École Normale Supérieure

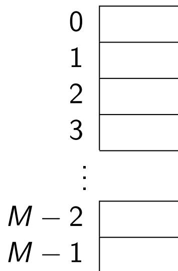
# Langages de programmation et compilation

Jean-Christophe Filliâtre

Cours 11 / 14 décembre 2015

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

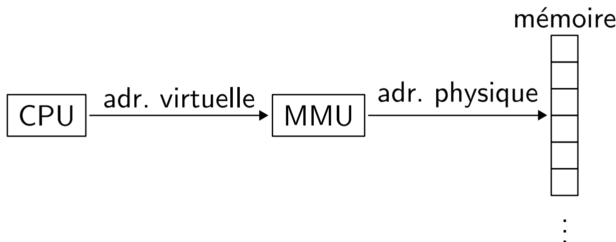
auxquels le CPU peut accéder en lecture et en écriture au moyen d'adresses physiques 0, 1, 2, etc.




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

depuis longtemps, cependant, 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 $0, 1, \ldots, M-1$)

c'est typiquement le **système d'exploitation** qui programme le MMU

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

chaque page est soit

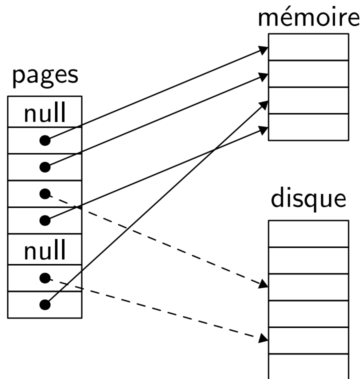
- non allouée
- allouée en mémoire physique (et le MMU renseigné)
- allouée sur le disque

le système d'exploitation maintient une table des pages

# exemple

8 pages

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



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 gestionnaire installe la page en mémoire physique
     (éventuellement en déplaçant vers le disque une autre page)
  3. l'exécution reprend sur la même instruction

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

chaque programme a donc l'illusion de disposer de l'intégralité de la mémoire (virtuelle) pour lui tout seul

cela facilite

- l'édition de liens
  (le code est toujours à la même adresse,
  par ex. `0x400000` sous Linux 64 bits)
- le chargement d'un programme
  (les pages sont déjà sur le disque)
- le partage de pages entre plusieurs processus
  (une même page physique = plusieurs pages virtuelles)
- l'allocation de mémoire
  (les pages physiques n'ont pas besoin d'être contiguës)

pour en savoir plus, notamment sur le mécanisme de traduction des adresses, lire

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

## allocation mémoire

il est facile d'allouer de la mémoire **statiquement**

- soit dans le segment `.data` (initialisée explicitement)
- soit dans le segment `.bss` (initialisée par zéro)

néanmoins...

la plupart des programmes doivent allouer de la mémoire **dynamiquement**

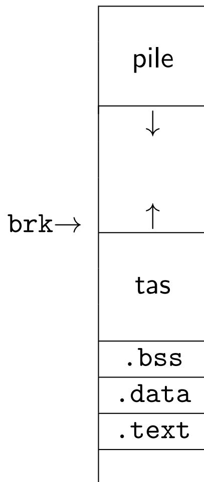
- soit implicitement, par des constructions du langage (objets, clôtures, etc.)
- soit explicitement, pour stocker des données dont la taille n'est pas connue statiquement

il convient généralement de la **libérer**

cette allocation dynamique se fait dans **le tas**

il est situé immédiatement au dessus du segment de données

le système maintient son sommet dans une variable `brk` (*program break*)



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

l'appel système

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

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

mais on ne sait toujours pas comment libérer la mémoire

on va devoir programmer un gestionnaire de mémoire,
permettant d'allouer et de libérer des blocs de mémoire

la libération peut être

- explicitement réalisée par le programmeur
  exemple : la bibliothèque C `malloc`
- automatiquement réalisé par le gestionnaire
  on parle alors de GC

à partir de sbrk, on veut fournir deux opérations

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

et

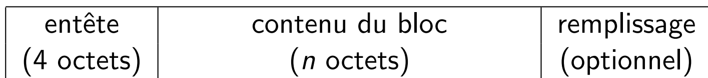
```
void free(void *ptr);
  // libère le bloc situé à l'adresse ptr
  // (doit avoir été alloué par malloc,
  //  sinon comportement non spécifié)
```

- on ne présume rien sur la séquence de `malloc` et de `free` à venir
- la réponse à `malloc` ne peut être différée
- tout structure de données nécessaire à `malloc` et `free` doit être stockée elle-même dans le tas
- tout bloc renvoyé par `malloc` doit être aligné sur 8 octets
- tout bloc alloué ne peut plus être modifié, ni déplacé

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

- 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



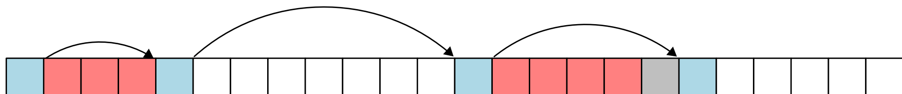
adresse renvoyée par `malloc`
(alignée sur 8 octets)

alloué
12 octets

libre
28 octets

alloué
16 octets

libre
20 octets



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

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 |

# `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

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*)

## `free`($p$)

il suffit de changer le statut du bloc $p$ (de alloué à libre)

# problème

la mémoire se **fragmente** : de plus en plus de petits blocs

$\Rightarrow$ de la mémoire est gâchée

$\Rightarrow$ la recherche devient coûteuse

il faut **compacter**

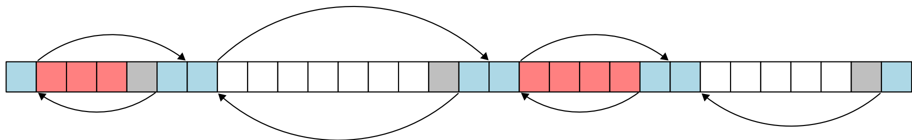
on raffine avec l'idée suivante : lorsqu'un bloc est libéré, on détermine s'il peut être **fusionné** avec un bloc libre adjacent (*coalescing*)

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

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



quand on libère un bloc $p$, on examine le précédent et le suivant

il y a 4 situations possibles

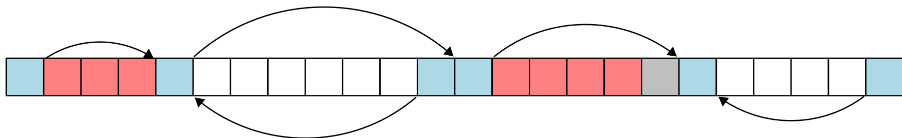
- alloué | $p$ | alloué : on ne fait rien
- alloué | $p$ | libre : fusion avec le suivant
- libre | $p$ | alloué : fusion avec le précédent
- libre | $p$ | libre : fusion des trois blocs

on maintient l'**invariant** qu'il n'y a jamais deux blocs libres adjacents

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)

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.

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 $n$

comme le voit, `malloc`/`free` sont plus subtiles qu'il n'y paraît
(le `malloc.c` de Linux fait plus de 5000 lignes)

beaucoup de paramètres, beaucoup de stratégies possibles

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

on trouve du code C mettant en œuvre ces idées dans

- Brian W. Kernighan et Dennis M. Ritchie
  *Le langage C*
- Randal E. Bryant et David R. O'Hallaron
  *Computer Systems : A Programmer's Perspective*

de nombreux langages (Lisp, OCaml, Haskell, Java, etc.) reposent sur un mécanisme **automatique** de libération des blocs mémoire, appelé **GC** pour *Garbage Collector*

en français, GC peut être traduit par ≪ ramasse-miettes ≫ ou encore ≪ glâneur de cellules ≫

**principe** : l'espace alloué sur le tas à une donnée (fermeture, 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
$\Rightarrow$ le GC fait donc partie de l'exécutable

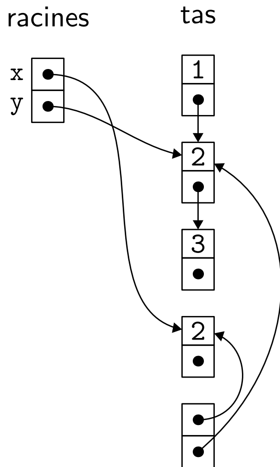
- soit comme une partie de l'interprète pour un langage interprété
- soit comme une bibliothèque liée avec le code compilé pour un langage compilé (*runtime*)

un bloc peut contenir un ou plusieurs pointeurs vers d'autres blocs (ses **fils**) 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

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



on considère une première solution, appelée **comptage des références** (*reference counting*)

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)

quand ce nombre tombe à zéro, on sait qu'on peut libérer le bloc, et on décrémente les compteurs de tous ses fils

la mise à jour du compteur a lieu lors d'une **affectation** (explicite ou implicite comme dans `1::x`) de la forme $b.f \leftarrow p$ ; il faut

- décrémenter le compteur du bloc correspondant à l'ancien pointeur $b.f$ ; s'il tombe à 0, désallouer ce bloc
- incrémenter le compteur du bloc $p$

problèmes :

- la mise à jour des compteurs est très coûteuse
- les **cycles** dans les structures de données rendent les blocs correspondant irrécupérables

le comptage des références est rarement utilisé dans les GC (une exception est le langage Perl) mais parfois explicitement par certains programmeurs

on considère une autre solution, plus efficace, appelée **marquer et balayer** (*mark and sweep*)

elle procède en deux temps

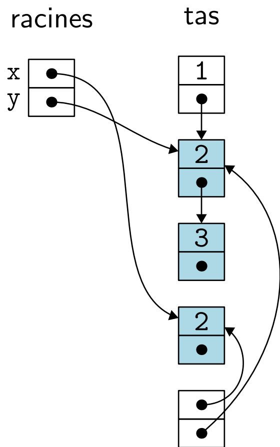
1. on marque tous les blocs atteignables à partir des racines (en utilisant un parcours en profondeur et un bit dans chaque bloc)
2. on examine tous les blocs et
   - on récupère ceux qui ne sont pas marqués (ils sont remis dans la *free list*)
   - on supprime les marques sur les autres

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

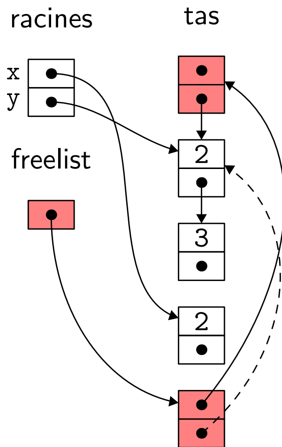
le marquage utilise un parcours en profondeur, comme ceci

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

**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

solution : on utilise la structure traversée elle-même pour encoder la pile d'appels récursifs (*pointer reversal*)

# pointer reversal (Deutsch / Schorr & Waite, 1965)

le parcours est un peu plus compliqué mais n'utilise plus que deux variables et un champ entier `done[x]` dans chaque bloc

```
parcours(x) =
   si x est un pointeur sur le tas non encore marqué
      t <- null; marquer x; done[x] <- 0
      tant que vrai
         i <- done[x]
         si i < nombre de champs de x
            y <- x.fi
            si y est un pointeur sur le tas non encore marqué
               x.fi <- t; t <- x; x <- y
               marquer x; done[x] <- 0
            sinon
               done[x] <- i+1
         sinon
            y <- x; x <- t
            si x = null alors c'est terminé
            i <- done[x]; t <- x.fi; x.fi <- y; done[x] <- i+1
```

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

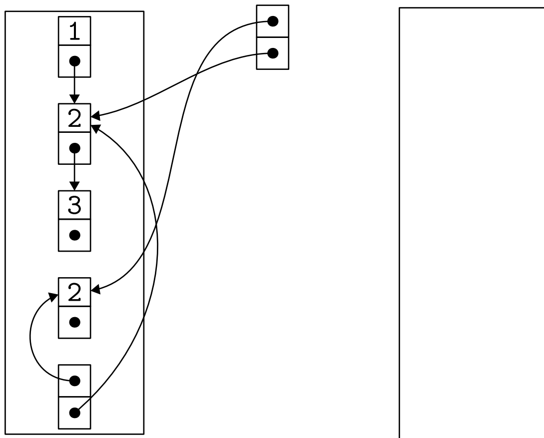
considérons encore une autre solution, appelée **s'arrêter et copier** (*stop and copy*)

l'idée est de découper le tas en deux moitiés

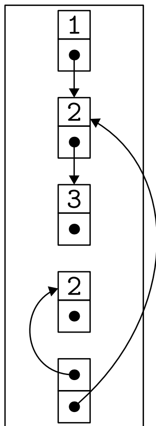
1. on n'en utilise qu'une seule, dans laquelle on alloue linéairement
2. lorsqu'elle est pleine, on copie tout ce qui est atteignable dans l'autre moitié, et on échange le rôle des deux moitiés

bénéfices immédiats :

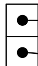
- l'allocation est très peu coûteuse (une addition et une comparaison)
- plus de problème de fragmentation


from-space
racines
to-space
1
2
3
2

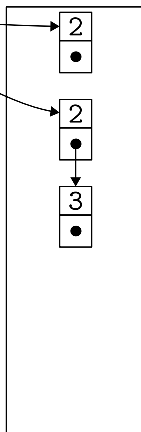
from-space




racines



to-space



problème : il faut effectuer la copie en utilisant un espace constant

solution : on va effectuer un parcours en largeur et utiliser l'espace d'arrivée comme zone de stockage des pointeurs à copier

lorsqu'un bloc a été déplacé de la première zone (`from-space`) vers la seconde (`to-space`) alors on utilise son premier champ pour indiquer où il a été copié

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 `to-space`

```
forward(p) =
   si p pointe dans from-space
      si p.f1 pointe dans to-space
         renvoyer p.f1
      sinon
         pour chaque champ fi de p
            next.fi <- p.fi
         p.f1 <- next
         next <- next + taille du bloc p
         renvoyer p.f1
   sinon
      renvoyer p
```

on peut alors réaliser la copie, en commençant par les racines

```
scan <- next <- début de to-space
pour chaque racine r
   r <- forward(r)
tant que scan < next
   pour chaque champ fi de scan
      scan.fi <- forward(scan.fi)
   scan <- scan + taille du bloc scan
```

la zone de `to-space` 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 !

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

dans un système des 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
(cf. Appel, chapitre 13)

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 (cf. Appel, chapitre 13)

enfin, il n'est pas souhaitable que le programme soit interrompu trop longtemps le temps d'une collection (gênant pour les programmes interactifs, critique pour les programmes temps-réel)

pour y remédier, on utilise un **GC incrémental**, qui marque les blocs petit à petit, au fur et à mesure des appels au GC

une simple marque ne suffit plus, il faut au moins trois couleurs (cf. Appel, chapitre 13)

le GC d'OCaml est un GC à deux générations

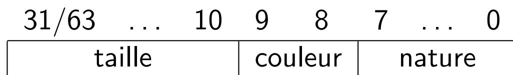
- un GC mineur (valeurs jeunes) : *Stop & Copy*
- un GC majeur (valeurs vieilles) : *Mark & Sweep* incrémental

la zone `to-space` du GC mineur est la zone du GC majeur

maintenant qu'on connaît les besoins du GC, on peut expliquer la **représentation des données** sur le tas (ici dans le cas d'OCaml)

chaque bloc sur le tas est précédé d'un **entête** dont la taille est un **mot** (4 octets sur une architecture 32 bits, 8 sur une architecture 64 bits)

l'entête contient la taille du bloc, sa nature et 2 bits utilisés par le GC




la taille est ici un nombre de mots ; si la taille est $n$, le bloc tout entier, avec son entête, occupe donc $n+1$ mots

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

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)

la taille du bloc étant codée sur 22/54 bits, on a

```
# Sys.max_array_length;;
- : int = 4194303             (* machine 32 bits *)
```

```
# Sys.max_array_length;;
- : int = 18014398509481983 (* machine 64 bits *)
```

les chaînes de caractères sont en revanche représentées de manière compacte (4 caractères par mot), ce qui donne

```
# Sys.max_string_length;;
- : int = 16777211             (* machine 32 bits *)
```

```
# Sys.max_string_length;;
- : int = 144115188075855863 (* machine 64 bits *)
```

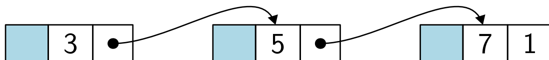
une valeur OCaml tient sur un mot ; c'est

- soit un entier impair $2n + 1$, représentant alors la valeur $n$ de type `int` ou un constructeur constant encodé par $n$ (`true`, `false`, `[]`, etc.)
- soit un pointeur (nécessairement pair pour des raisons d'alignement), qui peut pointer dans ou en dehors du tas

le GC teste donc le bit de poids faible pour déterminer si un champ est un pointeur ou non, car en présence de polymorphisme, le compilateur ne peut pas indiquer au GC quels sont les champs qui sont des pointeurs

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

exemple : la valeur `1 :: 2 :: 3 :: []` est ainsi représentée




conséquence : les entiers d'OCaml sont des entiers 31/63 bits signés
(mais la bibliothèque fournit des modules `Int32` et `Int64`)

enfin, il est important de rappeler que le mode de passage d'OCaml est le passage **par valeur**, même si de nombreuses valeurs se trouvent être des pointeurs

# la suite

- TD cette semaine
  - programmation d'un GC *stop & copy*

- cours lundi 4 janvier
  - production de code efficace, partie 1