CHAPITRE 1

Réalisation d'un allocateur mémoire

1.1 Objectif

Le but de cet exercice est de réaliser un allocateur dynamique de mémoire, c'est-à-dire un substitut aux fonctions `malloc()` et `free()` de C.

Les allocateurs de mémoire généraux sont parmi les programmes système les plus délicats à réaliser et à tester, mais aussi ceux qui peuvent avoir une influence considérable sur les performances en temps et en mémoire. Nous ne prétendons pas réaliser ici un allocateur très sophistiqué, seulement donner une idée des problèmes.

Cet exercice est aussi l'occasion de manipuler à un niveau fin les pointeurs de C, en mettant vraiment les mains dans le cambouis, comme on a souvent à le faire en programmation-système.

L'exercice n'est pas facile, même si le code en est court. Lisez bien les spécifications et les remarques qui suivent. Certains choix de conception ne deviendront clairs qu'après que vous ayez codé une solution. D'autres seront évidemment sujets à discussion (voir 1.4).

1.2 Présentation du problème

1.2.1 Les fonctions `malloc()` et `free()` d'ANSI C

En C, la fonction `malloc()` permet au programmeur d'allouer dynamiquement de la mémoire, et la fonction `free()` lui permet de rendre cette mémoire afin de la recycler pour l'utiliser dans un éventuel `malloc()` suivant. L'utilisation en est très simple (faites donc **man malloc** pour vous renseigner). Voici un exemple :

```
struct Data {  // une structure de données
    char nom[100];
    int age;
};
...
// On alloue dynamiquement un objet de ce type
// La fonction malloc() retourne un pointeur sur la zone allouée
struct Data *pdata = malloc(sizeof(struct Data));
```

```
// On peut maintenant utiliser librement cet objet
pdata->age = 12;
strcpy(pdata->nom, "Peter Pan");
...
// Et quand on n'en a plus besoin le libérer
free(pdata);
// Attention : ici le pointeur pdata n'est plus valide !
```

Cependant, la plupart des systèmes d'exploitation ne réalisent pas de manière primitive cette gestion du « recyclage ». Les fonctions `malloc()` et `free()` ne sont donc pas, en général, des appels-systèmes mais des fonctions de bibliothèque.

On peut d'ailleurs se demander pourquoi des fonctions aussi importantes ne sont pas directement réalisées par le système d'exploitation. La raison en est très simple. Il est très difficile d'écrire un allocateur général de mémoire dynamique, qui doit être à l'aise pour allouer un grand nombre de petits objets aussi bien qu'un grand nombre de très grands ou encore un mélange des deux. Des compromis de conception sont indispensables et les mauvais choix peuvent entraîner des pertes de performances parfois considérables. Donc il est préférable de ne pas figer les algorithmes de gestion mémoire dans le noyau. En les réalisant sous forme de fonctions de bibliothèque, on peut les changer et les remplacer facilement pour, par exemple, les adapter à un schéma d'utilisation mémoire particulier, pour lequel on peut imaginer des algorithmes plus efficaces que les compromis généraux.

1.2.2 La fonction UNIX sbrk()

Si le système d'exploitation ne réalise pas lui-même la gestion du « recyclage », il doit cependant collaborer un peu pour permettre la réalisation de la fonction `malloc()`. Le minimum qu'il ait à faire est de permettre d'augmenter l'espace mémoire d'un programme. Sous UNIX (et donc LINUX), ceci est réalisable par l'appel-système `sbrk()` [1] (**man sbrk**, donc...). Cette primitive s'utilise très simplement : il suffit de faire

```
void *pnew = sbrk(incr);
```

où `incr` est un entier non signé, pour que le segment de données du programme s'accroisse de (au moins) `incr` octets. La valeur de retour est un pointeur sur le début de la zone supplémentaire ainsi allouée. Noter bien que cette zone n'a absolument aucune structure ; c'est juste des octets à la suite les uns des autres, et c'est aux fonctions `malloc()` et `free()` qu'il appartiendra de la structurer.

Si le système ne peut plus allouer de mémoire supplémentaire, `sbrk()` retourne -1, ce qui n'est pas une très bonne idée car -1 n'est pas une valeur de pointeur (!) et cela rend le test un peu pénible :

```
if ((intptr_t)pnew == -1)
    fprintf(stderr, "Plus de mémoire\n");
```

Le type `intptr_t` est définit (en C99) dans le fichier `<stdint.h>` . Il s'agit du type entier dont la taille correspond à celle des pointeurs natifs de votre machine, e.g., 64 bits sur une machine 64 bits (**x86_64**).

1. Les fonctions `malloc()` et `free()` font partie de la norme ANSI C et donc de POSIX. Ce n'est pas le cas de `sbrk()` qui est spécifique à UNIX : d'autres systèmes d'exploitation peuvent proposer un mécanisme fondamentalement différent pour obtenir de la mémoire du système. On peut ajouter que `sbrk` est quelque peu obsolete, même sous UNIX, mais il suffira pour nos besoins.

1.3 UNE RÉALISATION D'UN ALLOCATEUR DYNAMIQUE SIMPLE

1.3.1 Spécification de l'interface

Bien que nous ayons annoncé que c'était très difficile, nous allons réaliser une version simple de `malloc()` et `free()`. Évidemment notre version ne sera pas aussi évoluée ni aussi efficace que celles que l'on trouve dans les systèmes modernes. Mais elle sera complète et permettra d'explorer les difficultés de la tâche.

Pour ne pas les confondre avec les versions standards, nous nommerons nos fonctions `mymalloc()` et `myfree()`. Leurs prototypes seront analogues à ceux du standard :

```
void *mymalloc(size_t size);
void myfree(void *p);
```

La fonction `mymalloc()` retourne un pointeur sur un bloc assez grand pour contenir un objet de taille `size` caractères (`size` est un entier non signé [2]). En cas d'échec, `mymalloc()` retourne le pointeur `NULL`. Quant à `myfree()`, elle libère la zone pointée par `p` afin qu'elle soit réutilisable par un futur `mymalloc()` dans le même programme ; après cet appel `p` est invalide (mais pas nul ! en fait sa valeur n'est pas modifiée). Bien entendu, pour pouvoir appeler `myfree()`, `p` doit avoir une valeur qui est le résultat d'un précédent `mymalloc()`.

1.3.2 Réalisation interne

Bloc utilisateur Afin de gérer le recyclage de la mémoire, le bloc retourné à l'utilisateur par `mymalloc()` doit contenir des informations de gestion qui n'on pas à être visibles ou accessibles de l'utilisateur. Nous supposerons que le bloc retourné a la structure représentée sur la figure 1.1.

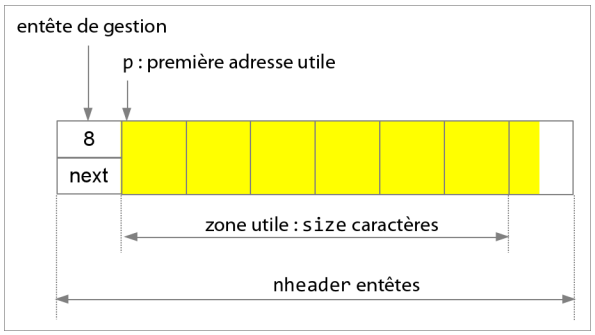

FIGURE 1.1 – Le bloc utilisateur.

Ce bloc utilisateur commence donc par un en-tête de gestion, qui est représenté par la structure C suivante :

```
struct Header {
    size_t nheaders;  // taille en nombre d'en-têtes
    struct Header *next;    // liste des blocs libres
};
```

Nous noterons `HEADER_SZ` la taille de cet en-tête. Donc

```
#define HEADER_SZ sizeof(struct Header)
```

Pour des raisons de commodité qui apparaîtront plus tard, `HEADER_SZ` sera en fait notre unité d'allocation mémoire. Nous allouerons toujours des blocs dont la taille est un multiple de `HEADER_SZ` et nous exprimerons leur taille en « nombre d'en-têtes ». Donc si l'utilisateur nous demande une taille de `size` caractères, nous lui fournirons un bloc comportant le nombre minimal d'en-têtes consécutifs permettant d'accommoder ces `size` caractères *plus*

2. Le type `size_t` est (pré)défini comme étant le type de retour de l'opérateur **`sizeof`**. Ce type est donc utilisé traditionnellement pour représenter des tailles mémoire ou des indices de tableaux. Il s'agit d'un entier non signé mais sa définition exacte dépend de la plate-forme matérielle utilisée (32 ou 64 bits en particulier).

un (pour l'en-tête de gestion). Le nombre total d'en-têtes nécessaires (y compris celui de gestion) sera rangé dans le champ `nheaders` de l'en-tête de gestion ; il est donné par la formule

```
nheaders = 2 + (size - 1) / HEADER_SZ
```

La première adresse utilisable (`p` sur la figure) est celle qui suit immédiatement l'en-tête de gestion ; c'est cette adresse que `mymalloc()` doit retourner à l'utilisateur, cachant ainsi l'en-tête de gestion.

Liste des blocs libres Notre bloc se trouve bien entendu dans une zone de mémoire que nous avons obtenue grâce à `sbrk()`. Nous avons intérêt à minimiser le nombre d'appels à `sbrk()` (ces appels-système sont chers). Donc nous ne demanderons jamais à `sbrk()` moins qu'une certaine quantité de mémoire, une constante, disons `MINSYSTBLOCK` (vous pouvez choisir une valeur quelconque, « ni trop grande, ni trop petite », mais il serait judicieux que cette constante soit un multiple de `HEADER_SZ`).

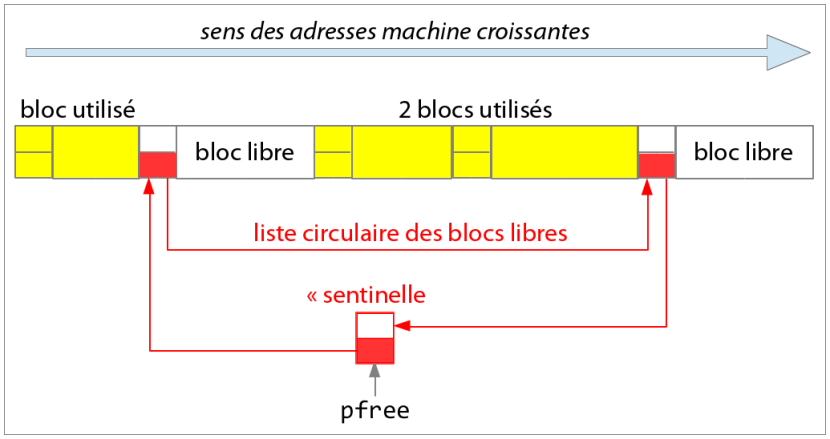

FIGURE 1.2 – La liste des blocs libres.

Pour cette raison et aussi par le jeu des recyclages, cette zone, que nous appellerons *bloc système*, peut en fait contenir plusieurs blocs utilisateur. Certains de ces blocs sont utilisés, d'autres ont pu être libérés. Grâce au pointeur `next` de l'en-tête, nous allons placer tous les blocs libres dans une liste simplement chaînée circulaire (Figure 1.2).

Le début (et la fin, puisqu'elle est circulaire !) de la liste sera marqué par un en-tête servant de sentinelle, pointé par le pointeur `pfree`. Cette sentinelle ne fait pas partie du bloc système (c'est une variable globale de `mymalloc()`/`myfree()`). Elle n'a aucune donnée associée (par exemple son champ `nheaders` est nul). Elle sert juste à accéder la liste qui est toujours parcourue de `pfree->next` (inclus) à `pfree` (exclus).

Notons que, par construction, si nous pouvons avoir deux blocs utilisés consécutifs, ce ne peut être le cas pour deux blocs libres (voir l'algorithme de `myfree()` plus loin). Par construction également, la liste des blocs libres sera toujours triée par adresses mémoires croissantes.

Algorithme de `mymalloc()` Compte tenu de la structure qui précède, l'algorithme pour trouver un bloc libre de taille `size` à retourner à l'utilisateur est le suivant :

1. Transformer `size` en le nombre d'en-têtes requis `nheaders`, grâce à la formule précédente.
2. Parcourir la liste libre (à partir de `pfree->next` inclus) jusqu'à trouver un bloc libre de taille suffisante.

3. Si on trouve un tel bloc et qu'il a la taille exacte nécessaire, le retirer simplement de la liste libre et retourner à l'utilisateur son adresse utile (le pointeur `p` de la figure 1.1). S'il est trop grand, le découper afin de pouvoir retourner un bloc de taille convenable à l'utilisateur et de retourner le tronçon restant dans la liste libre (voir plus bas une remarque sur la manière de découper).
4. Si on a fait un tour complet de la liste libre, c'est qu'un tel bloc n'existe pas. Alors appeler `sbrk()` pour demander de la mémoire au système (rappelez-vous que l'on ne demande jamais moins que `MINSYSTBLOCK` mais que l'on peut demander plus, c'est-à-dire un multiple de `MINSYSTBLOCK`, si nécessaire), structurer la mémoire correspondante en bloc avec en-tête, extraire la partie qui intéresse notre utilisateur et remettre ce qui reste dans la liste des blocs libres. Notez au passage que l'on a aucune garantie que les appels successifs à `sbrk()` rendent des zones de mémoire consécutives, ni même que ces zones sont dans l'ordre des adresses croissantes. **Suggestion :** quand vous aurez lu (et compris) l'algorithme proposé pour `myfree()` plus loin, revenez donc lire ce qui précède, cela pourrait vous donner des idées…

Quand on doit découper un bloc trop grand, il est commode de s'y prendre de la manière décrite sur la figure 1.3 : on retourne à l'utilisateur la fin du bloc libre, ceci minimise les mises à jour de pointeurs.

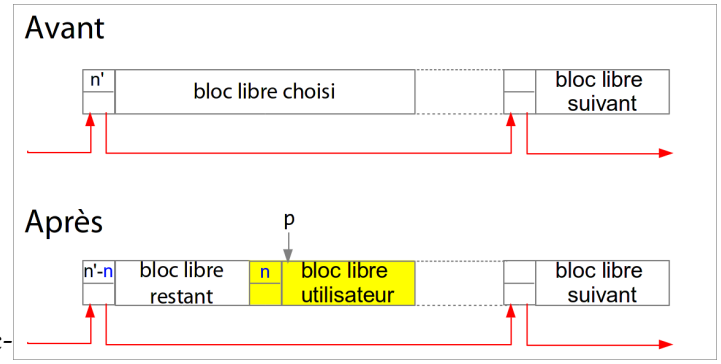

FIGURE 1.3 – Découpage d'un block libre trop grand : n est le nombre d'en-têtes requis par l'utilisateur.

Algorithme de `myfree()` La fonction `myfree()` reçoit comme unique paramètre le pointeur sur la partie utile du bloc (`p` sur la figure 1.1). Il suffit donc de parcourir la liste libre (rappelons qu'elle est triée par adresses croissantes) afin de trouver les deux blocs libres qui « encadrent » le bloc à libérer. D'où l'algorithme suivant.

Transformer le pointeur `p` fourni en un pointeur sur en-tête : `ptrh = (struct Header *)p - 1`. Puis parcourir la liste libre afin de trouver le premier bloc libre dont l'adresse est supérieure à `ptrh`. Lors de ce parcours, conserver non seulement un pointeur sur le bloc libre en cours d'examen (`ph`) mais aussi sur le bloc libre précédent (`phprev`).

Si l'on trouve un tel bloc, vous devez l'insérer dans la liste libre entre `phprev` et `ph`. Si on a fait un tour complet, c'est qu'il n'existe

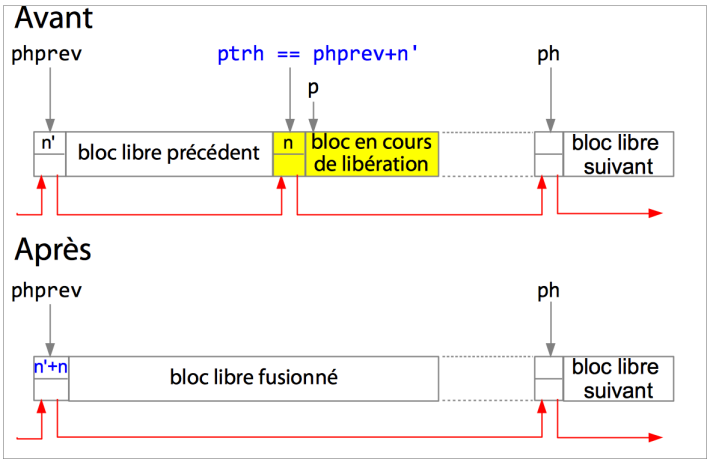

FIGURE 1.4 – Fusion des blocs libres adjacents dans `myfree()`.

aucun bloc libre dont l'adresse soit supérieure au bloc à libérer et celui-ci doit donc devenir le dernier de la liste libre.

Enfin vérifier si le bloc libre `phprev` précédent (`phprev`) et/ou suivant (`ph`) celui qu'on vient de libérer est/sont directement adjacent(s) à ce dernier. Si c'est le cas, fusionner les blocs libres pour former un seul bloc. La figure 1.4 illustre le cas où c'est le bloc précédent qui jouxte le bloc libéré, l'autre cas étant évidemment symétrique. Dans cette même figure 1.4, le premier schéma (marqué Avant) représente le bloc à libérer alors qu'il a déjà été insérer dans la liste libre.

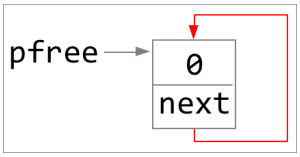

FIGURE 1.5 – Initialisation de la liste libre.

Initialisation La seule chose qui doive être initialisée correctement est la liste libre. C'est assez simple, elle est vide au début, ce qui veut dire que la sentinelle pointe sur elle-même (figure 1.5). Le premier appel à `myalloc()` ne trouvant aucun bloc `next` libre appelera `sbrk()` pour allouer le premier bloc système et l'ajouter à la liste des blocs libres. Ne pas oublier non plus d'initialiser la taille de la sentinelle (champ `nheaders`) à `0` ! (La sentinelle ne contient jamais de données utiles.)

Instrumentation et test Comme déjà mentionné, ce programme est délicat à écrire malgré sa faible taille [3] et il est encore plus délicat à tester. Cela donne évidemment l'occasion d'apprendre à utiliser un outil de mise au point comme **gdb** ou mieux **xxgdb** ou **ddd**.

Le test peut être bien entendu facilité en mettant quelques traces et aussi en instrumentant le code : par exemple on peut maintenir des informations sur les tailles totales allouées, le nombre de blocs couramment utilisés, le nombre de blocs système obtenus grâce à `sbrk()`, le nombre de blocs dans la liste libre, etc.

Pour faciliter le test, vous trouverez dans le répertoire `Malloc` deux programmes de test `main_mymalloc1.c` et `main_mymalloc2.c`. Ces programmes prennent en compte l'existence d'une fonction `mymalloc_instrum()` qui imprime les résultats de l'instrumentation. Évidemment, si vous ne disposez pas d'une telle fonction, vous pouvez toujours utiliser ces deux programmes, mais mettez les lignes correspondant aux appels de `mymalloc_instrum()` en commentaire.

> **Note**
>
> Dans le code fourni, vous avez aussi une version réduite du fichier `mymalloc.c` et de ses fichiers `.h`. Si vous exécutez **make** dans le répertoire fourni, vous obtenez des exécutables opérationnels, mais ceux-ci se contentent d'appeler les fonctions `malloc()` et `free` de la bibliothèque standard. Il faut bien entendu remplacer ces fichiers par votre propre code.

3. Pour donner un ordre de grandeur, dans ma propre solution, le code de `mymalloc()` et `myfree()` occupe un peu plus d'une centaine de lignes ; ceci inclut l'instrumentation, mais pas les commentaires (ma solution est vraiment très commentée) ni les programmes de test.

1.4 Analyse de l'allocateur ainsi réalisé

Un fois terminé, il n'est pas interdit de prendre un peu de recul par rapport à l'exercice. En particulier les questions suivantes méritent d'être posées :

- Le programme est-il robuste ? En particulier, supporte-t-il des valeurs incorrectes des paramètres ? Que serait-il possible d'améliorer ?
- Quel est le surcoût de l'allocation d'un bloc en terme de mémoire occupée ? Cela est-il acceptable ? pour les allocations des grandes tailles ? pour celles de petites tailles ?
- Quel est le coût en temps de l'allocation d'un bloc ? Est-ce améliorable ?