Harjit Lidder 4SI4

# Rapport : Conception d'un Allocateur en Rust :

**Introduction**

Dans le cadre de ce projet, j'ai développé un allocateur de mémoire basé sur une liste chaînée (Free List Allocator) en Rust en no_std. Ce rapport documente mes choix de conception, les défis rencontrés, et les avantages et inconvénients de mon approche par rapport à d'autres types d'allocateurs.

---

**Choix de Conception**

**Type d'allocateur : Free List Allocator**

J'ai choisi d'utiliser un allocateur basé sur une liste chaînée. Ce type d'allocateur gère la mémoire libre sous la forme de blocs liés entre eux, ce qui permet une gestion dynamique et flexible de la mémoire.

**Justifications**

1. **Simplicité** : Le modèle de liste chaînée est relativement simple à implémenter et à comprendre.
2. **Évolutivité** : Ce modèle permet d'ajouter ou de libérer dynamiquement de la mémoire sans besoin de gestion complexe.

3. **Adaptabilité** : Idéal pour des environnements sans système d'exploitation (bare-metal) où l'utilisation de bibliothèques standard est limitée.

**Configurations Techniques**

Pour supporter ce projet, j'ai défini une configuration spécifique dans le fichier config.toml au sein du répertoire .cargo :

```
[build]
target = "x86_64-unknown-none"
[target.x86_64-unknown-none]
rustflags = ["-C", "link-arg=-Ttext=0x1000"]
linker = "ld.lld"
```

Ce paramétrage a permis de résoudre l'erreur liée au "linking with cc" en désactivant les fichiers de démarrage standard, adaptés aux environnements sans système d'exploitation.

---

**Comparaison avec d'autres types d'allocateurs**

**Allocateur par "Bump"**

- **Avantages** :
    - Très rapide grâce à sa simplicité (augmentation linéaire d'un pointeur).
    - Idéal pour des usages où la désallocation n'est pas nécessaire.
- **Inconvénients** :
    - Ne supporte pas la désallocation, ce qui le rend inadapté pour des environnements complexes où la mémoire doit être fréquemment libérée.

**Allocateur de type "Slab"**

- **Avantages** :
    - Optimisé pour allouer de nombreux objets de taille fixe.
    - Très performant dans les environnements où la taille des objets est prévisible.
- **Inconvénients** :

- Moins flexible pour des tailles d'objets variables.
- Plus complexe à implémenter que le Free List Allocator.

**Allocateur chainé (mon choix)**

- **Avantages** :
  - Flexible pour des tailles d'allocation variables.
  - Supporte la désallocation, ce qui est essentiel pour une gestion dynamique de la mémoire.
- **Inconvénients** :
  - Plus lent que les allocateurs comme "Bump" en raison de la recherche dans la liste des blocs libres.
  - La fragmentation de la mémoire peut devenir un problème dans certains cas.

---

**Défis Rencontrés**

1. **Erreur "linking with cc failed"** :
   - **Problème** : Cette erreur est survenue en raison d'un mauvais paramétrage pour un environnement bare-metal.
   - **Solution** : Ajout des drapeaux rustflags dans le fichier .cargo/config.toml, désactivant les fichiers de démarrage standard.

2. **Gestion de la sécurité avec unsafe** :
   - **Problème** : La gestion directe des pointeurs avec unsafe a nécessité une attention particulière pour éviter les erreurs.
   - **Solution** : Chaque méthode unsafe a été soigneusement documentée avec des commentaires explicatifs, en utilisant les sections de sécurité de rustdoc.
3. **Manque de support pour les tests automatisés** :
   - **Problème** : En mode no_std, les outils de test standard comme cargo test ne sont pas utilisables.

- **Solution** : J'ai testé manuellement des scénarios basiques d'allocation et de désallocation dans la fonction _start.

---

**Fonctionnement de l'Allocateur**

**Initialisation**

L'allocateur est initialisé avec un tableau fixe de 1024 octets défini dans la fonction _start.

```
static mut HEAP: [u8; 1024] = [0; 1024];

unsafe {
    ALLOCATOR.init(HEAP.as_ptr() as usize, HEAP.len());
}
```

**Exemples de Fonctions Clés**

1. **Fonction insert_free_region**

Cette fonction insère une région de mémoire libre dans la liste chaînée.

```
/// # Safety
/// L'appelant doit garantir que l'adresse est alignée et que la taille est suffis
pub unsafe fn insert_free_region(&self, addr: usize, size: usize) {
    let alignment = mem::align_of::<Block>();

    if size < mem::size_of::<Block>() || addr % alignment != 0 {
        return;
    }

    let new_block = addr as *mut Block;
    (*new_block).size = size;

    (*new_block).next = *self.free_list.get();
    *self.free_list.get() = new_block;
}
```

**Utilité** : Permet d'ajouter une région de mémoire inutilisée, essentielle pour réutiliser des espaces libérés.

2. **Fonction find_block**

Cette fonction cherche un bloc libre correspondant à une taille et un alignement donnés.

```
pub unsafe fn find_block(&mut self, size: usize, alignment: usize) -> Option<(*mut
    let mut current_block = *self.free_list.get();
    let mut previous_block: *mut Block = null_mut();

    while !current_block.is_null() {
        if let Ok(allocation_address) = Self::check_block_allocation(current_block
            if !previous_block.is_null() {
                (*previous_block).next = (*current_block).next;
            } else {
                *self.free_list.get() = (*current_block).next;
            }

            return Some((current_block, allocation_address));
        }

        previous_block = current_block;
        current_block = (*current_block).next;
    }

    None
}
```

**Avantages et Inconvénients de mon Implémentation**

**Avantages**

- **Flexibilité** : Supporte des tailles d'allocation variables.
- **Dynamisme** : Permet une gestion fine de la mémoire allouée et libérée.
- **Compatibilité** : Bien adapté aux environnements sans système d'exploitation.

**Inconvénients**

- **Performance** : Recherche dans la liste chaînée peut être coûteuse en temps.
- **Fragmentation** : Risque accru de fragmentation mémoire dans certains scénarios.

---

**Documentation et Rustdoc**

J'ai soigneusement documenté toutes les sections unsafe en utilisant rustdoc pour garantir une compréhension claire des responsabilités de l'appelant. Chaque fonction critique est accompagnée de commentaires explicatifs sur sa sécurité et son fonctionnement.

Le trait GlobalAlloc :

L'implémentation de GlobalAlloc est cruciale pour tout allocateur personnalisé en Rust. Ce trait permet de définir la manière dont la mémoire brute est allouée et libérée dans l'application. En intégrant GlobalAlloc, j'ai pu rendre mon allocateur compatible avec des abstractions de haut niveau comme Box ou Vec, ce qui est essentiel pour une intégration fluide dans des projets plus complexes.

De plus, cela garantit que l'allocateur est utilisé globalement, en remplaçant le comportement par défaut de Rust. Cela simplifie la gestion et centralise toutes les opérations mémoire sous une seule implémentation, améliorant ainsi la cohérence et la maintenabilité.

---

**Conclusion**

Ce projet m'a permis d'explorer les concepts fondamentaux de la gestion de mémoire en Rust en no_std, dans un environnement sans système d'exploitation.