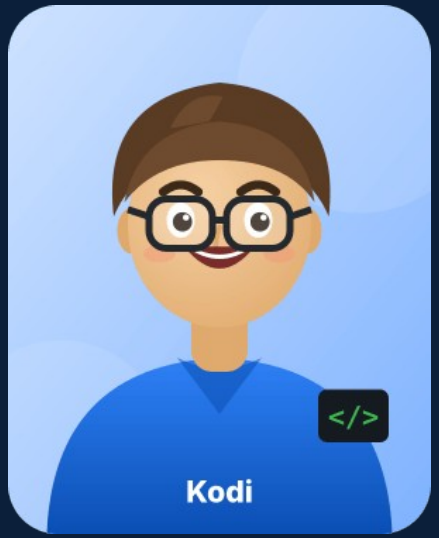
HEF-130 · MISSION 09 / 10

Concurrence et charge

Faire plusieurs choses à la fois n'accélère que ce qui attend. Et plus on ajoute de monde, plus il faut se coordonner.



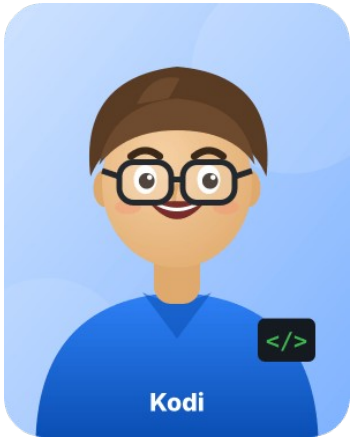


Badge : Chef d'orchestre du parallèle

105 min · avancé

L'histoire

Pour les inscriptions du lundi matin, HashBox ajoute des threads : de 8 à 64. Le débit monte… puis redescend. À 64 threads, le service est plus lent qu'à 8. Un test de charge révèle autre chose : sous un trafic constant, les temps de réponse sont bons ; sous un trafic en rafales, une minute de pic suffit pour que les files d'attente s'accumulent pendant dix minutes.



« Plus de concurrence n'est pas toujours plus vite, dit Sam. Tout dépend de ce qui attend : une base, un réseau, un verrou, un processeur. Et un système sous charge se comporte de façon non intuitive. Il faut apprendre à le tester. »

L'histoire (suite)

Tu apprends la différence entre concurrence et parallélisme, entre tâches limitées par les E/S et par le CPU, à utiliser threads et `asyncio`, à reconnaître courses et contention, à appliquer la contre-pression et à mener un test de charge.

Ta mission

À la fin, je serai capable de…

1. distinguer concurrence et parallélisme, tâches liées aux E/S et au CPU
2. mesurer le gain de threads et d'`asyncio` sur des tâches d'attente
3. comprendre le verrou global de Python (GIL) et ses conséquences
4. reconnaître une condition de course et la contention sur un verrou
5. simuler un test de charge, trouver le point de saturation et appliquer la contre-pression

Défi d'échauffement

Défi d'échauffement

Un cuisinier prépare 10 plats qui demandent chacun 2 minutes de travail actif puis 20 minutes au four. Il en a un seul à servir à la fois. Comment organiser le travail pour tout finir le plus vite ? Que se passe-t-il s'il n'y a qu'un four ? Et si les 10 plats demandent 22 minutes de travail actif et pas de four ?

Le concept

Concurrence ou parallélisme ?

	Définition	Exemple
Concurrence	gérer **plusieurs tâches en cours**, qui avancent en alternance	un serveur qui gère 1 000 connexions avec un seul thread
Parallélisme	exécuter **réellement en même temps** sur plusieurs cœurs	8 cœurs calculant 8 morceaux

Deux types de charge

Type	Le programme passe son temps à…	Remède
Lié aux E/S (*I/O-bound*)	attendre réseau, disque, base	concurrence : threads, `asyncio`
Lié au CPU (*CPU-bound*)	calculer	parallélisme : plusieurs processus, code natif, algorithme

Le concept

Python et le GIL

Dans CPython, le **verrou global de l'interpréteur (GIL)** n'autorise qu'**un seul thread à exécuter du code Python à la fois** (des versions expérimentales sans GIL existent). Conséquences :

- **Threads** : utiles pour les **E/S** (le GIL est relâché pendant l'attente), **inutiles** pour accélérer du calcul Python pur.
- **Processus** (`multiprocessing`, `ProcessPoolExecutor`) : contournent le GIL, au prix du coût de démarrage et de la **copie des données** entre processus.
- **Bibliothèques natives** (NumPy, etc.) relâchent souvent le GIL.

Les outils de concurrence en Python

Le concept

Les outils de concurrence en Python (suite)

Outil	Quand
`threading` / `ThreadPoolExecutor`	quelques dizaines à centaines de tâches d'E/S, code bloquant existant
`asyncio`	des milliers de connexions/tâches d'E/S, bibliothèques asynchrones
`multiprocessing` / `ProcessPoolExecutor`	calcul intensif sur plusieurs cœurs
File de tâches + travailleurs	découpler les producteurs et les consommateurs

Conditions de course

Une **condition de course** apparaît quand le résultat dépend de l'**entrelacement** de plusieurs fils d'exécution sur une donnée partagée. Classique : **lire-modifier-écrire**.

Le concept

Conditions de course (suite)

```
fil A : lit 10     fil B : lit 10
fil A : écrit 11   fil B : écrit 11      → un incrément est perdu
```

Remèdes : verrou (`threading.Lock`), opérations atomiques, file de messages, **éviter l'état partagé** (données immuables, un seul propriétaire).

Contention et coût de la coordination

Un verrou protège la donnée mais **sérialise** les accès : si tout le monde attend le même verrou, ajouter des fils n'accélère rien (Amdahl, mission 1). Au-delà d'un point, la **commutation de contexte** et la coordination coûtent plus que le gain : le débit **redescend**. D'où l'intérêt de :

- réduire la **portée** et la **durée** des verrous ;
- partitionner (un verrou par fragment) ;
- structures sans verrou ou file de messages ;

Le concept

Contention et coût de la coordination (suite)

- dimensionner les pools au **goulot** réel (taille du pool de connexions à la base, par exemple).

Tester la charge

Type de test	Question
Charge	tient-on le trafic attendu ?
Montée en charge / palier	où le système sature-t-il ?
Endurance	tient-il des heures (fuites) ?
Pic	comment réagit-il à une rafale ?
Stress	comment échoue-t-il au-delà de la capacité ?

À respecter : générateur de charge **en boucle ouverte** (mission 2), environnement représentatif, **percentiles**, surveillance des ressources, relevés à chaque palier.

Le concept

Saturation et contre-pression

Quand l'arrivée dépasse la capacité, une file **non bornée** grandit sans fin : latence infinie, mémoire épuisée. La **contre-pression** (*backpressure*) refuse ou ralentit le travail entrant :

Mécanisme	Effet
File **bornée**	le producteur est bloqué ou le travail refusé
Délestage (*load shedding*)	on rejette vite (HTTP 429/503) plutôt que de répondre tard
Limitation de débit	plafond par client
Délais d'attente	ne pas faire attendre pour rien
Priorités	les requêtes importantes d'abord

Mieux vaut **échouer vite pour une partie des requêtes** que **répondre lentement à toutes**.

Le concept

Saturation et contre-pression (suite)

🧠 ***Astuce*** *: avant d'ajouter des threads, trouve* ***ce qui attend*** *et* ***quel est le goulot*** *partagé. Si c'est un verrou ou une base saturée, plus de threads empire les choses.*

Exemple pas à pas

Problème : mesurer le gain des threads et d'`asyncio` sur des tâches d'attente, constater les limites avec du calcul, simuler une condition de course et son verrou, tracer une courbe de saturation et appliquer la contre-pression.

```
import asyncio, heapq, os, random, threading, time
from concurrent.futures import ThreadPoolExecutor

# ---------------- 1. tâches d'E/S : threads et asyncio ----------------
def tache_io(_): time.sleep(0.05)                                    # attent
   e simulée (réseau, base)

def sequentiel(n):
    d = time.perf_counter(); [tache_io(i) for i in range(n)]; return time.perf_cou
   nter() - d
def avec_threads(n, travailleurs):
    d = time.perf_counter()
    with ThreadPoolExecutor(travailleurs) as pool: list(pool.map(tache_io, range(n
   )))
```

Exemple pas à pas

Exemple pas à pas (suite)

```
    return time.perf_counter() - d
async def avec_asyncio(n):
    d = time.perf_counter(); await asyncio.gather(*(asyncio.sleep(0.05) for _ in r
   ange(n))); return time.perf_counter() - d

N = 40
t_seq, t_thr, t_asy = sequentiel(N), avec_threads(N, 20), asyncio.run(avec_asyncio
   (N))
print(f"{N} attentes de 50 ms : séquentiel {t_seq:.2f} s | 20 threads {t_thr:.2f}
   s | asyncio {t_asy:.2f} s")
assert t_seq > 1.9 and t_thr < t_seq / 5 and t_asy < t_seq / 10
t_1000 = asyncio.run(avec_asyncio(1000))
print(f"asyncio : 1000 attentes simultanées de 50 ms en {t_1000:.2f} s")
assert t_1000 < 1.0

# ---------------- 2. tâches de calcul : les threads n'aident pas (GIL) ----------
   ------
def calcul(_):
    x = 0
```

Exemple pas à pas (suite)

```
    for i in range(300000): x += i * i
    return x
def seq_calcul(n=8):
    d = time.perf_counter(); [calcul(i) for i in range(n)]; return time.perf_count
   er() - d
def thr_calcul(n=8):
    d = time.perf_counter()
    with ThreadPoolExecutor(8) as pool: list(pool.map(calcul, range(n)))
    return time.perf_counter() - d
a, b = min(seq_calcul() for _ in range(3)), min(thr_calcul() for _ in range(3))
print(f"8 calculs : séquentiel {a * 1000:.0f} ms | 8 threads {b * 1000:.0f} ms (ra
   pport {a / b:.2f}, avec le GIL on reste proche de 1)")
assert a / b < 2.5

# ---------------- 3. condition de course, rejouée de façon déterministe ---------
   -------
def incrementer(compteur, pas):                  # chaque « fil » est un générat
   eur : chaque yield est un point où l'autre peut s'intercaler
    valeur = compteur["n"]; yield                # lire
```

Exemple pas à pas (suite)

```
    compteur["n"] = valeur + 1; yield            # écrire

def executer(ordre, verrou=False):
    compteur = {"n": 0}; a, b = incrementer(compteur, 0), incrementer(compteur, 0)
    fils = {"A": a, "B": b}
    for nom in ordre:
        next(fils[nom], None)
    return compteur["n"]

assert executer("AABB") == 2                     # exécutions l'une après l'autr
   e : correct
assert executer("ABAB") == 1                     # entrelacement lire-lire-écrir
   e-écrire : un incrément perdu !
print("entrelacement AABB → 2 (correct) ; ABAB → 1 (incrément perdu)")

compteur, verrou = {"n": 0}, threading.Lock()
def travail_avec_verrou():
    for _ in range(20000):
        with verrou: compteur["n"] += 1
```

Exemple pas à pas (suite)

```
ts = [threading.Thread(target=travail_avec_verrou) for _ in range(4)]
[t.start() for t in ts]; [t.join() for t in ts]
assert compteur["n"] == 80000
print("avec un verrou : 4 fils × 20 000 incréments =", compteur["n"])

# ---------------- 4. courbe de saturation (simulation à événements discrets) ----
   ------------
def simuler_charge(clients, serveurs=4, service=0.020, reflexion=0.100, duree=60,
   graine=1):
    # boucle FERMÉE : chaque client attend sa réponse, « réfléchit » puis renvoie
   une requête
    rng = random.Random(graine)
    libre = [0.0] * serveurs; heap = [(rng.uniform(0, reflexion), i) for i in rang
   e(clients)]
    heapq.heapify(heap); latences, termines = [], 0
    while heap:
        t, c = heapq.heappop(heap)
        if t > duree: break
        s = min(range(serveurs), key=lambda k: libre[k])
```

Exemple pas à pas (suite)

```
        debut = max(t, libre[s]); fin = debut + rng.expovariate(1 / service); libr
   e[s] = fin
        latences.append(fin - t); termines += 1
        heapq.heappush(heap, (fin + rng.expovariate(1 / reflexion), c))
    latences.sort()
    return termines / duree, latences[int(0.95 * len(latences))]

print("clients | débit (req/s) | p95 (ms)")
courbe = {}
for c in (5, 10, 20, 40, 80, 160):
    debit, p95 = simuler_charge(c)
    courbe[c] = (debit, p95); print(f"  {c:4d}  |  {debit:8.0f}     | {p95 * 1000:
   7.0f}")
assert courbe[160][0] < 1.1 * 4 / 0.020                              # le débit pl
   afonne à la capacité (4 serveurs / 20 ms = 200 req/s)
assert courbe[160][1] > 10 * courbe[10][1]                           # la latence
   p95 explose après la saturation
assert courbe[80][0] > 0.85 * courbe[160][0]
```

Exemple pas à pas (suite)

```
# ---------------- 5. contre-pression : file bornée et délestage ----------------
def arrivees_en_pic(duree=120, lambda_normal=150, lambda_pic=400, debut_pic=30, fi
   n_pic=60, capacite=200, taille_file=None, graine=2):
    rng, file_, t, refuses, servis, latences = random.Random(graine), [], 0.0, 0,
   0, []
    prochain_service = 0.0
    while t < duree:
        lam = lambda_pic if debut_pic <= t < fin_pic else lambda_normal
        t += rng.expovariate(lam)
        while file_ and prochain_service <= t:                          # le serveur
    traite les requêtes en attente
            arrivee = file_.pop(0); debut = max(prochain_service, arrivee)
            prochain_service = debut + 1 / capacite; latences.append(prochain_serv
   ice - arrivee); servis += 1
        if taille_file is not None and len(file_) >= taille_file:
            refuses += 1                                                 # délestage
    : on refuse vite
        else:
            if not file_ and prochain_service < t: prochain_service = t
```

Exemple pas à pas

Exemple pas à pas (suite)

```
            file_.append(t)
    latences.sort()
    return servis, refuses, latences[int(0.99 * len(latences))], len(file_)

s1, r1, p99_1, reste1 = arrivees_en_pic(taille_file=None)
s2, r2, p99_2, reste2 = arrivees_en_pic(taille_file=100)
print(f"file illimitée : p99 = {p99_1:6.2f} s, {reste1} en attente à la fin | file
    bornée (100) : p99 = {p99_2:5.2f} s, {r2} refusées")
assert p99_1 > 20 * p99_2 and r2 > 0 and r1 == 0
```

Lecture :

1. Pour 40 attentes de 50 ms : **2 s en série**, ≈ 0,1 s avec 20 threads, ≈ 0,05 s avec `asyncio` — qui gère aussi **1 000 attentes simultanées en ~0,05 s**. La concurrence aide parce que les tâches **attendent**.

Exemple pas à pas (suite)

2. Pour du **calcul pur**, 8 threads ne sont pas plus rapides qu'un seul (le GIL les fait travailler à tour de rôle).
3. La **course** est rejouée de façon déterministe : l'entrelacement ABAB (les deux lisent avant que l'un écrive) perd un incrément ; avec un **verrou**, le résultat est exact.
4. La **courbe de saturation** : le débit croît puis **plafonne à la capacité** (200 req/s = 4 serveurs / 20 ms) ; la latence p95 explose après saturation.
5. Pendant un pic, une file **illimitée** donne des latences de plusieurs secondes (le travail s'accumule) ; une file **bornée** avec délestage garde des latences courtes pour les requêtes acceptées et **refuse vite** les autres.

À toi, avec aide

La loi d'extensibilité universelle (USL) modélise le débit avec `N` travailleurs :

```
X(N) = λ × N / (1 + σ × (N − 1) + κ × N × (N − 1))
```

`σ` mesure la **contention** (partie sérialisée), `κ` le **coût de cohérence**. Avec `λ = 100` req/s par travailleur, `σ = 0,05`, `κ = 0,002` :

1. calcule le débit pour N = 1, 2, 4, 8, 16, 32, 64, 128 ;
2. trouve le N qui maximise le débit (valeur théorique `√((1 − σ)/κ)`) et le débit correspondant ;
3. compare avec Amdahl seul (κ = 0) ;
4. que recommandes-tu pour dimensionner le pool de threads ?

Indices progressifs

Indice 1

Calcule les 8 valeurs avec une petite fonction.

Indice 2

N optimal ≈ √(0,95 / 0,002) ≈ 21,8.

Indice 3

Avec κ = 0, le débit ne fait que plafonner (à λ/σ = 2000).

Réponse

```
import math
def usl(N, lam=100, sigma=0.05, kappa=0.002):
    return lam * N / (1 + sigma * (N - 1) + kappa * N * (N - 1))
for N in (1, 2, 4, 8, 16, 32, 64, 128):
    print(f"N = {N:3d} : {usl(N):7.1f} req/s   (Amdahl seul : {usl(N, kappa=0):7.1
   f})")
n_opt = math.sqrt((1 - 0.05) / 0.002)
print(f"N optimal ≈ {n_opt:.1f} → {usl(round(n_opt)):.0f} req/s")
assert 21 <= round(n_opt) <= 22 and usl(round(n_opt)) > usl(64) > usl(128)
```

Le débit atteint son **maximum vers 22 travailleurs (≈ 740 req/s)** puis **redescend** (à 128 travailleurs il est inférieur à celui de 16). Avec Amdahl seul (sans coût de cohérence), le débit ne fait que plafonner vers 2 000 : le modèle simple **ne voit pas** la dégradation. **Recommandation** : dimensionner le pool vers 16-24 (jamais « le plus grand possible »), **mesurer** σ et κ par un test de charge, et chercher à réduire la contention (verrous plus fins, partitionnement) qui repousse le plafond.

À toi, en solo

Banc de charge réel sur un goulot partagé. Crée un « service » dont chaque requête (a) passe 20 ms dans une section limitée à **4 accès simultanés** (un sémaphore, comme un pool de connexions de 4) et (b) 5 ms hors section.

1. exécute 120 requêtes avec des pools de 1, 2, 4, 8, 16, 32 threads et mesure débit et durée totale ;
2. repère le **point de saturation** et explique-le avec la loi de Little (4 connexions × 1/20 ms = 200 req/s) ;
3. ajoute une **file bornée** devant le service : au-delà de 20 requêtes en attente, réponse immédiate « refusée » ; envoie 300 requêtes d'un coup et compte servies/refusées et la latence des servies ;
4. compare avec la même salve sans borne (latence de la dernière requête servie).

À toi, en solo · réponse

Pistes de correction

```
import queue, threading, time
from concurrent.futures import ThreadPoolExecutor

SECTION = threading.Semaphore(4)                                    # 4 « connexio
   ns »
def requete(_):
    time.sleep(0.005)
    with SECTION:
        time.sleep(0.020)
    return True

def debit(travailleurs, n=120):
    d = time.perf_counter()
    with ThreadPoolExecutor(travailleurs) as pool: list(pool.map(requete, range(n)
   ))
    duree = time.perf_counter() - d
    return n / duree, duree

res = {}
```

Pistes de correction (suite)

```
for t in (1, 2, 4, 8, 16, 32):
    res[t] = debit(t)
    print(f"{t:3d} threads : {res[t][0]:6.0f} req/s ({res[t][1]:.2f} s)")
assert res[4][0] > 3 * res[1][0] * 0.8                                  # jusqu'à 4
   , presque linéaire
assert res[32][0] < 1.35 * res[8][0]                                    # au-delà,
   plafond imposé par les 4 connexions (≈ 200 req/s)
assert res[32][0] < 260

# ---- file bornée : délestage ----
def salve(taille_max, n=300, travailleurs=8):
    q, servies, refusees, latences, verrou = queue.Queue(), [0], [0], [], threadin
   g.Lock()
    arret = object()
    def consommateur():
        while True:
            item = q.get()
            if item is arret: return
            requete(None)
```

À toi, en solo · réponse

Pistes de correction (suite)

```
            with verrou: servies[0] += 1; latences.append(time.perf_counter() - it
    em)
     ts = [threading.Thread(target=consommateur) for _ in range(travailleurs)]
     [t.start() for t in ts]
     for _ in range(n):                                                  # n requêt
    es d'un coup
         if taille_max is not None and q.qsize() >= taille_max:
             refusees[0] += 1
         else:
             q.put(time.perf_counter())
     for _ in ts: q.put(arret)
     [t.join() for t in ts]
     return servies[0], refusees[0], max(latences)

s_inf, r_inf, l_inf = salve(None)
s_b, r_b, l_b = salve(20)
print(f"sans borne : {s_inf} servies, {r_inf} refusées, latence max {l_inf:.2f} s
    | borne 20 : {s_b} servies, {r_b} refusées, latence max {l_b:.2f} s")
assert s_inf == 300 and r_inf == 0 and r_b > 0 and l_b < l_inf / 2
```

À toi, en solo · réponse

Pistes de correction (suite)

Le débit croît jusqu'à **4 threads** puis **plafonne** : le goulot est la section à 4 accès (Little : 4 / 0,020 s = 200 req/s). Ajouter des threads au-delà n'apporte rien. La **file bornée** refuse vite l'excès et garde des latences courtes pour ce qui est accepté ; sans borne, la dernière requête attend plusieurs fois plus longtemps.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Ajouter des threads pour du calcul Python	Aucun gain (GIL)	Processus, code natif, meilleur algorithme
État partagé sans protection	Courses : résultats faux et rares	Verrou, file de messages, données immuables
File d'attente sans limite	Latence infinie et mémoire saturée	File bornée, délestage, délais d'attente
Pool de threads « le plus grand possible »	Contention, commutations, débit qui redescend	Dimensionner au goulot, mesurer

Mini-boss

Mini-boss

Test de charge et dimensionnement du service d'inscription. Le service : 4 cœurs, une base à 20 connexions (requête de 15 ms), une réponse en 10 ms de CPU. Objectifs : 400 req/s, p99 < 300 ms, tenir un pic à 2× pendant 1 minute.

1. modélise le service (simulation à événements discrets) avec deux ressources (CPU et connexions) ;
2. génère la courbe de saturation (débit/latence contre clients) et identifie le goulot ;
3. teste en **boucle ouverte** avec un trafic constant puis un pic ; mesure p50/p95/p99 ;
4. ajoute file bornée + délestage + délais d'attente et compare (taux de refus, p99 des acceptées) ;
5. ajoute un second goulot (verrou partagé) et ajuste le modèle avec USL ;
6. propose un dimensionnement (nombre d'instances, taille de pool) satisfaisant les objectifs avec marge N+1 ;

Mini-boss

Mini-boss (suite)

7. rédige le plan du test de charge réel (paliers, métriques, critères d'arrêt).

Mini-boss · réponse

Pistes de correction

Un bon travail : une simulation à deux ressources où la capacité est `min(4 / 0,010 = 400, 20 / 0,015 ≈ 1333)` donc le CPU est le goulot (400 req/s) — un pic à 2× nécessite soit des instances supplémentaires, soit un délestage; comparaison file illimitée (p99 qui diverge pendant le pic) et file bornée (p99 stable, taux de refus chiffré); modèle USL ajusté; recommandation : 2 instances minimum (N+1 : 3) avec file bornée à ~50 et limitation du débit par client; plan de test avec paliers de 25 %, relevés de percentiles, arrêt si erreurs > 1 %.

Quiz éclair

Quiz éclair

1. Les threads Python accélèrent surtout :

A. le calcul numérique pur
B. les tâches limitées par les entrées-sorties (attentes)
C. la compilation

2. Une condition de course est :

A. une compétition entre deux serveurs
B. une erreur réseau
C. un résultat qui dépend de l'entrelacement des fils sur une donnée partagée

3. Face à une surcharge, la contre-pression consiste à :

A. refuser ou ralentir le travail entrant plutôt que d'accumuler une file infinie

Quiz éclair

Quiz éclair (suite)

B. augmenter la file sans limite
C. redémarrer le serveur

4. 🔁 *Révision mission 08* : Le temps d'accès moyen d'un cache dépend de :

A. uniquement de sa taille
B. son taux de succès et des temps du cache et de la source
C. de la couleur des données

5. 🔁 *Révision mission 08* : La ruée sur le cache (cache stampede) se prévient par :

A. l'augmentation du nombre de requêtes
B. la suppression du cache
C. un verrou par clé (un seul chargeur) et une expiration avec gigue

Quiz éclair · réponse

Réponses

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

Je réfléchis

1. Dans ton système, qu'est-ce qui attend le plus (base, réseau, verrou, CPU) ?
2. Que fait ton service quand le trafic double pendant dix minutes : il refuse, il ralentit, ou il s'effondre ?

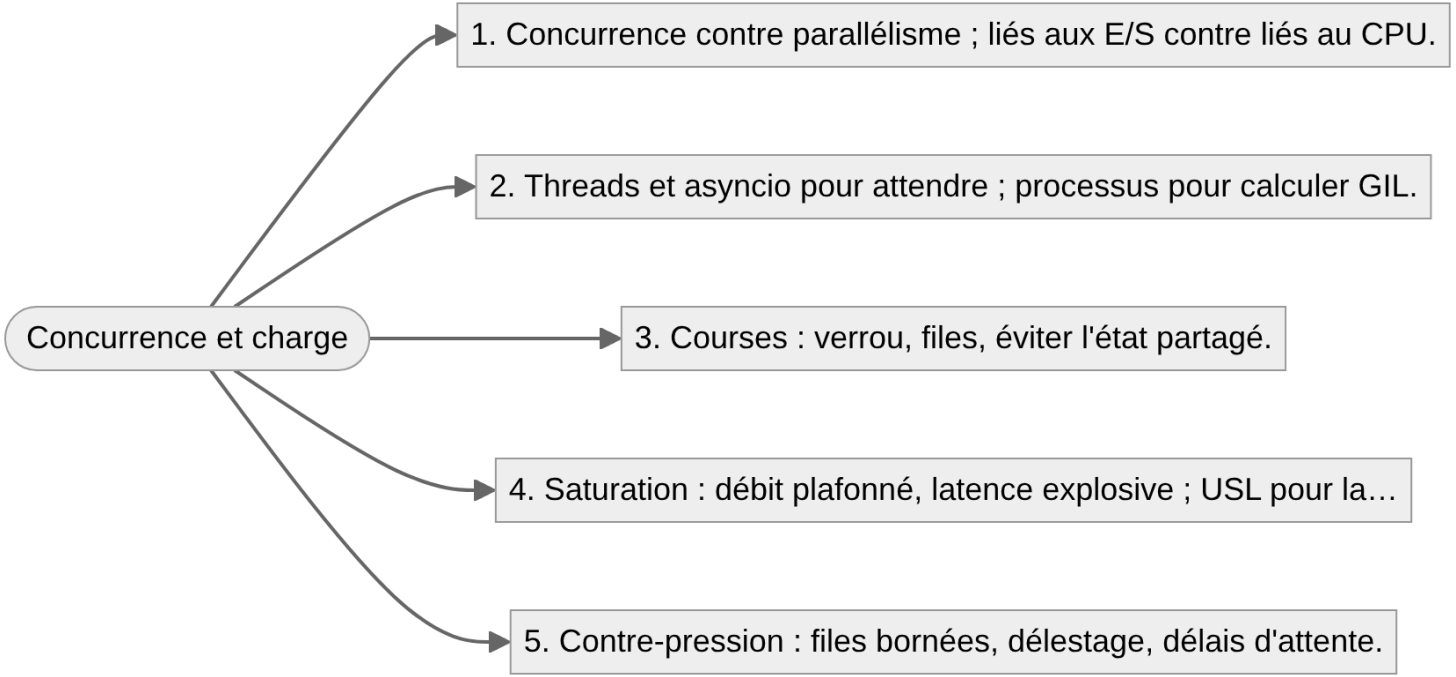
À retenir

À retenir

1. Concurrence contre parallélisme ; liés aux E/S contre liés au CPU.
2. Threads et asyncio pour attendre ; processus pour calculer (GIL).
3. Courses : verrou, files, éviter l'état partagé.
4. Saturation : débit plafonné, latence explosive ; USL pour la contention.
5. Contre-pression : files bornées, délestage, délais d'attente.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Chef d'orchestre du parallèle

Prochaine mission : Optimisation et validation

