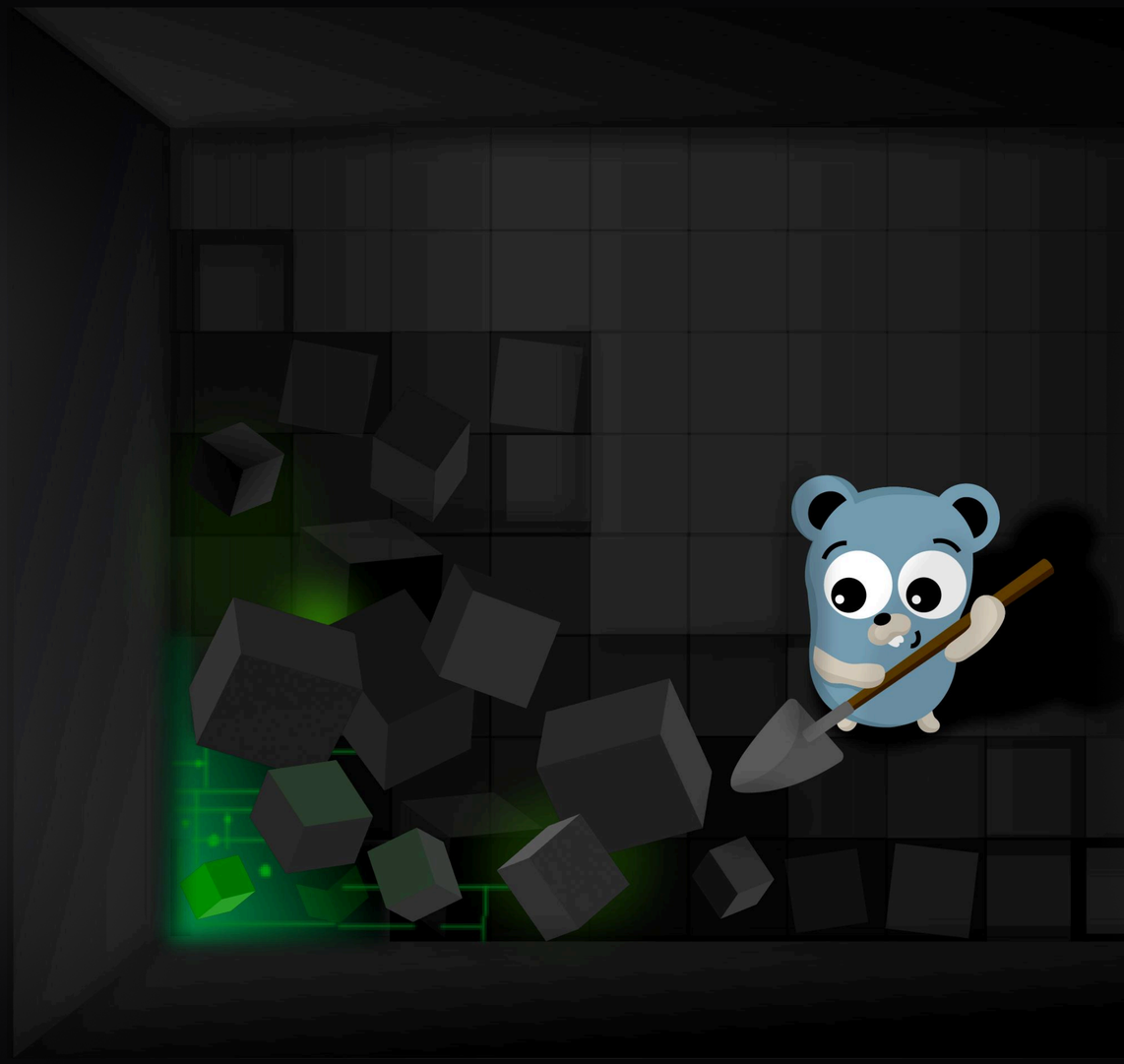
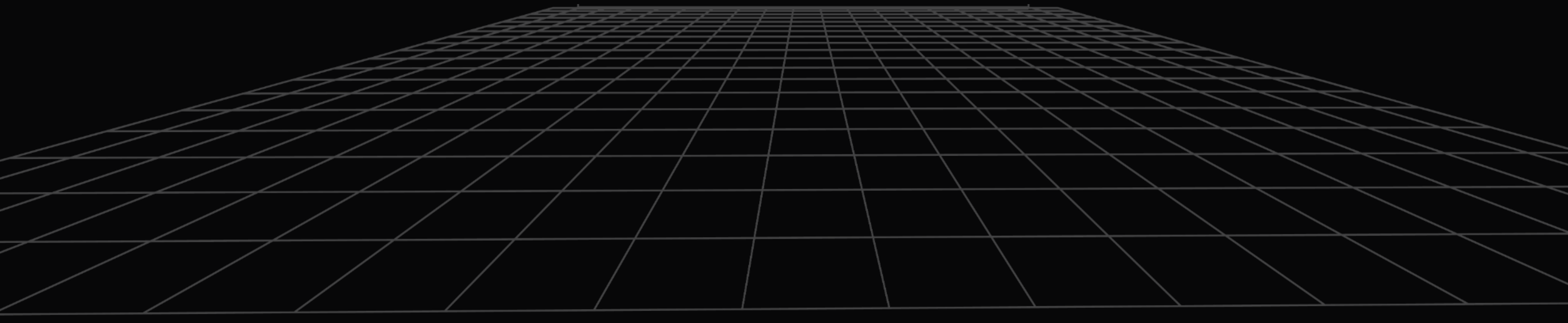
АЛЛОКАТОР В GOLANG



Balun.Courses

ПРОВЕРЬ ЗАПИСЬ

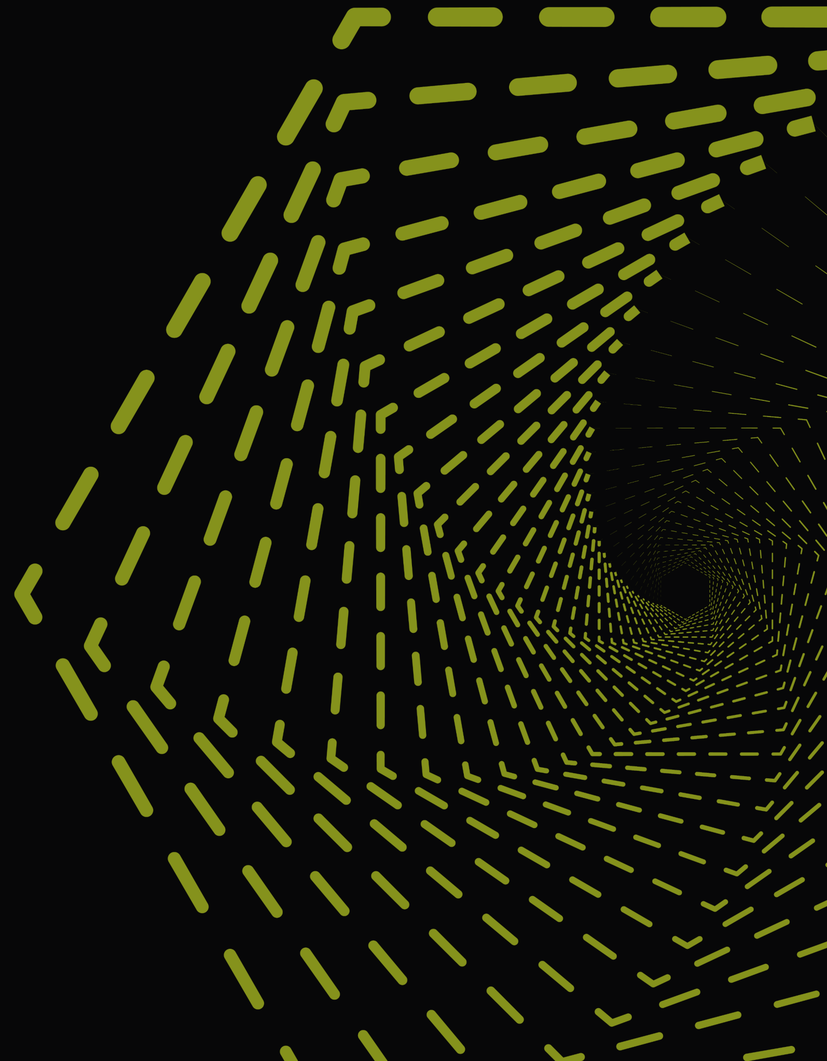


ПРАВИЛА ЗАНЯТИЯ

1. вопросы в чате можно задавать
   в любое время

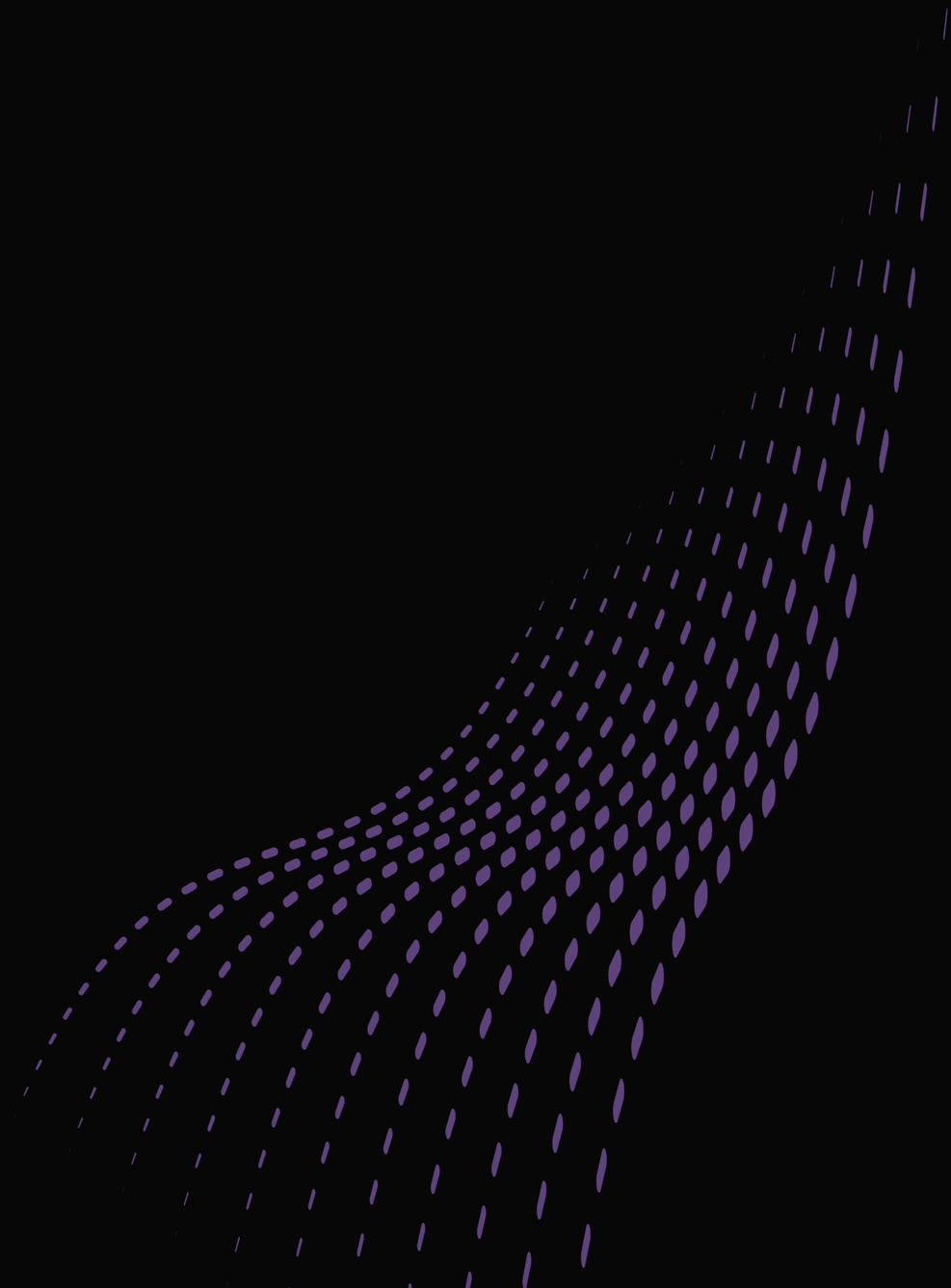
2. вопросы голосом задаем
   по поднятой руке в Zoom

3. ответы на вопросы будут
   в запланированных местах



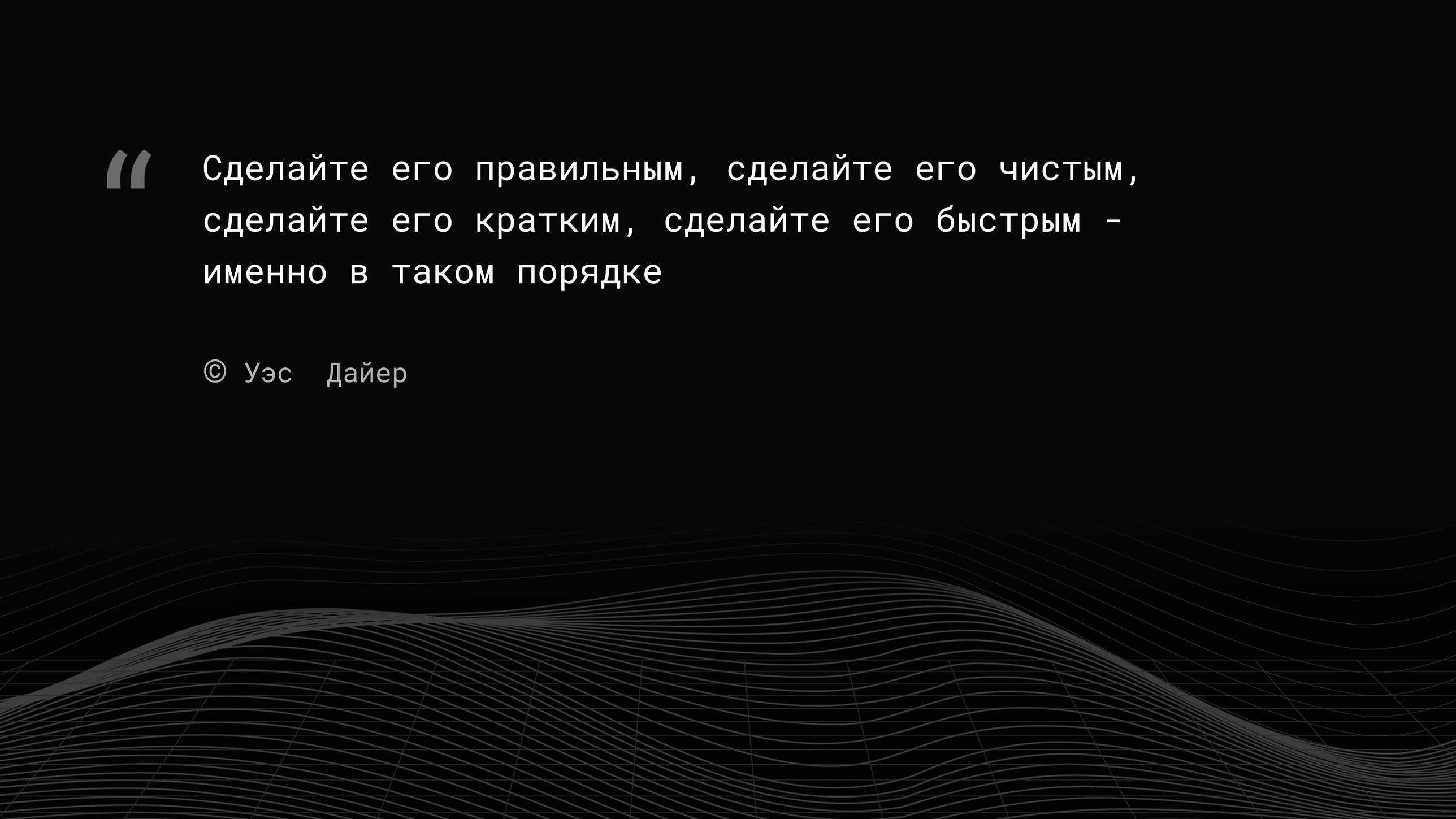
ПЛАН ЛЕКЦИИ

1. Алгоритмы распределения памяти
2. Внутреннее устройство аллокатора Go
3. Аллоцирование объектов, sync.Pool и арены



> Сделайте его правильным, сделайте его чистым, сделайте его кратким, сделайте его быстрым - именно в таком порядке

© Уэс Дайер



АЛГОРИТМЫ РАСПРЕДЕЛЕНИЯ ПАМЯТИ

АЛЛОКАТОР

Скорость работы

- Кеш-локальность
- Выдача недавно освобожденных блоков (которые не вымылись из кешей)

Масштабируемость

- Расширение на несколько CPU
- Взаимодействие с маленькими и большими участками памяти

Потребление памяти

- Эффективность структур хранения данных
- Низкая фрагментация

Безопасность

- Защита от неправильного использования
- Защита от атак (например переполнений)

Аспекты противоречат друг другу, поэтому приходится балансировать и находить компромисс. Либо нужно давать пользователю возможность крутить настройки

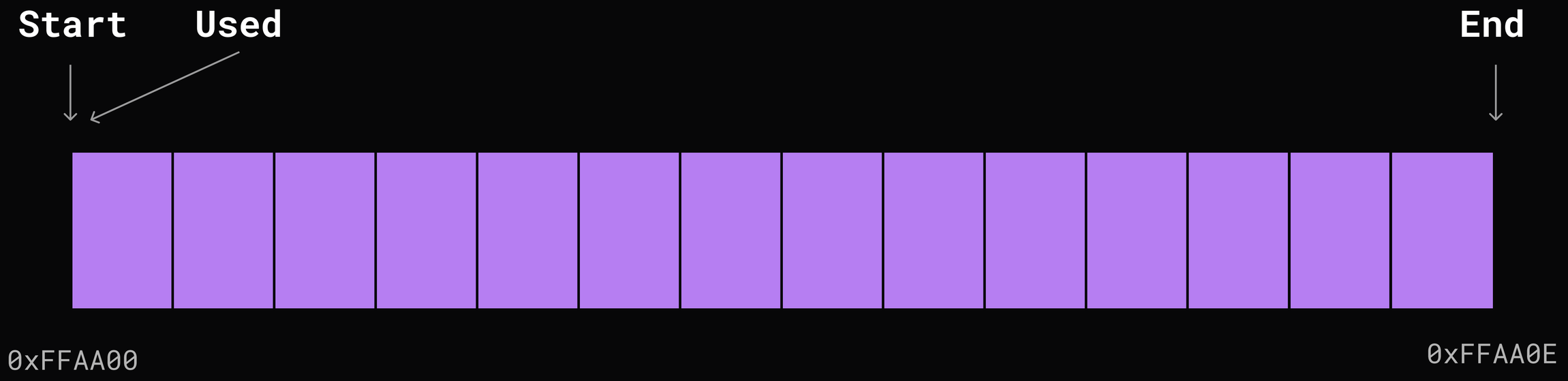
ИНТЕРФЕЙС

1. **create** – создает аллокатор и отдает ему в распоряжение некоторый объем памяти;
2. **allocate** – выделяет блок определенного размера из области памяти, которым распоряжается аллокатор;
3. **deallocate** – освобождает определенный блок;
4. **free** – освобождает все выделенные блоки из памяти аллокатора (память, выделенная аллокатору, не освобождается).

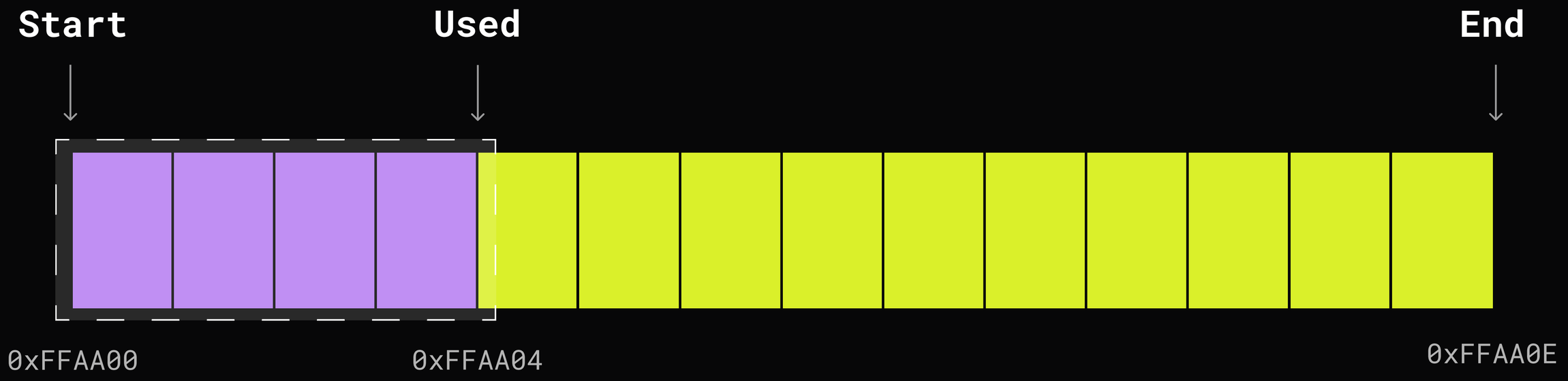
Terminal: deep_go

ЛИНЕЙНЫЙ АЛЛОКАТОР

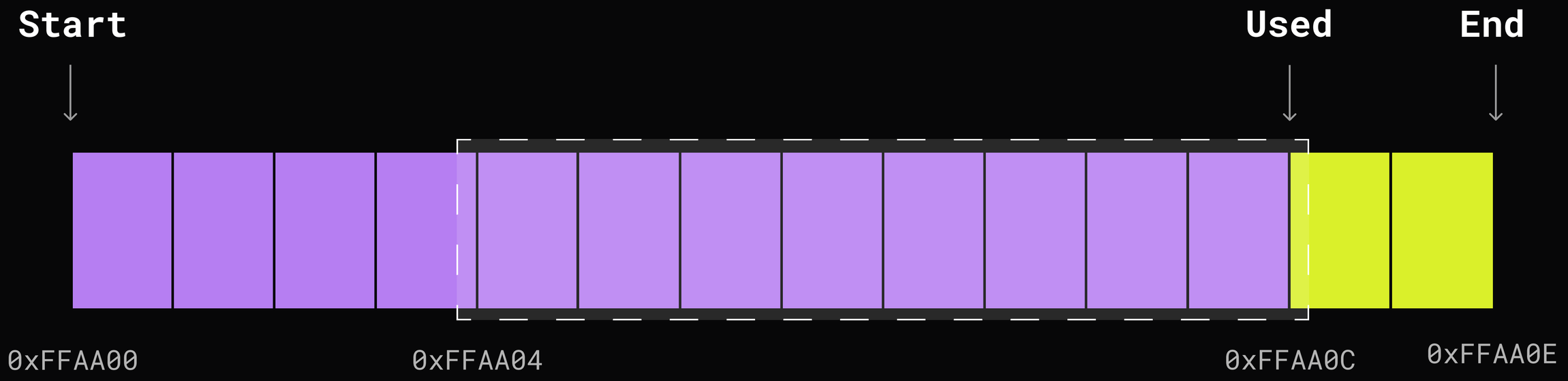
Самый простой вид аллокаторов. Идея состоит в том, чтобы сохранить указатель на начало блока памяти выделенному аллокатору, а также использовать другой указатель или числовое представление, которое необходимо будет перемещать каждый раз, когда выделение из аллокатора завершено


Start
Used
End
0xFFAA00
0xFFAA0E

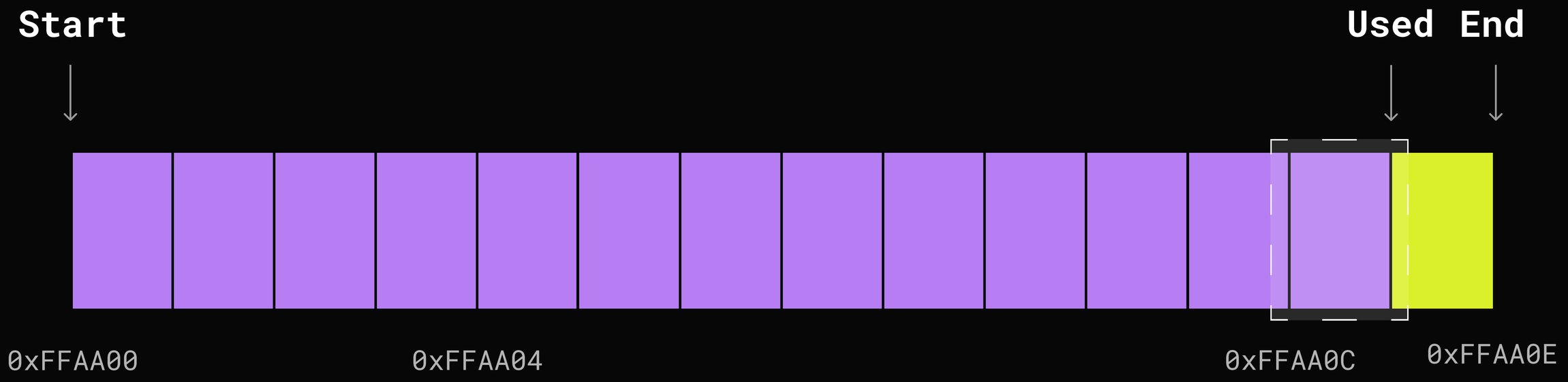
ВЫДЕЛЯЕМ 4 БАЙТА



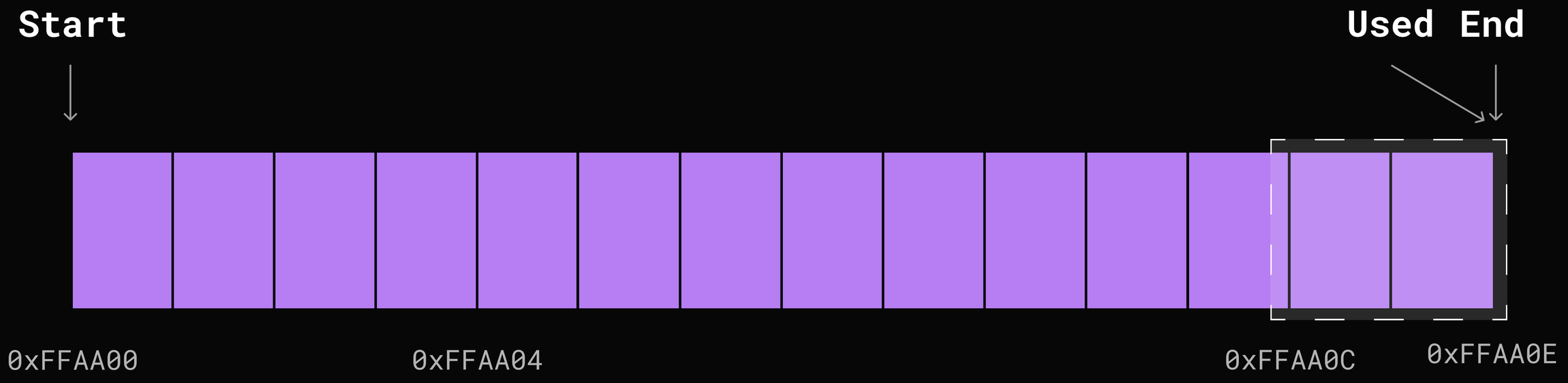
ВЫДЕЛЯЕМ 8 БАЙТ



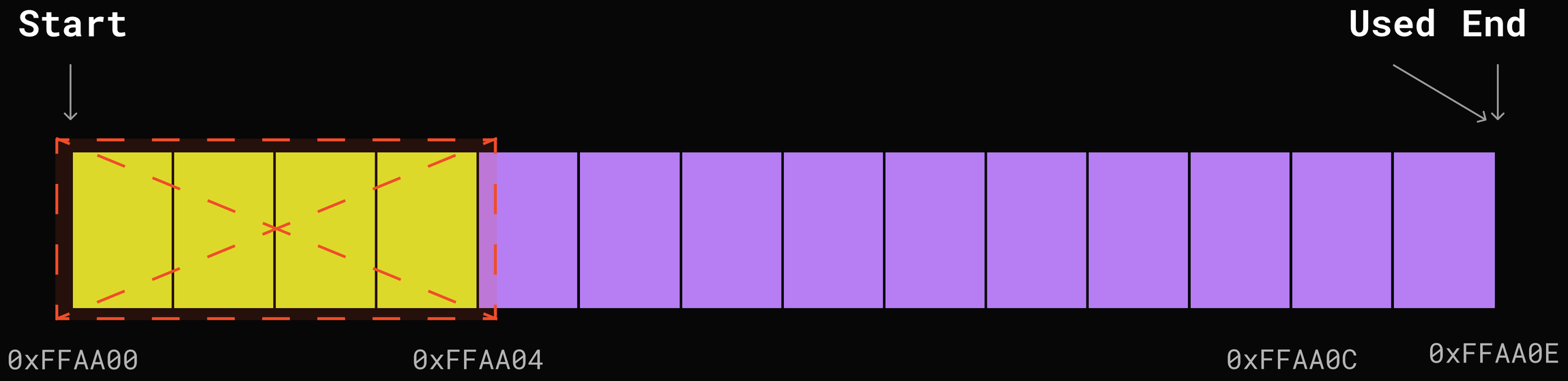
1. ВЫДЕЛЯЕМ 1 БАЙТ



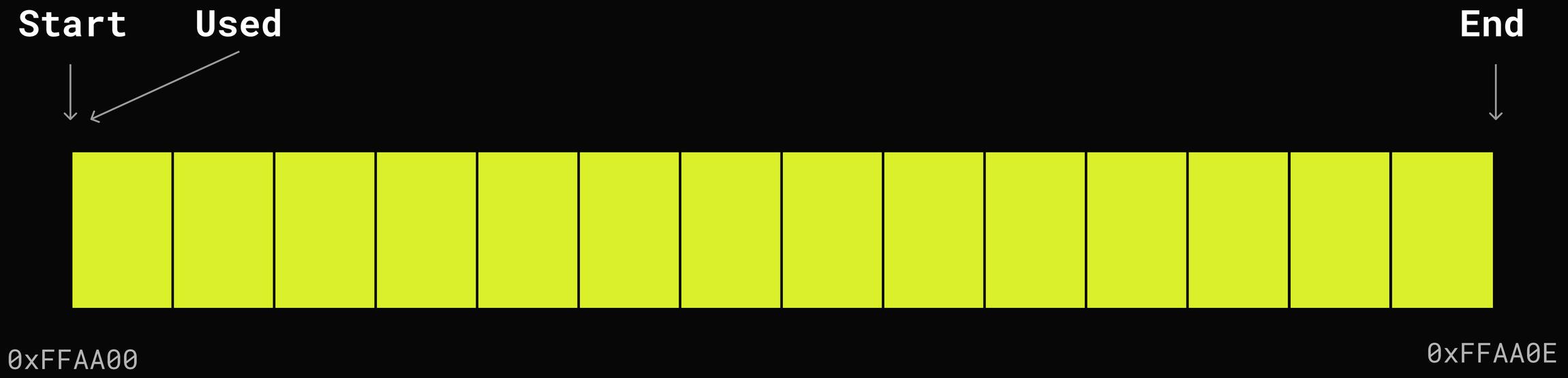
2. ВЫДЕЛЯЕМ 1 БАЙТ



НЕ ПОДДЕРЖИВАЕТСЯ ВЫБОРОЧНОЕ ОСВОБОЖДЕНИЕ ПАМЯТИ



ОСВОБОЖДАЕМ ВСЮ ЗАНЯТУЮ ПАМЯТЬ



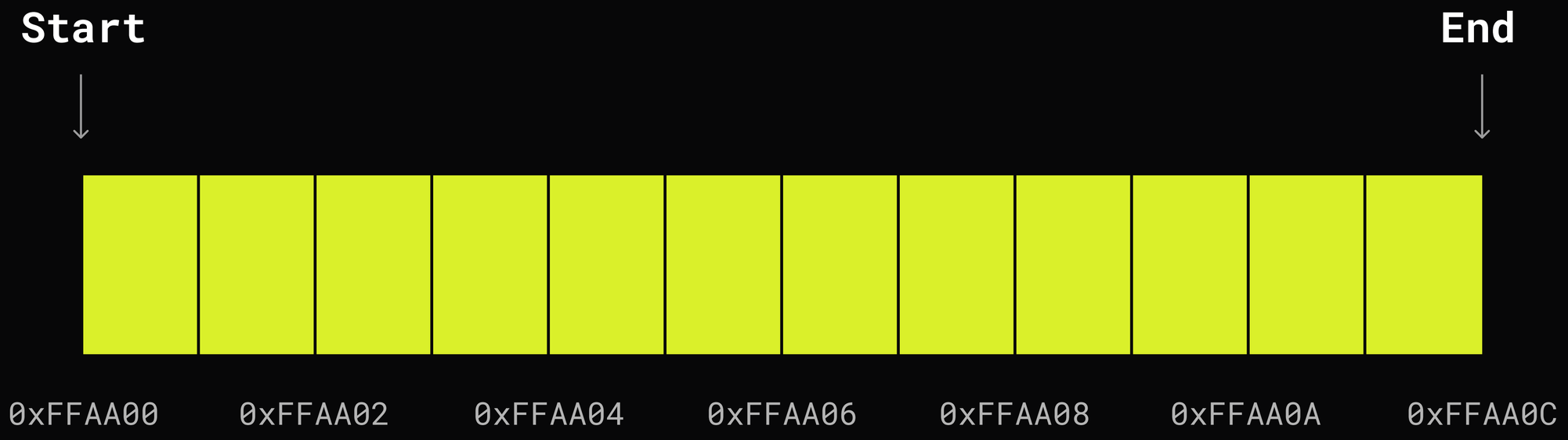
EXAMPLE

Linear allocator

Terminal: deep_go

БЛОЧНЫЙ АЛЛОКАТОР

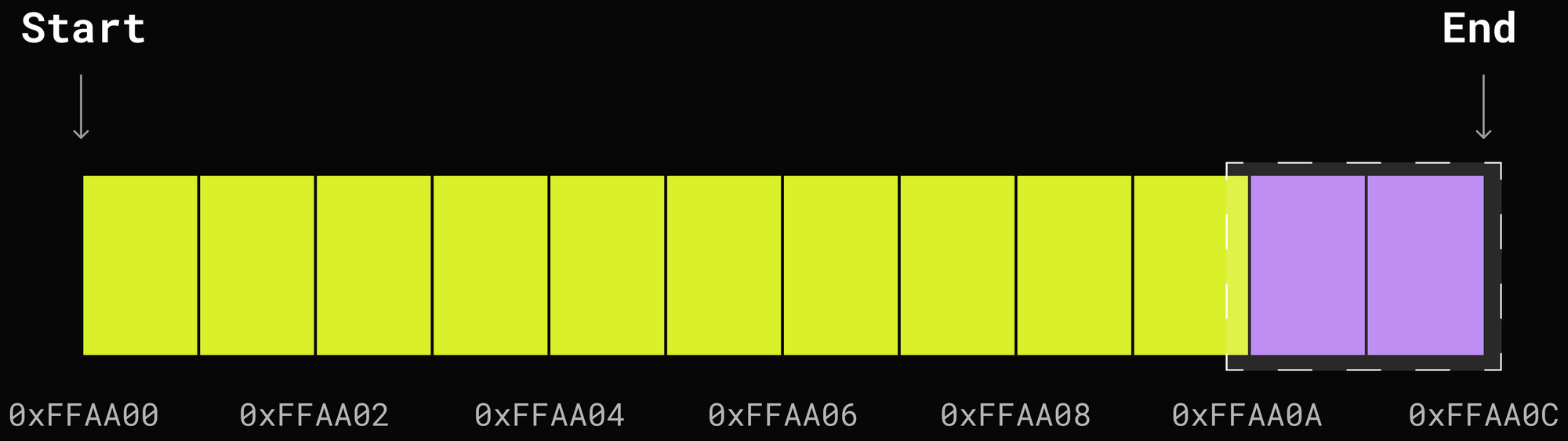
Идея блочного аллокатора заключается в том, что он разделяет некоторый большой участок памяти на более мелкие участки **одинакового размера**.


Start
End
0xFFAA00
0xFFAA02
0xFFAA04
0xFFAA06
0xFFAA08
0xFFAA0A
0xFFAA0C

FreeBlocks

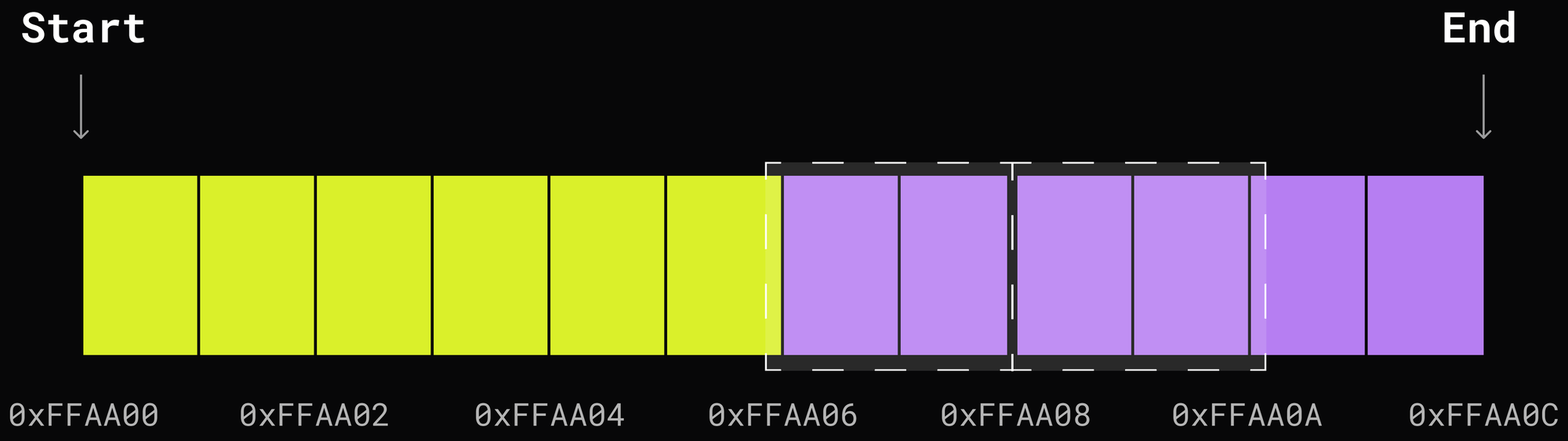

0xFFAA00
0xFFAA02
0xFFAA04
0xFFAA06
0xFFAA08
0xFFAA0A

ВЫДЕЛЯЕМ 2 БАЙТА

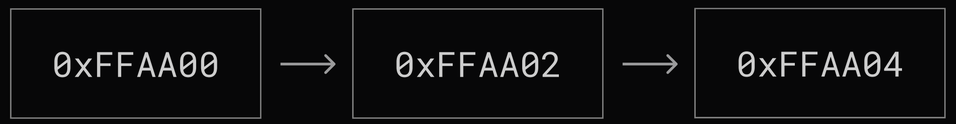


FreeBlocks

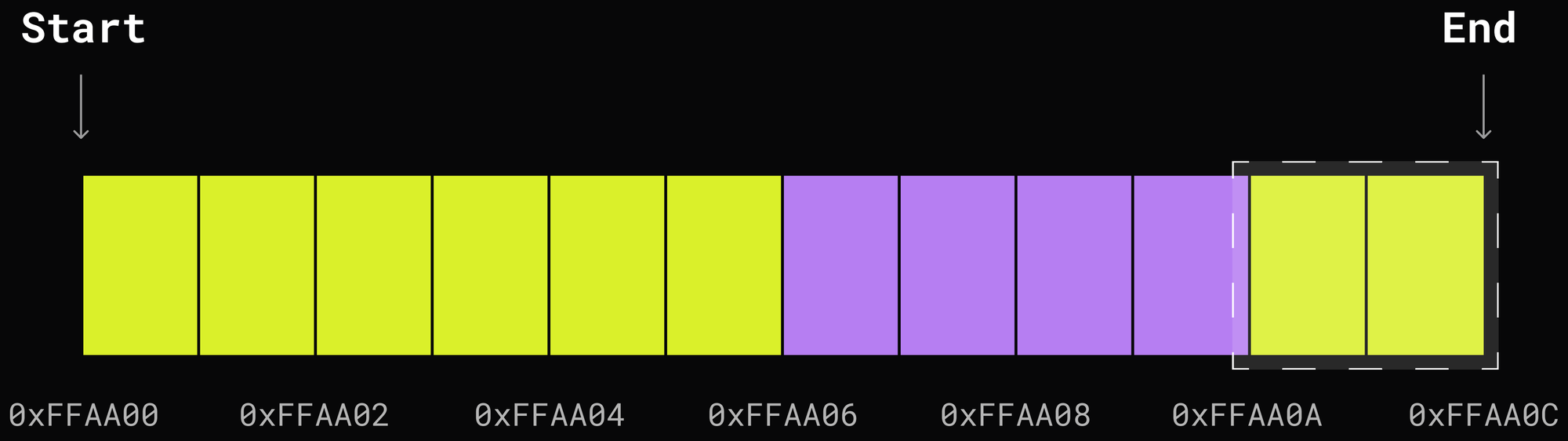
ВЫДЕЛЯЕМ 2 БАЙТА ДВА РАЗА



FreeBlocks

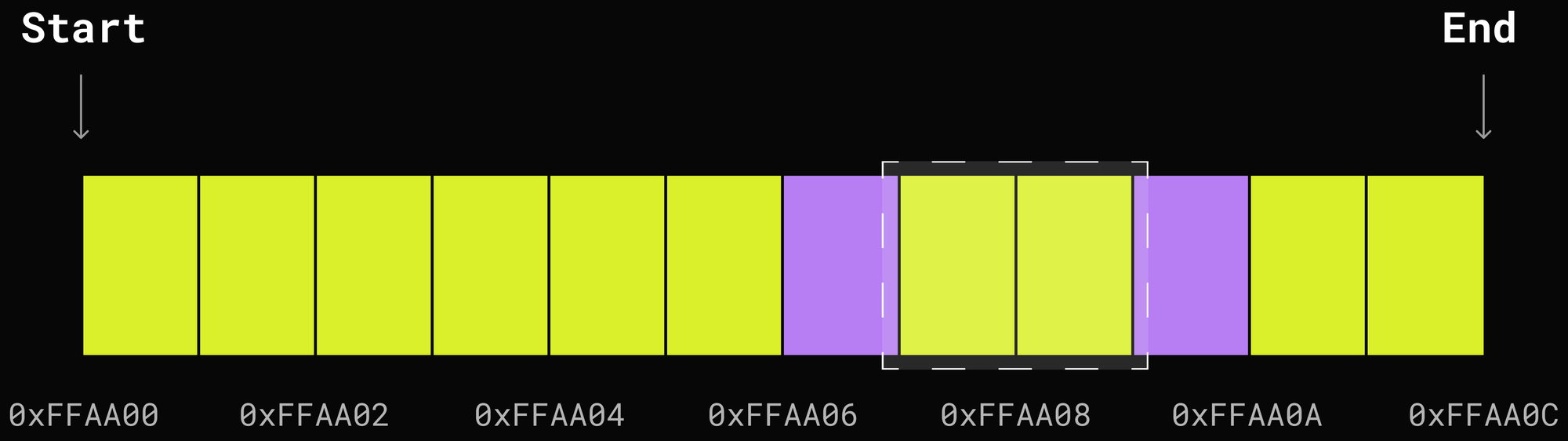


ОСВОБОЖДАЕМ 2 БАЙТА



FreeBlocks

ОСВОБОЖДЕНИЕ НЕКОРРЕКТНОГО БЛОКА



FreeBlocks

EXAMPLE

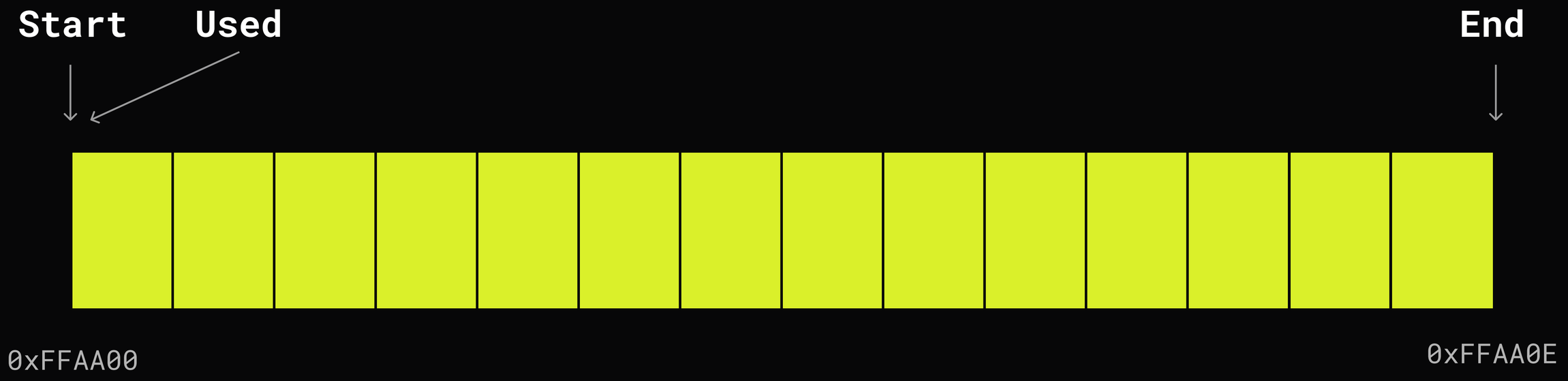
Pool allocator

Terminal: deep_go

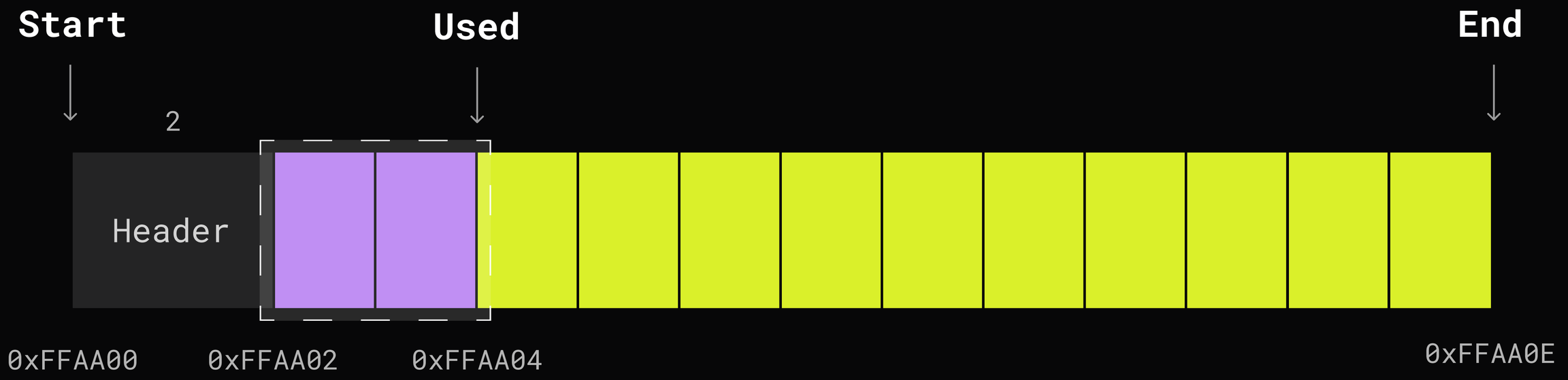
СТЕКОВЫЙ АЛЛОКАТОР

Умная эволюция линейного распределителя, которая позволяет управлять памятью, как стеком.

Все так же, как и раньше, мы сохраняем указатель вместе с «header» блоком (в дальнейшем будет употребляться, как заголовок) на текущий адрес памяти и перемещаем его вперед для каждого выделения.

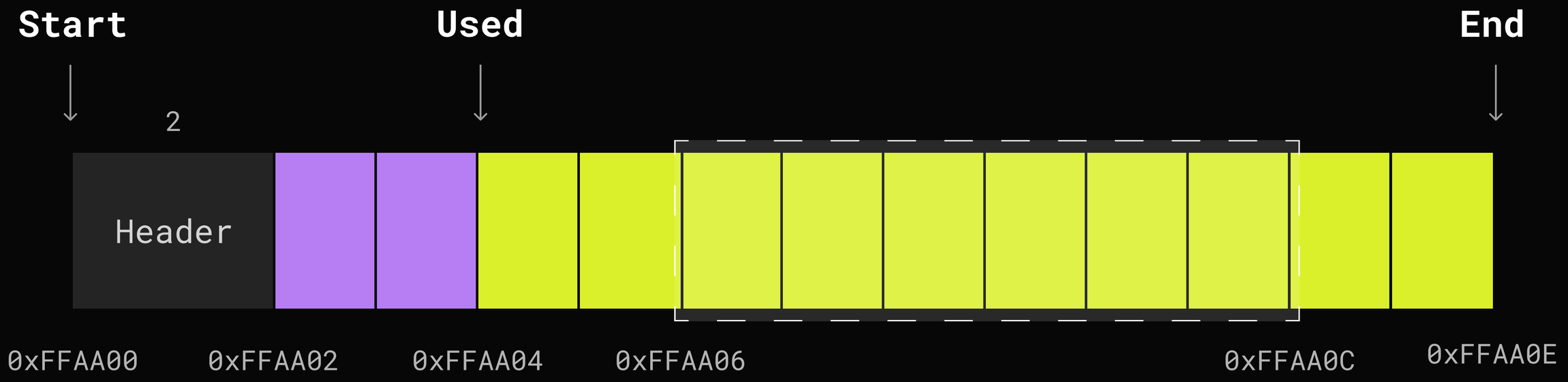

Start
Used
End
0xFFAA00
0xFFAA0E

ВЫДЕЛЯЕМ 2 БАЙТА



ВЫДЕЛЯЕМ 6 БАЙТ

ОСВОБОЖДАЕМ 6 БАЙТ



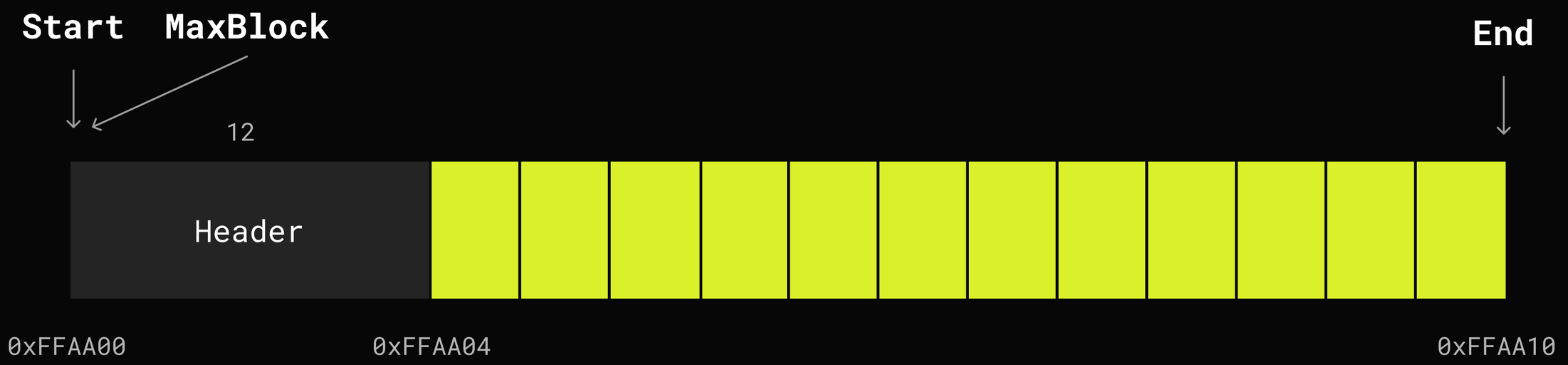
EXAMPLE

Stack allocator

MALLOC

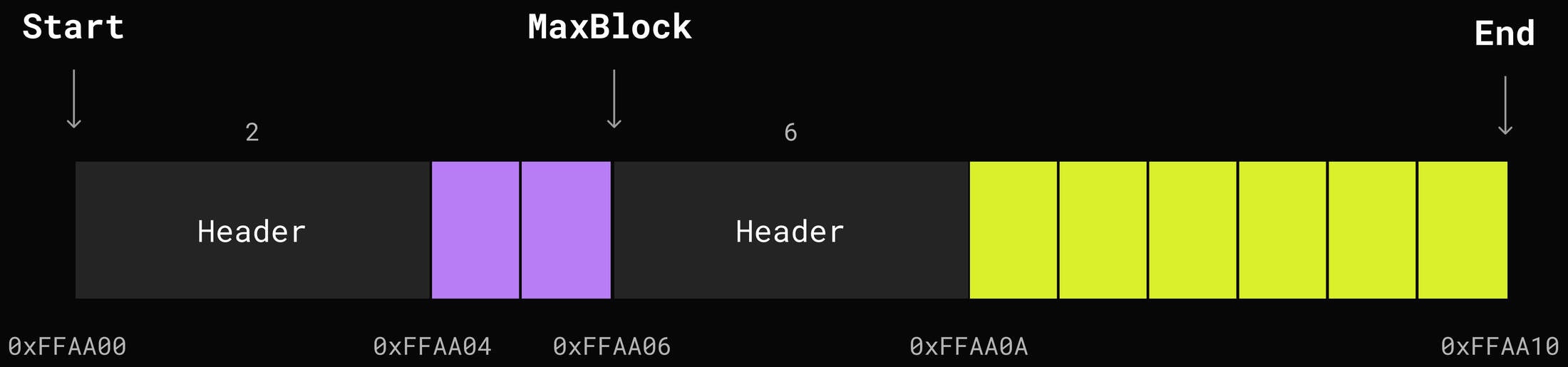
General purposes allocator

МОЖНО ИСПОЛЬЗОВАТЬ ЗАГОЛОВКИ МЕНЬШЕГО РАЗМЕРА



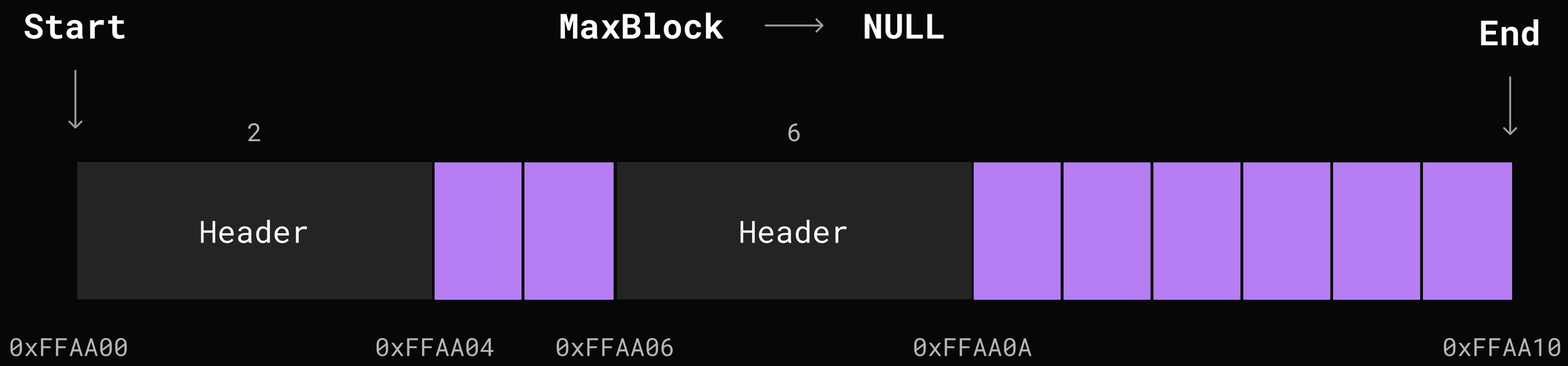
FreeBlocks = {0xFFA00}

ВЫДЕЛЯЕМ 2 БАЙТА



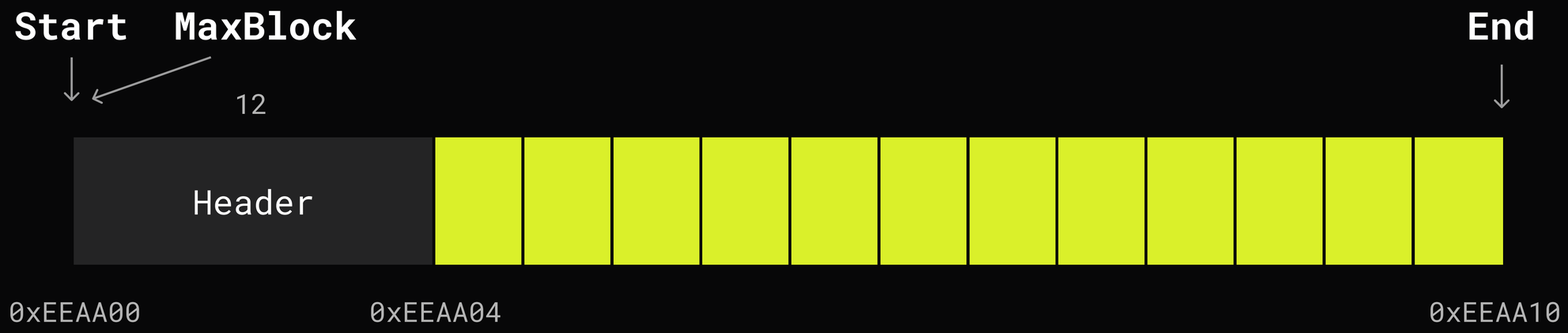
FreeBlocks = {0xFFA06}

ВЫДЕЛЯЕМ 6 БАЙТ



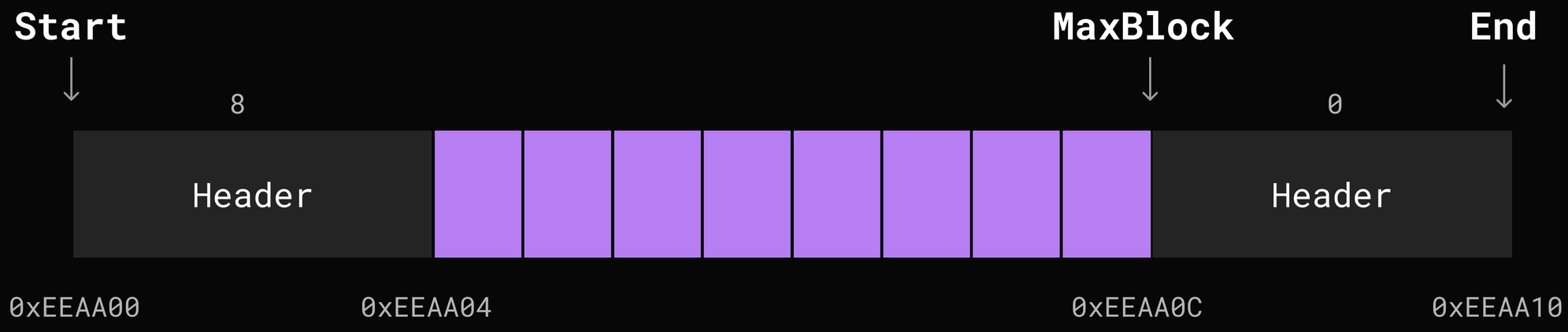
FreeBlocks = {}

FreeBlocks = {}



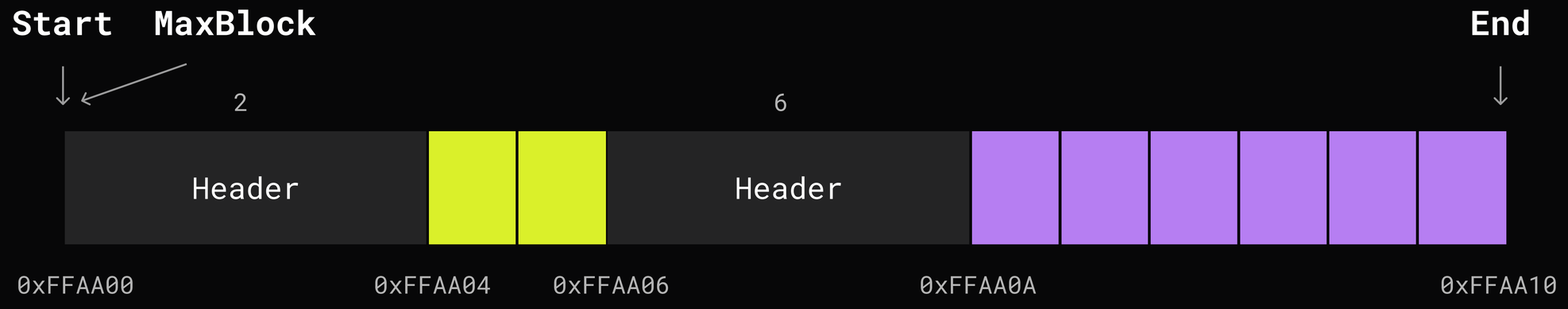
FreeBlocks = {0xEEAA00}

FreeBlocks = {}

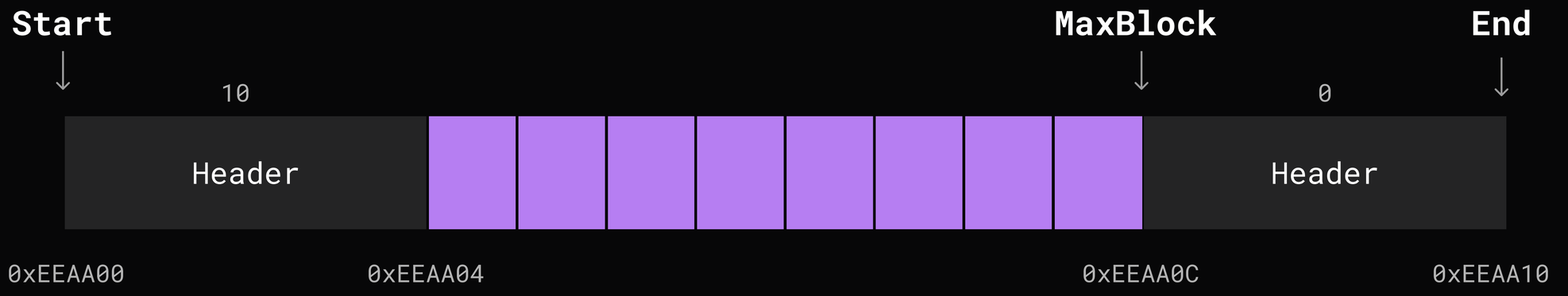


FreeBlocks = {0xEEAA0C}

ОСВОБОЖДАЕМ 2 БАЙТА

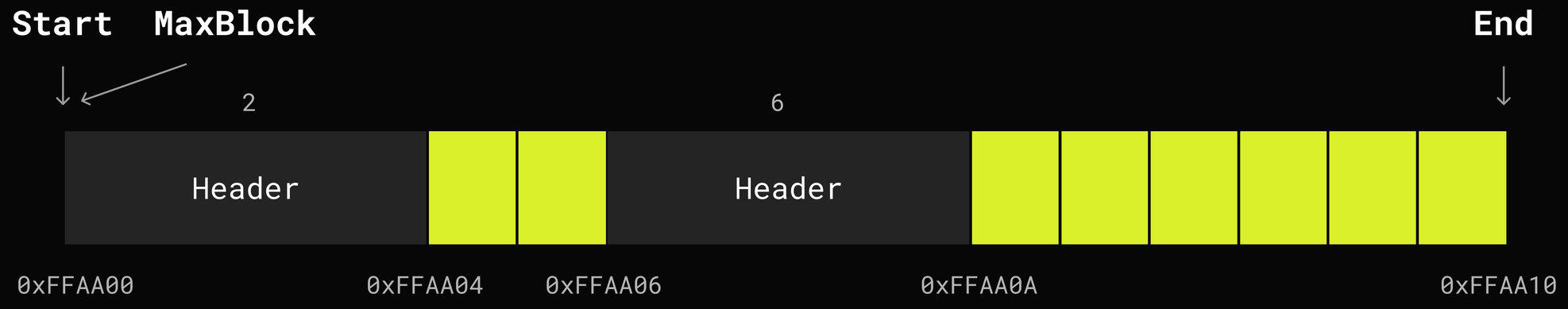


FreeBlocks = {0xFFAA00}

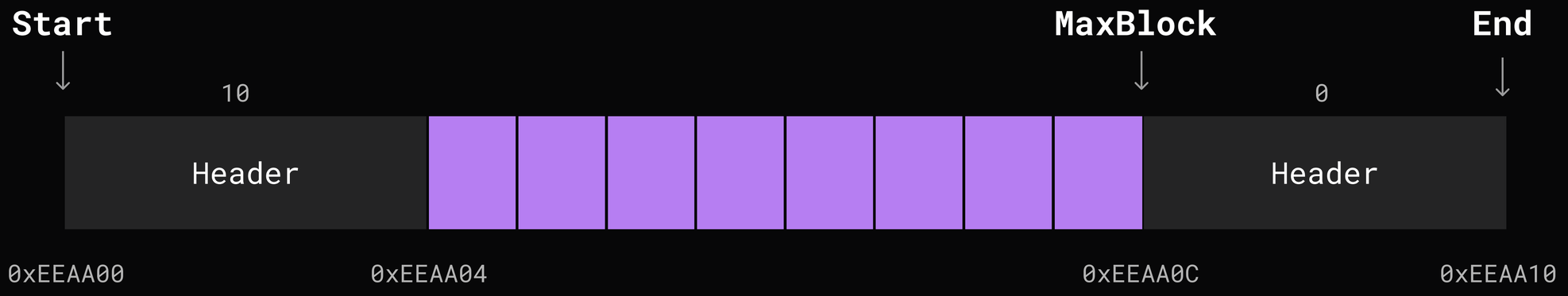


FreeBlocks = {0xEEAA0C}

ОСВОБОЖДАЕМ 6 БАЙТ

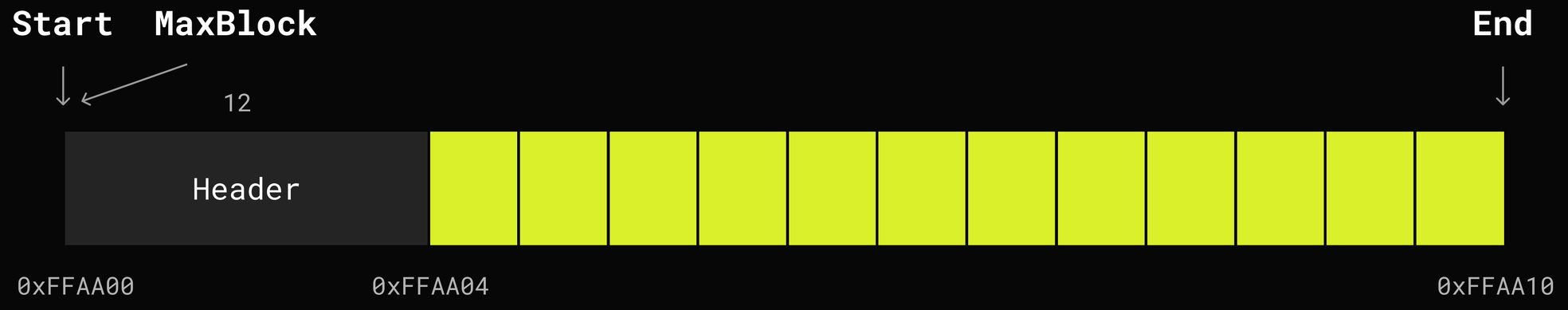


FreeBlocks = {0xFFAA00, 0XFFAA06}

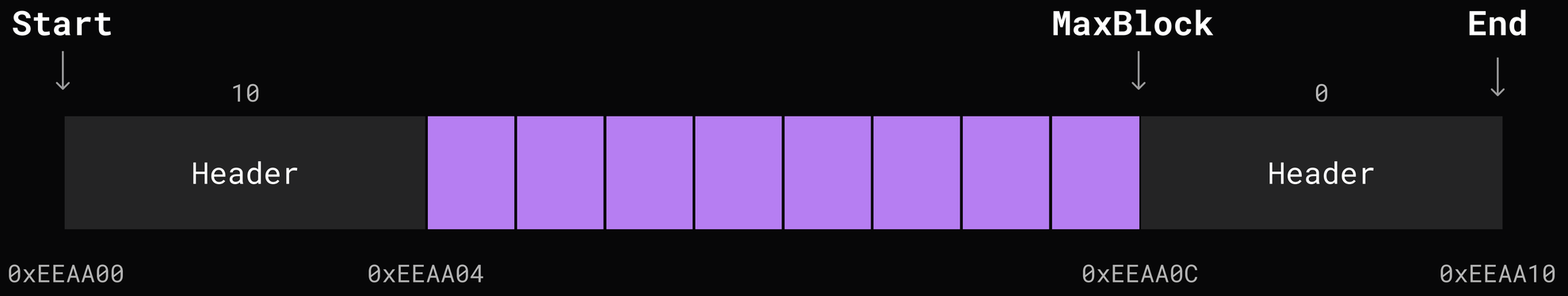


FreeBlocks = {0xEEAA0C}

ДЕФРАГМЕНТАЦИЯ



FreeBlocks = {0xFFAA00}



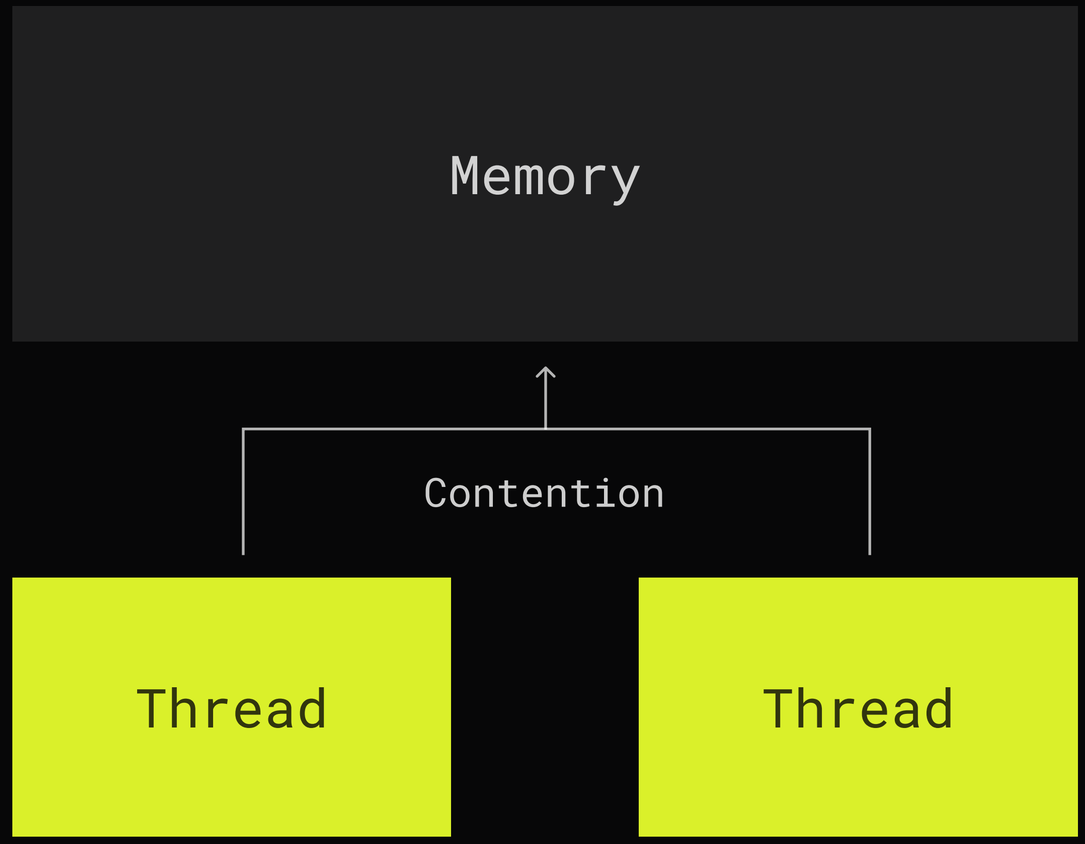
FreeBlocks = {0xEEAA0C}

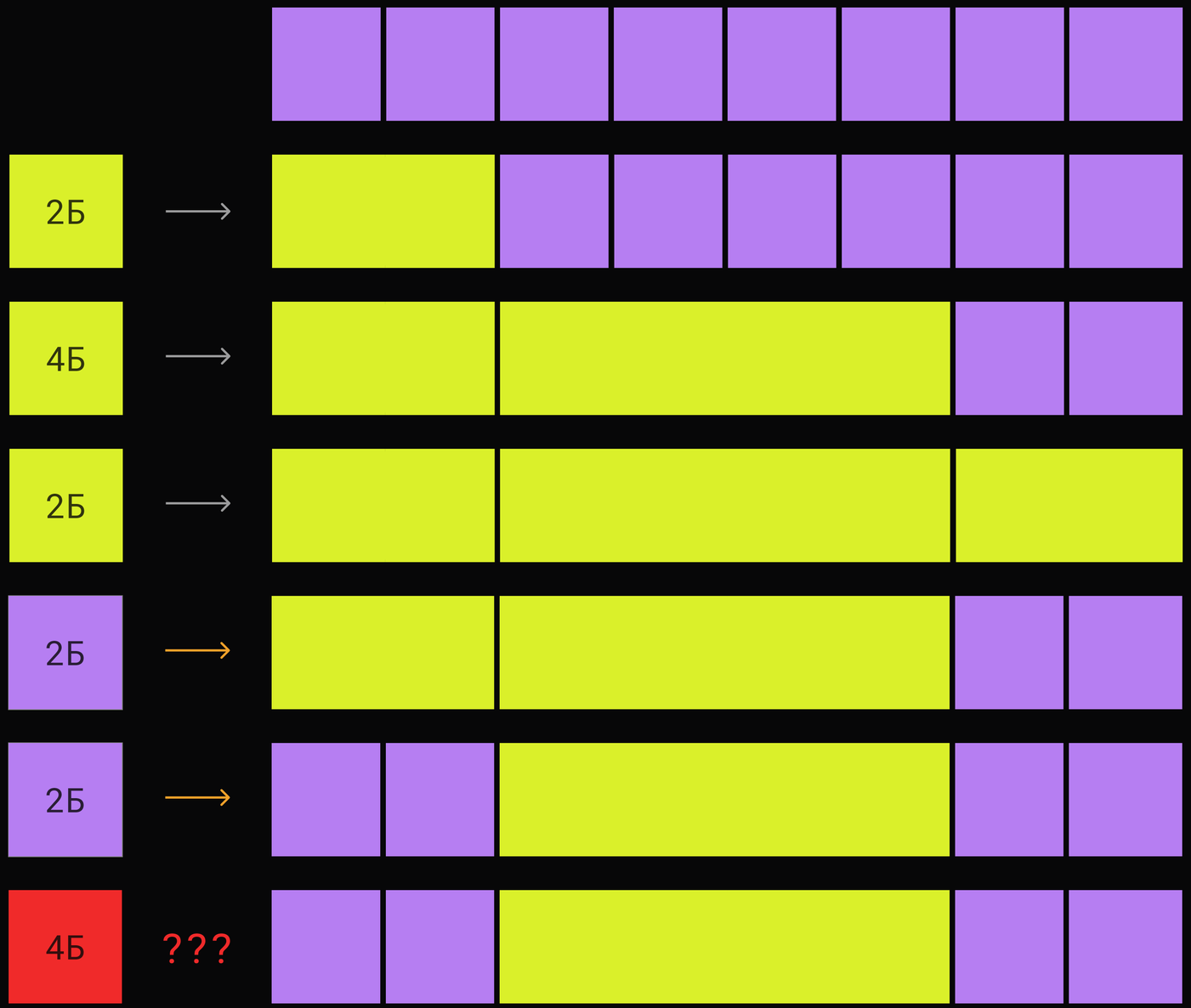
Terminal: question +

ПОЧЕМУ ВСЕ ТАК НЕ ЛЮБЯТ MALLOC?

MALLOC

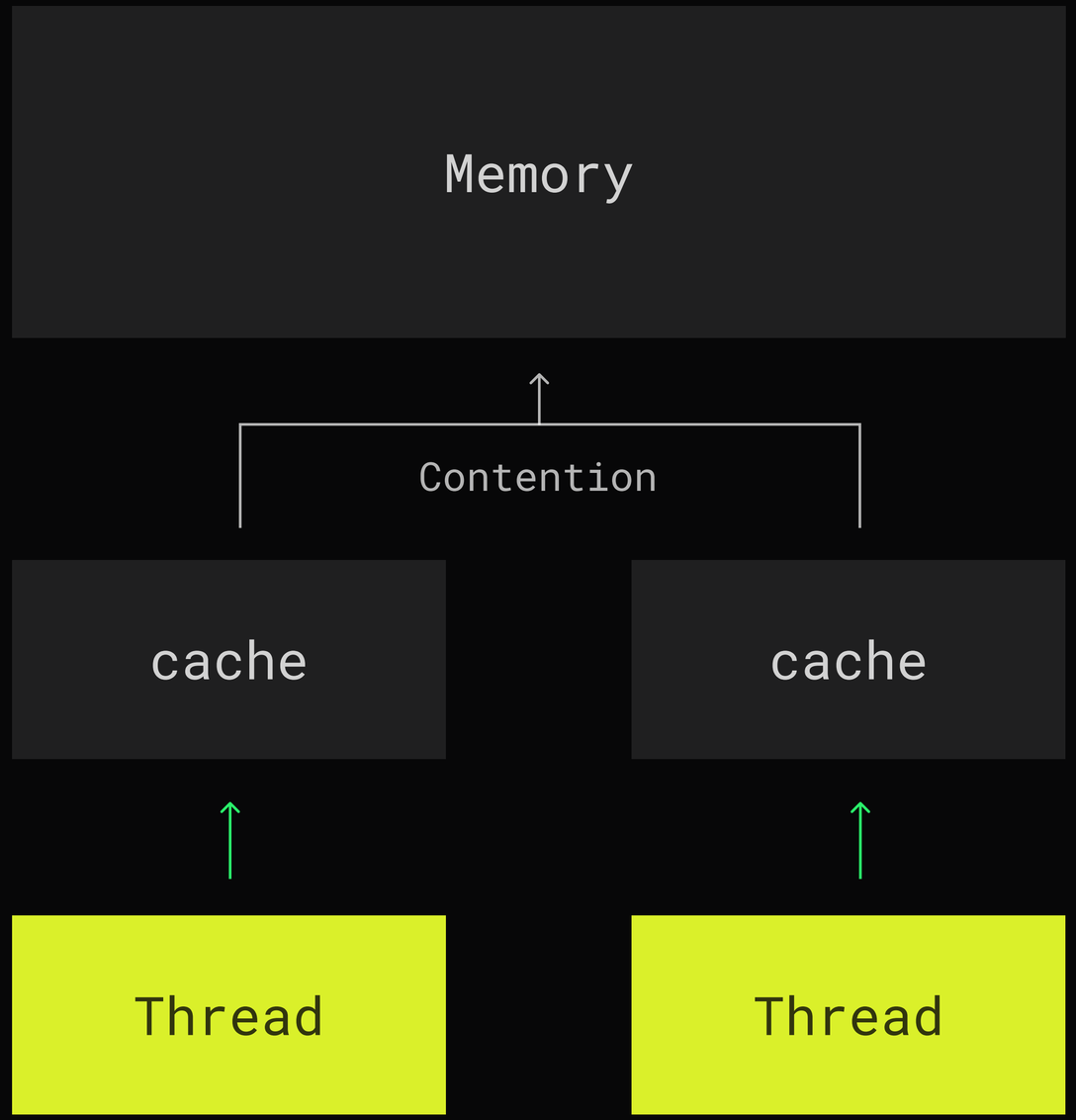
- Синхронизация потоков
- Фрагментация кучи

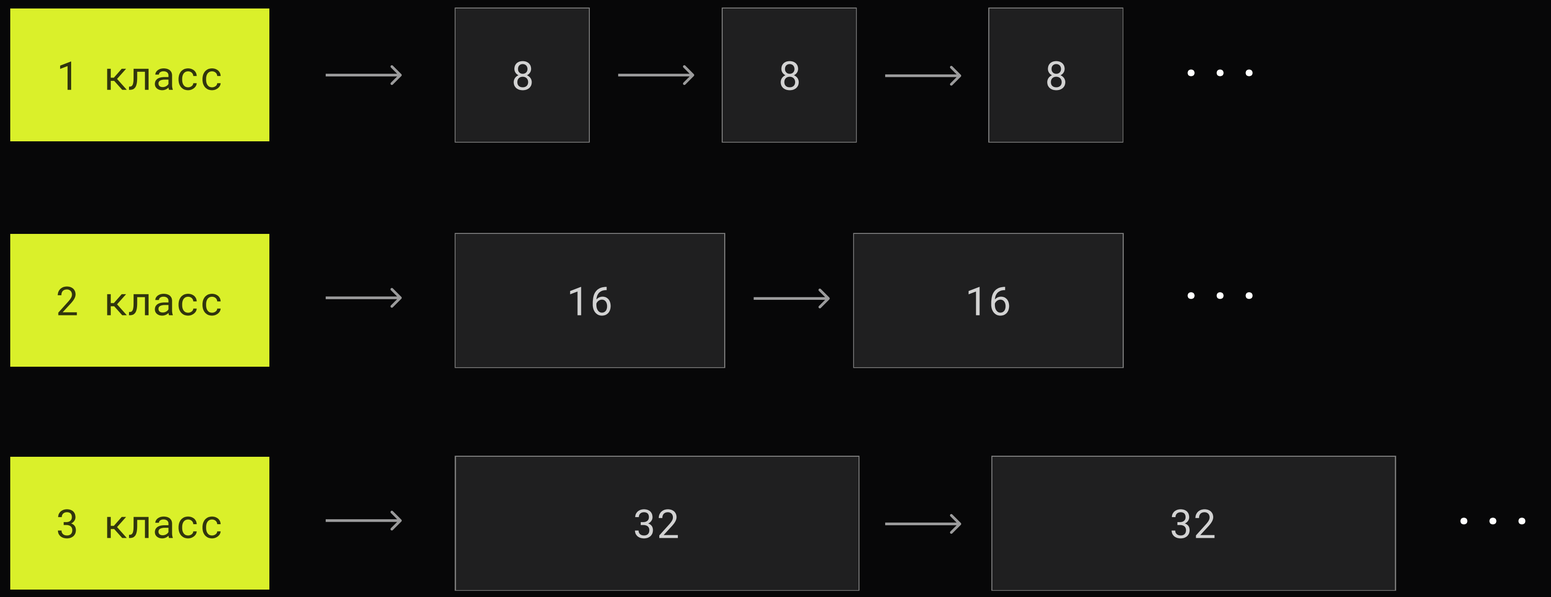



2Б
4Б
2Б
2Б
2Б
4Б
???

TCMALLOC

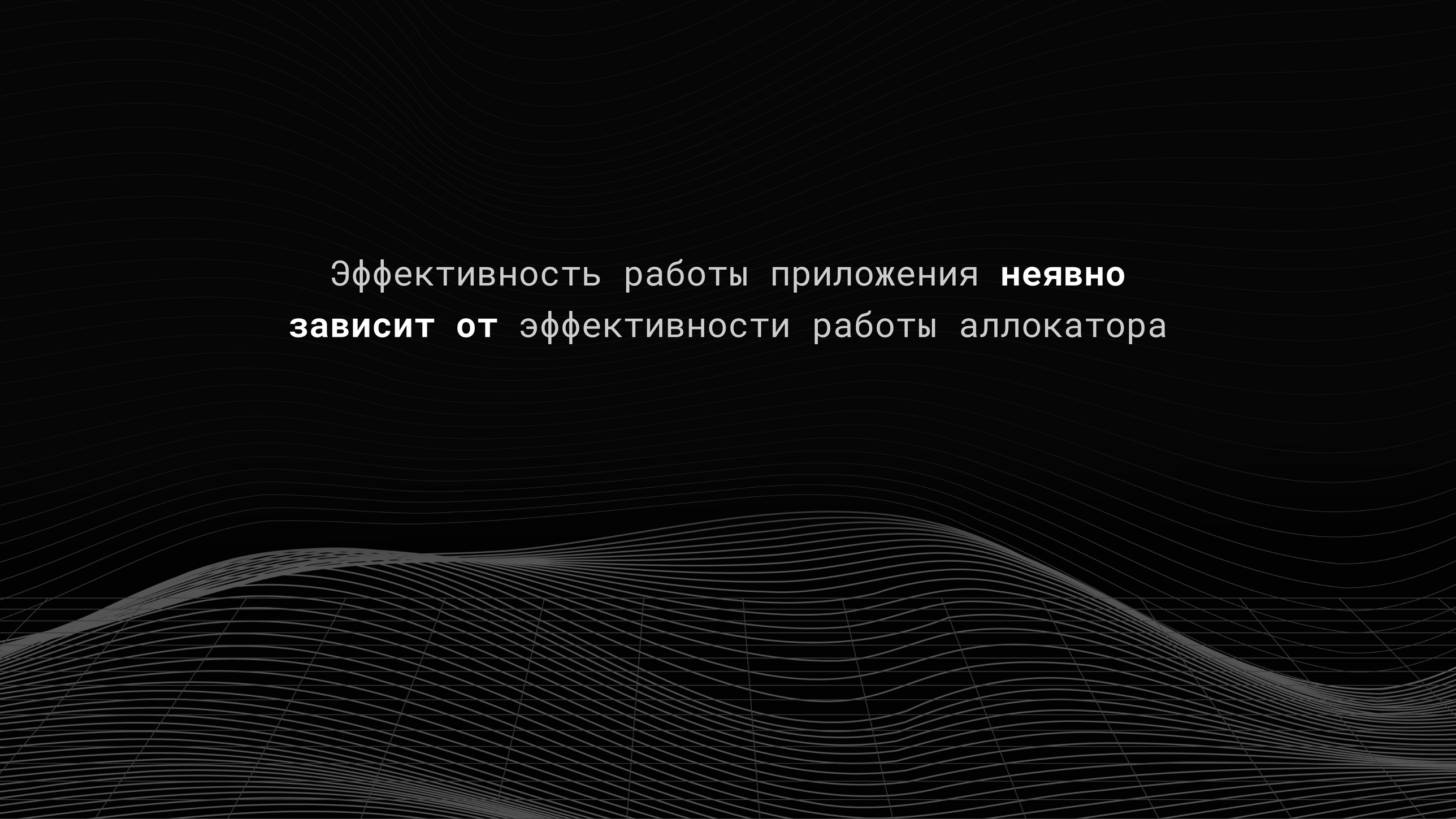
- Кэши для потоков
- Иерархическая модель хранения данных




1 класс
8
8
8
2 класс
16
16
3 класс
32
32

УСТРОЙСТВО АЛЛОКАТОРА GO

Эффективность работы приложения **неявно**
зависит от эффективности работы аллокатора



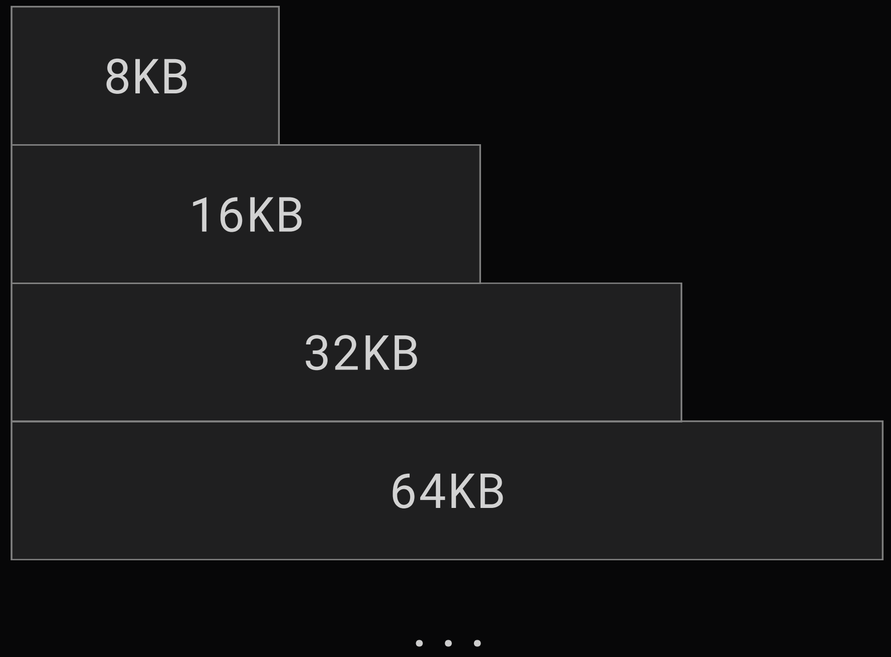
В Go, в случае необходимости, запрашивает память у операционной системы большими кусками (аренами), но размер арен может отличаться

Arena (64MB)

Из страниц арены Go выстраивает
иерархическую модель хранения данных,
образуя при этом спаны разного размера.

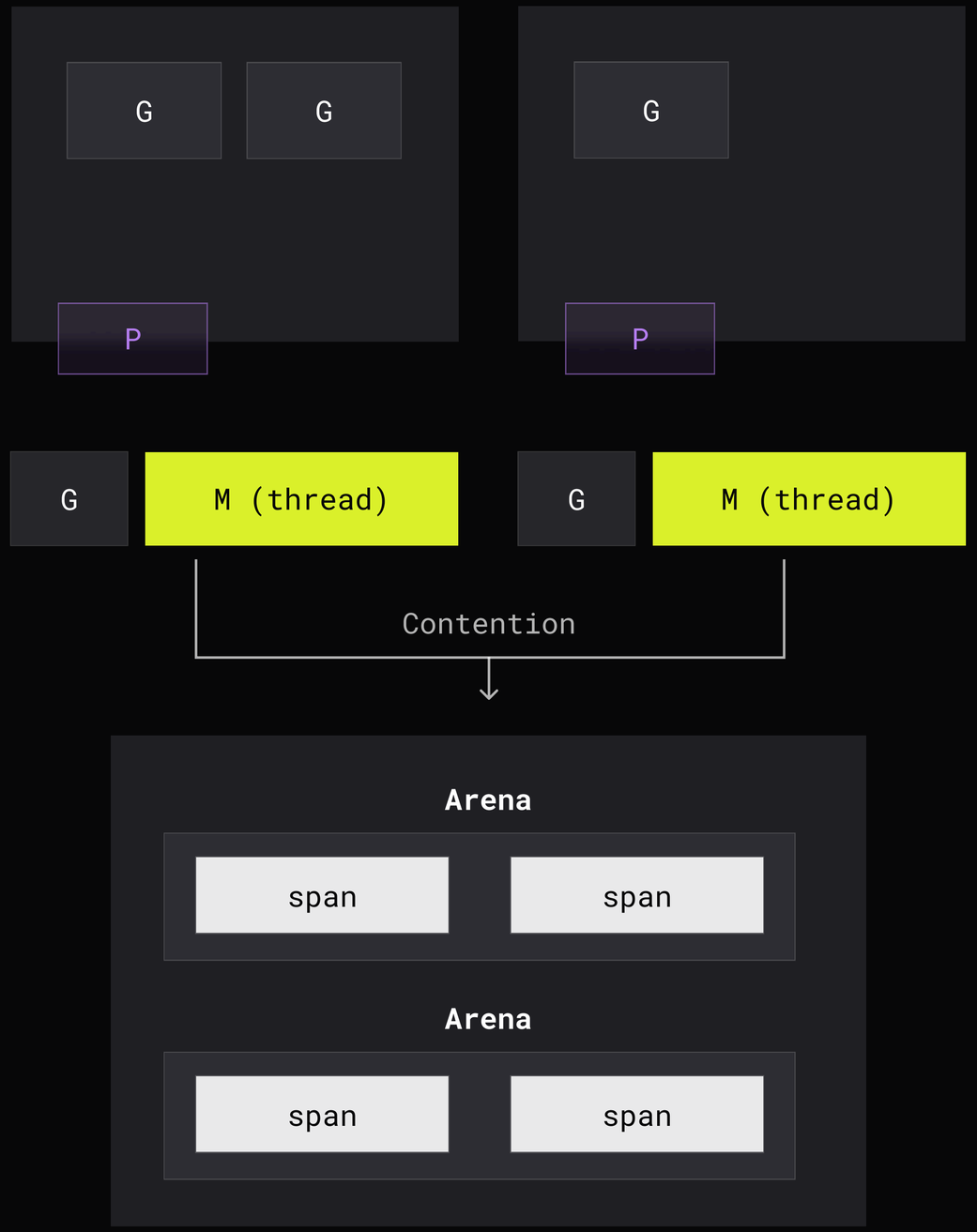
Arena (64MB)

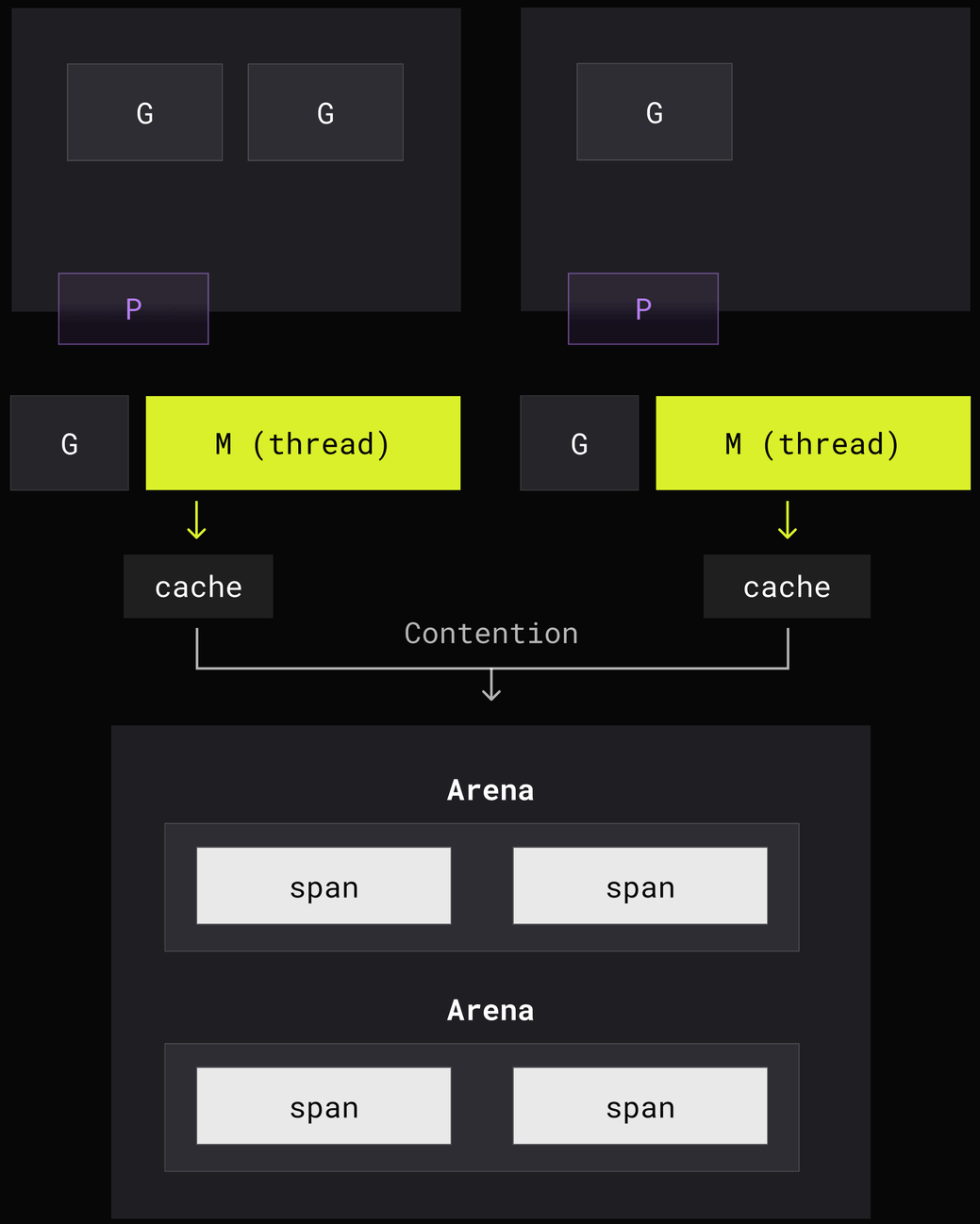
Spans



Terminal: question

КАК ПРОИСХОДИТ ПРОЦЕСС ВЫДЕЛЕНИЯ ПАМЯТИ?

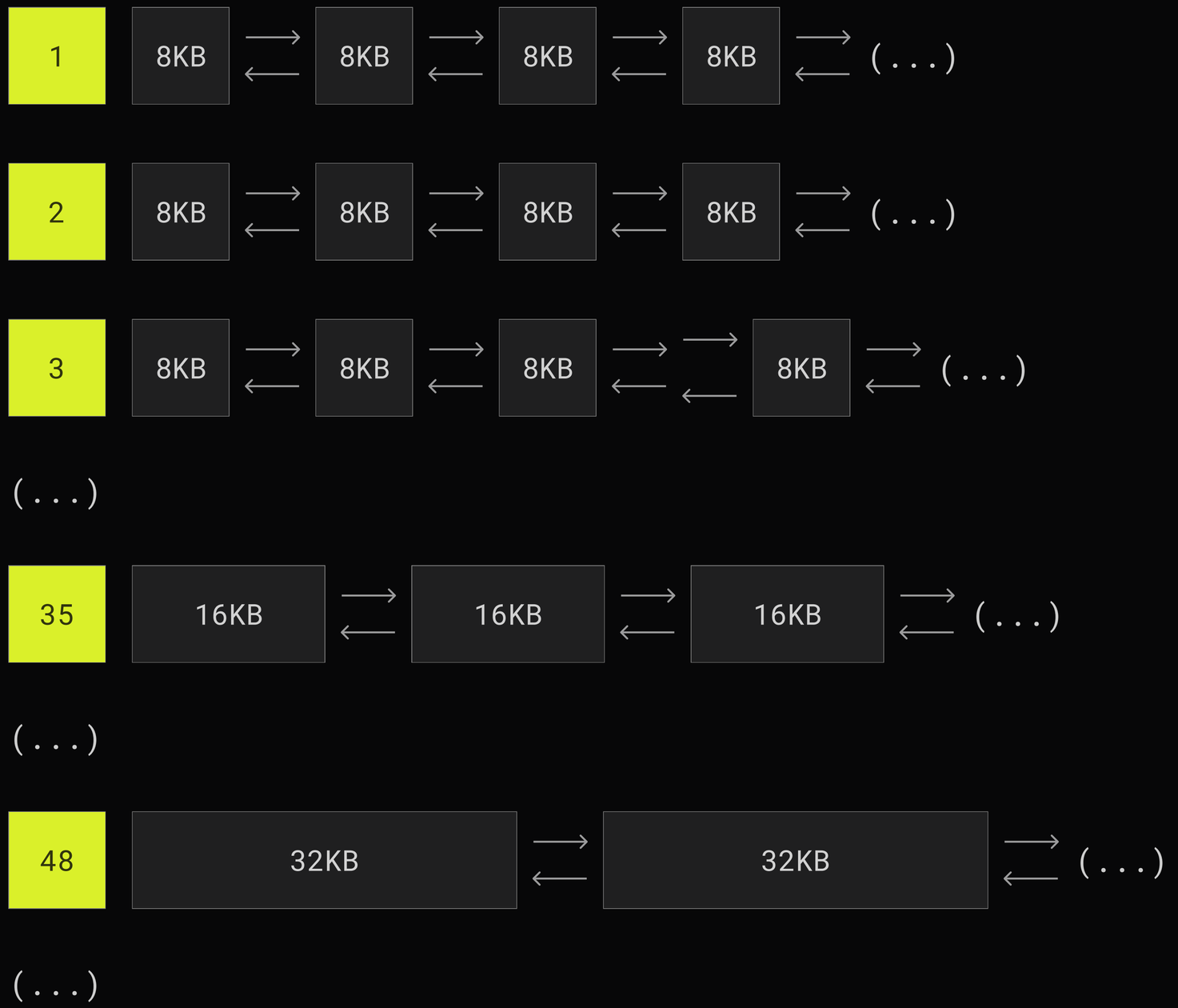

G
G
G
P
P
G
M (thread)
G
M (thread)
Contention
Arena
span
span
Arena
span
span


G
G
G
P
P
G
M (thread)
G
M (thread)
cache
cache
Contention
Arena
span
span
Arena
span
span

Terminal: question

ЗА СЧЕТ ЧЕГО ДОСТИГАЕТСЯ ПРЕДСКАЗУЕМАЯ ФРАГМЕНТАЦИЯ?

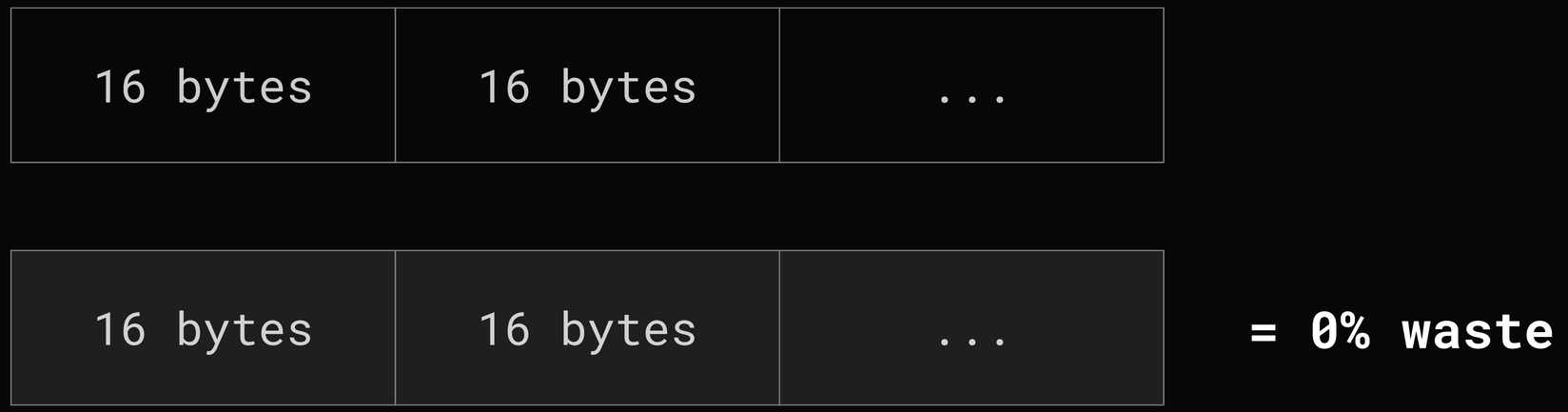
КЛАССЫ РАЗМЕРОВ



```
// class  bytes/obj  bytes/span  objects  tail waste  max waste  min align
//     1          8        8192     1024           0     87.50%          8
//     2         16        8192      512           0     43.75%         16
//     3         24        8192      341           8     29.24%          8
//     4         32        8192      256           0     21.88%         32
//     5         48        8192      170          32     31.52%         16
//     6         64        8192      128           0     23.44%         64
//     7         80        8192      102          32     19.07%         16
//     8         96        8192       85          32     15.95%         32
//     9        112        8192       73          16     13.56%         16
//    10        128        8192       64           0     11.72%        128
//    11        144        8192       56         128     11.82%         16
//    12        160        8192       51          32      9.73%         32
//    13        176        8192       46          96      9.59%         16
//    14        192        8192       42         128      9.25%         64
//    15        208        8192       39          80      8.12%         16
//    16        224        8192       36         128      8.15%         32
// [...]
//    67      32768       32768        1           0     12.50%       8192
```

```
1 var data [16]byte
```

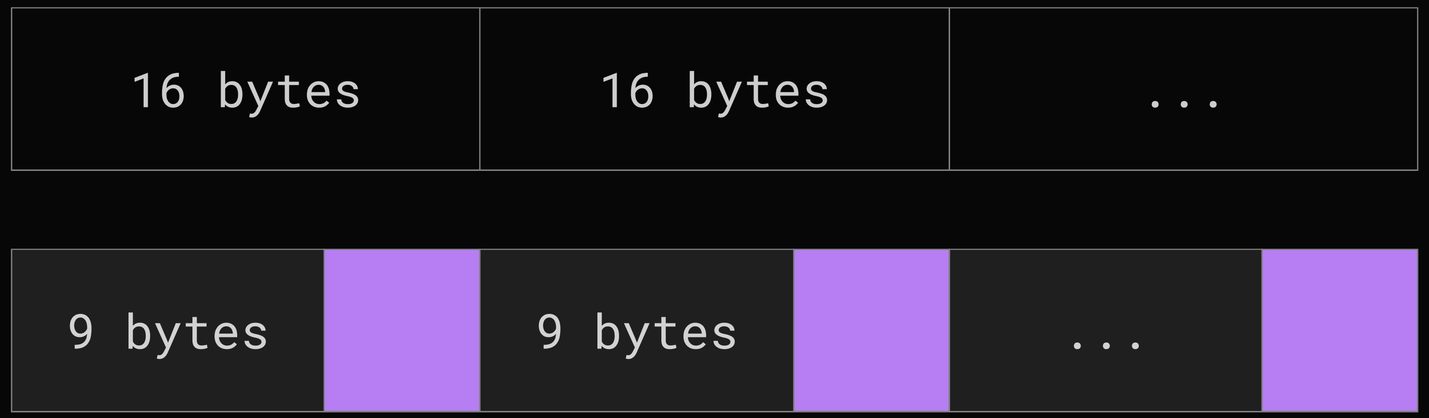
СПАН ВТОРОГО КЛАССА



= 0% waste

```
1 var data [9]byte
```

СПАН ВТОРОГО КЛАССА



= (16-9)/16
= 43.75% waste

EXAMPLE

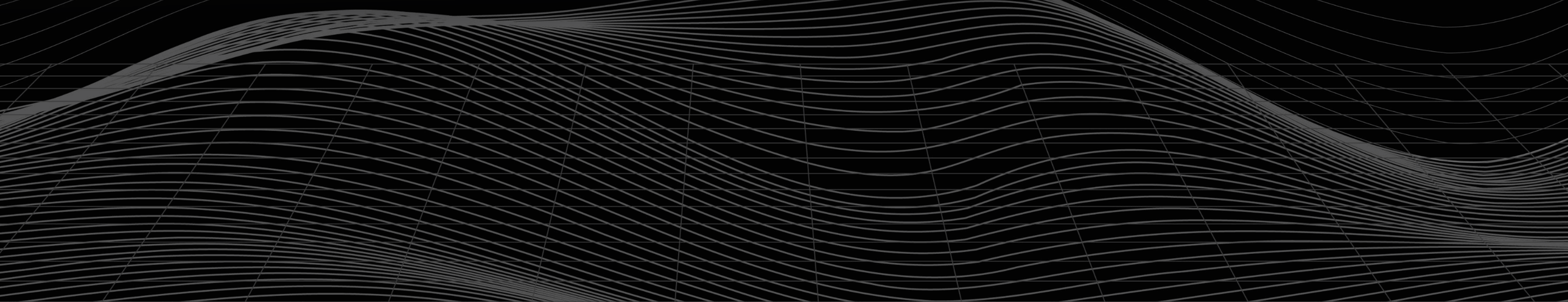
Allocations size

EXAMPLE

Allocations offset

Иногда бывает полезно помнить про классы объектов:
чем больше объектов — тем потенциально больший профит можно получить от правильно подогнанных размеров.

Но не стоит на этом зацикливаться: **в большинстве случаев читаемый и понятный код будет лучшим решением, чем оптимизированный, но менее понятный.**



ПЕРЕРЫВ 5 МИНУТ

АЛЛОЦИРОВАНИЕ ОБЪЕКТОВ

Terminal: deep_go

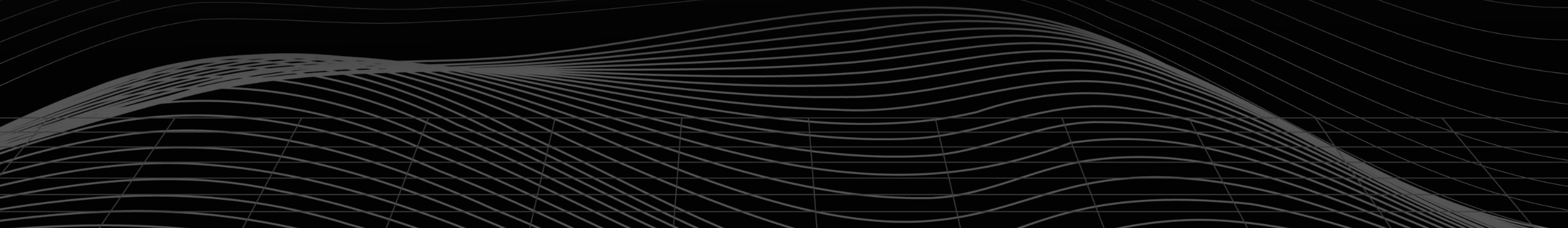
ESCAPE ANALYSIS

Процесс во время компиляции программы, который определяет где в программе создаются объекты и как они используются

Определяет, нужно ли выделять память для объекта в куче или можно разместить его в стеке

The storage location does have an effect on writing efficient programs. When possible, the Go compilers will allocate variables that are local to a function in that function’s stack frame.

However, if the compiler cannot prove that the variable is not referenced after the function returns, then the compiler must allocate the variable on the garbage-collected heap to avoid dangling pointer errors. Also, if a local variable is very large, it might make more sense to store it on the heap rather than the stack.



```
1     // MaxStackVarSize is the maximum size variable which we will allocate on the stack.
2     // This limit is for explicit variable declarations like "var x T" or "x := ...".
3     // Note: the flag smallframes can update this value.
4     MaxStackVarSize = int64(10 * 1024 * 1024)
5
6     // MaxImplicitStackVarSize is the maximum size of implicit variables that we will allocate on the stack.
7     //   p := new(T)          allocating T on the stack
8     //   p := &T{}            allocating T on the stack
9     //   s := make([]T, n)    allocating [n]T on the stack
10    //   s := []byte("...")   allocating [n]byte on the stack
11    // Note: the flag smallframes can update this value.
12    MaxImplicitStackVarSize = int64(64 * 1024)
```

EXAMPLE

Allocation 1

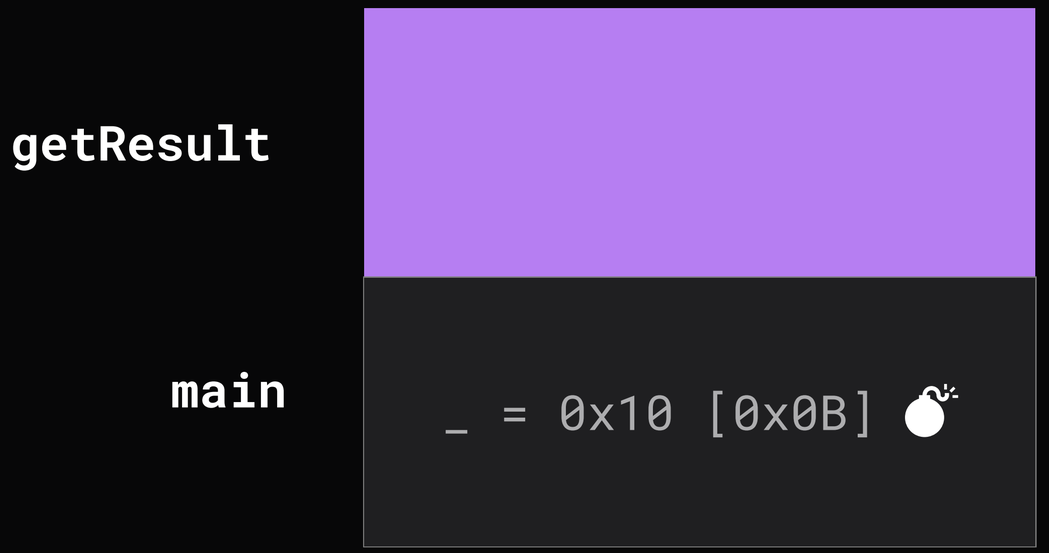

getResult
result = 200 [0x10]
main


getResult
main
_ = 200 [0x0B]

EXAMPLE

Allocation 2

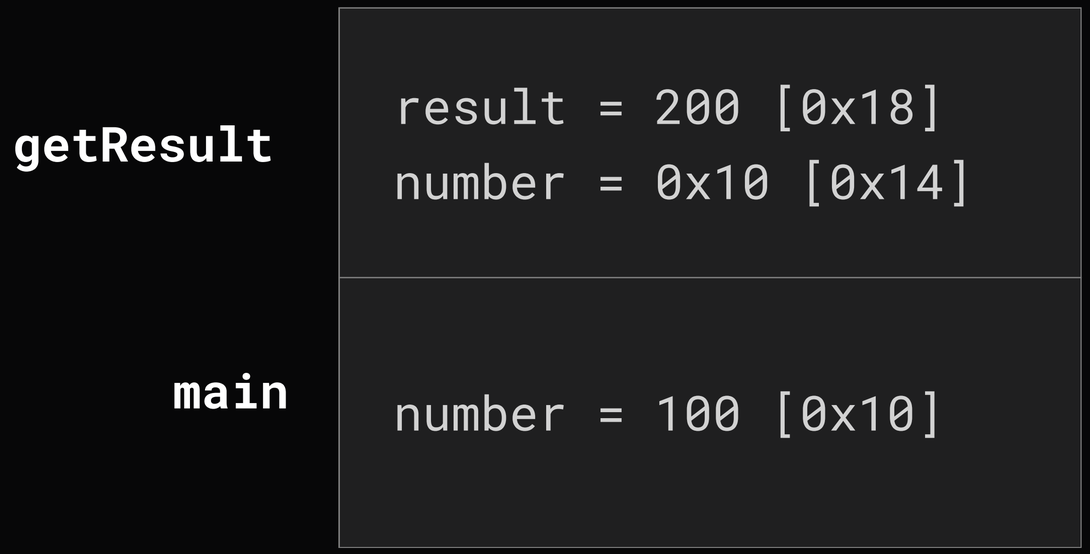

getResult
result = 200 [0x10]
main

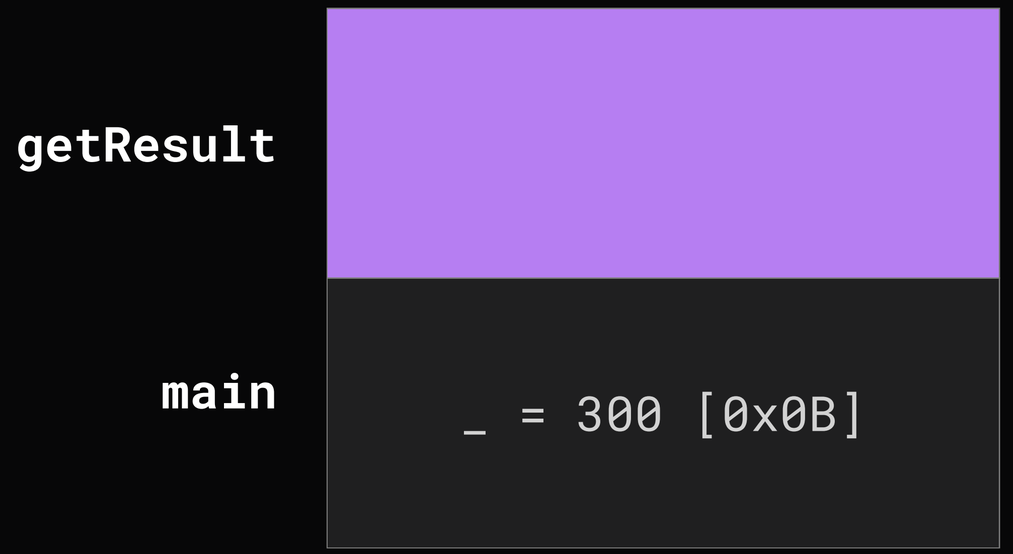

getResult
main
_ = 0x10 [0x0B]


HEAP
200 [0xBB]
getResult
result = 0xBB [0x10]
main
getResult
main
_ = 0xBB [0x0A]

EXAMPLE

Allocation 3


getResult
result = 200 [0x18]
number = 0x10 [0x14]
main
number = 100 [0x10]

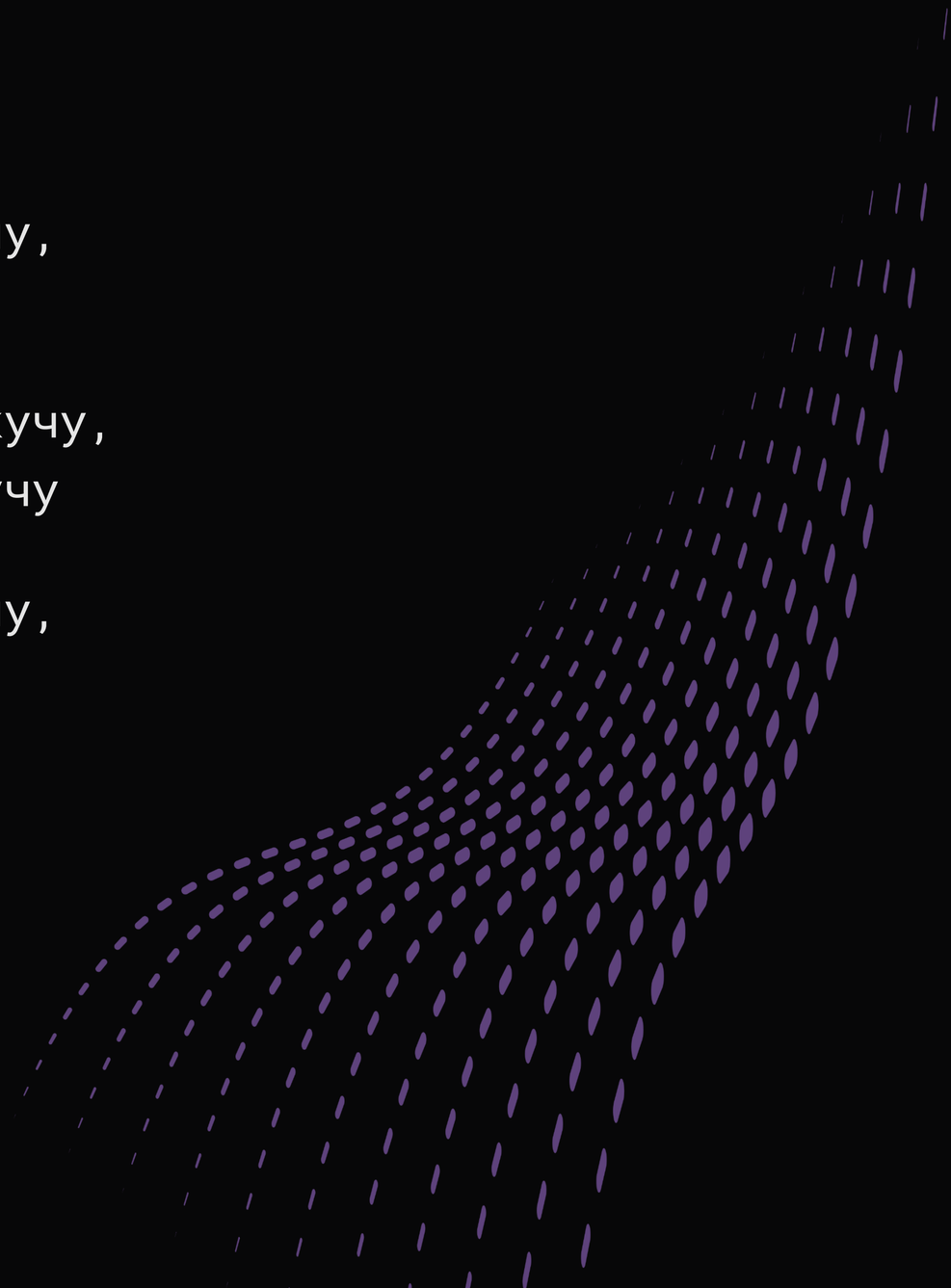

getResult
main
_ = 300 [0x0B]

EXAMPLE

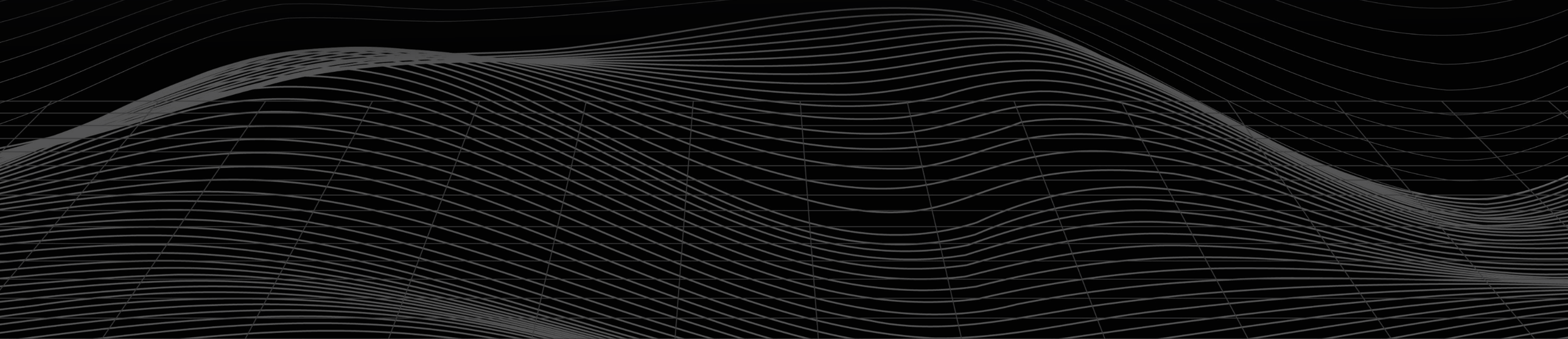
Allocation 4

НЕКОТОРЫЕ ФАКТЫ

- если одно из полей структуры «перемещается» в кучу, то вся структуры также «перемещается» в кучу
- если один из элементов массива «перемещается» в кучу, то все элементы массива также «перемещается» в кучу
- если один из элементов среза «перемещается» в кучу, то все элементы также «перемещаются» в кучу



Объект, созданный путем вызова функции new(), может быть проаллацирован **не только в куче, но еще и в стеке**

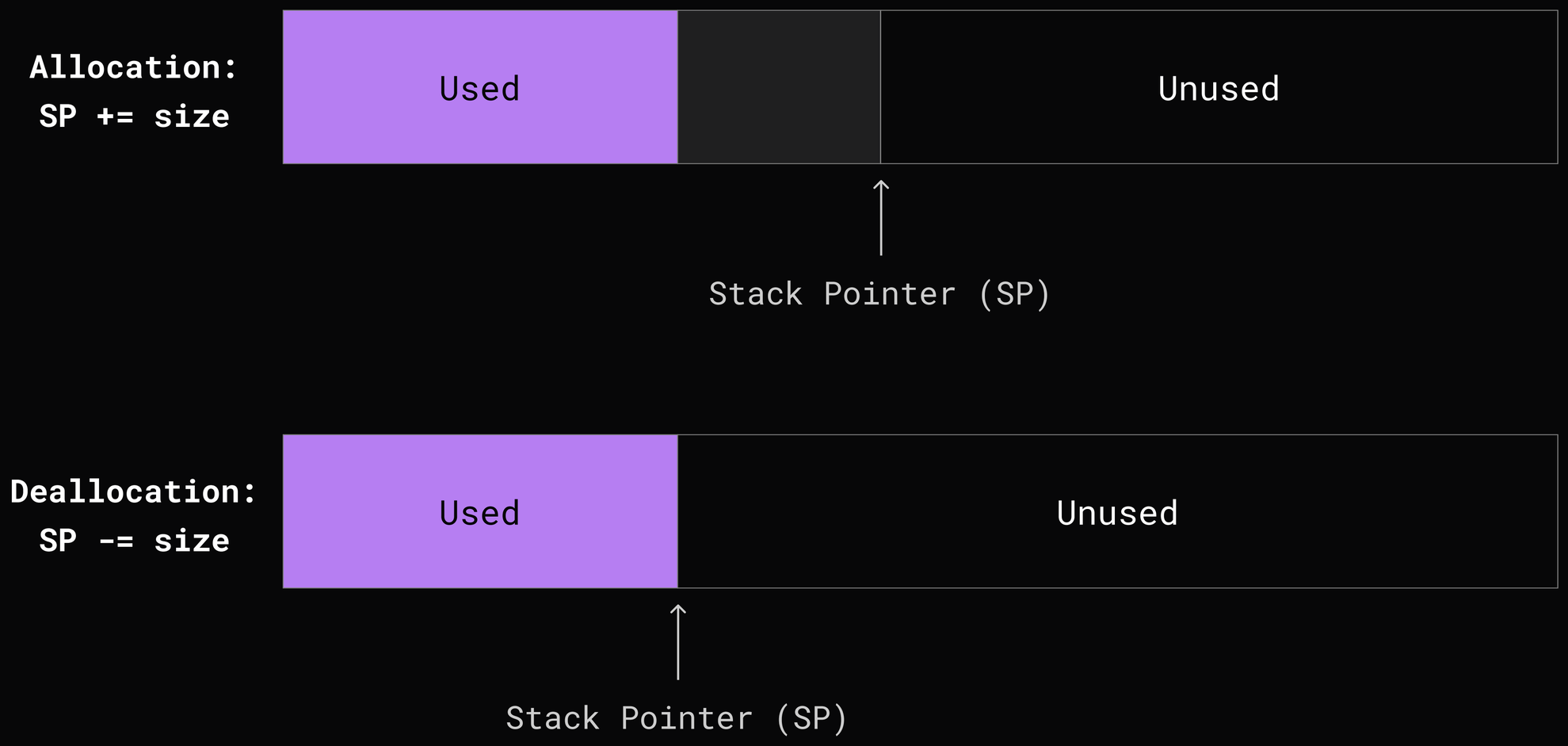


EXAMPLE

Allocation 5

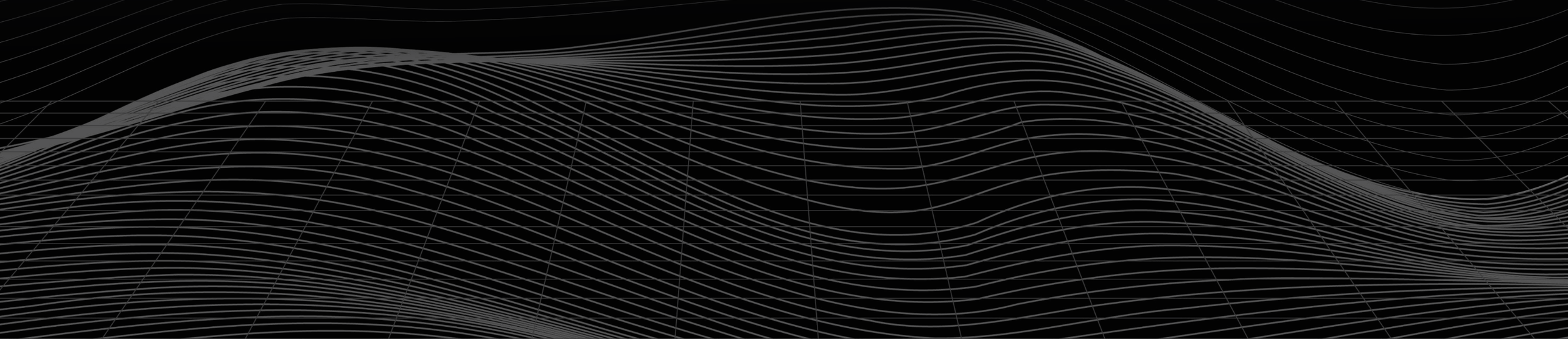
Terminal: question +  

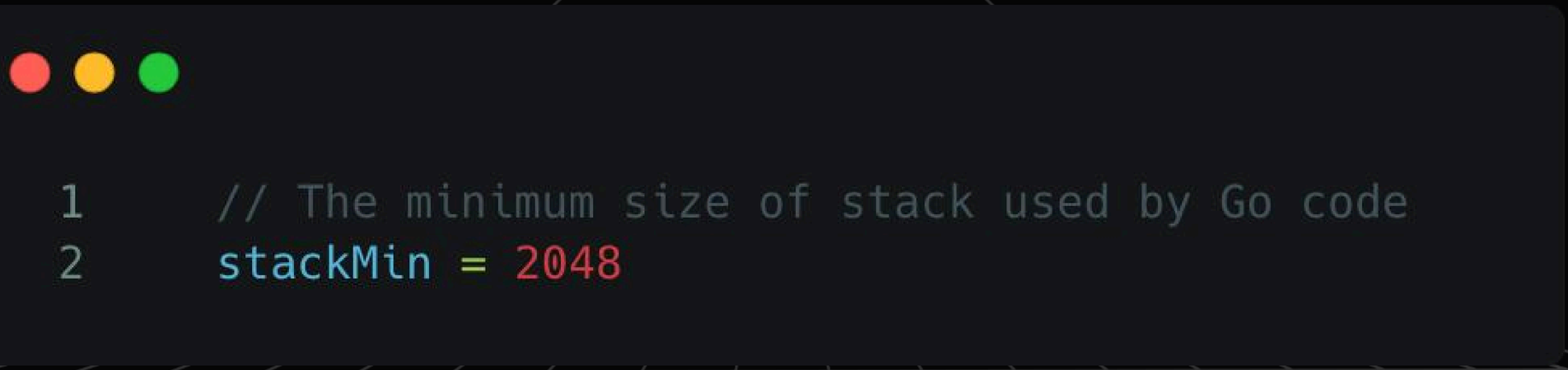
ПОЧЕМУ АЛЛОЦИРОВАНИЕ ОБЪЕКТОВ В СТЕКЕ ПРОИСХОДИТ БЫСТРЕЕ, ЧЕМ В КУЧЕ?


Allocation:
SP += size
Used
Unused
Stack Pointer (SP)
Deallocation:
SP -= size
Used
Unused
Stack Pointer (SP)

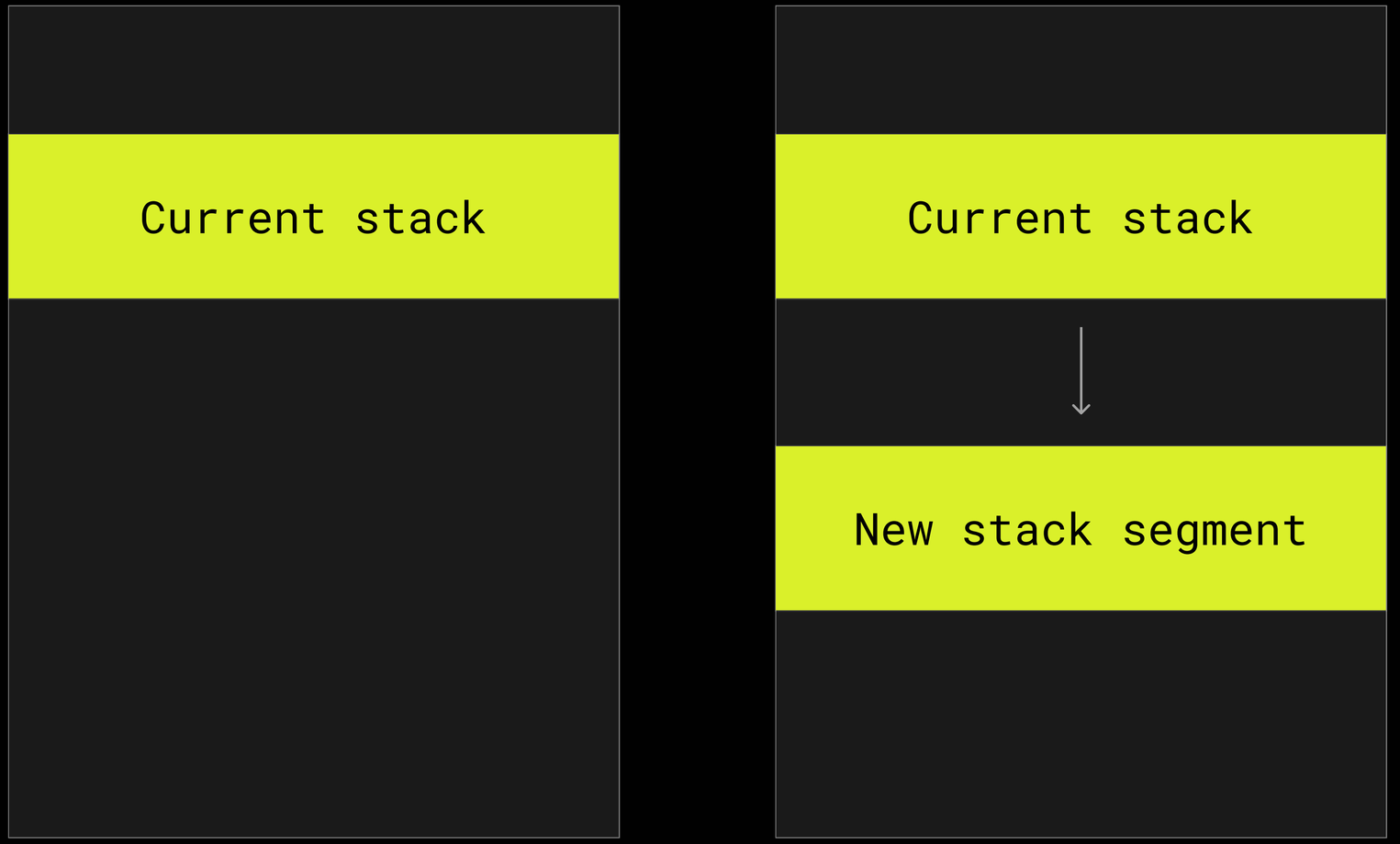
В Go есть некоторые накладные расходы,

когда стек растет или уменьшается

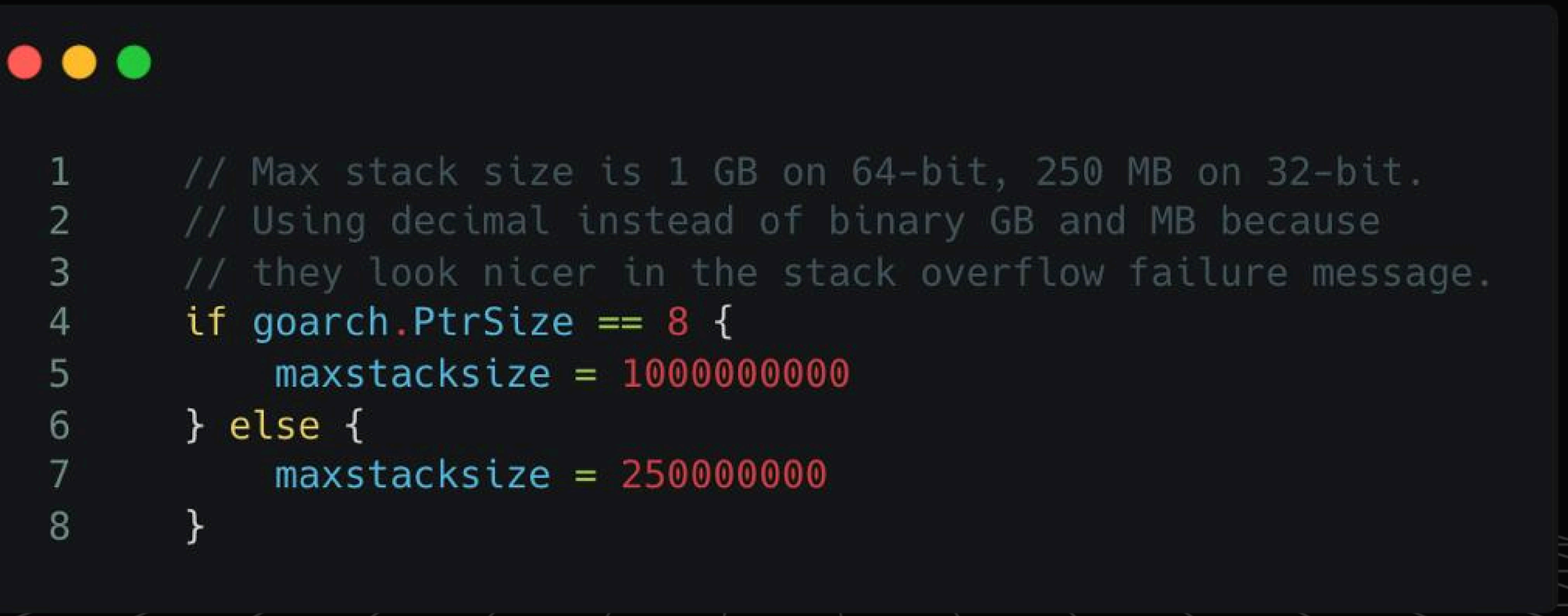



// The minimum size of stack used by Go code
stackMin = 2048

SEGMENTED STACK



CONTIGUOUS STACK


// Max stack size is 1 GB on 64-bit, 250 MB on 32-bit.
// Using decimal instead of binary GB and MB because
// they look nicer in the stack overflow failure message.
if goarch.PtrSize == 8 {
maxstacksize = 1000000000
} else {
maxstacksize = 250000000
}

EXAMPLE

Stack growth

Тем не менее, в целом, аллокации на стеке работают быстрее, чем в куче, потому что стек растет и уменьшается не так часто, а сам алгоритм распределения данных на стеке очень прост и более дружелюбен для кэшей процессора

КРОМЕ ТОГО

- **мы не грузим GC дополнительной работой**
- **стековую память не нужно освобождать по частям**

Стек горутины фактически можно рассматривать как один блок памяти, поэтому он будет освобожден, как единое целое, при завершении работы горутины

GC не занимается стеками горутин

Terminal: question + ⚙ –

ЗАЧЕМ ЭТО НУЖНО ЗНАТЬ?

EXAMPLE

Allocation effect 1

EXAMPLE

Allocation effect 2

EXAMPLE

Allocation effect 3

EXAMPLE

Loop allocations

Terminal: question + ⚙ –

КАК УМЕНЬШИТЬ КОЛИЧЕСТВО АЛЛОКАЦИЙ?

OBJECT POOL

Набор инициализированных и готовых к использованию объектов:

- когда системе требуется объект, он не создаётся, а берётся из пула
- когда объект больше не нужен, он не уничтожается, а возвращается в пул

API

```go
type Pool struct {
    New func() any
}

func (p *Pool) Get() any  // забирает объект из пула
func (p *Pool) Put(x any) // кладет объект в пул
```

EXAMPLE

Pool

```
p1 := pool.Get()
p2 := pool.Get()
p3 := pool.Get()
```

Memory in use

Memory pool

Garbage

```
p1 := pool.Get()
p2 := pool.Get()
p3 := pool.Get()
pool.Put(p1)
pool.Put(p2)
```

Memory in use

Memory pool

Garbage

```
p1 := pool.Get()
p2 := pool.Get()
p3 := pool.Get()
pool.Put(p1)
pool.Put(p2)
p4 := pool.Get()
```

Memory in use

Memory pool

Garbage

```
p1 := pool. Get()
p2 := pool. Get()
p3 := pool. Get()
pool.Put(p1)
pool.Put(p2)
p4 := pool.Get()
// a little later
```

Memory in use

Memory pool

Garbage

Terminal: question + ⌄ ⚙ –

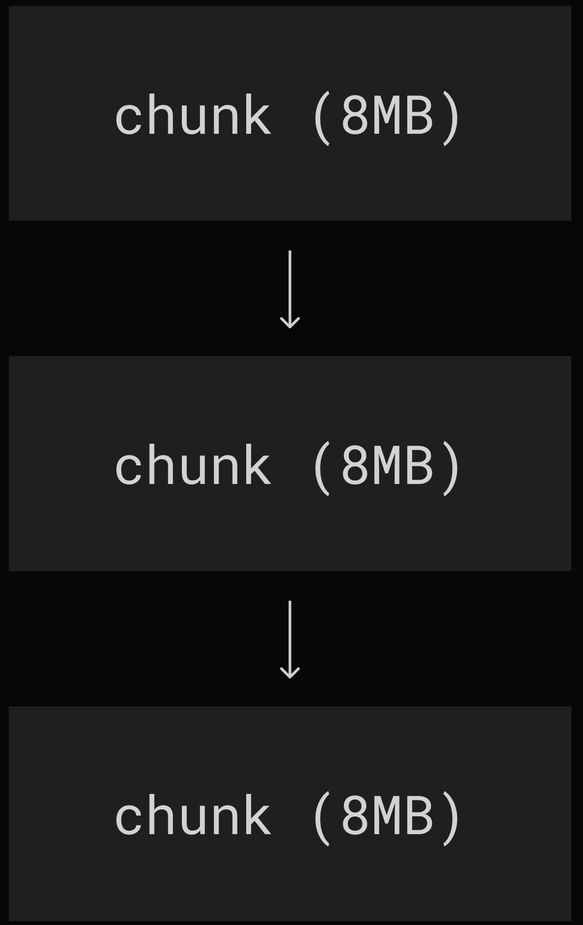
КАК ЕЩЕ МОЖНО УМЕНЬШИТЬ КОЛИЧЕСТВО АЛЛОКАЦИЙ?

REGION-BASED MEMORY MANAGEMENT

АРЕНЫ

Арены, как правило, предназначены
для батчевой обработки каких-либо данных.

Когда у нас в памяти много всего создается
по очереди и мы хотим очистить все одним махом,
не нагружая GC



EXAMPLE

Arena api

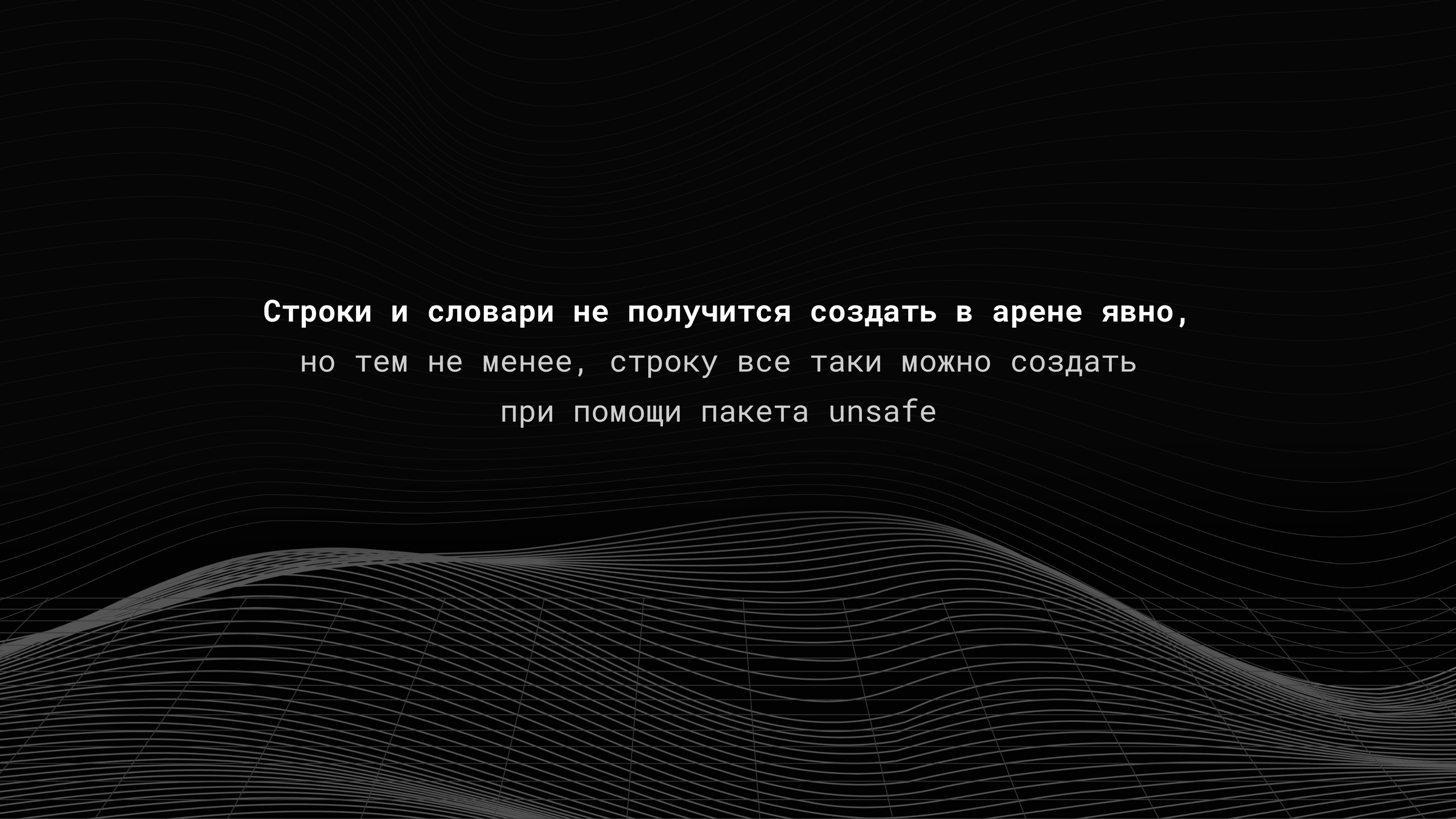
EXAMPLE

Arena problem 1

Строки и словари не получится создать в арене явно,

но тем не менее, строку все таки можно создать

при помощи пакета unsafe



EXAMPLE

Arena problem 2

EXAMPLE

Arena problem 3

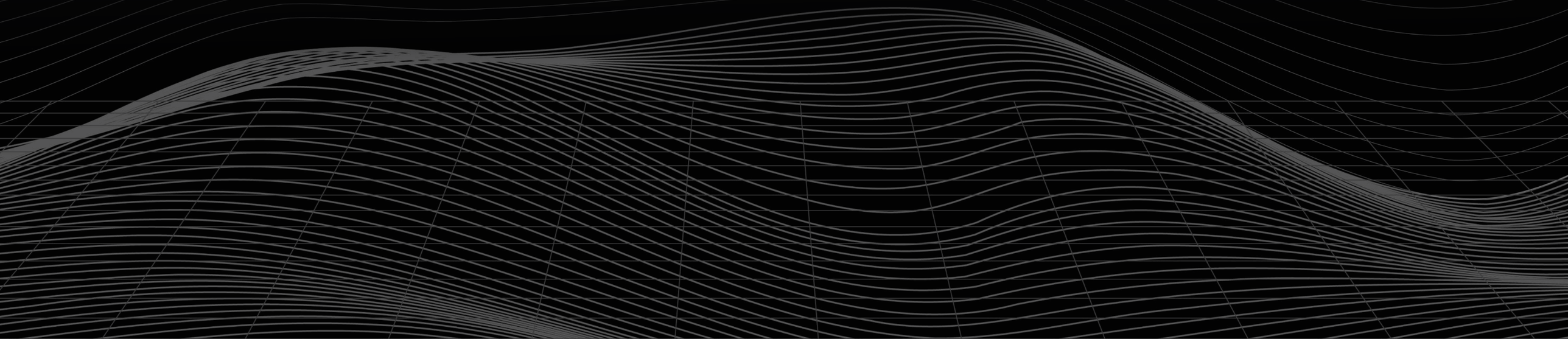
Memory sanitizer нужен для того, чтобы найти использование неинициализированной памяти, а адрес **address sanitizer** следит за обращением к адресам, которые не доступны (работает на Linux)

```
$ go help build
    ...
    -msan
            Link with C/C++ memory sanitizer support.
    -asan
            Link with C/C++ address sanitizer support.
    ...
```

В большинстве случаев читаемый и понятный код будет лучшим решением, чем оптимизированный, но более трудный для понимания

EXAMPLE

Big allocations



ПОЖАЛУЙСТА, ЗАПОЛНИ ОПРОС О ЗАНЯТИИ

Ссылка в чате и в группе участников