OSTEP
Virtualizing Memory
Paging

Questions answered in this lecture:

Review segmentation and fragmentation
What is paging?
Where are page tables stored?
What are advantages and disadvantages of paging?

Problem: Fragmentation

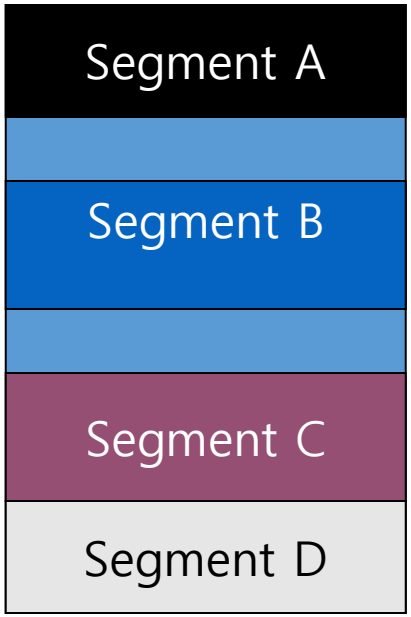
Definition: Free memory that can't be usefully allocated

Why?

- Free memory (hole) is too small and scattered
- Rules for allocating memory prohibit using this free space

Types of fragmentation

- External: Visible to allocator (e.g., OS)
- Internal: Visible to requester (e.g., if must allocate at some granularity)



External

No contiguous space!

Allocated to requester

Internal

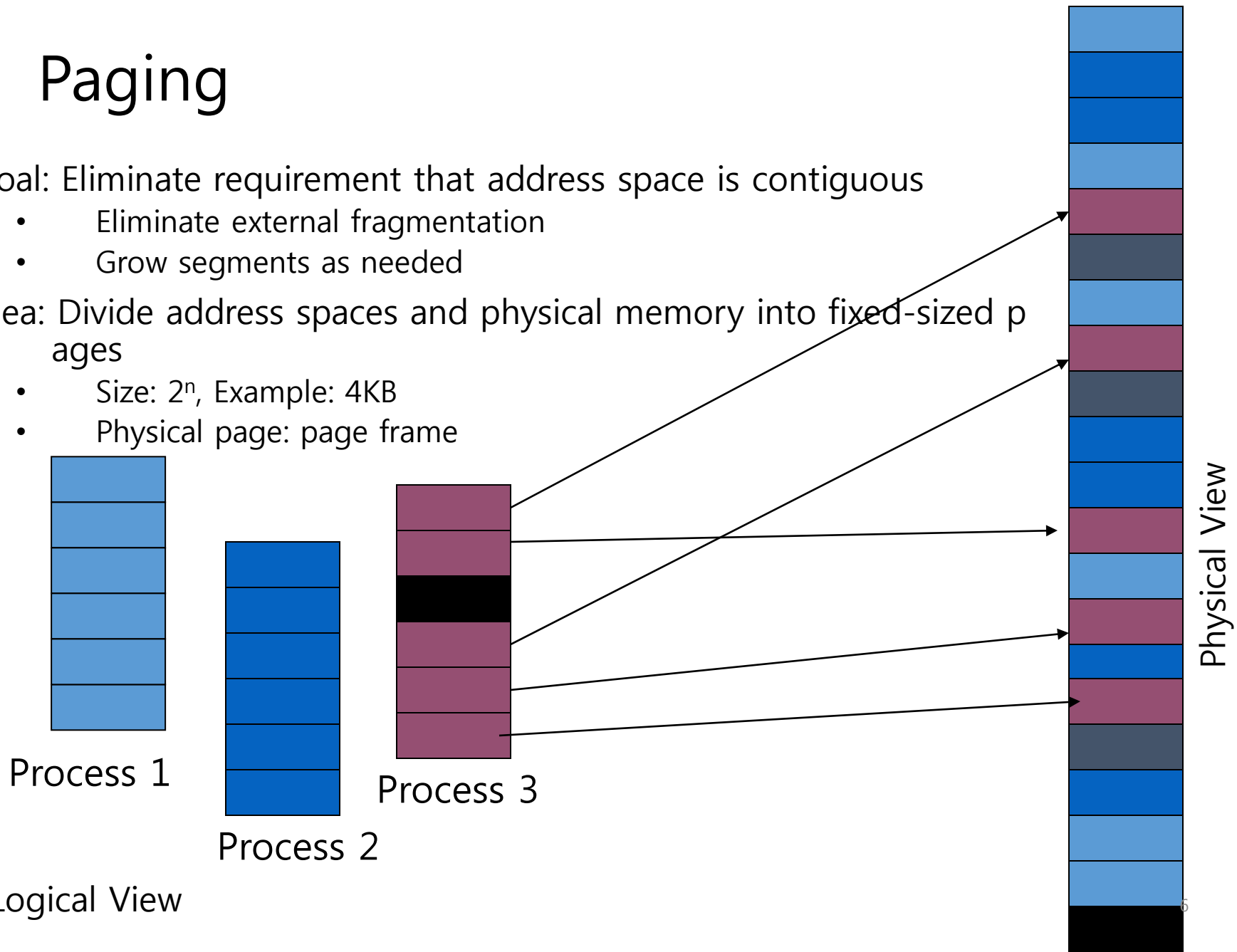
Paging

Goal: Eliminate requirement that address space is contiguous

- Eliminate external fragmentation
- Grow segments as needed

Idea: Divide address spaces and physical memory into fixed-sized p ages

- Size: 2^n, Example: 4KB
- Physical page: page frame



Advantages Of Paging

Flexibility: Supporting the abstraction of address space effectively

- Don’t need assumption how heap and stack grow and are used.

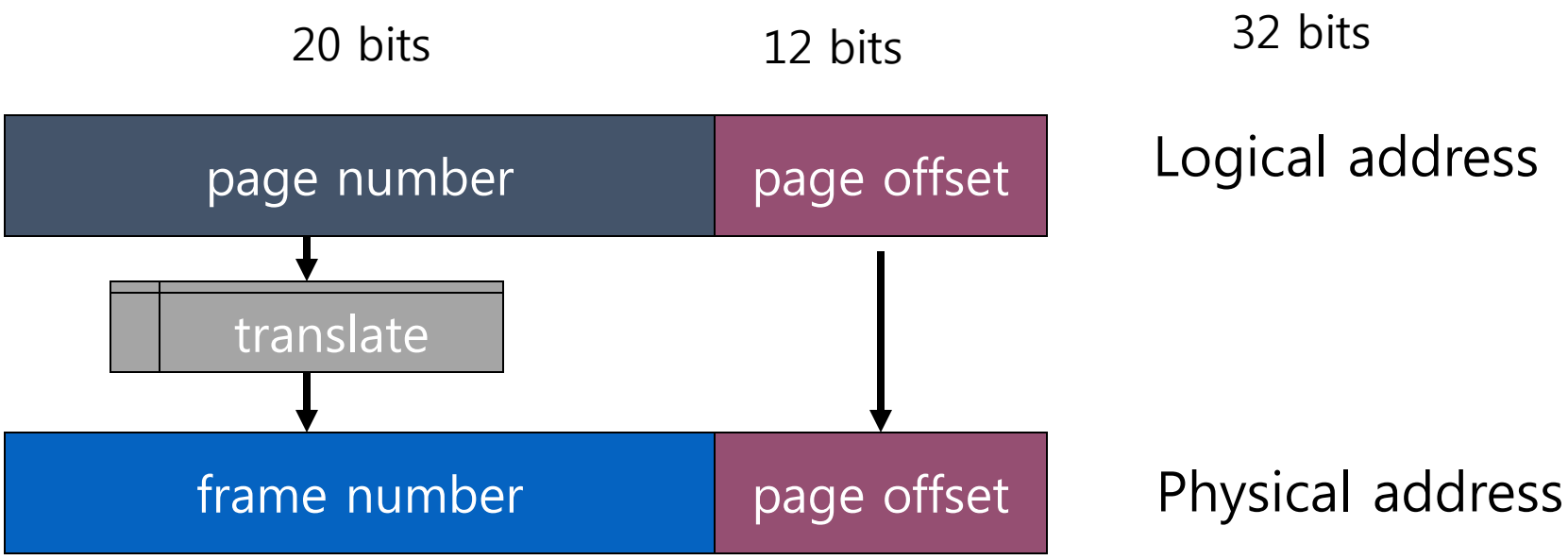
Simplicity: ease of free-space management

- The page in address space and the page frame are the same size.
- Easy to allocate and keep a free list

Translation of Page Addresses

How to translate logical address to physical address?

- High-order bits of address designate page number
- Low-order bits of address designate offset within page

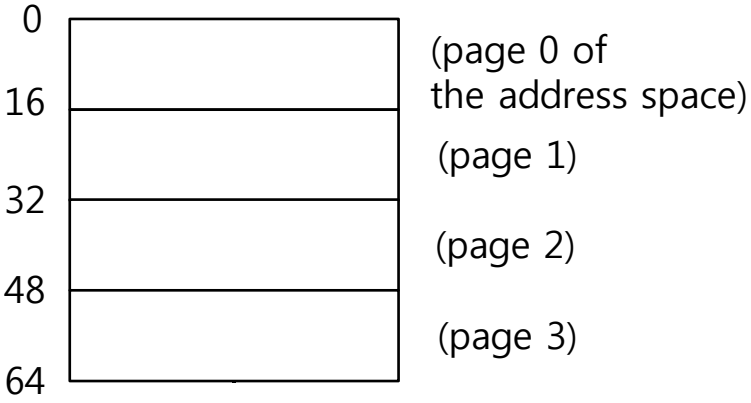


No addition needed; just append bits correctly…

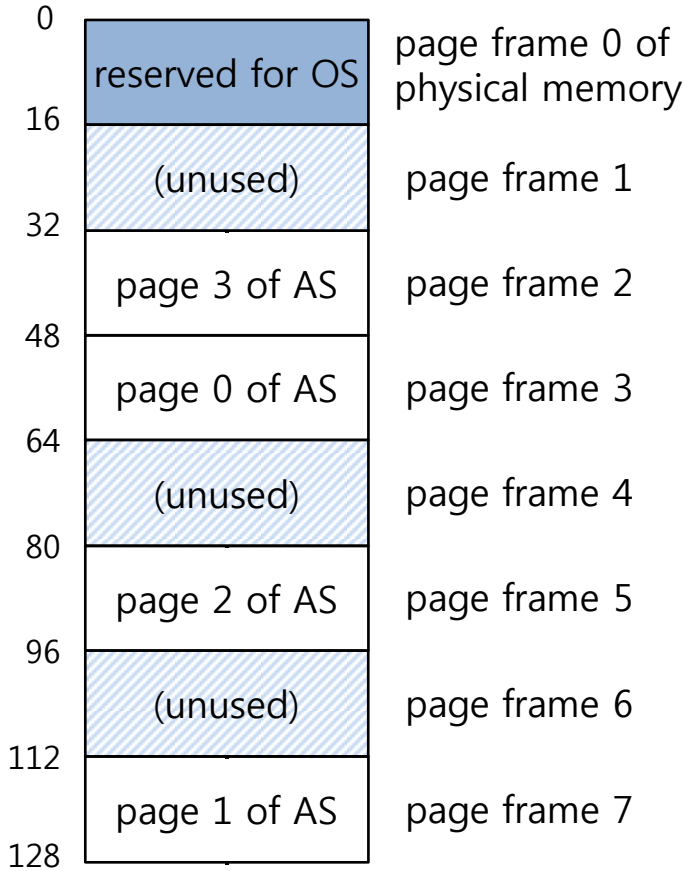
How does format of address space determine number of pages and size of pages?

Example: A Simple Paging

128-byte physical memory with 16 bytes page frames
64-byte address space with 16 bytes pages



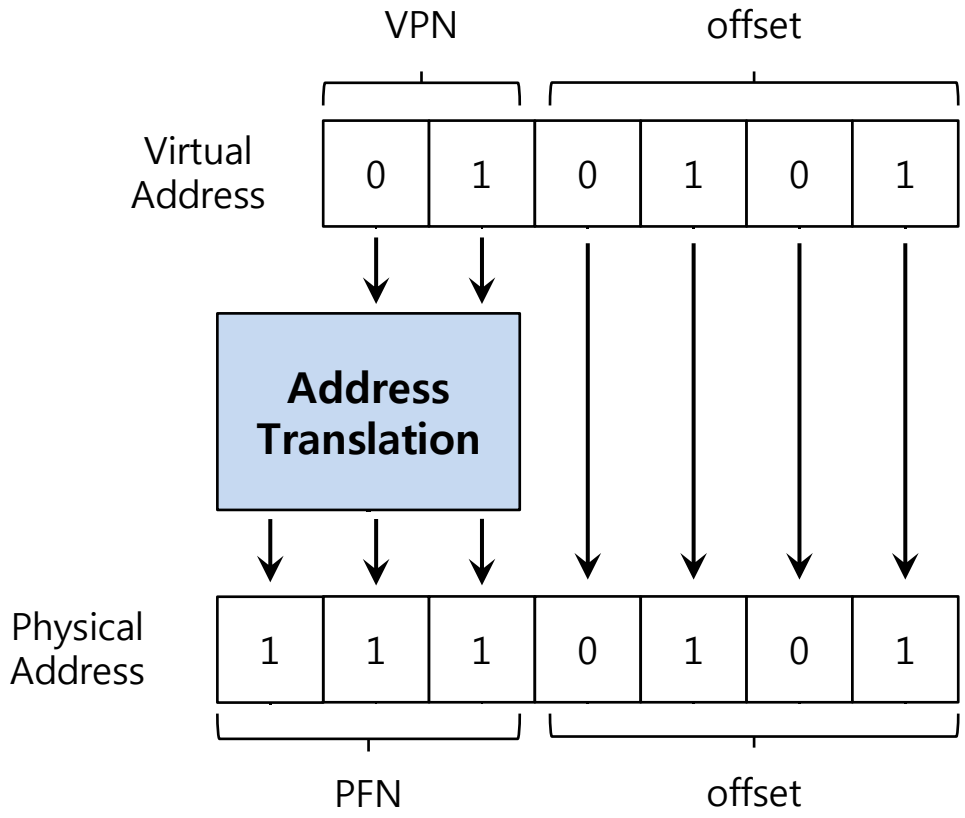
A Simple 64-byte Address Space



64-Byte Address Space Placed In Physical Memory

Example: Address Translation

The virtual address 21 in 64-byte address space



Quiz: Address Format

Given known page size, how many bits are needed in address to specify offset in page?

Page Size	Low Bits (offset)
16 bytes	4
1 KB	10
1 MB	20
512 bytes	9
4 KB	12

Quiz: Address Format

Given number of bits in virtual address and bits for offset, how many bits for virtual page number?

Page Size	Low Bits (offset)	Virt Addr Bits	High Bits (vpn)
16 bytes	4	10	6
1 KB	10	20	10
1 MB	20	32	12
512 bytes	9	16	5 7
4 KB	12	32	20

Correct?

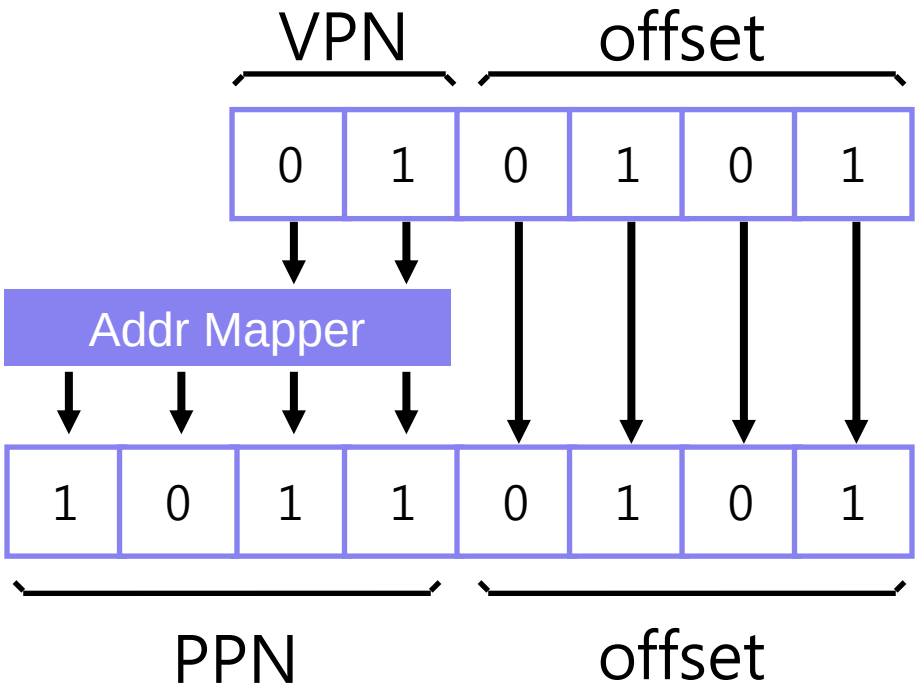
Quiz: Address Format

Given number of bits for vpn,
how many virtual pages can there be in an address space?

Page Size	Low Bits (offset)	Virt Addr Bits	High Bits (vpn)	Virt Pages
16 bytes	4	10	6	64
1 KB	10	20	10	1 K
1 MB	20	32	12	4 K
512 bytes	9	16	5	32
4 KB	12	32	20	1 MB

Virtual => Physical Page Mapping

Number of bits in virtual address format does not need to equal number of bits in physical address format



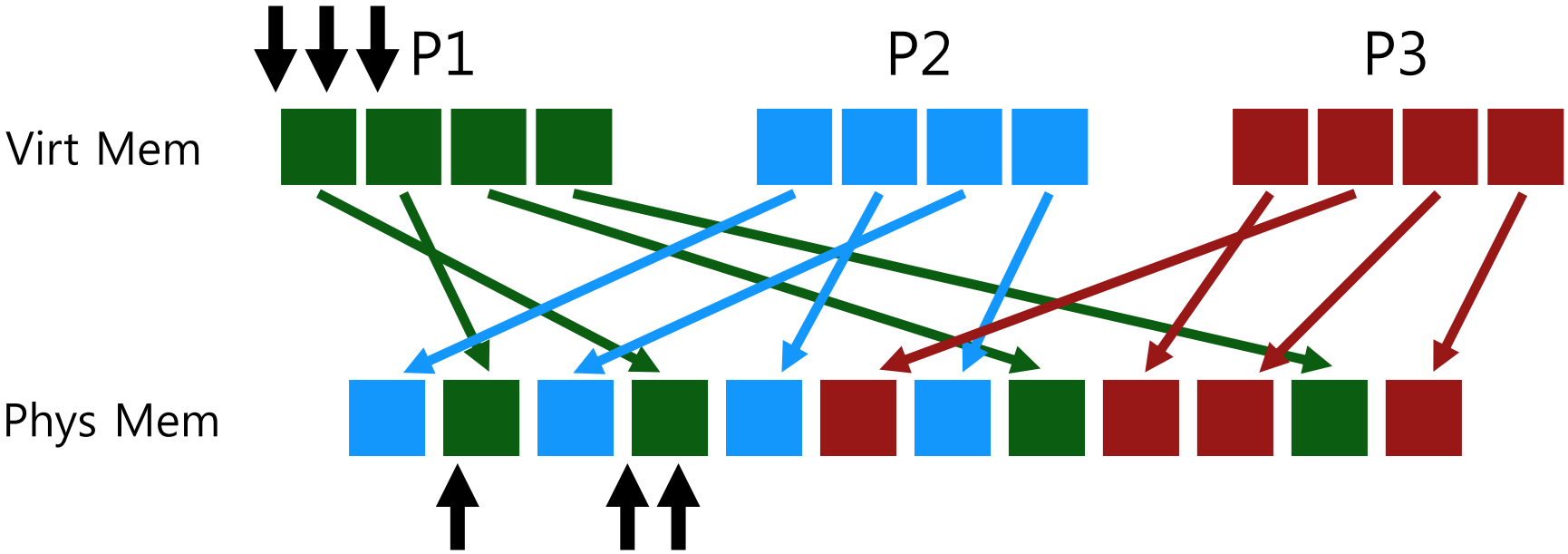
How should OS translate VPN to PPN?

For segmentation, OS used a formula (e.g., phys addr = virt_offset + base_reg)

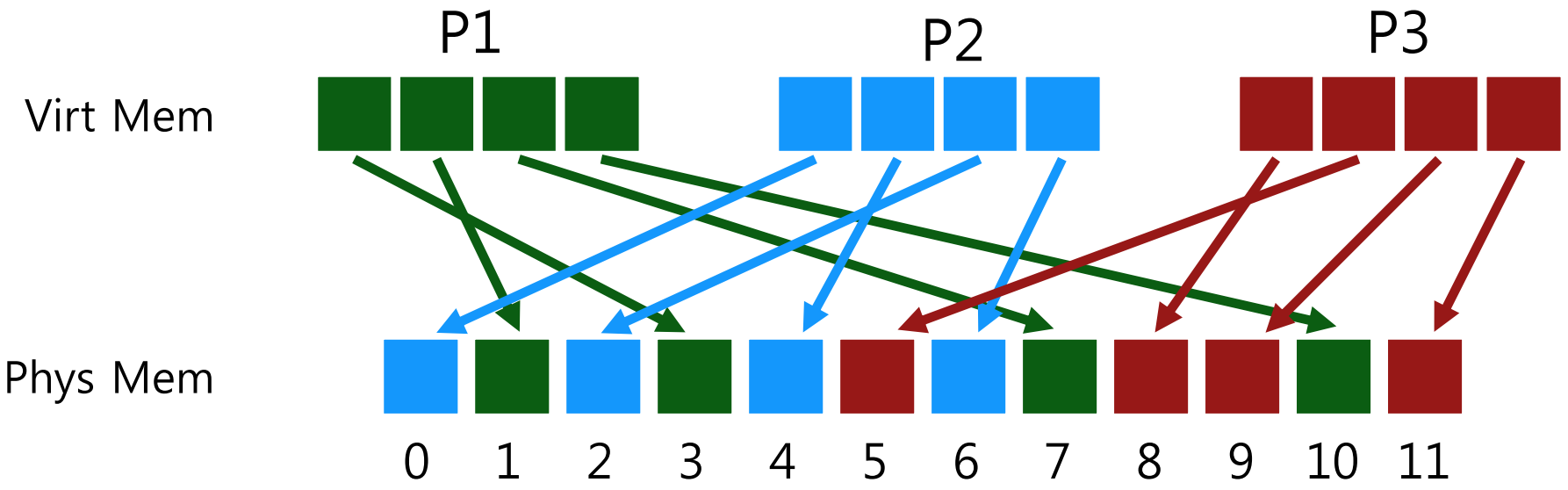
For paging, OS needs more general mapping mechanism

What data structure is good? Big array: pagetable

The Mapping



Quiz: Fill in Page Table



Page Tables:

P1	P2	P3
3	0	8
1	4	5
7	2	9
10	6	11

Where Are Pagetables Stored?

How big is a typical page table?
- assume **32-bit** address space
- assume 4 KB pages
- assume 4 byte entries

Final answer: 2 ^ (32 - log(4KB)) * 4 = **4 MB**

- Page table size = Num entries * size of each entry
- Num entries = num virtual pages = 2^(bits for vpn)
- Bits for vpn = 32– number of bits for page offset
  = 32 – lg(4KB) = 32 – 12 = 20
- Num entries = 2^20 = 1 MB
- Page table size = Num entries * 4 bytes = 4 MB

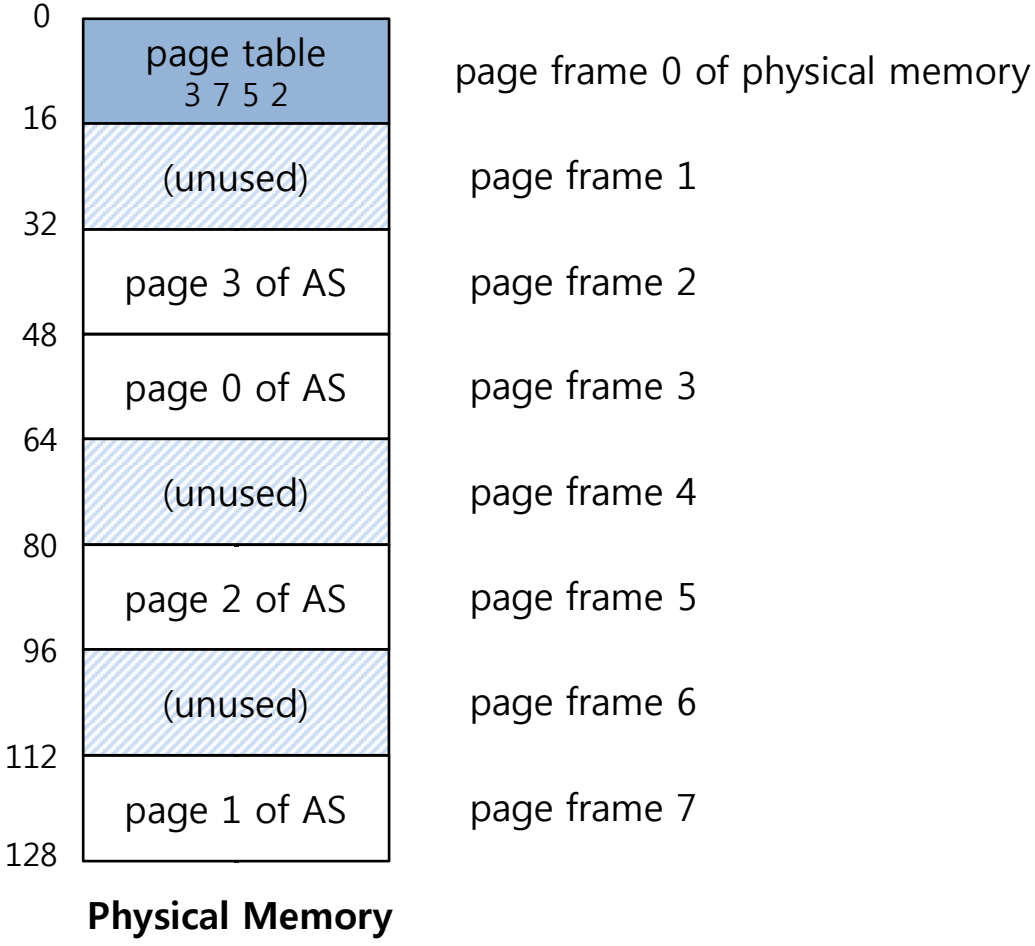
Implication: Store each page table in memory

- Hardware finds page table base with register (e.g., CR3 on x86)

What happens on a context-switch?

- Change contents of page table base register to newly scheduled process
- Save old page table base register in PCB of descheduled process

Example: Page Table in Kernel Physical Memory



What Is In The Page Table?

The page table is just a **data structure** that is used to map the virtual address to physical address.

- Simplest form: a linear page table, an array

The OS **indexes** the array by VPN, and looks up the page-table entry.

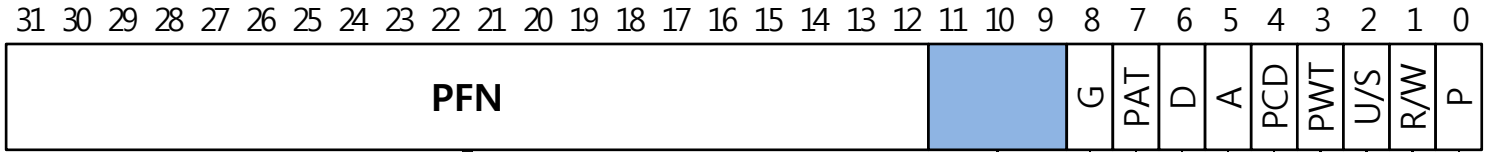
Common Flags Of Page Table Entry

- **Valid Bit**: Indicating whether the particular translation is valid.
- **Protection Bit**: Indicating whether the page could be read from, written to, or executed from
- **Present Bit**: Indicating whether this page is in physical memory or on disk(swapped out)
- **Dirty Bit**: Indicating whether the page has been modified since it was brought into memory
- **Reference Bit(Accessed Bit):** Indicating that a page has been accessed

Pagetable entries are just bits stored in memory

- Agreement between hw and OS about interpretation

Example: x86 Page Table Entry



An x86 Page Table Entry(PTE)

- P: present
- R/W: read/write bit
- U/S: supervisor
- A: accessed bit
- D: dirty bit
- PFN: the page frame number

Paging: Too Slow

- To find a location of the desired PTE, the **starting location** of the page table is **needed**.

- For every memory reference, paging requires the OS to perform one **extra memory reference**.

Accessing Memory With Paging

```
1         // Extract the VPN from the virtual address
2         VPN = (VirtualAddress & VPN_MASK) >> SHIFT
3
4         // Form the address of the page-table entry (PTE)
5         PTEAddr = PTBR + (VPN * sizeof(PTE))
6
7         // Fetch the PTE
8         PTE = AccessMemory(PTEAddr)
9
10        // Check if process can access the page
11        if (PTE.Valid == False)
12                  RaiseException(SEGMENTATION_FAULT)
13        else if (CanAccess(PTE.ProtectBits) == False)
14                  RaiseException(PROTECTION_FAULT)
15        else
16                  // Access is OK: form physical address and fetch it
17                  offset = VirtualAddress & OFFSET_MASK
18                  PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
19                  Register = AccessMemory(PhysAddr)
```

A Memory Trace

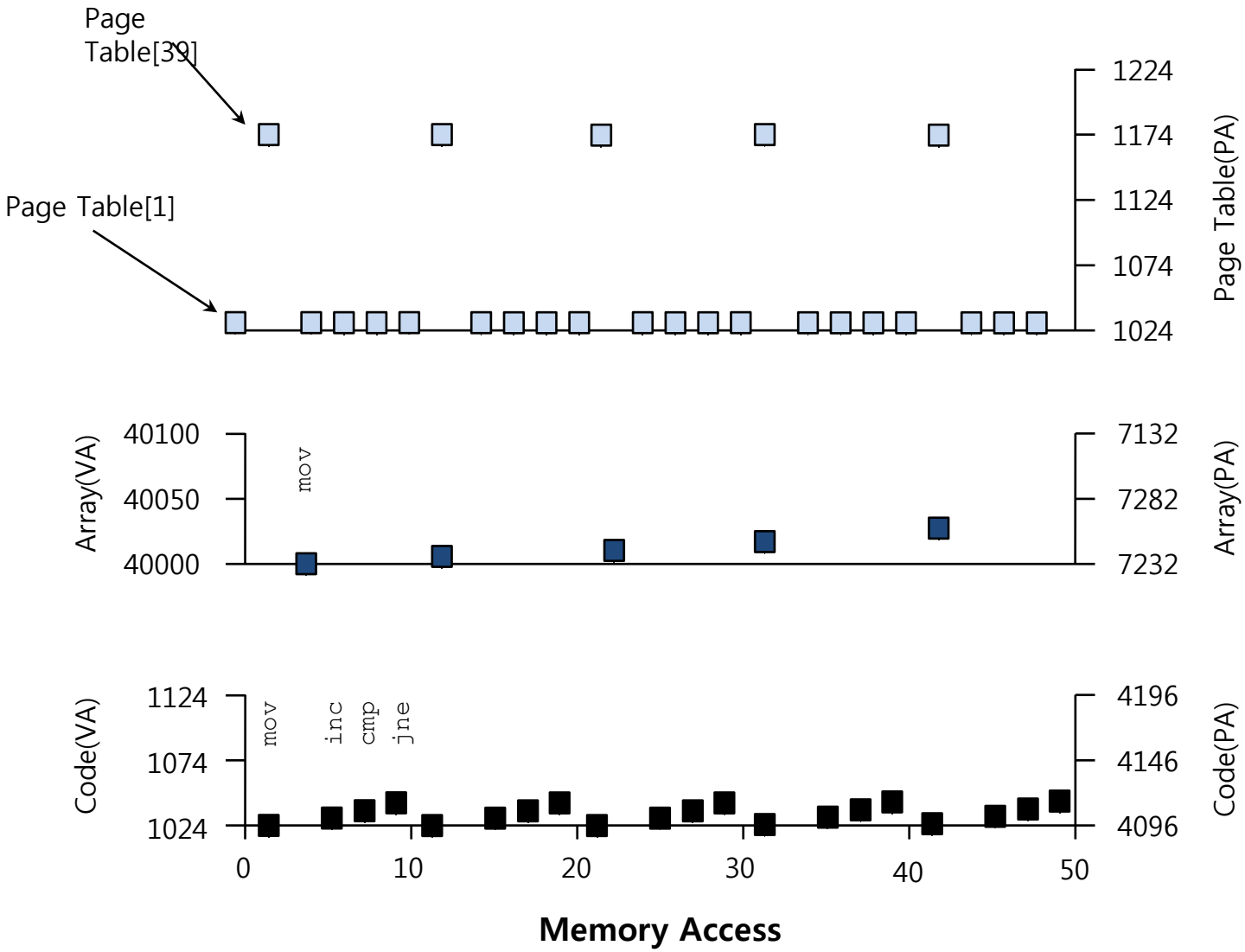
- Example: A Simple Memory Access

```
int array[1000];
...
for (i = 0; i < 1000; i++)
        array[i] = 0;
```

```
prompt> gcc –o array array.c –Wall –o
prompt>./array
```

```
0x1024 movl $0x0,(%edi,%eax,4)
0x1028 incl %eax
0x102c cmpl $0x03e8,%eax
0x1030 jne 0x1024
```

A Virtual(And Physical) Memory Trace

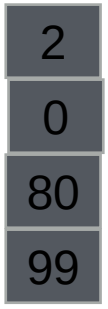


Memory Accesses with Pages

```
0x0010: movl  0x1100, %edi
0x0013: addl  $0x3, %edi
0x0019: movl  %edi, 0x1100
```

Assume PT is at phys addr 0x5000
Assume PTE's are 4 bytes
Assume 4KB pages
How many bits for offset? 12

Simplified view of page table



Old: How many mem refs with segmentation?

5 (3 instrs, 2 movl)

Physical Memory Accesses with Paging?

1) Fetch instruction at logical addr 0x0010; vpn?
   - Access page table to get ppn for vpn 0
   - Mem ref 1: 0x5000
   - Learn vpn 0 is at ppn 2
   - Fetch instruction at 0x2010 (Mem ref 2)

Exec, load from logical addr 0x1100; vpn?
   - Access page table to get ppn for vpn 1
   - Mem ref 3: 0x5004
   - Learn vpn 1 is at ppn 0
   - Movl from 0x0100 into reg (Mem ref 4)

Pagetable is slow!!! Doubles memory references

Advantages of Paging

No external fragmentation
- Any page can be placed in any frame in physical memory

Fast to allocate and free
- Alloc: No searching for suitable free space
- Free: Doesn't have to coallesce with adjacent free space
- Just use bitmap to show free/allocated page frames

Simple to swap-out portions of memory to disk (later lecture)
- Page size matches disk block size
- Can run process when some pages are on disk
- Add "present" bit to PTE

Disadvantages of Paging

Internal fragmentation: Page size may not match size needed by process

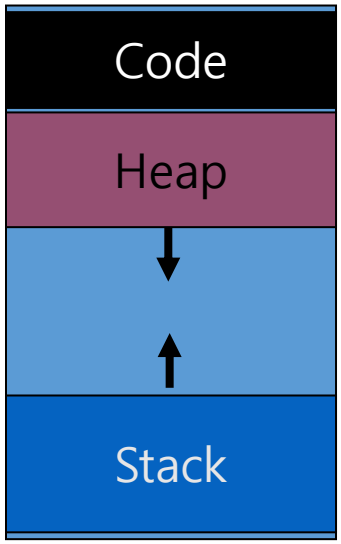
- Wasted memory grows with larger pages
- **Tension?**

Additional memory reference to page table --> Very inefficient

- Page table must be stored in memory
- MMU stores only base address of page table
- Solution: Add TLBs (future lecture)

Storage for page tables may be substantial

- Simple page table: Requires PTE for all pages in address space
  - Entry needed even if page not allocated
- Problematic with dynamic stack and heap within address space
- Page tables must be allocated contiguously in memory
- Solution: Combine paging and segmentation (future lecture)



Memory Allocators

Free-Space Management

Many systems need to manage/allocate space

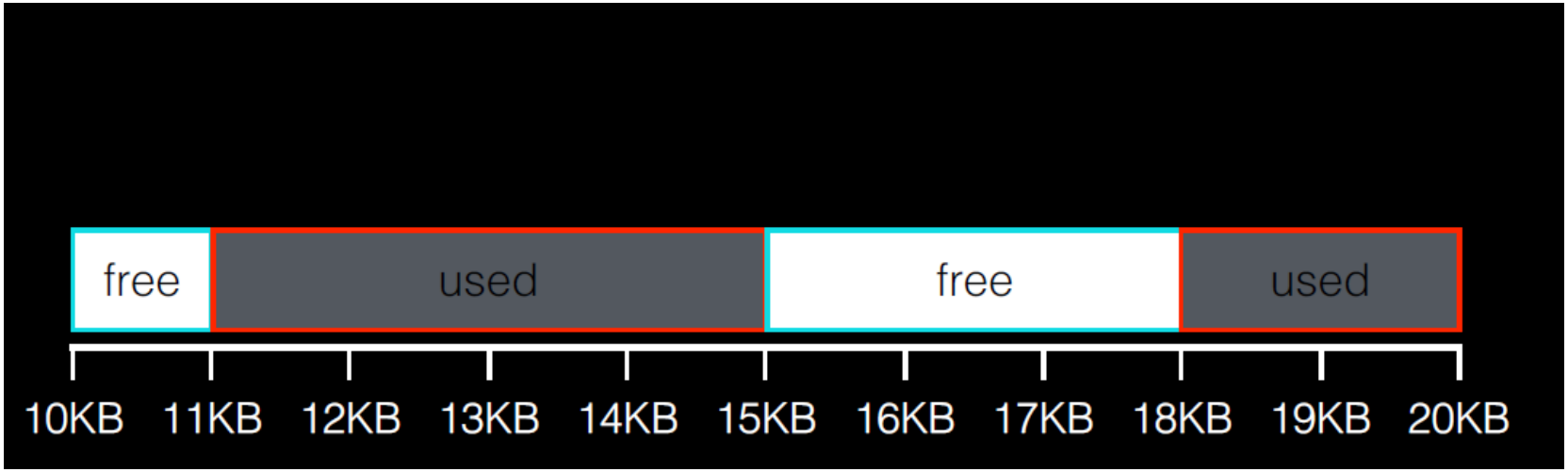
1. physical space for process segments
2. virtual space for malloc calls
3. disk blocks for files

Allocation API

void ***malloc**(size_t size);
void **free**(void *ptr);
void ***realloc**(void *ptr, size_t size);

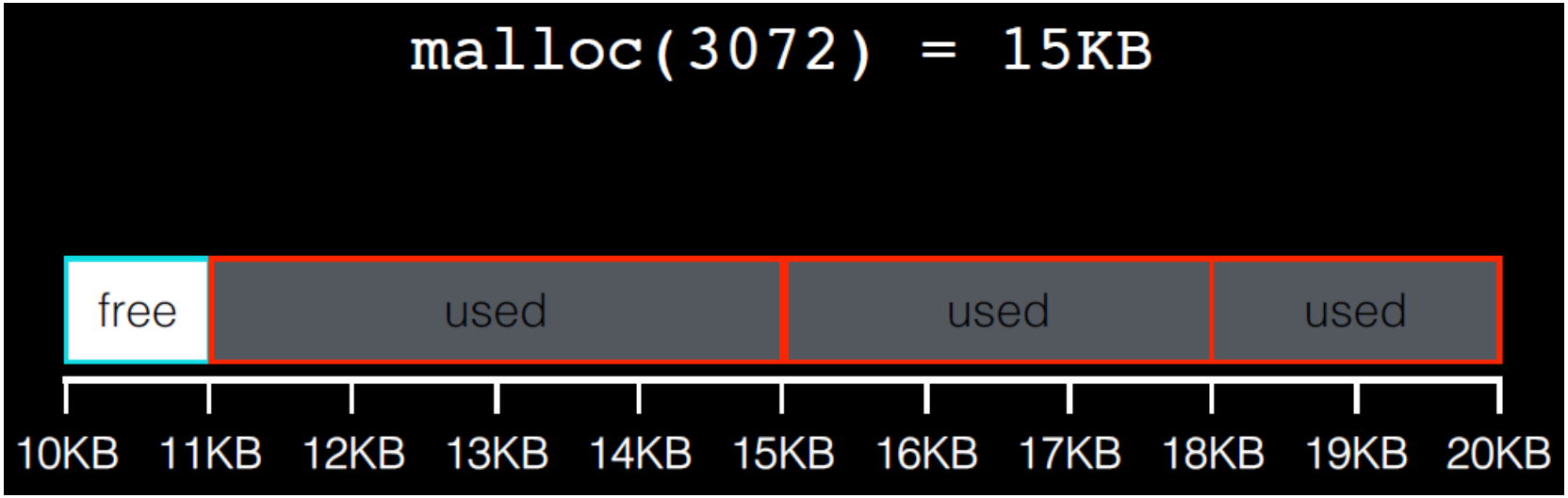
today

Malloc/Free Basics

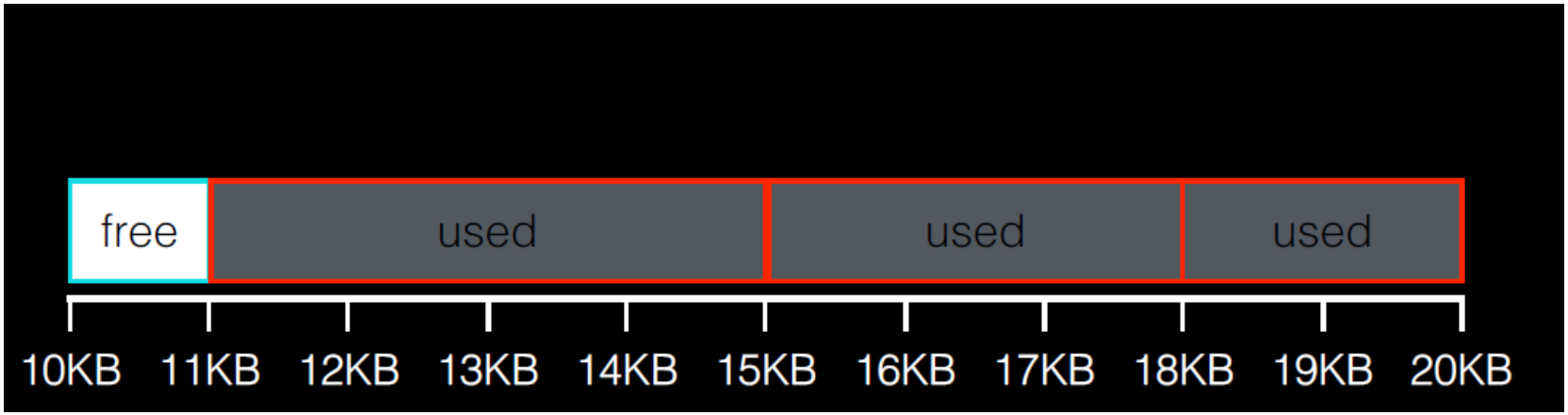


Malloc/Free Basics

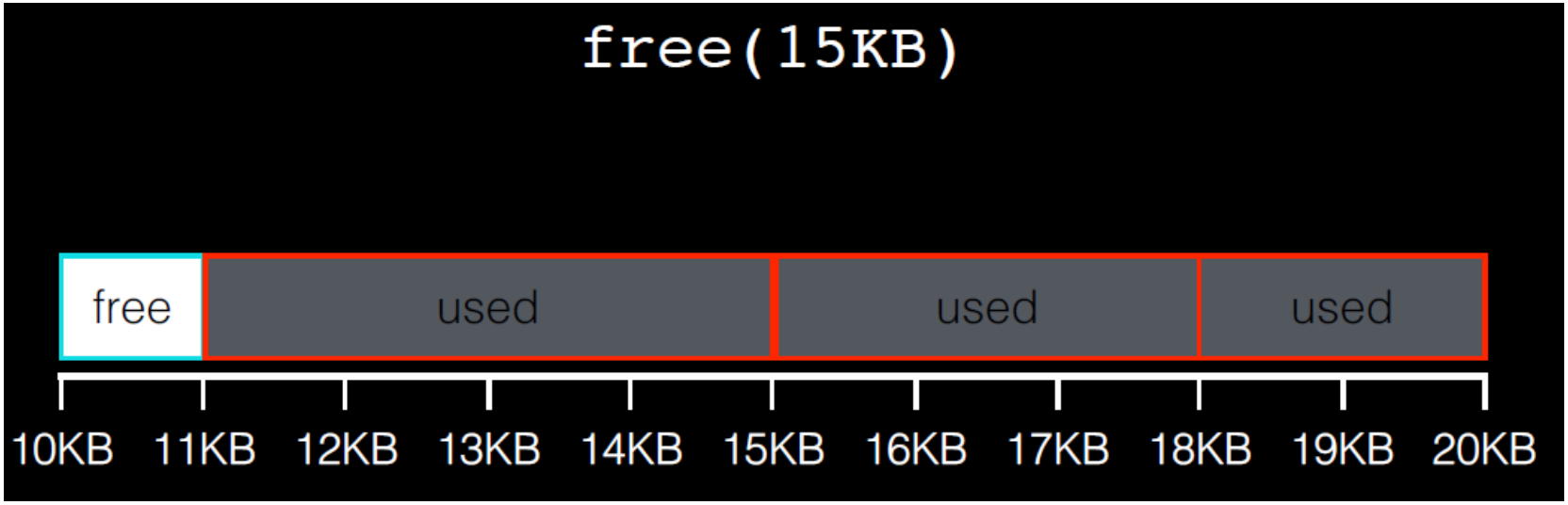
Malloc/Free Basics



Malloc/Free Basics

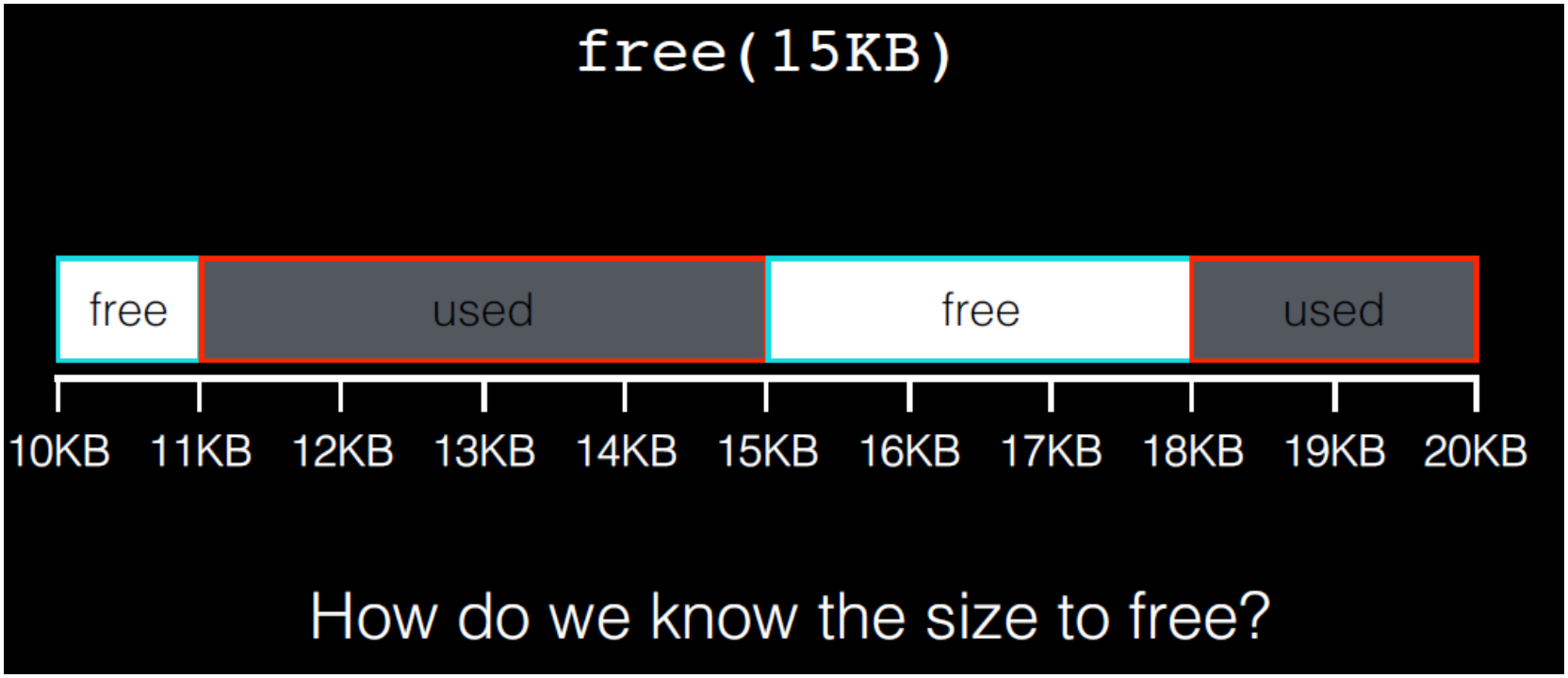


Malloc/Free Basics



Malloc/Free Basics

Malloc/Free Basics



Odd Object Sizes

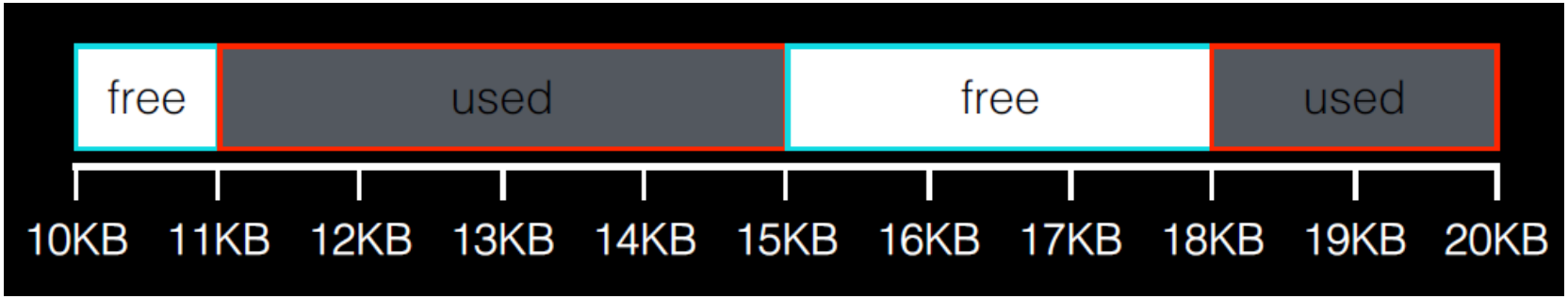
An object may not fit free space exactly:

split memory

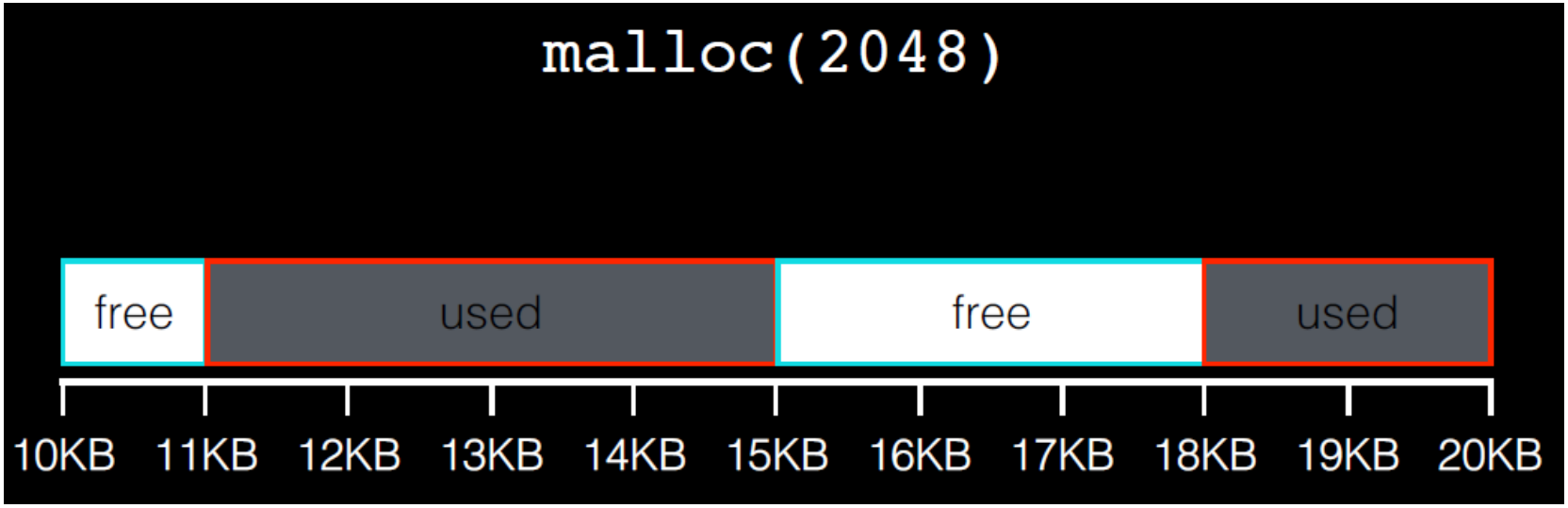
Free areas may be adjacent:

coalesce memory

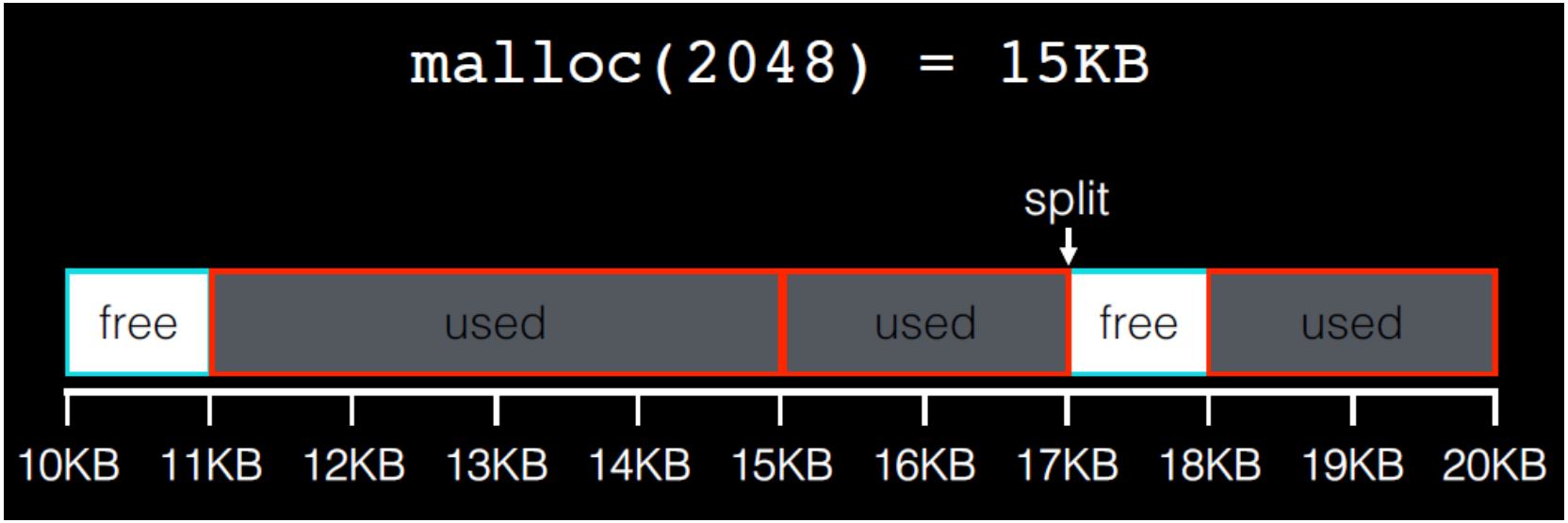
Splitting

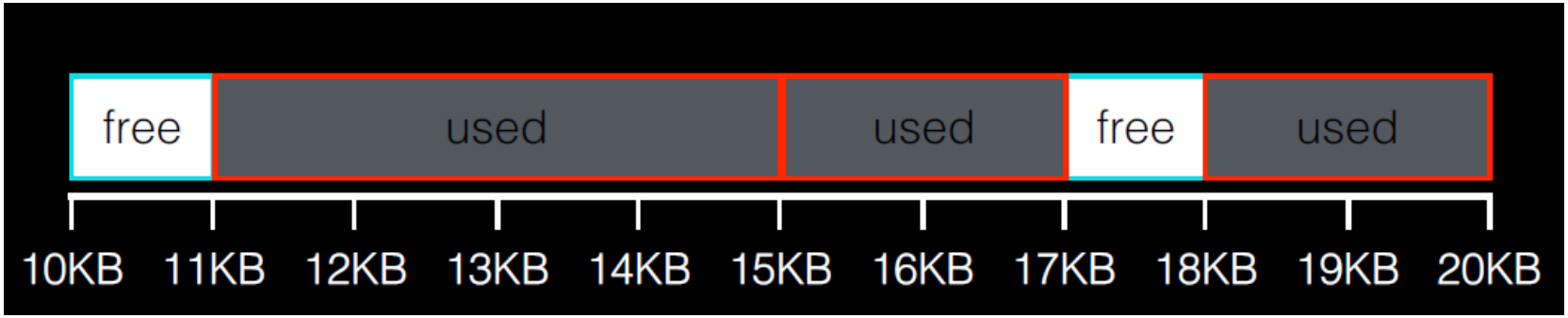

free
used
free
used
10KB
11KB
12KB
13KB
14KB
15KB
16KB
17KB
18KB
19KB
20KB

Splitting

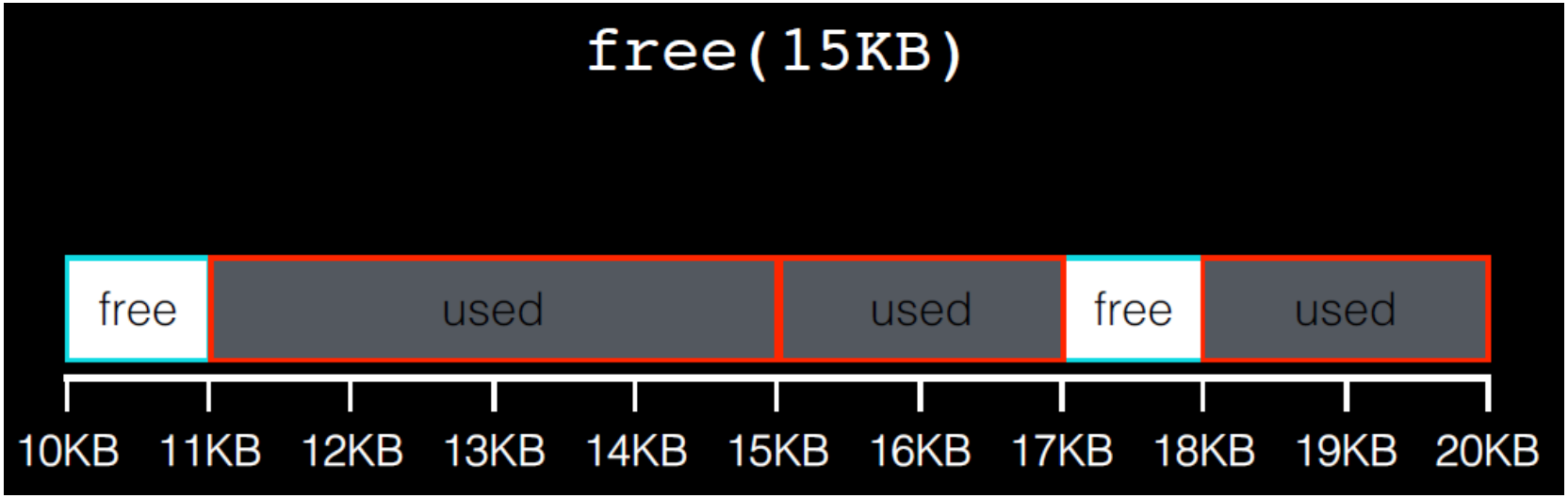

malloc(2048)
free
used
free
used
10KB
11KB
12KB
13KB
14KB
15KB
16KB
17KB
18KB
19KB
20KB

Splitting


malloc(2048) = 15KB
split
free
used
used
free
used
10KB
11KB
12KB
13KB
14KB
15KB
16KB
17KB
18KB
19KB
20KB


free
used
used
free
used
10KB
11KB
12KB
13KB
14KB
15KB
16KB
17KB
18KB
19KB
20KB

Coalescing


free(15KB)
free
used
used
free
used
10KB
11KB
12KB
13KB
14KB
15KB
16KB
17KB
18KB
19KB
20KB

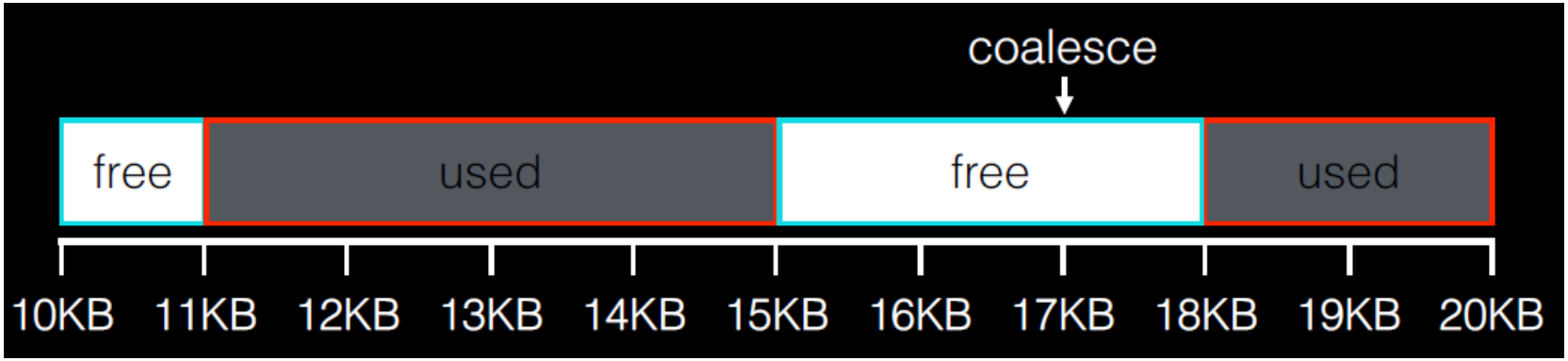
Coalescing


free(15KB)
free
used
free
free
used
10KB
11KB
12KB
13KB
14KB
15KB
16KB
17KB
18KB
19KB
20KB

Coalescing


free
used
free
free
used
10KB
11KB
12KB
13KB
14KB
15KB
16KB
17KB
18KB
19KB
20KB

Coalescing


coalesce
free
used
free
used
10KB
11KB
12KB
13KB
14KB
15KB
16KB
17KB
18KB
19KB
20KB

Bookkeeping

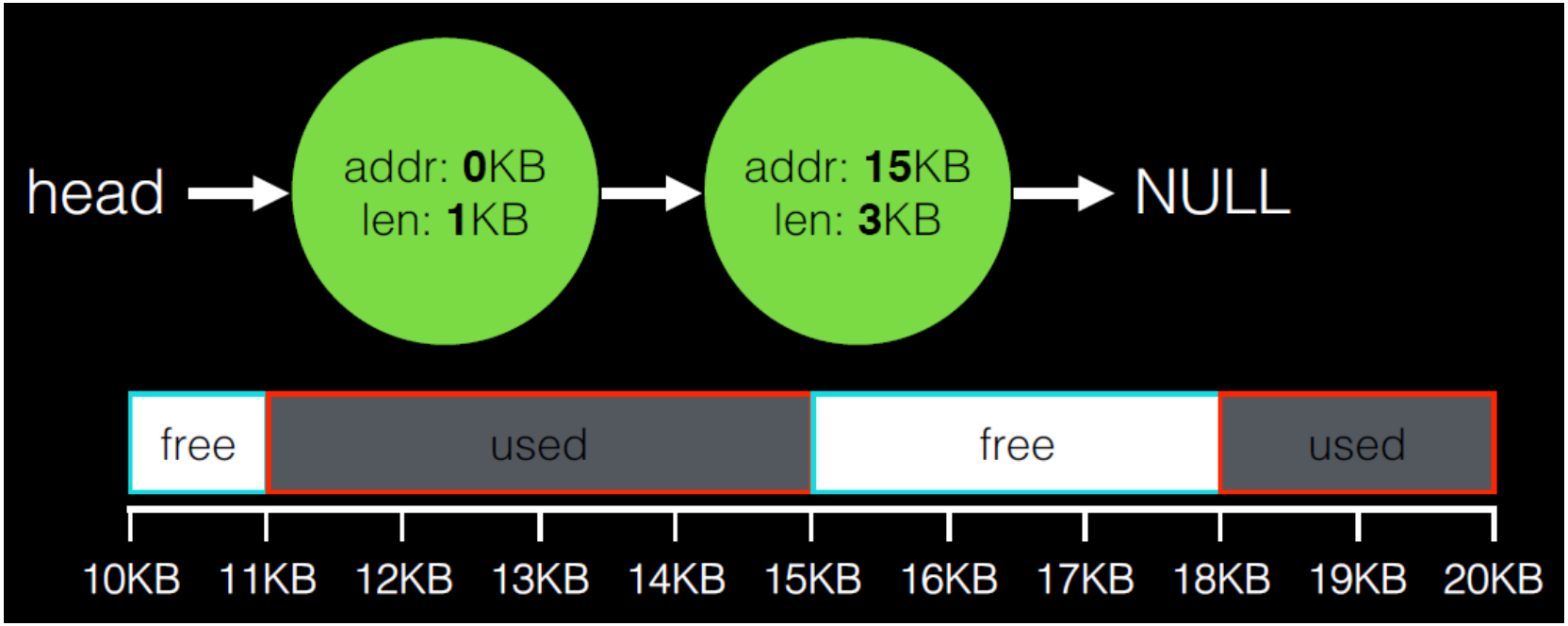
Need to know size+location of free spaces

- for malloc()

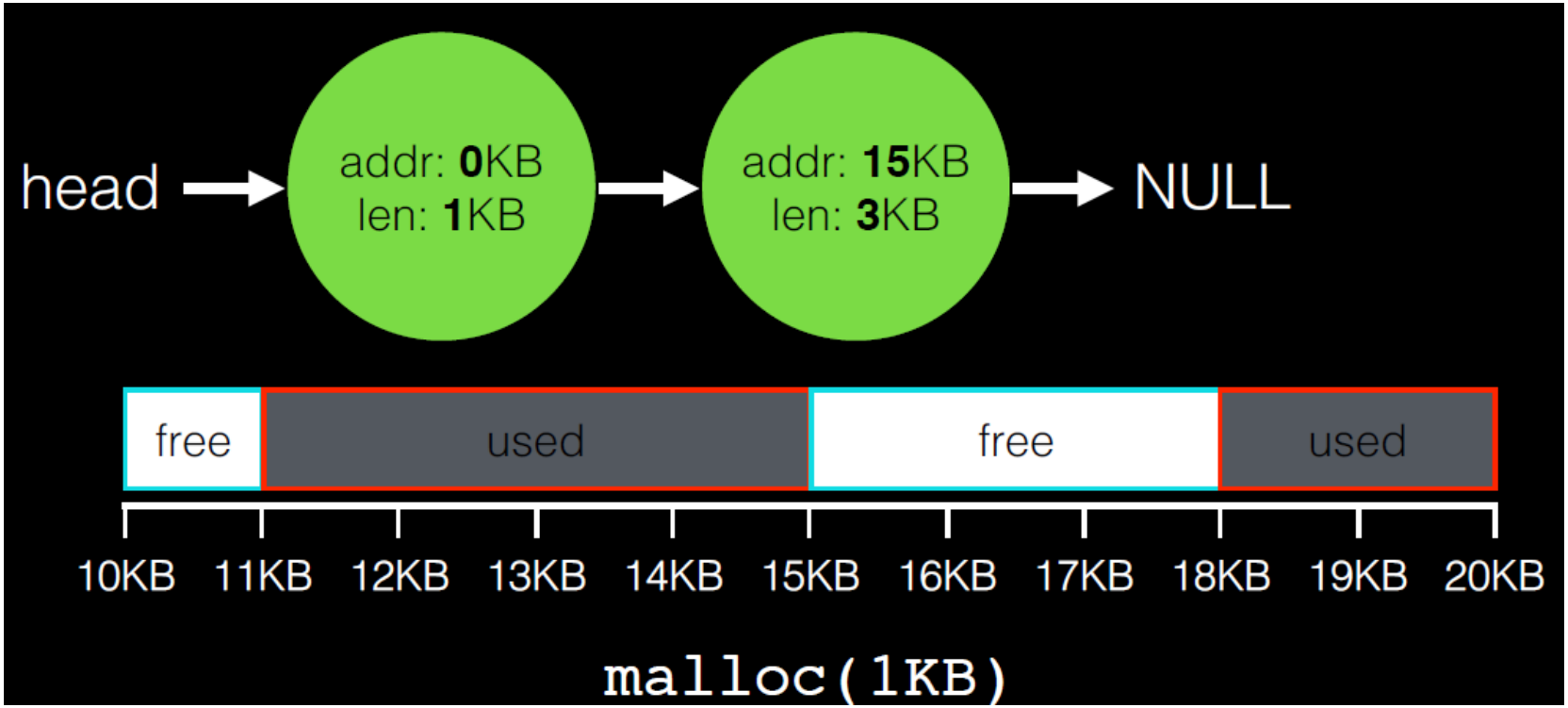
Need to know size of used spaces

- for free()

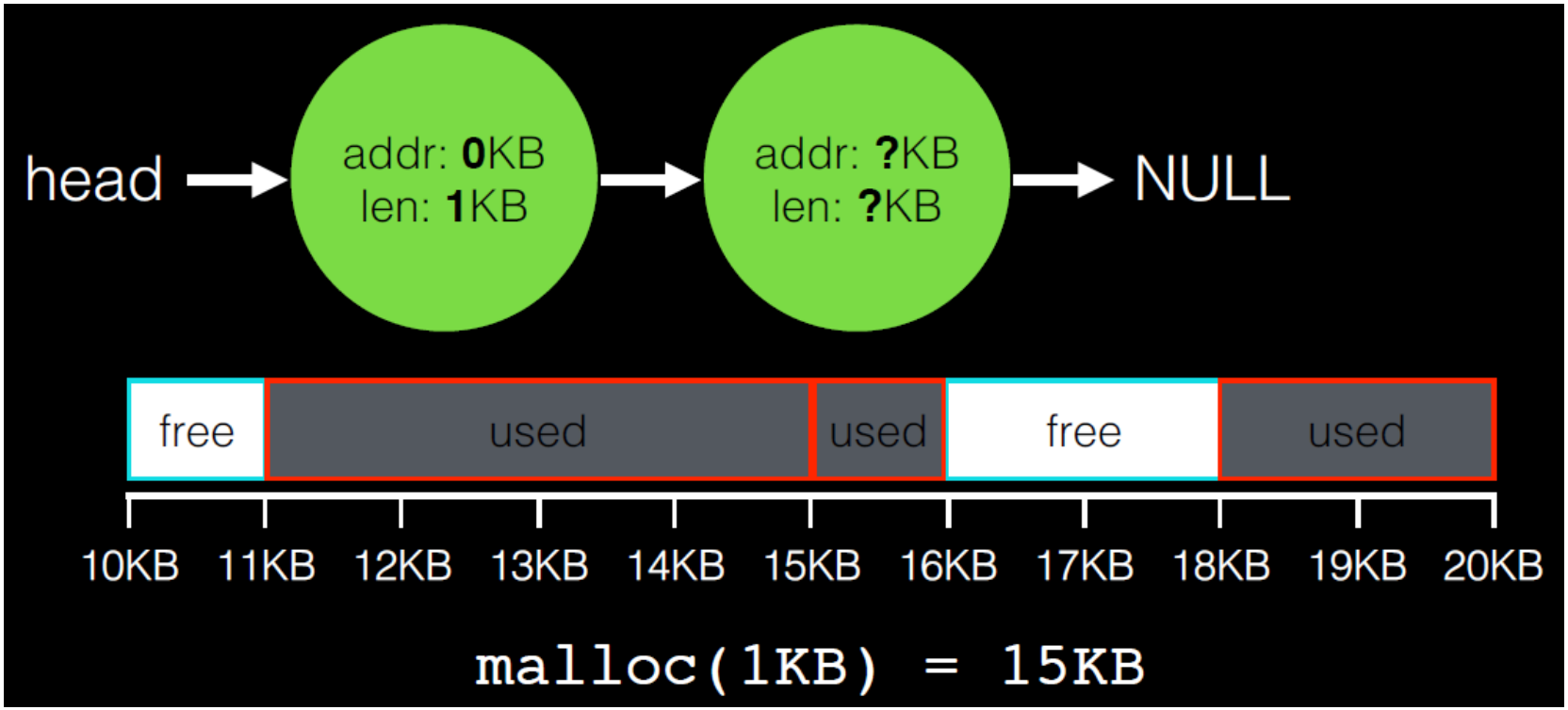
Free List: malloc



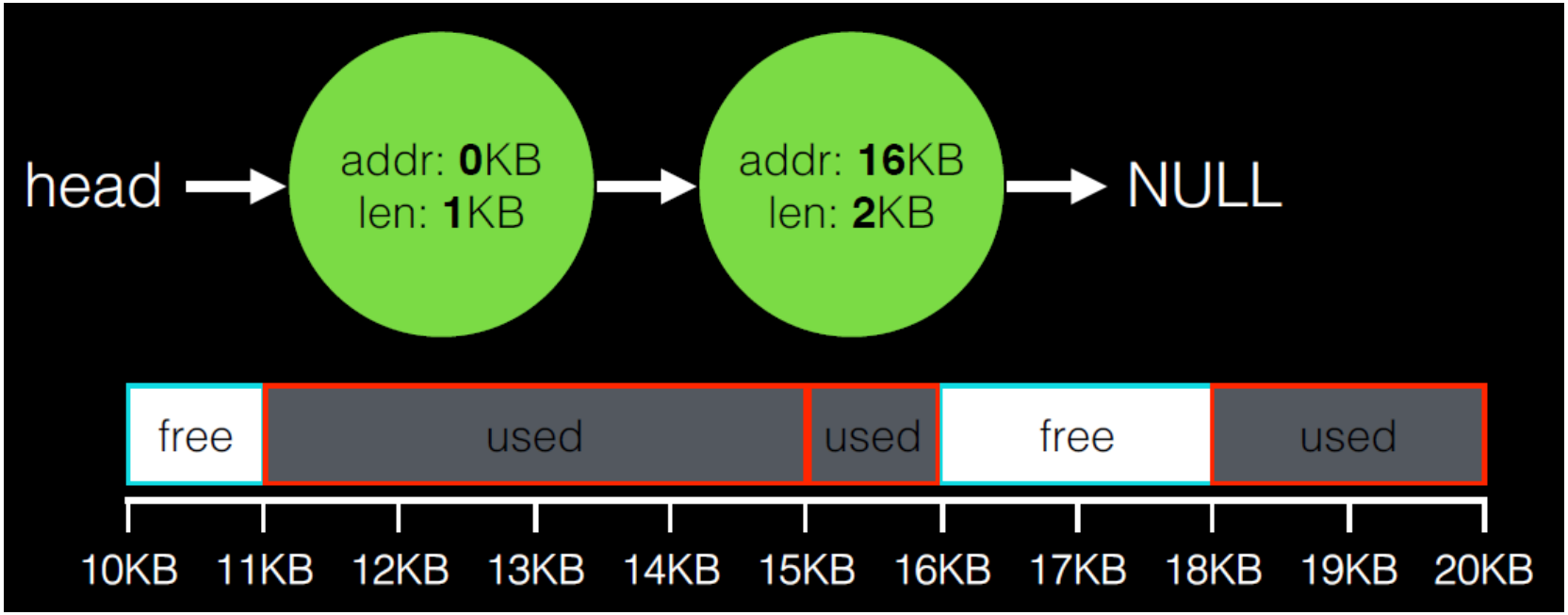
Free List: malloc



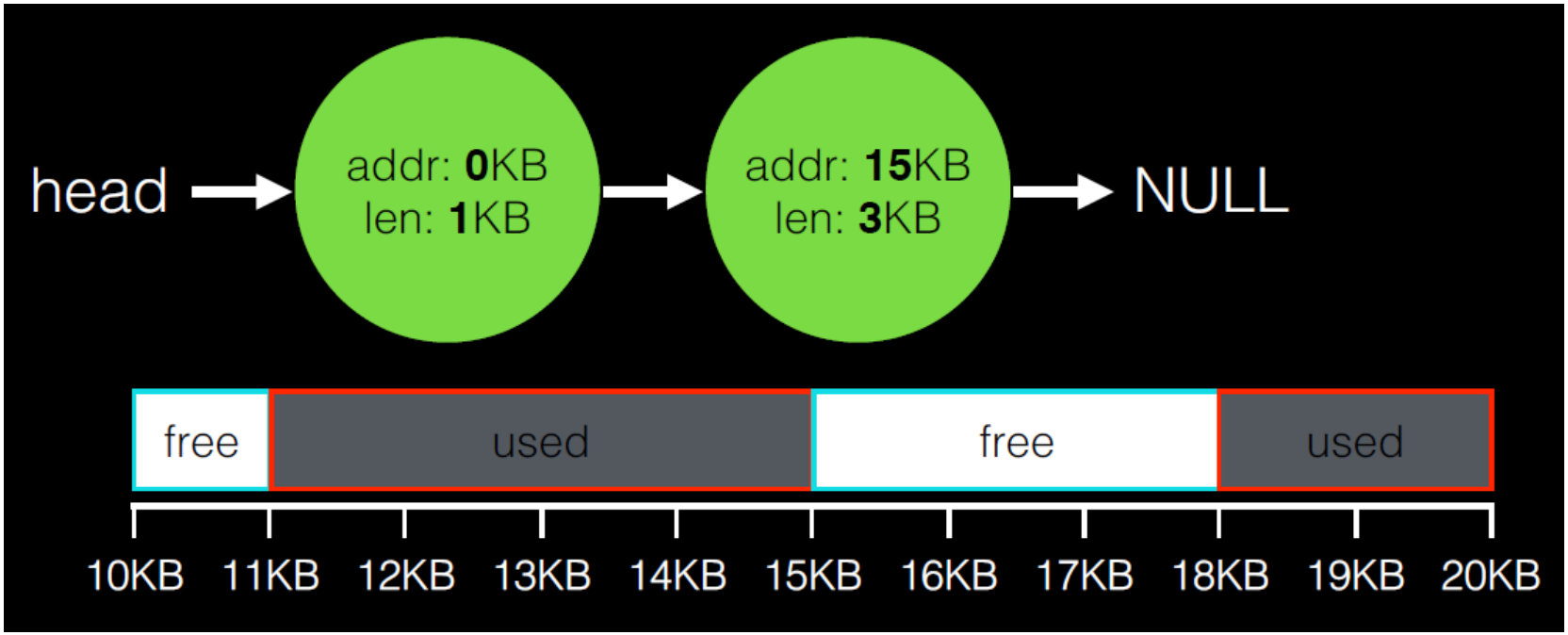
Free List: malloc



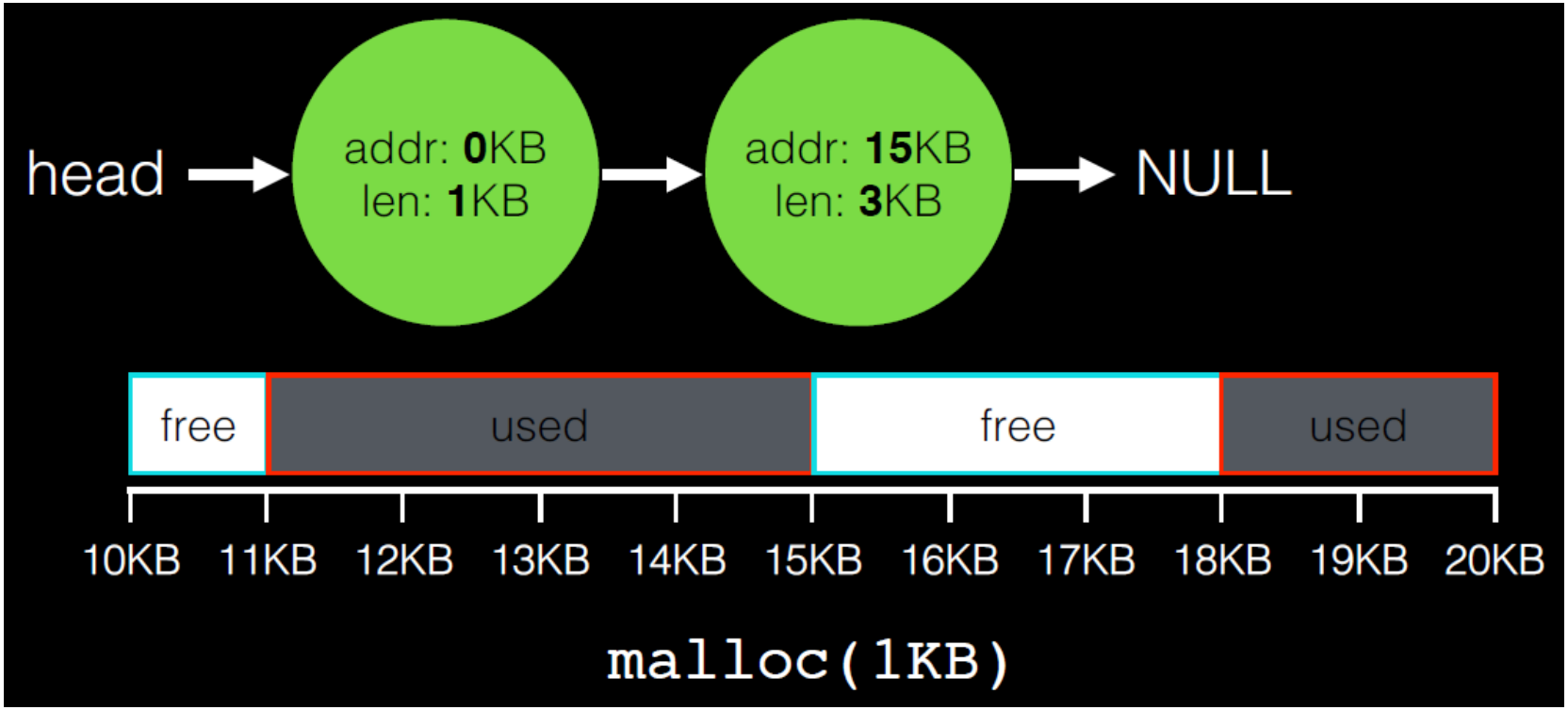
Free List: malloc



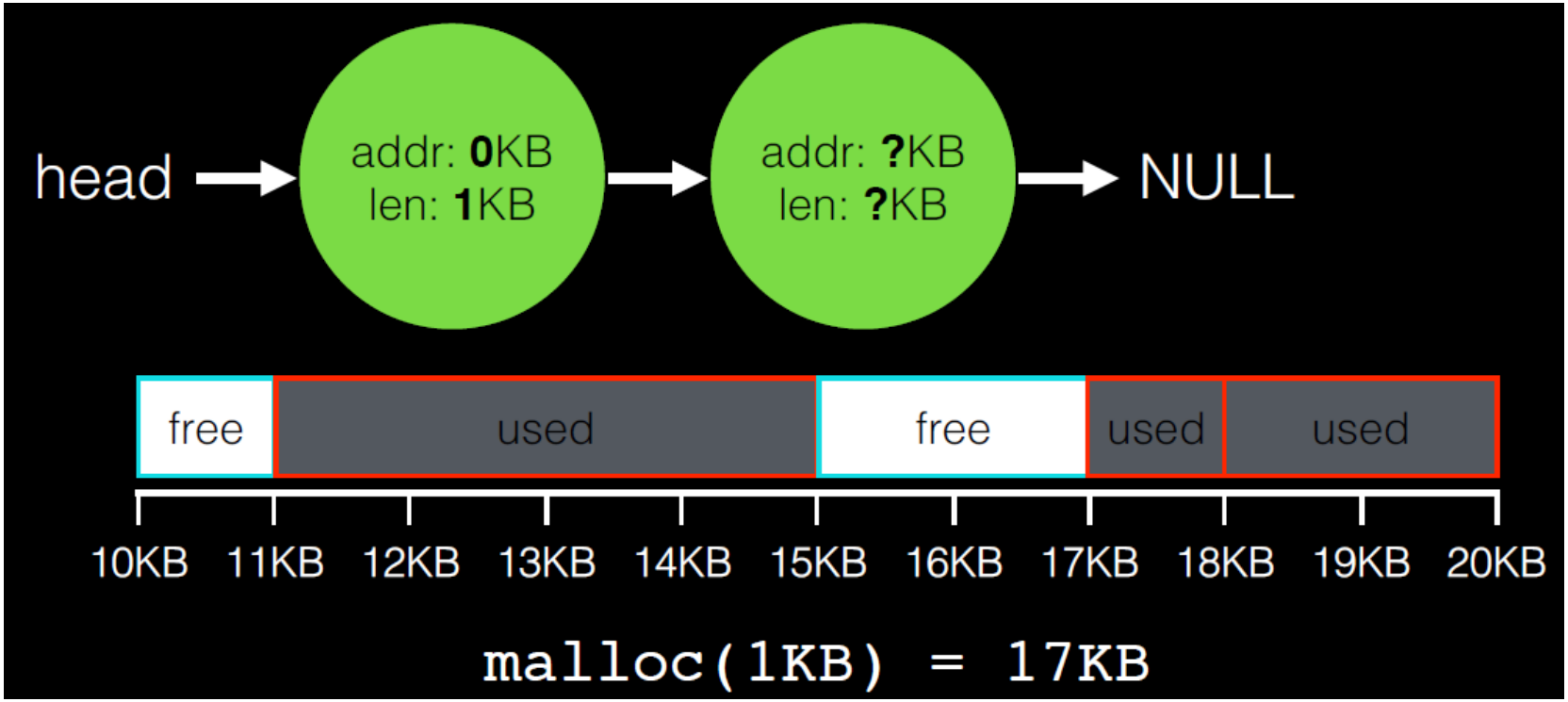
Start over


head
addr: 0KB
len: 1KB
addr: 15KB
len: 3KB
NULL
free
used
free
used
10KB
11KB
12KB
13KB
14KB
15KB
16KB
17KB
18KB
19KB
20KB

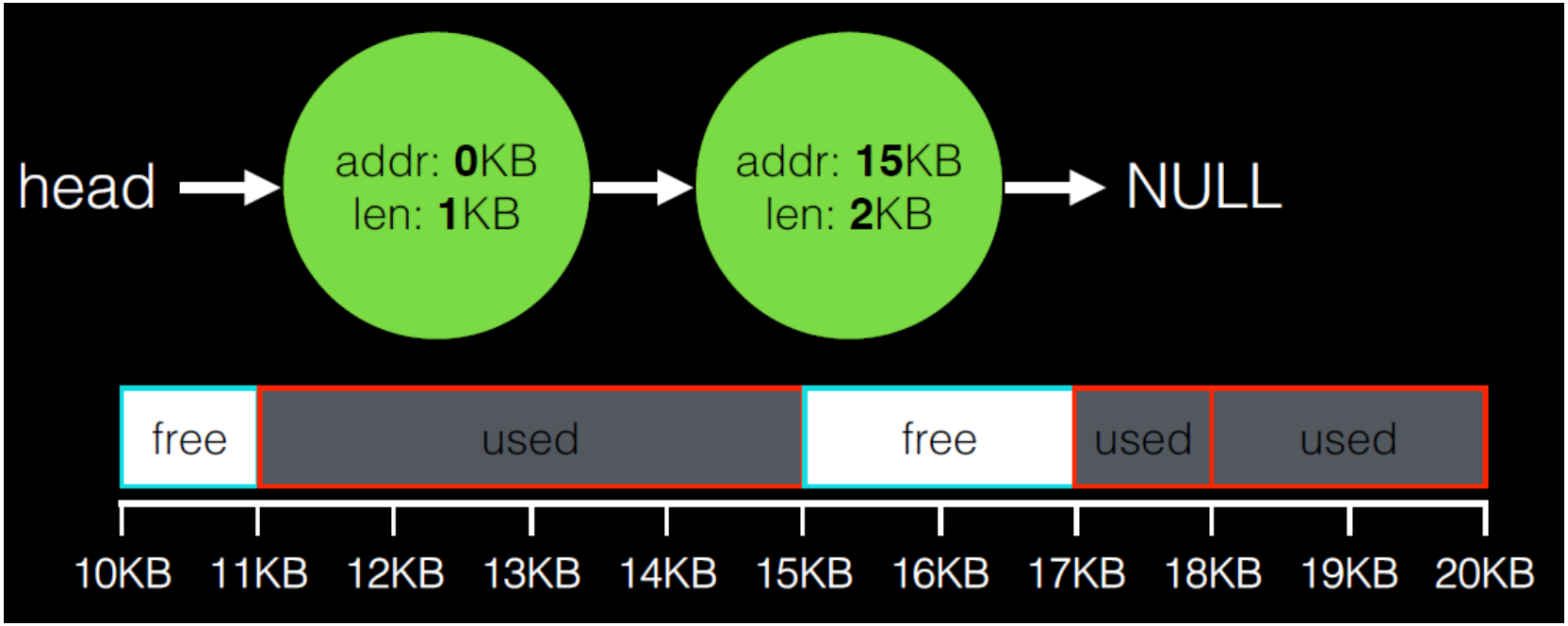
Start over


head
addr: 0KB
len: 1KB
addr: 15KB
len: 3KB
NULL
free
used
free
used
10KB
11KB
12KB
13KB
14KB
15KB
16KB
17KB
18KB
19KB
20KB
malloc(1KB)

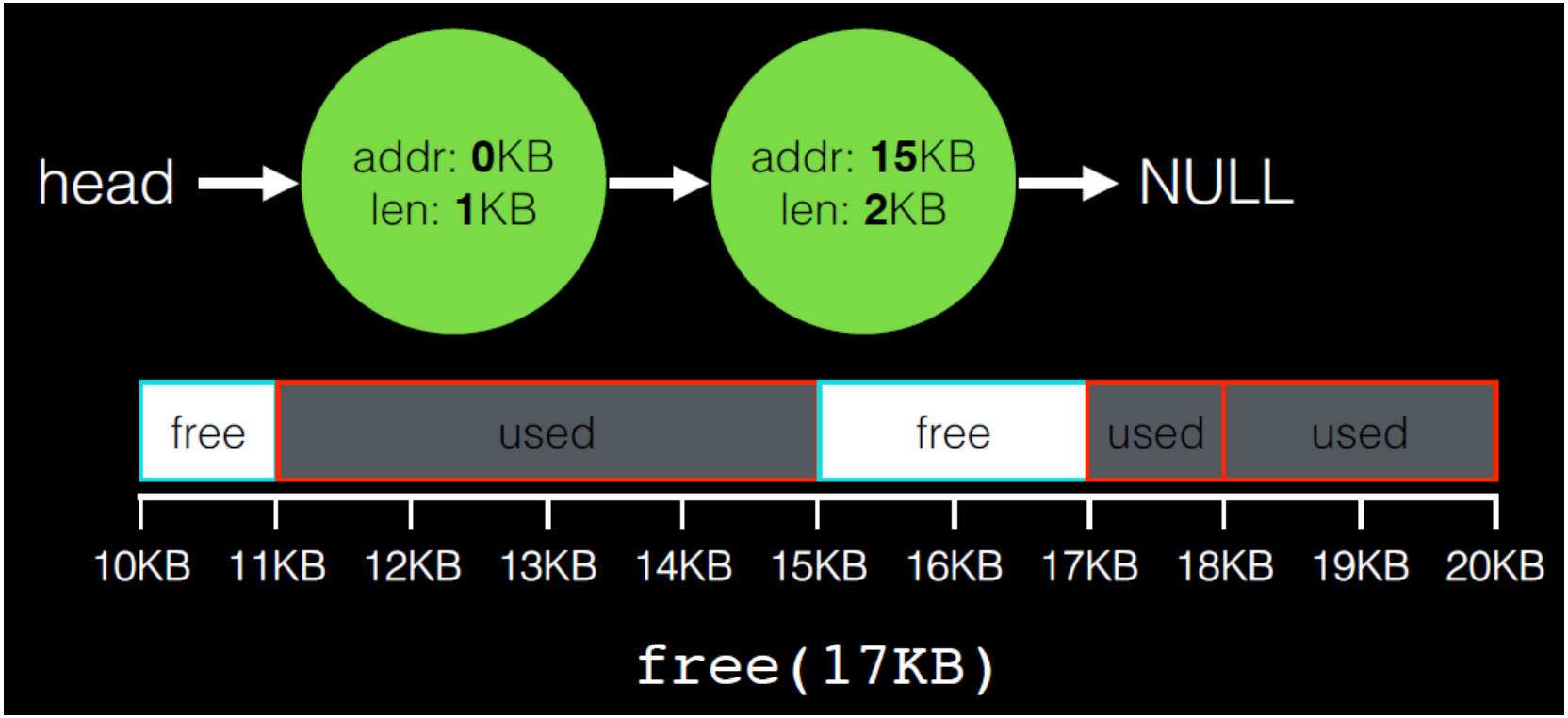
Free List: malloc



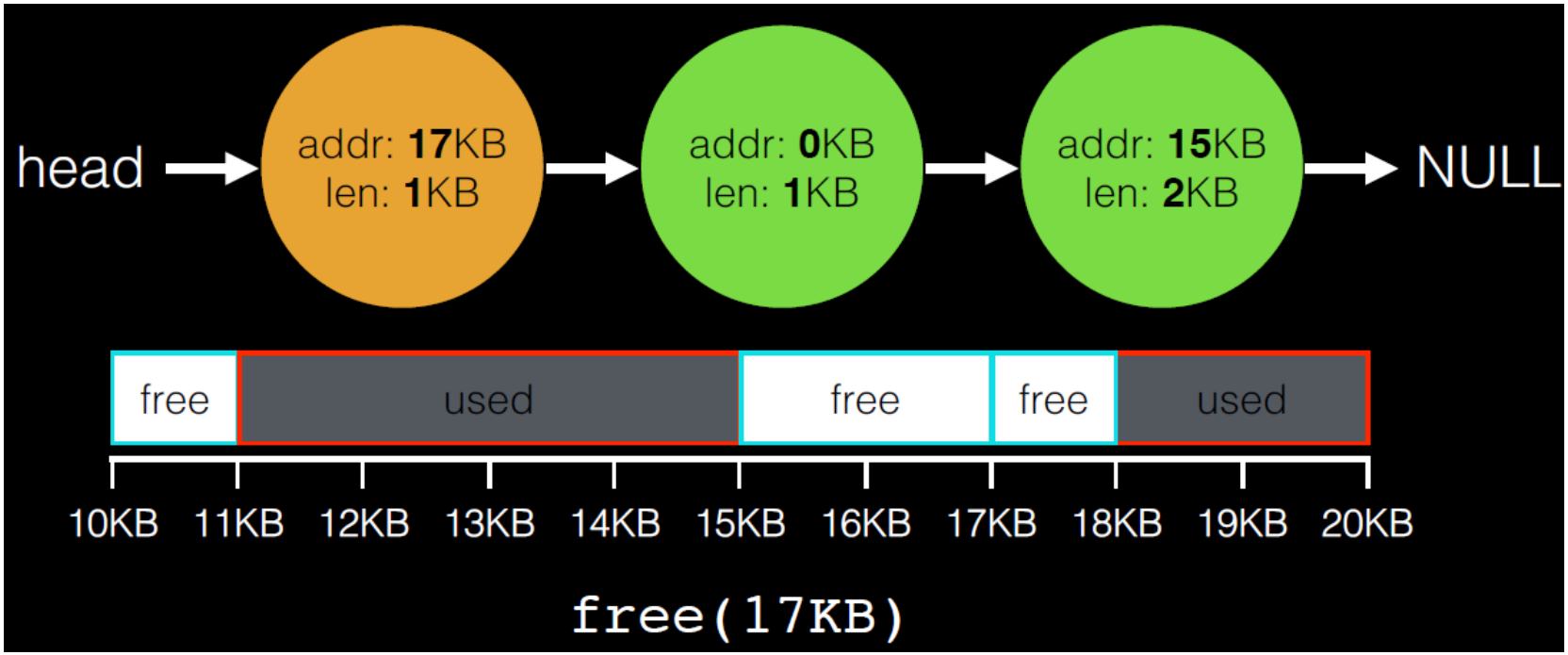
Free List: malloc



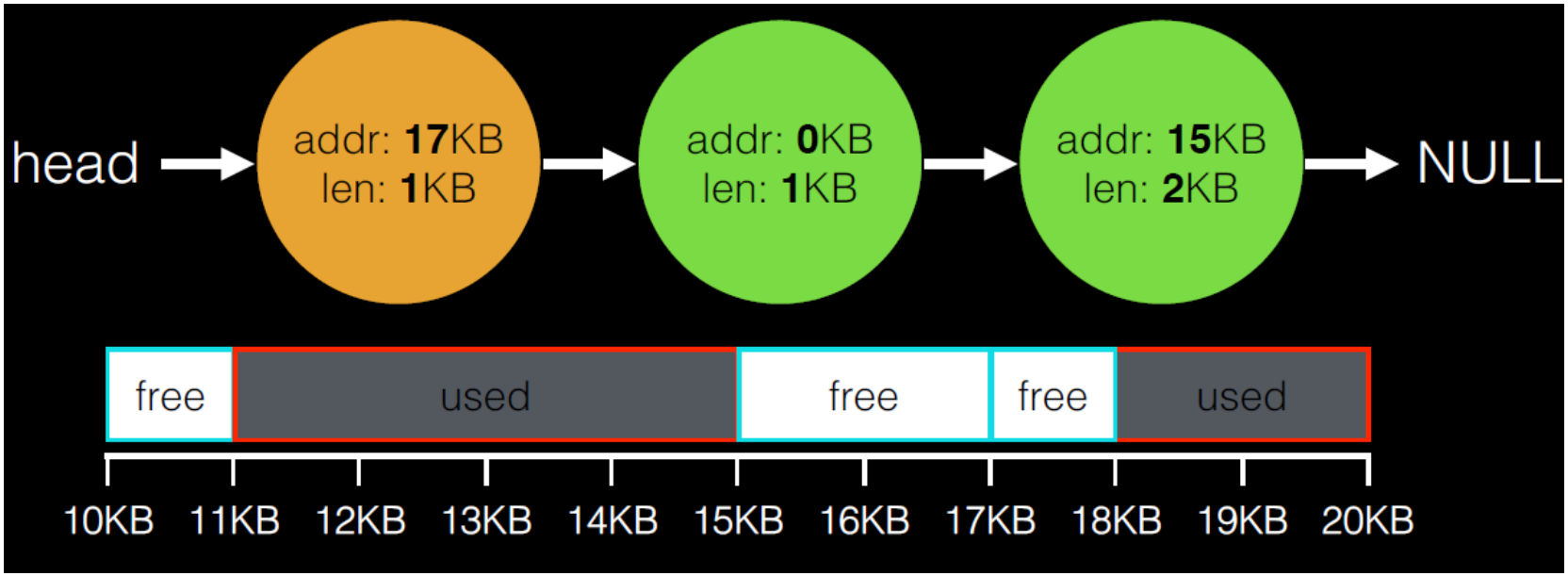
Free List: free



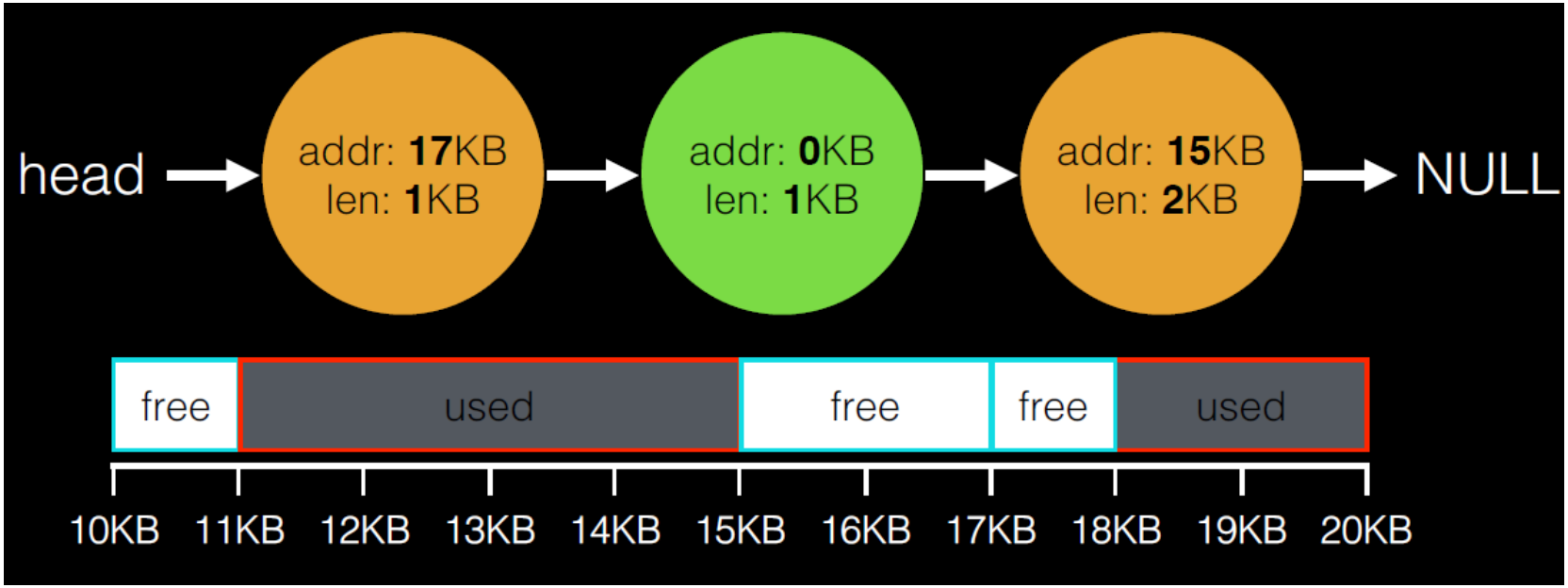
Free List: free



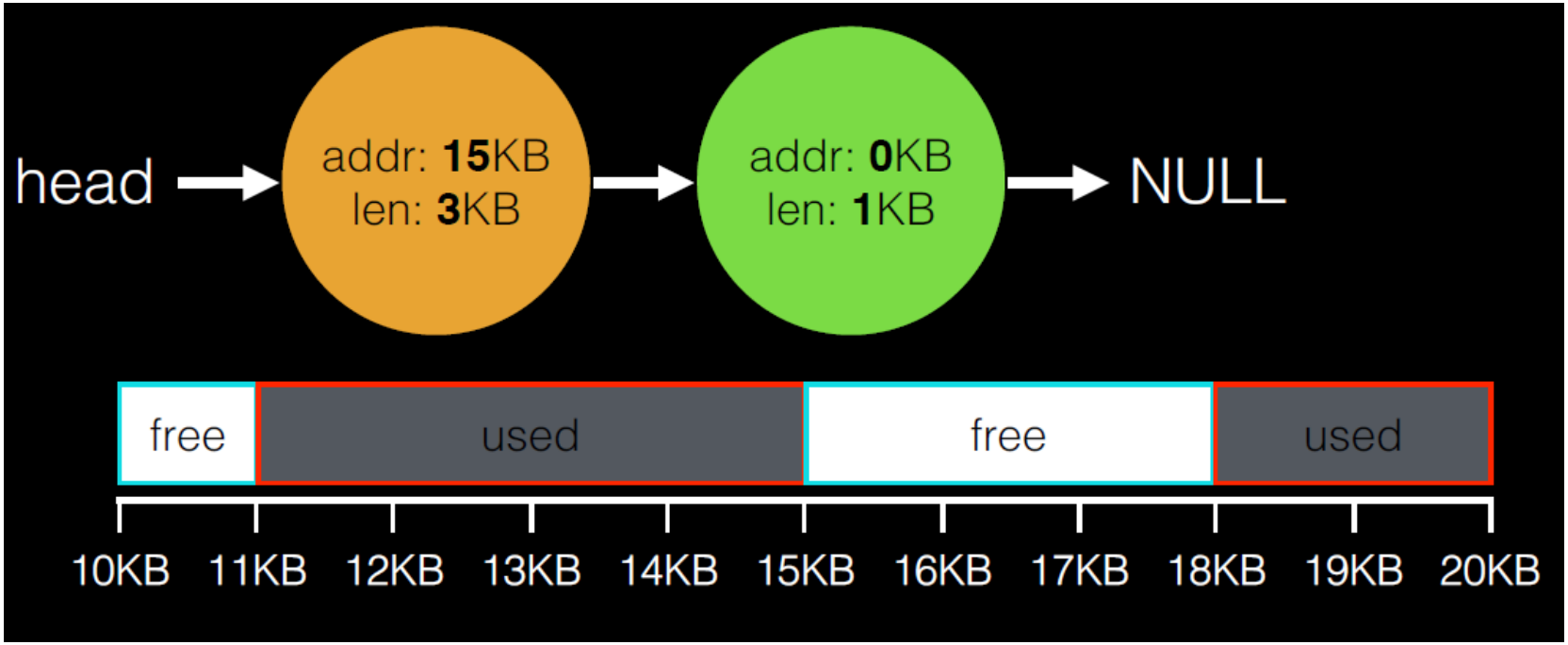
Free List: free



Free List: coalesce



Free List: coalesce

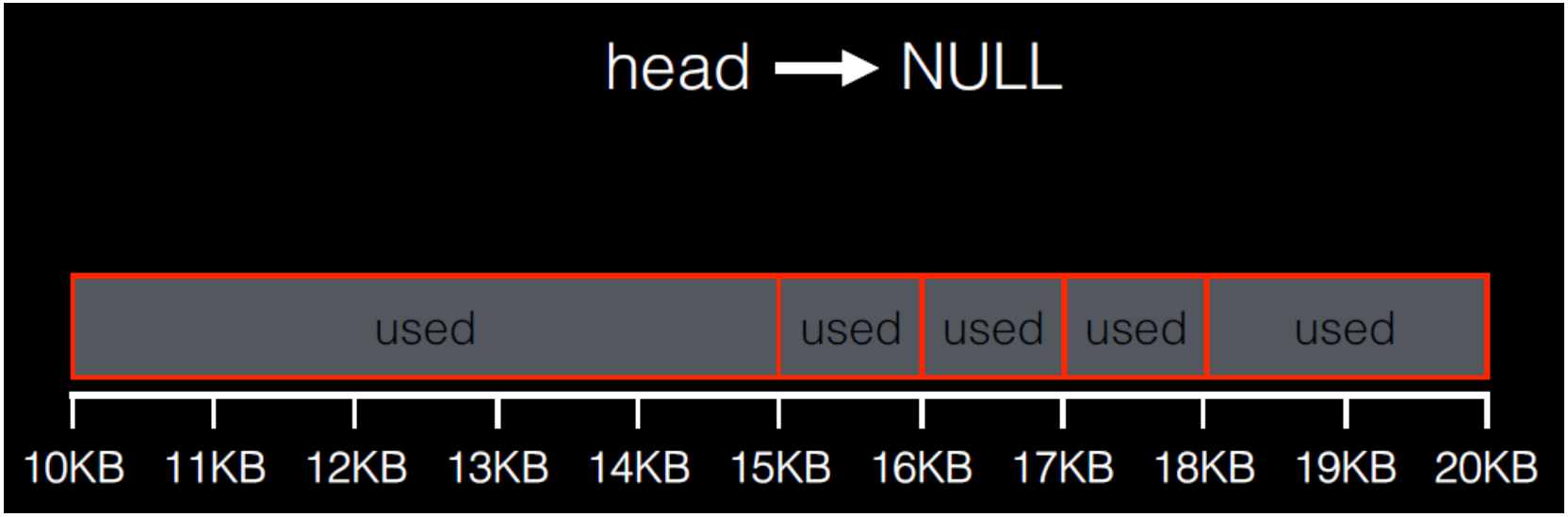


When coalescing is even trickier

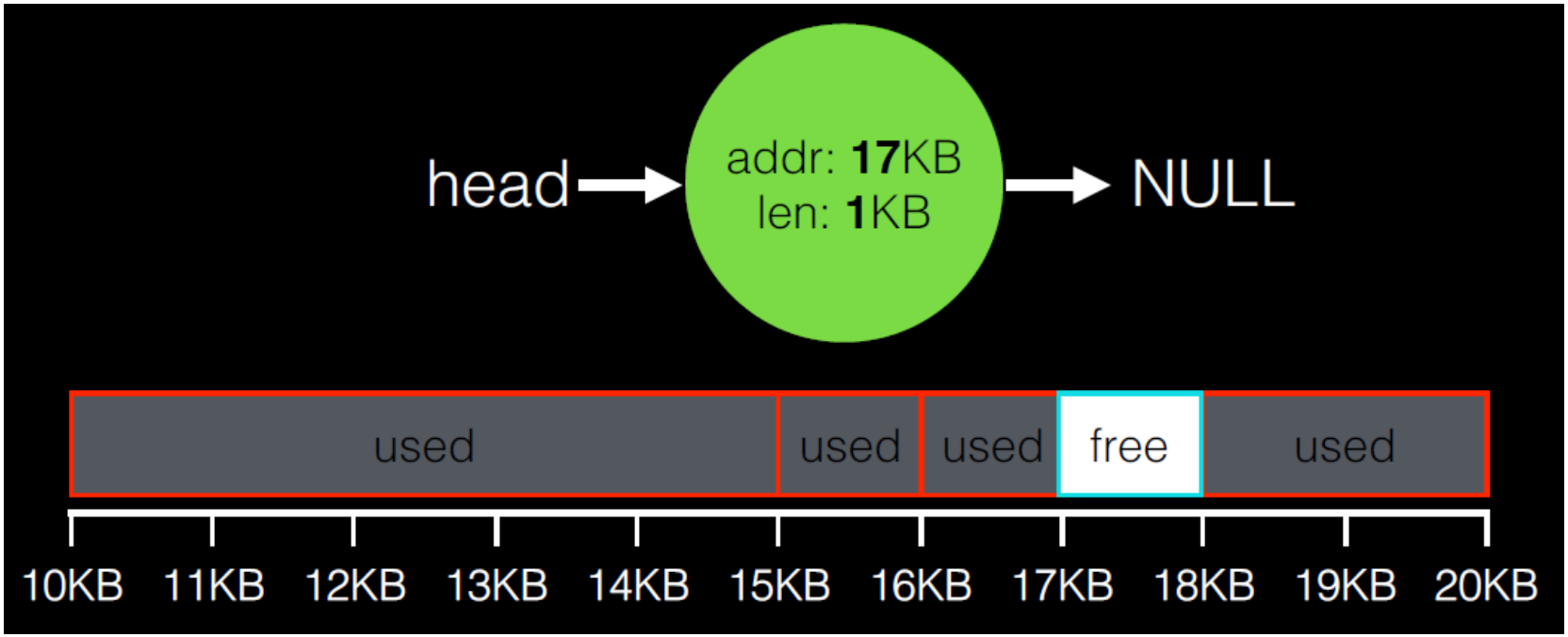
Do we ever have to coalesce multiple areas?

free order may be arbitrary:
free(17KB);
" free(15KB);
" free(16KB)

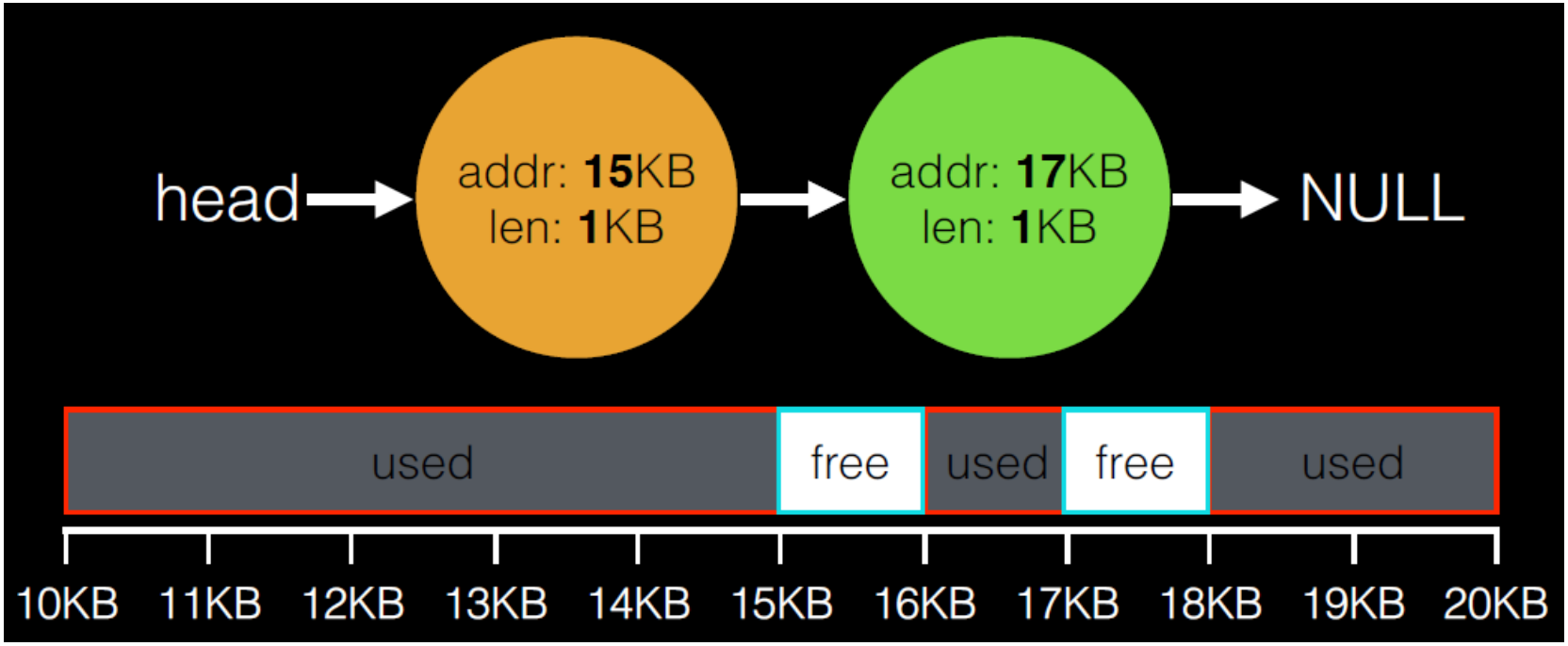
Double Coalesce



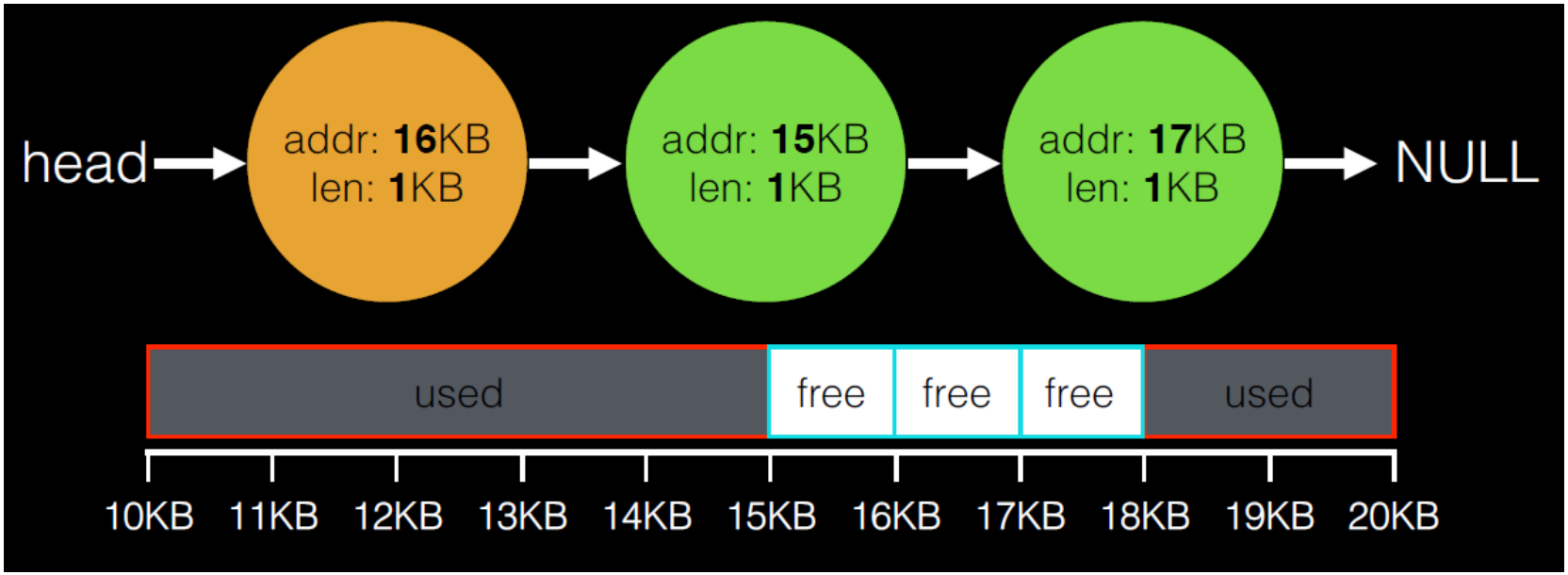
Double Coalesce



Double Coalesce



Double Coalesce



Bookkeeping: free list

What fields do we need for each node in linked list?

```
struct node {
        int size;
        void *addr;
        struct node* next;
 }
```

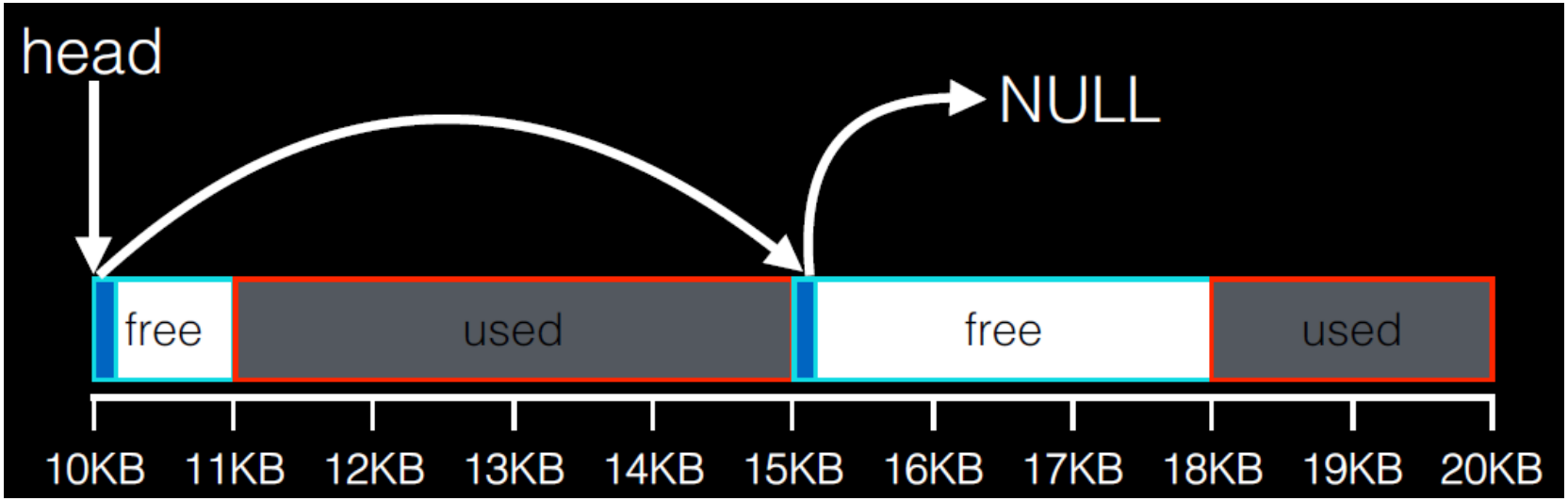
How do we allocate memory for new nodes?

- store them free space!
- addr = ((void *)node + sizeof(*node)

Free List


free
used
free
used
10KB
11KB
12KB
13KB
14KB
15KB
16KB
17KB
18KB
19KB
20KB

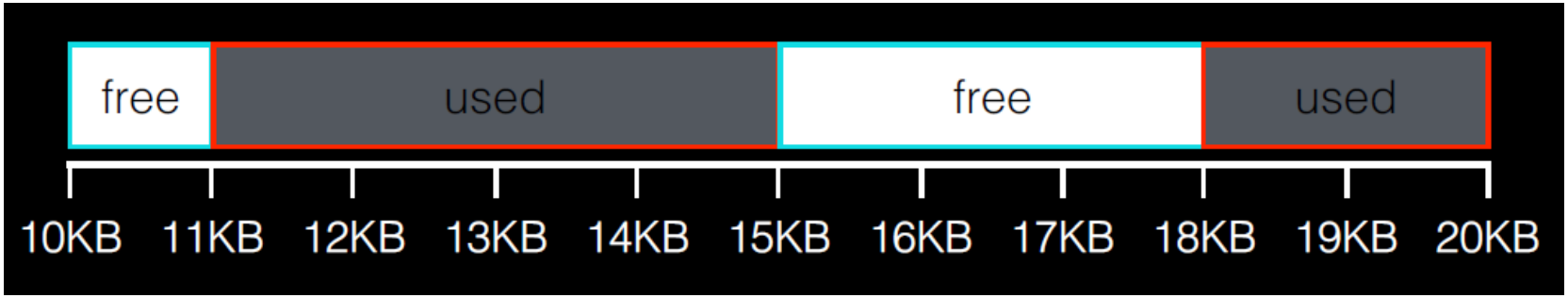
Free List


head
NULL
free
used
free
used
10KB
11KB
12KB
13KB
14KB
15KB
16KB
17KB
18KB
19KB
20KB

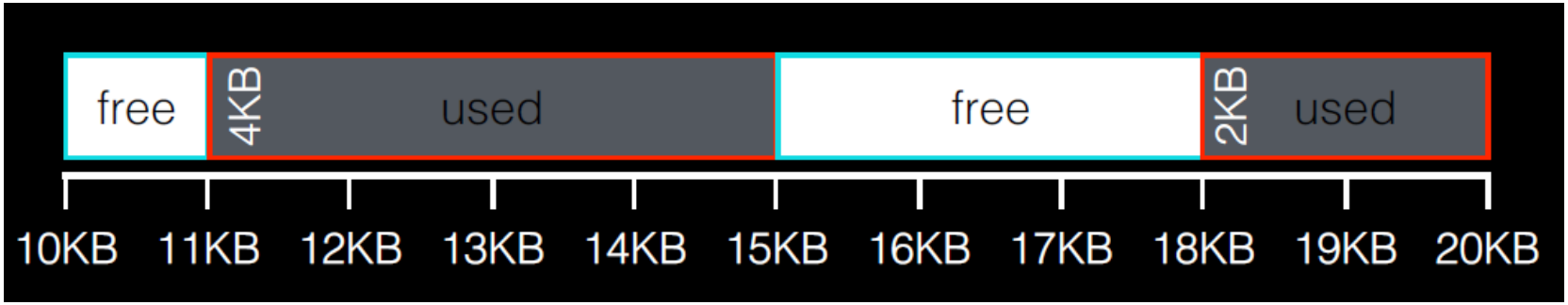
Bookkeeping

Need to know size+location of free spaces
 - for malloc()
Need to know size of used spaces
 - for free()

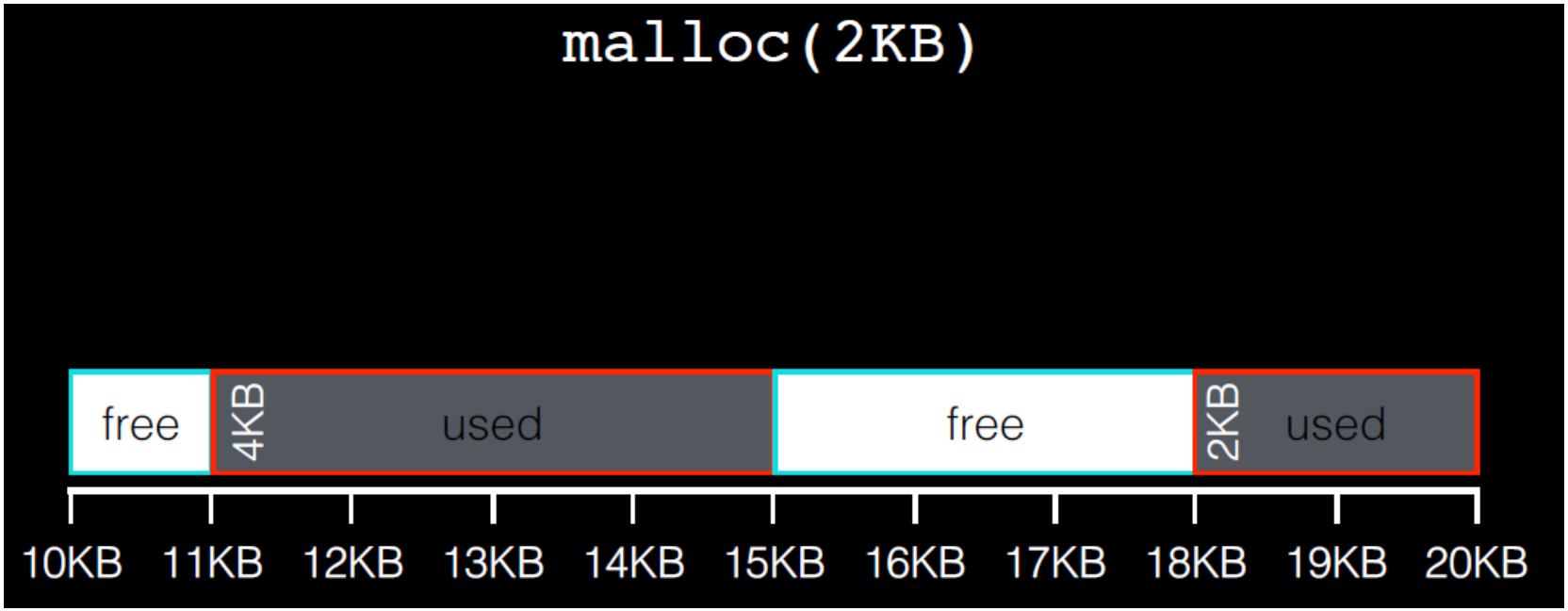
Used Size


free
used
free
used
10KB
11KB
12KB
13KB
14KB
15KB
16KB
17KB
18KB
19KB
20KB

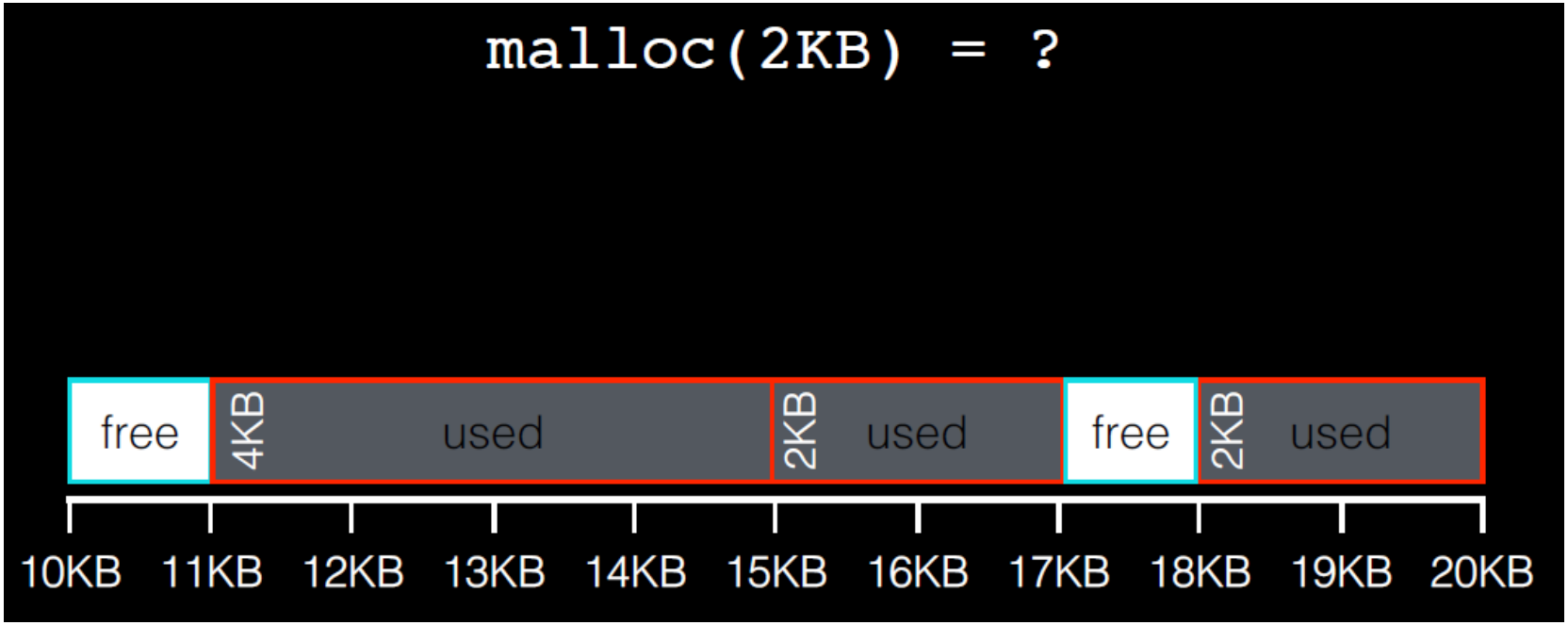
Used size


free
4KB
used
free
2KB
used
10KB
11KB
12KB
13KB
14KB
15KB
16KB
17KB
18KB
19KB
20KB

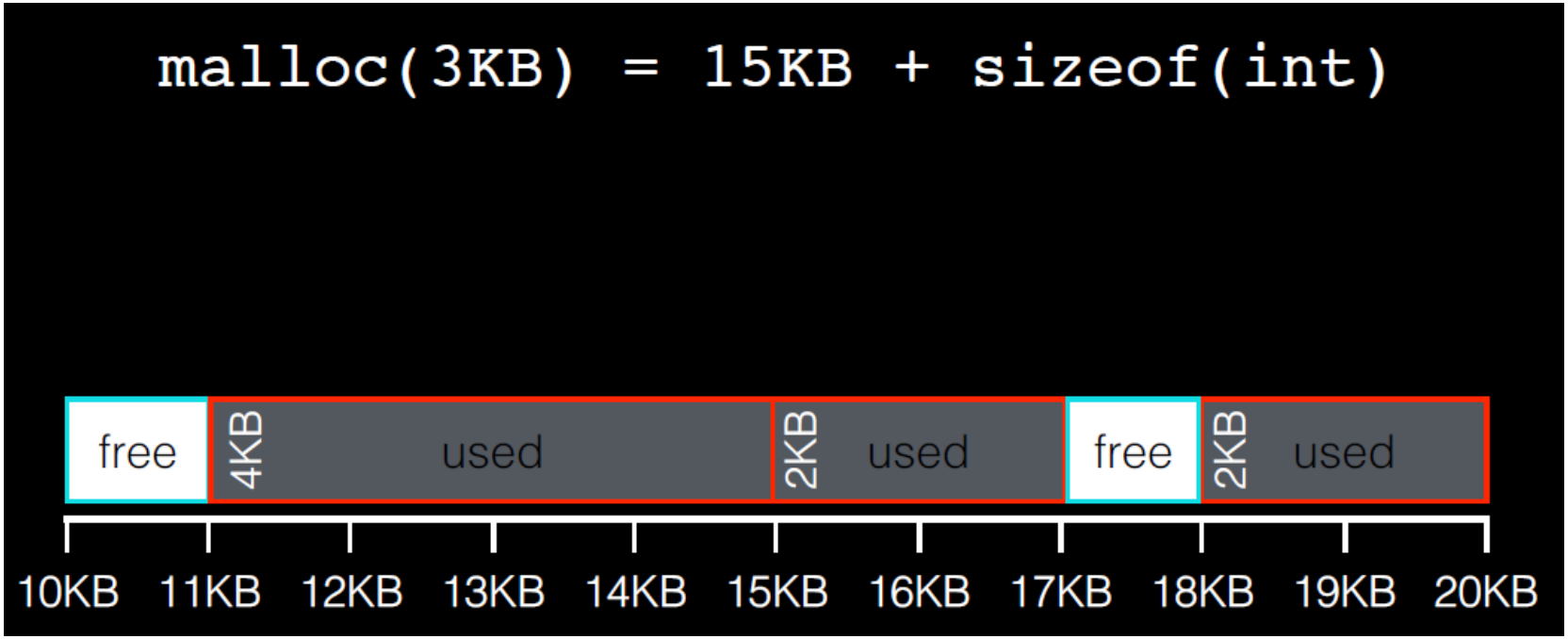
Used Size


malloc(2KB)
free
4KB
used
free
2KB
used
10KB
11KB
12KB
13KB
14KB
15KB
16KB
17KB
18KB
19KB
20KB

Used Size


malloc(2KB) = ?
free
4KB
used
2KB
used
free
2KB
used
10KB
11KB
12KB
13KB
14KB
15KB
16KB
17KB
18KB
19KB
20KB

Used Size


malloc(3KB) = 15KB + sizeof(int)
free
4KB
used
2KB
used
free
2KB
used
10KB
11KB
12KB
13KB
14KB
15KB
16KB
17KB
18KB
19KB
20KB

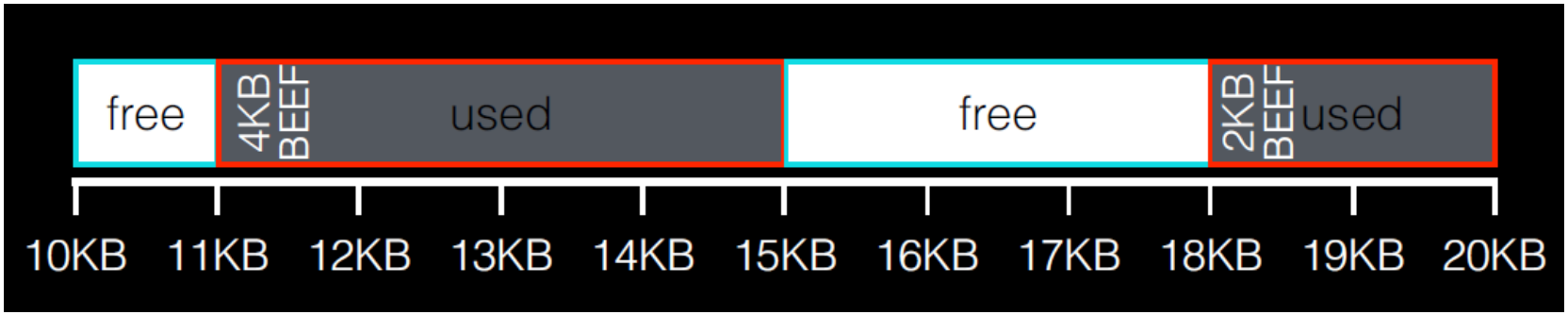
Magic Numbers

Can malloc/free catch bugs for you?
- double frees?
- overflows

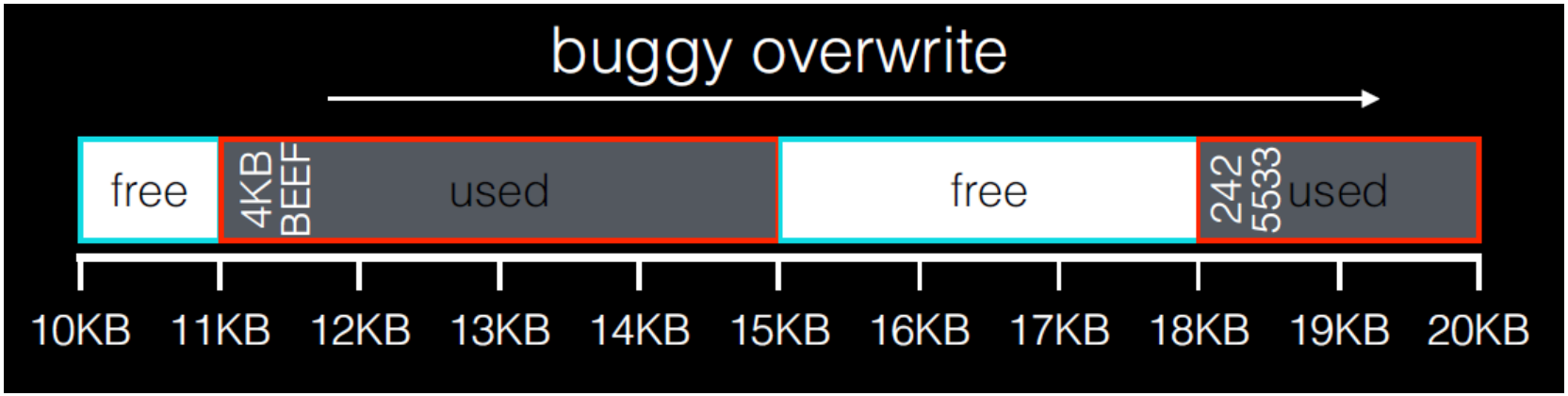
Add magic number to each allocated segment,
if it is overwritten, there's a bug!

What can you spell with 0 - 1 and A - F?
0xDEADBEEF, 0xFEEDFACE, …

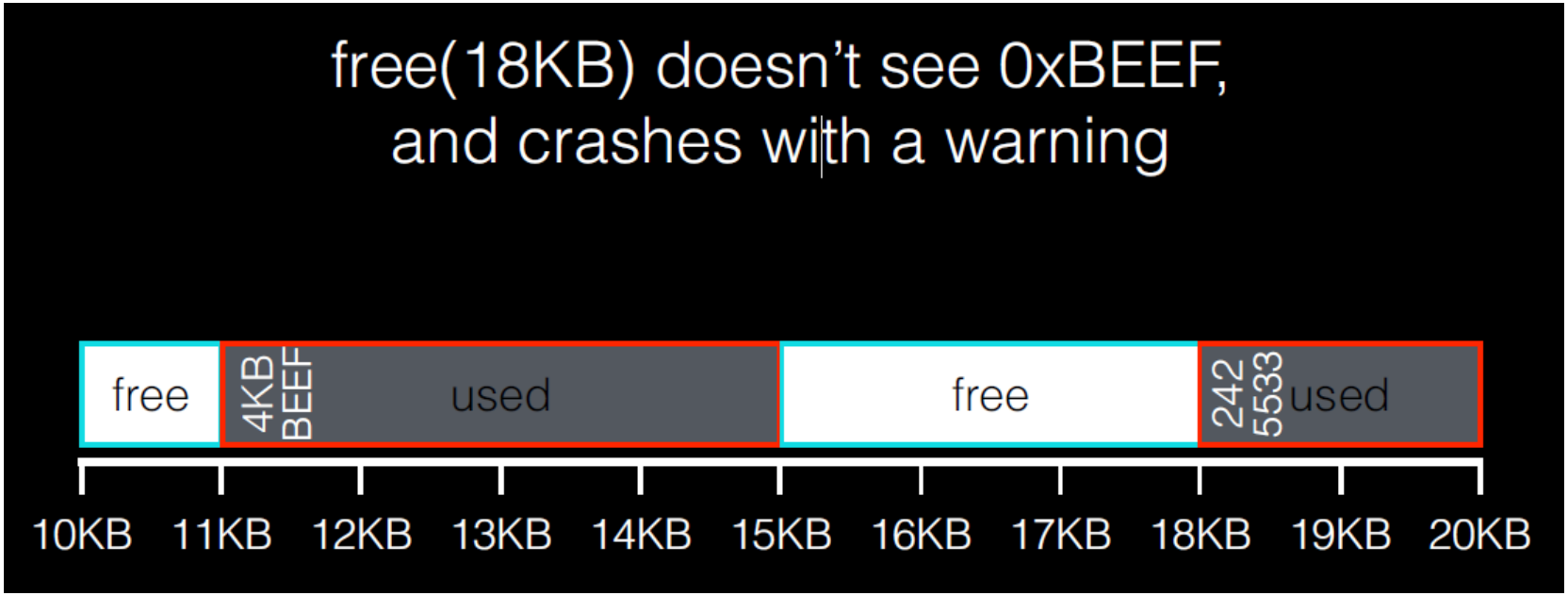
Magic


free
4KB
BEEF
used
free
2KB
BEEF
used
10KB
11KB
12KB
13KB
14KB
15KB
16KB
17KB
18KB
19KB
20KB

Magic


buggy overwrite
free
4KB
BEEF
used
free
242
5533
used
10KB
11KB
12KB
13KB
14KB
15KB
16KB
17KB
18KB
19KB
20KB

Magic


free(18KB) doesn't see 0xBEEF,
and crashes with a warning
free
4KB
BEEF
used
free
242
5533
used
10KB
11KB
12KB
13KB
14KB
15KB
16KB
17KB
18KB
19KB
20KB

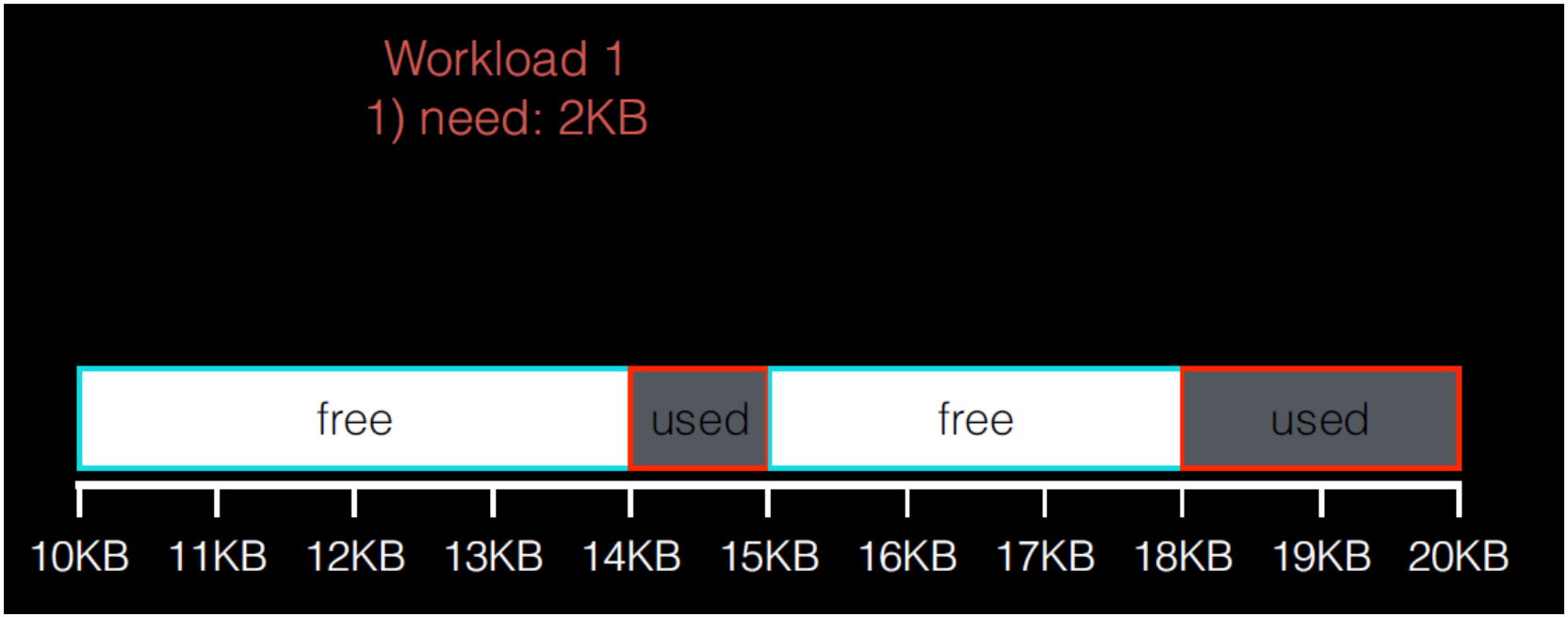
Allocator Policy

Which free space to consider?

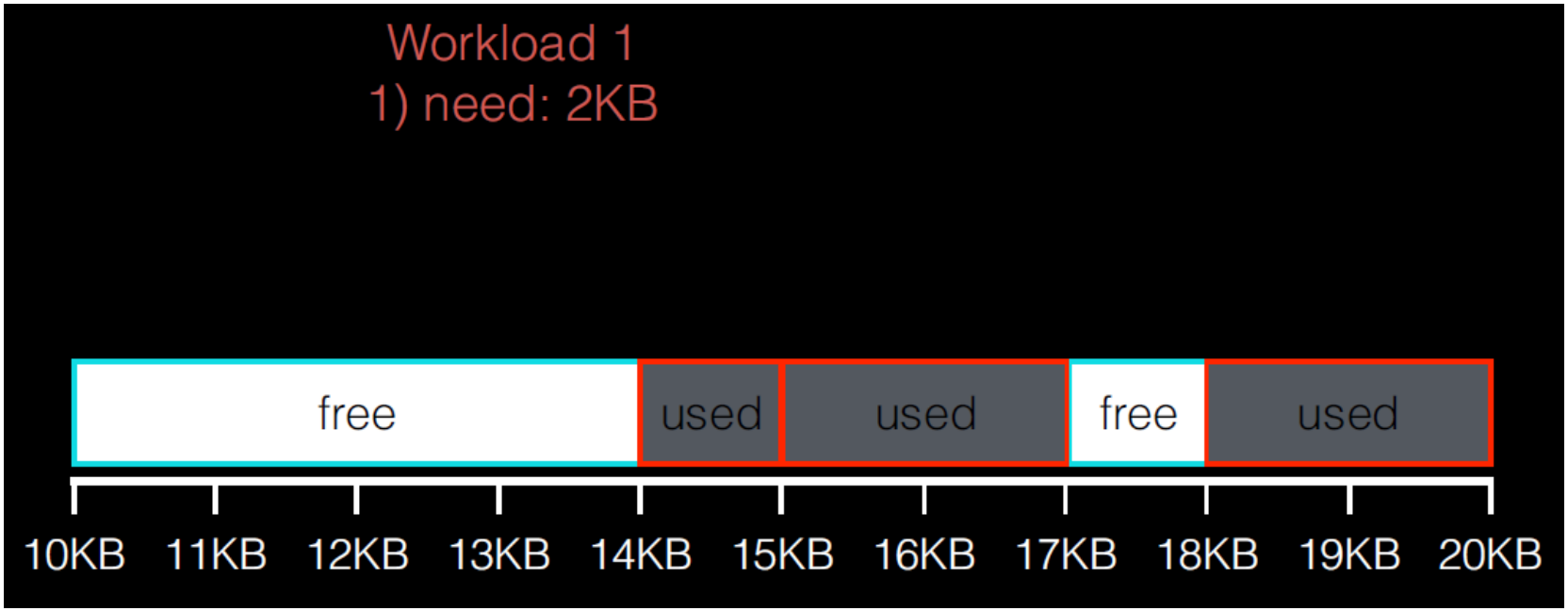
Of those considered, which to use?

No perfect solutions!

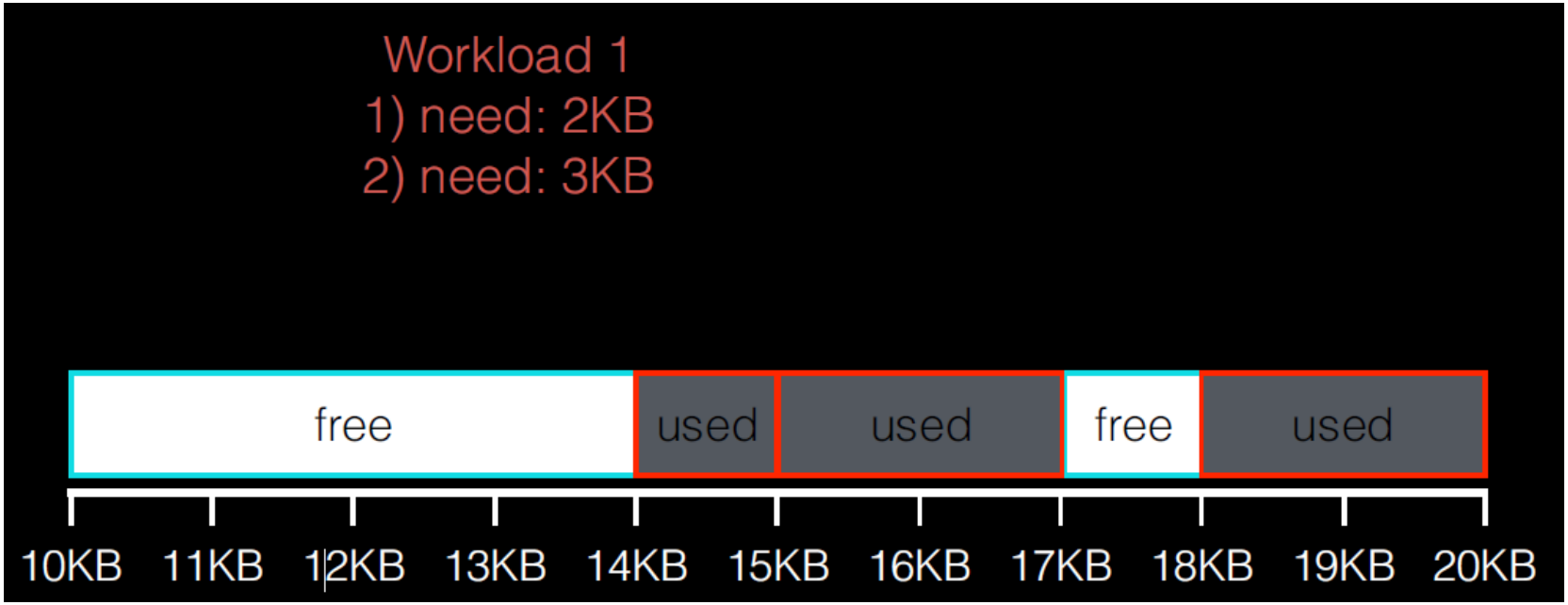
Where to allocate?



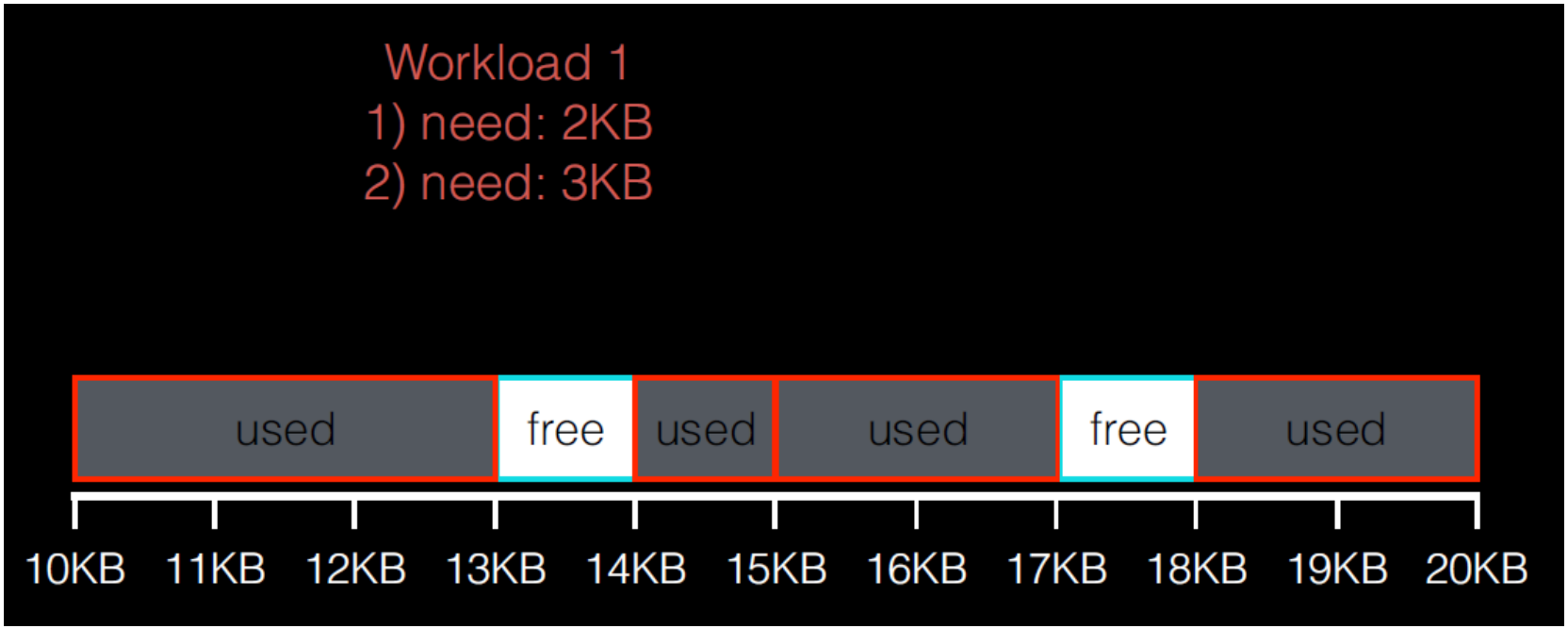
Where to allocate?



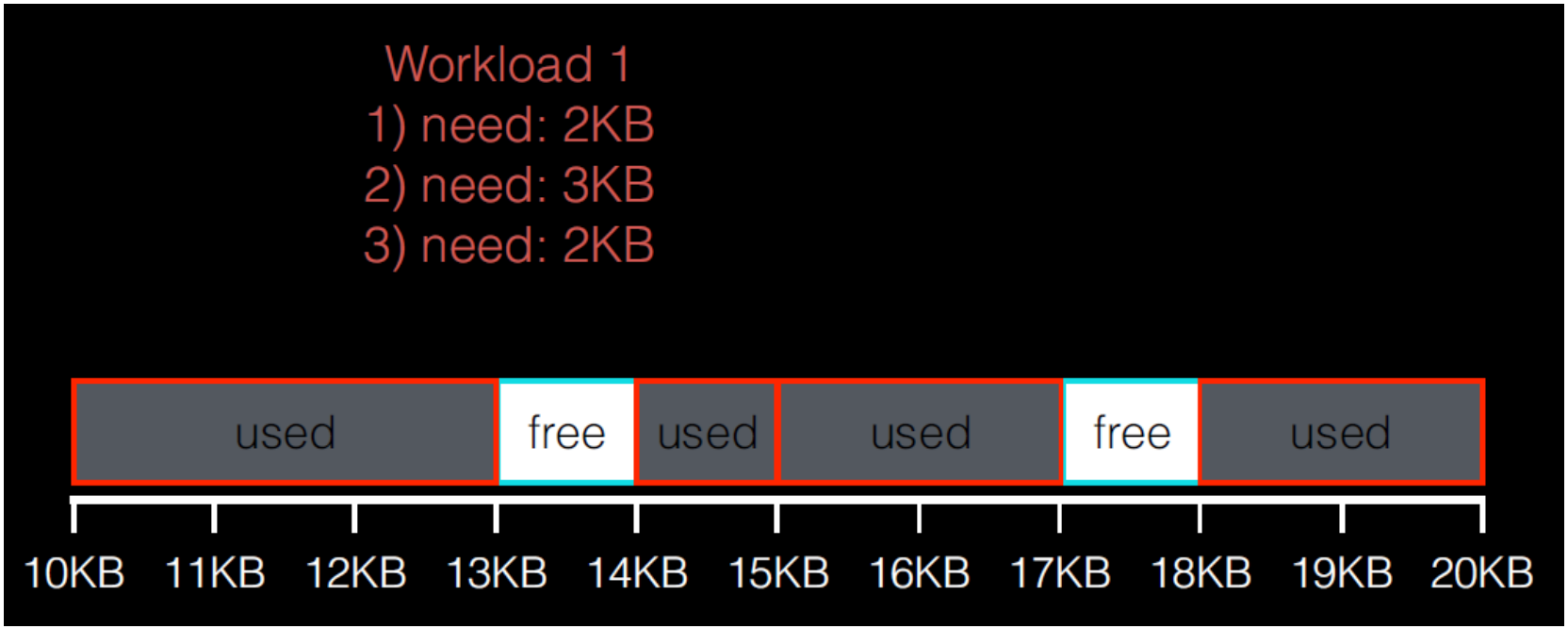
Where to allocate?



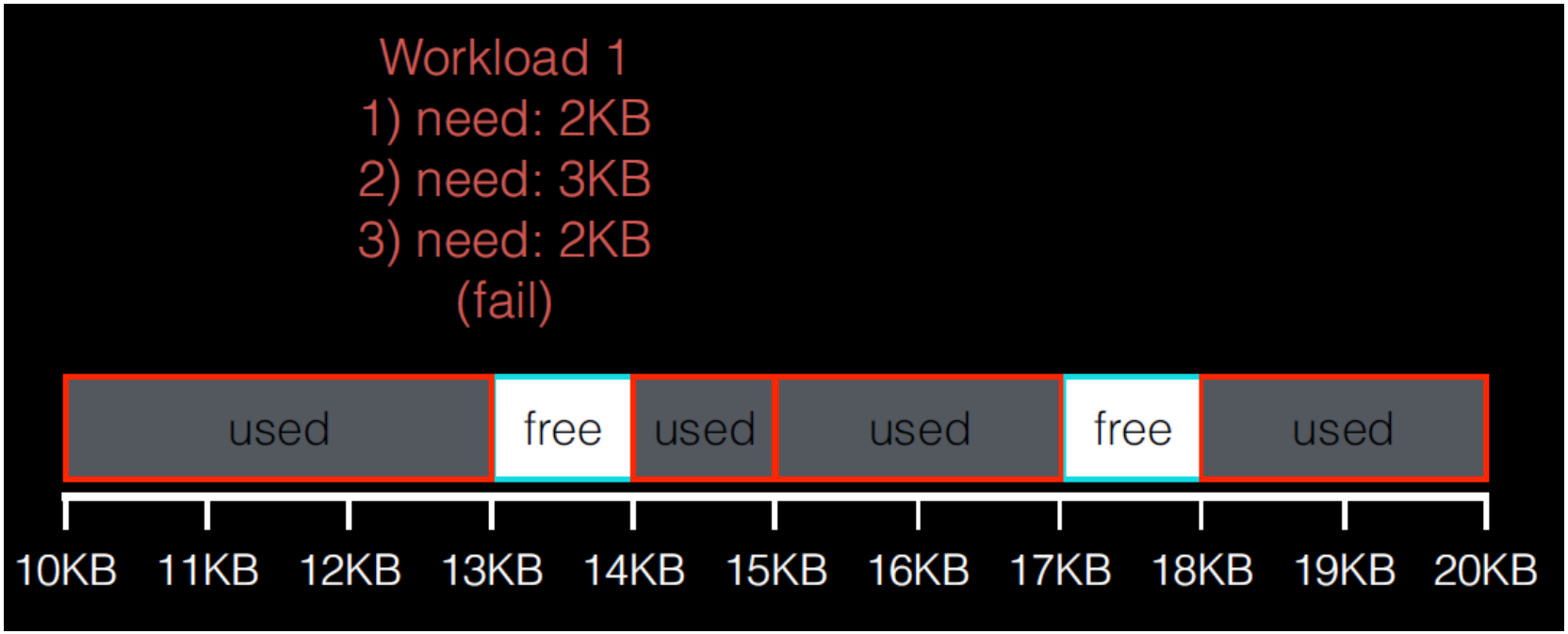
Where to allocate?



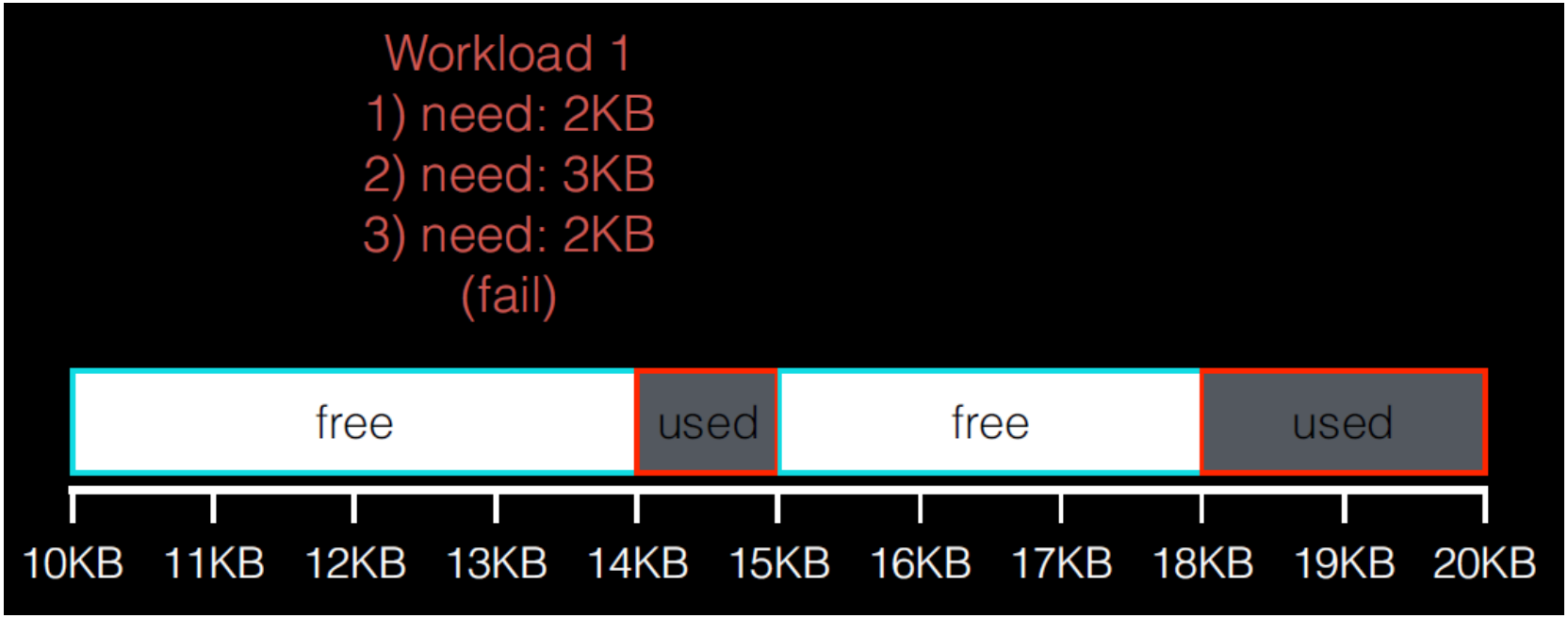
Where to allocate?



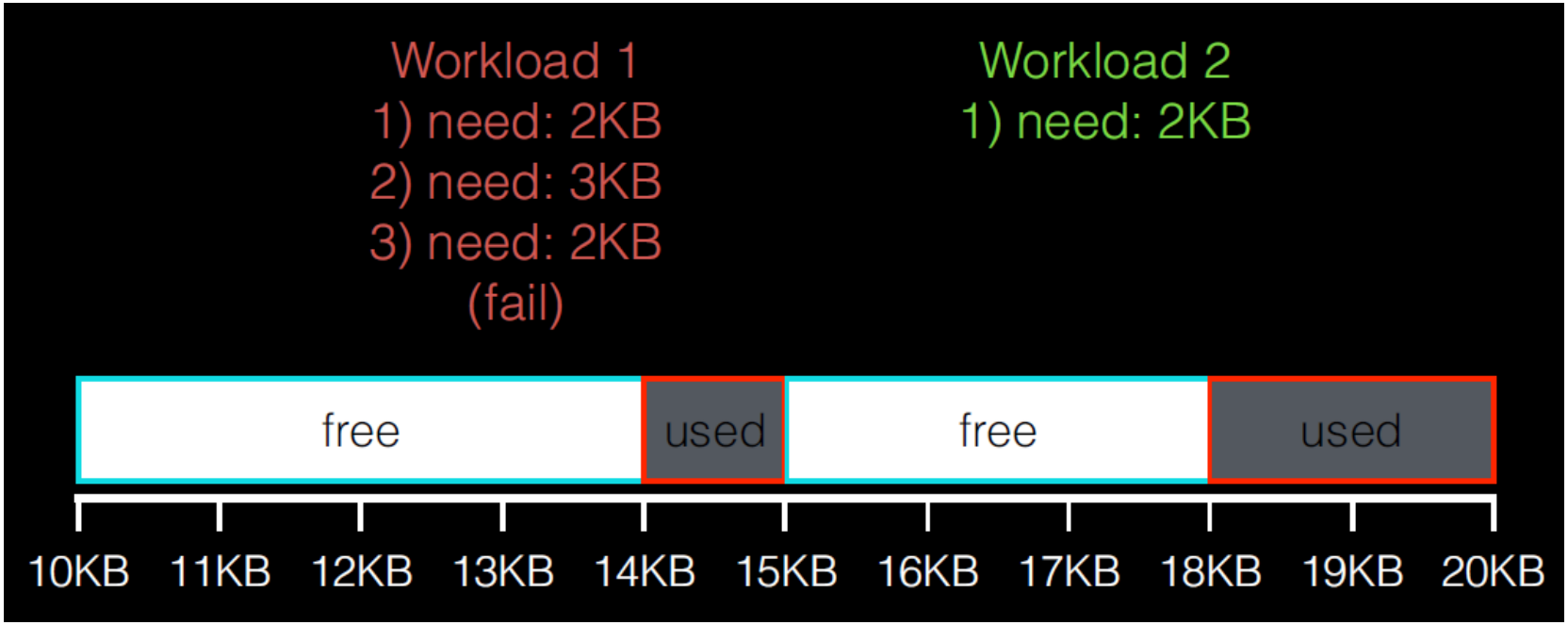
Where to allocate?



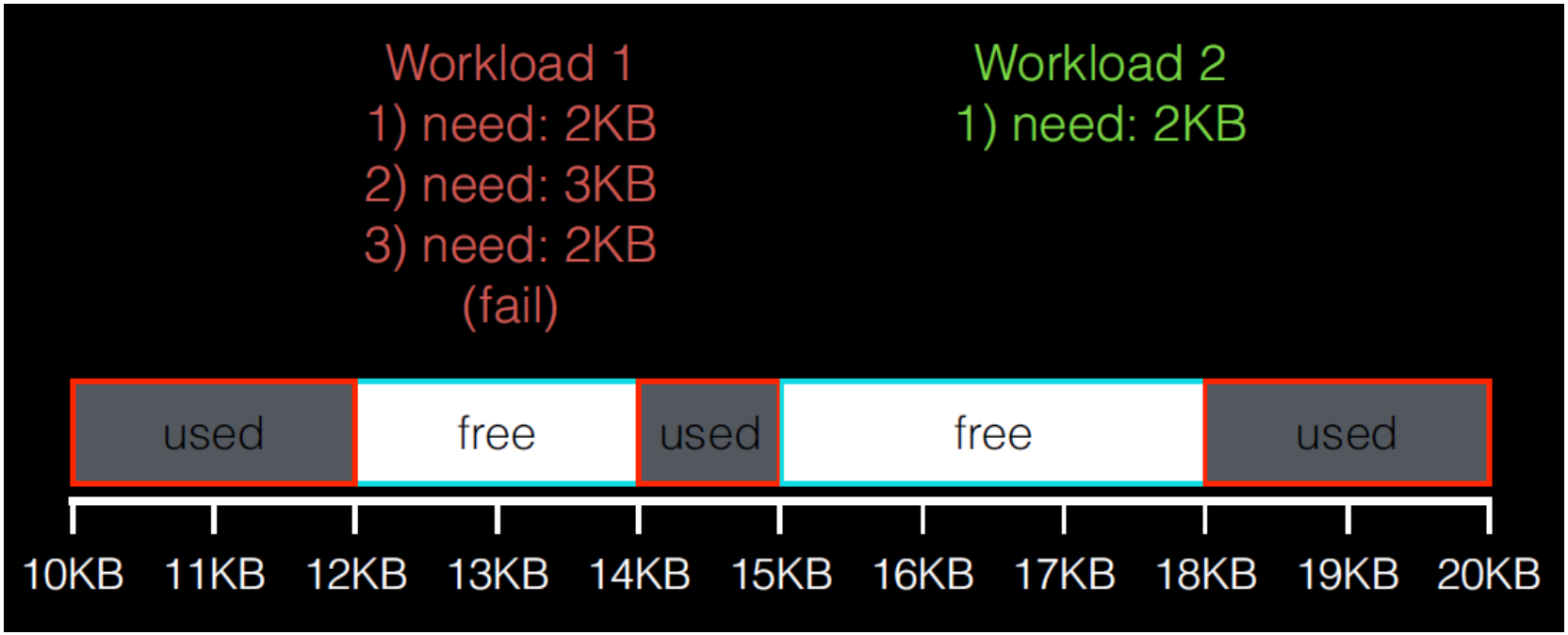
Where to allocate?



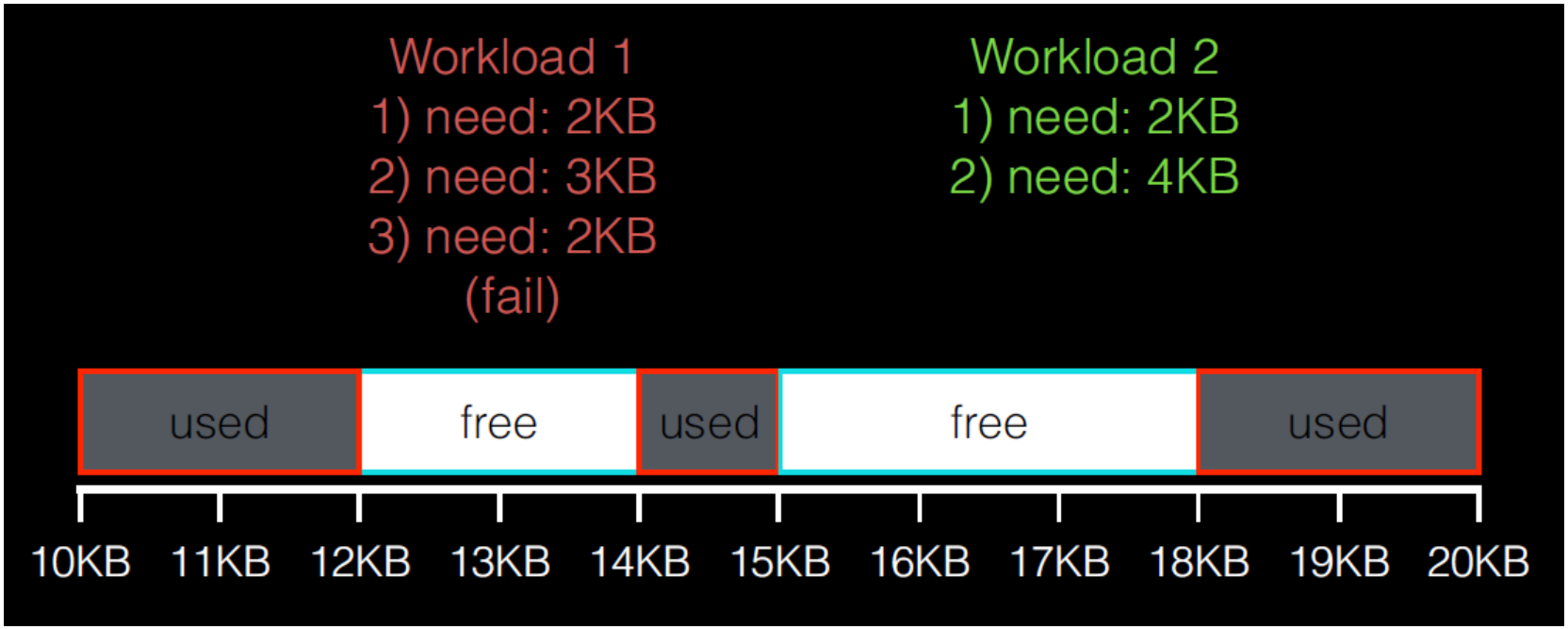
Where to allocate?



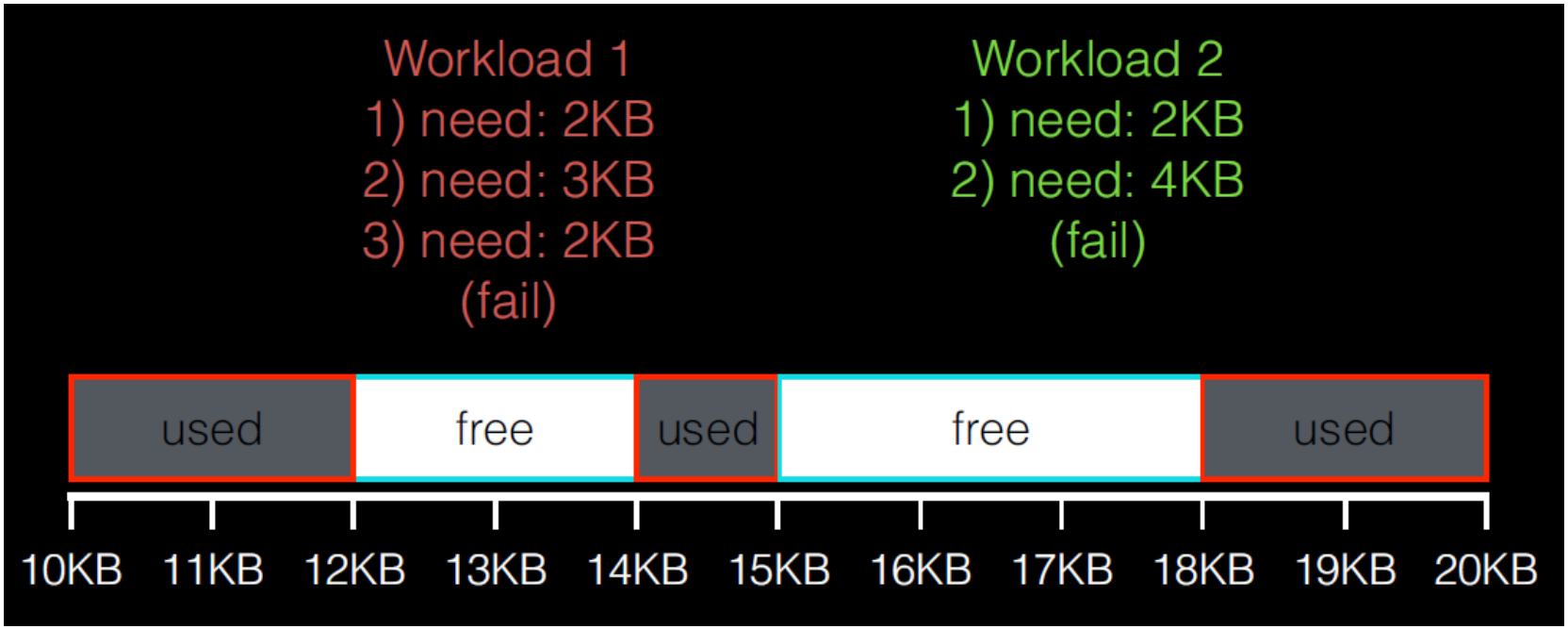
Where to allocate?



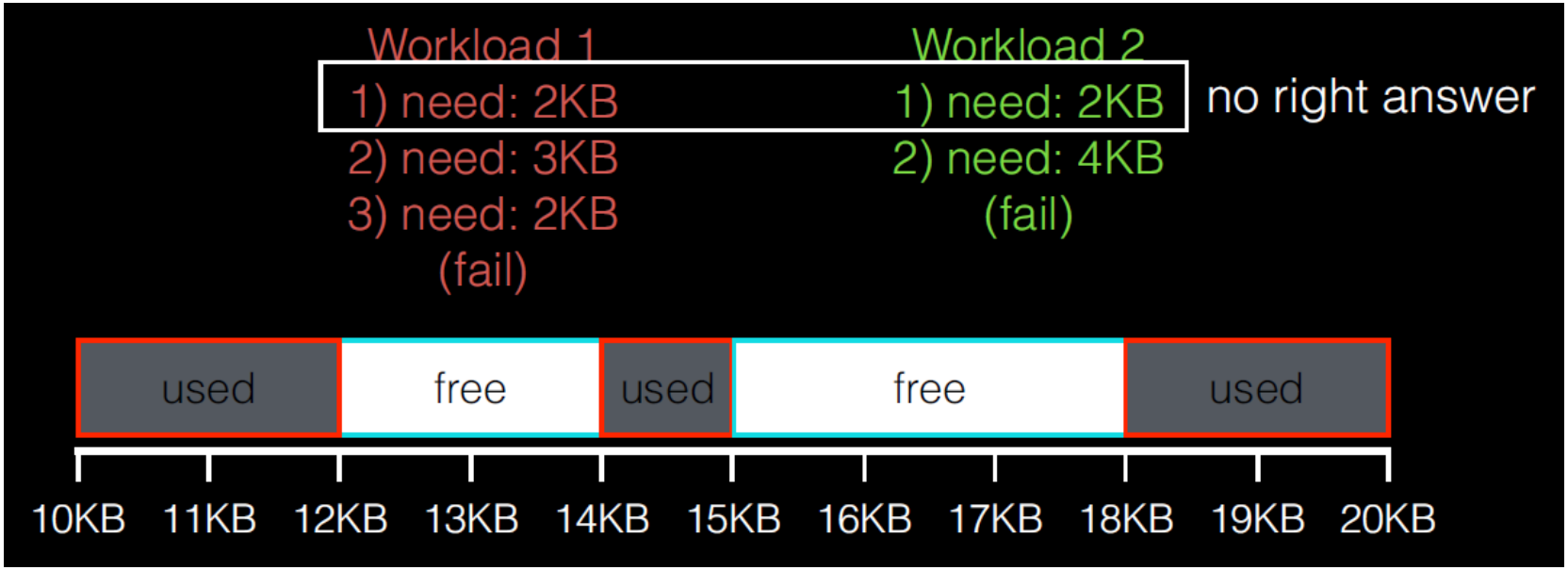
Where to allocate?



Where to allocate?



Where to allocate?



Review: Scheduler Vocabulary

Workload: set of job descriptions

Scheduler: logic that decides when jobs run

Metric: measurement of scheduling quality

Scheduler "algebra", given 2 variables, find the 3rd:

f(**W**, **S**) = **M**

Allocator Vocabulary

Workload: series of malloc()'s and free()'s

Allocator: logic that gives memory to processes

Metric: measurement of allocation quality

Allocator "algebra", given 2 variables, find the 3rd:

f(**W**, **A**) = **M**

Allocator Basics

Workload:
- operations
- addresses
- sizes

Allocators:
- Best fit
- Worst fit
- First fit
- Next fit
- Slab
- Buddy

Metrics:
- internal fragmentation
- external fragmentation
- search time

Summary

malloc provides a convenient library service to programs, abstracting the raw heap

Allocation is challenging because
- there is no right answer
- we can't use malloc ourselves
- expensive searching for ways to coalesce