Gestion mémoire

Programmation avancée

Armaël Guéneau & Jacques-Henri Jourdan
(et une partie des transparents de Jean-Christophe Filliâtre)

Gestion mémoire
Programmation avancée
Mémoire virtuelle
Gestion mémoire côté langages
Introduction
Allocateur mémoire de type `malloc`
Destructeurs
Garbage collection : concepts
Comptage de références
GC traçant : mark & sweep
GC traçant : stop & copy
GC à générations

Projet

Le projet est en ligne sur la page du cours :
Implémentation d'un borrow checker pour MiniRust

Date de rendu : dimanche 25 mai

Mémoire virtuelle

Gestion mémoire côté langages

Introduction

Allocateur mémoire de type `malloc`

Destructeurs

Garbage collection : concepts

Comptage de références

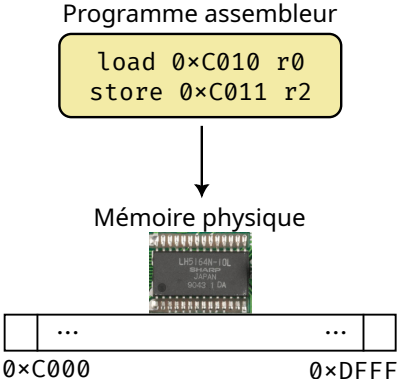
GC traçant : mark & sweep

GC traçant : stop & copy

GC à générations

Gestion mémoire, sur un système minimal

machine exécutant un seul programme à la fois, écrit en assembleur

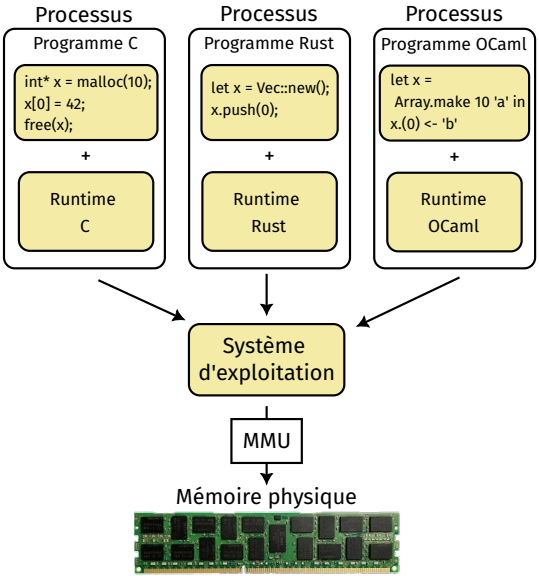


$\Longrightarrow$ le programme lit et écrit directement dans la mémoire physique

Gestion mémoire
Programmation avancée
Mémoire virtuelle
Gestion mémoire côté langages
Introduction
Allocateur mémoire de type malloc
Destructeurs
Garbage collection : concepts
Comptage de références
GC traçant : mark & sweep
GC traçant : stop & copy
GC à générations

Gestion mémoire, sur un système moderne

machine exécutant :
un système d'exploitation,
orchestrant plusieurs programmes
écrits dans des langages haut niveau



Mémoire virtuelle

Gestion mémoire côté langages

Introduction

Allocateur mémoire de type `malloc`

Destructeurs

Garbage collection : concepts

Comptage de références

GC traçant : mark & sweep

GC traçant : stop & copy

GC à générations

Comment on passe de “`Array.make 10 'a'`” à la mémoire physique de la machine ?

Mémoire virtuelle

Gestion mémoire côté langages

Introduction

Allocateur mémoire de type `malloc`

Destructeurs

Garbage collection : concepts

Comptage de références

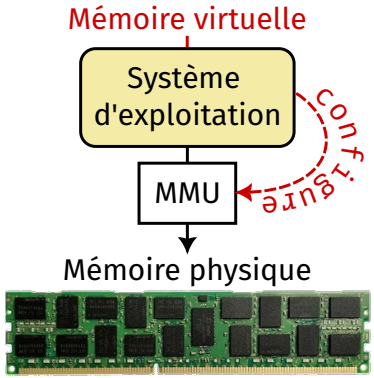
GC traçant : mark & sweep

GC traçant : stop & copy

GC à générations

1 Mémoire virtuelle

2 Gestion mémoire côté langages
- Introduction
- Allocateur mémoire de type `malloc`
- Destructeurs
- Garbage collection : concepts
- Comptage de références
- GC traçant : mark & sweep
- GC traçant : stop & copy
- GC à générations



Mémoire virtuelle

Gestion mémoire côté langages

Introduction

Allocateur mémoire de type `malloc`

Destructeurs

Garbage collection : concepts

Comptage de références

GC traçant : mark & sweep

GC traçant : stop & copy

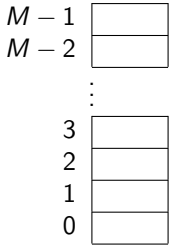
GC à générations

Mémoire physique

La mémoire physique d'un ordinateur est un grand tableau de M octets,

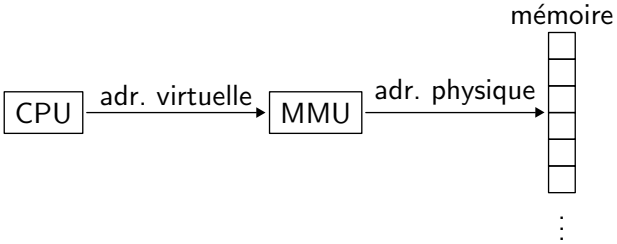
auxquels le CPU peut accéder en lecture et en écriture au moyen d'adresses physiques 0, 1, 2, etc.

M est de l'ordre de plusieurs milliards aujourd'hui (par exemple, $M = 2^{32}$ pour 4 Go de mémoire)



Mémoire virtuelle

Depuis longtemps, on n'accède plus directement à la mémoire : on utilise un mécanisme de mémoire virtuelle offert par le matériel,
à savoir le MMU (pour *Memory Management Unit*)



Il traduit des adresses virtuelles (dans $0, 1, \ldots, N-1$)
vers des adresses physiques (dans $0, 1, \ldots, M-1$).

Mémoire virtuelle, système d'exploitation et processus

La MMU est configurée par le système d'exploitation.

Le système d'exploitation configure la MMU différement pour chaque processus.

Le système d'exploitation assure que :
$\longrightarrow$ différents processus ne peuvent pas se marcher dessus en mémoire
$\longrightarrow$ un processus peut voir la mémoire comme une seule région contiguë

$\Longrightarrow$ **Chaque programme a l'illusion de disposer de l'intégralité de la mémoire (virtuelle) pour lui tout seul.**

Fonctionnement

La mémoire virtuelle est découpée en pages (par ex. de 4 ko chacune).

Le système d'exploitation maintient une table des pages **par processus**.

Dans la table, chaque page est soit :

- non allouée,
- allouée en mémoire physique (et le MMU renseigné),
- allouée sur le disque.

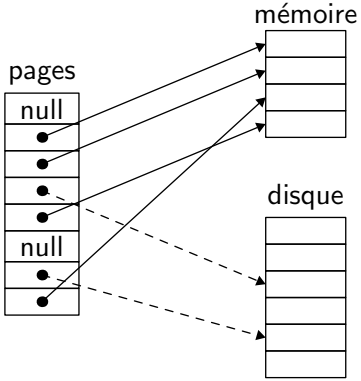
Quand l'exécution change de processus $\longrightarrow$ on change de table des pages dans la MMU

Gestion mémoire côté langages
Introduction
Allocateur mémoire de type `malloc`
Destructeurs
Garbage collection : concepts
Comptage de références
GC traçant : mark & sweep
GC traçant : stop & copy
GC à générations

Exemple

8 pages

- 2 non allouées
- 4 en mémoire physique
- 2 sur le disque



Fonctionnement (2)

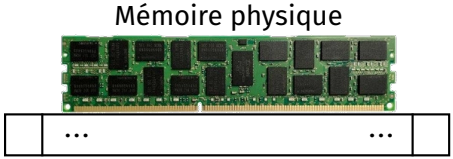
Quand le CPU veut lire ou écrire à une adresse virtuelle.
Le MMU effectue la traduction vers une adresse physique.

- Soit elle réussit et l'instruction est exécutée.
- Soit elle échoue et
  1. Une interruption est levée (*page fault*).
  2. Le système d'exploitation installe la page en mémoire physique (éventuellement en déplaçant vers le disque une autre page).
  3. L'exécution reprend sur la même instruction.

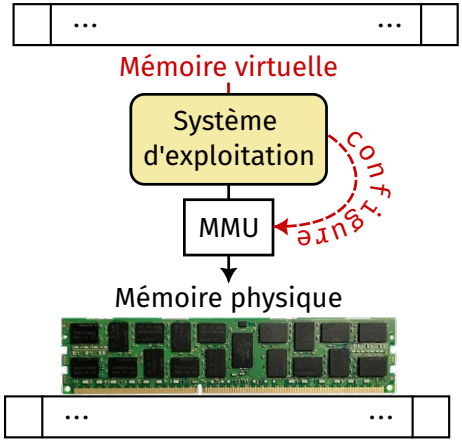
Pour en savoir plus :

Randal E. Bryant et David R. O'Hallaron
Computer Systems : A Programmer's Perspective
chapitre 9 Virtual Memory

Mémoire virtuelle : qu'a-t-on gagné ?

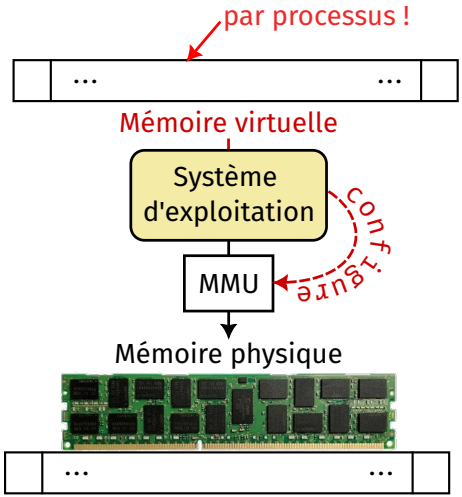


Mémoire virtuelle : qu'a-t-on gagné ?



Gestion mémoire côté langages
Introduction
Allocateur mémoire de type malloc
Destructeurs
Garbage collection : concepts
Comptage de références
GC traçant : mark & sweep
GC traçant : stop & copy
GC à générations

Mémoire virtuelle : qu'a-t-on gagné ?



Chaque programme peut faire comme s'il avait accès à la machine entière

1 Mémoire virtuelle

2 Gestion mémoire côté langages

- Introduction
- Allocateur mémoire de type `malloc`
- Destructeurs
- Garbage collection : concepts
- Comptage de références
- GC traçant : mark & sweep
- GC traçant : stop & copy
- GC à générations

Mémoire virtuelle

Gestion mémoire côté langages

Introduction

Allocateur mémoire de type `malloc`

Destructeurs

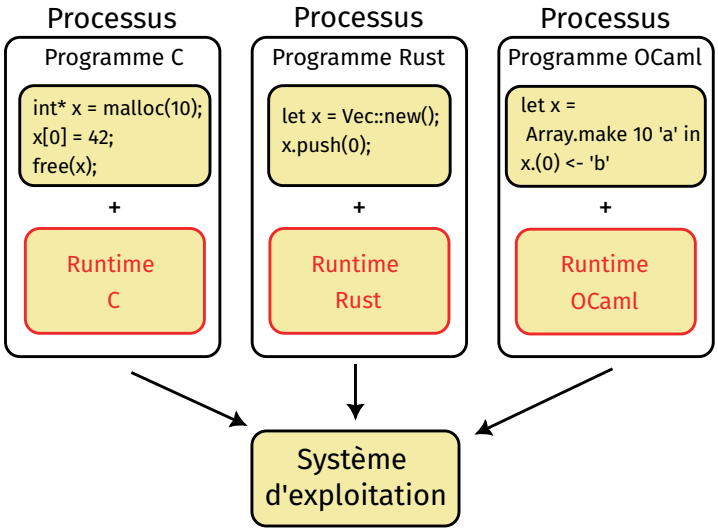
Garbage collection : concepts

Comptage de références

GC traçant : mark & sweep

GC traçant : stop & copy

GC à générations



Mémoire virtuelle
Gestion mémoire côté langages
Introduction
Allocateur mémoire de type malloc
Destructeurs
Garbage collection : concepts
Comptage de références
GC traçant : mark & sweep
GC traçant : stop & copy
GC à générations

Gestion mémoire

En pratique, un programme (C, Rust, OCaml) n'accède pas directement à la mémoire virtuelle. Ces langages fournissent des mécanismes plus flexibles de gestion mémoire.

Chaque langage fournit un mécanisme de **gestion mémoire** qui s'occupe de :

- réserver ("**allouer**") des morceaux de mémoire virtuelle
- les **libérer** quand ils ne sont plus utilisés

Le mécanisme de gestion mémoire choisi **varie beaucoup** entre langages de programmation, et est une partie importante du design d'un langage.

Mémoire virtuelle
Gestion mémoire côté langages
Introduction
Allocateur mémoire de type `malloc`
Destructeurs
Garbage collection : concepts
Comptage de références
GC traçant : mark & sweep
GC traçant : stop & copy
GC à générations

Allocation statique vs dynamique

Il est facile d'allouer de la mémoire statiquement, une fois pour toute au démarrage du programme.

Mais la plupart des programmes doivent allouer de la mémoire dynamiquement :

- soit implicitement, par des constructions du langage ;
  (objets, clôtures, etc.)
- soit explicitement, pour stocker des données dont la taille n'est pas connue statiquement (tableaux, listes, arbres, etc.).

Il convient généralement de la libérer.

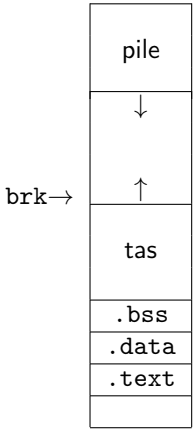
Mémoire virtuelle
Gestion mémoire côté langages
Introduction
Allocateur mémoire de type malloc
Destructeurs
Garbage collection : concepts
Comptage de références
GC traçant : mark & sweep
GC traçant : stop & copy
GC à générations

Le tas

Cette allocation dynamique se fait dans le tas.

Il est situé dans la mémoire virtuelle du processus, immédiatement au-dessus du segment de données.

Le système d'exploitation maintient son sommet dans une variable `brk` (*program break*).



Allocation simpliste

La façon la plus simple d'allouer de la mémoire consiste à augmenter `brk`.

L'appel au système d'exploitation

```
void *sbrk(int n);
```

incrémente `brk` de `n` octets et renvoie son ancienne valeur.

On peut inversement diminuer `brk` avec une valeur de `n` négative.

Cela nous limite à une utilisation du tas comme une pile.

(De nos jours, le programme peut faire appel un autre appel système, `mmap`, pour un peu plus de flexibilité, mais c'est hors de propos ici.)

Mémoire virtuelle
Gestion mémoire côté langages
Introduction
Allocateur mémoire de type `malloc`
Destructeurs
Garbage collection : concepts
Comptage de références
GC traçant : mark & sweep
GC traçant : stop & copy
GC à générations

Gestionnaire de mémoire

On veut un gestionnaire de mémoire permettant d'allouer et de libérer des blocs de mémoire **dans un ordre arbitraire**.

La libération peut être :

- explicitement réalisée par le programmeur
  (exemple : en C avec `malloc`/`free`),
- insérée par le compilateur à des points de programme spécifiques
  (exemple : RAII en Rust)
- automatiquement réalisée par le gestionnaire mémoire
  (on parle alors de GC, exemple : OCaml, ou `Rc` en Rust).

Mémoire virtuelle
Gestion mémoire côté langages
Introduction
Allocateur mémoire de type `malloc`
Destructeurs
Garbage collection : concepts
Comptage de références
GC traçant : mark & sweep
GC traçant : stop & copy
GC à générations

Allocateur mémoire de type `malloc`

Mémoire virtuelle

Gestion mémoire côté langages

Introduction

Allocateur mémoire de type `malloc`

Destructeurs

Garbage collection : concepts

Comptage de références

GC traçant : mark & sweep

GC traçant : stop & copy

GC à générations

Objectif

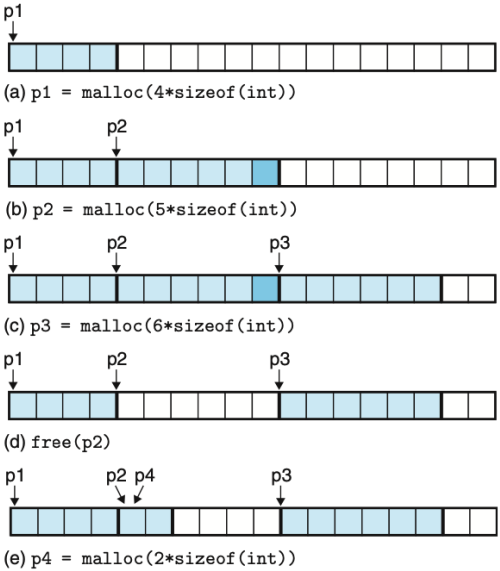
Un **allocateur** fournit deux opérations :

```
void *malloc(int size);
  // renvoie un pointeur vers un nouveau bloc
  // d'au moins size octets, ou NULL en cas d'échec

void free(void *ptr);
  // libère le bloc situé à l'adresse ptr
  // (doit avoir été alloué par malloc
  //  et ne pas avoir été déjà libéré,
  //  sinon comportement non spécifié)
```

En C, le langage ne fournit qu'un allocateur (gestion mémoire "manuelle") que le programmeur utilise directement.

En Rust, OCaml, le gestionnaire mémoire *inclut* un allocateur (plus d'autres choses...)

Exemple : séquence d'appels à malloc/free



Règles du jeu

Comment implémenter ces opérations par dessus `sbrk` ?

- On ne présume rien sur la séquence de `malloc` et de `free` à venir.
- La réponse à `malloc` ne peut être différée.
- Toute structure de données nécessaire à `malloc` et `free` doit être elle-même stockée dans le tas.
- Tout bloc renvoyé par `malloc` doit être aligné sur 8 octets.
- Tout bloc alloué ne peut plus être déplacé.

Il s'agit d'un problème délicat, avec de nombreuses réponses, souvent complexes.

Nous allons discuter une solution à ce problème.

Mémoire virtuelle
Gestion mémoire côté langages
Introduction
Allocateur mémoire de type `malloc`
Destructeurs
Garbage collection : concepts
Comptage de références
GC traçant : mark & sweep
GC traçant : stop & copy
GC à générations

Première idée

Les blocs, alloués ou libres, sont contigus en mémoire.

Ils sont chaînés : étant donnée l'adresse d'un bloc, on peut calculer l'adresse du suivant.

Mémoire virtuelle

Gestion mémoire côté langages

Introduction

Allocateur mémoire de type `malloc`

Destructeurs

Garbage collection : concepts

Comptage de références

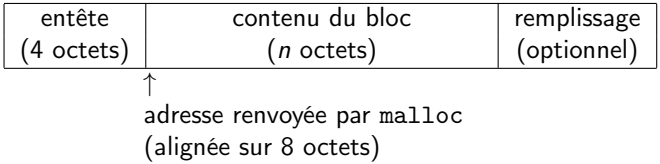
GC traçant : mark & sweep

GC traçant : stop & copy

GC à générations

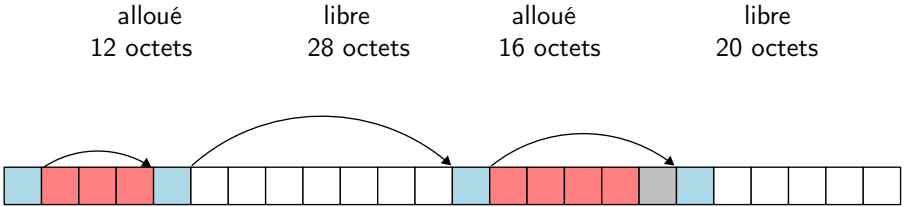
Mise en œuvre

- Un entête contient la taille (totale) et le statut (alloué / libre) ;
- suivent les n octets du bloc ;
- et un éventuel remplissage, garantissant une taille totale multiple de 8.



Mémoire virtuelle
Gestion mémoire côté langages
Introduction
Allocateur mémoire de type malloc
Destructeurs
Garbage collection : concepts
Comptage de références
GC traçant : mark & sweep
GC traçant : stop & copy
GC à générations

Exemple



- un carré = 4 octets
- bleu = entête / rouge = alloué / gris = remplissage / blanc = libre

Astuce

La taille totale étant un multiple de 8, ses trois bits de poids faible sont nuls.

On peut utiliser un de ces bits pour stocker le statut (alloué / libre).

Sur l'exemple précédent :

bit	5	4	3	2	1	0	
...	0	1	0	0	0	1	taille 16, alloué
...	1	0	0	0	0	0	taille 32, libre
...	0	1	1	0	0	1	taille 24, alloué
...	0	1	1	0	0	0	taille 24, libre

Mémoire virtuelle

Gestion mémoire côté langages

Introduction

Allocateur mémoire de type `malloc`

Destructeurs

Garbage collection : concepts

Comptage de références

GC traçant : mark & sweep

GC traçant : stop & copy

GC à générations

`malloc(`*n*`)`

On parcourt la liste des blocs
à la recherche d'un bloc libre suffisamment grand :

- si on en trouve un, alors
  - on le découpe éventuellement en deux blocs (un alloué + un libre),
  - on renvoie le bloc alloué ;
- sinon,
  - on alloue un nouveau bloc à la fin de la liste, avec `sbrk`,
  - on le renvoie.

Mémoire virtuelle
Gestion mémoire côté langages
Introduction
Allocateur mémoire de type `malloc`
Destructeurs
Garbage collection : concepts
Comptage de références
GC traçant : mark & sweep
GC traçant : stop & copy
GC à générations

Stratégie

Pour trouver un bloc libre, plusieurs stratégies sont possibles :

- on prend le premier bloc assez grand (*first fit*) ;
- même chose, mais en commençant là où s'était arrêtée la recherche précédente (*next fit*) ;
- on choisit un bloc assez grand de taille minimale (*best fit*).

Mémoire virtuelle

Gestion mémoire côté langages

Introduction

Allocateur mémoire de type `malloc`

Destructeurs

Garbage collection : concepts

Comptage de références

GC traçant : mark & sweep

GC traçant : stop & copy

GC à générations

`free(p)`

Il suffit de changer le statut du bloc p (de alloué à libre).

Mémoire virtuelle

Gestion mémoire côté langages

Introduction

Allocateur mémoire de type `malloc`

Destructeurs

Garbage collection : concepts

Comptage de références

GC traçant : mark & sweep

GC traçant : stop & copy

GC à générations

Problème

La mémoire se fragmente : de plus en plus de petits blocs

⇒ De la mémoire est gâchée.

⇒ La recherche devient coûteuse.

Il faut compacter.

Rappel : dans un allocateur type `malloc`, on ne peut pas déplacer les blocs une fois qu'ils ont été alloués ! (mais c'est une stratégie possible dans d'autres types d'allocateurs)

Deuxième idée

On raffine avec l'idée suivante : lorsqu'un bloc est libéré, on détermine s'il peut être fusionné avec un bloc libre adjacent (*coalescing*).

Il est facile de déterminer si le bloc suivant est libre
et de les fusionner le cas échéant (on additionne les tailles).

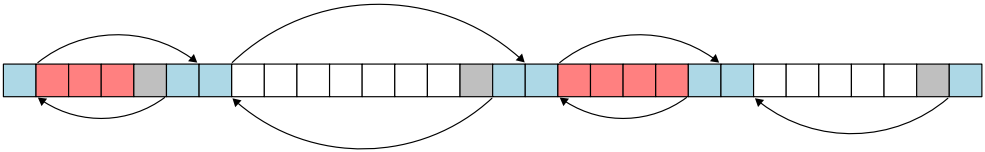
En revanche, on ne sait pas fusionner avec le bloc précédent facilement.

Mémoire virtuelle
Gestion mémoire côté langages
Introduction
Allocateur mémoire de type `malloc`
Destructeurs
Garbage collection : concepts
Comptage de références
GC traçant : mark & sweep
GC traçant : stop & copy
GC à générations

Troisième idée

Pour y parvenir, on duplique l'entête à la fin de chaque bloc (idée due à Knuth, appelée *boundary tags*).

Les blocs sont doublement chaînés :



Fusion

Quand on libère un bloc p, on examine le précédent et le suivant.

Il y a 4 situations possibles :

- alloué | p | alloué : on ne fait rien
- alloué | p | libre : fusion avec le suivant
- libre | p | alloué : fusion avec le précédent
- libre | p | libre : fusion des trois blocs

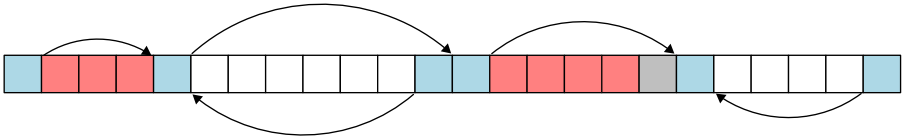
On maintient l'invariant qu'il n'y a jamais deux blocs libres adjacents.

Astuce

Dupliquer l'entête prend de la place.

Mais on peut :

- ne le faire que dans les blocs libres,
- utiliser un bit dans l'entête pour indiquer si le bloc précédent est libre.



Quatrième idée

Cela reste coûteux de parcourir tous les blocs pour allouer.

D'où l'idée de chaîner entre eux les blocs libres uniquement (*free list*).

Pour cela, on utilise le contenu du bloc, qui est libre, pour stocker deux pointeurs (impose une taille de bloc minimale).

Mémoire virtuelle

Gestion mémoire côté langages

Introduction

Allocateur mémoire de type `malloc`

Destructeurs

Garbage collection : concepts

Comptage de références

GC traçant : mark & sweep

GC traçant : stop & copy

GC à générations

Libération

Quand on libère un bloc, on a maintenant plusieurs possibilités pour le réinsérer dans la liste :

- insertion au début,
- liste triée par adresses croissantes,
- liste triée par taille de bloc,
- etc.

Mémoire virtuelle

Gestion mémoire côté langages

Introduction

Allocateur mémoire de type `malloc`

Destructeurs

Garbage collection : concepts

Comptage de références

GC traçant : mark & sweep

GC traçant : stop & copy

GC à générations

Cinquième idée

Parcourir toute la *free list* peut rester coûteux si de nombreux blocs sont trop petits.

D'où l'idée d'avoir plusieurs listes de blocs libres, organisées par taille.

Exemple : une liste de blocs libres de taille comprise entre 2^n et $2^{n+1} - 1$, pour chaque n.

Conclusion

Comme le voit, `malloc`/`free` sont plus subtiles qu'il n'y paraît
(le `malloc.c` de Linux fait plus de 5 000 lignes).

Beaucoup de paramètres, beaucoup de stratégies possibles.

Une énorme littérature sur la question, avec beaucoup d'évaluations empiriques.

(Voir par exemple Wilson, Johnstone, Neely, Boles.
Dynamic Storage Allocation : A Survey and Critical Review, 1995.)

Mémoire virtuelle
Gestion mémoire côté langages
Introduction
Allocateur mémoire de type `malloc`
Destructeurs
Garbage collection : concepts
Comptage de références
GC traçant : mark & sweep
GC traçant : stop & copy
GC à générations

Conclusion (2)

Un allocateur mémoire est une brique essentielle de tout système de gestion mémoire.

En C, c'est le seul mécanisme. Expose le programmeur à plusieurs erreurs :

- `malloc` avec la mauvaise taille ⇒ accès en dehors des bornes
- `free` oublié ⇒ fuite mémoire
- `free` appelé trop tôt ⇒ accès à de la mémoire non initialisée

Les langages plus haut niveau (Rust, OCaml, ...) fournissent des mécanismes supplémentaires qui complémentent l'allocateur, notamment pour automatiser la libération de mémoire. On va en parler maintenant.

Mémoire virtuelle
Gestion mémoire côté langages
Introduction
Allocateur mémoire de type `malloc`
Destructeurs
Garbage collection : concepts
Comptage de références
GC traçant : mark & sweep
GC traçant : stop & copy
GC à générations

Destructeurs

Mémoire virtuelle

Gestion mémoire côté langages

Introduction

Allocateur mémoire de type `malloc`

Destructeurs

Garbage collection : concepts

Comptage de références

GC traçant : mark & sweep

GC traçant : stop & copy

GC à générations

Exemple en Rust

```
{
  let v = vec![1, 2, 3];
  let b = Box::new(0);
  ...
}
```

- `vec![]` et `Box::new` **allouent** de la mémoire (ils appellent `malloc()`)
- ces allocations sont-elles **libérées** par le programme ci-dessus ? Pourquoi ?

Destructeurs

En C++, Rust, un type peut déclarer un destructeur pour les valeurs de ce type.

Le compilateur **appelle automatiquement** le destructeur quand la portée lexicale (*lexical scope*) d'une valeur se termine.

Le destructeur **libère la mémoire** (et autres ressources) pour cette valeur.

```
{
  let v = vec![1, 2, 3];
  let b = Box::new(0);
  ...
  b.drop(); // inséré par le compilateur Rust,
  v.drop(); // libère la mémoire
}
```


Mémoire virtuelle
Gestion mémoire côté langages
Introduction
Allocateur mémoire de type malloc
Destructeurs
Garbage collection : concepts
Comptage de références
GC traçant : mark & sweep
GC traçant : stop & copy
GC à générations

Avantages et Limitations

Avantages :

- la mémoire est libérée automatiquement à un point prévisible du programme
- fonctionne bien en combinaison avec les *emprunts* de Rust

Limitations :

- la durée de vie des valeurs doit pouvoir être inférée statiquement par le compilateur
- $\Rightarrow$ structures de données avec partage (ou pointeurs formant un graphe) impossibles

En pratique, il faut donc complémenter cette stratégie avec un mécanisme de libération de la mémoire plus **dynamique**...

Garbage collection : concepts

Mémoire virtuelle

Gestion mémoire côté langages

Introduction

Allocateur mémoire de type `malloc`

Destructeurs

Garbage collection : concepts

Comptage de références

GC traçant : mark & sweep

GC traçant : stop & copy

GC à générations

GC

De nombreux langages (Lisp, OCaml, Python, Java, etc. mais aussi Rust) reposent sur un mécanisme automatique et dynamique de libération des blocs mémoire, appelé GC pour *Garbage Collector*.

En français, GC peut être traduit par « ramasse-miettes » ou encore « glâneur de cellules ».

- le plus souvent, le GC est *intégré au langage* (les programmes du langage l'utilisent forcément : OCaml, Java, etc). C'est le cas pour les GC "traçants".
- mais certains mécanismes de GC (comptage de références) peuvent être fournis comme une *bibliothèque* dans un langage sans GC (Rust, C++)

Mémoire virtuelle
Gestion mémoire côté langages
Introduction
Allocateur mémoire de type `malloc`
Destructeurs
Garbage collection : concepts
Comptage de références
GC traçant : mark & sweep
GC traçant : stop & copy
GC à générations

GC

Principe : l'espace alloué sur le tas à une donnée (fermeture, objet, enregistrement, tableau, constructeur, etc.) qui n'est plus atteignable à partir des variables du programme peut être récupéré afin d'être réutilisé pour d'autres données.

Difficulté : on ne peut généralement pas déterminer statiquement (à la compilation) le moment où une donnée n'est plus atteignable.
⇒ Le GC fait donc partie de l'exécutable.

- Soit comme une partie de l'interprète pour un langage interprété.
- Soit comme une bibliothèque liée avec le code compilé pour un langage compilé (*runtime*).

Mémoire virtuelle
Gestion mémoire côté langages
Introduction
Allocateur mémoire de type `malloc`
Destructeurs
Garbage collection : concepts
Comptage de références
GC traçant : mark & sweep
GC traçant : stop & copy
GC à générations

Vocabulaire

Dans la suite, on appelle bloc toute portion élémentaire du tas allouée par le programme.

Un bloc peut contenir un ou plusieurs pointeurs vers d'autres blocs mais aussi des données autres (caractères, entiers, pointeurs en dehors du tas, etc.).

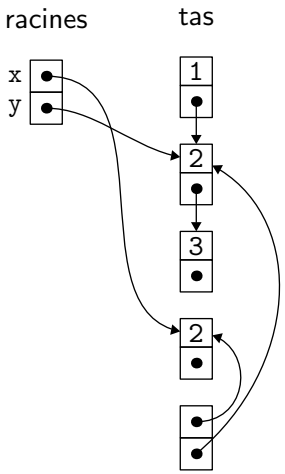
Étant donné un instant de l'exécution du programme, on appelle racine toute variable active à ce moment-là (variable globale ou variable locale contenue dans un tableau d'activation ou dans un registre).

On dit qu'un bloc est vivant s'il est accessible à partir d'une racine *i.e.*, s'il existe un chemin de pointeurs menant d'une racine à ce bloc.

Mémoire virtuelle
Gestion mémoire côté langages
Introduction
Allocateur mémoire de type `malloc`
Destructeurs
Garbage collection : concepts
Comptage de références
GC traçant : mark & sweep
GC traçant : stop & copy
GC à générations

Exemple

```
let x, y =
  let l = [1; 2; 3] in
  (List.filter even l, List.tl l)
...
```



Mémoire virtuelle
Gestion mémoire côté langages
Introduction
Allocateur mémoire de type malloc
Destructeurs
Garbage collection : concepts
Comptage de références
GC traçant : mark & sweep
GC traçant : stop & copy
GC à générations

Comptage de références

Mémoire virtuelle

Gestion mémoire côté langages

Introduction

Allocateur mémoire de type `malloc`

Destructeurs

Garbage collection : concepts

Comptage de références

GC traçant : mark & sweep

GC traçant : stop & copy

GC à générations

Comptage de références

Le comptage de références (*reference counting*) est une stratégie automatique de libération de la mémoire. C'est une forme de GC.

En Rust, disponible via la bibliothèque `Rc` (et `Arc` dans le cas concurrent).

L'idée est d'associer à chaque bloc le nombre de pointeurs qui pointent sur ce bloc (depuis des racines ou depuis d'autres blocs).

Lorsque l'on crée un nouvel alias vers un bloc, il faut incrémenter son compteur.

Lorsque l'on supprime un nouvel alias vers un bloc, il faut décrémenter son compteur.

Lorsque un compteur tombe à 0, alors on libère le bloc.

Quand on libère un bloc, on décrémente les compteurs de tous les blocs vers lesquels il pointe.

Mémoire virtuelle
Gestion mémoire côté langages
Introduction
Allocateur mémoire de type `malloc`
Destructeurs
Garbage collection : concepts
Comptage de références
GC traçant : mark & sweep
GC traçant : stop & copy
GC à générations

En Rust

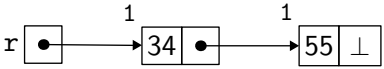
```
impl Rc<T> {
  // crée un bloc avec compteur de références (initialement à 1)
  // renvoie un pointeur vers ce bloc
  fn new(value: T) -> Rc<T>

  // crée un nouvel pointeur vers le bloc : incrémente le compteur
  fn clone(&self) -> Rc<T>

  // destructeur pour le pointeur : décrémente le compteur
  fn drop(&mut self)

  // obtient un pointeur sur la valeur à l'intérieur du bloc
  fn deref(&self) -> &T
}
```


Mémoire virtuelle
Gestion mémoire côté langages
Introduction
Allocateur mémoire de type malloc
Destructeurs
Garbage collection : concepts
Comptage de références
GC traçant : mark & sweep
GC traçant : stop & copy
GC à générations

Décompte des références

Limitations :

- la mise à jour des compteurs peut devenir coûteuse ;
- une désallocation peut entrainer une chaine de désallocations (latence difficile à prévoir)
- les cycles dans les structures de données rendent les blocs correspondant irrécupérables.



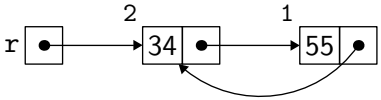
Le comptage des références est peu utilisé de manière systématique par un langage (exceptions : Perl, Python), il est plutôt exposé comme bibliothèque (C++, Rust).

Mémoire virtuelle
Gestion mémoire côté langages
Introduction
Allocateur mémoire de type malloc
Destructeurs
Garbage collection : concepts
Comptage de références
GC traçant : mark & sweep
GC traçant : stop & copy
GC à générations

Décompte des références

Limitations :

- la mise à jour des compteurs peut devenir coûteuse ;
- une désallocation peut entrainer une chaine de désallocations (latence difficile à prévoir)
- les cycles dans les structures de données rendent les blocs correspondant irrécupérables.



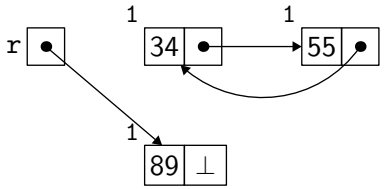
Le comptage des références est peu utilisé de manière systématique par un langage (exceptions : Perl, Python), il est plutôt exposé comme bibliothèque (C++, Rust).

Mémoire virtuelle
Gestion mémoire côté langages
Introduction
Allocateur mémoire de type malloc
Destructeurs
Garbage collection : concepts
Comptage de références
GC traçant : mark & sweep
GC traçant : stop & copy
GC à générations

Décompte des références

Limitations :

- la mise à jour des compteurs peut devenir coûteuse ;
- une désallocation peut entrainer une chaine de désallocations (latence difficile à prévoir)
- les cycles dans les structures de données rendent les blocs correspondant irrécupérables.



Le comptage des références est peu utilisé de manière systématique par un langage (exceptions : Perl, Python), il est plutôt exposé comme bibliothèque (C++, Rust).

Mémoire virtuelle
Gestion mémoire côté langages
Introduction
Allocateur mémoire de type malloc
Destructeurs
Garbage collection : concepts
Comptage de références
GC traçant : mark & sweep
GC traçant : stop & copy
GC à générations

GC “traçant”, mark & sweep

Marquer et balayer

On considère une autre solution, appelée marquer et balayer (*mark and sweep*).

Elle procède en deux temps :

1. on marque tous les blocs atteignables à partir des racines (en utilisant un parcours en profondeur et un bit dans chaque bloc) ;
2. on examine tous les blocs et
   - on récupère ceux qui ne sont pas marqués (ils sont remis dans la *free list*),
   - on supprime les marques sur les autres.

Quand on veut allouer un bloc, on examine la *free list* ; si elle est vide, c'est un bon moment pour effectuer un GC.

Mémoire virtuelle
Gestion mémoire côté langages
Introduction
Allocateur mémoire de type `malloc`
Destructeurs
Garbage collection : concepts
Comptage de références
GC traçant : mark & sweep
GC traçant : stop & copy
GC à générations

Marquer

Le marquage utilise un parcours en profondeur, comme ceci :

parcours(x) =
  si x est un pointeur sur le tas non encore marqué
    marquer x
    pour chaque champ f de x
      parcours($x.f$)

pour chaque racine r
  parcours(r)

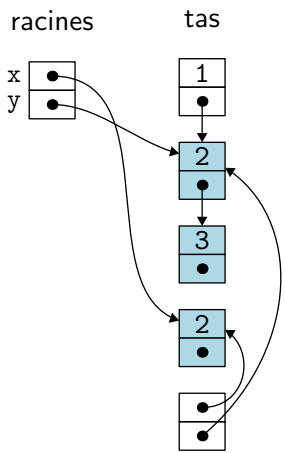
Balayer

Le balayage récupère les blocs non marqués :

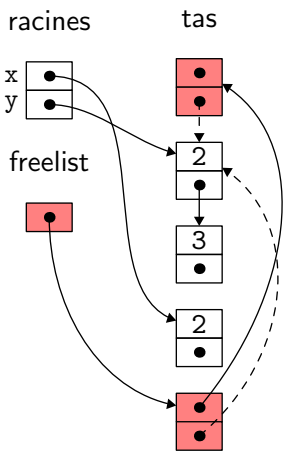
pour chaque bloc x
  si x est marqué
    effacer la marque de x
  sinon
    ajouter x à la *free list*

Exemple

marquer



balayer



Problème

Le marquage est un algorithme récursif, qui va donc utiliser une taille de pile proportionnelle à la profondeur du tas ; celle-ci peut être aussi grande que le tas lui-même.

On pourrait utiliser une pile explicite ou la structure traversée elle-même pour encoder la pile (*pointer reversal*).

Mémoire virtuelle
Gestion mémoire côté langages
Introduction
Allocateur mémoire de type `malloc`
Destructeurs
Garbage collection : concepts
Comptage de références
GC traçant : mark & sweep
GC traçant : stop & copy
GC à générations

Autre problème

Mais surtout, il n'est pas souhaitable que le programme soit interrompu trop longtemps pendant un marquage/balayage complet (ce serait gênant en pratique).

Pour y remédier, on marque les blocs petit à petit, au fur et à mesure des appels au GC.

On parle de GC incrémental.

Mémoire virtuelle

Gestion mémoire côté langages

Introduction

Allocateur mémoire de type `malloc`

Destructeurs

Garbage collection : concepts

Comptage de références

GC traçant : mark & sweep

GC traçant : stop & copy

GC à générations

Mark and sweep incrémental

Une simple marque ne suffit plus, il faut au moins trois couleurs.

On a des blocs :

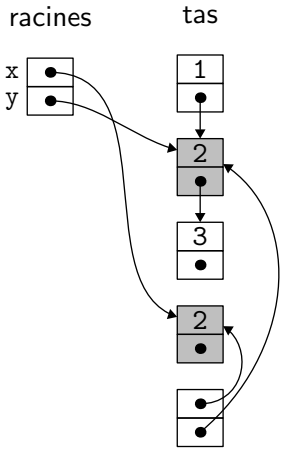
- blancs, candidats à être récupérés,
- noirs, atteignables depuis les racines et sans pointeurs vers des blocs blancs,
- gris, atteignables depuis les racines mais non encore examinés.

Initialement, les racines sont grises et tous les autres blocs sont blancs.

Mémoire virtuelle
Gestion mémoire côté langages
Introduction
Allocateur mémoire de type `malloc`
Destructeurs
Garbage collection : concepts
Comptage de références
GC traçant : mark & sweep
GC traçant : stop & copy
GC à générations

Mark and sweep incrémental

tant qu'il y a des blocs gris
  choisir un bloc gris x
  le colorier en noir
  pour chaque champ f de x
    si $x.f$ pointe vers un bloc blanc
      colorier ce bloc en gris



Intérêt : on peut faire un nombre quelconque de tours de cette boucle.

Mémoire virtuelle

Gestion mémoire côté langages

Introduction

Allocateur mémoire de type `malloc`

Destructeurs

Garbage collection : concepts

Comptage de références

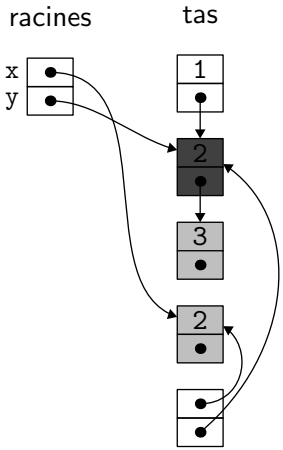
GC traçant : mark & sweep

GC traçant : stop & copy

GC à générations

Mark and sweep incrémental

tant qu'il y a des blocs gris
  choisir un bloc gris x
  le colorier en noir
  pour chaque champ f de x
    si $x.f$ pointe vers un bloc blanc
      colorier ce bloc en gris



Intérêt : on peut faire un nombre quelconque de tours de cette boucle.

Mémoire virtuelle

Gestion mémoire côté langages

Introduction

Allocateur mémoire de type `malloc`

Destructeurs

Garbage collection : concepts

Comptage de références

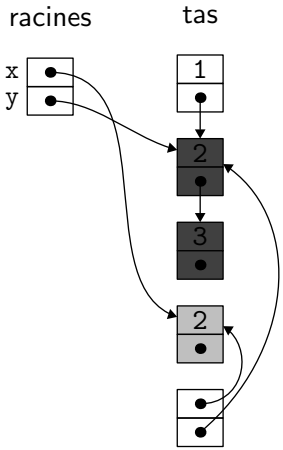
GC traçant : mark & sweep

GC traçant : stop & copy

GC à générations

Mark and sweep incrémental

tant qu'il y a des blocs gris
  choisir un bloc gris x
  le colorier en noir
  pour chaque champ f de x
    si $x.f$ pointe vers un bloc blanc
      colorier ce bloc en gris



Intérêt : on peut faire un nombre quelconque de tours de cette boucle.

Mémoire virtuelle

Gestion mémoire côté langages

Introduction

Allocateur mémoire de type `malloc`

Destructeurs

Garbage collection : concepts

Comptage de références

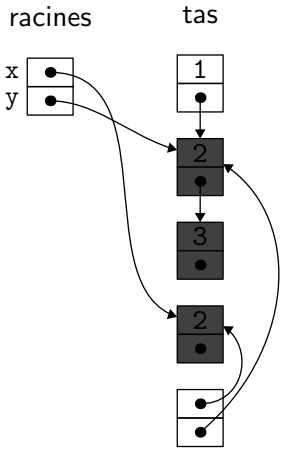
GC traçant : mark & sweep

GC traçant : stop & copy

GC à générations

Mark and sweep incrémental

tant qu'il y a des blocs gris
  choisir un bloc gris x
  le colorier en noir
  pour chaque champ f de x
    si $x.f$ pointe vers un bloc blanc
      colorier ce bloc en gris



Intérêt : on peut faire un nombre quelconque de tours de cette boucle.

Mémoire virtuelle

Gestion mémoire côté langages

Introduction

Allocateur mémoire de type `malloc`

Destructeurs

Garbage collection : concepts

Comptage de références

GC traçant : mark & sweep

GC traçant : stop & copy

GC à générations

Mark and sweep incrémental

Une fois qu'il n'y a plus de blocs gris :

- les blocs noirs sont tous atteignables depuis les racines ;
- les blocs blancs, en revanche, ne le sont pas
  car un bloc noir ne pointe jamais vers un bloc blanc.

Du coup,

1. on récupère les blocs blancs,
2. on blanchit les blocs noirs,
3. on remet toutes les racines en gris.

Mémoire virtuelle
Gestion mémoire côté langages
Introduction
Allocateur mémoire de type `malloc`
Destructeurs
Garbage collection : concepts
Comptage de références
GC traçant : mark & sweep
GC traçant : stop & copy
GC à générations

Marquer et balayer

C'est une bonne solution pour déterminer les blocs à récupérer (en particulier, on récupère bien les cycles inatteignables).

Pas encore une vraie solution au problème de la fragmentation.

Mémoire virtuelle

Gestion mémoire côté langages

Introduction

Allocateur mémoire de type `malloc`

Destructeurs

Garbage collection : concepts

Comptage de références

GC traçant : mark & sweep

GC traçant : stop & copy

GC à générations

GC “traçant”, stop & copy

GC par copie

Considérons encore une autre solution, appelée s'arrêter et copier (*stop and copy*).

L'idée est de découper le tas en deux moitiés :

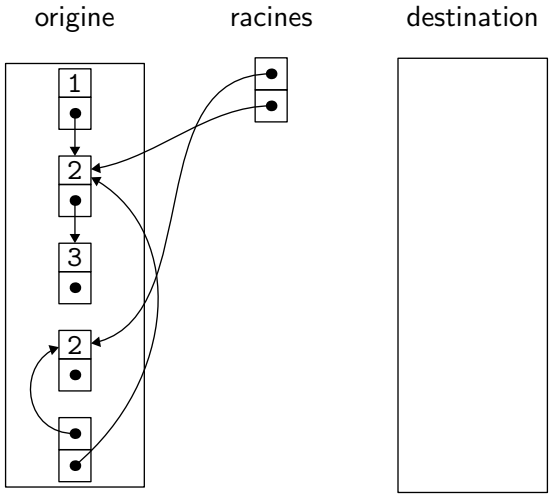
1. on n'en utilise qu'une seule, dans laquelle on alloue linéairement,
2. lorsqu'elle est pleine, on copie tout ce qui est atteignable dans l'autre moitié, et on échange le rôle des deux moitiés.

Bénéfices immédiats :

- l'allocation est très peu coûteuse (une addition et une comparaison) ;
- plus de problème de fragmentation.

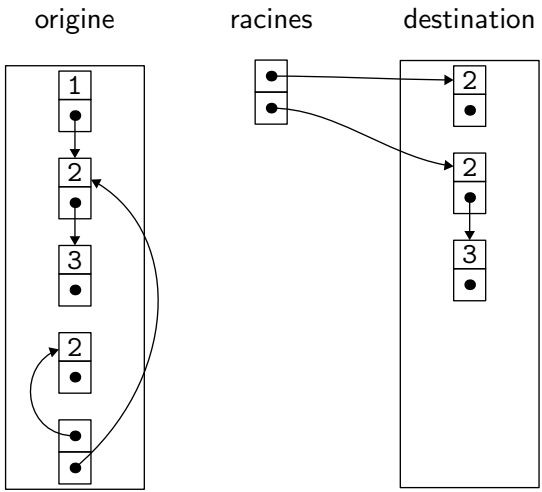
Mémoire virtuelle
Gestion mémoire côté langages
Introduction
Allocateur mémoire de type `malloc`
Destructeurs
Garbage collection : concepts
Comptage de références
GC traçant : mark & sweep
GC traçant : stop & copy
GC à générations

Exemple



Mémoire virtuelle
Gestion mémoire côté langages
Introduction
Allocateur mémoire de type malloc
Destructeurs
Garbage collection : concepts
Comptage de références
GC traçant : mark & sweep
GC traçant : stop & copy
GC à générations

Exemple



Algorithme de Cheney (1970)

Effectue la copie en utilisant un espace supplémentaire constant.

Principe : un parcours en largeur qui utilise :

- l'espace d'arrivée comme zone de stockage des pointeurs restant à mettre à jour,
- et l'espace de départ comme zone de stockage des pointeurs déjà mis à jour : lorsqu'un bloc a été déplacé de la première zone (`origine`) vers la seconde (`destination`) alors on utilise son premier champ pour indiquer où il a été copié.

Algorithme de Cheney

On commence par écrire une fonction qui copie le bloc à l'adresse p, si cela n'a pas encore été fait.

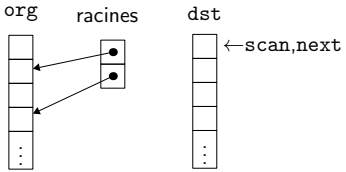
`next` désigne le premier emplacement libre dans `destination`.

déplace(p) =
  si p pointe dans `origine`
    si $p.f_1$ pointe dans `destination`
      renvoyer $p.f_1$
    sinon
      pour chaque champ f_i de p
        `next`.$f_i \leftarrow p.f_i$
      $p.f_1 \leftarrow$ `next`
      `next` $\leftarrow$ `next` + taille du bloc p
      renvoyer $p.f_1$
  sinon
    renvoyer p

Algorithme de Cheney

On peut alors réaliser la copie, en commençant par les racines.

`scan` $\leftarrow$ `next` $\leftarrow$ début de `destination`
pour chaque racine r
  $r \leftarrow$ déplace(r)
tant que `scan` $<$ `next`
  pour chaque champ f_i de `scan`
    `scan`.$f_i \leftarrow$ déplace(`scan`.f_i)
  `scan` $\leftarrow$ `scan` + taille du bloc `scan`



La zone de `destination` située entre `scan` et `next` représente les blocs dont les champs n'ont pas encore été mis à jour.

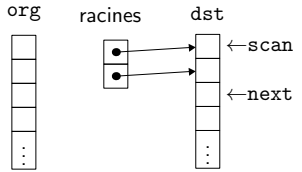
Noter que `scan` avance, mais que `next` aussi !

Mémoire virtuelle
Gestion mémoire côté langages
Introduction
Allocateur mémoire de type `malloc`
Destructeurs
Garbage collection : concepts
Comptage de références
GC traçant : mark & sweep
GC traçant : stop & copy
GC à générations

Algorithme de Cheney

On peut alors réaliser la copie, en commençant par les racines.

`scan` $\leftarrow$ `next` $\leftarrow$ début de `destination`
pour chaque racine r
  $r \leftarrow$ déplace(r)
tant que `scan` $<$ `next`
  pour chaque champ f_i de `scan`
    `scan`.$f_i \leftarrow$ déplace(`scan`.f_i)
  `scan` $\leftarrow$ `scan` + taille du bloc `scan`



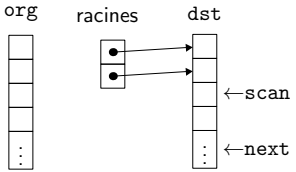
La zone de `destination` située entre `scan` et `next` représente les blocs dont les champs n'ont pas encore été mis à jour.

Noter que `scan` avance, mais que `next` aussi !

Algorithme de Cheney

On peut alors réaliser la copie, en commençant par les racines.

`scan` ← `next` ← début de `destination`
pour chaque racine r
  r ← déplace(r)
tant que `scan` < `next`
  pour chaque champ f_i de `scan`
    `scan`.f_i ← déplace(`scan`.f_i)
  `scan` ← `scan` + taille du bloc `scan`



La zone de `destination` située entre `scan` et `next` représente les blocs dont les champs n'ont pas encore été mis à jour.

Noter que `scan` avance, mais que `next` aussi !

Algorithme de Cheney

Bien que très élégant, cet algorithme a au moins un défaut : il modifie la localité des données *i.e.*, des blocs qui étaient proches avant la copie ne le sont plus nécessairement après.

En présence de caches mémoire, la localité est importante.

Il est possible de modifier l'algorithme de Cheney pour effectuer un mélange de parcours en largeur et de parcours en profondeur.

Mémoire virtuelle
Gestion mémoire côté langages
Introduction
Allocateur mémoire de type `malloc`
Destructeurs
Garbage collection : concepts
Comptage de références
GC traçant : mark & sweep
GC traçant : stop & copy
GC à générations

GC à générations

Dans de nombreux programmes, la plupart des valeurs ont une durée de vie courte, et celles qui survivent à plusieurs collections sont susceptibles de survivre à beaucoup d'autres collections.

D'où l'idée d'organiser le tas en plusieurs générations :

- G_0 contient les valeurs les plus récentes, et on y fait des collections fréquemment,
- G_1 contient des valeurs toutes plus anciennes que celles de G_0, et on y fait des collections moins fréquemment,
- etc.

En pratique, il y a quelques difficultés pour identifier les racines de chaque génération, en particulier parce qu'une affectation peut introduire un pointeur de G_1 vers G_0 par exemple.

Mémoire virtuelle
Gestion mémoire côté langages
Introduction
Allocateur mémoire de type malloc
Destructeurs
Garbage collection : concepts
Comptage de références
GC traçant : mark & sweep
GC traçant : stop & copy
GC à générations

Le GC d'OCaml

Le GC d'OCaml est un GC à deux générations :

- un GC mineur (valeurs jeunes) : *Stop & Copy*,
- un GC majeur (valeurs vieilles) : *Mark & Sweep* incrémental.

La zone `destination` du GC mineur est la zone du GC majeur.

Mémoire virtuelle

Gestion mémoire côté langages

Introduction

Allocateur mémoire de type `malloc`

Destructeurs

Garbage collection : concepts

Comptage de références

GC traçant : mark & sweep

GC traçant : stop & copy

GC à générations

Représentation des données

Maintenant qu'on connaît les besoins du GC, on peut expliquer la représentation des données choisie par OCaml.

Mémoire virtuelle

Gestion mémoire côté langages

Introduction

Allocateur mémoire de type `malloc`

Destructeurs

Garbage collection : concepts

Comptage de références

GC traçant : mark & sweep

GC traçant : stop & copy

GC à générations

Valeur

Une valeur OCaml est

- soit un entier, représentant alors une valeur de type `int` ou un constructeur constant (`true`, `false`, `[]`, etc.),
- soit un pointeur, qui peut pointer dans ou en dehors du tas.

Rappel : le mode de passage d'OCaml est par valeur.

Mémoire virtuelle

Gestion mémoire côté langages

Introduction

Allocateur mémoire de type `malloc`

Destructeurs

Garbage collection : concepts

Comptage de références

GC traçant : mark & sweep

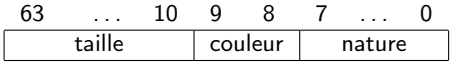
GC traçant : stop & copy

GC à générations

Bloc

Un pointeur sur le tas OCaml pointe vers un bloc de $n + 1$ mots
(un mot = 8 octets sur une architecture 64 bits).

Le premier de ces mots est un entête qui contient la taille n du bloc, sa nature et deux bits utilisés par le GC.



(Attention : c'est un autre entête que celui de `malloc`.)

Taille d'un bloc

La taille du bloc étant codée sur 54 bits, on a :

```
# Sys.max_array_length;;
- : int = 18014398509481983
```

Les chaînes de caractères sont en revanche représentées de manière compacte (8 caractères par mot), ce qui donne :

```
# Sys.max_string_length;;
- : int = 144115188075855863
```

Nature du bloc

La nature du bloc est un entier codé sur 8 bits (0..255) ; elle permet de distinguer :

- flottant,
- chaîne de caractères,
- tableau de flottants,
- objet,
- fermeture,
- le cas général d'un bloc structuré : enregistrement, tableau, *n*-uplet, constructeur ; dans ce dernier cas, l'entier indique de quel constructeur il s'agit (pour le filtrage).

Mémoire virtuelle

Gestion mémoire côté langages

Introduction

Allocateur mémoire de type `malloc`

Destructeurs

Garbage collection : concepts

Comptage de références

GC traçant : mark & sweep

GC traçant : stop & copy

GC à générations

Entiers et pointeurs

Lorsque le GC parcourt un bloc (pour le marquage ou la copie), il doit distinguer entiers et pointeurs.

Difficulté : le compilateur ne peut pas indiquer au GC quels sont les champs qui sont des pointeurs en présence de polymorphisme.

```
let f x = (x, x)

f 42    (* un bloc contenant deux entiers *)

f [42]  (* un bloc contenant deux pointeurs *)
```

Solution

Une valeur OCaml est :

- soit un pointeur, nécessairement pair pour des raisons d'alignement,
- soit un entier impair $2n + 1$, représentant alors la valeur n.

Le GC teste le bit de poids faible pour déterminer si un champ est un pointeur ou non.

Conséquence : les entiers d'OCaml sont des entiers 63 bits signés
(mais la bibliothèque fournit un module `Int64`).

Mémoire virtuelle

Gestion mémoire côté langages

Introduction

Allocateur mémoire de type `malloc`

Destructeurs

Garbage collection : concepts

Comptage de références

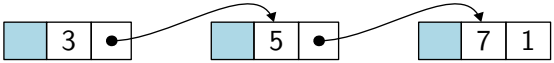
GC traçant : mark & sweep

GC traçant : stop & copy

GC à générations

Exemple

La valeur `3 :: 5 :: 7 :: []` est ainsi représentée :



Mémoire virtuelle

Gestion mémoire côté langages

Introduction

Allocateur mémoire de type `malloc`

Destructeurs

Garbage collection : concepts

Comptage de références

GC traçant : mark & sweep

GC traçant : stop & copy

GC à générations

Arithmétique

L'entier n étant représenté par $2n + 1$, l'arithmétique devient un peu plus compliquée.

D'où un usage intensif de l'instruction `lea` par le compilateur OCaml.

Ainsi, la fonction

```
let f x y = x + y
```

est compilée comme :

```
f:
    leaq -1(%rax,%rbx), %rax
    ret
```