TP Allocation dynamique de la mémoire sur le tas

2021/2022

Implémentation de `myalloc/myfree`

Vous allez implémenter vos propres fonctions d'allocation dynamique de la mémoire `myalloc` et `myfree`.

Ces fonctions seront des versions simplifiées, mais néanmoins fonctionnelles, de `malloc` et `free`.

Pour rappel, les fonctions `malloc` et `free` vont modifier le registre `brk` en utilisant des procédures de plus bas niveau: `brk()` et `sbrk()`.

Comme présenté en cours, c'est la modification de la valeur du registre `brk` qui va permettre de changer la taille du *tas* qui est le segment où résident toutes les variables allouées dynamiquement.

Notre implémentation de `myalloc` et `myfree` devra utiliser les fonctions `brk()/sbrk()` pour allouer et libérer les blocs mémoires dans le tas.

Nous vous proposons dans un premier temps de consulter la documentation des deux fonctions `brk()` et `sbrk()`.

Étape 0: Préparation de l'environnement de travail

Cette première étape va organiser notre environnement de travail et introduire deux outils:

- le programme `make` qui va gérer la compilation de notre projet automatiquement en consultant le fichier de dépendances `Makefile`
- la macro `#include` qui va recopier le contenu d'un fichier externe pour l'inclure comme contenu dans le fichier courant

Mais avant de présenter ces outils, nous allons préparer et configurer notre environnement de travail.

Organisation du code source

A faire:

- Télécharger le fichier source avec l'extension `.zip` associé à ce TP

Après avoir décompressé ce fichier, le code source de départ du TP est organisé comme suit:

```
src
  etape0
    Makefile
    blocinfo-0.inc.c
    malloctest.c
    myalloc-0.c
    myalloc-0.inc.c
    myalloc-bloc-entete-0.h
    myfree-0.inc.c
  etape1
    Makefile
    blocinfo-1.c.inc
    malloctest.c
    myalloc-1.c
    myalloc-1.c.inc
    myalloc-bloc-entete-1.h
    myfree-1.c.inc
  etape2
    Makefile
    blocinfo.inc.c
    malloctest.c
    myalloc-bloc-entete.h
    myalloc.c
    myalloc.inc.c
    myfree.inc.c
  generic
    align.h
    entete_size.h
    myalloc.h
```

Nous avons 4 répertoires: trois répertoires qui représentent les trois exercices du TP (etape0, etape1 et etape2); et un répertoire (generic) qui contient les fichiers entêtes communs.

Chaque répertoire d'étape contient un programme de test `malloctest.c`. Une

fois votre proposition implémentée, vous devez la tester avec ce programme.

Cette organisation n'est qu'une une proposition; vous pouvez adopter votre propre organisation pour réaliser le TP.

`Makefile` et `make`

Vous avez sûrement observé que pour chaque étape, nous avons un fichier nommé `Makefile`. Ce fichier est utile pour la commande `make` afin d'automatiser la compilation.

L'outil `make` permet d'exécuter une commande sous certaines conditions. Dans un projet C, nous allons utiliser cet outil pour recompiler les modules ainsi que le programme principal, si les fichiers sources ainsi que les modules intermédiaires sont modifiés.

Pour cela, `make` a besoin d'avoir un ensemble de *règles*. Ces règles sont écrites dans un fichier nommé `Makefile` qui devra se trouver dans le répertoire d'exécution du `make`.

Chaque règle va avoir un nom (a.k.a. cible), des dépendances et une commande associée. Si une des dépendances est modifiée alors la commande associée à la cible est lancée.

Voici un exemple d'une règle simple:

```
programme.o: programme.c programme.h
    gcc -o programme.c
```

Cette règle possède une cible nommée `programme.o`. La cible dépend de deux fichiers: `programme.c` et `programme.h`.

Si une de ces deux dépendances est modifiée alors la commande `gcc -c programme.c` sera exécutée.

Attention: Veuillez bien respecter la tabulation qui précède la commande à exécuter.

Si nous souhaitons compléter cet exemple pour générer un exécutable, alors nous devrions introduire une nouvelle règle:

```
programme.o: programme.c programme.h
    gcc -o programme.c

executable: programme.o
    gcc -o monprogramme programme.o
```

La nouvelle cible `executable` va permettre de générer le fichier exécutable en exécutant la commande associée.

Pour lancer une cible en particulier, il suffit de préciser à `make` le nom de la cible. Par exemple pour lancer la cible `executable`, il suffit de lancer:

```
make executable
```

La macro `#include`

`#include` n'est pas une instruction du langage C. C'est une macro du préprocesseur qui va copier le contenu du fichier spécifié dans le fichier courant.

Pour illustrer l'utilisation de `#include` nous proposons d'inclure les codes de vos fonctions, se trouvant dans des fichiers sources différents, dans un seul fichier source.

Par exemple, pour l'étape 0, les codes des fonctions `myalloc`, `myfree` et `blocinfo` seront contenus dans les fichiers `myalloc-0.inc.c`, `myfree-0.inc.c` et `blocinfo-0.inc.c`.

Le fichier `myalloc-0.c` va ensuite inclure ces codes en utilisant la macro `#include`.

Il est important de noter que seul le fichier `myalloc-0.c` sera compilé car il va inclure le contenu des autres fichiers.

Nous pouvons vérifier cela dans le `Makefile`. La cible `myalloc-0.o` précise bien qu'elle dépend des fichiers sources `myalloc-0.c`, `myfree-0.inc.c`, `myalloc-0.inc.c` et `blocinfo-0.inc.c`. Mais la commande de compilation `gcc -g -c myalloc-0.c` n'inclut que fichier `myalloc-0.c`.

Étape 1: Une première implémentation naïve

Notre première implémentation sera naïve: nous allons allouer simplement un bloc mémoire en augmentant la taille du tas.

Alignement de la mémoire

Pour optimiser les opérations d'accès à la mémoire, il est intéressant d'avoir des adresses qui soient des multiples d'une puissance de 2.

Par exemple, les adresses `0x00000008` et `0x00000010` sont toutes les deux des multiples de $8{=}2^3$. Ces adresses sont *alignées* sur 8.

Essayons de comprendre pourquoi l'alignement de la mémoire permet d'optimiser les accès aux variables.

Prenons un exemple où un utilisateur demande la création des variables suivantes:

- 1 variable de type int, que nous appellerons `x`
- 1 variable de type char, que nous appellerons `c`
- 1 variable de type int, que nous appellerons `y`

Comparons maintenant deux stratégies d'allocation de la mémoire.

La première stratégie va allouer des adresses mémoires non alignées. Le schéma de la mémoire sera le suivant après la création des trois variables:

0	1	2	3	4	5	6	7	8
&x				&c	&y			

Il faut savoir que nous pouvons lire de la mémoire des *mots* avec une taille qui correspond à la taille du *bus mémoire*.

Si dans notre exemple, la taille de notre bus est de 4 octets (un bus 32 bits), alors nous pouvons lire la variable `x` avec une seule opération de lecture. Il suffit de demander l'adresse `0x00000000` et le mot retiré de la mémoire va correspondre à `x`.

Si nous souhaitons maintenant lire la variable `c` alors nous allons demander la lecture de l'adresse `0x00000004`, nous allons retirer un mot de 4 octets mais nous nous intéressons simplement au premier octet qui correspond à la variable `c`.

Maintenant si nous souhaitons retirer la variable `y` nous allons devoir demander 2 opérations de lectures:

1. une première opération de lecture va lire un mot à l'adresse `0x00000004` pour avoir les 3 premiers octets de `y`
2. une deuxième opération de lecture va lire un mot à l'adresse `0x00000008` pour avoir le 4ème octet de `y`

En effet, la lecture avec le bus mémoire ne peut se faire que sur des adresses qui sont multiples de sa taille.

Nous voyons qu'avec ce schéma et avec la contrainte de lecture, nous allons réaliser deux opérations de lecture pour avoir le contenu de la variable `y`

Regardons maintenant la deuxième stratégie qui va réaliser un alignement des adresses mémoires sur 4 dans cet exemple. Cela veut dire que nous allons toujours allouer un bloc mémoire de taille plus grande ou égale à ce que nous avions demandé et que cette taille est un multiple de notre constante d'alignement (ici 4)

Si nous adoptons pour cet exemple une constante d'alignement égale à 4, alors le schéma de la mémoire pour avoir les trois variables `x`, `c` et `y` sera le suivant:

0	1	2	3	4	5	6	7	8	9	10	11
&x				&c				&y			

La variable `y` est alignée en mémoire sur un multiple de 4. Et si nous demandons à lire cette variable, alors *une seule* lecture du mot à l'adresse `0x00000008` suffit ! Mais cette amélioration de la performance, vient avec un prix: nous avons laisser un espace inutilisable entre `&c` et `&y`.

Dans la suite, nous allons utiliser cette astuce pour toujours allouer des blocs mémoires avec une taille multiple d'une constante `ALIGNMENT` qui sera spécifiée dans le fichier `generic/align.h`

Pour vous aider à calculer une taille de bloc avec alignement vous pouvez utiliser la macro `ALIGN` qui va vous donner le multiple de `ALIGNMENT`à utiliser. Cette macro se trouve dans le fichier `generic/align.h`

Par exemple, si la constante `ALIGNMENT` vaut 8 alors `ALIGN(5)` sera égale à 8 et `ALIGN(9)` sera égale a 16.

Méta-données du bloc mémoire

Pour gérer les blocs mémoires de l'utilisateur, il nous faut sauvegarder certaines informations. Ces données sur les données sont appelées des *méta-données*.

Par exemple, la taille du bloc mémoire de l'utilisateur est une information importante à sauvegarder.

Un bloc mémoire au total va toujours être composé d'une partie de méta-données rangées dans un *header* et d'une partie qui contient les données de l'utilisateur: le *payload*

Nous proposons la structure C suivante pour contenir les méta-données:

```
typedef struct bloc_entete
{
 size_t taille; //taille total du payload
 unsigned short libre ; //drapeau qui indique si le bloc est libre ou utilise: 1 libre, 0 u
} bloc_entete ;
```

Une macro utilitaire, `ENTETE_SIZE`, permet de connaître la taille de la partie des méta-données:

Implémentation de `myalloc` et `myfree`

```
void* myalloc(size_t t);
```

Implémentez la fonction `myalloc` qui retourne l'adresse du bloc mémoire alloué de taille t (au moins) avec les spécifications suivantes:

- Cette fonction doit utiliser `sbrk()` pour augmenter la taille du tas et gérer correctement l'entête du bloc.
- Attention, vous devez renvoyer l'adresse du bloc utilisateur sans le bloc d'en-tête (prévoir un décalage)

Vous pouvez utiliser le pseudo code suivant comme aide:

```
void* myalloc(size_t t){

    // 1. calculer la taille totale du bloc a allouer

    // 2. ne pas oublier d aligner la taille totale

    // 3. realiser un decalage de brk egale a la taille totale avec sbrk

    // 4. recuperer le pointeur donne par sbrk c est le bloc nouvellement alloue

    // 5. faire un cast pour considerer le pointeur comme un pointeur sur bloc_entete

    // 6. Mettre a jour les champs de bloc_entete

    // 7. Retourner l adresse de la partie utilisateur: decalage de la taille ENTETE_SIZE

}
```

Implémentation de `myfree`

```
void myfree(void* ptr);
```

Implémentez la fonction `myfree` avec les spécifications suivantes:

- Libère le bloc mémoire en mettant le flag de l'entête du bloc à la bonne valeur.
- Attention, ptr doit pointer sur le bloc utilisateur, il faut donc réaliser un décalage pour retrouver l'emplacement de l'entête.

Vous pouvez utiliser le pseudo code suivant comme aide:

```
void myfree(void* ptr){

    //1. realiser un decalage de ENTETE_SIZE a partir de ptr

    //2. faire un cast pour considerer la nouvelle adresse comme un pointeur sur bloc_entete

    //3. Mettre a jour les champs de bloc_entete
```

```
}
```

Etape 2: Recyclage des blocs

Notre solution alloue bien de la mémoire dynamiquement. Cependant, pour libérer de la mémoire nous avons simplement mis le flag à 1 et les blocs libérés ne sont jamais recyclés pour les prochaines demandes d'allocation.

C'est l'objectif de cette étape: avant d'augmenter le programme break nous allons vérifier si un bloc libre peut être réutilisé.

Nous allons adopter une stratégie de recyclage simple:

- Nous allons allouer le premier bloc qui convient (first fit)
- Si un bloc convient il est entièrement réutilisé (no split)

Afin de parcourir tous les blocs alloués, il nous faut sauvegarder l'adresse du premier bloc.

Nous pouvons ensuite parcourir l'ensemble des blocs connaissant la taille de chaque bloc.

Évidemment, nous ne devons pas dépasser le programme break courant (obtenu par l'astuce: sbrk(0))

Modifiez la fonction `myalloc` afin de recycler des blocs quand cela est possible.

Analyse de la solution

Nous avons amélioré notre première proposition par le recyclage de certains blocs.

Il serait intéressant d'analyser les performances de cette proposition.

Imaginons que notre processus utilise actuellement u blocs et qu'il a libéré l blocs. Le nombre total des blocs est $n = u + l$.

Pour notre recherche de blocs à recycler, le pire des cas serait que le bloc intéressant se trouve à la dernière position du tableau des blocs.

Dans ce cas, nous devons parcourir les n blocs. Si le traitement d'un bloc dure t unité de temps, alors le temps de réponse de la fonction `myalloc`, dans le pire des cas, sera de $t * n$.

Nous pouvons améliorer ce temps de réponse en traitant uniquement les blocs libres et comme $l \leq n$ alors nous pouvons espérer un temps de réponse plus rapide.

Étape 2: Recyclage des blocs avec une liste

L'objectif de cette étape est d'améliorer les performances de `myalloc`/`myfree` en gérant uniquement les blocs libres dans une liste. Cette liste sera parcourue pour trouver un bloc à recycler.

La première modification est au niveau de l'entête des blocs:

- Nous n'avons plus besoin du flag qui indique si le bloc est libre
- Nous ajoutons deux pointeurs pour gérer une liste doublement chaînée.

Voici donc la nouvelle structure pour le bloc entête:

```
typedef struct bloc_entete
{
  size_t taille; //taille du bloc utilisateur
  struct bloc_entete* suivant_ptr ;    // pointeur sur le bloc suivant dans la liste
  struct bloc_entete* precedent_ptr ;    // pointeur sur le bloc precedent dans la liste
} bloc_entete ;
```

Nous allons conserver la même stratégie de recyclage i.e:

- allocation du premier bloc qui convient (first fit)
- si un bloc convient, alors il est entièrement réutilisé (pas de split)

Modifiez la fonction `myalloc`:

- pour parcourir la liste des blocs libres afin de rechercher un bloc à recycler
- si un bloc à recycler est trouvé, alors il faut le retirer de la liste des blocs libres
- si aucun bloc de la liste ne convient, alors proposer un nouveau bloc avec l'extension du `brk`.

Modifier la fonction `myfree` pour:

- ajouter le bloc libéré dans la liste des blocs libres.