Réponses au TD — mémoire

Yahdhih ABDEL WEDOUD

5 octobre 2025

1 Dans quelle fonction est codé cet algorithme ? (fichier + ligne)

Réponse : Dans la fonction `benchRead` du fichier `bench.c`, autour de la ligne **121**.
Justification :

- Pré-chauffe : macro `PRE_CHAUFFE_DU_CACHE`.
- Chronométrage : `LANCER_CHRONO`/`ARRETER_CHRONO` (`gettimeofday()`).
- Boucles : externe (tailles) et interne (parcours du tampon avec `STEP_SIZE=16`, défini dans `bench.h`).

2 Fonctionnement du benchmark et but

But : Mettre en évidence la **hiérarchie mémoire** (L1/L2/L3/RAM) via le *temps moyen* d'une lecture dans un tampon de taille croissante.
Principe : Pour chaque taille, le code lit de nombreux éléments répartis dans le buffer, calcule la moyenne, puis passe à la taille suivante. Les **paliers** correspondent aux caches, les **sauts** à leur dépassement.

3 Pourquoi « pré-chauffer » le cache ?

On préchauffe le cache afin de stocker directement les données qu'on a besoin dedans, afin de ne pas les chercher directement dans la RAM à chaque fois qu'on en a besoin, car cela prend du temps. Les données présentes dans le premier niveau de cache (L1) sont des données qui sont en accès rapide par le processeur, ce niveau de cache a une petite taille. Le deuxième niveau de cache L2 est plus grand que L1 et son temps d'accès est un peu supérieure à celui de L1. Le troisième niveau de cache L3 est beaucoup plus grand que L2 et son temps d'accès est aussi supérieur à celui de L2.

L1 est le niveau le plus proche du/des processeur(s), ensuite il y a L2, puis L3 et après il y a la RAM.

4 Exécuter le benchmark et tracer la courbe

1) Compiler

```
make bench_malloc
```

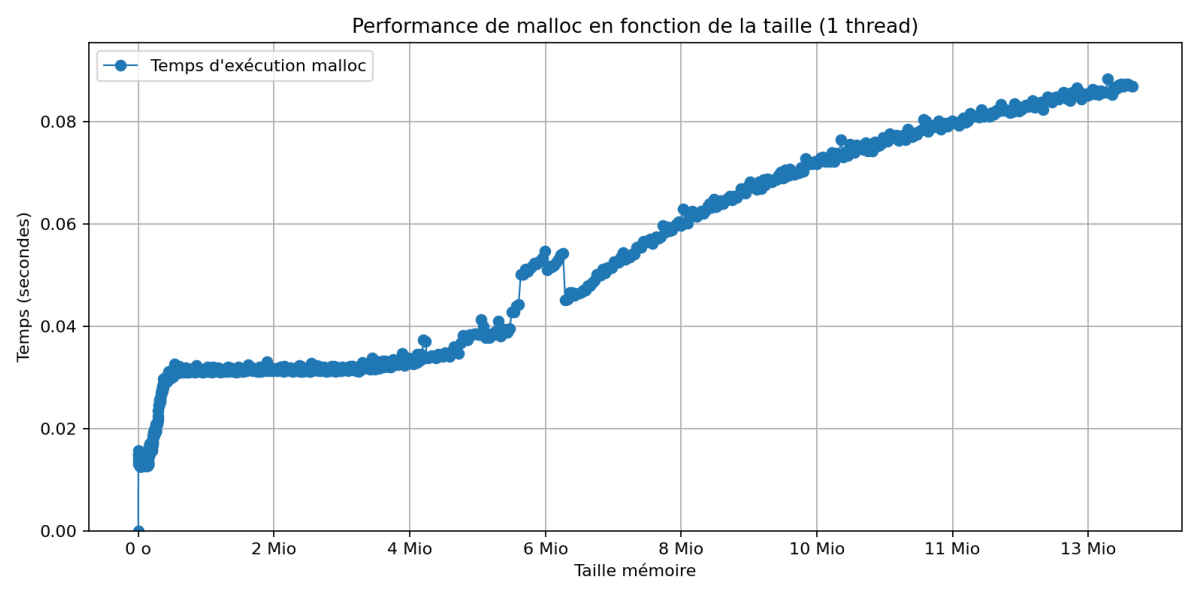
2) Exécuter et produire un CSV

```
./bench_malloc > result_malloc.csv
```

3) Tracer avec Python

```
python3 courbe.py result_malloc.csv
# Serveur/headless :
python3 courbe.py --no-show result_malloc.csv
```

Graphe généré :



5 Analyse de la courbe et rôle du cache

La courbe montre une progression en "escaliers" typique d'une hiérarchie mémoire. Pour de petites tailles, le temps reste bas et quasi plat : les données tiennent dans les caches proches (L1/L2, puis L3). On observe ensuite une rupture nette autour de ∼6 Mio : dépassement du dernier cache utile (probable L3). Au-delà, la pente devient quasi linéaire : on entre en régime RAM avec des latences plus élevées. Le pas d'accès fixe (stride) accentue les conflits de cache et rend les marches plus visibles. Les petites irrégularités près du seuil (6–6,5 Mio) viennent d'effets d'associativité, de prélecture et de TLB misses. En résumé : $<$ 6 Mio $\Rightarrow$ caches efficaces ; $\approx$ 6 Mio $\Rightarrow$ bascule ; $>$ 6 Mio $\Rightarrow$ coût DRAM/TLB dominant. Réduire le stride ou activer des huge pages lisserait la partie "grandes tailles".

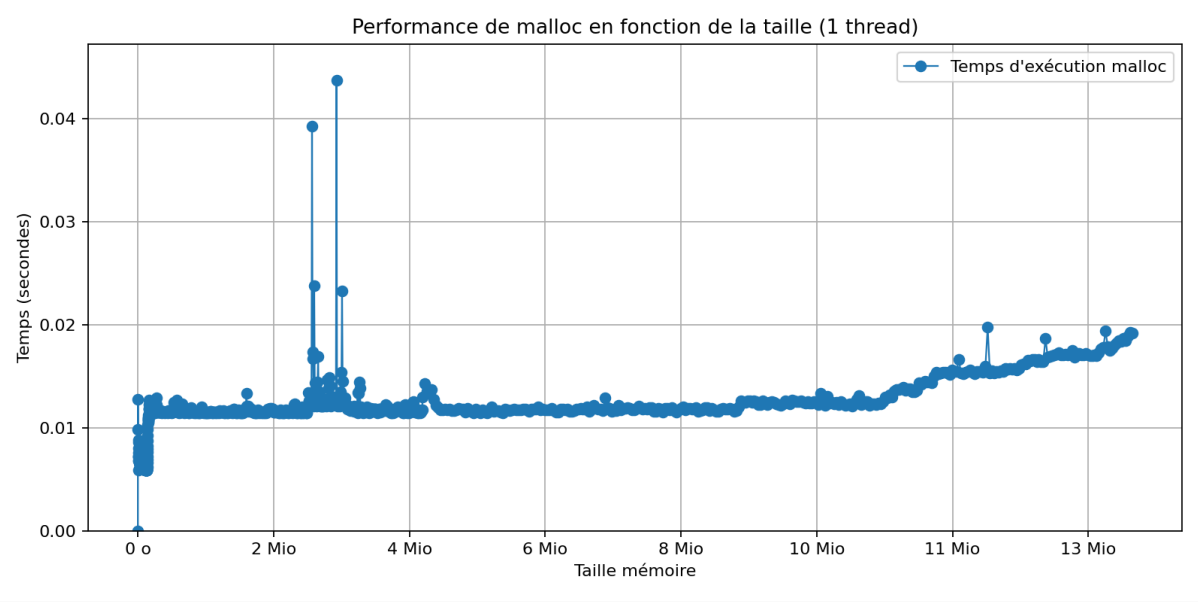
6 Qu'est-ce qu'une huge page ?

Une page mémoire **physique** de grande taille (p.ex. 2 MiB ou 1 GiB) au lieu des 4 KiB. Moins d'entrées TLB pour couvrir une zone donnée $\Rightarrow$ **moins de TLB misses**.

7 Activer les huge pages et relancer le bench

```
# Linux
echo 200 | sudo tee /proc/sys/vm/nr_hugepages
sudo mkdir -p /mnt/huge
sudo mount -t hugetlbfs -o mode=0777 none /mnt/huge
./bench_huge > result_huge.csv
```

Graphe généré (huge pages) :



Attendu : courbe plus **basse/stable** sur grandes tailles (moins de TLB misses).

8 Quand (ne pas) utiliser les huge pages ?

Après activation des huge pages, la courbe devient globalement **plus basse et plus stable** : on réduit les **TLB misses**, donc le coût par accès diminue. Les petites tailles restent sur un palier bas (les données tiennent en cache), puis la montée est **plus douce** jusqu'à plusieurs Mio. On observe quelques **pics isolés** vers ∼2–4 Mio : événements ponctuels (allocation/zeroing d'une huge page, alignement défavorable, conflit transitoire) plutôt que tendance. Au-delà de ∼6 Mio, la pente reste raisonnable : l'empreinte sort du dernier cache utile, mais les pages de 2 MiB limitent la pression sur le TLB. Un **léger relèvement** apparaît autour de 11–13 Mio, signe d'un régime davantage *RAM-bound* (latence DRAM, prélecture moins efficace). La variabilité fine est faible hors des pics, indiquant une *footprint* mieux "carrelée" par de grosses pages. En résumé : les huge pages **abaissent** la courbe et **lissent** les grandes tailles, les rares spikes d'allocation ayant peu d'impact structurel. Pour lisser encore, conserver un *stride* constant, répéter les mesures et reporter la **médiane** par taille.

9 Problèmes rencontrés avec l'utilisation des huges pages

- **Fragmentation mémoire** : réserver de grands blocs contigus (2 MiB/1 GiB) peut échouer si la RAM est fragmentée.
- **Réservation statique** : configuration via `nr_hugepages` immobilise de la mémoire au boot, indisponible pour le reste du système si elle n'est pas utilisée.
- **Fragmentation interne** : un besoin de 4 Ko occupe une huge page entière (2 MiB) ⇒ gaspillage possible.
- **Complexité d'exploitation** : nécessite un montage `hugetlbfs`, des droits/paramètres noyau et un code adapté (`mmap` aligné), donc déploiement/CI plus compliqués.
- **Portabilité/compatibilité** : indisponible sur certains OS/environnements (p. ex. macOS), et certaines bibliothèques ne gèrent pas bien ces pages.
- **NUMA & tuning** : une mauvaise affinité CPU/mémoire ou un mauvais dimensionnement

du pool peut dégrader les perfs au lieu de les améliorer.

10 Utilité de hp$_i$nit()ethp$_f$inalize()

— `hp_init()` — *Initialisation de l'allocateur*
Cette fonction initialise l'allocateur de mémoire basé sur les *huge pages*. Elle effectue les opérations suivantes :

1. **Ouvre un fichier dans `hugetlbfs`** :

   ```
   hp_fd = open(FILE_NAME, O_CREAT | O_RDWR, 0755);
   ```

   Crée/ou ouvre le fichier `/mnt/huge/bench` dans le système de fichiers spécial pour les huge pages.
2. **Mappe une grande zone de mémoire (14 MiB)** :

   ```
   reserve.mem = mmap(0, MEM_SIZE, PROTECTION, FLAGS, hp_fd, 0);
   ```

   Utilise `mmap()` pour créer un mapping mémoire de **14 MiB** (défini par `MEM_SIZE`). Cette zone constituera une "tas" pour toutes les allocations de l'allocateur.
3. **Gère les erreurs** : si `mmap()` échoue → affiche un message d'erreur et quitte.
   Si `open()` échoue → libère les ressources et quitte.

Résultat : à la fin de `hp_init()`, une grande zone de mémoire contiguë en *huge pages* est prête à être utilisée par l'allocateur.

— `hp_finalize()` — *Nettoyage / Libération*
Cette fonction libère proprement les ressources allouées par `hp_init()`. Elle effectue :

1. **Démappage de la mémoire** :

   ```
   munmap(reserve.mem, MEM_SIZE);
   ```

   Libère la zone mémoire de **14 MiB**.
2. **Fermeture du descripteur de fichier** :

   ```
   close(hp_fd);
   ```
3. **Suppression du fichier temporaire** :

   ```
   unlink(FILE_NAME);
   ```

   Supprime `/mnt/huge/bench` du système de fichiers.

Résultat : toutes les ressources système sont proprement libérées, sans fuite mémoire.

11 Utilité de `alloue` et la cause de sa déclaration en `static`

-Rôle de `alloue` : `alloue` est un **curseur** (ou « pointeur de position ») qui mémorise la **quantité totale de mémoire déjà distribuée** par l'allocateur dans la grande zone pré-allouée (`reserve.mem`).
Idée : on avance ce curseur à chaque allocation (stratégie *bump allocator*).
Fonctionnement détaillé :

```c
static void* __hp_malloc(size_t taille) {
    static size_t alloue = 0U; // Curseur de position (persistant)
    // ...
}
```

1. **Au démarrage :** `alloue = 0`.

Toute la zone (ex. 14 MiB) est libre, `reserve.mem` pointe sur le début.

2. **Première allocation** (ex. 100 octets) :
   — on renvoie l'adresse `reserve.mem + alloue` ⇒ début de zone,
   — on fait `alloue += 100`.
3. **Deuxième allocation** (ex. 200 octets) :
   — on renvoie `reserve.mem + 100`,
   — puis `alloue += 200` ⇒ `alloue = 300`.
4. **Calcul de l'adresse retournée :**

```
void* ret = (void*)((char*)reserve.mem + alloue);
alloue += taille;     // on avance le curseur
return ret;
```

En résumé : `alloue` indique où commence la **zone libre** suivante dans le grand bloc ; c'est un compteur qui augmente à chaque appel.

-Pourquoi `static` ? Deux effets essentiels.

1) Persistance entre appels. Sans `static` (variable locale normale) : `alloue` est réinitialisée à 0 à chaque appel ⇒ chevauchements et corruption.

```
//  Sans static : rinitialise  0  chaque appel (BUG)
void* __hp_malloc(size_t taille) {
    size_t alloue = 0U;  // <-- repart  0  chaque appel !
    // 1er appel : alloue = 0 -> OK
    // 2e appel : alloue = 0 -> ralloue au mme endroit -> donnes crases
}
```

Avec `static` (locale statique) : initialisée **une seule fois**, puis conservée pour toute la durée du programme.

```
//  Avec static : persiste entre appels
void* __hp_malloc(size_t taille) {
    static size_t alloue = 0U;
    // 1er appel : ret = base + 0, alloue = 100  (ex.)
    // 2e appel : ret = base + 100, alloue = 300 (ex.)
    // Chaque allocation suit la prcdente, sans chevauchement.
}
```

2) Portée limitée (encapsulation). `static size`$_t$ $alloue = 0U$; ⇒ `variable locale à la fonction (non visible en dehors), mais persistante.`
`Avantages : protection contre les modifications externes, pas de pollution d'espace de noms, débogage plus simple.`

Comparaison avec une variable globale.

- `Globale (size_t alloue_global;) : visible partout ⇒ risque de modification accidentelle (mauvaise pratique ici).`
- `Locale static : persiste` *et* `reste encapsulée ⇒ bonne pratique pour ce compteur interne.`

12 Q.12 : Explication du fonctionnement de la fonction `__hp_malloc`

Réponse — La fonction `__hp_malloc` est un **allocateur de mémoire simple** de type *bump allocator* (allocateur par avancement de curseur). Voici son fonctionnement étape par étape :

Algorithme :

1. **Vérification de la taille demandée :**

```
assert(taille > 0);
```

But : s'assurer qu'on demande au moins 1 octet.

2. **Alignement de la taille :**

```
taille = calcule_multiple_aligne(taille);
```

Idée : arrondir la taille au **multiple supérieur** de l'alignement matériel (généralement 8 ou 16 octets).
Pourquoi ? Pour garantir que tous les pointeurs retournés seront correctement alignés en mémoire.
Exemple : si on demande 13 octets avec un alignement de 8, la taille devient 16.

3. **Vérification de l'espace disponible :**

```
if (alloue + taille > MEM_SIZE) {
    fprintf(stderr, "Il n'y a plus assez de mmoire disponible.\n");
    return NULL;
}
```

But : vérifier s'il reste assez de place dans le grand bloc (ex. 14 MiB). Sinon, `NULL`.

4. **Calcul de l'adresse de retour :**

```
void* ret = (void*)((char*)reserve.mem + alloue);
```

Interprétation : l'adresse de retour est **base + offset courant**.
C'est la première adresse libre dans la zone.

5. **Mise à jour du curseur :**

```
alloue += taille;   // on avance le curseur
```

Conséquence : la prochaine allocation commencera juste après.

6. **Retour du pointeur :**

```
return ret;
```

13 Implémentation la fonction `__hp_malloc`

Réponse :

```
static void* __hp_malloc(size_t taille)
{
    /*
     * Alloue un bloc de mmoire au moins de taille `taille`.
     */

    static size_t alloue = 0U;   // Curseur de position (initialis une fois)
    void* ret;
```

```
    assert(taille > 0);

    // Arrondir la taille  l'alignement requis
    taille = calcule_multiple_aligne(taille);

    // Vrifier s'il reste assez de mmoire
    if (alloue + taille > MEM_SIZE) {
        fprintf(stderr, "Il n'y a plus assez de mmoire disponible.\n");
        return NULL;
    }

    // Calculer l'adresse de retour (dbut de la zone libre)
    ret = (void*)((char*)reserve.mem + alloue);

    // Avancer le curseur
    alloue += taille;

    return ret;
}
```

Explication ligne par ligne :

Ligne	Explication
`static size_t alloue = 0U;`	Curseur qui garde la trace de la mémoire déjà allouée. Initialisée à 0 au premier appel, puis persiste (variable locale `static`).
`assert(taille > 0);`	Sécurité : refuse les allocations de 0 octet.
`taille = calcule_multiple_aligne(taille);`	Arrondit pour garantir l'alignement mémoire.
`if (alloue + taille > MEM_SIZE)`	Vérifie qu'on ne dépasse pas les 14 MiB disponibles.
`ret = (void*)((char*)reserve.mem + alloue);`	Calcule l'adresse du début de la zone libre (*base* + *offset*).
`alloue += taille;`	Avance le curseur pour la prochaine allocation.
`return ret;`	Retourne le pointeur vers la mémoire allouée.

14 Le problème de cette implémentation

Réponse — Le problème majeur est qu'il **n'y a aucun mécanisme de libération** (`free`). Conséquences :

1. **Impossible de réutiliser la mémoire.**
   Une fois allouée, la mémoire reste occupée pour toute la durée du programme.

```
int *p1 = __hp_malloc(1000);   // Alloue 1000 octets
/* ... utilisation de p1 ... */
// __hp_free(p1);   //  Cette fonction n'existe pas

int *p2 = __hp_malloc(500);    // Alloue 500 NOUVEAUX octets
// Les 1000 octets de p1 ne sont JAMAIS rutiliss !
```

2. **Gaspillage massif (fuite mémoire).**
   À chaque appel, on « perd » le bloc précédent s'il n'est plus utile.

```c
// Scnario catastrophique :
for (int i = 0; i < 10000; i++) {
    char *temp = __hp_malloc(1024); // Alloue 1 KiB
    /* ... utiliser temp ... */
    // Pas de free !
}
// => Aprs ~10000 itrations, PLUS DE MMOIRE !
```

3. **Utilisable seulement dans des cas très spécifiques.**
   - *OK* : programme qui alloue toute sa mémoire au début, l'utilise, puis se termine.
   - *Pas OK* : serveur web, base de données, ou tout programme qui alloue/libère dynamiquement sur une longue durée.

15 Explication du fonctionnement de la fonction `recherche_bloc_libre`

Réponse — La fonction `recherche_bloc_libre` implémente une stratégie de recherche **First-Fit** (premier ajustement). Elle parcourt la **liste chaînée de blocs libres** et renvoie le **premier bloc suffisamment grand** pour satisfaire la demande.

Algorithme détaillé :

1) *Initialisation des pointeurs :*

```c
struct bloc *bloc = libre;     // Pointe vers le dbut de la liste des libres
struct bloc *precedent = NULL; // Mmorise le bloc prcdent
struct bloc *ret = NULL;       // Bloc  retourner
```

2) *Parcours de la liste chaînée :*

```c
while (bloc != NULL) {
```

On continue tant qu'il reste des blocs à examiner.

3) *Test de taille suffisante :*

```c
    if (bloc->taille >= taille) {
        // Vrifie si le bloc courant peut satisfaire la demande.
```

4) *Retrait du bloc de la liste (si trouvé) :*

```c
        if (precedent != NULL)      // Cas gnral (bloc au milieu/fin)
            precedent->suivant = bloc->suivant;
        else                        // Cas particulier : bloc en tte
            libre = bloc->suivant;
```

Le bloc trouvé est "détaché" de la liste des libres ; la chaîne reste continue.

5) *Sauvegarde et sortie :*

```c
        ret = bloc;                 // Premire occurrence convenable
        break;                      // On s'arrte (First-Fit)
    }
```

6) *Avancement si le bloc est trop petit :*

```c
    precedent = bloc;
    bloc = bloc->suivant;
}
```

7) *Retour :*

```c
return ret;     // Bloc trouv, ou NULL si aucun ne convient
```

Schéma visuel (exemple) :

```
Liste des blocs libres AVANT recherche_bloc_libre(150) :

libre
[100]->[200]->[80]->[300]->NULL
  ^     ^            ^
  |     |            |
  trop  200 est le PREMIER >= 150  -> choisi

Liste APRES :

libre
[100]->[80]->[300]->NULL

// Le bloc [200] est retir de la liste des libres et sera rutilis.
```

Caractéristiques :

- **Stratégie First-Fit** : s'arrête dès le *premier* bloc convenable.
- **Rapide** : un seul passage séquentiel sur la liste.
- **Fragmentation possible** : ne cherche pas le bloc optimal (contrairement à Best-Fit).
- **Gaspillage potentiel** : un bloc plus grand que nécessaire peut être utilisé pour une petite demande.

16 Implémentation de `recherche_bloc_libre`

```c
// Dans hp_allocator.c
static struct bloc *recherche_bloc_libre(size_t taille) {
  struct bloc *ret = NULL;
  struct bloc *precedent = NULL;
  struct bloc *bloc = libre;

  while (bloc != NULL) {                  // parcours
    if (bloc->taille >= taille) {         // assez grand ?
      if (precedent != NULL)
        precedent->suivant = bloc->suivant;
      else
        libre = bloc->suivant;            // tte de liste
      ret = bloc;
      break;
    }
    precedent = bloc;
    bloc = bloc->suivant;
  }
  return ret;
}
```

17 Explication du fonctionnement de la fonction `__hp_malloc`

Réponse — Cette nouvelle version de `__hp_malloc` est plus évoluée que la version simple (Q.12–Q.13). Elle combine deux stratégies :

- **Réutilisation** de blocs déjà libérés (parcours de la *liste des libres*).

— **Bump allocation** (à la fin de zone) en *dernier recours*.

Algorithme détaillé

Étape 1 — Alignement de la taille

```
taille = calcule_multiple_aligne(taille);  // garantit l'alignement mmoire
```

Étape 2 — Recherche d'un bloc libre

```
struct bloc *bloc_trouve = recherche_bloc_libre(taille);
```

On tente d'abord de **réutiliser** un bloc précédemment libéré.

Étape 3a — Cas 1 : un bloc libre convient

```
if (bloc_trouve != NULL) {
    // On rutilise ce bloc directement : les donnes commencent aprs l'en-tte
    return (void *)((char *)bloc_trouve + TAILLE_EN_TETE);
}
```

Réutilisation réussie : pas d'allocation supplémentaire. Le pointeur renvoyé vise la **zone de données** (après l'en-tête).

Étape 3b — Cas 2 : aucun bloc libre ne convient

```
if (bloc_trouve == NULL) {
    // Vrifier la place restante pour une allocation en fin de zone
    if (MEM_SIZE - alloue < TAILLE_EN_TETE + taille) {
        fprintf(stderr, "Il n'y a plus assez de mmoire disponible.\n");
        return NULL;
    }

    // Allouer un nouveau bloc  la fin (bump allocation)
    struct bloc *bloc = (struct bloc *)((char *)reserve.mem + alloue);
    alloue += TAILLE_EN_TETE + taille;     // on consomme en-tte + donnes
    bloc_init(bloc, taille);               // initialise l'en-tte (taille, liens, etc
    .)

    // Retourner le pointeur vers la zone de donnes
    return (void *)((char *)bloc + TAILLE_EN_TETE);
}
```

Étape 4 — Retour `return (void*)((char*)bloc + TAILLE_EN_TETE);` renvoie toujours l'adresse de la **zone de données**, c.-à-d. *après l'en-tête*.

Résumé

1. Aligne la taille demandée.
2. Cherche un bloc libre suffisant (*First-Fit*) ; s'il existe, le réutilise.
3. Sinon, vérifie l'espace restant et alloue un *nouveau bloc* en fin de zone (bump).
4. Met à jour le curseur `alloue`, initialise l'en-tête, et renvoie l'adresse des données.

18 Implémentation de la fonction `__hp_malloc`

Réponse — Votre fonction `__hp_malloc` est déjà complètement implémentée. Voici le code final, avec commentaires intégrés :

```
void* __hp_malloc(size_t taille)
{
    /*
```

```
     * Alloue un bloc de mmoire d'au moins 'taille' octets.
     */

    static size_t alloue = 0U;   // Curseur de fin de zone (bump allocation)
    void* ret;

    assert(taille > 0);

    // 1) Aligner la taille demande
    taille = calcule_multiple_aligne(taille);

    // 2) Chercher d'abord dans les blocs libres (rutilisation)
    struct bloc* bloc_trouve = recherche_bloc_libre(taille);

    // 3) Si aucun bloc libre ne convient -> bump allocation (fin de zone)
    if (bloc_trouve == NULL) {
        // Vrifier l'espace restant (en-tte + donnes)
        if (MEM_SIZE - alloue < TAILLE_EN_TETE + taille) {
            fprintf(stderr, "Il n'y a plus assez de mmoire disponible.\n");
            return NULL;
        }

        // Crer un nouveau bloc  la fin de la zone
        bloc_trouve = (struct bloc *)((char *)reserve.mem + alloue);
        alloue += TAILLE_EN_TETE + taille;    // Avancer le curseur
        bloc_init(bloc_trouve, taille);       // Initialiser l'en-tte
    }

    // 4) Retourner l'adresse des donnes (aprs l'en-tte)
    return (void *)((char *)bloc_trouve + TAILLE_EN_TETE);
}
```

Points clés de l'implémentation :

1. **Alignement** : garantit que les adresses retournées sont correctement alignées.
2. **Réutilisation** : tente d'abord de réutiliser un bloc libre (liste des libres).
3. **Bump allocation** : en dernier recours, alloue en fin de zone.
4. **Vérification mémoire** : contrôle `MEM_SIZE - alloue >= TAILLE_EN_TETE + taille`.
5. **Adresse de retour** : toujours la *zone de données*, pas l'en-tête.
6. **Mise à jour du curseur** : avance de `TAILLE_EN_TETE + taille` après création du bloc.
7. **Robustesse** : `assert(taille > 0)` et message d'erreur si mémoire insuffisante.

19 Explication du fonctionnement de la fonction `__hp_free`

Réponse — `__hp_free` libère un bloc en l'ajoutant à la **liste chaînée des blocs libres**, pour qu'il puisse être **réutilisé** plus tard par `__hp_malloc`.

Algorithme détaillé
Étape 1 — Vérification du pointeur `NULL`

```
if (ptr == NULL)
    return;
```

Protection : comme `free(NULL)`, ne fait rien si le pointeur est nul.
Étape 2 — Retrouver l'adresse de l'en-tête

```
struct bloc *nouveau = (struct bloc *)((char *)ptr - TAILLE_EN_TETE);
```

Le pointeur utilisateur `ptr` vise la **zone de données**. On remonte à l'**en-tête** du bloc en soustrayant `TAILLE_EN_TETE`.

Étape 3 — Insertion en tête de la liste libre

```
nouveau->suivant = libre;  // Le nouveau bloc pointe vers l'ancienne tte
libre = nouveau;           // Le nouveau bloc devient la nouvelle tte
```

Insertion en $\mathcal{O}(1)$: pas besoin de parcourir la liste. Le bloc est ajouté **au début** de la liste chaînée des libres.

Résumé :

- `NULL` → no-op (comportement standard).
- Conversion `data → header` via `ptr - TAILLE_EN_TETE`.
- Ajout en tête de la liste libre ($\mathcal{O}(1)$), prêt pour une future réutilisation.

20 Implémentation de la fonction `__hp_free`

Réponse — Votre fonction `__hp_free` est complètement implémentée. Voici le code final et les explications ligne par ligne.

```
void __hp_free(void *ptr)
{
    /*
     * Ajoute le bloc fourni  la liste des blocs libres.
     */

    struct bloc *nouveau;

    // 1. Vrifier que le pointeur n'est pas NULL
    if (ptr == NULL)
        return;

    // 2. Retrouver l'adresse de l'en-tte
    nouveau = (struct bloc *)((char *)ptr - TAILLE_EN_TETE);

    // 3. Insertion en tte de la liste des blocs libres
    nouveau->suivant = libre;  // Nouveau bloc pointe vers l'ancienne tte
    libre = nouveau;           // Nouveau bloc devient la nouvelle tte
}
```

Détails d'implémentation :

Ligne	Explication
if (ptr == NULL) return;	Évite les erreurs : même comportement que `free(NULL)`.
nouveau = (struct bloc *)((char *)ptr - TAILLE_EN_TETE);	Calcule l'adresse de l'en-tête en reculant de `TAILLE_EN_TETE` octets depuis la zone de données. Le cast en `(char*)` permet l'arithmétique d'octets.
nouveau->suivant = libre;	Le bloc libéré pointe vers l'ancienne tête de la liste des libres.
libre = nouveau;	Le bloc libéré devient la nouvelle tête de liste.

21 Construire une bibliothèque dynamique de l'allocateur

Compilation en PIC puis en .so

```
gcc -Wall -Wextra -fPIC -c hp_allocator.c -o hp_allocator.o
gcc -shared hp_allocator.o -o libhp.so
```

Le fichier `libhp.so` est la bibliothèque dynamique exportant les fonctions de l'allocateur.

22 Utiliser la bibliothèque avec le bench via `LD_PRELOAD`

```
# Dans le dossier bench avec l'excutable compil
LD_PRELOAD=./libhp.so ./bench > result_lib.csv

# Tracer :
python3 courbe.py result_lib.csv
```

`LD_PRELOAD` force le chargeur dynamique à lier `malloc/free` depuis `libhp.so` avant la `libc`, selon ce que la bibliothèque exporte (voir Q23).

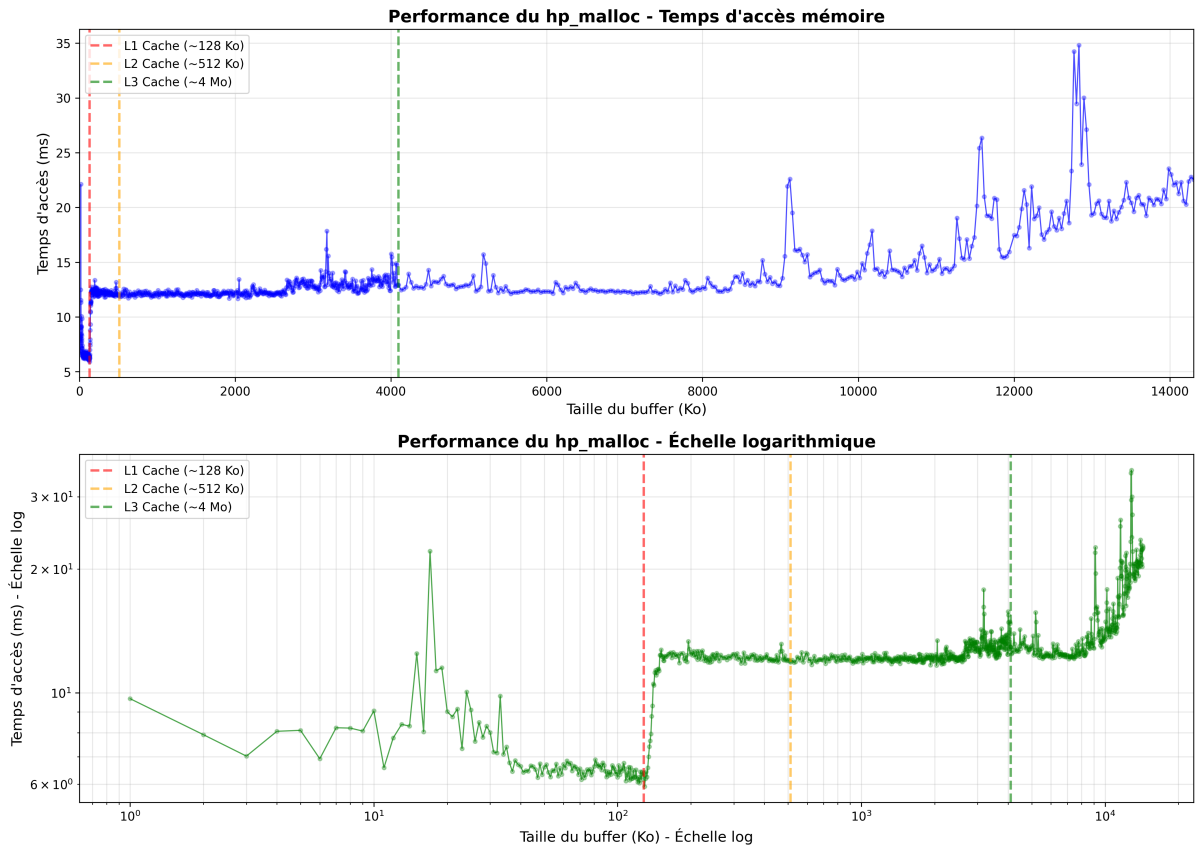


FIGURE 1 – Performance de `hp_malloc` : temps d'accès mémoire (linéaire en haut, log en bas). Les lignes pointillées indiquent L1 (128 KiB), L2 (512 KiB), L3 (4 MiB).

Interprétation (lecture du graphe).

- **Petits buffers (** L1**)** : latence faible et assez stable — tout tient en cache L1.
- **Autour de L1 → L2** : palier/hausse nette du temps d'accès quand la taille dépasse ∼128 KiB, signe que les accès quittent L1 pour L2.
- **Autour de L2 → L3** : nouveau palier autour de ∼512 KiB — on amortit sur L3.
- **Au-delà de L3** (4 MiB) : la latence et la variance augmentent (accès mémoire principale, plus de *miss*, bruit TLB/page faults éventuels). Les pics traduisent des passages de frontières d'alignement ou des effets de parcours.
- **Panneau log (bas)** : met en évidence les *régimes* de croissance et les cassures aux frontières de caches ; la tendance devient plus claire sur plusieurs ordres de grandeur.

23 Wrappe des fonction malloc et free

Le wrapper implémenté dans `malloc_wrapper.c` **intercepte** les appels à `malloc()` et `free()` pour les **rediriger** vers notre allocateur personnalisé (`hp_malloc`/`hp_free`) via le mécanisme `LD_PRELOAD`.

Architecture du wrapper

Principe de base :

```c
// Redirection simple des symboles
void *malloc(size_t size) {
    return hp_malloc(size);  // Utilise notre allocateur
}

void free(void *ptr) {
    hp_free(ptr);  // Utilise notre liberation
}
```

Fonctionnalites implementees

1. Gestion de l'initialisation :

```c
static int initializing = 0;
static int initialized = 0;
static char temp_buffer[1024];  // Buffer temporaire pour dlsym()

static void init(void) {
    if (initializing || initialized) return;
    initializing = 1;

    // Recuperer les vraies fonctions de la libc
    real_malloc = dlsym(RTLD_NEXT, "malloc");
    real_free = dlsym(RTLD_NEXT, "free");

    fprintf(stderr, "[WRAPPER] malloc/free interceptes\n");
    initialized = 1;
    initializing = 0;
}
```

Probleme resolu : `dlsym()` appelle `malloc()` en interne ⇒ risque de recursion infinie.
Solution : Buffer temporaire pendant l'initialisation.

2. Famille complete de fonctions Le wrapper implemente **toutes les variantes** d'allocation :

Fonction	Implementation
`malloc()`	Redirige vers `hp_malloc()`
`free()`	Redirige vers `hp_free()`
`calloc()`	`malloc()` + initialisation a zero
`realloc()`	Nouvel `malloc()` + copie des donnees
`posix_memalign()`	`malloc()` (deja aligne)
`aligned_alloc()`	`malloc()` (deja aligne)
`memalign()`	`malloc()` (deja aligne)
`valloc()`	`malloc()` (alignement page)

3. Mecanisme de fallback

```c
void *malloc(size_t size) {
    // Pendant l'initialisation : buffer temporaire
    if (initializing) {
        if (temp_buffer_offset + size < TEMP_BUFFER_SIZE) {
            void *ptr = temp_buffer + temp_buffer_offset;
            temp_buffer_offset += size;
            return ptr;
        }
    }
```

```
    // Utiliser hp_malloc
    void *ptr = hp_malloc(size);

    // Fallback vers libc si echec
    if (!ptr && real_malloc) {
        ptr = real_malloc(size);
    }

    return ptr;
}
```

4. Gestion de `free()`

```
void free(void *ptr) {
    // Ignorer les free du buffer temporaire
    if (ptr >= (void *)temp_buffer &&
        ptr < (void *)(temp_buffer + TEMP_BUFFER_SIZE)) {
        return;
    }

    // Ignorer NULL
    if (ptr == NULL) return;

    // Utiliser notre implementation
    hp_free(ptr);
}
```

Utilisation

Compilation :

```
gcc -fPIC -shared malloc_wrapper.c hp_allocator.c \
    -o libmalloc_wrapper.so -ldl
```

Execution avec interception :

```
# Avec votre allocateur
LD_PRELOAD=./libmalloc_wrapper.so ./bench > result_custom.csv

# Sans (libc standard)
./bench > result_libc.csv

# Tracer les deux courbes
python3 courbe.py result_libc.csv result_custom.csv
```

Avantages de cette approche

Transparent : Aucune modification du code du benchmark

Complet : Toutes les fonctions d'allocation sont gerees

Securise : Gestion de la recursion `dlsym()` → `malloc()`

Robuste : Fallback vers libc en cas d'echec

Realiste : Technique utilisee par jemalloc, tcmalloc, Valgrind

Flux d'execution

```
Programme benchmark
        |
   malloc(1024)  <- Appel dans le code
        |
LD_PRELOAD intercepte -> libmalloc_wrapper.so
        |
malloc() du wrapper
        |
hp_malloc(1024)  <- Votre allocateur avec huge pages
        |
Retour du pointeur
```

Points techniques importants

1. `_GNU_SOURCE` : Active `dlsym(RTLD_NEXT, ...)` pour trouver la libc
2. `dlsym(RTLD_NEXT)` : Recupere les vraies fonctions libc
3. **Buffer temporaire** : Evite la recursion infinie lors de l'init
4. **Verification NULL** : Protection contre `free(NULL)` et pointeurs invalides

Resultat

Le wrapper permet de **benchmarker votre allocateur** sur n'importe quel programme existant, sans recompilation, en remplacçant dynamiquement toutes les allocations par votre implementation avec huge pages.

```
# Message au lancement
[WRAPPER] malloc/free interceptes et rediriges vers hp_malloc/hp_free
```