**Université de Saint-Quentin-en-Yvelines**
**Master 1: Calcul Haut Performance et Simulation**

# *Rapport de projet Allocateur mémoire*

---

# PROJET D'ALLOCATEUR MÉMOIRE

---

***Préparé par:***

M$^r$.Khadimou Rassoul DIOP
M$^r$.Hery ANDRIANANTENAINA

M$^r$. Julien ADAM Encadreur

***Année 2019-2020***

# Contents

# 1 INTRODUCTION

Ce projet a pour but d'appliquer les connaissances acquises durant les cours. Premièrement, nous serons amenés à faire des supports des fonctions d'allocation de mémoire, c'est à dire d'intercepter des fonctions depuis une bibliothèque tierce par exemple la fonction malloc(), qui alloue des tailles en octet, et renvoie un pointeur sur la mémoire allouée, ainsi que les fonction realloc() et calloc(). Ensuite, détailler les implémentations sur les algorithmes que nous avons utilisé. Puis nous allons expliquer la méthode de compilation, le recyclage de blocs effectués ainsi que le mapping mémoire.Par la suite, le projet sera consacré à une amélioration en tenant compte des problèmes d'alignement de mémoire, du multithreading et à la fin des mesures de performances pour faire des comparaisons de nos allocateurs avec celle des allocateurs existant.

# 2 Support des fonctions d'allocation

Dans cette partie nous nous sommes inspirés du manuel de programmeur Linux qui détaille très bien ce que font les fonctions d'allocateur mémoire comme malloc, calloc et realloc qu'on a réimplémenté. Pour ce faire, on commence par initialiser nos fonctions d'allocation en les définissant static et de valeur NULL. Ainsi on définit des allocateurs statiques qui vont allouer avant le dlsym renvoit les fonctions d'allocation de la bibliothèque standard. Puis on a fait une fonction init qui se charge de l'interception des bibliothèques tierces c'est à dire qu'on renvoie l'adresse des allocateurs de la bibliothèque tierce dans la zone mémoire où on a chargé nos fonc-

tions en appliquant un dlsym à nos fonctions initialisées précédemment. Ceci fait, si notre dlsym ne renvoie pas d'erreur on aura réussi à charger notre bibliothèque avec nos fonctions d'allocateur mémoire dynamique.

### 2.1 MALLOC :

La fonction malloc() alloue size octets, et renvoie un pointeur sur la mémoire allouée. Le contenu de la zone de mémoire n'est pas initialisé. Si size est nulle, malloc renvoie soit NULL ou un unique pointeur qui pourra être passé ultérieurement à free() avec succès. Pour l'implémentation de cette fonction on commence par définir une valeur d'initialisation qui vaut zéro. On définit un buffer de 1024 octets et un compteur en variables globales. Si notre fonction d'allocation donc notre malloc vaut NULL, si la valeur d'initialisation ne vaut pas 0 on le réinitialise et on rappelle la fonction init pour réintercepter malloc avec dlsym. Sinon, on regarde la taille allouée avec le malloc si elle est inférieure à celle du buffer, on définit un pointeur de taille supérieure ou égale à celle du buffer, on met à jour la valeur du compteur en lui additionnant la taille allouée puis on retourne le pointeur sinon on exit. A la fin de la fonction on appelle notre fonction d'allocateur initialisée au début dans un pointeur que l'on va retourner.

### 2.2 FREE :

La fonction free() libère l'espace mémoire pointé par ptr, qui a été obtenu lors d'un appel antérieur à malloc().Si le pointeur ptr n'a pas été obtenu par l'un de ces appels, ou s'il a déjà été libéré avec free(ptr), le comportement est indéterminé. Si ptr est NULL, aucune

opération n'est effectuée. Pour cette implémentation si le pointeur est entre le buffer et le buffer + count on affiche le message de libération de la mémoire sinon on appelle la fonction free initialisée dans init().

### 2.3 CALLOC :

La fonction calloc() alloue la mémoire nécessaire pour un tableau de n éléments de size octets, et renvoie un pointeur vers la mémoire allouée. Cette zone est remplie avec des zéros. Si n ou si size est nulle, calloc renvoie soit NULL ou un unique pointeur qui pourra être passé ultérieurement à free() avec succès. Pour notre implémentation, notre fonction prend en paramètres n éléments et n octets, si notre malloc implémenté précédemment est NULL on appelle malloc dans un pointeur alloué à n éléments fois n octets et on remplit la zone mémoire de ce pointeur avec des 0. Puis on renvoie ce pointeur. A la fin on appelle notre fonction d'allocateur initialisée au début dans un pointeur que l'on va retourner.

### 2.4 REALLOC :

Pour l'implémentation de cette fonction si notre malloc implémenté précédemment est NULL l'appel est équivalent à malloc on appelle donc la fonction malloc. Si la zone pointée était déplacée(on peut vérifier ceci avec la fonction memmove), un free(ptr) est effectué et on retourne le pointeur. On appelle notre allocateur défini dans init() assigné dans un nouveau pointeur pour modifier la taille du pointeur à n octects puis on renvoie ce pointeur.

## 3 Méthodes de compilation et système de build

On fait un Makefile et une bibliothèque dynamique via le Makefile qu'on va analyser avec ldd. Dans notre makefile on utilise les compilateurs gcc et icc. Par ailleurs on rajoute la variable d'environnement PRELOAD pour effectuer le préchargement de l'allocateur.

## 4 Mapping mémoire

Malloc contient une taille et mmap permet une allocation des pages (mémoire virtuelle) pour satisfaire cette taille. Le but est ici de répondre à ces demandes de mémoire efficacement. Dans l'implémentation de cette fonction on ouvre l'application compilée contre l'allocateur en read/write. On crée une nouvelle projection dans l'espace d'adressage virtuel du processus en utilisant mmap. On initialise le contenu de la projection du fichier avec 1024 octects. On tronque le fichier avec la taille passée en argument puis à la fin on retourne la projection.

## 5 Recyclage de blocs

On a appliqué l'algorithme du pool de recyclage. On définit une structure chaînée de données free element. On alloue un gros tableau de blocs (pool), chaque allocation consiste à sélectionner une case non utilisée dans le tableau. À la libération, il suffit de marquer le bloc comme libres. Potentiellement, on peut agrandir le pool (mais pas avec realloc).On initialise la structure de stockage d'éléments à NULL que l'on a appelé ici freestore puis le nombre de blocs numchunk. Dans la

fonction de recyclage on définit le pointeur de structure e. Puis on regarde si le freestore n'est pas bien initialisé, la liste chaînée est vide et on alloue un gros bloc à e. Ainsi on traite le gros bloc comme une liste chaînée de numchunk - 1 free element et on fait pointer le dernier petit bloc vers NULL. A la fin on retourne les éléments du free store. Enfin on a fait une fonction deallocate afin de pouvoir libérer les blocs si on a envie. Cette fonction est simple, on met juste les éléments du pointeur alloué dans le freestore après l'avoir casté en structure de données.

# 6 Test de performances

Les performances de notre malloc
(Byte, KiByte, MiByte, GiByte), min, max, moyenne, mediane, byte per cycle, giga byte per second
( 2400, 2, 0, 0); 22532.2600000000; 12431.6800000000; 17481.9700000000; 12915.0500000000; 0.1858297103; 0 GiB/s;

Les performances du malloc de la lib C
(Byte, KiByte, MiByte, GiByte), min, max, moyenne, mediane, byte per cycle, giga byte per second
( 2400, 2, 0, 0); 22539.8000000000; 11271.0900000000; 16905.4450000000; 10364.3500000000; 0.2315630020; 0 GiB/s;

Les performances du jemalloc (Byte, KiByte, MiByte, GiByte), min, max, moyenne, mediane, byte per cycle, giga byte per second
( 2400, 2, 0, 0); 23406.1000000000; 11641.3800000000; 17523.7400000000; 10962.6900000000; 0.2189243698; 0 GiB/s;

Dans nos tests on itère 100 fois et on mesure les cycles et on chronomètre le temps d'éxécution avec la fonction rdtsc(). Ainsi on obtient les résultats ci-dessus. En comparant les bytes per cycle on remarque que celui de notre malloc est le plus petit avec une valeur de 0,18 puis vient celui du jemalloc qui vaut 0,21 et enfin on a celui de la lib C qui vaut 0,23.

# References

[1] https://stackoverflow.com/questions/6083337/overriding-malloc-using-the-ld-preload-mechanism

[2] https://medium.com/a-42-journey/how-to-create-your-own-malloc-library-b86fedd3

[3] https://embeddedartistry.com/blog/2017/02/22/generating-aligned-memory/#dynamic-memory-alignment