Shenandoah GC

сборщик мусора, который смог

Aleksey Shipilёv

shade@redhat.com
@shipilev

Safe Harbor / Тихая Гавань

Everything on this and any subsequent slides may be a lie. Do not base your decisions on this talk. If you do, ask for professional help.

Всё что угодно на этом слайде, как и на всех следующих, может быть враньём. Не принимайте решений на основании этого доклада. Если всё-таки решите принять, то наймите профессионалов.

redhat.

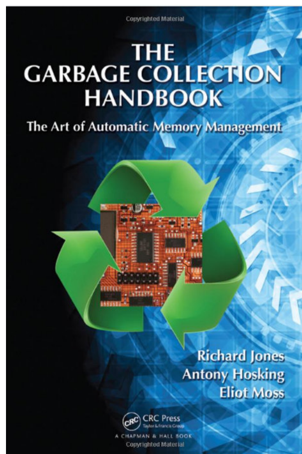
Дисклеймеры (как обычно)

Этот доклад:

1. ~~...рассказывает про многопоточность, а не сборку мусора.~~ **Наконец-то** рассказывает про сборку мусора!
2. Рассказывает про сборку мусора **вообще** (насколько это позволяет время), и про Shenandoah **в частности**
3. Рассказывает быстро, бодро, беспощадно
4. Рассказывает про **плюсы** и **минусы** алгоритмов сборки

redhat.

Минутка рекламы

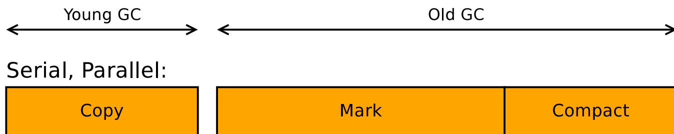


- По моему ~~скромному~~ мнению, серьёзно что-то обсуждать, не прочитав GC Handbook – к долгим печалям и хождению по кругу
- Это только кажется, что $\$name$ GC – это супер-инновация: на деле многие GC используют учебник в хвост и гриву

redhat

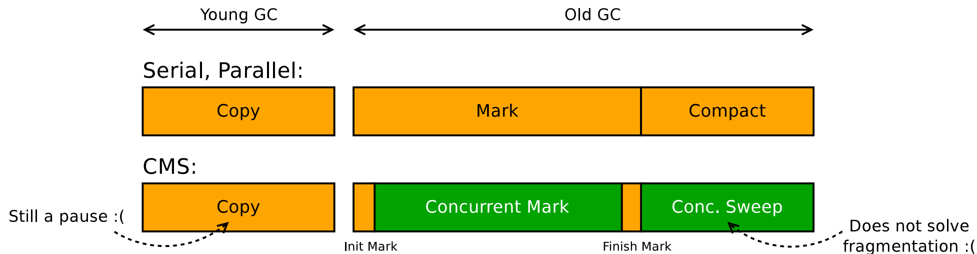
Крупно

Крупно: ландшафт



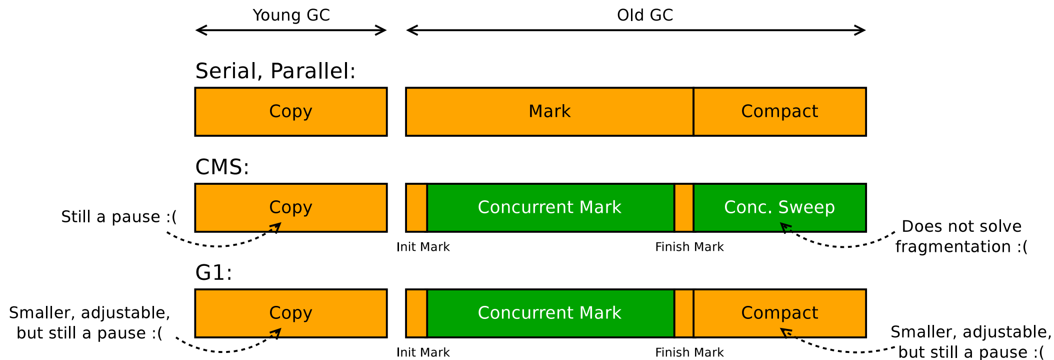
redhat

Крупно: ландшафт



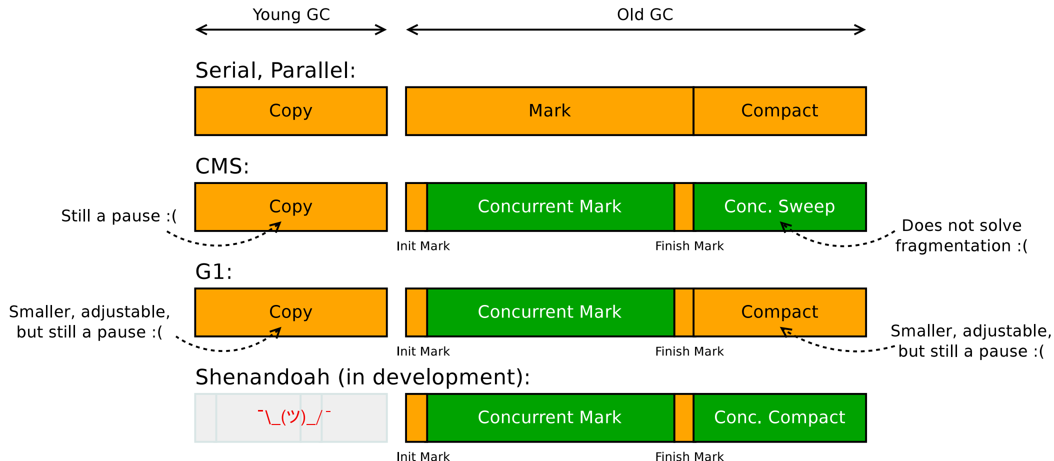
redhat.

Крупно: ландшафт



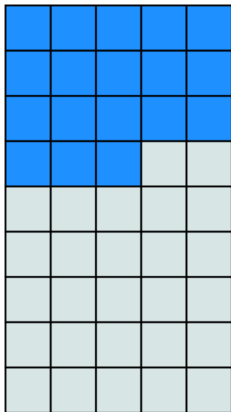
redhat

Крупно: ландшафт



redhat.

Крупно: куча



Shenandoah – регионализованный коллектор

Похож на G1:

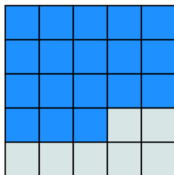
- Разбиением на регионы
- *Пока что* политикой сборки: в первую очередь собирает регионы с большим количеством мусора

Не похож на G1:

- Отсутствием деления на young/old сборки
- Учётом ссылок между регионами

redhat.

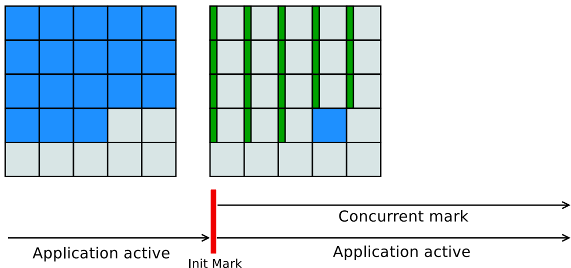
Крупно: цикл



Application active

redhat

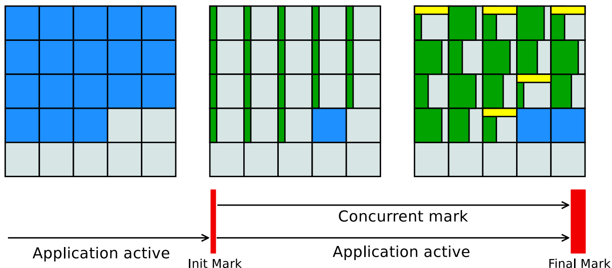
Крупно: цикл




```
Pause Init Mark 0.419ms
```


redhat

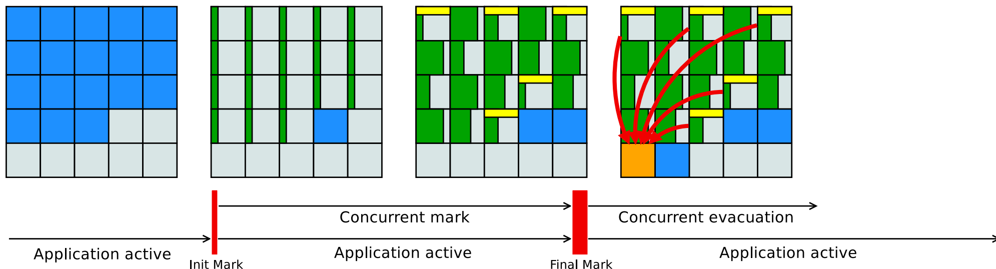
Крупно: цикл




```
Pause Init Mark 0.419ms
Concurrent marking 13664M->13808M(16384M) 458.434ms
Pause Final Mark 13808M->8408M(16384M) 0.986ms
```


redhat.

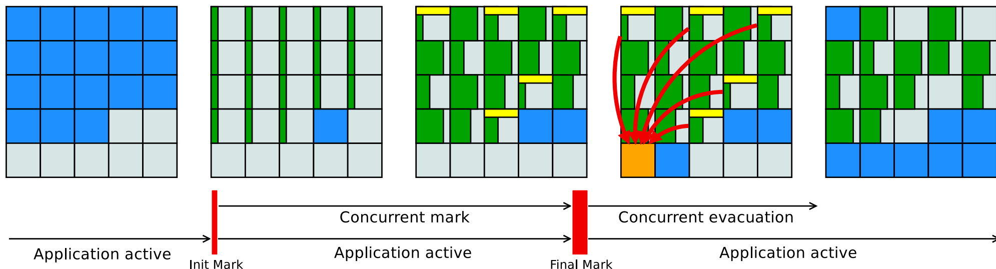
Крупно: цикл




```
Pause Init Mark 0.419ms
Concurrent marking 13664M->13808M(16384M) 458.434ms
Pause Final Mark 13808M->8408M(16384M) 0.986ms
Concurrent evacuation 8408M->9704M(16384M) 229.654ms
```


redhat.

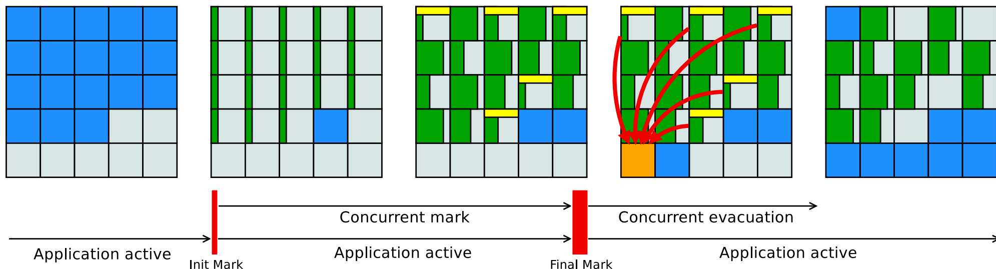
Крупно: цикл




```
Pause Init Mark 0.419ms
Concurrent marking 13664M->13808M(16384M) 458.434ms
Pause Final Mark 13808M->8408M(16384M) 0.986ms
Concurrent evacuation 8408M->9704M(16384M) 229.654ms
```


redhat.

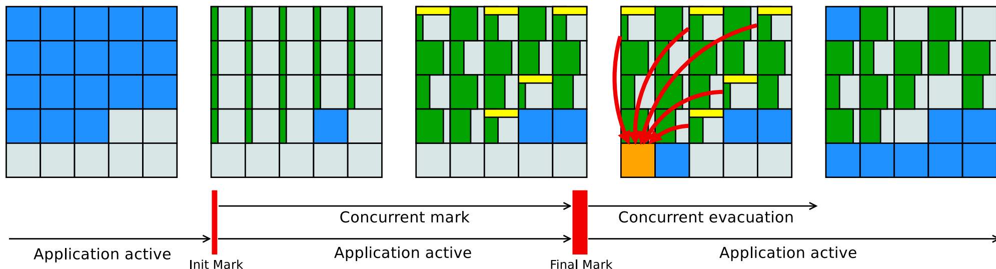
Крупно: цикл




```
Pause Init Mark 0.419ms
Concurrent marking 13664M->13808M(16384M) 458.434ms
Pause Final Mark 13808M->8408M(16384M) 0.986ms
Concurrent evacuation 8408M->9704M(16384M) 229.654ms
```


redhat.

Крупно: цикл




```
Pause Init Mark 0.419ms
Concurrent marking 13664M->13808M(16384M) 458.434ms
Pause Final Mark 13808M->8408M(16384M) 0.986ms
Concurrent evacuation 8408M->9704M(16384M) 229.654ms
```


redhat.

Concurrent Mark

Concurrent Mark: достижимость

Чтобы найти мусор, нужно ~~думать как мусор~~
узнать, есть ли ссылки на объект

Три подхода:

1. **No-op**: забить и считать всё достижимым
2. **Mark-*** : Пробежаться по графу объектов, найти достижимое и посчитать *всё остальное* мусором
3. **Reference counting**: на каждом чтении/записи считать количество ссылок на объект, при refcount=0 считать объект мусором

redhat

Concurrent Mark: трёхцветная абстракция

Граф объектов можно его обойти, назначая объектам цвета:

1. Белый: ещё не посещён
2. Серый: посещён, но ссылки не просканированы
3. Чёрный: посещён и ссылки просканированы

redhat.

Concurrent Mark: трёхцветная абстракция

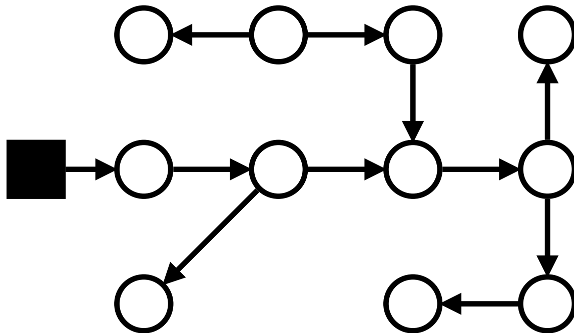
Граф объектов можно его обойти, назначая объектам цвета:

1. Белый: ещё не посещён
2. Серый: посещён, но ссылки не просканированы
3. Чёрный: посещён и ссылки просканированы

Минутка уныния: вся жизнь алгоритма маркировки – это покраска белого в серое, а серого в чёрное.

redhat

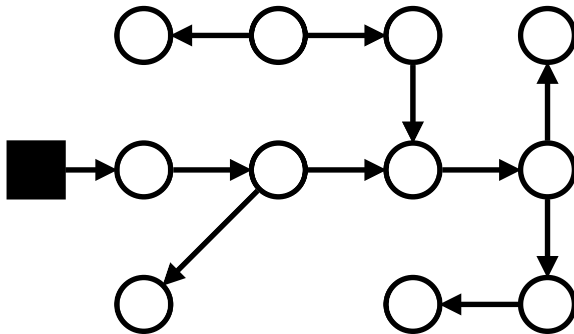
Concurrent Mark: stop-the-world mark



Когда приложение остановлено, всё тривиально!
Никто не мешается под ногами.

redhat

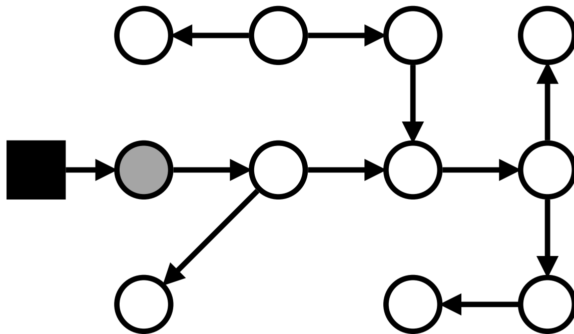
Concurrent Mark: stop-the-world mark



Нашли все корни, покрасили их в чёрный,
т.к. они по определению достижимы

redhat

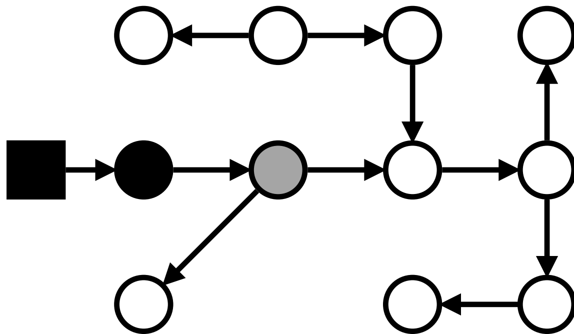
Concurrent Mark: stop-the-world mark



Ссылки из чёрных теперь серые, сканируем ссылки из серых;
серые – это wavefront обхода

redhat

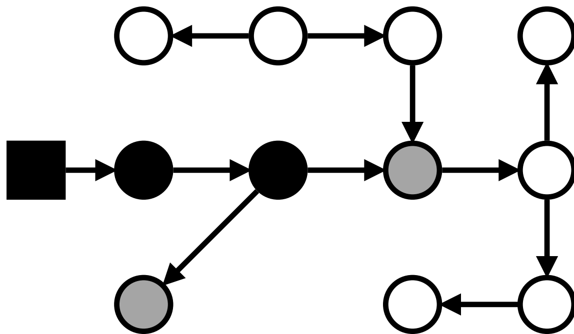
Concurrent Mark: stop-the-world mark



Сканирование из серых завершено, красим их в чёрные;
новые ссылки – серые

redhat

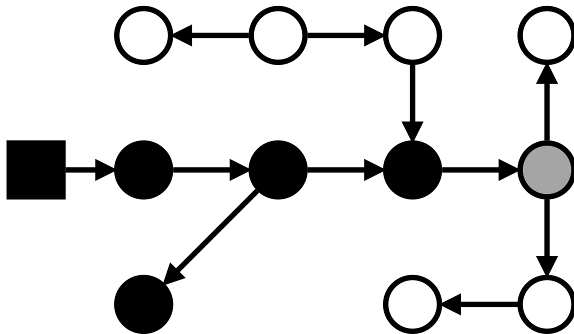
Concurrent Mark: stop-the-world mark



Серые → чёрные;
достижимые из серых → серые

redhat.

Concurrent Mark: stop-the-world mark

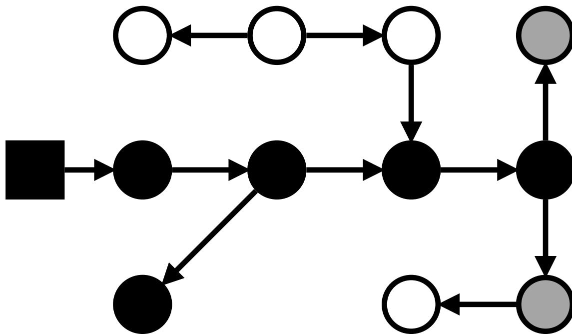


Серые → чёрные;

достижимые из серых → серые

redhat.

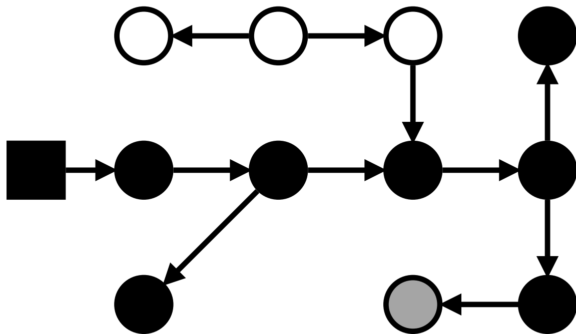
Concurrent Mark: stop-the-world mark



Серые → чёрные;
достижимые из серых → серые

redhat.

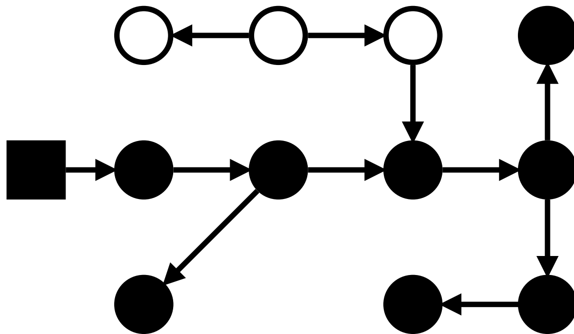
Concurrent Mark: stop-the-world mark



Серые → чёрные;
достижимые из серых → серые

redhat.

Concurrent Mark: stop-the-world mark



Конец: всё достижимое – чёрное;
весь мусор – белый

redhat.

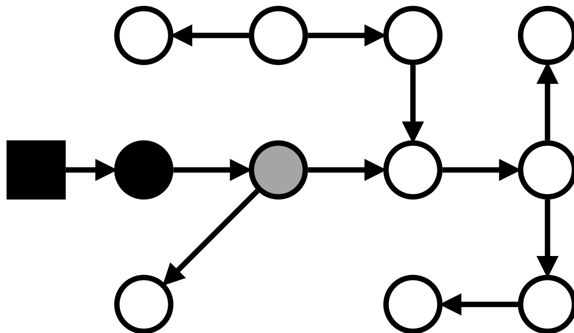
Concurrent Mark: проблемы с мутатором

В **concurrent** mark всё сложнее: там есть приложение, которое меняет граф объектов.

За это его презрительно называют *мутатором*.

redhat

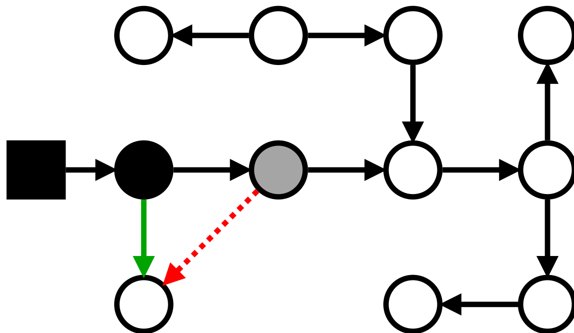
Concurrent Mark: проблемы с мутатором



Добрался wavefront сюда,
и только он начал сканировать ссылки...

redhat.

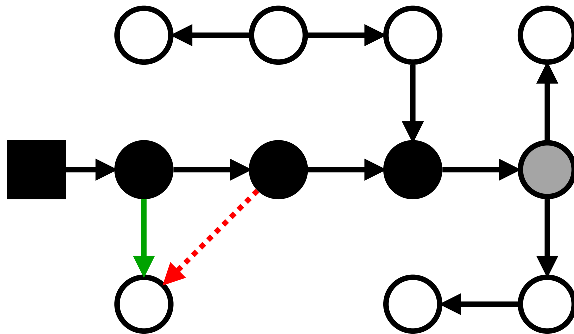
Concurrent Mark: проблемы с мутатором



Мутатор снёс ссылку из серого ...
и вставил её в чёрный!

redhat

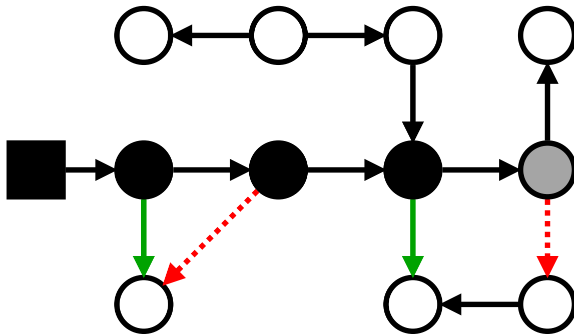
Concurrent Mark: проблемы с мутатором



Или даже когда-нибудь потом вставил ссылку на *транзитивно достижимый* белый объект

redhat

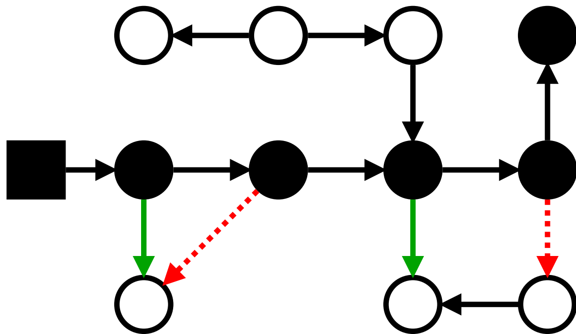
Concurrent Mark: проблемы с мутатором



Или даже когда-нибудь потом вставил ссылку на *транзитивно достижимый* белый объект

redhat.

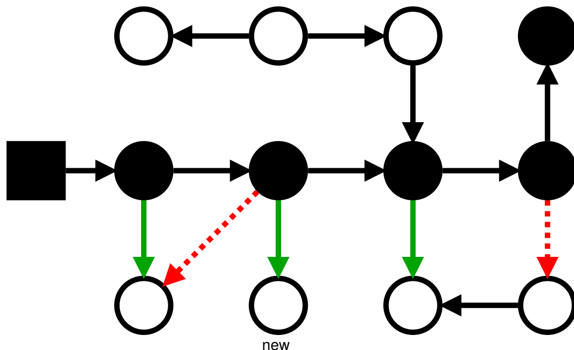
Concurrent Mark: проблемы с мутатором



Марк завершился, и опаньки: есть **достижимые** белые объекты, которые мы сейчас снесём!

redhat

Concurrent Mark: проблемы с мутатором



Ещё хуже: появился **новый** объект и ссылку на него записали под конец марка

redhat

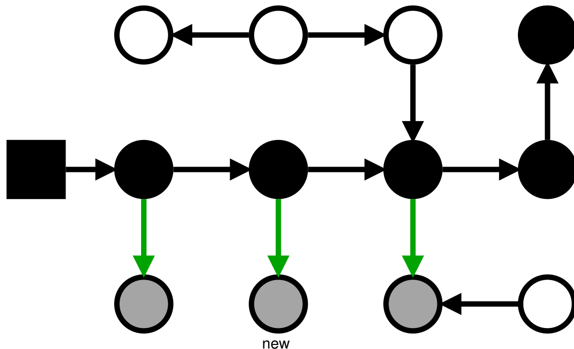
Concurrent Mark: способы решения

Оказывается, есть два способа решить эту проблему:

1. **Incremental Update**: перехватить записи и обработать *вставки*, обойдя новые ссылки – принимая новое на лету
2. **Snapshot-at-the-Beginning**: перехватить записи и обработать *удаления*, запомнив старые ссылки – уворачиваясь от деструктивных изменений

redhat.

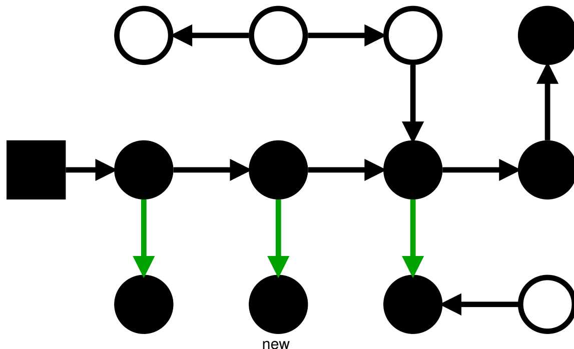
Concurrent Mark: Incremental Update



Красим все новые ссылки в серый

redhat.

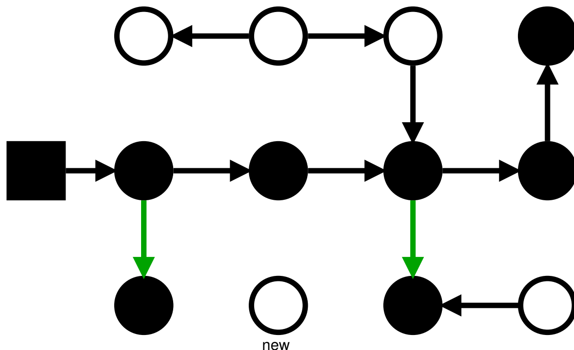
Concurrent Mark: Incremental Update



Конец!

redhat

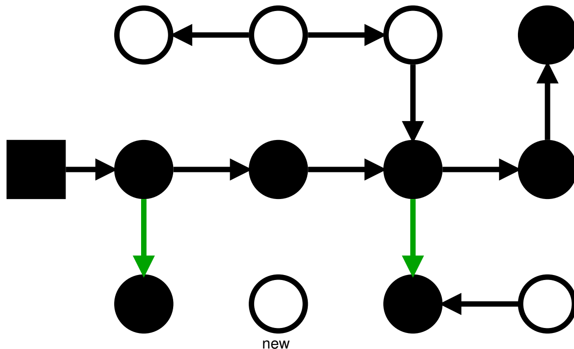
Concurrent Mark: Incremental Update



Бонус: если объект создали, но не записали, его не маркаем

redhat

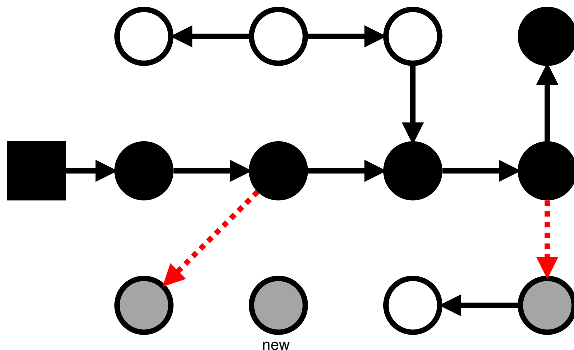
Concurrent Mark: Incremental Update



Бонус: если ссылка на объект пропала, ну и ладно!

redhat

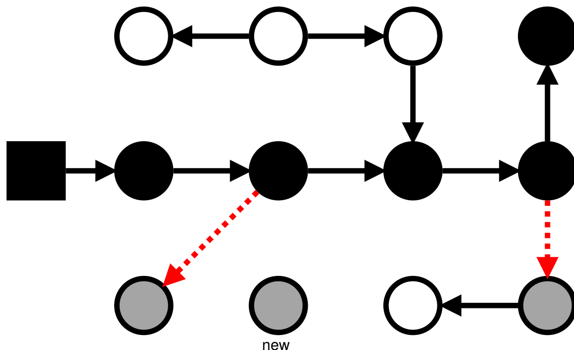
Concurrent Mark: SATB



Красим все старые ссылки в серый

redhat.

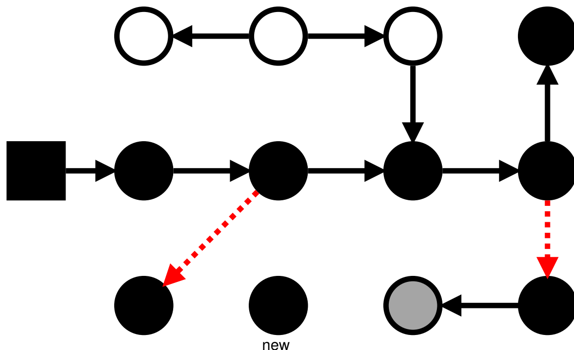
Concurrent Mark: SATB



Красим новые объекты в серый

redhat.

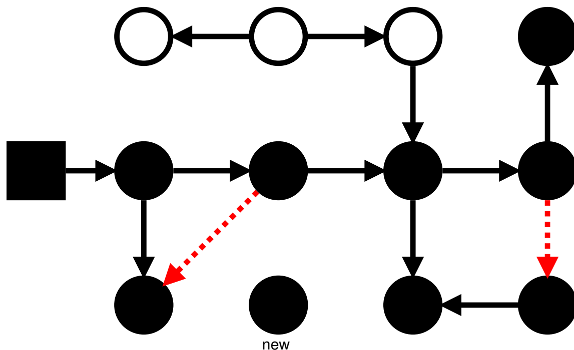
Concurrent Mark: SATB



Доделываем...

redhat.

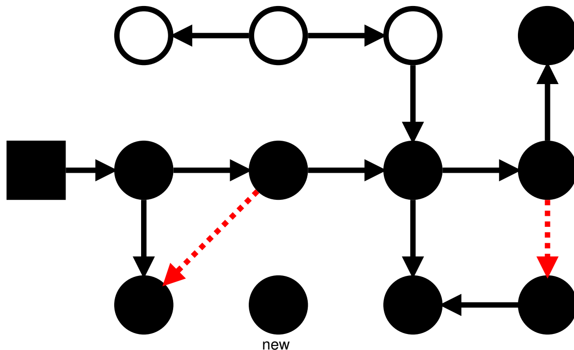
Concurrent Mark: SATB



Конец!

redhat.

Concurrent Mark: SATB



«Snapshot At The Beginning»:
пометили *все* достижимые на начало сборки

redhat.

Concurrent Mark: SATB барьер – вершки

```
# прочитали из TLS флажок
movsbl 0x378(%r15),%r10      # flag = *(TLS + 0x378)

# если он взведён, то прыгаем в барьер
test   %r10,%r10             # if (flag) ...
jne    OMG-SATB-ENABLED

# ну и пишем в объект %r12 по офсету 0x42
mov    %r11,0x42(%r12)       # *(obj + 0x42) = r11
```


redhat.

Concurrent Mark: SATB барьер – корешки

```
OMG-SATB-ENABLED:
  # прочитали старое значение
  mov     0x2c(%rbp),%r10d      # oldval = *(obj + 0x2c)

  # взяли голову тредлокальной очереди...
  mov     0x388(%r15),%r11      # qhead = *(TLS + 0x388)

  # дальше десяток инструкций с вакханалией
  # по добавлению в очередь, проверки переполнения
  # ухода в настоящий VM slowpath и т.п.
```


redhat

Concurrent Mark: отсюда две паузы

Init Mark:

1. Остановить мутатор, чтобы избежать гонок
2. Покрасить весь rootset в чёрный
3. Взвести SATB/IU-барьеры в готовность

Final Mark:

1. Остановить мутатор, чтобы избежать гонок
2. Слить остатки из SATB/IU-очередей
3. Доделать из остатков и изменений в rootset

redhat.

Concurrent Mark: отсюда две паузы

Init Mark:

1. Остановить мутатор, чтобы избежать гонок
2. Покрасить весь rootset в чёрный $\leftarrow$ самое жирное
3. Взвести SATB/IU-барьеры в готовность

Final Mark:

1. Остановить мутатор, чтобы избежать гонок
2. Слить остатки из SATB/IU-очередей
3. Доделать из остатков и изменений в rootset $\leftarrow$ самое жирное

redhat.

Concurrent Mark: стоимость барьеров

	Overhead, %
	SATB
CMP	-2.8
CPS	
CRY	
DER	
MPG	
SMK	
SER	
SFL	
XML	-1.4

redhat.

Concurrent Mark: наблюдения

1. Хорошо сделанный STW GC побьёт хорошо сделанный concurrent GC по чистой пропускной способности

   Перевод: Если вам плевать на паузы, не выдумывайте и пользуйтесь STW GC

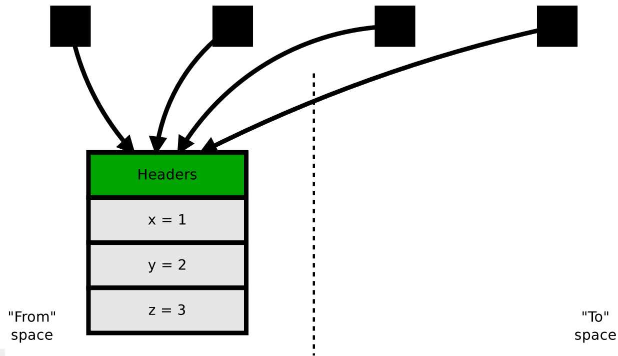
2. Разные GC по-разному будут влиять на приложение, даже если самих сборок не происходит.

   Перевод: Если gc log молчит о сборках, вы всё равно платите за барьеры

redhat

Concurrent Copy

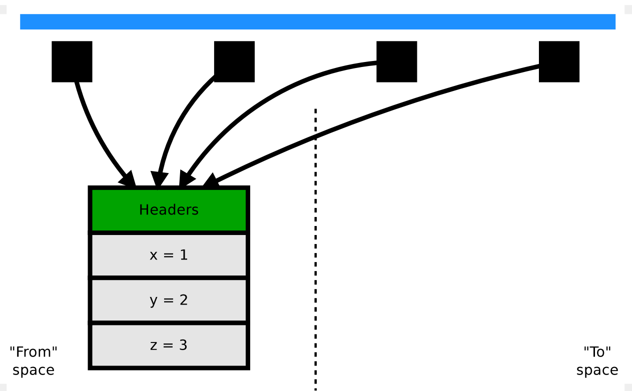
Concurrent Copy: stop-the-world



Задача:
есть объект, на него есть ссылки, надо объект переместить

redhat

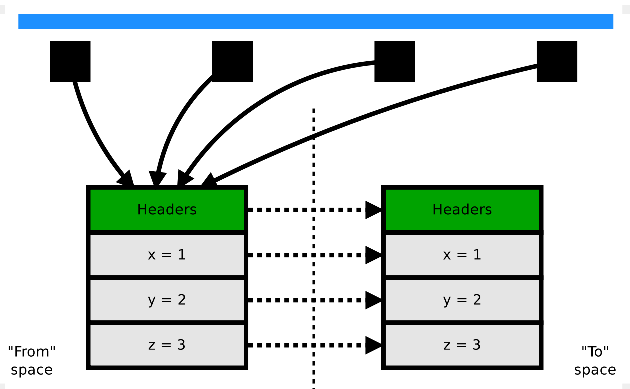
Concurrent Copy: stop-the-world



Stop The World, чтобы никто не видел, что мы под ковром делаем

redhat

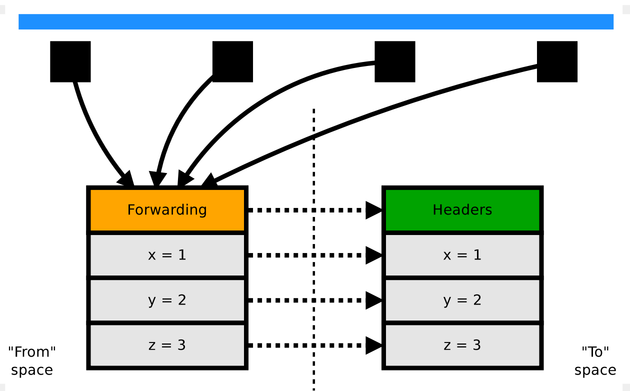
Concurrent Copy: stop-the-world



Фаза 1:
Копируем объект вместе с содержимым

redhat.

Concurrent Copy: stop-the-world

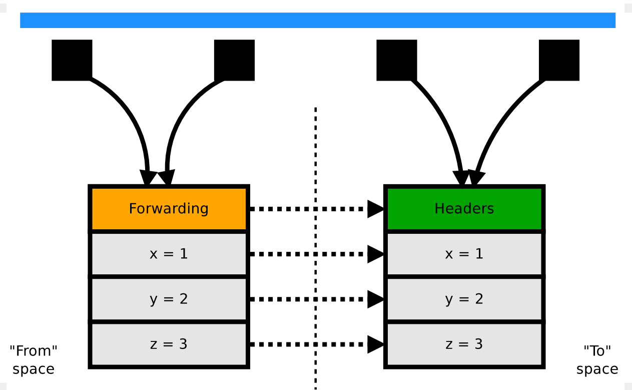


Фаза 2:

Переписываем все ссылки: сохраняем fwdptr на копию

redhat

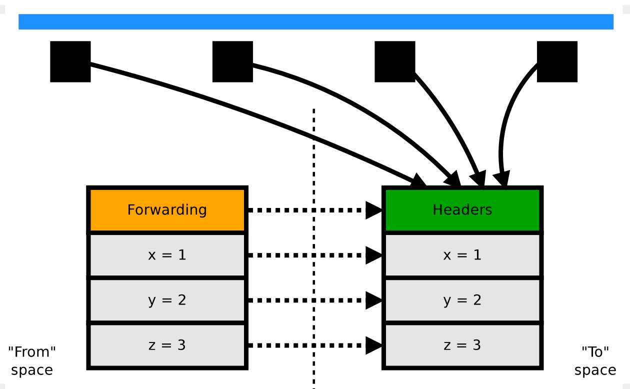
Concurrent Copy: stop-the-world



Фаза 2:
Переписываем все ссылки: если видим fwdptr, переписываем на новый адрес

redhat

Concurrent Copy: stop-the-world

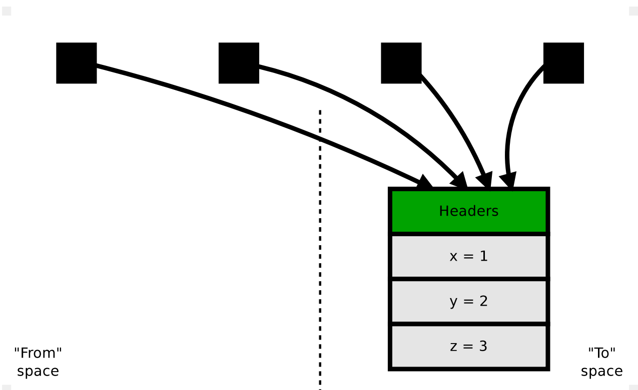


Фаза 2:

Переписываем все ссылки: если видим fwdptr, переписываем на новый адрес

redhat

Concurrent Copy: stop-the-world



На границе всё спокойно, отпускаем мутаторы на волю. Готово!

redhat.

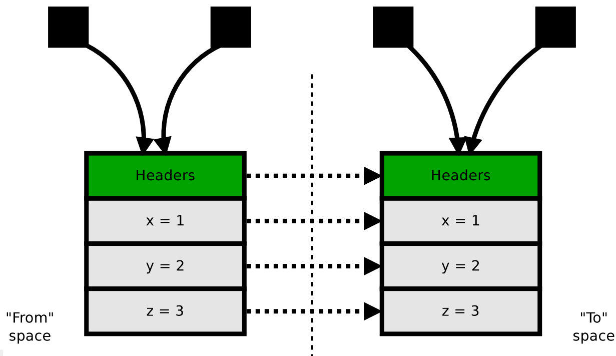
Concurrent Copy: проблемы с мутатором

http://vernova-dasha.livejournal.com/77066.html

В **concurrent** перемещении всё сложнее: есть приложение, которое читает и пишет в кучу как не в себя.

redhat

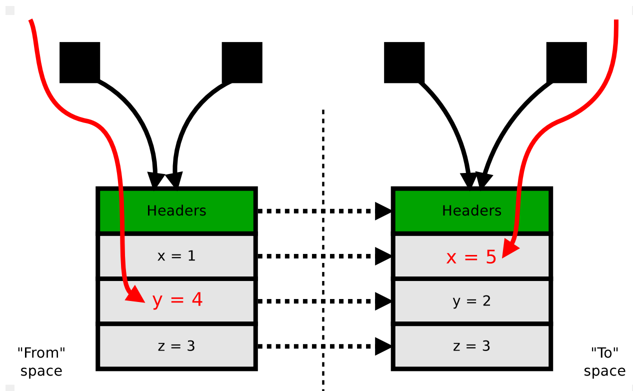
Concurrent Copy: проблемы с мутатором



При перемещении объекта будут существовать *две* копии объекта, и обе эти копии будут достижимы!

redhat

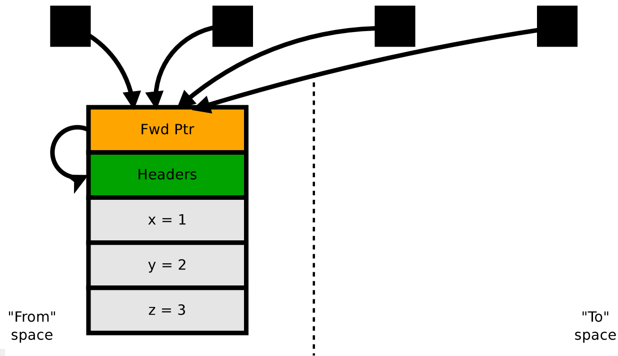
Concurrent Copy: проблемы с мутатором



Один поток записал $y = 4$ в одну копию, а другой $x = 5$ в другую. Какая копия верна?

redhat

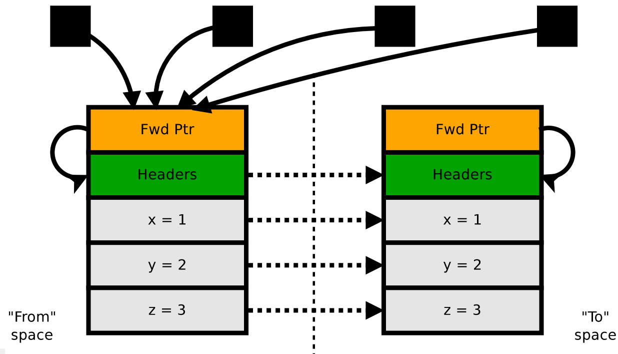
Concurrent Copy: Brooks pointers



Идея:
атомарность переписывания ссылок достигается через дополнительное перенаправление

redhat.

Concurrent Copy: Brooks pointers

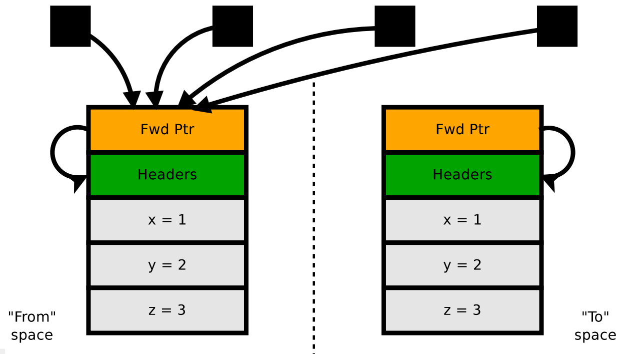


Шаг 1:

Спокойно копируем объект, инициализируем его fwdptr

redhat.

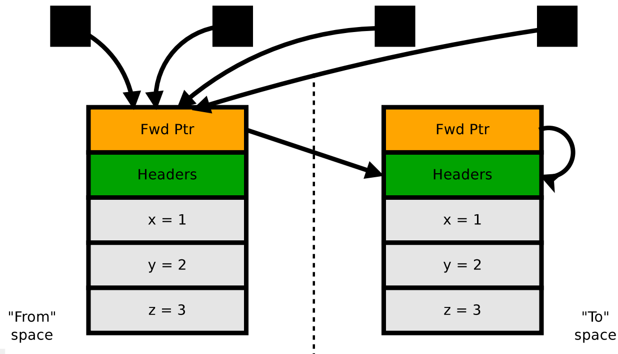
Concurrent Copy: Brooks pointers



Теперь у нас есть копия объекта, но про неё никто не знает

redhat.

Concurrent Copy: Brooks pointers

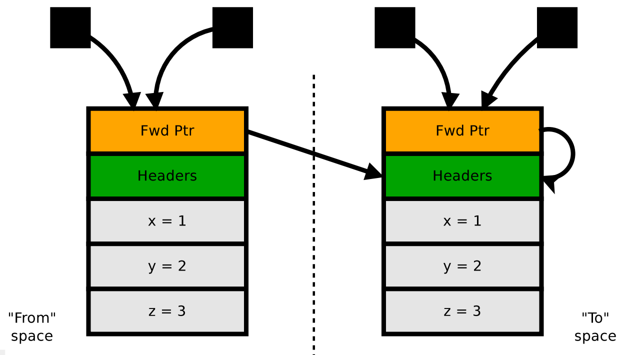


Шаг 2:

CAS! Атомарно меняем fwdptr на новую копию

redhat.

Concurrent Copy: Brooks pointers

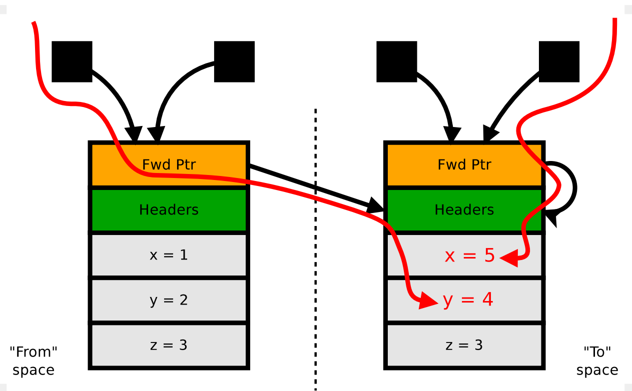


Шаг 3:

Спокойно (не атомарно) переписываем ссылки в в остальной куче

redhat

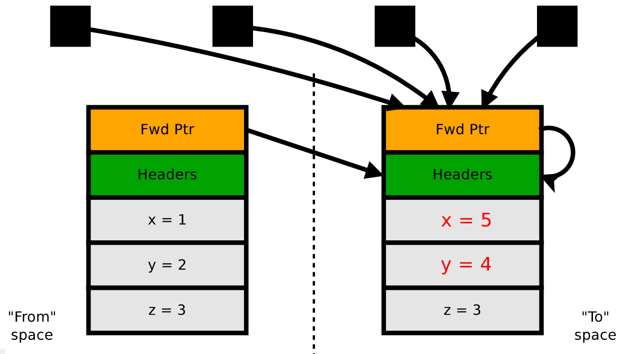
Concurrent Copy: Brooks pointers



Даже если кто-то придёт через старую ссылку, он вынужден прочитать fwdptr и узнать, где лежит новая копия

redhat

Concurrent Copy: Brooks pointers

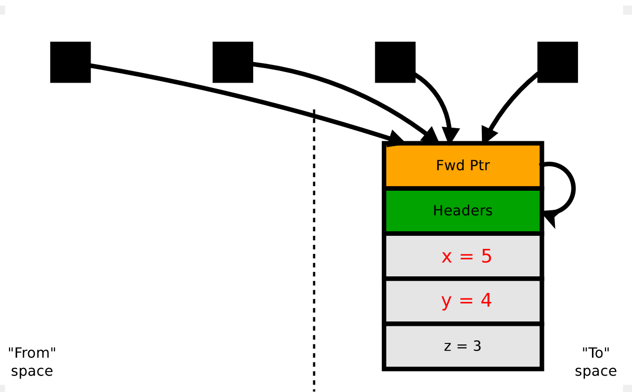


Шаг 4:

Все ссылки переписаны,
старая копия не нужна

redhat

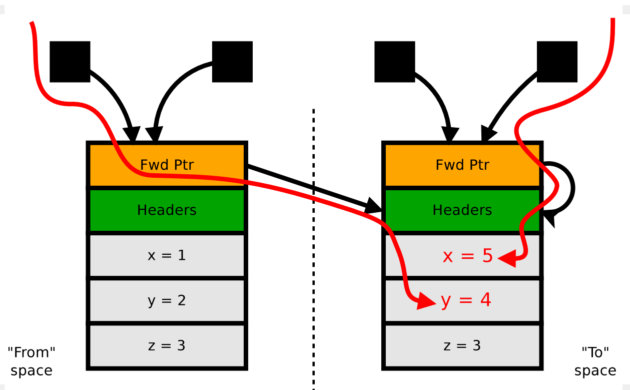
Concurrent Copy: Brooks pointers



Конец!

redhat.

Write Barriers: мотивация



To-space invariant: записи должны случаться **только** в to-space, иначе они потенциально теряются

redhat

Write Barriers: реализация – вершки

```
# прочитали из тредлокала флажок
movzbl 0x3d8(%r15),%r11d    # flag = *(TLS + 0x3d8)

# если он взведён, то прыгаем в эвакуацию
test   %r11d,%r11d          # if (flag) ...
jne    OMG-EVAC-ENABLED

# взяли to-copy
mov    -0x8(%rbp),%r10      # obj = *(obj - 8)

# ну и пишем в to-copy-объект %r10 по офсету 0x30
mov    %r10,0x30(%r10)      # *(obj + 0x30) = r10
```


redhat

Write Barriers: реализация – корешки

```
stub Write(val, obj, offset) {
  if (evac-in-progress &&
      in-collection-set(obj) &&
      fwd-ptrs-to-self(obj)) {
    // in to-space, no copy yet, can't write
    copy = copy(obj);
    obj = CAS(fwd-ptr-addr(obj), obj, copy);
  } else {
    obj = fwd-ptr(obj);
  }
  *(obj + offset) = val;
}
```


redhat

Write Barriers: сам GC

```
stub evacuate(obj) {
  if (in-collection-set(obj) &&
      fwd-ptrs-to-self(obj)) {
    // in to-space, no copy yet, can't write
    copy = copy(obj);
    CAS(fwd-ptr-addr(obj), obj, copy);
  }
}
```


redhat

Write Barriers: стоимость барьеров

	Overhead, %	
	SATB	WB
CMP	-2.8	-0.6
CPS		-1.5
CRY		
DER		-0.9
MPG		-6.9
SMK		
SER		-0.9
SFL		
XML	-1.4	

redhat.

Write Barriers: наблюдения

1. Shenandoah требует WB на **всех** записях

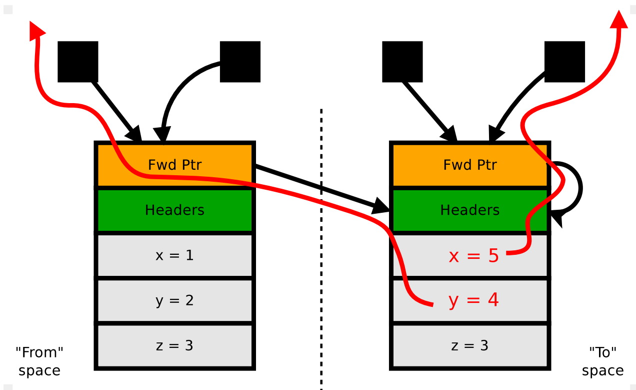
   Перевод: Теоретически, это большая проблема!

2. На практике, WB срабатывают редко: только во время эвакуации, на редких эвакуируемых объектах, в момент когда они ещё не были перемещены

   Перевод: На практике, влияют мало на большинство приложений

redhat.

Read Barriers: мотивация



Каждое чтение из кучи обязано (?) пройти через чтение forwarding pointer'а, чтобы обнаружить новую копию объекта.

redhat

Read Barriers: реализация и стоимость

```
# read barrier
mov     -0x8(%r10),%r10     # obj = *(obj - 8)

# а вот и чтение
mov     0x30(%r10),%r10d    # val = *(obj + 0x30)
```


redhat

Read Barriers: реализация и стоимость

```
# read barrier
mov     -0x8(%r10),%r10     # obj = *(obj - 8)

# а вот и чтение
mov     0x30(%r10),%r10d    # val = *(obj + 0x30)
```

Benchmark	Score				Units
	base		+3 RBs		
time	4.6	± 0.1	5.3	± 0.1	ns/op
L1-dcache-loads	12.3	± 0.2	15.1	± 0.3	#/op
cycles	18.7	± 0.3	21.6	± 0.3	#/op
instructions	26.6	± 0.2	30.3	± 0.3	#/op

redhat

Read Barriers: стоимость барьеров

	Overhead, %		
	SATB	WB	RB
CMP	-2.8	-0.6	-10.5
CPS		-1.5	-15.0
CRY			
DER		-0.9	-8.6
MPG		-6.9	-13.6
SMK			
SER		-0.9	-8.9
SFL			-13.2
XML	-1.4		-16.6

redhat.

Read Barriers: наблюдения

1. RB в принципе дешёвые, но их **очень** много

   Перевод: утяжелять RB дальше нельзя

2. Накладные расходы сильно зависят от возможностей оптимизатора по удалению и поклейке барьеров

   Перевод: работа над GC подразумевает работу над оптимизирующим компилятором

redhat

Read Barriers: JMM спешит на помощь

Можно читать из from-copy, то есть не делать RB, пока:

1. Не встретим lock, `volatile` read/write, memory barrier
2. Не встретим непрозрачный вызов

redhat.

Read Barriers: JMM спешит на помощь

Можно читать из from-copy, то есть не делать RB, пока:

1. Не встретим lock, `volatile` read/write, memory barrier
2. Не встретим непрозрачный вызов

По умолчанию можно:

1. Не переделывать RB после safepoint-а
2. Стирать RB при чтении `final`-ов

redhat.

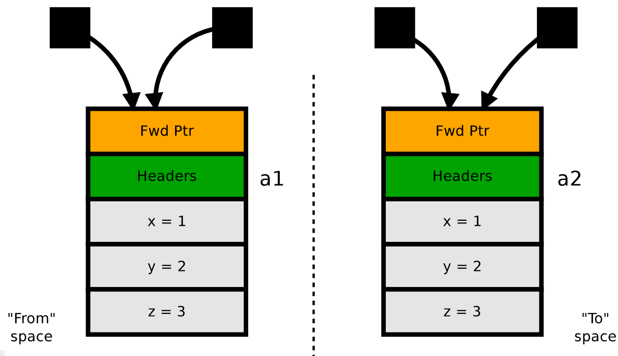
Read Barriers: пример с `final`

На полном серьёзе: `final` наконец-то улучшает производительность!

Benchmark	Score		Units
	plain	final	
time	2.7 ± 0.1	2.6 ± 0.1	ns/op
L1-dcache-loads	13.2 ± 0.1	11.2 ± 0.1	#/op
instructions	29.6 ± 0.6	28.5 ± 0.3	#/op

redhat.

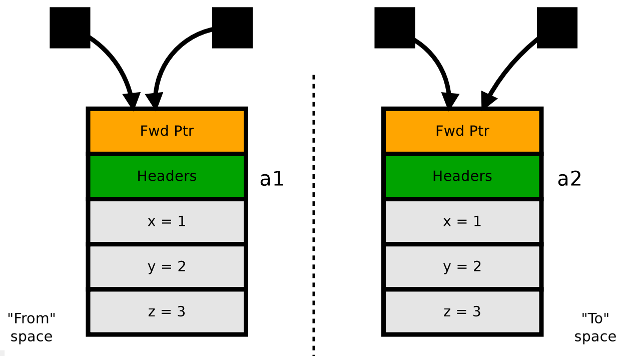
ACMP: незадача



JMM нам разрешает читать **из** from-copy!

redhat

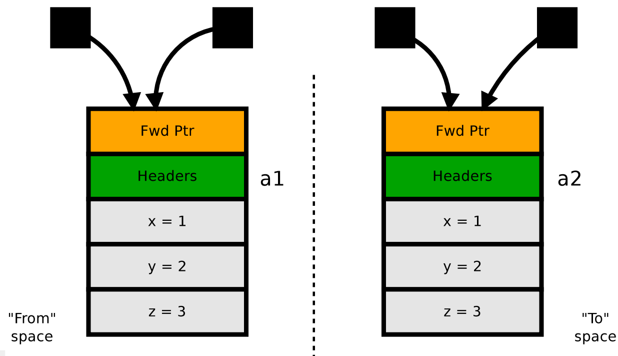
ACMP: незадача



Но что если мы сравниваем *сами* from-copy и to-copy?

$(a1 == a2) \rightarrow ???$

ACMP: незадача



Но что если мы сравниваем *сами* from-copy и to-copy?

(a1 == a2) → ???

Но *машинные указатели* не равны! Упс...

redhat

ACMP: барьер

Даже если у нас две *физические* копии одного *логического* объекта, то «==» должно сравнивать *логически*

```
# сравним указатели; если уже равны, до свидания
cmp     %rcx,%rdx           # if (a1 == a2) ...
je      EQUALS

# может быть false negative; сравним to-copy
mov     -0x8(%rcx),%rcx    # a1 = *(a1 - 8)
mov     -0x8(%rdx),%rdx    # a2 = *(a2 - 8)

# сравниваем опять:
cmp     %rcx,%rdx           # if (a1 == a2) ...
```


redhat.

ACMP: стоимость барьеров

	Overhead, %			
	SATB	WB	RB	ACMP
CMP	-2.8	-0.6	-10.5	-5.0
CPS		-1.5	-15.0	
CRY				-1.7
DER		-0.9	-8.6	
MPG		-6.9	-13.6	
SMK				
SER		-0.9	-8.9	
SFL			-13.2	
XML	-1.4		-16.6	-1.0

redhat.

ACMP: наблюдения

1. Полные сравнения «==» достаточно редки

   Перевод: acmp-барьеры особенно не тормозят штатные использования

2. Специальные виды сравнений хорошо оптимизируются

   Перевод: `a == null` не требует барьеров; предшествующие read-barriers клеятся с acmp, и т.д.

redhat.

ACMP: есть похожие проблемы?

Есть ли ещё какие-нибудь операции,
которые наступают на те же грабли:
ломаются, когда есть две физические копии объекта?

redhat.

CAS: та же проблема

```
boolean compareAndSet(Object expected, Object value);
```

Ещё хуже, чем acmp: сравнивает со значением в памяти!

На failure-пути:

1. Может мы сравниваем to-ptr в памяти с from-ptr
2. Может в памяти from-ptr, а мы ломимся с to-ptr
3. Может в памяти from-ptr, а потом туда лёг to-ptr
4. Может в памяти to-ptr, а потом туда лёг from-ptr

redhat.

CAS: стоимость барьеров

	Overhead, %				
	SATB	WB	RB	ACMP	CAS
CMP	-2.8	-0.6	-10.5	-5.0	
CPS		-1.5	-15.0		
CRY				-1.7	
DER		-0.9	-8.6		
MPG		-6.9	-13.6		
SMK					-0.2
SER		-0.9	-8.9		
SFL			-13.2		
XML	-1.4		-16.6	-1.0	-0.5

redhat.

CAS: наблюдения

1. В большинстве алгоритмов отказ в CAS достаточно редок

   Перевод: если у вас отказал CAS, то скорее всего у вас куда бОльшие проблемы с производительностью, чем барьер

2. Большая часть CAS-барьера – многократная перестраховка от ультра-редких событий

   Перевод: Она вынесена из критического пути и в норме не влияет

redhat

В целом

В целом: наблюдения

1. Барьеры стоят, и стоят много.

   Перевод: Concurrent GC будет стоить даже без сборок

2. Чем больше GC делает без пауз, тем больше оверхед

   Перевод: Паузы vs пропускная способность, привет

3. Размышления об оверхедах в отрыве от выигрышей, которые они дают, технически интересны, но упускают главное

   Перевод: Барьеры жрут циклы не просто так

redhat.

Partial

Partial: сравнивая с ParallelOld

	G1	Shenandoah		
		default	max-mark	max-copy
CMP	-15%	-20%	-73%	-76%
CPS	0%	-18%	-66%	-80%
CRY	0%	0%	-57%	-64%
DER	-7%	-18%	-66%	-76%
MPG	-1%	-19%	-64%	-74%
SMK	-2%	-4%	-55%	-67%
SER	-5%	-20%	-68%	-72%
SFL	-6%	-17%	-64%	-64%
XML	-12%	-27%	-75%	-75%

redhat.

Partial: сравнивая с ParallelOld

	G1	Shenandoah		
		default	max-mark	max-copy
CMP	-15%	-20%	-73%	-76%
CPS	0%	-18%	-66%	-80%
CRY	0%	0%	-57%	-64%
DER	-7%	-18%	-66%	-76%
MPG	-1%	-19%	-64%	-74%
SMK	-2%	-4%	-55%	-67%
SER	-5%	-20%	-68%	-72%
SFL	-6%	-17%	-64%	-64%
XML	-12%	-27%	-75%	-75%

redhat.

Partial: сравнивая с ParallelOld

	G1	Shenandoah		
		default	max-mark	max-copy
CMP	-15%	-20%	-73%	-76%
CPS	0%	-18%	-66%	-80%
CRY	0%	0%	-57%	-64%
DER	-7%	-18%	-66%	-76%
MPG	-1%	-19%	-64%	-74%
SMK	-2%	-4%	-55%	-67%
SER	-5%	-20%	-68%	-72%
SFL	-6%	-17%	-64%	-64%
XML	-12%	-27%	-75%	-75%

redhat.

Partial: сравнивая с ParallelOld

	G1	Shenandoah		
		default	max-mark	max-copy
CMP	-15%	-20%	-73%	-76%
CPS	0%	-18%	-66%	-80%
CRY	0%	0%	-57%	-64%
DER	-7%	-18%	-66%	-76%
MPG	-1%	-19%	-64%	-74%
SMK	-2%	-4%	-55%	-67%
SER	-5%	-20%	-68%	-72%
SFL	-6%	-17%	-64%	-64%
XML	-12%	-27%	-75%	-75%

redhat.

Partial: сравнивая с ParallelOld

	G1	Shenandoah		
		default	max-mark	max-copy
CMP	-15%	-20%	-73%	-76%
CPS	0%	-18%	-66%	-80%
CRY	0%	0%	-57%	-64%
DER	-7%	-18%	-66%	-76%
MPG	-1%	-19%	-64%	-74%
SMK	-2%	-4%	-55%	-67%
SER	-5%	-20%	-68%	-72%
SFL	-6%	-17%	-64%	-64%
XML	-12%	-27%	-75%	-75%

redhat.

Partial: деление кучи

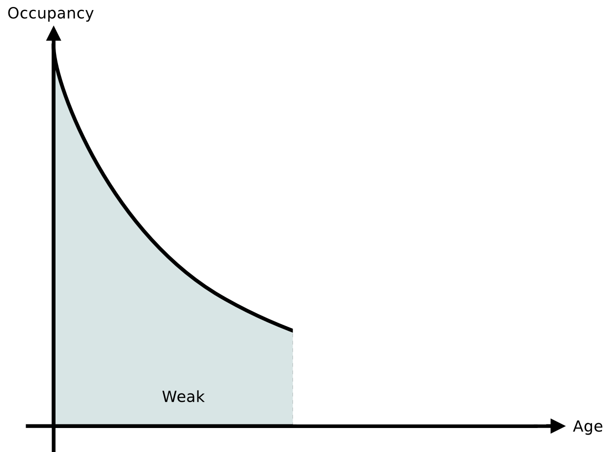
Проблема: собирать всю кучу дорого.
Для STW GC «дорого» = «большие паузы»

Идея:

1. Разделим кучу по какому-нибудь признаку: возраст, размер, класс, контекст, поток
2. Будем собирать подкучи отдельно

redhat.

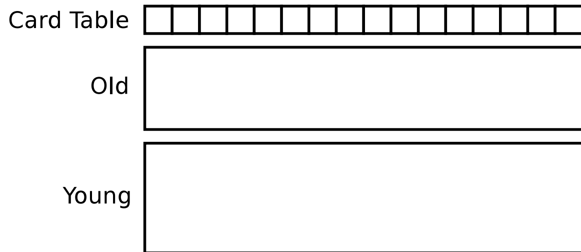
Partial: гипотезы о поколениях



Слабая гипотеза:
большинство объектов
умирают молодыми

redhat.

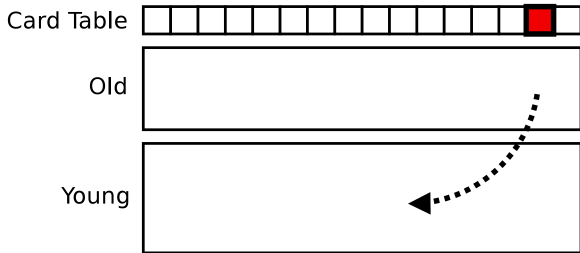
Partial: Serial/Parallel/CMS



Serial/Parallel это эксплуатируют делением кучи на *поколения*

redhat.

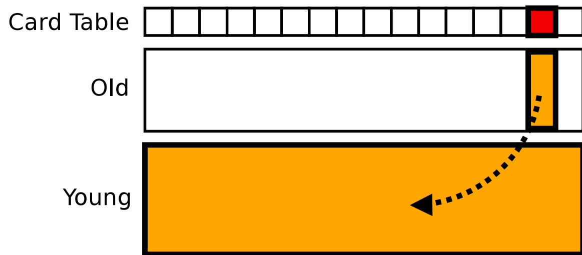
Partial: Serial/Parallel/CMS



Young можно собрать отдельно, если мы знаем все входящие ссылки из Old – для этого есть Card Table

redhat

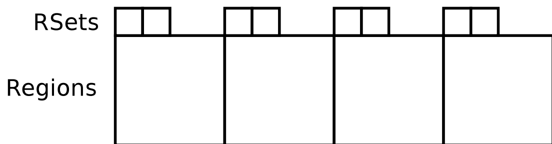
Partial: Serial/Parallel/CMS



Во время Young собираем его целиком и «грязные» части Old

redhat.

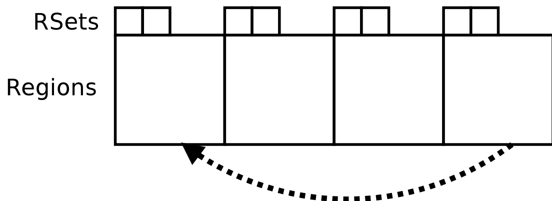
Partial: G1



G1 делает это сложнее: у него есть как Card Table, так и Remembered Sets

redhat.

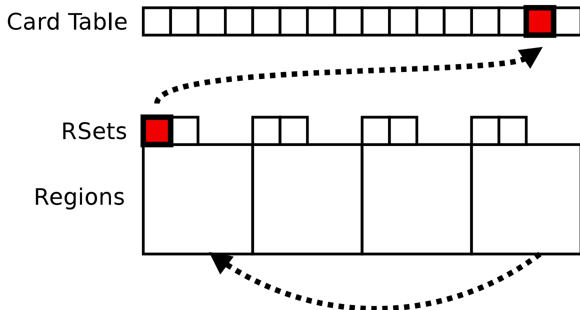
Partial: G1



При записи помечаем Card Table, но этого *не хватает*, чтобы собрать только один регион

redhat.

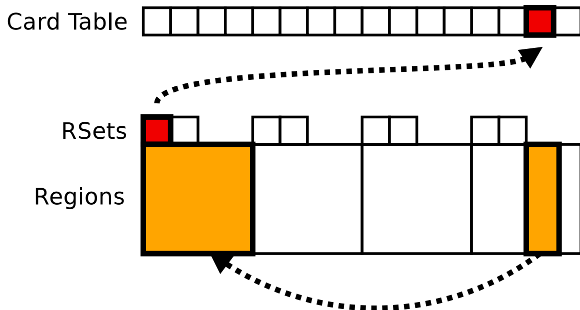
Partial: G1



По Card Table G1 асинхронно строит Remembered Sets: список грязных кусков для *каждого региона*

redhat.

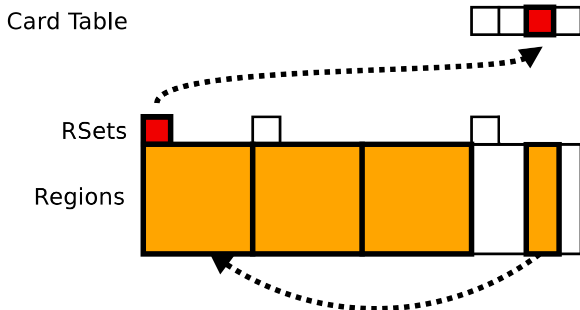
Partial: G1



Теперь можно собрать один регион: RSet нам показывает, что ещё сканировать

redhat.

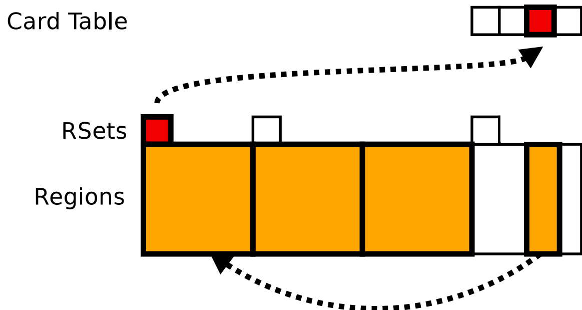
Partial: G1



Практика показала, что RSet-ы огромные, и поэтому G1 стал *generational*: часть регионов считается молодыми

redhat.

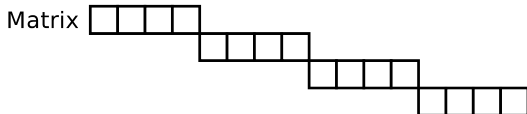
Partial: G1

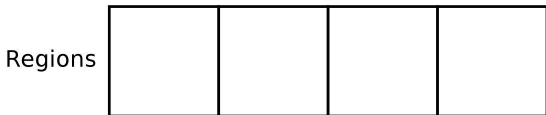


Интересные вилы: теперь young-регионы можно собрать только целиком, нельзя собрать один young-регион! Ссылки-то между young не записаны...

redhat.

Partial: Shenandoah

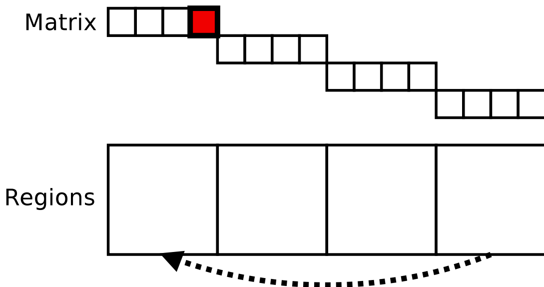




Идея: почему бы не сделать *более грубый* card table, но для каждого региона?

redhat.

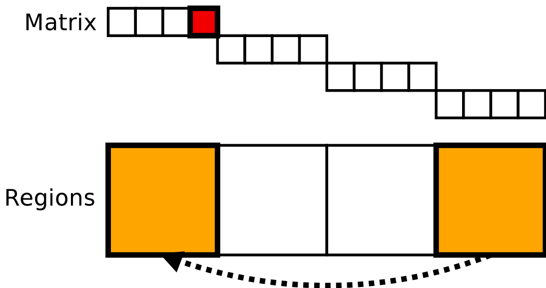
Partial: Shenandoah



Тогда мы можем поддержать *матрицу связности* регионов, и точно знать какие группы регионов мы можем собирать

redhat

Partial: Shenandoah



Собираем первый регион, и матрица говорит, что надо ещё просмотреть четвёртый. Конец.

redhat.

Partial: наблюдения

1. STW GC очень важно не собирать всю кучу

   Перевод: будут поколения, LOH, и т.п.

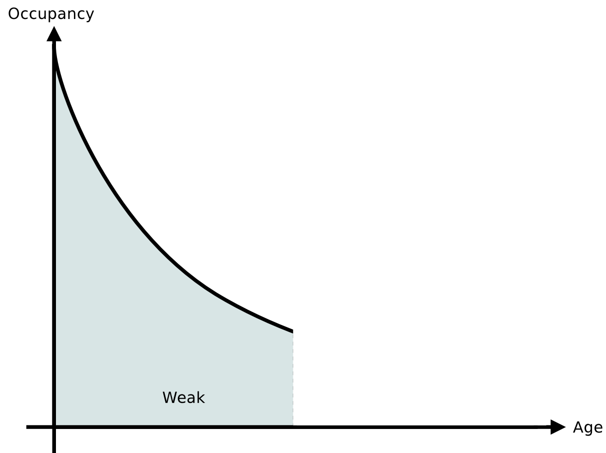
2. *Паузам* в concurrent GC всё равно, собираем всю кучу или только часть. *Пропускная способность* concurrent GC становится лучше с частичными сборками.

   Перевод: Частичные сборки в принципе не нужны, но улучшают незаметность коллектора.

redhat.

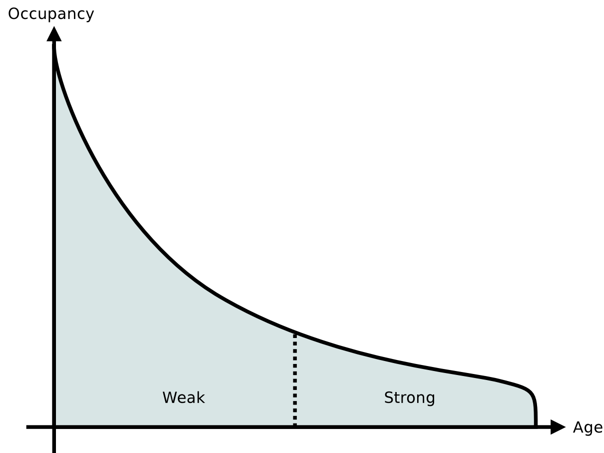
XXX

XXX: гипотезы о поколениях



redhat

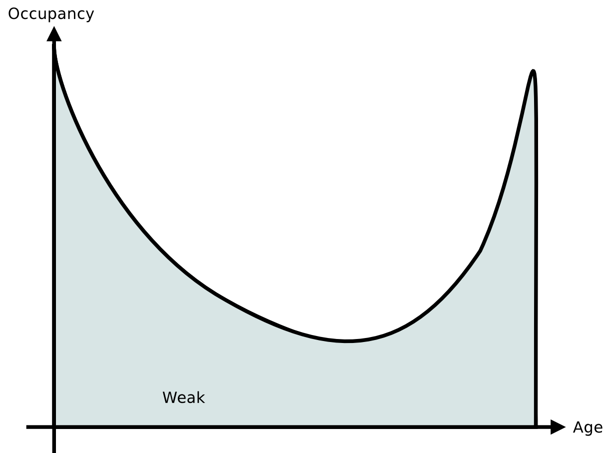
XXX: гипотезы о поколениях



Сильная гипотеза:
чем старше объект, тем дольше он проживёт.

redhat

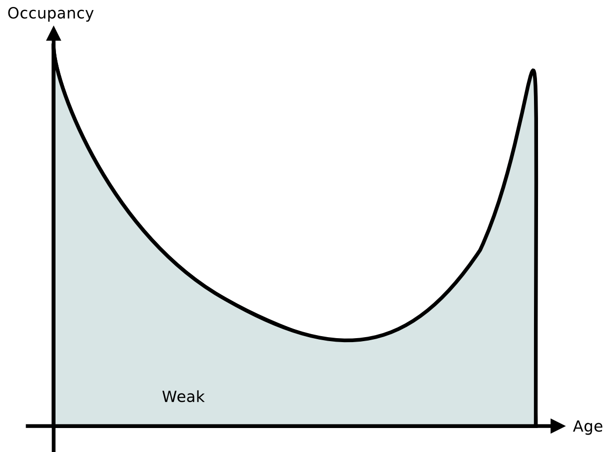
XXX: гипотезы о поколениях



Сильная гипотеза:
чем старше объект, тем
дольше он проживёт.

redhat.

XXX: гипотезы о поколениях



Сильная гипотеза:
чем старше объект, тем дольше он проживёт.

Упс, поведение кешей с политикой Least Recently Used ей прямо противоречит.

redhat.

LRU

LRU: гадкий ворклоад

Чудовищно неудобная нагрузка для *простых* generational GC
(если следуют слабой гипотезе о поколениях, и верят в сильную)

1. Прикидывается young-gc нагрузкой, особенно на чтениях
2. По мере заполнения кеша растёт Live Data Set (LDS)
3. LDS обычно измеряется десятками гигабайт – это ж кэш
4. Как кэш набухнет, начинает отпускать старые объекты
5. Как правило, удаление старых объектов фрагментирует кучу

redhat.

LRU: самый простой LRU

Самая простая реализация LRU в JDK?

redhat.

LRU: самый простой LRU

Самая простая реализация LRU в JDK?

```
cache = new LinkedHashMap<>(size*4/3, 0.75f, true) {
  @Override
  protected boolean removeEldestEntry(Map.Entry<> eldest) {
    return size() > size;
  }
};
```


redhat.

LRU: тест

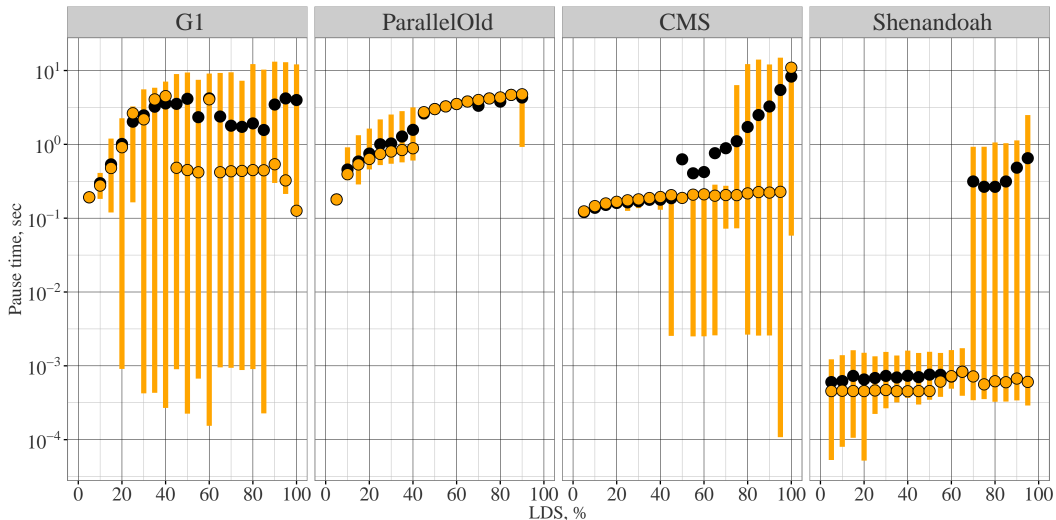
Скучная конфигурация:

1. Shenandoah JDK 9 (latest)
2. 8 потоков на i7-4790K
3. Средненькая куча: `-Xmx8g -Xms8g`
4. 90% hit rate, 90% чтений, 10% записей
5. Size (LDS) = 0..100% от `-Xmx`

Меняя размер кеша, меняем LDS $\Rightarrow$ ставим GC в неудобные положения

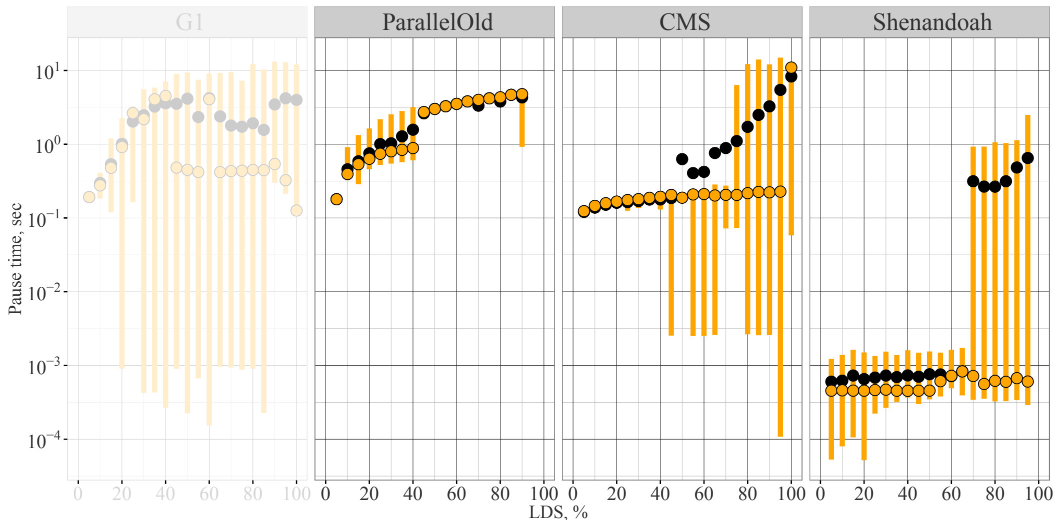
redhat

LRU: Pauses vs. LDS



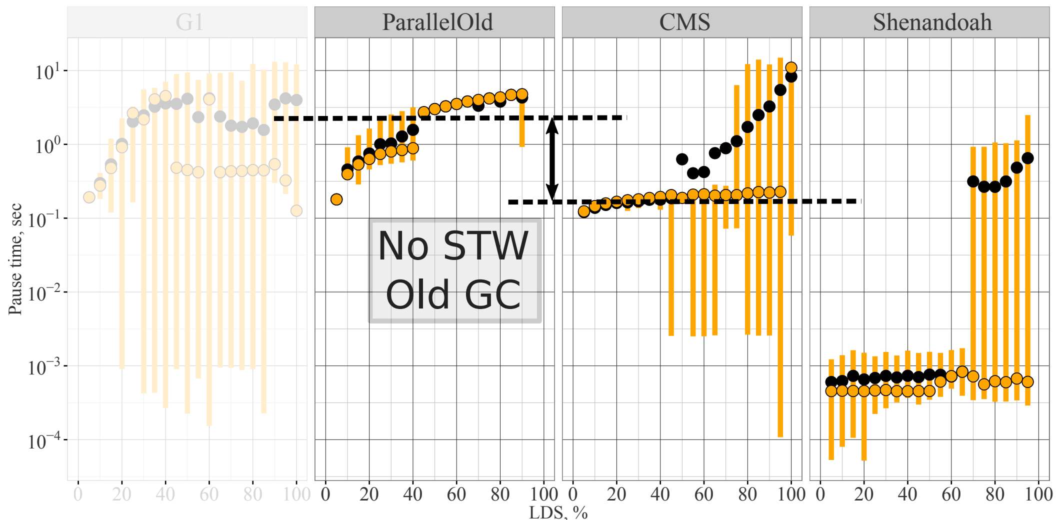
redhat.

LRU: Pauses vs. LDS



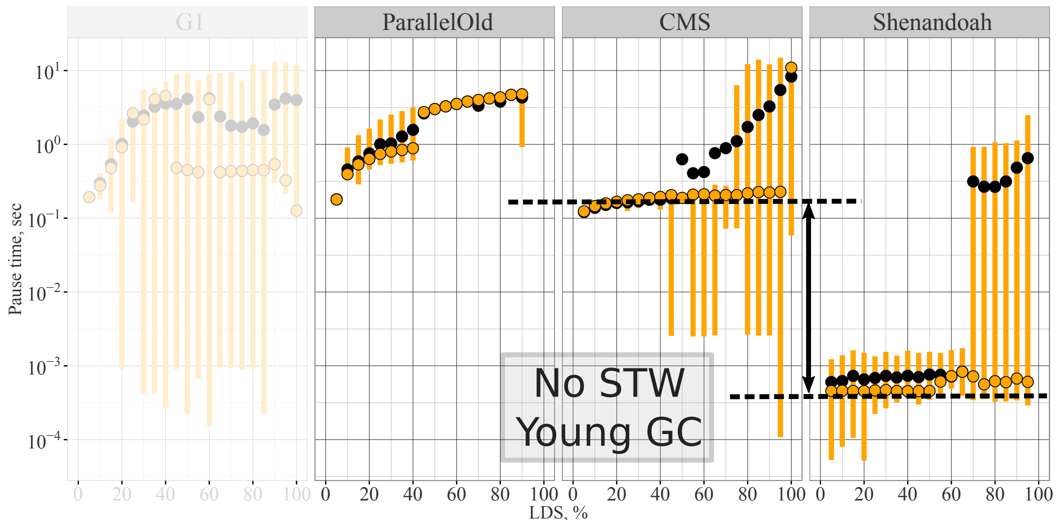
redhat.

LRU: Pauses vs. LDS



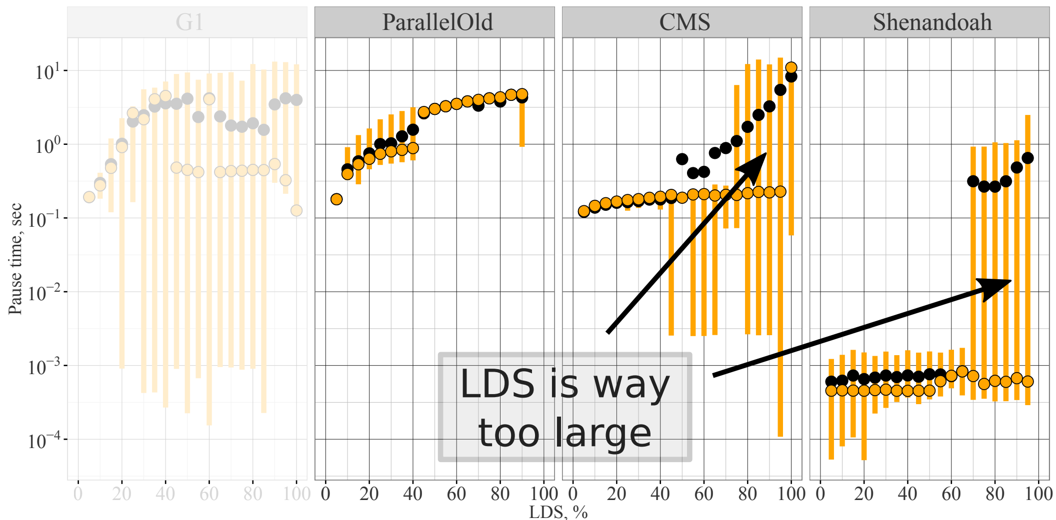
redhat

LRU: Pauses vs. LDS



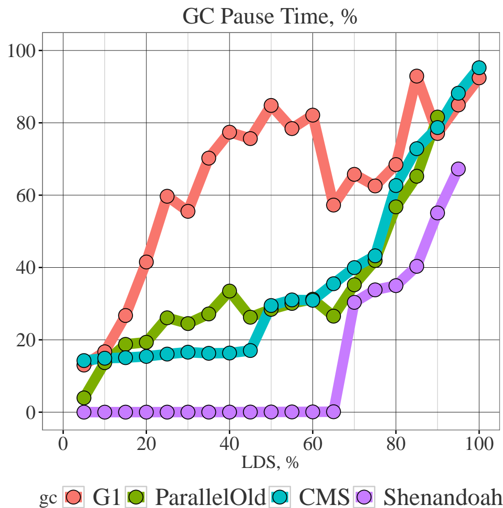
redhat

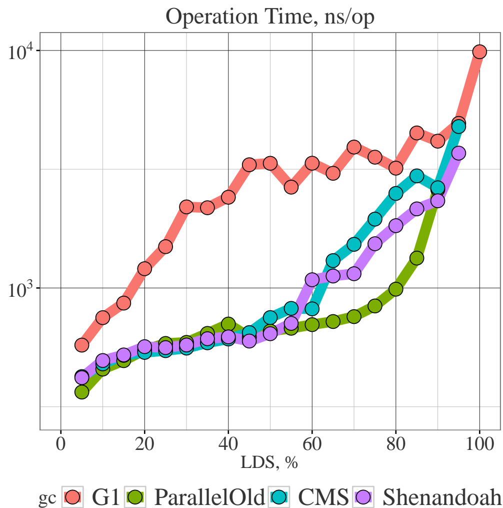
LRU: Pauses vs. LDS



redhat.

LRU: Perf vs. LDS



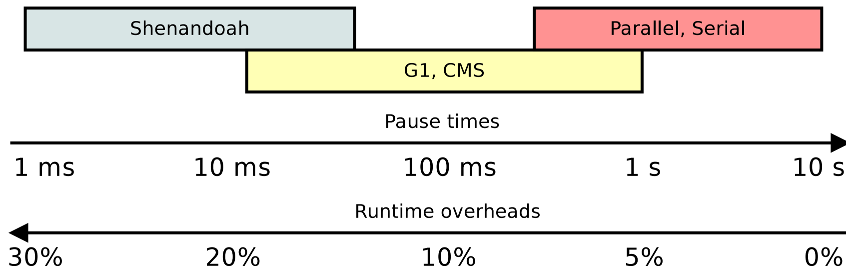


redhat

Выводы

Выводы: одной картинкой

Универсального GC не существует:
либо низкие паузы, либо низкие накладные расходы
(, либо дикие расходы по памяти)



Выбирайте под нагрузку!

redhat.

Выводы: одной строкой

1. Знание того, как работают коллекторы, позволяет выбрать нужный коллектор под нагрузку
2. Concurrent Mark – в целом решённая проблема, и избавляет от существенной части пауз. G1 и CMS уже готовы в этом помочь.
3. Concurrent Copy/Compact – проблема, которая должна быть решена для ещё более низких пауз. В этом месте появляется Shenandoah.

redhat.

Выводы: где взять

`https://wiki.openjdk.java.net/display/shenandoah/`

Несколько способов:

1. Долгий: дождаться включения в upstream OpenJDK: где-то в районе Java 10
2. Средний: дождаться включения/обновлений в IcedTea: т.е. RHEL, Fedora, Debian, Gentoo; в том числе 8u, 9
3. Короткий: взять бинарные билды или построить самим: строится и запускается как обычный OpenJDK

redhat

Шиза

Шиза: жирная JVM

-Xmx4T -Xms4T, 1T живых данных

Shenandoah:

```
Pause Init Mark 28.901ms
Concurrent marking 1942G->2069120M(4194G) 3844.822ms
Pause Final Mark 2069G->1639G(4194G) 136.814ms
Concurrent evacuation 1639G->1837G(4194G) 6961.820ms
Concurrent reset bitmaps 973.670ms
```

Шиза: жирная JVM

-Xmx4T -Xms4T, 1T живых данных

Shenandoah:

```
Pause Init Mark 28.901ms
Concurrent marking 1942G->2069120M(4194G) 3844.822ms
Pause Final Mark 2069G->1639G(4194G) 136.814ms
Concurrent evacuation 1639G->1837G(4194G) 6961.820ms
Concurrent reset bitmaps 973.670ms
```

G1:

```
Pause Young (G1 Evac Pause) 2813G->2790G(4194G) 26890.949ms
Pause Mixed (G1 Evac Pause) 2974G->2697G(4194G) 31789.592ms
```


redhat