

# Systeme d'exploitation

## Allocation - désallocation mémoire

### Licence Informatique

Jean-Louis Lanet / Guillaume Bouffard

Jean-louis.lanet@unilim.fr

# Plan



- Garbage collector
  - Garbage collector
    - Garbage collector
      - Garbage collector
        - » Garbage collector
          - » Garbage collector
            - » Garbage collector



# Gestion de la mémoire

- Lorsque le programme s'exécute il peut avoir besoin de mémoire dynamiquement,
  - Allocation d'objet, création de tableau, extension de tableau etc.
  - La mémoire allouée n'est pas infinie et il faut la gérer proprement,
  - La gestion manuelle est source d'erreur, mauvaise libération fuite de mémoire etc.
  - La gestion manuelle se fait à l'aide d'appel à la librairie (`free,..`),
  - Déléguer au système cette gestion soulage le programmeur et permet une plus grand efficacité.



# Différentes façons de faire

- Tous les langages de programmation moderne autorisent les programmeurs à allouer de l'espace de stockage dynamiquement,
  - Nouveaux enregistrements, des tableaux, des objets, etc.
- De même, ces langages ont besoin d'un mécanisme pour recycler la mémoire abandonnée,
- C'est généralement l'aspect le plus difficile des run time de tout langage moderne (Java, ML, Lisp, Scheme, Modula, ...)



# Plan mémoire

Mémoire virtuelle  
Pour un processus

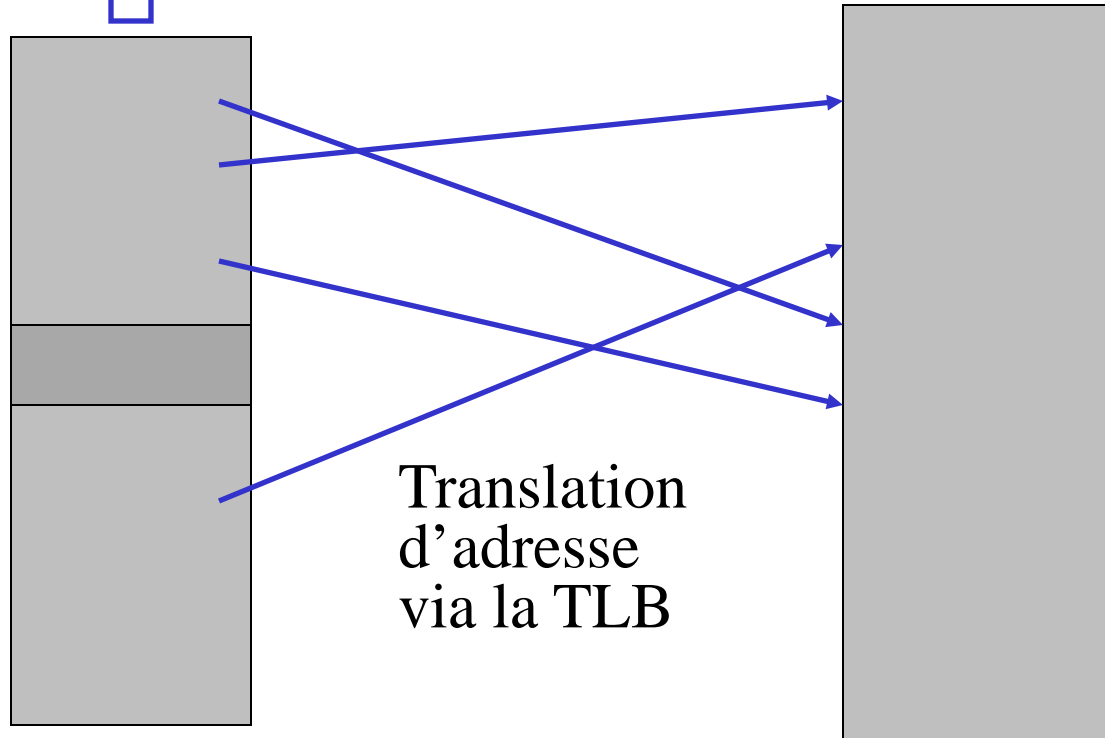
Nouvelles allocation  
de page via des appels  
système

Mémoire physique

Tas

Zone  
statique

Pile



Translation  
d'adresse  
via la TLB

Sens de décalage de la pile

# Garbage = détritius

- C'est quoi un détritius ?
  - une cellule mémoire est considérée comme détritius si elle ne sera plus utilisée par le programme quelque soit le chemin utilisé par le programme.
- Est il simple de décider si une cellule mémoire doit être considérée comme un détritius ?

# Garbage = détritius

- C'est quoi un détritius ?
  - une cellule mémoire est considérée comme détritius si elle ne sera plus utilisée par le programme quelque soit le chemin utilisé par le programme.
- Est il simple de décider si une cellule mémoire doit être considérée comme un détritius ?
- C'est un problème indécidable (cf. théorème de Rice)



# Détritus

- Puisque déterminer si une cellule mémoire est un détritius est indécidable, on utilise des techniques approchées.
  - Soit on laisse le travail au programmeur par allocation et désallocation explicite
  - On évite en les détritius en faisant le ménage grâce à la pile,
  - On compte les références sur un objet,
  - On utilise un GC automatique : Mark-sweep, copying collection, Generational GC





# Type d'allocation

- Allocation statique
  - Le plus simple si tout est connu à la compilation
  - Pas de récursivité, taille fixe des données
  - Fortran 77
- Allocation via la pile
  - À chaque appel de la mémoire est allouée,
  - Taille des données fonction de paramètres d'appel,
  - Pas de rémanence des données,
  - Algol 58
- Allocation du tas
  - Au-delà de la durée de vie d'un appel, rémanence, objet dynamique,
  - Gestion explicite Pascal, C, C++, implicite ... Java, Scheme,...



# Gestion mémoire en pile

- Exemple:

```
void les_20_preiers_preiers() {  
    int premiers[] = new int[20];  
    // variable premiers en pile,  
  
    for (int i=0;i<20;i++)  
        premiers[i]=calculer_premier(i);  
    afficher_tableau_entiers(20,premiers);  
  
} // ici, premiers est dépilé, elle peut  
// être automatiquement recyclée/libérée
```



# Le programmeur fait le travail

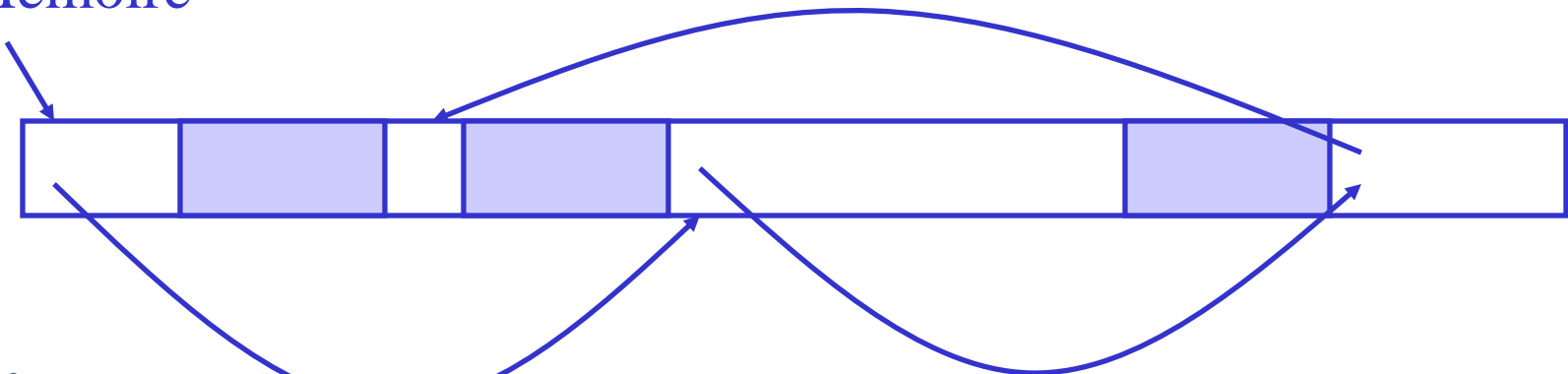
- Le programmeur utilise la librairie pour choisir quand allouer et dés allouer :
  - `void* malloc(long n)`
  - `void free(void *addr)`
  - Le système gère le besoin en mémoire en demandant des pages supplémentaires si nécessaire,
  - L'avantage c'est que le programmeur est intelligent ,
  - L'inconvénient c'est que le programmeur est généralement un peu bête , oublie de dés allouer et n'a pas envie de s'ennuyer avec ces détails.



# Gestion explicite de la mémoire

- Fonctionnement de `malloc/free` ?
  - Les blocs de mémoire libres sont stockés dans une liste chaînée,
  - `malloc`: recherche dans cette liste des blocs de mémoire libre,
  - `free`: insère en tête de liste le bloc libéré

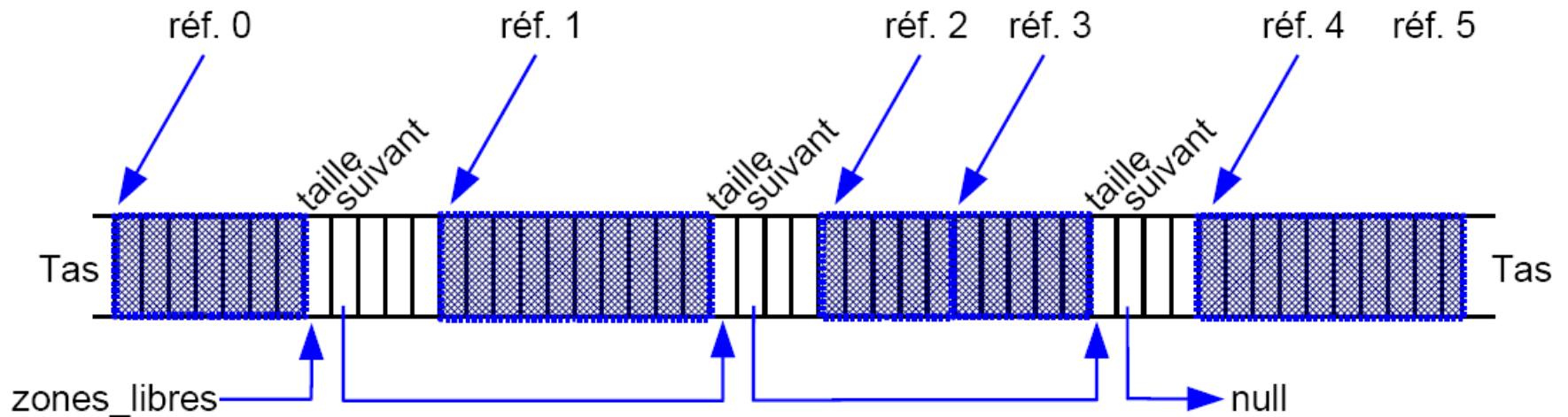
listeMémoire



# Gestion explicite de la mémoire

- Solutions:
  - Utilisation de liste multiple par taille de bloc,
    - `Malloc` and `free` sont alors en  $O(1)$
    - Mais si le besoin est de 4 blocs et que cette liste est vide on est en famine alors que des blocs plus grands existent.
  - Les blocs sont de taille en puissance de 2
    - Subdiviser les blocs pour obtenir la bonne taille
    - Les blocs adjacent sont réunis pour former des blocs plus gros,
    - Néanmoins on a 30% d'espace mémoire perdu en moyenne.
  - La gestion mémoire a toujours un coût.

# Liste des blocs libres



# Gestion mémoire automatique

- Système automatique de gestion mémoire = ramasse-miettes
- Doit pouvoir garder (ou retrouver) les données/objets encore actifs, et du coup être capable de libérer les autres
- Connaît les zones mémoire libres (allouables) et celles qui sont occupées,
  - Avant très consommateur de temps,
  - Surcote dépend de la façon de programmer, du type d'objet etc.



# Gestion automatique

- Un système remplace le développeur en allouant et surtout dés allouant automatiquement
  - impossible d'oublier de dés allouer (mais possible de continuer de référencer à tort)
  - plus difficile (si pointeurs) voire impossible (si références) d'utiliser une donnée déjà libérée
  - Détritus : plus de référence avant la dés allocation: la mémoire est perdue (*memory leakage*),
  - Pointeur flottant (*dangling reference*) deux pointeurs sur le même objet, l'un dés alloue la mémoire et l'autre pointe sur du vide...



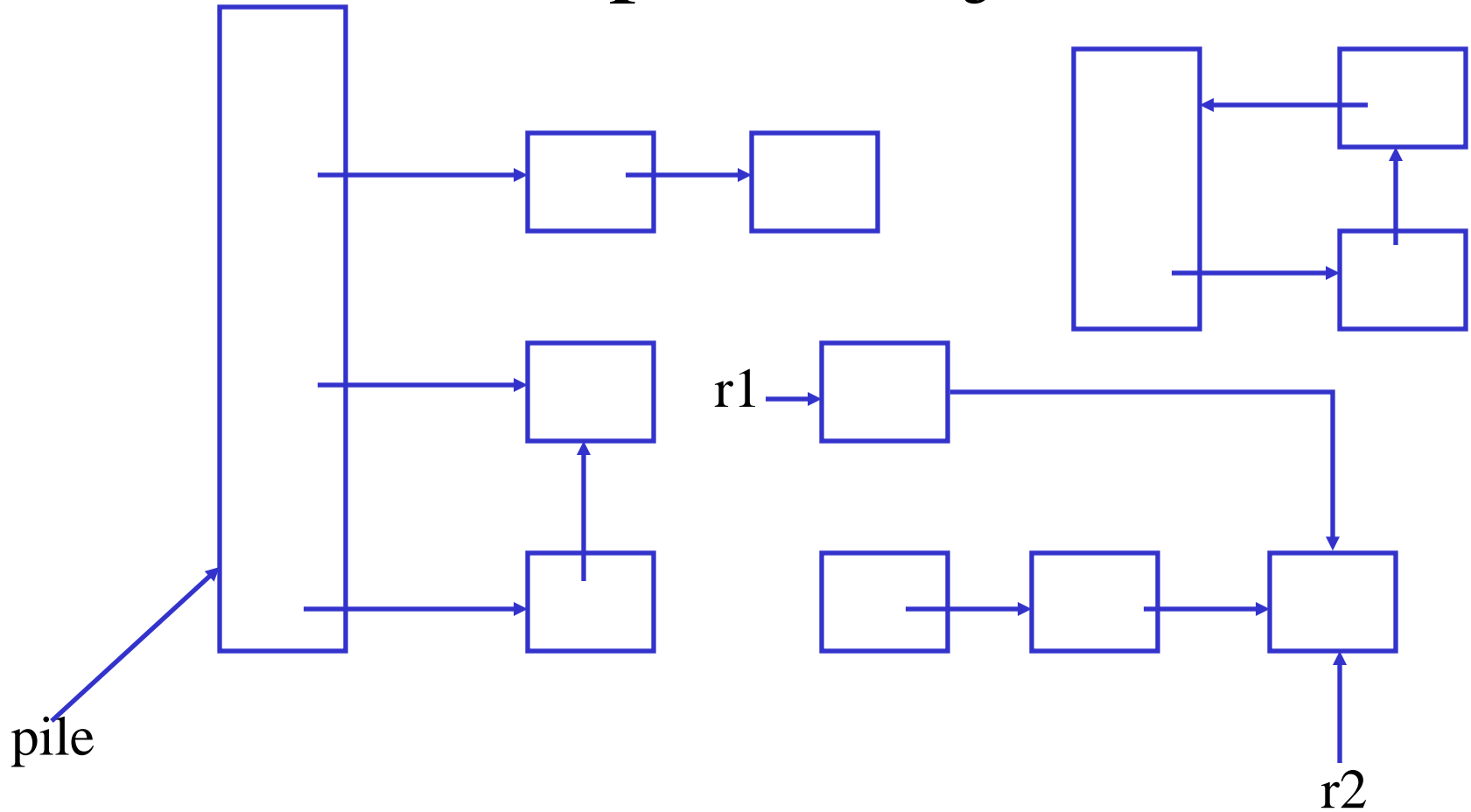


# Gestion automatique

- Quand peut on décider qu'il est temps de dés allouer la mémoire ?
  - On ne le sait pas précisément,
  - Donc on prend une approche conservative
  - Solution évidente lorsqu'il devient inatteignable depuis une **racine**,
    - Les racines: les registres, la pile, les données statiques
    - Si à partir d'une racines l'objet est inatteignable alors c'est un détrit.



# Graphe d'objet



– Comment évaluer l'atteignabilité ?



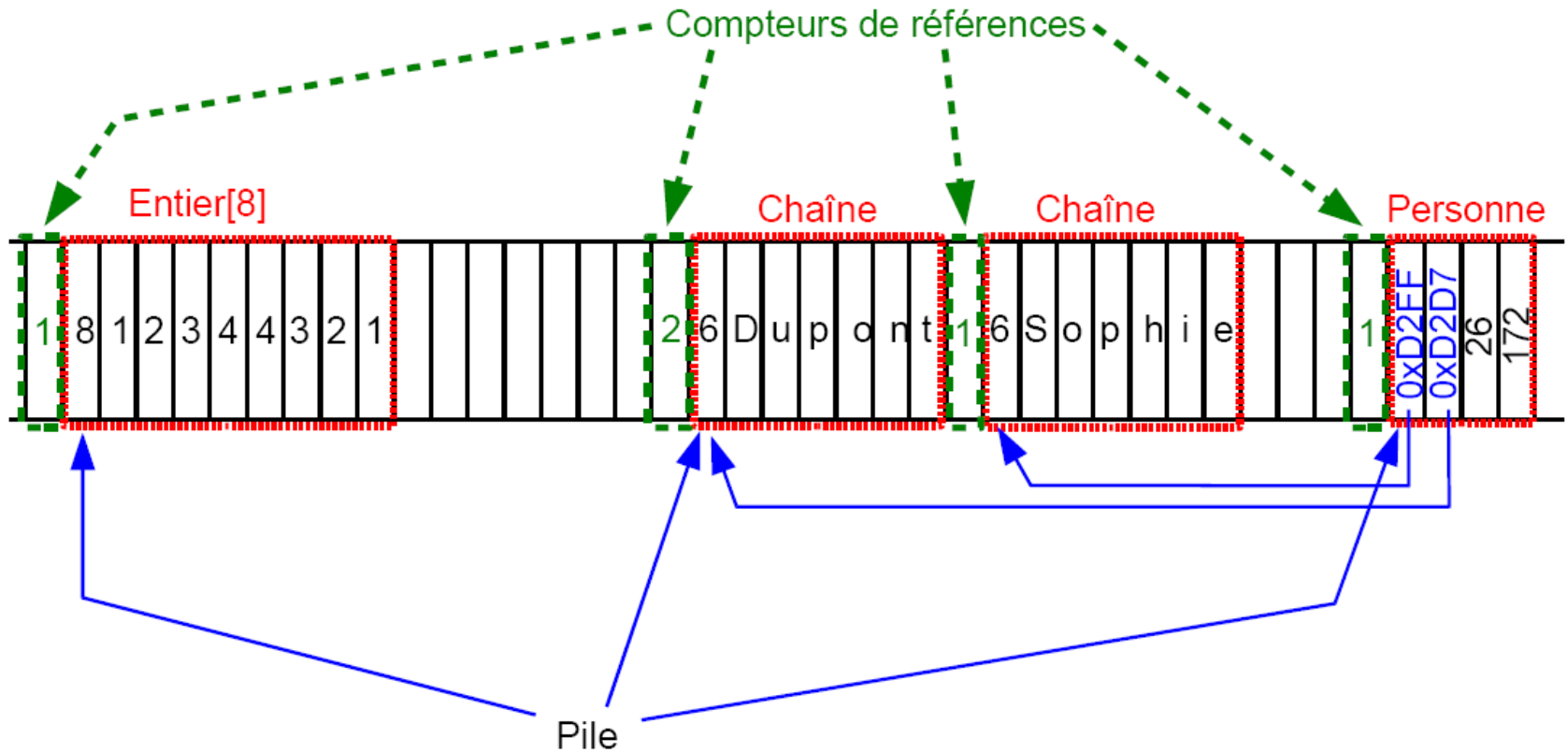
# Algorithmes de GC

- Comptage de références
- Marquage - balayage
- Copie - compactage

# Comptage de référence

- À chaque objet (ou donnée, structure, zone mémoire) alloué est associé un compteur (entier) indiquant le nombre de références sur cet objet.

# Comptage de référence



# Comptage de référence

- A chaque affectation, on met à jour les compteurs concernés:

```
a = new X(); // nb_réf(X1)=1
b = a;      // nb_réf(X1)=2
a = new X(); // nb_réf(X1)=1; nb_réf(X2)=1
b = null;   // nb_réf(X1)=0: X1 libérable
```

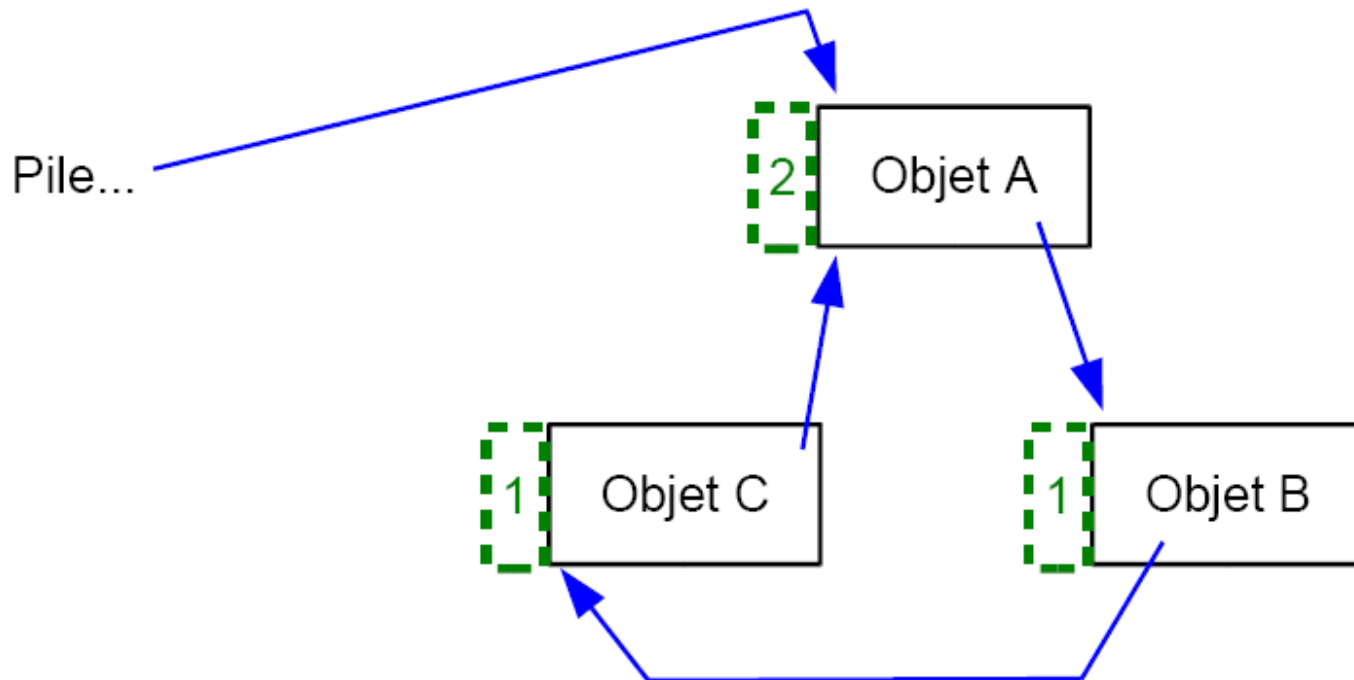
- Libération peut être immédiate (+ simple) ou différée

# Comptage par référence

- Libérer un objet X1:
  - récupérer la mémoire de X1 (remise en liste libre)
  - diminuer les compteurs de références des objets pointés par X1
  - libérations en cascade possibles (peut prendre du temps)

# Comptage de référence

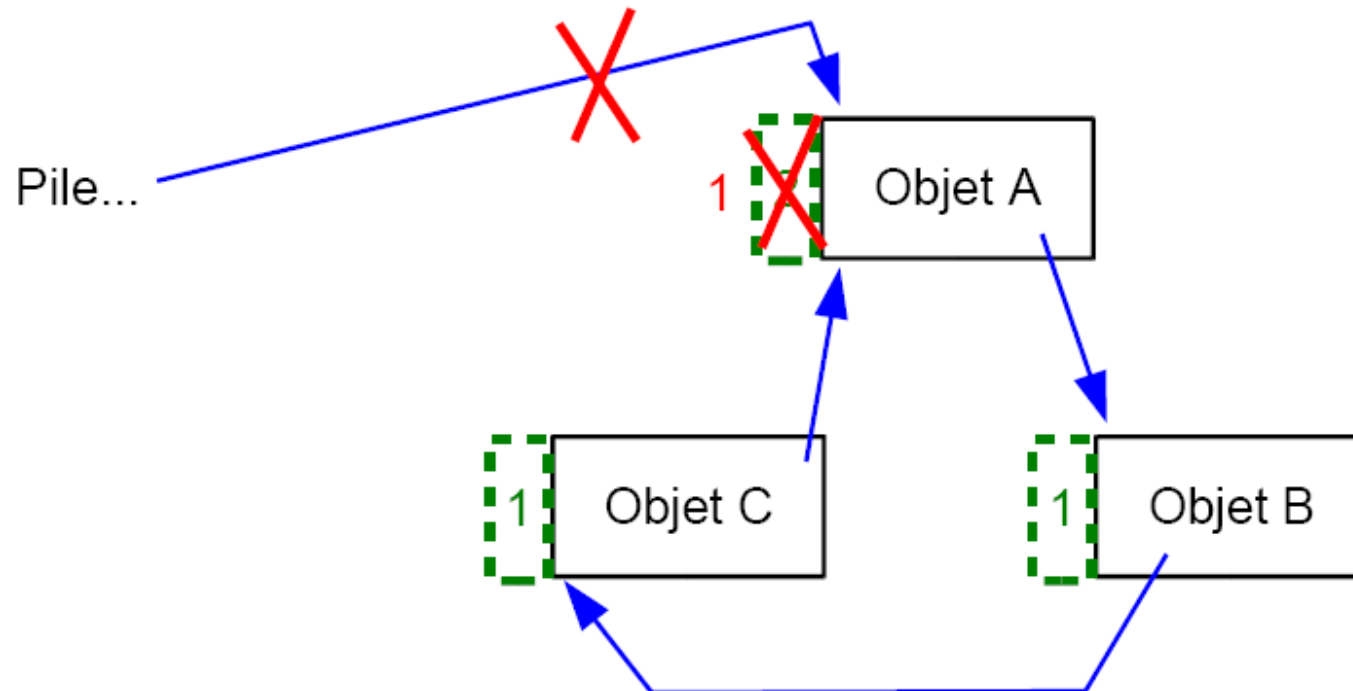
- Le problème des cycles





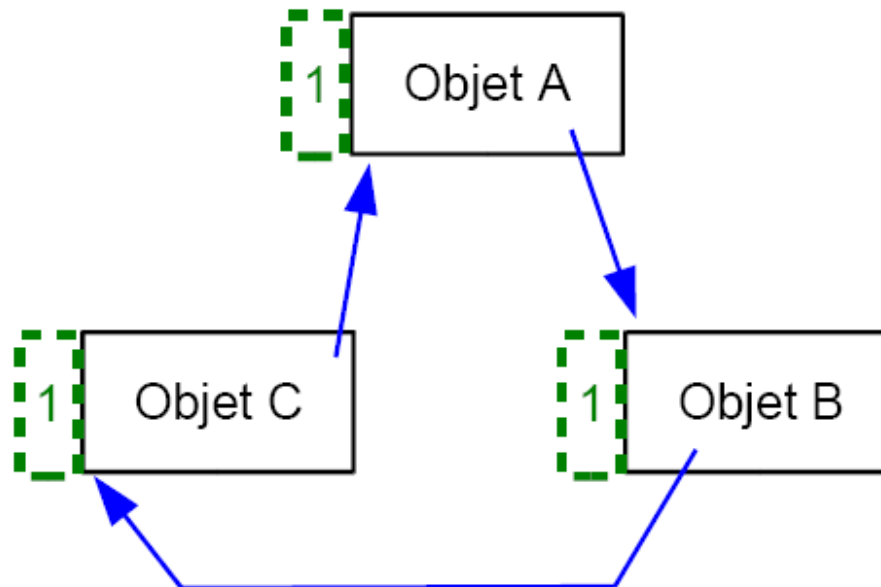
# Comptage de référence

- Le problème des cycles



# Comptage de référence

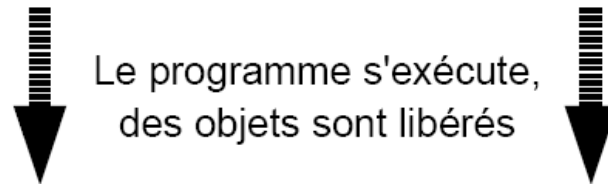
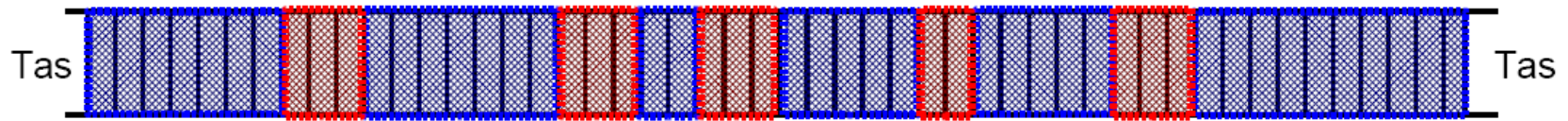
- Cycles non détectés
  - Besoin en plus d'un système de détection de cycles
  - Le cycle A/B/C n'est plus référencé, mais ses compteurs sont  $>0$ , donc les objets ne sont pas collectés...



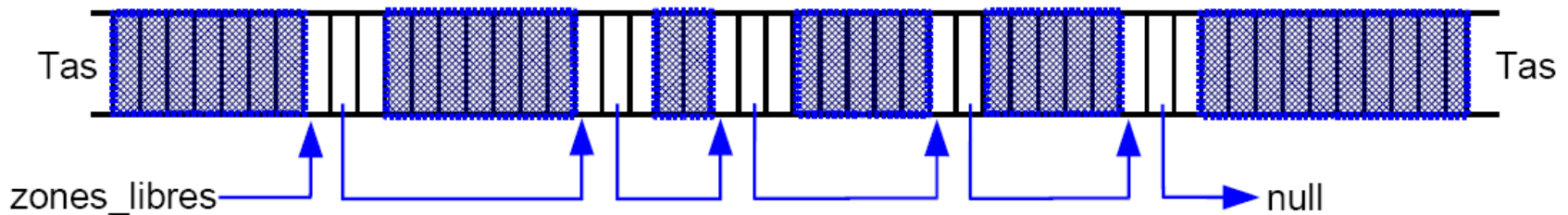
# Comptage de référence

- Problème de la fragmentation

A l'instant T1:



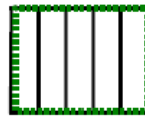
A l'instant  $T2 > T1$ :



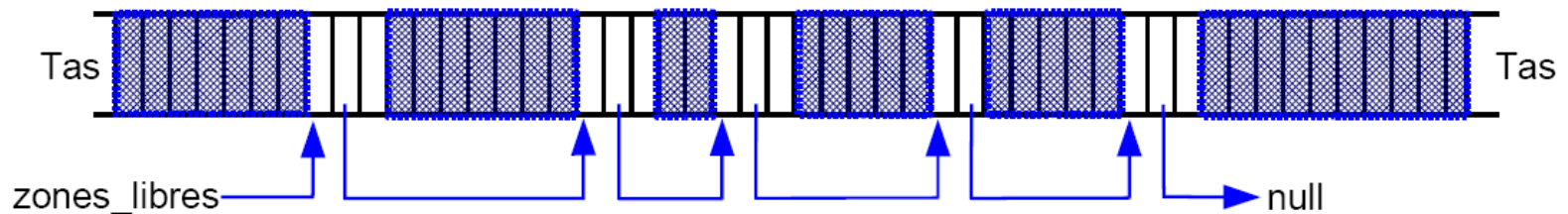
# Comptage de référence

- Fragmentation possible

A l'instant T2, on veut allouer 5:



Mais pas de zone mémoire assez grande disponible:



Pourtant il y a de la mémoire libre (14 en tout): c'est la fragmentation.

# Comptage de référence

- Bilan
  - Simple
  - Exécution répartie le long de celle du programme.
  - Coûteux au total: MAJ de compteur(s) à chaque affectation
  - Pas de gros délai si objets libérés un par un.
  - Délais si cascades...
  - Problème des cycles
  - Fragmentation possible



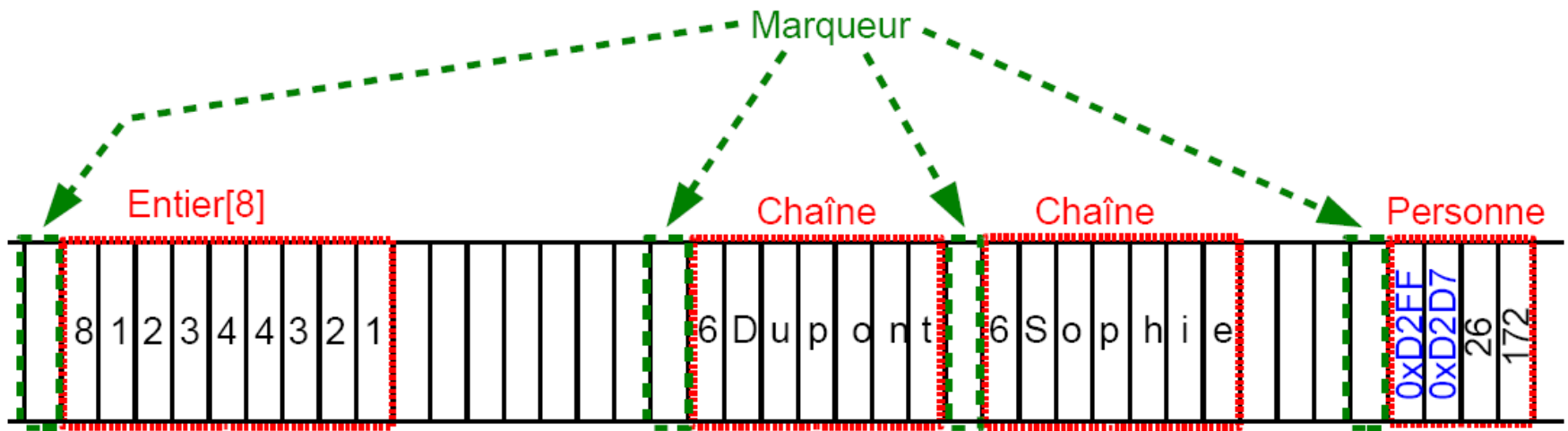
# Marquage balayage

- Le ramasse-miettes se déclenche par intermittence
  - Exécution du ramasse-miettes arrête le programme temporairement
- Lorsque ramasse-miettes se déclenche:
  - phase de marquage : trouver les vivants
  - phase de balayage : recycler les morts



# Marquage balayage

- A chaque objet alloué est associé un marqueur (ou drapeau, ou *mark flag*)



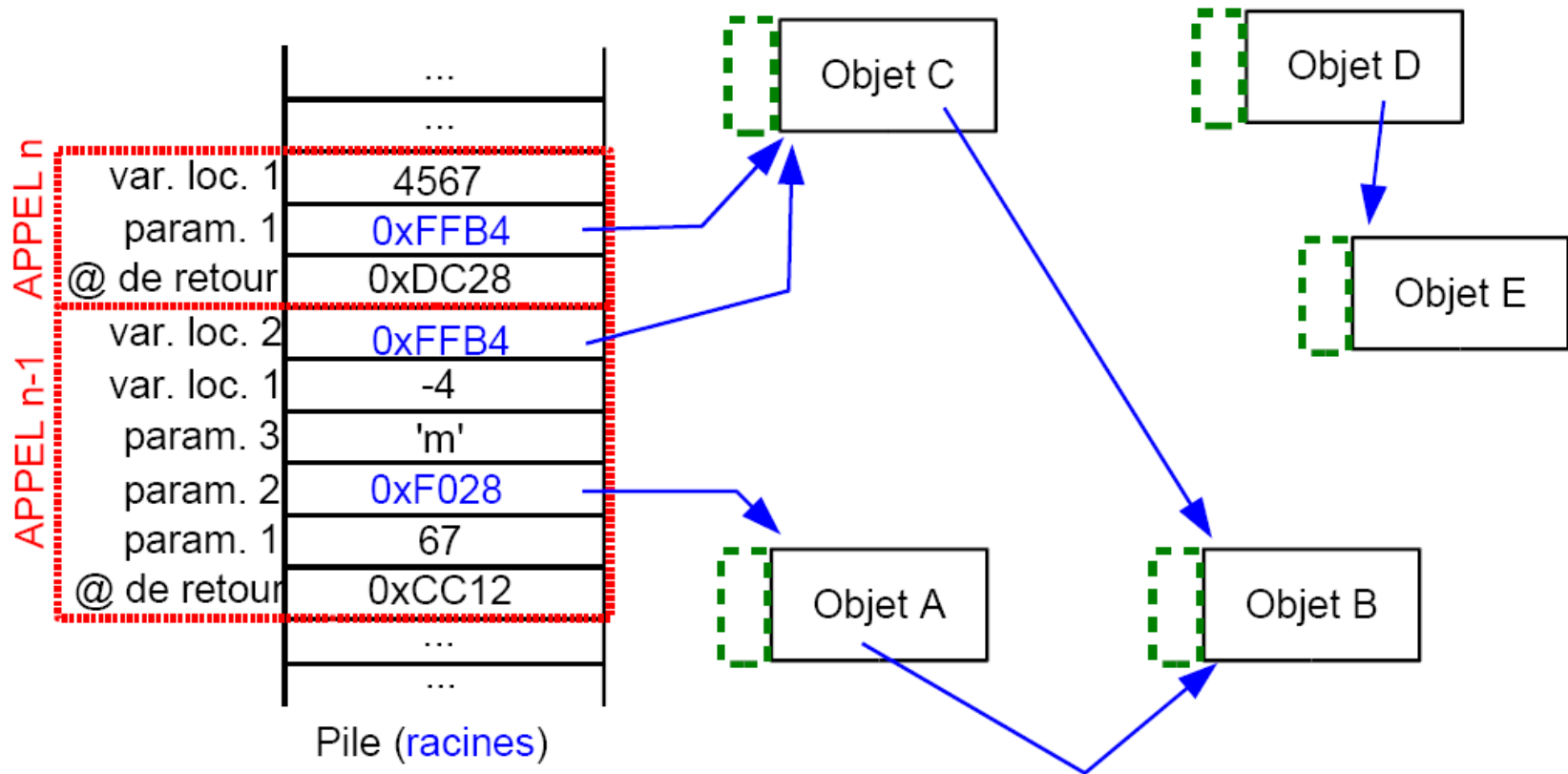
# Algorithme de marquage

- Partir des racines (piles, zone statique) du graphe d'objets
- Pour chaque objet rencontré
  - s'il est déjà marqué, rien à faire
  - sinon le marquer et propager l'algorithme sur tous les objets qu'il référence
- Quand le marquage se termine, on a tous les actifs. Les autres sont morts.





# Algorithme de marquage



# Algorithme de balayage

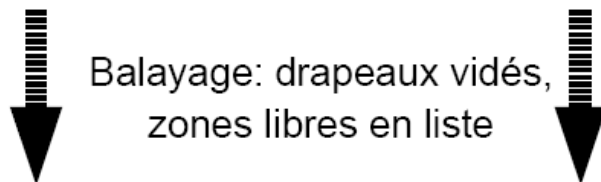
- On parcourt la liste de toutes les zones mémoires.
- Si marqué, on conserve (on démarque pour le coup suivant),
- Si pas marqué, on intègre la zone dans la liste libre.

# Algorithme de balayage

Après marquage, avant balayage:



zones\_libres → null



Après balayage:



zones\_libres → [pointing to first free zone] → [pointing to second free zone] → null

# Marquage - balayage: bilan

- Pauses longues: marquage + balayage
  - durée du marquage dépend de la taille du graphe d'objets (surtout les vivants)
- Amélioration: incrémental (pauses fractionnées)
- Les cycles ne sont pas un problème
- Fragmentation possible

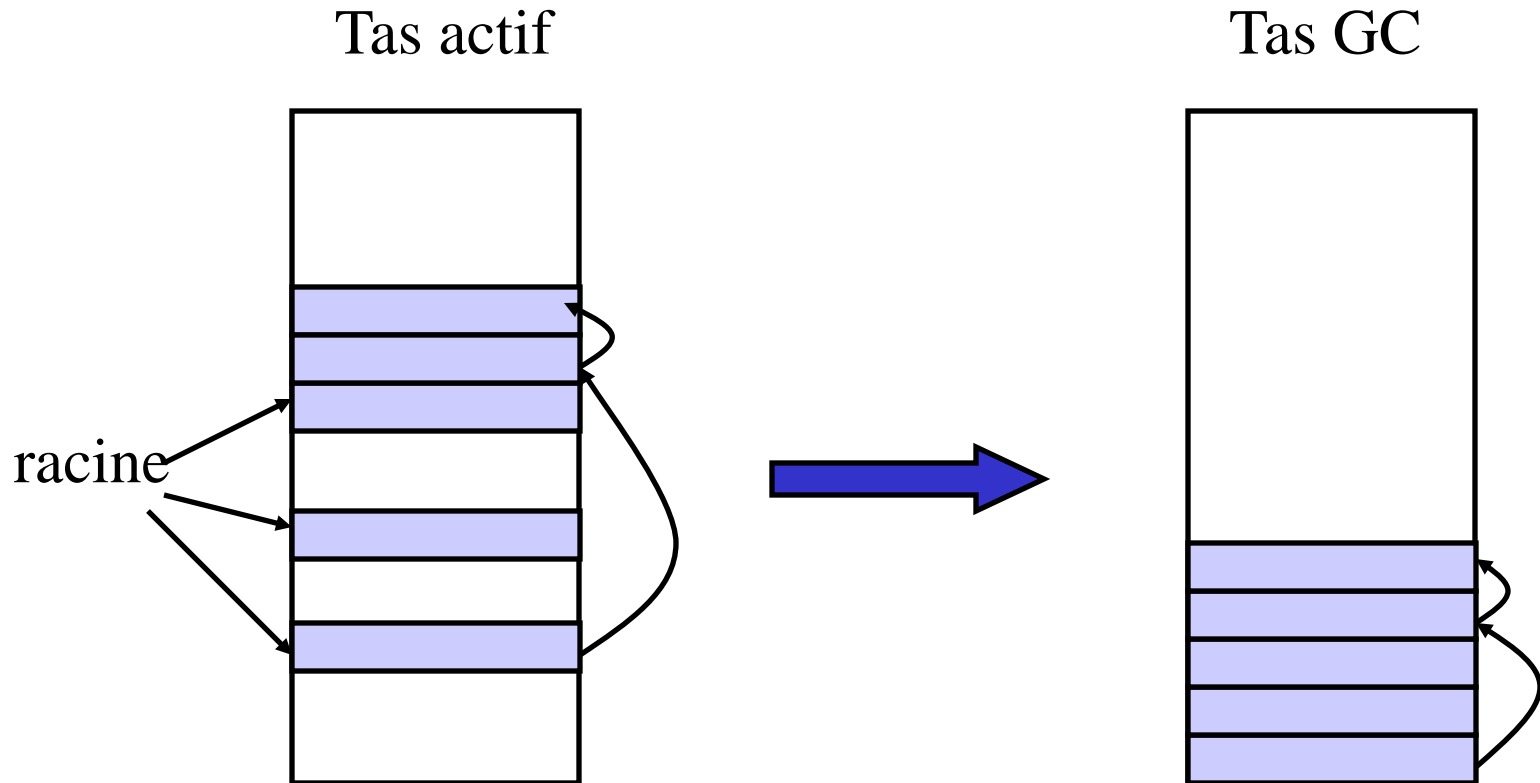


# Copie compactage

- Principe: parcours du graphe d'objets (comme marquage-balayage)
- Recopie des vivants dans nouvel espace mémoire (de façon contigüe), les morts sont abandonnés sur le champs de bataille,
- Le nouveau tas est le recopié,
- L'autre devient le futur tas du GC.

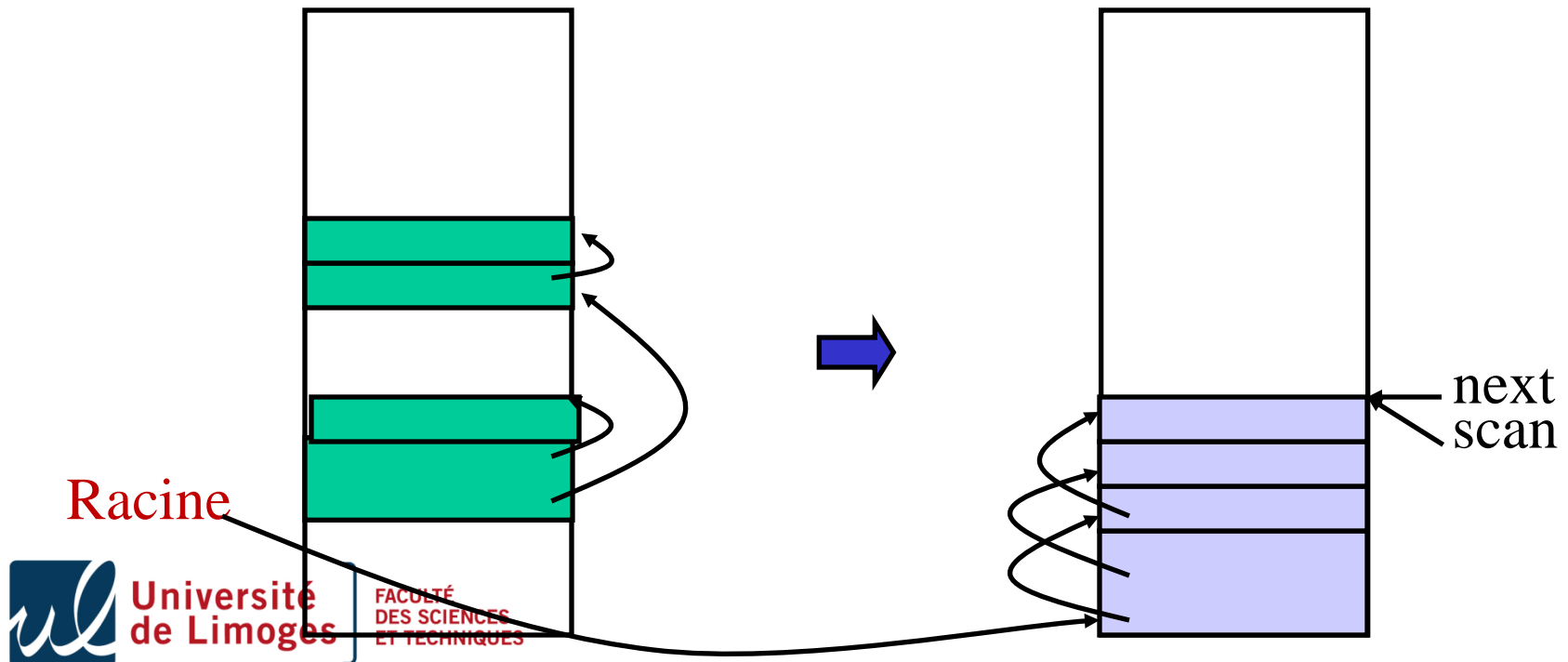


# Copie compactage



# Copie

- Algorithme de Cheney's
  - Traversée en largeur, copie du tas



# Copie - compactage: bilan

- Parcours comme marquage
- Recopie coûteuse
- Gestion de « forwarding pointers »
- Double espace mémoire
- Pas de fragmentation





# GC générationnel

- Observation 1 : un objet vieux et vivant a une très forte probabilité de le rester,
- Observation 2 : de nombreux objets sont créés avec une durée de vie très courte,
- Conclusion : la durée de vie d'un objet est le signe de non détritius.



# GC générationnel

- Assigner aux objets différentes générations G0, G1, ...
  - G0 contient des objets récents forte probabilité à recycler
  - G0 parcouru plus souvent que G1
  - Cas général utilisation de deux générations
  - Racines de G0 inclus tous les objets de G1 en plus de la pile et des registres.



# GC générationnel

- Comment éviter de scanner de vieux objets ?
  - Observation: les vieux pointent rarement les jeunes
    - Après la création d'un objet, son initialisation pointera vers des objets plus vieux
    - Si un objet ancien est modifié longtemps après sa création alors ceci peut arriver.
- Quand est on assez vieux pour changer de génération ?
  - Généralement après la passe de scan si il survit il est promu
- On peut utiliser des algorithmes différents suivant les générations,
  - Copie compactage pour les jeunes
  - Mark and sweep pour les vieux.



Any question ?

