Сборка мусора в Java и Go

для прикладных разработчиков

МИР Plat.Form

Прогревочная

t.me/pro_grevy

Обо мне

linkedin.com/in/alantsov

Мир Plat.Form

mir-platform.ru

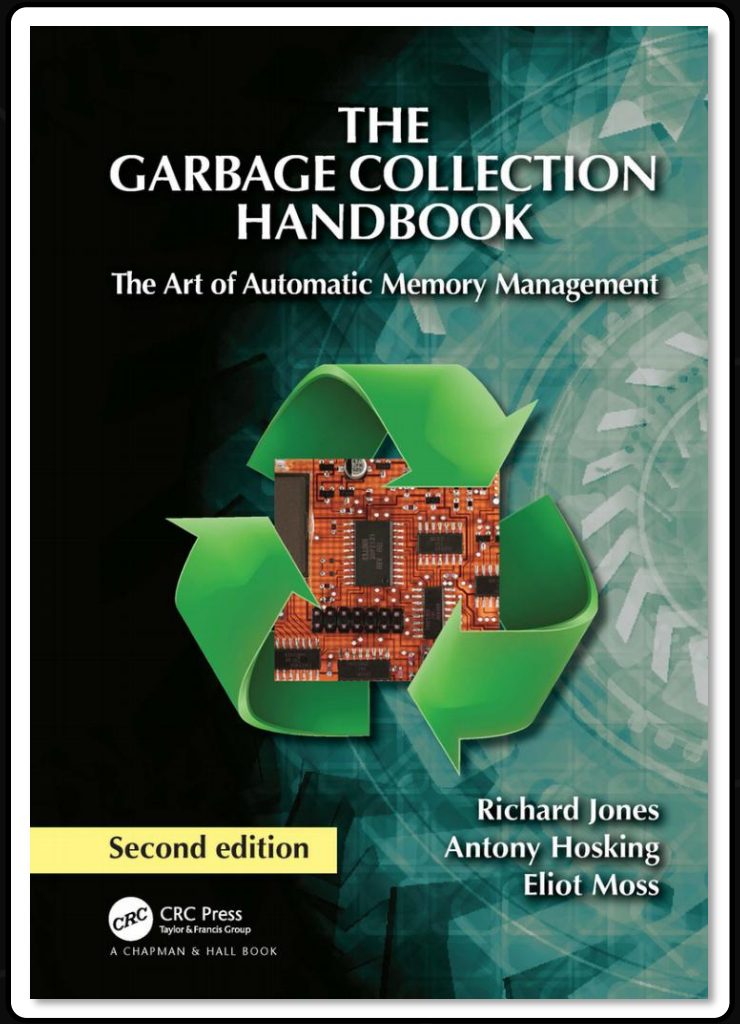
Польза прикладному разработчику?

- **Понимание принципов работы GC**
  не делаем сборщику мусора больно
- **Выбор алгоритма GC**
  актуализация: многие описания устаревшие
- **Сравнить Java и Go**
  разные инженерные подходы

- **Фундаментальные алгоритмы**
  и некоторые прикладные выводы из них
- **STW-коллекторы**
  Java: Serial & Parallel, древний Go
- **Конкурентная маркировка**
  Java G1; Go 1.5: Concurrent Mark Sweep
- **Современные конкурентные GC**
  Java: Shenandoah, ZGC; Go: Green Tea GC
- **Какой алгоритм самый лучший?**

Часть 1
Фундаментальные алгоритмы

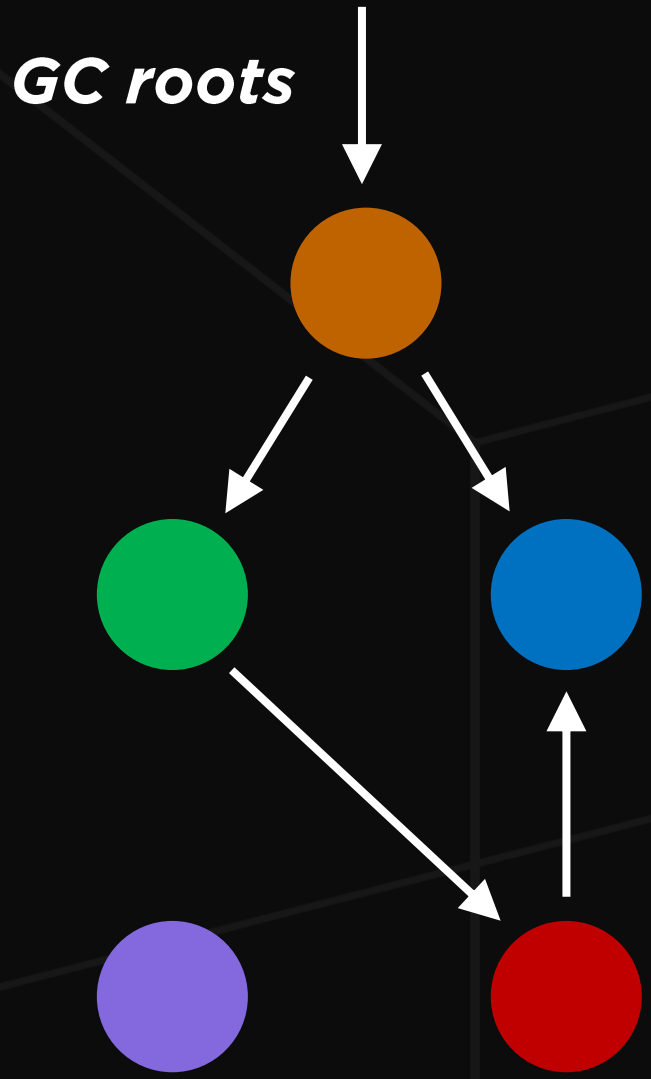
Четыре подхода к GC

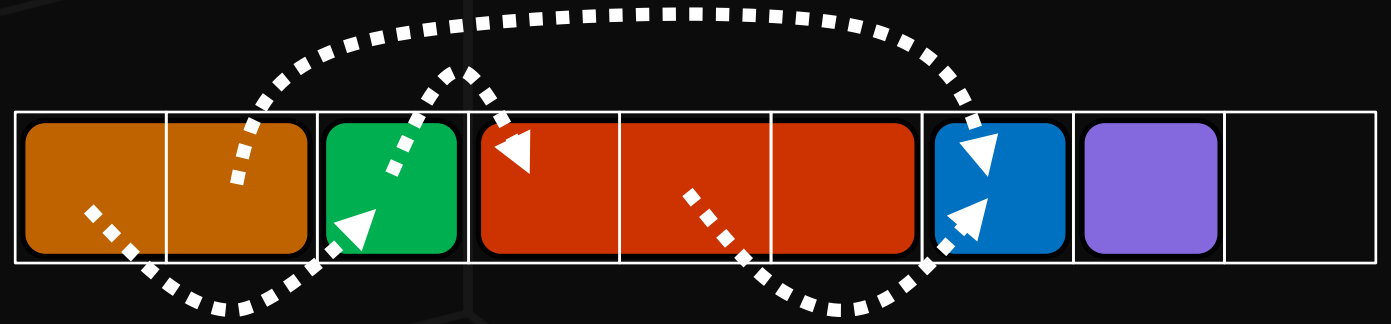


- ~~Reference counting~~
- Mark-Sweep
- Mark-Compact
- Mark-Evacuate

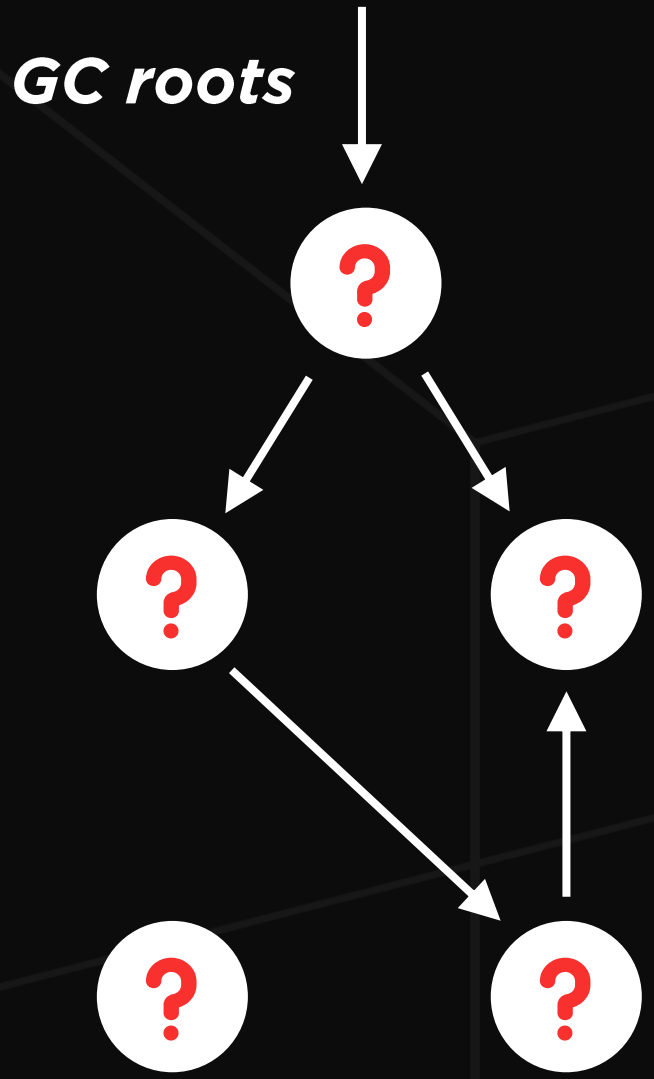
пока что:
останавливаем приложение при GC
aka “stop-the-world”

Маркировка (tri-color)





Маркировка (tri-color)

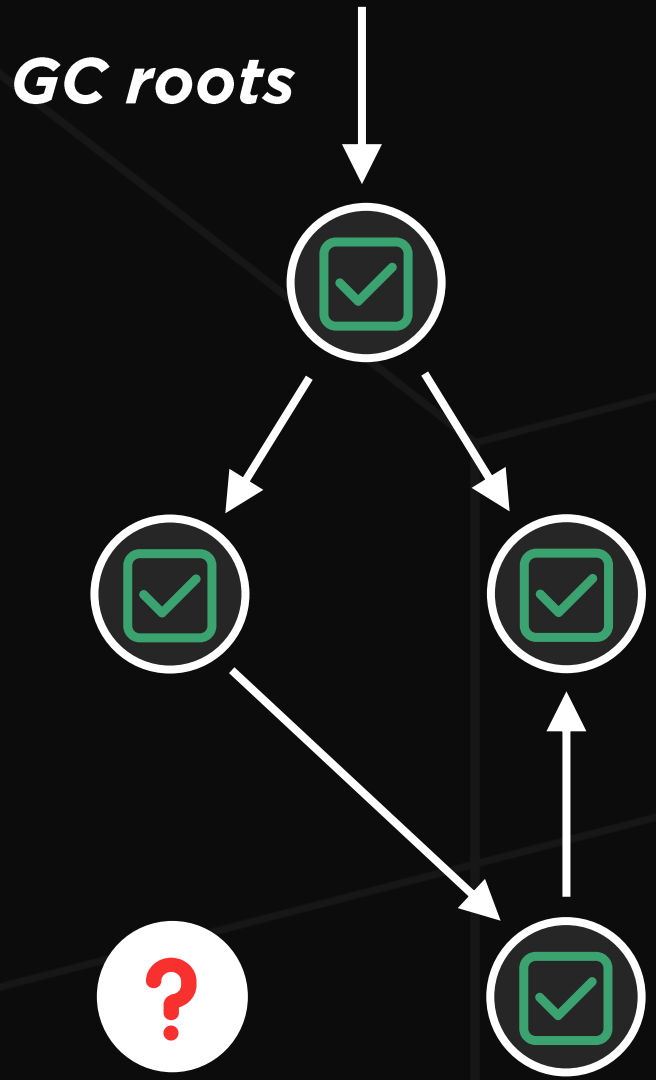


- пока не встречали

- объект живой, но его исходящие указатели не обработаны полностью

- объект живой и обработан

Маркировка (tri-color)

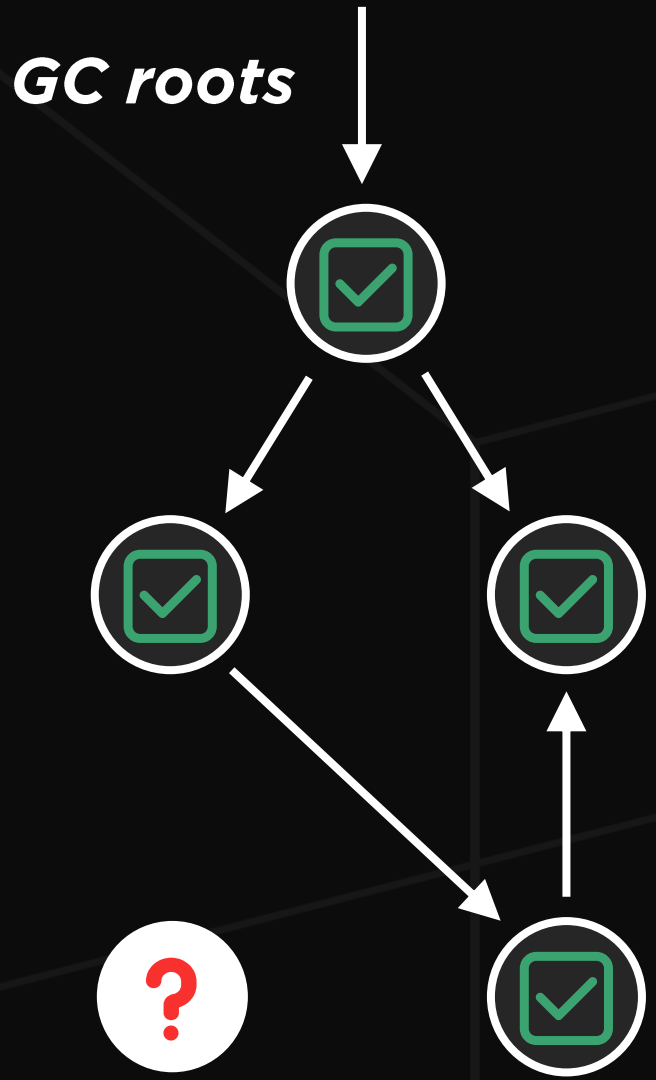


- пока не встречали

- объект живой, но его исходящие указатели не обработаны полностью

- объект живой и обработан

Маркировка (tri-color)



? - это мусор

- объект живой

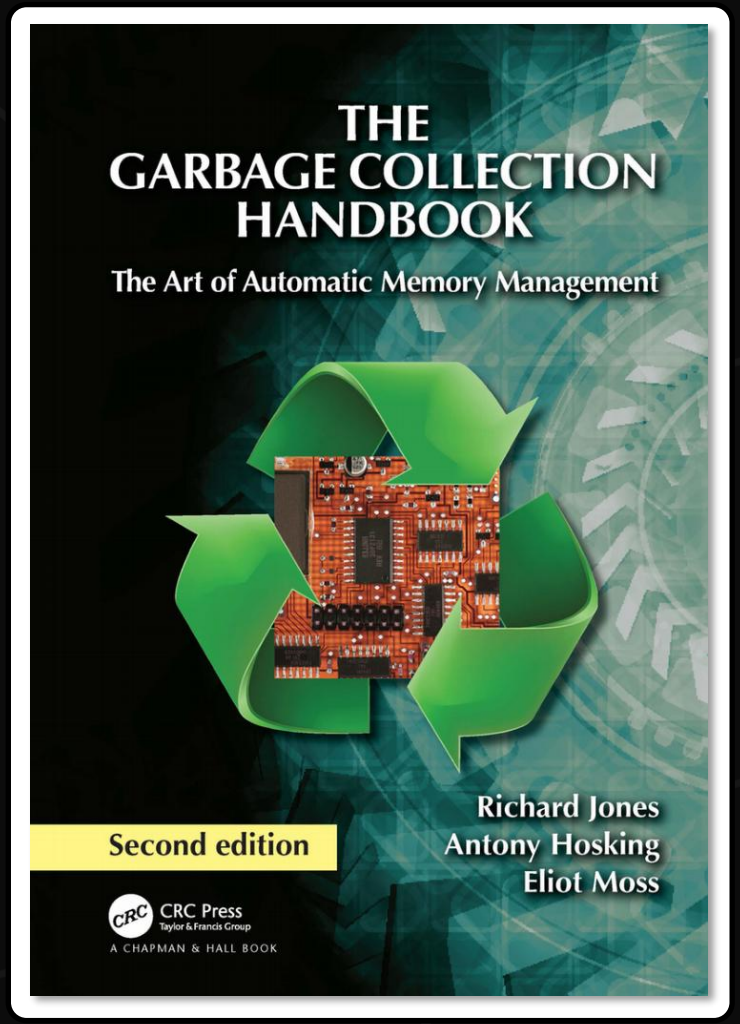
инвариант: запрещены → ?

Маркировка

- **Пропорциональна количеству живых объектов на момент GC**
  потенциальная ситуация: spiral of death

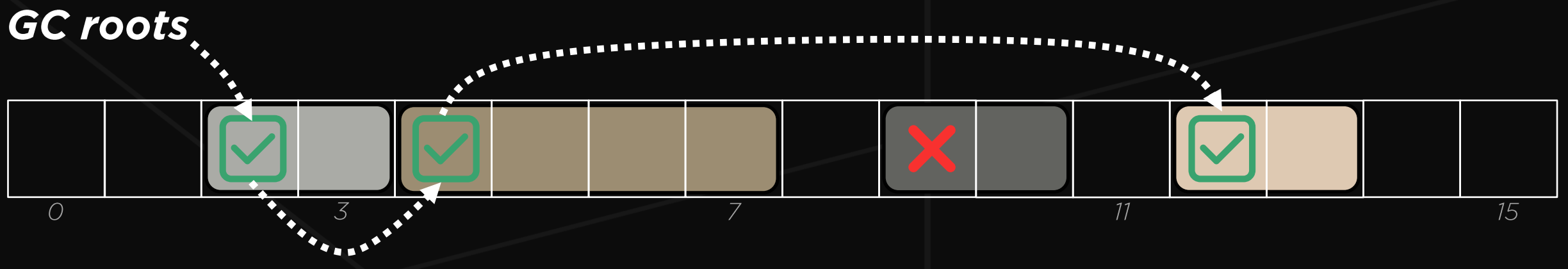
- **Нужно где-то хранить информацию об достижимости объекта**
  mark word, bitmap, 64 bits pointer

Четыре подхода к GC



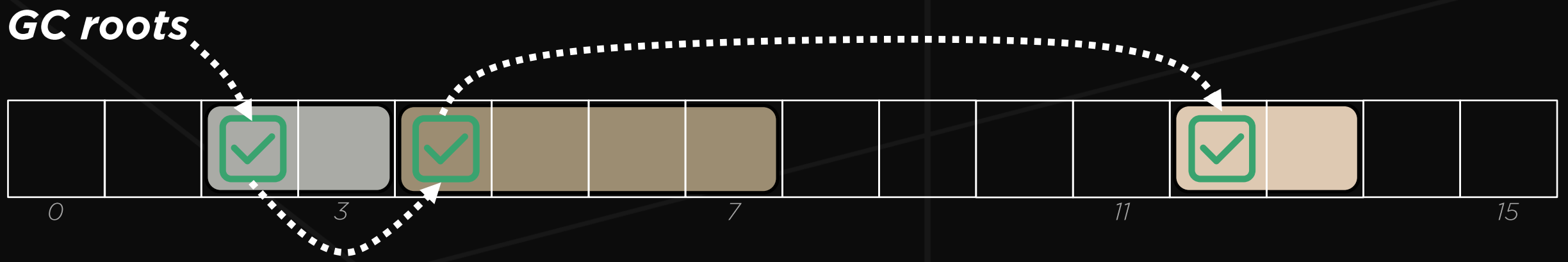
- ~~Reference counting~~
- Mark-Sweep
- Mark-Compact
- Mark-Evacuate

Mark-Sweep



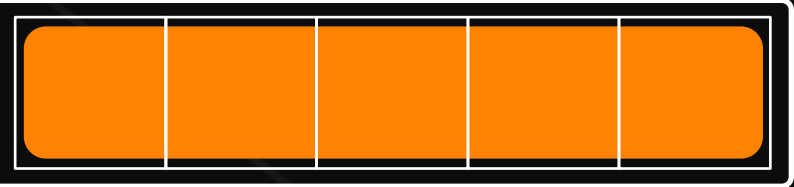

```
free blocks list:  [0-1], [8], [11], [14-15]
```

Mark-Sweep



free blocks list: [0-1], [8-11], [14-15]

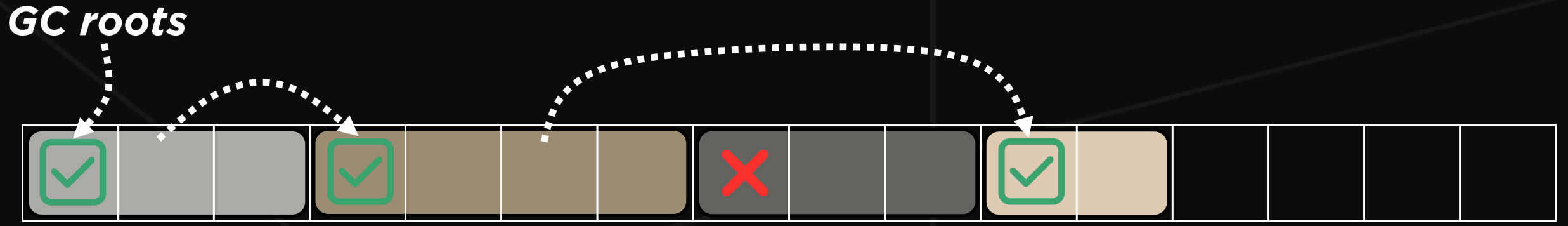
куда положить новый объект?



Mark-Sweep

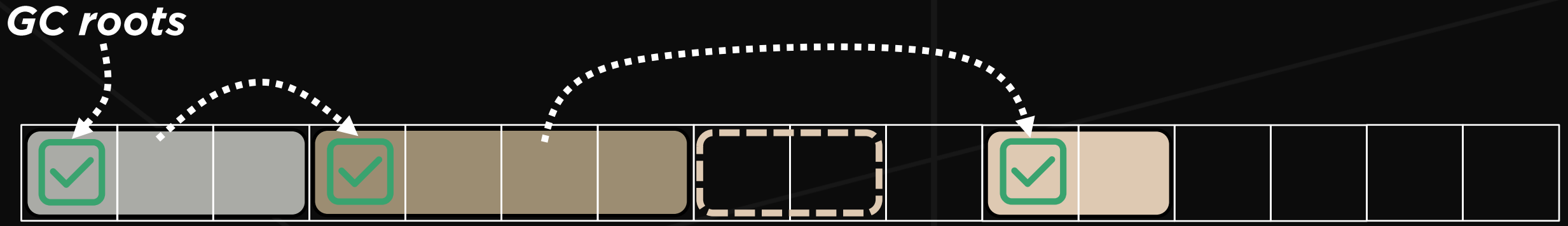
- **Sweep: помечаем память мертвых объектов как вновь доступную**
  влияет на логику работы аллокации
- **Никаких перемещений объектов**
  указатели не инвалидируются в процессе GC
- **Нужно как-то нивелировать проблему внешней фрагментации**

Mark-Compact



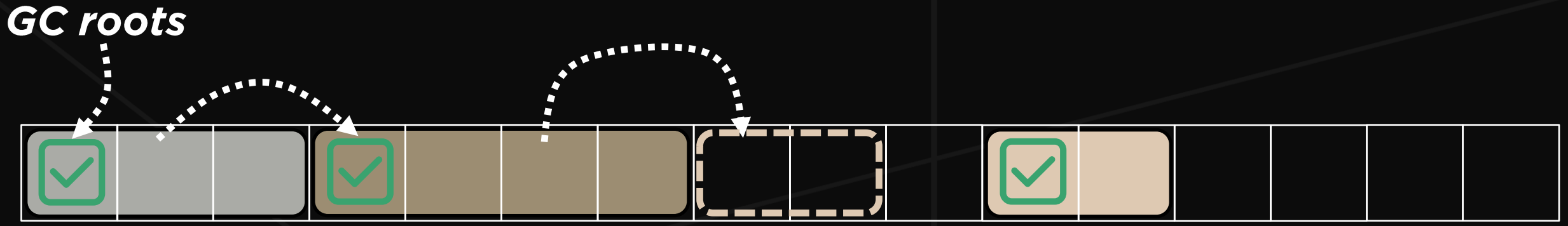
- Маркировка

Mark-Compact



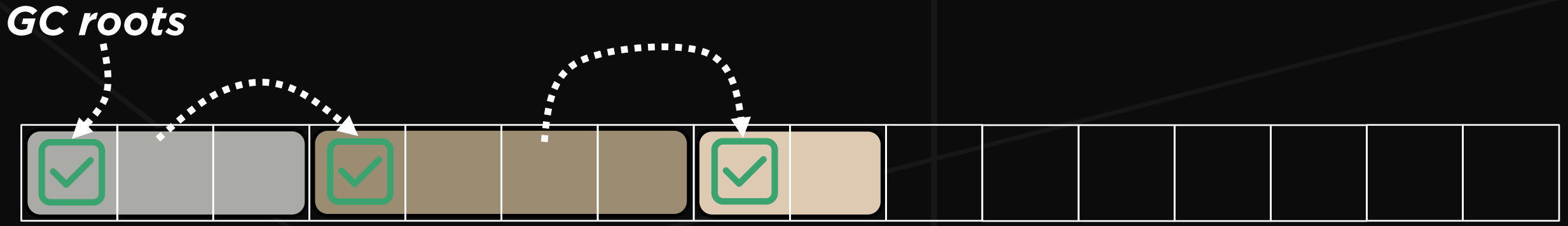
- Маркировка
- Рассчитываем новые адреса объектов (пока не переносим)

Mark-Compact



- Маркировка
- Рассчитываем новые адреса объектов (пока не переносим)
- В существующих объектах переписываем адреса указателей

Mark-Compact

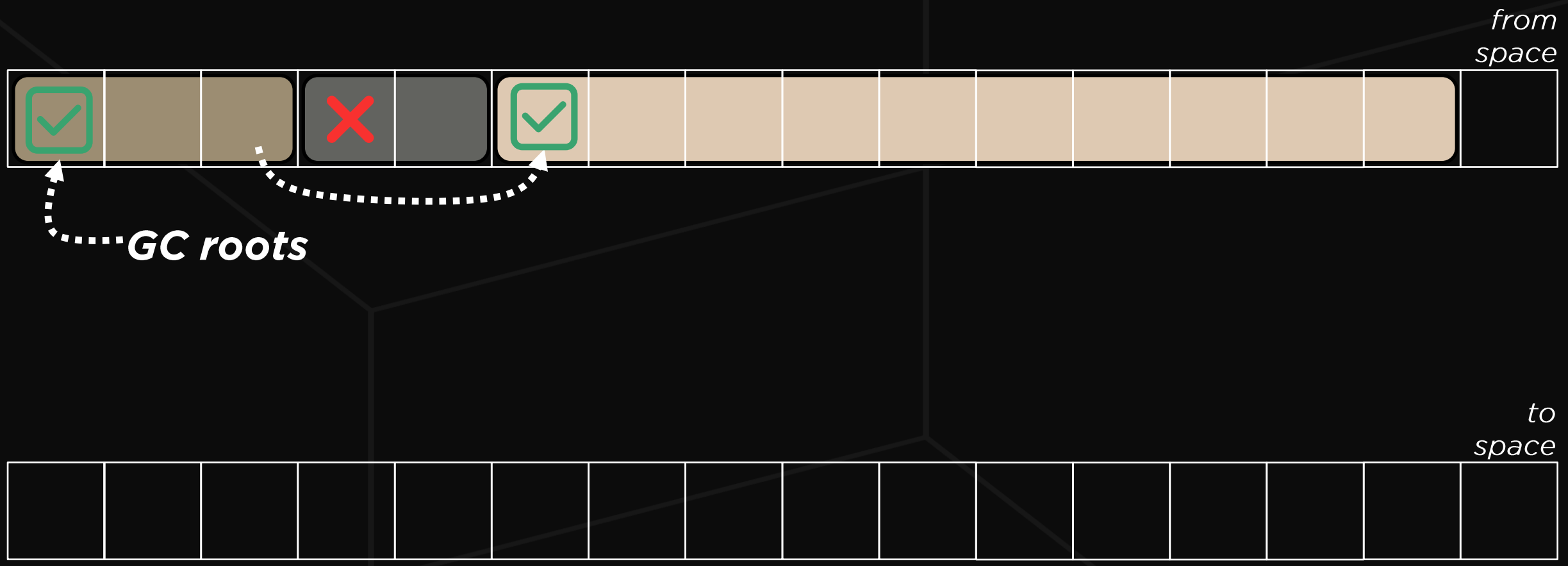


- Маркировка
- Рассчитываем новые адреса объектов (пока не переносим)
- В существующих объектах переписываем адреса указателей
- Перенос объектов

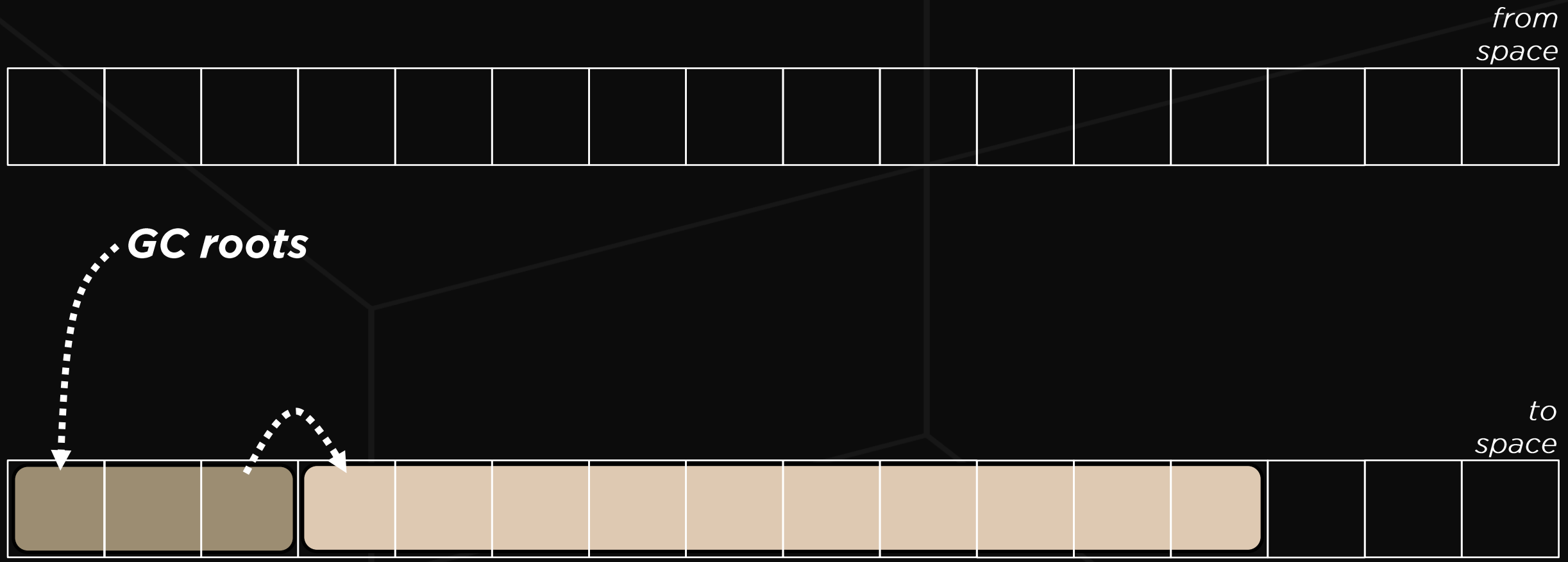
Mark-Compact

- **Compact: собираем все живые объекты рядом в хипе**
- **Много работы**
  маркировка, изменение адресов, перенос
- **Переносим объекты**
  нет проблем с фрагментацией
  улучшаем локальность данных
- **Аллокация**
  через bumping pointer

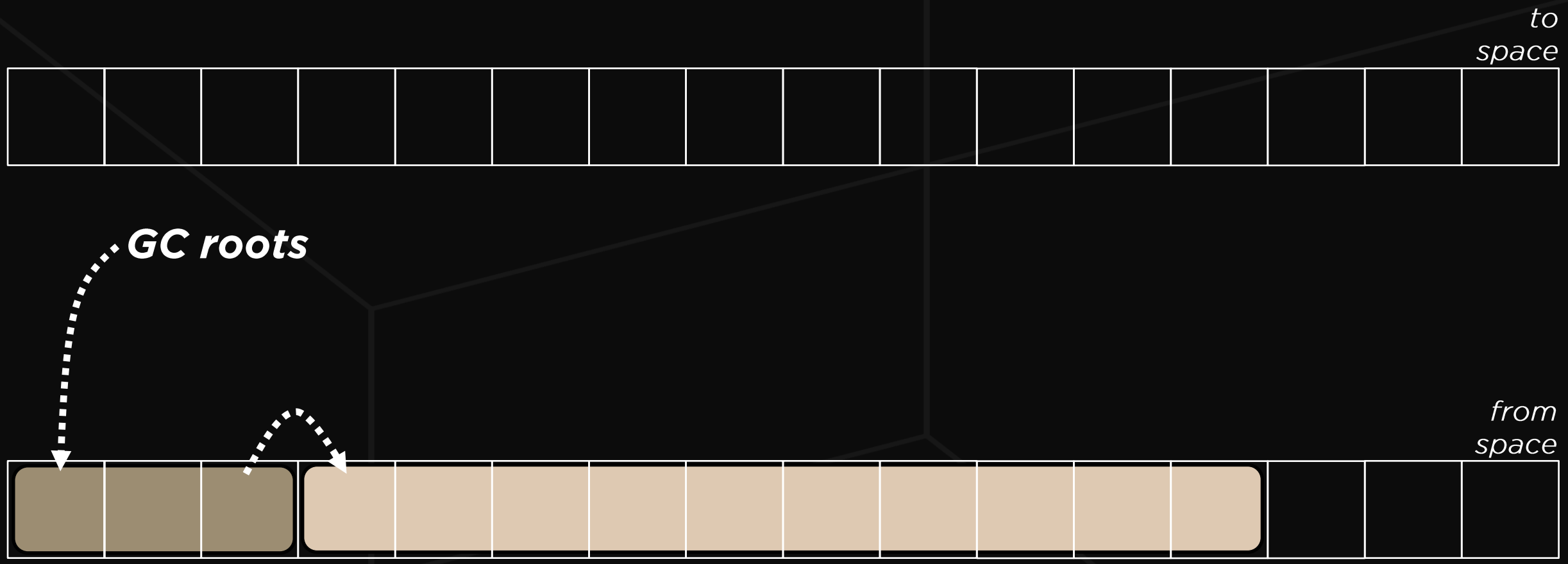
Mark-Evacuate



Mark-Evacuate



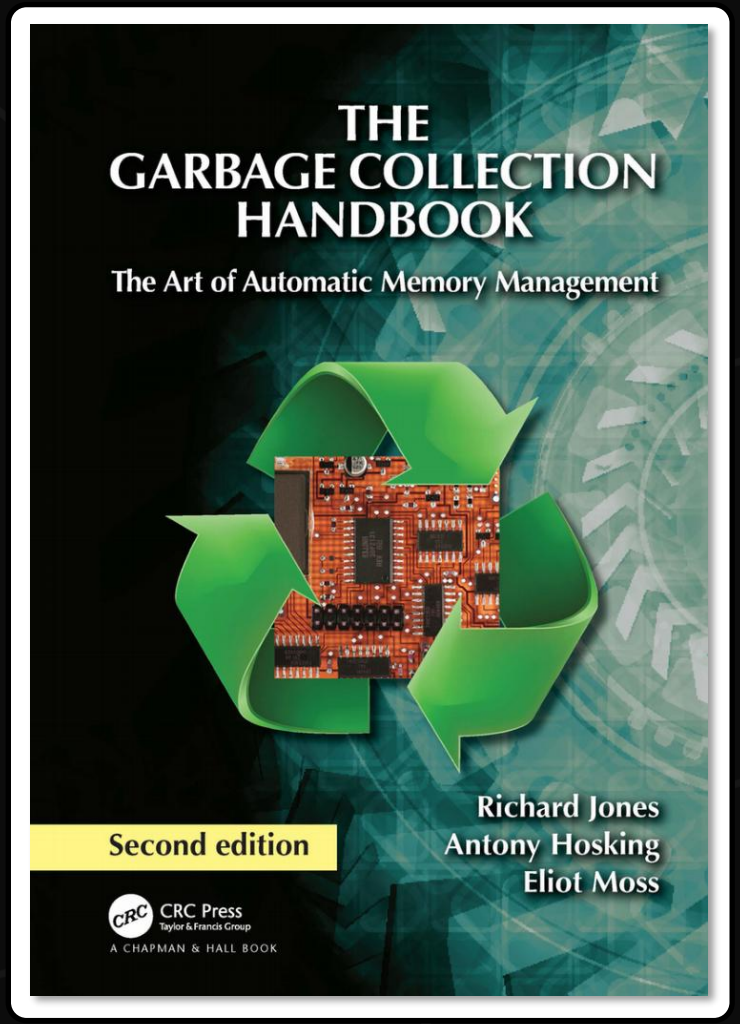
Mark-Evacuate



Mark-Evacuate

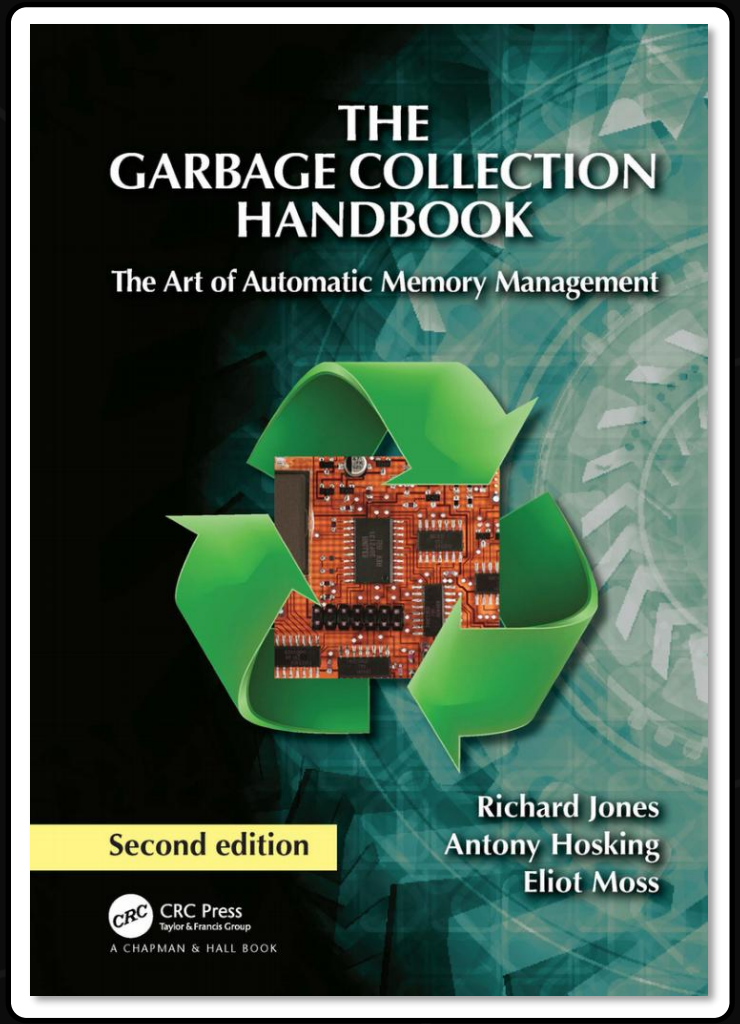
- **Две области хипа: *from-space* и *to-space***
  инвариант: to-space пуст в начале GC
- **Переносим объекты**
  нет проблем с фрагментацией
  улучшаем локальность данных
- **Аллокация**
  через bumping pointer

Четыре подхода к GC



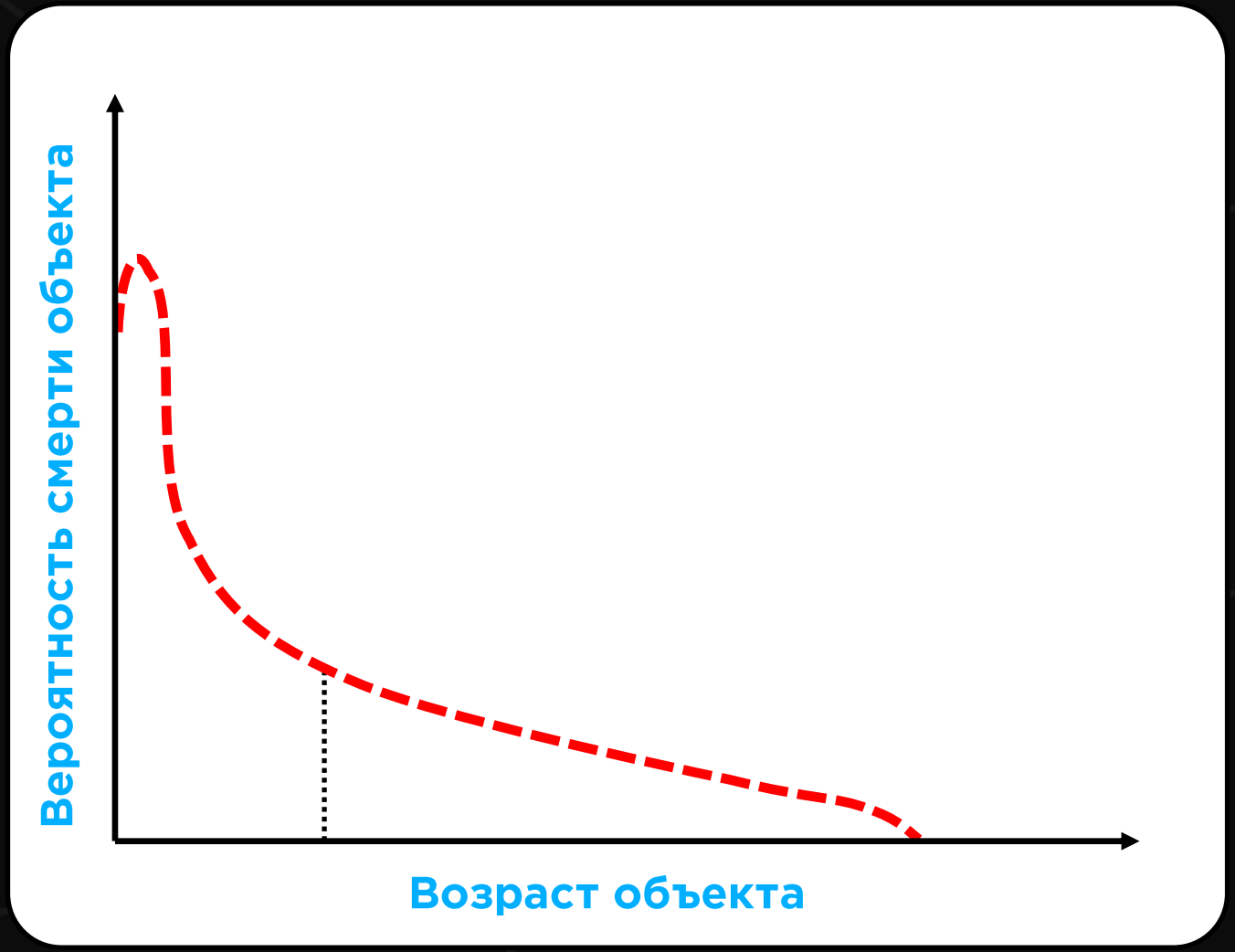
- ~~Reference counting~~
- Mark-Sweep
- Mark-Compact
- Mark-Evacuate

Четыре подхода к GC

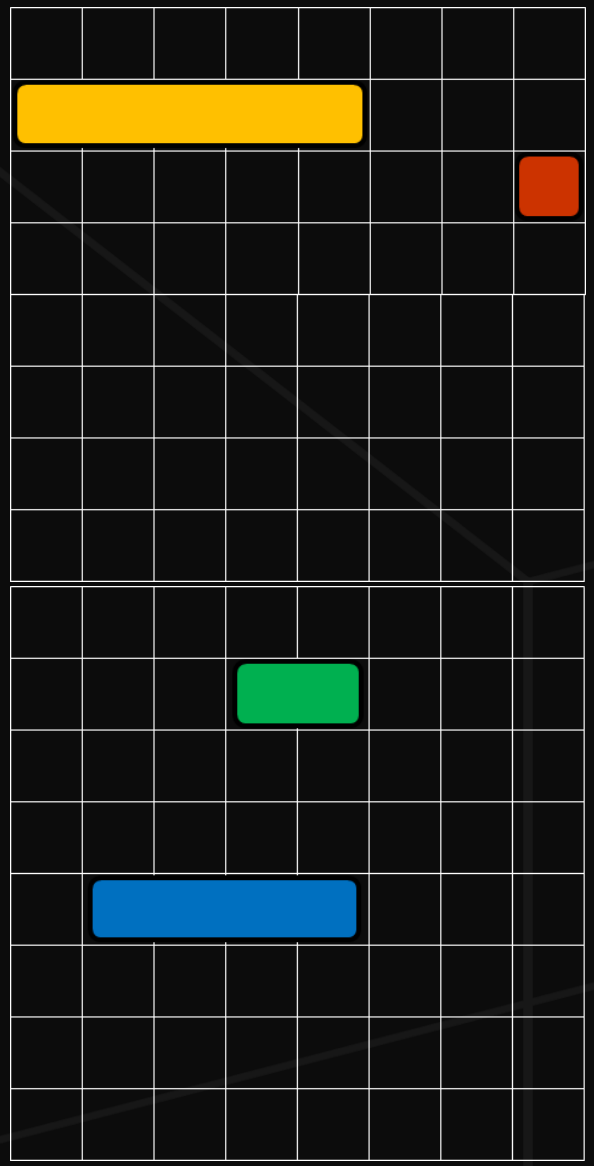


- ~~Reference counting~~
- ~~Mark-Sweep~~
- ~~Mark-Compact~~
- ~~Mark-Evacuate~~

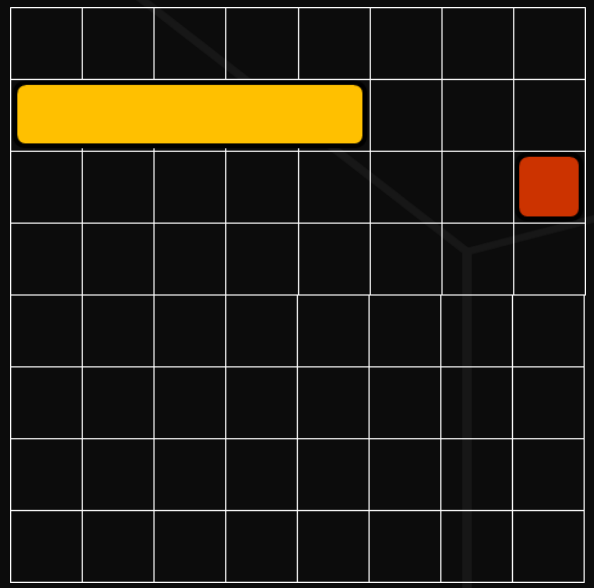
Гипотезы о поколениях



- **Weak**
  Молодые объекты по большей части быстро умирают
- **Strong**
  Чем старше объект, тем меньше вероятность его смерти

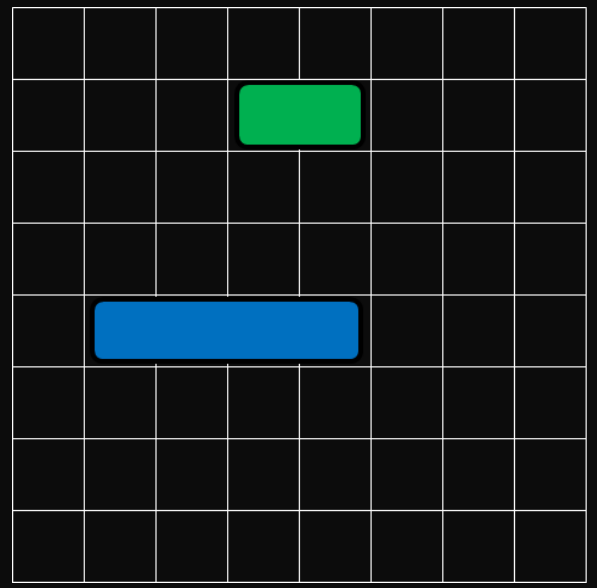


GC: *хорошая инвестиция*
потратим CPU ➜ много RAM



молодые

GC: *сомнительная инвестиция*
потратим CPU ➜ немного RAM



старые

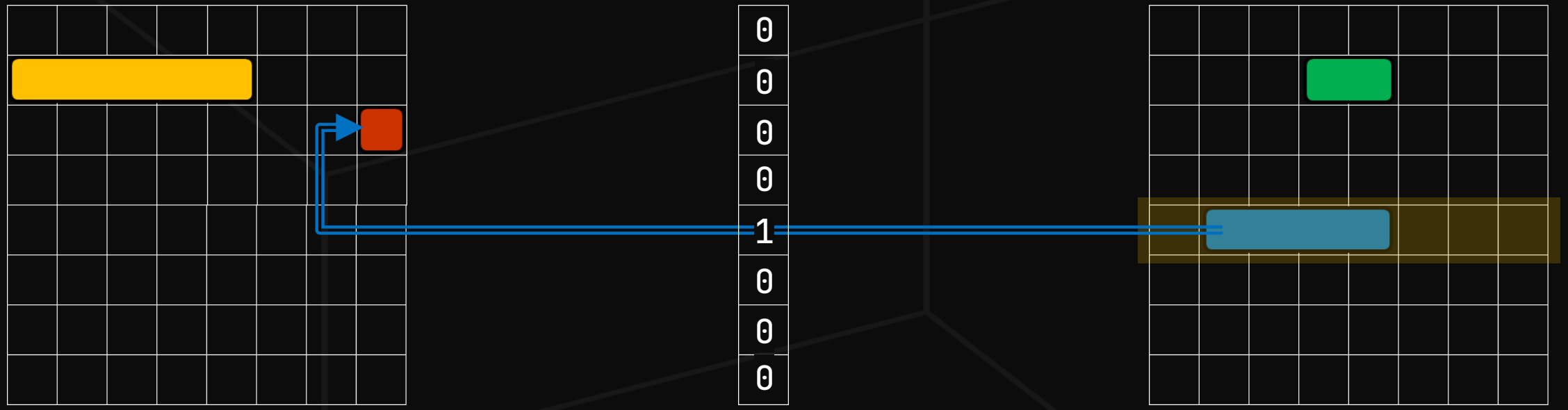
GC: *хорошая инвестиция*
потратим CPU → много RAM

GC: *сомнительная инвестиция*
потратим CPU → немного RAM

мало указателей из старого поколения в молодое

молодые

старые


0
0
0
0
1
0
0
0
молодые
card table
старые

Поколения

- **Младшая сборка – для эффективности**
- **Не бесплатно –**
  нужна специальная логика: *барьер на запись*
  даже в STW-коллекторах
- **Гипотезы о поколениях могут нарушатся**
  например, LRU-кэш

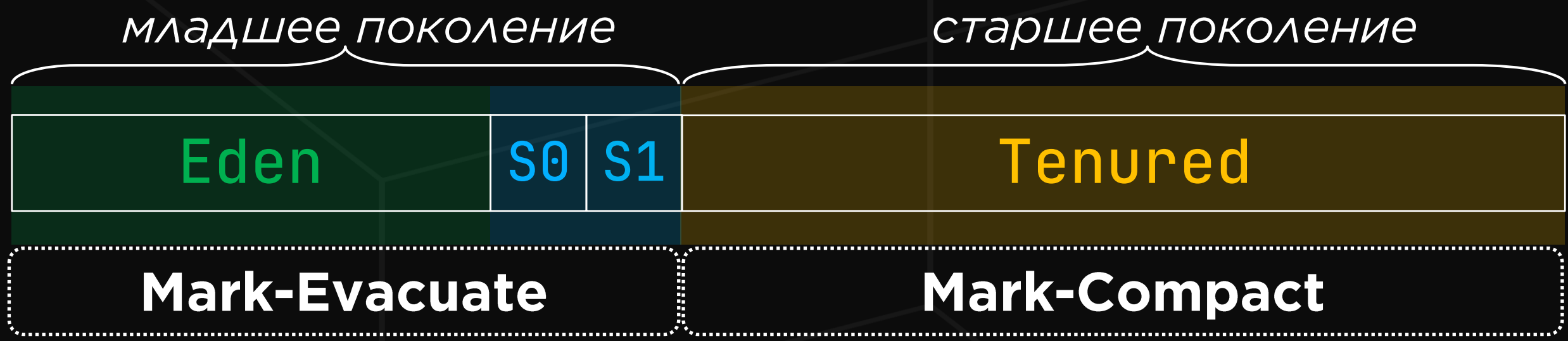
Часть 2
STW-коллекторы

- **Основные потоки приложения – *мутаторы***

- **Останавливаем всех мутаторов → выполняем GC → возобновляем работу всех мутаторов**

Java: Serial и Parallel
Go: версия 1.4

Serial и Parallel



Serial и Parallel

- **Usecase: максимизация throughput**

  даже ценой большей latency

- **Постоянно обновляются и улучшаются**

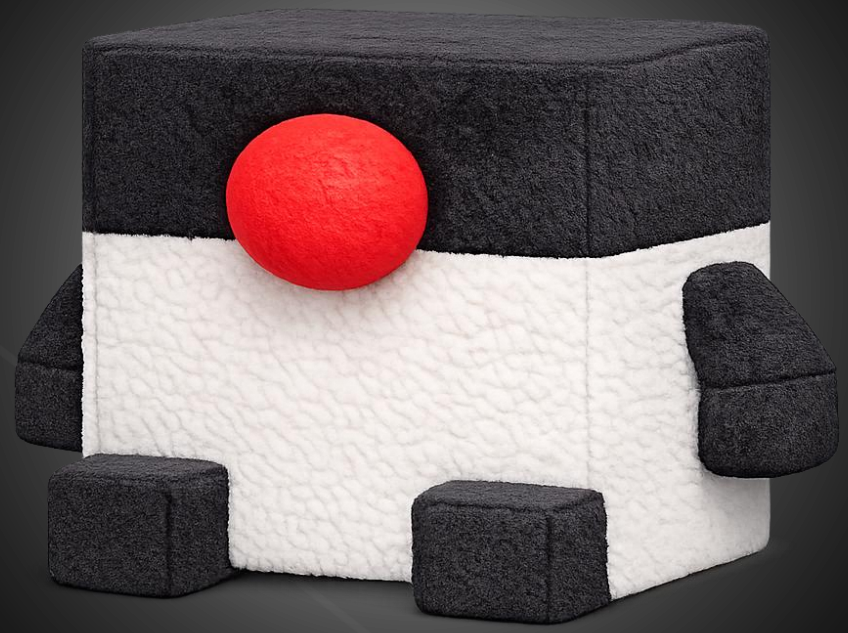
JDK / JDK-8320165

Parallel: Full GC code is very slow due to quadratic calc_new_address

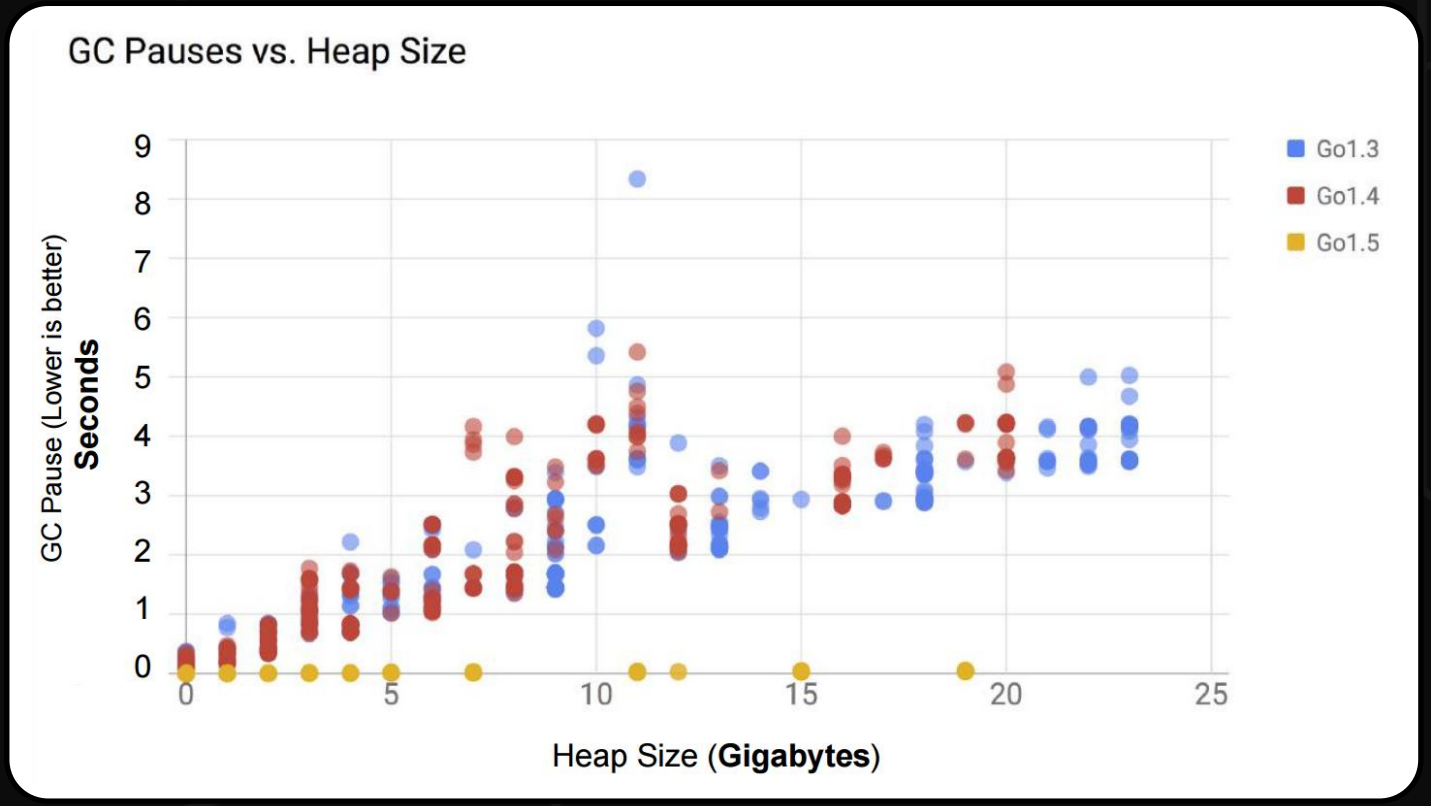
Closed

Details

Type:	Enhancement	Resolution:	Duplicate
Priority:	P4	Fix Version/s:	tbd
Affects Version/s:	11, 17, 21, 22		
Component/s:	hotspot		
Labels:	amazon-interest, gc-parallel		
Subcomponent:	gc		



Древний Go: STW Mark-Sweep



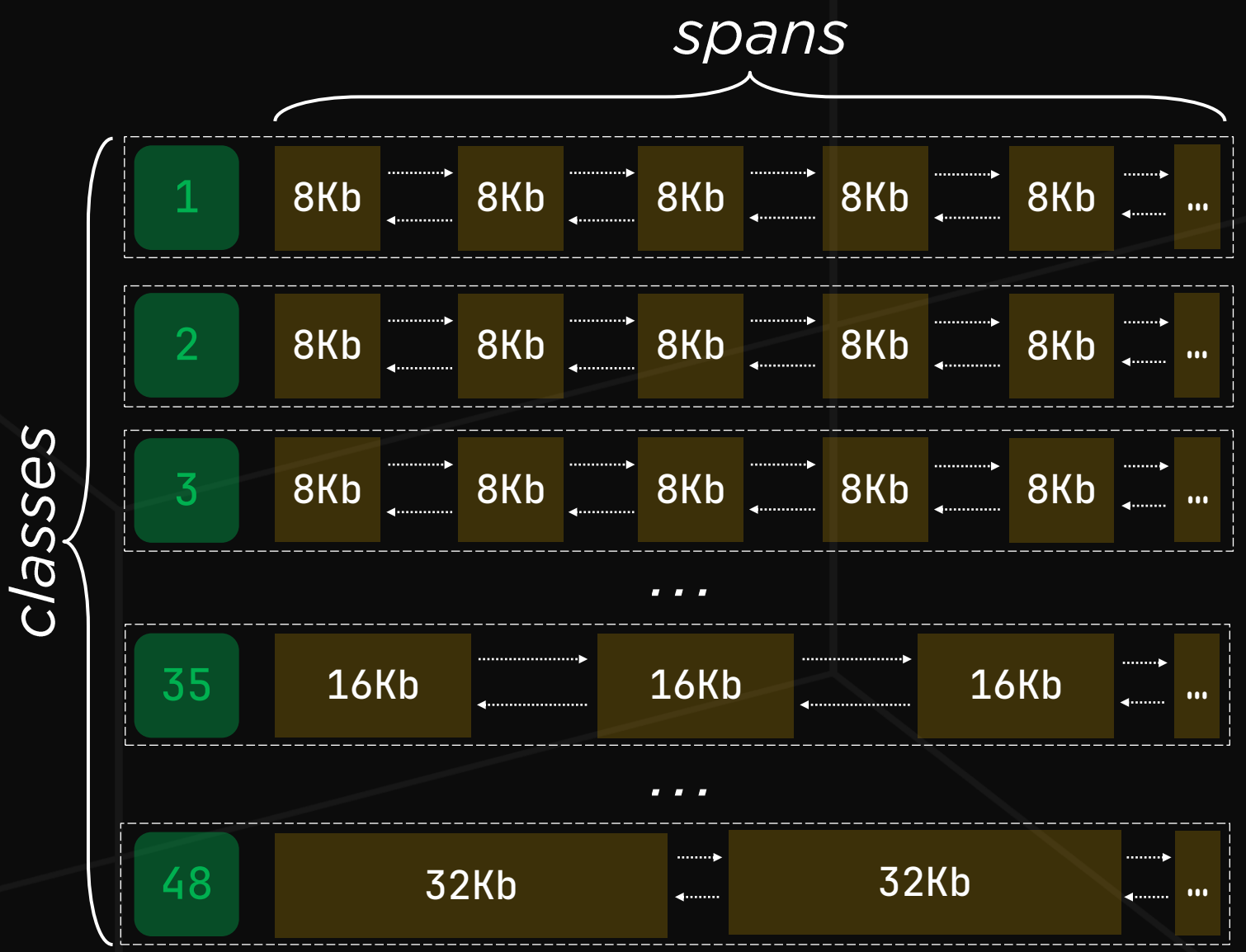
- Классический алгоритм
- Нужно что-то сделать с фрагментаций

Внешняя фрагментация

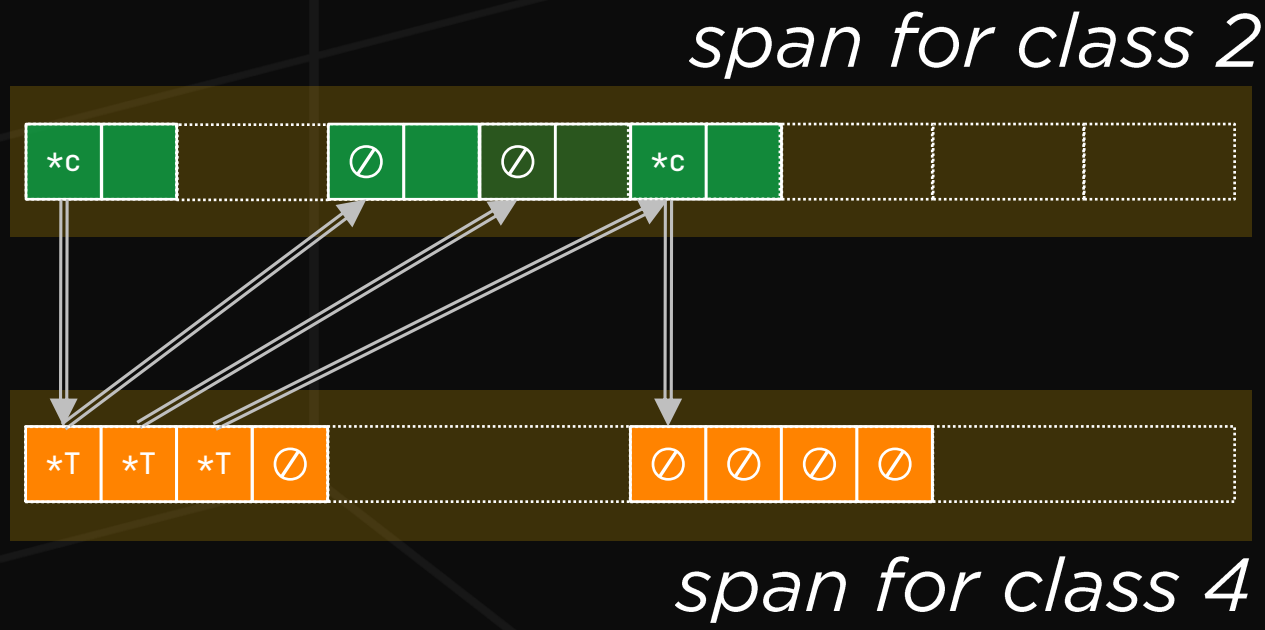
куда положить новый объект?

Source file src/internal/runtime/gc/sizeclasses.go

```
1  // Code generated by mksizeclasses.go; DO NOT EDIT.
2  //go:generate go -C ../../../runtime/_mkmalloc run mksizeclasses.go
3
4  package gc
5
6  // class  bytes/obj  bytes/span  objects  tail waste  max waste  min align
7  //     1          8        8192     1024           0     87.50%          8
8  //     2         16        8192      512           0     43.75%         16
9  //     3         24        8192      341           8     29.24%          8
10 //     4         32        8192      256           0     21.88%         32
11 //     5         48        8192      170          32     31.52%         16
12 //     6         64        8192      128           0     23.44%         64
13 //     7         80        8192      102          32     19.07%         16
14 //     8         96        8192       85          32     15.95%         32
15 //     9        112        8192       73          16     13.56%         16
16 //    10        128        8192       64           0     11.72%        128
17 //    11        144        8192       56         128     11.82%         16
18 //    12        160        8192       51          32      9.73%         32
19 //    13        176        8192       46          96      9.59%         16
20 //    14        192        8192       42         128      9.25%         64
21 //    15        208        8192       39          80      8.12%         16
22 //    16        224        8192       36         128      8.15%         32
```


spans
classes
1
8Kb
8Kb
8Kb
8Kb
8Kb
...
2
8Kb
8Kb
8Kb
8Kb
8Kb
...
3
8Kb
8Kb
8Kb
8Kb
8Kb
...
...
35
16Kb
16Kb
16Kb
...
...
48
32Kb
32Kb
...

```
type T struct{
    children *[4]*T
    value    int
}
```

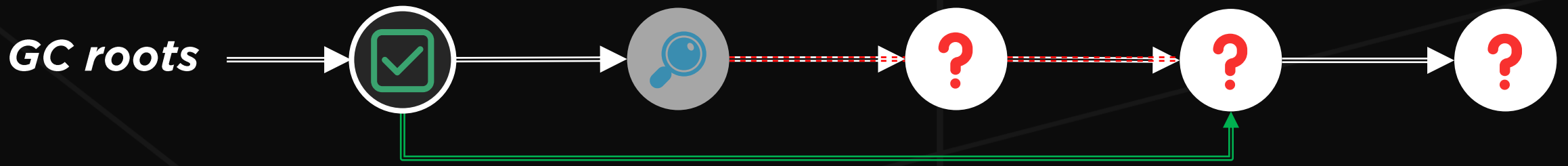


Борьба с внешней фрагментацией

- **Классифицируем объекты**
  выбираем класс в зависимости от размера объекта
- **Свои спаны у каждого класса**
  особый случай для очень маленьких и очень больших
- **Цена - внутренняя фрагментация**
  потери ~10%
- **Не пишем на Go, как на Java**
  пользуемся встраиванием структур, а не указателями

Часть 3
Конкурентная маркировка

Конкурентная маркировка



- ? - пока не встречали
- - объект живой, но его исходящие указатели не обработаны полностью
- - объект живой и обработан

инвариант: запрещены → ?

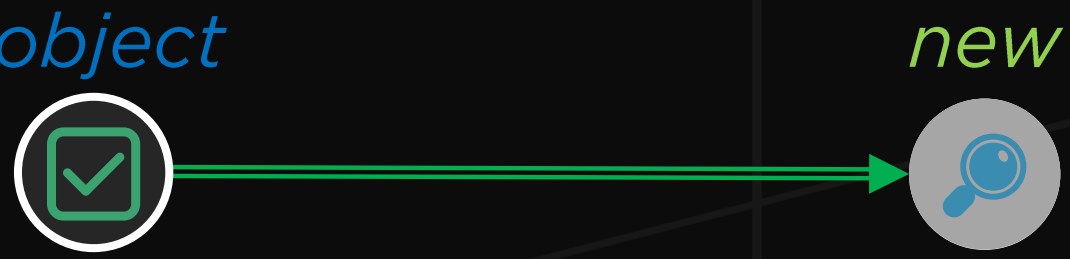
- **Нужно *подловить* мутатора во время маркировки**

Конкурентная маркировка

- **Решение: пока работает маркировка – нужно включать *барьер на запись***

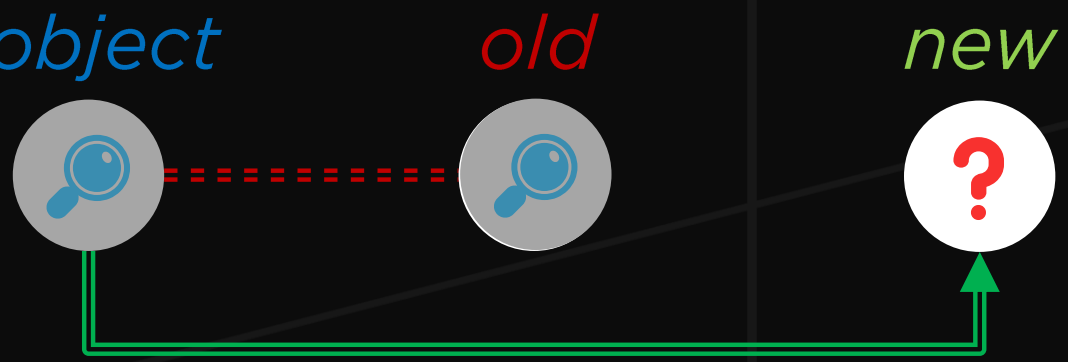
```
object.field = <newRef>
```

IU / Dijkstra write barrier



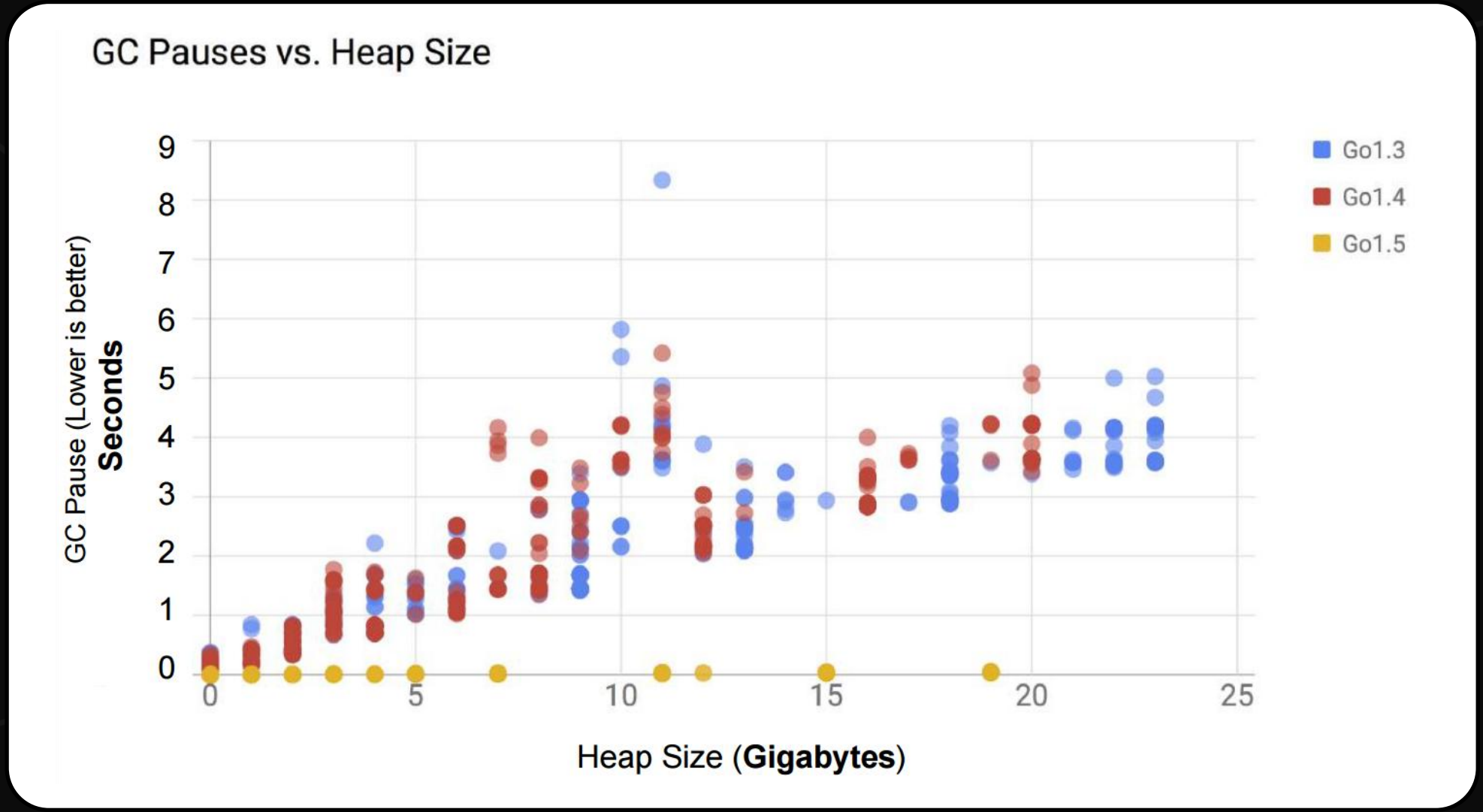

```
write_barrier(object.field, newRef):
    object.field = newRef
    if newRef != null and isNotMarked(newRef):
        mark(newRef)
```

SATB / Yuasa write barrier




```
write_barrier(object.field, newRef):
    var oldRef = object.field
    if oldRef != null:
        mark(oldRef)
    object.field = newRef
```

Go 1.5: Concurrent Mark-Sweep



Go 1.5: Concurrent Mark-Sweep

…While other goals may intervene, 1.6 will most likely be used to improve throughput by adding bump pointer allocation as well as a generational copy collector for nursery (new) spaces…

…The work being pulled in includes logic for starting and stopping Go routines individually and adding a write barrier infrastructure suitable not only for concurrent GC but also for the potential 1.6 generational copying collector…

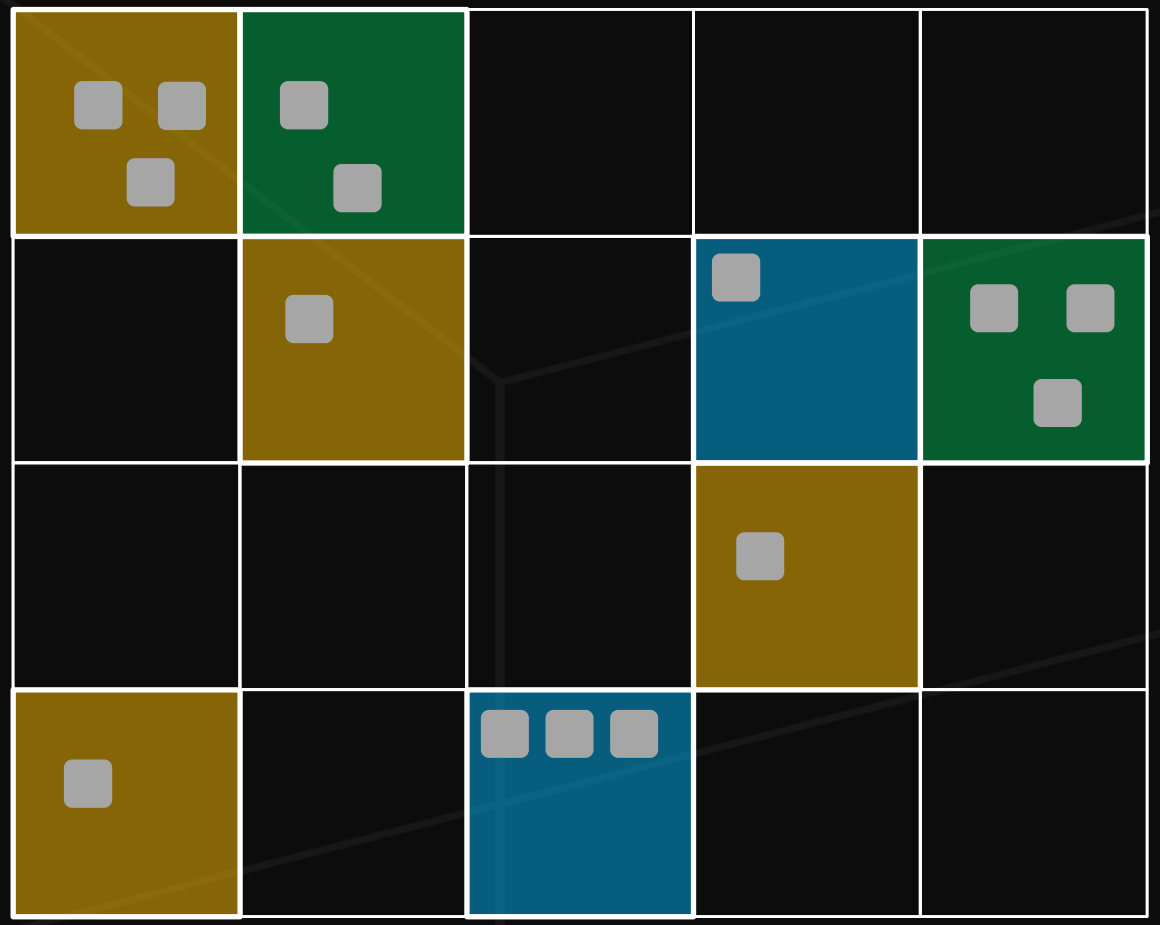
golang.org/s/go14gc

Почему в Go нет поколений?

github.com/golang/go/discussions/70257#discussioncomment-11201305

- Барьер на запись должен быть включен всегда, а не во время маркировки
- Слабая гипотеза о поколениях не соблюдается: смертность ~60%, а не 90+%
- Почему: compile-time escape analysis + structs

G1



Eden

Survivors

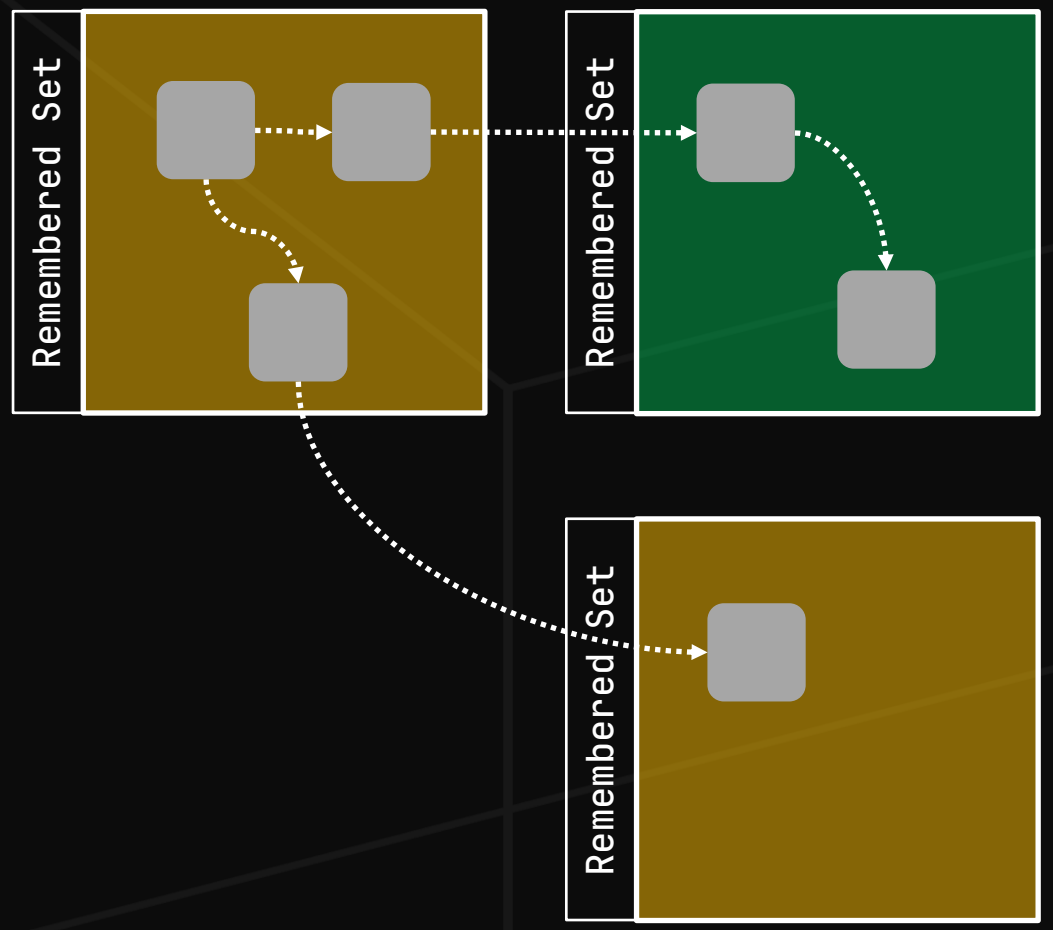
младшее

Old

старшее

Как отслеживать ссылки между регионами?

Remembered Set



- Храним информацию об входящих указателях из регионов старого поколения
- Механизм нетривиальный и неоднократно переделывался
- Нужны барьеры на запись!

Write Barrier: Parallel

```
; %r10 = store location
; %ebp = reference (compressed)
; %r12b = 0

shrq $0x9, %r10
movabsq $0x7f4975093000, %r14
movb %r12b, (%r14,%r10)
```

Write Barrier: G1

```
; %r11 = store location
; %ebx = reference (compressed)
; %r12b = 0
;
; Fast-path
; Cross-region check:
xorq %r11, %r10
shrq $0x15, %r10
testq %r10, %r10
je 0x22
; Null pointer check:
testl %ebx, %ebx
je 0x1e
; Card address calculation:
shrq $0x9, %r11
movabsq $0x7f48d0c00000, %rdi
addq %r11, %rdi
nopl (%rax)
; Young gen. card check:
cmpb $0x2, (%rdi)
jne 0x166 ; => slow-path

; Slow-path
; Thread-local dirty card buffer:
movq 0x48(%r15), %r10
movq 0x58(%r15), %r11
```

```
; Memory barrier:
lock
addl $0x0, -0x40(%rsp)
nop
; Dirty card check:
cmpb $0x0, (%rdi)
je -0x180
; Dirty the card:
movb %r12b, (%rdi)
; Check if the buffer is full
testq %r10, %r10
jne 0x21
; If full => call runtime
movq %r15, %rsi
nopl (%rax,%rax)
nop
movabsq $0x7f48f236f860, %r10
callq *%r10
jmp -0x1a9
; Else => enqueue the card
movq %rdi, -0x8(%r11,%r10)
addq $-0x8, %r10
movq %r10, 0x48(%r15)
nop
jmp -0x1bc
```

Write Barrier: G1

```
1  ; %r11 = store location
2  ; %ebx = reference (compressed)
3  ; %r12b = 0
4  ;
5  ; H
6  ; C
7  xor
8  shr
9  tes
10 je
11 ; M
12 tes
13 je
14 ; C
15 shr
16 mov
17 add
18 nop
19 ; M
20 cmp
21 jne
22
23 ; S
24 ; In
25 movq 0x48(%r15), %r10
26 movq 0x58(%r15), %r11
```

```
27 ; Memory barrier:
28 lock
29 addl $0x0, -0x40(%rsp)
```

```
51 jmp -0x1bc
```

JDK / JDK-8340827

JEP 522: G1 GC: Improve Throughput by Reducing Synchronization

Closed

Details

Type:	JEP	Resolution:	Delivered
Priority:	P4	Fix Version/s:	26
Component/s:	hotspot		
Labels:	gc-g1 gc-g1-barrier gc-g1-refinement jdk26-ptt-2025-09-15		
Author:	Ivan Walulya & Thomas Schatzl		
JEP Type:	Feature		

G1

- **Ухудшаем *throughput* ради *latency***

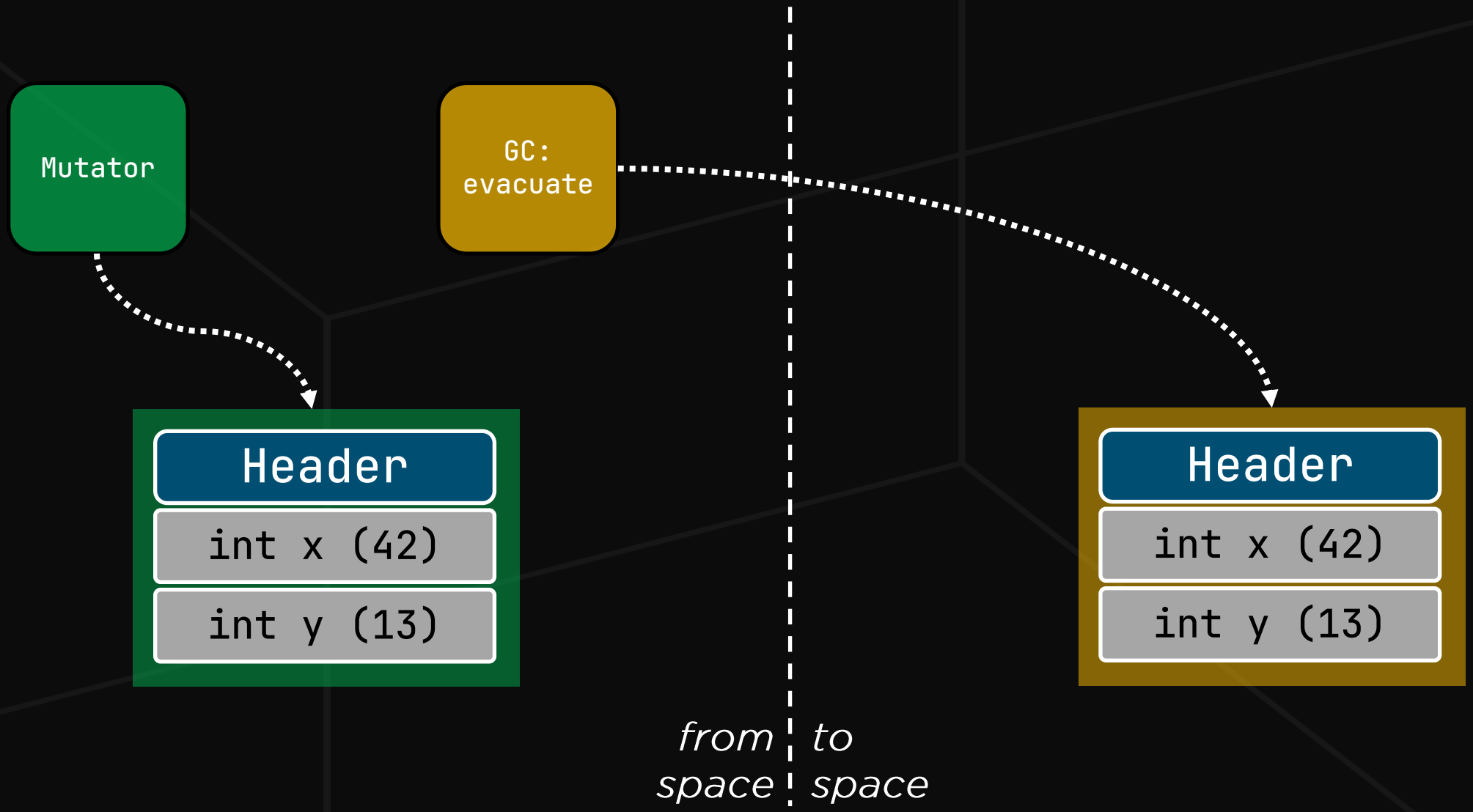
  целевая пауза ~ 200 ms

- **Практически полностью переписан**

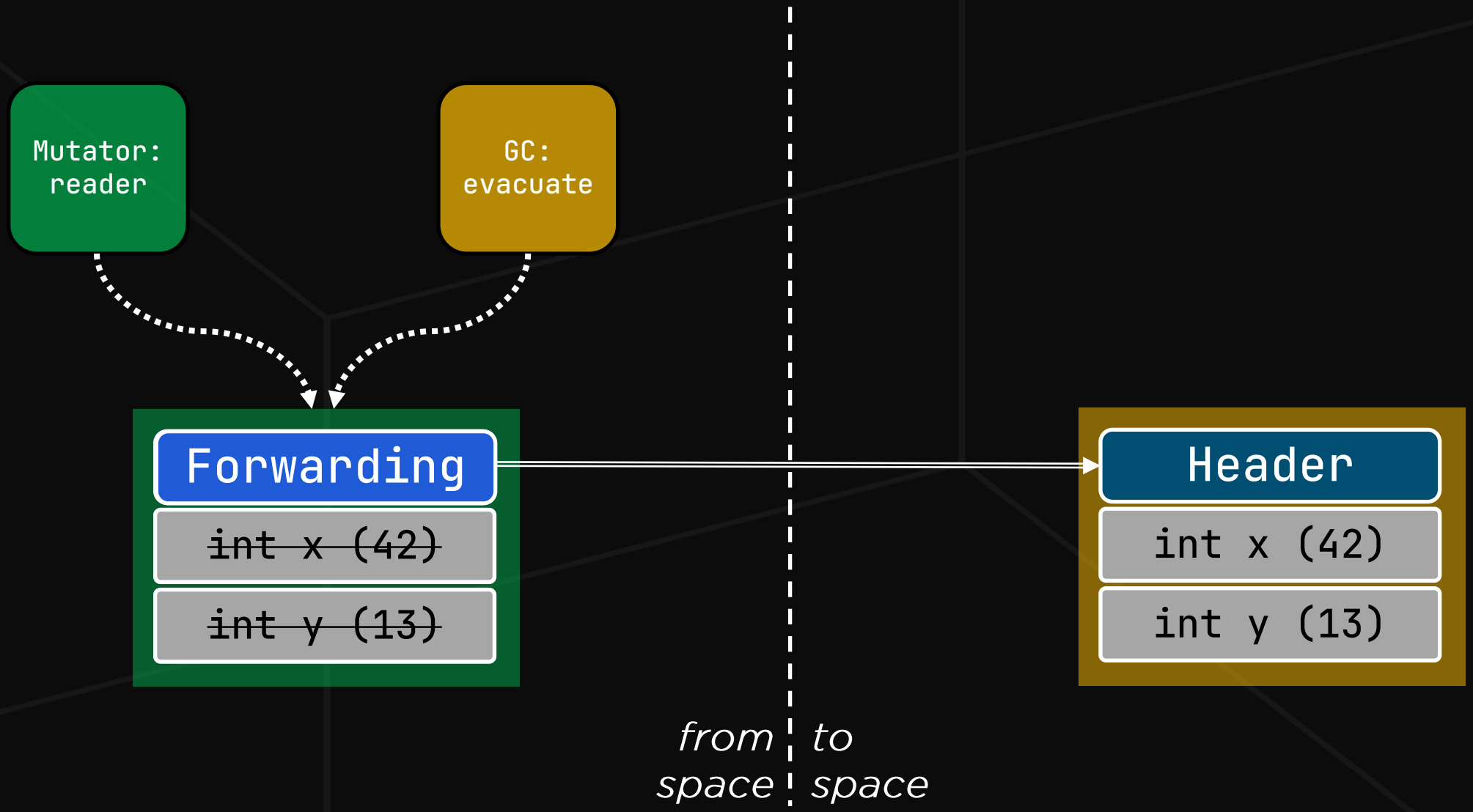
- **Если не подходил раньше -
  дайте ему второй шанс!**

Часть 4
Современные конкурентные коллекторы

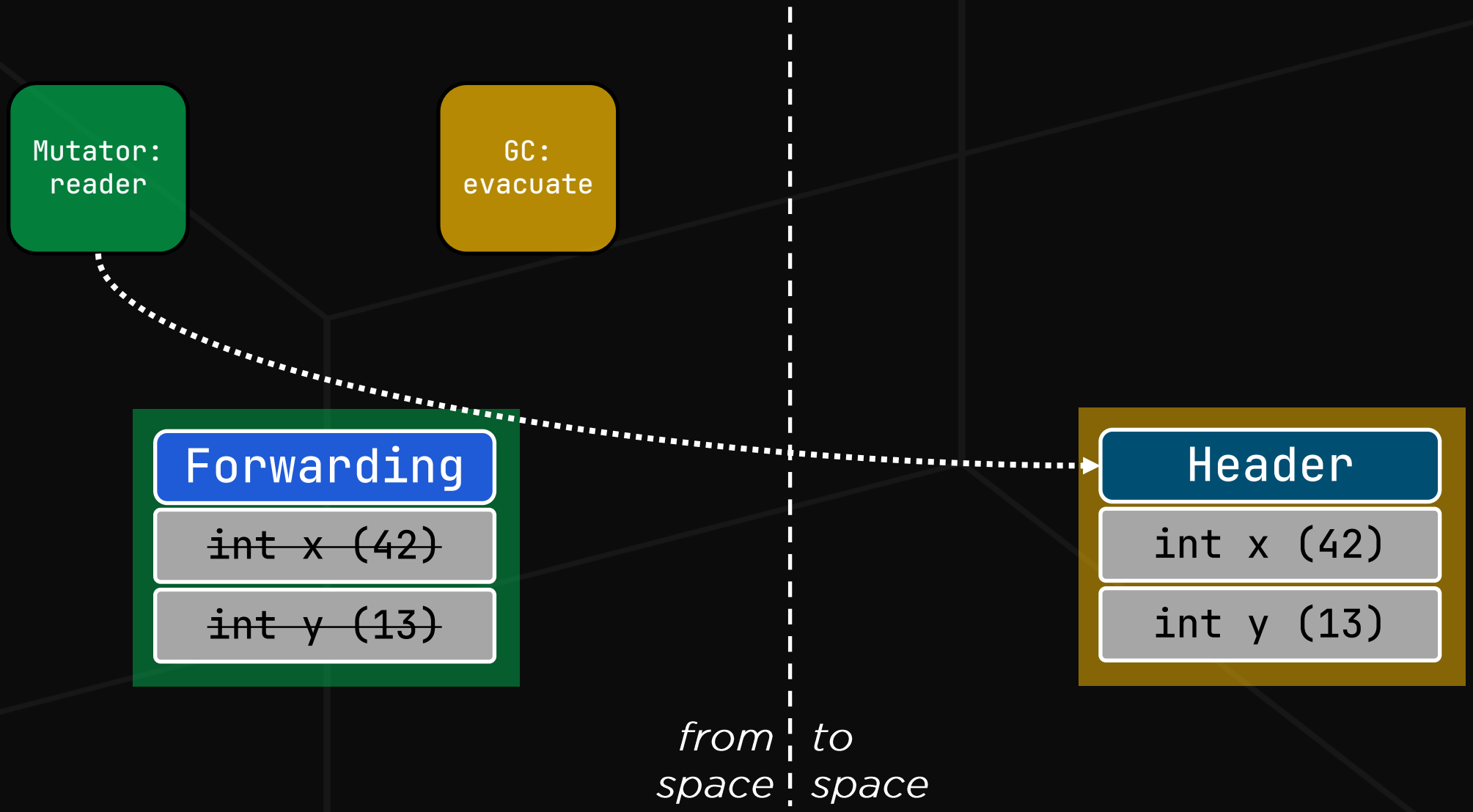
Concurrent Mark-Evacuate



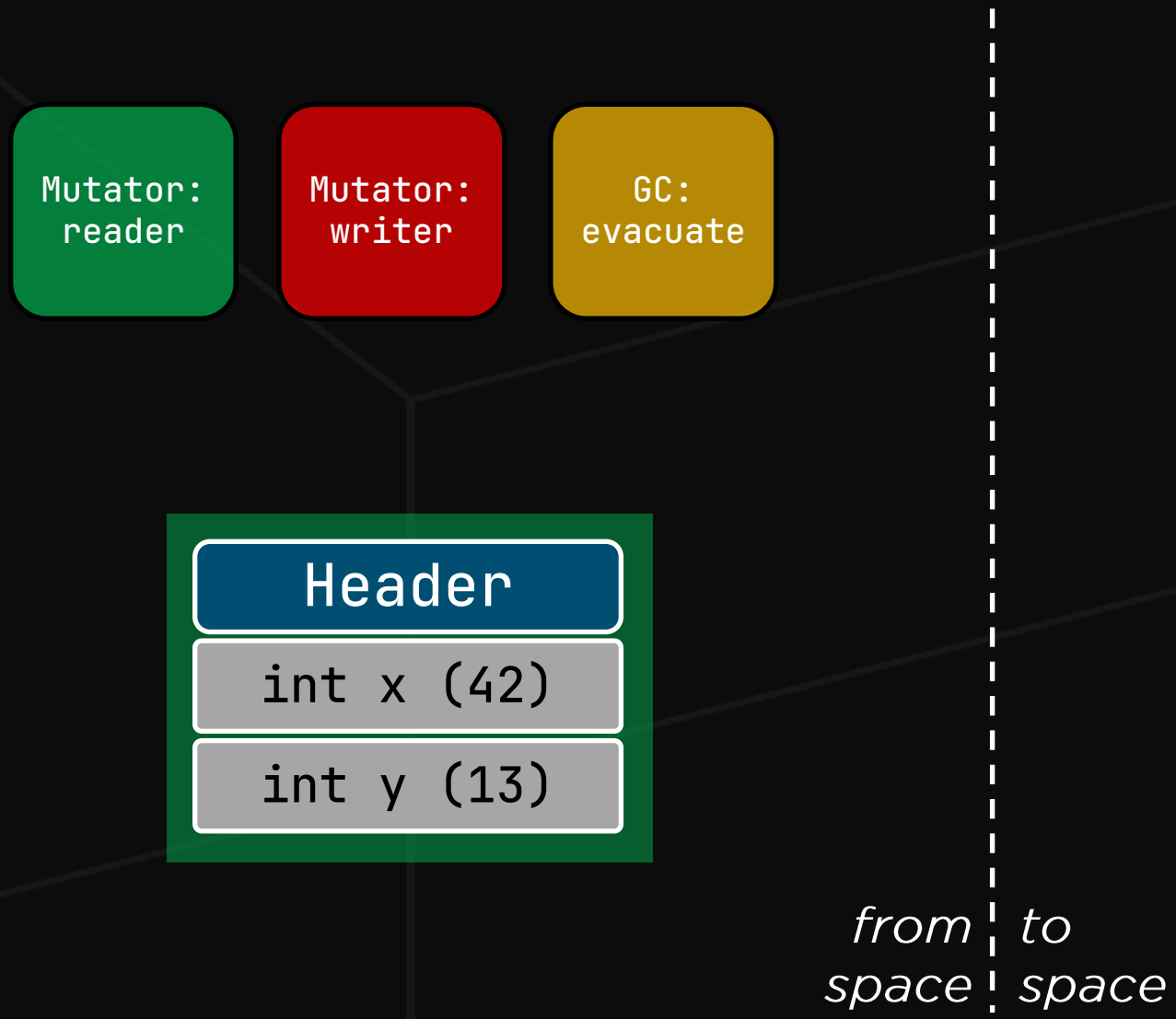
Shenandoah 2.0



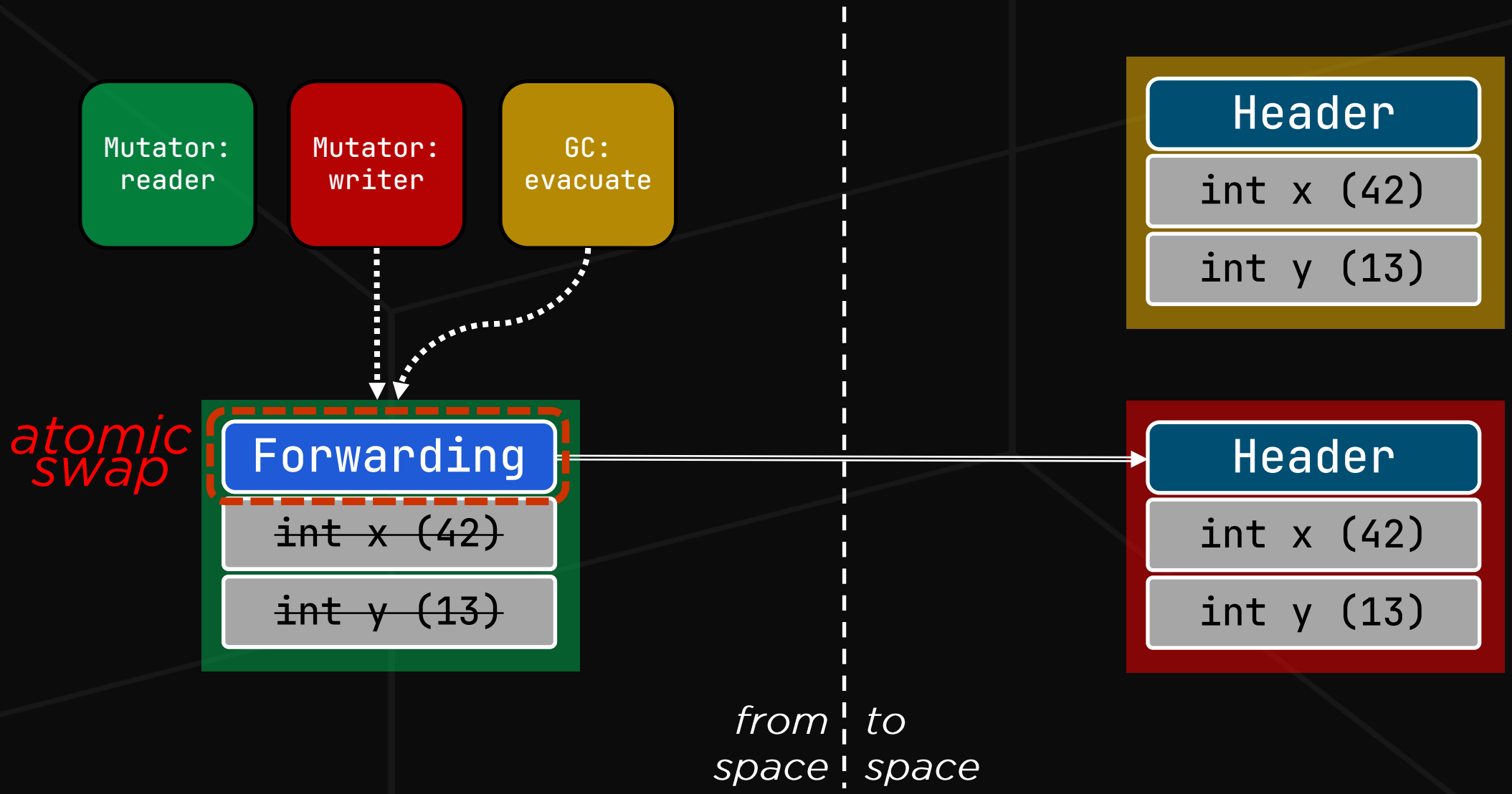
Shenandoah 2.0



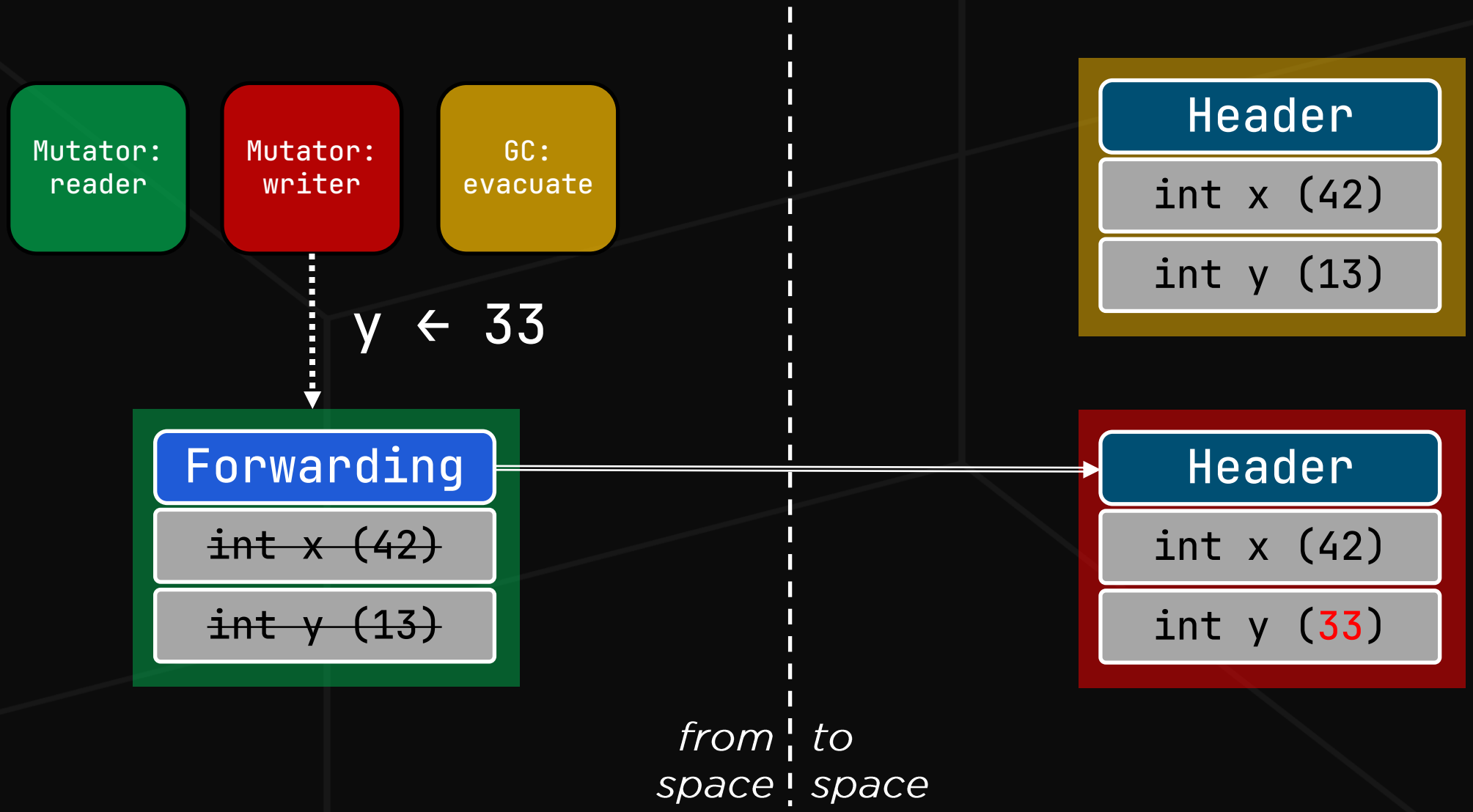
Shenandoah 2.0



Shenandoah 2.0



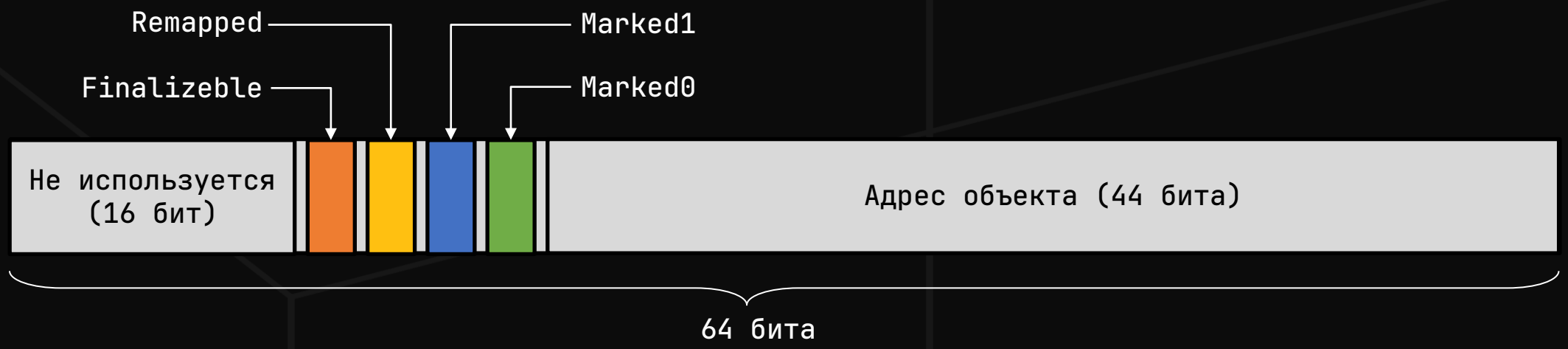
Shenandoah 2.0



Shenandoah 2.0

- **Регионализированный, конкурентный, копирующий GC**
  фокус на latency
- **Опциональная поддержка поколений**
  -XX:ShenandoahGCMode=generational
- **Информация об эвакуации**
  через mark word
- **Нужны барьеры**
  и на чтение и на запись

Non-Generational ZGC

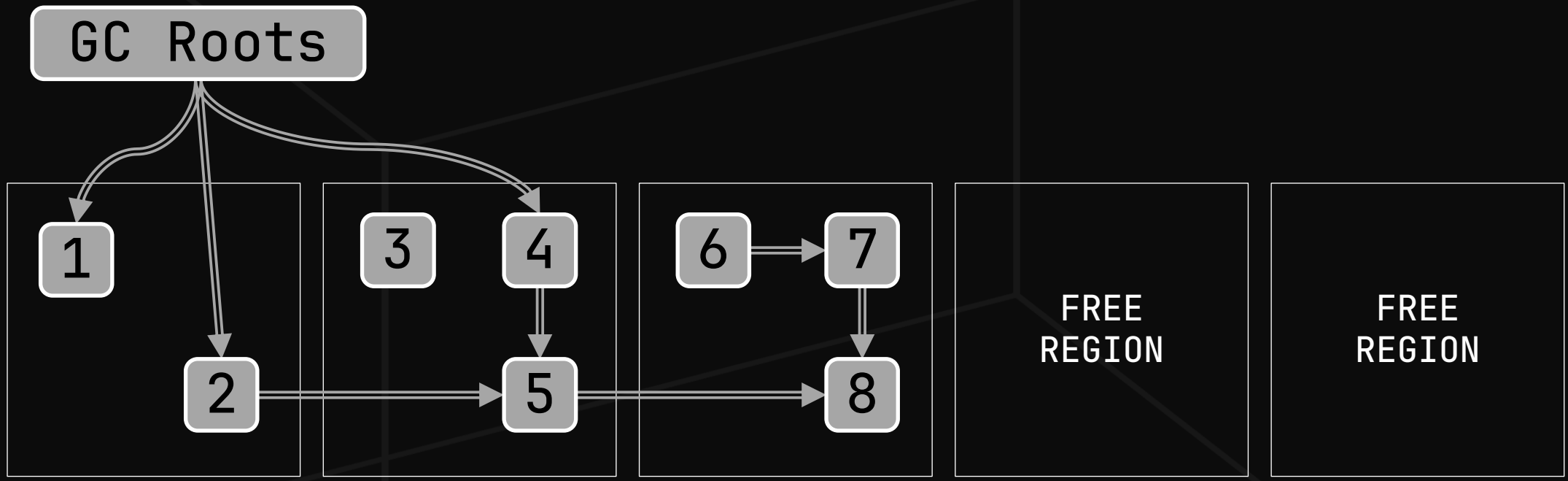


- **В 64-х битном указателе хранится «цвет»**
  никаких CompressedOops
- **Барьеры только на чтение!**
- **Трюк с тройным virtual mapping**

Цвета: Finalizeble | Marked1 | Remapped | Marked0

Стадия GC: *Pause Mark Start*

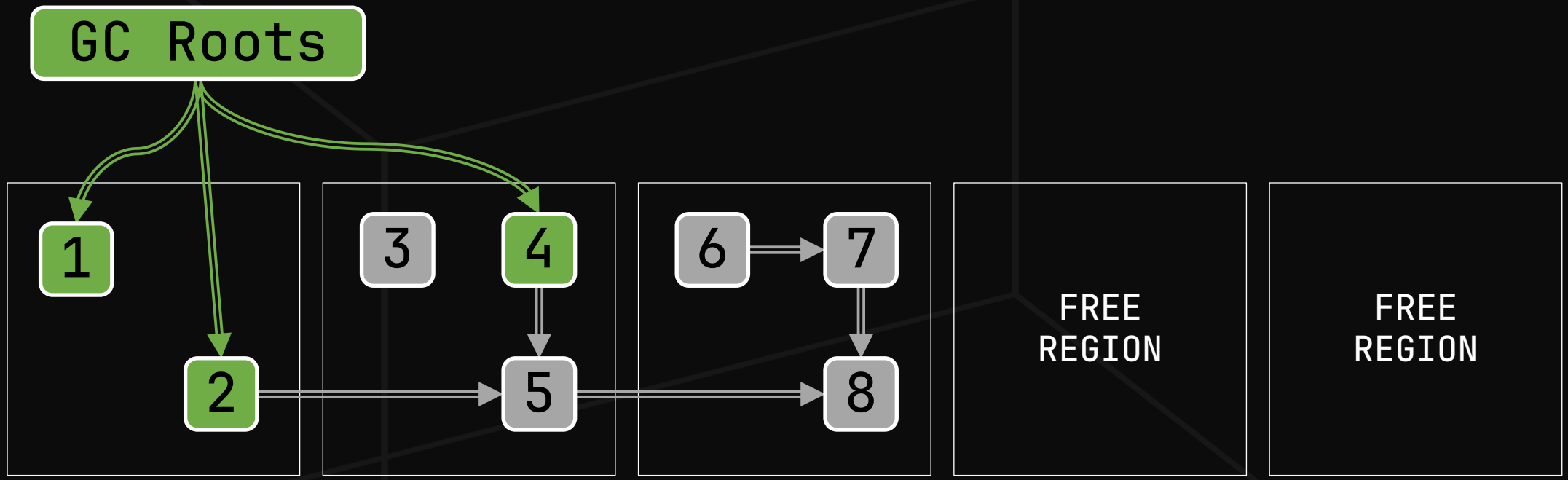
Правильный цвет: Marked0



Цвета: Finalizeble | Marked1 | Remapped | Marked0

Стадия GC: *Pause Mark Start*

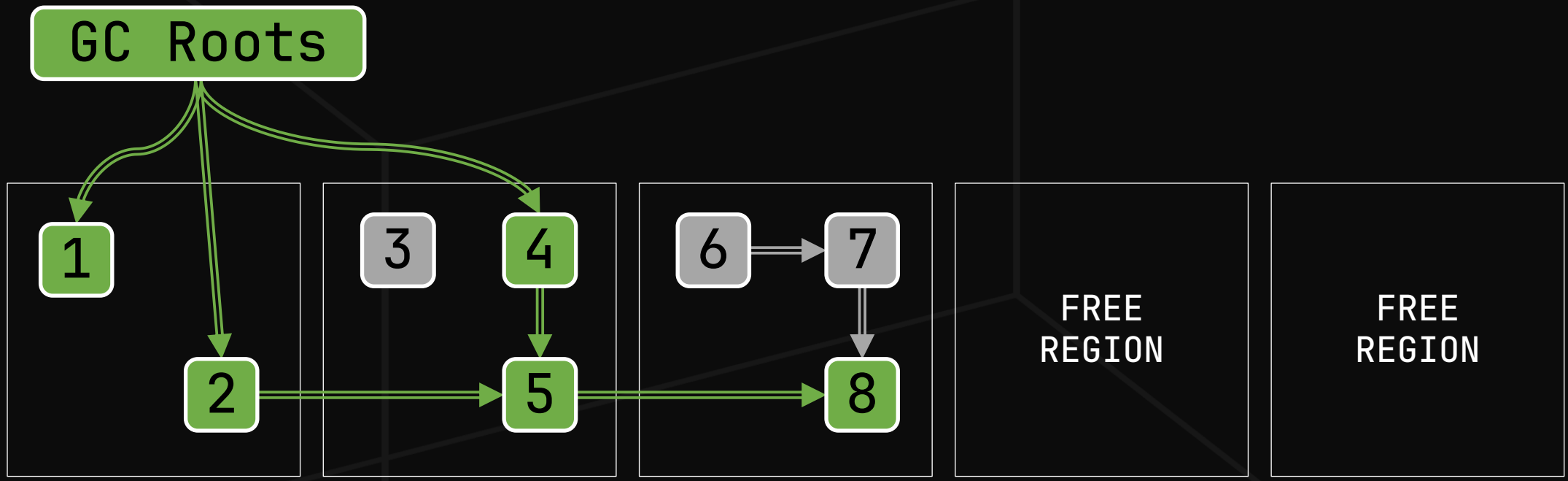
Правильный цвет: Marked0



Цвета: Finalizeble | Marked1 | Remapped | Marked0

Стадия GC: *Concurrent Mark*

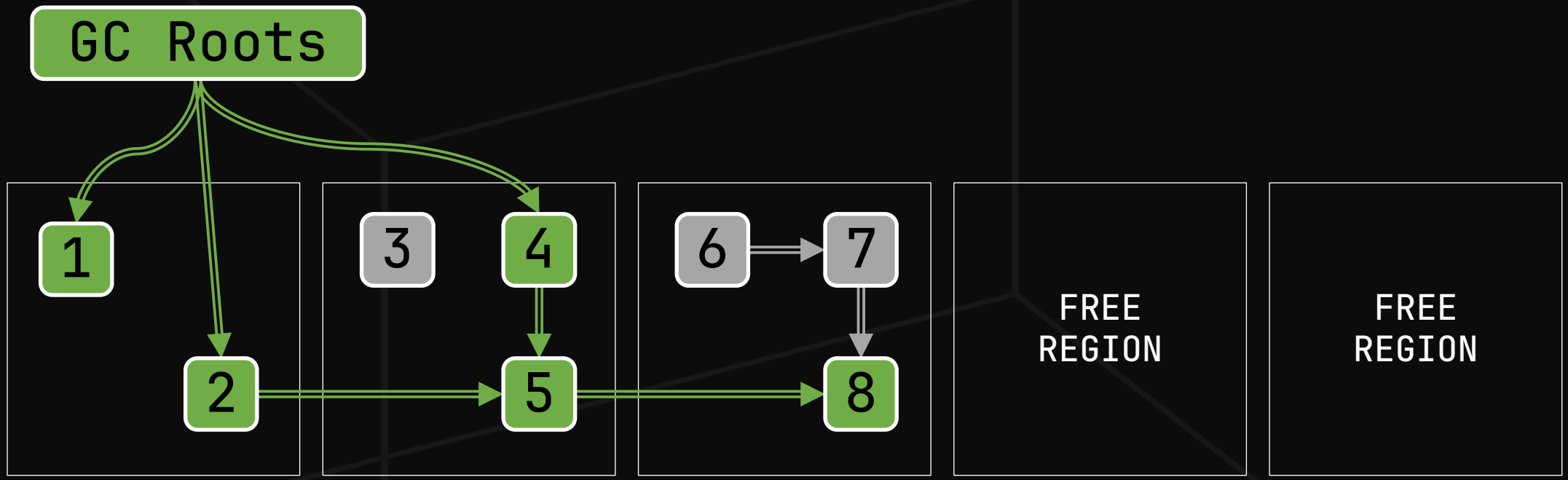
Правильный цвет: Marked0



Цвета: Finalizeble | Marked1 | Remapped | Marked0

Стадия GC: *Pause Mark End*

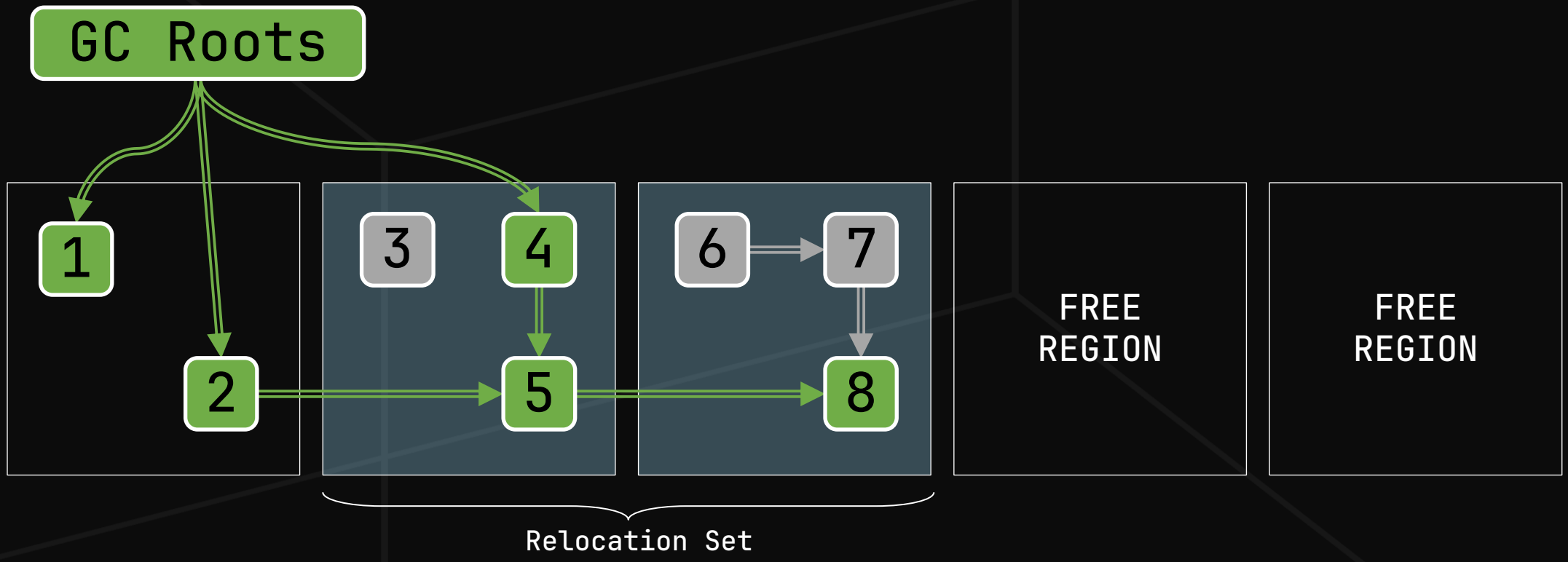
Правильный цвет: Marked0



Цвета: Finalizeble | Marked1 | Remapped | Marked0

Стадия GC: *Prepare for Relocate*

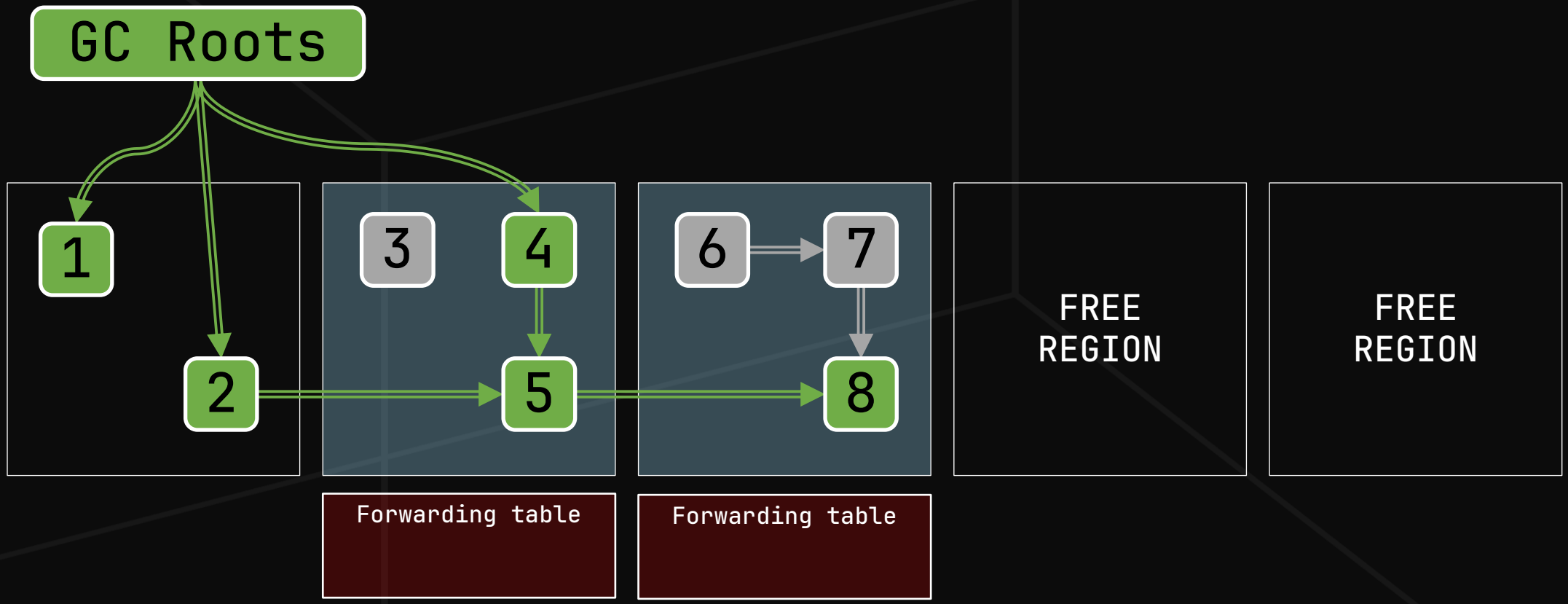
Правильный цвет: Marked0



Цвета: Finalizeble | Marked1 | Remapped | Marked0

Стадия GC: *Prepare for Relocate*

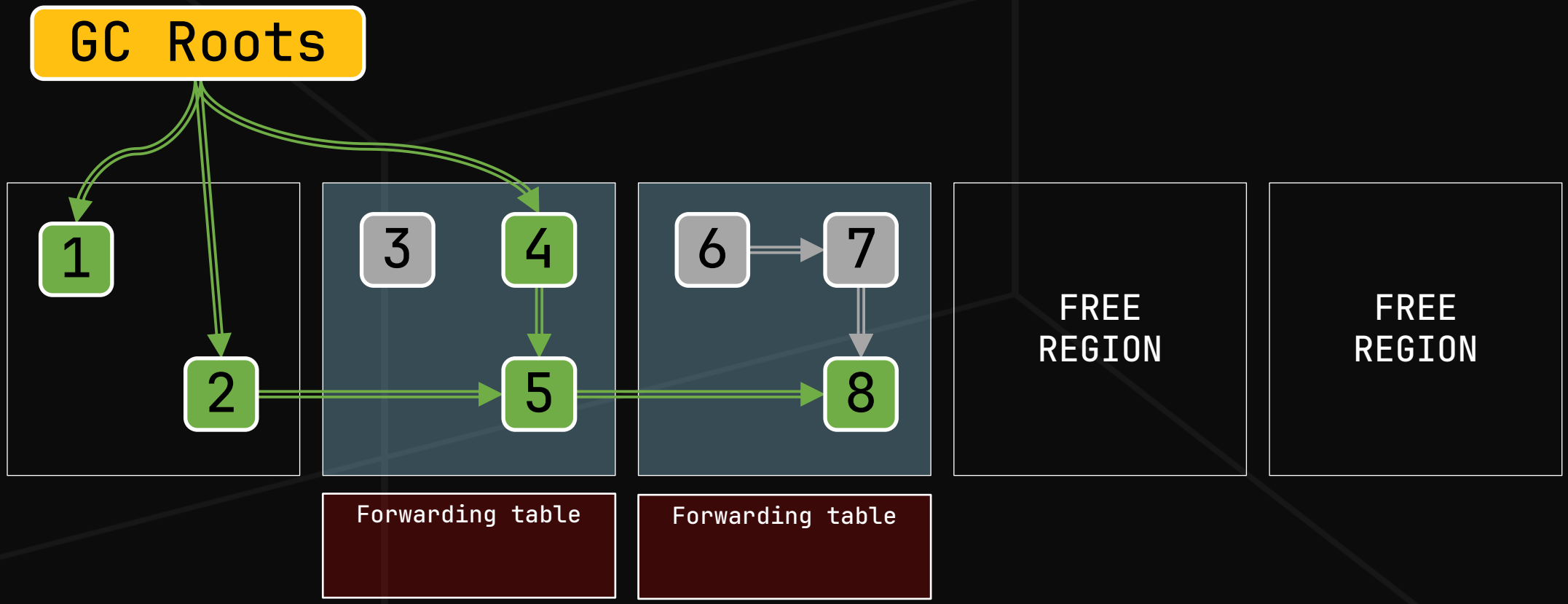
Правильный цвет: Marked0



Цвета: Finalizeble | Marked1 | Remapped | Marked0

Стадия GC: *Pause Relocate Start*

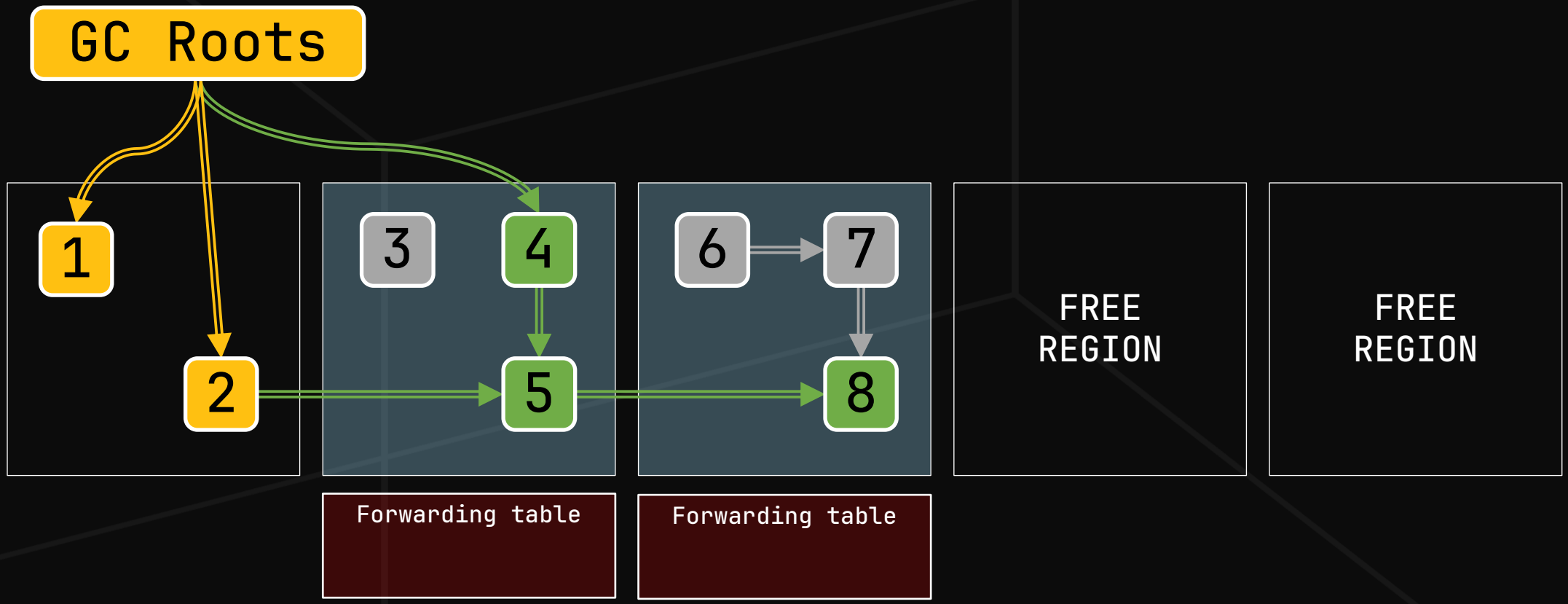
Правильный цвет: Remapped



Цвета: Finalizeble | Marked1 | Remapped | Marked0

Стадия GC: *Pause Relocate Start*

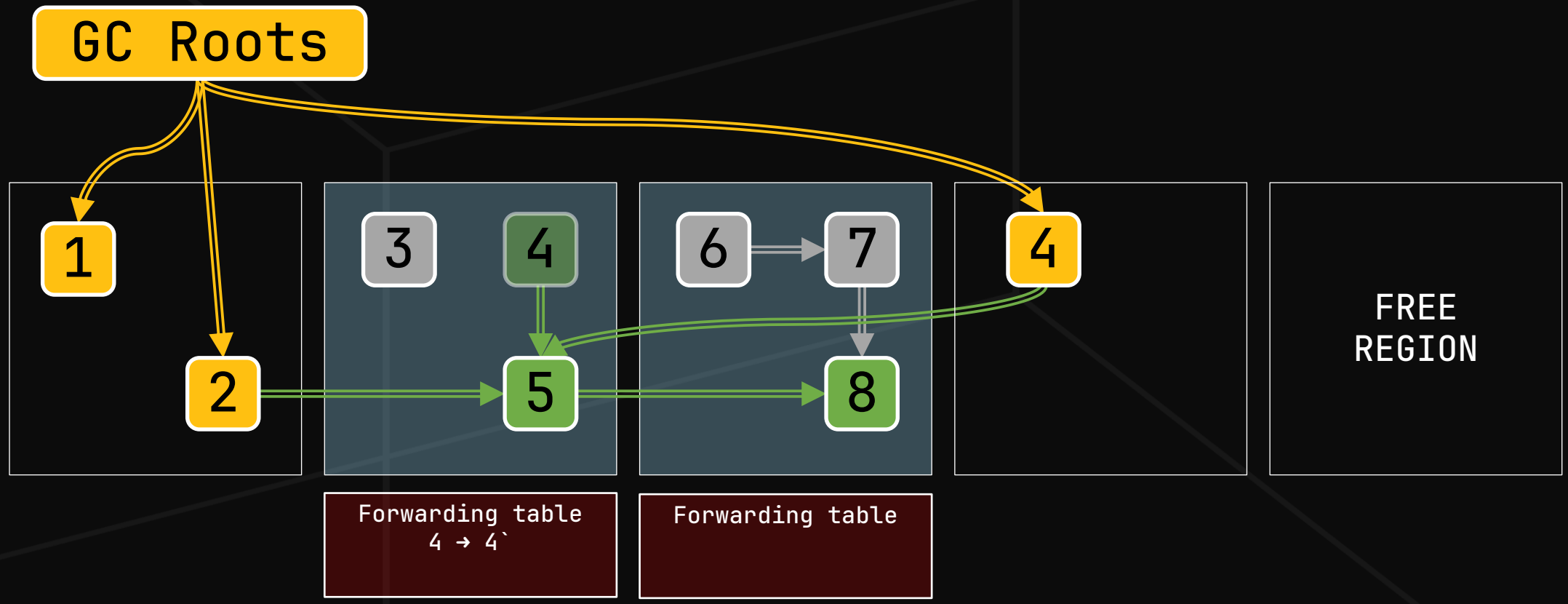
Правильный цвет: Remapped



Цвета: Finalizeble | Marked1 | Remapped | Marked0

Стадия GC: *Pause Relocate Start*

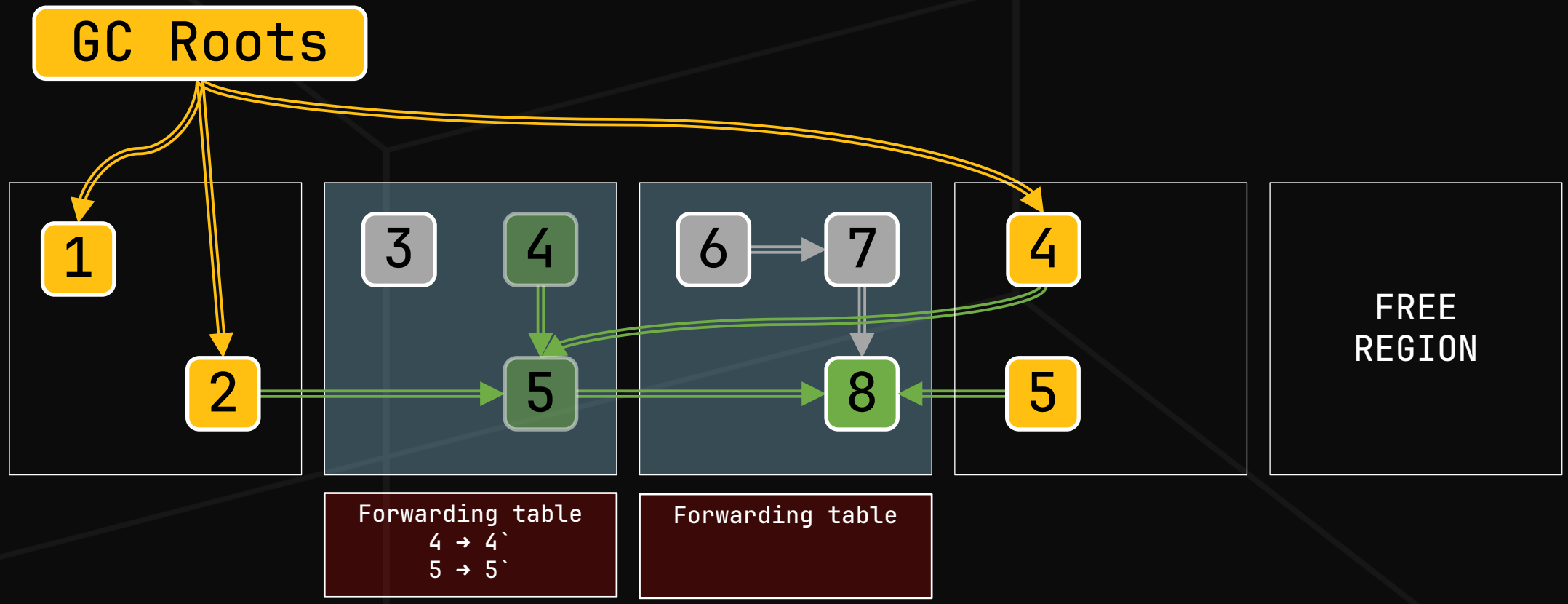
Правильный цвет: Remapped



Цвета: Finalizeble | Marked1 | Remapped | Marked0

Стадия GC: *Concurrent Relocate*

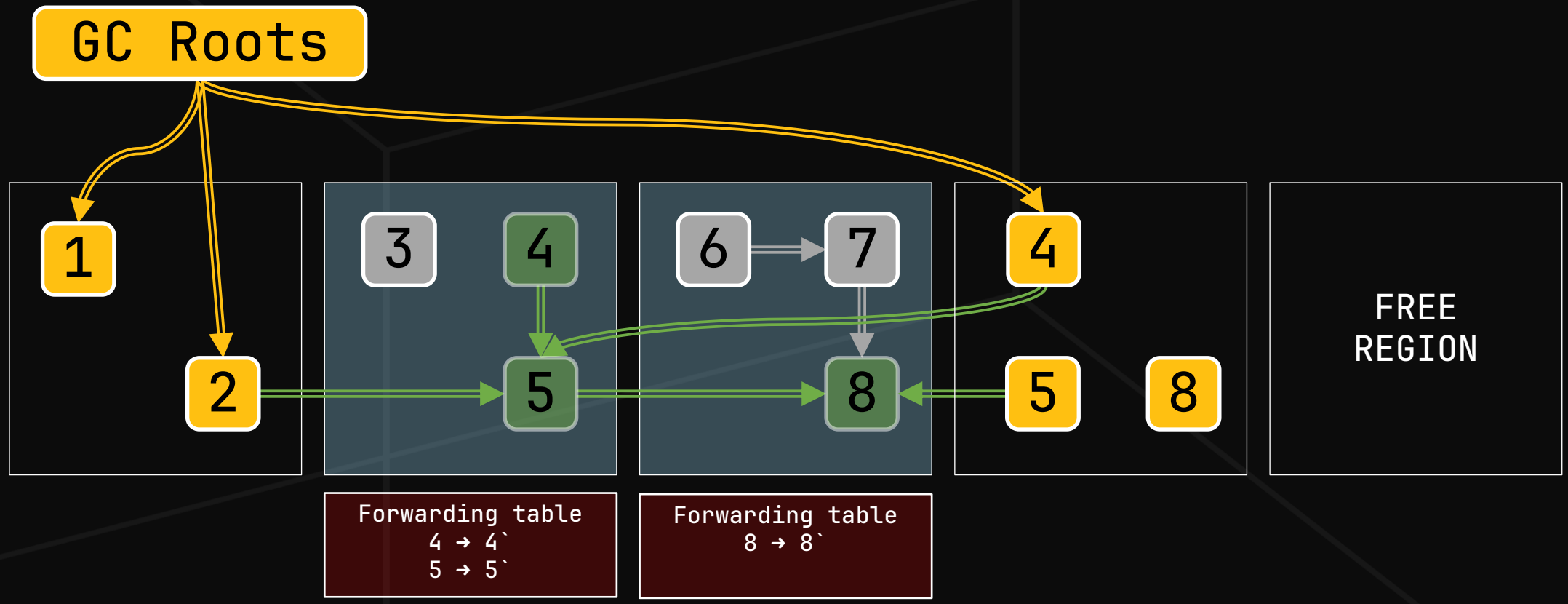
Правильный цвет: Remapped



Цвета: Finalizeble | Marked1 | Remapped | Marked0

Стадия GC: *Concurrent Relocate*

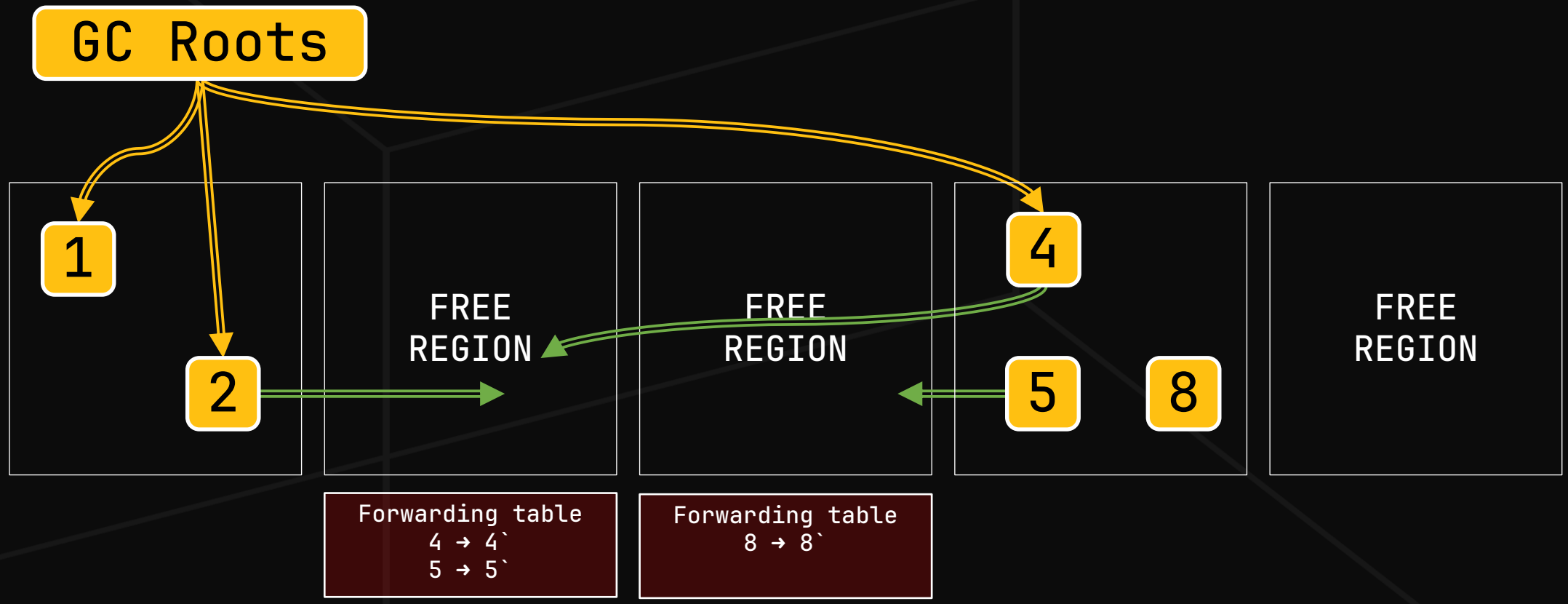
Правильный цвет: Remapped



Цвета: Finalizeble | Marked1 | Remapped | Marked0

Стадия GC: *Concurrent Relocate*

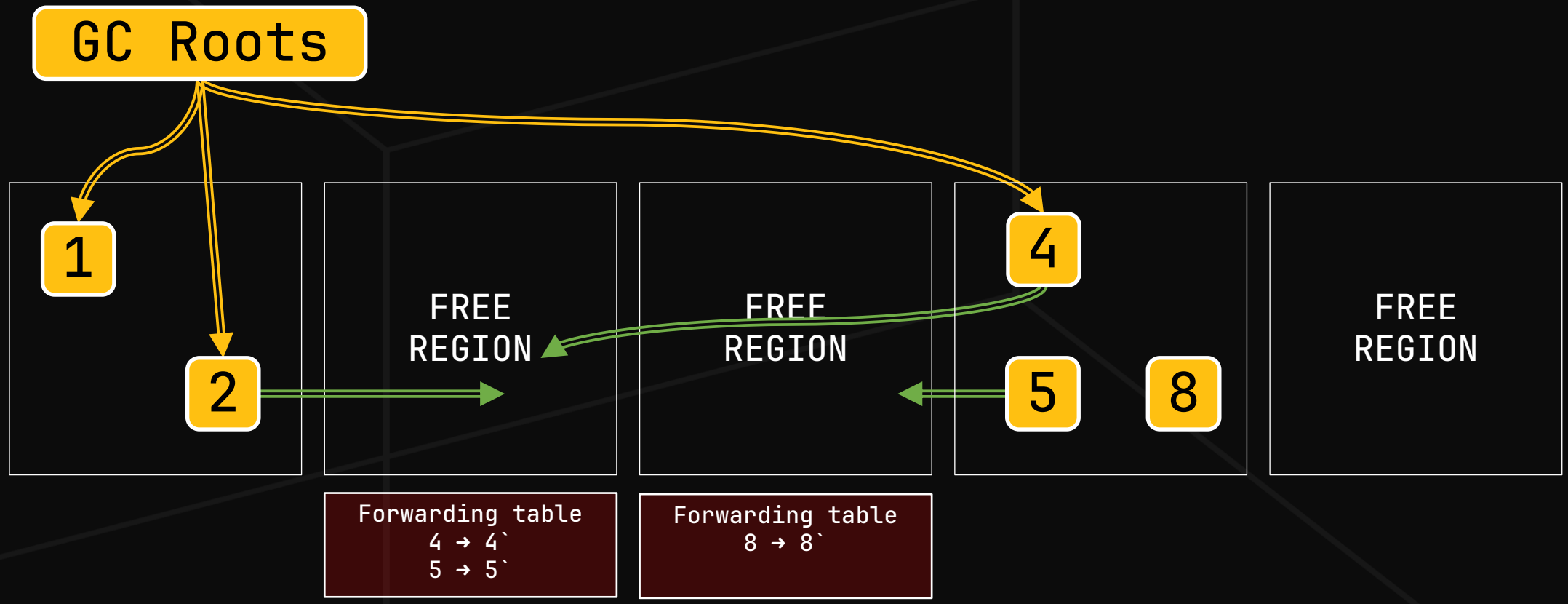
Правильный цвет: Remapped



Цвета: Finalizeble | Marked1 | Remapped | Marked0

Стадия GC: *Cycle Completed*

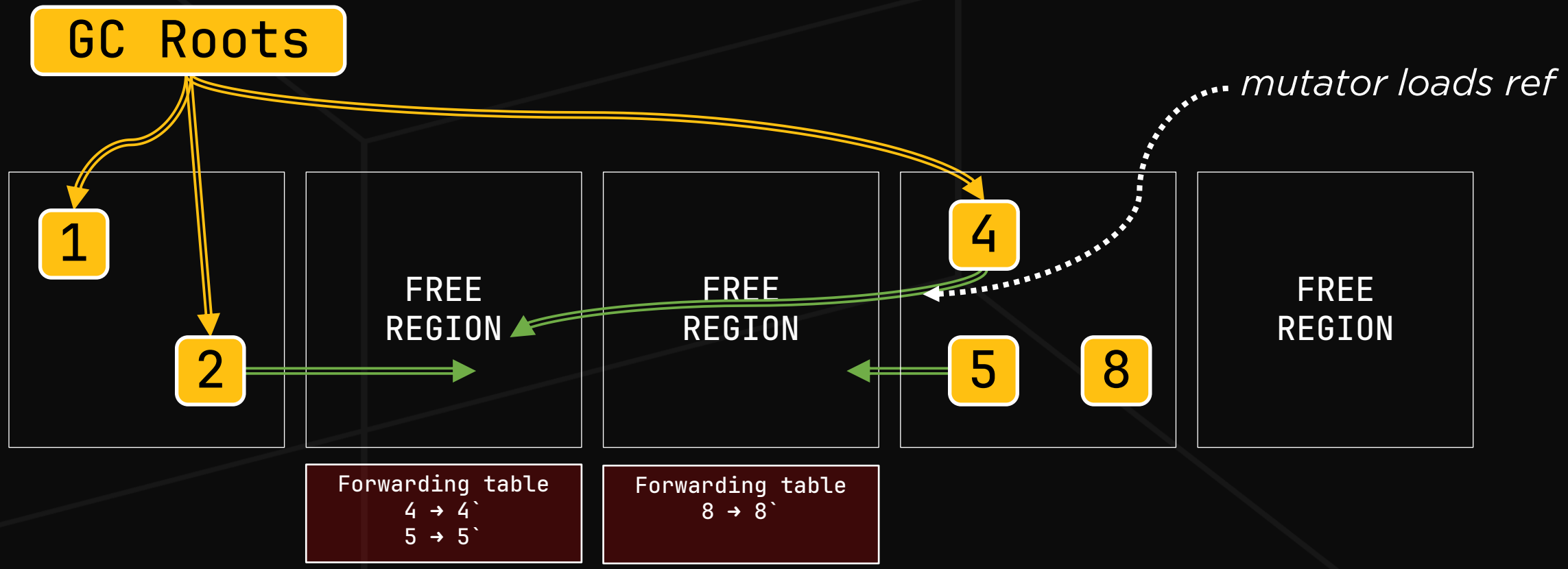
Правильный цвет: Remapped



Цвета: Finalizeble | Marked1 | Remapped | Marked0

Стадия GC: *Cycle Completed*

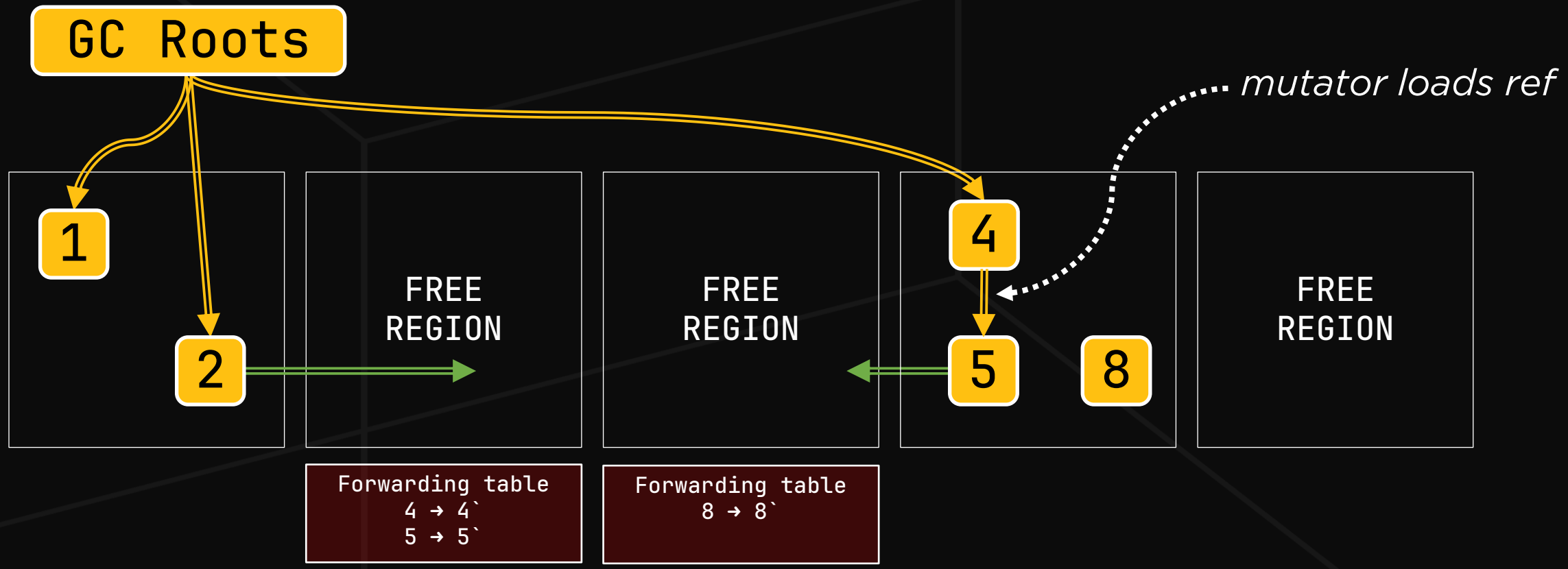
Правильный цвет: Remapped



Цвета: Finalizeble | Marked1 | Remapped | Marked0

Стадия GC: *Cycle Completed*

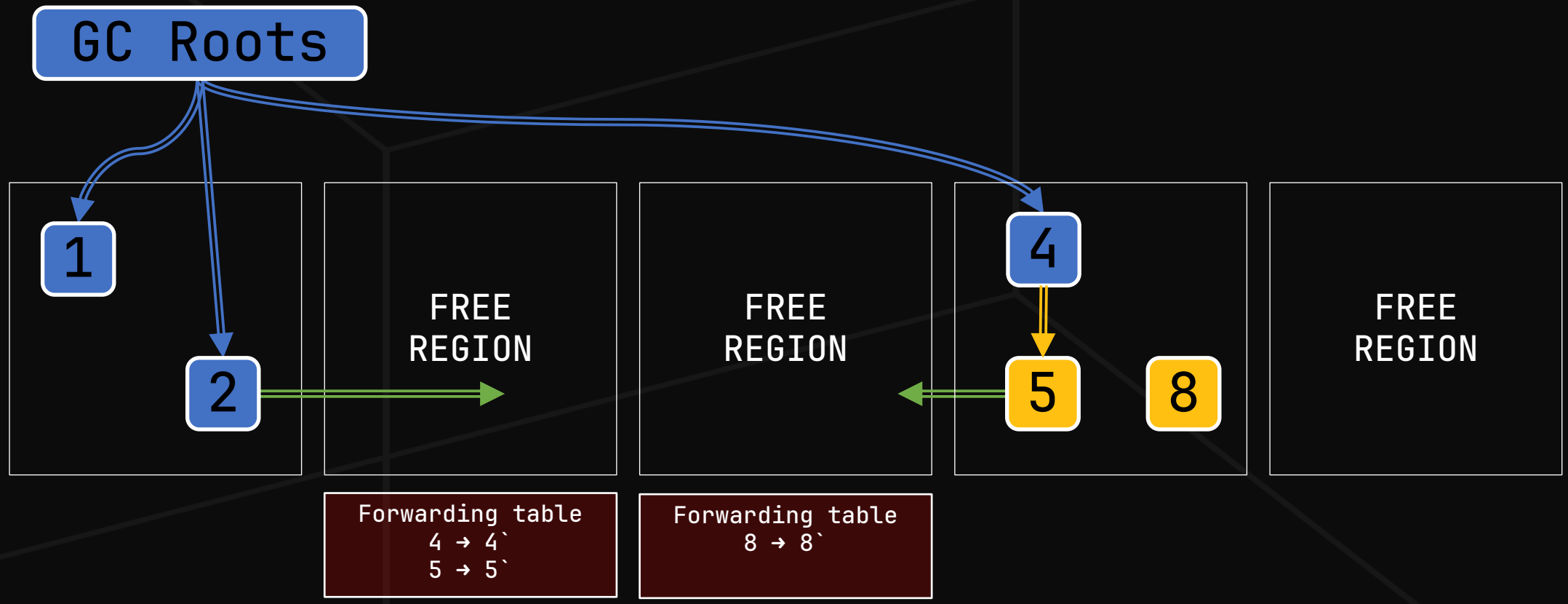
Правильный цвет: Remapped



Цвета: Finalizeble | Marked1 | Remapped | Marked0

Стадия GC: *Pause Mark Start*

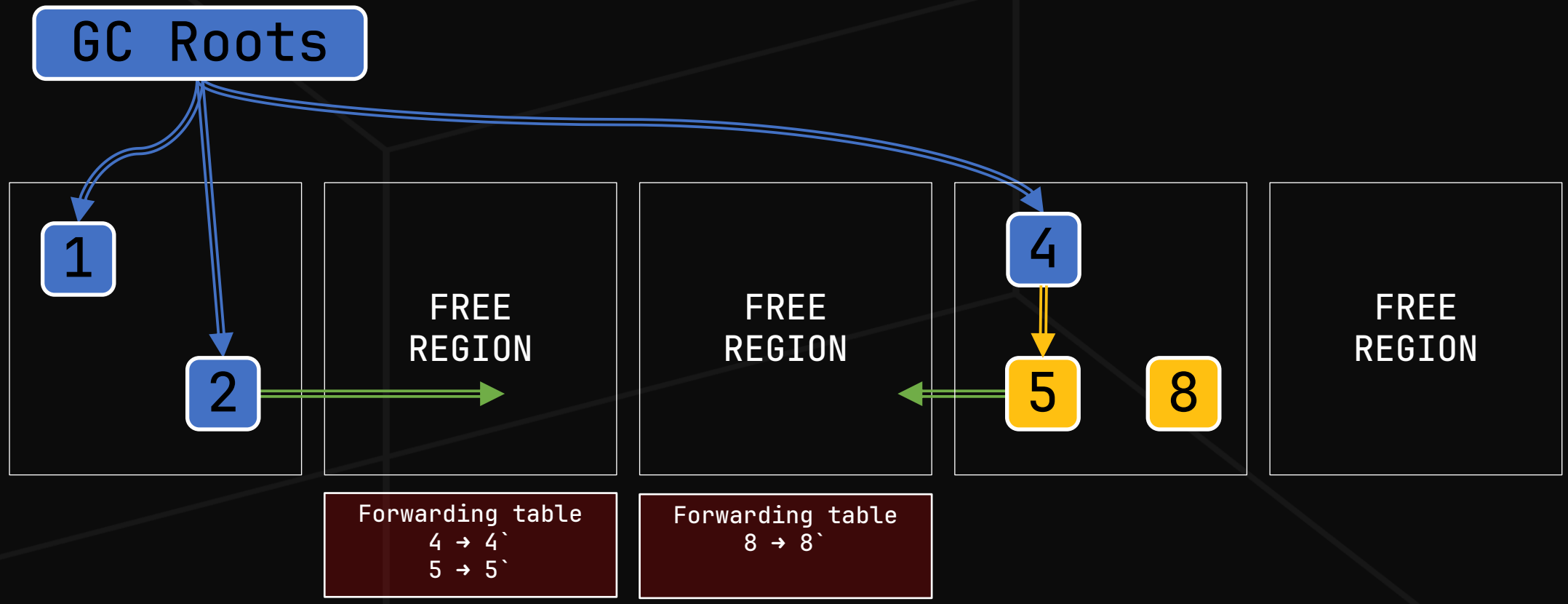
Правильный цвет: Marked1



Цвета: Finalizeble | Marked1 | Remapped | Marked0

Стадия GC: *Concurrent Mark*

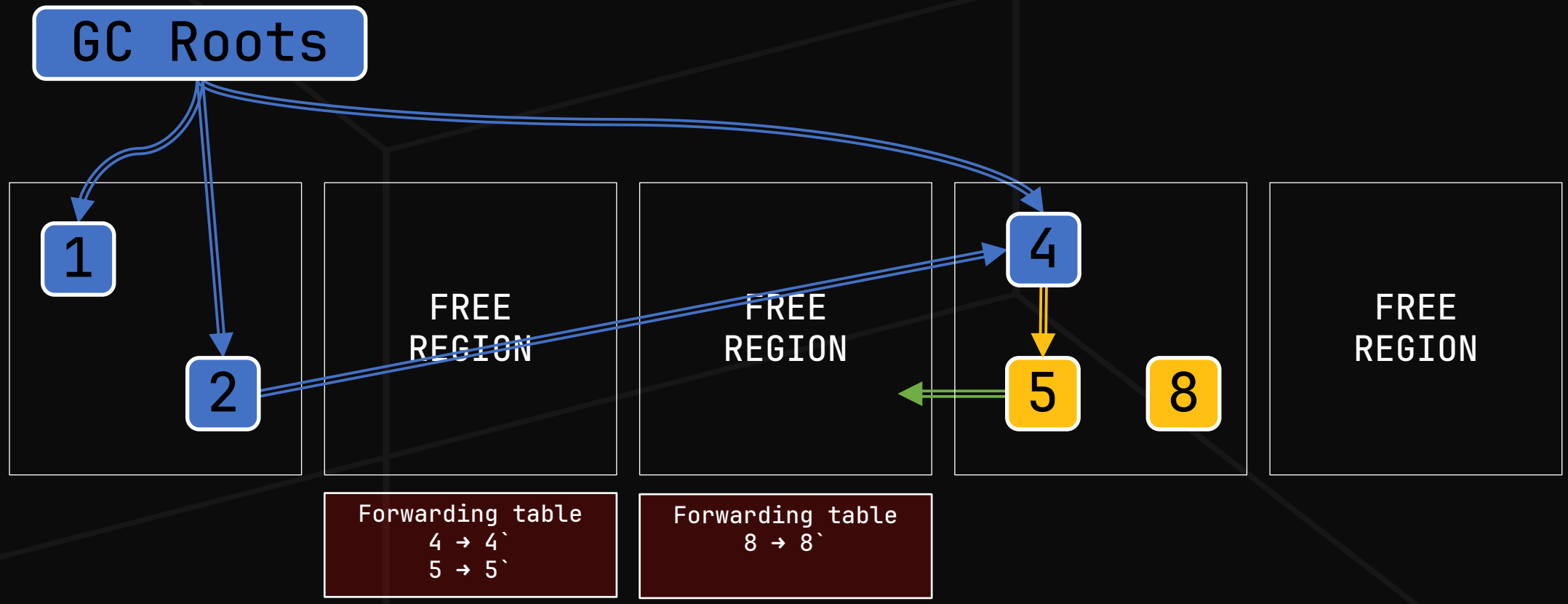
Правильный цвет: Marked1



Цвета: Finalizeble | Marked1 | Remapped | Marked0

Стадия GC: *Concurrent Mark*

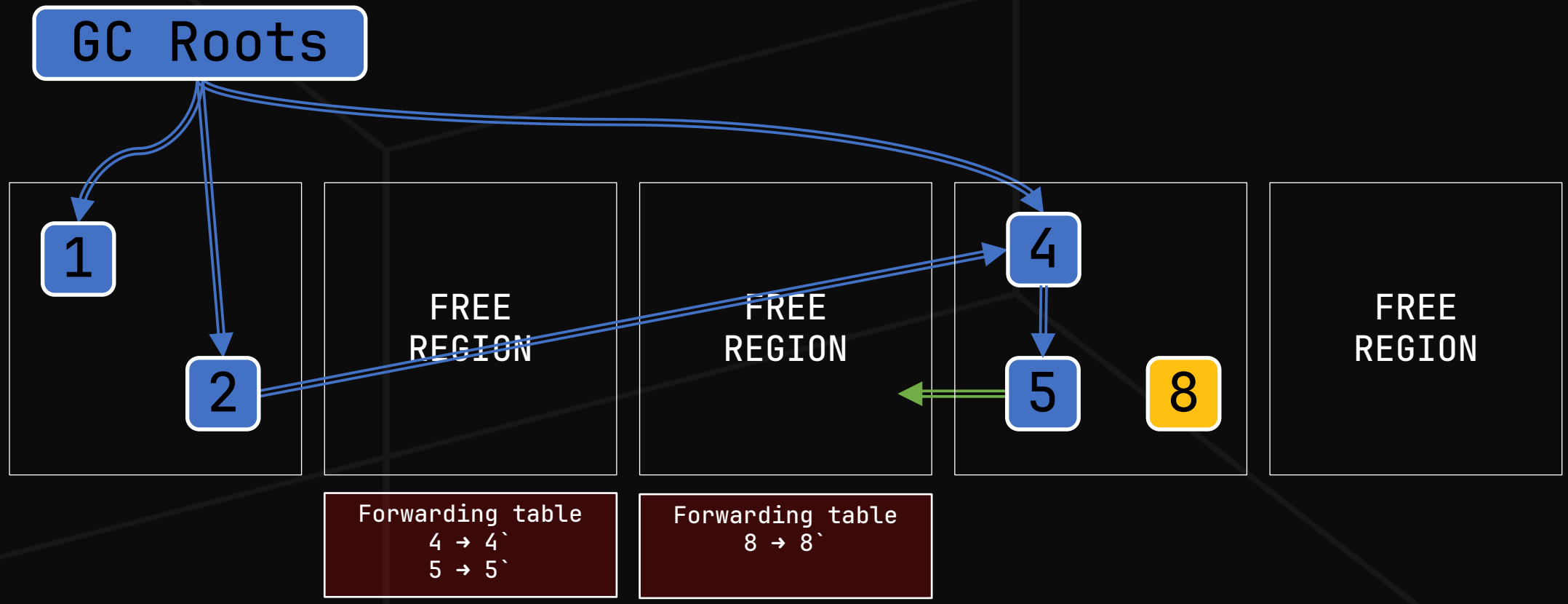
Правильный цвет: Marked1



Цвета: Finalizeble | Marked1 | Remapped | Marked0

Стадия GC: *Concurrent Mark*

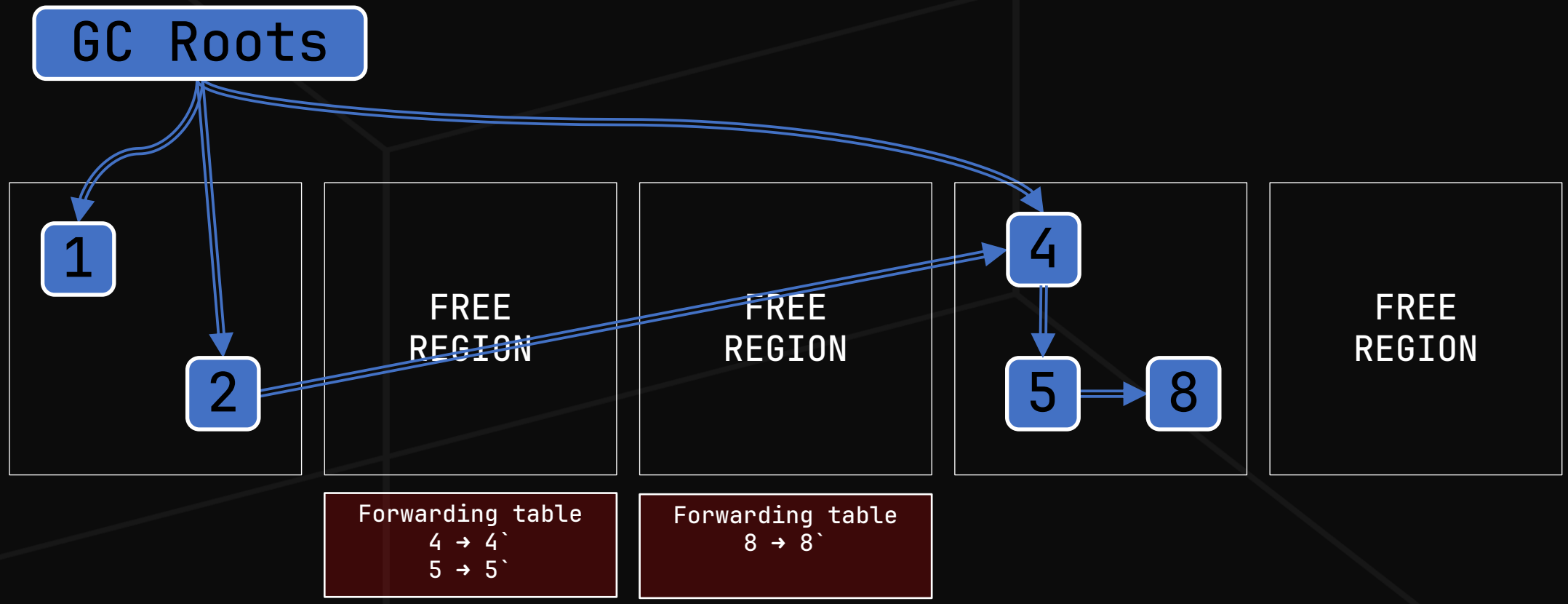
Правильный цвет: Marked1



Цвета: Finalizeble | Marked1 | Remapped | Marked0

Стадия GC: *Concurrent Mark*

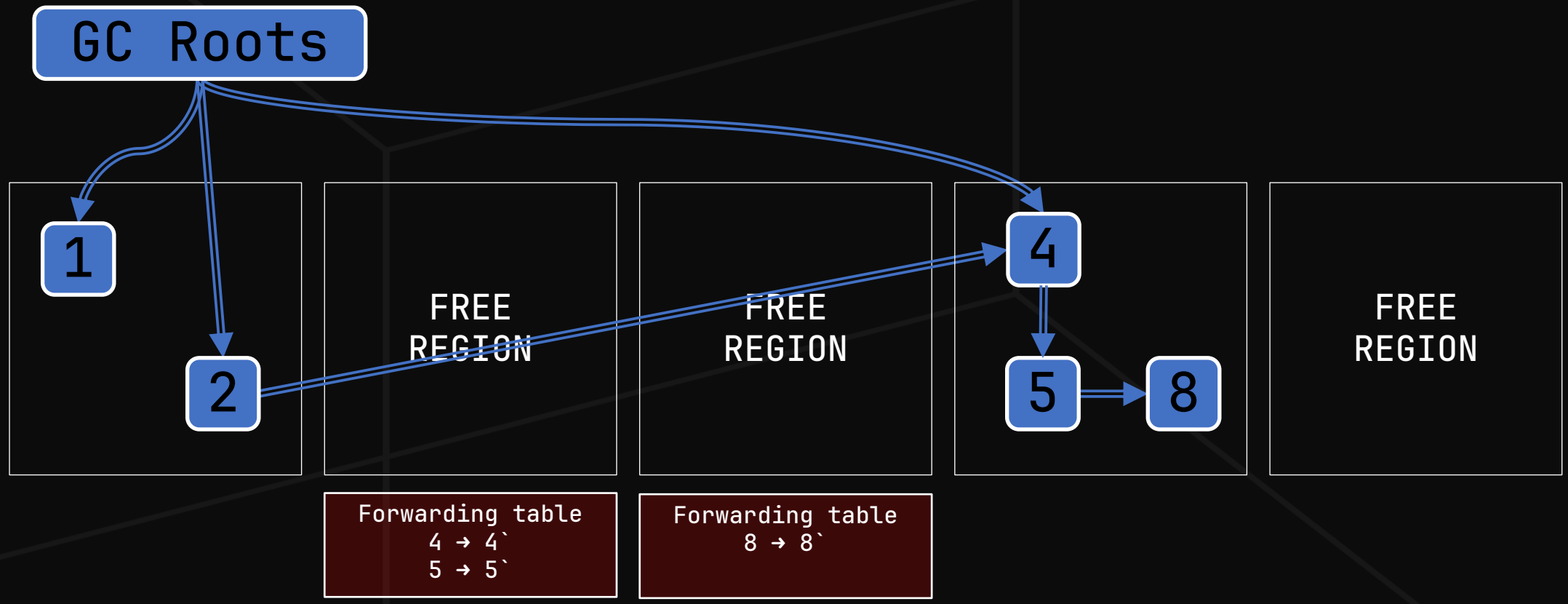
Правильный цвет: Marked1



Цвета: Finalizeble | Marked1 | Remapped | Marked0

Стадия GC: *Concurrent Mark*

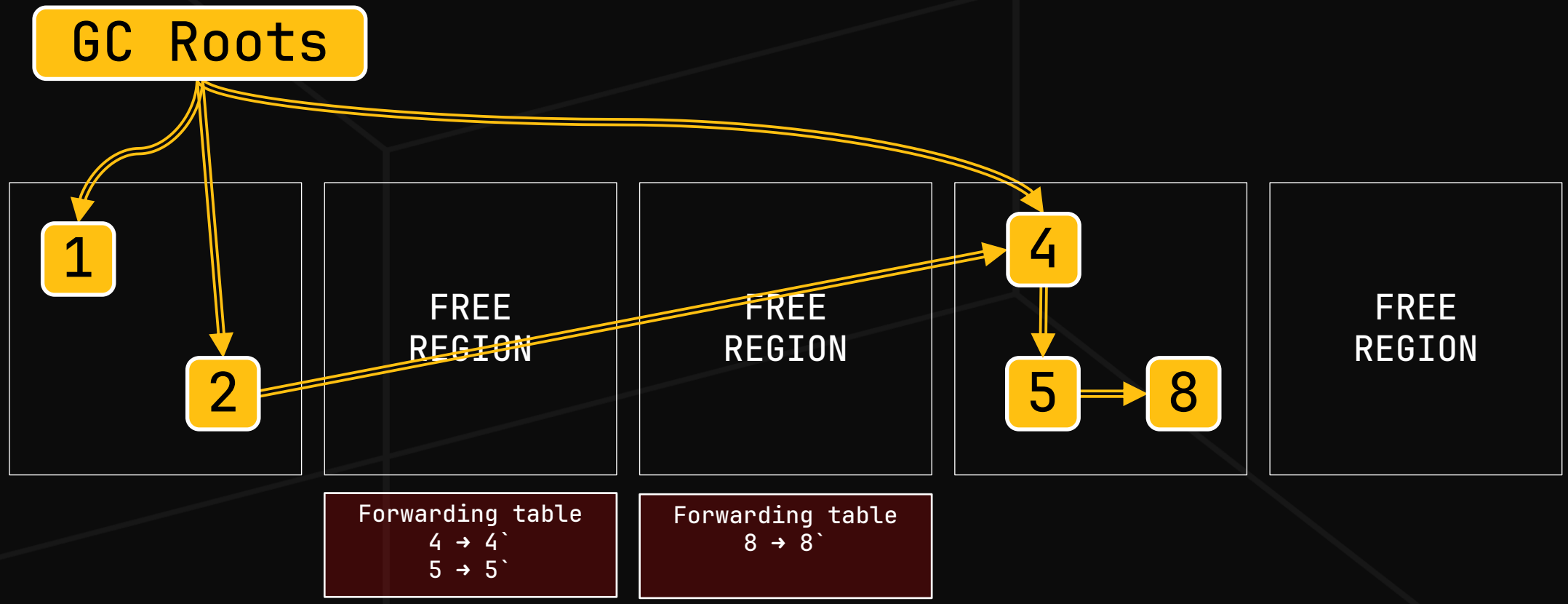
Правильный цвет: Marked1



Цвета: Finalizeble | Marked1 | Remapped | Marked0

Стадия GC: *Pause End + Relocate*

Правильный цвет: Remapped

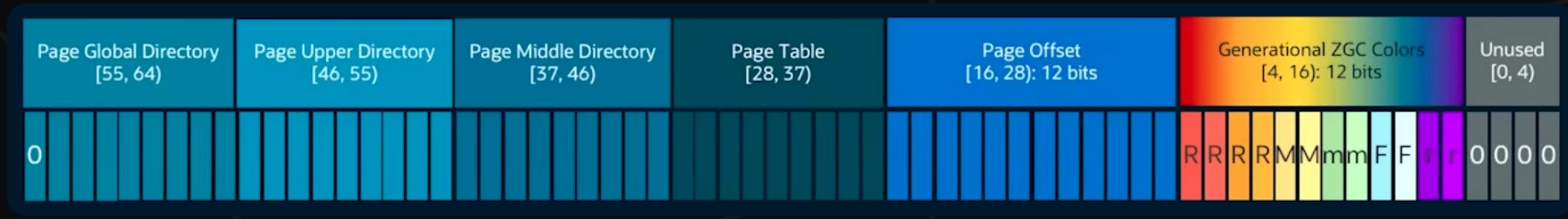


Цвета: Finalizeble | Marked1 | Remapped | Marked0

Стадия GC: *Pause End + Relocate*

Правильный цвет: Remapped

Generational ZGC



- Поддержка поколений
- Барьеры и на чтение и на запись
- Изменение схемы раскраски: 12 цветов
- Никаких мультимаппингов

«цвет» убирается/добавляется барьерами

ZGC Load/Store Barriers

```
1 ; NonGen ZGC Load Barrier
2 movq 0x10(%rax), %rcx
3 testq %rcx, 0x20(%r15)
4 jnz slow-path
```

```
1 ; Gen ZGC Load Barrier
2 movq 0x10(%rax), %rcx
3 shrq %rcx, $good_remap_bit_index
4 ja slow-path
```

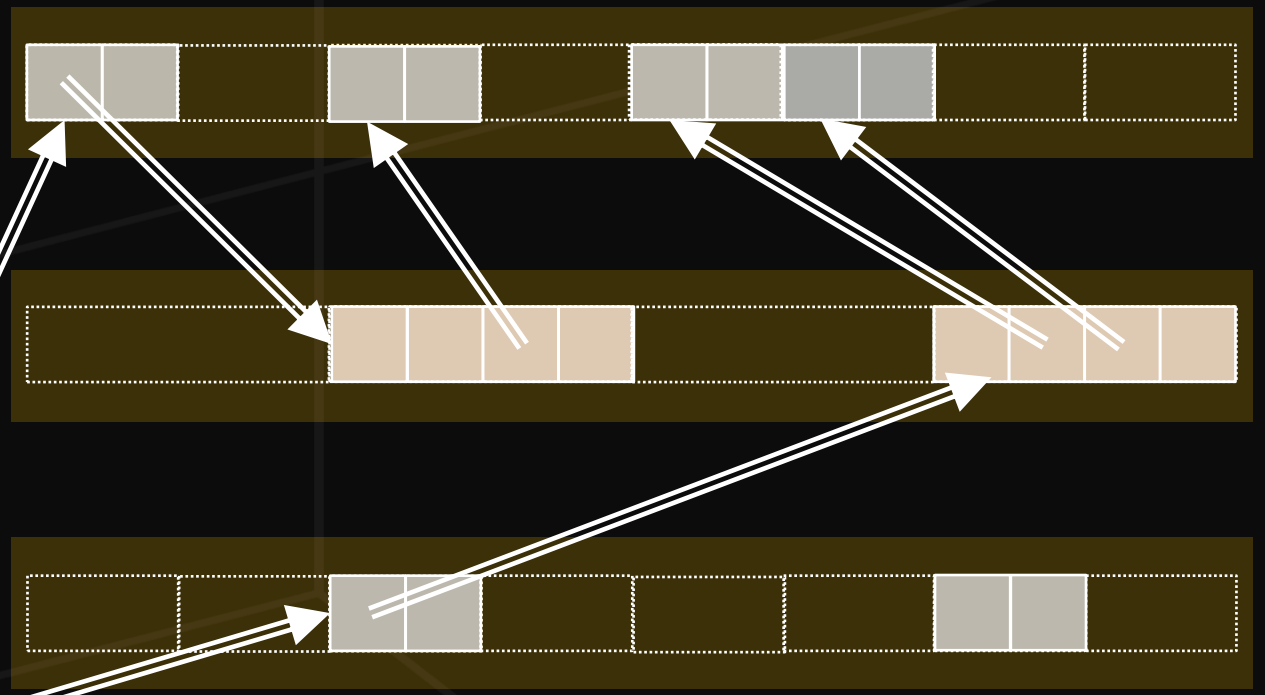
```
1 ; Gen ZGC Store Barrier
2 testl 0x10(%rax), $store_bad_mask
3 jnz slow_path
4 ; add color pointer before store
5 shlq %rcx, $good_remap_bit_index
6 orq %rcx, $store_good_color
7 movq %rcx, 0x10(%rax)
```

Generational ZGC

- **Практически полностью конкурентный GC**
  субмиллисекундные паузы
- **Региональный, поколенческий**
  12 цветов в 64-х битном указателе
- **Цена: снижение throughput**
  барьеры на чтение и на запись
- **Нет поддержки CompressedOops**

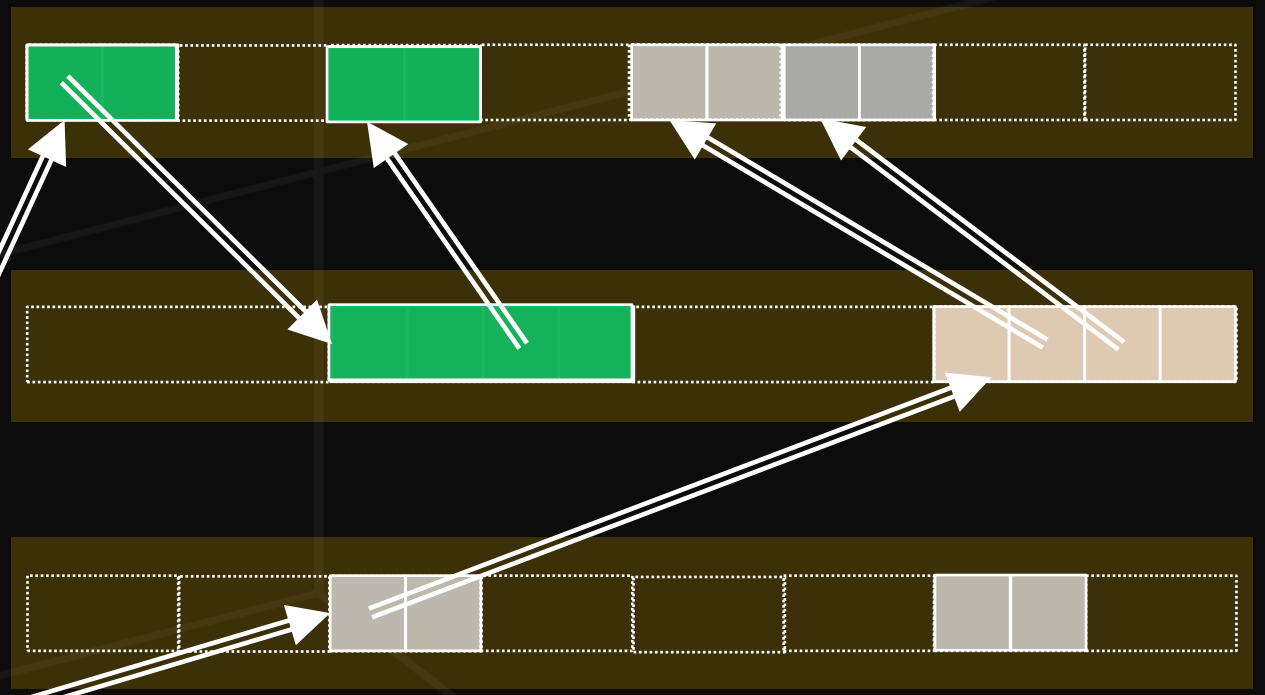
Go 1.25: Marking

```
type T struct{
    children *[4]*T
    value    int
}
```



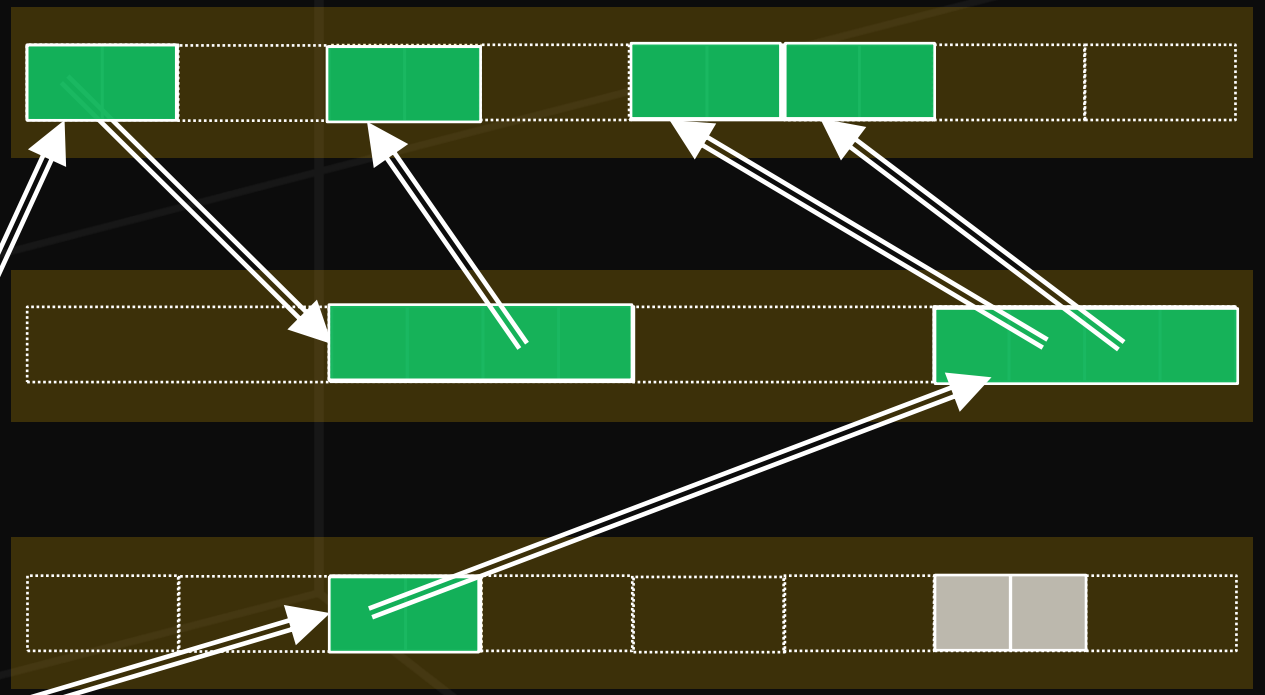
Go 1.25: Marking

```
type T struct{
    children *[4]*T
    value    int
}
```



Go 1.25: Marking

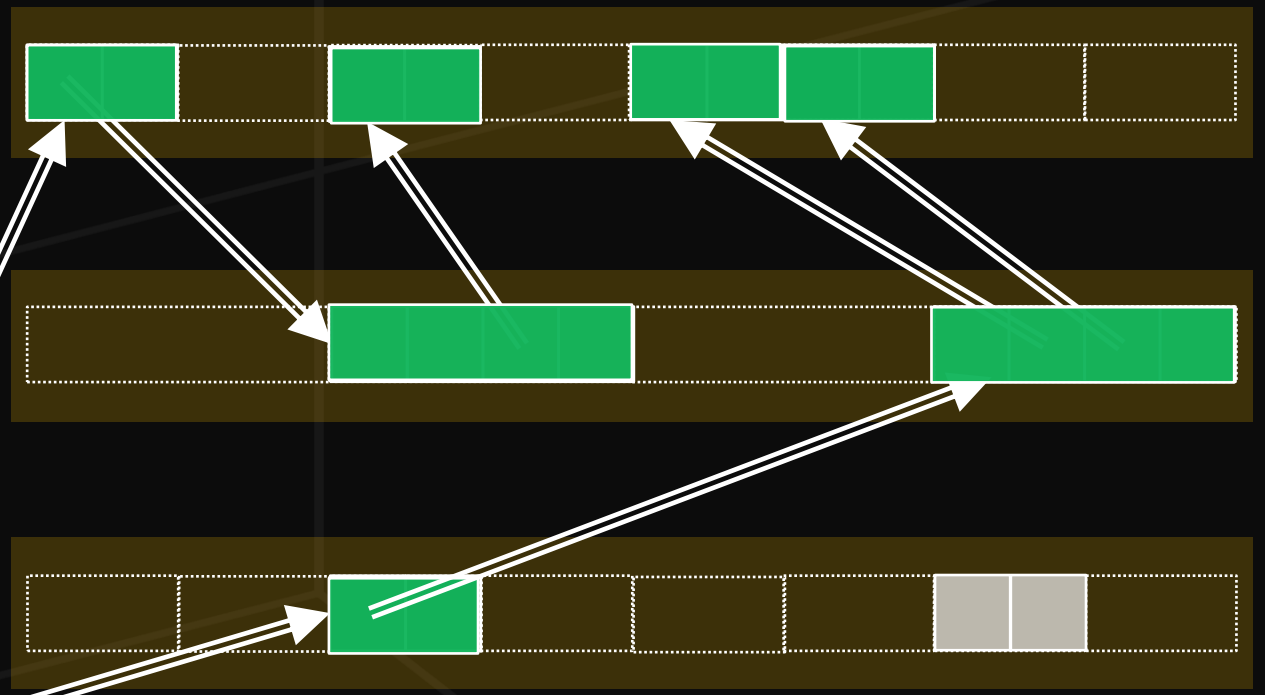
```
type T struct{
    children *[4]*T
    value    int
}
```



7 сканирований

Go 1.26: The Green Tea GC

```
type T struct{
    children *[4]*T
    value    int
}
```



4 сканирования

Go 1.26: The Green Tea GC

- **Concurrent Mark Sweep**
- **Оптимизация маркировки**
  на нее уходит 90% CPU от GC
- **Ходим не по объектам, а по спану целиком**

ШОК! Попугай научился говорить "зависит от контекста" и **устроился стафф-инженером!**

Java

- Копирование и перенос объектов
- Дешевая аллокация
- Все GC поддерживают поколения
- Возможность выбора алгоритма
  latency vs throughput

Go

- Объекты не перемещаются
- Хитроумный аллокатор
- Гипотезы о поколениях не находят своего места
- Выбор сделан: *latency*
  Пишите горизонтально масштабируемые приложения

Java: как выбрать коллектор?

Важно latency	*ZGC, Shenandoah*
Важен throughput	*Parallel*
Баланс latency / throughput	*современный G1*
Избыток CPU	*Zgc, Shenandoah*
Гигантские хипы	*Zgc*
Хип меньше 32Gb	*CompressedOops: не Zgc*

Спасибо за внимание!