Systèmes d'exploitation - Mémoire

Othmane AJDOR

2018-2019

Table des matières

1 Allocateurs mémoires

— Coté kernel (+embarqué minimaliste)
— mémoire virtuelle <== le cours du jour

Ca necessite un partage efficace (allocation au plus juste) des espaces mémoire dans un espace trop petit et borné

— satisfaire le client dans le working set si trop gros => simuler une mémoire plus grande.

La mémoire virtuelle est découpée en zone :
— les zones libres sont libres : l'allocateur peut ecrire dedans
— ca coute moins cher en terme de cycles CPU de gerer une liste de mémoire libre que de gérer une liste de toutes les zones utilisées.

Les algorithmes de chainage simple comme FAT32 fragmentent, ce qui n'est pas recommendé sur un espace large. EXT4 et NTFS ne le font pas.

2 Mémoires

2.1 Malloc & free

— petit -> la limite est de 64octets, si on demande 1o, on en aura 64.
— moyen
— grand -> 128 ko, la gestion de l'unité est gérée par l'OS != mise en RAM paresseuse et par morceau

2.2 SLAB, SLOB, SLUB

Allocateur spécialisé (1 taille, 1 type d'objet, ...) L'avantage de ces allocateurs c'est qu'ils sont plus simple que les autres, ce qui nous permet de faire des choses comme lock_free en multi-thread.

2.3 alloca

Allocateur historique qui ne sert plus à rien. Il faisait des allocations en pile et ne fait pas de libération.
Elle est faite automatiquement à la fin de la fonction.

2.4 Garbage collector

Allocation classique avec une libération à la charge du runtime

```
#include "gc.h"
{
    GC_INIT();
    ...
    void *p = GC_malloc(taille);
    ...
}
```

Le coût du garbage collector est un problème.

2.4.1 Base

On part d'une liste des racines en scannant les objets pour trouver ceux n'étant plus accessibles des racines.

2.4.2 Stétegie de scan

— Stop the world (Java)
— Mark and sweep (Go) [Dijkstra], on scan tout le temps

En java, on dispose de plusieurs zones :
— (les jeunes) eden -> si plein, on lance un scan mineur. Sinon on les passe dans S0 (survivor space) puis dans S1.
— (old generation) tenured et encore plus vieux les objets permanents
On ne s'occupe que des zones jeunes qui sont plus susceptibles d'être liberées.

Dans le cas d'objets qui pointent sur eux memes, on a besoin de tout parcourir depuis la racine et ce n'est pas suffisant.
C'est pour ça qu'on utilise des indicateurs sur les liens entre objets :
— weak, phantom. EZ, on libère
— soft (possible de GC ?) -> il faut vérifier tous les circuits et vérifier que tous les liens qui menent vers celui là sont soft.
— strong.
En GO, on dispose de la notion de tache qui permet de faciliter le lancement du garbage collector :
— On scan tout et tout le temps
— mark and sweep
GC Pacer : equilibrer l'execution entre calcul et GC.

En Rust, tout est à expliquer au compilateur qui fera les malloc et free

```
let mut a = 12; // a est mutable
let b = &a;
println!("{}", a); // affiche 12
println!("{}", b); // affiche 12, auto unref
a = 11;
println!("{}", a); // affiche 11
println!("{}", b); // ça ne compile pas
```

Pour savoir à quel moment il faut détruire, Rust utilise un paradigme appelé "proprietaire unique" où les pointeurs font des emprunts temporaires. Dans le code ci-dessus, b fait un emprunt temporaire. Entre le premier print de b et le deuxiemme , a a changé. Le deuxieme print de b n'est plus valide comme c'était un emprunt temporaire. Il faut refaire un emprunt. On n'a pas le droit de modifier une valeur après un emprunt. On n'a non plus le droit d'emprunter a et le modifier dans b.
C'est le borrow checker qui s'occupe de tout cela.

```
let v:Vect<T> = (1=1000000).map(|x| x*x).take(10).collect();
// (1=1000000) est un iterateur
// map est un iterateur
// take est un iterateur
// collect crée le vecteur (10 multiplications)
```