

Projet

Allocateurs mémoire

Philippe MARQUET

Mars 2017

Ce projet propose d'écrire plusieurs versions d'allocateurs mémoire.

Un allocateur mémoire est un mécanisme qui permet au sein d'un programme C d'obtenir dynamiquement de la mémoire. La bibliothèque C standard contient un allocateur mémoire utilisé via les appels aux fonctions `malloc()` et `free()`.

Nous allons débiter le projet par la mise en place d'un allocateur mémoire rudimentaire, pour en améliorer l'implémentation dans un second temps, pour l'enrichir ensuite, et enfin proposer une alternative complète à `malloc()`/`free()`.

Nous nous attacherons à tester et valider chacune de nos propositions.

1 Allocateur de blocs de taille fixe – atf

Notre premier allocateur va être capable d'allouer et de libérer des blocs mémoire d'une taille fixe donnée.

1.1 Interface de notre allocateur de taille fixe

L'interface de cet allocateur définit la constante

```
#define ATF_BLOCSIZE    ...
```

qui précise la taille des blocs.

Il est possible de demander l'allocation d'un bloc par un appel à

```
void * atf_newbloc();
```

qui retourne l'adresse d'un bloc mémoire de `ATF_BLOCSIZE` octets. Cette fonction `atf_newbloc()` retourne `NULL` en cas d'erreur, c'est-à-dire si il n'y a plus de mémoire disponible.

Un bloc précédemment alloué peut être libéré par un appel à

```
int atf_freebloc(void *bloc);
```

qui retourne 0 en seul cas de succès.

L'allocateur offre aussi une fonction `atf_init()` d'initialisation qui devra être appelée une et une unique fois avant l'utilisation des autres primitives.

1.2 Implémentation de notre allocateur de taille fixe

Une première implémentation consiste à utiliser un tableau alloué statiquement pour y garder les blocs.

On définit le nombre de blocs pouvant être alloués et déclare ce tableau :

```
#define NBLOCS    ...  
static char membloc[NBLOCS][ATF_BLOCSIZE];
```

Il s'agit ensuite de pouvoir identifier parmi les `NBLOCS` ceux qui sont libres.

On utilise un autre tableau

```
static char memstatus[NBLOCS];
```

qui pour chaque bloc indique s'il est libre (0) ou non (une valeur non nulle).

L'allocation d'un bloc consiste donc à rechercher une valeur nulle dans le tableau `memstatus`.

La libération d'un bloc consiste à retrouver le numéro du bloc à partir de l'adresse passée en paramètre et à marquer le bloc comme libre.

Exercice 1 (Allocateur de taille fixe — `atf`)

Proposez une implémentation d'un allocateur de taille fixe sous forme d'une bibliothèque `atf.h` et `atf.c` conforme à l'interface proposée. □

1.3 Validation à minima de notre allocateur de taille fixe

Une première validation à minima de notre bibliothèque peut consister en un programme qui

- alloue des blocs tant que possible via des appels à `atf_newbloc()` ;
- tout en mémorisant les adresses retournées certains de ces appels à `atf_newbloc()`, disons 42 adresses ;
- libère les blocs correspondant à ces 42 adresses ;
- alloue des blocs tant que possible ;
- vérifie que cela a permis d'allouer exactement 42 blocs ;
- vérifie que les adresses de ces 42 blocs correspondent exactement à celles des blocs libérés.

Exercice 2

Proposez un programme qui valide à minima notre bibliothèque `atf`. □

1.4 Test par le jeu de la lettre qui saute

Vous trouverez dans le dépôt

https://gitlab-etu.fil.univ-lille1.fr/ls4-pdc/validation_atf

- un exécutable `jeu` (ELF linux) ;
- un dictionnaire de mots de 4 lettres (`dico4lettres.h`) ;
- un `Makefile` ;
- les fichiers sources `jeu.c` (similaire à ce qui a été vu en cours) et `gestionpile.c` (similaire à ce qui va être vu en TD).

Le jeu consiste à se donner deux mots dans un dictionnaire (`dico4lettres.h`) et à voir s'il est possible de passer de l'un à l'autre en respectant la règle suivante : on passe d'un mot à un autre par une suite de mots ne différant que d'une lettre. Par exemple, on peut passer de `lion` à `peur` par la suite de mots :

`lion -> pion -> paon -> pain -> paix -> poix -> poux -> pour -> peur`

Pour tester le programme `jeu`, vous pouvez passer de `doux` à `dure`. Ceci fait, effacez l'exécutable `jeu`.

Exercice 3

Pour le reconstruire, vous devez ajouter à ces fichiers vos versions de `atf.h` et `atf.c`.

Un simple `make` devrait vous permettre de reconstruire l'exécutable et de rejouer.

On ne vous demande pas de coder le jeu ou la gestion de la pile mais de vérifier le bon fonctionnement de votre code d'allocation. □

1.5 Implémentation alternative de notre allocateur de taille fixe

En comparant la taille des tableaux `membloc` et `memstatus`, nous pouvons questionner le coût mémoire de cet allocateur et proposer une alternative.

Chaque valeur du tableau `memstatus` pourrait être mémorisée sur un bit, nous pourrions enregistrer 8 valeurs par octets, et ainsi déclarer (en supposant `NBLOCS` multiple de 8) :

```
static char memstatus_b[NBLOCS/8];
```

Nous pourrions aussi imaginer déclarer un tableau

```
enum memstatus_e {MEM_FREE=0, MEM_USED};  
static enum memstatus_e memstatus_s[NBLOCS];
```

Exercice 4

Question 4.1 Comparez les tailles respectives des tableaux `memstatus`, `memstatus_b`, `memstatus_e`. □

Question 4.2 Proposez une implémentation de notre bibliothèque `atf` plus efficace en terme de mémoire. □

2 Chaînage des blocs libres – `cbl`

Notre allocateur précédent doit parcourir successivement l'ensemble des éléments d'un tableau de grande taille pour trouver un bloc libre. Ce temps de parcours peut être pénalisant.

Nous allons proposer une implémentation plus rapide de notre allocateur mémoire qui permettra de trouver un bloc libre en un temps constant.

2.1 Ensemble des blocs libres

Notre allocateur va identifier des *séries de blocs libres contigus*. L'ensemble des séries de blocs libres contigus seront chaînées entre-elles.

- Nous allons utiliser le premier bloc de chaque série de blocs libres contigus pour mémoriser
- le nombre de blocs libres de la série;
 - le numéro du premier bloc de la prochaine série de blocs libres contigus (-1 si il n'y a pas de prochaine série).

L'allocateur mémorisera le numéro du premier bloc de la première série de blocs libres contigus (-1 si il n'y a plus de blocs libres) :

```
#define BLOC_END -1  
static int first_free_blocs;
```

La recherche d'un bloc libre est donc immédiate à partir de ce numéro `first_free_blocs`. On remarque aussi que le coût mémoire de nouvel allocateur est réduit.

2.2 Interface de l'allocateur `cbl`

L'allocateur `cbl` fournit l'interface suivante :

```
#define CBL_BLOCKSIZE ...  
  
void * cbl_newbloc();  
int cbl_freebloc(void *bloc);  
int cbl_init();
```

2.3 Implémentation de notre allocateur `cbl`

L'allocateur définit un type union pour mémoriser dans un bloc

- un tableau de `CBL_BLOCKSIZE` octets – pour les blocs alloués – ; ou
- une structure (nombre de blocs libres, prochaine série) – pour les premiers blocs libres des séries.

On conserve un tableau `membloc` alloué statiquement pour y garder les blocs.

Initialement, l'ensemble des blocs du tableau sont libres, formant une unique série de blocs libres contigus. La variable `first_free_blocs` sera donc initialisée à 0. Le bloc 0 sera initialisée avec une structure (`NBLOCS`, `CBL_END`).

Lors de la désallocation d'un bloc, on ne cherchera pas à le rattacher à une série de blocs libres contigus qui le précéderait ou le suivrait. On créera une nouvelle série de blocs libres contigus constituée d'un unique bloc.

Exercice 5 (Allocateur chaînage des blocs libres — `cbl`)

Proposez une implémentation d'un allocateur chaînant les blocs libres sous forme d'une bibliothèque `cbl.h` et `cbl.c`. □

Exercice 6 (Validation a minima de l'allocateur `cbl`)

Proposez un programme qui valide a minima notre bibliothèque `cbl` sur le même modèle que celui défini pour `atf`. □

3 Allocation de multiples blocs – `amb`

Nos allocateurs précédents proposent l'allocation d'un unique bloc d'une taille donnée.

Nous allons proposer une nouvelle interface permettant l'allocation d'un bloc mémoire d'une taille choisie, ou ce qui revient au même, de plusieurs blocs mémoire contigus.

3.1 Gestion des blocs mémoire

Supposant que l'on réponde à une demande d'allocation de plusieurs blocs mémoire, disons n blocs contigus. La primitive d'allocation retourne l'adresse de début de cette zone mémoire.

Lors de la désallocation de cette zone mémoire, la primitive de désallocation reçoit cette adresse. Il lui faut pouvoir retrouver combien de blocs mémoire avaient été alloués.

Nous allons donc, lors de l'allocation, mémoriser cette information, cette valeur n , dans le bloc précédant le premier bloc.

Ainsi une demande d'allocation de n blocs va rechercher $n + 1$ blocs libres contigus, utiliser le bloc 0 pour y mémoriser la taille n de la zone mémoire allouée, et retourner la zone mémoire constituée des blocs 1 à n .

Exercice 7 (Structure de donnée pour l'allocateur de taille variable)

Un bloc peut être

- un bloc de données, c'est-à-dire un bloc alloué ;
- le premier bloc d'une série de blocs libres ;
- le « bloc 0 » d'une série de blocs alloués.

Proposez la définition d'une structure de données pour représenter un bloc. □

3.2 Interface de l'allocateur `amb`

L'allocateur `amb` propose l'interface suivante :

```
#define AMB_BLOCKSIZE    ...

void * amb_newbloc(unsigned int nbloc);
int amb_freebloc(void *bloc);
int amb_init();
```

La fonction `amb_newbloc()` retourne l'adresse d'une zone mémoire de `nbloc` de `AMB_BLOCKSIZE` octets.

3.3 Allocation d'une zone mémoire

Il s'agit de trouver une série de blocs libres contigus d'une taille donnée. Comme pour l'allocateur précédent `cb1`, nos séries de blocs libres contigus sont gardées dans une liste chaînée.

Deux algorithmes sont possibles :

- choisir la première série de blocs libres contigus qui convient, qui est assez grande. On parle de *first-fit* ;
- choisir la liste de blocs libres de taille la plus proche de celle demandée. On parle de *best-fit*.

Une fois la série de blocs libres identifiée, il s'agit de la scinder en deux parties : la zone mémoire qui sera utilisée (y compris son bloc d'entête), et le reste des blocs de la série qui forme une nouvelle série de blocs libres contigus.

3.4 Désallocation d'une zone mémoire

Lors de la libération d'une zone mémoire, il est possible

- simplement de considérer cette zone mémoire, ex-série de blocs alloués, comme une nouvelle série indépendante de blocs libres, et de l'ajouter à la liste des séries de blocs libres ; ou
- de chercher parmi les séries de blocs libres si les blocs libres précédents ou suivant la nouvelle série ne seraient pas libres. Auquel cas, procéder à la fusion de ces séries pour former une unique série de blocs libres.

La première solution entraîne un émiettement de la mémoire, on parle aussi de fragmentation.

La seconde solution est coûteuse, un parcours de l'ensemble des séries de blocs libres étant nécessaire.

3.5 Allocateur `amb`

Exercice 8 (Allocateur de multiples blocs — `amb`)

Proposez une implémentation d'un allocateur de multiples blocs sous forme d'une bibliothèque `amb.h` et `amb.c`. □

Exercice 9 (Validation à minima de l'allocateur `amb`)

Proposez une évolution des validations à minima d'allocateurs précédents pour valider l'implantation la bibliothèque `amb`. □

3.6 Décompression d'un texte compressé par ajout de préfixe

On considère dans la suite des chaînes de caractères ASCII *décompressées* exclusivement composées de lettres (elles ne comportent aucun chiffre).

Une méthode primaire de compression consiste à repérer les motifs répétés dans la chaîne à compresser puis à construire une nouvelle chaîne constituée du nombre de répétitions suivi du motif. Par exemple, la chaîne :

AAAAAAAAAABABABCCD

est compressée en

10A2BA1B2C1D

Vous trouverez dans le dépôt

https://gitlab-etu.fil.univ-lille1.fr/ls4-pdc/validation_amb

du code implantant l'opération de décompression par le biais d'une fonction de prototype :

```
char * decompresse (char *) ;
```

qui prend en argument un pointeur sur une chaîne de caractères compressée et qui renvoie un pointeur sur une nouvelle chaîne de caractères non compressée.

Dans cette fonction, vous devez utiliser la fonction `amb_newbloc` de votre allocateur mémoire.

Nous allons maintenant utiliser cette fonction dans un filtre pour manipuler des fichiers. Pour ce faire, on considère le format de fichier d'entrée suivant :

- chaque ligne du fichier est composée par des chaînes compressées avec la méthode précédente ;
- la fin d'une chaîne compressées est définie par un caractère saut de ligne ;
- ces lignes ont strictement moins de 81 caractères ;
- le caractère EOF de fin de fichier se trouve au tout début de la dernière ligne du fichier.

Le programme `decompression.c` dont la compilation donne un filtre (`decompression`) prend sur l'entrée standard un fichier au format ci-dessus et retourne sur la sortie standard pour chaque ligne compressée du fichier d'entrée, la forme décompressée.

Vous prendrez soin à ce que les chaînes de caractères *décompressée* déjà retournée sur la sortie standard soit désallouée avec votre fonction `amb_freebloc`.

Pour tester votre code, vous pouvez utiliser le fichier d'entrée `decompression.input` et comparer votre sortie (en utilisant par exemple le filtre `diff`) avec le fichier de sortie `decompression.output`.

Exercice 10 (Validation de l'allocateur `amb`)

Vérifiez que votre mécanisme d'allocation permet au code de décompression de fonctionner. □

4 Bibliothèque d'allocation mémoire – `bam`

La phase ultime de ce projet est de développer une bibliothèque de remplacement des fonctions d'allocation dynamique fournies par la bibliothèque standard.

4.1 Bibliothèque et appels système

La bibliothèque Unix propose un système d'allocation dynamique de mémoire principalement utilisé au travers les fonctions `malloc()` et `free()`. Il existe aussi un appel système d'allocation de mémoire `sbrk()`¹.

Les fonctions de la bibliothèque sont implémentées au dessus des appels système. Les fonctions `malloc()` et `free()` sont donc implémentées à l'aide d'appels à `sbrk()`.

Détaillons ces trois fonctions.

`void *malloc (unsigned size);` La fonction `malloc()` de la bibliothèque retourne un pointeur sur un bloc d'au moins `size` octets, qui est correctement aligné selon la contrainte la plus forte de tous les types manipulables en C.

`void free (void *ptr);` La fonction `free()` de la bibliothèque libère un bloc précédemment alloué; `ptr` est un pointeur sur ce bloc.

`void *sbrk (int incr);` L'appel système `sbrk()` incrémente l'espace mémoire utilisateur de `incr` octets et retourne un pointeur sur le début de cette zone nouvellement allouée.

Le principe de fonctionnement est le suivant :

- l'utilisateur demande un bloc mémoire via un appel à `malloc()` ;

1. On se contentera de l'explication sommaire suivante : un appel système est un appel d'une fonction de bas niveau du système. La notion d'appel système sera détaillée dans le cours de « Programmation des systèmes » de 3e année de licence. Par ailleurs, l'emploi de `sbrk()` est maintenant obsolète; on s'en accommodera dans le cadre de ce projet pour ne pas compliquer les choses.

- `malloc()` demande lui même un bloc mémoire au système via `sbrk()`. Pour éviter le coût de cet appel système, `malloc()` demande un « gros » bloc mémoire à `sbrk()` et n'en utilise qu'une partie pour satisfaire la demande de l'utilisateur. Une structure de données interne à la bibliothèque d'allocation mémoire conserve la zone mémoire obtenue et non utilisée ;
- lors de prochaines demandes de blocs mémoire par l'utilisateur via des appels à `malloc()`, la mémoire excédentaire précédemment obtenue par `sbrk()` est utilisée ;
- les blocs mémoire libérés par `free()` sont gardés dans l'espace mémoire utilisateur et pourront aussi servir à répondre aux demandes suivantes. La mémoire n'est jamais rendue au système avant la fin de l'exécution du programme.

4.2 Possible implémentation de `malloc()`/`free()`

Comparé à nos allocateurs précédents, notre bibliothèque `bam` :

- n'utilise plus de tableau alloué statiquement (le tableau `membloc`), mais les résultats d'appels à `sbrk()` pour allouer les zones mémoire ;
- permet l'allocation de zones mémoire dont la taille n'est plus exprimée en nombre de blocs, mais en octets.

Le maintien de la liste des blocs mémoire libres se fera cependant selon une méthode proche de notre allocateur `amb`.

L'ensemble des blocs mémoire libres est par exemple gardé par la bibliothèque d'allocation mémoire sous la forme d'une liste chaînée :

- chaque bloc contient une taille et un pointeur sur le bloc suivant, ces deux informations formant en quelque sorte un entête. Cet entête est suivi de la zone mémoire disponible proprement dite ;
- quand une zone mémoire est retournée par `malloc()`, on conserve la taille de cette zone dans l'entête ;
- pour permettre un plus facile « recollement » de blocs, on peut par exemple conserver les blocs dans l'ordre croissant des adresses mémoire ;
- il peut être intéressant d'organiser la liste chaînée de manière circulaire : le dernier bloc pointe sur le premier.

Exercice 11 (Structures de données pour `bam`)

Définissez les structures de données nécessaires pour la gestion de cette liste de blocs. □

L'implémentation de `malloc()` peut parcourir la liste des blocs libres jusqu'à rencontrer un bloc suffisamment grand (« first-fit »). Si le bloc a exactement la taille demandée, on l'enlève de la liste et on le retourne à l'utilisateur. Si le bloc est trop grand, on le divise et on retourne à l'utilisateur un bloc de la taille demandée ; le reste du bloc est gardé dans la liste des blocs libres. Si la recherche d'un bloc de taille suffisante a échoué, on demande une augmentation de la mémoire utilisateur au système.

La libération d'une zone mémoire par `free()` recherche l'emplacement auquel insérer ce bloc dans la liste des blocs libres. Si le bloc libéré est adjacent à un bloc libre, on les fusionne pour former un bloc de plus grande taille. Cela évite une fragmentation de la mémoire et autorise ensuite de retourner des blocs de grande taille sans faire des appels au système.

Exercice 12 (Allocation et désallocation)

Proposez une implémentation d'un allocateur fonctionnant à l'image de `malloc()` et `free()` sous forme d'une bibliothèque `bam.h` et `bam.c`. □

Exercice 13 (Validation de la bibliothèque `bam`)

Proposez une validation de cette bibliothèque `bam`. Inspirez-vous des validations des allocateurs développés jusqu'ici. □

À suivre...