CR – TP allocateur

API/CPS INFO3, Polytech Grenoble, 2021-2022

DECORSAIRE Mattéo et ONGHENA Edgar

En résumé

Nous avons réussi à implémenter l'allocateur ainsi que les 6 extensions proposées (ci-dessous). Des informations complémentaires à ce compte-rendu, notamment des explications sur la structure de l'allocateur, se trouvent dans le `README.md` à la racine du dépôt. Nous avons beaucoup utilisé `git` pour collaborer efficacement et le graphe (https://gricad-gitlab.univ-grenoble-alpes.fr/onghenae/info3-allocateur/-/network/main) en témoigne.

Les cibles make fournies à l'origine marchent toujours, et peuvent être utilisées pour tester notre allocateur. Le paragraphe suivant détaille le fonctionnement des tests.

Tests

Notre code comporte différent types de tests. Le fichier `/tests/general.c` contient des tests unitaires généraux. Il y a également des tests spécifiques à l'intégration valgrind dans le même dossier (on en parle après). Tous les tests peuvent être lancés avec `make tests`. Le fichier `.gitlab-ci.yml` les fait s'exécuter en intégration continue pour vérifier qu'à tout moment dans notre développement, il n'y a pas de régression.

5.1 Débordement mémoire

Nous avons implémenté le système de gardes de sorte à ce qu'il soit activable ou désactivable sans recompiler (il faut cependant réinitialiser l'allocateur à chaque fois comme la réprésentation mémoire change).

Un type `guard` et une valeur constante sont définies dans un `#if` du préprocesseur. Cela permet de définir une taille de garde qui dépend de l'alignement. En prenant une taille de garde égale à l'alignement, on se facilite beaucoup la tâche pour garder l'entièreté de la structure de l'allocateur alignée. Sur une cible 64 bits, le type `guard` est représenté par un `__uint128_t`, un nombre sur 16 octets car l'alignement fait 16 octets. Les alignements 16, 8 et 4 octets sont gérés.

Lors de l'allocation, si les gardes sont activés, la taille allouée réellement fait `2*sizeof(guard)` de plus que ce que l'utilisateur demande. Cela permet de placer deux gardes autour de la zone que l'utilisateur aura a sa libre disposition. Le pointeur renvoyé est bien sûr celui de la zone modifiable, prise en sandwich. Les gardes sont initialisés à leur valeur constante.

Lors d'une libération, si les gardes sont activés, le pointeur passé en paramètre est décrémenté de `sizeof(guard)` pour retrouver la "vraie" zone allouée, comprenant le garde de gauche. On vérifie que les gardes ont la bonne valeur et on libère la zone. Sinon, on renvoie `false` pour signaler l'erreur et la variable globale `LAST_ERROR` se voit affecter la constante d'enum `GUARD_VIOLATION`.

5.2 Corruption de l'allocateur

Deux techniques sont mises en œuvre. D'abord, on est capables de savoir si une adresse passée à `mem_free` correspond à une zone allouée ou non. Si elle ne l'est pas, `LAST_ERROR = NOT_ALLOCATED` et `mem_free` renvoie faux.

De plus, on peut détecter lorsque la taille d'une zone libre est plus grande que l'espacement entre l'adresse de cette zone libre et la suivante (si `.next != NULL`). Si on se rend compte d'une corruption sur ce point là, le code d'erreur est `FB_LINK_BROKEN`.

5.3 Autres politiques (worst fit, best fit)

Les deux autres politiques ont été implémentées et sont testées dans nos tests unitaires.

5.4 Comparaison de `mem_fit_worst` et `mem_fit_best`

mem_fit_best permet d'obtenir l'adresse de la plus petite zone suffisament grande pour y allouer l'espace demandé.
Avantage : Cela évite de gacher les grosses zones inutilement en les decoupant en zones plus petites. Une zone petite sera allouée pour une petite quantité d'espace demandé
Inconvenient : on risque d'obtenir de minusculules zones libres inalouables en pratique si la zone

alloué ne prend pas totalement l'espace libre

mem_fit_worst permet d'obtenir l'adresse de la plus grande zone suffisament grande pour y allouer l'espace demandé.
Avantage : Cela permet d'éviter les petites zones libres inalouables comme décrites precedemment.
Inconvénient : on risque de remplir un grand espace libre par pleins de zones allouées, et quand on voudra allouer une zone grande, on ne pourra plus car on aura commencé par remplir les zones le plus grandes.

5.5 Compatibilité avec valgrind

Notre allocateur dépend de `<valgrind/valgrind.h>` et utilise 3 de ses macros (init, alloc, free) pour être compatible avec l'analyse de mémoire fournie par ce dernier outil.

Un test est mis en place pour vérifier que valgrind détecte bien les fuites. Ce test peut être lancé avec `make tests/valgrind`. Il contient un test où la mémoire est bien libérée et valgrind se termine sans erreur, ainsi qu'un test "controle" où l'on fait volontairement fuiter de la mémoire et on s'assure que valgrind renvoie une erreur comme attendu.

On notera que lorsque les gardes sont activés, valgrind ne se voit notifier que des zones renvoyées à l'utilisateur. Les gardes ne sont pas inclus. L'idée est de ne pas surprendre un utilisateur qui déboguerait et trouverait des addresses incohérentes avec ce qu'il reçoit de notre allocateur.

5.6 Autre extension: FFI Rust

En "bonus", j'ai (Edgar) implémenté un wrapper Rust pour l'allocateur.

Rust est un langage Orienté Objet basé sur la composition (à la différence de Java, C#, Python, JavaScript, OCaml et d'autres qui sont basés sur l'héritage), et dont la gestion de mémoire est déclarative (il n'utilise pas de ramasse-miettes, ni de gestion manuelle de la mémoire). C'est un langage de bas niveau, qui peut notamment être utilisé pour créer des systèmes d'exploitation, mais sa bibliothèque standard est tellement facile à utiliser qu'il a des airs de langage de haut niveau et peut être utilisé comme tel. De plus, Rust est entièrement memory-safe, c'est à dire qu'il est impossible de créer des segfaults, sauf à l'intérieur d'un bloc `unsafe { ... }` où certaines règles sont assouplies. Cette gestion de la mémoire repose sur l'idée de traiter les pointeurs comme étant de type complètement différents selon leur provenance. Un pointeur vers un `int` dans la mémoire statique s'apellera `&'static i32` tandis qu'un `int` alloué sur le tas aura comme type `Box<i32>` et une liste de `int` aura comme type `Vec<i32>`. En traitant différemment chacun de ces pointeurs, les destructeurs associés aux types peuvent libérer la mémoire si nécéssaire, ne rien faire sinon. Rust garantit également l'absence de race conditions lors de la programmation de logiciels asynchrones.

Il est possible, en Rust, de définir soi-même une structure qui joue le rôle d'allocateur, en implémentant le trait `GlobalAllocator` dessus. Un trait est l'équivalent d'une interface en Java. Cela étant fait, on peut définir une instance de cette structure comme étant effectivement l'allocateur à utiliser:

```
6 #[global_allocator]
7 static ALLOCATOR: Info3AllocateurGlobal = Info3AllocateurGlobal;
```

En combinant cette fonctionalité avec la possibilité de faire de la FFI avec du C, j'ai raccordé proprement notre allocateur à Rust, sous la forme d'une bibliothèque, et ai également ajouté la possibilité de changer la fonction de fit depuis Rust.

Il y a en réalité deux façons de faire des allocateurs en Rust. Celle présentée au dessus est plus ancienne, et créé un allocateur *global*, mais la possibilité de créer des allocateurs *spécifiques à certains types nécéssitant des allocations* est en cours d'implémentation. En définissant ma toolchain en "nightly", j'ai accès à la version instable du langage avec ce type d'allocateur expérimental et j'ai également implémenté notre allocateur avec cette nouvelle méthode.

Le code Rust est testé. Les tests sont dans `/rust/tests/` et peuvent être exécuteés avec la commande `cargo test` une fois l'outillage Rust installé[1] (https://md.edgar.bzh/s/4QeWck_8x#fn1). Certains tests testent l'allocateur global, d'autres testent l'allocateur spécifique à certains types.

Une documentation HTML de la bibliothèque Rust peut être obtenue avec `cargo doc --open`. L'intégration continue la génère automatiquement depuis le dépôt, elle est accessible publiquement ici (https://onghenae.gricad-pages.univ-grenoble-alpes.fr/info3-allocateur/info3_allocateur/index.html).

1. https://rustup.rs/ (https://rustup.rs/) ↩ (https://md.edgar.bzh/s/4QeWck_8x#fnref1)