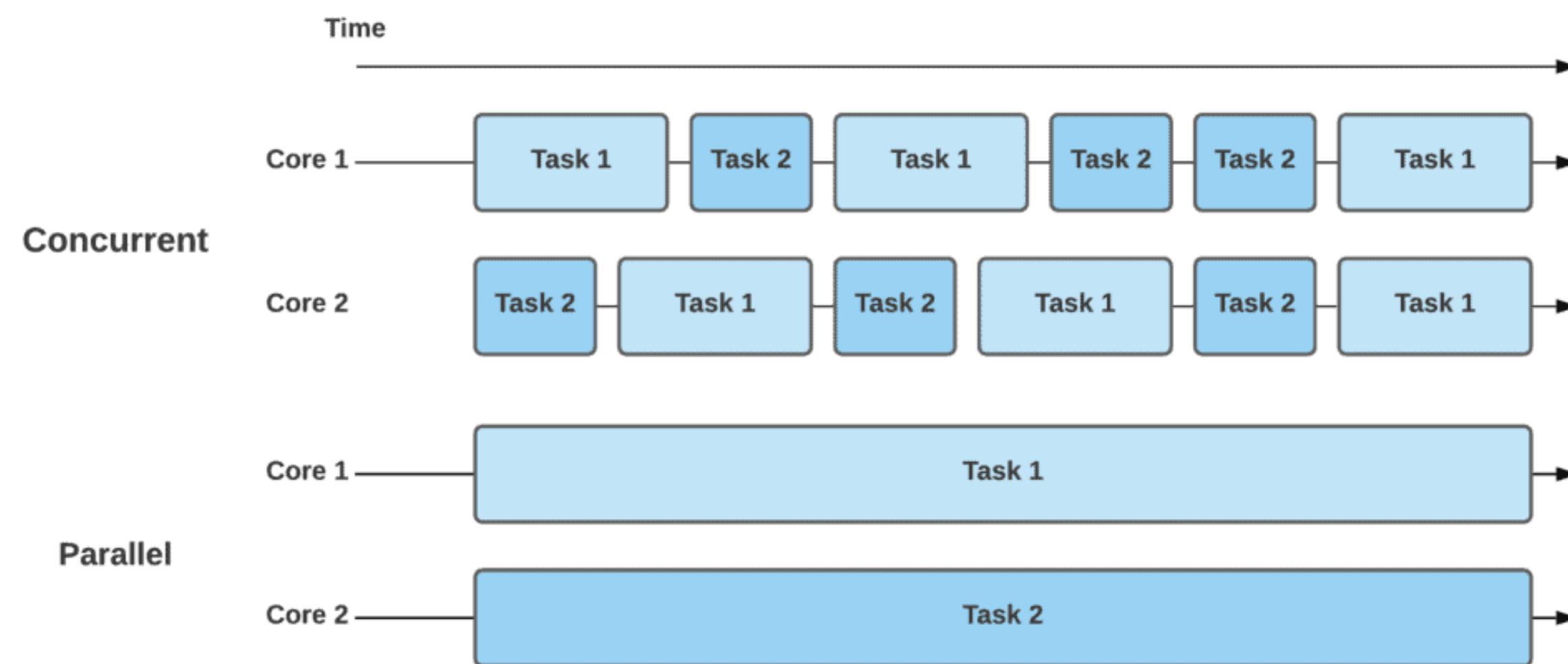


Многопоточное программирование

Конкурентность и параллелизм

Конкурентность и параллелизм

Конкурентность - выполнение нескольких задач поочерёдно.
Параллелизм - выполнение нескольких задач одновременно.



Работа с ОС

Работа с ОС

Примеры взаимодействия с ОС в Golang

- 1) *Пакеты os, syscall (запись , чтение файлов , работа с каталогами etc...)*
- 2) *Сетевые вызовы*
- 3) *Выделение ресурсов (памяти , процессорного времени)*

Абстракция от деталей ОС

Golang, как язык высокого уровня, скрывает сложности взаимодействия с разными ОС , предоставляя единый интерфейс.

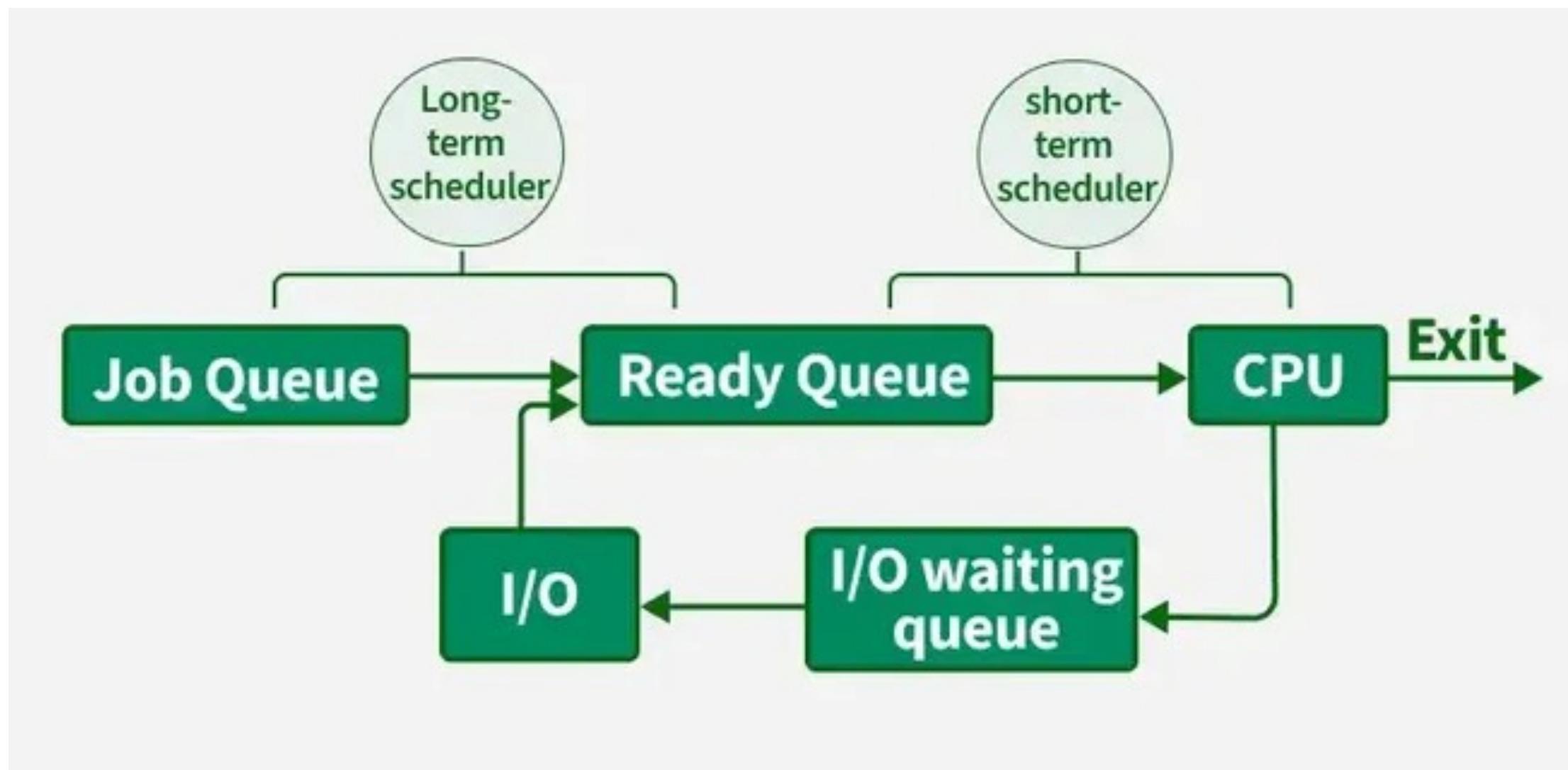
Планировщик ОС

Планировщик ОС

В ОС существует несколько планировщиков. Основные из них:

Долговременный планировщик - решает, какие задачи или процессы будут добавлены в очередь процессов, готовых к выполнению

Краткосрочный планировщик - решает, какие из готовых и загруженных в память процессов будут запущены на ЦПУ



Планировщик GO

Для чего нужен планировщик

Задача планировщика - распределение вычислительных ресурсов между задачами.

Планировщики ОС распределяют потоки запущенных процессов ОС между ядрами процессора.

Планировщик Go распределяет горутины между потоками ОС.

Для чего нужны горутины

1) Проблемы потоков

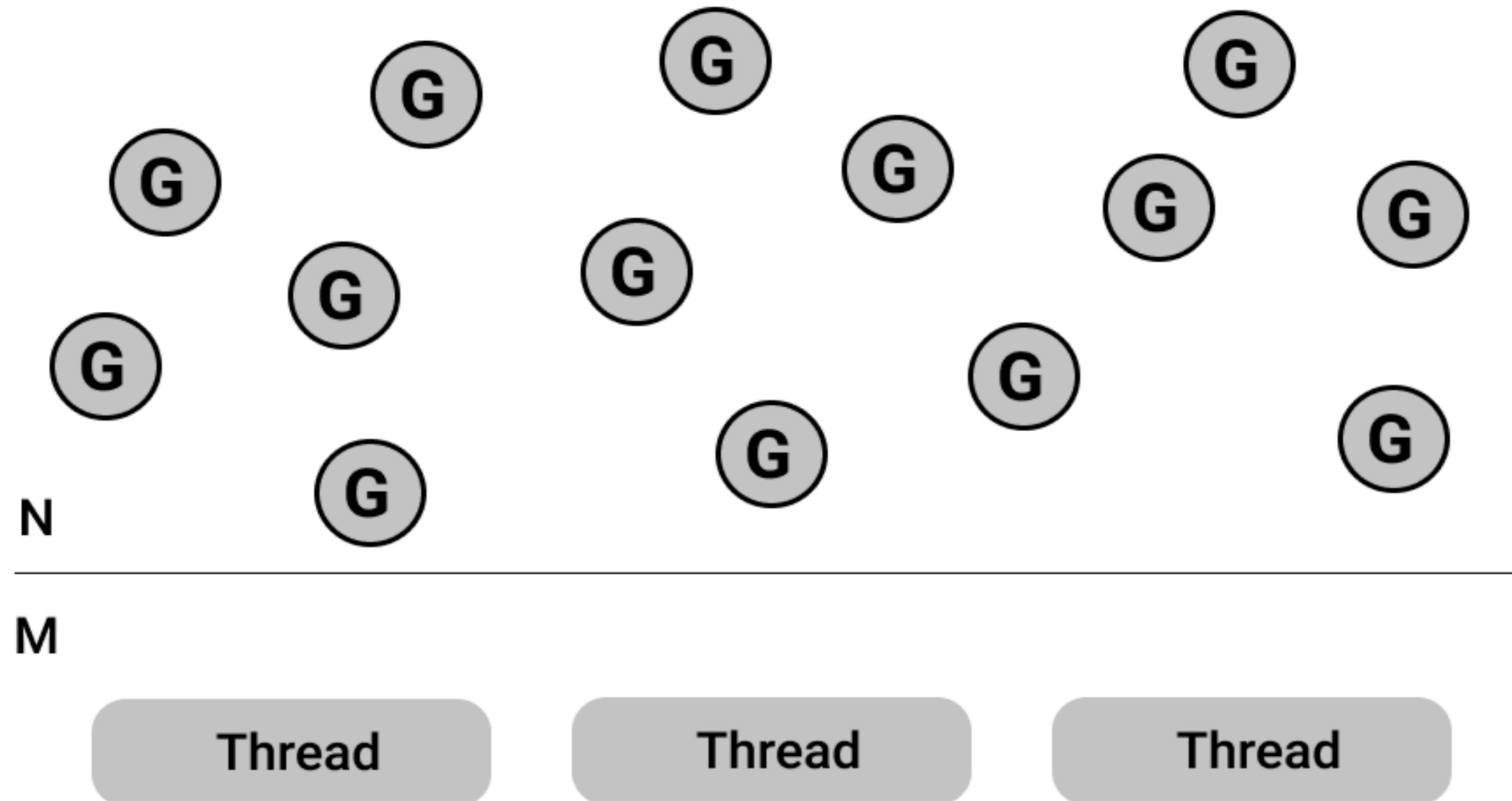
- Потоки дорогие по памяти (*каждому потоку выделяется стэк памяти, который больше стэка горутины*).
- Потоки дорого переключать (*переключение контекста*).

2) Решения в Go

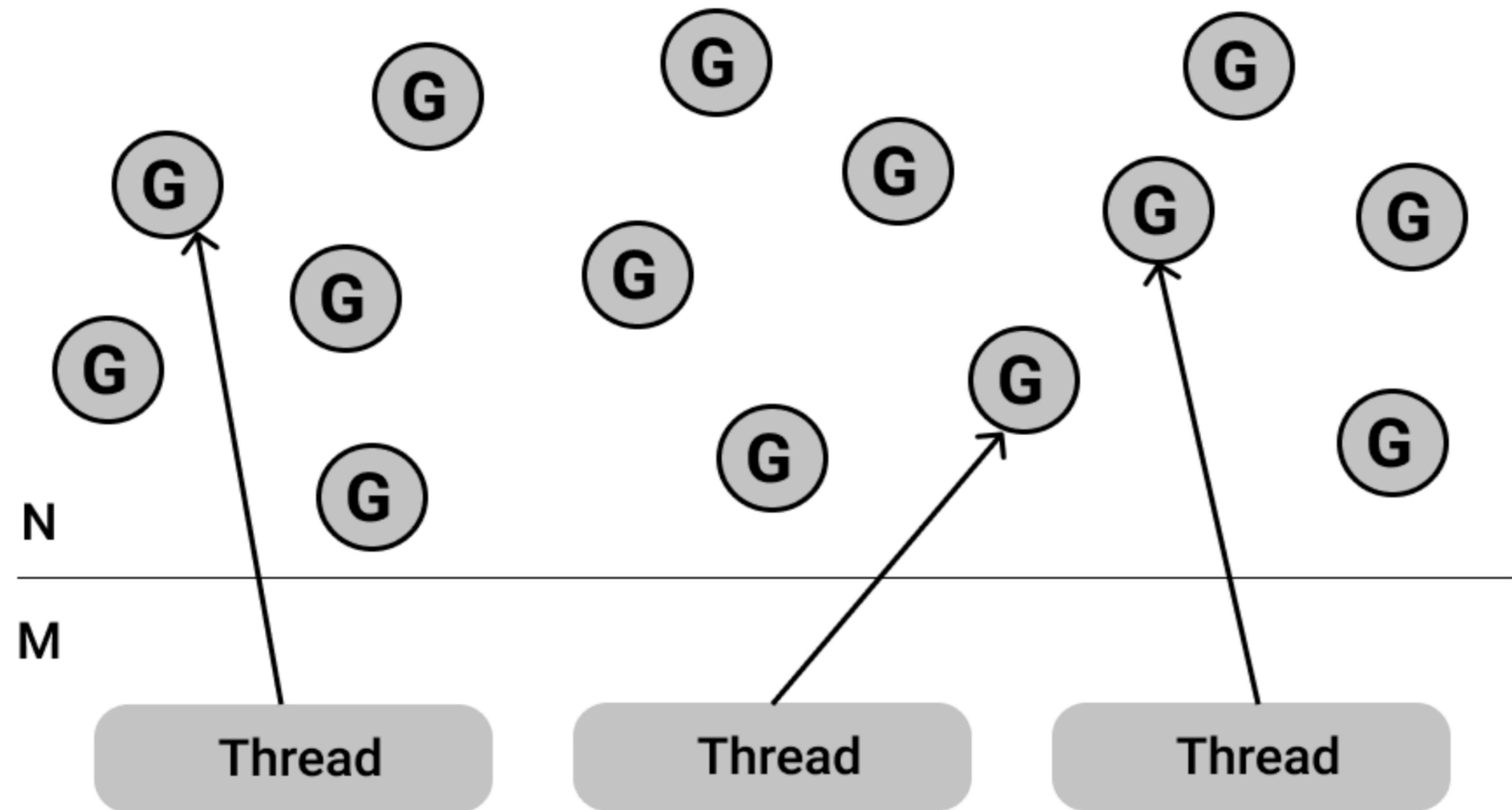
- В Go используются горутины (*легковесные потоки*). Изначальный объём памяти, выделяемый горутине, меньше изначального объёма памяти выделяемого потоку ОС. Горутины имеют *growable stack*, предел ограничен объёмом операционной памяти ОС.
- Go выбирает моменты , когда переключение дешевое.

Проектирование планировщика

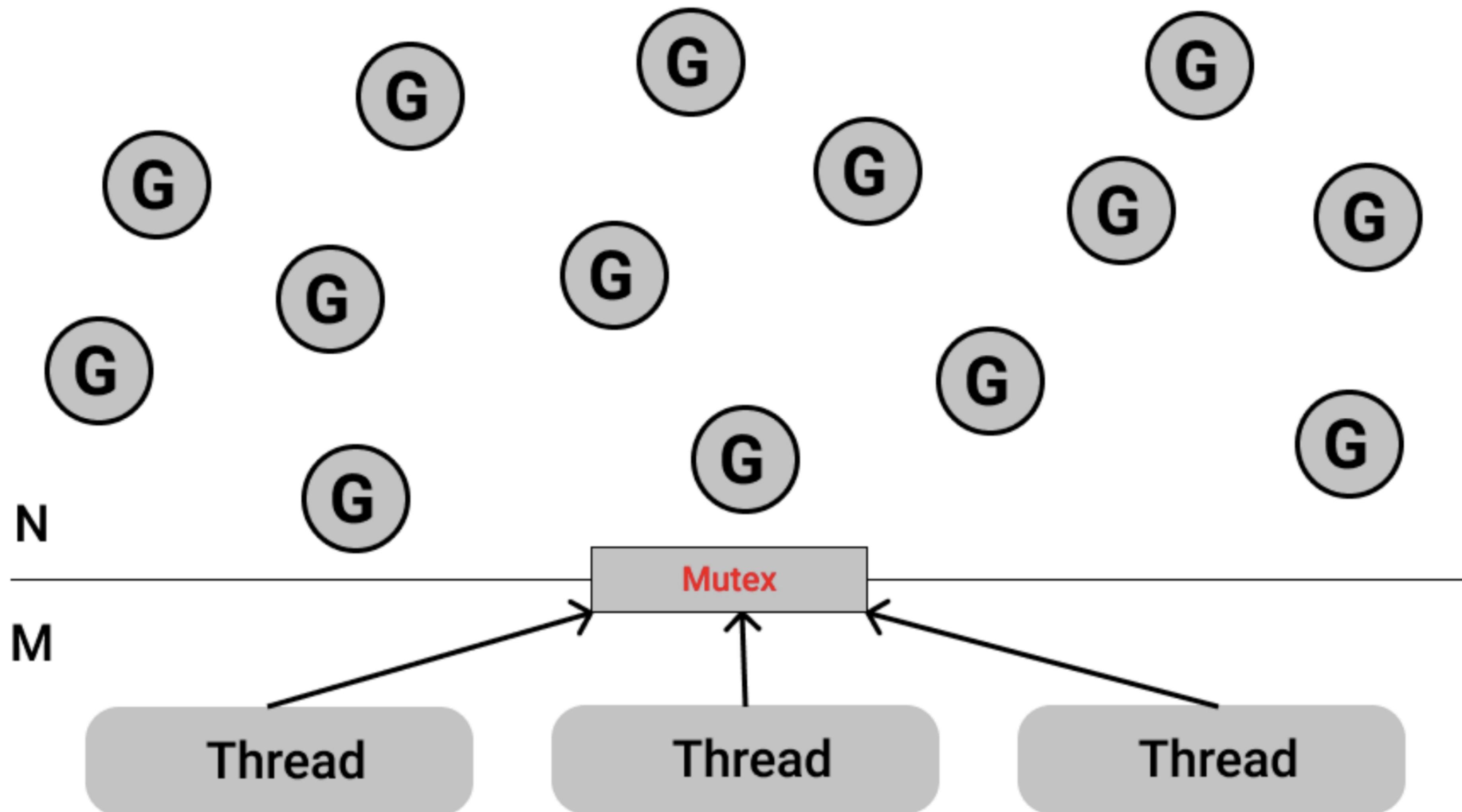
Проектирование планировщика : m-n threading



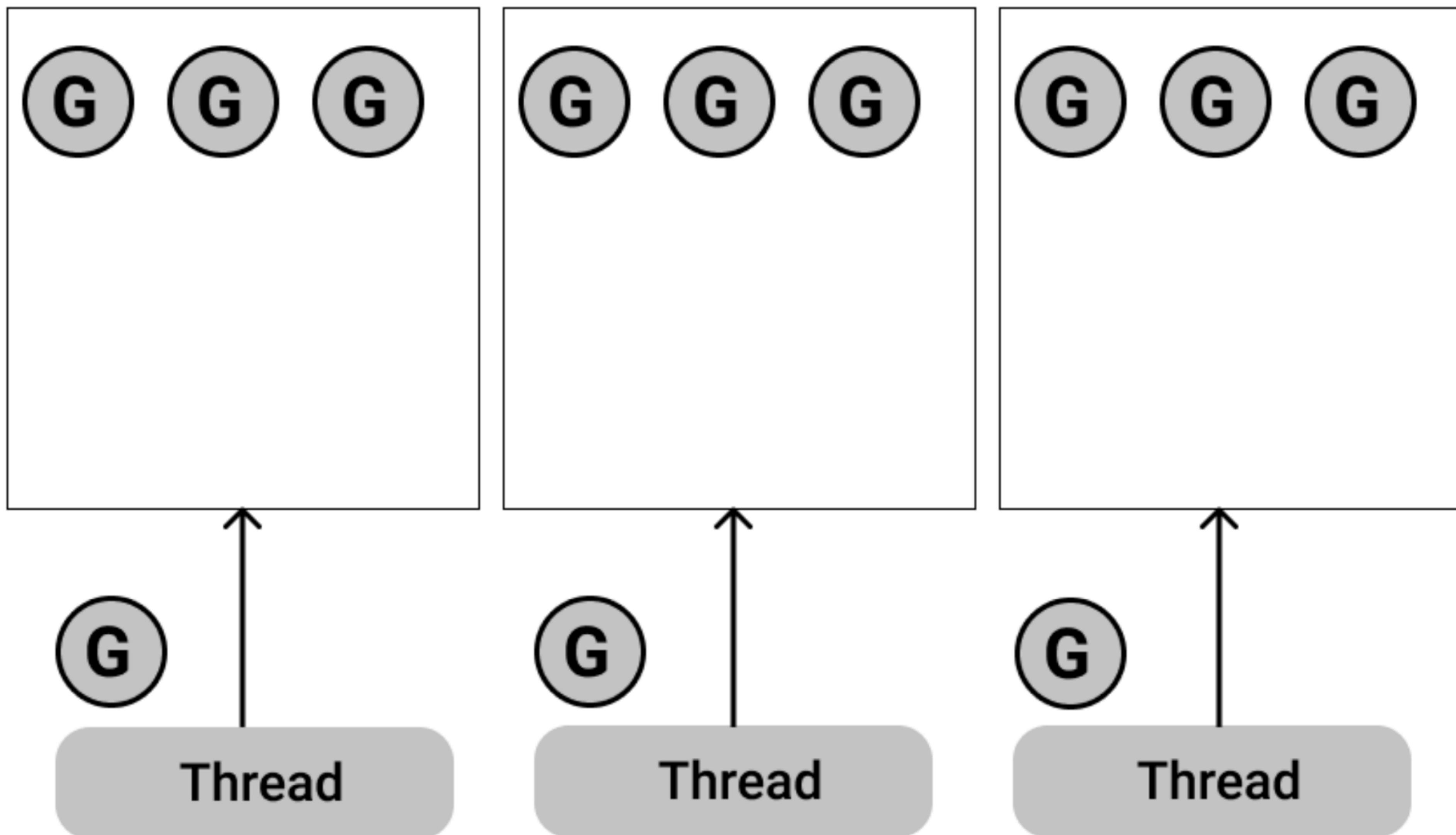
Проектирование планировщика : m-n threading



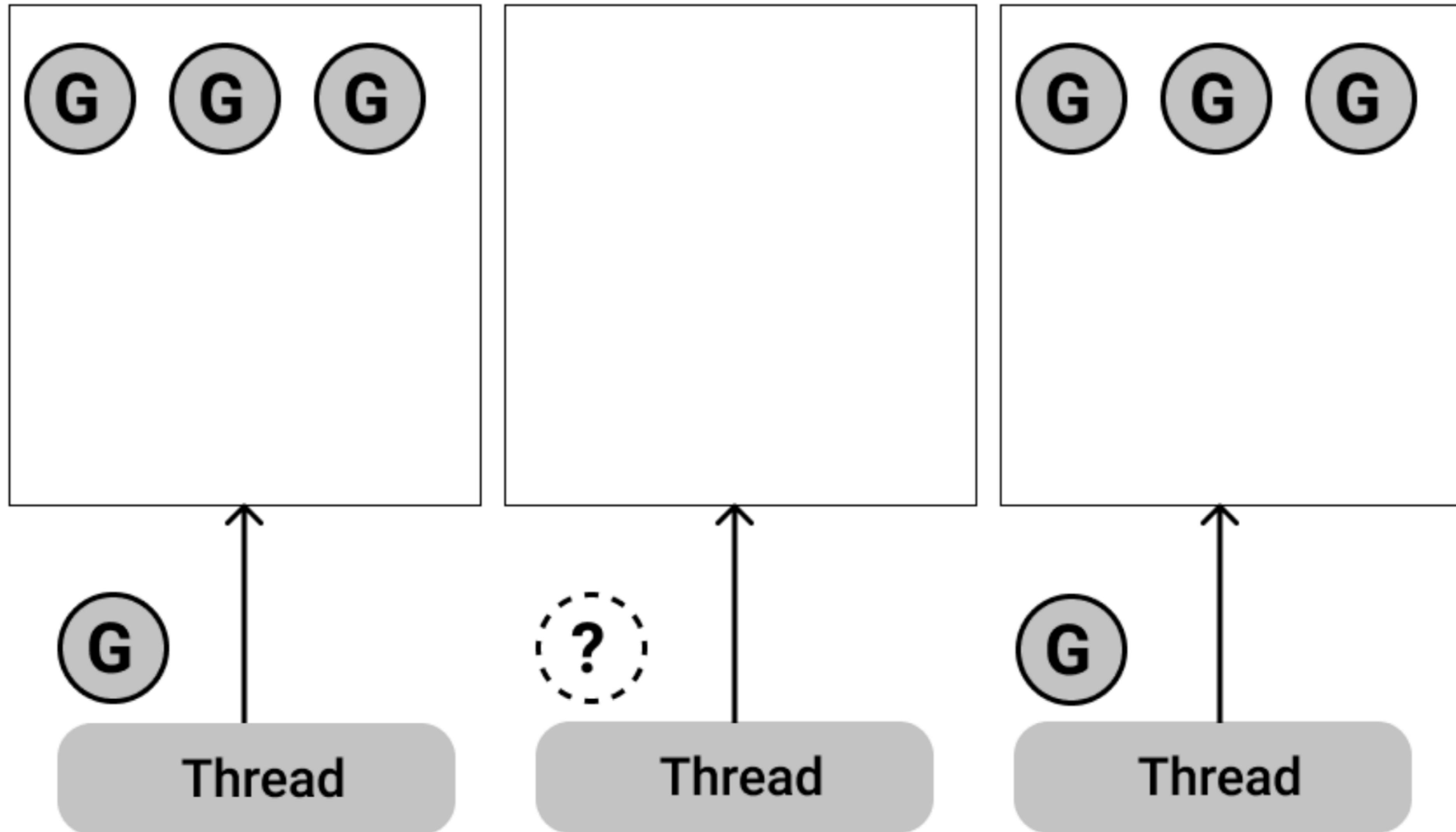
Проектирование планировщика : m-n threading



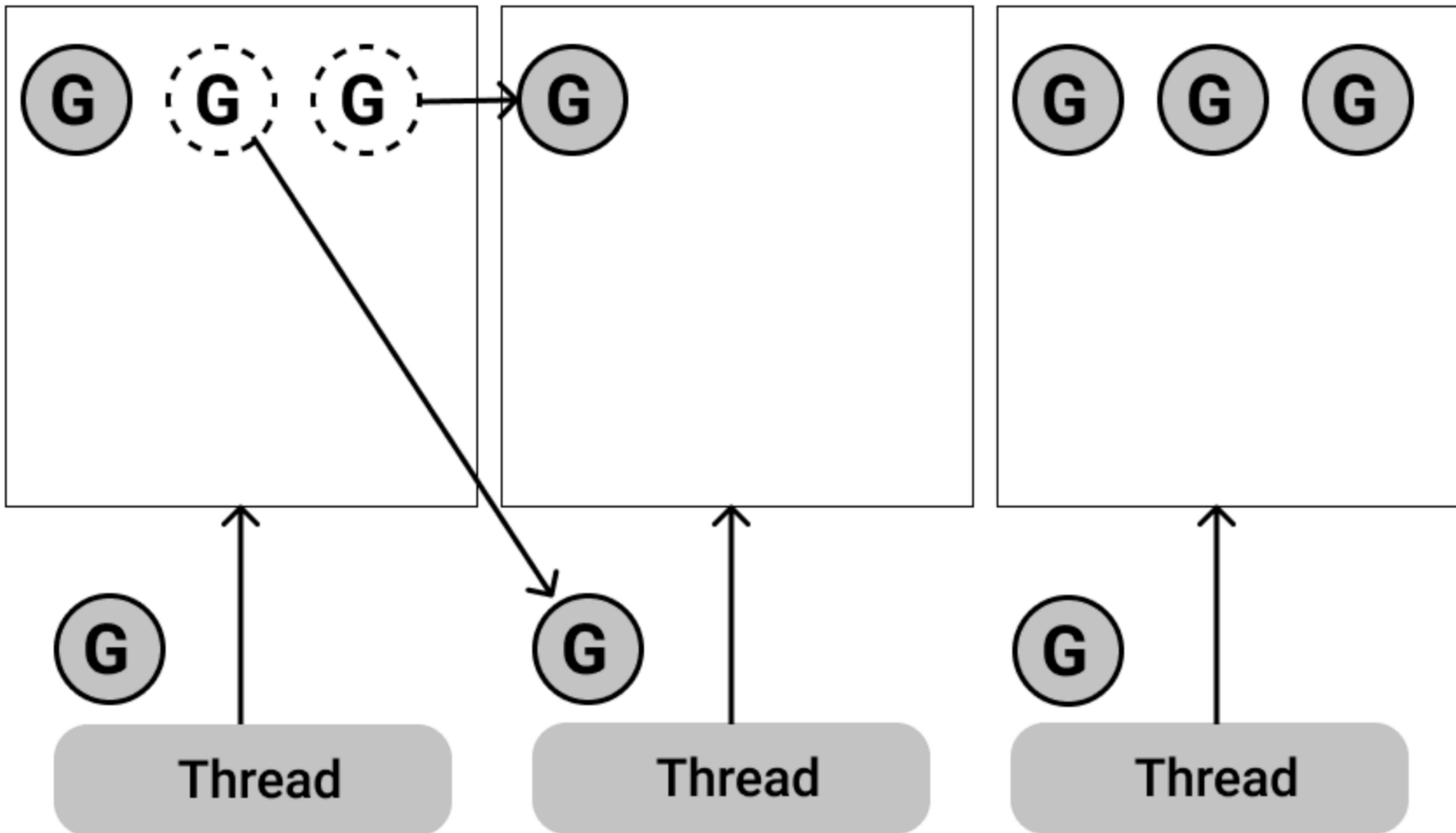
Проектирование планировщика : отдельные очереди



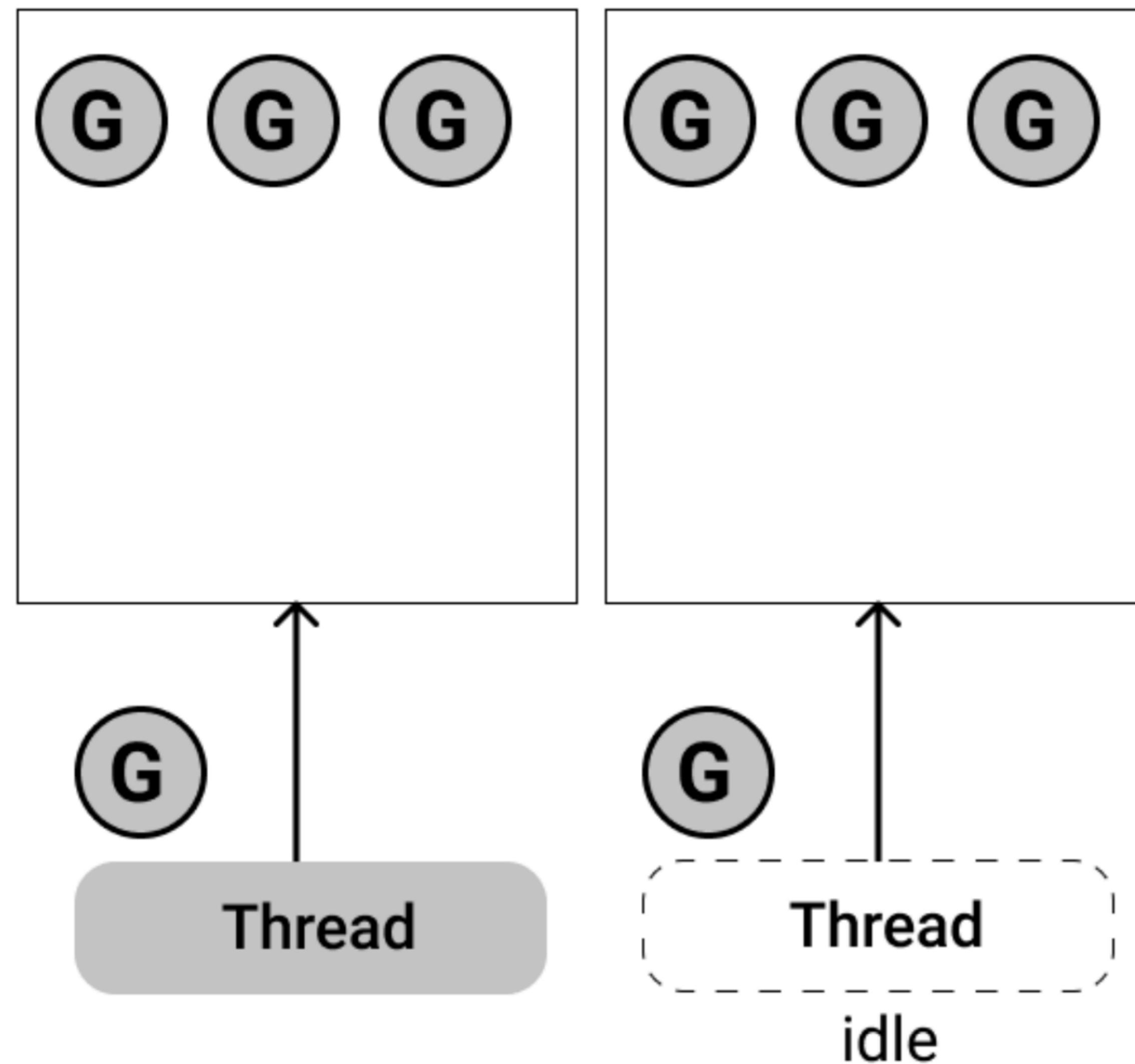
Проектирование планировщика : закончилась очередь



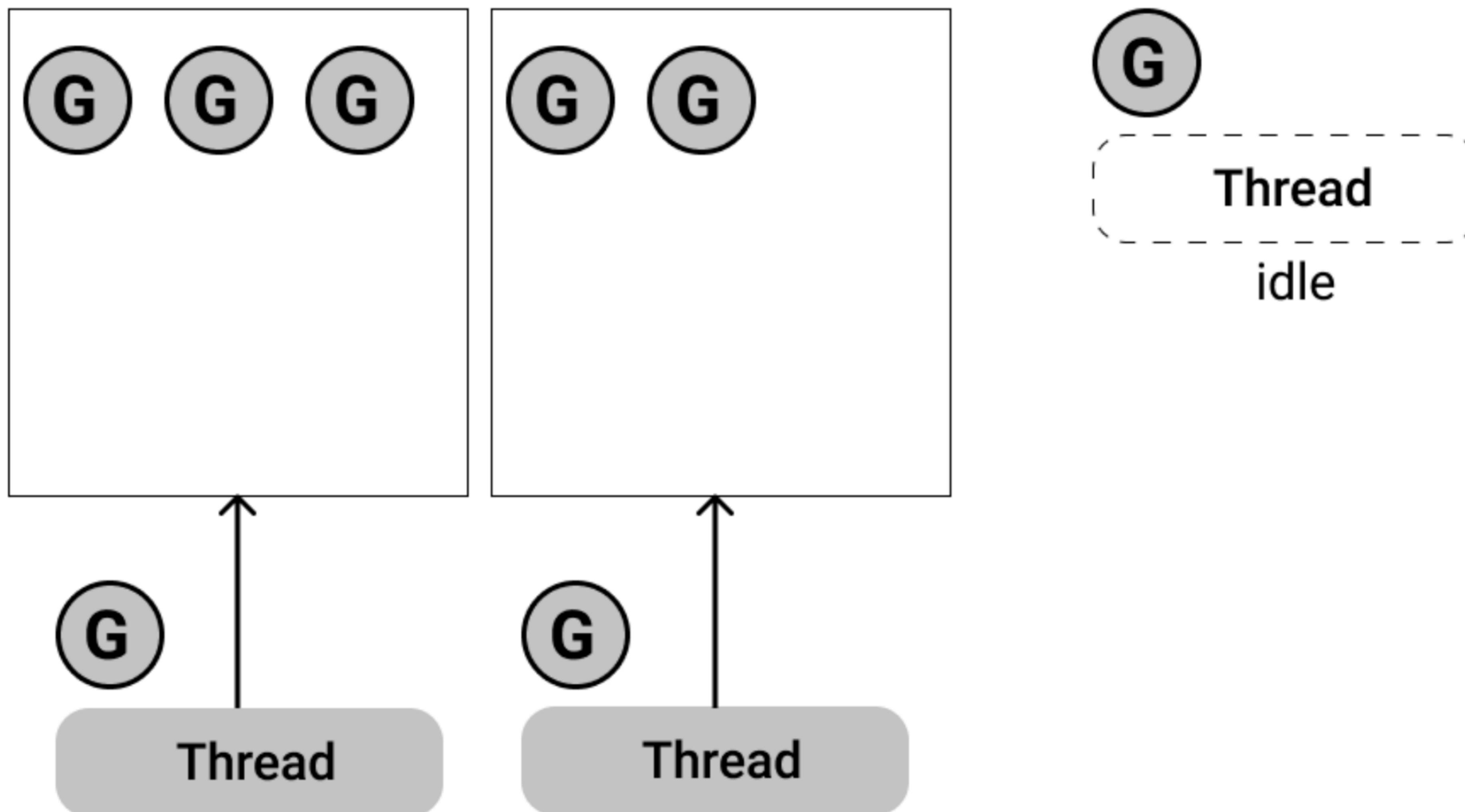
Проектирование планировщика : work stealing



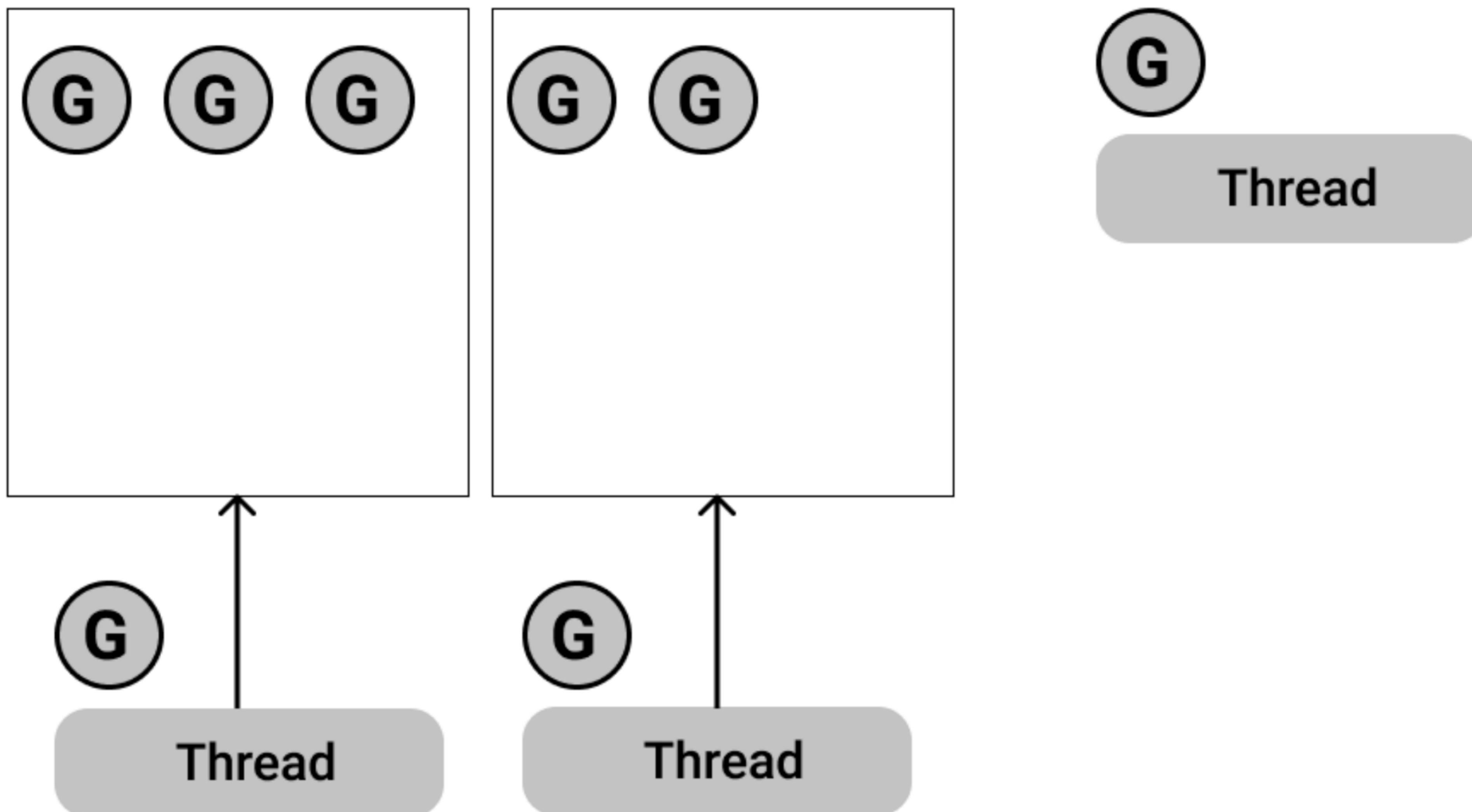
Проектирование планировщика : syscall



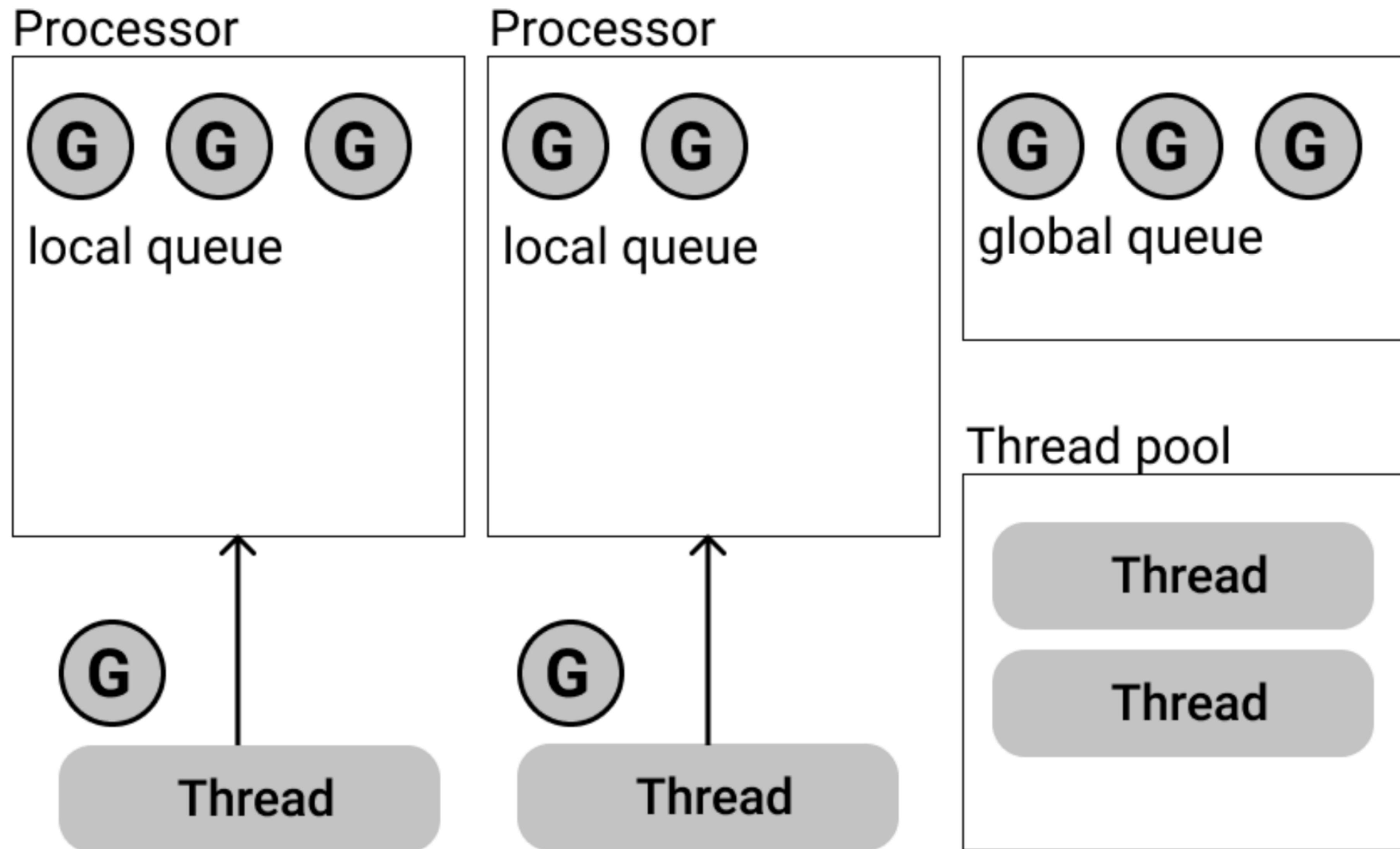
Проектирование планировщика : syscall



Проектирование планировщика : syscall



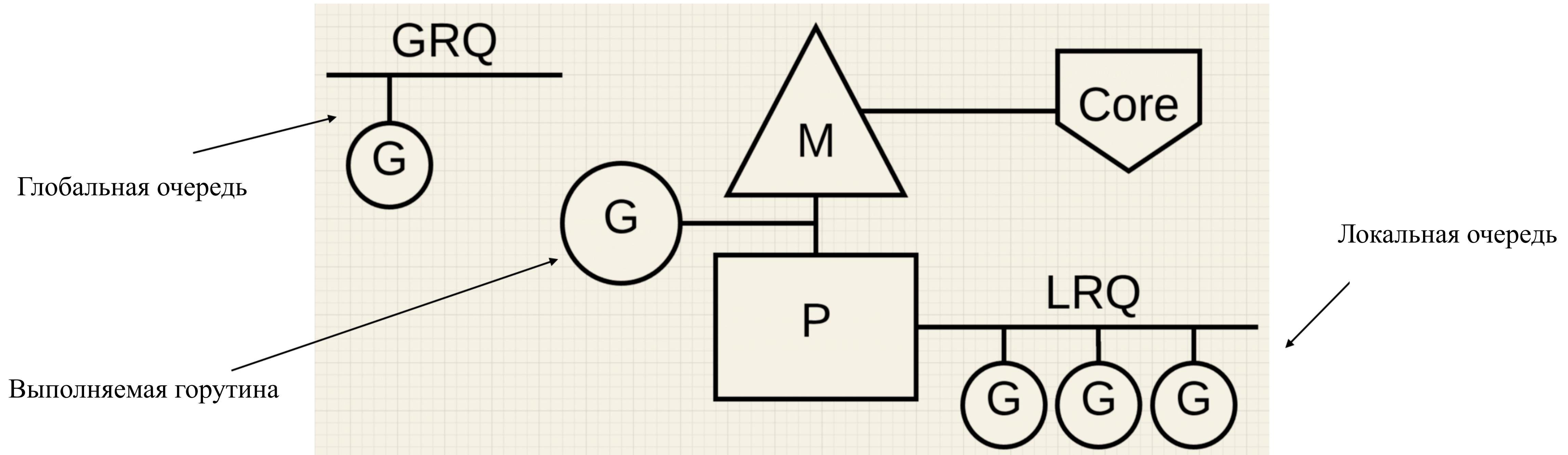
Проектирование планировщика : все идеи вместе



Планировщик Go

PMG модель

- 1) **P (processor)** – логический процессор (не железо). Это условный контекст , который объединяет поток операционной системы (M) и очередь горутин . Количество горутин , привязанных к P неограниченно . По умолчанию количество Р берётся из значения переменной среды GOMAXPROCS и равно количеству логических ядер компьютера .
- 2) **M (machine thread)** – поток OS. Он закреплён за P и имеет с ним отношение один к одному .
- 3) **G (goroutine)** – горутина



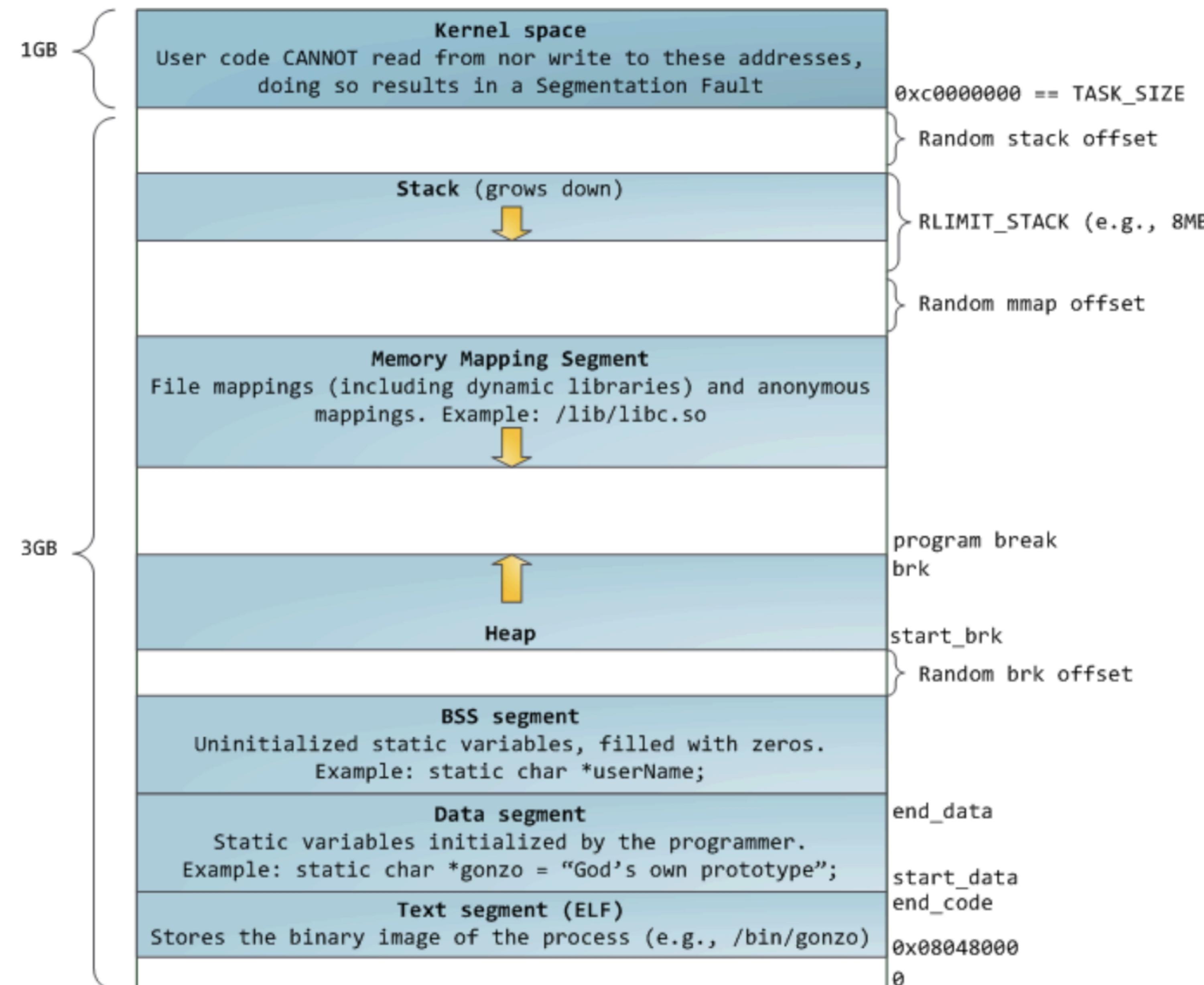
Проектирование планировщика

По какому принципу выбираем что делать?

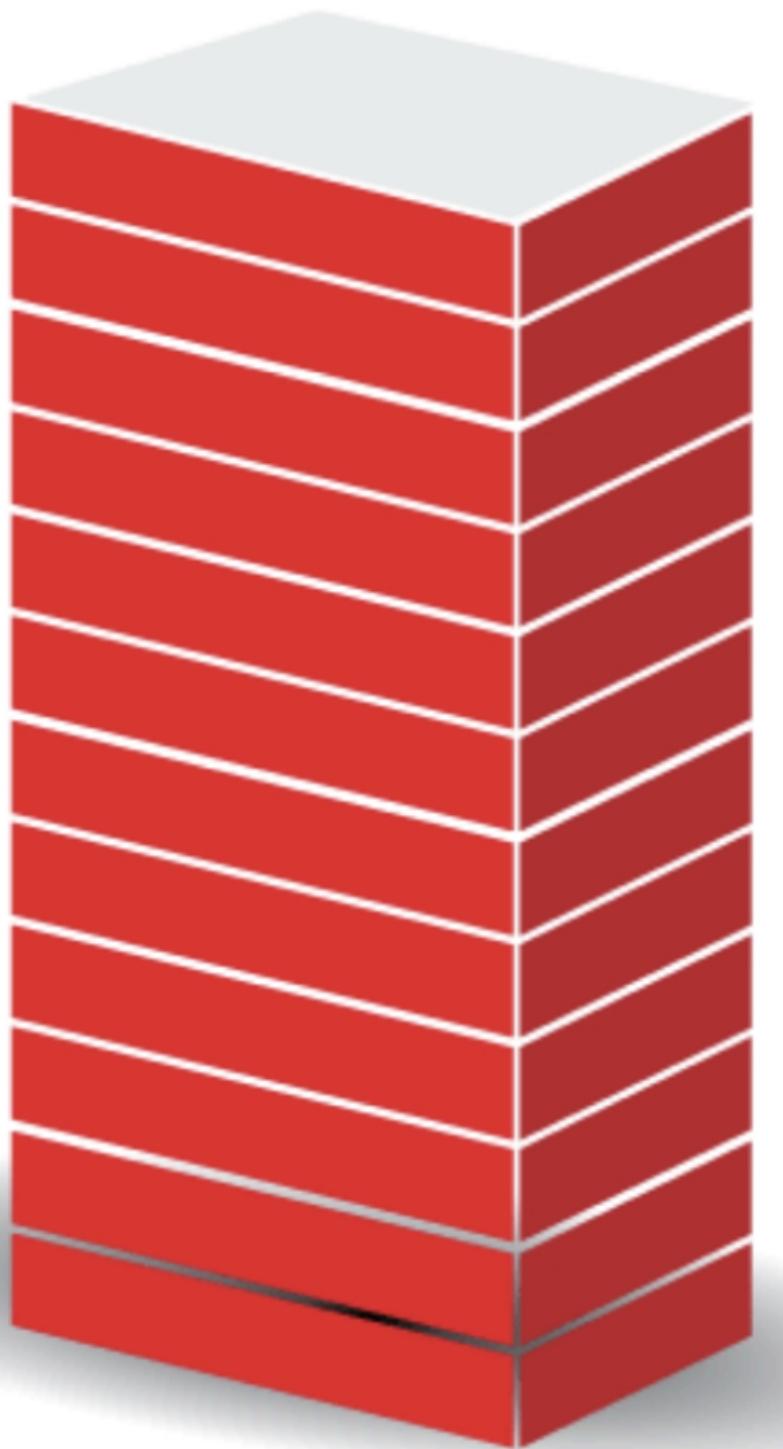
- 1) *Локальная очередь*
- 2) *Глобальная очередь*
- 3) *Work stealing*

Работа с памятью

Память



Память



Stack

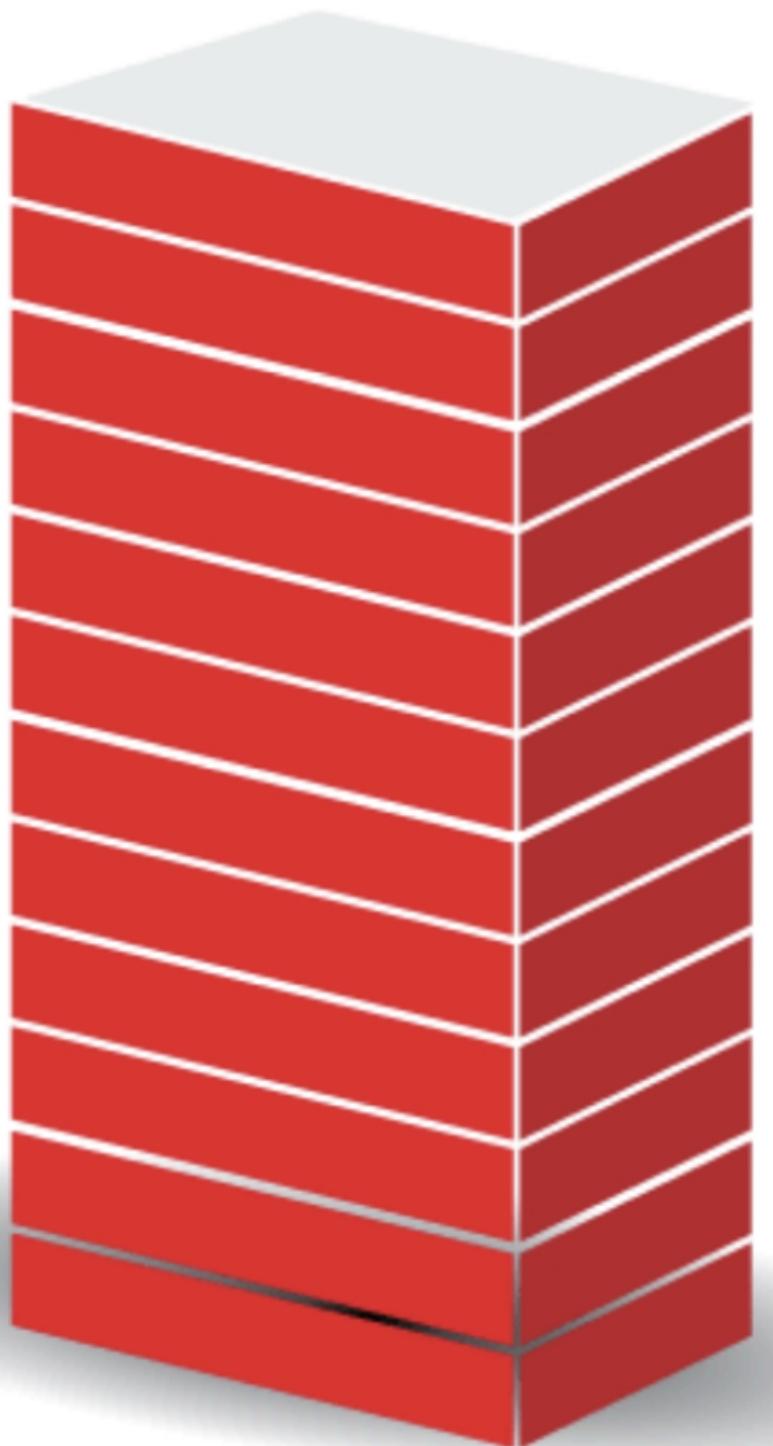
Ordered, on top of eachother!

No particular order!



Heap

Память



Stack

Ordered, on top of eachother!

No particular order!



Heap

Память

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

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

Память

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

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

func main

n = 4

n2 = 0

Память

```
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

Память

```
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

Память

```
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

Память

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

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

Память

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

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

func main

n = 4

Память

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

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

func main

→ n = 4

func square

— x = 0xc000078dab5

Память

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

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

func main

→ n = 16

func square

x = 0xc000078dab5

Память

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

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

func main

→ n = 16

func square

x = 0xc000078dab5

Память

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

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

```
func main
```

```
n = 16
```

```
func println
```

```
a = 16
```

Память

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

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

Память

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

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

```
func main
```

```
n = nil
```

Память

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

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

func main

n = nil

func answer

x = 42

Память

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

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

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

```
func answer
x = 42
```

Память

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

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

func main

n = nil

func answer

x = 42

Память

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

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

func main

n = nil

func answer

x = 42 —————

0xc000078da = 42 ←

Память: escape analysis

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

...

```
./full.go:12:2: moved to heap: x
```

...

```
./full.go:7:14: *n escapes to heap
```

...

Сборщик мусора

Сборщик мусора

1) *Зачем нужен ?*

- *Очищать кучу от не используемых переменных .*
- *Предотвращать утечки памяти .*

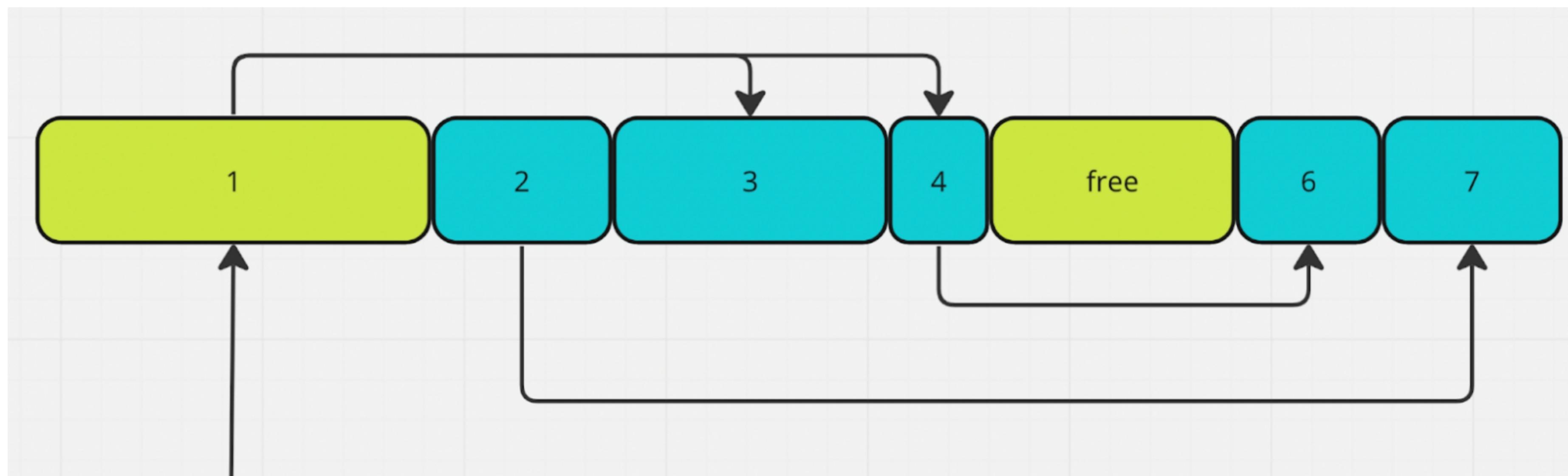
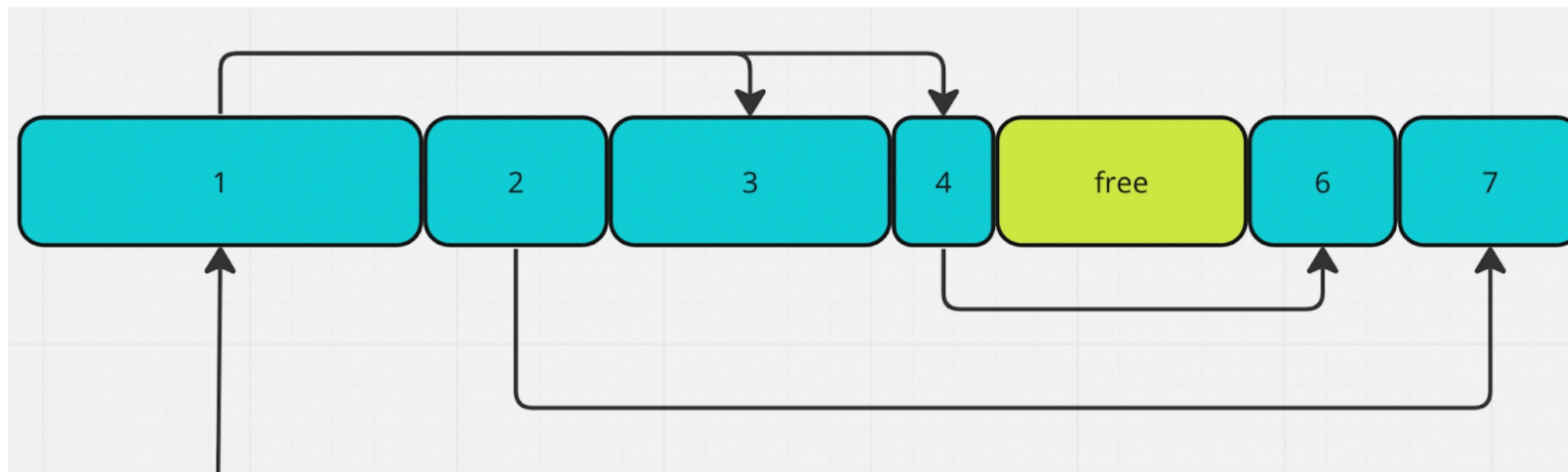
2) *Принцип работы (Mark and Sweep):*

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

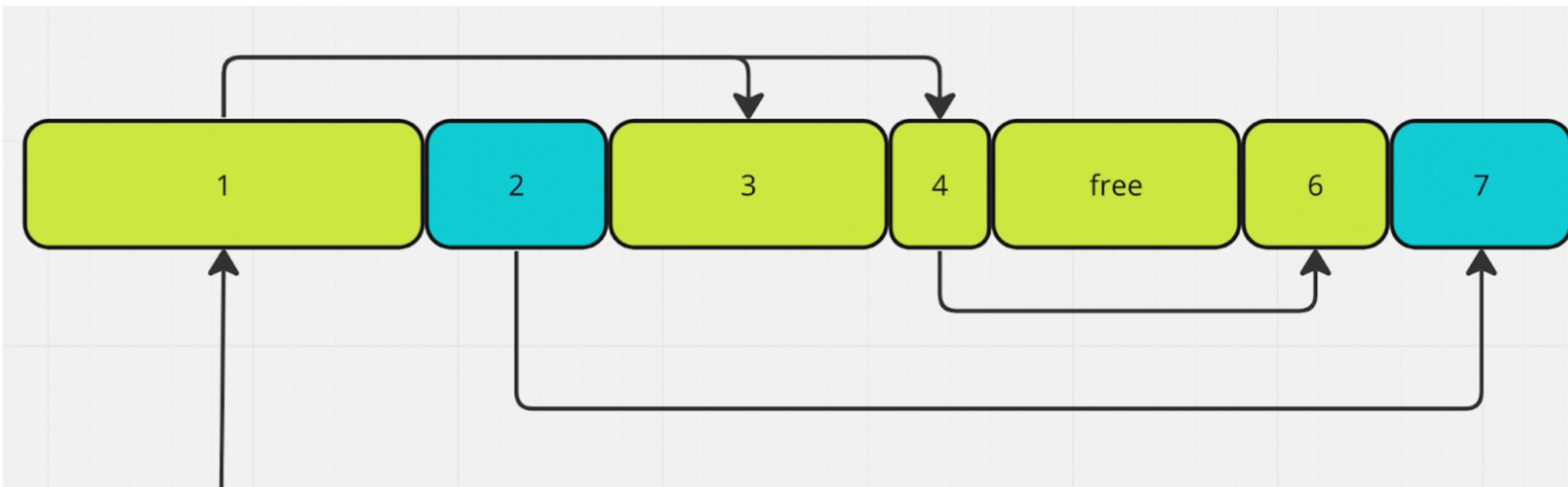
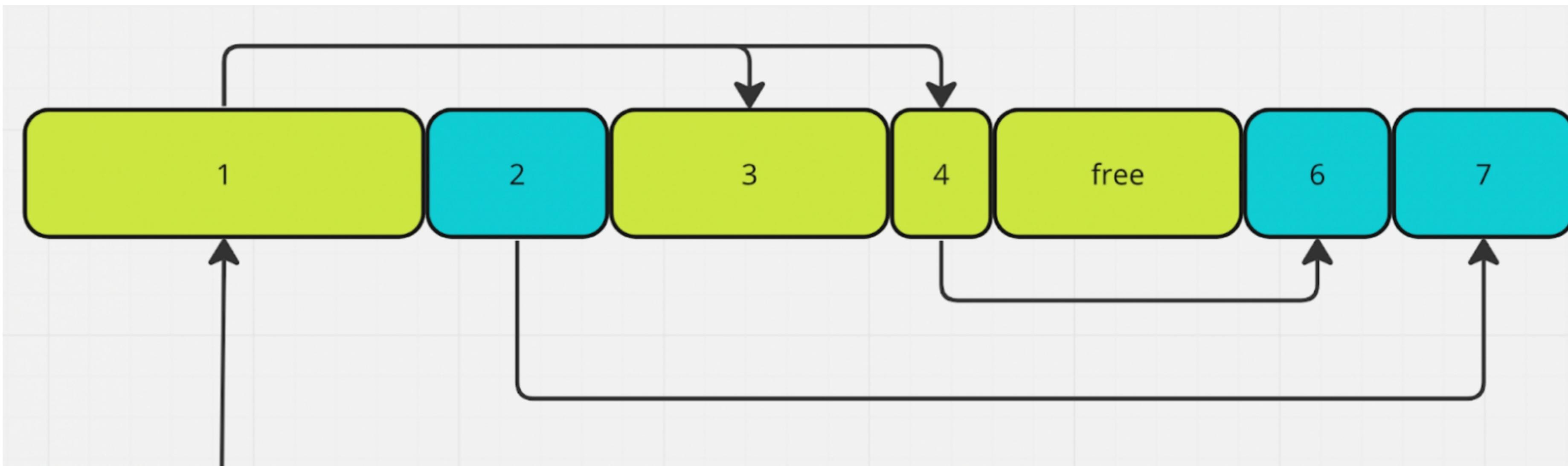
3) *Использует трехцветную маркировку*

- *Белый* — потенциальный мусор , ещё не затронутые алгоритмом объекты ;
- *Серый* — объекты « на рассмотрении »;
- *Чёрный* — активные объекты.

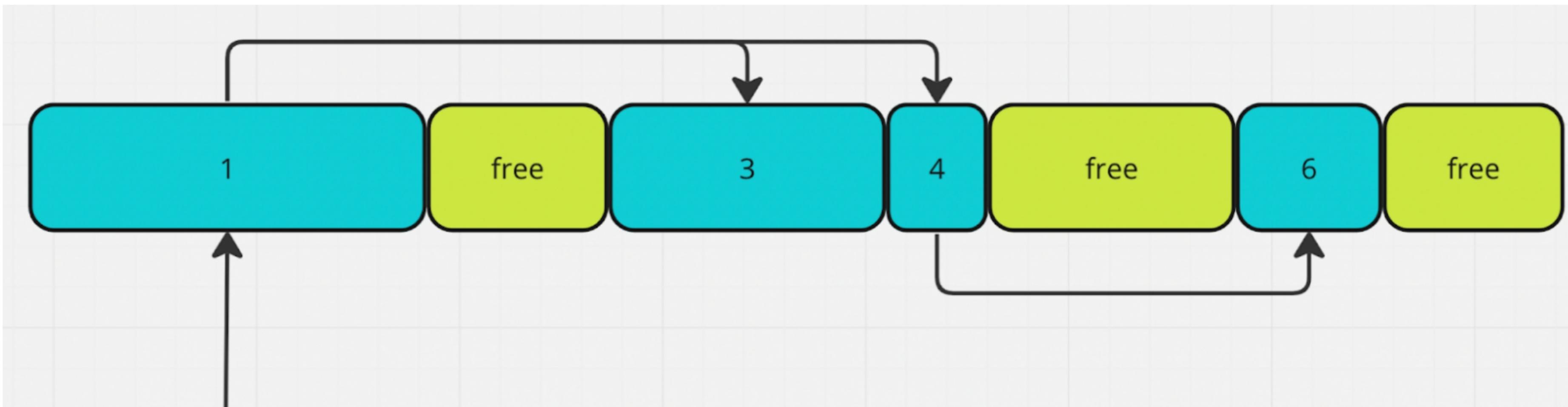
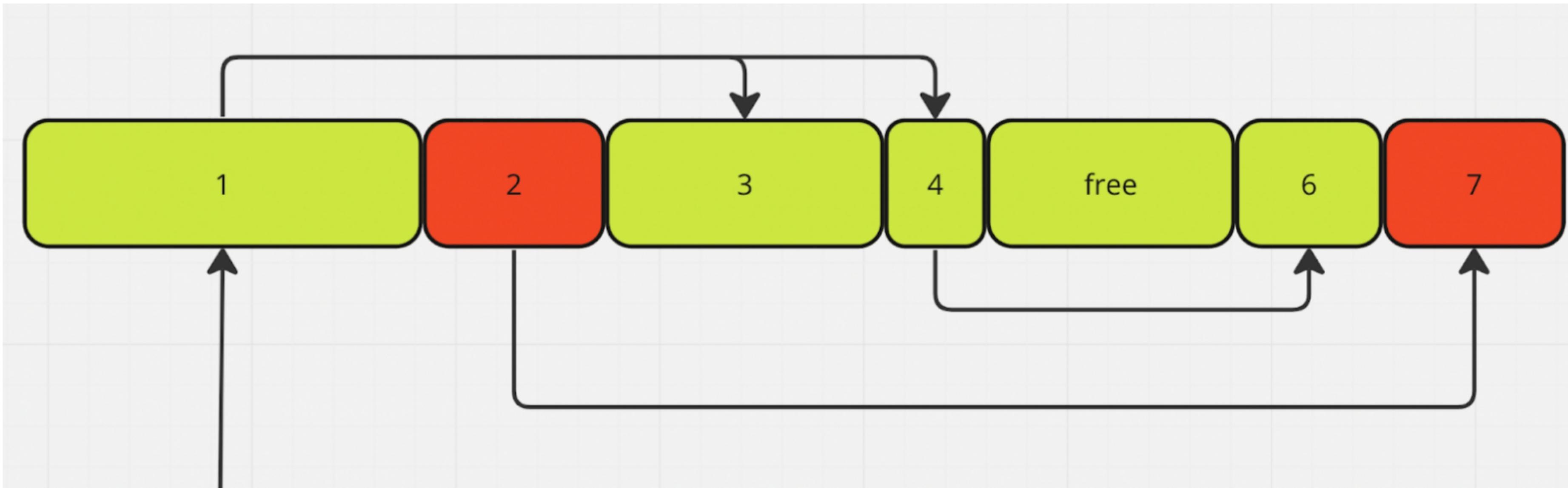
Сборщик мусора



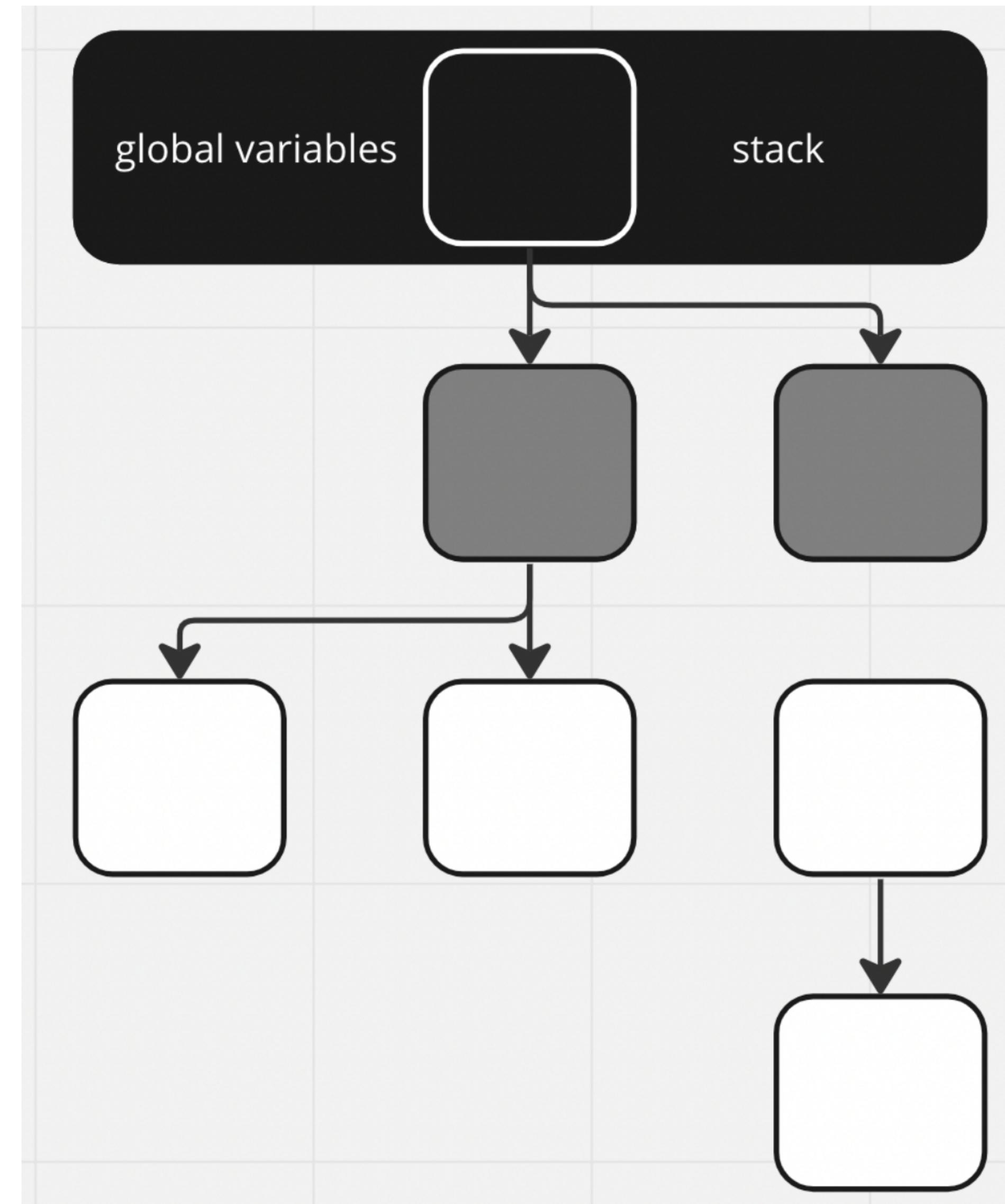
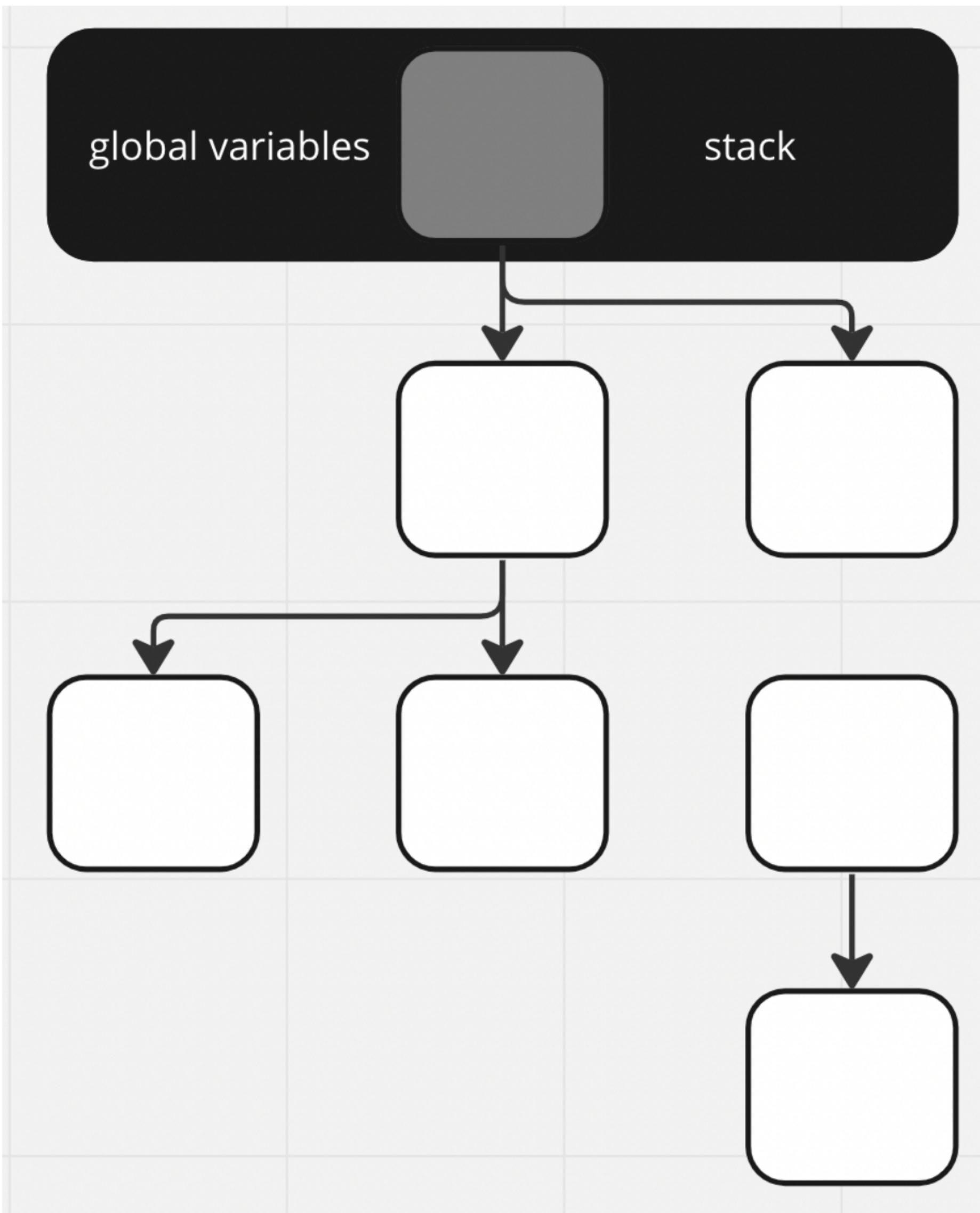
Сборщик мусора



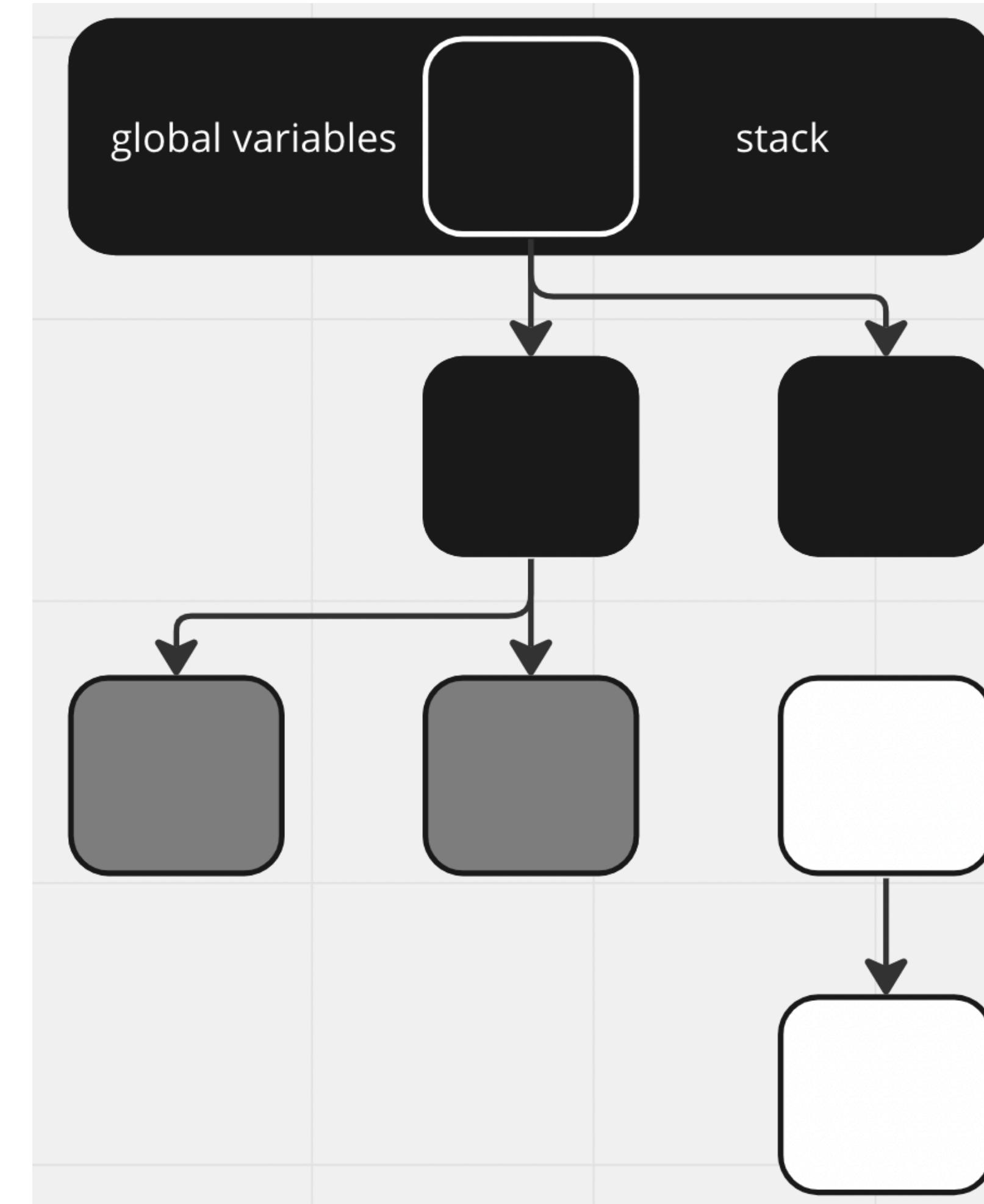
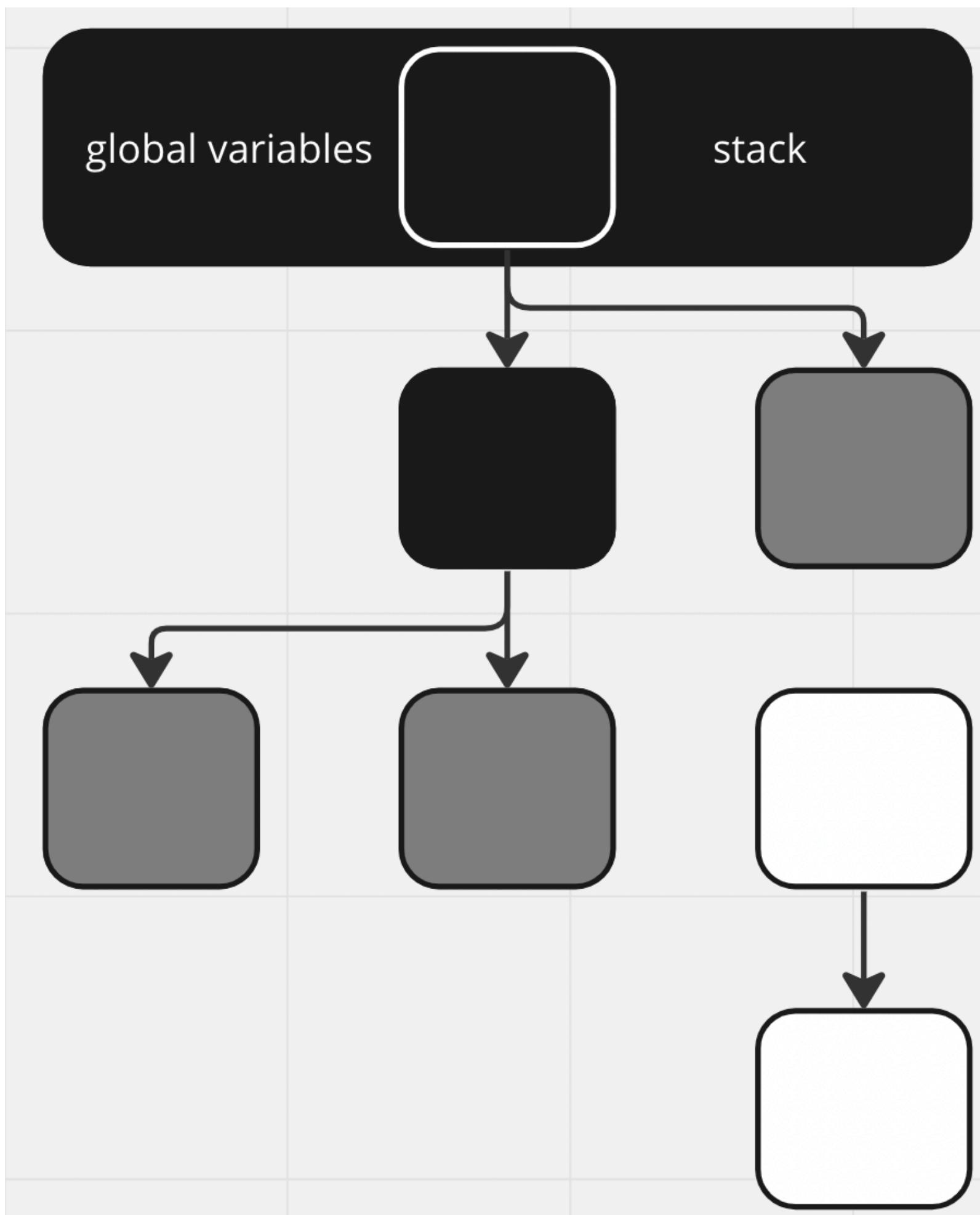
Сборщик мусора



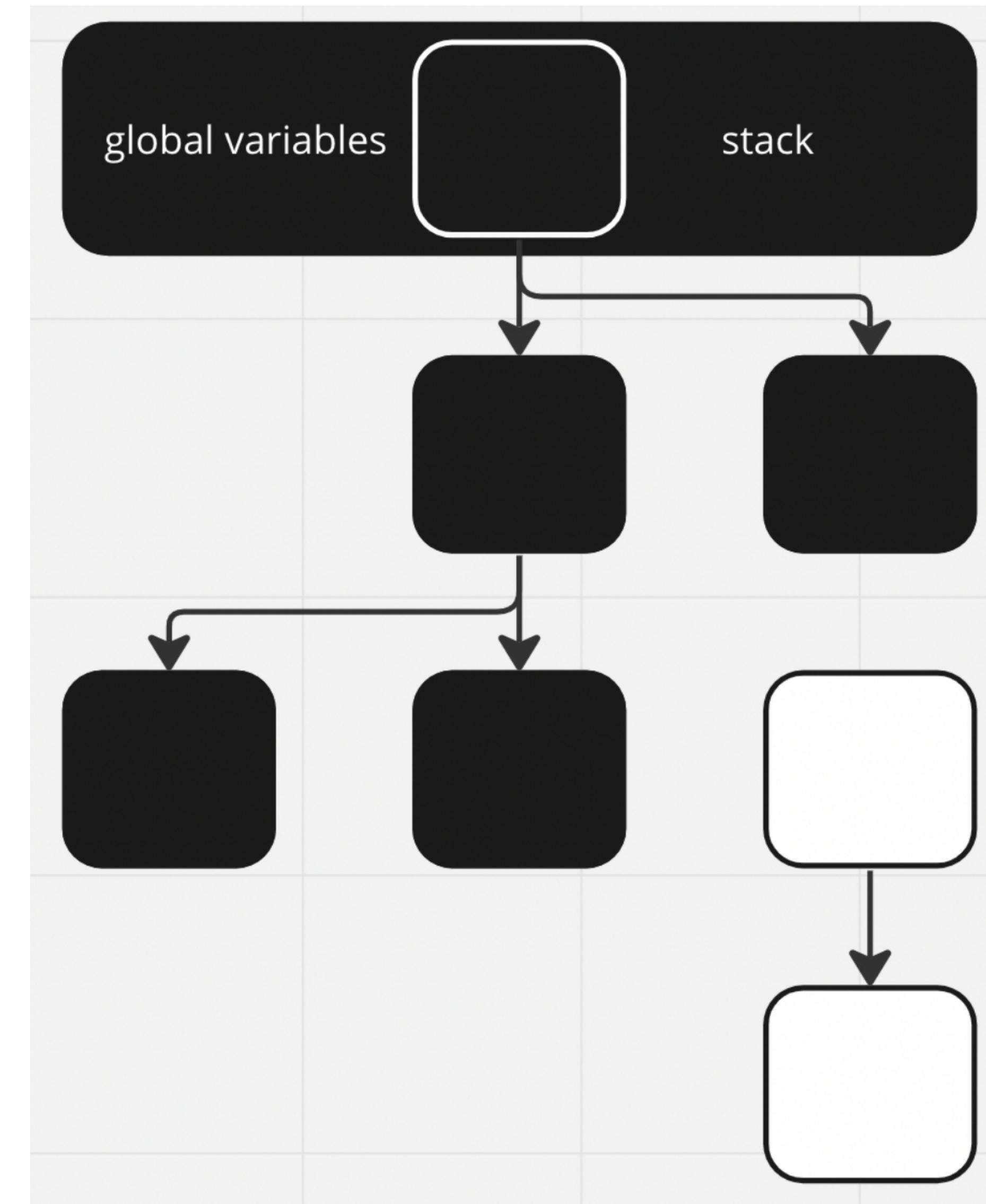
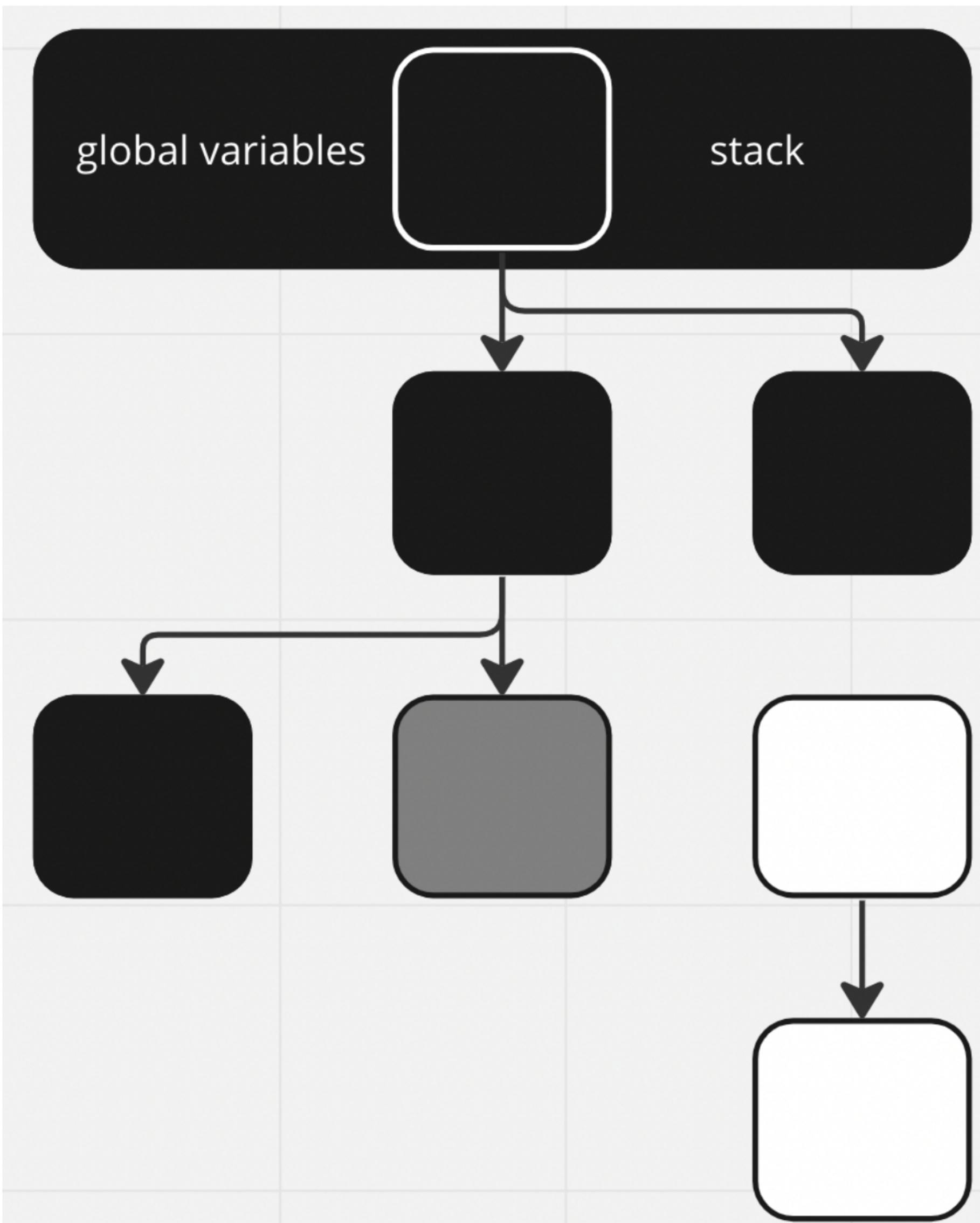
Сборщик мусора



Сборщик мусора



Сборщик мусора



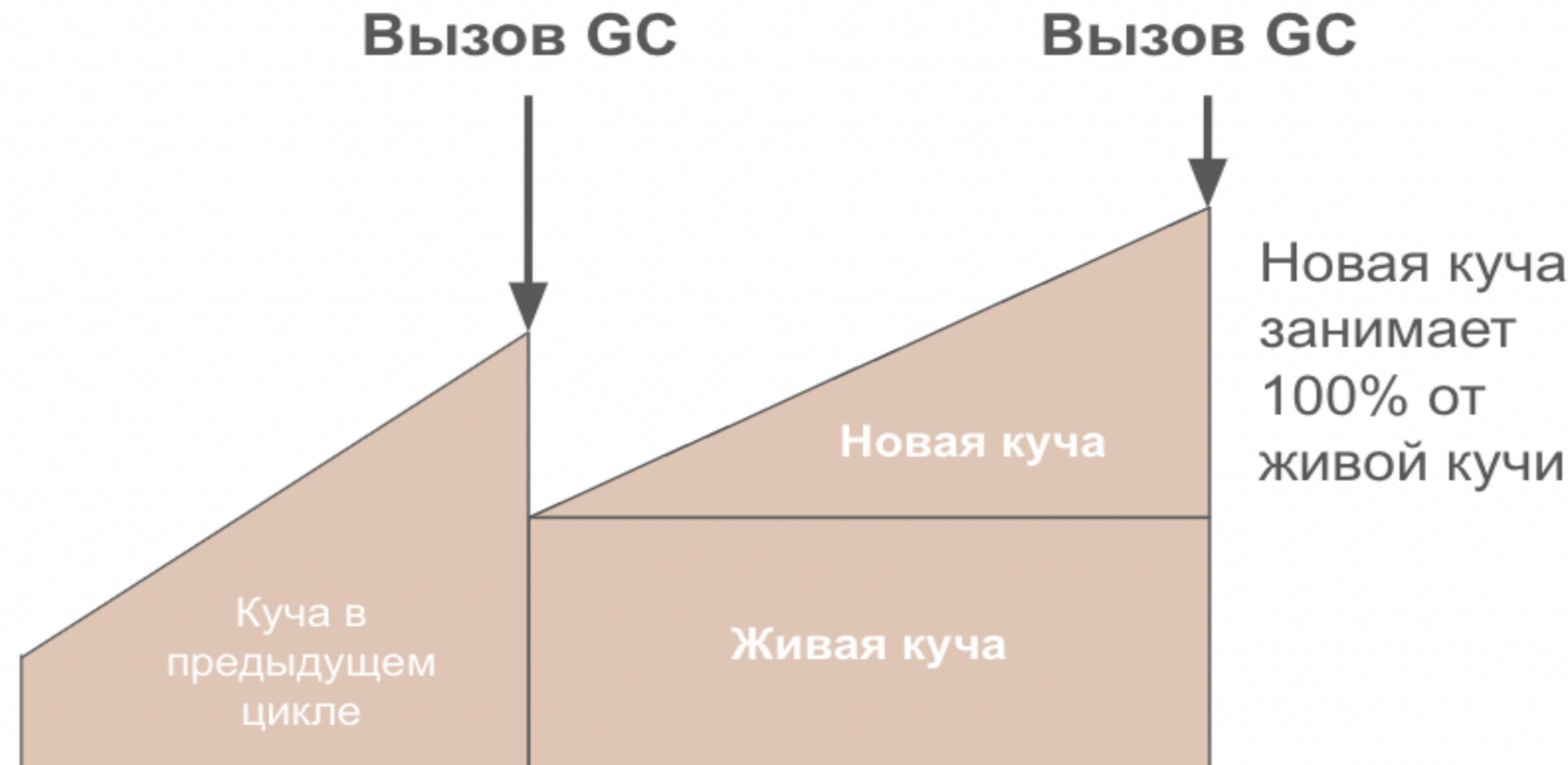
Сборщик мусора

GC начинает работу, когда размер кучи достигает целевого.

Целевой размер кучи вычисляется по формуле $\text{Live heap} + (\text{Live heap} + \text{GC roots}) * \text{GOGC} / 100$.

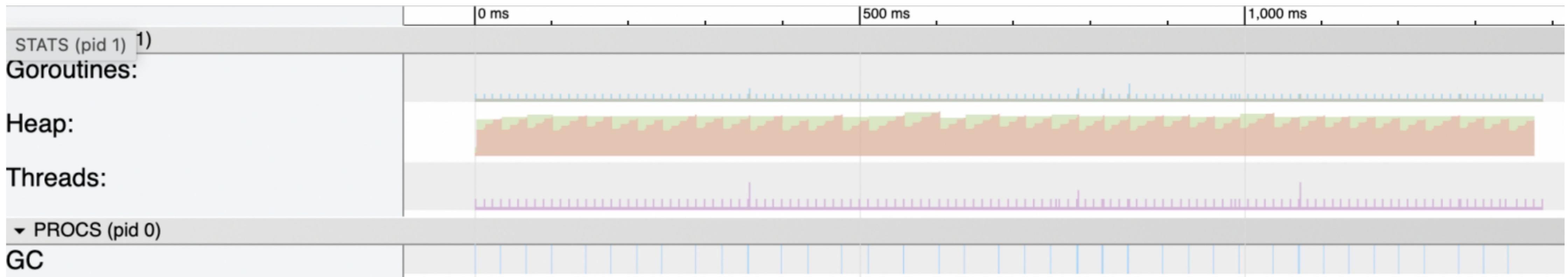
Частоту сборки мусора можно регулировать с помощью переменной окружения GOGC или функции [SetGCPercent](#) из пакета runtime/debug.

<https://tip.golang.org/doc/gc-guide#GOGC>



Сборщик мусора

Посмотреть работу GC можно с помощью утилиты go tool trace



Список материалов для изучения

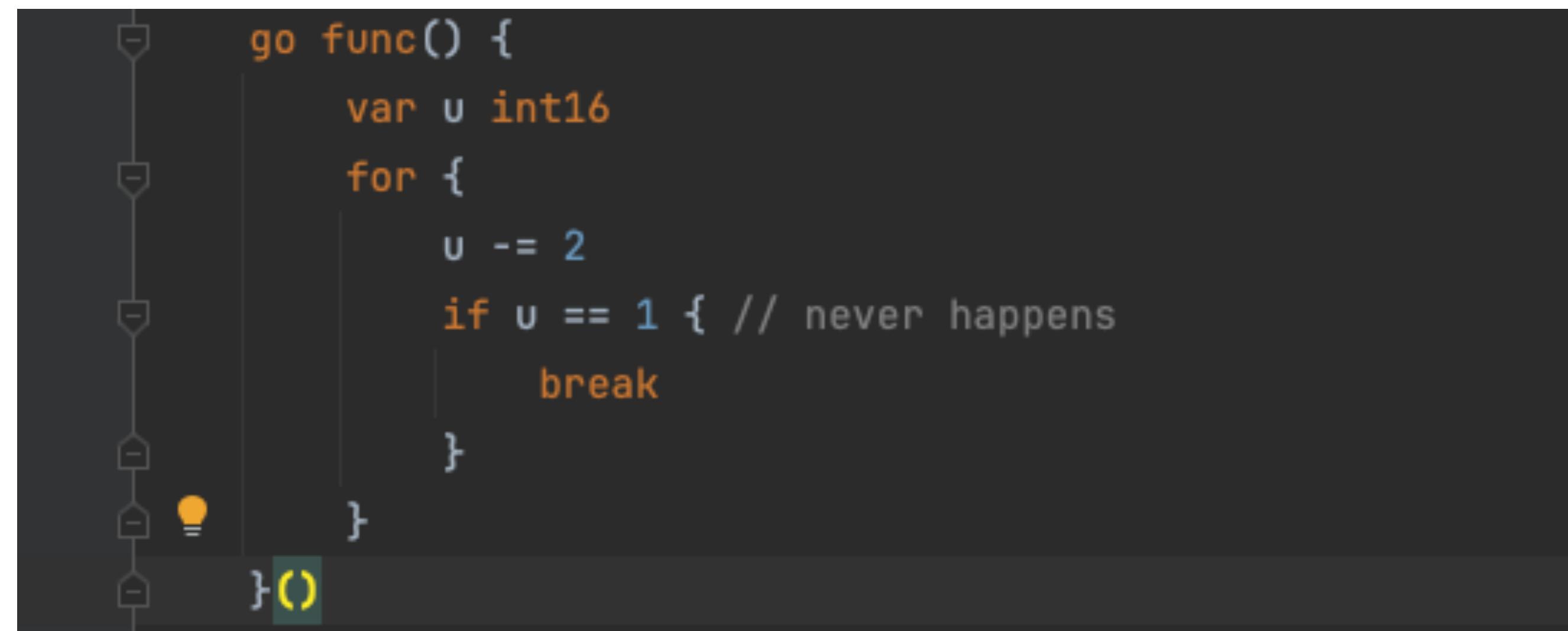
1. Планировщик Go – <https://habr.com/ru/articles/743266/>
2. Планировщик , но подробнее – <https://habr.com/ru/articles/804145/>
3. Асинхронное выполнение – <https://medium.com/a-journey-with-go/go-asynchronous-preemption-b5194227371c>

Горутины и каналы

Горутины

Горутины в Go создаются и запускаются с помощью ключевого слова `go`)

После которого идёт вызов функции, которая будет запущена в этой горутине.



The screenshot shows a portion of a Go code editor interface. On the left, there are several small, light-gray rectangular markers with thin black outlines, likely indicating code completion or selection points. The main area contains the following Go code:

```
go func() {  
    var u int16  
    for {  
        u -= 2  
        if u == 1 { // never happens  
            break  
        }  
    }  
}(0)
```

The code defines a goroutine using the `go` keyword. Inside the goroutine, a variable `u` of type `int16` is declared. A `for` loop is used to decrement `u` by 2 in each iteration. An `if` statement checks if `u` equals 1, with a note in the code that this never happens. If it did, the `break` keyword would exit the loop. The entire goroutine is enclosed in parentheses and followed by a closing brace, which is highlighted in yellow. The final closing brace is also highlighted in yellow, and the entire expression is enclosed in another set of parentheses, also highlighted in yellow.

Каналы

Go представляет несколько способов управления горутинами:

- примитивы синхронизации;
- каналы.



Сегодня рассматриваем каналы:

См так же: <https://go101.org/article/channel.html>

Буферизированные каналы

Каналы могут быть буферизированными и не буферизированными.

Горутина, которая записывает в канал, блокируется до тех пор, пока из него не прочитают.

Горутина, которая записывает в не буферизированный канал, блокируется после каждой записи.

Горутина, которая записывает в буферизированный канал, блокируется если буфер канала полон.

Select

Используется, если надо одновременно ждать сигнал от нескольких каналов:

```
select {
    case x := <-ch1:
        fmt.Println(x)
    case t := <-time.NewTimer(2 * time.Millisecond).C:
        fmt.Println(t)
}
```

select and send

еще select можно использовать для отправки в каналы.

close

- закрывает ТОЛЬКО источник (тот кто шлёт)
- всё ожидания прерываются - точно все? Что принимает select в пустом канале?, `for ... range` - завершается.

Примитивы синхронизации

WaitGroup

WaitGroup

1) *Что выведет эта программа ?*

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     for i := 0; i < 5; i++ {
7         go func() {
8             fmt.Println("go-go-go")
9         }()
10    }
11 }
12 }
```

WaitGroup

1) *Как решить проблему ожидания отработки горутин?*

```
5  func main() {
6      const goCount = 5
7      ch := make(chan struct{})
8      for i := 0; i < goCount; i++ {
9          go func() {
10             fmt.Println("go-go-go")
11             ch <- struct{}{}
12         }()
13     }
14
15     for i := 0; i < goCount; i++ {
16         <-ch
17     }
18 }
19 |
```

WaitGroup

1) *Как решить проблему ожидания отработки горутин?*

```
8  func main() {
9      const goCount = 5
10     wg := sync.WaitGroup{}
11     wg.Add(goCount) // <===
12     for i := 0; i < goCount; i++ {
13         go func() {
14             defer wg.Done() // <===
15             fmt.Println("go-go-go")
16         }()
17     }
18     wg.Wait() // <===
19 }
20 }
```

WaitGroup

1) Пример работы sync.WaitGroup

```
8  func main() {
9      const goCount = 5
10     wg := sync.WaitGroup{}
11     wg.Add(goCount) // <===
12     for i := 0; i < goCount; i++ {
13         go func() {
14             defer wg.Done() // <===
15             fmt.Println("go-go-go")
16         }()
17     }
18     wg.Wait() // <===
19 }
```

WaitGroup

1) *Методы sync.WaitGroup*

```
type WaitGroup struct {  
}
```

`func (wg *WaitGroup) Add(delta int)` – увеличивает счетчик `WaitGroup`.

`func (wg *WaitGroup) Done()` – уменьшает счетчик на 1.

`func (wg *WaitGroup) Wait()` – блокируется, пока счетчик `WaitGroup` не обнулится.

Mutex

Mutex

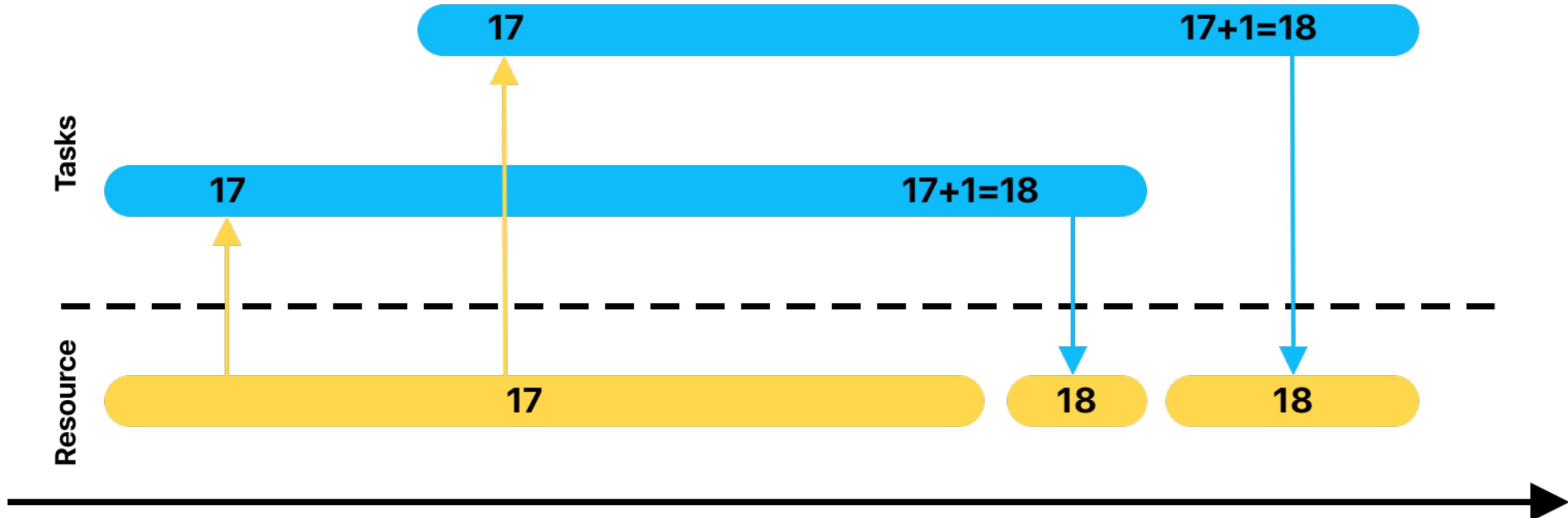
1) *Что выведет эта программа?*

```
8  func main() {
9      wg := sync.WaitGroup{}
10     v := 0
11     for i := 0; i < 1000; i++ {
12         wg.Add(1)
13         go func() {
14             old_v := v
15             new_v := old_v + 1
16             v = new_v
17             wg.Done()
18         }()
19     }
20     wg.Wait()
21     fmt.Println(v)
22 }
```

Асинхронная работа Горутины и каналы

```
// GOMAXPROCS=1 go run main.go  
// GOMAXPROCS=4 go run main.go  
// GOMAXPROCS=8 go run main.go
```

- 1) Одна из частых проблем в многопоточной программе это гонка за ресурсы
- 2) Возникает когда две и более горутины взаимодействуют с одной областью памяти



Mutex

```
7
8 func main() {
9     wg := sync.WaitGroup{}
10    v := 0
11    var mu sync.Mutex
12
13    for i := 0; i < 1000; i++ {
14        wg.Add(1)
15        go func() {
16            defer wg.Done()
17            defer mu.Unlock()
18
19            mu.Lock()
20
21            v++
22        }()
23    }
24
25    wg.Wait()
26    fmt.Println(v)
27 }
28 }
```

Mutex

- 1) Мьютекс (англ. mutex, от mutual exclusion — « взаимное исключение »).
- 2) Код между Lock и Unlock выполняет только однаgorутина , остальные ждут :

mutex.Lock()

v++

mutex.Unlock()

Mutex

```
5  type Counters struct {
6      mu sync.Mutex
7      m map[int]int
8  }
9
10 func New() *Counters {
11     return &Counters{
12         mu: sync.Mutex{},
13         m: make(map[int]int),
14     }
15 }
16
17 func (c *Counters) Load(key int) (int, bool) {
18     c.mu.Lock()
19     defer c.mu.Unlock()
20     val, ok := c.m[key]
21     return val, ok
22 }
23 func (c *Counters) Store(key int, value int) {
24     c.mu.Lock()
25     defer c.mu.Unlock()
26     c.m[key] = value
27 }
```

RWMutex

RWMutex

- 1) Позволяет более оптимально использовать мьютексы, не блокируя чтение

```
type Counters struct {
    mu sync.RWMutex
    m map[int]int
}

func New() *Counters {
    return &Counters{
        mu: sync.RWMutex{},
        m: make(map[int]int),
    }
}

func (c *Counters) Load(key int) (int, bool) {
    c.mu.RLock()
    defer c.mu.RUnlock()
    val, ok := c.m[key]
    return val, ok
}

func (c *Counters) Store(key int, value int) {
    c.mu.Lock()
    defer c.mu.Unlock()
    c.m[key] = value
}
```

sync.Map

sync.Map

```
5  func main() {
6      v := sync.Map{}
7
8      var wg sync.WaitGroup
9
10     for i := 0; i < 1000; i++ {
11         wg.Add(1)
12         go func(i int) {
13             defer wg.Done()
14
15             v.Store(i, i)
16         }(i)
17     }
18
19     wg.Wait()
20 }
21 }
```

sync.Map

```
type Map struct {  
}  
  
func (m *Map) Delete(key interface{})  
func (m *Map) Load(key interface{}) (value interface{}, ok bool)  
func (m *Map) LoadOrStore(key, value interface{}) (actual interface{}, loaded bool)  
func (m *Map) Range(f func(key, value interface{}) bool)  
func (m *Map) Store(key, value interface{})
```

atomic

atomic

```
func main() {
    var counter int32
    wg := sync.WaitGroup{}

    for i := 0; i < 100; i++ {
        wg.Add(1)
        go func() {
            atomic.AddInt32(&counter, 1)
            wg.Done()
        }()
    }

    wg.Wait()
    fmt.Println("Counter:", counter)
}
```

```
var dd atomic.Int32
dd.Add(123)
fmt.Println(dd.Load())
```

sync.Once

Sync.Once

- 1) Единожды выполнит переданную в Do() функцию.
При повторном вызове прогнозирует

```
8  var (
9    one sync.Once
10 )
11
12 func getSomething() int {
13     var result int
14     one.Do(
15         func() {
16             fmt.Println("sync.Once")
17             result += 1
18         },
19     )
20     return result
21 }
22
23 func main() {
24     wg := sync.WaitGroup{}
25     countG := 5
26
27     for i := 0; i < countG; i++ {
28         wg.Add(1)
29         go func() {
30             defer wg.Done()
31             fmt.Println(getSomething())
32         }()
33     }
34     wg.Wait()
35 }
36 }
```

Список материалов для изучения

1. Примитивы синхронизации в Go – <https://medium.com/german-gorelkin/synchronization-primitives-go-8857747d9660>
2. Горутины – <https://habr.com/ru/articles/830460/>
3. Каналы – <https://habr.com/ru/articles/490336/>

Context

Что такое Context?

Это интерфейс с четырьмя методами

- 1) Deadline() (deadline `time.Time`, ok `bool`)
- 2) Done() <-chan `struct{}`
- 3) Err() `error`
- 4) Value(key `any`) `any`

Создание контекста

- 1) `context.Background()` Context - возвращает пустой контекст, который никогда не завершается, не хранит значения и не имеет дедлайна. Используется в функции `main`, инициализации, в тестах, и как верхнеуровневый контекст для входящих запросов.
- 2) `context.TODO()` - также возвращает пустой контекст. Используется когда не ясно какой контекст использовать или когда функция, в которой он создаётся, ещё не принимает контекст как параметр.

Создание дочернего контекста

Дочерний контекст создаётся с помощью методов:

- 1) context.WithCancel(parent Context) (ctx Context, cancel CancelFunc)
 - 2) context.WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)
 - 3) context.WithDeadline(parent Context, d time.Time) (Context, CancelFunc)
 - 4) contextWithValue(parent Context, key, val any) Context
- ...

Сигналы ОС

Что такое сигнал ОС?

Сигнал в операционных системах семейства Unix — асинхронное уведомление процесса о каком-либо событии.

Они посылаются программе для остановки программы или обработки ошибки.

[https://en.wikipedia.org/wiki/Signal_\(IPC\)](https://en.wikipedia.org/wiki/Signal_(IPC))

Обработка сигналов ОС

Функции для обработки из пакета signal (<https://pkg.go.dev/os/signal>):

```
Notify(c chan<- os.Signal, sig ...os.Signal)
NotifyContext(parent context.Context, signals ...os.Signal) (ctx context.Context, stop
context.CancelFunc)
Ignore(sig ...os.Signal)
...
```