

プログラミング言語 6  
ガベージコレクション (2)  
Programming Languages 6  
Garbage Collection (2)

田浦

# 目次 / Contents

- ① 世代別 GC / Generational GC
- ② マーク&スイープ GC のトピック
  - 空き領域管理
  - マーク&スイープ GC の性能改善
    - 空き領域管理
    - 遅延スイープ
  - 保守的 GC / Conservative GC
- ③ インクリメンタル GC / Incremental GC
- ④ 参照カウントの変種
  - 遅延参照カウント
  - Sticky 参照カウント
  - 1 bit 参照カウント

# Contents

## ① 世代別 GC / Generational GC

## ② マーク&スイープ GC のトピック

- 空き領域管理
- マーク&スイープ GC の性能改善
  - 空き領域管理
  - 遅延スイープ
- 保守的 GC / Conservative GC

## ③ インクリメンタル GC / Incremental GC

## ④ 参照カウントの変種

- 遅延参照カウント
- Sticky 参照カウント
- 1 bit 参照カウント

# 世代別 GC: はじめに

- 目的: 走査型 GC における GC オーバーヘッド (mark-cons 比) の低下
- 手段: 普段は最近作られたオブジェクト (若い世代) だけを走査・回収する
  - ▶ 普段は若い世代だけを走査
  - ▶ それだけでは回収しきれなくなったら全体を走査
- なぜこれでうまく行く (かもしれない) のだろう?

# 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?

# mark-cons 比 (復習)

GC オーバーヘッド

≡ 1 バイトのメモリ割り当てに要した GC の仕事量

## mark-cons 比 (復習)

$$\begin{aligned} & \text{GC オーバーヘッド} \\ \equiv & \text{1 バイトのメモリ割当てに要した GC の仕事量} \\ = & \frac{\text{GC が行った仕事量}}{\text{メモリ割当量}} \end{aligned}$$

# mark-cons 比 (復習)

$$\begin{aligned} & \text{GC オーバーヘッド} \\ \equiv & \text{1 バイトのメモリ割り当てに要した GC の仕事量} \\ = & \frac{\text{GC が行った仕事量}}{\text{メモリ割当量}} \\ & (\text{以降走査型を仮定. ある一回の GC に着目}) \end{aligned}$$



# mark-cons 比 (復習)

$$\begin{aligned} & \text{GC オーバーヘッド} \\ \equiv & \text{1 バイトのメモリ割り当てに要した GC の仕事量} \\ = & \frac{\text{GC が行った仕事量}}{\text{メモリ割当量}} \\ & (\text{以降走査型を仮定. ある一回の GC に着目}) \\ \propto & \frac{\text{ルートから到達可能だったオブジェクトの量}}{\text{回収したオブジェクトの量}} \end{aligned}$$

# mark-cons 比 (復習)

$$\begin{aligned} & \text{GC オーバーヘッド} \\ \equiv & \text{1 バイトのメモリ割り当てに要した GC の仕事量} \\ & \text{GC が行った仕事量} \\ = & \frac{\text{メモリ割当量}}{\text{(以降走査型を仮定. ある一回の GC に着目)}} \\ \propto & \frac{\text{ルートから到達可能だったオブジェクトの量}}{\text{回収したオブジェクトの量}} \\ = & \frac{\text{ルートから到達可能だったオブジェクトの量}}{\text{ルートから到達不能だったオブジェクトの量}} \end{aligned}$$

# mark-cons 比 (復習)

GC オーバーヘッド

≡ 1 バイトのメモリ割り当てに要した GC の仕事量

GC が行った仕事量

=  $\frac{\text{メモリ割当量}}{\text{メモリ割当量}}$

(以降走査型を仮定. ある一回の GC に着目)

ルートから到達可能だったオブジェクトの量

∝  $\frac{\text{ルートから到達可能だったオブジェクトの量}}{\text{回収したオブジェクトの量}}$

ルートから到達可能だったオブジェクトの量

=  $\frac{\text{ルートから到達可能だったオブジェクトの量}}{\text{ルートから到達不能だったオブジェクトの量}}$

- つまり, 「ルートから到達可能なオブジェクトの割合」が少ないほど効率が良くなる

# mark-cons 比 (復習)

GC オーバーヘッド

≡ 1 バイトのメモリ割り当てに要した GC の仕事量

GC が行った仕事量

=  $\frac{\text{メモリ割当量}}{\text{メモリ割当量}}$

(以降走査型を仮定. ある一回の GC に着目)

ルートから到達可能だったオブジェクトの量

∝  $\frac{\text{回収したオブジェクトの量}}{\text{回収したオブジェクトの量}}$

ルートから到達可能だったオブジェクトの量

=  $\frac{\text{ルートから到達可能だったオブジェクトの量}}{\text{ルートから到達不能だったオブジェクトの量}}$

- つまり, 「ルートから到達可能なオブジェクトの割合」が少ないほど効率が良くなる
- 以降, 「ルートから到達可能」を「生きてる」と言うことにする (厳密には誤用)

# mark-cons ratio (review)

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)

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



# mark-cons ratio (review)

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

# mark-cons ratio (review)

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

- *the less reachable objects there are, the smaller it becomes*

# mark-cons ratio (review)

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

- *the less reachable objects 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)

# 世代別 GC の基本発想

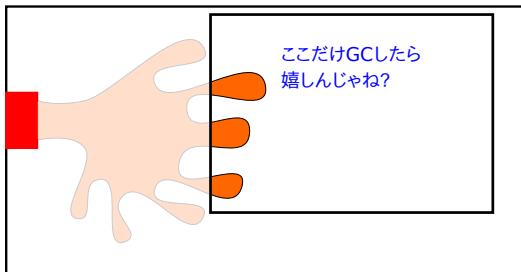
- 基本発想: 「生きてるオブジェクトの割合が少ない」領域だけを GC すれば?



- 2つの課題:
  - ① 狙う領域: 生きているオブジェクトの割合が少ないのはどこ?
  - ② 正しさ: その領域「だけ」を GC(走査) して, その領域内の, 生きてるオブジェクトをすべてを見つけるには?

# 世代別 GC の基本発想

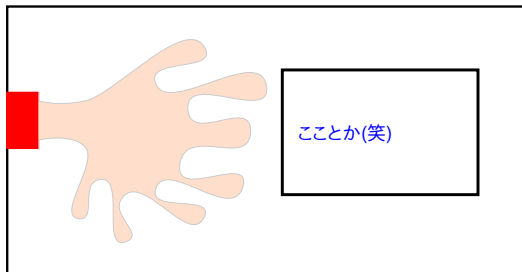
- 基本発想: 「生きてるオブジェクトの割合が少ない」領域だけを GC すれば?



- 2つの課題:
  - ① 狙う領域: 生きているオブジェクトの割合が少ないのはどこ?
  - ② 正しさ: その領域「だけ」を GC(走査) して、その領域内の、生きてるオブジェクトをすべてを見つけるには?

# 世代別 GC の基本発想

- 基本発想: 「生きてるオブジェクトの割合が少ない」領域だけを GC すれば?



- 2つの課題:
  - ① 狙う領域: 生きているオブジェクトの割合が少ないのはどこ?
  - ② 正しさ: その領域「だけ」を GC(走査)して、その領域内の、生きてるオブジェクトをすべてを見つけるには?

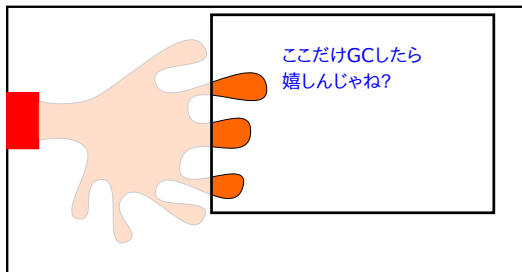
# Generational GC: the basic idea

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



# Generational GC: the basic idea

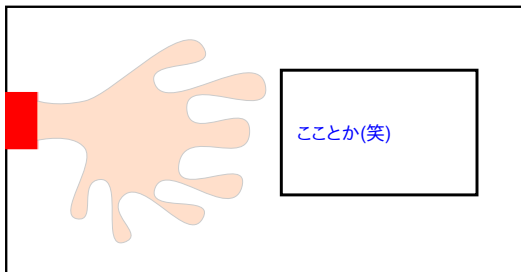
- basic idea: traverse (collect) only *a region that has a lesser live object ratio*





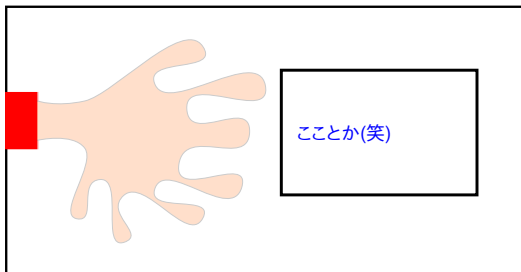
# Generational GC: the basic idea

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



# Generational GC: the basic idea

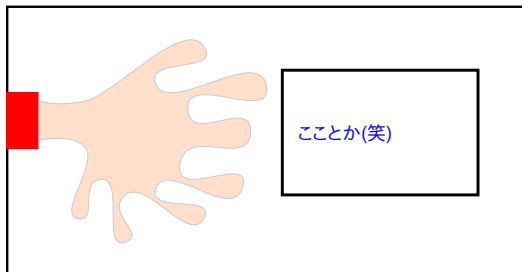
- basic idea: traverse (collect) only *a region that has a lesser live object ratio*



- two problems:

# Generational GC: the basic idea

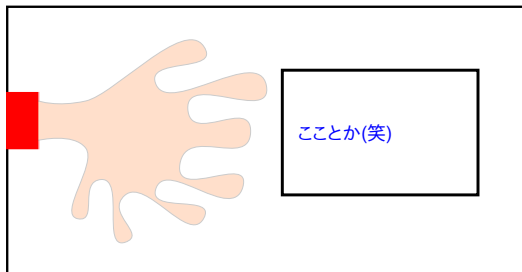
- basic idea: traverse (collect) only *a region that has a lesser live object ratio*



- 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?*

# 課題 1: 世代別 GC が狙う領域

若い (最近作られた) オブジェクトがいる領域

# 課題 1: 世代別 GC が狙う領域

若い (最近作られた) オブジェクトがいる領域

Q: なぜ (または, いつ) これが功を奏するのか?

# Problem 1: where generational GC targets

a region holding young (recently created) objects

# Problem 1: where generational GC targets

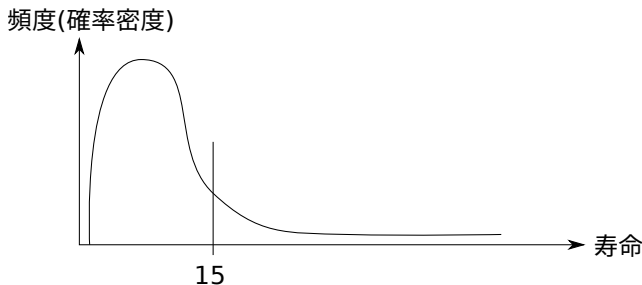
a region holding young (recently created) objects

Q: why (or when) is this effective?



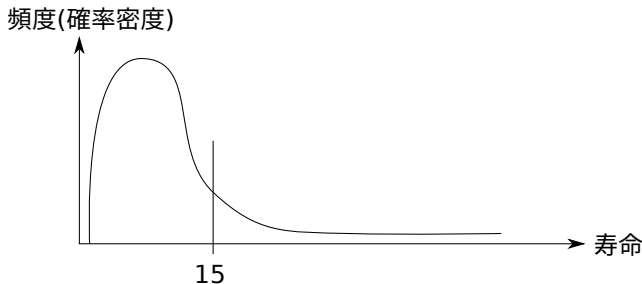
# (Weak) Generational Hypothesis

- 「ほとんどのオブジェクトは短命」 (“most objects die young”)
- (メモリ割り当て手段がヒープしかない言語では) 多くのプログラムで成り立っていそうな性質



# (Weak) generational hypothesis

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



# (Weak) Generational Hypothesis に関する study

- 様々な言語・プログラムで、「ある (高い) 割合  $d$  のオブジェクト, ある (若い) 年齢  $y$  以前に, 死んだ」ことが報告されている
  - ▶ 注: あるオブジェクト  $o$  の年齢 =  $o$  生成後に行われたメモリ割り当ての量 (つまり, メモリ割り当て量で時間を測る)

著者	言語	死亡率 ( $d$ )	年齢 ( $y$ )
Zorn	Common Lisp	50-90%	10KB
Sanson and Jones	Haskell	75-95%	10KB
Hayes	Cedar	99%	721KB
Appel	SML/NJ	98%	可変
Barret and Zorn	C	50%	10KB
	C	90%	32KB

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

# 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

# 「ほとんどが短命」と世代別 GC の根拠

- 例えば 90% が 10KB 以内に死ぬのであれば,

最近 10KB を走査した時の mark-cons 比  $\approx 0.1$

- 一方, 生きているオブジェクトの 2~3 倍のヒープサイズを使う場合,

全てを走査した時の mark-cons 比  $\approx 1/3 \sim 1/2 > 0.1$

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

- say 90% die younger than 10KB, then

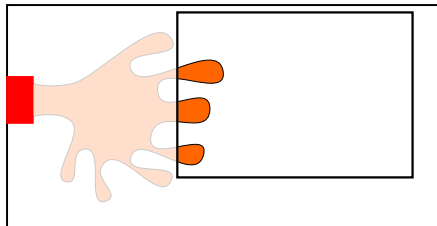
mark-cons ratio when traversing most recent 10KB  $\approx 0.1$

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

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

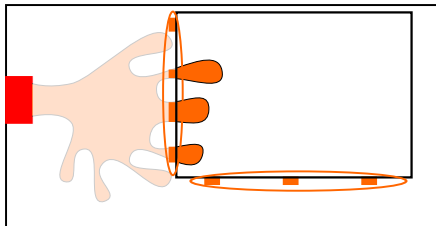
## 課題2: 世代別 GC が正しく動作するには?

- あくまで見つけるべきは、「ルートから (老若問わず) 全部のポインタをたどって」到達可能な、若いオブジェクト
- 古いオブジェクトを無視するだけではダメ



## 課題2: 世代別 GC が正しく動作するには?

- あくまで見つけるべきは、「ルートから (老若問わず) 全部のポインタをたどって」到達可能な、若いオブジェクト
- 古いオブジェクトを無視するだけではダメ

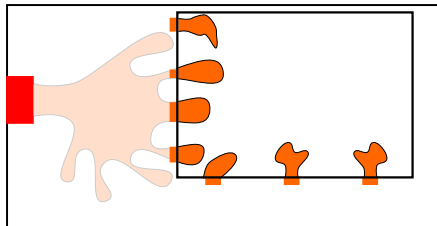


- 解: プログラム実行中にできる「老 → 若」というポインタを「全部」記録しておきそれらをルートと見なす.



## 課題2: 世代別 GC が正しく動作するには?

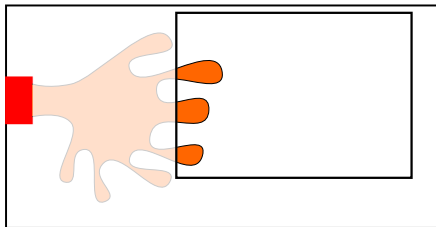
- あくまで見つけるべきは、「ルートから (老若問わず) 全部のポインタをたどって」到達可能な、若いオブジェクト
- 古いオブジェクトを無視するだけではダメ



- 解: プログラム実行中にできる「老 → 若」というポインタを「全部」記録しておきそれらをルートと見なす.
- 注: ルートから到達不能でも, 回収されないことがある.

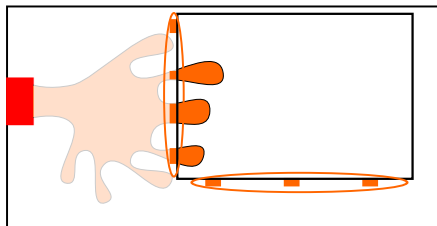
## Problem 2: how to make it correct?

- 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?

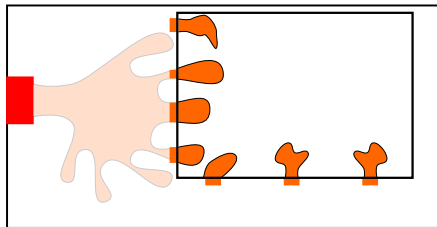
- we need to find all young objects reachable from the root, through “*all pointers, young or old*”
- simply ignoring old objects won't work



- 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

# 書き込みバリア (write barrier)

- すべての「老 → 若」ポインタを捕捉するために mutator のアクションに介入する
- 介入が必要な mutator のアクション:

古い (かもしれない) オブジェクトのフィールド  
← 新しい (かもしれない) オブジェクト

という代入文

- OCaml で言えば,

式の例	説明	補足の要否
<code>o.x &lt;- a</code>	mutable なフィールドの更新	要
<code>{ x = ...; ... }</code>	レコード etc. の生成	不要
<code>let b = o.x</code>	変数の初期化	不要

「ほとんど関数型」な言語では稀にしか発生しないと期待

# 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

# Write Barrier の実装 (1) Remembered Set

- 特に工夫のない write barrier

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

に対し,

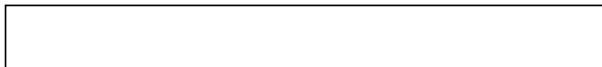
```
1 if (generation(a) < generation(o)) {  
2   if (o ∉ R) add(R, o)  
3 }
```

- Remembered Set 方式
- オーバーヘッド大
  - ▶ `generation(·)` の計算 (コピー GC でもアドレスの比較)
  - ▶  $o \in R$  のチェック
  - ▶  $R$  の管理

# Write Barrierの実装 (2) カードマーキング

- 基本アイデア: 書き込みがあったアドレスを無条件に記録
- ヒープを, 「カード」と呼ばれる連続領域に区切る
  - ▶ カード: アドレスの上位何 bit かが共通の領域
    - ★ 例えば 64 bit アドレスの上位 57 bit が共通
    - ★ → カード 1 枚  $2^7 = 128$  バイト

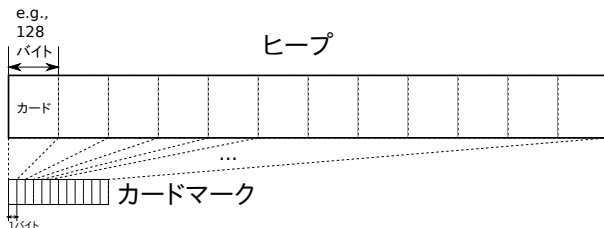
ヒープ





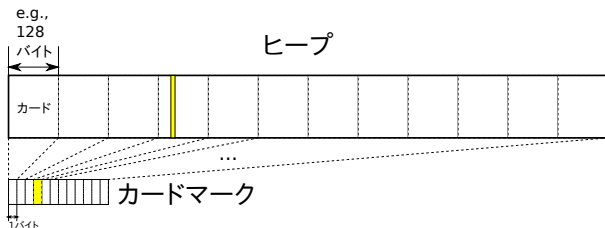
# Write Barrierの実装 (2) カードマーキング

- 基本アイデア: 書き込みがあったアドレスを無条件に記録
- ヒープを, 「カード」と呼ばれる連続領域に区切る
  - ▶ カード: アドレスの上位何 bit かが共通の領域
    - ★ 例えば 64 bit アドレスの上位 57 bit が共通
    - ★ → カード 1 枚  $2^7 = 128$  バイト



# Write Barrierの実装(2) カードマーキング

- 基本アイデア: 書き込みがあったアドレスを無条件に記録
- ヒープを,「カード」と呼ばれる連続領域に区切る
  - ▶ カード: アドレスの上位何 bit かが共通の領域
    - ★ 例えば 64 bit アドレスの上位 57 bit が共通
    - ★ → カード 1 枚  $2^7 = 128$  バイト



- 各カードに書き込みがあったかどうかだけ記録する (1 バイト/カード; カードマーク)

# カードマーキングのオーバーヘッド

- 例: 以下の更新に対し,

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

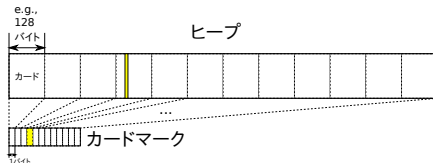
「&o->x を含むカードに書き込みあった」ことを無条件に記録

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

$C$  は, ヒープ先頭アドレス  $\text{heap}$ , カードマークの先頭アドレス  $\text{cards}$  として,

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

を満たすアドレス.



# カードマーキング法の利点と欠点

- write barrier のオーバーヘッド小 ( $C$  をレジスタに保持できれば, 3 命令程度)

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

- カードの大きさ次第で, メモリオーバーヘッド調節可 (例: 128 バイトなら,  $1/128$ )
- 「書き込みがあったカード」だけを効率的に列挙できない. 全カード ( $\propto$  ヒープ) を見る必要がある
- カードのどこかに書き込みがあったら, そのカードすべてをルートと見なす必要がある

# Contents

- ① 世代別 GC / Generational GC
- ② マーク&スイープ GC のトピック
  - 空き領域管理
  - マーク&スイープ GC の性能改善
    - 空き領域管理
    - 遅延スイープ
  - 保守的 GC / Conservative GC
- ③ インクリメンタル GC / Incremental GC
- ④ 参照カウントの変種
  - 遅延参照カウント
  - Sticky 参照カウント
  - 1 bit 参照カウント

# Contents

## ① 世代別 GC / Generational GC

## ② マーク&スイープ GC のトピック

- 空き領域管理
- マーク&スイープ GC の性能改善
  - 空き領域管理
  - 遅延スイープ
- 保守的 GC / Conservative GC

## ③ インクリメンタル GC / Incremental GC

## ④ 参照カウントの変種

- 遅延参照カウント
- Sticky 参照カウント
- 1 bit 参照カウント

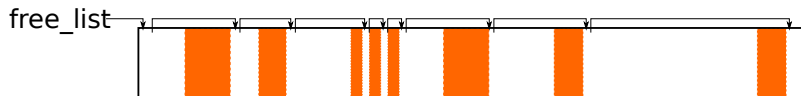
# 空き領域管理・探索

- コピー GC 以外のあらゆる方式 (マーク&スイープ GC, 参照カウント, malloc/free) では, 空き領域は連続していない
- → 空き領域の把握・管理が必要
- 目標:
  - ▶ メモリ割り当て速度: メモリ割り当て要求に応じたサイズの空き領域を素早く見つける
  - ▶ メモリ利用効率: 空いている領域はなるべく使う
- 基本データ構造: フリーリスト (空き領域のリスト)



# 空き領域管理・探索

- コピー GC 以外のあらゆる方式 (マーク&スイープ GC, 参照カウント, malloc/free) では, 空き領域は連続していない
- → 空き領域の把握・管理が必要
- 目標:
  - ▶ メモリ割り当て速度: メモリ割り当て要求に応じたサイズの空き領域を素早く見つける
  - ▶ メモリ利用効率: 空いている領域はなるべく使う
- 基本データ構造: フリーリスト (空き領域のリスト)





# フリーリスト

- 空き領域のリスト

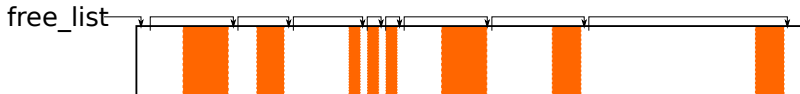
```
1 typedef struct cell {  
2     struct cell * next;  
3     size_t sz;  
4     /* 必要ならばその他の情報 */  
5 } cell;  
6 cell * free_list;
```

- 割り当て (malloc) ≈

- ① 要求サイズ以上の空き領域の (線型) 探索
- ② 領域が余ったら余りをまたリストへ

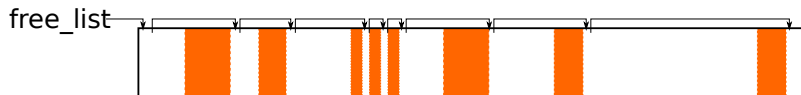
- 回収 (free) ≈

- ① 開放されたセルをリストへ戻す (問題: そのサイズは?)
- ② 開放されたセルとアドレスが隣接しているセルがあったら統合 (coalescing)



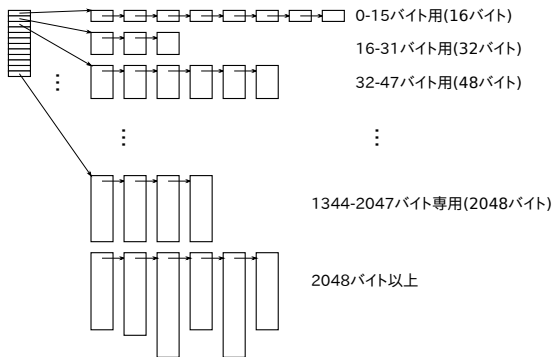
# 単純な方式の問題

- 割り当て:
  - ▶ それなりに走査が必要
  - ▶ → サイズごとに別れたフリーリスト (segregated free lists)
- 回収:
  - ▶ coalescing 可能かを調べるオーバーヘッド
  - ▶ 回収されたセルのサイズ
  - ▶ → 大きな塊 (ページ) 単位で同じサイズ専用にする (Big Bags of Pages; BiBOP)



# サイズごとのフリーリスト

- 小さなサイズ (e.g., 2K バイト以下) について、代表的なサイズのフリーリストを作る
- それ以上大きな空き領域はすべてひとつのリストにまとめる
- 例:
  - ▶ 16, 32, 48, 64, ..., 448, 512, 672, 800, 1024, 1344, 2048
  - ▶ 2048 バイト以上すべて



# 割り当てシーケンスの例

- **色付き部分**: 小さいオブジェクト用の速いパス

- **青部分**: サイズ (sz) がコンパイル時の定数であれば、原理的には (コンパイラの定数畳み込みで) 除去可能な部分

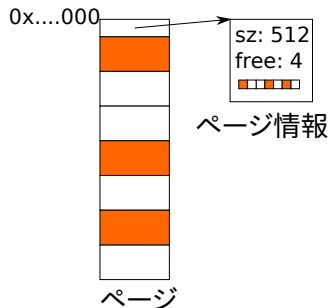
- **赤部分**: 本質部分. リストを一回手繰る (6-7 命令)

- 注: マルチスレッド環境では
  - ▶ `free_lists` をスレッドごとに持たせる, または
  - ▶ **`free_lists` の読み出し～更新**を不可分に (スケューラビリティ障害)

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

# Big Bags of Pages

- ヒープをある固定サイズのブロック (ページ) に分けて管理
  - ▶ ページ: アドレスの上位 bit を共有するアドレスの集合
  - ▶ 例: 64 bit アドレス, 上位 52 bit 共有  $\rightarrow 2^{12} = 4096$  バイト/ページ
- 1 つのページは,
  - ▶ まるごと空き, または
  - ▶ 一つのサイズのオブジェクト専用 (e.g., 48 バイト専用)
- Coalescing: 「1 ページまるごと空いた」場合だけ, 他のサイズに流用
  - ▶  $\rightarrow$  各ページごとに空きセル数を数えるだけで良い
- サイズもオブジェクト毎に持たせる必要なし



# Contents

## ① 世代別 GC / Generational GC

## ② マーク&スイープ GC のトピック

- 空き領域管理
- マーク&スイープ GC の性能改善
  - 空き領域管理
  - 遅延スイープ
- 保守的 GC / Conservative GC

## ③ インクリメンタル GC / Incremental GC

## ④ 参照カウントの変種

- 遅延参照カウント
- Sticky 参照カウント
- 1 bit 参照カウント

# マーク&スイープGCの性能改善

- 全体構成:

- ① マークフェーズ: ルートからポインタをたどり, 到達可能なオブジェクトに印をつける
- ② スweepフェーズ: 印のついていないオブジェクトを回収 → フリーリストに戻す

- 基本的な工夫:

- ▶ サイズごとのフリーリスト
- ▶ BiBOP による管理
- ▶ マークビットとオブジェクトの分離
- ▶ 遅延スイープ

# マークビットとオブジェクトの分離

- マークフェーズで「到達可能」の印をどこにつけるか?
- 方法 1: オブジェクト内の 1 ワード
- 方法 2: オブジェクト外の領域にまとめて持つ
  - ▶ 具体的にはどこ? ページヘッダ. そのページ内のオブジェクトの分をまとめて持つ (1 バイト/オブジェクト)

```
1 mark(void * o) {  
2     page * page = o & 0xFFF...000; /* ページの先頭アドレス */  
3     page->header->mark[(o & 0x000...FFF) >> 4] = 1;  
4 }
```

- ▶ ポイント: 書き込まれるキャッシュラインの数が減る



# 遅延スイープ

- スイープの目的: 空き領域の回収
- 自然には, 空き領域を適切なサイズのフリーリストに戻す (BiBOP 法)
- **遅延スイープ**: フリーリストの構築を, メモリ割り当てで必要になるまで遅らせる

# スweepフェーズの全体像

- マークフェーズ終了後、ページは3種類に分類できる
  - ▶ 空ページ: 到達したオブジェクト 0
  - ▶ 一部空ページ: 到達したオブジェクト  $> 0$ , 到達しなかったオブジェクト  $> 0$
  - ▶ 満杯ページ: 到達しなかったオブジェクト = 0
- Sweepの実際: 素直には,

```
1  for (全ページ p) {  
2      if (p が空) {  
3          p を空ページリストにつなぐ;  
4          /* 任意サイズのページとして再利用可能 */  
5      } else if (p が一部空) {  
6          sz = そのページ内のオブジェクトのサイズ;  
7          p 中の空き領域を sz バイト用のフリーリストにつなぐ;  
8      }  
9  }
```

# 遅延スイープ

- フリーリストの構築をすぐに行わない
- 各サイズ用の「回収リスト」につなぐ

```
1 for (全ページ p) {  
2     if (p が空) {  
3         p を空ページリストにつなぐ;  
4         /* 任意サイズ用の空領域として再利用可能 */  
5     } else if (p が一部空) {  
6         sz = そのページ内のオブジェクトのサイズ;  
7         p を sz バイト用の「回収リスト」につなぐ;  
8     }  
9 }
```

# 回収リスト

- 空き領域が1つ以上含まれる，完全に空ではないページのリスト
- フリーリスト同様，サイズごとに作る
- メモリ割り当て時，フリーリストが空だった時に回収リストからセルを補充

# なぜスweepを遅延させる？

- 仕事を「後回し」にしてるだけ？ それだけではない
- Coalescing の機会向上:
  - ▶ 何回か後の GC までに使われず、それまでに空ページになるかもしれない
- 参照の時間局所性改善:
  - ▶ フリーリスト充填のためのメモリ参照と、実際に mutator に使われるまでの時間を短くする
- Sweepの時間を少しでも短くする

# Contents

## ① 世代別 GC / Generational GC

## ② マーク&スイープ GC のトピック

- 空き領域管理
- マーク&スイープ GC の性能改善
  - 空き領域管理
  - 遅延スイープ
- 保守的 GC / Conservative GC

## ③ インクリメンタル GC / Incremental GC

## ④ 参照カウントの変種

- 遅延参照カウント
- Sticky 参照カウント
- 1 bit 参照カウント

# 保守的 GC (Conservative GC)

- 保守的 GC

- ▶  $\approx$  C/C++など, GC を前提に設計されていない言語用の GC
- ▶  $\approx$  各語がポインタか否かわからない (わからない時は「保守的に」ポインタと見なす) 前提での GC

- 反対語: 正確な (accurate) GC

- ▶ 正確といっても, 「死んでいる (以降アクセスされない)」ゴミが全部とれるわけではない
- ▶ 正確・保守的 GC の意味は「ポインタの判別 (pointer identification)」が正確か保守的かということ
- ▶ 正確な GC を行う言語では例えば, ある語を見ただけでポインタか否かがわかるようなデータ表現を使う
  - ★ 例: 最下位 bit = 0 (ポインタ), = 1 (非ポインタ)

# 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)

## C/C++言語の基本的な困難: ポインタの曖昧さ

- ポインタとそれ以外のデータの厳密な区別がつかない．ある語に書かれていた「7596272344674820427  
(10110100101101011011000010111001101100001011101110110000)は，
  - ▶ 7596272344674820427 番地に割り当てられたオブジェクトへのポインタ？
  - ▶ 整数の 7596272344674820427
  - ▶ 文字列の一部 (“Kawasaki”)
  - ▶ 倍精度浮動小数点 ( $6.549545 \dots \times 10^{199}$ )

のいずれであるかはっきりしない (曖昧)

- 基本:
  - ▶ ある語が、現在割り当て中のアドレス範囲と一致したら、それへのポインタと仮定
  - ▶ 本当はポインタでないものがポインタと間違われることもある
  - ▶ それによる被害(リーク)を防ぐアイデア → [Blacklisting](#)

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

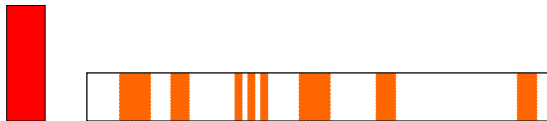
- a pointer and a non-pointers cannot be told apart; a word “7596272344674820427 (10110100101101011011...011000010100101<sub>2</sub>)” can be any of the following
  - ▶ a pointer to an object at address 7596272344674820427,
  - ▶ an integer 7596272344674820427,
  - ▶ a part of a string (“Kawasaki”),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

# 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**

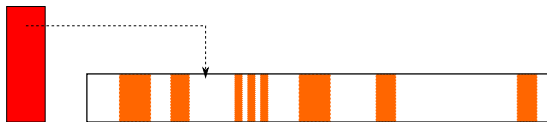
# Blacklisting

- マークフェーズ中に以下のような語  $p$  (「**要注意アドレス**」) を記憶しておく (Blacklisting)



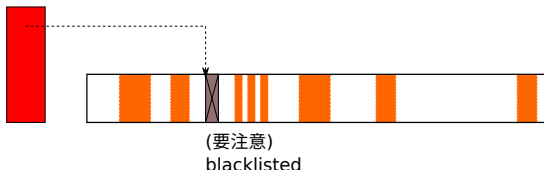
# Blacklisting

- マークフェーズ中に以下のような語  $p$  (「**要注意アドレス**」) を記憶しておく (Blacklisting)
  - ▶  $p$  は現在割り当て中のアドレス**ではない**
  - ▶  $p$  は, 将来割り当て対象になりそう (現在のヒープの一部である)



# Blacklisting

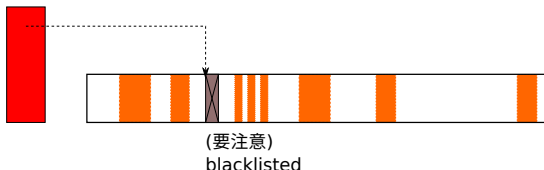
- マークフェーズ中に以下のような語  $p$  (「**要注意アドレス**」) を記憶しておく (Blacklisting)
  - ▶  $p$  は現在割り当て中のアドレス**ではない**
  - ▶  $p$  は, 将来割り当て対象になりそう (現在のヒープの一部である)
- そのような  $p$  は**当面, 割り当ての対象から外す**





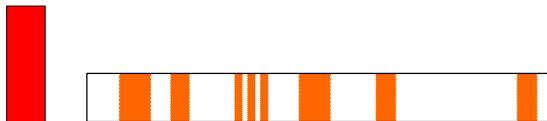
# Blacklisting

- マークフェーズ中に以下のような語  $p$  (「**要注意アドレス**」) を記憶しておく (Blacklisting)
  - ▶  $p$  は現在割り当て中のアドレス**ではない**
  - ▶  $p$  は, 将来割り当て対象になりそう (現在のヒープの一部である)
- そのような  $p$  は**当面, 割り当ての対象から外す**
- 「 $p$  が割り当てられない」時点で損害だが,  $p$  を割り当てて,  $p$  およびそこから到達可能なものが全て回収できなくなるのは大損害



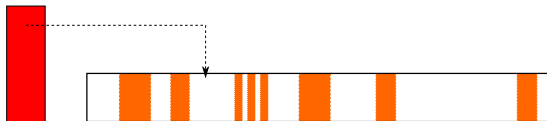
# Blacklisting

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



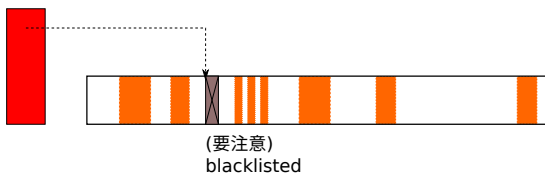
# Blacklisting

- 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)



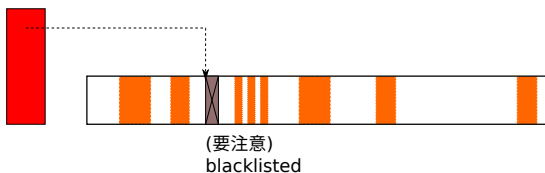
# Blacklisting

- 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*)



# その他の保守的 GC の tips

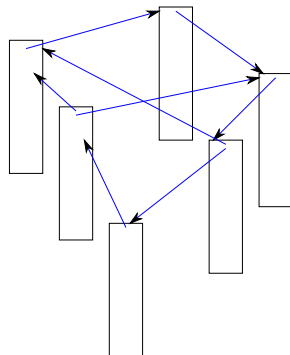
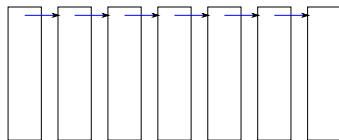
- <http://hboehm.info/gc/gcinterface.html> 参照
- GC\_MALLOC\_ATOMIC :
  - ▶ GC\_MALLOC と同じだが、「この中にポインタは含めません」宣言 (文字列, 数値の配列など)
  - ▶ ポインタ誤認の確率を減らす
  - ▶ mark の量を減らす
- GC\_MALLOC\_IGNORE\_OFF\_PAGE : 「最初の 512 バイト以降にはポインタは含めません」宣言
- いらないデータに対するポインタはこまめに NULL に
  - ▶ データ構造中のポインタ
  - ▶ 一つが誤認されて生き残ったときの「道連れ」をなくす
- データのリンク構造の工夫
  - ▶ ポインタの誤認によって「たくさんが道連れに」なりにくデータ構造

# 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

# 道連れになりやすい・にくいデータ構造

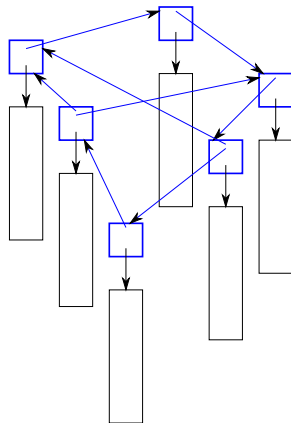
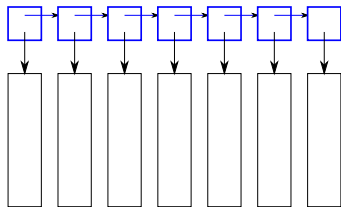
- リンクリスト, 木構造, グラフなどの作り方
- (NG): 大きな構造体同士が直接リンクしあう
- (GOOD): リンク構造 (峰) と (大きな) データ本体を分ける → 本体が誤認されても道連れはおきない





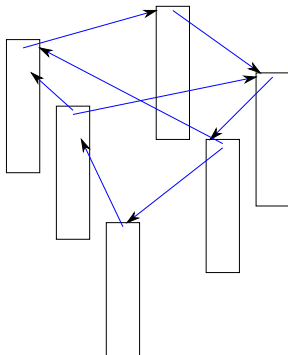
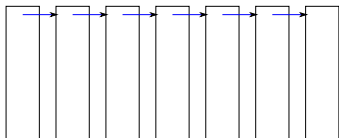
# 道連れになりやすい・にくいデータ構造

- リンクリスト, 木構造, グラフなどの作り方
- (NG): 大きな構造体同士が直接リンクしあう
- (GOOD): リンク構造 (峰) と (大きな) データ本体を分ける → 本体が誤認されても道連れはおきない



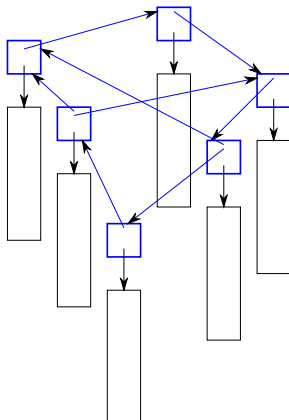
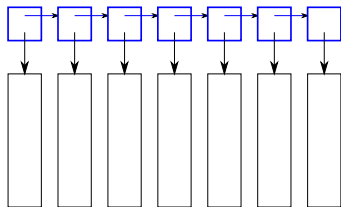
# Data structure (not) prone to the domino effect

- 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

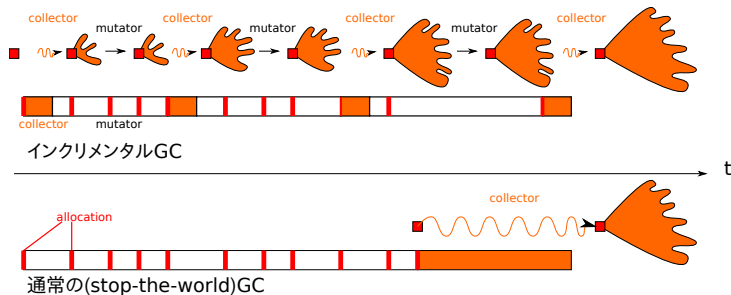


# Contents

- ① 世代別 GC / Generational GC
- ② マーク&スイープ GC のトピック
  - 空き領域管理
  - マーク&スイープ GC の性能改善
    - 空き領域管理
    - 遅延スイープ
  - 保守的 GC / Conservative GC
- ③ インクリメンタル GC / Incremental GC
- ④ 参照カウントの変種
  - 遅延参照カウント
  - Sticky 参照カウント
  - 1 bit 参照カウント

# インクリメンタルGC

- 走査型 GC の「停止時間」を短くする GC
  - ▶ 実時間性や応答性を必要とするアプリケーション
- 停止時間  $\approx$  到達可能オブジェクトすべてをマークする時間
- インクリメンタル GC は、到達可能オブジェクトの走査を、「細切れ」に行って、停止時間を短くする
  - ▶ 1GB「一気に」走査せずに、10MB ずつ 100 回に分けて走査

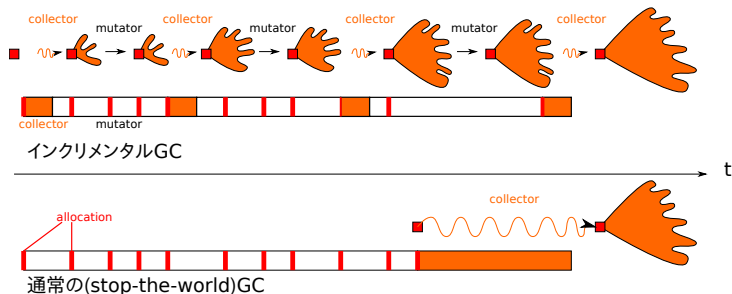


# Incremental GC

- 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

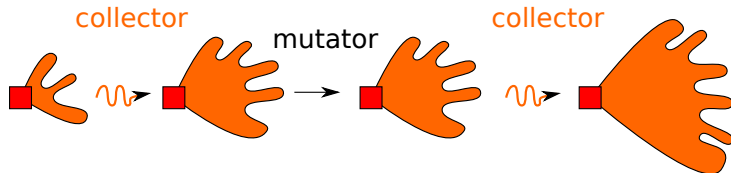
# Incremental GC

- 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



# インクリメンタルGCの難しさ

- (GC 目線で) GC がグラフを走査している間に，グラフが変化する

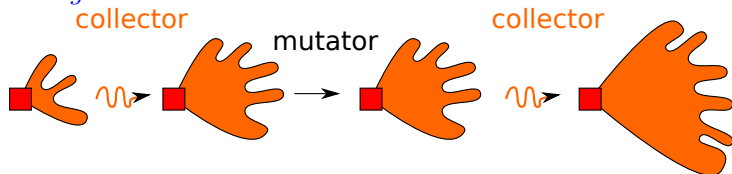


- それでも、「ルートから到達可能」なオブジェクトを見逃さないためには？
- グラフ走査の基本に立ち返って考える
- 以降の前提:
  - ▶ mutator は 1 つ (1 スレッドのアプリケーション)
  - ▶ mutator と collector は同じスレッド内で「交互に」動く
    - ★ メモリ割り当て要求時に GC が少し動く
  - ▶ → 両者が別スレッドであればおきるような競合は考えない



# Challenges in incremental GC

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

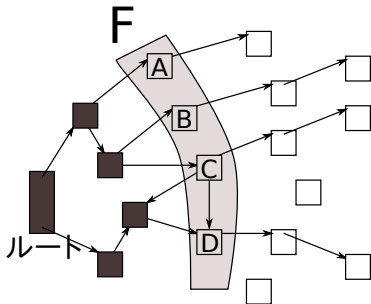


- how to guarantee it finds all “reachable objects” nevertheless?
- we'll get back to the basics of graph traversal
- assumptions for later discussions:
  - ▶ only a single mutator (the app is single-threaded)
  - ▶ the mutator and the collector run “*alternately*” (*not at the same time*)
    - ★ the collector does a little bit of its work upon a memory allocation
  - ▶ → we do not consider race conditions that would happen when they are truly concurrent

# グラフの走査：基本

- マーク&スweepでもコピーでも、本質的には同じ
- データ構造の詳細を省略して書くと：

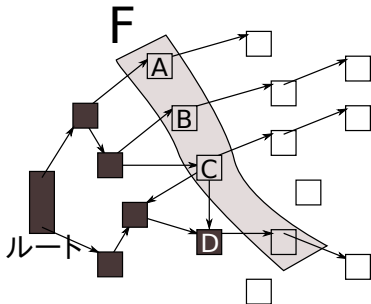
```
1  F = { ルート };  
2  while (F が空でない) {  
3      o = pop(F);  
4      for (o 内のポインタ p)  
5          if (!marked(p)) {  
6              mark(p);  
7              add(F, p);  
8          }  
9  }
```



# グラフの走査：基本

- マーク&スweepでもコピーでも、本質的には同じ
- データ構造の詳細を省略して書くと：

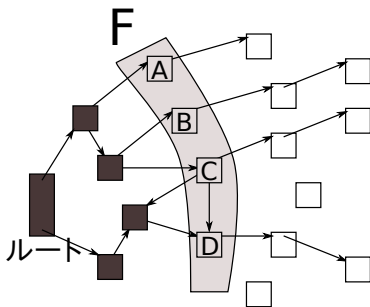
```
1  F = { ルート };  
2  while (F が空でない) {  
3      o = pop(F);  
4      for (o 内のポインタ p)  
5          if (!marked(p)) {  
6              mark(p);  
7              add(F, p);  
8          }  
9  }
```



# 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  }
```



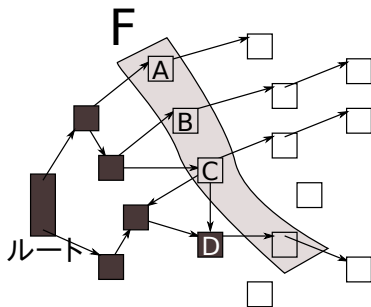
# Graph traversal : basics

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

```

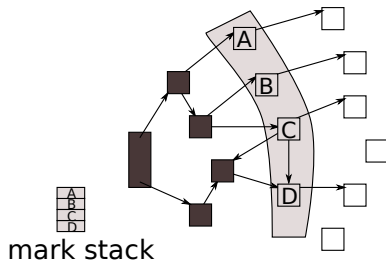
1  F = { root };
2  while (F is not empty) {
3      o = pop(F);
4      for (all pointers p in o)
5          if (!marked(p)) {
6              mark(p);
7              add(F, p);
8          }
9  }

```

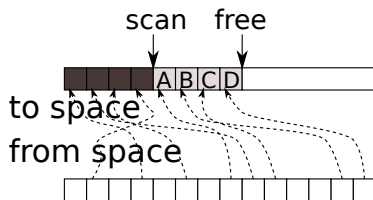


# 鍵となるデータ: 最前線

- $F$ : 最前線 (frontier)
- 自身は訪問されたが、子供はまだ訪問されていない (可能性がある) オブジェクトの集合
- 実際のデータ構造
  - ▶ マーク&スweep: マークスタック
  - ▶ コピー GC: to space の一部



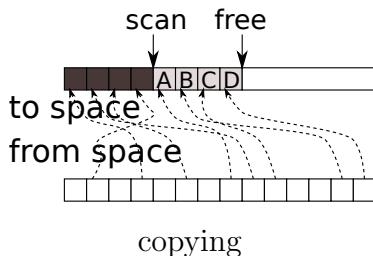
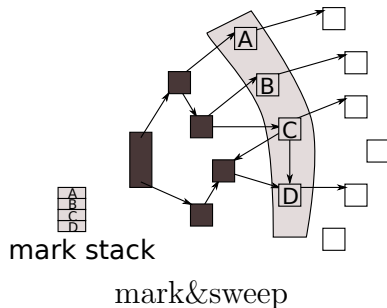
マーク&スweep



コピー

# 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



# インクリメンタルGCが解決すべき問題

```
F = { ルート };  
while (F が空でない) {  
  o = pop(F);  
  for (o 内のポインタ p)  
    if (!marked(p)) {  
      mark(p);  
      add(F, p);  
    }  
  if (何度か回った)  
    // この間にグラフが書き換わる  
    resume_mutator();  
}
```

- 通常の GC : 上記 while ループが終了するまで mutator が動かない → オブジェクトグラフも書き換わらない
- インクリメンタル GC :
  - ▶ 上記 while ループを何度か回ったら, mutator に制御を返す
  - ▶ ... またしばらくしたら続きをやる
  - ▶ つまり, while ループの iteration 間でグラフが書き換わっていることにどう対処するかという問題



# The issue that an incremental GC must address

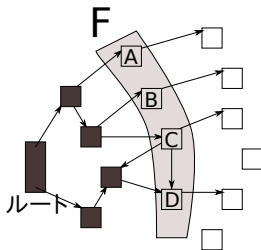
```
F = { root };
while (F is not empty) {
  o = pop(F);
  for (all pointers p in o)
    if (!marked(p)) {
      mark(p);
      add(F, p);
    }
  if (has iterated a few times)
    // the graph changes below
    resume_mutator();
}
```

- **ordinary GC:** the while loop runs until the end keeping the mutator stopped  $\rightarrow$  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*

# 3色 (tri-color) アブストラクション

- 走査を，オブジェクトに「色を塗る」操作で例える
  - ▶ **黒**：自分も，その子供も訪問された
  - ▶ **灰色**：自分は訪問された，子供はまだ ( $\iff$  自分  $\in F$ )
  - ▶ **白**：自分も訪問されていない
- 3色を用いたグラフ走査の記述

```
ルートを灰色に;  
while (灰色がいる) {  
  o = 灰色オブジェクトを一個選び, 黒に塗る;  
  for (o 内のポインタ p)  
    if (p が差すオブジェクト白)  
      それを灰色に塗る;  
  mutator によるグラフ書き換え;  
}
```

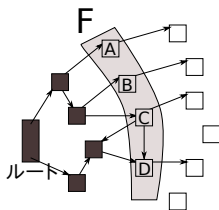


- アルゴリズムの正しさ:  
灰色がなくなった時，ルートから到達可能なオブジェクトは全て黒 (白は到達不能)

# The tri-color abstraction

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

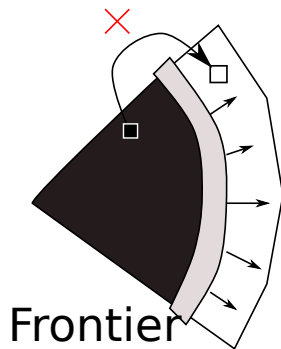
```
gray the root;  
while (there is a gray object) {  
  o = pick a gray object and blacken it;  
  for (all pointers in o)  
    if (p points to a white object)  
      gray it;  
  the mutator changes the graph; }  
}
```



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

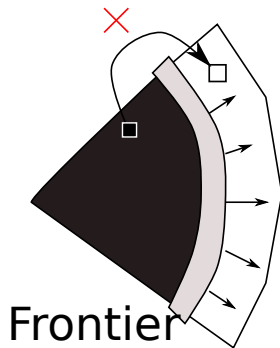
# 問題が生ずるグラフの書き換え

- 直感的には問題は、「黒 → 白」というポインタを作られてしまうこと
  - ▶ 黒: GC がすでに「処理済み」と思っているオブジェクト
  - ▶ 白: このまま発見されなければ回収されてしまうオブジェクト
- ⇒ 「黒 → 白ポインタ」が決して出来ないようにする



# A problematic mutation to the graph

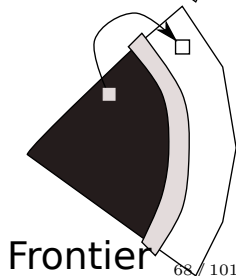
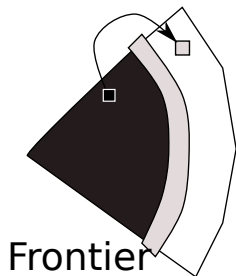
- intuitively, *the issue seems the mutator may create “black  $\rightarrow$  white” pointers*
  - `black` : GC thinks it has “done” with them
  - `white` : going to be reclaimed, unless found in other paths
- $\Rightarrow$  prevent “black  $\rightarrow$  white pointers” from being created



# 黒 → 白を防ぐアプローチ

「黒 → 白」ができる瞬間を捕捉

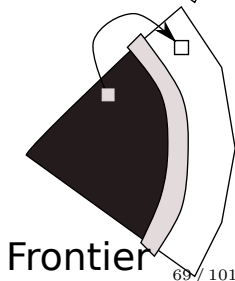
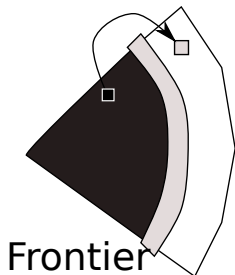
- ① 方法 1: 白 を灰色にする (黒 → 灰色 にする)
  - ▶ 利点: 最前線が常に「前進」する
  - ▶ 利点: コピー GC と共存しやすい (後述)
  - ▶ 欠点: 回収率が低い.  $o$  に再び書き込みが起きて  $p$  が到達不能になっても, (その GC では) 回収されない
- ② 方法 2: 黒 を灰色に戻す (灰色 → 白 にする)
  - ▶ 利点: 回収率が高い
  - ▶ 欠点: 最前線が「後退」する



# Two approaches to preventing black $\rightarrow$ white

capture the point where “black $\rightarrow$ white” is about to be created

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



# 捕捉が必要なアクション

素直にはポインタが動くあらゆるアクションを捕捉する必要がある

- オブジェクトのフィールドへポインタを書き込む (write barrier)

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

- ルートへポインタを書き込む  $\equiv$  変数へポインタを書き込む (read barrier)

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

後者は頻度が大きいので避けるケースも有り (実例 2: Boehm GC)



# 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
```

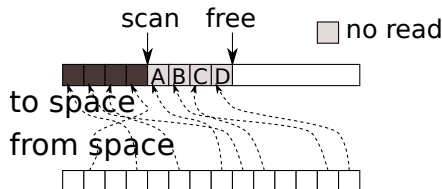
- write a pointer into a root  $\equiv$  write a pointer to a variable (read barrier)

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

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

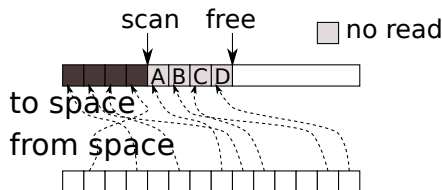
# 実例 1: Appel-Ellis-Li

- コピー GC + インクリメンタル
- 方法 1 に基づく．より正確には以下を保って動く：  
*mutator* は決して白へのポインタを入手しない
- どのように保つか？
  - ▶ 灰色オブジェクトのフィールド読み出しを補足 (read barrier)
- 灰色オブジェクトがいる領域  $\subset$  scan ~ free からの読み出しを仮想記憶のページ保護を用いて補足



# Example #1: Appel-Ellis-Li

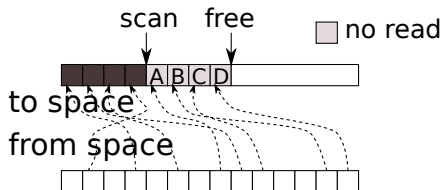
- copying GC + incremental
- based on the approach # 1. more precisely, maintain the following invariant  
*the mutator never sees a pointer to white*
- how?
  - ▶ intervene in reading a field from gray objects (read barrier)
- read-protect the region of gray objects  $\subset$  scan  $\sim$  free, by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



# Appel-Ellis-Li : Read Barrier の実際

- 灰色オブジェクトのフィールドが読み込まれたら、それを含むページ内のオブジェクトを黒にする (= それらのオブジェクトをスキャンする → それらが指しているオブジェクトは灰色になる)

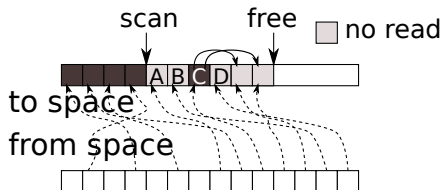
```
1 trap_read_from_grey(a) {  
2   page = a を含むページ;  
3   for (page に含まれるオブジェクト o) {  
4     scan(o); // o の子供をコピー  
5   }  
6   unprotect(page);  
7 }
```



# Appel-Ellis-Li : Read Barrier の実際

- 灰色オブジェクトのフィールドが読み込まれたら、それを含むページ内のオブジェクトを黒にする (= それらのオブジェクトをスキャンする → それらが指しているオブジェクトは灰色になる)

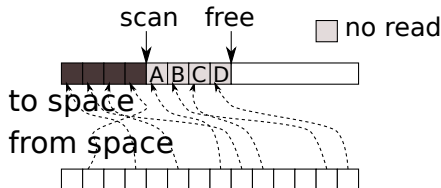
```
1 trap_read_from_grey(a) {  
2   page = a を含むページ;  
3   for (page に含まれるオブジェクト o) {  
4     scan(o); // o の子供をコピー  
5   }  
6   unprotect(page);  
7 }
```



# 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)

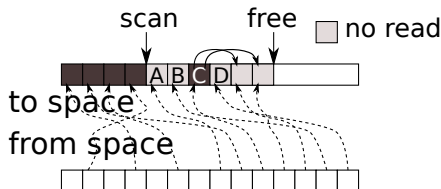
```
1 trap_read_from_grey(a) {  
2   page = the page including a;  
3   for (all objects o in the page) {  
4     scan(o); // copy o's children  
5   }  
6   unprotect(page);  
7 }
```



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

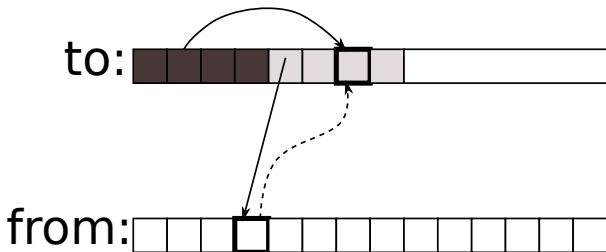
- when a field of a gray object is read, blacken objects in the page containing it (= scan those objects → they become gray)

```
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 }
```



## 補足：コピーGCとの相性

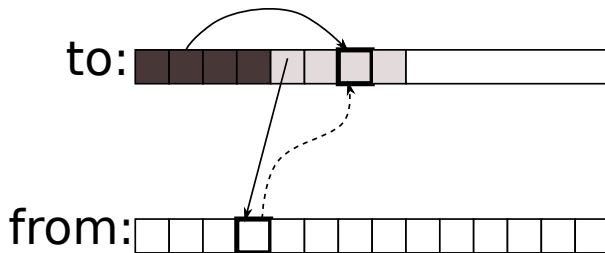
- コピーGCにおいては、GC中コピー済み(灰または黒)オブジェクトのコピーが2つ存在する(from spaceとto space)
- immutableなオブジェクトならどちらを見ても問題はないが、mutableなオブジェクトの場合「どちらを見るか」を決める必要がある
- いずれはto spaceを見るのだから、「from spaceは決して見ない」とするのが自然
- → 白(from space)ポインタ自体をmutatorに渡さない(見せない)とするのが自然





## 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



## 実例 2: Boehm GC

- 保守的 GC (→ マーク&スイープ) + インクリメンタル
- 保つ条件:
  - ▶ 「ルートでない黒オブジェクト → 白」ポインタを作らない
- どうやって?
  - ▶ 「オブジェクトのフィールドに対する書き込み」を捕捉 (write barrier)
- 注: 「ルート (黒かもしれない) → 白」というポインタはできうる
  - ▶ 防ごうと思うと, ルートへの書き込み → オブジェクトの読み出しを捕捉する必要がある
  - ▶ オーバーヘッドが大きすぎるので, 別途対処 (後述)

## 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)

# Boehm GC の write barrier

- やはり仮想記憶を用い (C/C++用では唯一の選択肢), オブジェクトへの書き込みを補足
- 書き込まれたオブジェクトを灰色へ戻す
  - ▶ マークスタックへ挿入
- Read barrier は行わない → 「ルート (黒) → 白」というポインタは許容 (放置) する
- マークフェーズの終わりに改めて, ルートから走査し直す (再走査)
- 再走査中は mutator を止める → 潜在的には停止時間が長くなる

# 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

## 補足: もう少し厳密な正しさの証明

- 「黒 → 白」というポインタが問題なのはわかるが、それさえ防げば OK というのは自明ではない
- 改めて、示すべき命題: 以下のアルゴリズム終了時点で、  
ルートから到達可能 → 黒

```
1 ルートを灰色に;  
2 while (灰色がいる) {  
3     o = 灰色オブジェクトを一個選び, 黒に塗る;  
4     for (o 内のポインタ p)  
5         if (p が差すオブジェクト白)  
6             それを灰色に塗る;  
7     mutator によるグラフ書き換え;  
8 }
```

## 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;  
8 }
```

# 鍵となる条件

- GC, mutator の実行中「常に」以下が成り立つ  
(I): ルートから到達可能な「白」オブジェクトは, ある「灰色」オブジェクトから到達可能
- これが言えたとすると,
  - (I) かつ終了状態 (灰色は存在しない)
    - ルートから到達可能な「白」オブジェクトは存在しない
    - 「白」オブジェクトは回収可能となり正しさが示される. 後は (I) を示せばよい



# 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).

# (I) の証明

- $w$  をルートから到達可能な白オブジェクトとする



- ルートは黒または灰色で，かつ黒  $\rightarrow$  白ポインタは存在しない (\*) から，ルートから  $w$  に至るパス  $P$  中に，灰色オブジェクトがひとつ以上ある．証明終わり．



- \*: mutator だけでなく，GC が「黒  $\rightarrow$  白ポインタを作らない」ことも証明の必要があるがそれは易しいので省略

# 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

- ① 世代別 GC / Generational GC
- ② マーク&スイープ GC のトピック
  - 空き領域管理
  - マーク&スイープ GC の性能改善
    - 空き領域管理
    - 遅延スイープ
  - 保守的 GC / Conservative GC
- ③ インクリメンタル GC / Incremental GC
- ④ 参照カウントの変種
  - 遅延参照カウント
  - Sticky 参照カウント
  - 1 bit 参照カウント

# 参照数変更が必要な操作

- ポインタが変数に代入 (束縛) されるあらゆる場面で必要
- 注: 以下は NULL ポインタの検査を省略している

- 変数の代入

```
1 p = e; /* e->rc++; p->rc--; if (p->rc == 0) free(p); */
```

- 変数初期化

```
1 void * p = e; /* e->rc++; */
```

- オブジェクトのフィールドへ代入

```
1 o->p = e; /* e->rc++; o->p->rc--; if (o->p->rc == 0) free(o->p); */
```

# 参照数変更が必要な操作

- 関数への引数渡し

```
1 f(e); /* e->rc++ */
```

- 変数が dead になる (以降使われなくなる)

```
1 foo() {  
2     object * p = e;  
3     ... p->x ...  
4     /* 以降p は dead(使われない).  
5         p->rc--; if (p->rc == 0) free(p) */  
6 }
```

# 参照カウントのオーバーヘッド

- つまり、オブジェクトを書き換えていなくても、参照数はしょっちゅう書き換わる

```
1 list * p = 0;  
2 for (p = head; p; p = p->next) {  
3     ...  
4 }
```

# Contents

- ① 世代別 GC / Generational GC
- ② マーク&スイープ GC のトピック
  - 空き領域管理
  - マーク&スイープ GC の性能改善
    - 空き領域管理
    - 遅延スイープ
  - 保守的 GC / Conservative GC
- ③ インクリメンタル GC / Incremental GC
- ④ 参照カウントの変種
  - 遅延参照カウント
  - Sticky 参照カウント
  - 1 bit 参照カウント



# 遅延参照カウント (Deferred Reference Counting)

- 要点: ルート (局所変数・大域変数) からの参照を数えない. オブジェクトからの参照だけを数える
- 利点: オブジェクトのフィールドへの代入時だけ, 参照数を変更すればよい
- 欠点: 参照数フィールド (rc) は, ルートからの参照を含まないため, 「rc == 0 → 到達不能」は成り立たない
- → rc == 0 → ゴミ候補として, 表 (Zero Count Table; ZCT) へ登録
- 時折ルートをスキャンして, 本当に参照数 0 の (ZCT にあり, ルートからさされていない) ものを回収

# 遅延参照カウント

- オブジェクトのフィールド更新時

```
1 o->p = e;
```

```
1 if (o->p) {  
2   o->p->rc--;  
3   if (o->p->rc == 0) add(o, ZCT);  
4   e->rc++;  
5   remove(e, ZCT);  
6 }
```

- 時折 (e.g., メモリ割り当て時に空き領域が見つからない)

```
1 R = ルート中のポインタ全て;  
2 ZCT に含まれ, R に含まれないオブジェクトを開放;
```

# Contents

- ① 世代別 GC / Generational GC
- ② マーク&スイープ GC のトピック
  - 空き領域管理
  - マーク&スイープ GC の性能改善
    - 空き領域管理
    - 遅延スイープ
  - 保守的 GC / Conservative GC
- ③ インクリメンタル GC / Incremental GC
- ④ 参照カウントの変種
  - 遅延参照カウント
  - Sticky 参照カウント
  - 1 bit 参照カウント

# Sticky 参照カウント

- 参照数は最大いくつになりうるのか?

$$\text{理論的な最大値} = \frac{\text{ヒープサイズ}}{\text{ポインタサイズ}}$$

ポインタとほぼ同じだけの bit 数 (e.g., 64 bit) が必要

- Sticky 参照カウント
  - ▶ 参照数の bit 数を制限 (e.g., 5 bit. 1 ... 31, および “32 以上 (sticky)”)
  - ▶ 一旦オーバーフローしたら (“sticky” になったら) 参照カウントは, sticky のまま
  - ▶ 走査型 GC との併用が前提 (循環ゴミを回収するのにどのみち必要)
  - ▶ 一旦 sticky になったら, 走査型 GC 後に本当の参照数が回復できる (その時点でオーバーフローしていなければ)

# Contents

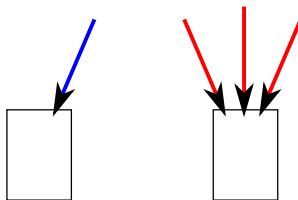
- ① 世代別 GC / Generational GC
- ② マーク&スイープ GC のトピック
  - 空き領域管理
  - マーク&スイープ GC の性能改善
    - 空き領域管理
    - 遅延スイープ
  - 保守的 GC / Conservative GC
- ③ インクリメンタル GC / Incremental GC
- ④ 参照カウントの変種
  - 遅延参照カウント
  - Sticky 参照カウント
  - 1 bit 参照カウント

# 1 bit 参照カウント

- sticky 参照カウントで、参照数を 1 bit にしたもの
- 1 bit = 参照数 1 または 2 以上 (sticky) の区別
- さらに、参照数はオブジェクトに持つ必要はなく、「ポインタ内」に持たせることができる

# 1 bit 参照カウントの基本

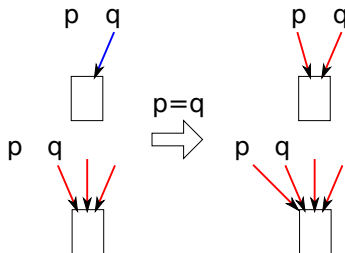
- 用語
  - ▶ **unique** ポインタ: 参照数 1 のオブジェクトへのポインタ
  - ▶ **shared** ポインタ: 参照数 2 以上のオブジェクトへのポインタ
- 不変条件: あるオブジェクトを指すポインタの数は以下のどちらか
  - ▶ **unique** 1 個 + **shared** 0 個
  - ▶ **unique** 0 個 + **shared** 任意個



# 1 bit 参照カウンタの動き (1)

- 代入:

```
1 p = q;
```



- unique/shared をポインタの最下位ビットで区別 (unique=0, shared=1) するとする

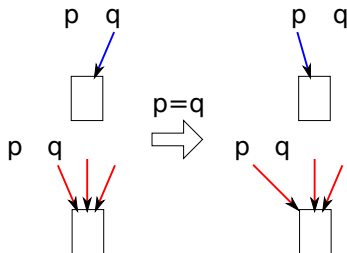
```
1 if ((p & 1) == 0) free(p);  
2 p = q = (q | 1);
```

- 正しいが，これでは回収できるオブジェクトは稀



## 1 bit 参照カウンタの動き (2)

- 変数  $q$  が代入  $p = q$  以降  $\text{dead}$ (参照されない) とすると、代入に際して特別な操作は不要



```
1 if ((p & 1) == 0) free(p);  
2 p = q;
```