SINF1252 - Implémentation de malloc et free

2016

1 Énoncé

1.1 Description du projet

La fonction `void *malloc(size_t size);` fournie par la librairie standard permet d'allouer de la mémoire de manière dynamique sur le heap. Concrètement, `malloc` connait la taille actuelle du heap grâce à l'appel système `sbrk` et peut augmenter cette taille grâce à ce même appel. Lorsqu'on demande à `malloc` d'allouer une zone mémoire de taille n, la fonction se débrouille pour trouver dans le heap un bloc suffisamment grand que pour contenir n bytes consécutifs. S'il n'existe pas de tel bloc, alors `malloc` demande au kernel d'étendre le heap du programme via `sbrk` et retente l'opération. Si le kernel refuse d'augmenter le heap (par exemple, lorsqu'il n'y a plus assez de mémoire disponible), alors `malloc` échoue et retourne NULL.

Il existe également la fonction `void free(void *ptr);` qui permet de libérer la mémoire associée à un pointeur précédemment retourné par `malloc`.

Lors de ce projet, vous allez devoir implémenter, par **binôme**, votre propre version, simplifiée, de `malloc` et `free`, que nous appellerons `mymalloc` et `myfree`. Votre implémentation devra satisfaire les contraintes suivantes:

1. **Séquences d'appels arbitraires**. Un programme qui utilise votre implémentation peut effectuer des appels à malloc/free n'importe quand, et dans n'importe quel ordre.

2. **Réponse immédiate**. Votre implémentation doit répondre immédiatement aux requêtes qui lui sont effectuées. Par exemple, il est interdit de réordonner ou placer les requêtes dans un buffer.

3. **Seul le heap doit être utilisé**. Les structures contenant les informations sur les blocs de mémoire alloués doivent également se trouver sur le heap.

4. **Alignement**. Les blocs de mémoire que votre implémentation alloue doivent être alignés sur 32 bits (cela signifie notamment que le pointeur que `mymalloc` retourne doit être un multiple de 4).

5. **Ne pas modifier les blocs alloués**. Il est interdit de modifier ou de déplacer des blocs déjà alloués. Seuls les blocs libres peuvent être manipulés.

On distingue souvent deux objectifs orthogonaux pour les *memory allocators*: maximiser la vitesse d'allocation, et maximiser l'utilisation de la mémoire. Le premier objectif est très facile à atteindre: il suffit d'allouer le premier bloc suffisamment grand

que l'on trouve. Cependant, l'utilisation de la mémoire d'une telle solution est inacceptable. Pour ce projet, on vous demande de vous focaliser sur le deuxième objectif, c'est à dire de maximiser l'utilisation de la mémoire.

1.2 Fragmentation

Le principal problème que vous allez rencontrer dans ce projet est la fragmentation, qui caractérise une situation où il existe suffisamment de mémoire libre correpondant à une requête, mais où l'agencement des blocs libres ne permet pas de satisfaire cette requête. Il existe deux types de fragmentation: la fragmentation interne et la fragmentation externe.

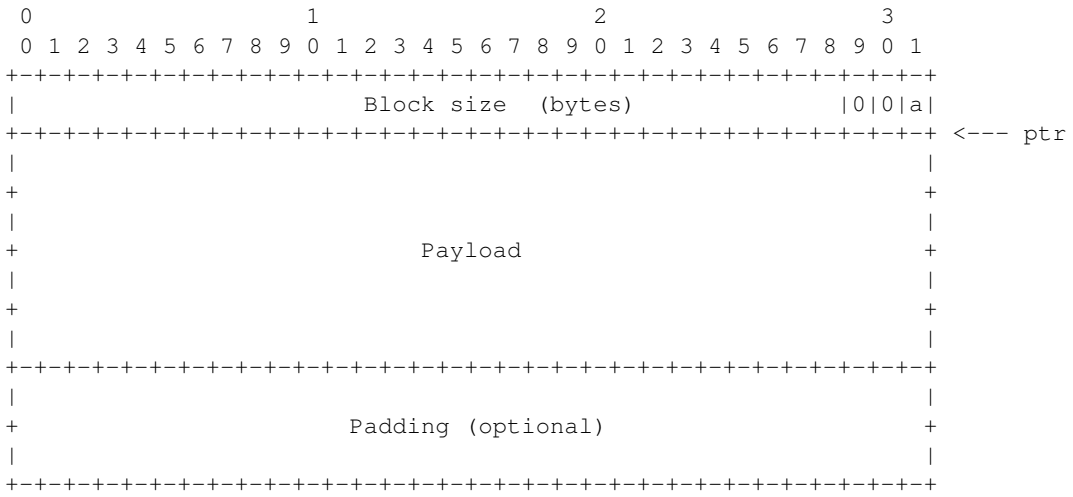
La fragmentation interne apparait lorsque la taille d'un bloc est supérieure à la quantité de mémoire demandée. Cela arrive notamment lorsqu'il faut satisfaire des contraintes d'alignement. Par exemple, on vous demande d'avoir un alignement sur 32 bits. Cela signifie que si l'on appelle `mymalloc` en demandant 510 bytes de mémoire, votre fonction devra allouer un bloc arrondi au multiple de 4 suivant, c'est à dire 512. Il y aura donc 2 bytes de gaspillés.

La fragmentation externe apparait lorsqu'il existe des blocs libres entre des blocs alloués, trop petits pour satisfaire les demandes d'allocation. Par exemple, si trois appels sont faits à `mymalloc` de resp. 64, 16, 64 bytes, puis qu'un appel à `myfree` est effectué pour libérer le bloc de 16 bytes, et finalement qu'un dernier appel à `mymalloc` est fait avec une valeur supérieure à 16 bytes, alors le bloc libéré ne pourra pas être utilisé car trop petit. On se trouve alors en présence de fragmentation externe.

Il est impossible d'éliminer totalement ces deux types de fragmentation, particulièrement la fragmentation externe car elle dépend de requêtes futures. Cependant, il est possible d'en limiter les impacts en implémentant des heuristiques (e.g., *best fit*, *worst fit*, *first fit*).

1.3 Implémentation

On vous demande de représenter les blocs en mémoire de la manière suivante.



Un bloc est donc composé d'un header de 32 bits, d'une payload et d'un padding optionnel. Les 29 bits de poids fort du header contiennent la taille du bloc (header,

payload et padding). Le bit de poids faible du header (représenté par a ci-dessus) vaut 0 si le bloc est libre et 1 si le bloc est alloué. Le payload représente la quantité de mémoire effectivement demandée lors de l'appel à `mymalloc` et le padding est la taille supplémentaire nécessaire pour satisfaire les contraintes d'alignement ou de taille de bloc. Notez que la distinction payload/padding est purement conceptuelle. En pratique, lorsque le bloc a été alloué, `mymalloc` n'a pas besoin de connaitre la limite entre le payload et le padding.

Dans l'exemple ci-dessus, ptr représente donc le pointeur qui retourné par `mymalloc`.

Notez que la taille du bloc présente dans le header permet de connaitre immédiatemment la position du bloc suivant. En effet, si on a p un pointeur vers un bloc donné et k la valeur du champ `Block size`, alors le bloc suivant se trouve à l'adresse $p + k$. Il peut être également utile d'implémenter un moyen de connaître la fin de la liste des blocs.

Voici la structure à utiliser afin de représenter le header.

```
struct block_header {
        unsigned int size : 29,
                     zero : 2,
                     alloc : 1;
}
```

On vous demande également d'implémenter la fonction `void *calloc(size_t size);`, qui est identique à malloc excepté que les données du bloc alloué doivent être initialisées à zéro.

1.4 Tests unitaires

Afin de vérifier le bon fonctionnement de votre programme, nous vous demandons d'implémenter des tests unitaire à l'aide de `CUnit` [1]. L'utilisation de `CUnit` est obligatoire. Ces tests sont **importants** et peuvent vous éviter de longues heures de debug.

1.5 Rapport

Vous devez écrire un rapport au format **PDF** d'une ou deux pages, contenant les explications de votre implémentation, les difficultés rencontrées, les cas couverts par vos tests unitaires ainsi qu'une figure montrant les résultats d'une mesure de performance de votre implémentation.

1.6 Reviews

Lorsque vous aurez rendu le projet, chaque membre d'un groupe recevra indépendamment deux projets à review. Vous devrez donc analyser et commenter un autre code que le vôtre. Les informations sur les reviews arriveront en temps voulu.

2 Délivrables

La timeline du projet est la suivante:

- 24/02: Lancement du projet
- 16/03: Remise du projet

[1] `https://sites.uclouvain.be/SystInfo/notes/Outils/html/cunit.html`

- 16/03: Lancement des reviews
- 11/04: Remise des reviews

Le projet devra être remis sur Moodle dans une archive au format **nom1_nom2.zip** où nom1 et nom2 sont le nom des membres de votre groupe. Cette archive doit contenir le code source de votre implémentation avec un **Makefile** permettant de compiler votre projet et de lancer les tests, ainsi que votre rapport au format **PDF**.