ОСНОВЫ WEB-РАЗРАБОТКИ

Лекция 13. Внутреннее устройство языка Golang

Курс читают:

Шульман В.Д. @ShtuzerVD
Пелевина Т.В. @anivelat
Шабанов В.В. @ZeroHug
Шумилин В.В @Nodthar1107

ТЕМЫ НА СЕГОДНЯ

- **Массивы**
- **Слайсы**
- **Мапы**
- **Сборщик мусора**
- **Планировщик горутин**

МАССИВЫ

Массивы - коллекция элементов одного типа. Длина массива *не* может изменяться. Вот как мы можем создать массив в Go:

```
arr := [4]int{3,2,5,4}
```

МАССИВЫ

Если мы создадим два массива в Go с разными длинами, то два массива будут иметь разные типы, так как **длина массива в Go, входит в его тип**:

```
a := [3]int{}
b := [2]int{}

// (a) [2]int и (b) [3]int - разные типы
```

МАССИВЫ


```go
5  func main() {
6      var arr = [3]int{1,2,3}
7      foo(arr)
8  }
9
10 func foo(input [4]int) {
11     fmt.Println(input)
12 }
```

```
# command-line-arguments 1017421/main.go:7:6:
cannot use arr (variable of type [3]int) as
[4]int value in argument to foo
```

МАССИВЫ

Более того, если нам лень писать длину массива, то мы можем сказать компилятору, чтобы он сам подсчитал длину:

```
a := [...]int{1, 2, 3} // [3]int
```

МАССИВЫ

СЛАЙСЫ

Слайсы можно создать двумя способами:

```go
// С помощью make
var foo []byte
foo = make([]byte, 5, 5)

// С помощью shorthand syntax
bar := []byte{}
```

СЛАЙСЫ

A slice can be created with the built-in function called make, which has the signature,

```
func make([]T, len, cap) []T
```

where T stands for the element type of the slice to be created. The make function takes a type, a length, and an optional capacity. When called, make allocates an array and returns a slice that refers to that array.

```
var s []byte
s = make([]byte, 5, 5)
// s == []byte{0, 0, 0, 0, 0}
```

СЛАЙСЫ

When the capacity argument is omitted, it defaults to the specified length. Here's a more succinct version of the same code:

```
s := make([]byte, 5)
```

СЛАЙСЫ


```go
func main() {
    var foo = make([]byte, 5)
    var bar = make([]int, 10)
    var fee = make([]string, 2)

    fmt.Println(foo, bar, fee)
}

/*
Output:
  [0 0 0 0 0] [0 0 0 0 0 0 0 0 0 0] [ ]
*/
```

СЛАЙСЫ

The length and capacity of a slice can be inspected using the built-in `len` and `cap` functions.

```
len(s) == 5
cap(s) == 5
```

СЛАЙСЫ


```
b := []byte{'g', 'o', 'l', 'a', 'n', 'g'}
// b[1:4] == []byte{'o', 'l', 'a'}, sharing the same storage as b
```

```
// b[:2] == []byte{'g', 'o'}
// b[2:] == []byte{'l', 'a', 'n', 'g'}
// b[:] == b
```

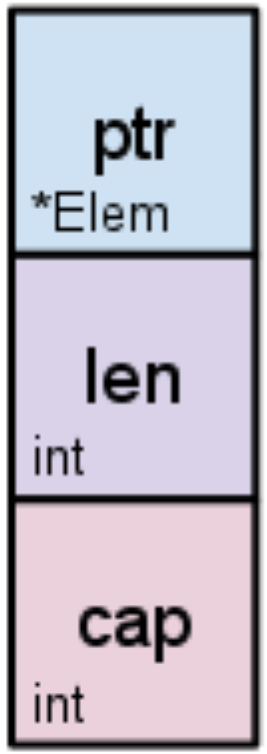
СЛАЙСЫ

This is also the syntax to create a slice given an array:

```
x := [3]string{"Лайка", "Белка", "Стрелка"}
s := x[:] // a slice referencing the storage of x
```

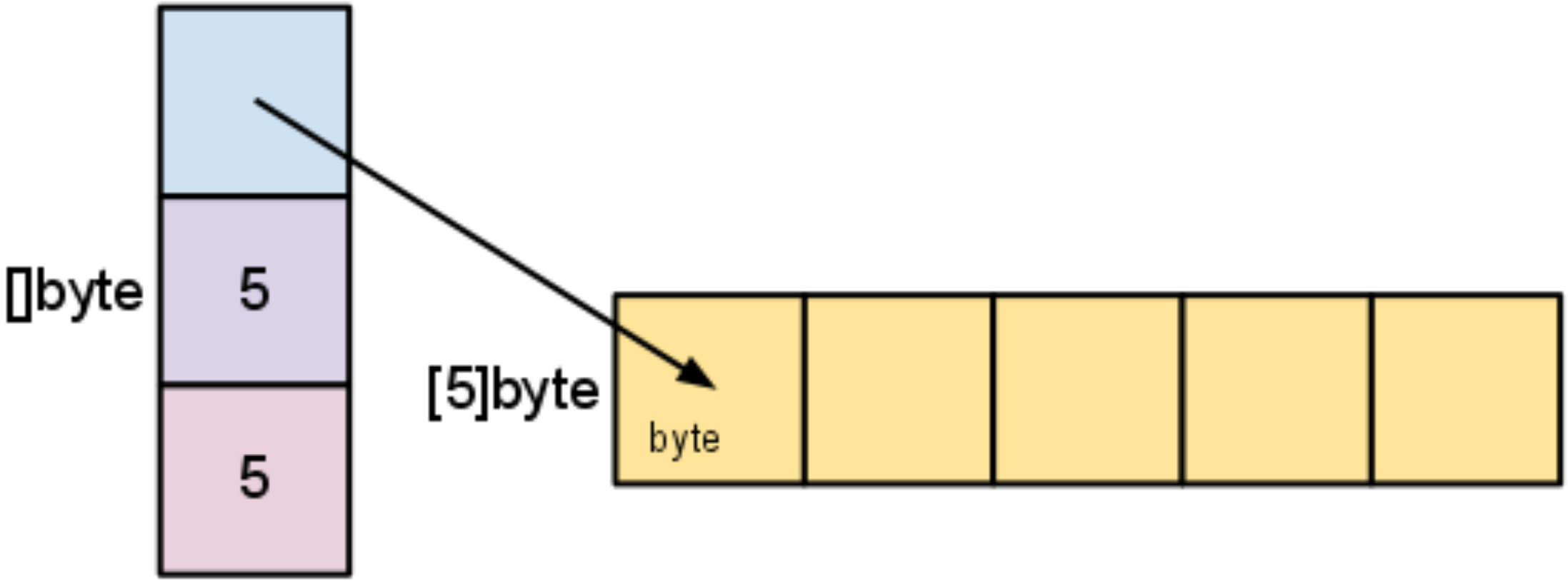
СЛАЙСЫ

A slice is a descriptor of an array segment. It consists of a pointer to the array, the length of the segment, and its capacity (the maximum length of the segment).



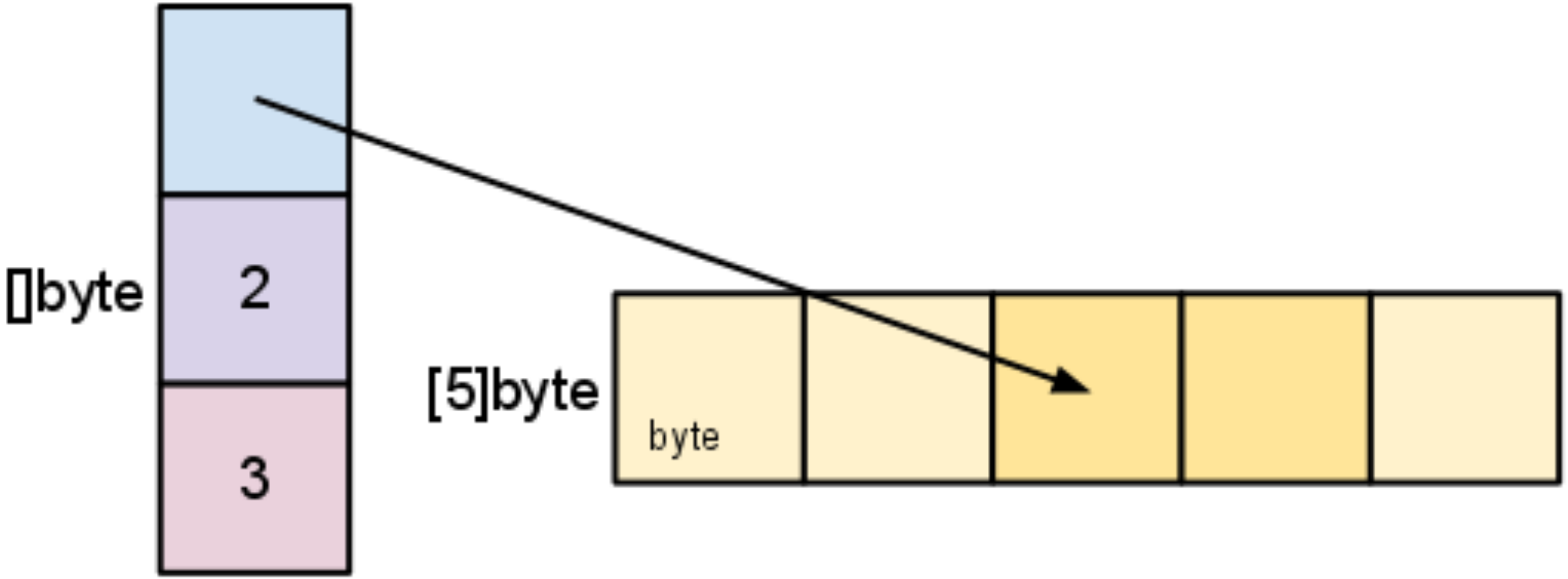
СЛАЙСЫ

Our variable s, created earlier by `make([]byte, 5)`, is structured like this:



СЛАЙСЫ

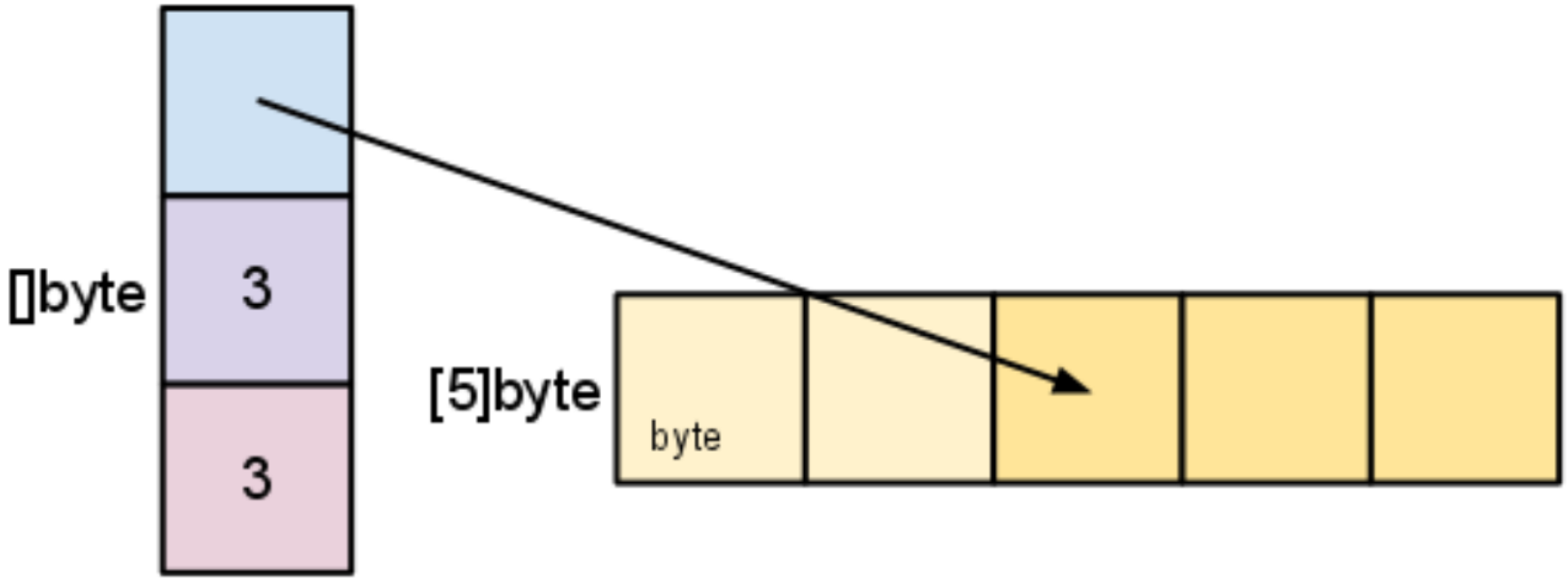

```
s = s[2:4]
```



СЛАЙСЫ

Earlier we sliced s to a length shorter than its capacity. We can grow s to its capacity by slicing it again:

```
s = s[:cap(s)]
```



СЛАЙСЫ

Slicing does not copy the slice's data. It creates a new slice value that points to the original array. This makes slice operations as efficient as manipulating array indices. Therefore, modifying the *elements* (not the slice itself) of a re-slice modifies the elements of the original slice:

```
d := []byte{'r', 'o', 'a', 'd'}
e := d[2:]
// e == []byte{'a', 'd'}
e[1] = 'm'
// e == []byte{'a', 'm'}
// d == []byte{'r', 'o', 'a', 'm'}
```

СЛАЙСЫ


```
t := make([]byte, len(s), (cap(s)+1)*2) // +1 in case cap(s) == 0
for i := range s {
        t[i] = s[i]
}
s = t
```

```
t := make([]byte, len(s), (cap(s)+1)*2)
copy(t, s)
s = t
```

СЛАЙСЫ

But most programs don't need complete control, so Go provides a built-in append function that's good for most purposes; it has the signature

```
func append(s []T, x ...T) []T
```

The append function appends the elements x to the end of the slice s, and grows the slice if a greater capacity is needed.

```
a := make([]int, 1)
// a == []int{0}
a = append(a, 1, 2, 3)
// a == []int{0, 1, 2, 3}
```

СЛАЙСЫ


```
a := make([]int, 1)
// a == []int{0}
a = append(a, 1, 2, 3)
// a == []int{0, 1, 2, 3}
```

To append one slice to another, use `...` to expand the second argument to a list of arguments.

```
a := []string{"John", "Paul"}
b := []string{"George", "Ringo", "Pete"}
a = append(a, b...) // equivalent to "append(a, b[0], b[1], b[2])"
// a == []string{"John", "Paul", "George", "Ringo", "Pete"}
```

МАПЫ

A Go map type looks like this:

```
map[KeyType]ValueType
```

where `KeyType` may be any type that is comparable (more on this later), and `ValueType` may be any type at all, including another map!

МАПЫ

This variable `m` is a map of string keys to int values:

```
var m map[string]int
```

Map types are reference types, like pointers or slices, and so the value of `m` above is `nil`; it doesn't point to an initialized map. A nil map behaves like an empty map when reading, but attempts to write to a nil map will cause a runtime panic; don't do that. To initialize a map, use the built in `make` function:

```
m = make(map[string]int)
```

МАПЫ


```
m := make(map[key_type]value_type)
m := new(map[key_type]value_type)
var m map[key_type]value_type
m := map[key_type]value_type{key1: val1, key2: val2}
```

МАПЫ

- Вставки: `insert(map, key, value)`
- Удаления: `delete(map, key)`
- Поиска: `lookup(key) → value`

МАПЫ

- Вставка:

```
m[key] = value
```

- Удаление:

```
delete(m, key)
```

МАПЫ

- Поиск:

```
value = m[key]
```

или

```
value, ok = m[key]
```

МАПЫ


```
package main

func main() {
    var m map[string]int
    for _, word := range []string{"hello", "world", "from", "the",
        "best", "language", "in", "the", "world"} {
        m[word]++
    }
    for k, v := range m {
        println(k, v)
    }
}
```

Вы ~~гофера~~ подвох видите? — А он есть!

МАПЫ

При попытке запуска такой программы получим панику и сообщение «assignment to entry in nil map». А все потому что мапа — ссылочный тип и мало объявить переменную, надо ее проинициализировать:

```
m := make(map[string]int)
```

Чуть пониже будет понятно почему это работает именно так. В начале было представлено аж 4 способа создания мапы, два из них мы рассмотрели — это объявление как переменную и создание через make. Еще можно создать с помощью "Composite literals" конструкции

```
map[key_type]value_type{}
```

МАПЫ


```go
package main

import "fmt"

func main() {
    m := map[int]bool{}
    for i := 0; i < 5; i++ {
        m[i] = ((i % 2) == 0)
    }
    for k, v := range m {
        fmt.Printf("key: %d, value: %t\n", k, v)
    }
}
```

МАПЫ

Запуск 1:

```
key: 3, value: false
key: 4, value: true
key: 0, value: true
key: 1, value: false
key: 2, value: true
```

Запуск 2:

```
key: 4, value: true
key: 0, value: true
key: 1, value: false
key: 2, value: true
key: 3, value: false
```

МАПЫ

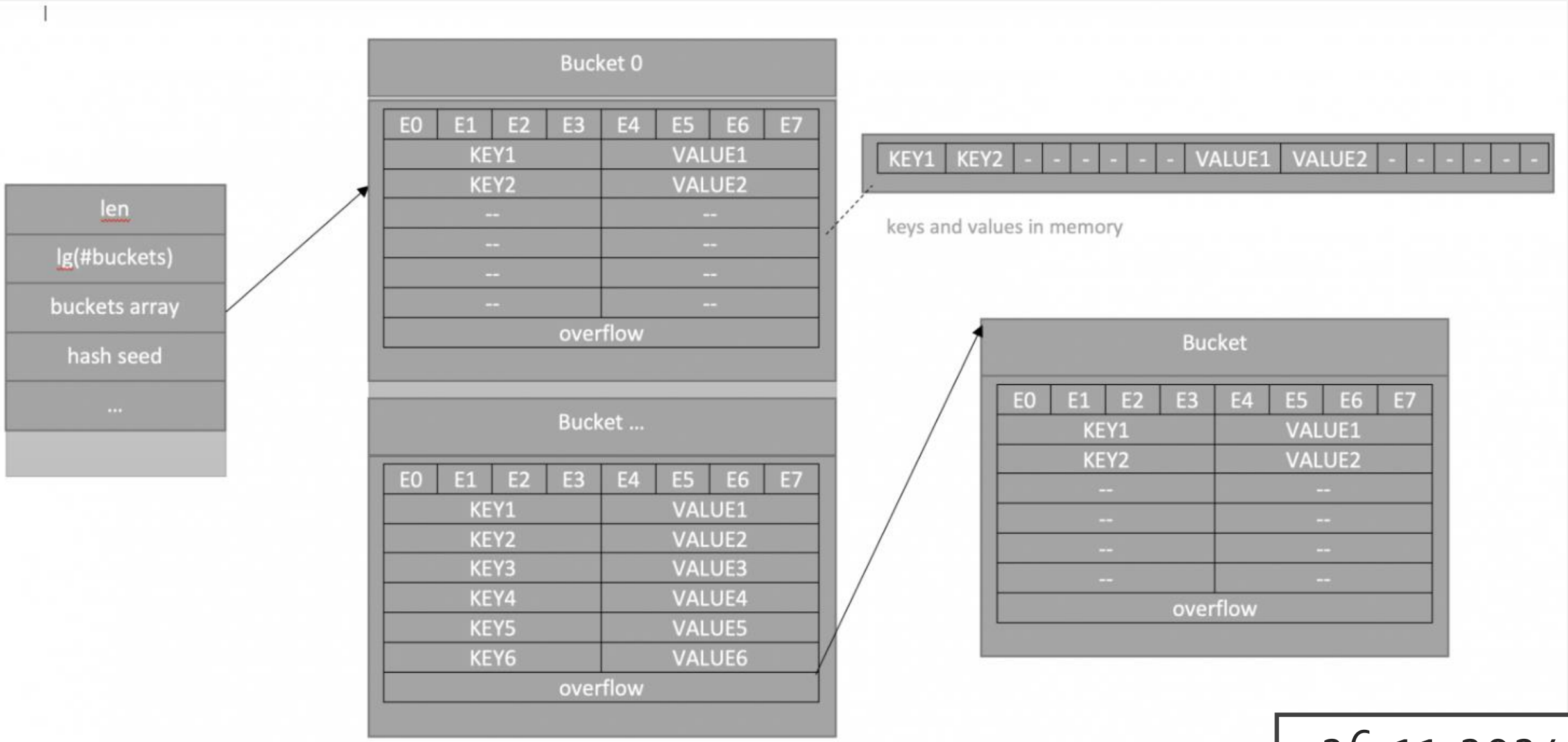
Как видим, вывод разнится от запуска к запуску. А все потому, что мапа в Go unordered, то есть не упорядоченная. Это значит, что полагаться на порядок при обходе не надо. Причину можно найти в исходном коде рантайма языка:

```
// mapiterinit initializes the hiter struct used for ranging over maps.
func mapiterinit(t *maptype, h *hmap, it *hiter) {...
// decide where to start
r := uintptr(fastrand())
 ...
it.startBucket = r & bucketMask(h.B)...}
```

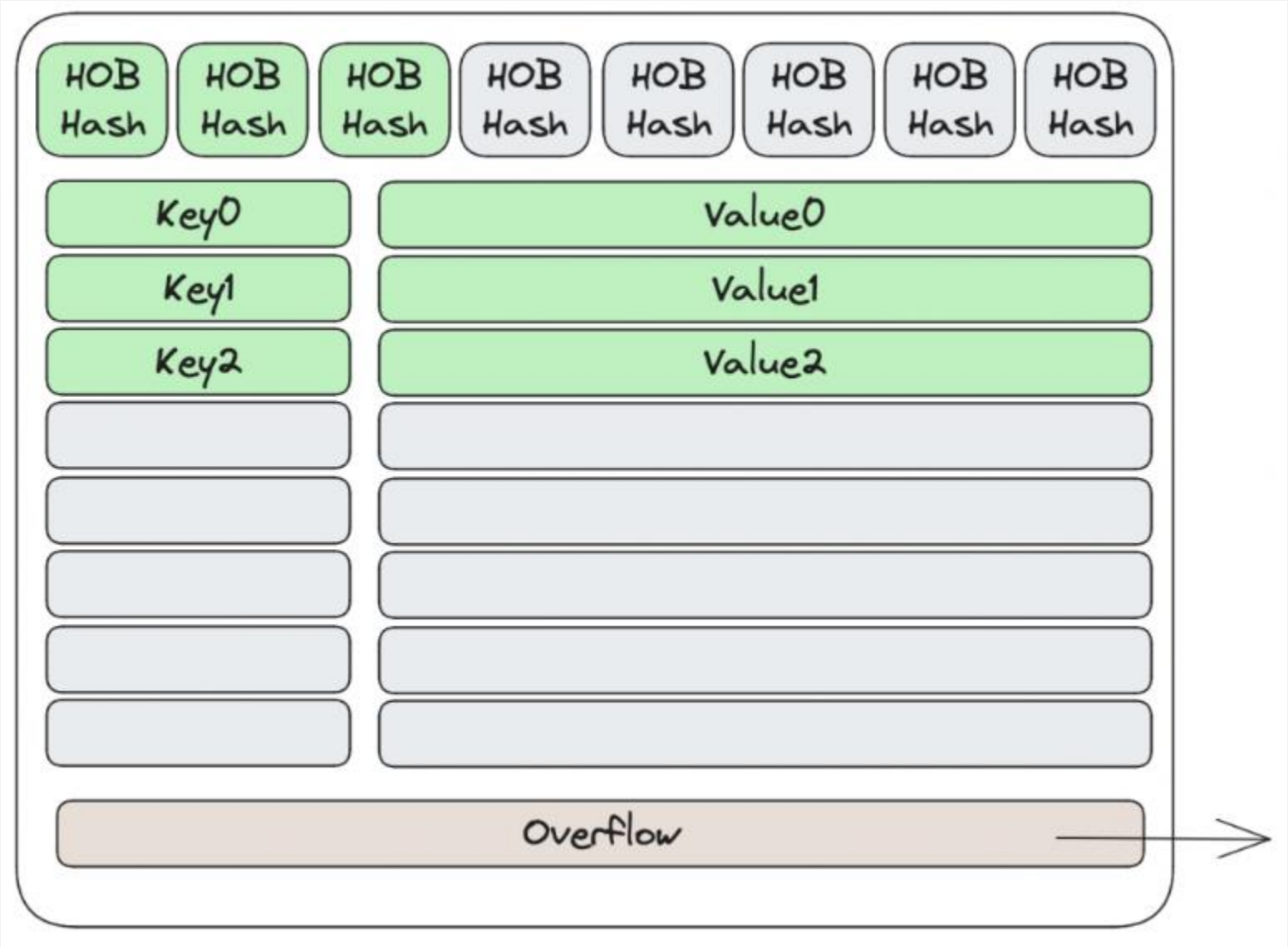
МАПЫ


```go
// A header for a Go map.
type hmap struct {
    // Note: the format of the hmap is also encoded in cmd/compile/internal/gc/reflect.go.
    // Make sure this stays in sync with the compiler's definition.
    count     int // # live cells == size of map.  Must be first (used by len() builtin)
    flags     uint8
    B         uint8  // log_2 of # of buckets (can hold up to loadFactor * 2^B items)
    noverflow uint16 // approximate number of overflow buckets; see incrnoverflow for details
    hash0     uint32 // hash seed

    buckets    unsafe.Pointer // array of 2^B Buckets. may be nil if count==0.
    oldbuckets unsafe.Pointer // previous bucket array of half the size, non-nil only when gr
    nevacuate  uintptr        // progress counter for evacuation (buckets less than this have

    extra *mapextra // optional fields
}
```

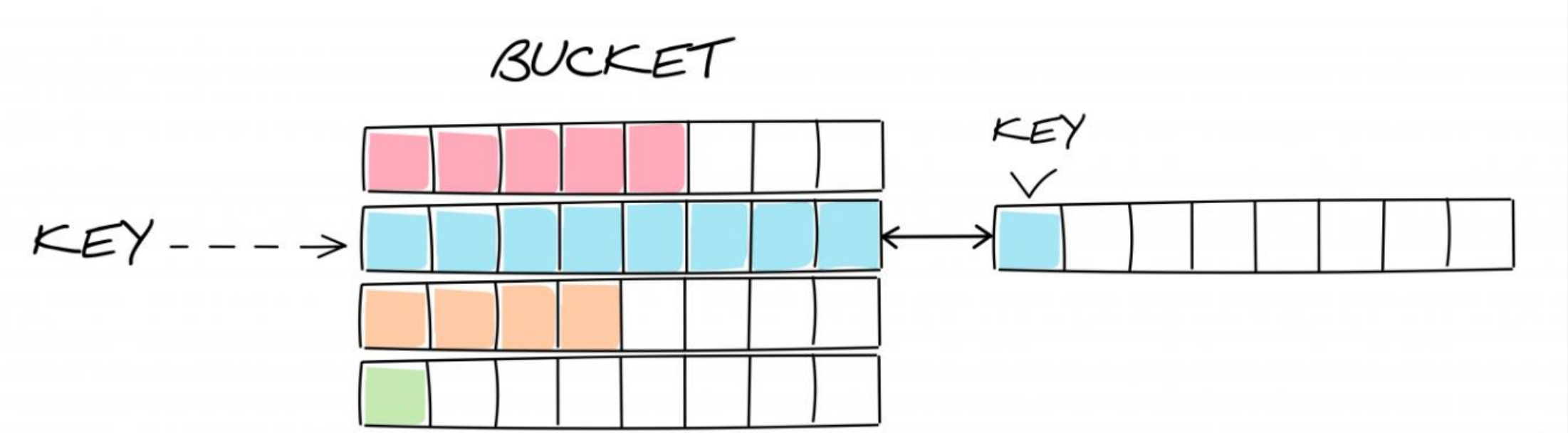
МАПЫ


len
lg(#buckets)
buckets array
hash seed
...
Bucket 0
E0 E1 E2 E3 E4 E5 E6 E7
KEY1 VALUE1
KEY2 VALUE2
-- --
-- --
-- --
-- --
overflow
Bucket ...
E0 E1 E2 E3 E4 E5 E6 E7
KEY1 VALUE1
KEY2 VALUE2
KEY3 VALUE3
KEY4 VALUE4
KEY5 VALUE5
KEY6 VALUE6
overflow
KEY1 KEY2 - - - - - - VALUE1 VALUE2 - - - - - -
keys and values in memory
Bucket
E0 E1 E2 E3 E4 E5 E6 E7
KEY1 VALUE1
KEY2 VALUE2
-- --
-- --
-- --
-- --
overflow

МАПЫ


HOB Hash
HOB Hash
HOB Hash
HOB Hash
HOB Hash
HOB Hash
HOB Hash
HOB Hash
Key0
Value0
Key1
Value1
Key2
Value2
Overflow

МАПЫ



МАПЫ

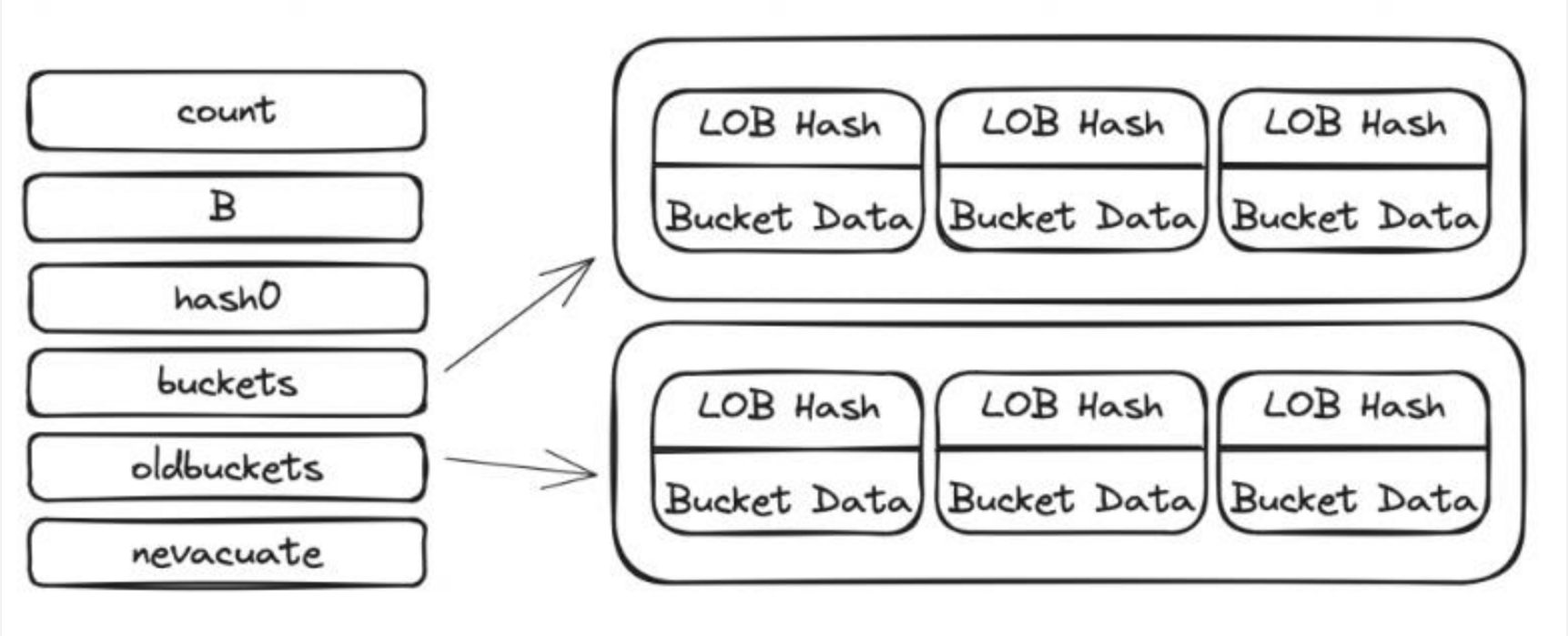
Как растет map?

В исходном коде можно найти строчку:

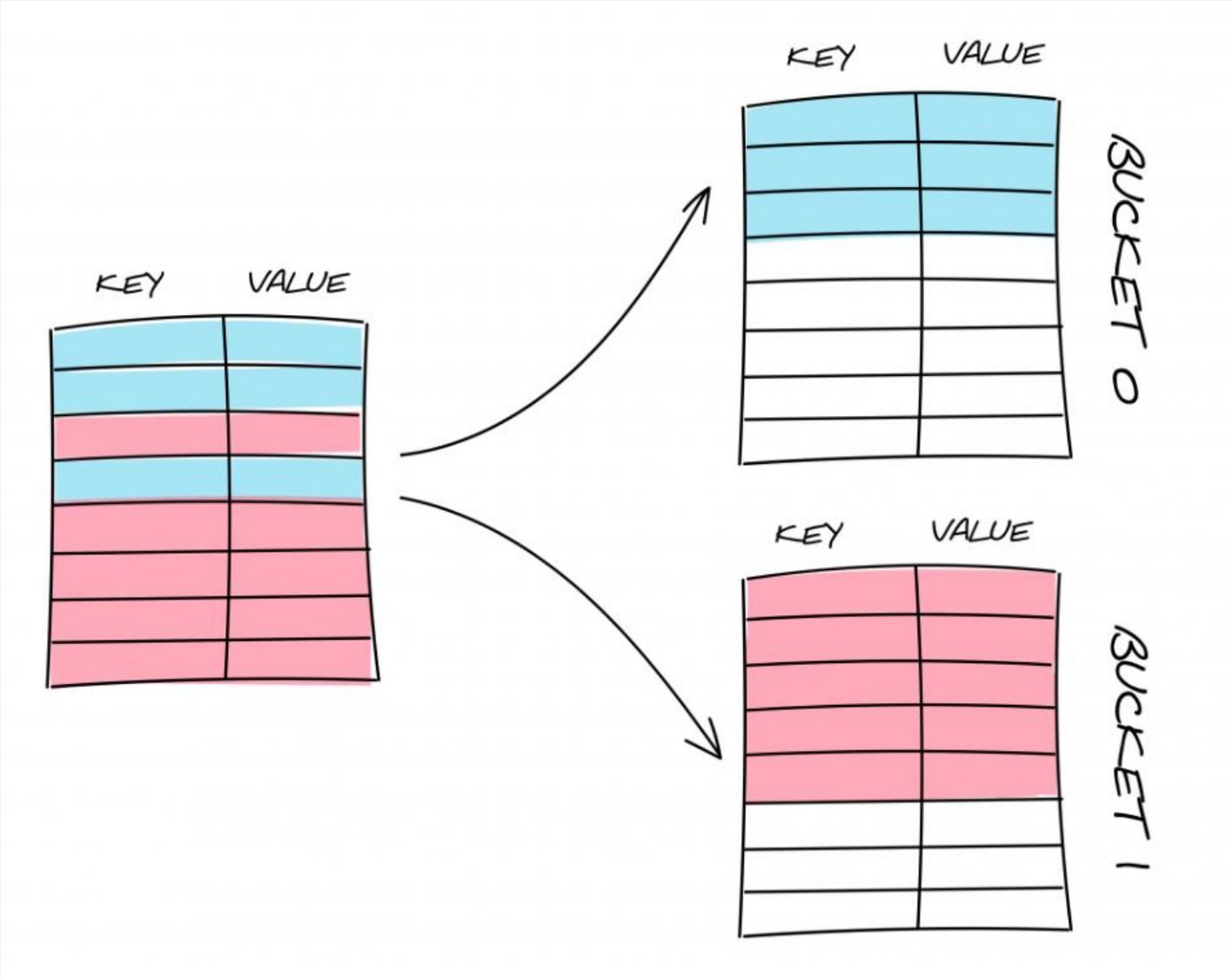
```
// Maximum average load of a bucket that triggers growth is 6.5.
```

то есть, если в каждом «ведре» в среднем более 6,5 элементов, происходит увеличение массива buckets. При этом выделяется массив в 2 раза больше, а старые данные копируются в него маленькими порциями каждые вставку или удаление, чтобы не создавать очень крупные задержки. Поэтому все операции будут чуть медленнее в процессе эвакуации данных (при поиске тоже, нам же приходится искать в двух местах). После успешной эвакуации начинают использоваться новые данные.

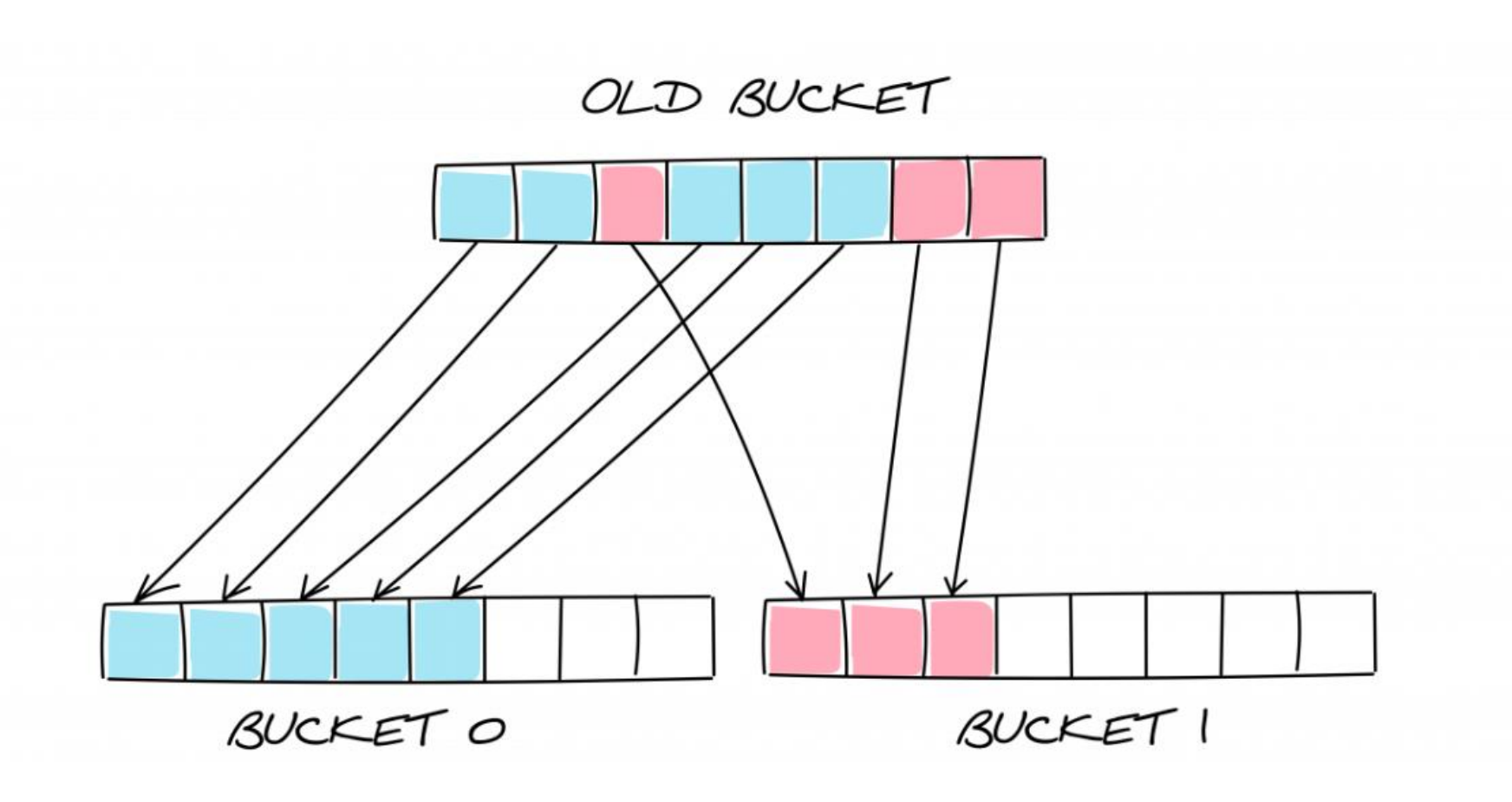
МАПЫ


count
B
hash0
buckets
oldbuckets
nevacuate
LOB Hash
Bucket Data
LOB Hash
Bucket Data
LOB Hash
Bucket Data
LOB Hash
Bucket Data
LOB Hash
Bucket Data
LOB Hash
Bucket Data

МАПЫ

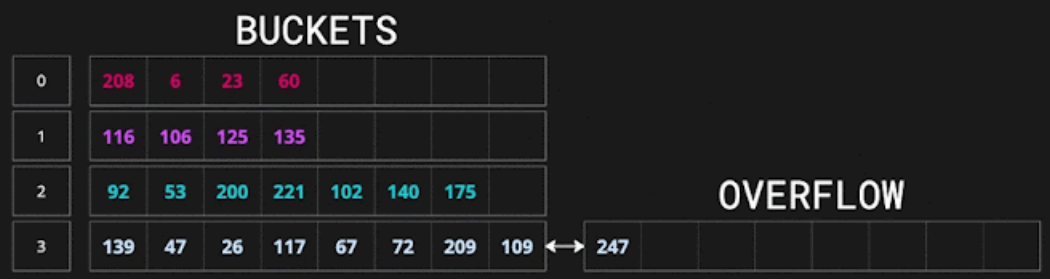


МАПЫ



МАПЫ


```
map["asparagus"]    tophash=92   // hash=6676846025543344914
map["parsley"]      tophash=116  // hash=8362162976769191333
map["dill"]         tophash=139  // hash=10059918011756683267
map["celery"]       tophash=208  // hash=14992832796737831644
map["spinach"]      tophash=53   // hash=3845764748768971554
map["ginger"]       tophash=6    // hash=129487862405433856
map["onion"]        tophash=47   // hash=3405719629583682135
map["garlic"]       tophash=23   // hash=1660642884483801968
map["carrot"]       tophash=200  // hash=14425631749397988578
map["potato"]       tophash=26   // hash=19335412573092058883
map["tomato"]       tophash=106  // hash=7649745411964669949
map["cucumber"]     tophash=221  // hash=159900803025728833334
map["orange"]       tophash=117  // hash=84627946350616075111
map["watermelon"]   tophash=60   // hash=43384266893119521688
map["banana"]       tophash=102  // hash=73725344714819994466
map["grape"]        tophash=125  // hash=90667813675925812899
map["pomegranate"]  tophash=135  // hash=97997741960697345577
map["grapefruit"]   tophash=67   // hash=48990523149081700711
map["pear"]         tophash=72   // hash=52007753084715932199
map["guava"]        tophash=209  // hash=15074973090940376547
map["melon"]        tophash=140  // hash=10118944801101152178
map["fig"]          tophash=109  // hash=7859390306932591675
map["kiwi"]         tophash=175  // hash=12677119218972580930
map["lemon"]        tophash=247  // hash=17858657349740820359
```



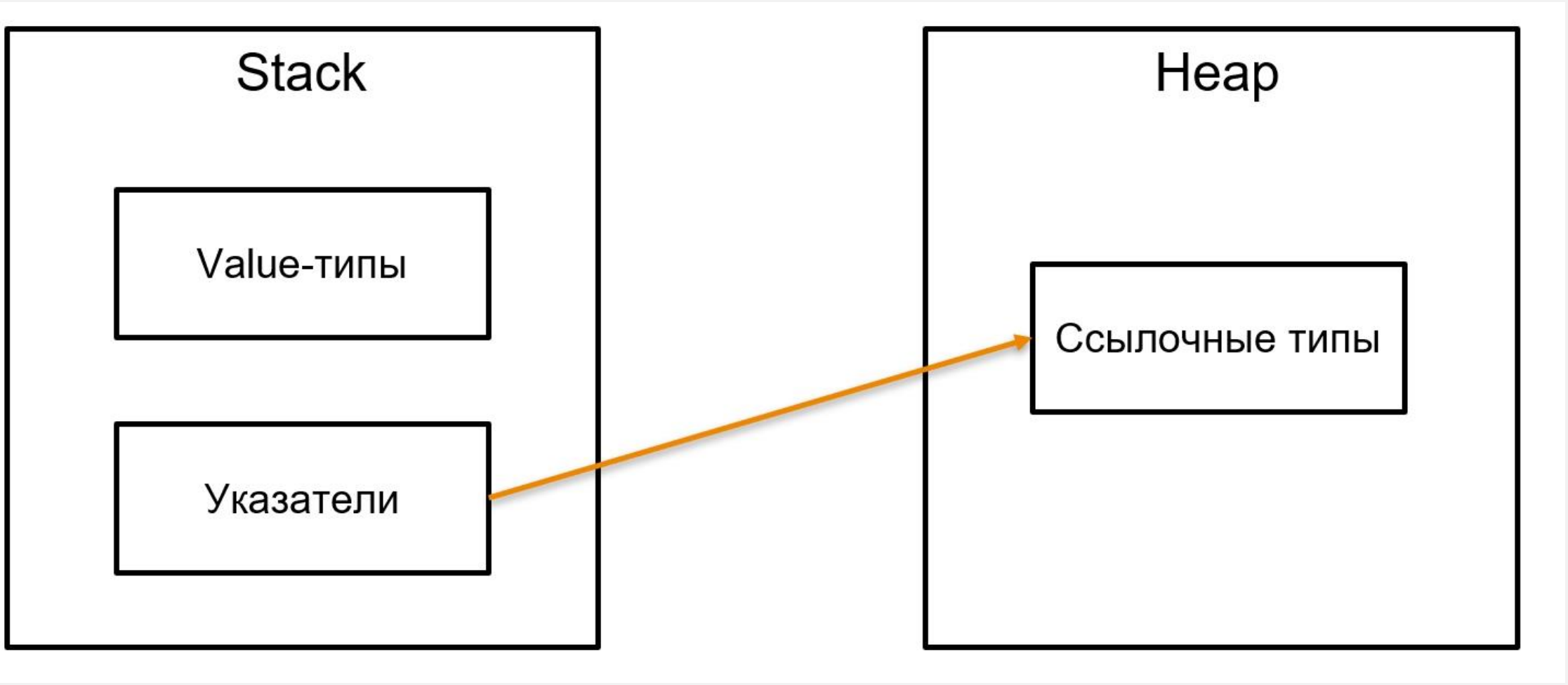
МАПЫ

Эвакуация становится одной из причин, почему мы не можем обратиться к ключу по адресу, так как мы не можем предсказать, в каком месте будет находиться наша пара. Возможно, она уже находится в новом бакете, а возможно, еще в старом. Кроме того, ясно, что процесс эвакуации данных не является бесплатным, и чтобы избежать его, можно заранее определить количество элементов в нашей мапе.

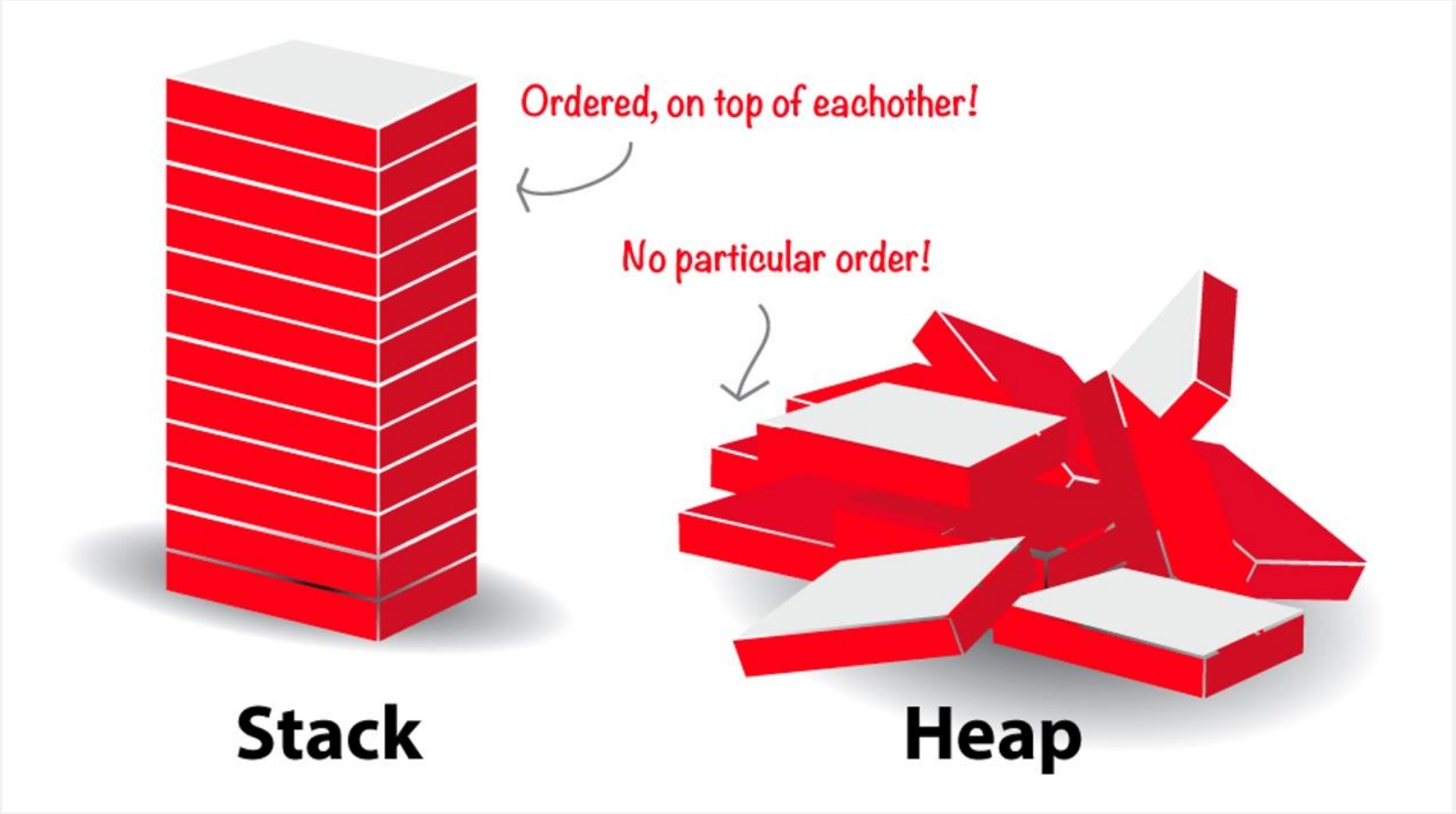
```
myMap := make(map[string]string, 1000)
```

Если вы скажете, сколько элементов будет на этапе создания мапы, выделится нужное количество бакетов под это количество элементов и тогда вы избежите эвакуации.

СБОРЩИК МУСОРА



СБОРЩИК МУСОРА



СБОРЩИК МУСОРА

Escape analysis (Escape analysis) в языке Go является одной из оптимизаций компилятора, которая позволяет определить, будет ли объект или переменная "выброшена" из локальной области и будет ли использоваться вне нее. Этот анализ позволяет определить, следует ли выделить память для объекта на куче или же можно использовать стек для его хранения. Вот некоторые особенности анализа эскейпа в Go:

СБОРЩИК МУСОРА

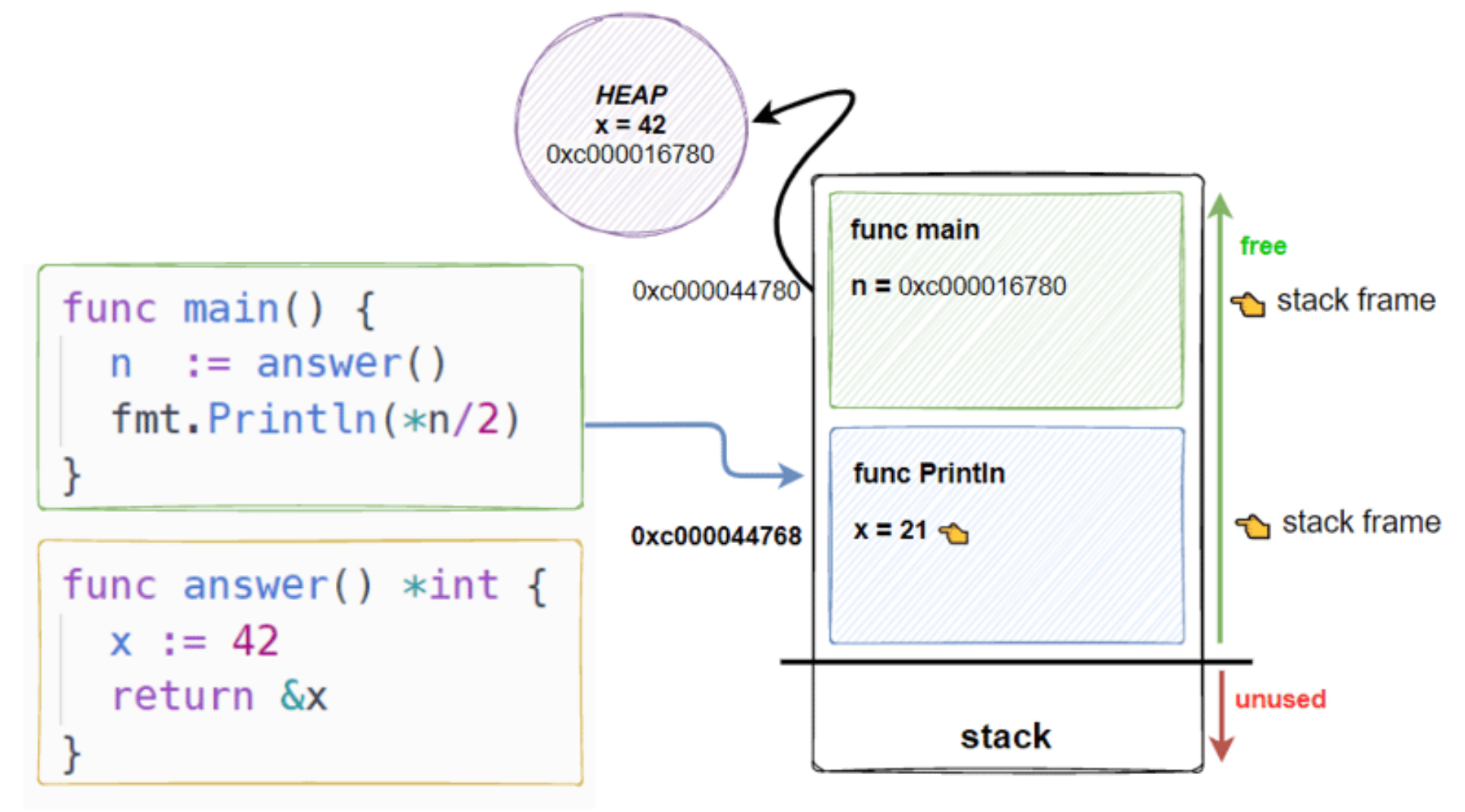
Стековое распределение (Stack Allocation)

Escape analysis пытается найти переменные, которые могут быть безопасно распределены на стеке вместо кучи. Это происходит, когда переменная не будет использоваться после выхода из локальной области, например, когда она не передается в другие функции или не сохраняется для использования после возврата из текущей функции.

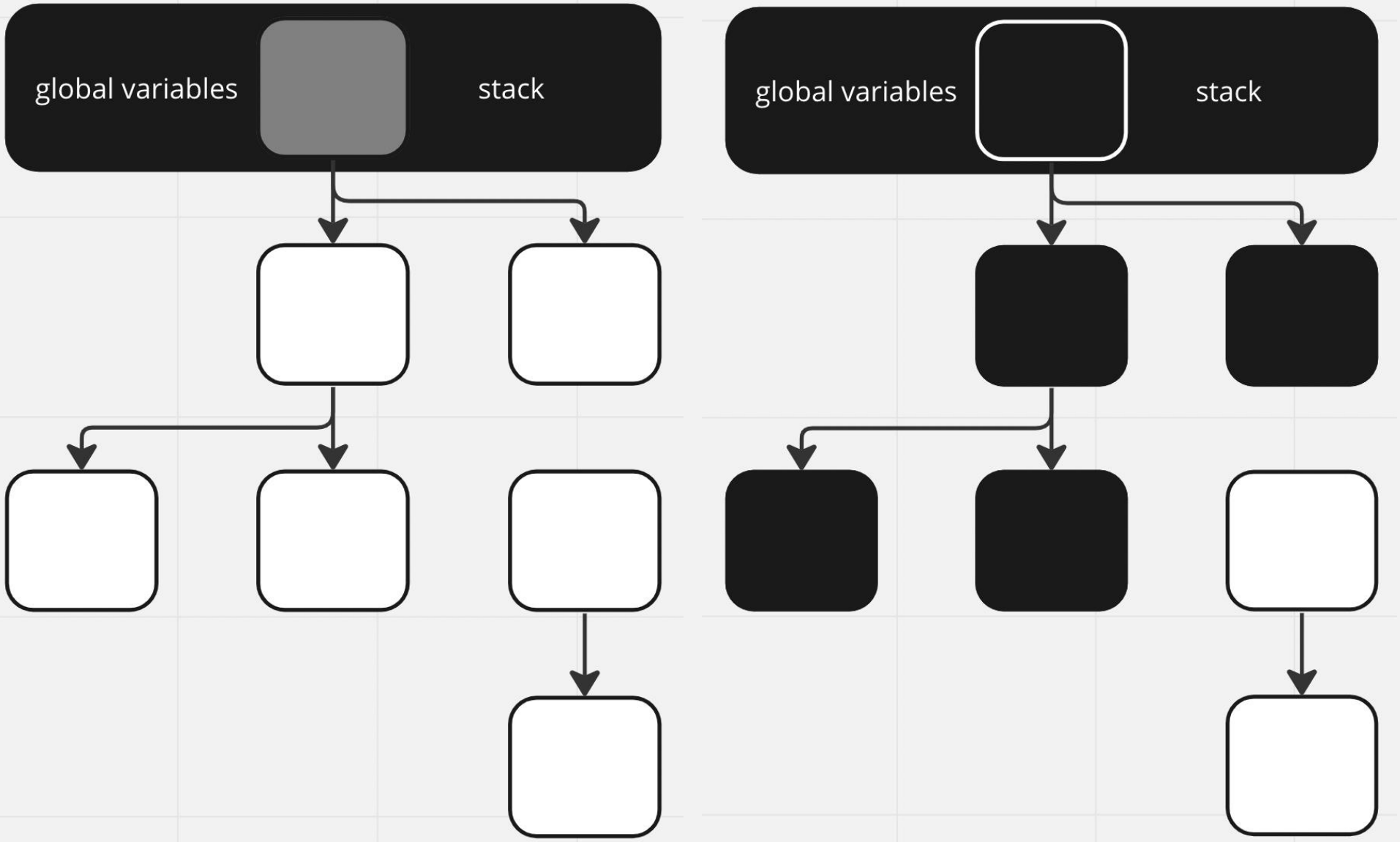
Кучевое распределение (Heap Allocation)

Если Escape analysis обнаруживает, что переменная будет использоваться за пределами локальной области, то объект или переменная будет выделена на куче. Например, когда переменная передается по указателю или сохраняется в глобальной области, она считается "выброшенной" из локальной области и требует распределения памяти на куче.

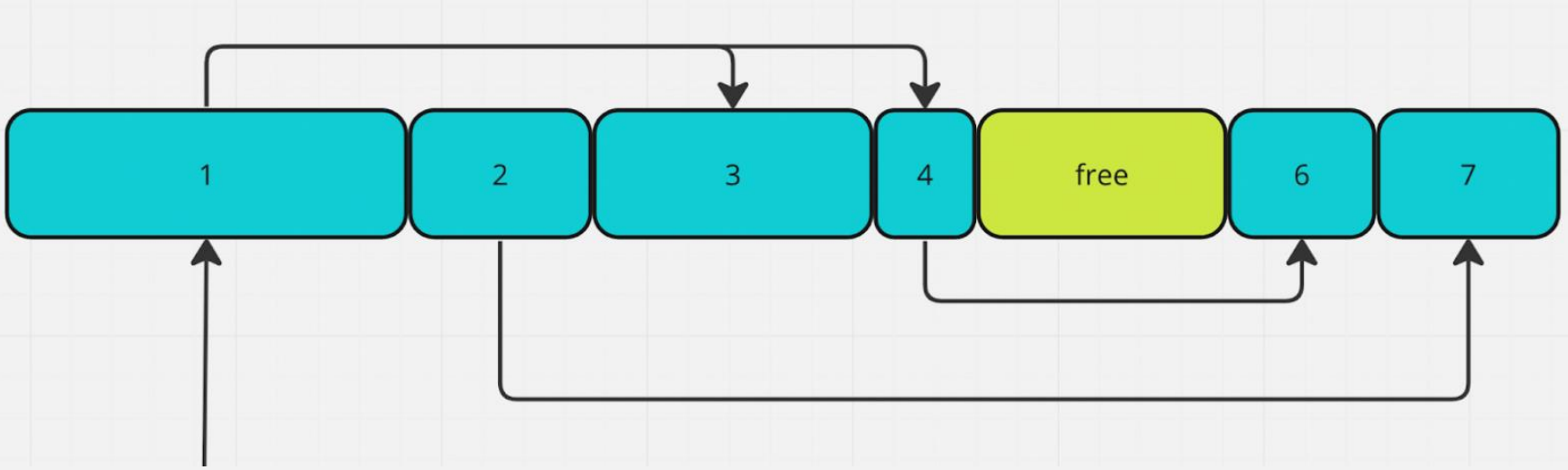
СБОРЩИК МУСОРА

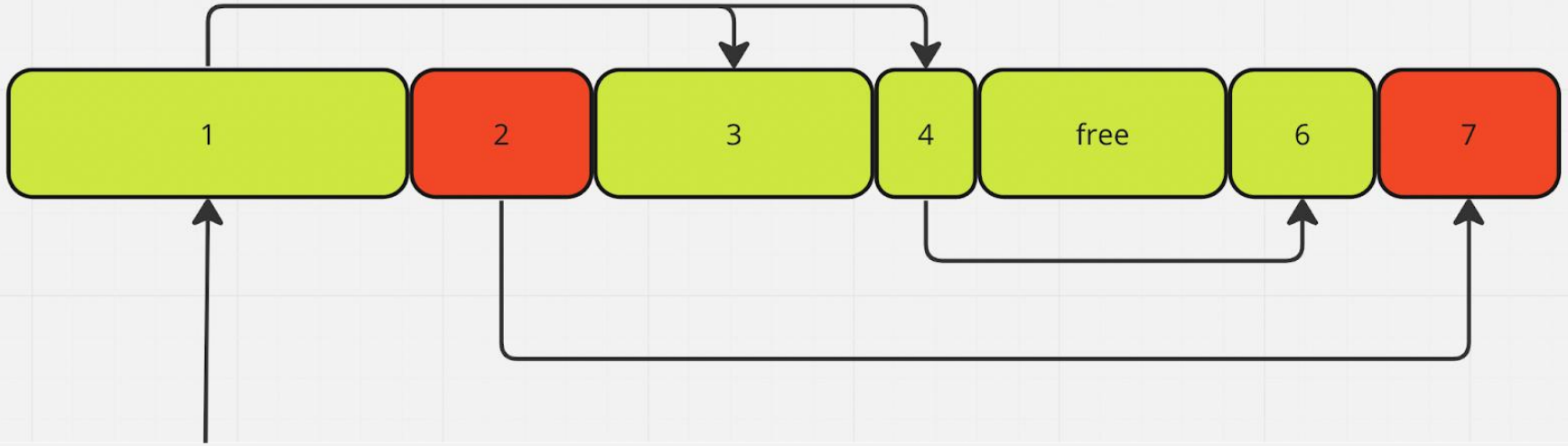


СБОРЩИК МУСОРА



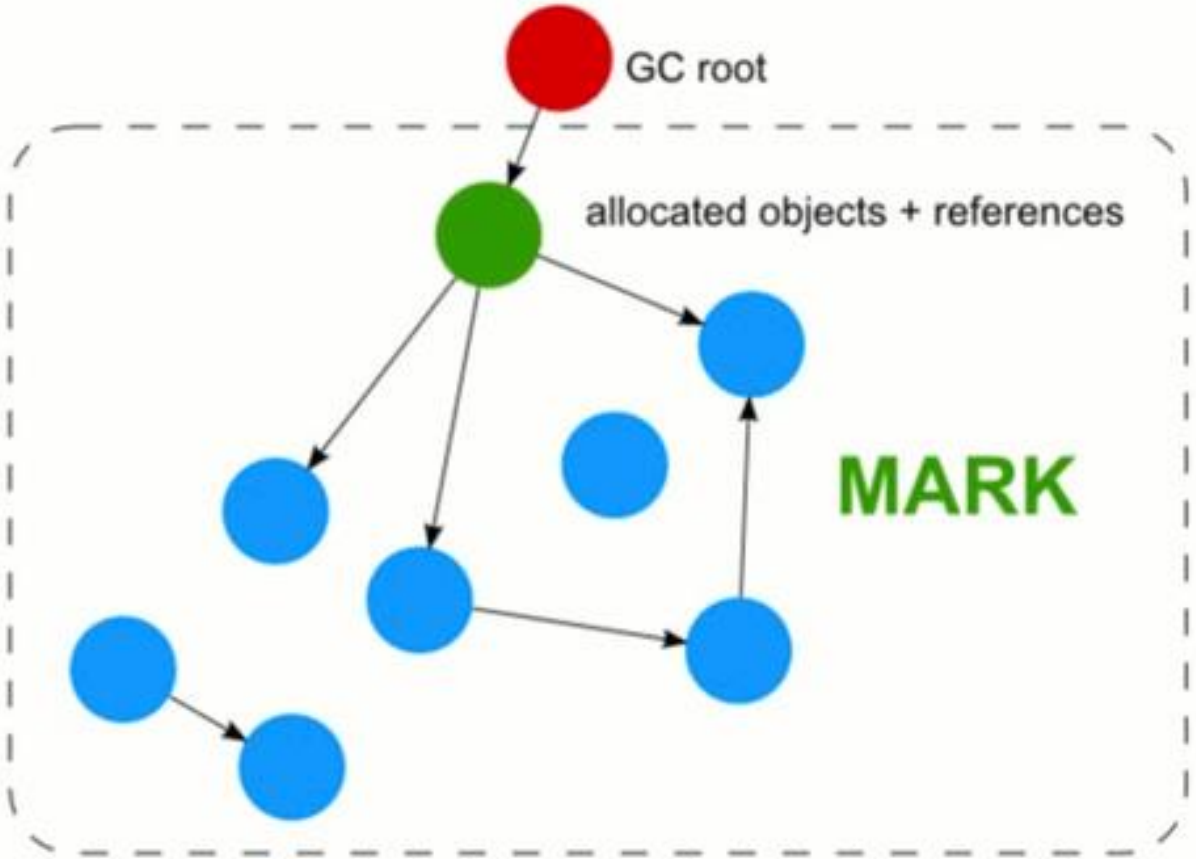
СБОРЩИК МУСОРА



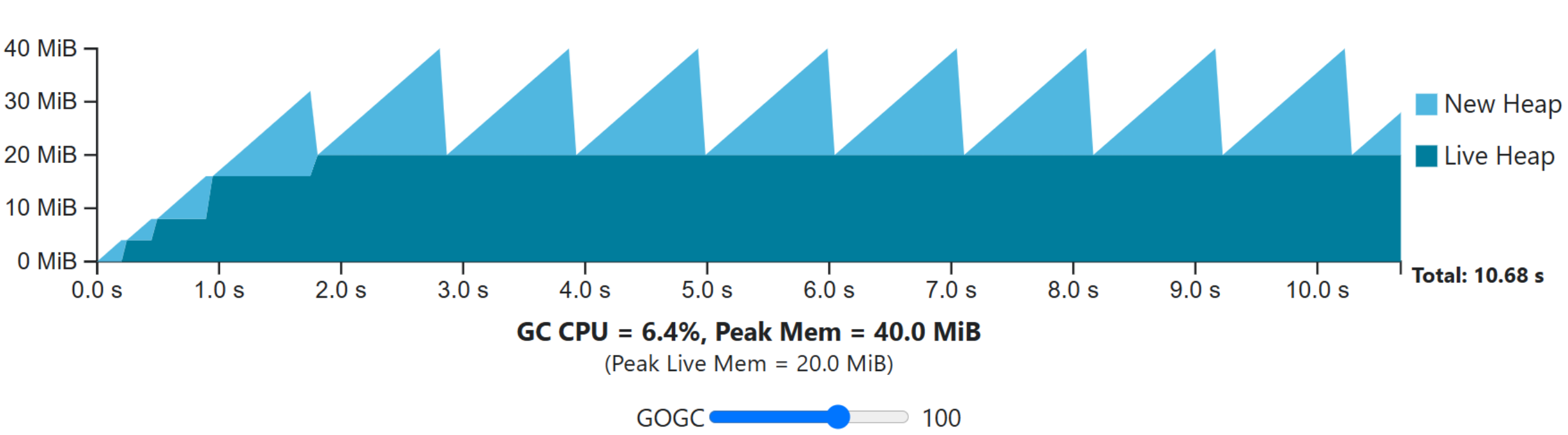


СБОРЩИК МУСОРА

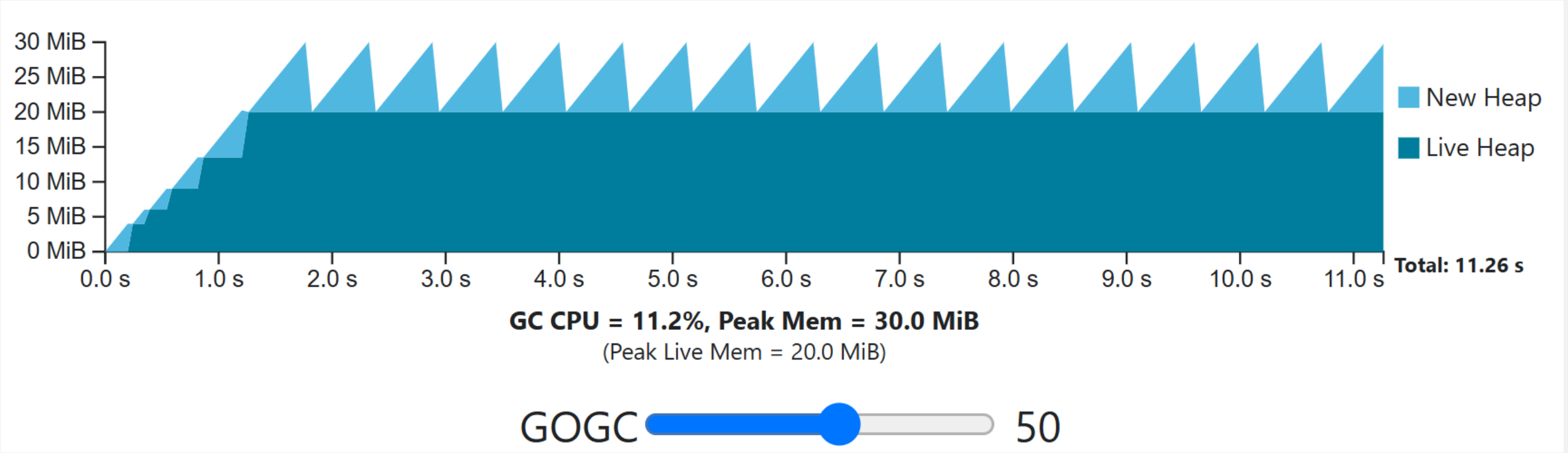
Mark and sweep (MARK)



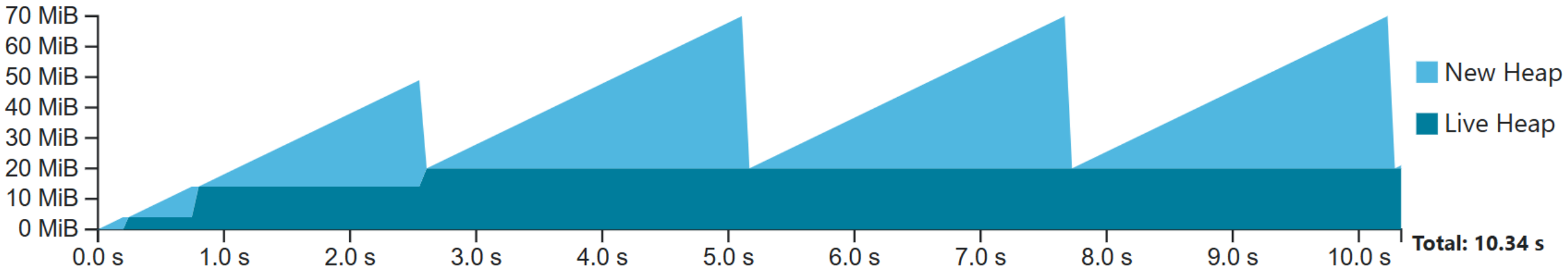
СБОРЩИК МУСОРА



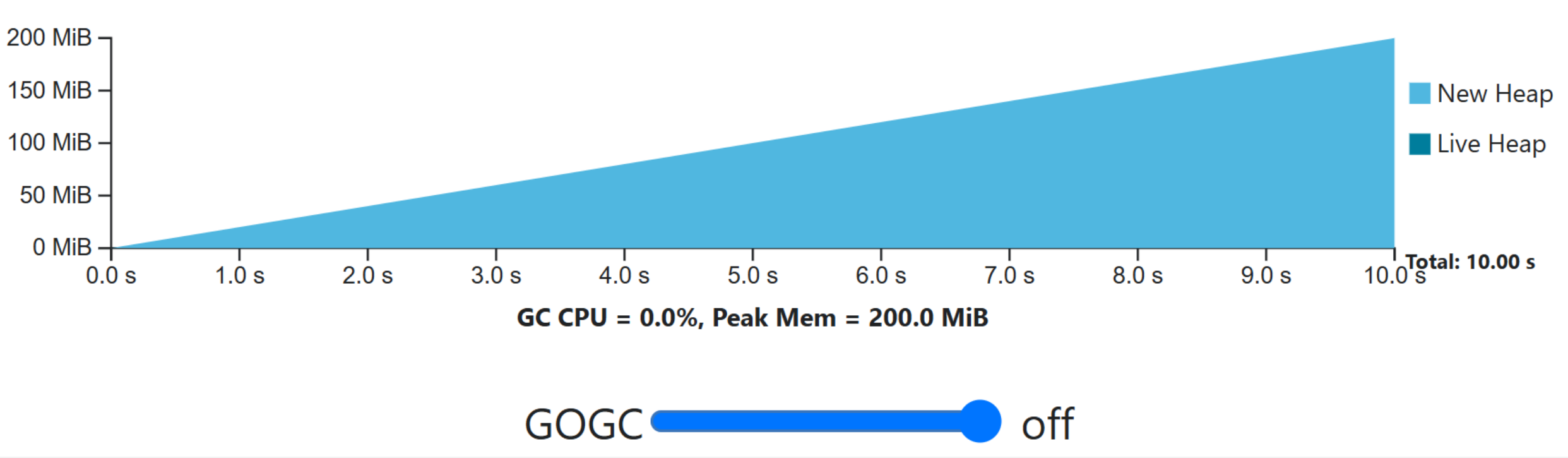
СБОРЩИК МУСОРА



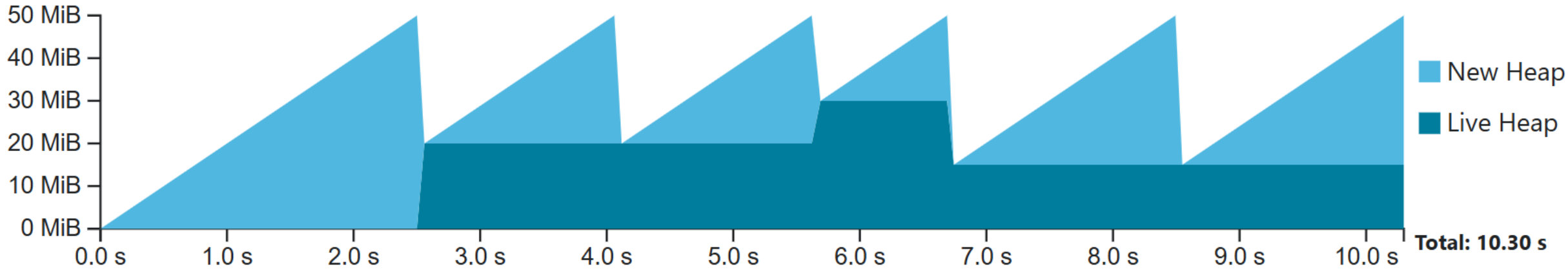
СБОРЩИК МУСОРА



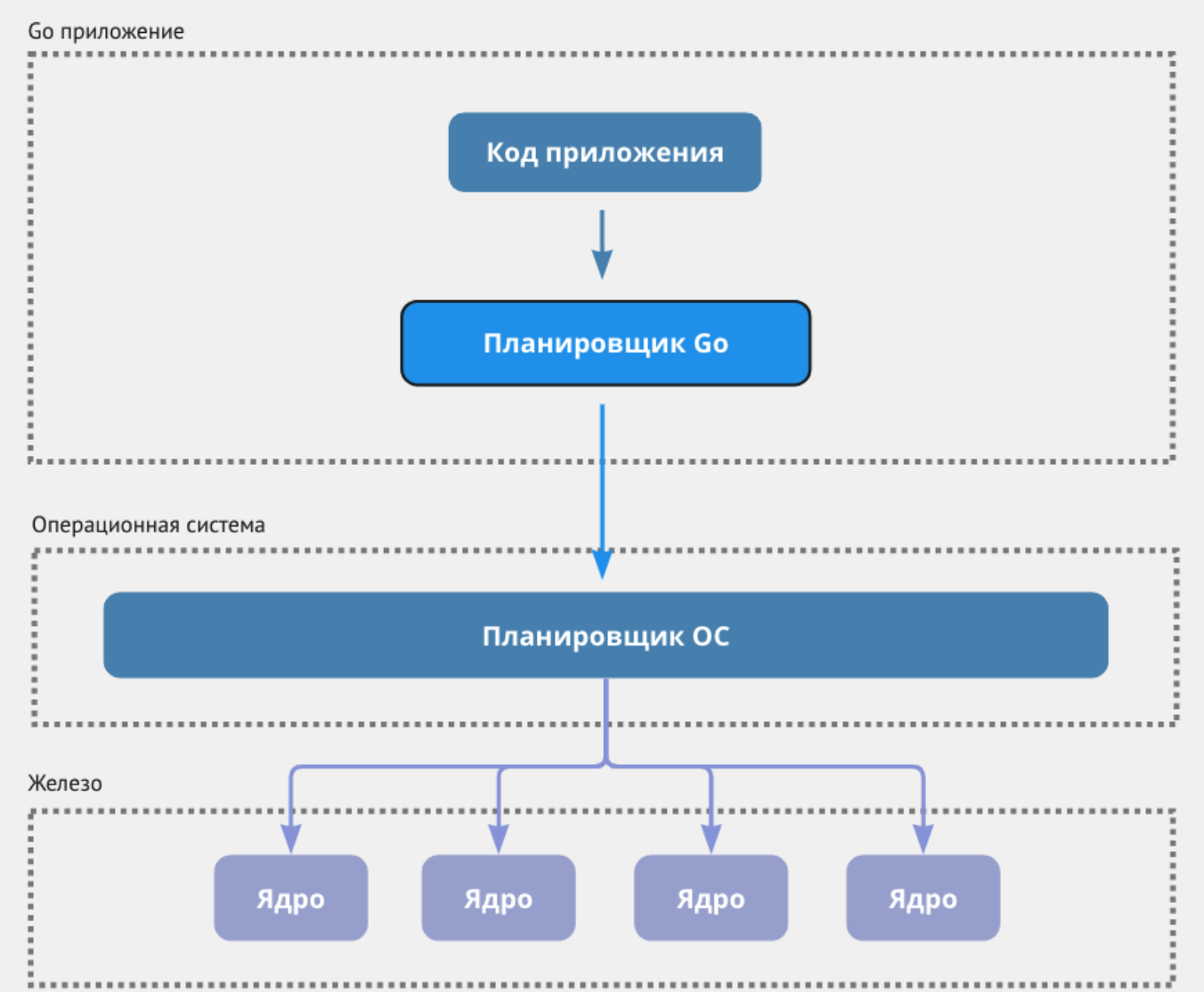
СБОРЩИК МУСОРА



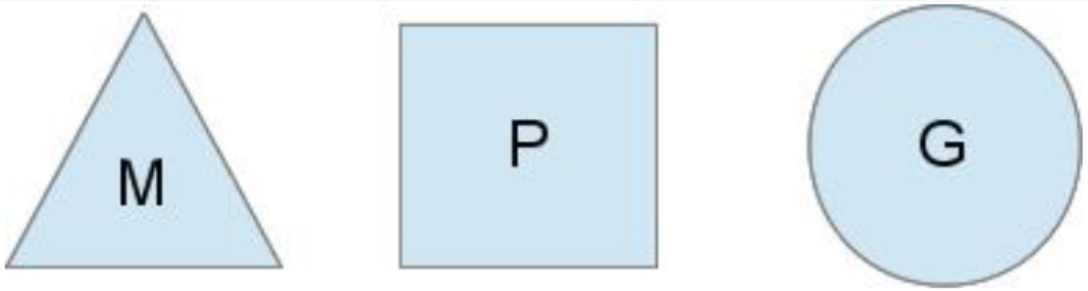
СБОРЩИК МУСОРА



ПЛАНИРОВЩИК



ПЛАНИРОВЩИК

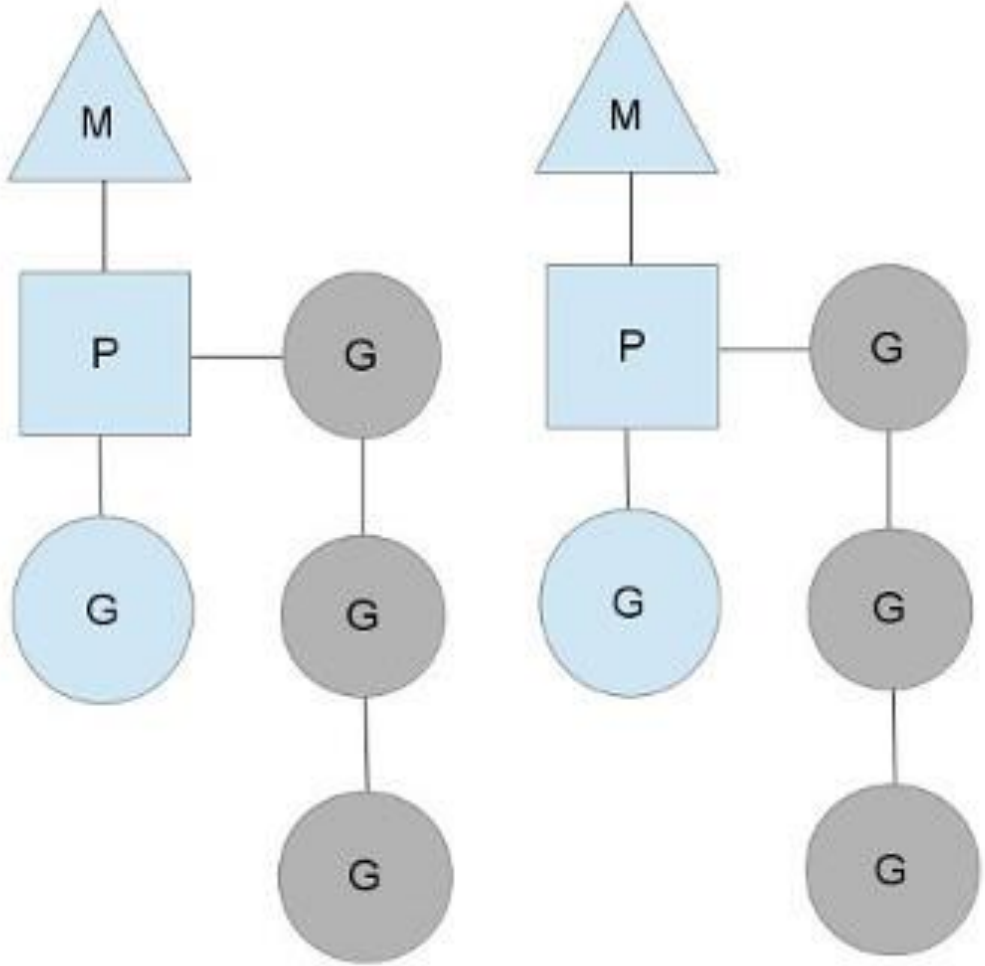


Треугольник представляет поток операционной системы. Выполнением такого потока управляет операционная система, и работает это во многом подобно вашим стандартным потокам POSIX. В исполнимом коде это называется **M машиной**.

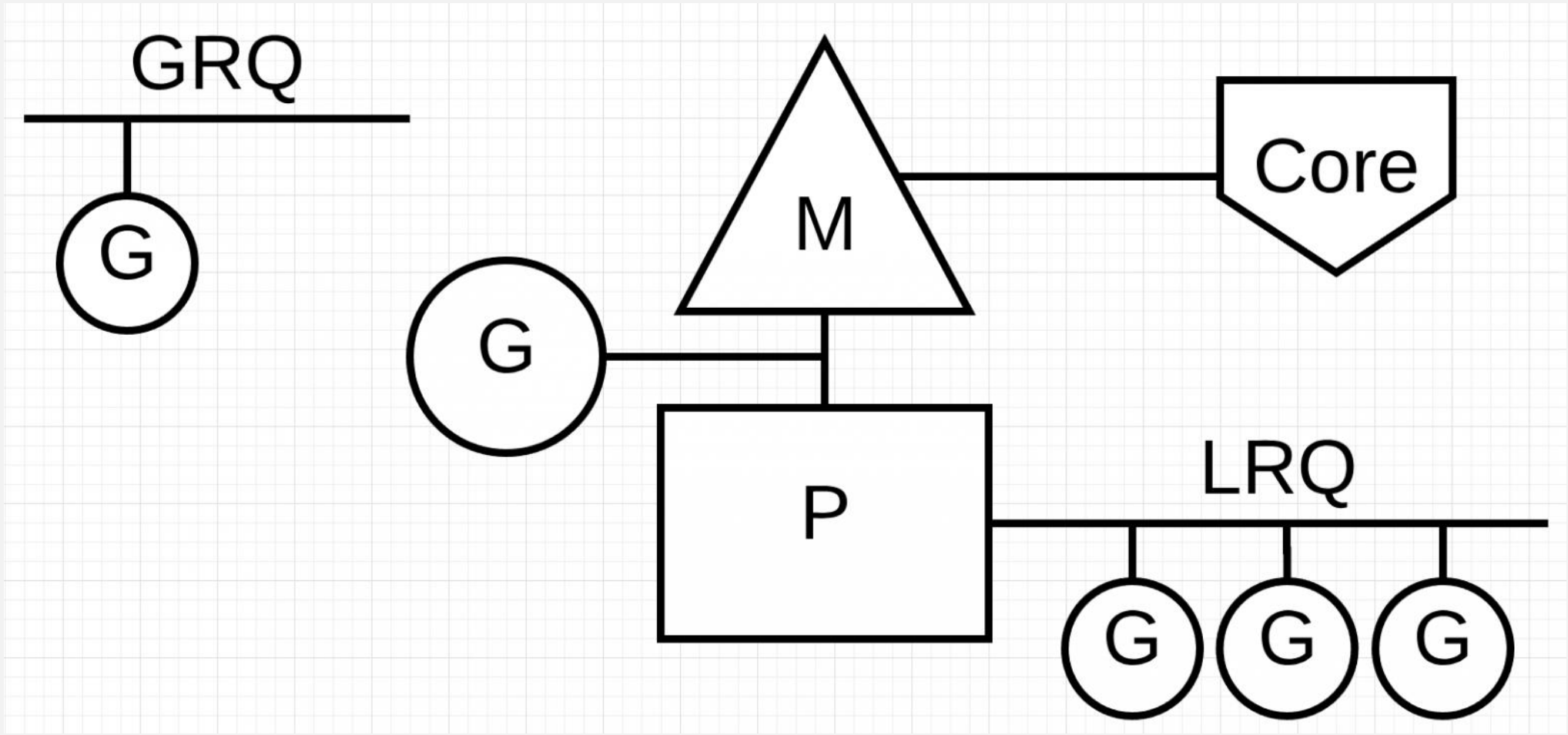
Круг представляет Go-рутину. Он включает стек, указатель команд и другую важную информацию для планирования Go-рутины, такую как канал, который на ней может быть блокирован. В исполнимом коде это обозначается как **G**.

Прямоугольник представляет контекст планирования. Вы можете понимать его как локализованная версию планировщика, который выполняет код Go-рутин в единственном потоке ядра. **Это важная часть, которая позволяет нам уйти от N:1 планировщика к M:N планировщику.** Во время выполнения кода контекст обозначается как **P для процессора**. В общем это и всё, если коротко.

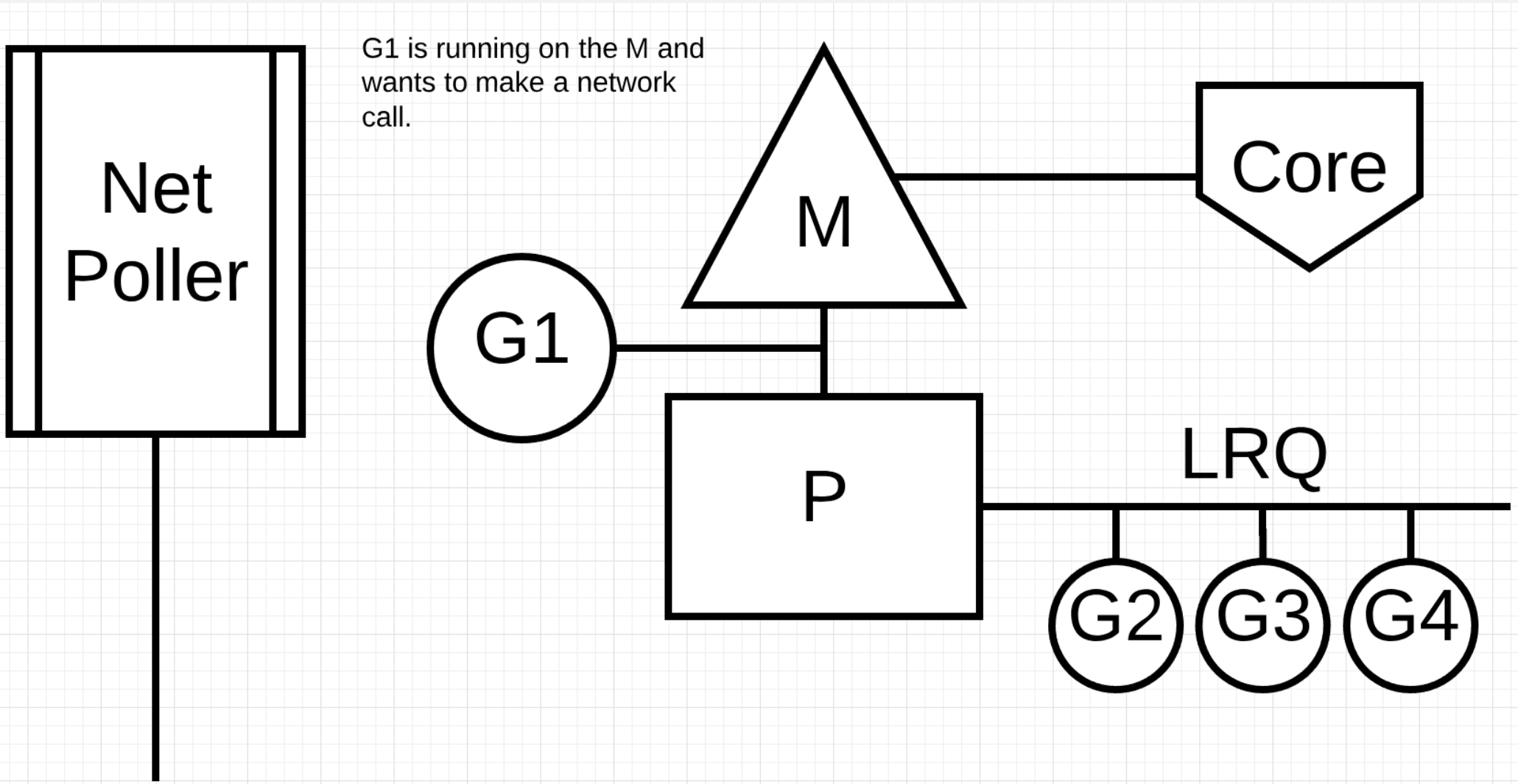
ПЛАНИРОВЩИК



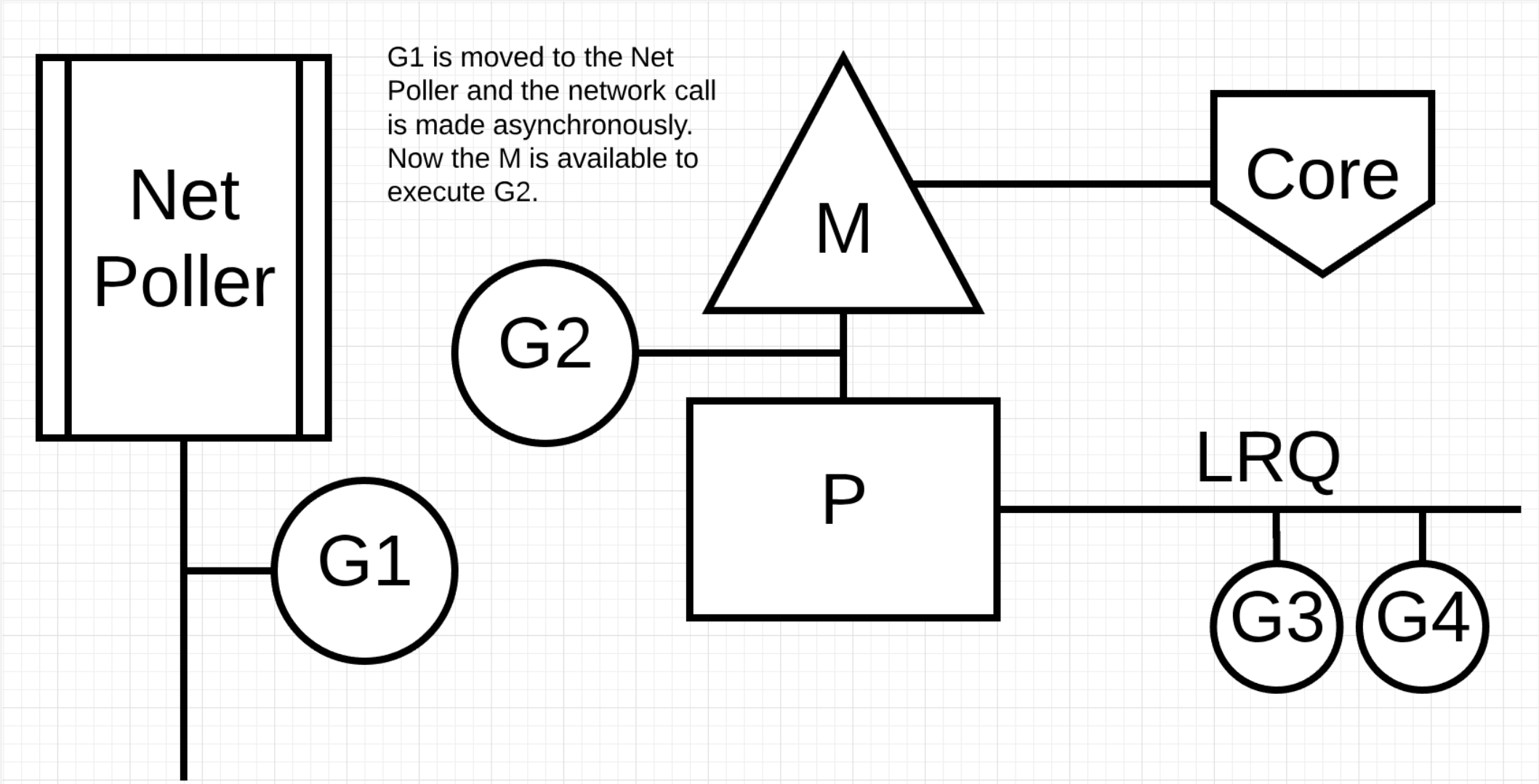
ПЛАНИРОВЩИК



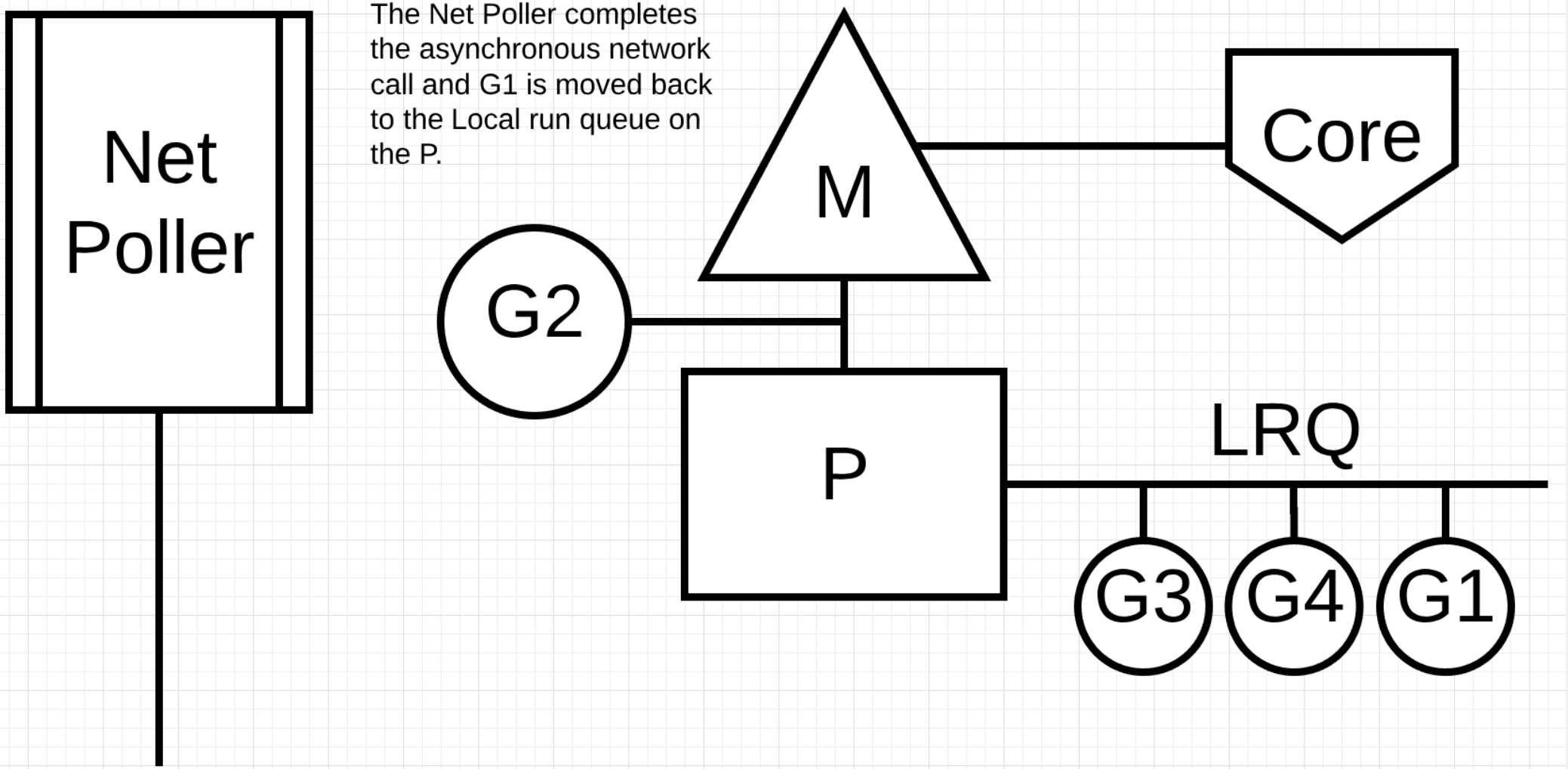
ПЛАНИРОВЩИК



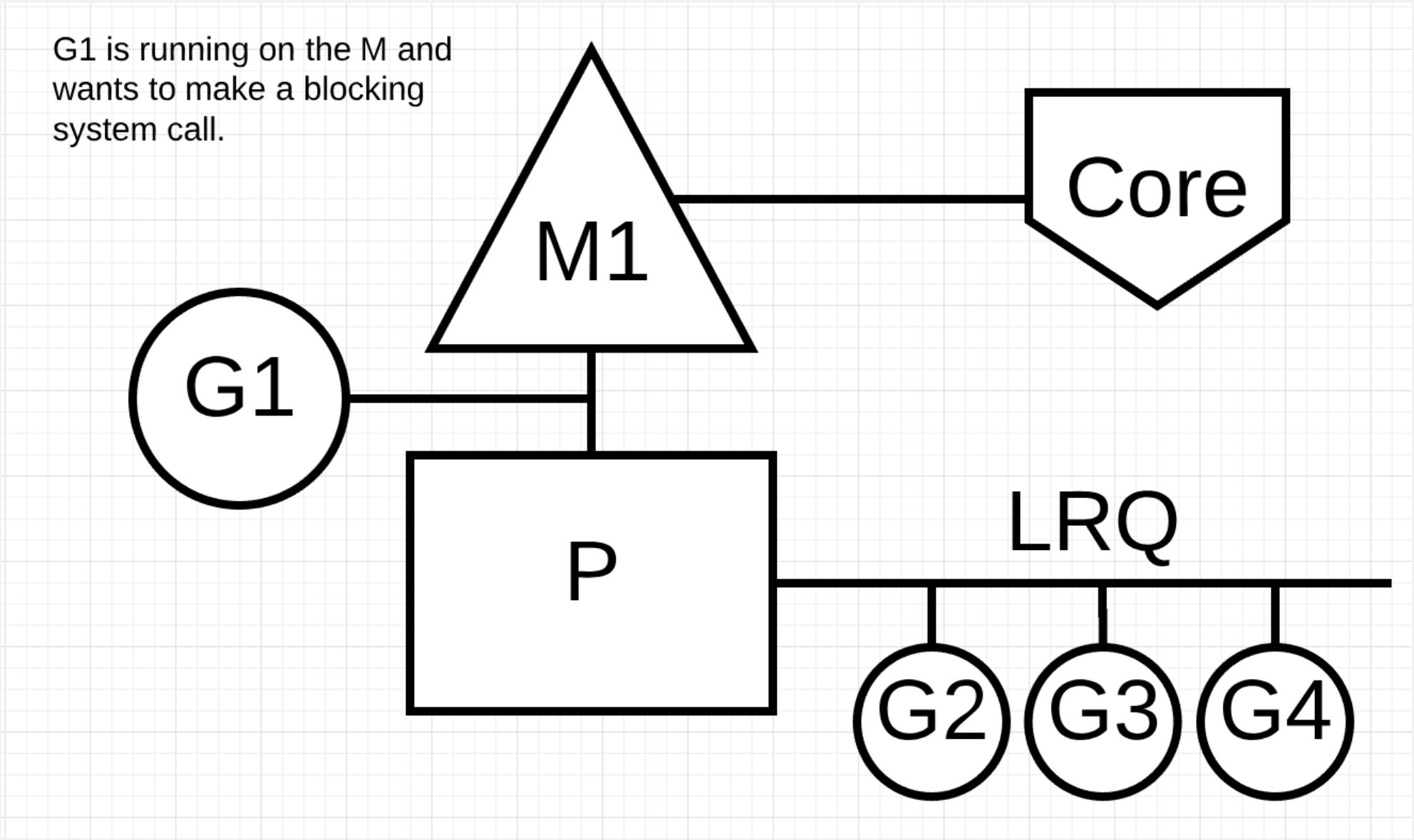
ПЛАНИРОВЩИК



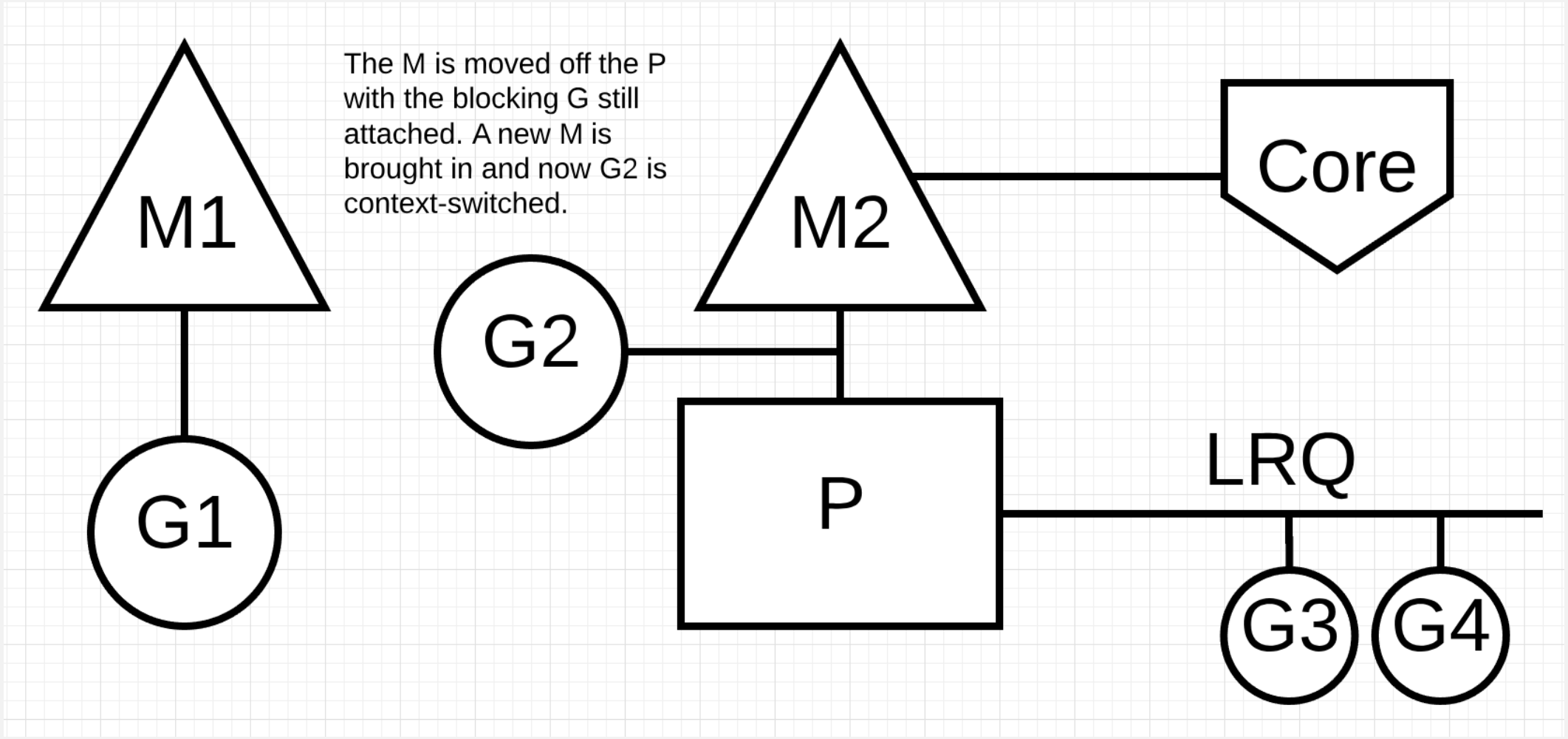
ПЛАНИРОВЩИК



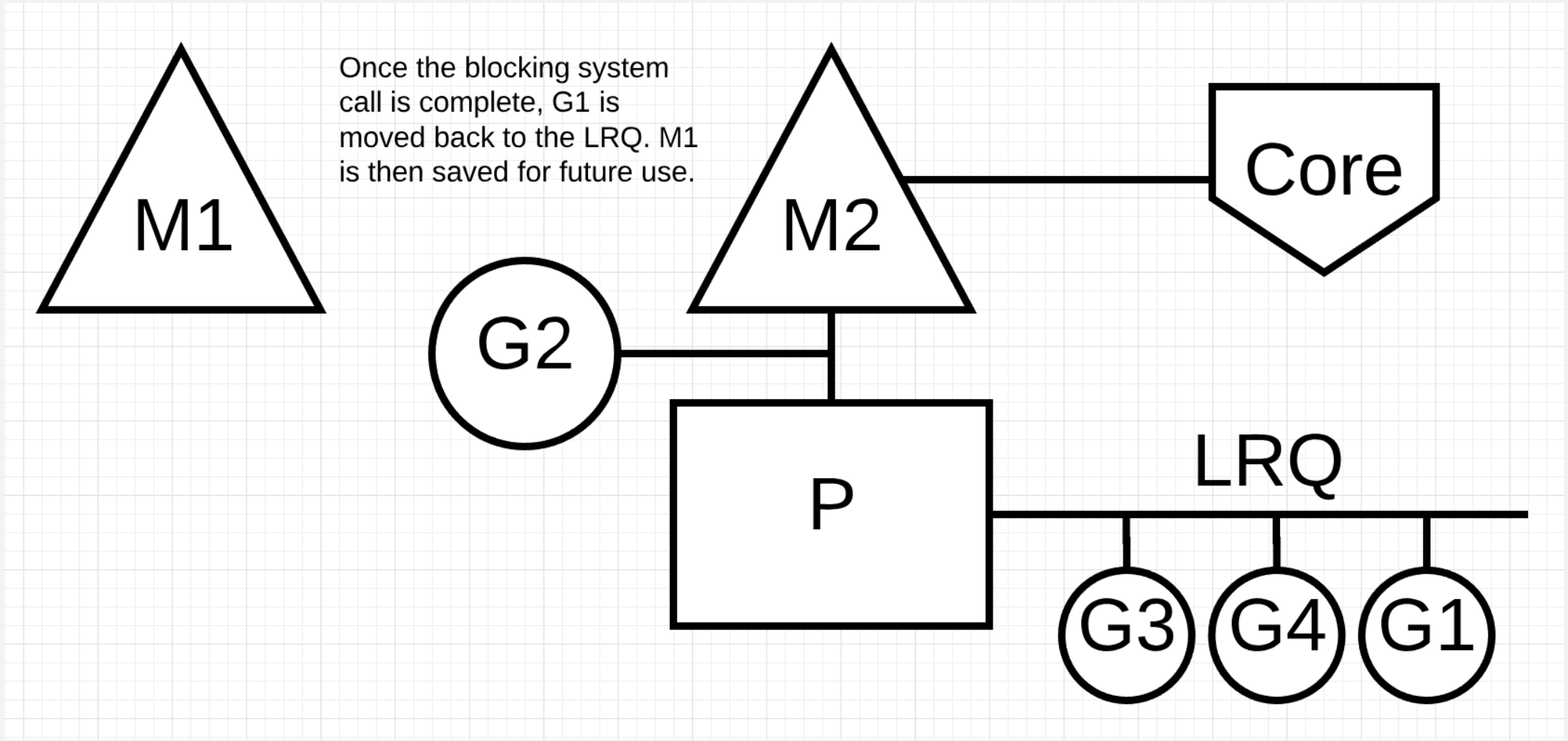
ПЛАНИРОВЩИК



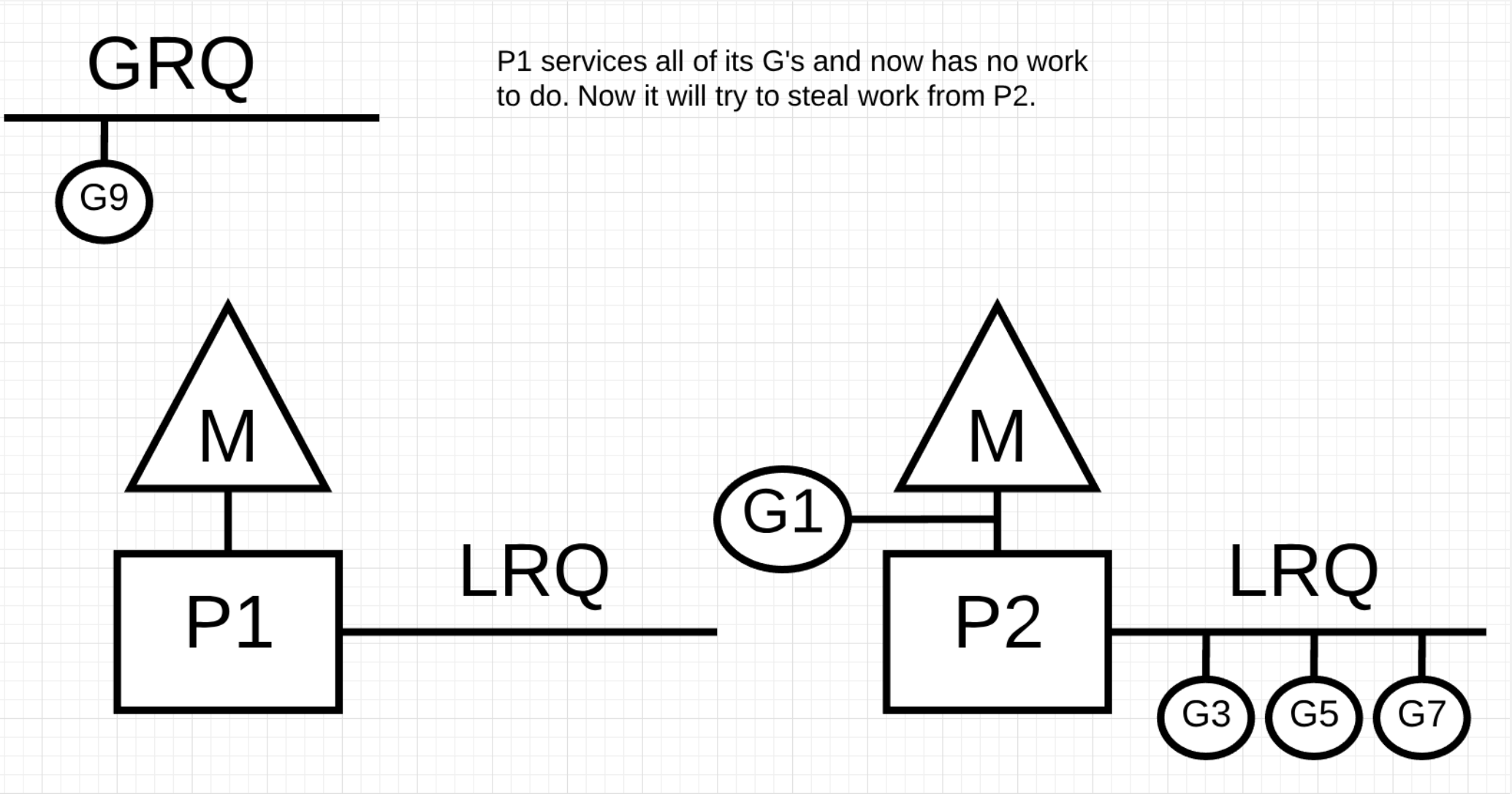
ПЛАНИРОВЩИК



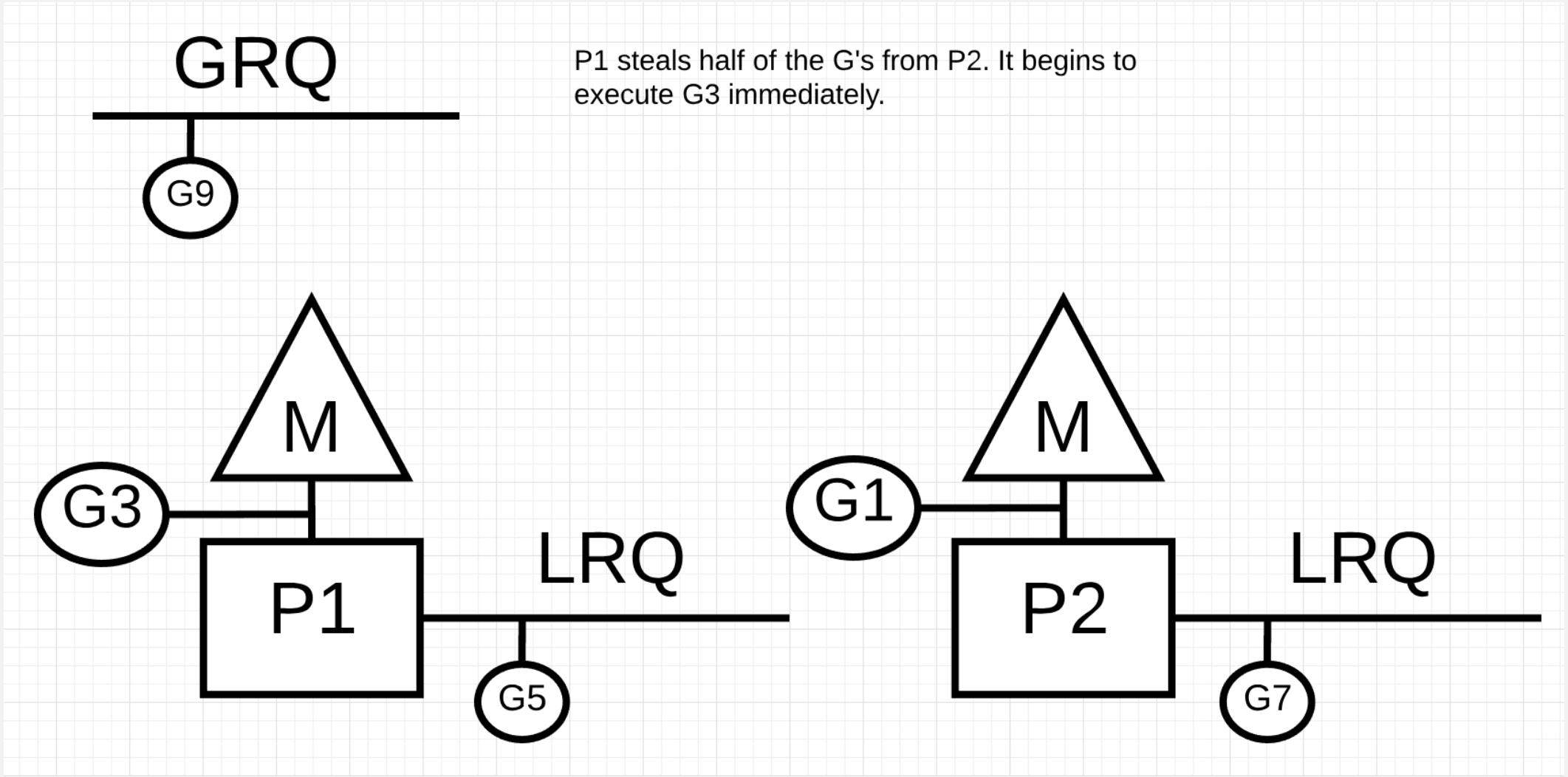
ПЛАНИРОВЩИК



ПЛАНИРОВЩИК



ПЛАНИРОВЩИК



ИСТОЧНИКИ

- Введение в слайсы
  https://go.dev/blog/slices-intro
- Хэш таблицы в Go. Детали реализации
  https://habr.com/ru/articles/457728/
- Мапы в Go: уровень Pro
  https://habr.com/ru/companies/avito/articles/774618/
- A Guide to the Go Garbage Collector
  https://go.dev/doc/gc-guide
- Планировщик Go
  https://habr.com/ru/articles/489862/

СПАСИБО ЗА ВНИМАНИЕ :3

26.11.2024