

# Programming Language (9)

## Garbage Collection

田浦

# Contents

Criteria of evaluating GCs (RC vs. traversing)

Two traversing GCs (mark&sweep vs. copying)

Memory allocation cost of traversing GCs (mark-cons ratio)

Generational GC

Incremental GC

Topics on Mark&Sweep GCs

- Free Area Management

- Improving mark&sweep GCs

  - Separated Mark Bits

  - Lazy Sweep

Conservative GC

# Contents

Criteria of evaluating GCs (RC vs. traversing)

Two traversing GCs (mark&sweep vs. copying)

Memory allocation cost of traversing GCs (mark-cons ratio)

Generational GC

Incremental GC

Topics on Mark&Sweep GCs

- Free Area Management

- Improving mark&sweep GCs

  - Separated Mark Bits

  - Lazy Sweep

Conservative GC

# Evaluating GCs

1. **preciseness:**
  - ▶ garbage that can be collected
2. **memory allocation cost:**
  - ▶ the work (including GC) required to allocate memory
3. **pause time:**
  - ▶ the (worst case) time the mutator has to (temporarily) suspend for GC to function
4. **mutator overhead:**
  - ▶ the overhead imposed on the mutator for GC to function



## Criteria #2: memory allocation cost

- ▶ difficult to say in a few words (more details ahead)

## Criteria #2: memory allocation cost

- ▶ difficult to say in a few words (more details ahead)
- ▶ traversing GC:
  - ▶ *the cost is determined by the ratio “reachable objects” / “unreachable (reclaimed) objects” (later)*
  - ▶ totally depending on apps and memory size, it can be anywhere from the minimum to infinity
  - ▶ an advanced technique: **generational GC**

## Criteria #2: memory allocation cost

- ▶ difficult to say in a few words (more details ahead)
- ▶ traversing GC:
  - ▶ *the cost is determined by the ratio “reachable objects” / “unreachable (reclaimed) objects” (later)*
  - ▶ totally depending on apps and memory size, it can be anywhere from the minimum to infinity
  - ▶ an advanced technique: **generational GC**
- ▶ **reference counting**:
  - ▶ the cost of reclaiming an object once its RC drop to zero is small and constant
  - ▶ it is constant even if memory is scarce (good)



## Criteria #3: pause time

- ▶ reference counting < traversing GC (reference counting is better)

## Criteria #3: pause time

- ▶ reference counting < traversing GC (reference counting is better)
- ▶ traversing GC:
  - ▶ traverse *all* live objects, *en masse*, and reclaim *all* unreachable objects, *en masse*
  - ▶ do a whole bunch of work and get a whole bunch of free blocks

## Criteria #3: pause time

- ▶ reference counting < traversing GC (reference counting is better)
- ▶ traversing GC:
  - ▶ traverse *all* live objects, *en masse*, and reclaim *all* unreachable objects, *en masse*
  - ▶ do a whole bunch of work and get a whole bunch of free blocks
  - ▶ why so? troubled if the mutator runs (= changes the graph of objects) during traversing
    - ▶ a solution: [incremental GC](#)
    - ▶ generational GCs mitigate it too

## Criteria #3: pause time

- ▶ reference counting < traversing GC (reference counting is better)
- ▶ traversing GC:
  - ▶ traverse *all* live objects, *en masse*, and reclaim *all* unreachable objects, *en masse*
  - ▶ do a whole bunch of work and get a whole bunch of free blocks
  - ▶ why so? troubled if the mutator runs (= changes the graph of objects) during traversing
    - ▶ a solution: incremental GC
    - ▶ generational GCs mitigate it too
- ▶ reference counting:
  - ▶ when an object's RC drops to zero (as a result of mutator's action), it can be reclaimed immediately
  - ▶ reclaim garbage as they arise

## Criteria #4: mutator overhead

- ▶ traversing < reference counting (traversing GC is better)
- ▶ reference counting has a large overhead for updating RCs

```
1 object * p, * q;  
2 p = q;
```

will do:

```
1 if (p) p->rc--;  
2 if (q) q->rc++;  
3 p = q;
```

Moreover,

- ▶ what about multithreaded programs?
- ▶ what if the counter overflows (how to check it)?
- ▶ techniques: [deferred reference counting](#), [sticky reference counting](#), [1 bit reference counting](#)
- ▶ remark: some traversing GCs (e.g., generational and incremental) add overhead to pointer updates too

# Summary

	traversing	reference counting
preciseness	+	—
allocation cost	? (*)	+
pause time	— (†)	+
mutator overhead	+ (‡)	—

- (\*) depends on size of reachable graph and memory;  
generational garbage collector helps
- (†) incremental garbage collector helps
- (‡) both generational and incremental garbage collectors  
impose some mutator overheads

# Contents

Criteria of evaluating GCs (RC vs. traversing)

Two traversing GCs (mark&sweep vs. copying)

Memory allocation cost of traversing GCs (mark-cons ratio)

Generational GC

Incremental GC

Topics on Mark&Sweep GCs

- Free Area Management

- Improving mark&sweep GCs

  - Separated Mark Bits

  - Lazy Sweep

Conservative GC

## mark&sweep GC vs. copying GC

they differ in what to do on reachable objects

- ▶ mark&sweep GC: mark them as “visited”



## mark&sweep GC vs. copying GC

they differ in what to do on reachable objects

- ▶ mark&sweep GC: mark them as “visited”
- ▶ copying GC: copy them into a distinct (contiguous) region

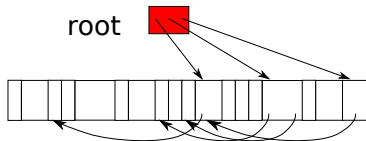
# Mark&sweep GC

## 1. mark-phase:

- ▶ traverses objects from the root, *marking* objects it encounters
- ▶ maintains *mark stack (not shown in the figure)*, marked objects whose children may have not been marked (= light gray objects)

## 2. sweep phase:

- ▶ reclaims all memory blocks that were not visited
- ▶ free memory blocks are not contiguous, so must be managed by an appropriate data structure (*free lists*)



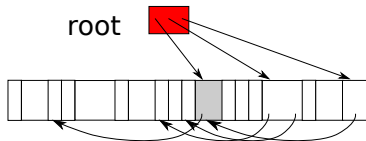
# Mark&sweep GC

## 1. mark-phase:

- ▶ traverses objects from the root, *marking* objects it encounters
- ▶ maintains *mark stack* (*not shown in the figure*), marked objects whose children may have not been marked (= light gray objects)

## 2. sweep phase:

- ▶ reclaims all memory blocks that were not visited
- ▶ free memory blocks are not contiguous, so must be managed by an appropriate data structure (*free lists*)



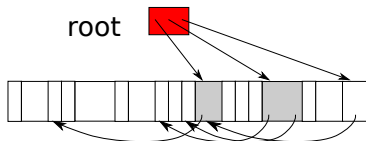
# Mark&sweep GC

## 1. mark-phase:

- ▶ traverses objects from the root, *marking* objects it encounters
- ▶ maintains *mark stack* (*not shown in the figure*), marked objects whose children may have not been marked (= light gray objects)

## 2. sweep phase:

- ▶ reclaims all memory blocks that were not visited
- ▶ free memory blocks are not contiguous, so must be managed by an appropriate data structure (*free lists*)



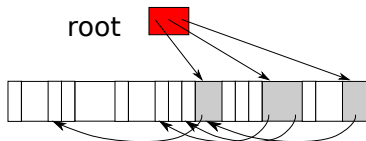
# Mark&sweep GC

## 1. mark-phase:

- ▶ traverses objects from the root, *marking* objects it encounters
- ▶ maintains *mark stack (not shown in the figure)*, marked objects whose children may have not been marked (= light gray objects)

## 2. sweep phase:

- ▶ reclaims all memory blocks that were not visited
- ▶ free memory blocks are not contiguous, so must be managed by an appropriate data structure (*free lists*)



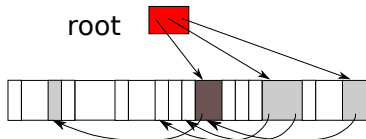
# Mark&sweep GC

## 1. mark-phase:

- ▶ traverses objects from the root, *marking* objects it encounters
- ▶ maintains *mark stack* (*not shown in the figure*), marked objects whose children may have not been marked (= light gray objects)

## 2. sweep phase:

- ▶ reclaims all memory blocks that were not visited
- ▶ free memory blocks are not contiguous, so must be managed by an appropriate data structure (*free lists*)



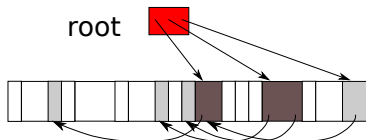
# Mark&sweep GC

## 1. mark-phase:

- ▶ traverses objects from the root, *marking* objects it encounters
- ▶ maintains *mark stack* (*not shown in the figure*), marked objects whose children may have not been marked (= light gray objects)

## 2. sweep phase:

- ▶ reclaims all memory blocks that were not visited
- ▶ free memory blocks are not contiguous, so must be managed by an appropriate data structure (*free lists*)



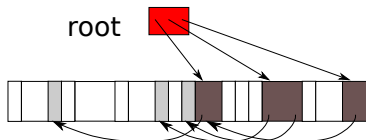
# Mark&sweep GC

## 1. mark-phase:

- ▶ traverses objects from the root, *marking* objects it encounters
- ▶ maintains *mark stack (not shown in the figure)*, marked objects whose children may have not been marked (= light gray objects)

## 2. sweep phase:

- ▶ reclaims all memory blocks that were not visited
- ▶ free memory blocks are not contiguous, so must be managed by an appropriate data structure (*free lists*)





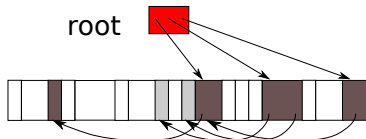
# Mark&sweep GC

## 1. mark-phase:

- ▶ traverses objects from the root, *marking* objects it encounters
- ▶ maintains *mark stack* (*not shown in the figure*), marked objects whose children may have not been marked (= light gray objects)

## 2. sweep phase:

- ▶ reclaims all memory blocks that were not visited
- ▶ free memory blocks are not contiguous, so must be managed by an appropriate data structure (*free lists*)



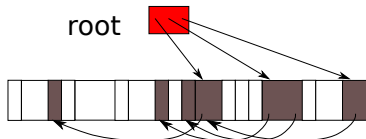
# Mark&sweep GC

## 1. mark-phase:

- ▶ traverses objects from the root, *marking* objects it encounters
- ▶ maintains *mark stack* (*not shown in the figure*), marked objects whose children may have not been marked (= light gray objects)

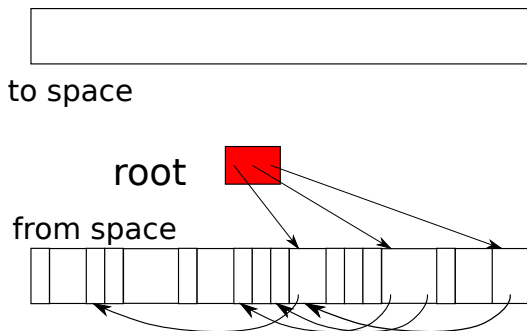
## 2. sweep phase:

- ▶ reclaims all memory blocks that were not visited
- ▶ free memory blocks are not contiguous, so must be managed by an appropriate data structure (*free lists*)



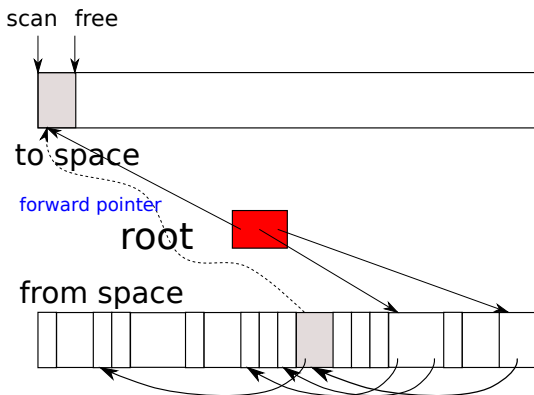
# Copying GC

- ▶ in essence,  $\approx$  copying a graph ( $\approx$  serialization)
  - ▶ the same pointers must remain the same after the copy
- ▶ **semi-space GC** (copy all objects reachable from the root into another space)



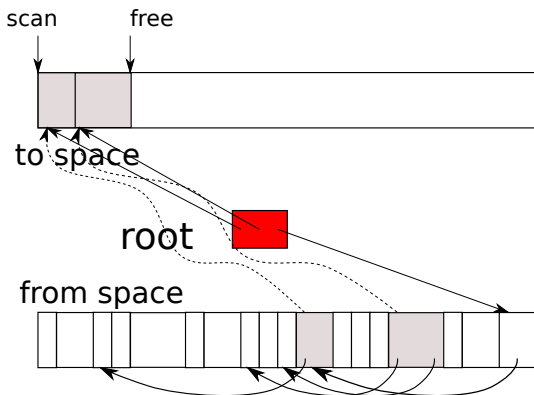
# Copying GC

- ▶ in essence,  $\approx$  copying a graph ( $\approx$  serialization)
  - ▶ the same pointers must remain the same after the copy
- ▶ **semi-space GC** (copy all objects reachable from the root into another space)



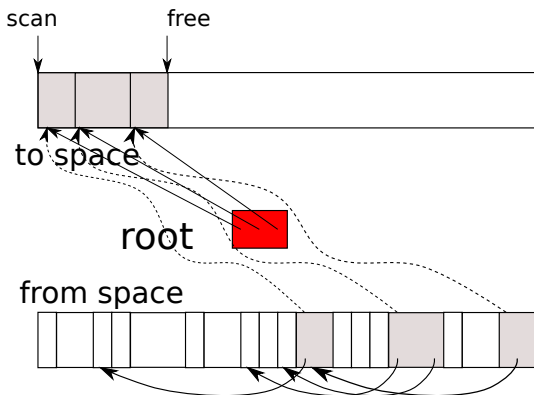
# Copying GC

- ▶ in essence,  $\approx$  copying a graph ( $\approx$  serialization)
  - ▶ the same pointers must remain the same after the copy
- ▶ **semi-space GC** (copy all objects reachable from the root into another space)



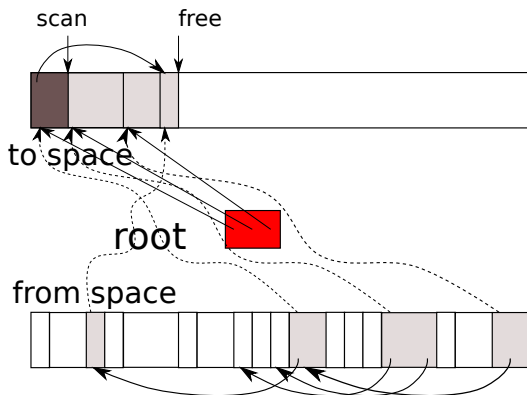
# Copying GC

- ▶ in essence,  $\approx$  copying a graph ( $\approx$  serialization)
  - ▶ the same pointers must remain the same after the copy
- ▶ **semi-space GC** (copy all objects reachable from the root into another space)



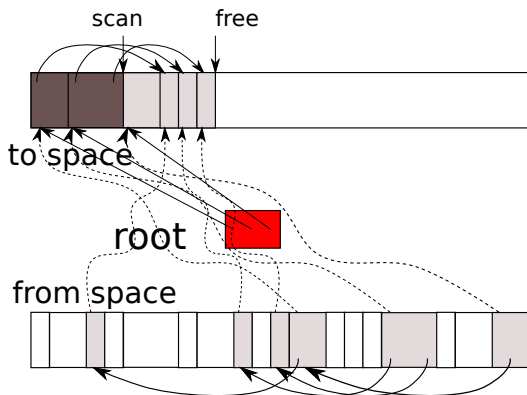
# Copying GC

- ▶ in essence,  $\approx$  copying a graph ( $\approx$  serialization)
  - ▶ the same pointers must remain the same after the copy
- ▶ **semi-space GC** (copy all objects reachable from the root into another space)



# Copying GC

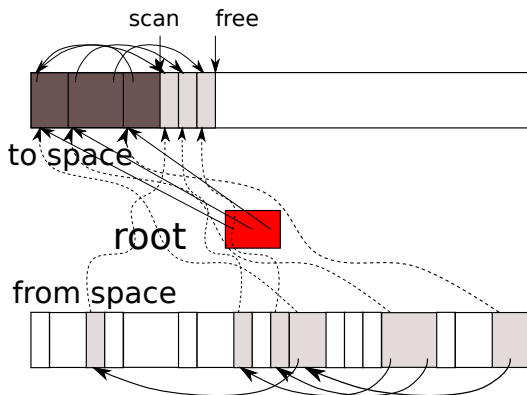
- ▶ in essence,  $\approx$  copying a graph ( $\approx$  serialization)
  - ▶ the same pointers must remain the same after the copy
- ▶ **semi-space GC** (copy all objects reachable from the root into another space)





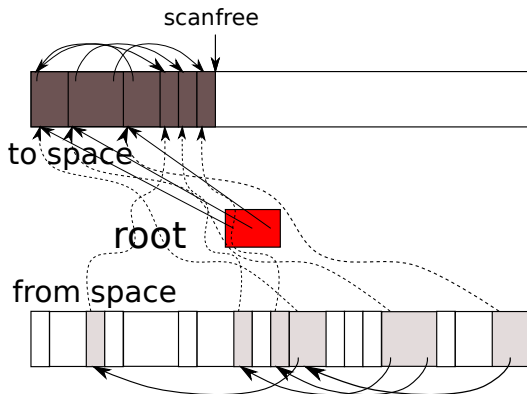
# Copying GC

- ▶ in essence,  $\approx$  copying a graph ( $\approx$  serialization)
  - ▶ the same pointers must remain the same after the copy
- ▶ **semi-space GC** (copy all objects reachable from the root into another space)



# Copying GC

- ▶ in essence,  $\approx$  copying a graph ( $\approx$  serialization)
  - ▶ the same pointers must remain the same after the copy
- ▶ **semi-space GC** (copy all objects reachable from the root into another space)

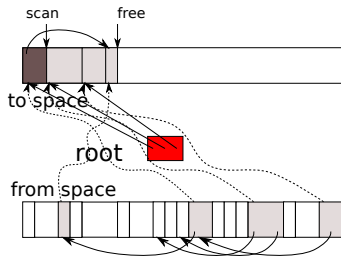


# Copying GC: algorithm

```
1 void *free, *scan;
2 copy_gc() {
3     free = scan = to_space;
4     redirect_ptrs(root);
5     while (scan < free) {
6         redirect_ptrs(scan);
7         scan += the size of object scan points to;
8     }
9 }
10 redirect_ptrs(void * o) {
11     for (p ∈ pointers in o) {
12         if (p has been copied) {
13             p = p's forward pointer;
14         } else {
15             copy p to free;
16             p = free;
17             p's forward pointer = free;
18             free += the size of object p points to;
19         }
20     }
21 }
```

invariant

- ▶  $p < \text{scan} \Rightarrow p$  has been reached; so has its direct children
- ▶  $p < \text{free} \Rightarrow p$  has been reached; but its children may not

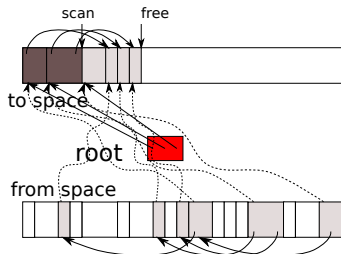


# Copying GC: algorithm

```
1 void *free, *scan;
2 copy_gc() {
3     free = scan = to_space;
4     redirect_ptrs(root);
5     while (scan < free) {
6         redirect_ptrs(scan);
7         scan += the size of object scan points to;
8     }
9 }
10 redirect_ptrs(void * o) {
11     for (p ∈ pointers in o) {
12         if (p has been copied) {
13             p = p's forward pointer;
14         } else {
15             copy p to free;
16             p = free;
17             p's forward pointer = free;
18             free += the size of object p points to;
19         }
20     }
21 }
```

invariant

- ▶  $p < \text{scan} \Rightarrow p$  has been reached; so has its direct children
- ▶  $p < \text{free} \Rightarrow p$  has been reached; but its children may not



# Mark&sweep vs. copying GC

- ▶ copying GC pros:
  - ▶ live objects occupy a contiguous region after a GC
  - ▶ → the free region becomes contiguous too
  - ▶ → the overhead for memory allocation is small (no need to “search” the free region)
- ▶ copying GC cons:
  - ▶ copy is expensive, obviously
  - ▶ the free region must be reserved to accommodate objects copied (low memory utilization)
    - ▶ must ensure “size of objects that may be copied”  $\leq$  “size of the region to copy them into”
    - ▶ → “from space” = “to space”
  - ▶ pointers must be “precisely” distinguished from non-pointers (ambiguous pointers are not allowed)
    - ▶ pointers are updated to the destinations of copies
    - ▶ a disaster occurs if you update non-pointers

# Contents

Criteria of evaluating GCs (RC vs. traversing)

Two traversing GCs (mark&sweep vs. copying)

Memory allocation cost of traversing GCs (mark-cons ratio)

Generational GC

Incremental GC

Topics on Mark&Sweep GCs

- Free Area Management

- Improving mark&sweep GCs

  - Separated Mark Bits

  - Lazy Sweep

Conservative GC

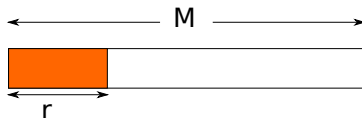
# Memory allocation cost of traversing GCs

- ▶ let's quantify the cost of allocating a byte including GC's work
- ▶ assume:
  - ▶ heap size (size of a semi-space in case of copying GC) =  $M$
  - ▶ reached objects =  $r$
  - ▶ assume for the sake of argument it's *always*  $r$



# Memory allocation cost of traversing GCs

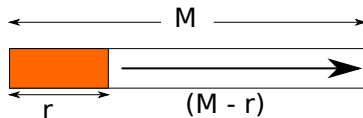
- ▶ let's quantify the cost of allocating a byte including GC's work
- ▶ assume:
  - ▶ heap size (size of a semi-space in case of copying GC) =  $M$
  - ▶ reached objects =  $r$
  - ▶ assume for the sake of argument it's *always*  $r$





# Memory allocation cost of traversing GCs

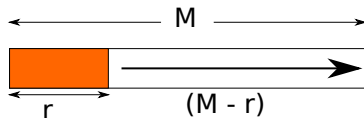
- ▶ let's quantify the cost of allocating a byte including GC's work
- ▶ assume:
  - ▶ heap size (size of a semi-space in case of copying GC) =  $M$
  - ▶ reached objects =  $r$
  - ▶ assume for the sake of argument it's *always*  $r$
- ▶ behavior at equilibrium: the program repeats:
  1. a GC occurs  $\rightarrow$  scan (or copy)  $r$  bytes, to make a free space of  $(M - r)$  bytes
  2. allocate  $(M - r)$  bytes without triggering a GC



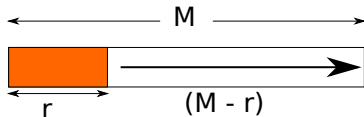
# Memory allocation cost of traversing GCs

- ▶ let's quantify the cost of allocating a byte including GC's work
- ▶ assume:
  - ▶ heap size (size of a semi-space in case of copying GC) =  $M$
  - ▶ reached objects =  $r$
  - ▶ assume for the sake of argument it's *always*  $r$
- ▶ behavior at equilibrium: the program repeats:
  1. a GC occurs  $\rightarrow$  scan (or copy)  $r$  bytes, to make a free space of  $(M - r)$  bytes
  2. allocate  $(M - r)$  bytes without triggering a GC
- ▶ a key observation

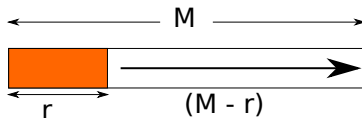
*the time (cost) of a single GC is roughly proportional to the amount of reached objects (i.e.,  $\propto r$ )*



# Memory allocation cost of traversing GCs

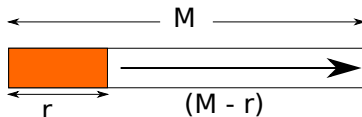


# Memory allocation cost of traversing GCs



$\therefore$  the cost of allocating a byte

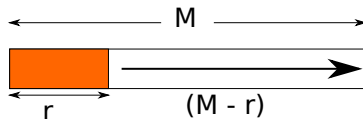
# Memory allocation cost of traversing GCs



$$\begin{aligned} \therefore & \quad \text{the cost of allocating a byte} \\ = & \quad \alpha + \frac{\text{the amount of time spent on a GC}}{\text{the amount of space reclaimed by a GC}} \end{aligned}$$

- ▶  $\alpha$  : a constant cost needed anyway, even if you don't need to reclaim memory at all

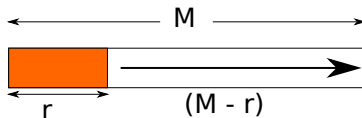
# Memory allocation cost of traversing GCs



$$\begin{aligned} \therefore & \text{the cost of allocating a byte} \\ = & \alpha + \frac{\text{the amount of time spent on a GC}}{\text{the amount of space reclaimed by a GC}} \\ = & \alpha + \beta \frac{\text{the amount of space visited by a GC}}{\text{the amount of space reclaimed by a GC}} \end{aligned}$$

- $\alpha$  : a constant cost needed anyway, even if you don't need to reclaim memory at all

# Memory allocation cost of traversing GCs



$$\begin{aligned} \therefore & \quad \text{the cost of allocating a byte} \\ = & \quad \alpha + \frac{\text{the amount of time spent on a GC}}{\text{the amount of space reclaimed by a GC}} \\ = & \quad \alpha + \beta \frac{\text{the amount of space visited by a GC}}{\text{the amount of space reclaimed by a GC}} \\ = & \quad \alpha + \beta \frac{r}{M - r} \end{aligned}$$

- ▶  $\alpha$  : a constant cost needed anyway, even if you don't need to reclaim memory at all
- ▶  $\beta$  : an average cost to examine a single byte
  - ▶ copy it (in a copying GC)
  - ▶ see if it is a pointer to an unvisited object

## Note on copying GC vs mark-sweep GC

- ▶ the key observation  
*the time (cost) of a single GC is roughly proportional to the amount of reached objects (i.e.,  $\propto r$ )*

ignores the cost of so-called “sweep phase”

- ▶ a more accurate quantification will be

the time (cost) of a single GC  $\approx \beta r + \gamma(M - r)$ ,

which adds a constant ( $\gamma$ ) to an allocation cost per byte, which any memory allocator will incur anyway

- ▶ i.e., the cost will be

$$\begin{aligned} & \alpha + \frac{\beta r + \gamma(M - r)}{M - r} \\ = & \alpha + \gamma + \beta \frac{r}{M - r} \end{aligned}$$



# Memory allocation cost of traversing GCs

- ▶ important formula:

$$\text{allocation cost per byte} \propto \text{const.} + \frac{r}{M - r}$$

- ▶  $r/(M - r)$  is often called *mark-cons ratio*. its origin:
  - ▶ mark : the amount of work to *mark* reachable objects
  - ▶ cons : the synonym of memory allocation in the ancient Lisp language = (cons x y)

# Memory allocation cost of traversing GCs

$$\text{cost per byte} \propto \text{const.} + \frac{r}{M - r}$$

- ▶  $r$  (primarily) depends only on app (not dependent of GCs)
  - ▶ remark:  $r$  may fluctuate depending on “when” GCs occur
- ▶  $M$  is an adjustable parameter (up to GC’s choice)
- ▶  $M$  is large  $\rightarrow$  the cost is small
- ▶  $\rightarrow$  you can reduce the cost by making  $M$  (memory usage) larger
- ▶ may sound obvious, but remember that what is important is the cost *per allocation (byte)*, not the frequency of GCs

# How large do we make $M$ (memory usage)?

- ▶ alright, the larger we make  $M$ , the smaller the cost becomes
  - ▶  $\rightarrow$  why don't we make it arbitrarily large (up to physical memory)?
- ▶ we normally set  $M$  “modestly”, like:

$$M \propto r$$

e.g., choose a constant  $k > 1$  and set:

$$M = kr$$

- ▶ a GC measures the amount of reachable objects to get  $r$  and set  $M$  according to the above formula

# How large do we make $M$ (memory usage)?

- ▶ in this setting,

- ▶ cost

$$\text{mark-cons ratio} = \frac{r}{kr - r} = \frac{1}{k - 1} = \text{const}$$

- ▶ memory usage

$\propto$  the size of reachable objects at a point during execution

both are “reasonable”

- ▶ most GCs allow you to set  $k$  (or  $M$  directly)
- ▶ normally,  $k = 1.5 \sim 2$ , but it is worth knowing that you can reduce the cost by setting it large

# Contents

Criteria of evaluating GCs (RC vs. traversing)

Two traversing GCs (mark&sweep vs. copying)

Memory allocation cost of traversing GCs (mark-cons ratio)

Generational GC

Incremental GC

Topics on Mark&Sweep GCs

- Free Area Management

- Improving mark&sweep GCs

  - Separated Mark Bits

  - Lazy Sweep

Conservative GC

# Generational GC: introduction

- ▶ **objective:** reduce *mark-cons ratio* in traversing GCs
- ▶ **how:** traverse and reclaim only *recently created objects* (*young generation*)
  - ▶ traverse only young generations often
  - ▶ traverse the entire heap occasionally when it does not reclaim enough space
- ▶ why does it work?

GC overhead

$\equiv$  GC's work per allocating a byte

## mark-cons ratio (review)

$$\begin{aligned} & \text{GC overhead} \\ \equiv & \text{GC's work per allocating a byte} \\ = & \frac{\text{GC's work}}{\text{memory allocated}} \end{aligned}$$



## mark-cons ratio (review)

$$\begin{aligned} & \text{GC overhead} \\ \equiv & \text{GC's work per allocating a byte} \\ = & \frac{\text{GC's work}}{\text{memory allocated}} \\ & (\text{assume a traversing GC; look at a specific GC}) \end{aligned}$$

## mark-cons ratio (review)

$$\begin{aligned} & \text{GC overhead} \\ \equiv & \text{GC's work per allocating a byte} \\ = & \frac{\text{GC's work}}{\text{memory allocated}} \\ & (\text{assume a traversing GC; look at a specific GC}) \\ \propto & \frac{\text{space reachable from the root}}{\text{space reclaimed}} \end{aligned}$$

## mark-cons ratio (review)

$$\begin{aligned} & \text{GC overhead} \\ \equiv & \text{GC's work per allocating a byte} \\ = & \frac{\text{GC's work}}{\text{memory allocated}} \\ & (\text{assume a traversing GC; look at a specific GC}) \\ \propto & \frac{\text{space reachable from the root}}{\text{space reclaimed}} \\ = & \frac{\text{space reachable from the root}}{\text{space unreachable from the root}} \end{aligned}$$

## mark-cons ratio (review)

$$\begin{aligned} & \text{GC overhead} \\ \equiv & \text{GC's work per allocating a byte} \\ = & \frac{\text{GC's work}}{\text{memory allocated}} \\ & (\text{assume a traversing GC; look at a specific GC}) \\ \propto & \frac{\text{space reachable from the root}}{\text{space reclaimed}} \\ = & \frac{\text{space reachable from the root}}{\text{space unreachable from the root}} \end{aligned}$$

► *the less reachable space there are, the smaller it becomes*

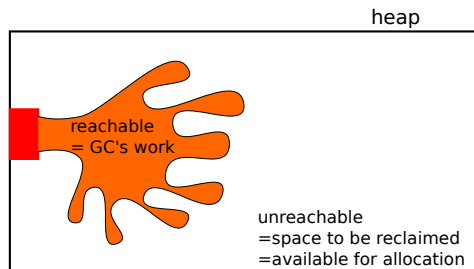
## mark-cons ratio (review)

$$\begin{aligned} & \text{GC overhead} \\ \equiv & \text{GC's work per allocating a byte} \\ = & \frac{\text{GC's work}}{\text{memory allocated}} \\ & (\text{assume a traversing GC; look at a specific GC}) \\ \propto & \frac{\text{space reachable from the root}}{\text{space reclaimed}} \\ = & \frac{\text{space reachable from the root}}{\text{space unreachable from the root}} \end{aligned}$$

- ▶ *the less reachable space there are, the smaller it becomes*
- ▶ below, we simply say an object is “alive” when it is “reachable from the root” (strictly, not a correct usage)

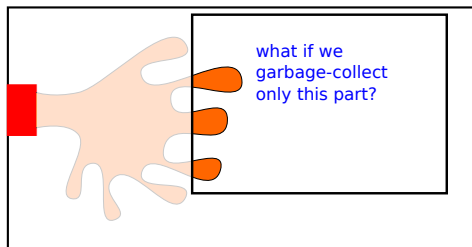
# Generational GC: the basic idea

- ▶ basic idea: traverse (collect) only *a region that has a lesser live object ratio*



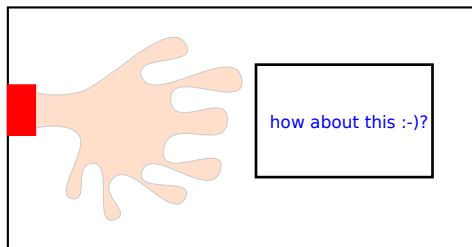
# Generational GC: the basic idea

- ▶ basic idea: traverse (collect) only *a region that has a lesser live object ratio*



# Generational GC: the basic idea

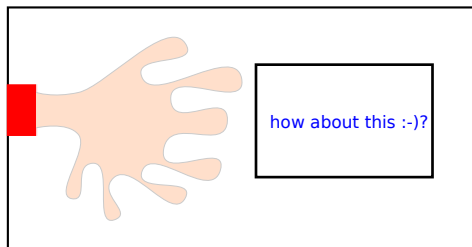
- ▶ basic idea: traverse (collect) only *a region that has a lesser live object ratio*





# Generational GC: the basic idea

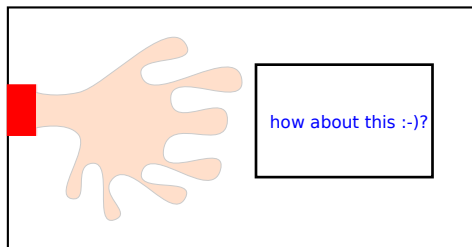
- ▶ basic idea: traverse (collect) only *a region that has a lesser live object ratio*



- ▶ two problems:

# Generational GC: the basic idea

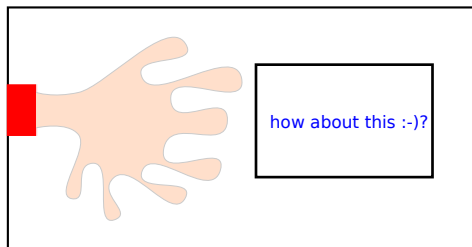
- ▶ basic idea: traverse (collect) only *a region that has a lesser live object ratio*



- ▶ two problems:
  1. where to target: *which region has a lesser live object ratio?*

# Generational GC: the basic idea

- ▶ basic idea: traverse (collect) only *a region that has a lesser live object ratio*



- ▶ two problems:
  1. where to target: *which region has a lesser live object ratio?*
  2. correctness: how to find all live objects in a region, *by traversing "only" that region?*

## Problem 1: where generational GC targets

a region holding young (recently created) objects

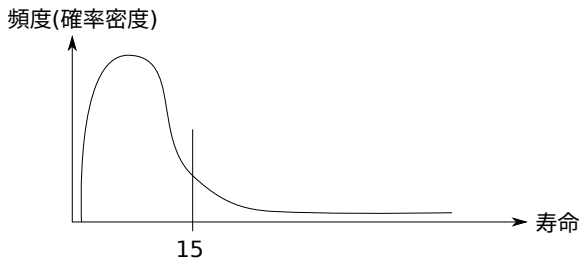
## Problem 1: where generational GC targets

a region holding young (recently created) objects

Q: why (or when) is this effective?

## (Weak) generational hypothesis

- ▶ “*most objects die young*”
- ▶ it seems to hold in most languages (where all memory allocations are served from the heap)



# Studies on (weak) generational hypothesis

- ▶ studies show “a (large) fraction  $d$  of objects die before a (young) age  $y$ ” in various languages
  - ▶ note: an “age” of an object  $o$  = the total size of memory allocated after  $o$  is created (that is, *the time is measured by the amount of memory allocation*)

authors	lang.	mortality rate ( $d$ )	age ( $y$ )
Zorn	Common Lisp	50-90%	10KB
Sanson and Jones	Haskell	75-95%	10KB
Hayes	Cedar	99%	721KB
Appel	SML/NJ	98%	varies
Barret and Zorn	C	50%	10KB
	C	90%	32KB

source: Richard Jones and Rafael Lins. “Garbage Collection. Algorithms for Automatic Memory Management” Chapter 7.1

# “most objects die young” and a rationale of generational GCs

- ▶ say 90% die younger than 10KB, then

mark-cons ratio when traversing most recent 10KB  $\approx 0.1$

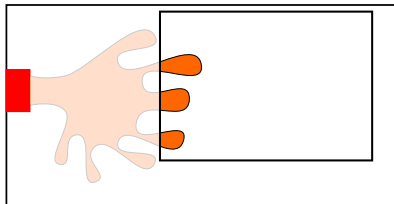
- ▶ if we use heap 2-3 times larger than the live objects,

the ratio when traversing the entire heap  $\approx 1/3 \sim 1/2 > 0.1$



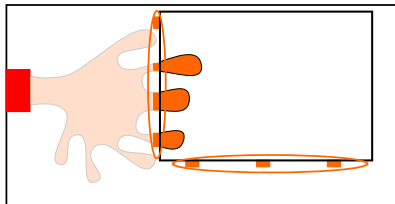
## Problem 2: how to make it correct?

- ▶ we need to find all young objects reachable from the root, through “*all pointers, young or old*”
- ▶ simply ignoring old objects won't work



## Problem 2: how to make it correct?

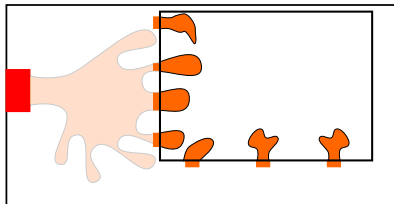
- ▶ we need to find all young objects reachable from the root, through “*all pointers, young or old*”
- ▶ simply ignoring old objects won't work



- ▶ solution: *record “all” pointers from “old  $\rightarrow$  young”* during the execution and consider them as part of the root

## Problem 2: how to make it correct?

- ▶ we need to find all young objects reachable from the root, through “*all pointers, young or old*”
- ▶ simply ignoring old objects won't work



- ▶ solution: *record “all” pointers from “old  $\rightarrow$  young”* during the execution and consider them as part of the root
- ▶ note: some may not be reclaimed, despite being unreachable from the root

# Write barrier

- ▶ an intervention in mutator actions to capture all “old  $\rightarrow$  young” pointers
- ▶ mutator actions that need an intervention: assignments:

(possibly) old object's field  $\leftarrow$  (possibly) young object

- ▶ in OCaml,

expression	description	need intervention?
<code>o.x &lt;- a</code>	update a mutable field	yes
<code>{ x = ...; ... }</code>	create a record etc.	no
<code>let b = o.x</code>	initialize a variable	no

- ▶ hopefully they rarely occur in “mostly functional” languages

# Implementing Write Barrier (1) Remembered Set

► given

```
1 o.x <- a;
```

we do

```
1 if (generation(a) < generation(o)) {  
2   if (o ∉ R) add(R, o)  
3 }
```

► the overhead is large

- obtain `generation(·)` (address comparison in copying GC)
- check if  $o \in R$
- manage `R`

## Implementing Write Barrier (2) Card Marking

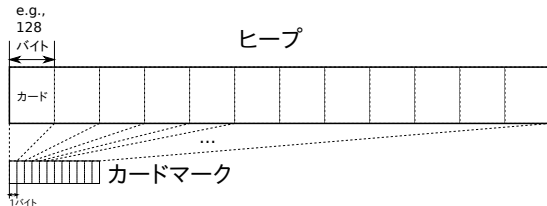
- ▶ basic idea: unconditionally record addresses pointers are written to
- ▶ partition the heap into constant-sized “cards”
  - ▶ a card: a region whose addresses share a number of most significant bits
    - ▶ e.g., share the highest 57 of 64 bit addresses
    - ▶  $\rightarrow$  a single card  $2^7 = 128$  bytes

ヒープ



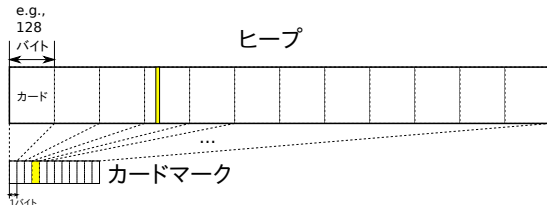
# Implementing Write Barrier (2) Card Marking

- ▶ basic idea: unconditionally record addresses pointers are written to
- ▶ partition the heap into constant-sized “cards”
  - ▶ a card: a region whose addresses share a number of most significant bits
    - ▶ e.g., share the highest 57 of 64 bit addresses
    - ▶ → a single card  $2^7 = 128$  bytes



# Implementing Write Barrier (2) Card Marking

- ▶ basic idea: unconditionally record addresses pointers are written to
- ▶ partition the heap into constant-sized “cards”
  - ▶ a card: a region whose addresses share a number of most significant bits
    - ▶ e.g., share the highest 57 of 64 bit addresses
    - ▶ → a single card  $2^7 = 128$  bytes



- ▶ record only whether **each card receives any pointer write** (1 byte/card; **card mark**)



# The overhead of card-marking

- ▶ e.g.: given the following pointer update,

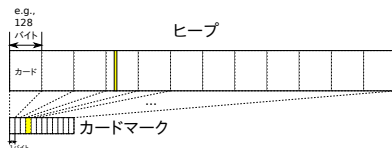
```
1 o->x <- y;
```

unconditionally record “a card containing  $\&o \rightarrow x$  is written”

```
1 C[(&o->x) >> 9] = 1;
```

$C$  is the base address to obtain the card address. that is,

```
1 C[heap >> 9] == card
```



# Card-marking : Pros and Cons

- ▶ a small write barrier overhead (if you hold  $C$  in a register, it takes three RISC instructions)

```
1 C[(&o->x) >> 9] = 1;
```

- ▶ memory overhead adjustable by adjusting card size (e.g. a card is 128 bytes  $\rightarrow 1/128$ )
- ▶ you cannot efficiently list written cards; you must check all cards ( $\propto$  heap)
- ▶ when any address of a card is written, we must consider all addresses of the card a root

# Contents

Criteria of evaluating GCs (RC vs. traversing)

Two traversing GCs (mark&sweep vs. copying)

Memory allocation cost of traversing GCs (mark-cons ratio)

Generational GC

**Incremental GC**

Topics on Mark&Sweep GCs

- Free Area Management

- Improving mark&sweep GCs

  - Separated Mark Bits

  - Lazy Sweep

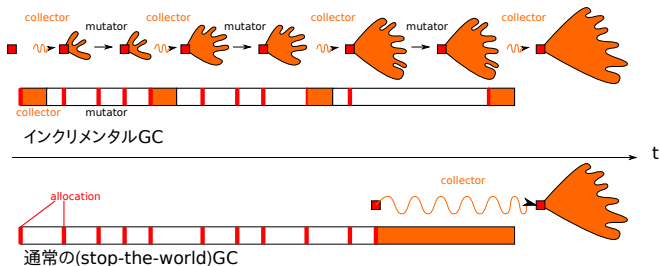
Conservative GC

# Incremental GC

- ▶ objective: *reduce the “pause time”* of traversing GC
  - ▶ good for applications that need real time or interactive responses
- ▶ recall that pause time  $\approx$  time to traverse all reachable objects

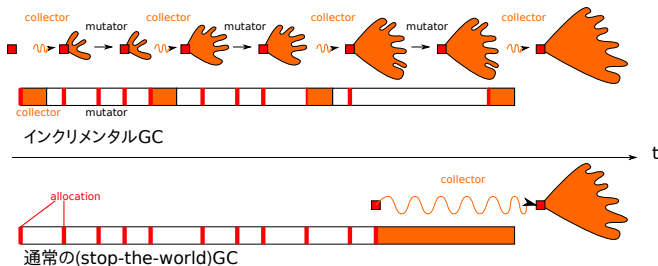
# Incremental GC

- ▶ objective: *reduce the “pause time”* of traversing GC
  - ▶ good for applications that need real time or interactive responses
- ▶ recall that pause time  $\approx$  time to traverse all reachable objects
- ▶ how: by traversing reachable objects *“a little bit at a time”*
  - ▶ instead of traversing 1 GB in one stroke, traverse 10 MB at a time, 100 times



# Challenges in incremental GC

- ▶ (from GC's view point) *the object graph changes while GC is traversing it*



- ▶ how to guarantee it does not miss any reachable object?
- ▶  $\Rightarrow$  we'll get back to the basics of graph traversal

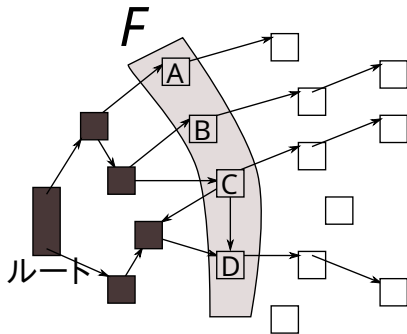
# Assumptions for later discussions

- ▶ only a single mutator (the app is single-threaded)
- ▶ the mutator and the collector run *“alternately” (not at the same time)*
  - ▶ the collector does a little bit of its work upon a memory allocation
- ▶ i.e., we do not consider race conditions that would happen when they are truly concurrent

# Graph traversal : basics

- ▶ traversing GC  $\approx$  graph traversal
- ▶ the principle is the same whether it's mark&sweep or copying
- ▶ omitting details, it is:

```
1  F = { root };  
2  while (F is not empty) {  
3    o = pop(F);  
4    for (all pointers p in o)  
5      if (!marked(p)) {  
6        mark(p);  
7        add(F, p);  
8      }  
9  }
```

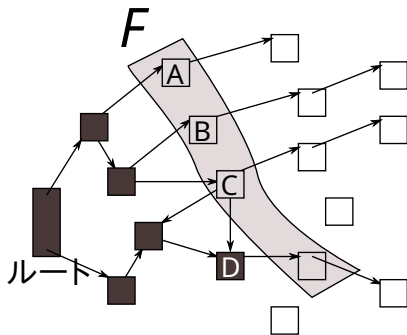




# Graph traversal : basics

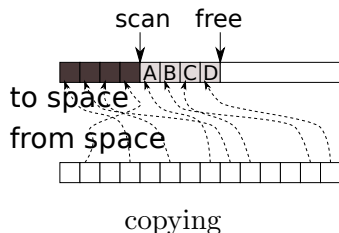
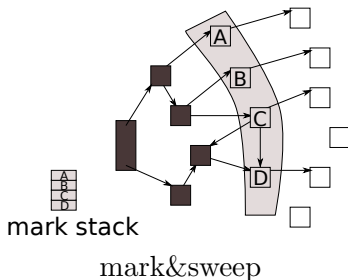
- ▶ traversing GC  $\approx$  graph traversal
- ▶ the principle is the same whether it's mark&sweep or copying
- ▶ omitting details, it is:

```
1  F = { root };  
2  while (F is not empty) {  
3    o = pop(F);  
4    for (all pointers p in o)  
5      if (!marked(p)) {  
6        mark(p);  
7        add(F, p);  
8      }  
9  }
```



# Key data : the frontier

- ▶  $F$  : frontier
- ▶ the set of objects that have been visited but whose children may have not
- ▶ the actual data structure
  - ▶ mark&sweep : mark stack
  - ▶ copying : a part of the to space



# The issue that an incremental GC must address

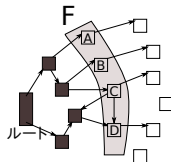
```
1 F = { root };
2 while (F is not empty) {
3   o = pop(F);
4   for (all pointers p in o)
5     if (!marked(p)) {
6       mark(p);
7       add(F, p);
8     }
9   if (has iterated a few times)
10    // the graph changes below
11    resume_mutator();
12 }
```

- ▶ **ordinary GC:** the while loop runs until the end keeping the mutator stopped → the object graph does not change during the loop
- ▶ **incremental GC:**
  - ▶ *the collector gets interrupted by the mutator every once in a while*
  - ▶ ... and continues after a while
  - ▶ that is, the issues is how to do with the fact that *the graph may change between iterations of the while loop*

# The tri-color abstraction

- ▶ likens a graph traversal to coloring its nodes
- ▶ visiting an object  $\approx$  coloring an object
  - ▶ **black**: the object and its children have been visited
  - ▶ **gray**: it has been visited but its children may not
  - ▶ **white**: it has not been visited
- ▶ the graph traversal using the tri-color abstraction

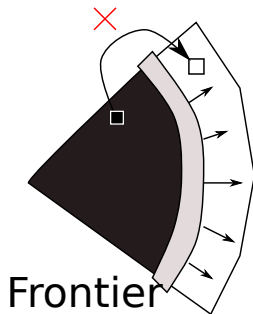
```
1 gray the root;  
2 while (there is a gray object) {  
3   o = pick a gray object and blacken it;  
4   for (all pointers in o)  
5     if (p points to a white object)  
6       gray it;  
7   the mutator changes the graph; }
```



- ▶ correctness of the algorithm: *when there are no gray objects, all objects reachable from the root are black (i.e., white objects are unreachable)*

# A problematic mutation to the graph

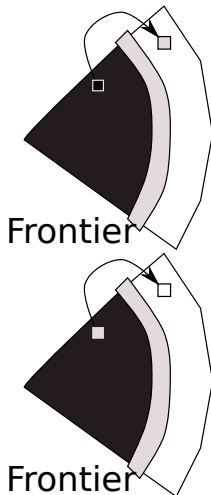
- ▶ intuitively, *the issue seems the mutator may create “black  $\rightarrow$  white” pointers*
  - ▶ **black**: GC thinks it has “done” with it
  - ▶ **white**: going to be reclaimed, unless found in other paths
- ▶  $\Rightarrow$  prevent “black  $\rightarrow$  white pointers” from being created



# Two approaches to preventing black→white

capture the point where “**black**→**white**” is about to be created

1. approach #1: gray the **white** (make **black**→**gray**)
  - ▶ pros: the frontier always progresses
  - ▶ pros: easier to work with for copying GCs
  - ▶ cons: reclaim less objects. even if **white** becomes unreachable due to another update to **black**, it won't be reclaimed (by the current GC)
2. approach #2: get the **black** back to gray (make **gray**→**white**)
  - ▶ pros: reclaim more objects
  - ▶ cons: the frontier retreats



# Mutator actions that need to be captured

naively all pointer movements must be captured

- ▶ write a pointer into an object field (write barrier)

```
1  o->x = p
```

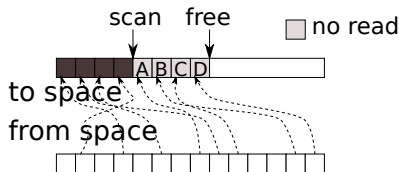
- ▶ write a pointer into a root  $\equiv$  write a pointer to a variable (read barrier)

```
1  p = o->x
```

the latter is so frequent that some approaches avoid them  
(example #2: Boehm GC)

# Example #1: Appel-Ellis-Li

- ▶ copying GC + incremental
- ▶ based on the approach # 1. more precisely, maintain the following invariant  
*the mutator never sees a pointer to white*
- ▶ how?
  - ▶ intervene in reading a field from gray objects (read barrier)
- ▶ read-protect the region of gray objects  $\subset$  scan  $\sim$  free, by the virtual memory primitive of operating systems

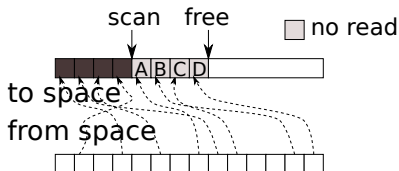




# Appel-Ellis-Li : the read barrier in action

- ▶ when a field of a gray object is read, blacken objects in the page containing it (= scan those objects → they become gray)

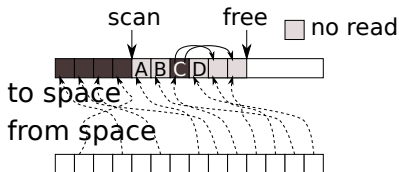
```
1 trap_read_from_grey(a) {  
2   page = the page including a;  
3   for (all objects o in the page) {  
4     scan(o); // copy o's children  
5   }  
6   unprotect(page);  
7 }
```



# Appel-Ellis-Li : the read barrier in action

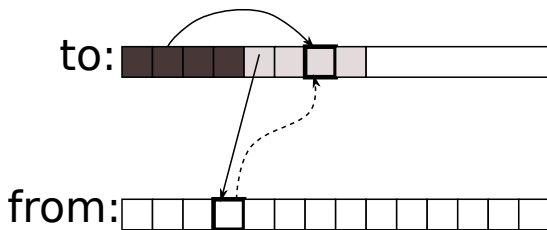
- ▶ when a field of a gray object is read, blacken objects in the page containing it (= scan those objects → they become gray)

```
1 trap_read_from_grey(a) {  
2   page = the page including a;  
3   for (all objects o in the page) {  
4     scan(o); // copy o's children  
5   }  
6   unprotect(page);  
7 }
```



## Remark : it's easier for copying GC

- ▶ during a copying GC, there are two versions of each visited object (one in the from space and the other in the to space)
- ▶ immutable objects do not care which one the mutator sees, but mutable ones do
- ▶ it will eventually see the one in to space anyways, so it's natural to maintain “it never sees the one in the from space”
- ▶ → it's natural to let the mutator never see (get a pointer to) a white object



## Example #2: Boehm GC

- ▶ conservative GC ( $\rightarrow$  mark&sweep) + incremental
- ▶ invariants:
  - ▶ “non-root **black**  $\rightarrow$  white” pointers never exist
- ▶ how?
  - ▶ capture “*writing to an object field*” (*write barrier*)
- ▶ remark: “**root**  $\rightarrow$  **white**” pointers *may* exist
  - ▶ prevention requires us to capture writing to the root  $\rightarrow$  *reading* from an object
  - ▶ the overhead is so large that it deserves a separate treatment (covered later)

# Write barrier in Boehm GC

- ▶ capture writing into objects by virtual memory (the only choice in C/C++)
- ▶ gray the “written-to” object
  - ▶ push it onto the mark stack
- ▶ no read barriers → “root (black) → white” pointers are allowed
- ▶ at the end of a mark phase, it traverses from the root again
- ▶ during this second traversal, the mutator is stopped → it may cause a long pause time

## Appendix: a more rigorous correctness proof

- ▶ while it is clear “black $\rightarrow$ white” pointers cause a problem, it is not trivial that preventing them is sufficient to solve the problem
- ▶ the proposition to prove: after the following algorithm finished,

*reachable from the root  $\rightarrow$  black*

```
1 gray the root;  
2 while (there are gray objects) {  
3   o = pick and blacken a gray object;  
4   for (pointers p in o)  
5     if (p points to a white object)  
6       gray it;  
7   the mutator changes the graph in such a way  
8   that does not create black  $\rightarrow$  white pointer;  
9 }
```

# The key invariant

- ▶ the following “always” holds during the execution (GC or mutator)  
*(I): all “white” objects reachable from the root are reachable from some “gray” objects*
- ▶ if this is true,
  - (I) and the termination condition (i.e. there are no grays)
  - no white objects are reachable from the root
  - white objects can be reclaimedand we are done. the only remaining task is to prove (I).

# Proof of (I)

- ▶ say  $w$  is a white object reachable from the root



- ▶ since the root is always black or gray and there are no “black  $\rightarrow$  white” pointers (\*), there must be a gray object on each path  $P$  from the root to  $w$  (QED).



- ▶ (\*) : you need to show that not only the mutator but also the collector never creates “black  $\rightarrow$  white” pointers. it’s easy and left as an exercise.



# Contents

Criteria of evaluating GCs (RC vs. traversing)

Two traversing GCs (mark&sweep vs. copying)

Memory allocation cost of traversing GCs (mark-cons ratio)

Generational GC

Incremental GC

Topics on Mark&Sweep GCs

- Free Area Management

- Improving mark&sweep GCs

  - Separated Mark Bits

  - Lazy Sweep

Conservative GC

# Contents

Criteria of evaluating GCs (RC vs. traversing)

Two traversing GCs (mark&sweep vs. copying)

Memory allocation cost of traversing GCs (mark-cons ratio)

Generational GC

Incremental GC

Topics on Mark&Sweep GCs

- Free Area Management

- Improving mark&sweep GCs

  - Separated Mark Bits

  - Lazy Sweep

Conservative GC

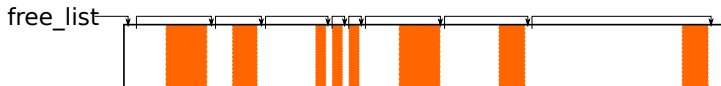
# Managing and finding free space

- ▶ in any method except for copying GC (mark&sweep GC, reference counting, malloc/free), free space are not contiguous
- ▶ → tracking and managing free blocks is required
- ▶ goal:
  - ▶ good allocation speed: quickly find a region that fits the request size
  - ▶ good memory utilization: do not waste available space
- ▶ basic data structure: free list (list of free blocks)



# Managing and finding free space

- ▶ in any method except for copying GC (mark&sweep GC, reference counting, malloc/free), free space are not contiguous
- ▶ → tracking and managing free blocks is required
- ▶ goal:
  - ▶ good allocation speed: quickly find a region that fits the request size
  - ▶ good memory utilization: do not waste available space
- ▶ basic data structure: free list (list of free blocks)



# Free list

- ▶ list of free blocks (or cells)

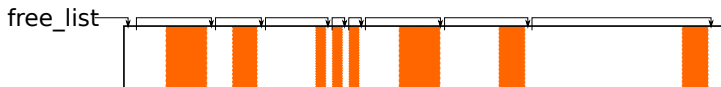
```
1 typedef struct cell {  
2     struct cell * next;  
3     size_t sz;  
4     /* other info as necessary */  
5 } cell;  
6 cell * free_list;
```

- ▶ allocation (malloc)  $\approx$

1. (linearly) search for a cell large enough for the requested size
2. if a free space remains in the cell, put it back to the free list

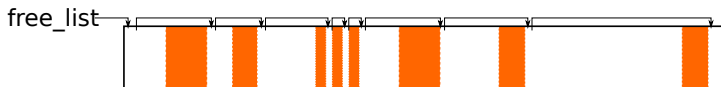
- ▶ reclamation (free)  $\approx$

1. put it back to the free list (issue: how to know its size?)
2. if the cell just freed is adjacent to another free cell, merge them (coalescing)



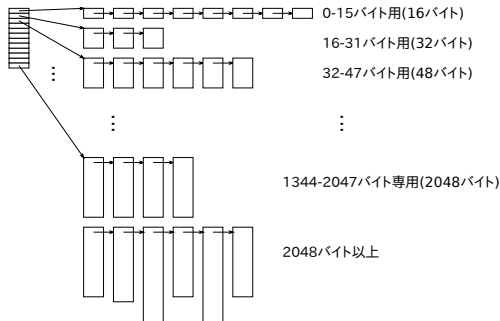
# Issues in the simple method

- ▶ allocation:
  - ▶ needs to traverse a fair amount of cells (until you find a cell that fits)
  - ▶ → make many free lists, one for a fixed size (*segregated free lists*)
- ▶ reclamation:
  - ▶ needs to check if coalescing is possible
  - ▶ needs to know the size of the freed cell
  - ▶ → manage memory in a larger unit (*page*) and dedicate a page to a single size (*Big Bags of Pages; BiBOP*)



# Segregated free lists

- ▶ for small sizes (e.g., < 2KB), make a free list for various representative sizes
- ▶ a single list for large sizes
- ▶ ex:
  - ▶ 16, 32, 48, 64, ..., 448, 512, 672, 800, 1024, 1344, 2048
  - ▶ 2048 bytes or larger



# Allocation sequence

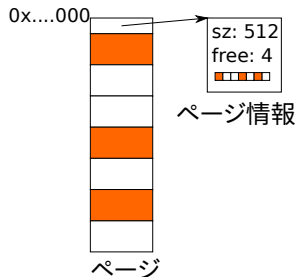
- ▶ **Colored**: a fast path for small objects
- ▶ **blue**: the overhead removable if the size (*sz*) is a compile-time constant
- ▶ **red**: the essential cost. traverse a list once (6-7 instructions)
- ▶ note: in multithreaded programs, we either have to
  - ▶ let each thread have its own `free_lists`, or
  - ▶ atomically perform **the read-modify-write on `free_lists`** (this hinders scalability)

```
1 void ** free_lists;
2
3 void * malloc(size_t sz) {
4     if (SMALL(sz)) {
5         size_t idx =
6             bytes_to_idx(sz);
7         cell * a =
8             free_lists[idx];
9         if (a) {
10             free_lists[idx] =
11                 a->next;
12             return a;
13         } else {
14             return malloc_slow(sz);
15         }
16     }
17 }
```



# Big Bags of Pages

- ▶ manage the heap by dividing it into constant-sized blocks (page)
  - ▶ a page: a set of addresses sharing a number of highest bits
  - ▶ e.g. 64 bit addresses, sharing the highest 52 bits  $\rightarrow 2^{12} = 4096$  bytes/page
- ▶ each page is either
  - ▶ completely free or
  - ▶ used only for a single size (e.g., only for 48 bytes)
- ▶ Coalescing: repurpose a page only when the page becomes completely empty
  - ▶  $\rightarrow$  only need to count the number of free cells in the page
- ▶ does not require per-object size field either



# Contents

Criteria of evaluating GCs (RC vs. traversing)

Two traversing GCs (mark&sweep vs. copying)

Memory allocation cost of traversing GCs (mark-cons ratio)

Generational GC

Incremental GC

Topics on Mark&Sweep GCs

- Free Area Management

- Improving mark&sweep GCs

  - Separated Mark Bits

  - Lazy Sweep

Conservative GC

# Improving performance of mark&sweep GC

- ▶ overall structure:
  1. **mark phase**: traverses pointers from the root, **marking** reached objects along the way
  2. **sweep phase**: reclaims unmarked objects → **pushes them back to an appropriate free list**
- ▶ basics:
  - ▶ segregated free lists
  - ▶ BiBOP
  - ▶ **mark bits separated from objects**
  - ▶ **lazy sweep**

# Separated mark bits

- ▶ question: where do you put the mark bit of an object?
- ▶ Method 1: use a word within an object
- ▶ Method 2: use a separate space dedicated for mark-bits outside objects
  - ▶ where is the separate space, exactly? → page header; holds mark bits of all the objects in the page together (1 byte/object)

```
1 mark(void * o) {  
2     page * page = o & 0xFFF...000; /* page header address */  
3     page->header->mark[(o & 0x000...FFF) >> 4] = 1;  
4 }
```

- ▶ point: gather spaces that are written

# Lazy sweep

- ▶ why do we need to sweep: reclaim space that has become free
- ▶ naturally, you would put them back to an appropriate free list (cf. BiBOP)
- ▶ **lazy sweep**: defer this operation until you need to allocate them

# Overview of the sweep phase

- ▶ after a mark phase is finished, a page is either
  - ▶ empty: zero objects have been reached
  - ▶ partial:  $> 0$  objects have been reached,  $> 0$  objects have not been reached
  - ▶ full: zero objects have not been reached
- ▶ a naive implementation of a sweep phase:

```
1 for (all pages p) {  
2     if (p is empty) {  
3         put p in the empty page list;  
4         /* can be repurposed for any size */  
5     } else if (p is partial) {  
6         sz = the size of objects in the page;  
7         put free cells in p to the free list fo sz bytes;  
8     }  
9 }
```

# Lazy sweep

- ▶ does not rebuild free lists immediately
- ▶ instead puts the page into the list of “to-reclaim” pages

```
1 for (all pages p) {  
2   if (p is empty) {  
3     put p in the empty page list;  
4     /* can be repurposed for any size */  
5   } else if (p is partial) {  
6     sz = the size of objects in the page;  
7     put p into the reclaim list for sz bytes;  
8   }  
9 }
```

# Reclaim list

- ▶ list of pages that have at least one free cell
- ▶ like free lists, there is a list per size
- ▶ when an allocation finds the free list empty, look at the reclaim list and if there is any page, move free cells of a page into the free list



# What's the point?

- ▶ simply deferring the task you need to do anyway? not exactly so
- ▶ make more coalescing opportunities:
  - ▶ after a few GCs, a page may become empty before it needs to be reused for allocation
- ▶ improve temporal locality of references:
  - ▶ by touching free cells to put them back to free list, closely before they are used by the mutator
- ▶ shorten the pause time due to the sweep phase

# Contents

Criteria of evaluating GCs (RC vs. traversing)

Two traversing GCs (mark&sweep vs. copying)

Memory allocation cost of traversing GCs (mark-cons ratio)

Generational GC

Incremental GC

Topics on Mark&Sweep GCs

- Free Area Management

- Improving mark&sweep GCs

  - Separated Mark Bits

  - Lazy Sweep

Conservative GC

# Conservative GC

- ▶ *conservative* GC
  - ▶  $\approx$  GC for languages designed without assuming GC, such as C/C++

# Conservative GC

- ▶ *conservative* GC
  - ▶  $\approx$  GC for languages designed without assuming GC, such as C/C++
  - ▶  $\approx$  GC in the presence of words that may or may not be pointers (conservatively assumed to be pointers)

# Conservative GC

- ▶ *conservative* GC
  - ▶  $\approx$  GC for languages designed without assuming GC, such as C/C++
  - ▶  $\approx$  GC in the presence of words that may or may not be pointers (conservatively assumed to be pointers)
- ▶ antonym: *accurate* GC
  - ▶ does not necessarily reclaim all dead (no longer used) objects
  - ▶ *accurate* or *conservative* refers to whether “pointer identifications” are accurate or not
  - ▶ languages that implement an accurate GC normally use a data representation in which looking at a single word can tell you whether it is a pointer or not
    - ▶ ex: the last bit = 0 (pointer), = 1 (non-pointer)

## A challenge in C/C++: pointer ambiguity

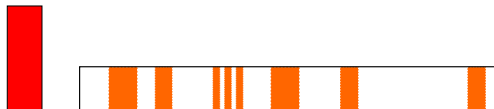
- ▶ a pointer and a non-pointers cannot be told apart; a word “7596272344674820427 (10110100101101011011...011000010100101<sub>2</sub>)” can be any of the following
  - ▶ a pointer to an object at address 7596272344674820427,
  - ▶ an integer 7596272344674820427,
  - ▶ a part of a string (“Kawasaki”),
  - ▶ a double precision floating point number (6.549545...  $\times 10^{199}$ ),
  - ▶ ...

# A challenge in C/C++: pointer ambiguity

- ▶ a pointer and a non-pointers cannot be told apart; a word “7596272344674820427 (10110100101101011011...011000010100101<sub>2</sub>)” can be any of the following
  - ▶ a pointer to an object at address 7596272344674820427,
  - ▶ an integer 7596272344674820427,
  - ▶ a part of a string (“Kawasaki”),
  - ▶ a double precision floating point number  $(6.549545 \dots \times 10^{199})$ ,
  - ▶ ...
- ▶ the basic principle:
  - ▶ *if a word is an address of a block being used, it is assumed to be the pointer to it*
  - ▶ a non-pointer may be misidentified as a pointer
  - ▶ a method to minimize the loss (leak) caused by misidentified pointers → [blacklisting](#)

# Blacklisting

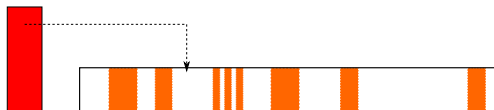
- ▶ during marking, record (“blacklist”) words  $p$  (suspicious addresses) satisfying:





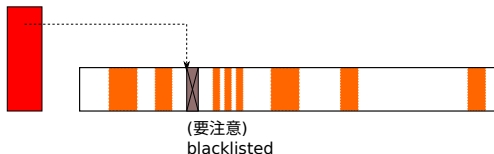
# Blacklisting

- ▶ during marking, record (“blacklist”) words  $p$  (suspicious addresses) satisfying:
  - ▶ address  $p$  is currently *not* used and
  - ▶  $p$  is a subject of future allocation (i.e., an address within the current heap)



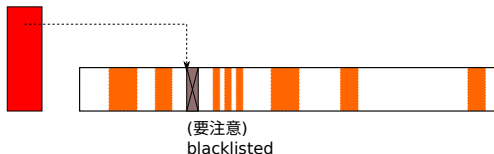
# Blacklisting

- ▶ during marking, record (“blacklist”) words  $p$  (suspicious addresses) satisfying:
  - ▶ address  $p$  is currently *not* used and
  - ▶  $p$  is a subject of future allocation (i.e., an address within the current heap)
- ▶ do not use such  $p$ ’s for *future allocation*



# Blacklisting

- ▶ during marking, record (“blacklist”) words  $p$  (suspicious addresses) satisfying:
  - ▶ address  $p$  is currently *not* used and
  - ▶  $p$  is a subject of future allocation (i.e., an address within the current heap)
- ▶ do not use such  $p$ 's for *future allocation*
- ▶ note that we already lose (a memory associated with)  $p$ , but it would be much worse and devastating to allocate  $p$  and make  *$p$  and all objects reachable from  $p$*  uncollectable (*the domino effect*)

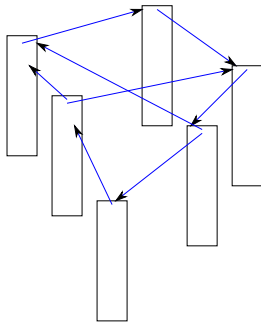
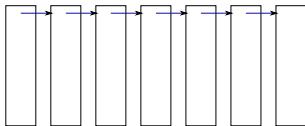


## Other tips in conservative GC

1. see <http://hboehm.info/gc/gcinterface.html>
2. `GC_MALLOC_ATOMIC` :
  - ▶ same as `GC_MALLOC`, except you indicate (declare) you never put pointers in it (good for strings and numerical arrays)
  - ▶ reduce the probability of pointer misidentification
  - ▶ reduce the space that must be traversed
3. `GC_MALLOC_IGNORE_OFF_PAGE` : declares “you never put pointers except in the first 512 bytes”
4. clear pointers no longer necessary with NULL
  - ▶ pointers within a data structure
  - ▶ prevent the domino effect when a single object is mistakenly kept alive
5. tips in how you link data structures
  - ▶ data structures less prone to the domino effect due to a pointer misidentification

# Data structure (not) prone to the domino effect

- ▶ suppose you make link lists, trees and graphs
- ▶ (NG): directly link large nodes with payload
- ▶ (GOOD): separate the structure linking nodes (*the spine*) and the payloads → misidentifying a payload does not lead to another object



# Data structure (not) prone to the domino effect

- ▶ suppose you make link lists, trees and graphs
- ▶ **(NG)**: directly link large nodes with payload
- ▶ **(GOOD)**: separate the structure linking nodes (*the spine*) and the payloads → misidentifying a payload does not lead to another object

