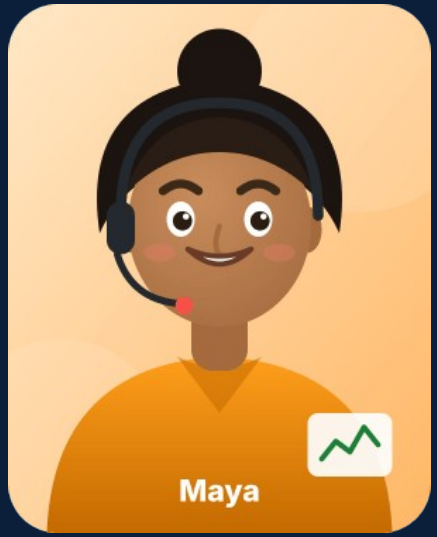
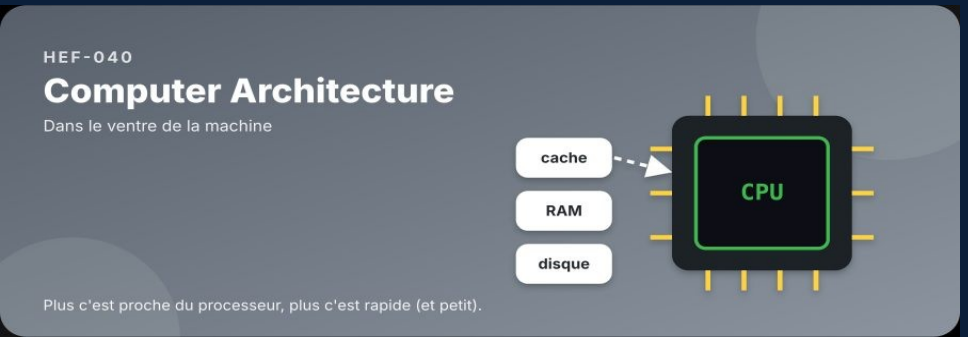
HEF-040 · MISSION 06 / 10

Stockage et entrées/sorties

Le disque est lent, le réseau est lent. Savoir en tirer le meilleur, c'est grouper, séquencer et mettre en tampon.



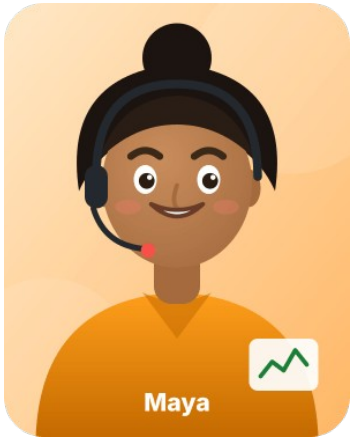


Badge : Maître du flux

70 min · débutant

L'histoire

Le script d'import de HashBox écrit 200 000 lignes dans un fichier. Il met plusieurs minutes. Un collègue réécrit la même opération et elle prend 2 secondes.



« Il n'a pas changé la logique, dit Sam. Il a changé la façon de parler au disque : au lieu de lui envoyer 200 000 petits messages, il en envoie quelques gros. »

Tu comprends comment le stockage fonctionne et pourquoi la façon d'écrire compte plus que la quantité.

Ta mission

À la fin, je serai capable de…

1. comparer disque dur (HDD) et SSD
2. distinguer accès séquentiel et accès aléatoire
3. expliquer la mise en tampon (buffering) et le regroupement
4. mesurer l'écart entre écritures non groupées et groupées
5. expliquer ce qu'est un appel système d'entrée/sortie

Défi d'échauffement

Défi d'échauffement

Tu dois envoyer 1 000 lettres à la poste. Option A : tu fais 1 000 allers-retours avec une lettre à chaque fois. Option B : tu les mets toutes dans un sac et tu fais un seul trajet. Dans quelle mesure l'option A est-elle plus lente ? Qu'est-ce qui coûte le plus : la lettre ou le trajet ?

Le concept

Les supports de stockage

Support	Fonctionnement	Accès aléatoire	Débit séquentiel
Disque dur (HDD)	plateaux magnétiques + tête de lecture mécanique	lent (ms, il faut déplacer la tête)	~100 à 250 Mo/s
SSD (SATA)	puces mémoire flash, sans pièce mécanique	rapide (~100 µs)	~500 Mo/s
SSD NVMe	flash sur le bus PCIe	très rapide	plusieurs Go/s

Séquentiel ou aléatoire

- **Séquentiel** : lire/écrire des blocs voisins, dans l'ordre. Rapide.
- **Aléatoire** : sauter d'un endroit à un autre. Lent, surtout sur HDD.

Le concept

Le coût d'une opération d'entrée/sortie

Chaque lecture ou écriture passe par le **système d'exploitation** (un *appel système*). Il y a un **coût fixe** par appel (changement de contexte, vérifications), indépendant de la quantité de données.

> *Écrire 1 octet 1 000 000 de fois coûte beaucoup plus cher qu'écrire 1 Mo en une fois.*

Mise en tampon (buffering)

Un **tampon** (*buffer*) est une zone de mémoire où l'on accumule les données avant de les envoyer en un seul bloc. Les fichiers Python en utilisent un par défaut.

Le concept

Mise en tampon (buffering) (suite)

```
# Chaque write est mis dans un tampon, envoyé au système par blocs.
with open("sortie.txt", "w") as f:
    for i in range(100000):
        f.write(f"{i}\n")
```

On force l'envoi avec `f.flush()`. `os.fsync` demande au système de garantir l'écriture physique.

Le cache de pages du système

Le système d'exploitation garde en RAM les fichiers récemment lus (**cache de pages**). La deuxième lecture d'un fichier est donc bien plus rapide que la première.

Écrire de façon sûre

Le concept

Écrire de façon sûre (suite)

Une écriture « terminée » n'est pas forcément **sur le disque**. En cas de panne de courant, un tampon non écrit est perdu. Pour des données critiques (journal, base de données), on force l'écriture avec `fsync`, au prix de la vitesse.

Les mêmes idées sur le réseau

Envoyer 1 000 petites requêtes est plus lent qu'une requête groupée : même principe.

Exemple pas à pas

Problème : mesurer l'écart entre écrire ligne par ligne avec envoi forcé à chaque ligne, et écrire avec tampon.

```
import os, tempfile, time

def ecrire_sans_tampon(chemin, n):
    with open(chemin, "w") as f:
        for i in range(n):
            f.write(f"{i}\n")
            f.flush()                # force l'envoi au système à chaque ligne

def ecrire_avec_tampon(chemin, n):
    with open(chemin, "w") as f:
        for i in range(n):
            f.write(f"{i}\n")

def ecrire_en_bloc(chemin, n):
    with open(chemin, "w") as f:
```

Exemple pas à pas (suite)

```
        f.write("".join(f"{i}\n" for i in range(n)))

with tempfile.TemporaryDirectory() as dossier:
    chemin = os.path.join(dossier, "test.txt")
    for fonction in (ecrire_sans_tampon, ecrire_avec_tampon, ecrire_en_bloc):
        t = time.perf_counter()
        fonction(chemin, 100000)
        print(fonction.__name__, round(time.perf_counter() - t, 3), "s")
```

Lecture : écrire avec `flush` à chaque ligne est nettement plus lent que d'utiliser le tampon par défaut. L'écriture en un seul bloc est la plus rapide. Le contenu final est identique : seule la façon de **parler au système** change.

Compromis : le tampon améliore la vitesse mais retarde l'écriture physique. En cas de panne, on peut perdre les dernières lignes.

À toi, avec aide

Lis un fichier de 200 000 lignes de deux façons et compare :

1. `f.read(1)` caractère par caractère ;
2. `for ligne in f` (ligne par ligne, avec tampon) ;
3. `f.read()` en une fois.

Pourquoi la troisième est-elle rapide mais parfois dangereuse ?

À toi, avec aide · réponse

Indices progressifs

Indice 1

Crée d'abord le fichier de test dans un dossier temporaire.

Indice 2

Mesure chaque lecture avec `time.perf_counter()`.

Indice 3

Un fichier de 50 Go lu en une fois ne tient pas en mémoire.

À toi, avec aide · réponse

Réponse

```
import os, tempfile, time

with tempfile.TemporaryDirectory() as d:
    chemin = os.path.join(d, "donnees.txt")
    with open(chemin, "w") as f:
        f.write("".join(f"ligne {i}\n" for i in range(200000)))

    def lire_caractere():
        n = 0
        with open(chemin) as f:
            while f.read(1):
                n += 1
        return n

    def lire_lignes():
        with open(chemin) as f:
            return sum(1 for _ in f)

    def lire_tout():
```

À toi, avec aide · réponse

Réponse (suite)

```
        with open(chemin) as f:
            return len(f.read().splitlines())

    for fonction in (lire_caractere, lire_lignes, lire_tout):
        t = time.perf_counter(); fonction()
        print(fonction.__name__, round(time.perf_counter() - t, 3), "s")
```

`lire_tout` est rapide mais consomme toute la taille du fichier en mémoire : sur un très gros fichier, c'est dangereux. Lire par lignes ou par blocs est un bon compromis (mémoire constante, vitesse proche).

À toi, en solo

Écris `copier(source, destination, taille_bloc)` qui copie un fichier par blocs de `taille_bloc` octets. Mesure le temps pour une taille de fichier de 20 Mo avec des blocs de 1 octet (sur 200 Ko seulement, sinon trop long), 512 octets, 64 Ko et 1 Mo.
Quelle taille de bloc te semble un bon choix ? Pourquoi le gain devient-il faible au-delà d'une certaine taille ?

À toi, en solo · réponse

Pistes de correction

```
import os, tempfile, time

def copier(source, destination, taille_bloc):
    with open(source, "rb") as entree, open(destination, "wb") as sortie:
        while True:
            bloc = entree.read(taille_bloc)
            if not bloc:
                break
            sortie.write(bloc)

with tempfile.TemporaryDirectory() as d:
    src, dst = os.path.join(d, "src.bin"), os.path.join(d, "dst.bin")
    with open(src, "wb") as f:
        f.write(os.urandom(2_000_000))
    for taille in (512, 4096, 65536, 1_048_576):
        t = time.perf_counter()
        copier(src, dst, taille)
        print(taille, round(time.perf_counter() - t, 4), "s")
    assert open(src, "rb").read() == open(dst, "rb").read()
```

Pistes de correction (suite)

Un bloc de 64 Ko est un bon compromis. Au-delà, le coût fixe par appel est déjà négligeable devant le transfert lui-même, et un bloc très grand consomme plus de mémoire.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Écrire octet par octet	Un appel système par octet	Écrire par blocs ou utiliser le tampon
Lire tout un gros fichier en mémoire	Épuisement mémoire	Traiter par lignes ou blocs
Croire que `write` = sur le disque	Perte de données en cas de panne	`flush` et `fsync` pour les données critiques
Accès aléatoires sur HDD	Très lent (déplacements mécaniques)	Grouper et trier les accès

Mini-boss

Mini-boss

Importateur robuste. Écris `importer(fichier_csv, fichier_sortie)` qui lit un gros fichier CSV de 200 000 lignes `id;nom;score`, garde les lignes dont le score est ≥ 10, et écrit le résultat.
Contraintes : mémoire constante (pas de `read()` complet), écriture par blocs de 1 000 lignes, comptage des lignes lues/écrites/rejetées.

1. Écris la version simple (ligne par ligne, `flush` à chaque ligne) et la version optimisée.
2. Vérifie qu'elles produisent exactement le même fichier.
3. Mesure et explique l'écart.
4. Que se passe-t-il si le programme est interrompu au milieu ? Comment limiter la perte (écrire dans un fichier temporaire puis renommer) ?

Mini-boss · réponse

Pistes de correction

```
import os, tempfile, time

def importer_simple(src, dst):
    lues = ecrites = 0
    with open(src) as f, open(dst, "w") as g:
        for ligne in f:
            lues += 1
            id_, nom, score = ligne.rstrip("\n").split(";")
            if int(score) >= 10:
                g.write(ligne); g.flush(); ecrites += 1
    return lues, ecrites

def importer_optimise(src, dst, bloc=1000):
    lues = ecrites = 0
    tmp = dst + ".tmp"
    tampon = []
    with open(src) as f, open(tmp, "w") as g:
        for ligne in f:
            lues += 1
```

Pistes de correction (suite)

```
            if int(ligne.rstrip("\n").split(";")[2]) >= 10:
                tampon.append(ligne); ecrites += 1
                if len(tampon) >= bloc:
                    g.write("".join(tampon)); tampon.clear()
        g.write("".join(tampon))
    os.replace(tmp, dst)          # remplacement atomique : fichier complet ou anc
   ien fichier
    return lues, ecrites

with tempfile.TemporaryDirectory() as d:
    src = os.path.join(d, "in.csv")
    with open(src, "w") as f:
        f.write("".join(f"{i};nom{i};{i % 20}\n" for i in range(50000)))
    for fn, out in ((importer_simple, "a.csv"), (importer_optimise, "b.csv")):
        t = time.perf_counter(); r = fn(src, os.path.join(d, out))
        print(fn.__name__, r, round(time.perf_counter() - t, 3), "s")
    assert open(os.path.join(d, "a.csv")).read() == open(os.path.join(d, "b.csv"))
   .read()
```

Mini-boss · réponse

Pistes de correction (suite)

Le fichier temporaire suivi de `os.replace` garantit qu'on ne laisse jamais un fichier de sortie à moitié écrit.

Quiz éclair

Quiz éclair

1. Un SSD est plus rapide qu'un HDD en accès aléatoire parce que :

A. il est plus gros
B. il n'a pas de pièce mécanique à déplacer
C. il est moins cher

2. Écrire 1 octet un million de fois est généralement :

A. aussi rapide
B. impossible
C. bien plus lent qu'écrire 1 Mo d'un coup

3. Un tampon (buffer) sert à :

A. regrouper des données avant de les envoyer

Quiz éclair

Quiz éclair (suite)

B. effacer des fichiers
C. chiffrer les données

4. 🔁 *Révision mission 05* : Un cache stocke :

A. les programmes installés
B. des copies de données récemment utilisées
C. les mots de passe

5. 🔁 *Révision mission 04* : La RAM est :

A. persistante
B. aussi rapide que le cache
C. volatile

Quiz éclair · réponse

Réponses

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

Je réfléchis

1. Quelle tâche de ta vie bénéficierait d'être « groupée » (courses, rendez-vous, e-mails) ?
2. Pourquoi vaut-il parfois mieux accepter de perdre quelques secondes de données que de forcer chaque écriture ?

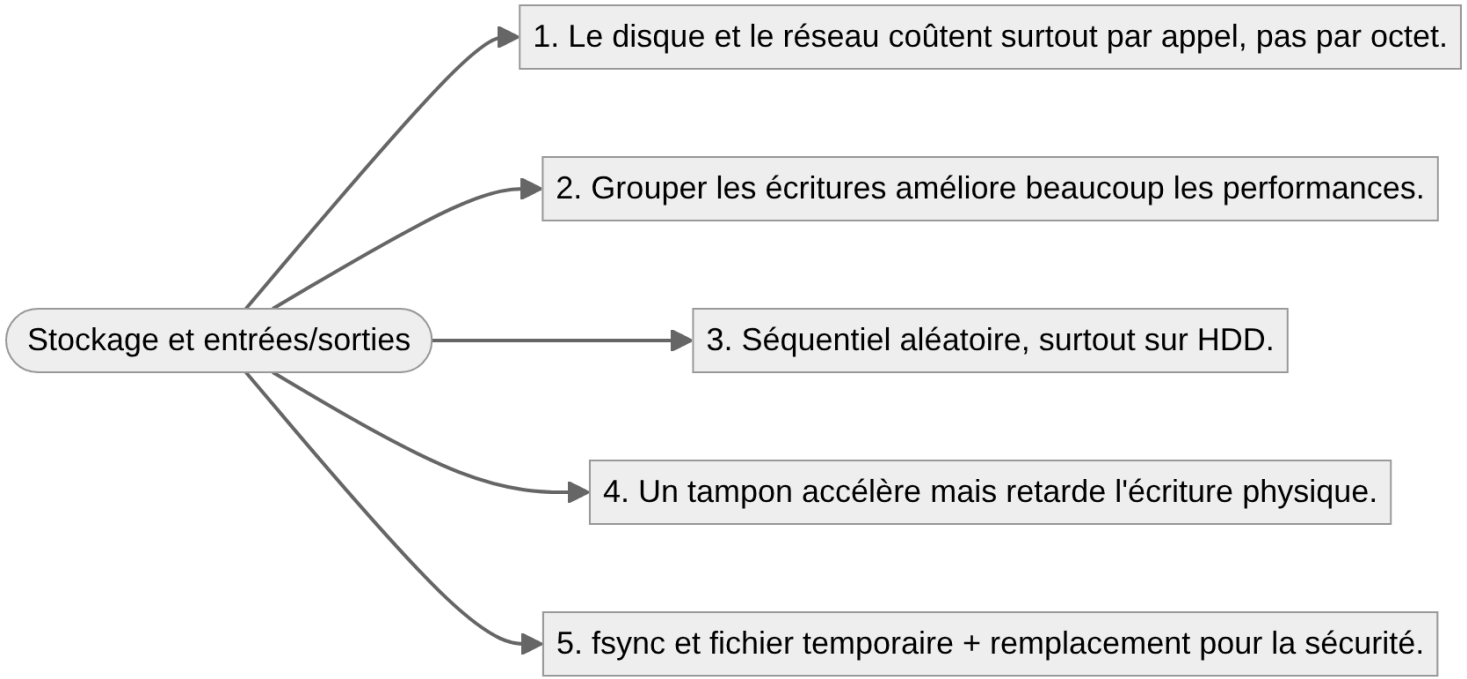
À retenir

À retenir

1. Le disque et le réseau coûtent surtout par appel, pas par octet.
2. Grouper les écritures améliore beaucoup les performances.
3. Séquentiel > aléatoire, surtout sur HDD.
4. Un tampon accélère mais retarde l'écriture physique.
5. `fsync` et fichier temporaire + remplacement pour la sécurité.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Maître du flux

Prochaine mission : Bus, interruptions et périphériques

