Rapport

Nathan Caracciolo, Arnaud Bihan

January 27, 2022

1 Préambule

La gestion de la mémoire fait référence à la gestion de la mémoire du Tas. En effet c'est la seule section du programme dont la taille peut varier pendant l'exécution de ce dernier, l'ensemble des autres zones mémoires sont allouées au chargement du programme. Même la pile dont le contenu évolue dynamiquement a un espace restreint dont les dimensions sont fixes. Le tas contrairement à la pile n'est pas ordonné et structuré ce qui permet d'y ajouter et d'y supprimer des éléments peu importe l'ordre.

2 Dans la théorie

Dans les premières théories informatiques(cf machine de Turing), on suppose l'existence d'une mémoire infinie permettant l'exécution de tout programme car on s'intéresse principalement aux capacités de cette discipline nouvelle à résoudre des problèmes complexes et concrets.

3 Les débuts de la programmation

Les tout premiers ordinateurs sont conçus pour des calculs simples (arithmétiques et logiques), sont mono-threadé et possèdent une quantité très limité de mémoire. Par conséquent, les premiers langages de programmation n'ont pas forcément implémenté de gestion de la mémoire dynamique, les quantités de mémoires sont humainement représentable et on préfère séquencer la mémoire avant l'exécution. C'est une approche qui reste très fréquente pour des programmes embarqués où les quantités de mémoires sont minimales.

4 Gestion manuelle de la mémoire

C'est avec l'apparition des premiers systèmes d'exploitation que l'on voit apparaître le besoin de gérer la mémoire dynamiquement. En effet au lieu de considérer le matériel comme destiné à une seule tâche ou un ensemble de tâches fixé par avance. On commence à imaginer des systèmes généralistes capables de s'adapter à la demande des programmes.

L'implémentation de la gestion de la mémoire manuelle est rudimentaire, il s'agit d'une encapsulation simple des appels systèmes permettant les allocations dynamiques de mémoire des différents systèmes d'exploitation.

Cette solution est souvent donnée dans une librairie du langage, pas dans la syntaxe du langage à proprement parler. C'est le cas pour le langage C dont la gestion de la mémoire dynamique se fait en passant par la librairie standard "stdlib.h". C'est la solution choisit par la quasi-totalité des langages contemporains de C comme Fortran ou Algol.

4.1 Les outils permettant l'allocation manuelle de la mémoire

Un système de gestion manuelle de la mémoire nécessite un ensemble élémentaire de fonctions.

- Une fonction d'allocation de la mémoire demandant au système d'exploitation une zone mémoire d'une certaine taille. Il s'agit par exemple des fonctions malloc et calloc pour le langage C.

```
int* tab = (int*)malloc(sizeof(int) * 100);

int* tab = (int*)calloc(100, sizeof(int));
```

- Une fonction de libération de la mémoire rendant au système d'exploitation une zone mémoire devenue inutile. Il s'agit par exemple de la fonction free pour le langage C.

```
free(tab);
```

- Une fonction de modification de la zone mémoire demandant au système d'exploitation l'augmentation ou la réduction de la dite zone. Il s'agit par exemple de la fonction realloc pour le langage C.

```
tab = realloc(tab, sizeof(int) * 200);
```

4.2 Qualités et défauts

C'est une solution simple à implémenter, et qui permet de garder des performances cruciales pour certaines applications. Elle est cependant beaucoup plus difficile à utiliser pour un développeur et peu mener à des programmes buggés.
Les trois principaux bugs sont:

- L'utilisation de zones libérées, ou non allouées
- La libération de zones déjà libérées
- La non libération de zones non utilisées (fuite de mémoire)

La solution manuelle reste très utilisée dans des programmes critiques qui nécessitent les meilleurs performances comme les systèmes d'exploitations, les applications embarquées ou les moteurs de jeux vidéos.

5 Gestion de la mémoire automatique

5.1 Garbage Collector

À la même période un langage en tout point différent du C et de ses concepts, le langage Lisp développe un nouveau concept. La gestion automatique de la mémoire à l'aide d'un outil : le garbage collector.

Le principe est simple : automatiser la gestion de la mémoire pour l'enlever de l'esprit des programmeurs. Permettant de grandement simplifier certaines conceptions de programmes. Le but du garbage collector est d'assurer que l'on ne manquera jamais de mémoire à cause d'objets non atteignable par le programme. Par exemple le garbage collector de java, contenu dans la JVM peut choisir ou non de ne pas activer le nettoyage de la mémoire si l'espace occupée par ces derniers n'est pas important mais lorsqu'il atteindra un certain niveau, le nettoyage s'effectuera.

Cette approche va énormément se répandre et devenir l'approche majoritaire. La quasi-totalité des langages récents utilisent un garbage collector, c'est le cas de Java, C#, Kotlin, Go ... En somme, il existe plusieurs algorithmes de Garbage collector:

- le Mark-Sweep consiste en un parcours de graphe et qui marque les sommets que l'on atteint. Les sommets marqués deviennent alors les objets atteignables par le programme et les sommets non-marqués sont donc des objets morts sur lesquels il est impossible d'obtenir une référence et qui peuvent donc être ramassé par le Garbage Collector.

- le marquage Tricolor consiste à marquer les sommets en 3 couleurs. Si un sommet est blanc l'objet associé est alors candidat pour le recyclage, s'il est marqué gris, alors l'algorithme de marquage Tricolor n'est pas fini car il s'agit des sommets sur lesquels nous devront raisonner. Enfin, s'il marqué noir, il n'y a plus rien à faire avec ce dernier et il ne sera pas concerné par la phase de recyclage

Les phases de détections des objets morts ainsi que leurs suppressions peuvent se faire de manière parallèle mais après ces deux phases, il y a une phase de restructuration de la mémoire, une phase de "compact". Cette phase est la plus dur dans la réalisation d'un garbage collector. Nous pensions cette phase impossible à rendre parallèle mais il s'est avérer que nous avions tort et que le créateur du garbage collector d'OCaml a récemment réussi. Il s'agit tout de même d'une implémentation complexe à mettre en place.

5.2 Reference counting

On peut aussi parler de l'approche de gestion de mémoire automatique par comptage de référence qui est un algorithme extrêmement simple implémenté par Python qui consiste juste à associer à chaque objet un compteur de référence, c'est à dire combien de références pointent vers cet objet et une fois que ce chiffre tombe à zéro on peut libérer la ressource. En théorie cet algorithme semble correct mais montre des problèmes dans la pratique lorsque l'on tombe dans le cas de référence circulaire. Pour pallier à ce problème Python implémente un garbage collector qui passe en seconde main afin de régler ces problèmes laissés par la passe de comptage de référence.

6 Le cas Rust

Pour éviter le sur-coût des garbages collectors tout en gardant la simplicité de programmation des langages ayant une gestion automatique de la mémoire, les créateurs du langage Rust se sont tournés vers le champs des analyses statiques afin d'obtenir une solution alliant puissance et simplicité. Le langage Rust utilise deux concepts pour modéliser la gestion de la mémoire étant: Borowing et OwnerShip. Le choix de Rust est intéressant car il n'autorise qu'un seul parent, qu'un seul "owner" à un objet. Tous les autres objets qui voudront accéder à un objet devront emprunter (Borrowing) une référence de cet objet et l'objet mourra simplement lorsque son "owner" meurt ou lorsque décide de le relâcher (à la manière d'un free en c)

Dans Rust chaque référence que l'on utilise possède son temps de vie (Lifetime), Rust peut alors analyser statiquement si les lifetimes son cohérent entre eux et prévenir le programmeur qu'il y a une erreur dans l'utilisation de référence. Il est aussi possible d'annoter une variable d'un lifetime pour être sûr de ce que Rust comprendra de notre programme.

Cette particularité réduit tout compte fait l'expressivité du langage et Rust n'est pas exempt de tout défauts, il existe malgré tout des fuites de mémoires dû aux RefCell qui utilisent un comptage de référence similaire à python et donc apporte les mêmes problèmes à Rust.

7 Références:

- https://fr.wikipedia.org/wiki/John_McCarthy
- https://fr.wikipedia.org/wiki/Lisp
- https://en.wikipedia.org/wiki/Memory_management
- https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html
- https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html
- https://towardsdatascience.com/memory-management-and-garbage-collection-in-python-c1cb51d1612c
- https://en.wikipedia.org/wiki/Tracing_garbage_collection
- https://doc.rust-lang.org/book/
- https://howtodoinjava.com/java/garbage-collection/all-garbage-collection-algorithms/
- https://www.geeksforgeeks.org/mark-and-sweep-garbage-collection-algorithm/
- https://mobiskill.fr/blog/conseils-emploi-tech/quest-ce-que-rust-et-pourquoi-lutiliser/
- Rust présentation : https://www.youtube.com/watch?v=5C_HPTJg5ek
- Rust lifetime : https://www.youtube.com/watch?v=juIINGuZyBc
- Rust ownership : https://www.youtube.com/watch?v=VFIOSWy93H0
- Rust Memory Leaks : https://www.youtube.com/watch?v=XtA7n9_3bTM