Garbage Collectors

Rémi Forax

Plan

- Histoire
- Garbage Collection
  - Compteur de références vs Tracing
  - Allocation / Desallocation
  - Compacter vs Copier
- *Garbage Collectors* de Java
- Les références faibles

Histoire

Bref histoire des *Garbage Collectors*

- 1970
  - LISP (John McCarty 1969)
  - Cheney algorithm (semispace)
- 1980
  - BASIC, PERL
- 2000
  - Sun Concurrent Mark And Sweep: CMS
- 2010
  - Garbage First: G1
  - Azul Continuous Concurrent Compacting Collector: C4

Langage

Langages qui utilisent un GC

- Languages typés dynamiquement
  - SmallTalk, PERL, Python, VB, PHP, JavaScript, etc
- Langages typés statiquement
  - Objective-C, Java, D, C#, Go, etc

Lexique

Concurrent

– Un GC concurrent est un GC qui s’exécute en même temps que l’application

Parallele

– Un GC parallele est un GC qui effectue une des tâches du GC en utilisant plusieurs threads

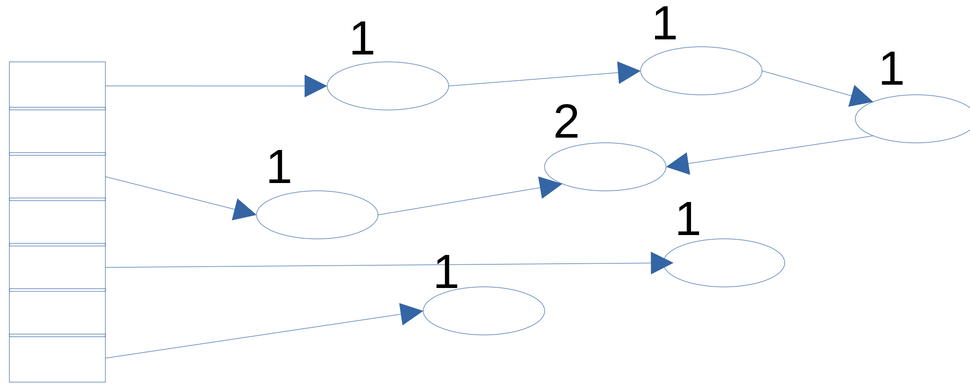
Incrémentale

– Un GC incrémentale est un GC qui effectue une partie de la tâche au lieu d’effectuer toute la tâche

Compteur de références

Compteurs de référence

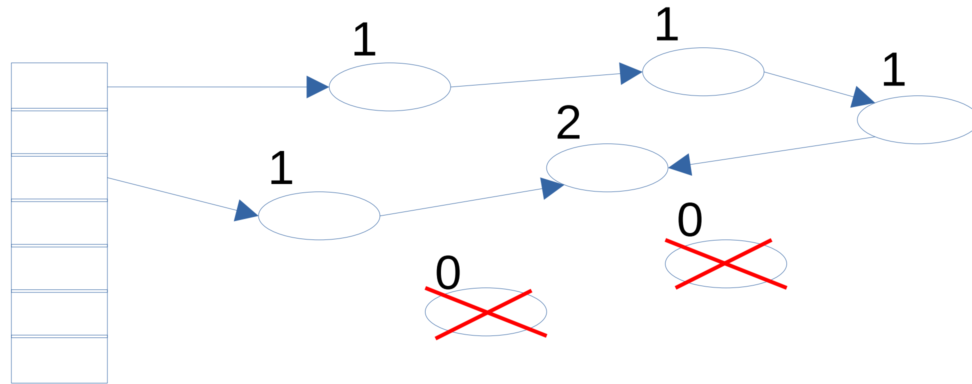
Chaque objet a un compteur de références



À chaque fois que l'on crée une référence sur un objet,
on incrémente le compteur

Compteurs de référence

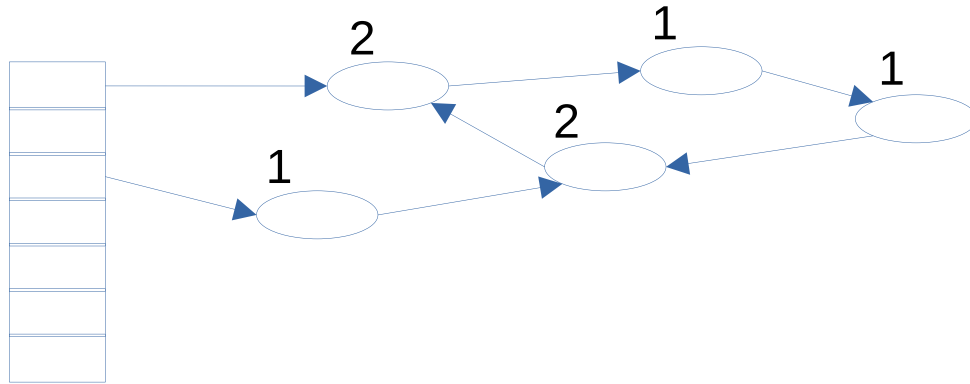
Chaque objet à un compteur de références



Si le compteur tombe à zéro, on supprime l’objet

Compteurs de référence

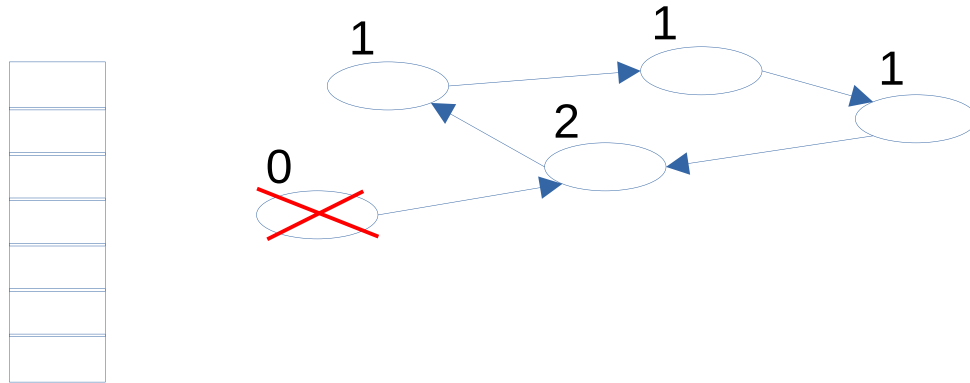
Chaque objet à un compteur de références



Problème avec les cycles !

Compteurs de référence

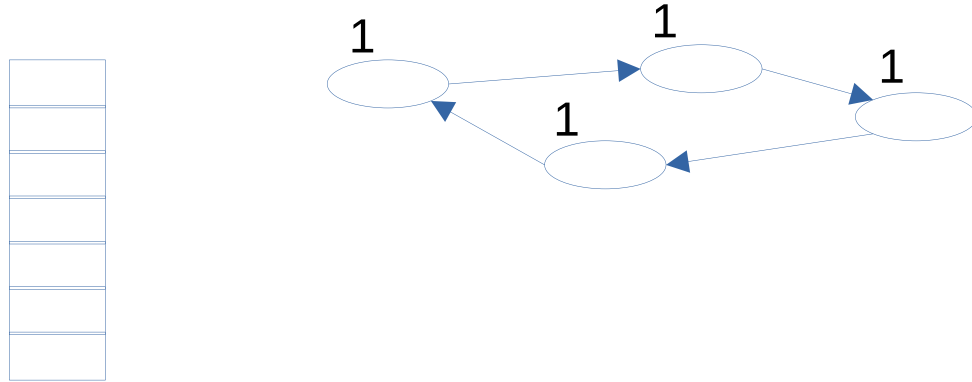
Chaque objet à un compteur de références



Problème avec les cycles !

Compteurs de référence

Chaque objet à un compteur de références



Python détecte les cycles, Objective-C non !

Compteur de référence

Avantage: la libération est prédictible

- Cela revient à faire les free() automatiquement

Deux problémes

- Les cycles
- Le multi-threads,
  il faut mettre à jour les compteurs de façon atomique
  - Les opérations atomiques coûtent cher

Tracing

Tracing *roots*

Ce sont les endroits où l’on va trouver les objets initiaux qui référencent les autres objets

- La pile de chaque thread
- Les globales (static en Java)
- Les globales de JNI/VM
  (objets Java stockés dans le code C)

Tracing

On part des *roots*, on marque tous les objets que l'on rencontre récursivement

- 2 problèmes
  - Comment on sait si on a une référence et pas un grand entier sur la pile ?
  - Comment on fait pour que l'algo ne soit pas récursif ?

Conservative Tracing

Boehm GC

- Marche sur une pile C
  - Pas d’information sur le type des valeurs sur la pile
- On sait que le tas est entre deux adresses
  - Toutes les valeurs entre les deux adresses sont des pointeurs

  => Cela veut dire que l’on garde des objets que l’on ne devrait pas !

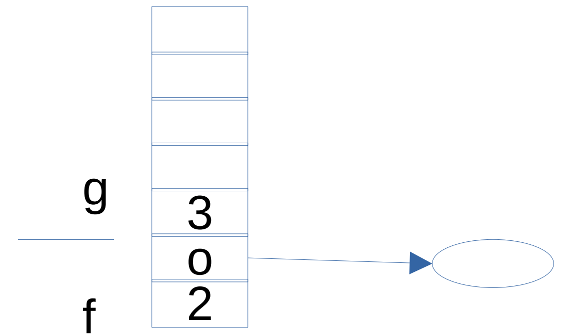
Precise Tracing

- Le garbage collecteur ne se déclenche qu’à des endroits (*safepoints*) précis du programme
  - Lors d’une allocation (new, etc)
  - En début (V8) ou en fin de fonction (OpenJDK)
  - Avant le goto d’une boucle (dont on ne connait pas le temps d’exécution)
- Le bytecode est typé donc on peut calculer par interpretation abstraite les positions des pointeurs sur la pile

Exemple

```
int f(Object o) {
  return 2 + g(o);
}

int g(Object o) {
  return 3;
  // Le GC se déclenche ici !
}
```



Tracing

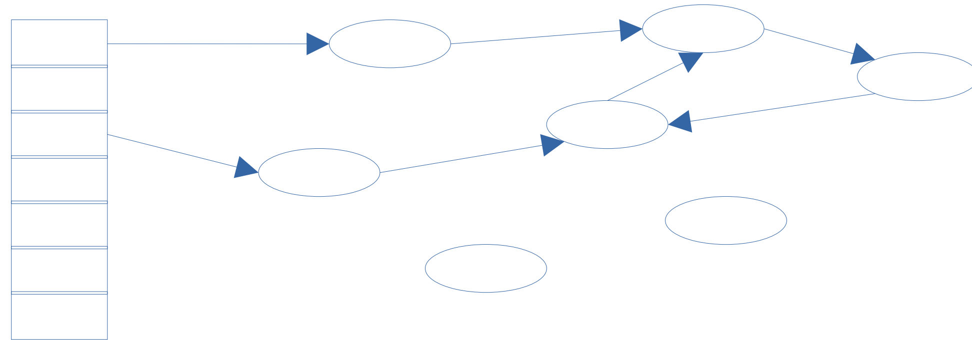
Le *3-color marking*

- Fait un parcours en largeur
- Au lieu d'avoir 2 couleurs white/black (marqué/pas marqué)
  - On ajoute une couleur (grey) qui indique qu'il faut parcourir les fils

3-color Marking

Virtuellement, on a 3 ensembles White / Grey / Black

- White tous les objets à marquer
- Grey: objets dont les fils ne sont pas encore marqués
- Black: objets dont les fils sont marqués

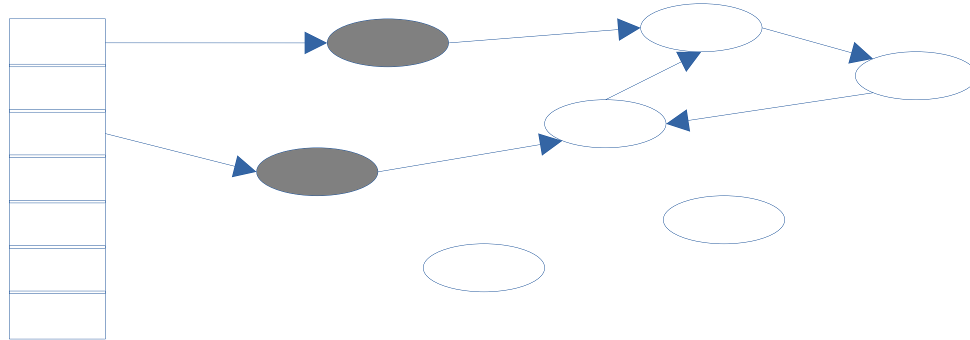


Au début tous les objets sont blancs

3-color Marking

Grey: objets dont les fils ne sont pas encore marqués

Black: objets dont les fils sont marqués

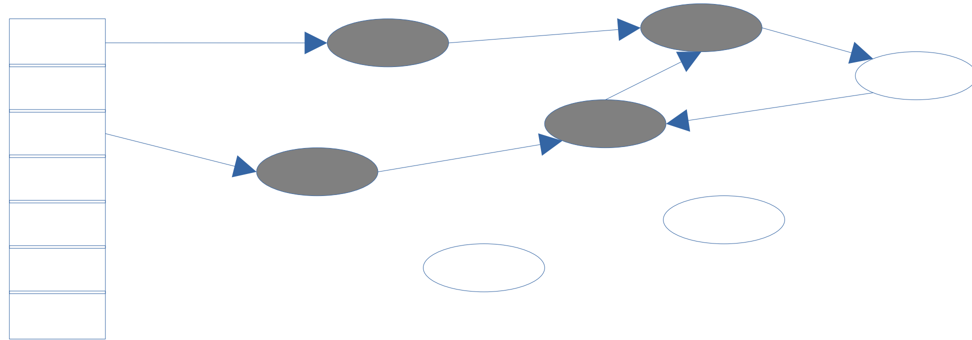


On marque les objets accessibles à partir de l’ensemble gris

3-color Marking

Grey: objets dont les fils ne sont pas encore marqués

Black: objets dont les fils sont marqués

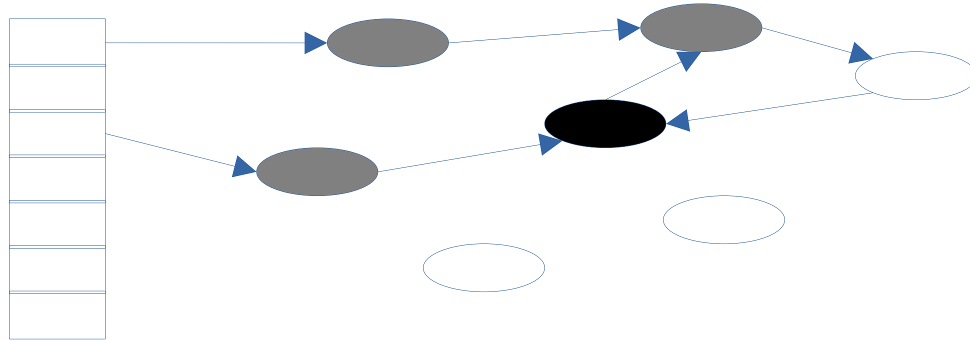


Si on arrive sur un objet gris ou noir, on s’arrète

3-color Marking

Grey: objets dont les fils ne sont pas encore marqués

Black: objets dont les fils sont marqués

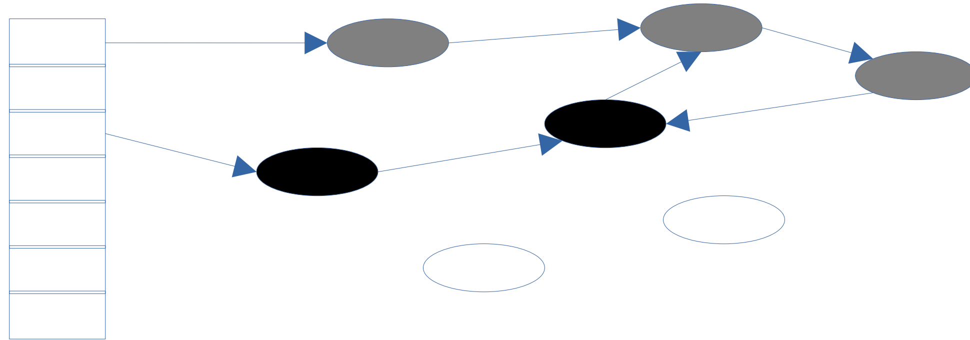


Les objets dont les fils sont parcourus sont noirs

3-color Marking

Grey: objets dont les fils ne sont pas encore marqués

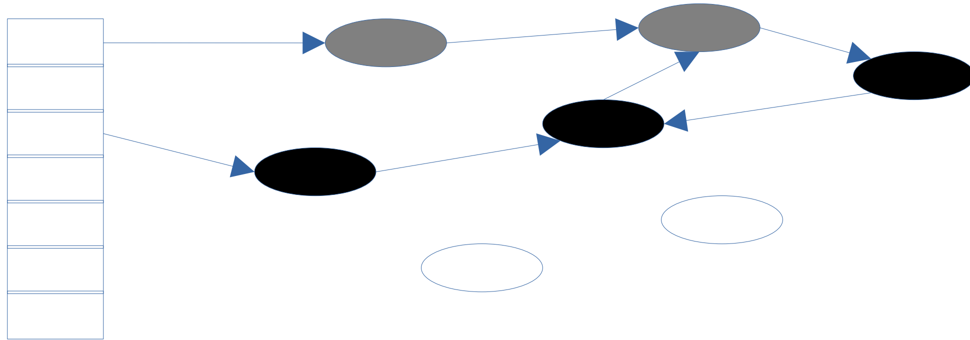
Black: objets dont les fils sont marqués



Les objets dont les fils sont parcourus sont noirs

3-color Marking

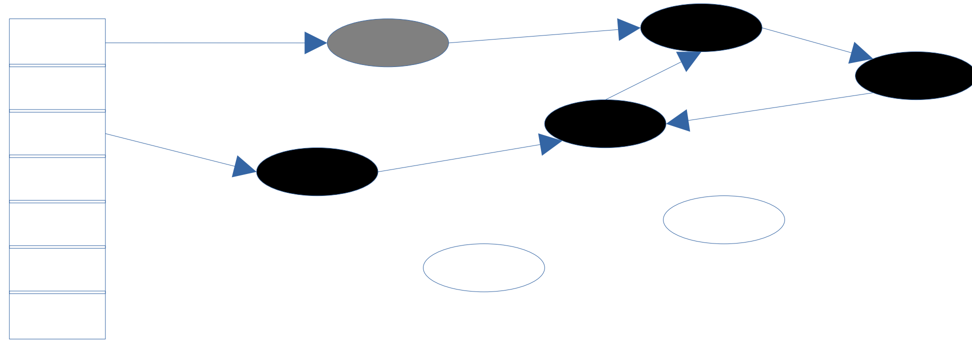
Black: objets dont les fils sont marqués



Les objets dont les fils sont parcourus sont noirs

3-color Marking

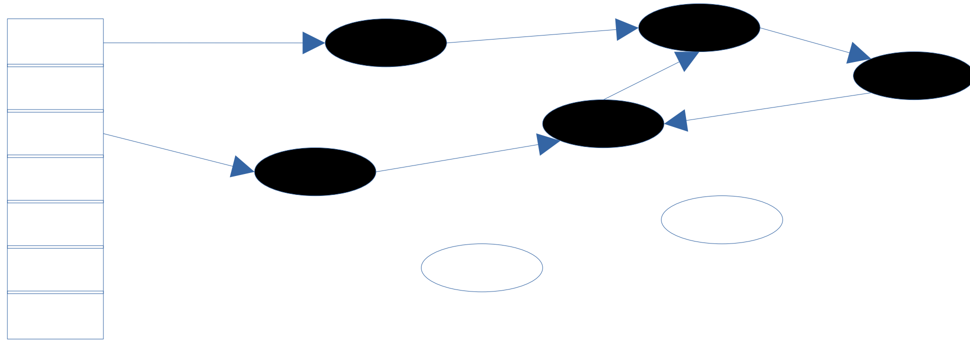
Black: objets dont les fils sont marqués



Les objets dont les fils sont parcourus sont noirs

3-color Marking

Black: objets dont les fils sont marqués

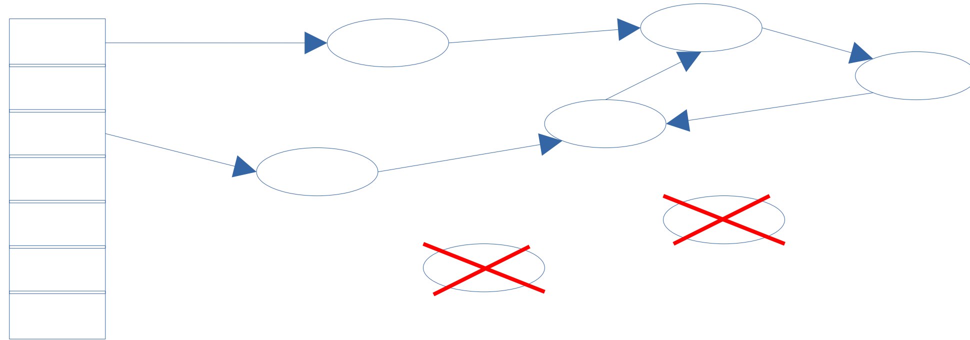


Les objets dont les fils sont parcourus sont noirs

3-color Marking

Une fois que l’on a fini

- On doit libèrer les objets white
- Et black devient le nouveau white



En pratique: marquer les objets

On marque les objets en gardant 2 bits dans le header des objets

- Plein d’écritures en mémoire (lent !)

Les objets en gris sont aussi stockés dans une table intermédiaire (side table) par thread

Concurrent Tracing

Il existe une version concurrente du *3-color marking*

- Tous les nouveaux objets créés prendant que le tracing s’éxécute sont black
- Si on mute un champ qui contient une référence, on passe l’objet en grey
  - Il faudra le re-parcourir dans le futur (re-trace)

Concurrent <==> s’exécute en même temps que le programme

Allocation / désallocation

Stratégie Sweep : Malloc / Free

Si on veut être compatible avec le C

On appelle malloc() lors de l'allocation

On appelle free() sur les objets pas marqués (white)

BoehmGC, les GCs de Python/Go utilise malloc/free

Mais

- Malloc est lent, il merge les morceaux de mémoire libérés par free()
- Problèmes : fragmentation en mémoire (localité) et concurrence => locks

Stratégie : Moving GC

Pour éviter la fragmentation, on bouge les objets en mémoire

Cela demande de <u>mettre à jour</u> les pointeurs dans les globales, sur la pile, et dans les champs

Allocation: *Pointer Bump*

- Rapide : on alloue les objets les uns derrière les autres
  - Pas de liste chainée de suivants comme malloc

- *Thread Local Buffer*: pour le multi-thread, chaque thread à sa propre zone d’allocation
  - Opérations atomiques que si une zone est pleine

Moving GC : Copier vs Compacter

Deux techniques

- Copier : deux zones, on copie les objets vivants dans une nouvelle zone, l’ancienne zone est considérée comme vide
- Compacter : on bouge les objets vers le début du tas. On a précalculer là où il devrait aller.

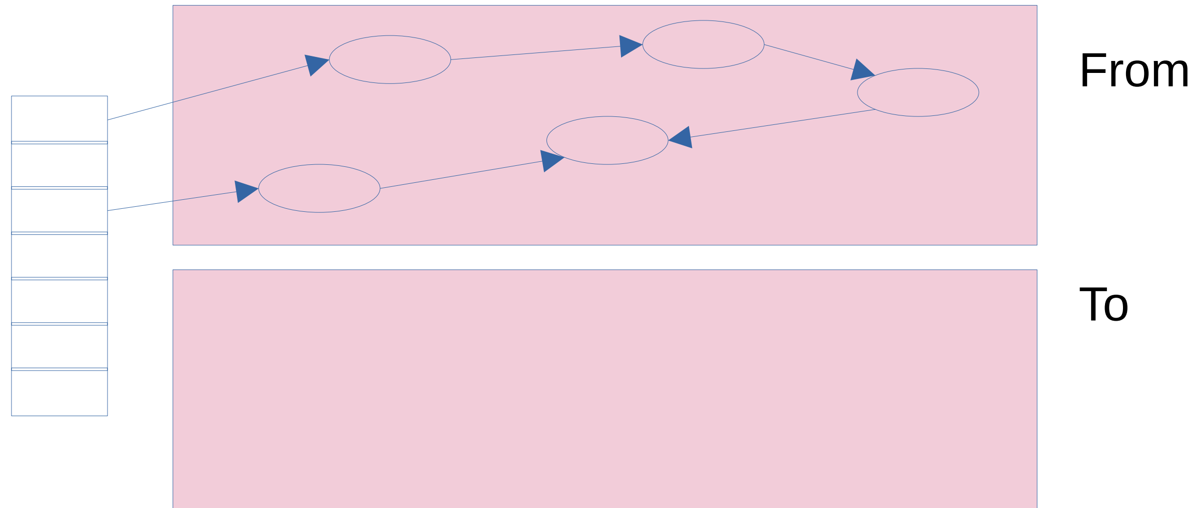
GC Semispace – Cheney (1970)

On sépare la pile en 2 zones (“from”, “to”)

- On alloue dans la zone “from”, une fois celle-ci pleine
- **Marking**: On marque les objets vivants
- **Copy**: Pour chaque référence sur la pile, on copie l’objets de “from” vers “to”. On écrit dans l’objet dans “from” la *forward address* vers “to”. Pour chaque champs, on fait la même chose s’il n’y a pas de *forward address*.
- On swap from et to.

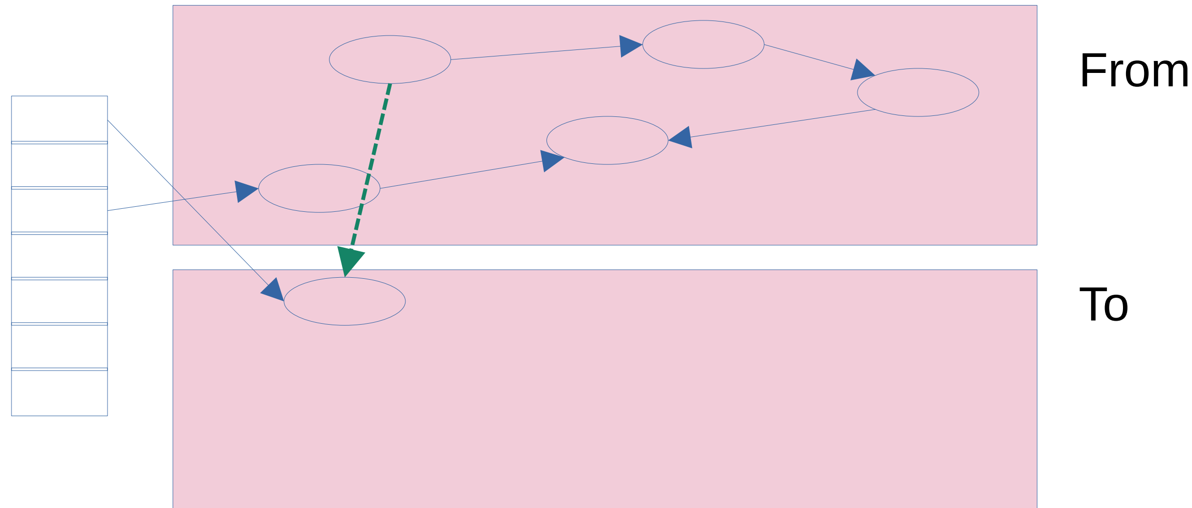
Algorithme de Cheney

On s’intéresse aux objets marqués



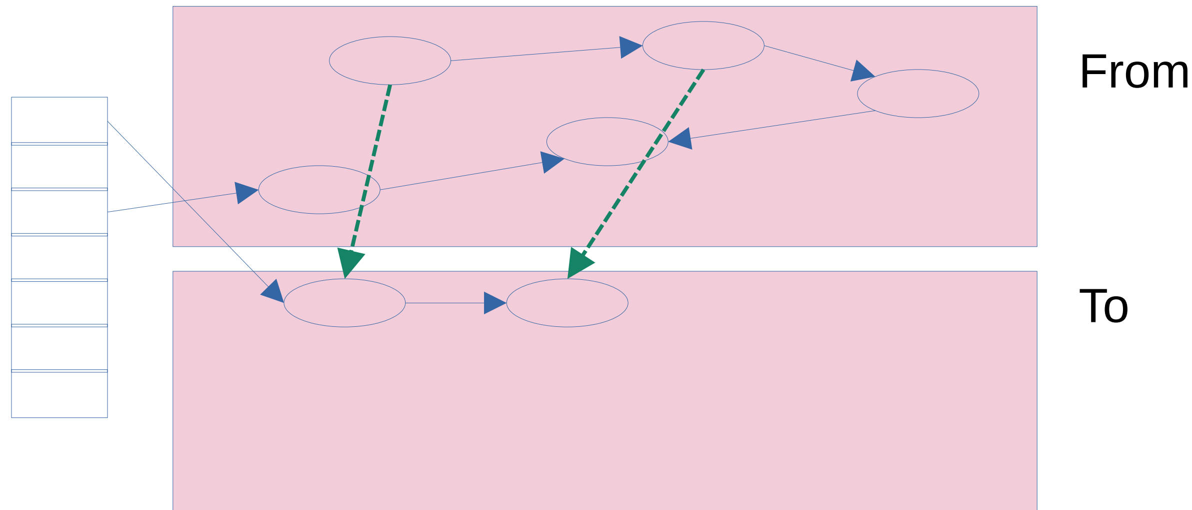
Algorithme de Cheney

On copie l’objet + *forward address*



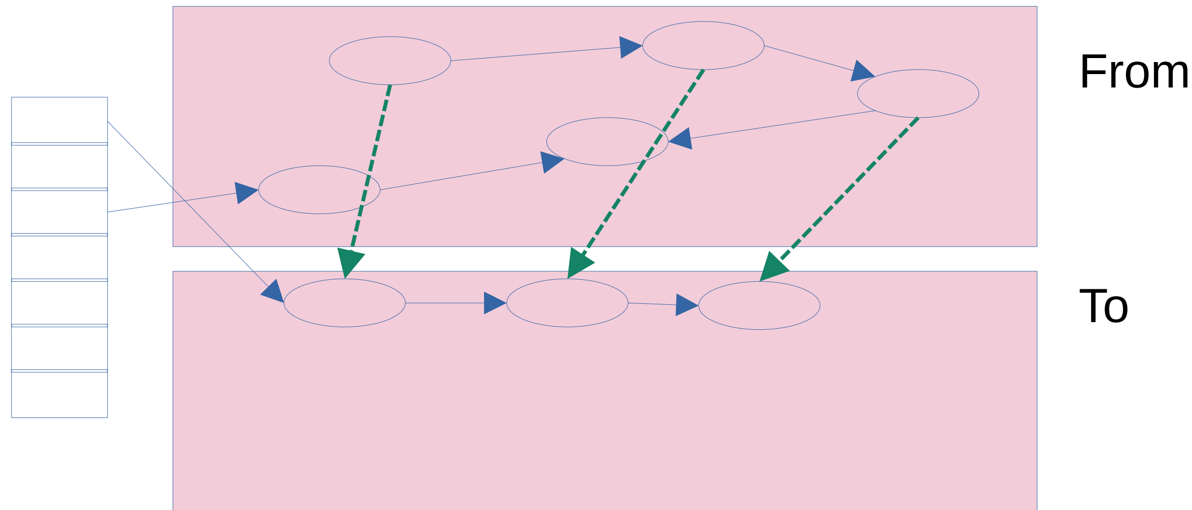
Algorithme de Cheney

On parcours les champs, copie l’objet + *forward address*



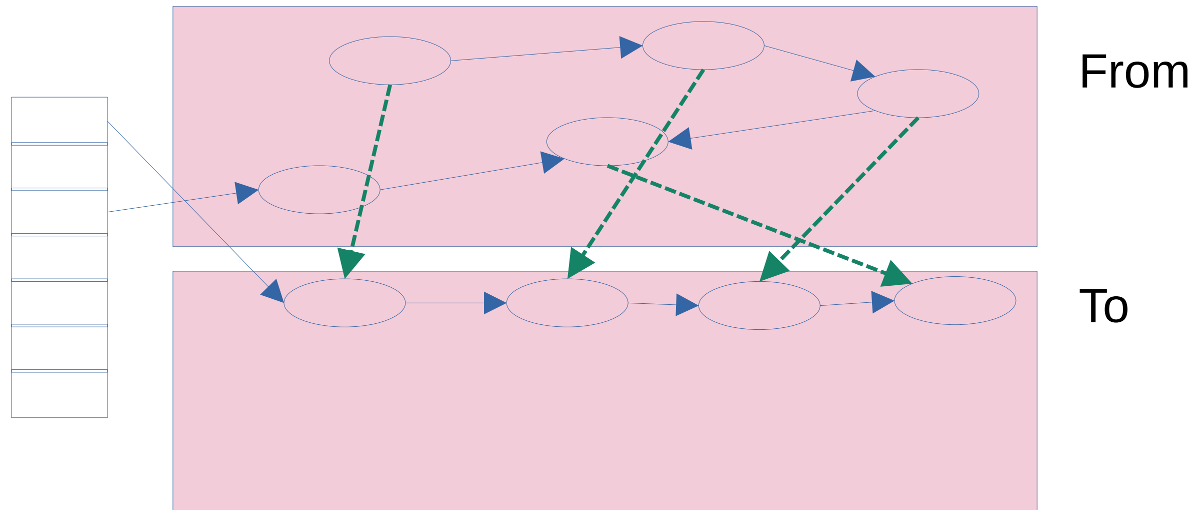
Algorithme de Cheney

On parcours les champs, copie l’objet + *forward address*



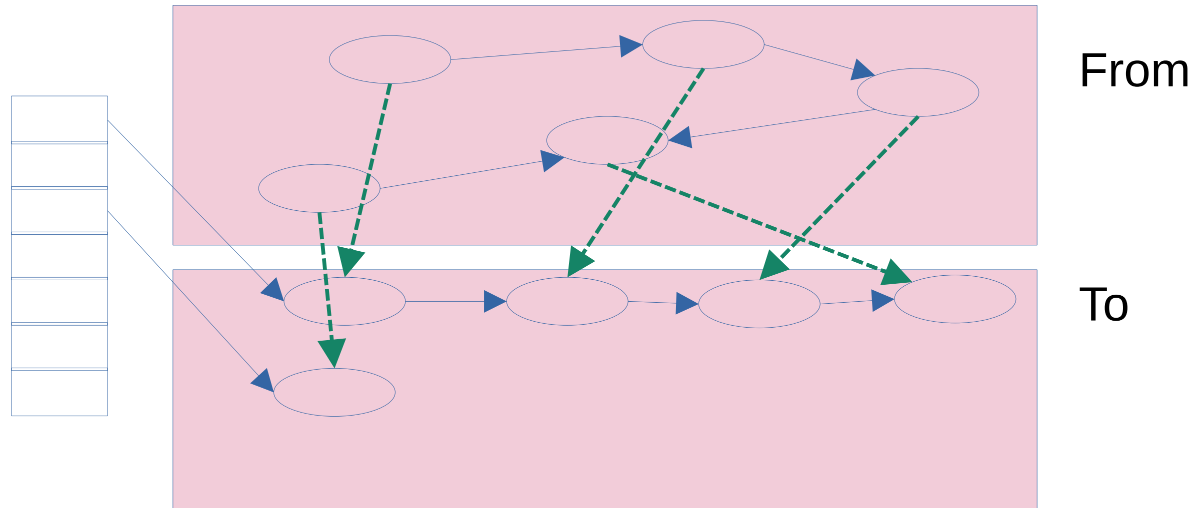
Algorithme de Cheney

On parcours les champs, copie l’objet + *forward address*



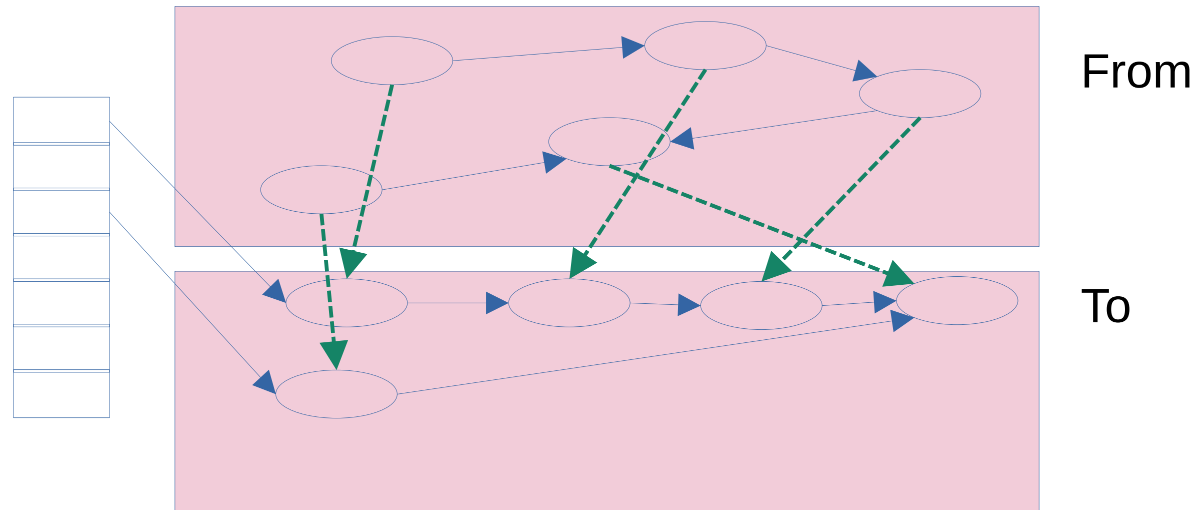
Algorithme de Cheney

On parcours la pile, copie l'objet + *forward address*



Algorithme de Cheney

On parcours les champs qui ont déjà une *forward address*



From

To

Et, “to” devient le nouveau “from”

GC Lisp-2 (1970)

Marking: On parcours la pile et on marque tous les objets vivants récursivement

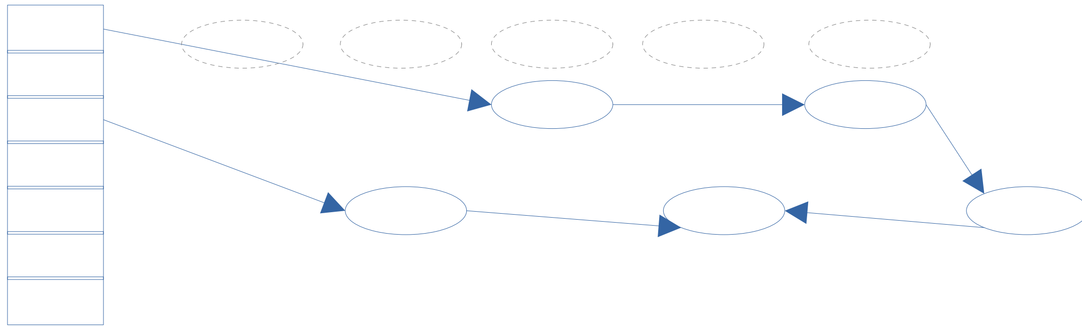
Planning: On calcule pour chaque objet vivant son nouvel emplacement (*forward address*). Il faut un espace dans l'entête de chaque objet (*Brooks pointer*)

Fixing: on parcours la pile et on met à jour les pointeurs

Moving: on copie les objets à leur nouvelle place (et on met à jour les champs)

Algorithme de Lisp2

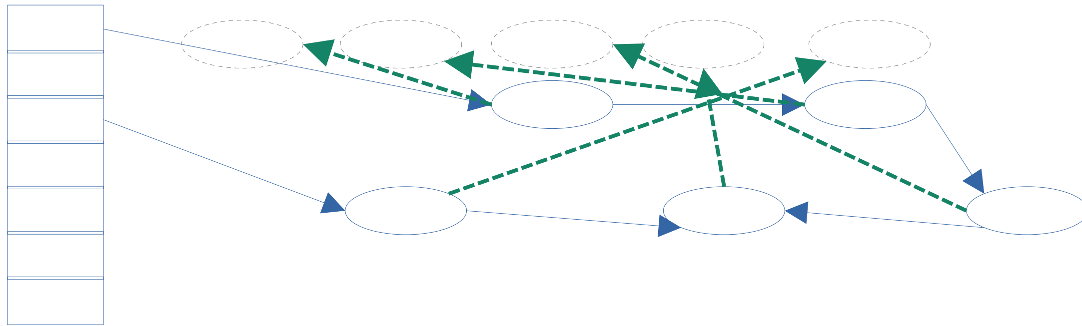
Marking: on marque les objets vivants



Chaque objet possède un espace pour stocker la *forward address*

Algorithme de Lisp2

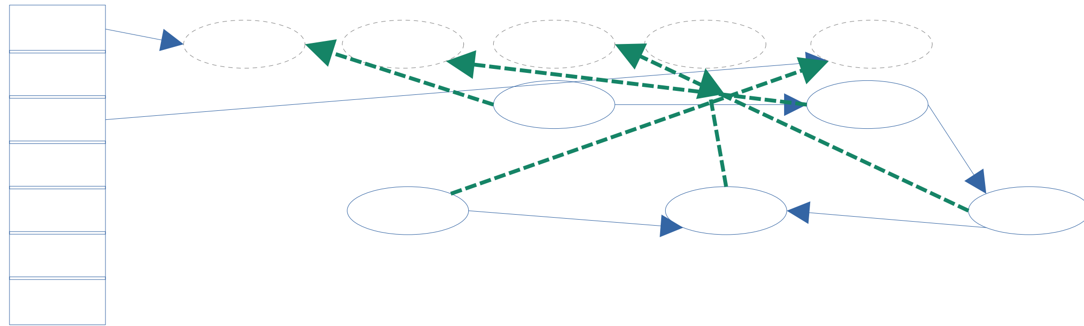
Panning : on calcule les *forward addresses*



La *forward address* est l’endroit où sera l’objet à la fin

Algorithme de Lisp2

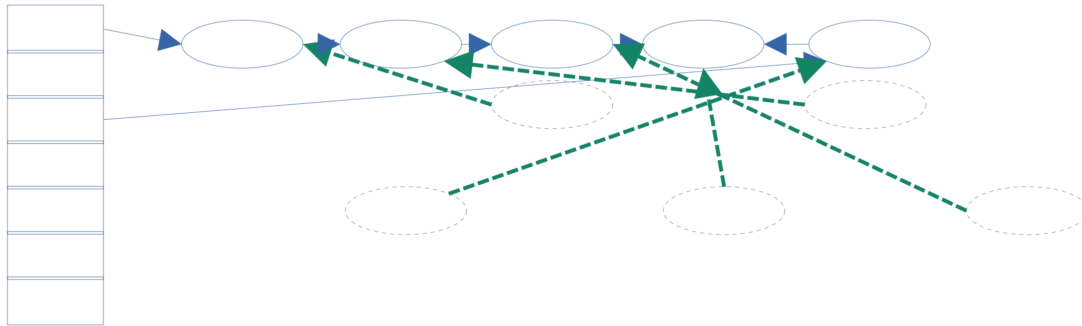
Fixing: on met à jour les pointeurs de la pile



Pour chaque pointeur, on suit la *forward address*

Algorithme de Lisp2

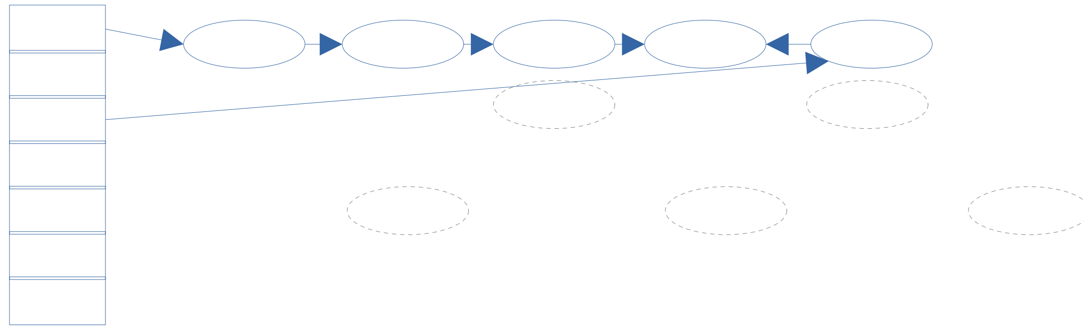
Fixing: on copie les objets en mettant à jour les fils



Pour chaque fils qui est une référence, on suit la *forward address*

Algorithme de Lisp2

La mémoire après le dernier objet est considérée comme libre



Compacter vs Copier

Les algorithmes historiques

- Compacter est lent
  - La complexité dépend de la taille du tas
- Copier demande plus de mémoire que nécessaire
  - La complexité dépend uniquement des objets vivants

=> Découper l’espace en différentes zones,
on marque/compacte/copie pas tout !

GC incrémentale
(+ hypothèse générationnelle)

Hypothèse générationnelle

La plupart des objets meurt jeunes

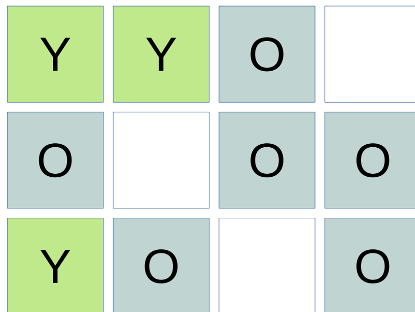
- On doit concentrer les efforts du GC dans les zones où l’on vient d’allouer des objets

- On attribut à chaque objet/zone un numéro de génération
  - Le GC privilégie les zones avec un petit numéro

Type de zones

Zones linéaires générationnelle (< 2008)

Zones matricielles générationnelle (> 2008)



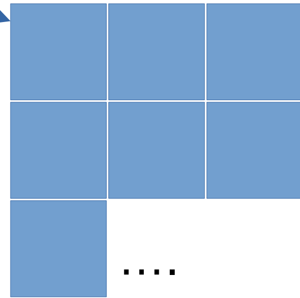
Marquage par zones

- On découpe le tas en différente zone !
  - Mais l'algo de marking nécessite de parcourir tous les objets
    - Donc marche pas par zone

- Idée: chaque zone maintient la liste des pointeurs sur une autre zone
  - On peut utiliser 1 bit mais trop gros grain
    - On utilise 1 bits pour une partie de la zone (une *card*) ~ 256 bytes
      - La structure de données est appelée CardMark

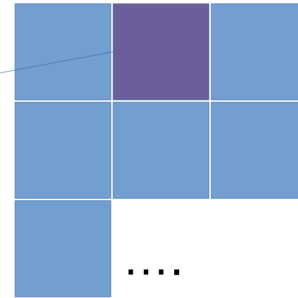
Card Marking

une *card*

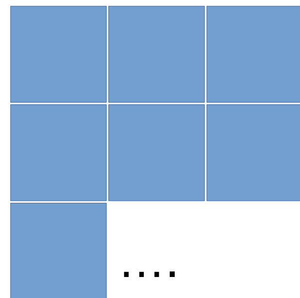


zone0

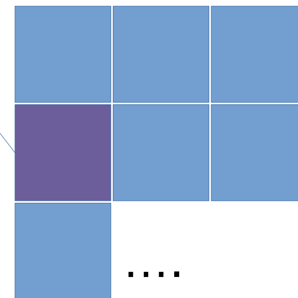
z1: 0100…
z2: 0000...
z3: 0001...



zone1



zone2



zone3

Une fois que la zone0 est évacuée, on reset les bits

Incremental Card Marking

- Comment mettre à jour la structure de CardMark ?
  - À chaque fois que l'on fait une écriture
    a.b = c
  - Dans la zone qui contient 'c', dans bitset de la zone de 'a', le bit de la *card* qui contient 'a' est allumé
    - On appel le code qui fait la mise à jour une ***GC WriteBarrier*** ou barrière en écriture pour le GC

Incremental Card Marking (2)

- Marking incrémentale des zones jeunes
  - On ne marque pas les objets en dehors des zones jeunes
  - On marque à partir des racines uniquement les pointeurs vers les zones jeunes
  - Et on parcours les objets contenus dans les *cards* qui pointent vers les zones jeunes

Collection Mineur / Majeur

- Collection mineur
  - on récupère uniquement les objets qui sont jeunes
- Collection majeur
  - on récupère tous les objets
- Collection full (*allocation failure* = mode panic)
  - on récupère tous les objets

Evacuation/Copie concurrente

Evacuation concurrente

On peut faire la copie des objets en même temps que le programme s’exécute

- Il faut plannifier où les objets doivent être copiés (calculer la *forward address*)
- Si le GC n’a pas encore déplacer un objet, le programme doit le faire :)
  - Il faut une ***GC Read Barrier*** (une barrière en lecture pour le GC)
    - Si le GC est en phase de copie des objets, avec a.b

      il faut regarder si l’objet “a” appartient à une zone copiée, faire la copie vers la *forward address* puis modifier la variable ‘a’.

Les différentes types de mémoire pour la machine virtuelle Java

Types de mémoire

Les piles (stack)

- Une pour chaque thread (pas d’allocation d’objets sur la pile)

Le Tas (heap)

- Mémoire gérer par le GC, découpé en zones

Mémoire Native

- Mémoire gérée par malloc (pour les IOs et appels systèmes) (MemorySegment et ByteBuffer)

Metaspace (ex: PermGen)

- Metadonnées associées aux classes (Klass, constant pool, bytecode, oopMap, profile/inlining cache ...)
- Code cache contient le code assembleur des méthodes JITée

Les *garbages collectors* de Java

Garbage Collectors de l’OpenJDK

L’OpenJDK possède 5 *garbage collectors* différents

- Peu de mémoire/core : Serial, Parallel, G1
- Beaucoup de mémoire/core : G1, Shenandoah, ZGC

Tous les GCs sont générationnels. L’allocation utilise des TLB (NUMA aware) + *bump pointer.*

Serial et Parallel

Serial et Parallel sont *stop the world*

- Minor: mark & copy, Major: mark & compact
- Serial GC utilise 1 thread, mémoire < 1 G
  - Ligne de commande, serverless lambda, IOT
- Parallel GC utilise toutes les threads, mémoire < 128 G
  - GC le plus rapide (throughput) mais avec des temps de pause (latence) qui peuvent être horribles
  - Application en batch

Garbage First (aka G1)

Le GC par défaut

- Minor/Major: mark & copy/compact
- memoire < 128 G
- On indique le temps de pause maximal
  (> 100ms)
- Le marking est concurrent, la copie est incrémental mais *stop the world*
  - Rapide mais peut louper des échéances

Shenandoah

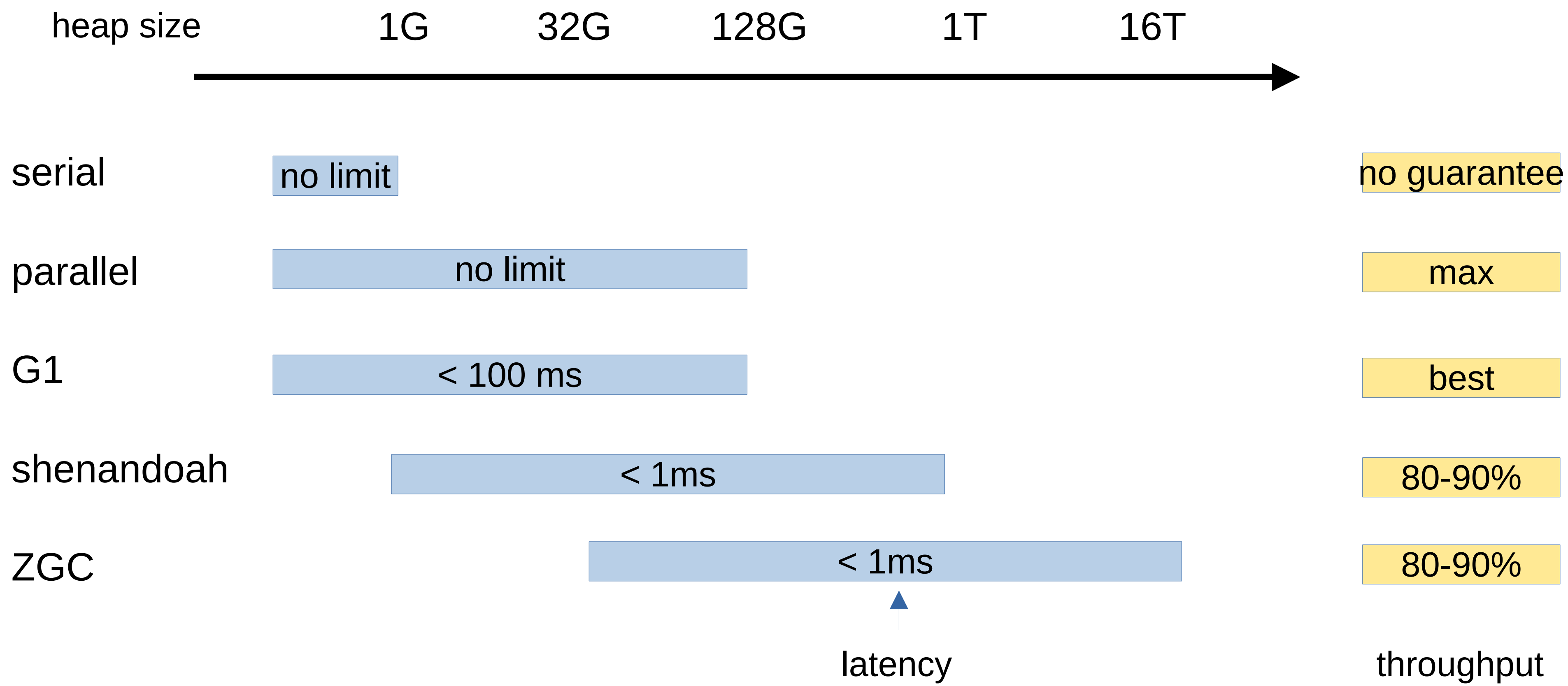
- GC developpé par RedHat puis Amazon
  - Minor/Major: mark & copy/compact
  - Préfère la latence au débit (throughput)
    - Mémoire < 1T
    - Temps de pause < 1~3 ms
  - Le marking et la copie sont concurrents
    - Le parcours des piles est concurrent aussi

ZGC

est un GC developpé par Oracle

- Minor, Major : mark & copy/compact
- Préfère la latence au débit (throughput)
  - Mémoire > 32G (pointeur sur 64 bits), < 16T
  - Temps de pause < 1 ms
- Le marking et la copie sont concurrents
  - Le parcours des piles est concurrent aussi

Java GCs



Les Références faibles (Weak Reference)

Références Faibles

Une référence faible est une référence qui n’est pas compter/parcouru par le GC

Le package java.lang.ref contient plusieurs classes (qui hérite de Reference) et qui permet de définir différentes références faibles

Comment créer une référence ?

Les références faibles possèdent un constructeur qui prend en paramètre une référence (forte)

```
var myObject = new MyObject();
var ref = new WeakRef<>(myObject);
ref.refersTo(myObject);  // true
myObject = null;
```

Plus tard

- `ref.get();  // renvoie null si myObject a été GC`
- `ref.refersTo(null);  // true si myObject a été GC`

ReferenceQueue

FIFO qui contient les références des objets qui ont été GC

```
var myObject = new MyObject();
var queue = new ReferenceQueue<MyObject>();
var ref = new WeakRef<>(myObject, queue);
myObject = null;
```

- Lorsque myObject est GC
  - queue.poll() // renvoie ref ou null
  - queue.remove() // block et renvoie ref
  - queue.remove(timeout); // block et renvoie ref, ou null si timeout

Types de références faibles

Il existe 3 types de références faibles

- PhantomReference & WeakReference
  - Le référent devient null lorsqu’il est GC
- SoftReference
  - Si le GC a besoin de mémoire, la référence devient faible
    - Le référent devient null lorsqu’il est GC

Finalization (Legacy)

Finalization

- java.lang.Object possède une méthode
  - protected void finalize()

- Appelée par un thread système (après un GC) lorsqu'une instance devrait mourir
  - Peut faire revivre l'objet une fois
  - Rend l'objet pas thread safe (aahhh)

Finalization

Historiquement, utiliser pour appeler close()

```
class MyObject implements java.io.Closeable {
  private final InputStream input;   // doit être final ou volatile !
  ...

  protected void finalize() {
    input.close();
  }

  public void close() {
    input.close();
  }
}
```

Finalization (deprecated, java 9)

En réalité

- finalize() est appelée que quand on manque de mémoire
  - close() est jamais fait si on manque pas de mémoire (aaaahh)
- Pas efficace, on fait revivre l’objet pour pouvoir appeler close()
  - La VM doit lancer une thread (ou plusieurs !)

Object.finalize()

```
@Deprecated(since="9",
            forRemoval=true)
protected void finalize()
                 throws Throwable
```

Deprecated, for removal: This API element is subject to removal in a future version.
Finalization is deprecated and subject to removal in a future release. The use of finalization can lead to problems with security, performance, and reliability. See JEP 421 for discussion and alternatives.

Subclasses that override `finalize` to perform cleanup should use alternative cleanup mechanisms and remove the `finalize` method. Use `Cleaner` and `PhantomReference` as safer ways to release resources when an object becomes unreachable. Alternatively, add a `close` method to explicitly release resources, and implement `AutoCloseable` to enable use of the `try`-with-resources statement.

This method will remain in place until finalizers have been removed from most existing code.

Called by the garbage collector on an object when garbage collection determines that there are no more references to the object. A subclass overrides the `finalize` method to dispose of system resources or to perform other cleanup.

When running in a Java virtual machine in which finalization has been disabled or removed, the garbage collector will never call `finalize()`. In a Java virtual machine in which finalization is enabled, the garbage collector might call `finalize` only after an indefinite delay.

Finalization (for removal, java 18)

Les classes du JDK n’implantent plus finalize()

- Marche encore pour les autres classes

Flag pour éviter la finalization (--finalization=disabled)

Bientôt

- Flag pour que la finalization marche (--finalization=enabled)
- Puis la finalization marchera plus !
  - finalize() va être supprimer de java.lang.Object

PhantomReference vs WeakReference

Le référent

- d’une phantom ref devient null avant la finalization
- d’une weak ref devient null après la finalization

Pas de différence, si pas de finalization !