



OTUS

# Golang Developer. Professional

[otus.ru](http://otus.ru)

• REC

Проверить, идет ли запись

# Меня хорошо видно && слышно?



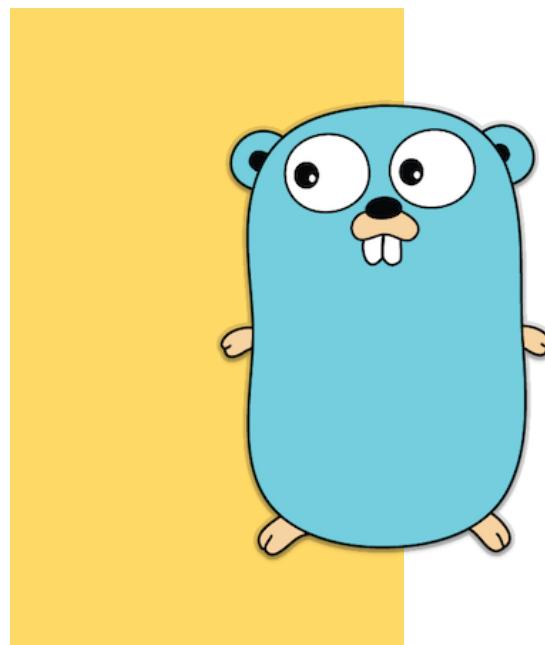
Ставим "+", если все хорошо  
"-", если есть проблемы



Тема вебинара

# Go внутри. Память и сборка мусора

Рубаха Юрий



# Правила вебинара



Активно  
участвуем



Off-topic обсуждаем  
в учебной группе



Задаем вопрос  
в чат или голосом



Вопросы вижу в чате,  
могу ответить не сразу

## Условные обозначения



Индивидуально



Время, необходимое  
на активность



Пишем в чат



Говорим голосом



Документ



Ответьте себе или  
задайте вопрос

# О чём будем говорить

- Зачем знать?
- Выделение памяти
- Сборка мусора

# Зачем?

Зачем нам знать про работу Go с памятью?

Ведь все автоматически выделяется и удаляется.

# Зачем?

- Не нужно, чтобы писать хорошие программы на Go.
- .
- .
- .

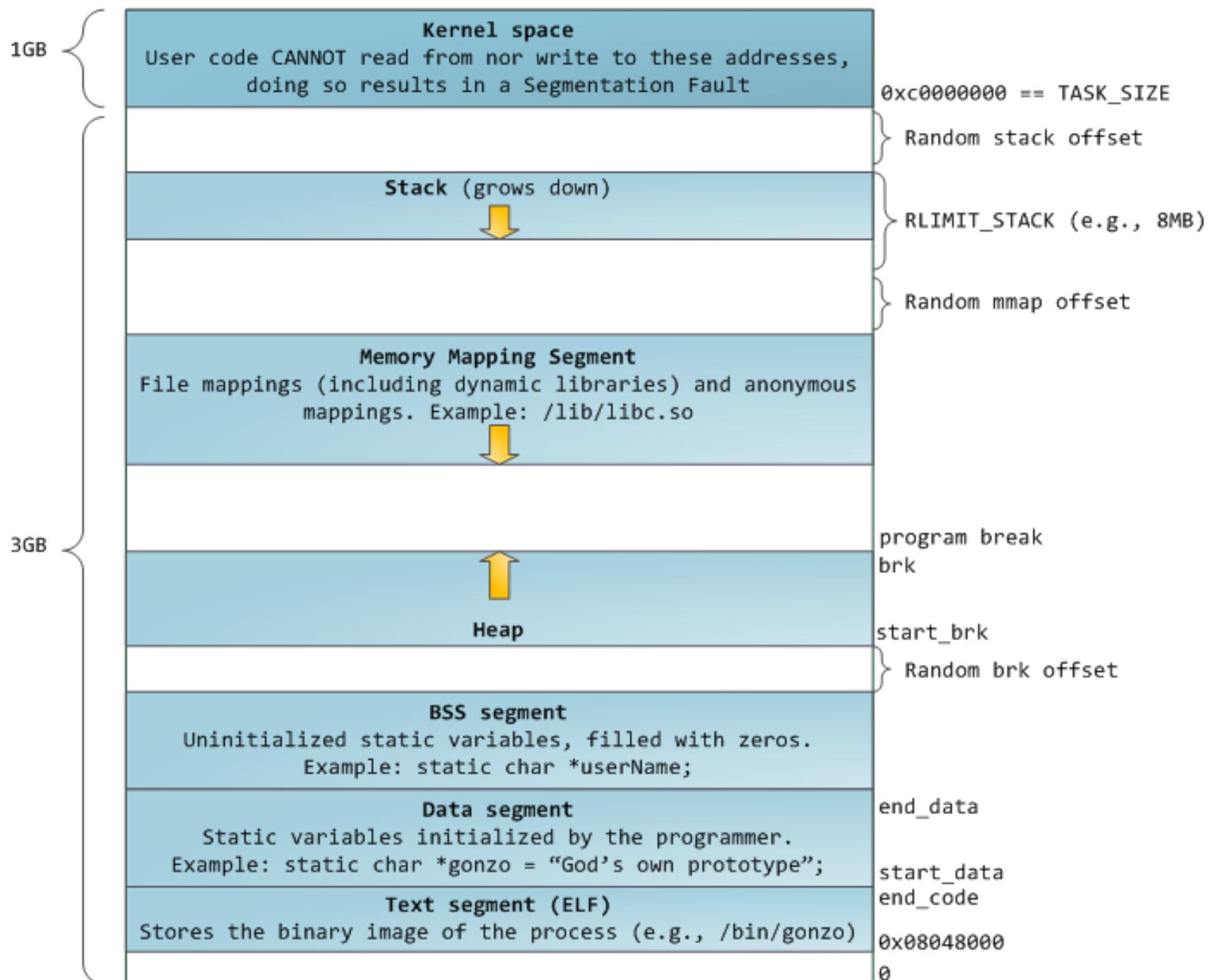
# Зачем?

- Не нужно, чтобы писать хорошие программы на Go.
- Нужно, когда есть проблемы с производительностью из-за памяти.  
(и есть пруфы!)
- .

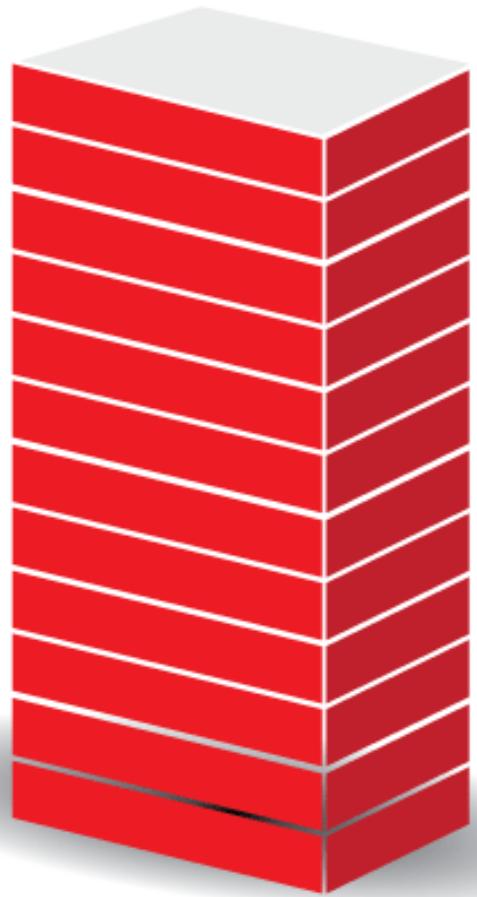
# Зачем?

- Не нужно, чтобы писать хорошие программы на Go.
- Нужно, когда есть проблемы с производительностью из-за памяти.  
(и есть пруфы!)
- Важно знать поведение, а не то, как все устроено.

# Память процесса в Linux



# Stack vs Heap



**Stack**

Ordered, on top of eachother!

No particular order!



**Heap**

# Go: Stack vs Heap

Как узнать, где Go выделит память?

.

# Go: Stack vs Heap

Как узнать, где Go выделит память?

Зачем это знать? :)

## Go: Stack

```
func main() {
    n := 4
    n2 := square(n)
    fmt.Println(n2)
}

func square(x int) int {
    return x * x
}
```

## Go: Stack

```
func main() {
    n := 4
    n2 := square(n)
    fmt.Println(n2)
}

func square(x int) int {
    return x * x
}
```

```
func main
n = 4
n2 = 0
```

---



## Go: Stack

```
func main() {
    n := 4
    n2 := square(n)
    fmt.Println(n2)
}

func square(x int) int {
    return x * x
}
```

func main

n = 4

n2 = 0

func square

x = 4



## Go: Stack

```
func main() {
    n := 4
    n2 := square(n)
    fmt.Println(n2)
}

func square(x int) int {
    return x * x
}
```

func main

n = 4

n2 = 16

---

func square

x = 4



## Go: Stack

```
func main() {
    n := 4
    n2 := square(n)
    fmt.Println(n2)
}

func square(x int) int {
    return x * x
}
```

func main

n = 4

n2 = 16

func println

a = 16



## Go: Stack pointer

```
func main() {
    n := 4
    square(&n)
    fmt.Println(n)
}

func square(x *int) {
    *x = (*x) * (*x)
}
```



## Go: Stack pointer

```
func main() {
    n := 4
    square(&n)
    fmt.Println(n)
}

func square(x *int) {
    *x = (*x) * (*x)
}
```

```
func main
```

```
n = 4
```



## Go: Stack pointer

```
func main() {
    n := 4
    square(&n)
    fmt.Println(n)
}

func square(x *int) {
    *x = (*x) * (*x)
}
```

func main

→n = 4

func square

—x = 0xc000078dab5



## Go: Stack pointer

```
func main() {
    n := 4
    square(&n)
    fmt.Println(n)
}

func square(x *int) {
    *x = (*x) * (*x)
}
```

func main

→n = 16

func square

—x = 0xc000078dab5



## Go: Stack pointer

```
func main() {
    n := 4
    square(&n)
    fmt.Println(n)
}

func square(x *int) {
    *x = (*x) * (*x)
}
```

func main

→n = 16

func square

x = 0xc000078dab5



## Go: Stack pointer

```
func main() {
    n := 4
    square(&n)
    fmt.Println(n)
}

func square(x *int) {
    *x = (*x) * (*x)
}
```

func main

n = 16

func println

a = 16



## Go: Heap

```
func main() {
    n := answer()
    fmt.Println(*n)
}

func answer() *int {
    x := 42
    return &x
}
```



## Go: Heap

```
func main() {
    n := answer()
    fmt.Println(*n)
}

func answer() *int {
    x := 42
    return &x
}
```

```
func main
```

```
n = nil
```



## Go: Heap

```
func main() {
    n := answer()
    fmt.Println(*n)
}

func answer() *int {
    x := 42
    return &x
}
```

func main

n = nil

func answer

x = 42



## Go: Heap

```
func main() {
    n := answer()
    fmt.Println(*n)
}

func answer() *int {
    x := 42
    return &x
}
```

func main

n = 0xc000078da

func answer

→ x = 42



## Go: Heap

```
func main() {
    n := answer()
    fmt.Println(*n)
}

func answer() *int {
    x := 42
    return &x
}
```

```
func main
n = 0xc000078da
```

```
func println
→a = ?
```



## Go: Heap

```
func main() {
    n := answer()
    fmt.Println(*n)
}

func answer() *int {
    x := 42
    return &x
}
```

func main

n = nil

func answer

x = 42



## Go: Heap

```
func main() {
    n := answer()
    fmt.Println(*n)
}

func answer() *int {
    x := 42
    return &x
}
```

func main

n = nil

func answer

x = 42

0xc000078da = 42 ←



## Go: Heap

```
func main() {
    n := answer()
    fmt.Println(*n)
}

func answer() *int {
    x := 42
    return &x
}
```

func main  
n = 0xc000078da

func answer  
x = 42

→ 0xc000078da = 42



## Go: Heap

```
func main() {
    n := answer()
    fmt.Println(*n)
}

func answer() *int {
    x := 42
    return &x
}
```

func main  
n = 0xc000078da

func println  
a = 42

→ 0xc000078da = 42



# Go: Stack vs Heap

Как узнать, где Go выделит память?

# Go: Stack vs Heap

Никак.

Только компилятор знает, где будет выделена память.

## Go: escape analysis

```
go build -gcflags="-m"
```

```
...
./full.go:12:2: moved to heap: x
...
./full.go:7:14: *n escapes to heap
...
```

# Go: Stack vs Heap

Зачем знать, где Go выделит память?

## Простое правило

**Уменьшайте количество ненужных аллокаций.**

# Go: Stack vs Heap (вопрос)

Почему `io.Reader` имеет такой интерфейс?

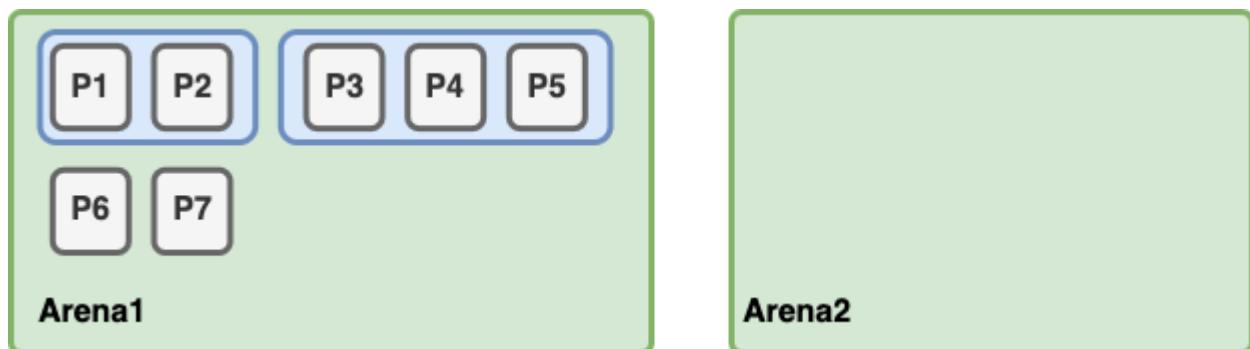
```
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

А не такой?

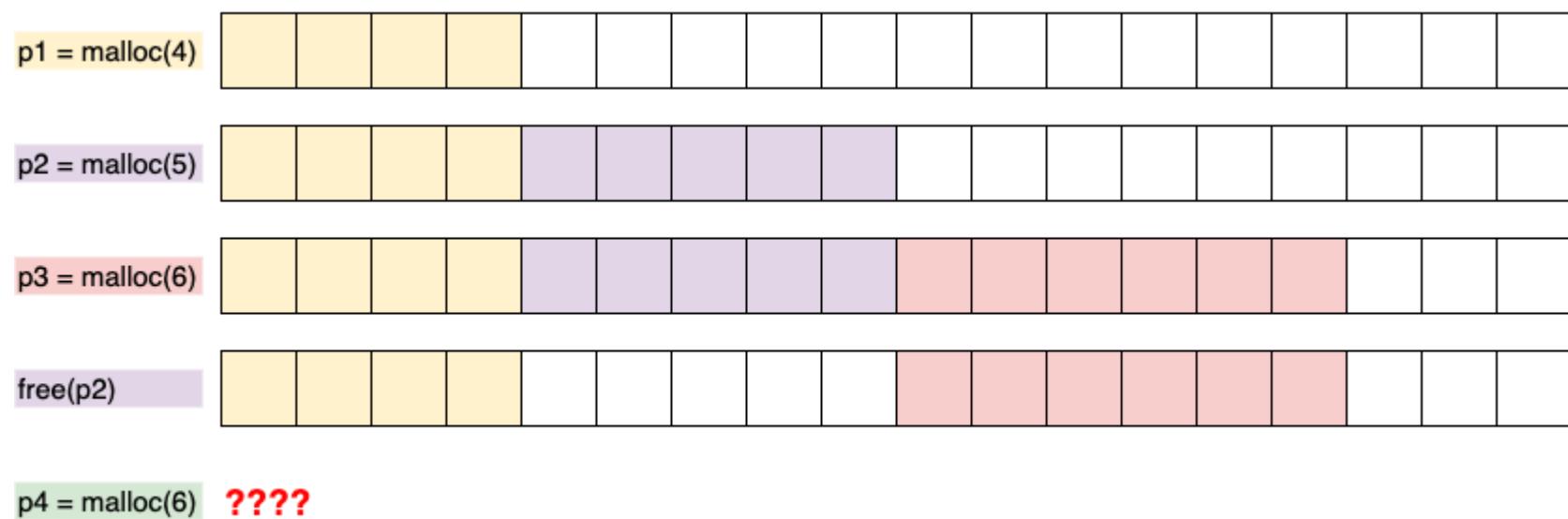
```
type Reader interface {
    Read(n int) (p []byte, err error)
}
```

# Выделение памяти: основные понятия

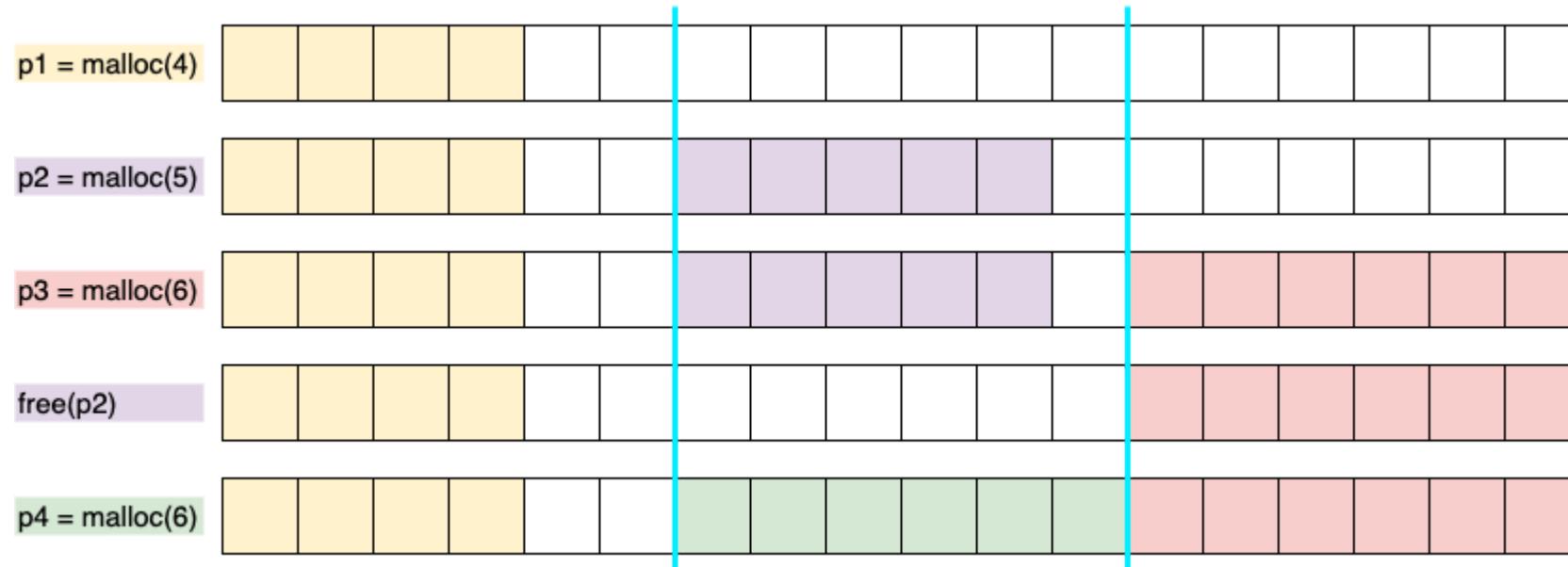
- Arena - большой кусок памяти (64 МБ)
- Page - кусочек Арены (8 КБ)
- Span - несколько страничек подряд



# Выделение памяти: фрагментация



# Выделение памяти: фрагментация (спаны)



# Выделение памяти: классы Span'ов

Конкретный Span хранит:

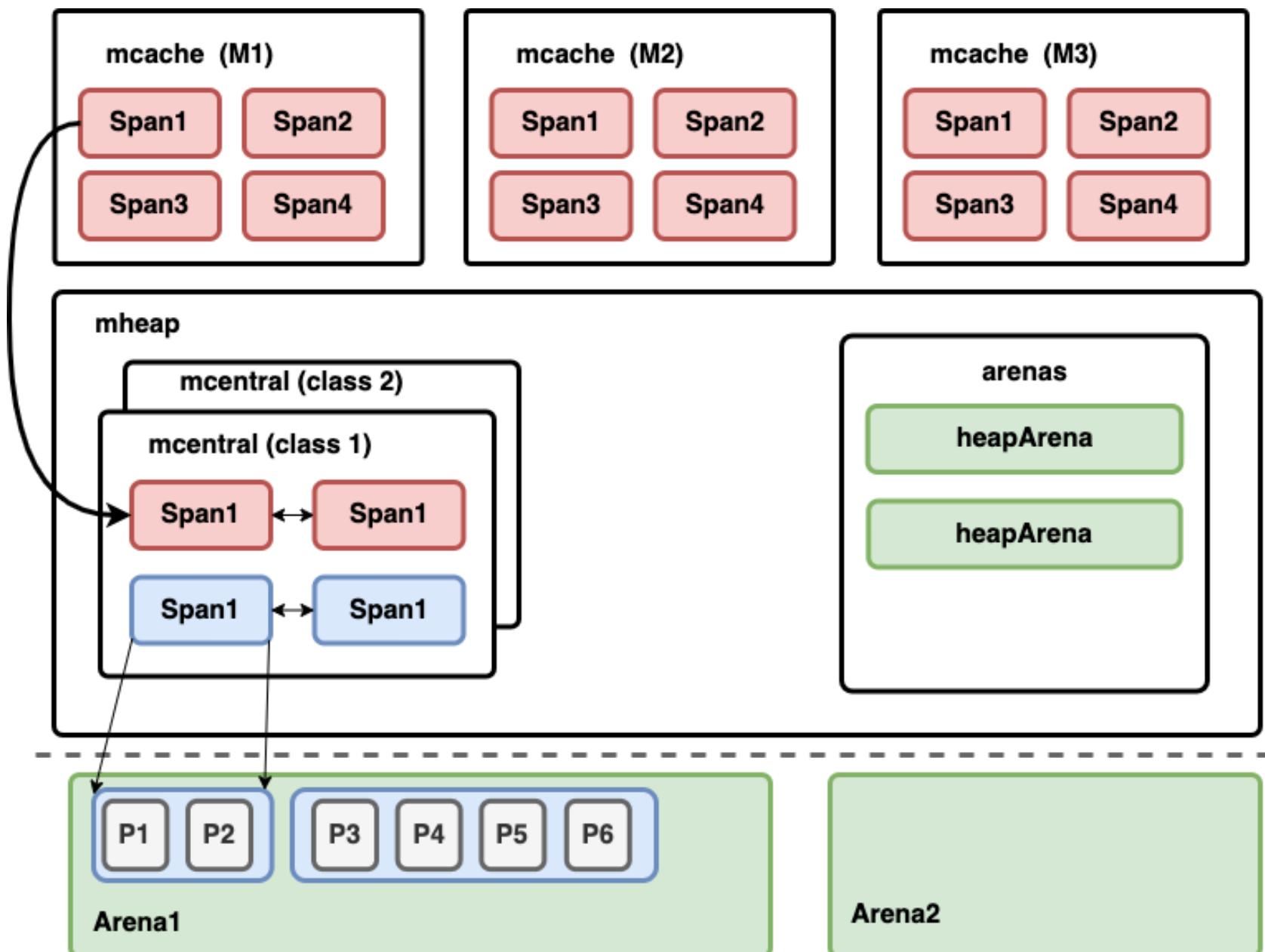
- объекты одного размера (классы)
- объекты либо с указателями, либо без (нужно для gc)

```
// runtime/sizeclasses.go

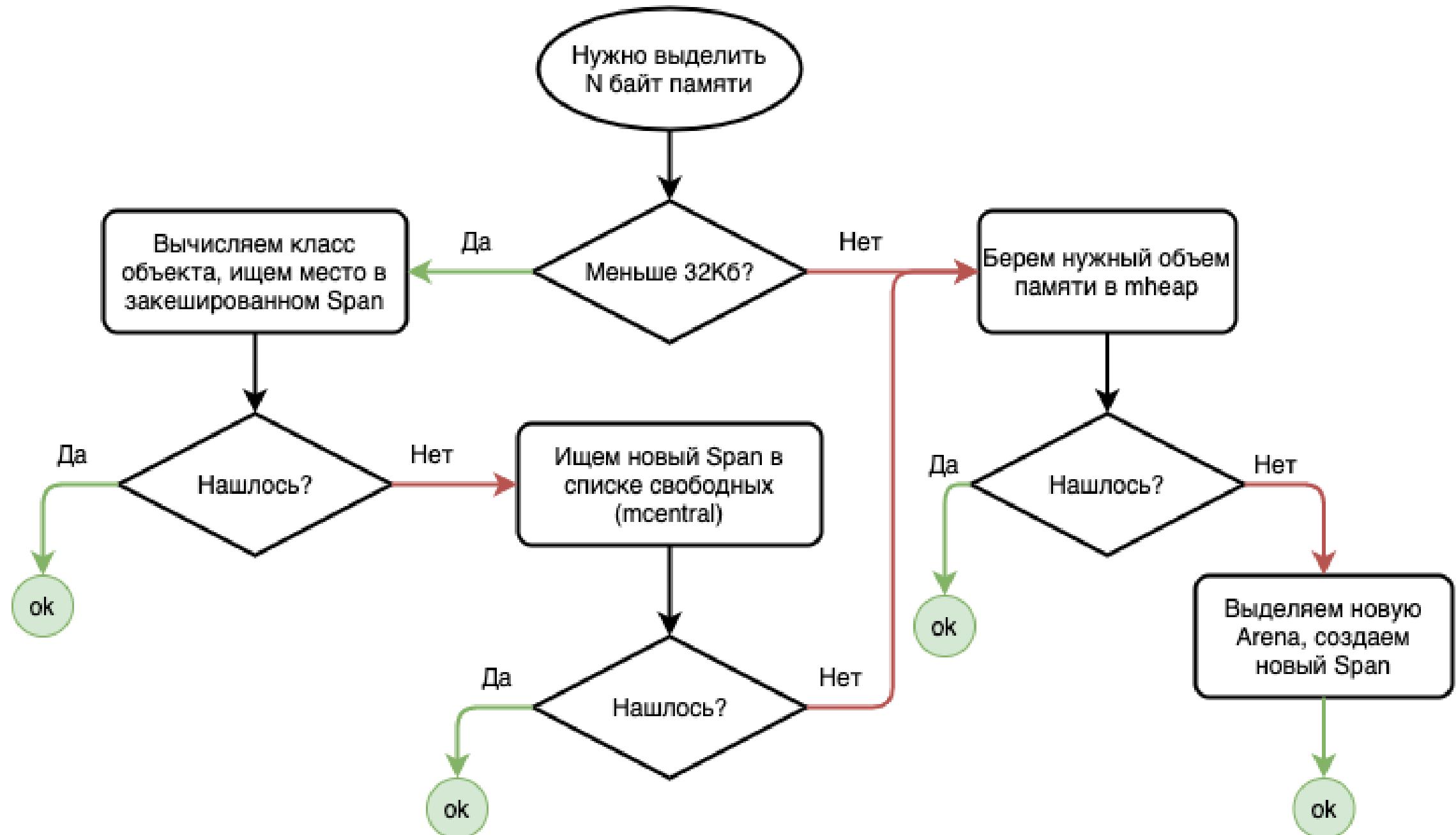
// class bytes/obj bytes/span objects tail waste max waste
// 1 8 8192 1024 0 87.50%
// 2 16 8192 512 0 43.75%
// 3 32 8192 256 0 46.88%
...
// 33 1152 8192 7 128 12.41%
// 34 1280 8192 6 512 15.55%
// 35 1408 16384 11 896 14.00%
...
// 65 27264 81920 3 128 10.00%
// 66 28672 57344 2 0 4.91%
// 67 32768 32768 1 0 12.50%
```



# Выделение памяти: общая картина



# Алгоритм выделения памяти



<https://golang.org/src/runtime/malloc.go#L905>

# Сборка мусора: mark and sweep

- **Mark:** находим и отмечаем все достижимые объекты из набора (например, в куче).
- **Sweep:** проходим по всем объектам в куче, затираем недостижимые и возвращаем их в пул свободной памяти.



# Сборка мусора: Three-color Marker Algorithm

1. Покрасить все корневые объекты (стек и глобальные переменные) в серый.
2. Выбрать серый объект из набора серых объектов и пометить его как чёрный.
3. Все объекты, на которые указывает чёрный объект, пометить серым. Это гарантирует, что сам объект и объекты, на которые он ссылается, не будут выброшены в мусор.
4. Если в графе остались серые объекты, вернуться к шагу 2.



# Сборка мусора: алгоритм

- Остановить ненадолго выполнение (STW - Stop The World)
- Разметить объекты в куче (алгоритм tricolor)
- Еще раз остановить все (STW - Stop The World)
- Освобождать память по ходу работы



## Сборка мусора: STW

- Останавливает все горутины.
- .
- .

# Сборка мусора: STW

- Останавливает все горутины.
- Обычно отрабатывает очень быстро (~100 мкс).
- .

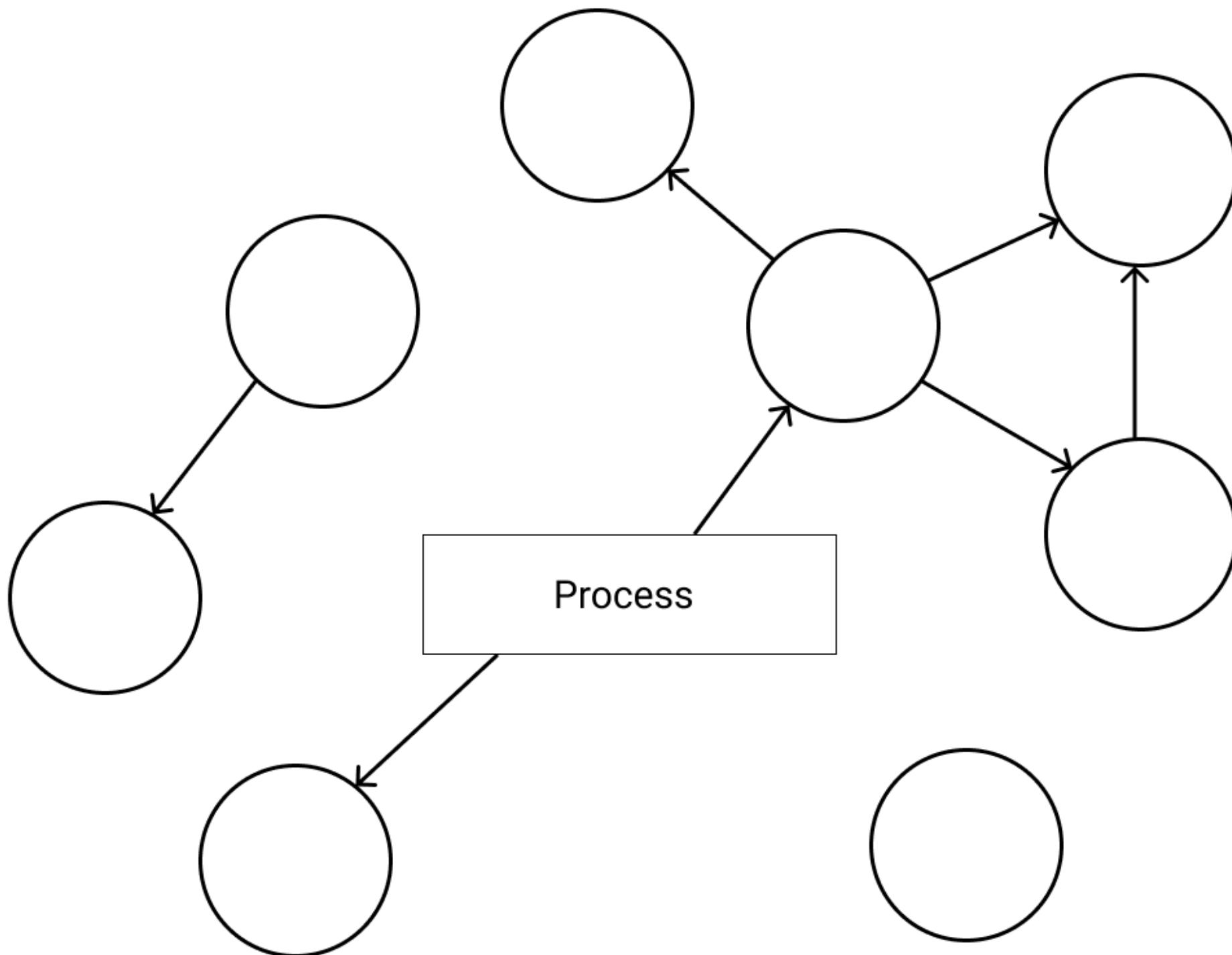
## Сборка мусора: STW

- Останавливает всеgorутины.
- Обычно отрабатывает очень быстро (~100 мкс).
- Не ждет горутины в syscalls.

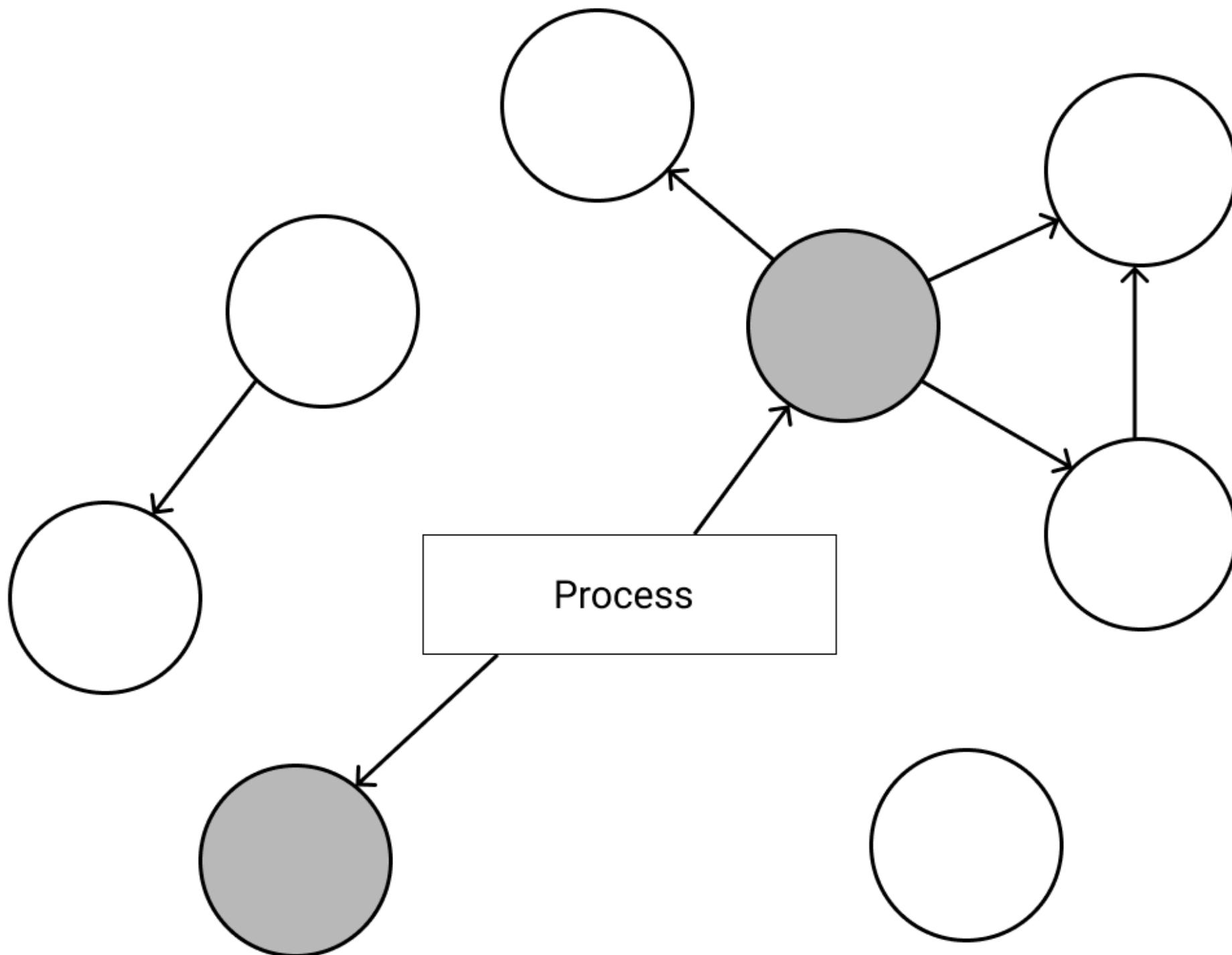
## Сборка мусора: STW

- Переключает рантайм в режим Marking.
- Включает Write Barrier.

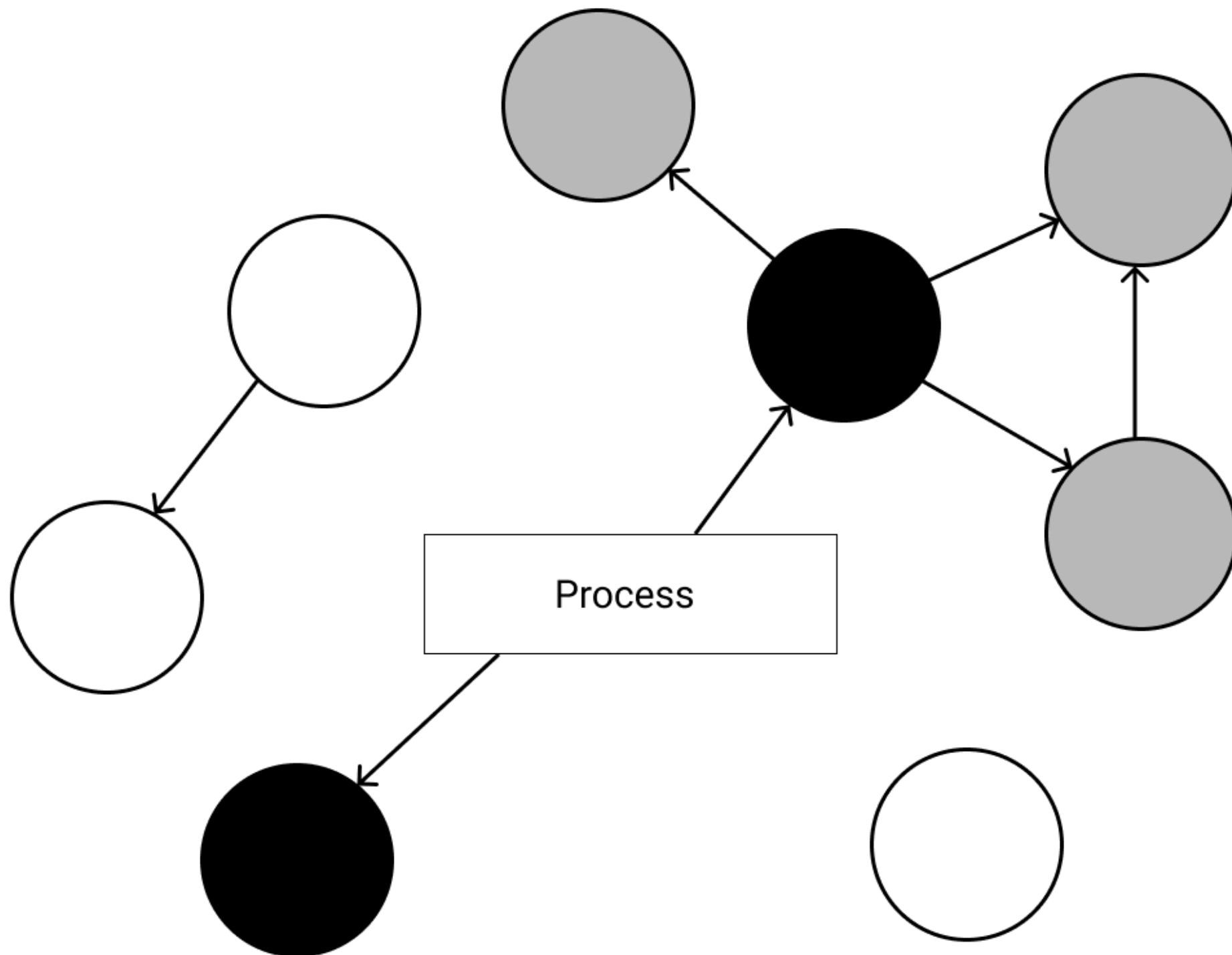
## Сборка мусора: разметка объектов



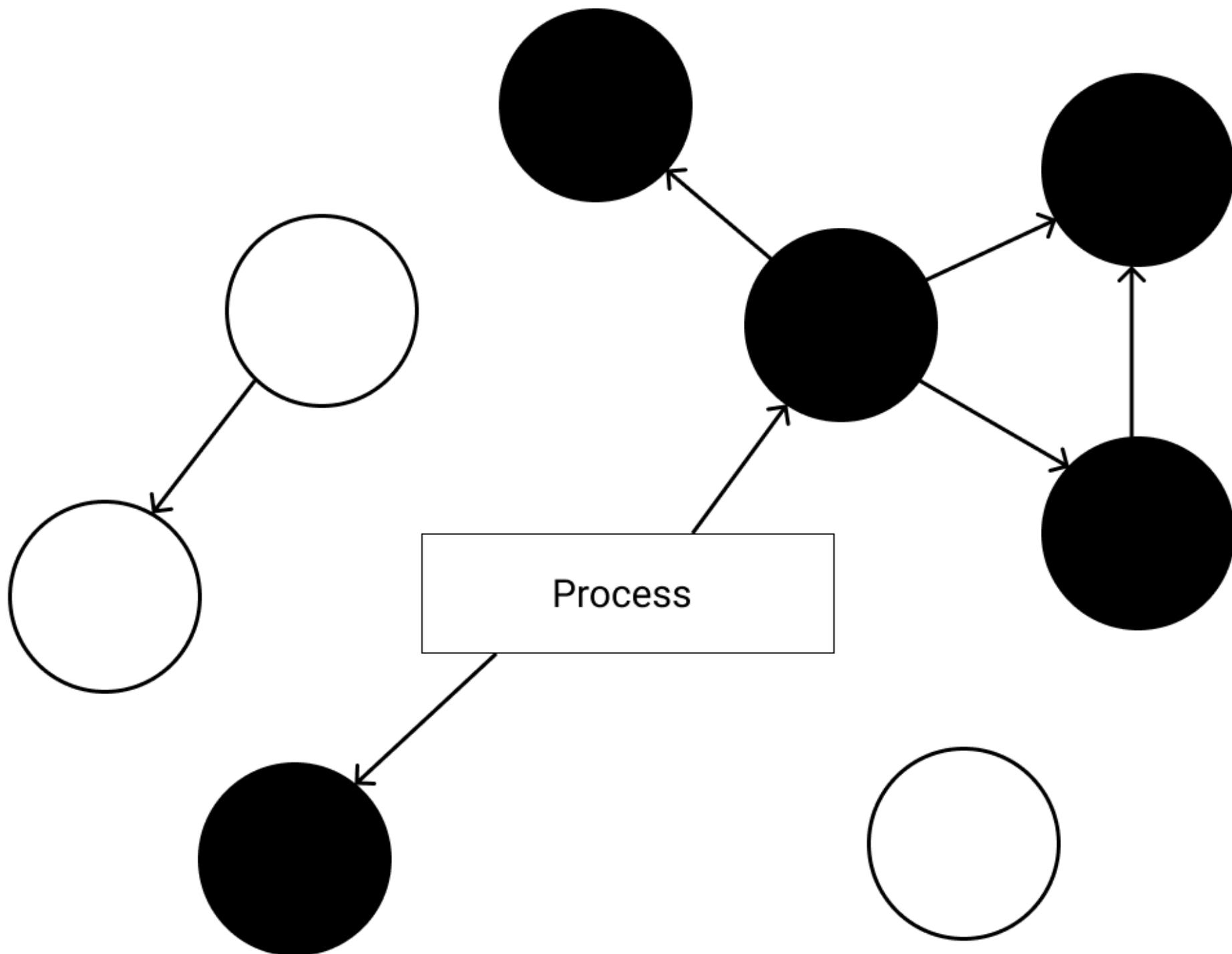
## Сборка мусора: разметка объектов



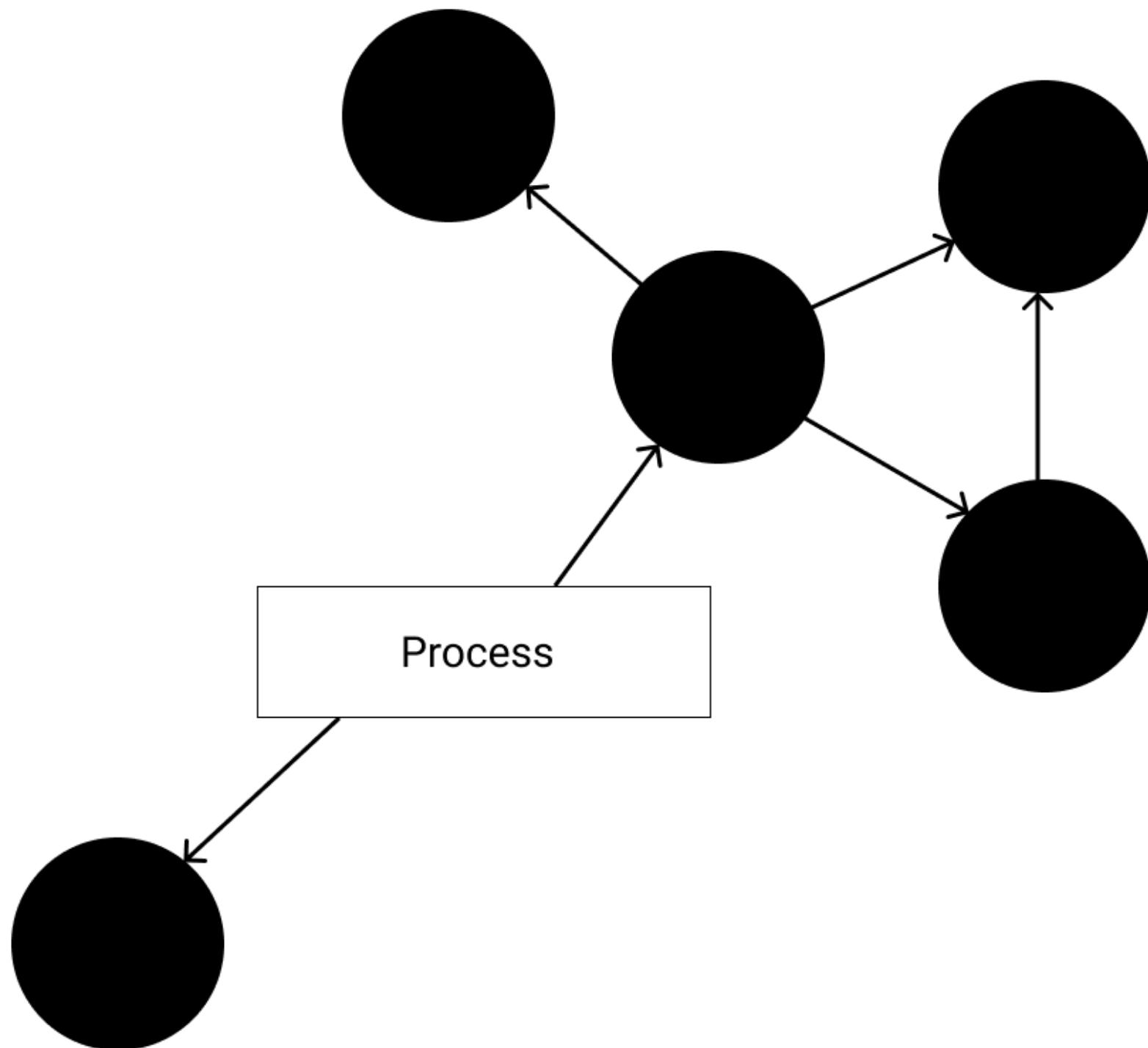
## Сборка мусора: разметка объектов



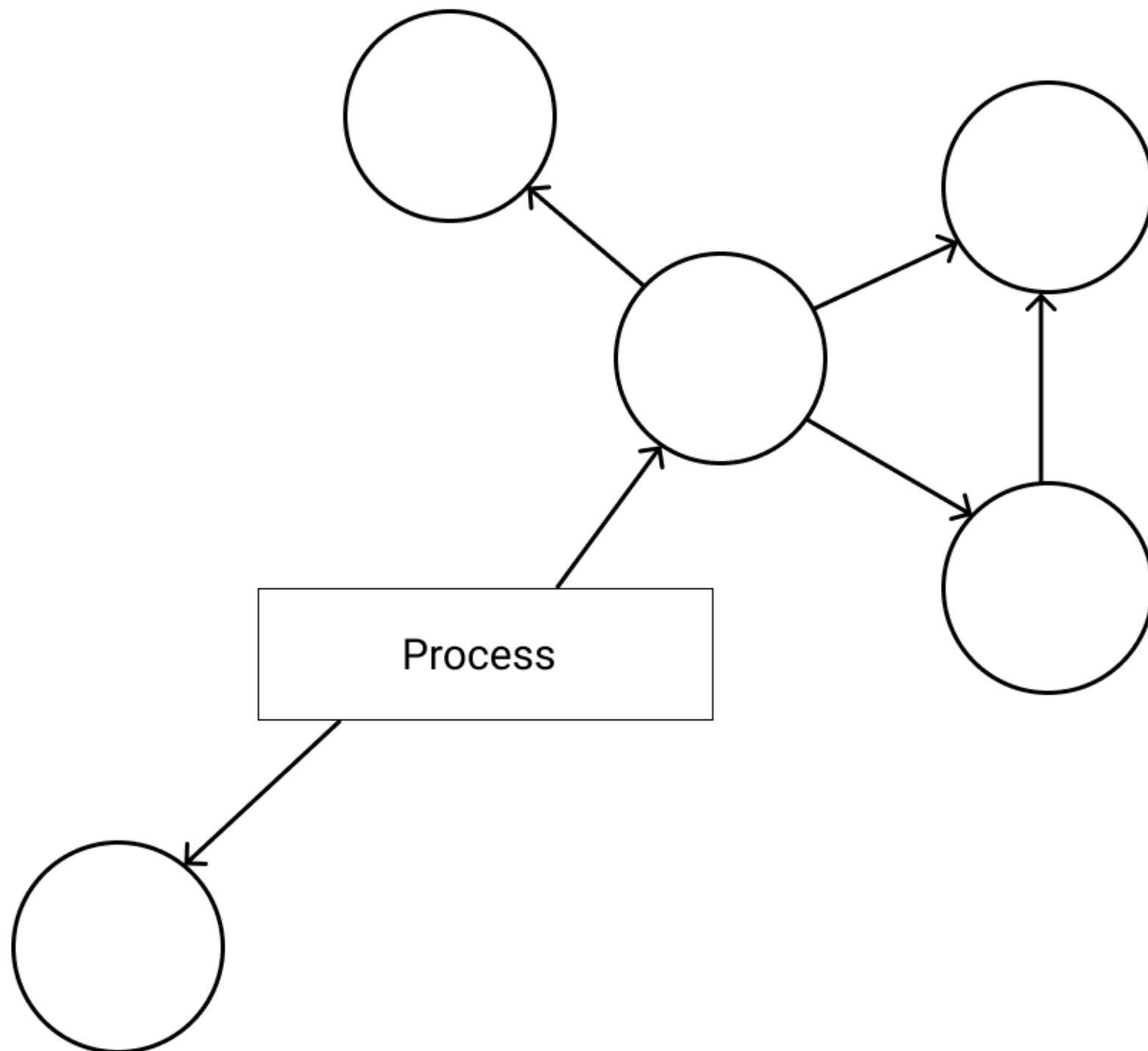
## Сборка мусора: разметка объектов



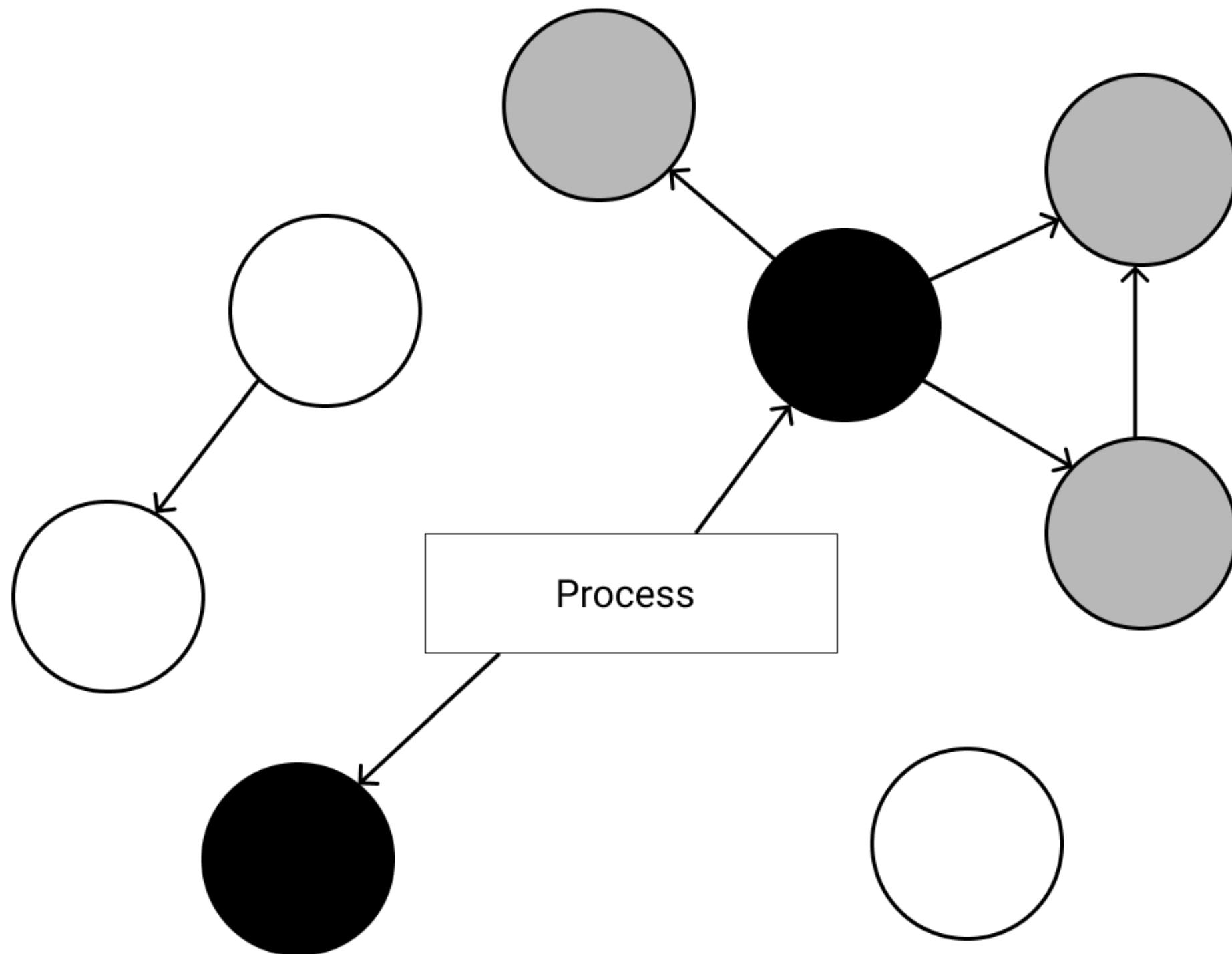
## Сборка мусора: разметка объектов



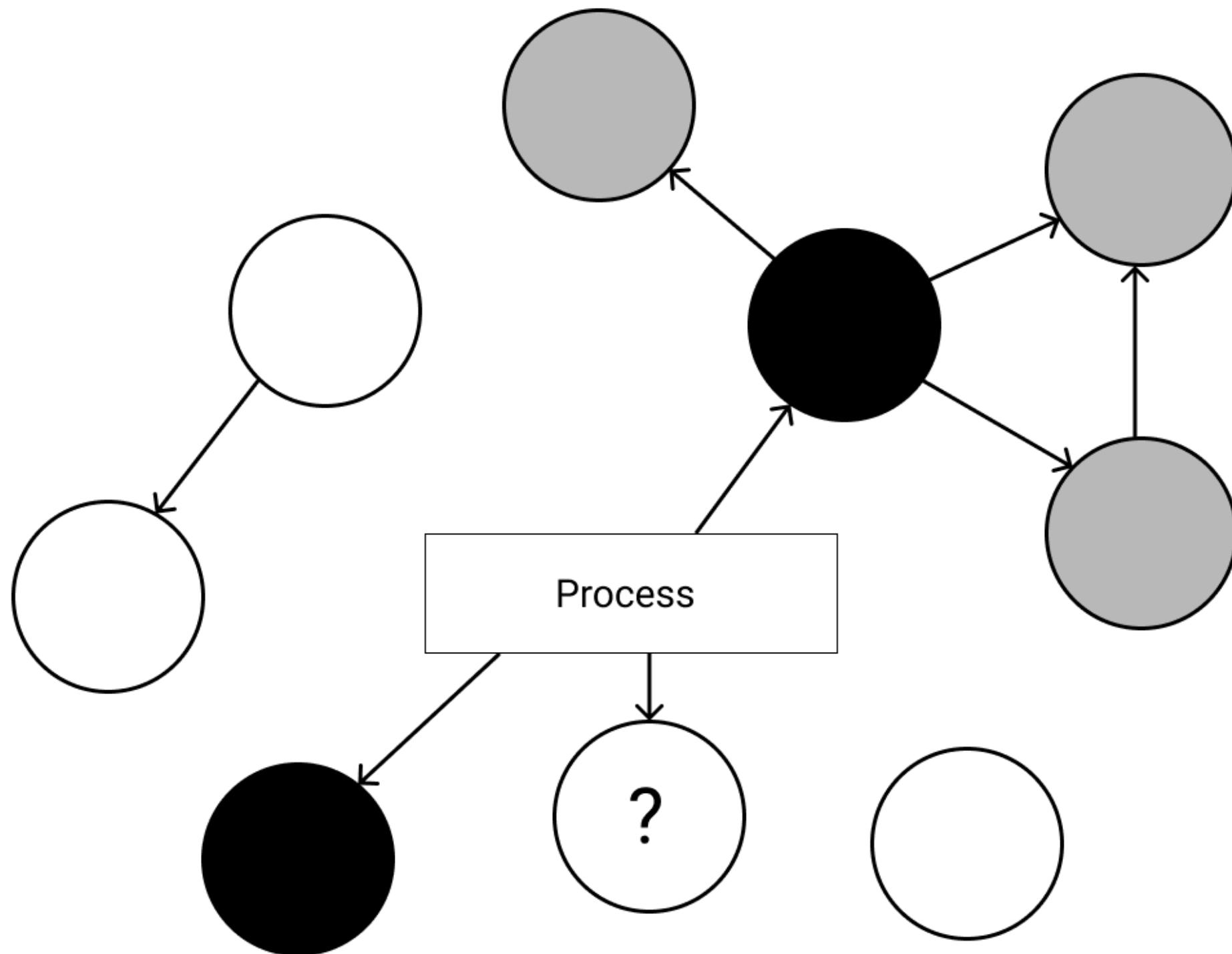
## Сборка мусора: STW



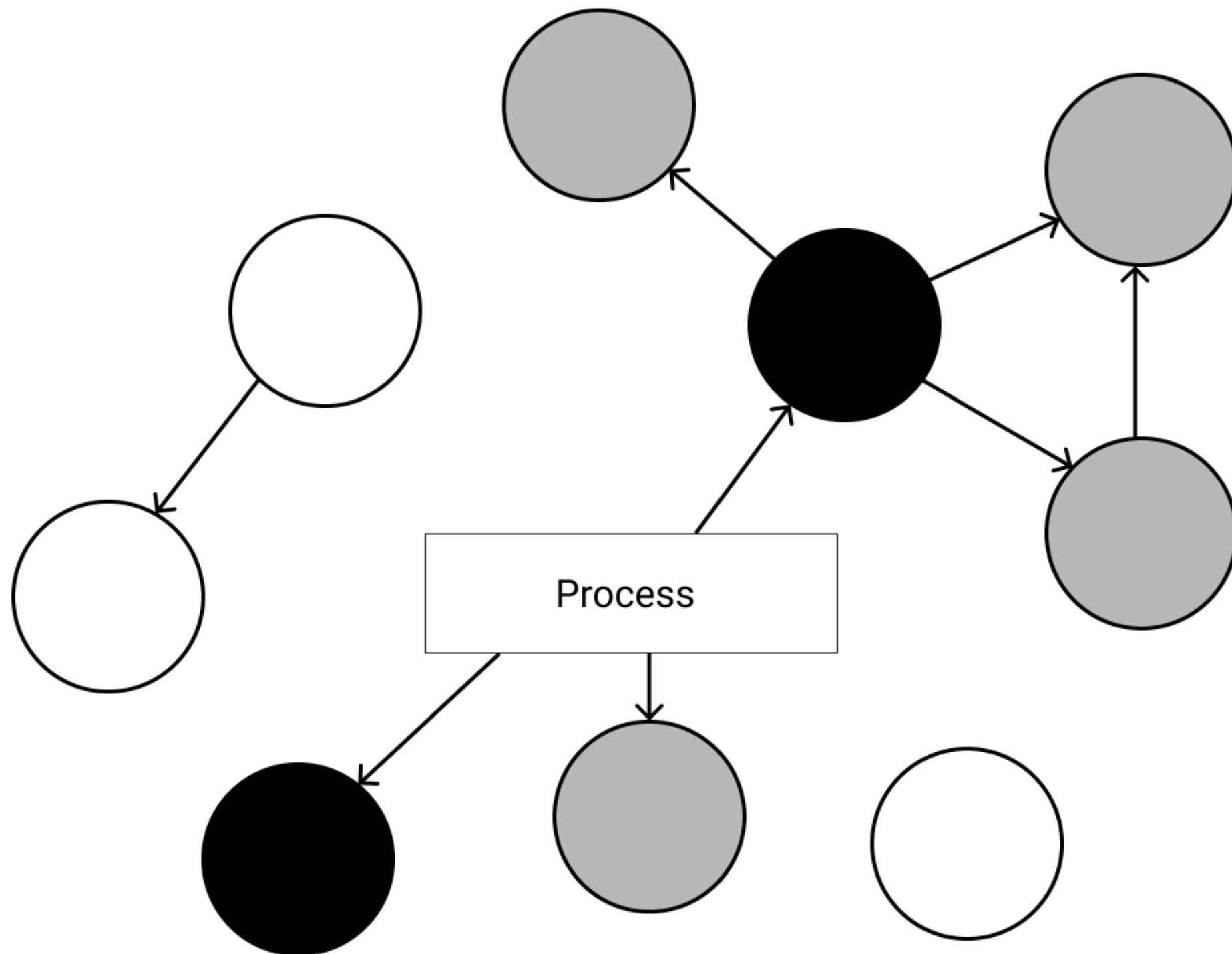
## Сборка мусора: write barrier



## Сборка мусора: write barrier



## Сборка мусора: write barrier



# Сборка мусора: чего это стоит?

# Сборка мусора: чего это стоит?

- 25% CPU
- .

# Сборка мусора: чего это стоит?

- 25% CPU (иногда больше, если GC не успевает)
- .

# Сборка мусора: чего это стоит?

- 25% CPU (иногда больше, если GC не успевает)
- Замедление выделения памяти.

# Простое правило

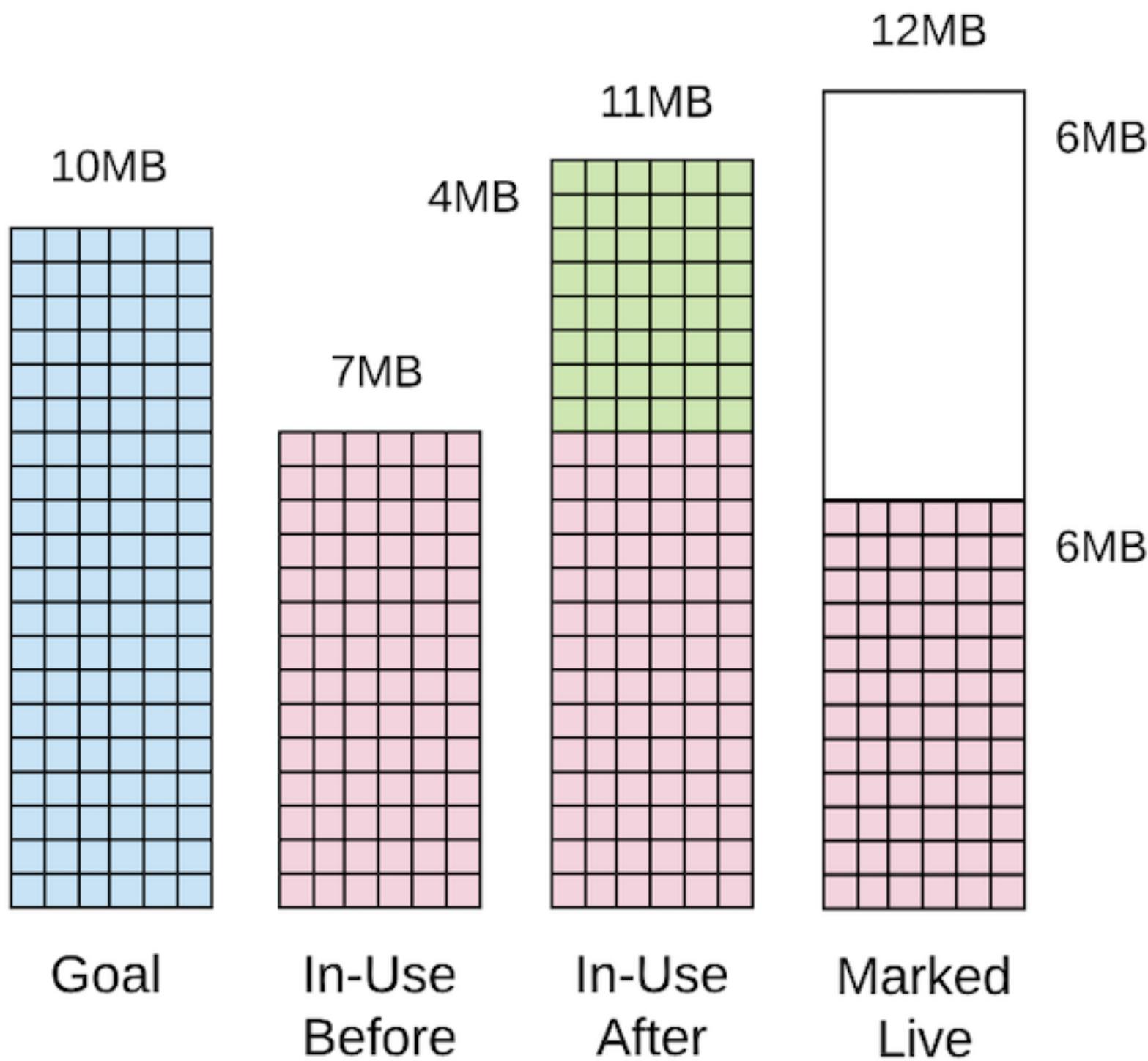
**Уменьшайте количество  
мусора.**

Чисто не там где убирают, а там, где не мусорят.

# Сборка мусора: моменты запуска

- По времени.
- По количеству выделенной памяти.
- Вручную (`runtime.GC()`).

## Сборка мусора: GOGC



## Простое правило

Не трогайте GOGC.

Уменьшайте количество  
ненужных аллокаций  
и мусора.

# Вопросы?



Ставим “+”,  
если вопросы есть



Ставим “-”,  
если вопросов нет

**Заполните, пожалуйста,  
опрос о занятии  
по ссылке в чате**



# Следующий вебинар

## Работа с вводом/выводом в Go



✓ Ссылка на вебинар будет в ЛК за 15 минут

✓ Материалы к занятию в ЛК – можно изучать

ribbon icon  
Обязательный материал обозначен красной лентой

Спасибо за внимание!

# Приходите на следующие вебинары

