Travaux Pratiques Systèmes d'Exploitation
allocation dynamique de la mémoire

2014/2015

1 Implémentation de myalloc/myfree

Vous allez implémenter vos propres fonctions d'allocation dynamique de la mémoire myalloc/myfree. Ces fonctions seront des versions simplifiées mais fonctionnelles de malloc/free.

Comme malloc/free nous allons nous baser sur les fonctions brk/sbrk pour gérer le pointeur program break et ainsi modifier la taille du tas (heap)

Question 1: (Re)lisez attentivement la documentation des fonctions brk/sbrk.

2 Étape 1: Une première implémentation naïve

Pour notre première implémentation nous proposons de réaliser une implémentation naïve: nous allons allouer simplement un bloc mémoire en augmentant la taille du tas.

Attention la taille réelle du bloc alloué doit être alignée c'est à dire elle doit commencer à une adresse multiple d'une constante donnée (puissance de 2).

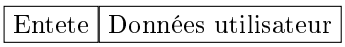
Pour vous aider à calculer une adresse alignée vous pouvez utiliser la macro ALIGN:

```
#define ALIGNMENT 8
#define ALIGN(size) (((size) + (ALIGNMENT-1)) & ~(ALIGNMENT-1))
```

2.1 Méta-données du bloc mémoire

Pour gérer les blocs mémoires de l'utilisateur, il nous faut sauvegarder certaines données. Par exemple, la taille du bloc mémoire de l'utilisateur est une information importante à sauvegarder. Ces données sur les données sont appelées des *méta-données*.

Un bloc mémoire réel va toujours être composé d'une partie de méta-données rangées dans un header et d'une partie pour ranger les données utiles (payload)



Voici une première structure C qui permet de réaliser ce schéma:

```
#define ENTETE_SIZE (ALIGN(sizeof(bloc_entete)))
```

Pour avoir la taille que prend notre entête, nous allons définir la macro ENTETE_SIZE:

```
#include <stdlib.h>
typedef struct bloc_entete
{
 size_t taille; //taille du bloc utilisateur
 unsigned short libre : 1 ; //drapeau de 1 bit qui indique si le
    bloc est libre ou utilise: 1 libre, 0 utilise.
} bloc_entete ;
```

2.2 Implémentation de myalloc et myfree

Question 2: Implémentez la fonction myalloc avec les spécifications suivantes:

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

Retourne l'adresse du bloc mémoire alloué de taille t (au moins)
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'entête (prévoir un décalage)

Question 3: Implémentez la fonction myfree avec les spécifications suivantes:

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

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

2.3 Etape 2: Recyclage des blocs

Notre solution alloue bien de la mémoire dynamiquement. Cependant, pour l'instant pour libérer de la mémoire nous avons simplement mis le flag à 1. 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 ne peut pas être réutilisé.

Pour l'instant nous allons adopter une stratégie de recyclage simple:

- Nous allons allouer le premier bloc qui convient (firt fit)
- Si un bloc convient il est entièrement réutilisé (sans 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 program break courant (obtenu par l'astuce: sbrk(0))

Question 4: Modifiez la fonction myalloc afin de recycler des blocs quand cela est possible.

2.3.1 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 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.

2.4 Étape 2: Recyclage des blocs avec une liste

L'objectif de cette étape est d'améliorer les performances de myalloc/myfree. Pour cela, nous proposons de gérer uniquement les blocs libres avec une liste. Cette liste sera parcourue pour trouver un bloc à recycler.

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

```
#include <stdlib.h>
/*
 * Gestion des blocs avec une liste chainee
 */
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 n'avons plus besoin du flag qui indique si le bloc est libre et nous rajoutons deux pointeurs pour gérer une liste doublement chainée.

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)

Question 5: Modifiez la fonction myalloc pour:

- parcourir la liste des blocs libres pour rechercher un bloc à recycler
- si un bloc à recycler est trouvé, alors le retirer de la liste des blocs libres
- si aucun bloc à recycler n'est disponible, alors proposer un nouveau bloc.

Modifier la fonction myfree pour:

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

2.5 Recyclage amélioré des blocs avec une liste (optionnel)

Pour cette étape, améliorez la solution précédente en implémentant la stratégie suivante:

- allocation du premier bloc qui convient le mieux (best fit)
- si un bloc est plus grand que la taille demandée alors ne prendre que la taille requise et considérer le bloc restant comme toujours libre (avec split)