
ÉCOLE
POLYTECHNIQUE
MONTRÉAL

# LOG3210
# Cours 10

## Environnements d'exécution
## (Gestion de la mémoire)

# Gestion du monceau (heap)

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Contrairement à la pile (stack) qui permet de stocker les données qui "vivent" le temps d'un appel, le monceau (heap) permet quant à lui de stocker les données qui ne sont pas locales à une fonction.

- La gestion du monceau (allocation / déallocation de mémoire et garbage collection) s'effectue aussi dans le contexte des environnements d'exécution.

- Plus précisément, on désignera sous l'appellation de gestionnaire de mémoire le sous-système responsable de la gestion du monceau.

# Gestionnaire de mémoire

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Comme nous l'avons mentionné précédemment, le gestionnaire de mémoire a deux tâches principales:
  1. **Allocation**: Lorsqu'un programme requière un bloc de mémoire de taille X, le gestionnaire tente de trouver un bloc libre de taille ≥ X et le retourne au programme. S'il n'y a plus d'espace disponible, le gestionnaire doit le signaler.
  2. **Déallocation**: Lorsqu'un bloc de mémoire est libéré, le gestionnaire doit l'insérer dans la liste des blocs libres (free list)
- Typiquement, le gestionnaire interagira avec l'OS pour obtenir davantage de mémoire, mais la retournera rarement à l'OS après déallocation!

# Propriétés du gestionnaire de mémoire

ÉCOLE POLYTECHNIQUE MONTRÉAL

- En plus de correctement allouer et désallouer la mémoire, on attend généralement d'un gestionnaire de mémoire qu'il soit aussi:
  1. **Efficace en terme d'espace.** Il doit minimiser l'espace mémoire nécessaire pour l'exécution d'un programme.
  2. **Efficace en terme d'exécution.** Les ordinateurs modernes ont plusieurs "couches" de mémoire. Le gestionnaire doit placer les objets les plus fréquemment accédés dans les couches les plus rapides.
  3. **Faible surcharge** (low overhead). Le coût associé aux opérations d'allocation / déallocation doit être le plus bas possible.

# Hiérarchie de mémoire

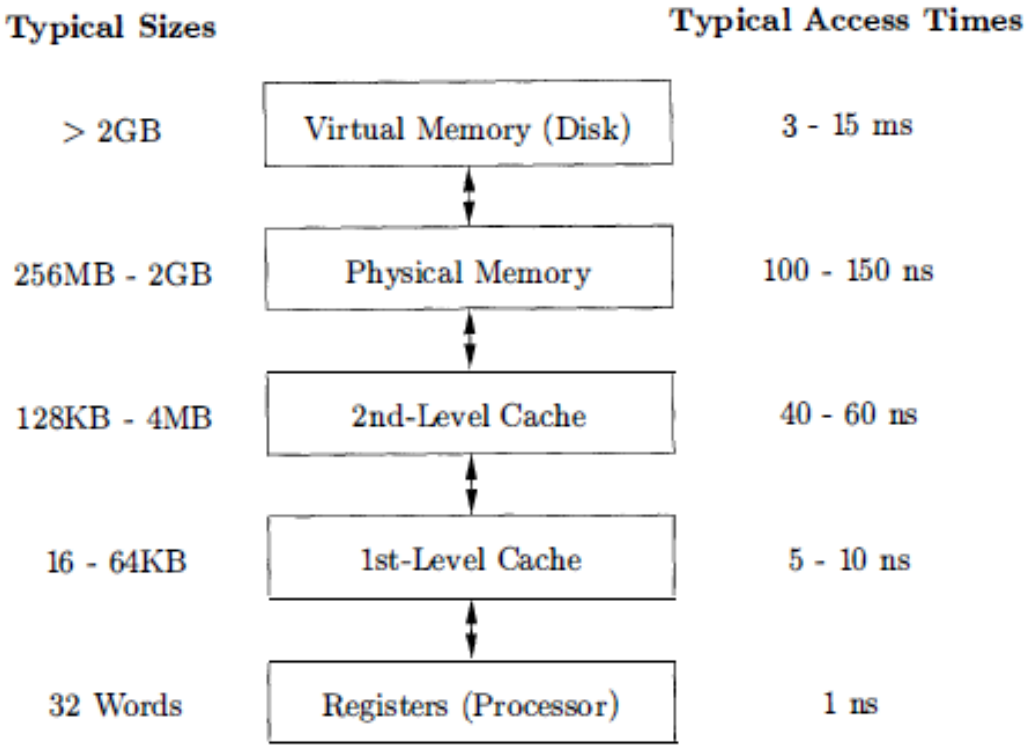



Figure 7.16: Typical Memory Hierarchy Configurations

# Localité des programmes

ÉCOLE POLYTECHNIQUE MONTRÉAL

- La plupart des programmes consacrent la majeure partie de leur temps d'exécution à une portion relativement restreinte du code et des données. On réfère à ce phénomène comme le degré de *localité* des programmes.

- Plus le temps d'exécution est concentré sur un sous-ensemble restreint d'instructions et de données, plus le degré de localité est élevé.

# Localité temporelle et spatiale

ÉCOLE POLYTECHNIQUE MONTRÉAL

- **Localité spatiale**: Étant donné un accès à une adresse mémoire, si les adresses mémoire voisines ont une plus forte probabilité d'être accédées, il y a localité spatiale.
- Les instructions ont généralement une localité spatiale élevée. Moins vrai pour les données sur le monceau.

- **Localité temporelle**: Étant donné un accès à une adresse mémoire, si cette adresse a de fortes chances d'être ré-accédée dans un court laps de temps, il y a localité temporelle.
- La localité temporelle est très difficile à estimer statiquement hormis pour certaines structures (ex. boucles). Stratégies dynamiques.

# Optimisations d'accès mémoire

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Différentes stratégies peuvent être déployées pour tirer profit de la localité spatiale et temporelle:
  - Conserver les instructions les plus récentes en cache.
  - Stocker des blocs d'instructions consécutives en cache.
  - Mettre les objets du monceau fréquemment accédés en cache.
  - Etc.

# Fragmentation

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Au début de l'exécution d'un programme, la mémoire du monceau consiste en un seul bloc de mémoire libre.


- Au fur et à mesure des allocations / dé-allocations, la mémoire se fragmente:


- Si la mémoire est mal gérée, il peut devenir difficile d'allouer des gros blocs de mémoire, et ce, même si la quantité totale de mémoire libre est suffisante.

# Stratégies d'allocation

ÉCOLE POLYTECHNIQUE MONTRÉAL

- **First-fit**: Au moment de l'allocation, le premier bloc libre de taille suffisante est fragmenté et retourné. Tend à créer beaucoup de fragmentation.

- **Best-fit**: Au moment de l'allocation, le plus petit bloc libre dont la taille est suffisante est fragmenté et retourné. Plus lent, mais plus efficace que first-fit pour combattre la fragmentation.

# Implémentation best-fit

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Afin d'accélérer la recherche d'un bloc de taille adéquate, on peut partitionner les blocs libres dans des *bins*.

- Par exemple, gcc définit les *bins* suivants: [1], [2], …, [16, 24[, [24, 32[, … , [502, 512[, [512, 1024[, [1024, 2048[, …, *wilderness chunk*

- Au moment de l'allocation, il suffit de trouver le *bin* correspondant, parcourir la liste de blocs libres dans ce *bin*, retourner le plus petit bloc de taille appropriée et insérer le fragment restant dans le *bin* approprié.

# Next-fit

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Bien que la stratégie best-fit réduise la fragmentation, elle ne tire pas profit de la localité spatiale.

- Les blocs mémoire qui sont alloués au même moment ont une probabilité élevée d'avoir des durées de vie similaires. S'ils sont contigüs en mémoire, au moment de la déallocation, il sera possible de reformer un gros bloc et de réduire davantage la fragmentation.

- Next-fit tente d'abord d'allouer la mémoire dans le *dernier* fragment créé. Ce faisant, on tire profit de la localité spatiale des instructions et on améliore la rapidité des allocations de mémoire.

# Déallocation et fusion de blocs

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Quand il y a déallocation, le gestionnaire de mémoire doit retourner le bloc dans la *free list* et possiblement dans le *bin* approprié.

- De plus, si le bloc déalloué est contigu à un autre bloc libre, on voudra généralement fusionner les deux blocs et réduire la fragmentation.

- Pour ce faire, nous utiliserons deux concepts:
  - Tags frontières: Les extrémités des blocs contiennent un bit qui indique si le bloc est libre ou occuppé, immédiatement suivis d'un compteur qui indique le nombre de byte dans le bloc.
  - Les blocs libres seront liés entre eux dans une liste doublement chaînée.

# Exemple de déallocation

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Le bloc B vient d'être retourné dans la *free list*.
- Le bit à l'extrême droite de A indique que A est libre tandis que le bit à l'extrême gauche de C indique qu'il est occupé. A et B seront fusionnés.
- Les compteurs de A et de B permettent de savoir que le nouveau bloc contiendra 300 bytes + l'espace précédemment occupé par les pointeurs et les tags frontière.
- Il ne reste qu'à ajuster les pointeurs *previous* et *next* de A et de B afin d'insérer le nouveau bloc dans le *bin* approprié.

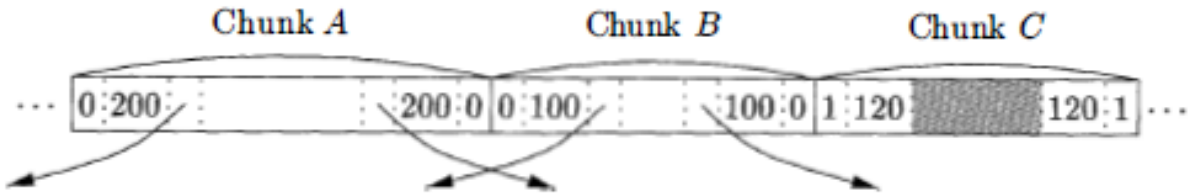



Figure 7.17: Part of a heap and a doubly linked free list

# Ramasse-miettes

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Le but du *garbage collector* est de libérer automatiquement les blocs de mémoires qui ne peuvent plus être référencés.

- En d'autres mots, s'il n'existe aucune référence vers un bloc de données sur le monceau, ce bloc est considéré comme *garbage* et peut être libéré.

# Hypothèses

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Afin de bâtir un *garbage collector* nous devons faire certaines hypothèses:
  1. Les objets ont un type et le *garbage collector* peut accéder au type des objets sur le monceau.
  2. Le type permet de déterminer la taille que l'objet occuppe en mémoire.
  3. Le type permet de savoir si l'objet contient des références vers d'autres objets.
  4. Les références contiennent l'adresse correspondant au début de l'objet. Les références ne contiennent jamais une adresse au milieu d'un objet.

# Pré-requis

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Pour que le garbage collector puisse accomplir sa tâche, il est nécessaire d'être en mesure de *garantir* qu'un bloc mémoire ne puisse être référé dans le futur.

- Si un langage est *type unsafe*, les objets alloués sur le monceau n'ont pas de type garanti.

- En conséquence, toutes références, peu importe leurs types, peuvent référer à n'importe quelle adresse mémoire (arithmétique de pointeurs, cast d'entiers en pointeurs, etc.) et on ne peut jamais garantir qu'un bloc ne puisse plus être référé.

# Performances

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Bien que la technologie des *garbage collectors* date de plusieurs décennies, ses coûts en performance sont tels que plusieurs langages populaires (C, C++) ne les supportent pas.

- Voici quelques critères à considérer lors du design d'un garbage collector:
  1. Temps d'exécution: Minimiser le supplément de temps.
  2. Espace mémoire: Minimiser la fragmentation.
  3. Temps de pause: Le garbage collector ne doit pas bloquer l'exécution du programme sur de (trop) longues périodes.
  4. Localité des programmes: Maximiser la localité spatiale et temporelle.

# Accessibilité des objets

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Lorsqu'un objet ne peut plus être référé, on dit qu'il n'est plus accessible (reachable). Le *garbage collector* peut seulement collecter les objets inaccessibles.

- Afin de construire un *garbage collector*, il faut donc considérer les actions d'un programme qui influencent l'accessibilité des objets.

# Accessibilité des objets (suite)

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Les principaux facteurs qui influencent l'accessibilité des objets sont:
  1. **Allocation d'objets.** Le gestionnaire de mémoire retourne une référence vers le bloc mémoire (objet) nouvellement alloué. Cet objet est désormais accessible.
  2. **Passage de paramètres et valeurs de retour.** Lorsqu'une référence à un objet est passé en paramètre ou retourné, l'objet référé demeure accessible.
  3. **Assignations de références.** Les instructions du type u = v où u et v sont des références ont deux effets. L'objet référé par v a maintenant une référence supplémentaire et l'ancien objet référé par u a maintenant une référence en moins.
  4. **Retour de fonctions.** Lors d'un retour, un enregistrement d'activation est retiré de la pile. Toutes les références dans cet enregistrement sont perdues.

# Inaccessibilité

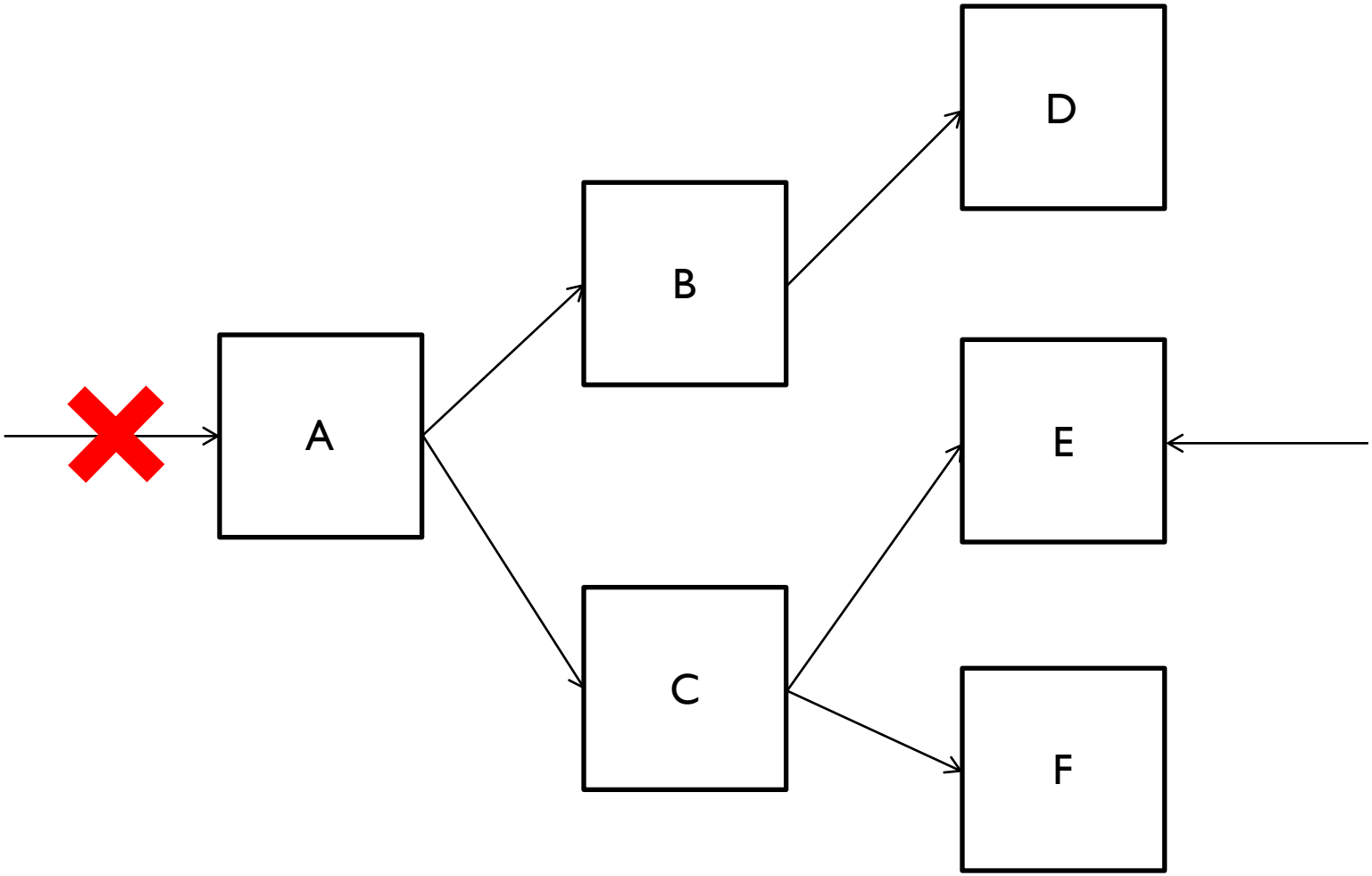
ÉCOLE POLYTECHNIQUE MONTRÉAL

- Dans la diapositive précédente, nous avons vu différentes conditions qui éliminent des références à des objets.

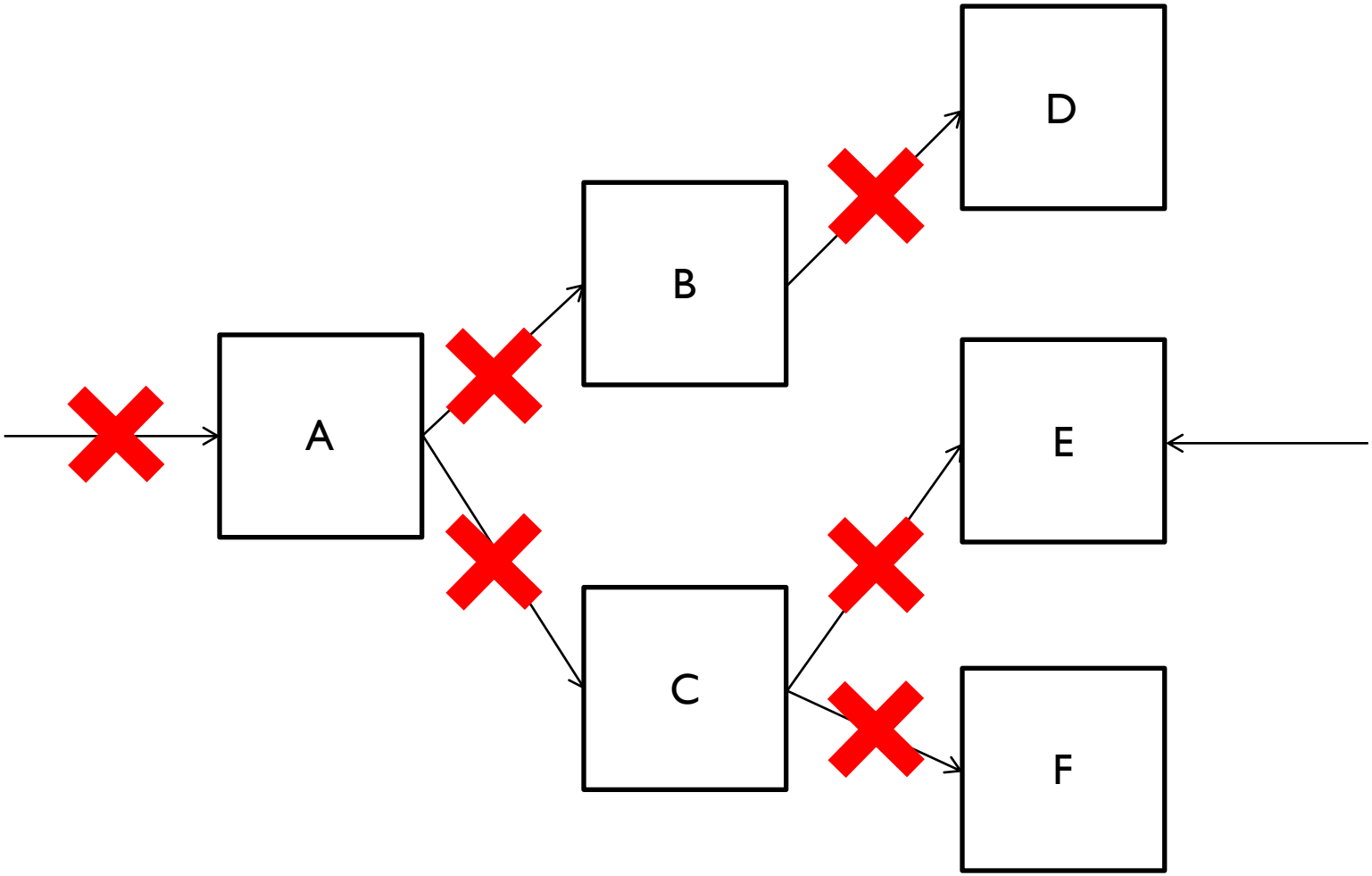
- Si la référence éliminée était la dernière qui référait à l'objet, cet objet devient inaccessible **et ne peut redevenir accessible par la suite**. (Pré-requis pour le garbage collector)

- Lorsqu'un objet devient inaccessible, il faut récursivement inspecter les références qu'il contient afin de les éliminer. Ce faisant, il est possible qu'on élimine d'autres objets…

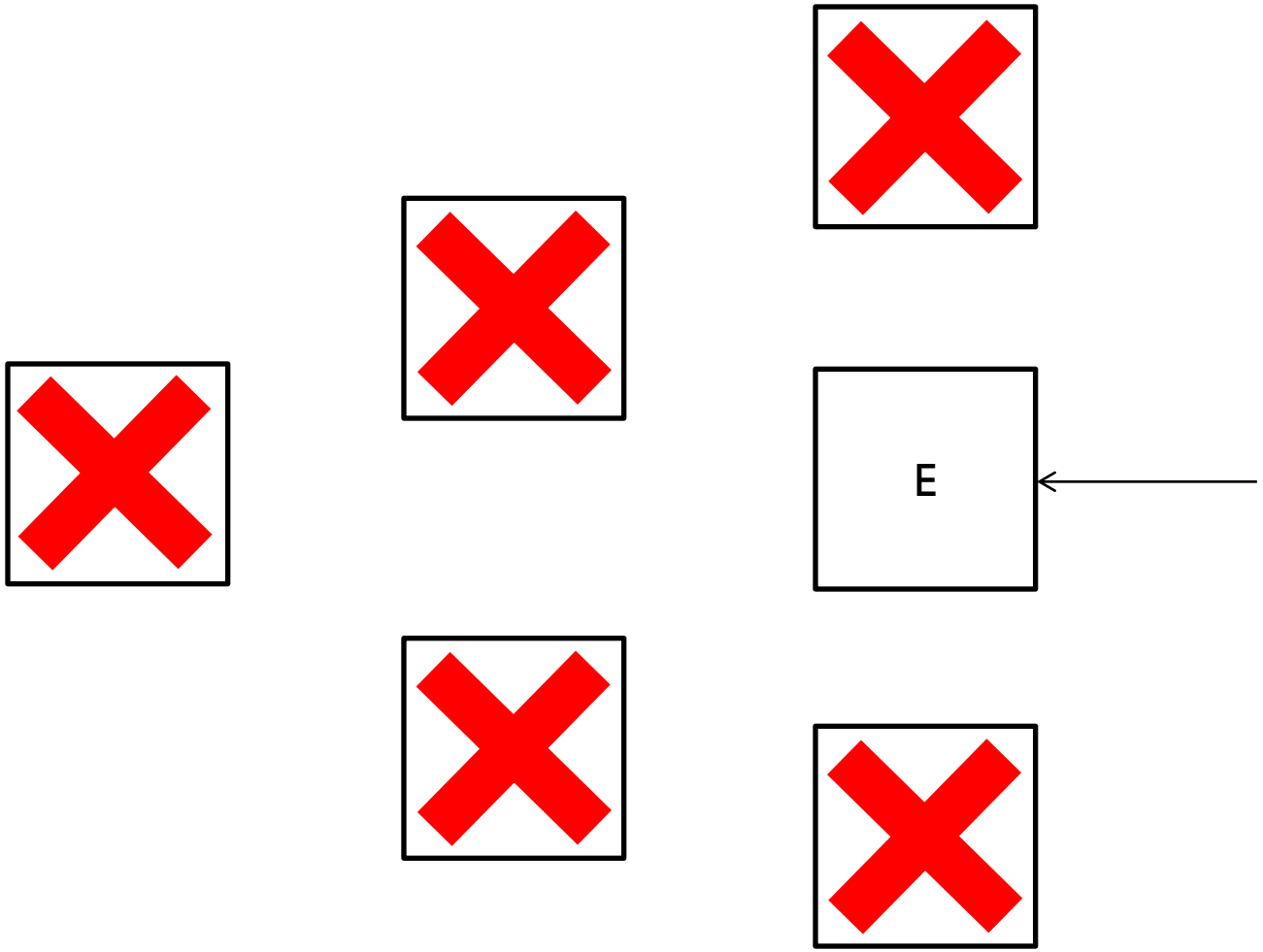
# Exemple élimination de références



# Exemple élimination de références



# Exemple élimination de références



# Décompte de références

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Intuitivement, si on attache un compteur de références à chaque objet, il est possible de libérer un objet lorsque son compteur atteint zéro.
- Les *garbage collectors* par décompte de références fonctionnent selon ce principe.
- Considérons un *garbage collector* par décompte de références simple…

# Garbage collector par décompte de références

ÉCOLE POLYTECHNIQUE MONTRÉAL

1. **Allocation d'objets:** Le compteur de l'objet est initialisé à la valeur 1.
2. **Passage de paramètres:** Le compteur de chaque objet passé en paramètre est incrémenté de 1 (On suppose que les **références** des objets sont passées en paramètre et non les objets eux-mêmes).
3. **Assignation de références:** Supposons u=v. Le compteur de l'objet référé par v augmente de 1 et le compteur de l'ancien objet référé par u diminue de 1.
4. **Retours de fonction:** Toutes les références locales à la fonction sont détruites et les objets auxquels elles pointaient voient leur compteur diminuer de 1.
5. **Perte transitive d'accessibilité:** Quand un compteur atteint zéro, on doit diminuer de 1 les compteurs de tous les objets référés par l'objet dont le compteur vient d'atteindre zéro.

# Références cycliques

- Les références cycliques posent problème pour les *garbage collector* par décompte de références.

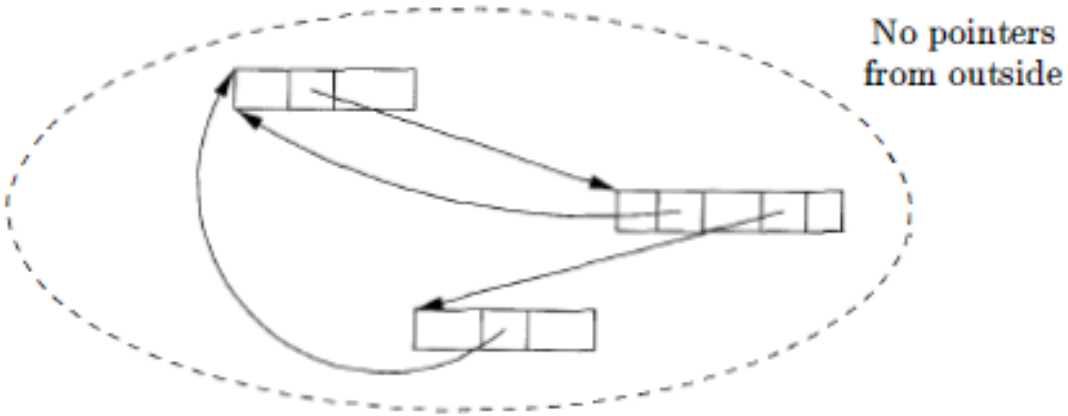



Figure 7.18: An unreachable, cyclic data structure

# Avantages et désavantages du décompte de références

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Désavantages:
  1. Lent. Beaucoup d'opérations supplémentaires doivent être faites à l'exécution pour maintenir les compteurs.
  2. Supporte mal les références cycliques, fuites de mémoire possibles.

- Avantages:
  1. Incrémentiel. Les opérations associées au *garbage collector* sont réparties sur l'ensemble du programme. Il n'est pas nécessaire de stopper l'exécution sur de longues périodes.
  2. La mémoire est libérée rapidement. Dès que le compteur tombe à zéro, le *garbage collector* peut collecter l'objet.

# Garbage collectors basés sur le traçage de références

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Contrairement aux *garbage collectors* par décompte de références qui sont incrémentiels, ceux basés sur le traçage s'exécutent périodiquement et requièrent de stopper l'exécution du programme.

- Ces *garbage collectors* examinent les objets sur le monceau, marquent ceux qui sont inaccessibles et les libèrent.

- Ils seront généralement exécutés lorsqu'il n'y a plus de mémoire de disponible ou qu'elle atteint un certain seuil.

# Marque-et-collecte

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Les *garbage collector* marque-et-collecte (Mark-and-Sweep) fonctionnent en deux phases: 1. marquer les objets inaccessibles sur le monceau et 2. les collecter.

- Le garbage collector marque-et-collecte a toutefois besoin d'un point de départ; un ensemble d'objets qui sont assurément accessibles afin de déterminer transitivement tous les autres objets accessibles.

- Cet ensemble de départ s'appelle l'ensemble racine (*root set*). En Java, il contiendra entre autre tous les champs statiques ainsi que toutes les variables sur la pile.

# Algorithme marque-et-collecte

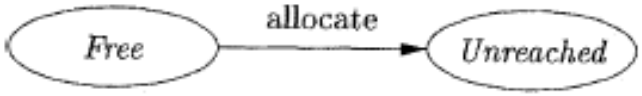
ÉCOLE POLYTECHNIQUE MONTRÉAL

```
    /* marking phase */
1)  set the reached-bit to 1 and add to list Unscanned each object
        referenced by the root set;
2)  while (Unscanned ≠ ∅) {
3)      remove some object o from Unscanned;
4)      for (each object o' referenced in o) {
5)          if (o' is unreached; i.e., its reached-bit is 0) {
6)              set the reached-bit of o' to 1;
7)              put o' in Unscanned;
            }
        }
    }
    /* sweeping phase */
8)  Free = ∅;
9)  for (each chunk of memory o in the heap) {
10)     if (o is unreached, i.e., its reached-bit is 0) add o to Free;
11)     else set the reached-bit of o to 0;
    }
```

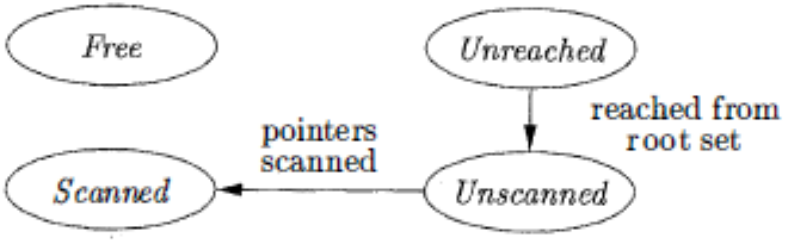
Figure 7.21: A Mark-and-Sweep Garbage Collector

# États de la mémoire dans un cycle marque-et-collecte

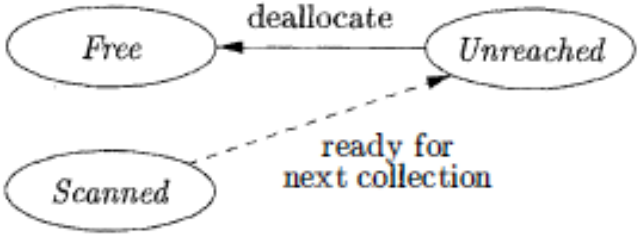





(a) Before tracing: action of mutator




(b) Discovering reachability by tracing




(c) Reclaiming storage

# Optimisation marque-et-collecte

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Dans l'algorithme initial, la boucle finale (ligne 9) est très coûteuse car il faut itérer sur tout le monceau.
- Une optimisation simple consiste à faire en sorte que le gestionnaire de mémoire maintienne deux listes: *Free* et *Unreached*.
- *Free* contient les blocs libres comme précédemment.
- *Unreached* contient les blocs alloués par le gestionnaire de mémoire.

# Optimisation marque-et-collecte (suite)

ÉCOLE POLYTECHNIQUE MONTRÉAL

- L'algorithme optimisé défini une liste supplémentaire *Scanned* en plus de la liste *Unscanned*.
- L'appartenance d'un objet à une liste (ligne 6) est vérifiée en O(1) en consultant un ensemble de bits dans l'objet. (Ex. 00 = *Free*, 01= *Unreached*, 10 = *Unscanned*, 11 = *Scanned*).

```
1)  Scanned = ∅;
2)  Unscanned = set of objects referenced in the root set;
3)  while (Unscanned ≠ ∅) {
4)          move object o from Unscanned to Scanned;
5)          for (each object o' referenced in o) {
6)                  if (o' is in Unreached)
7)                          move o' from Unreached to Unscanned;
            }
    }
8)  Free = Free ∪ Unreached;
9)  Unreached = Scanned;
```

Figure 7.25: Baker's mark-and-sweep algorithm

# Marque-et-compacte

ÉCOLE POLYTECHNIQUE MONTRÉAL

- La stratégie marque-et-compacte est une variation sur le thème marque-et-collecte qui permet de réduire la fragmentation.
- Un *garbage collector* marque-et-compacte déplace les objets du monceau de telle sorte qu'ils soient contigus et que la fragmentation soit éliminée.
- Il est par la suite plus facile d'allouer de larges objets.
- Permet de tirer plus facilement profit de la localité spatiale et temporelle puisque les prochains blocs alloués seront contigus.

# Algorithme marque-et-compacte

ÉCOLE POLYTECHNIQUE MONTRÉAL

- On peut considérer que *NewLocation* est une table de hachage contenant des paires (objet, adresse).

1. Marque-et-collecte classique (1 à 7).
2. Calcule les nouvelles adresses des objets (8 à 12).
3. Ajuste les références contenues dans les objets sur le monceau (13 à 17).
4. Ajuste les références contenues dans les objets de l'ensemble racine (18 à 19).

```
    /* mark */
1)  Unscanned = set of objects referenced by the root set;
2)  while (Unscanned ≠ ∅) {
3)      remove object o from Unscanned;
4)      for (each object o' referenced in o) {
5)          if (o' is unreached) {
6)              mark o' as reached;
7)              put o' on list Unscanned;
            }
        }
    }
    /* compute new locations */
8)  free = starting location of heap storage;
9)  for (each chunk of memory o in the heap, from the low end) {
10)     if (o is reached {
11)         NewLocation(o) = free;
12)         free = free + sizeof(o);
        }
    }
    /* retarget references and move reached objects */
13) for (each chunk of memory o in the heap, from the low end) {
14)     if (o is reached) {
15)         for (each reference o.r in o)
16)             o.r = NewLocation(o.r);
17)         copy o to NewLocation(o);
        }
    }
18) for (each reference r in the root set)
19)     r = NewLocation(r);
```

Figure 7.26: A Mark-and-Compact Collector

# Exemple marque-et-compacte

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Les pointillés représentent les références contenues dans les objets du monceau.

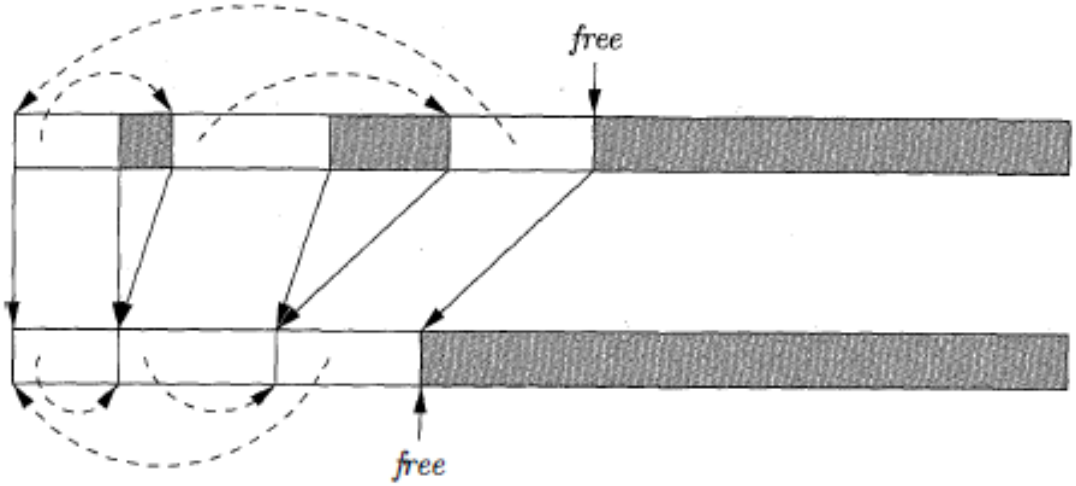



Figure 7.27: Moving reached objects to the front of the heap, while preserving internal pointers

# Collecte par copie

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Le collecteur marque-et-compacte nous force à nouveau à parcourir tout le monceau: une fois pour calculer les nouvelles adresses et une autre pour ajuster les références.

- La collecte par copie permet d'éviter le parcours complet du monceau.

- On divisera l'espace mémoire du monceau en deux blocs disjoints: *From* et *To*. À l'exécution, l'espace mémoire sera toujours alloué dans *From*, à la collecte, les blocs accessibles seront copiés dans *To* et à la fin de la collecte, *From* et *To* sont inversés.

# Algorithme de collecte par copie

ÉCOLE POLYTECHNIQUE MONTRÉAL

- *NewLocation* est encore une table de hachage qui contient des paires (objet, adresse).
- À la fin de l'algorithme, *From* et *To* sont inversés.

```
1)  CopyingCollector () {
2)        for (all objects o in From space) NewLocation(o) =NULL;
3)        unscanned = free = starting address of To space;
4)        for (each reference r in the root set)
5)              replace r with LookupNewLocation(r);
6)        while (unscanned ≠ free) {
7)              o = object at location unscanned;
8)              for (each reference o.r within o)
9)                    o.r = LookupNewLocation(o.r);
10)             unscanned = unscanned + sizeof(o);
          }
    }

    /* Look up the new location for object if it has been moved. */
    /* Place object in Unscanned state otherwise. */
11) LookupNewLocation(o) {
12)       if (NewLocation(o) = NULL) {
13)             NewLocation(o) = free;
14)             free = free + sizeof(o);
15)             copy o to NewLocation(o);
          }
16)       return NewLocation(o);
    }
```

Figure 7.28: A Copying Garbage Collector

# Garbage collector courte-pause

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Comme nous l'avons vu, les *garbage collector* basés sur le traçage de références doivent stopper l'exécution du programme, faire la collecte et relancer l'exécution.

- Afin de réduire les temps d'arrêt associés aux *garbage collectors* par traçage de références, deux stratégies peuvent être mises en place:
  1. Collecte incrémentielle: Les exécutions du programme et du *garbage collector* seront entrelacées.
  2. Collecte partielle: Un sous-ensemble des déchets sera collecté.

# Collecte incrémentielle

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Les *garbage collectors* incrémentiels doivent composer avec le fait que le programme s'exécute en même temps que la collecte.

- Contrairement aux *garbage collectors* basés sur le traçage, qui doivent récupérer *tous* les blocs libres, on exige seulement des collecteurs incrémentiels qu'ils libèrent *au moins* les blocs qui étaient inaccessibles au *début* de la collecte.

- Si cette exigence n'est pas respectée, certains blocs pourraient ne jamais être libérés et il pourrait y avoir des fuites de mémoire!

# Collecte incrémentielle (suite)

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Bien sûr, comme pour tout *garbage collector*, on exige aussi des collecteurs incrémentiels qu'ils ne libèrent *jamais* un objet qui est encore accessible.

- Cette condition peut s'avérer difficile à respecter compte tenu que le programme continue de s'exécuter en même temps que la collecte.

- Les collecteurs incrémentiels feront donc une *sur-approximation* de l'ensemble d'objets accessibles.

# Précision de la collecte incrémentielle

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Considérons *R*, l'ensemble d'objets accessibles (reachable) au début de la collecte. Comme le programme continue de s'exécuter en même temps que la collecte, *R* peut:
  1. Grandir si le programme alloue de nouveaux objets après le début de la collecte.
  2. Rétrécir si le programme élimine toutes les références à un objet dans *R*. L'objet devient effectivement inaccessible.

- Appellons *New* l'ensemble d'objets créés après le début de la collecte et *Lost* l'ensemble d'objets devenus inaccessibles après le début de la collecte.

# Bornes de précision de la collecte incrémentielle

ÉCOLE POLYTECHNIQUE MONTRÉAL

- La tâche du *garbage collector* est de déterminer les objets accessibles et d'éliminer ceux qui ne le sont pas.

- L'ensemble d'objets accessibles à la fin du traçage peut s'exprimer comme:

$$(R \cup New) - Lost$$

- Idéalement, l'ensemble d'objets accessibles S, tel que déterminé par le *garbage collector* devrait s'approcher au maximum de cette valeur. C'est notre borne inférieure.

# Bornes de précision de la collecte incrémentielle (suite)

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Pour atteindre la borne inférieure, le *garbage collector* devrait évaluer les conséquences de chaque élimination de référence par le programme. Ce serait très coûteux.

- Si on exige seulement du *garbage collector* qu'il libère les objets qui étaient libres au début de la collecte, il est conservateur de sur-approximer l'ensemble *S* comme:

$$R \cup New$$

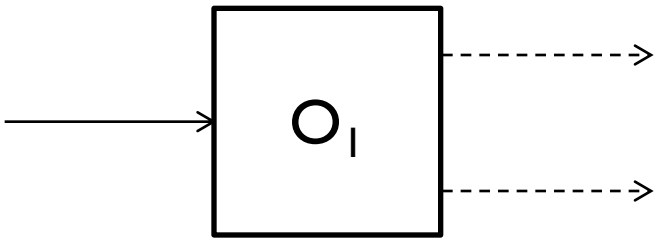
- En conséquence, l'ensemble *S* peut être borné comme suit:

$$((R \cup New) - Lost) \subseteq S \subseteq (R \cup New)$$

# Analyse d'accessibilité incrémentielle

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Si on tente de simplement entrelacer l'exécution d'un *garbage collector* marque-et-collecte et l'exécution d'un programme, des problèmes peuvent survenir:



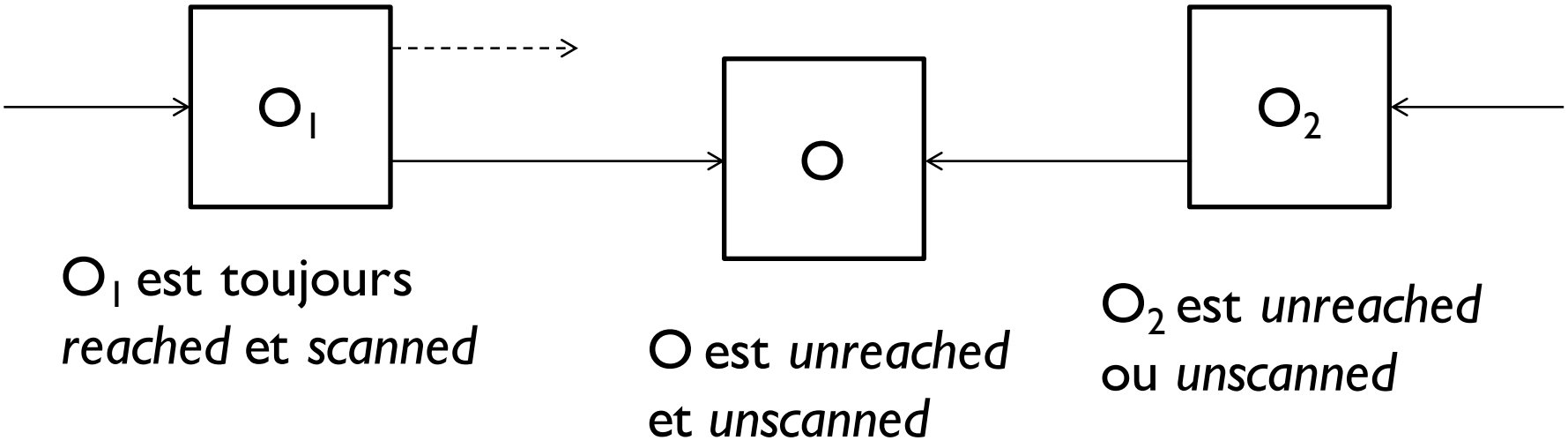

Le *garbage collector* trouve
$O_1$ et explore ses références.

$O_1$ est maintenant *reached* et *scanned*

# Analyse d'accessibilité incrémentielle (suite)

ÉCOLE POLYTECHNIQUE MONTRÉAL

- L'exécution est transférée au programme, qui stocke une référence à un objet *unreached* O dans $O_1$. Cette référence provient d'un objet $O_2$ qui est *unreached* ou *unscanned*.

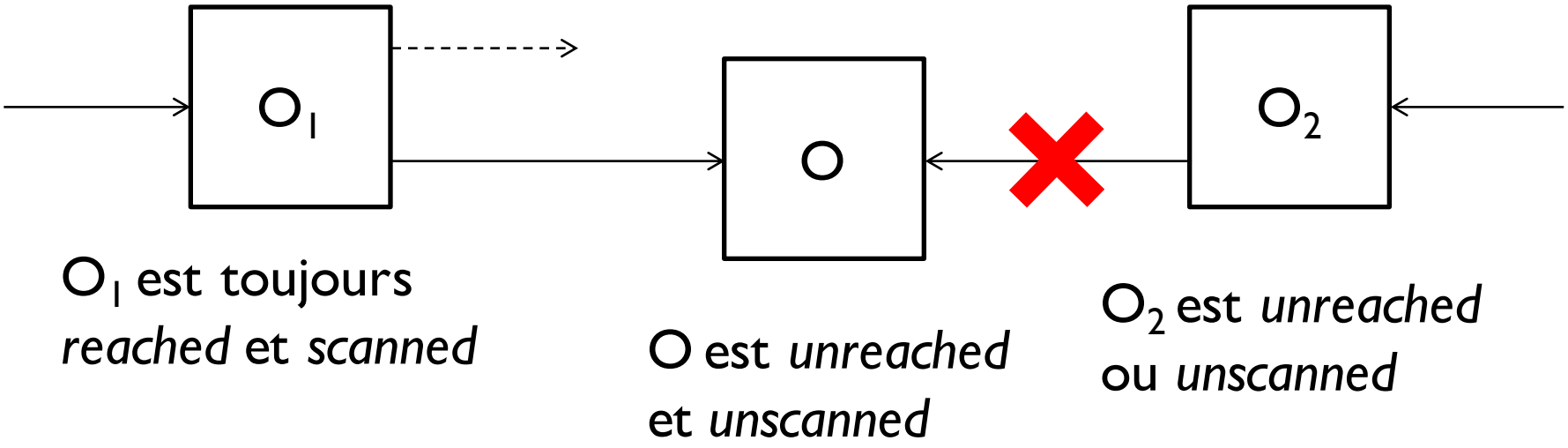


# Analyse d'accessibilité incrémentielle (suite)

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Le programme détruit la référence de $O_2$ vers O.

- Le *garbage collector* ne revisitera pas $O_1$, O demeurera *unreached* et sera donc détruit!



# Analyse d'accessibilité incrémentielle correcte

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Le problème provient du fait que le programme a violé un invariant de l'algorithme marque-et-collecte: tout objet *scanned* ne peut contenir que des références à d'autres objets *scanned* ou *unscanned*, jamais à des objets *unreached*.

- Une solution à ce problème est d'implémenter des barrières d'écriture:
  - Chaque fois qu'une référence à un objet O *unreached* est écrite dans un objet $O_1$ *scanned*, placer O dans la liste *unscanned* ou remettre $O_1$ dans *unscanned* afin qu'il soit revisité.

# Collectes partielles

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Plutôt que de partager l'exécution entre le programme et le *garbage collector*, la collecte partielle limite les impacts sur le temps d'exécution en visant un sous-ensemble du monceau.

- Il a été déterminé empiriquement qu'entre 80% et 98% de tous les objets nouvellement alloués meurent à l'intérieur de quelques millions d'instructions.

- En conséquence, il est avantageux pour un *garbage collector* de viser les objets les plus « jeunes ».

# Collectes partielles (suite)

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Pour la collecte partielle, nous définirons deux ensembles d'objets: *Target*, les objets visés par le collecteur et *Stable*, les objets qui seront ignorés par le collecteur.

- Il est possible d'adapter les *garbage collectors* non incrémentiels basés sur le traçage de références en modifiant la définition de l'ensemble racine:
  - L'ensemble racine contiendra désormais les champs statiques et les variables sur la pile en plus de tous les objets dans l'ensemble *Stable*.
  - Lors de l'exécution de l'algorithme, les références d'objets de *Target* vers des objets de *Stable* peuvent être ignorées.

# Collecte générationnelle

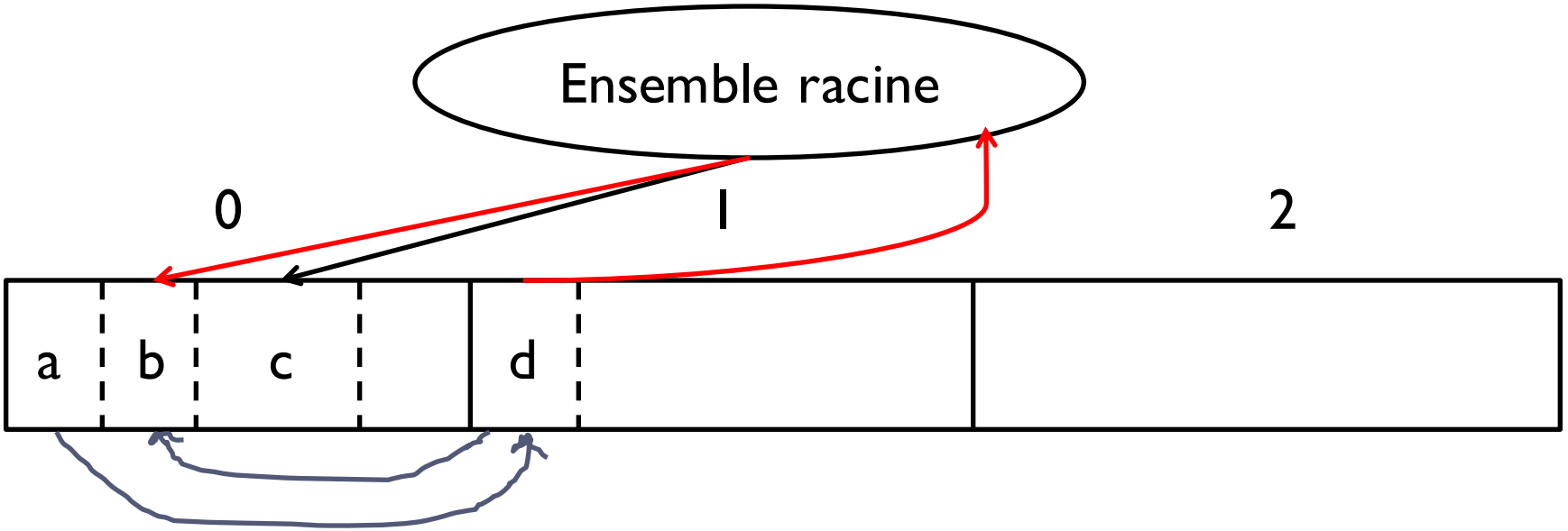
ÉCOLE POLYTECHNIQUE MONTRÉAL

- Idée: Collecter les objets les plus jeunes.

- Implémentation: Le monceau sera partitionné en « générations » de 0 à *n*.
  - L'allocation se fait toujours dans la partition 0.
  - Lorsque la partition 0 se remplit pour la première fois, le *garbage collector* est lancé avec *Target* = [0] et *Stable* = [1,n]. Les objets qui ne sont pas collectés sont copiés dans 1.
  - Dans le cas général, quand 0 se remplit, on fait la collecte avec *Target* = [0,i] et *Stable* = [i+1,n] où i est la partition pleine avec le plus grand index.

- Pour simplifier la construction de l'ensemble racine, chaque partition i maintient un ensemble souvenir (remembered set) qui contient tous les objets de partitions > i qui référencent à des objets dans i. On ajoute dans l'ensemble racine tous les ensembles souvenirs $\in$ *Target*.

# Collecte générationnelle (suite)



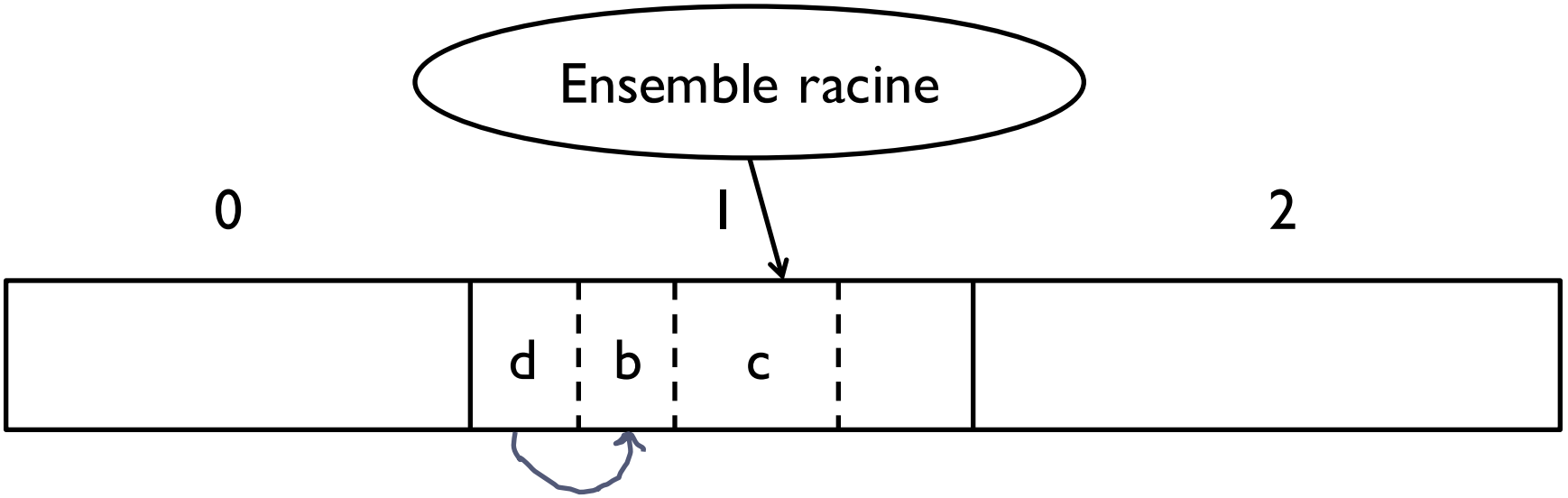

La partition 0 est considérée pleine.

*Target* = [0]

*Stable* = [1,2]

L'objet d fait parti de l'ensemble souvenir de b puisque d réfère b et est ajouté à l'ensemble racine (flèches rouges).

# Collecte générationnelle (suite)



# Avantages et inconvénients de la collecte générationnelle

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Les objets « jeunes » sont collectés souvent. Libère beaucoup de mémoire à peu de frais.

- Les « vieux » objets seront rarement collectés et leur collecte est coûteuse car pour collecter un bloc *m*, il faut traiter tous les blocs [0,m].

- Une solution possible est d'utiliser deux stratégies: collecte générationnelle pour les jeunes objets et algorithme du train pour les vieux objets.

# Algorithme du train

ÉCOLE POLYTECHNIQUE MONTRÉAL

- L'algorithme du train utilise des partitions de taille fixe, appelées des wagons. Les wagons sont liés entre eux et forment des trains.
- Il y a deux manières de libérer la mémoire:
  1. La mémoire du premier wagon du premier train est libéré de manière incrémentielle.
     Chaque wagon maintient un ensemble souvenir qui contient toutes les références en provenance de l'extérieur du wagon et cet ensemble souvenir est ajouté à l'ensemble racine.
     À la fin du traçage, les objets accessibles sont transférés dans un autre wagon.
     Le wagon contient donc seulement des objets inaccessibles et des références cycliques (objets qui ne sont pas référés de l'extérieur du wagon).
     Le premier wagon est détruit.

# Algorithme du train (suite)

ÉCOLE POLYTECHNIQUE MONTRÉAL

2. Parfois, le premier train ne contient aucune référence externe.
   En d'autres termes, l'ensemble racine ne contient aucune référence vers un objet du train et les ensembles souvenirs de tous les wagons du train contiennent des références provenant d'autres wagons du même train.
   Le train ne contient que des références cycliques et peut être éliminé.

- En résumé, à l'implémentation, on tente de sortir les objets qui ne sont pas en références cycliques du premier train.
  Simultanément, on tente de concentrer les références cycliques dans un même wagon.
  Ce faisant, on tente de maximiser la mémoire qui peut être libérée.