Département d'informatique et de recherche opérationnelle (DIRO)
IFT2035 — Concepts des langages de programmation (automne 2025)
Auteur: *Mattéo Delabre* · Dernière mise à jour: *2025-11-02*

Devoir 2 — *Allocateur de mémoire*

Instructions Ce devoir doit être remis sur Studium **en équipe de deux** avant *le dimanche 16 novembre à 23 h 59*. Vous devez inscrire votre équipe sur Studium avant de pouvoir remettre votre devoir.

Intégrité Il n'est pas autorisé de copier du contenu de sources extérieures, y compris des devoirs d'autres équipes, sans les mettre entre guillemets ou sans en citer l'origine. L'utilisation de modèles de langue tel ChatGPT est autorisée seulement pour l'écriture d'une partie du code. Le rapport doit être entièrement écrit par votre équipe. L'utilisation de ces modèles est toutefois autorisée pour effectuer une révision linguistique du rapport. Au moins 50% de votre code doit avoir été écrit par votre équipe.

Pointage Le devoir vaut pour 5% de la note du cours.

Votre objectif pour ce devoir est d'utiliser le langage **Zig** pour réaliser un **allocateur de mémoire** à pile permettant le recyclage de mémoire. La tâche est divisée en trois étapes de difficulté croissante.

Simple allocateur à pile

Comme première étape pour se familiariser avec la création d'un allocateur, mettons au point un allocateur *à pile*. Dans ce style d'allocateur, les allocations se font les unes à la suite des autres en laissant, au besoin, de l'espace pour l'alignement. L'adresse de la prochaine case mémoire disponible dans le bloc de mémoire est conservée et avancée à chaque nouvelle allocation. Dès que cette adresse atteint la fin du bloc de mémoire disponible, aucune autre allocation n'est possible.

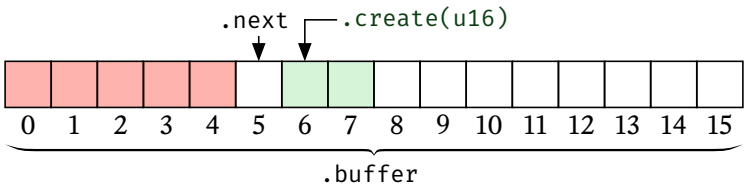

Fig. 1. – État de la mémoire gérée par l'allocateur à pile lors d'un appel à `.create(u16)`. Les cases rouges sont déjà allouées, les vertes sont utilisées pour la nouvelle allocation.

Avec cette stratégie, il n'est pas possible de libérer une allocation individuelle. Les méthodes `.destroy` et `.free` sont donc inopérantes. Il est seulement possible de libérer toutes les allocations simultanément, en détruisant et reconstruisant l'allocateur.

Notre allocateur, nommé `AllocateurPile`, dispose d'un attribut `.buffer`, qui est la tranche de mémoire dans laquelle les allocations doivent se faire, et d'un attribut `.next`, qui est le numéro de la prochaine case mémoire disponible.

```
const AllocateurPile = struct {
    buffer: []u8,
    next: usize,
    // ...
};
```

> TÂCHE Téléchargez le fichier `1-pile.zig`. Complétez la méthode `.alloc` de l'allocateur `AllocateurPile` afin de le rendre opérant. Laissez les autres éléments du fichier inchangés.

(Utilisez la commande `zig test 1-pile.zig` *pour tester votre programme.)*

Étiquetage des allocations

Afin de permettre la libération des allocations individuelles et la réutilisation de la mémoire libérée, il faut *étiqueter* chacune des allocations effectuées avec leur taille et l'information de si elles ont été libérées ou non.

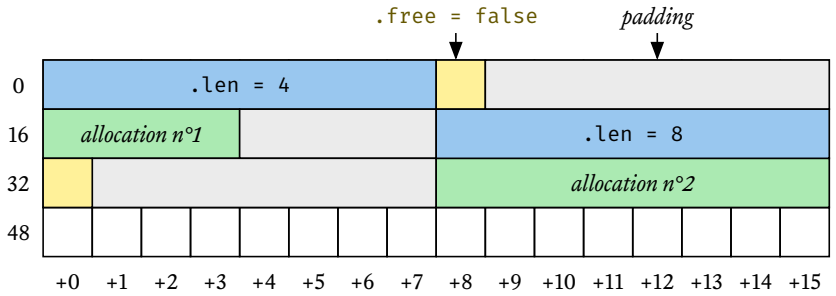

Fig. 2. – État de la mémoire gérée par l'allocateur avec étiquettes après deux allocations, l'une pour 4 octets et l'autre pour 8 octets. La prochaine allocation se fera à l'adresse 48.

Nous choisissons de stocker cette information à côté de chaque allocation. Ainsi, chaque allocation est précédée d'un en-tête: une valeur de type `Header`. L'attribut `.len` contient la taille de l'allocation (sans l'espace occupé par l'en-tête). L'attribut `.free` permet de savoir si l'allocation en question a été libérée.

```
const Header = struct {
    len: usize,
    free: bool,
};
```

À cette étape, l'objectif est seulement de stocker cette information additionnelle à côté de chaque allocation et de la mettre à jour lors de l'appel de la méthode `.free`. Pour le moment, nous ne permettons pas encore la réutilisation de la mémoire libérée.

TÂCHE Téléchargez le fichier `2-etiquette.zig`. Complétez les méthodes `.alloc`, `.getHeader` et `.free` de l'allocateur `AllocateurEtiquette` afin de le rendre opérant. Laissez les autres éléments du fichier inchangés.

(Utilisez la commande `zig test 2-etiquette.zig` *pour tester votre programme.)*

Recyclage de mémoire

Finalement, modifions notre stratégie d'allocation pour permettre le *recyclage*, c'est-à-dire la réutilisation des allocations libérées individuellement. À chaque demande d'allocation de n octets, il faut:

- Passer au travers des allocations existantes pour trouver la première allocation libérée dont la taille est $\geq n$.
- S'il existe une telle allocation libérée, la réutiliser.
- Sinon, créer une nouvelle allocation de taille n à la fin du bloc de mémoire.

TÂCHE Téléchargez le fichier `3-recycle.zig`. Complétez les méthodes `.alloc`, `.getHeader` et `.free` de l'allocateur `AllocateurRecycle` afin de le rendre opérant. Laissez les autres éléments du fichier inchangés.

(Utilisez la commande `zig test 3-recycle.zig` *pour tester votre programme.)*

Critères d'évaluation

Votre travail est évalué sur les critères suivants:

- **3 points** sont attribués sur le code remis. Il est évalué à l'aide de tests automatiques. Le critère le plus important est que votre code fonctionne correctement. L'exécution de votre code ne devrait générer aucun avertissement ou erreur. La qualité du code est également évaluée : votre code devrait être concis et clair et inclure des commentaires au besoin. Sachez que le devoir peut être résolu en ajoutant ~60 lignes aux fichiers.
- **2 points** seront attribués sur le rapport remis. Votre rapport, d'une longueur maximale de 3 pages, devrait faire état de votre démarche dans la résolution du devoir. Expliquez-y les choix que vous avez faits lors de l'écriture du code. Si applicable, détaillez-y votre utilisation de modèles de langue. Montrez-y les tests que vous avez réalisés pour vérifier que votre code est correct.