TP Systèmes 1 : Allocateur de mémoire virtuelle (aka malloc)

Ensimag 2A, édition 2023-2024

1 Allocation de mémoire virtuelle : malloc et free

Ce sujet vous propose d'implanter un allocateur dynamique de mémoire, très semblable à l'implantation du `malloc()` sous Linux (GNU libc). Il utilise en particulier la même gestion globale (petite, moyenne et grande taille), des seuils et des algorithmes similaires. Votre allocateur est utilisable à la place de celui du ńvéritableż `malloc()` pour tous vos processus, dans les cas simples.

1.1 Attendus

Acceptable : Faire l'implémentation correcte de l'allocateur pour les grandes et les petites tailles.

Bien : Faire l'implémentation correcte pour les tailles moyennes.

Très bien : Avoir une implémenation qui fonctionne en pour les processus 32 et 64 bits.

Excellent : Avoir une bibliothèque qui fonctionne en multi-thread

1.2 Fondamentaux sur les allocateurs

Il y a deux types d'allocateurs. Les allocateurs en mémoire virtuelle (celui de ce sujet) et les allocateurs pour la mémoire physique. Un système d'exploitation utilise, en interne, un allocateur de mémoire physique, pour gérer la répartition de la mémoire physique (RAM). À quelques détails, ce sont les mêmes principes de bases. Sous Linux, l'allocateur physique utilise l'algorithme du *Buddy*, l'algorithme que nous utiliserons dans ce TP pour gérer les tailles moyennes de notre allocateur virtuel.

La gestion de la mémoire signifie :

— connaître les zones mémoires utilisées ainsi que celles libres, i.e. disponibles pour le système ou l'utilisateur lorsqu'ils en demandent l'allocation,
— allouer des zones mémoires libres lorsque l'utilisateur, les demande,
— libérer des zones mémoires utilisées lorsque l'utilisateur, les rend.

Le code final est petit (109 lignes de code dans notre solution), mais comme toujours en "Système", le temps pour faire le sujet dépendra surtout de vous, de votre compréhension du sujet et votre maîtrise de C. **Pour aller plus vite, nous vous conseillons de : travailler à deux SIMULTANÉMENT sur le code, avec votre binôme ; de vous mettre d'accord en dessinant sur une feuille de papier ce que vous voulez faire avec vos pointeurs avant de coder ; les parties du sujet étant de difficulté croissante, de ne commencer à coder la partie suivante qu'après avoir débugué (et donc compris) la partie précédente**.

Ce sujet se veut progressif et construit pas-à-pas la solution. Il propose de faire d'abord les grandes tailles, pour manipuler des pointeurs, et comprendre comment `malloc()` détecte les débordements et conserver la trace des tailles allouées.

Ensuite, il traite les petites tailles, qui sont gérées avec un *pool* (une liste simplement chaînée) de *chunks* disponibles, des morceaux de taille identique. Lorsqu'il n'a plus de *chunk*, une demande est faite au système, avec un mécanisme de *recursive doubling* pour amortir le coût.

Enfin, pour les tailles intermédiaires, l'algorithme essaie de conserver des zones libres contiguës les plus grandes possibles. L'algorithme choisi, celui du buddy, s'écarte un peu de celui de GNU `malloc()`.

1.3 Travail demandé

Pour les différents TP et l'examen de TP final, les différents squelettes utilisent le même environnement de compilation.

Vous devez créer un fichier AUTHORS à la racine avec vos prénoms, noms et logins dedans. Par exemple pour l'équipe d'Alice Recoque et Robert Tarjan

```
Alice Recoque recoquea
Bob Tarjan tarjantr
```

Ensuite vous devriez pouvoir compiler le squelette fourni dans le répertoire `build/`. La première compilation s'effectue donc en tapant :

```
cd ensimag-malloc
cd build
cmake .. # deux points pour indiquer que le CMakeList.txt est
         # dans le répertoire parent
make
make test
make check
```

2 Marquage des blocs

La mémoire disponible dans un ordinateur, même si elle est grande, n'est pas infinie. Pour chaque zone de mémoire allouée, il faut être capable de conserver sa taille afin de, lors de sa libération, soit la conserver pour réutilisation (allocation petite et moyenne), soit la rendre au système (allocation grande).

Pour cela, chaque bloc alloué par `malloc()` est marqué. Aux deux bouts de la zone, deux valeurs sont écrites : un nombre *magique* qui servira à détecter les débordements, et la taille (cf fig 1). Pour ce TP, notre marquage est un trop gros, mais simple et robuste.

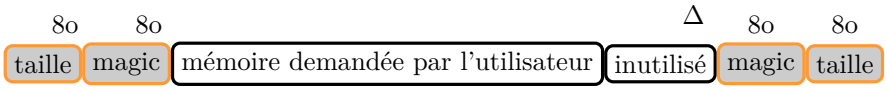


FIGURE 1 – Marquage d'un bloc

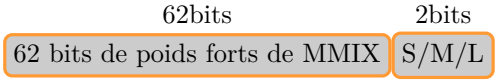
La *taille* vraiment allouée est : la taille demandée par l'utilisateur + $\Delta + 4 * 8$ octets. C'est cette `taille` totale qui est stockée dans le marquage.

2.1 Calcul de la valeur magique

Pour notre implantation, la valeur magique sera calculée comme suit :

L'adresse de la zone allouée (donc commençant par le marquage) est utilisé comme initialisation d'un mécanisme de tirage aléatoire simple sur 64 bits (générateur congruentiel linéaire) sur 64 bits, proposé par Donald Knuth, un des fondateurs de l'algorithmique moderne, et de quelques autres contributions majeures.

Les deux bits de poids faible de cette valeur sont ensuite modifiés pour prendre une valeur codée entre 0 et 3 désignant le type d'allocation (petite, moyenne ou grande).



2.2 Travail demandé

Dans le projet, vous devez coder les fonctions d'écriture et de lecture du marquage dans `src/mem_internals.c`

— `void *mark_memarea_and_get_user_ptr(void *ptr, unsigned long size, MemKind k)` écrit le marquage dans les 16 premiers et les 16 derniers octets du bloc pointé par `ptr` et d'une longueur de `size` octets. Elle renvoie l'adresse de la zone utilisable par l'utilisateur, 16 octets après `ptr` .
Pour calculer la valeur magique vous aurez besoin de la fonction `knuth_mmix_one_round(unsigned long l)` et des opérateurs binaires de C.

Pour rappel, `A & 0b111UL` garde les trois bits de poids faible de A, `A & ~(0b111UL)` garde le complémentaire (UL pour avoir la valeure comme un `unsigned long`), et `A | B` pour faire un OR bits à bits (une fusion, si les bits à 1 de A et B ne sont pas aux mêmes endroits).

— `Alloc mark_check_and_get_alloc(void *ptr)` lit le marquage, qui commence 16 octets avant `ptr` et renvoie les valeurs lues : début de l'adresse de la zone, taille totale et type d'allocation dans une variable de type `Alloc`. Elle vérifie la cohérence de la valeur magique.
Cette fonction vérifie enfin si le marquage de fin de la zone est identique au marquage de début. Pour cela vous pouvez utiliser les `assert(...)` afin de « planter ». Dans la « vraie vie », lorsque qu'une écriture de l'utilisateur déborde de la partie qui lui est dévolue, cela change la valeur magique. C'est souvent pour cela que la fonction `free()` plante en présence de bugs mémoire.

Après l'implantation de ces deux fonctions, vous pouvez déjà utiliser les grandes allocations (plus de 128 ko) dans le `memshell`. Elles sont gérées une par une par le système d'exploitation. La fonction qui les allouent est la fonction `mmap()` que vous verrez plus tard à nouveau dans ce cours.

Attention à l'arithmétique des pointeurs ! Elle multiplie les opérations de la taille du type pointée !

```
1 TOTO *ptr = 0;
2 TOTO *decal = ptr + 1;
3 assert( decal == (TOTO *)sizeof(TOTO));
```

`decal` est donc 1 TOTO plus loin en mémoire. Si TOTO est le type `int`, `decal` vaut 4. Si TOTO est le type `long` ou `void *` (`void **ptr`), `decal` vaut 8, sur les architectures 64 bits. Si TOTO est le type `char` ou `void`, `decal` vaut 1.

3 Les petites allocations : un pool de taille fixe

Une liste chaînées de petites zones libre, toutes de taille identique, est conservée dans un *Pool*. Lorsqu'une allocation de moins de 64 octets est demandée, cette allocation est servie en utilisant un des éléments libres, de taille 96 octets (64 + 32 pour le marquage). Il suffit de prendre le premier élément de la liste chaînée. Lorsqu'une petite allocation est libérée, elle est remise dans le Pool. Il suffit de l'insérer en tête de liste. Par souci de simplicité, lorsque la liste est vide, il faudra demander une réallocation avec la fonction `mem_realloc_small` qui fera pointer la tête de liste vers le début de la nouvelle zone de mémoire, puis de remplir, en utilisant cette zone, la liste chaînée. À chaque réallocation, la taille demandée au système double.

Pour initialiser la zone réallouée lorsque la liste est vide, il suffit donc de placer tous les 96 octets, l'adresse de l'élément suivant qui est 96 octets plus loin.

Pour écrire une adresse, on peut utiliser comme type un pointeur universel (`void *`). Pour placer cette adresse à une adresse spécifique, on peut utiliser un pointeur de pointeur universel (`void **`).

```
void **A = 0x0001234;
*A = (void *)0x000F00F;
```

Pour écrire à l'adresse 0x0001234, l'adresse Ox000FOOF. Attention à l'arithmétique des pointeurs lors de vos calculs d'adresses !

3.1 L'arène

La tête de liste, ainsi que la taille courante pour l'allocation sont stockée dans une variable globale (`MemArena arena;`). On trouve aussi dans cette variable les têtes de listes pour l'algorithme du buddy pour les allocations de taille moyenne, ainsi que les variables traçant les tailles utilisées par le *recursive doubling* des réallocations.

La tête de liste pour les petites allocations est donc `arena.chunkpool` .

Bonus multithreadé (ne pas implanter avant que tout ne marche)

Une idée simple est de n'utiliser qu'une arène et de la protéger des accès concurrents avec un mutex (cf. les cours, TD et TP sur la programmation concurrente).

Une seconde idée simple est d'avoir une arène par thread. Ainsi il n'y a pas de partage, il n'y a pas besoin de synchronisation entre les threads et ainsi pas de contention sur l'accès à l'arène. Comme toutes les données importantes sont dans l'arène, pour obtenir un allocateur multi-thread qui utilise une arène par thread, il suffit d'ajouter le mot clef `_Thread_local` devant la déclaration de la variable. Il n'a pas été ajouté par défaut, car il devient alors plus compliqué d'avoir accès à la structure avec un débogueur.

Mais dans cette version simpliste, il y a de nombreuses limitations sur la gestion des libérations et une accumulation potentielle au fil du temps d'une occupation de la mémoire liée à des threads terminés depuis longtemps.

Pour avoir un allocateur multi-thread efficace avec peu de contention, le nombre d'arènes est lié au nombre de curs (le carré du nombre de curs par exemple) et les threads sont assignés à une des arènes à leurs créations. La contention est réduite et l'usage de la mémoire reste controllée.

3.2 Travail demandé

Dans le projet, vous devez coder les fonctions d'allocations `emalloc_small()` et `efree_small()` dans `src/emalloc_small.c` .

— La fonction `void *emalloc_small(unsigned long size)` enlève le premier élément de la liste chainée pointée par `arena.chunkpool`, appelle la fonction de marquage en lui donnant l'adresse de l'élément, sa taille (`CHUNKSIZE`) et le type (`SMALL_KIND`) puis renvoie le résultat de la fonction de marquage.
Lorsque la liste `arena.chunkpool` est vide (`NULL`), vous devez appeller la fonction
`unsigned long mem_realloc_small()`. Cette fonction mettra à jour la variable et renvoie la taille en octet du bloc dont l'adresse est maintenant dans `arena.chunkpool`.
Vous devez construire une liste chainée de chunks en utilisant ce bloc. Il suffit de chainer vos pointeurs vers le chunk suivant tous les 96 octets (`CHUNKSIZE`).
— La fonction `void efree_small(Alloc a)` replace le chunk, décrit dans la variable `a`, en tête de la liste chainée.

4 Les allocations moyennes : algorithme du compagnon (buddy)

Dans les méthodes d'allocation par subdivision, les tailles de mémoires allouées sont quantifiées : elles sont exprimées en multiples d'une certaine unité d'allocation et les tailles permises sont définies par une relation de récurrence. Deux systèmes usuels sont :

— le système binaire (1,2,4,8...)
— le système Fibonacci (1,2,3,5,8,13...)

Nous allons utiliser le système binaire.

Pour chacune des tailles, une liste des blocs libres est conservée dans une des cases d'une table appelée "Table des Zones Libres" (TZL), qui est dans ce TP le tableau `arena.TZL[]`). Dans le système binaire, il y a donc une liste des blocs de taille 2^0 dans la case 0, 2^1 dans la case 1, 2^2 dans la case 2, ...2^i dans la case i.

Les informations sur les zones libres sont stockées dans les zones libres elle-même, en particulier le pointeur vers le suivant dans la liste.

Lorsque l'on manque de blocs d'une certaine taille S_i, on subdivise un bloc de taille S_{i+1}. Dans le système binaire, un bloc de 8 est donc divisé en 2 blocs de 4. Ces deux blocs issus d'un même bloc original sont dit "compagnons" ("buddy" en anglais). En pratique, pour chaque bloc, l'adresse de son compagnon est fixe.

Ce mode d'allocation a 2 avantages :

— la recherche d'un bloc libre de la bonne taille est rapide
— trouver le compagnon d'un bloc est facile puisque l'on peut calculer son adresse et donc le chercher dans la liste des zones libres de la même taille que le bloc libéré.

4.1 Calcul du compagnon

Si les blocs de mémoire sont correctement alignés, l'adresse du compagnon est exactement l'adresse du bloc XOR (opérateur `^` en C) la taille des deux blocs compagnons.
Par exemple pour un bloc de taille 2, à l'adresse 4, son compagnon est à l'adresse 6.

```
0b100 ^ 0b010 == 0b110 // 4 XOR 2 == 6
0b110 ^ 0b010 == 0b100 // 6 XOR 2 == 4
```

Les blocs dans l'allocateur ont une taille minimale de 128 octets. La fonction `mem_realloc_medium` les alignent correctement. Et le premier bloc alloué est de la taille des allocations larges : il n'y aura donc jamais plus d'une réallocation à la fois.

4.2 Travail demandé

Il est demandé de réaliser la gestion des allocations de taille moyenne avec l'algorithme du buddy. Pour cela, vous fournirez les fonctions suivantes :

— `void *emalloc_medium(unsigned long size)` calcule l'indice de la TZL contenant la bonne taille (la plus petite pouvant ontenir l'allocation) en utilisant la fonction `puiss2()`. Si un bloc est disponible, il est marqué et l'adresse utilisateur est retournée.
S'il n'y a pas de bloc de la bonne taille, il faut découper, récursivement, un bloc plus gros. Comme toutes les tailles sont des puissances de 2, couper un bloc en deux, au milieu, donne deux blocs dont la taille est la puissance de 2 immédiatement inférieure. Chaque bloc inutilisé est inséré dans la bonne liste de `arena.TZL` (cf fig 2).
S'il n'y a pas de grands bloc disponible avant d'atteindre l'indice `(FIRST_ALLOC_MEDIUM_EXPOSANT + arena.medium_next_exponant)`, il n'y en a pas de plus grands et il faut appeller la fonction `mem_realloc_medium()` qui ajoutera un bloc de la bonne taille, bien aligné, à cet indice.

— `void efree_medium(Alloc a)` libère le bloc commençant à l'adresse décrite dans la structure `a`. Pour cela, il faut calculer l'adresse du buddy. Vérifier s'il est présent dans la bonne liste de `arena.TZL`. S'il n'est pas présent, il faut insérer le bloc dans la liste. S'il est présent, il faut enlever le buddy de la liste et recommencer avec le bloc fusionné et la liste de l'indice suivant.

Ces fonctions seront réalisées au sein du module `mem_medium.c`.

5 Compléments techniques

5.1 Rendu des sources

L'archive des sources que vous devez rendre dans `Teide` est généré par le makefile créé par cmake :

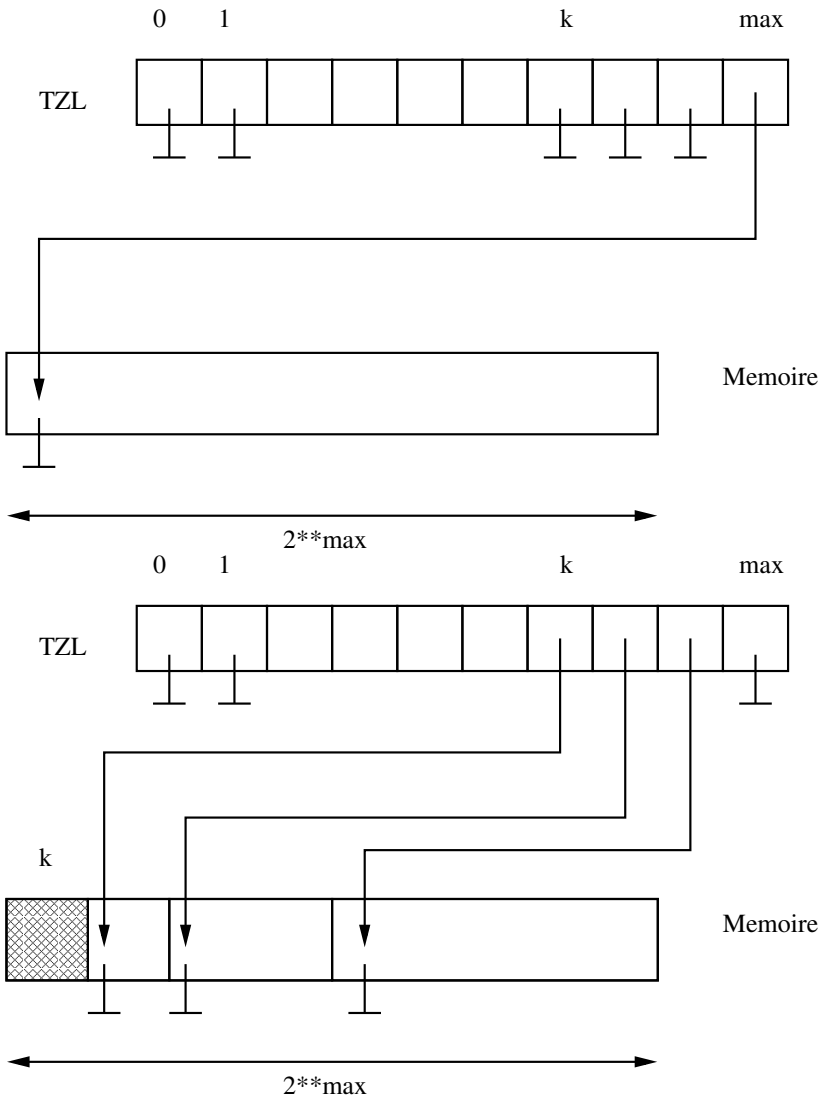


FIGURE 2 – Découpage d'un bloc de taille 2^{max} pour obtenir un bloc de taille 2^k

```
cd ensimag-allocateur
cd build
make package_source
```

Il produit dans le répertoire `build`, un fichier ayant pour nom (à vos login près) `Emalloc-2.0.login1-login2-login3-Source.tar.gz`.

C'est ce fichier tar qu'il faut rendre.

5.2 Déboguage

Pour vous aider à comprendre ce qui se passe dans votre implantation, deux programmes sont compilés par le squelette : un shell interactif en C, des tests unitaires en C++.

Dans tous les cas, en plus d'utiliser des printf pour avoir une idée de l'état de votre programme vous avez tout intérêt à utiliser un débogueur.

5.2.1 Comment arrêter le débogueur sur un changement à une adresse particulière

Pour lancer le débogueur sur le programme memshell

```
gdb memshell
```

Puis dans le débogueur, vous pouvez utiliser les commandes « standards » que vous trouverez dans tous les débogueurs (souvent avec le même nom).

```
break emalloc_small # pour demander à s'arrêter au début de la
 ↪  fonction
run # pour démarrer
print/x ptr # pour afficher la variable ptr en hexadécimal
```

Mais vous avez aussi accès à de nombreuses fonctions très utiles pour ce TP, comme des breaks conditionnels ou des watchpoints. Un watchpoint sert à découvrir les instructions exactes qui modifient une position particulière de la mémoire (ici Oxffaba500) (par exemple une adresse pointée dans vos structures), il suffit de faire :

```
watch *0xfffaba500 # similaire à un break, mais s'active lorsque
                   # la position mémoire indiquée est écrite
run
```

5.3 Adressage 32 et 64 bits

Votre code devra compiler et tourner sur des machines 64 bits. Les types de base et les pointeurs n'étant pas toujours de la même taille, il vous faudra donc utiliser `sizeof()` pour obtenir la taille d'un type de manière portable, ou la fixer avec `#include <stdint.h>`.

5.4 Compilation et test unitaires

Le squelette fourni inclut l'utilisation de `cmake` pour construire automatiquement les Makefile utiles.

Vous devez créer un fichier AUTHORS à la racine avec vos prénoms, noms et logins dedans.

La création des Makefile s'effectue en utilisant `cmake` dans un répertoire où seront créés les fichiers générés. Le répertoire "build" du squelette sert à cet usage. Tout ce qui apparaît dans "build" pourra donc être facilement effacé.

La première compilation s'effectue donc en tapant :

```
cd ensimag-malloc
cd build
cmake ..
make
make test
make check
```

et les suivantes, dans le répertoire `build`, avec

```
make
make test
make check
```

Une batterie de test vous est fournie pour vous aider à construire plus rapidement un code correct. "make test" devrait donc trouver la plupart des bugs de votre programme au fur est à mesure que vous l'écrivez. Vous pouvez lancer directement l'exécutable `alloctest` pour avoir plus de détails avec "make check", ou lancer individuellement les tests qui vous intéressent `./alloctest --help`.

5.5 Shell interactif

Un petit shell interactif `memshell` vous est fourni. Il est utilisable seul, mais il permet aussi de réaliser des tests en utilisant votre débogueur pour inspecter l'état de votre allocateur, par exemple l'état de la TZL.

```
gdb ./memshell
break emalloc_medium
```

```
layout
run
alloc 64
print arena.TZL
```