Architecture Système d’Exploitation

TD2 - Interaction de la Pagination avec les Caches

Tàzio Gennuso

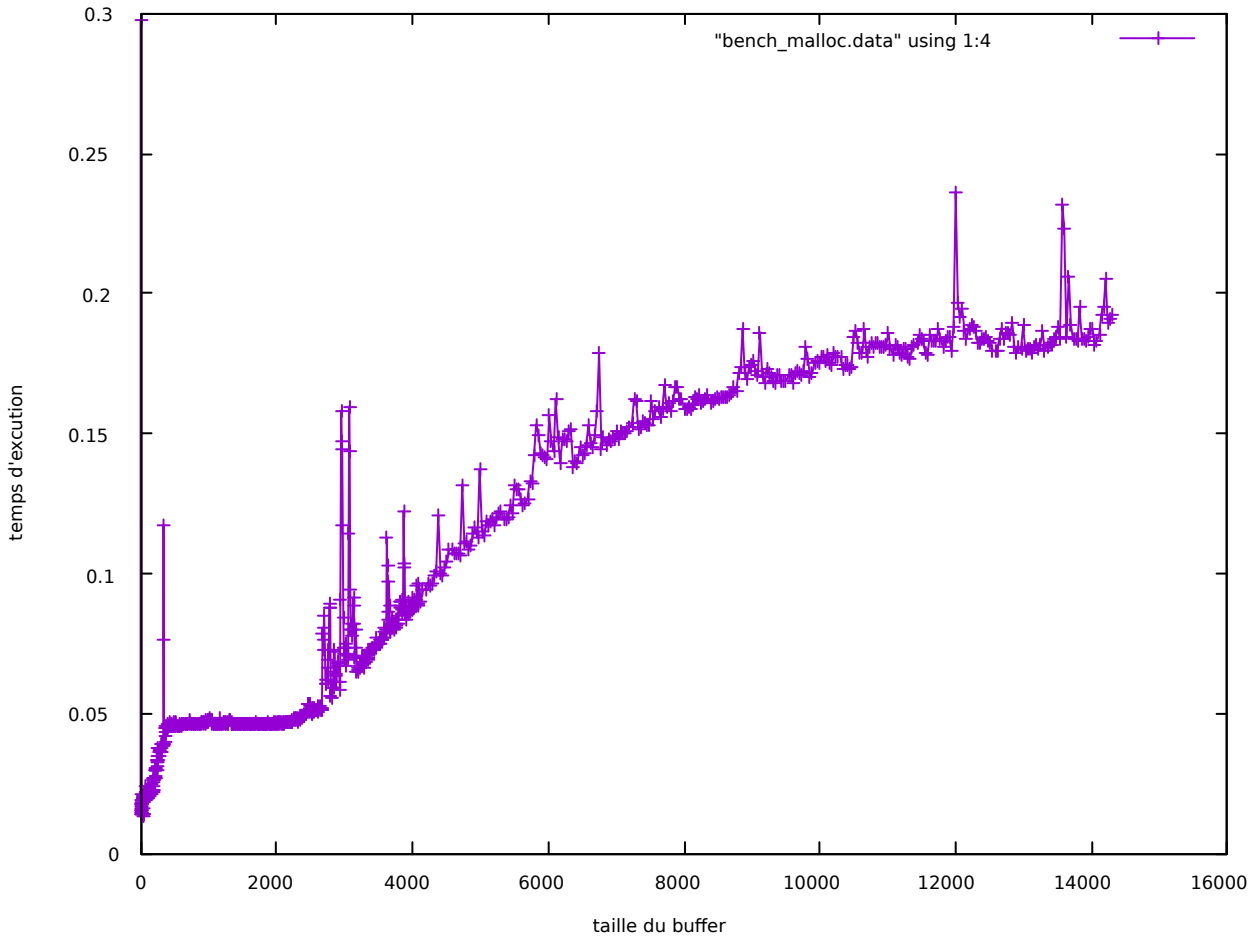
II – Analyse d’un Benchmark

Q.1 : Cette fonction est codée dans le fichier svalat_bench/bench.c, dans la fonction `benchRead()`, de la ligne 111 à 142.

Q.2 : Le benchmark chronomètre le temps de lecture du cache selon des paramètres fixés.

Q.3 : La préchauffe du cache sert à recréer artificiellement le contexte du problème, en remplissant le cache avant l’exécution des tests chronométrés.

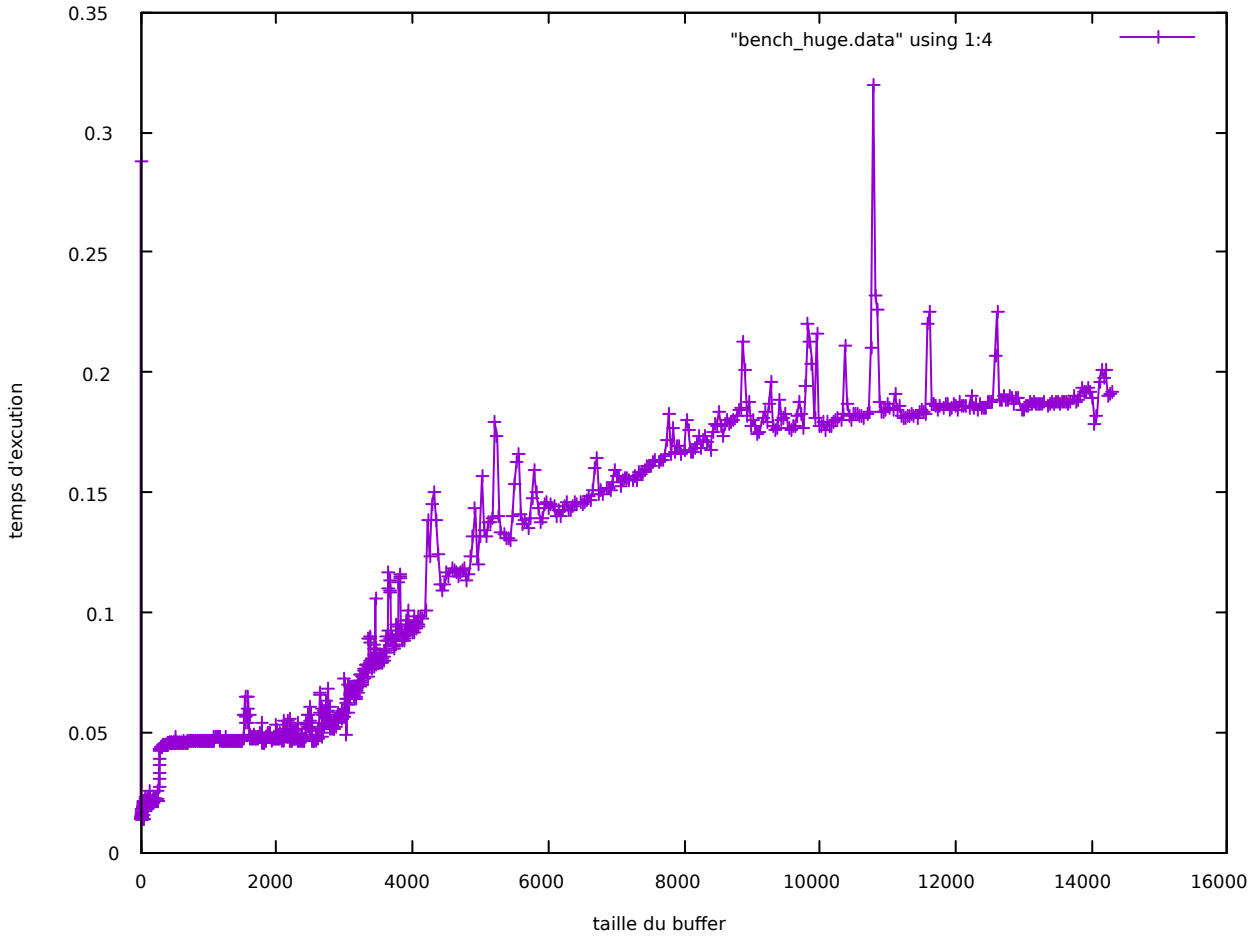
Q.4 : On trace la courbe avec gnuplot, avec la commande `plot "monfichier" using 1:4 with lp`.



Q.5 : On remarque que pour une taille de buffer inférieure à 2Mo, le temps d’exécution est constant (5ms), et augmente avec la taille au-delà. L’utilisation du cache n’est pas optimale puisque le temps d’exécution augmente alors qu’il n’est pas encore rempli.

Q.6 : Une *huge page* est un page considérablement plus grande que la taille standard.

Q.7 :



Q.8 : On remarque que pour une taille de buffer inférieure à 3Mo, le temps d'exécution est constant (5ms), et augmente avec la taille au-delà. L'utilisation du cache n'est pas optimale puisque le temps d'exécution augmente alors qu'il n'est pas encore rempli, cependant, cela est mieux que le cas précédent.

Q.9 : La mémoire perdue sur des huge pages est plus importante que sur des pages standards car il est plus difficile d'obtenir une quantité précise de mémoire avec de plus gros blocs unitaires.

III – Création d'un Allocateur en Mode Utilisateur

1 – Gestion du malloc

Q.10 : Les fonction `hp_init()` et `hp_finalize()` allouent et libère les *huges pages*. Elles sont appelées avant et après le main grâce à `__attribute__((constructor))` et `__attribute__((destructor))` qui indiquent cela au compilateur. Plus précisément, au moment du chargement (au #include associé) et du « déchargement » de la bibliothèque (à la sortie du programme).

Q.11 : La variable `alloue` contient la quantité de mémoire allouée. Elle est déclarée en static afin que sa valeur soit conservée d'un appel à l'autre à la fonction.

Q.12 : La fonction `__hp_malloc()` alloue de la mémoire parmi la mémoire préalablement réservée. Elle ne fait qu'allouer un bloc de la taille demandée à la suite de la mémoire déjà allouée.

Q.13 : On implémente les lignes manquantes de la façon suivante :

```
static void *__hp_malloc(size_t taille)
{
      static size_t alloue = 0UL;
      void *ret;
      assert(taille > 0);
      taille = calcule_multiple_align(taille);

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

      ret = &reserve.mem[alloue];
      alloue += taille;
      return ret;
}
```

Q.14 : Le problème de cette implémentation est qu' on ne libère jamais la mémoire, on se contente d'allouer à la suite.

2 – Gestion du free

Q.15 : `recherche_bloc_libre()` parcourt la liste chaînée des blocs jusqu'à trouver un bloc ou une suite de blocs de la taille voulue. On peut ainsi trouver la quantité de mémoire recherchée parmi de la mémoire qui a été désallouée précédemment.

Q.16 : On implémente les lignes manquantes de la façon suivante :

```
static struct bloc *recherche_bloc_libre(size_t taille)
{
    struct bloc *bloc = libre;
    struct bloc *precedent = NULL;
    struct bloc *ret = NULL;

    while (bloc != NULL)
    {
        if (bloc->taille >= taille)
        {
            if (precedent != NULL)
                precedent->suivant = bloc->suivant;
            else
                libre = bloc->suivant;

            ret = bloc;
            break;
        }
        precedent = bloc;
        bloc = bloc->suivant;
    }
    return ret;
```

```
}
```

Q.17 : `__hp_malloc()` commence par rechercher un bloc de mémoire libre. S'il y en a un, c'est celui-ci qui est renvoyé, sinon un nouveau bloc est créé et renvoyé (pourvu qu'il y ai suffisamment de place en mémoire pour un bloc de la taille demandée).

Q.18 : On implémente les lignes manquantes de la façon suivante :

```
static void *__hp_malloc(size_t taille)
{
    static size_t alloue = 0UL;
    void *ret;

    assert(taille > 0);

    taille = calcule_multiple_align(taille);

    ret = recherche_bloc_libre(taille); //recherche d'un bloc libre

    if (ret == NULL)
    {
        if (MEM_SIZE - alloue < TAILLE_EN_TETE + taille)
        {
            fprintf(stderr, "Il n'y a plus assez de mémoire disponible.\n");
            return NULL;
        }

        ret = &reserve.mem[alloue];
        alloue += taille + TAILLE_EN_TETE;
        bloc_init(ret, taille);
    }

    return ret;
}
```

Q.19 : À travers `__hp_free()`, le bloc argument devient le dernier bloc de la liste chaîné des blocs libres.

Q.20 : On implémente les lignes manquantes de la façon suivante :

```
static void __hp_free(void *ptr)
{
    struct bloc *bloc = libre;
    struct bloc *nouveau;

    if (ptr == NULL)
        return;

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

    while (bloc != NULL && bloc->suivant != NULL)
        bloc = bloc->suivant;

    if (bloc == NULL)
        libre = nouveau;
    else
        bloc->suivant = nouveau;
}
```

IV – Bibliothèque dynamique

Q.21 : On crée une bibliothèque dynamique avec les commandes suivantes :

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