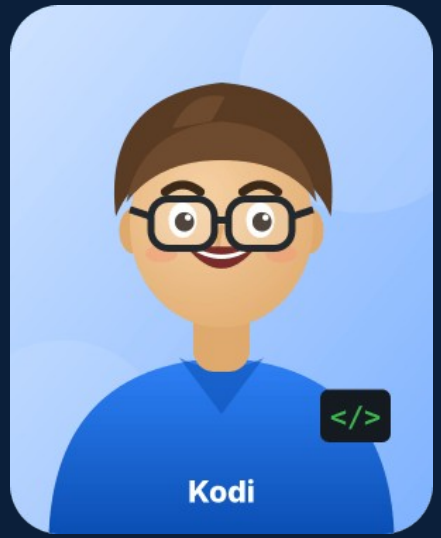
HEF-130 · MISSION 05 / 10

I/O et stockage

Le processeur compte en nanosecondes, le disque et le réseau en millisecondes. Quand le programme attend, c'est souvent une entrée-sortie qu'il attend.



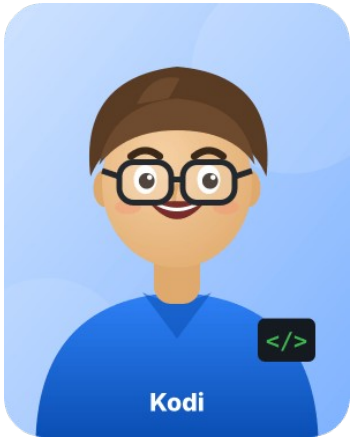


Badge : Plombier des entrées-sorties

100 min · avancé

L'histoire

L'export des factures de HashBox écrit 200 000 lignes dans un fichier. Il prend 38 secondes sur le serveur de production, alors que la même opération dure 0,4 s sur l'ordinateur de Kodi. Après enquête, Sam trouve la cause : le code appelle `flush()` après chaque ligne, et le serveur de production utilise un disque réseau où chaque vidage coûte 0,2 ms.



> *« 200 000 vidages × 0,2 ms, ça fait 40 secondes, dit Sam. Le calcul ne coûtait rien. Ce qui coûtait, c'était d'aller voir le disque 200 000 fois au lieu de une. En entrée-sortie, on gagne en regroupant : moins d'appels, plus gros, plus séquentiels. »*

Tu apprends comment un programme parle au disque (tampons, appels système, page cache), pourquoi regrouper et lire séquentiellement, et quels formats et compressions rendent le stockage plus efficace.

Ta mission

À la fin, je serai capable de…

1. expliquer la hiérarchie tampon / cache de pages / disque et le coût d'un appel système
2. regrouper lectures et écritures pour réduire le nombre d'appels
3. comparer accès séquentiel et aléatoire, tailles de blocs
4. comparer formats de sérialisation (texte, binaire) et compression en taille et en temps
5. savoir quand utiliser `mmap`, `fsync` et des écritures atomiques

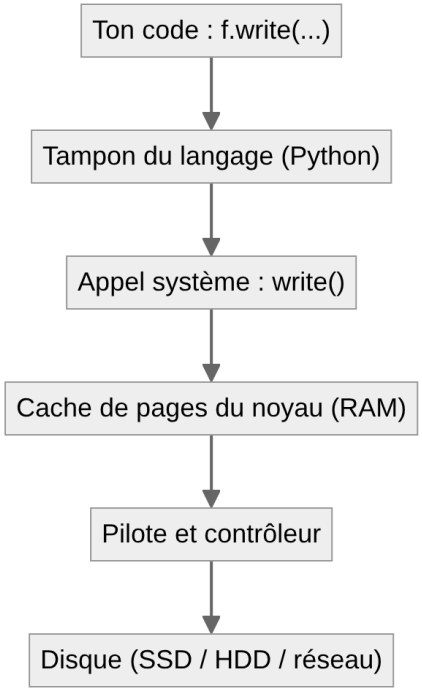
Défi d'échauffement

Défi d'échauffement

Tu dois déplacer 10 000 briques d'un tas à un autre, à 50 m de distance. Tu peux en porter 1 à la fois, ou 50 avec une brouette. Le trajet coûte plus que le chargement. Calcule le nombre de trajets dans chaque cas. Si en plus on te demande de signer un reçu à chaque trajet, où est la vraie perte de temps ?

Le concept

La pile des entrées-sorties (suite)



Le concept

La pile des entrées-sorties (suite)

Description du schéma : une écriture traverse le tampon du langage, un appel système, le cache de pages du noyau (en mémoire vive), puis le pilote et enfin le disque physique.

- Le **tampon** regroupe plusieurs petites écritures en une grosse.
- L'**appel système** (`read`, `write`) coûte ~0,2 à 2 µs à lui seul (bascule en mode noyau).
- Le **cache de pages** garde en mémoire les fichiers récemment lus ou écrits : une relecture est quasi instantanée.
- `flush()` vide le tampon Python **vers le noyau** ; `os.fsync()` force le noyau à écrire **sur le disque** (coûteux, mais nécessaire pour la durabilité).

Le coût d'une opération

Opération	Ordre de grandeur
Appel système	~1 µs

Le concept

Le coût d'une opération (suite)

Opération	Ordre de grandeur
Lecture dans le cache de pages	~1 µs par Ko
Lecture aléatoire SSD	~100 µs
`fsync` sur SSD	~1 ms
Lecture aléatoire disque dur	~10 ms

Les règles d'or

1. **Moins d'appels, plus gros** : regrouper (*batching*). 1 écriture de 1 Mo vaut mieux que 1 000 000 d'écritures de 1 octet.
2. **Séquentiel plutôt qu'aléatoire** : surtout sur disque dur ; sur SSD aussi (lecture anticipée, blocs).
3. **Taille de bloc adaptée** : 64 Ko à 1 Mo pour des lectures de flux (pas 1 octet, pas 1 Go).

Le concept

Les règles d'or (suite)

4. **Éviter de relire** : garder en mémoire ce qui sert plusieurs fois.
5. **Ne flusher/fsyncer que si nécessaire** (durabilité) et le faire **par lot**.
6. **Écrire de façon atomique** : écrire dans un fichier temporaire, `fsync`, puis renommer (`os.replace`) : un lecteur voit l'ancien fichier ou le nouveau, jamais un fichier à moitié écrit.

Formats de stockage

Format	Lisible ?	Taille	Vitesse	Usage
CSV / texte	oui	grande	moyenne	échange, débogage
JSON	oui	grande (noms répétés)	moyenne	API, configuration
Binaire structuré (`struct`, MessagePack, Protocol Buffers)	non	petite	rapide	échanges internes, grandes quantités

Le concept

Formats de stockage (suite)

Format	Lisible ?	Taille	Vitesse	Usage
Colonnaire (Parquet)	non	très petite	rapide pour l'analyse	analyses, entrepôts
SQLite / base	via requêtes	moyenne	indexé	données interrogées

Compression

Compresser **réduit la taille** (donc le temps de lecture disque/réseau) mais **coûte du CPU**. Selon le débit du support :

Algorithme	Rapport	Vitesse
`zlib` / gzip niveau 1	moyen	rapide
`zlib` niveau 9	meilleur	lent

Le concept

Compression (suite)

Algorithme	Rapport	Vitesse
`lzma`	très bon	très lent
zstd, lz4 (hors bibliothèque standard)	bon / moyen	très rapide

Pour des données **très redondantes** (texte, JSON, journaux), le gain est de ×5 à ×20. Pour des données **déjà compressées** (images, vidéos), pas de gain.

Projection en mémoire : `mmap`

`mmap` associe un fichier à l'espace d'adressage : on le lit comme un tableau d'octets, le système charge les pages à la demande. Utile pour de **grandes recherches en lecture** sans copies, ou pour partager des données entre processus.

Le concept

Durabilité et cohérence

- **Écriture atomique** (fichier temporaire + renommage) ;
- **Journal d'écriture** (*write-ahead log*) : consigner l'opération avant de l'appliquer, pour pouvoir la rejouer après une panne (c'est ce que font les bases de données, HEF-070) ;
- `fsync` du fichier **et** du dossier pour garantir qu'un renommage survit à une coupure de courant.

🧠 ***Astuce** : pour trouver un problème d'E/S, compte les appels système (strace -c sous Linux) : un nombre énorme d'appels minuscules est le signe d'un manque de regroupement.*

Exemple pas à pas

Problème : compter les écritures réelles avec un fichier instrumenté, montrer l'effet du regroupement, comparer des tailles de blocs, des formats et des compressions, et écrire un fichier de façon atomique.

```
import io, json, os, struct, tempfile, time, zlib

# ---------------- un fichier qui compte les écritures « système » ---------------
   -
class FichierCompteur(io.RawIOBase):
    def __init__(self, chemin):
        self.f = open(chemin, "wb", buffering=0)            # non tamponné : chaq
   ue write() est un appel système
        self.appels, self.octets = 0, 0
    def writable(self): return True
    def write(self, donnees):
        self.appels += 1; self.octets += len(donnees)
        return self.f.write(donnees)
    def close(self): self.f.close(); super().close()
```

Exemple pas à pas

Exemple pas à pas (suite)

```
LIGNES = [f"{i};m{i % 800};HEF-{i % 90:03d};{(i * 37) % 200}\n" for i in range(200
   00)]

with tempfile.TemporaryDirectory() as d:
    # 1. une écriture par ligne, sans tampon
    brut = FichierCompteur(os.path.join(d, "a.txt"))
    for l in LIGNES: brut.write(l.encode())
    brut.close()
    # 2. avec un tampon de 64 Ko
    brut2 = FichierCompteur(os.path.join(d, "b.txt"))
    tampon = io.BufferedWriter(brut2, buffer_size=64 * 1024)
    for l in LIGNES: tampon.write(l.encode())
    tampon.close()
    print(f"sans tampon : {brut.appels:6d} appels système | tampon 64 Ko : {brut2.
   appels:4d} appels (pour {brut.octets} octets)")
    assert brut.appels == 20000 and brut2.appels < 100 and brut.octets == brut2.oc
   tets
    assert open(os.path.join(d, "a.txt"), "rb").read() == open(os.path.join(d, "b.
   txt"), "rb").read()
```

Exemple pas à pas (suite)

```
 # 3. mesure du temps : regrouper vaut mieux que flusher à chaque ligne
 def ecrire_flush(chemin):
     with open(chemin, "w") as f:
         for l in LIGNES:
             f.write(l); f.flush()
 def ecrire_groupe(chemin):
     with open(chemin, "w") as f:
         f.write("".join(LIGNES))
 def meilleur(f, chemin, n=3):
     t = float("inf")
     for _ in range(n):
         d0 = time.perf_counter(); f(chemin); t = min(t, time.perf_counter() -
d0)
     return t
 t_flush, t_groupe = meilleur(ecrire_flush, os.path.join(d, "c.txt")), meilleur
(ecrire_groupe, os.path.join(d, "d.txt"))
 print(f"flush à chaque ligne : {t_flush * 1000:7.1f} ms | écriture groupée : {
t_groupe * 1000:6.1f} ms (x{t_flush / t_groupe:.0f})")
```

Exemple pas à pas (suite)

```
    assert t_flush > 3 * t_groupe

    # 4. lecture par blocs de différentes tailles : nombre d'appels
    chemin = os.path.join(d, "b.txt")
    for taille in (1, 64, 4096, 65536):
        n = 0
        with open(chemin, "rb", buffering=0) as f:
            while f.read(taille):
                n += 1
                if taille == 1 and n > 5000: break
        total = os.path.getsize(chemin)
        print(f"  blocs de {taille:6d} octets : {(-(-total // taille)):8d} lecture
s pour {total} octets")

    # 5. formats : taille d'un même jeu de données
    enregistrements = [{"membre": f"m{i % 800}", "cours": f"HEF-{i % 90:03d}", "mo
ntant": (i * 37) % 200} for i in range(20000)]
    en_json = json.dumps(enregistrements).encode()
    en_csv = "".join(f"{e['membre']};{e['cours']};{e['montant']}\n" for e in enreg
```

Exemple pas à pas

Exemple pas à pas (suite)

```
    istrements).encode()
     en_binaire = b"".join(struct.pack("<HBB", int(e["membre"][1:]), int(e["cours"]
    [4:]), e["montant"]) for e in enregistrements)
     for nom, donnees in (("JSON", en_json), ("CSV", en_csv), ("binaire (struct)",
    en_binaire)):
         z1, z9 = zlib.compress(donnees, 1), zlib.compress(donnees, 9)
         print(f"  {nom:17s} brut {len(donnees):8d} o | zlib niveau 1 : {len(z1):7d
    } o | niveau 9 : {len(z9):7d} o")
     assert len(en_binaire) < len(en_csv) < len(en_json)
     assert len(zlib.compress(en_json, 9)) < len(en_json) / 5

     # 6. écriture atomique : on ne voit jamais un fichier à moitié écrit
     def ecrire_atomique(chemin, texte):
         temporaire = chemin + ".tmp"
         with open(temporaire, "w") as f:
             f.write(texte); f.flush(); os.fsync(f.fileno())
         os.replace(temporaire, chemin)                          # remplacement atomiq
    ue
     cible = os.path.join(d, "etat.json")
```

Exemple pas à pas (suite)

```
    ecrire_atomique(cible, '{"version": 1}')
    ecrire_atomique(cible, '{"version": 2}')
    assert json.load(open(cible)) == {"version": 2} and not os.path.exists(cible +
    ".tmp")
    print("écriture atomique : OK")
```

Lecture :

1. Sans tampon : **20 000 appels système** pour 20 000 lignes. Avec un tampon de 64 Ko : moins de 100 appels, même contenu octet pour octet.
2. Flusher à chaque ligne est **plusieurs fois plus lent** que d'écrire en un bloc, même sur un disque local rapide ; sur un disque réseau, l'écart se compte en dizaines de fois.
3. Plus la taille de bloc est petite, plus le nombre de lectures explose (1 octet par lecture → des centaines de milliers d'appels).

Exemple pas à pas (suite)

4. **Format** : le binaire est le plus compact, puis CSV, puis JSON (noms répétés). La compression `zlib` réduit le JSON de plus de cinq fois : les données répétitives se compressent très bien.
5. L'**écriture atomique** (fichier temporaire, `fsync`, `os.replace`) garantit qu'un lecteur ne voit jamais un état partiel.

À toi, avec aide

Mesure l'effet de la **taille du tampon de lecture** sur la lecture d'un fichier de 20 Mo, pour des tailles de 1 Ko, 16 Ko, 256 Ko, 4 Mo. Compte le nombre d'appels `read` (fichier non tamponné + compteur) et le temps. Quelle taille choisirais-tu, et pourquoi ne pas simplement prendre la plus grande ?

Indices progressifs

Indice 1

Ouvre avec `open(chemin, 'rb', buffering=0)` et compte tes propres appels à `read(taille)`.

Indice 2

Génère le fichier avec des octets pseudo-aléatoires de graine fixée.

Indice 3

Au-delà d'une certaine taille, le gain est nul.

Réponse

```
import os, random, tempfile, time
with tempfile.TemporaryDirectory() as d:
    chemin = os.path.join(d, "donnees.bin")
    rng = random.Random(1)
    with open(chemin, "wb") as f:
        for _ in range(20):
            f.write(rng.randbytes(1024 * 1024))
    for taille in (1024, 16 * 1024, 256 * 1024, 4 * 1024 * 1024):
        meilleur, appels = float("inf"), 0
        for _ in range(3):
            appels = 0; debut = time.perf_counter()
            with open(chemin, "rb", buffering=0) as f:
                while f.read(taille):
                    appels += 1
            meilleur = min(meilleur, time.perf_counter() - debut)
        print(f"bloc {taille // 1024:5d} Ko : {appels:6d} lectures, {meilleur * 10
   00:7.1f} ms")
```

Réponse (suite)

Le nombre d'appels diminue proportionnellement à la taille du bloc (20 480 pour 1 Ko, 5 pour 4 Mo) mais le temps **plafonne** vers 64-256 Ko : au-delà, l'appel système n'est plus le facteur dominant, et un bloc énorme **consomme de la mémoire** et peut dégrader l'utilisation du cache du processeur. On choisit en pratique **64 Ko à 1 Mo**.

À toi, en solo

Un petit journal d'écritures (write-ahead log) durable. Implémente `JournalDurable` :

1. `ajouter(enregistrement)` écrit une ligne JSON avec somme de contrôle (CRC32) ;
2. écritures **groupées** : `fsync` seulement tous les N enregistrements ou à `valider()` (compromis latence/durabilité) ;
3. `relire()` renvoie les enregistrements valides et **s'arrête** au premier enregistrement corrompu ou incomplet (cas d'une coupure en cours d'écriture) ;
4. simule une coupure en tronquant le fichier au milieu de la dernière ligne, et vérifie que `relire()` renvoie tout sauf la dernière ;
5. compte les `fsync` pour 10 000 enregistrements avec N = 1, 100, 1000 et conclus sur le compromis.

À toi, en solo · réponse

Pistes de correction

```
import json, os, tempfile, zlib

class JournalDurable:
    def __init__(self, chemin, fsync_tous_les=100):
        self.chemin, self.n_fsync, self.fsyncs, self.en_attente = chemin, fsync_to
   us_les, 0, 0
        self.f = open(chemin, "ab")
    def _synchroniser(self):
        self.f.flush(); os.fsync(self.f.fileno()); self.fsyncs += 1; self.en_atten
   te = 0
    def ajouter(self, enregistrement):
        corps = json.dumps(enregistrement, sort_keys=True)
        self.f.write(f"{zlib.crc32(corps.encode()):08x} {corps}\n".encode())
        self.en_attente += 1
        if self.en_attente >= self.n_fsync:
            self._synchroniser()
    def valider(self):
        if self.en_attente:
            self._synchroniser()
```

À toi, en solo · réponse

Pistes de correction (suite)

```
    def fermer(self):
        self.valider(); self.f.close()

def relire(chemin):
    valides = []
    with open(chemin, "rb") as f:
        for ligne in f:
            try:
                texte = ligne.decode()
                if not texte.endswith("\n"):
                    break                                   # lign
   e incomplète
                crc, corps = texte[:-1].split(" ", 1)
                if int(crc, 16) != zlib.crc32(corps.encode()):
                    break                                   # corr
   uption
                valides.append(json.loads(corps))
            except (ValueError, UnicodeDecodeError):
                break
```

Pistes de correction (suite)

```
    return valides

with tempfile.TemporaryDirectory() as d:
    chemin = os.path.join(d, "wal.log")
    j = JournalDurable(chemin, fsync_tous_les=10)
    for i in range(25): j.ajouter({"n": i, "action": "inscription"})
    j.fermer()
    assert [r["n"] for r in relire(chemin)] == list(range(25))
    taille = os.path.getsize(chemin)
    with open(chemin, "r+b") as f:
        f.truncate(taille - 7)                                  # coupur
   e en pleine écriture de la dernière ligne
    reste = relire(chemin)
    print("après coupure :", len(reste), "enregistrements valides sur 25")
    assert [r["n"] for r in reste] == list(range(24))

    for n in (1, 100, 1000):
        j = JournalDurable(os.path.join(d, f"w{n}.log"), fsync_tous_les=n)
        for i in range(10000): j.ajouter({"n": i})
```

À toi, en solo · réponse

Pistes de correction (suite)

```
        j.fermer()
        print(f"  fsync tous les {n:4d} enregistrements → {j.fsyncs:5d} fsync pour
    10 000 enregistrements")
```

Compromis : `fsync` à chaque enregistrement garantit qu'**aucune donnée acceptée n'est perdue** mais coûte ~1 ms (10 s pour 10 000) ; tous les 1000 enregistrements, c'est ~1000 fois moins cher, mais on peut perdre jusqu'à 999 enregistrements en cas de coupure. Les bases de données proposent ce réglage (*group commit*). Le **CRC** détecte les écritures partielles.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
`flush()` ou `fsync()` à chaque ligne	Des milliers d'appels coûteux	Regrouper ; synchroniser par lot
Lire octet par octet	Appels système en masse	Blocs de 64 Ko à 1 Mo, tampon
Écrire un fichier en place	Lecteur voyant un état partiel, perte en cas de panne	Fichier temporaire + `fsync` + `os.replace`
Format texte verbeux pour de gros volumes	Taille, temps de lecture et d'analyse	Binaire ou colonnaire, compression adaptée

Mini-boss

Mini-boss

Optimiser l'export des factures. Écris un exporteur lent (200 000 factures, une ligne JSON par facture, `flush` à chaque ligne, lecture octet par octet pour recompter), puis améliore-le en 5 étapes mesurées :

1. écriture groupée avec tampon ;
2. format plus compact (CSV ou binaire) ;
3. compression adaptée (justifie le niveau) ;
4. écriture atomique du fichier final ;
5. lecture par blocs et projection `mmap` pour retrouver une facture par son numéro grâce à un index d'offsets.

Pour chaque étape : appels système comptés, taille, temps (meilleur de 5), facteur cumulé. Termine par un test de relecture complète prouvant que les données sont identiques et une conclusion sur ce qui compte le plus (nombre d'appels, taille, format).

Mini-boss · réponse

Pistes de correction

Un bon travail : fichier instrumenté pour compter les appels, tableau final avec des gains cumulés (nombre d'appels divisé par plus de 1000, taille divisée par 5 à 10 avec binaire + zlib), index d'offsets (liste d'`int`) et `mmap` pour accéder à la facture n en O(1) sans lire le reste. La conclusion attendue : le facteur dominant est le **nombre d'opérations d'E/S**, puis la **taille** des données, puis le format.

Quiz éclair

Quiz éclair

1. Pour accélérer l'écriture d'un gros volume de lignes, on :

A. appelle flush après chaque ligne
B. écrit octet par octet
C. regroupe les écritures dans un tampon

2. Une écriture atomique de fichier se fait en :

A. écrivant dans un fichier temporaire puis en le renommant
B. écrivant directement sur place
C. désactivant le disque

3. Compresser des données :

A. est toujours plus rapide

Quiz éclair

Quiz éclair (suite)

B. réduit la taille mais coûte du CPU
C. n'a jamais d'intérêt

4. 🔁 *Révision mission 04* : Pour traiter un fichier plus gros que la mémoire, on utilise :

A. une liste de toutes les lignes
B. un second écran
C. un flux ou un générateur, ligne par ligne

5. 🔁 *Révision mission 04* : Une fuite mémoire en Python correspond le plus souvent à :

A. des objets encore référencés (cache, historique) qu'on n'utilise plus
B. un manque de RAM sur la machine
C. une erreur de syntaxe

Quiz éclair · réponse

Réponses

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

Je réfléchis

1. Dans ton code, où écris-tu ou lis-tu en petites quantités un grand nombre de fois ?
2. Quelle donnée critique de ton application pourrait être perdue en cas de coupure de courant maintenant ?

À retenir

À retenir

1. Tampon → appel système → cache de pages → disque.
2. Moins d'appels, plus gros, plus séquentiels.
3. `flush` (vers le noyau) contre `fsync` (vers le disque) : à utiliser par lot.
4. Formats binaires et compression : taille contre CPU.
5. Écriture atomique et journal avec somme de contrôle pour la durabilité.

MISSION ACCOMPLIE

+100 XP

Badge débloqué : Plombier des entrées-sorties

Prochaine mission : Réseau et latence

