Programmation – TD9 : Pointeurs en folie

{ Jeremie.Detrey, Emmanuel.Jeandel } @ens-lyon.fr

8/9 décembre 2004

⚠ *Attention ! Aujourd'hui on fait du C.*

1 Mise en jambes : balade dans les arbres

1.1 Introduction

Nous allons travailler pour cet exercice avec des arbres binaires classiques, représentés par la structure de données suivantes :

```
struct node {
  int value;
  struct node *left, *right;
};
```

Ainsi, chaque nœud contient une valeur (ici un entier), ainsi que des pointeurs vers ses fils gauche et droit. Une feuille est représentée par un pointeur `NULL`.

1.2 Parcours classiques

1.a. Écrivez une fonction `depth_first` qui effectue un parcours en profondeur d'un arbre. Affichez par exemple la valeur des nœuds rencontrés pour vérifier que votre fonction marche bien.

1.b. Si vous avez écrit une fonction récursive pour la question précédente, reprenez-la pour qu'elle ne le soit plus.

2. Écrivez la fonction `breadth_first` qui effectue un parcours en largeur d'un arbre.

1.3 Parcours en profondeur en place[1]

Attention : ceci est du bonus. Amusez-vous à chercher si vous êtes en avance. Sinon passez à la suite.

Pour répondre à la question **1.** vous avez dû avoir recours à une pile (que ce soit la pile de récursion, ou une pile de votre cru pour mémoriser les nœuds qu'il vous restait à parcourir). En fait, grâce à l'incroyable toute puissance du C[2], il est possible (moyennant creusage de neurones conséquent) de se passer de cette structure auxiliaire, et donc d'effectuer le parcours en espace constant (on dit *en place* pour faire savant).

[1] Aucun pointeur n'a été blessé durant la réalisation de cet exercice.

[2] Bon, OK, on en fait peut-être un peu trop, là.

L'idée ici est de n'utiliser que deux pointeurs supplémentaires : un pour désigner le nœud en cours (`cur`), et un autre pour désigner celui d'où l'on vient (`prev`). Il faut juste trouver un moyen de sauvegarder `prev` lorsque l'on descend dans l'arbre pour pouvoir le récupérer à la remontée.

3. Allez-y ! N'hésitez pas à appeler votre TDman à la rescousse si vous bloquez.

2 Spaghetti à la sauce `malloc`

On reprend dans cet exercice une version adaptée à notre sauce du `malloc` Kernighan & Ritchie, avec des idées de la version de Doug Lea.

2.1 Préambule

Nous allons vous faire écrire quelques versions de `malloc` (et de son comparse `free`), que vous allez tester sur des exemples simples. Pour cela, vous appellerez vos fonctions `mymalloc` et `myfree`. Vous aurez aussi besoin d'une fonction `myinit` qui allouera un bloc de mémoire de 1Mo qui représentera le segment mémoire sur lequel travailleront vos fonctions. `myinit` se chargera aussi d'initialiser les structures de données nécéssaires. Pour information, voici le prototype de ces trois fonctions :

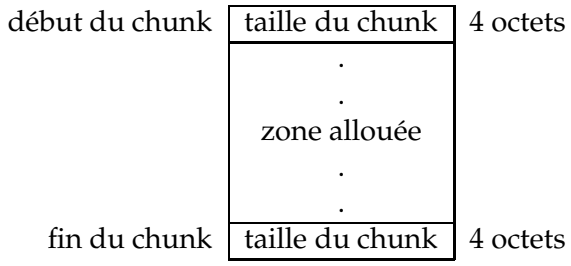
```
void myinit();
void *mymalloc(size_t size);
void myfree(void *ptr);
```

Pour tester votre code, vos TDmen adorés ont écrit une version du programme `sort` qui trie par ordre alphabétique une liste de mots passés en entrée standard (le tout avec plein d'allocations inutiles exprès). Vous pouvez le récupérer sur la page habituelle, ainsi qu'un fichier d'exemple.

2.2 `malloc`, comment ça marche ?

La mémoire est représentée comme un ensemble de blocs (ou *chunks*) pouvant chacun être soit alloué soit libre. Au début de l'exécution du programme, le segment de mémoire n'est qu'un énorme chunk libre, qui se fractionne au fur et à mesure que les appels à `malloc` le grignottent.

Un chunk alloué sera représenté de la manière suivante :



Et pour un chunk libre, on aura :

début du chunk	taille du chunk	4 octets
	pointeur vers le chunk libre précédent	4 octets
	pointeur vers le chunk libre suivant	4 octets
	. . zone libre . .	
fin du chunk	taille du chunk	4 octets

L'information sur la taille des chunks est redondante, car la taille d'un chunk est indiquée au début et à la fin du chunk. Cela permet en fait de traverser la mémoire de chunk en chunk dans les deux sens : ça nous servira par la suite pour fusionner des chunks libres contigus.

Les deux pointeurs contenus dans les chunks libres nous permettent quant à eux de maintenir une liste doublement chaînée de chunks libres, que l'on va parcourir à chaque allocation pour trouver un chunk de la bonne taille.

La taille minimale d'un chunk est donc de 16 octets. De plus, pour des raisons d'alignement, les tailles des chunks doivent être des multiples de 4 octets. On va donc se servir de cette propriété pour coder de l'information supplémentaire : pour pouvoir différencier les chunks alloués des chunks libres, on ajoutera 1 à la valeur de la taille des chunks alloués. Ainsi, lorsque l'on rencontrera un chunk de taille n (telle qu'indiquée dans la structure) il suffira de faire un test de parité sur n : si n est pair, il s'agit d'un chunk libre de taille n, alors que si n est impair, il s'agit d'un chunk alloué de taille $n - 1$.

2.3 À vous maintenant !

1. Écrivez une première version de `mymalloc` et `myfree` : `mymalloc` recherche un chunk libre de taille suffisante, le découpe s'il est trop gros, et `myfree` se contente de libérer le chunk et de le placer dans la liste.
2. Améliorez-la un peu en fusionnant des chunks libres adjacents.
3. Faites en sorte de maintenir les chunks libres triés par taille croissante : cela vous permettra de faire du *best-fit* pour l'allocation et donc éviter de fragmenter trop la mémoire.
4. Réfléchissez à comment améliorer encore la chose. Vous pouvez aller jeter un œil à l'article de Doug Lea pour vous aider : `http://gee.cs.oswego.edu/dl/html/malloc.html`.