[537] Paging

Tyler Harter
9/17/14

Overview

Review Segmentation

Paging (Chapter 18)

Break + Announcements?

Memory Allocators (Chapter 17)

Discuss P2

Review: Segmentation

Virtual Memory Approaches

Approaches (covered Monday):

- **Time Sharing**: one process uses RAM at a time
- **Static Relocation**: rewrite code before run
- **Base**: add a base to virt addr to get phys
- **Base+Bounds**: also check phys is in range
- **Segmentation**: many base+bound pairs

Virtual Memory Approaches

Approaches (covered Monday):

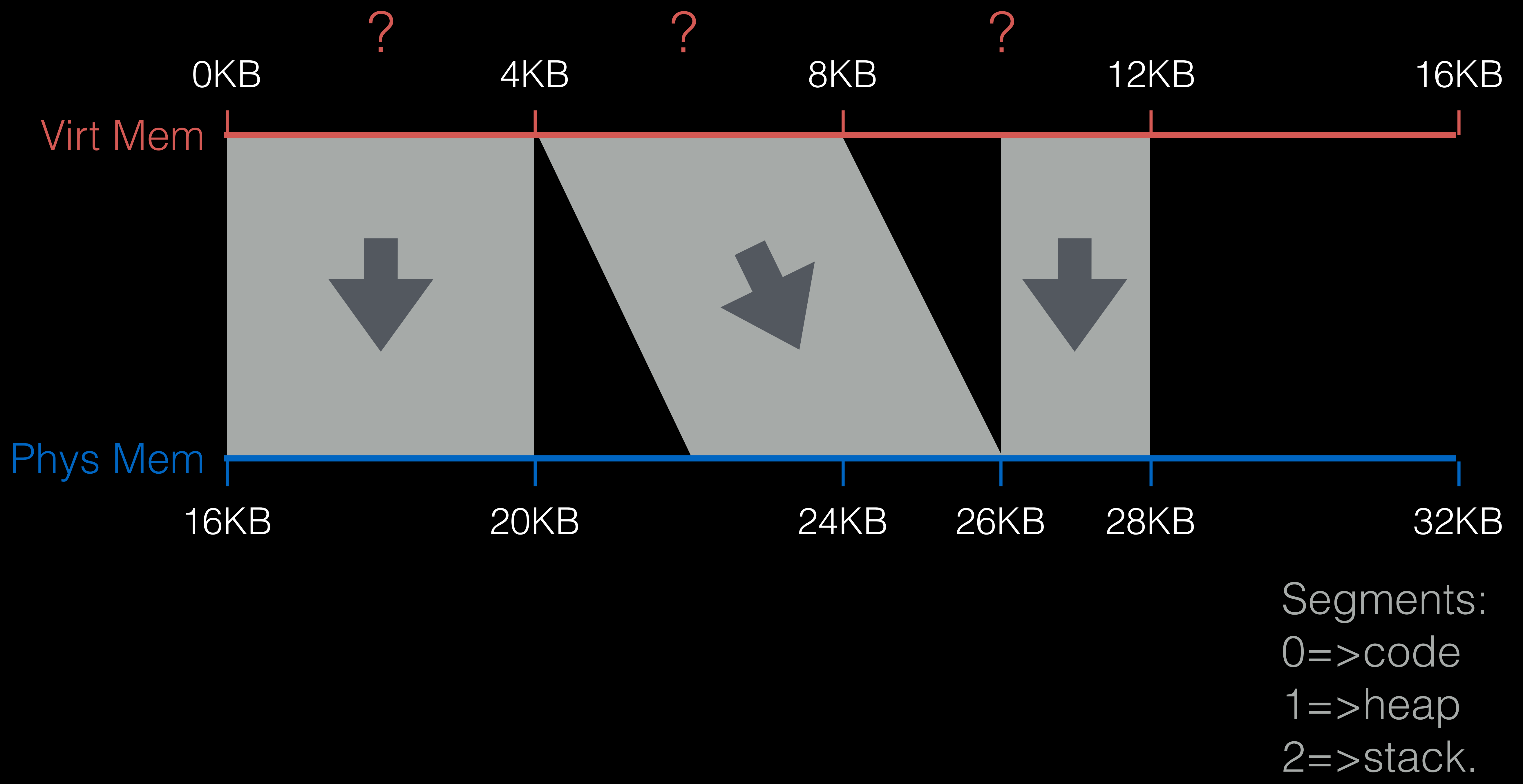
- **Time Sharing**: one process uses RAM at a time
- **Static Relocation**: rewrite code before run
- **Base**: add a base to virt addr to get phys
- **Base+Bounds**: also check phys is in range
- **Segmentation**: many base+bound pairs

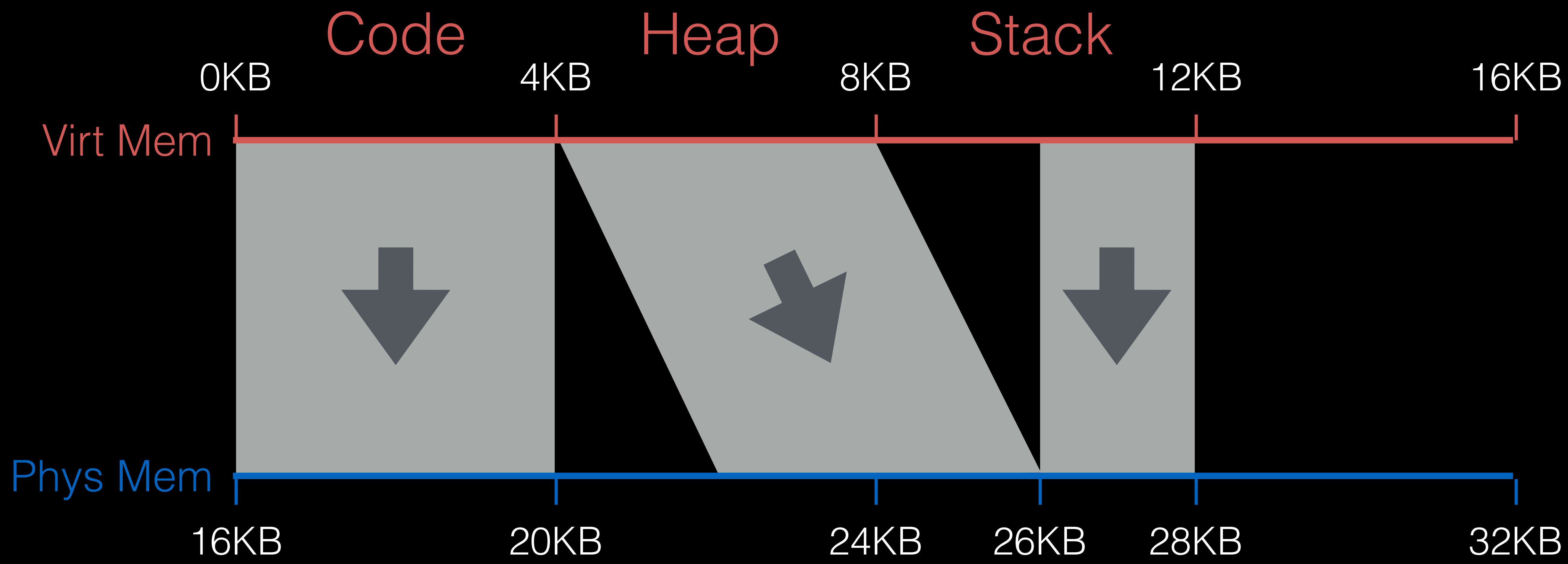
Segmentation Example

Assume a 14-bit virtual addresses,
with the high 2 bits indicating the segment.

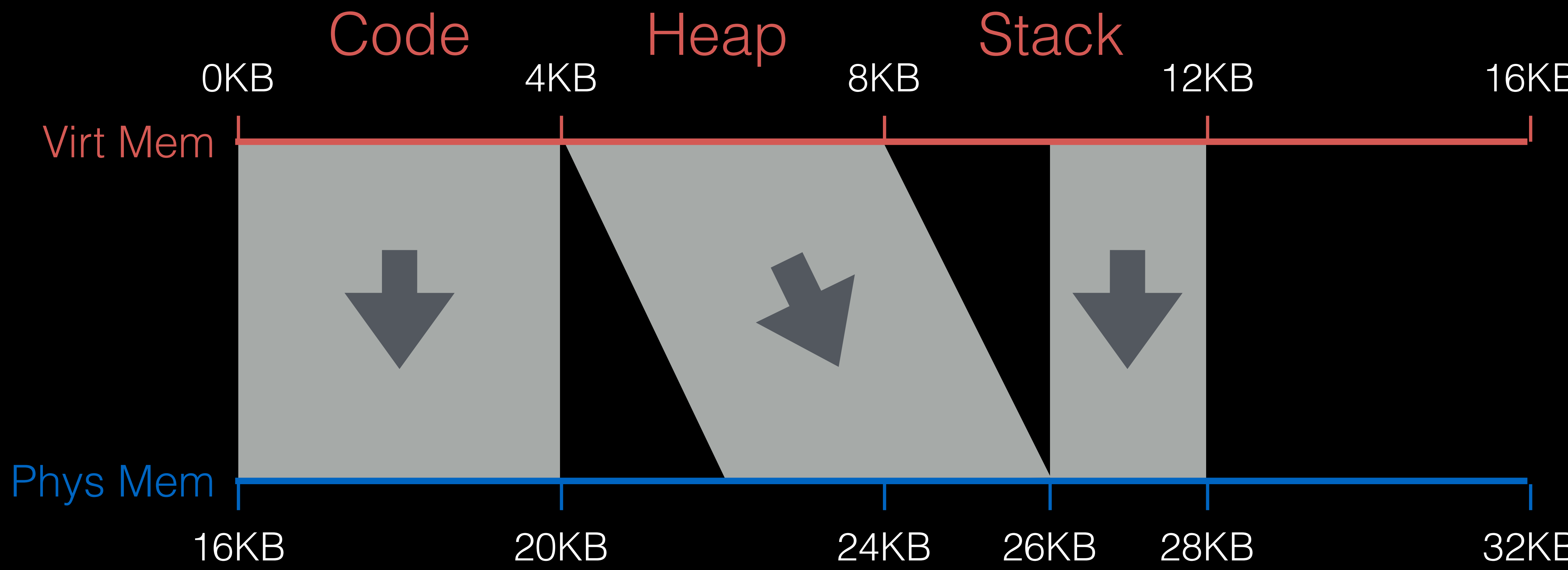
Assume 0=>code, 1=>heap, and 2=>stack.

What virtual addresses *could* be valid for each segment?


?
?
?
0KB
4KB
8KB
12KB
16KB
Virt Mem
Phys Mem
16KB
20KB
24KB
26KB
28KB
32KB
Segments:
0=>code
1=>heap
2=>stack.

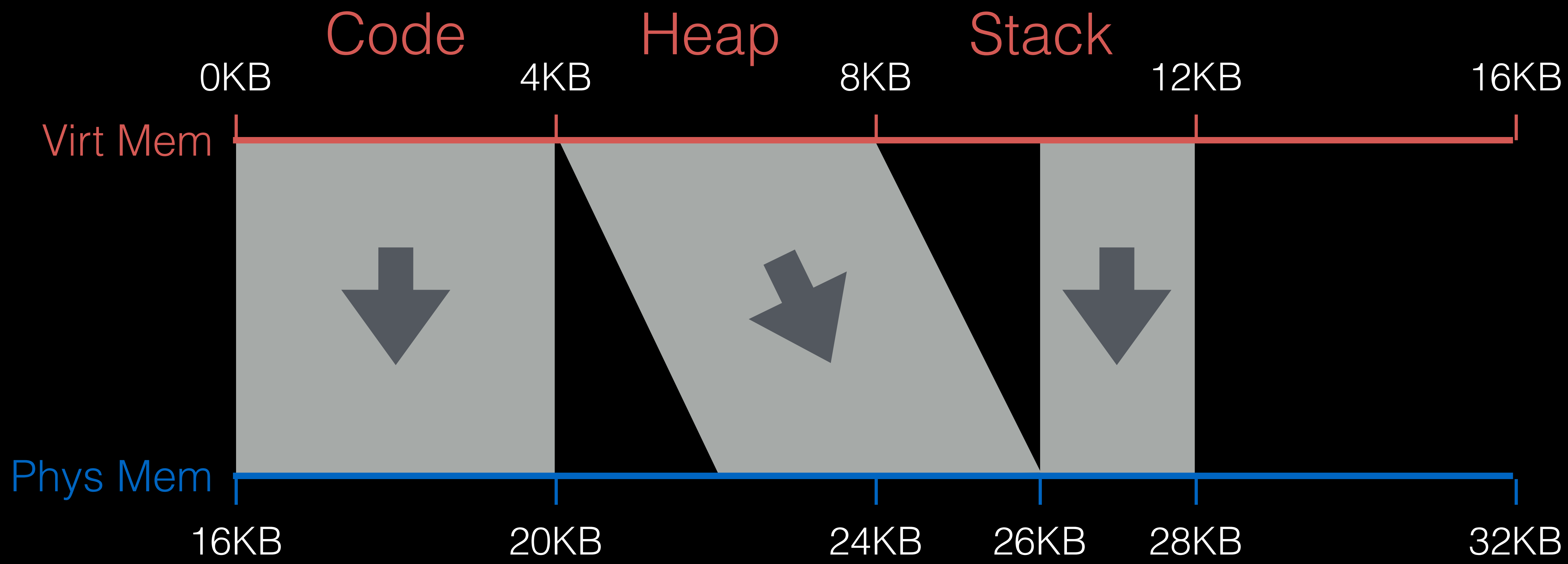


Segments:
0=>code
1=>heap
2=>stack.



Seg	Base	Bounds
0	?	?
1	?	?
2	?	?

Segments:
0=>code
1=>heap
2=>stack.



Seg	Base	Bounds
0	16KB	20KB
1	22KB	26KB
2	26KB	28KB

Segments:
0=>code
1=>heap
2=>stack.

Memory Accesses

Memory Accesses:

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

Seg	Base	Bounds
code	16KB (0x4000)	20KB (0x5000)
heap	22KB (0x5800)	26KB (0x6800)
stack	26KB (0x6800)	28KB (0x7000)

Memory Accesses

Memory Accesses:

Fetch instruction at addr 0x4010

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

Seg	Base	Bounds
code	16KB (0x4000)	20KB (0x5000)
heap	22KB (0x5800)	26KB (0x6800)
stack	26KB (0x6800)	28KB (0x7000)

Memory Accesses

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

Memory Accesses:

Fetch instruction at addr 0x4010

Exec, load from addr 0x5900

Seg	Base	Bounds
code	16KB (0x4000)	20KB (0x5000)
heap	22KB (0x5800)	26KB (0x6800)
stack	26KB (0x6800)	28KB (0x7000)

Memory Accesses

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

Memory Accesses:

Fetch instruction at addr 0x4010

Exec, load from addr 0x5900

Seg	Base	Bounds
code	16KB (0x4000)	20KB (0x5000)
heap	22KB (0x5800)	26KB (0x6800)
stack	26KB (0x6800)	28KB (0x7000)

Memory Accesses

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

Memory Accesses:

Fetch instruction at addr 0x4010
Exec, load from addr 0x5900
Fetch instruction at addr 0x4013

Seg	Base	Bounds
code	16KB (0x4000)	20KB (0x5000)
heap	22KB (0x5800)	26KB (0x6800)
stack	26KB (0x6800)	28KB (0x7000)

Memory Accesses

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

Memory Accesses:

Fetch instruction at addr 0x4010
Exec, load from addr 0x5900
Fetch instruction at addr 0x4013
Exec, no load

Seg	Base	Bounds
code	16KB (0x4000)	20KB (0x5000)
heap	22KB (0x5800)	26KB (0x6800)
stack	26KB (0x6800)	28KB (0x7000)

Memory Accesses

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

Memory Accesses:

Fetch instruction at addr 0x4010
Exec, load from addr 0x5900
Fetch instruction at addr 0x4013
Exec, no load

Seg	Base	Bounds
code	16KB (0x4000)	20KB (0x5000)
heap	22KB (0x5800)	26KB (0x6800)
stack	26KB (0x6800)	28KB (0x7000)

Memory Accesses

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

Memory Accesses:

Fetch instruction at addr 0x4010
Exec, load from addr 0x5900
Fetch instruction at addr 0x4013
Exec, no load
Fetch instruction at addr 0x4019

Seg	Base	Bounds
code	16KB (0x4000)	20KB (0x5000)
heap	22KB (0x5800)	26KB (0x6800)
stack	26KB (0x6800)	28KB (0x7000)

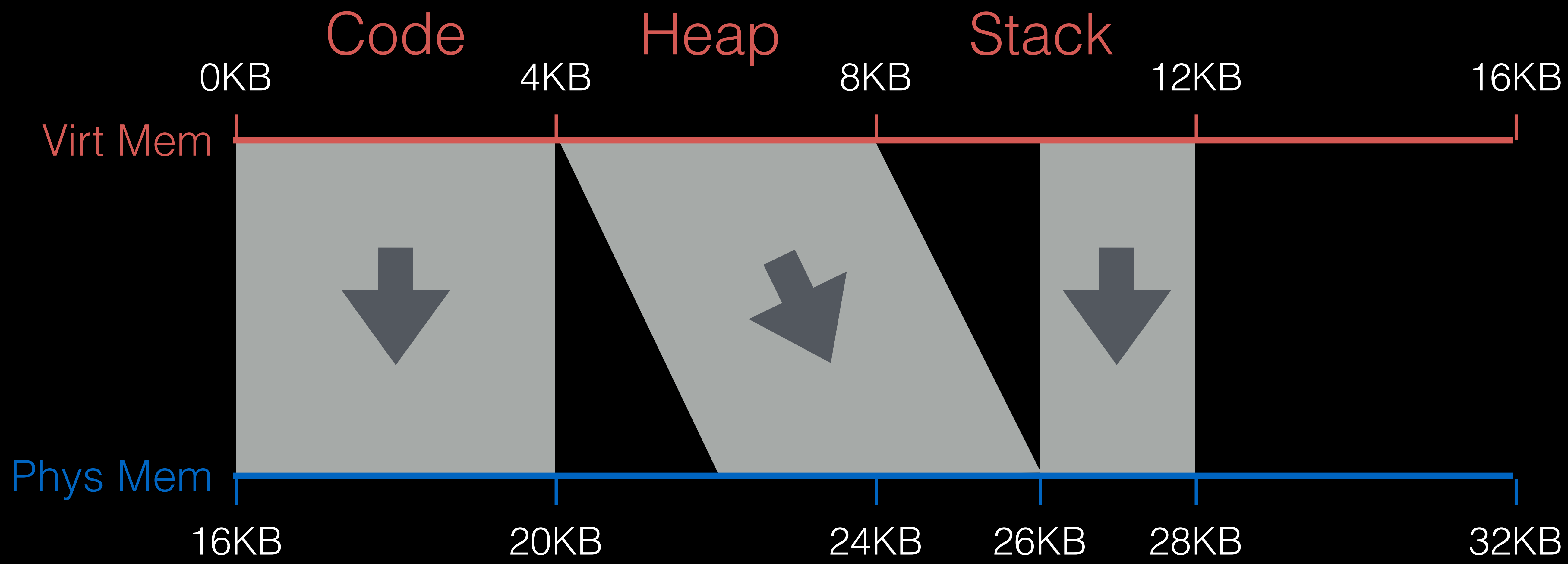
Memory Accesses

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

Memory Accesses:

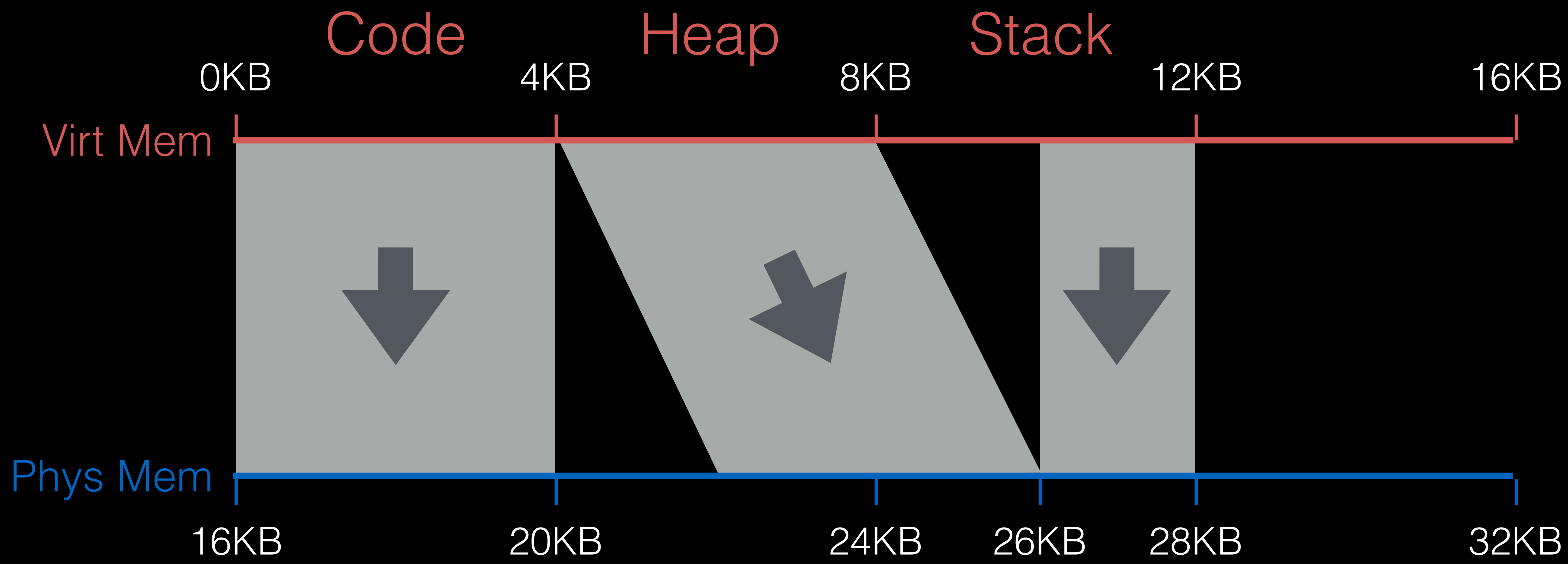
Fetch instruction at addr 0x4010
Exec, load from addr 0x5900
Fetch instruction at addr 0x4013
Exec, no load
Fetch instruction at addr 0x4019
Exec, store to addr 0x5900

Seg	Base	Bounds
code	16KB (0x4000)	20KB (0x5000)
heap	22KB (0x5800)	26KB (0x6800)
stack	26KB (0x6800)	28KB (0x7000)



Seg	Base	Bounds
0	16KB	20KB
1	22KB	26KB
2	26KB	28KB

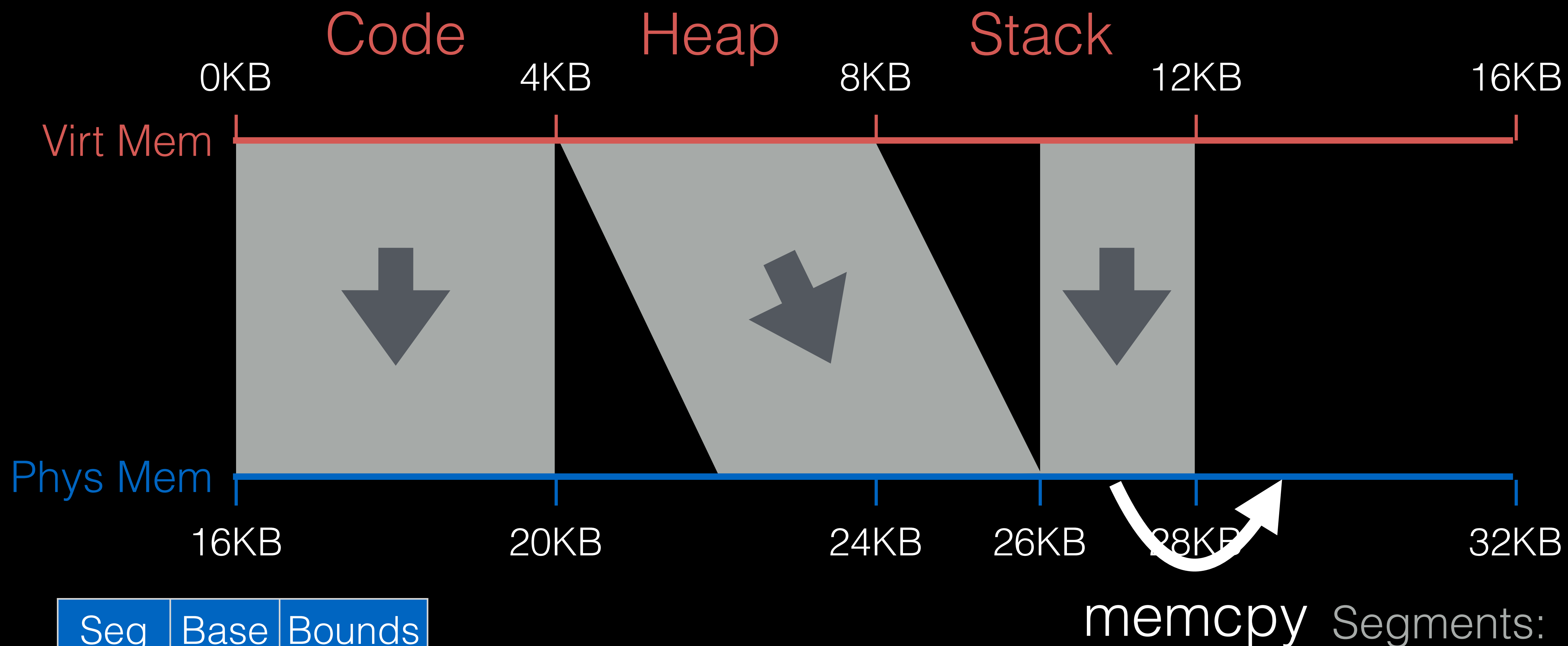
Segments:
0=>code
1=>heap
2=>stack.



Seg	Base	Bounds
0	16KB	20KB
1	22KB	26KB
2	26KB	28KB

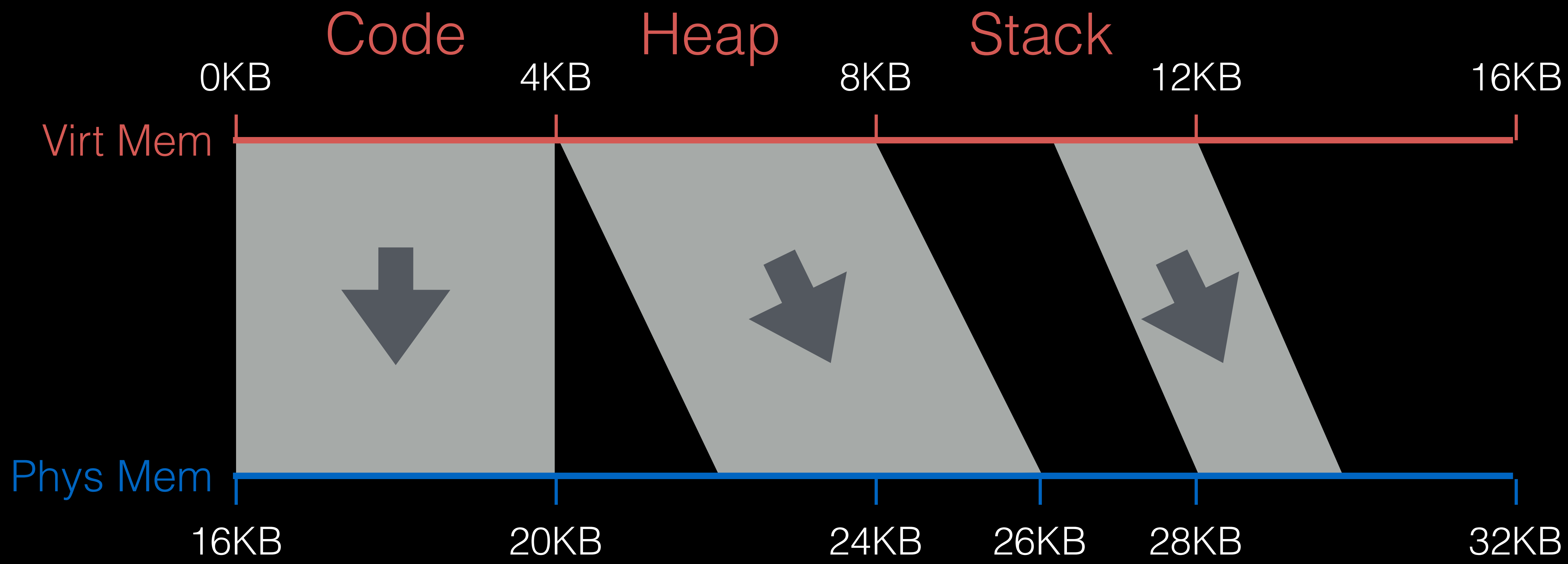
How to grow stack?

Segments:
0=>code
1=>heap
2=>stack.



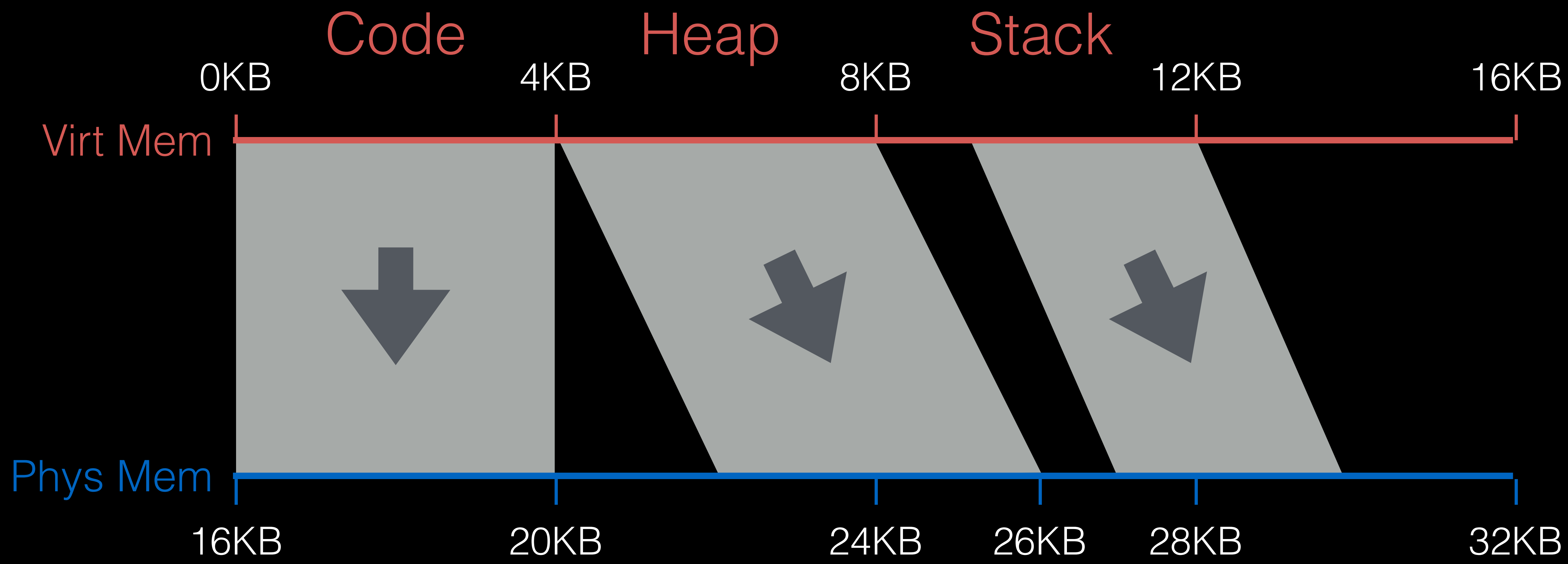
Seg	Base	Bounds
0	16KB	20KB
1	22KB	26KB
2	26KB	28KB

Segments:
0=>code
1=>heap
2=>stack.



Seg	Base	Bounds
0	16KB	20KB
1	22KB	26KB
2	28KB	30KB

Segments:
0=>code
1=>heap
2=>stack.



Seg	Base	Bounds
0	16KB	20KB
1	22KB	26KB
2	27KB	30KB

Segments:
0=>code
1=>heap
2=>stack.

Paging

Paging

Segmentation is too coarse-grained.
Either waste space *OR* memcpy often.

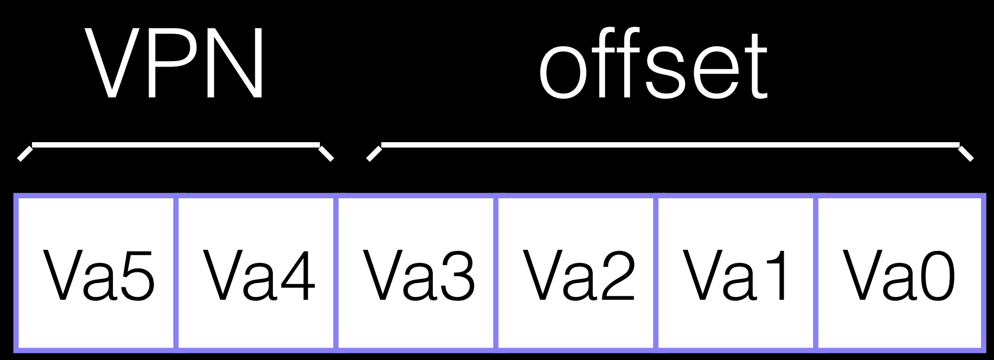
We need a fine-grained alternative!

Paging idea:
- break mem into small, fix-sized chunks (aka pages)
- each virt page is independently mapped to a phys page
- grow memory segments however we please!

Addressing

For segmentation, high bits => segment, low bits => offset

For paging, high bits => page, low bits => offset



How many low bits do we need?

Address Examples

Page Size	Low Bits (offset)
16 bytes	

Address Examples

Page Size	Low Bits (offset)
16 bytes	4

Address Examples

Page Size	Low Bits (offset)
16 bytes	4
1 KB	

Address Examples

Page Size	Low Bits (offset)
16 bytes	4
1 KB	10

Address Examples

Page Size	Low Bits (offset)
16 bytes	4
1 KB	10
1 MB	

Address Examples

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

Address Examples

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

Address Examples

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

Address Examples

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

Address Examples

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

Address Examples

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

Address Examples

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

Address Examples

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	
512 bytes	9	16	
4 KB	12	32	

Address Examples

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
4 KB	12	32	20

Address Examples

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

Address Examples

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 MB	20	32	12	
512 bytes	9	16	5	
4 KB	12	32	20	

Address Examples

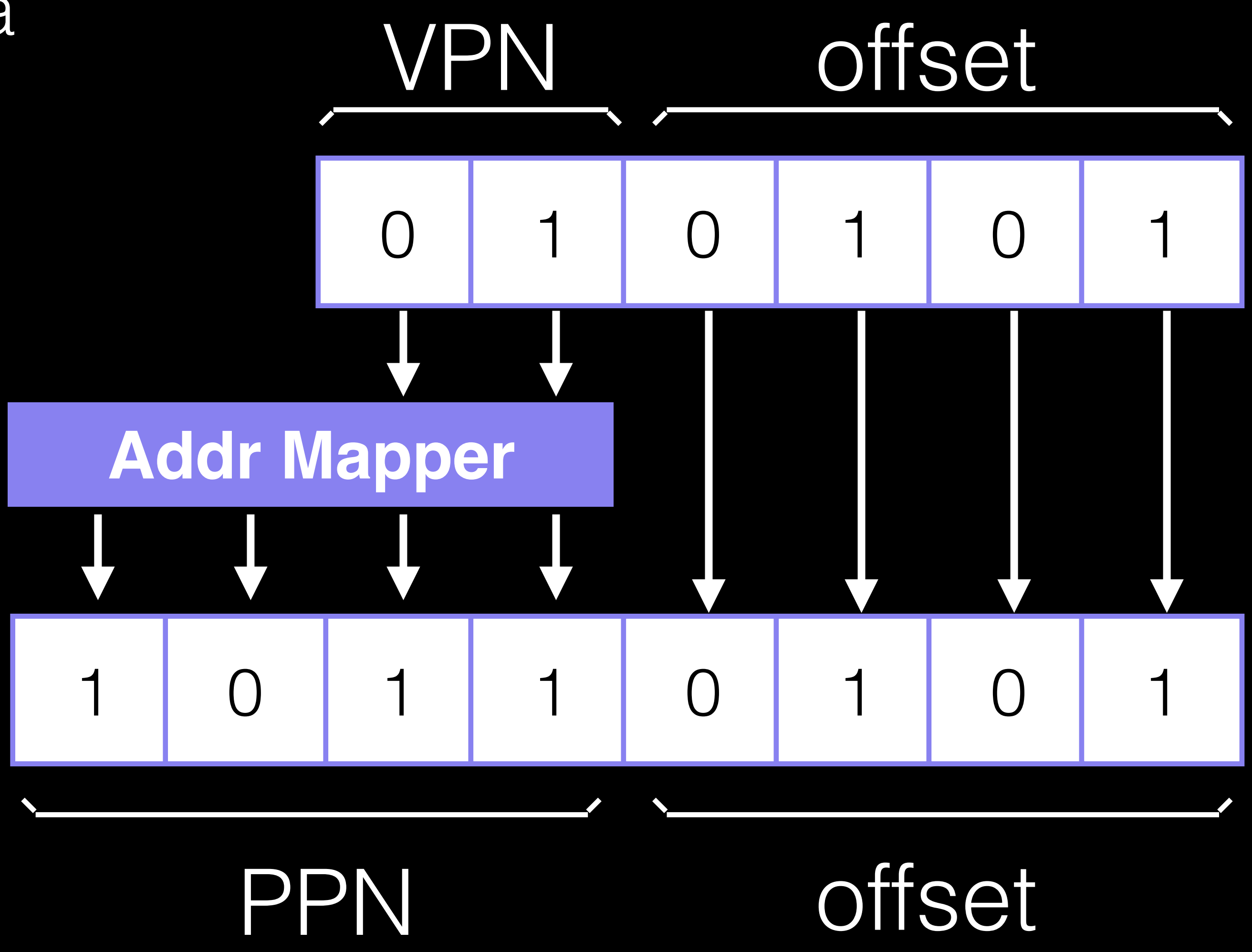
Page Size	Low Bits (offset)	Virt Addr Bits	High Bits (vpn)	Virt Pages
16 bytes	4	10	6	64
1 KB	10	20	10	1 K
1 MB	20	32	12	4 K
512 bytes	9	16	5	32
4 KB	12	32	20	1 MB

Virt => Phys Mapping

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

Now, we need a more general mapping mechanism.

What data structure is good?

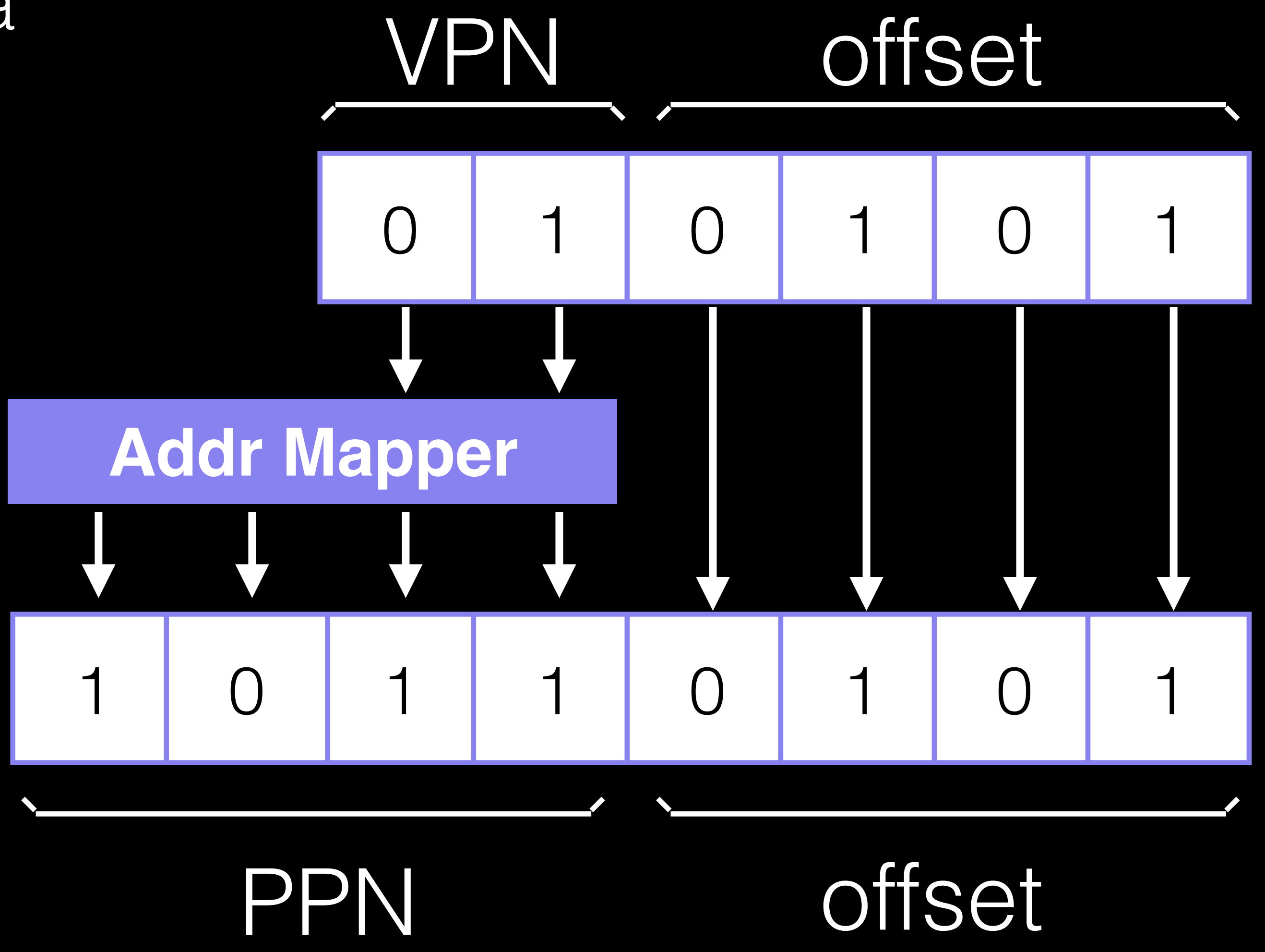


Virt => Phys Mapping

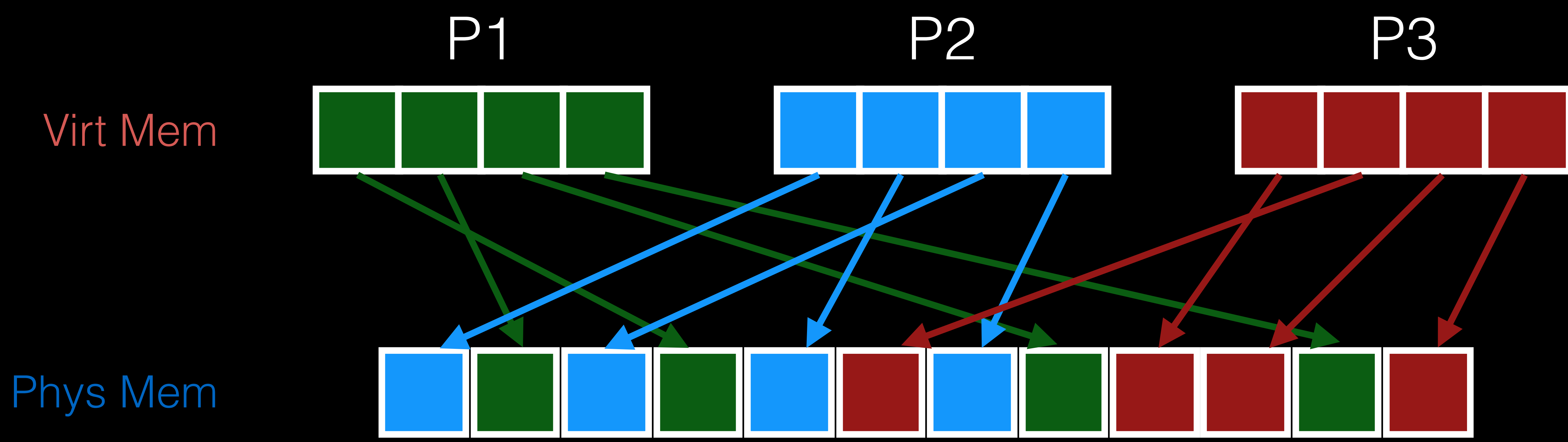
For segmentation, we used a formula (e.g., phys = virt_offset + base_reg)

Now, we need a more general mapping mechanism.

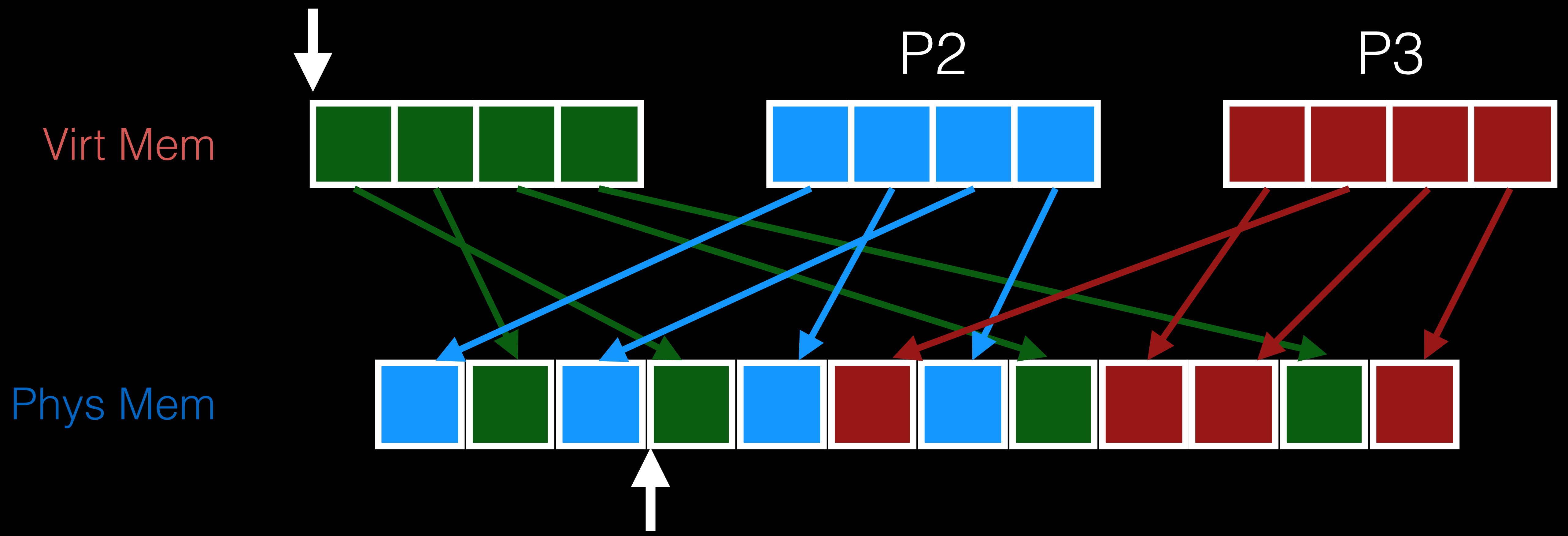
What data structure is good?
Big array, called a pagetable



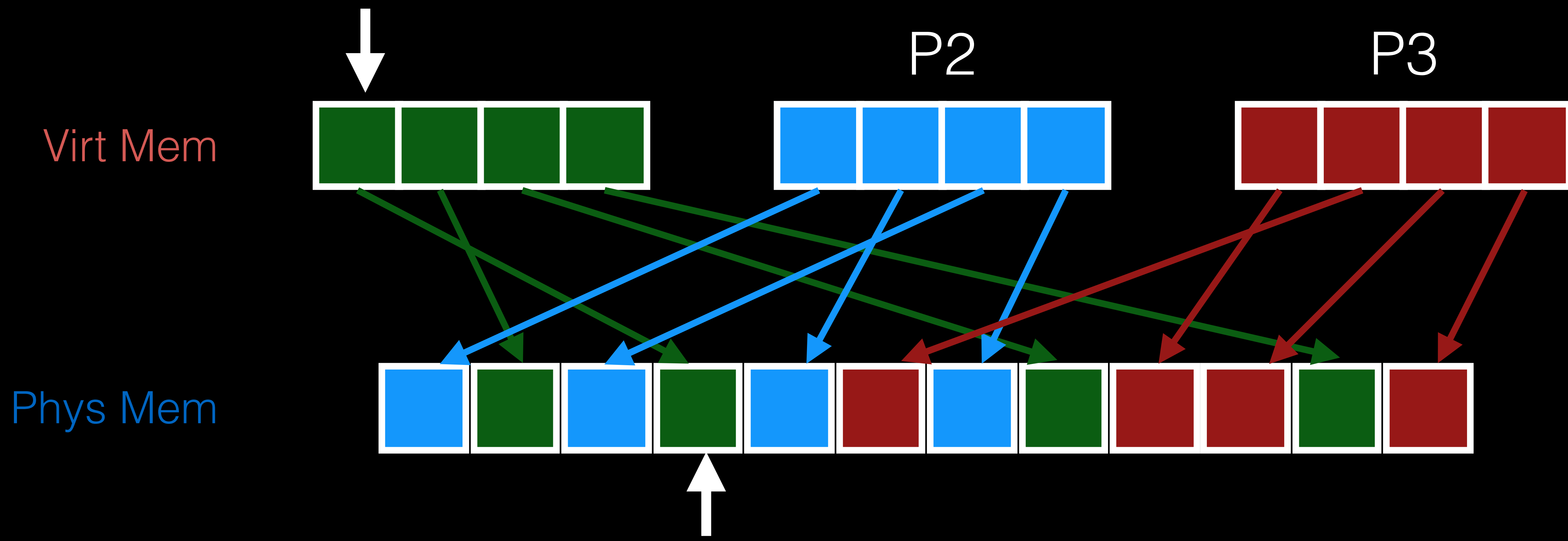
The Mapping



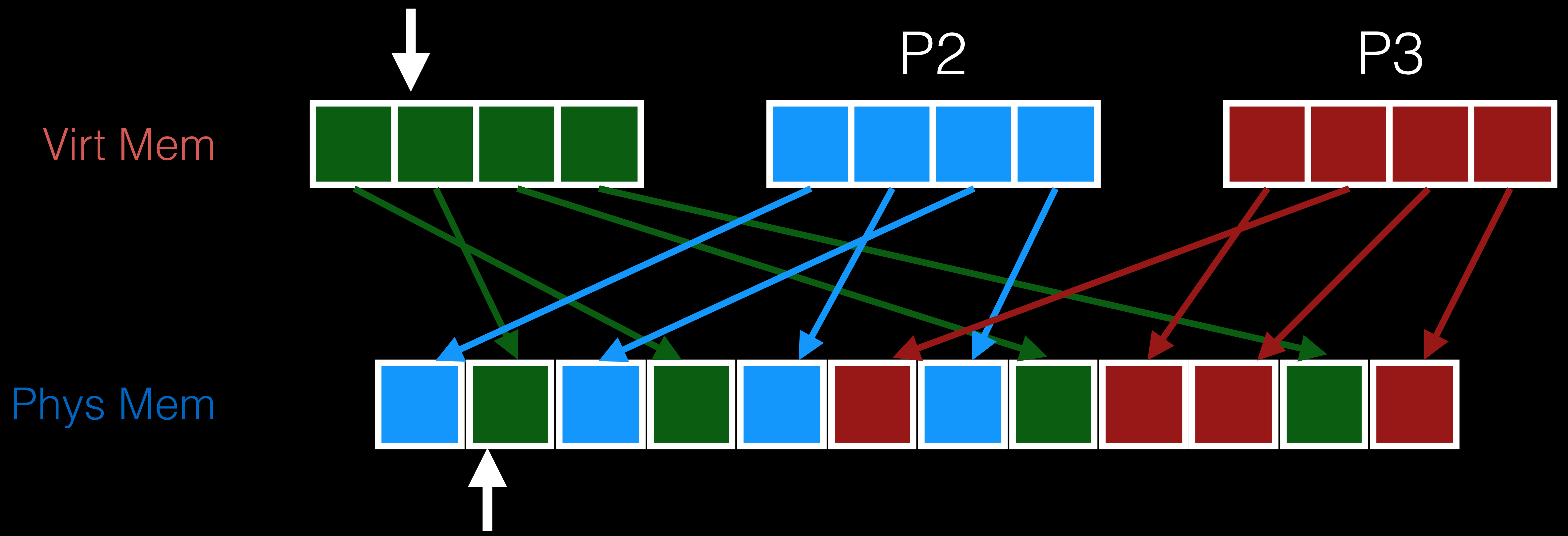
The Mapping



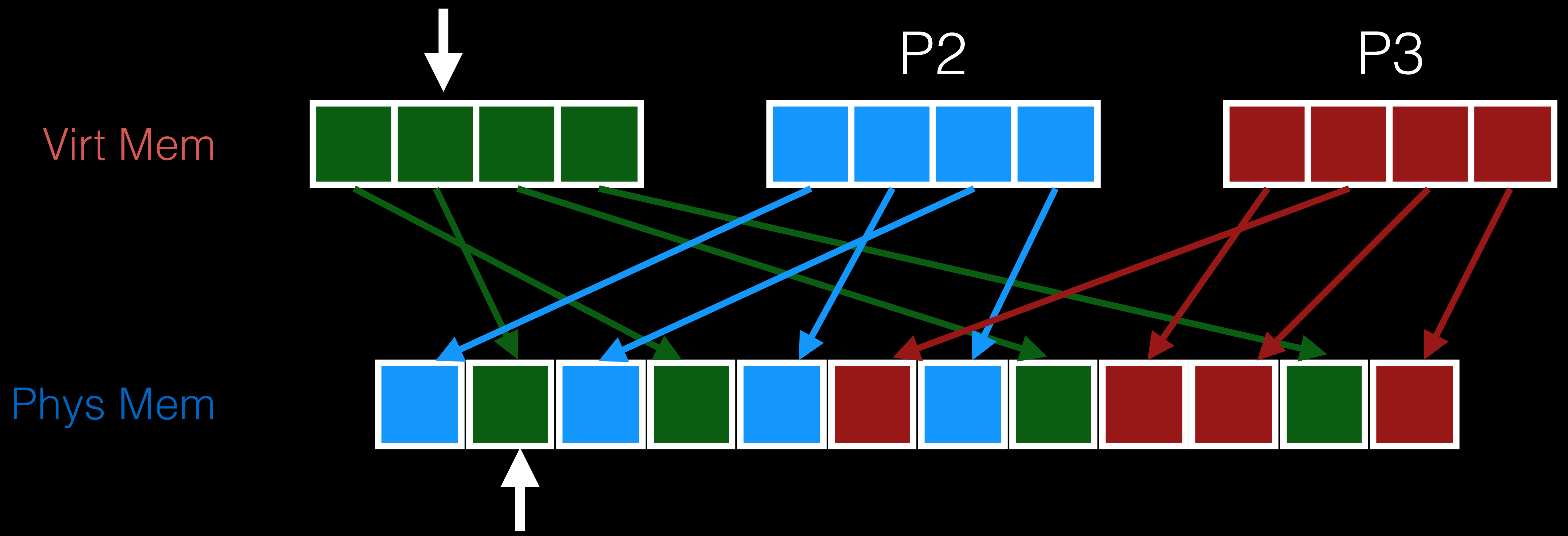
The Mapping

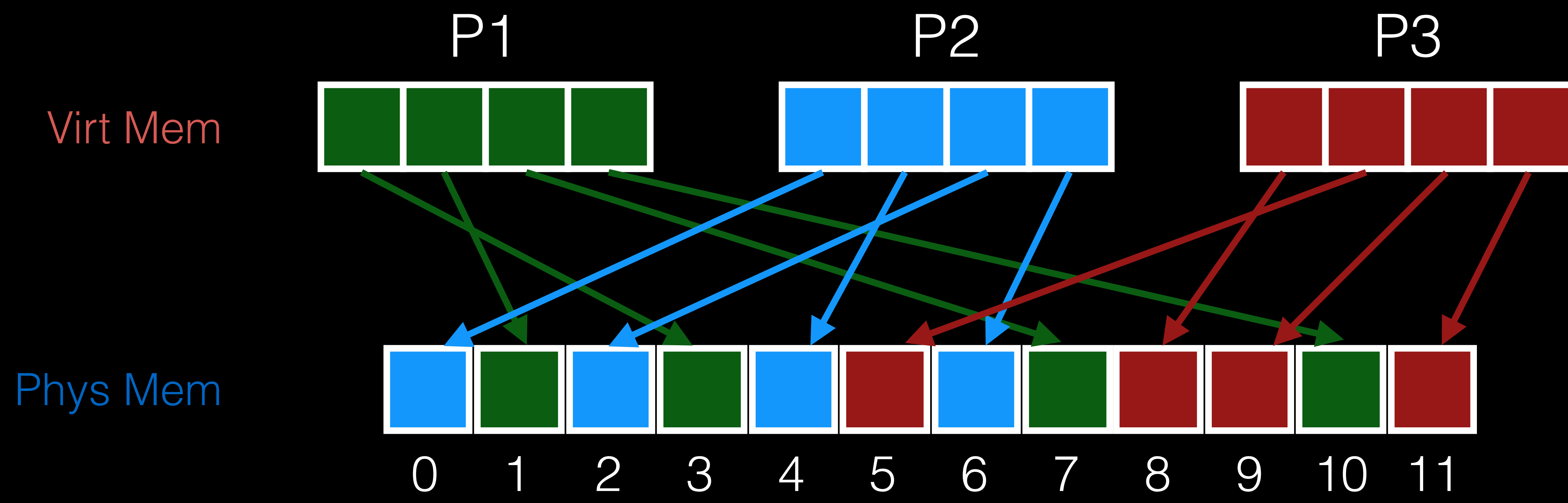


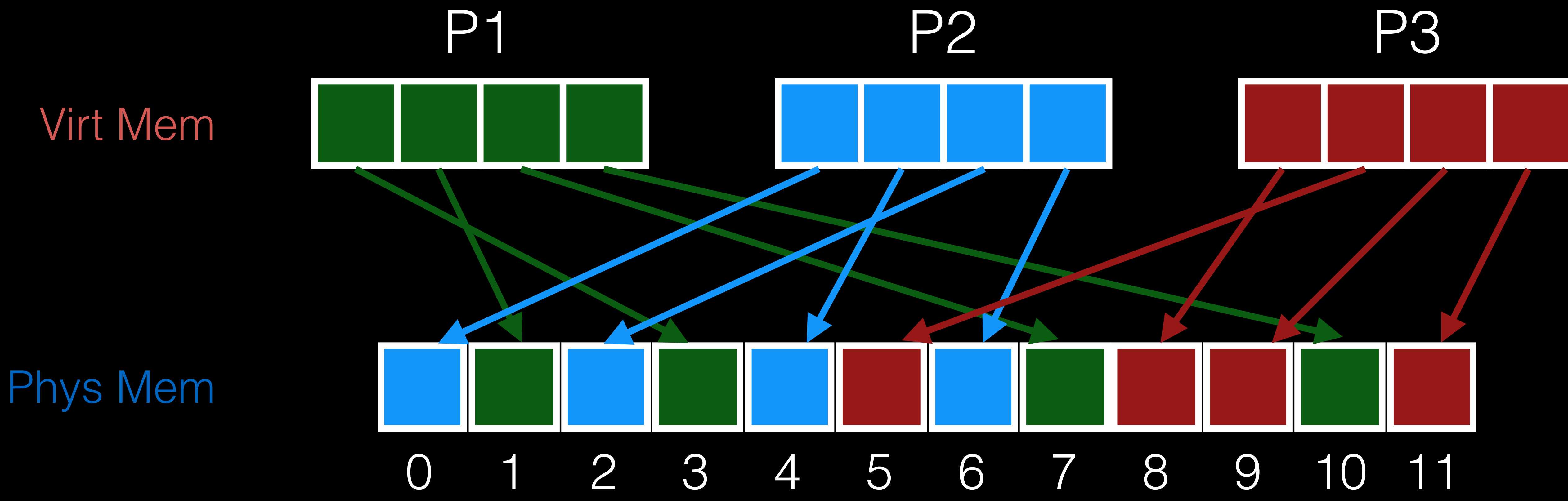
The Mapping



The Mapping

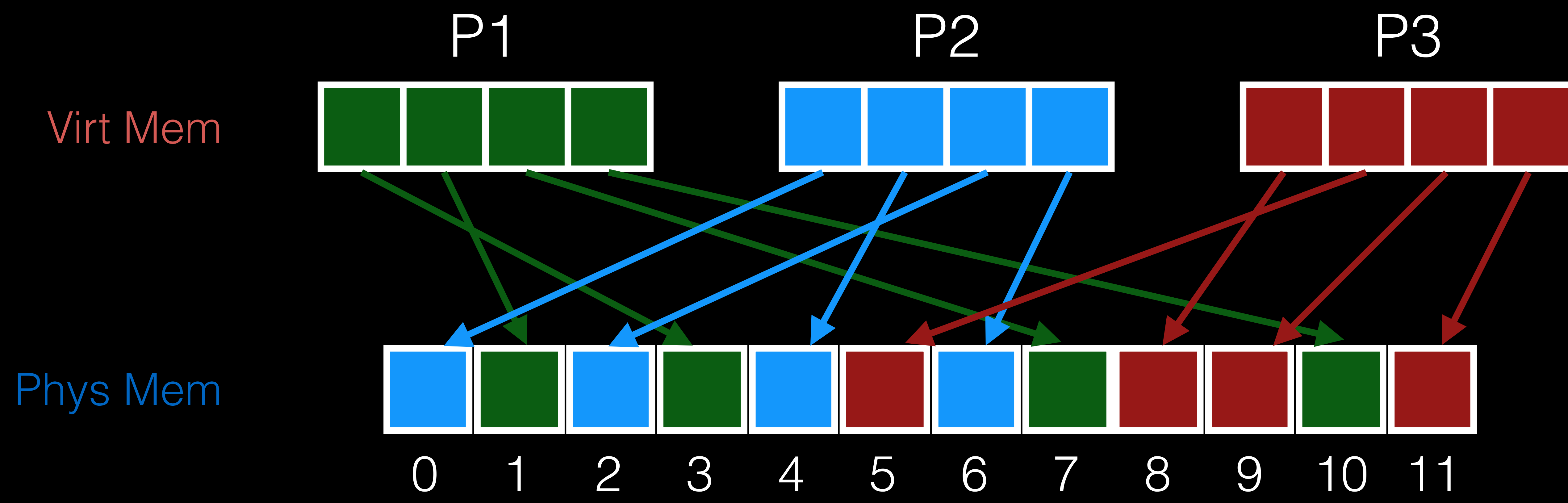



P1
P2
P3
Virt Mem
Phys Mem
0
1
2
3
4
5
6
7
8
9
10
11



Page Tables:

P1	P2	P3
3	0	?
1	4	?
7	2	?
10	6	?



Page Tables:

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

Where Are Pagetable's Stored?

How big is a typical page table?

Where Are Pagetable's Stored?

How big is a typical page table?

- assume 32-bit address space
- assume 4 KB pages
- assume 4 byte entries (or this could be less)
- 2 ^ (32 - log(4KB)) * 4 = **4 MB**

Where Are Pagetable's Stored?

How big is a typical page table?
- assume 32-bit address space
- assume 4 KB pages
- assume 4 byte entries (or this could be less)
- 2 ^ (32 - log(4KB)) * 4 = **4 MB**

Store in memory.
CPU finds it via register (e.g., CR3 on x86)

Final PT example

Memory Accesses:

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

PT

2
0
80
99

Assume PT is at 0x5000
Assume PTE's are 4 bytes
Assume 4KB pages

Final PT example

Memory Accesses:

PT, load from addr 0x5000

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

PT

2
0
80
99

Assume PT is at 0x5000
Assume PTE's are 4 bytes
Assume 4KB pages

Final PT example

Memory Accesses:

PT, load from addr 0x5000

Fetch instruction at addr 0x2010

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

PT

2
0
80
99

Assume PT is at 0x5000

Assume PTE's are 4 bytes

Assume 4KB pages

Final PT example

Memory Accesses:

PT, load from addr 0x5000
Fetch instruction at addr 0x2010
PT, load from addr 0x5004

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

PT

2
0
80
99

Assume PT is at 0x5000
Assume PTE's are 4 bytes
Assume 4KB pages

Final PT example

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

Memory Accesses:

PT, load from addr 0x5000
Fetch instruction at addr 0x2010
PT, load from addr 0x5004
Exec, load from addr 0x5900

PT

2
0
80
99

Assume PT is at 0x5000
Assume PTE's are 4 bytes
Assume 4KB pages

Final PT example

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

Memory Accesses:

PT, load from addr 0x5000
Fetch instruction at addr 0x2010
PT, load from addr 0x5004
Exec, load from addr 0x5900

PT

2
0
80
99

Assume PT is at 0x5000
Assume PTE's are 4 bytes
Assume 4KB pages

Final PT example

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

Memory Accesses:

PT, load from addr 0x5000
Fetch instruction at addr 0x2010
PT, load from addr 0x5004
Exec, load from addr 0x5900
...

PT

2
0
80
99

Assume PT is at 0x5000
Assume PTE's are 4 bytes
Assume 4KB pages

Final PT example

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

Memory Accesses:

PT, load from addr 0x5000
Fetch instruction at addr 0x2010
PT, load from addr 0x5004
Exec, load from addr 0x0100
…

PT

2
0
80
99

Assume PT is at 0x5000
Assume PTE's are 4 bytes
Assume 4KB pages

Our pagetable is slow!!!

Other PT info

What other data should go in pagetable entries besides translation?

Other PT info

What other data should go in pagetable entries besides translation?

- valid bit
- protection bits
- present bit
- reference bit
- dirty bit

Summary

Pros?

Cons?

Summary

Pros?

- very flexible
- no external fragmentation
- no need to shuffle around data

Cons?

- expensive translation
- huge space overheads

Announcements

P1 due tonight!

Tests: sorry.

P2 released.

Office hours, 1-2pm today (room 7373).

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

Free-Space Management

Many systems need to manage/allocate space

1. physical space for process segments

2. virtual space for malloc calls ← today

3. disk blocks for files

Allocation API

```
void *malloc(size_t size);

void free(void *ptr);

void *realloc(void *ptr, size_t size);
```

Allocation API

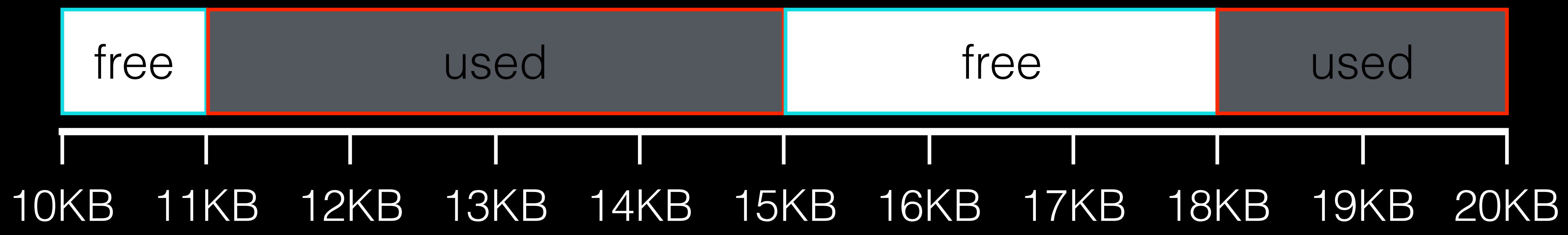
```
void *malloc(size_t size);

void free(void *ptr);

void *realloc(void *ptr, size_t size);
```

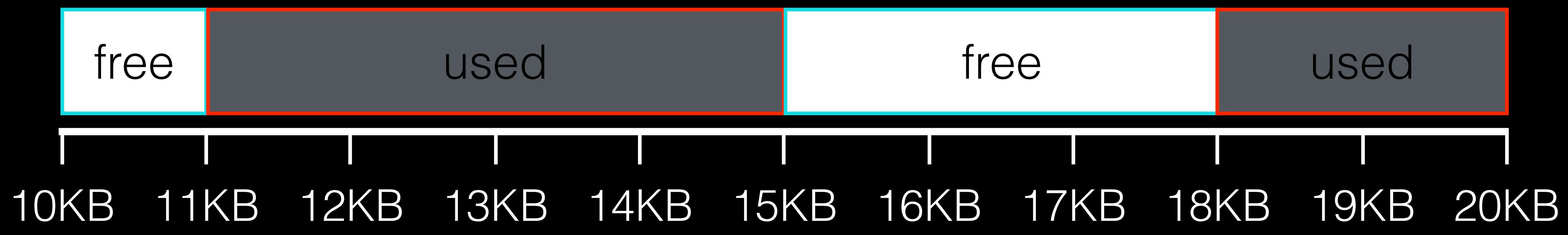
today

Malloc/Free Basics



Malloc/Free Basics

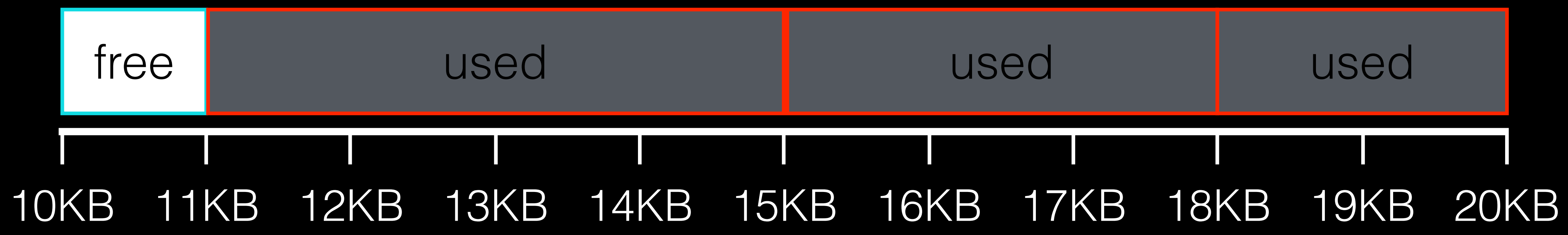
`malloc(3072)`



Malloc/Free Basics

`malloc(3072) = 15KB`

Malloc/Free Basics

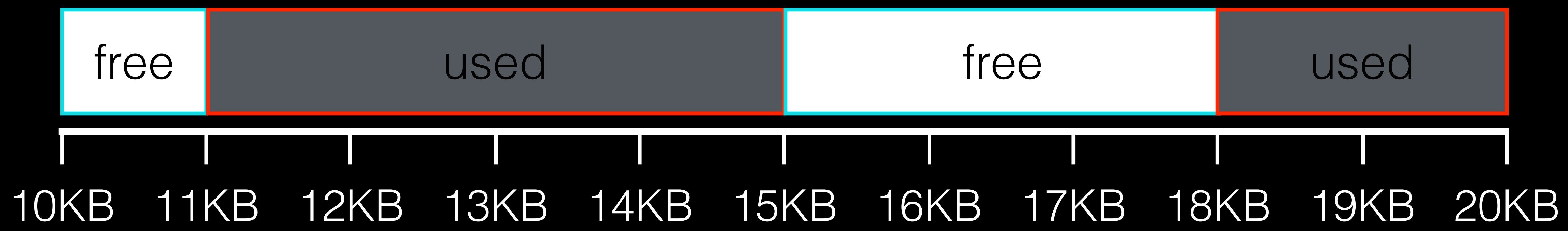


Malloc/Free Basics

`free(15KB)`

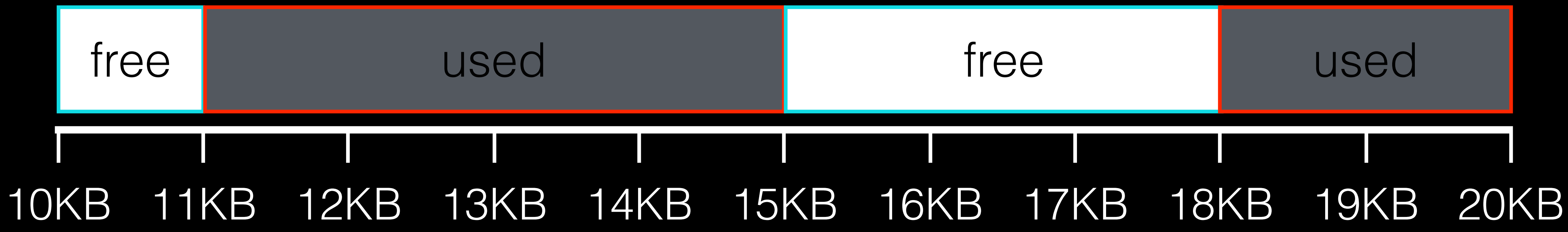
Malloc/Free Basics

`free(15KB)`



Malloc/Free Basics

`free(15KB)`



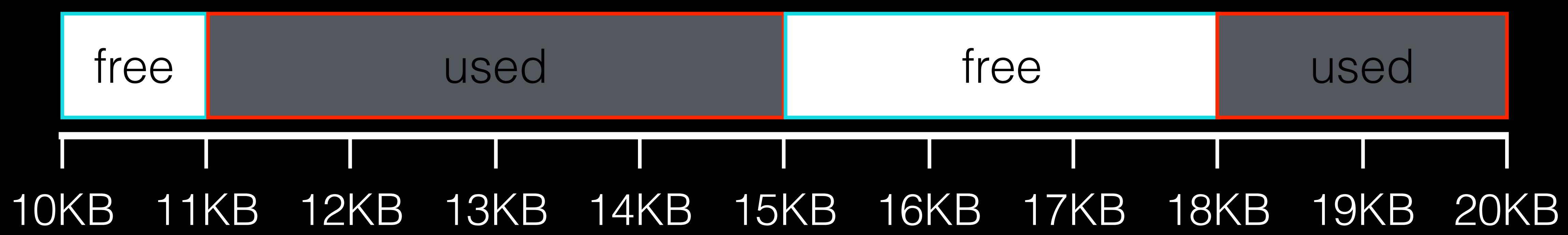
How do we know the size to free?

Odd Object Sizes

An object may not fit free space exactly:
    split memory

Free areas may be adjacent:
    coalesce memory

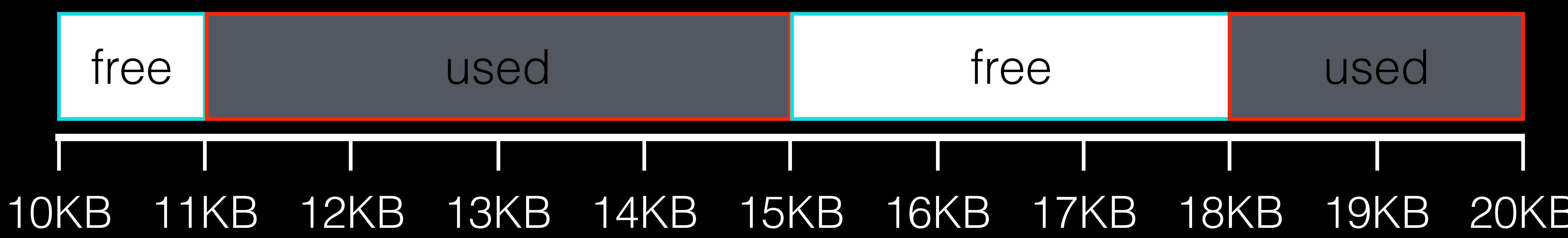
Splitting



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

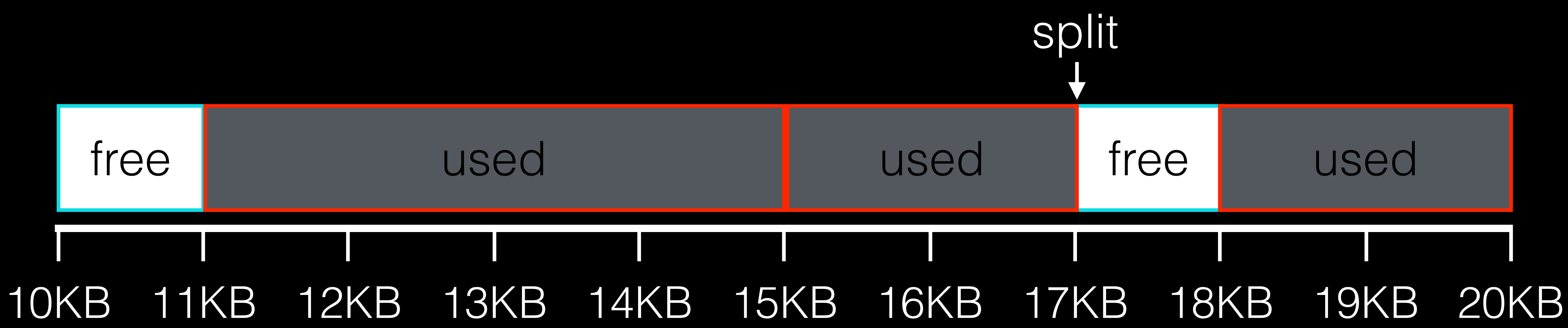
Splitting

`malloc(2048)`



Splitting

```
malloc(2048) = 15KB
```



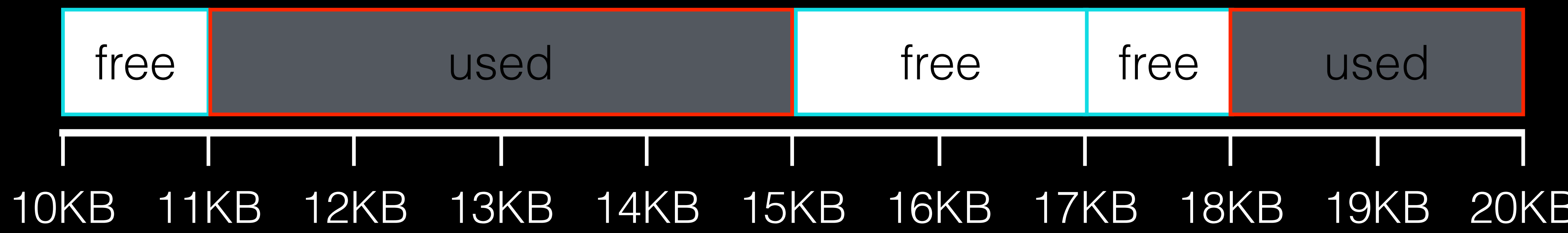

free
used
used
free
used
10KB
11KB
12KB
13KB
14KB
15KB
16KB
17KB
18KB
19KB
20KB

Coalescing

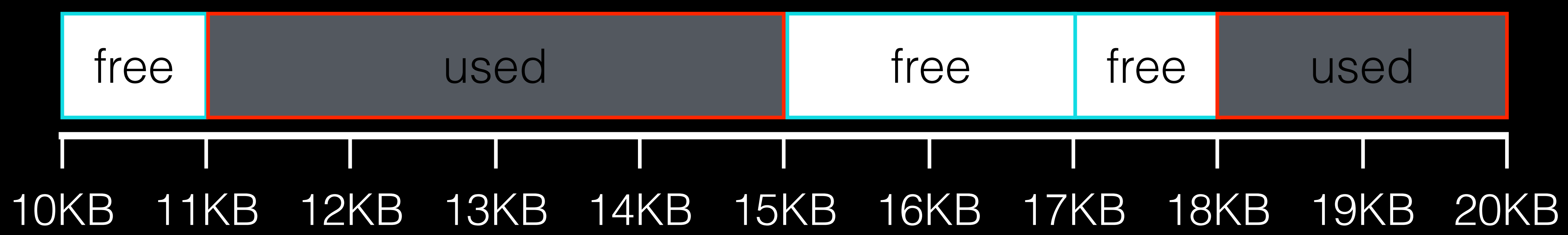
`free(15KB)`

Coalescing

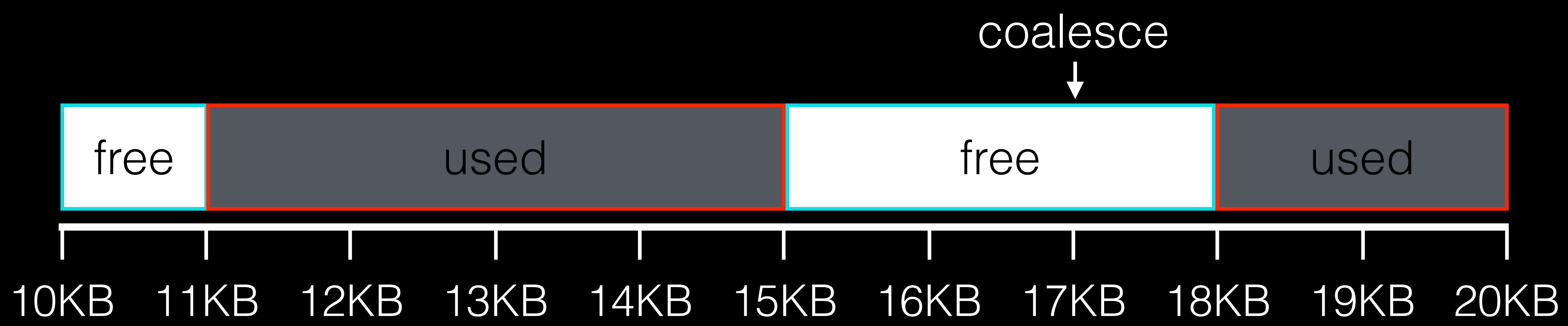
`free(15KB)`



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()

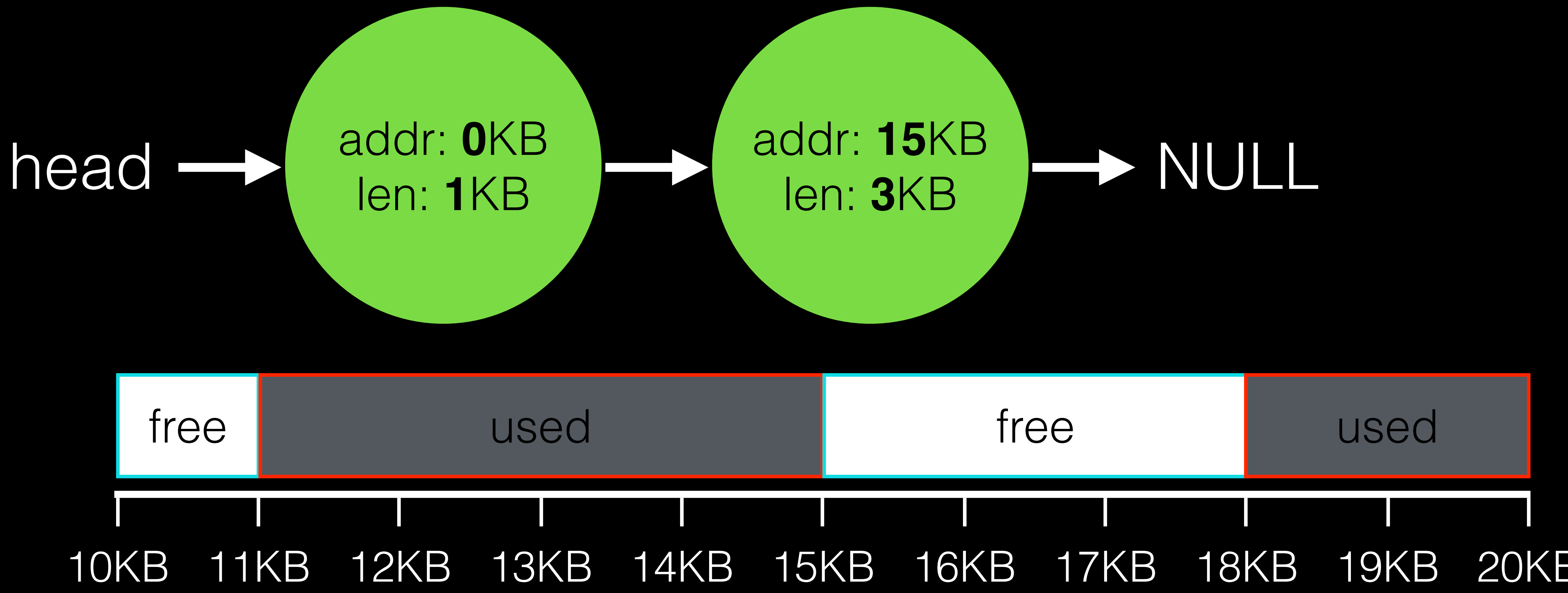
Need to know size of used spaces
- for free()

Bookkeeping

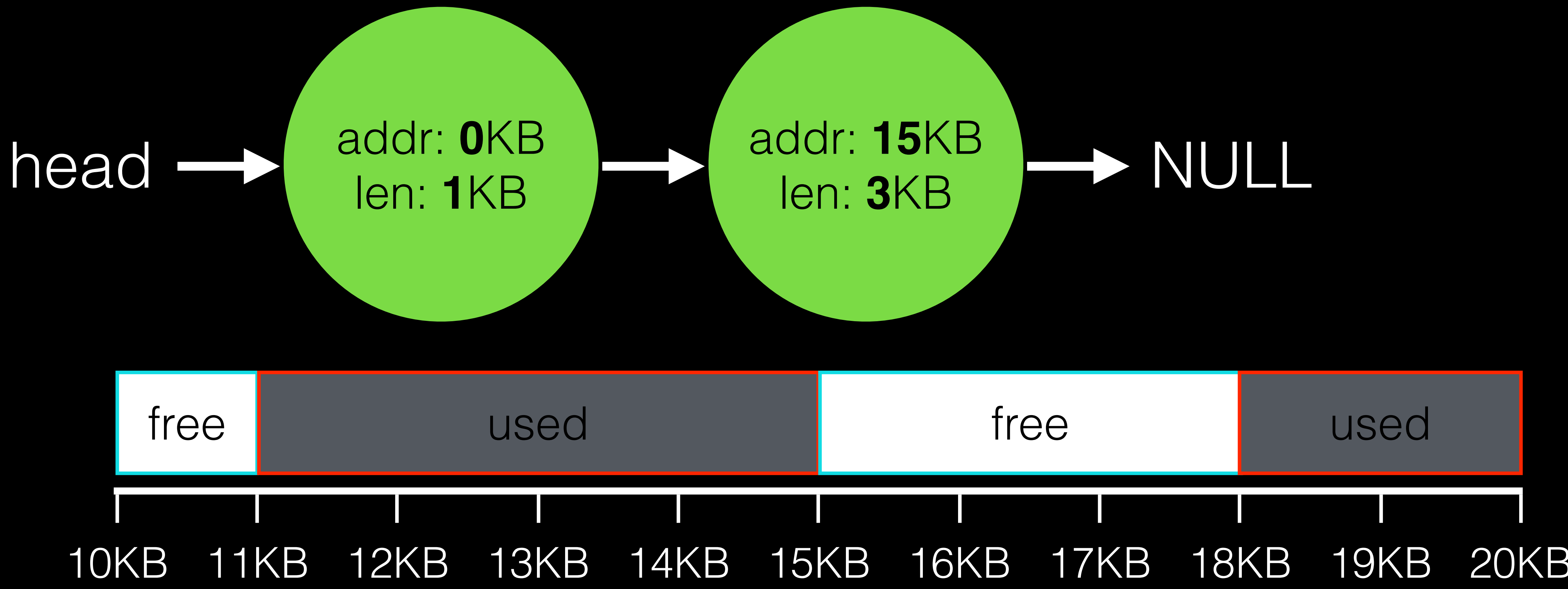
Need to know size+location of free spaces
- for malloc()

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

Free List: malloc

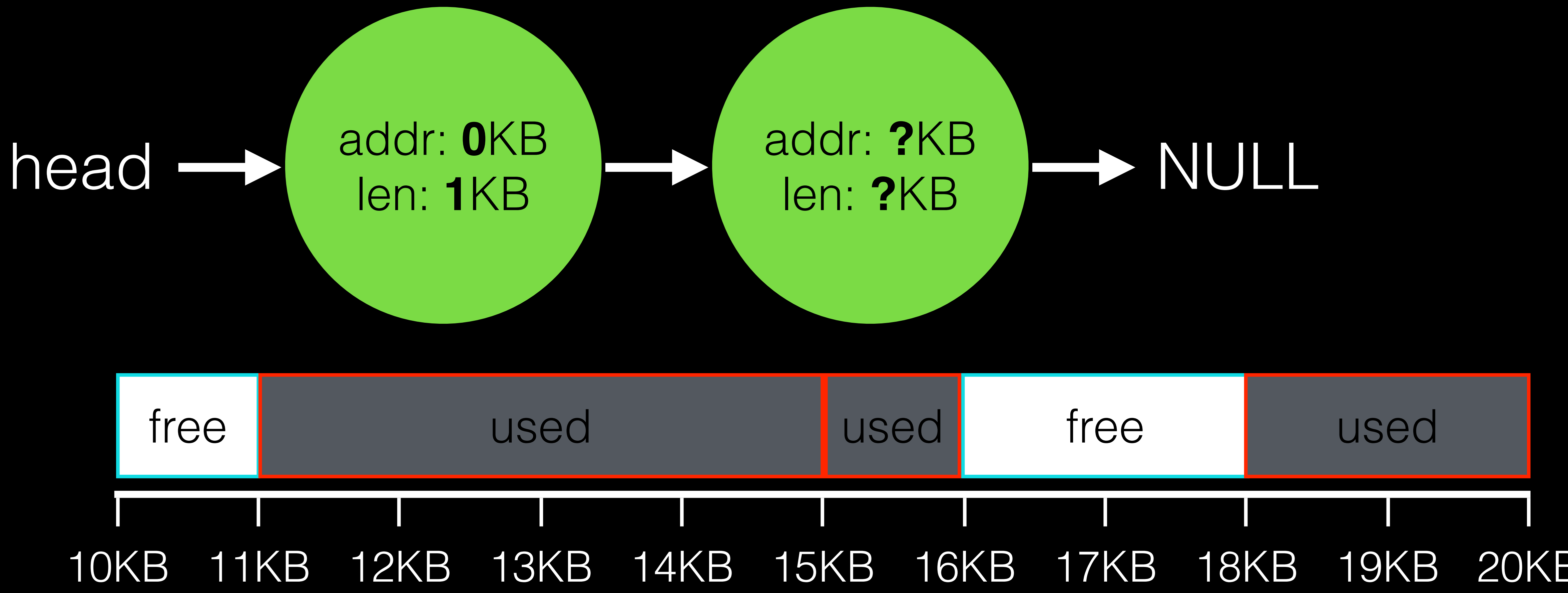


Free List: malloc



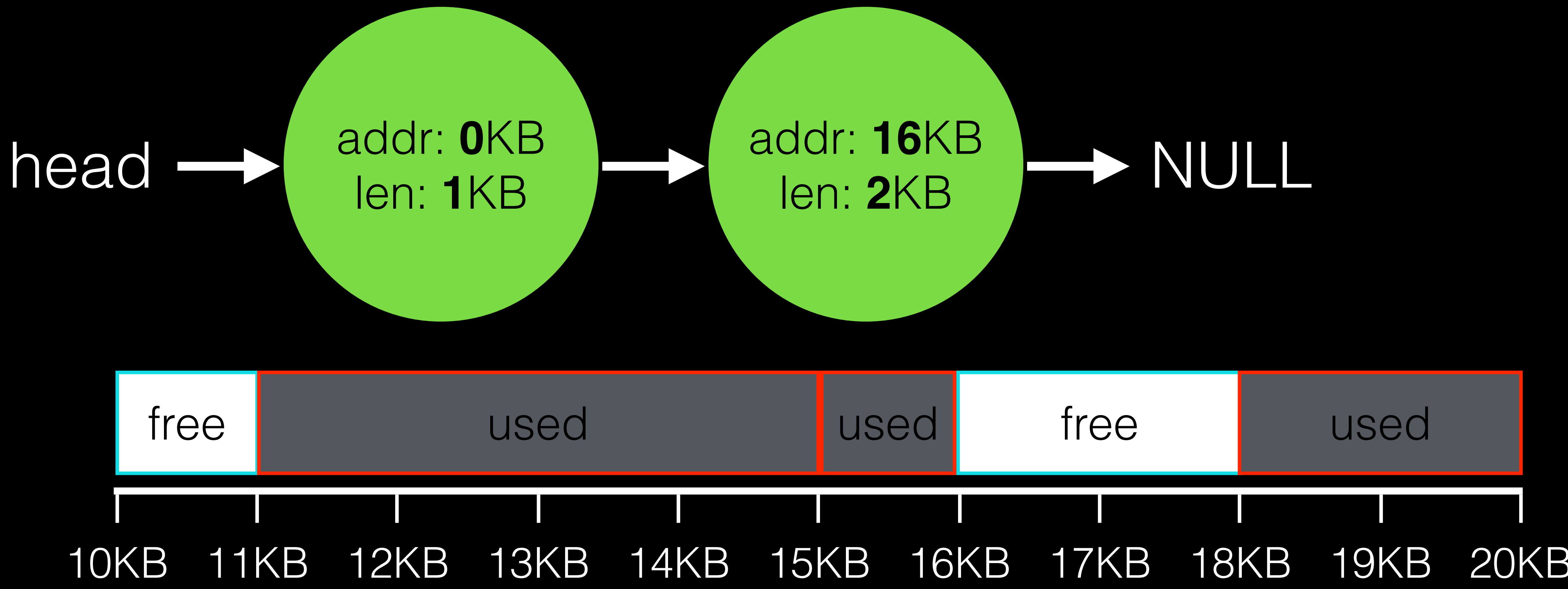
`malloc(1KB)`

Free List: malloc

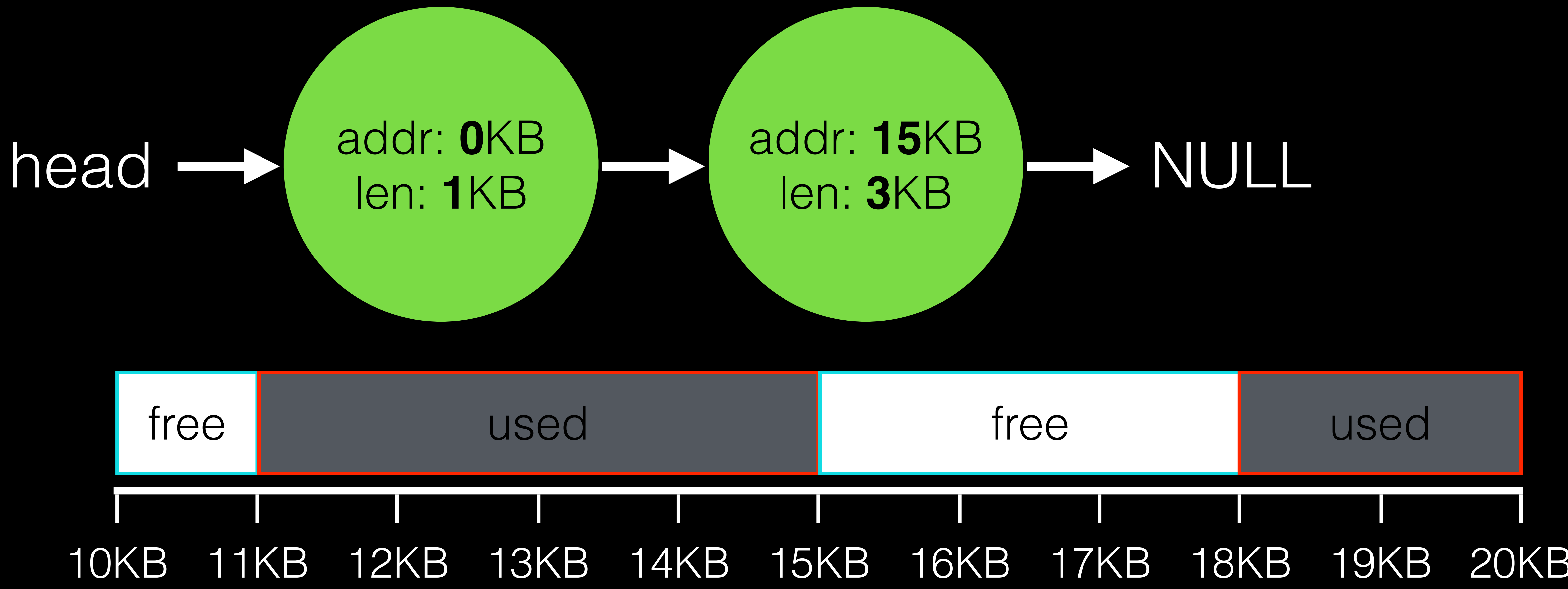



```
malloc(1KB) = 15KB
```

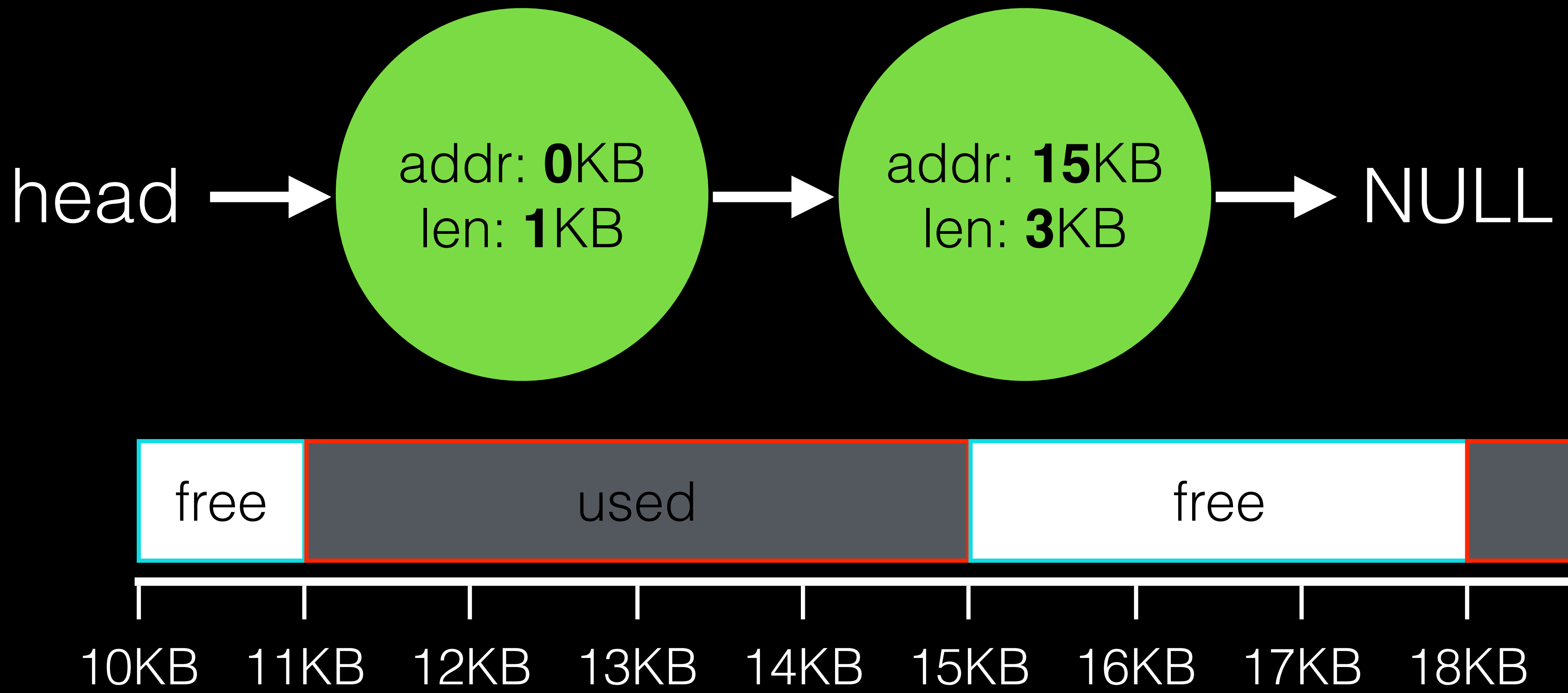
Free List: malloc



Start Over

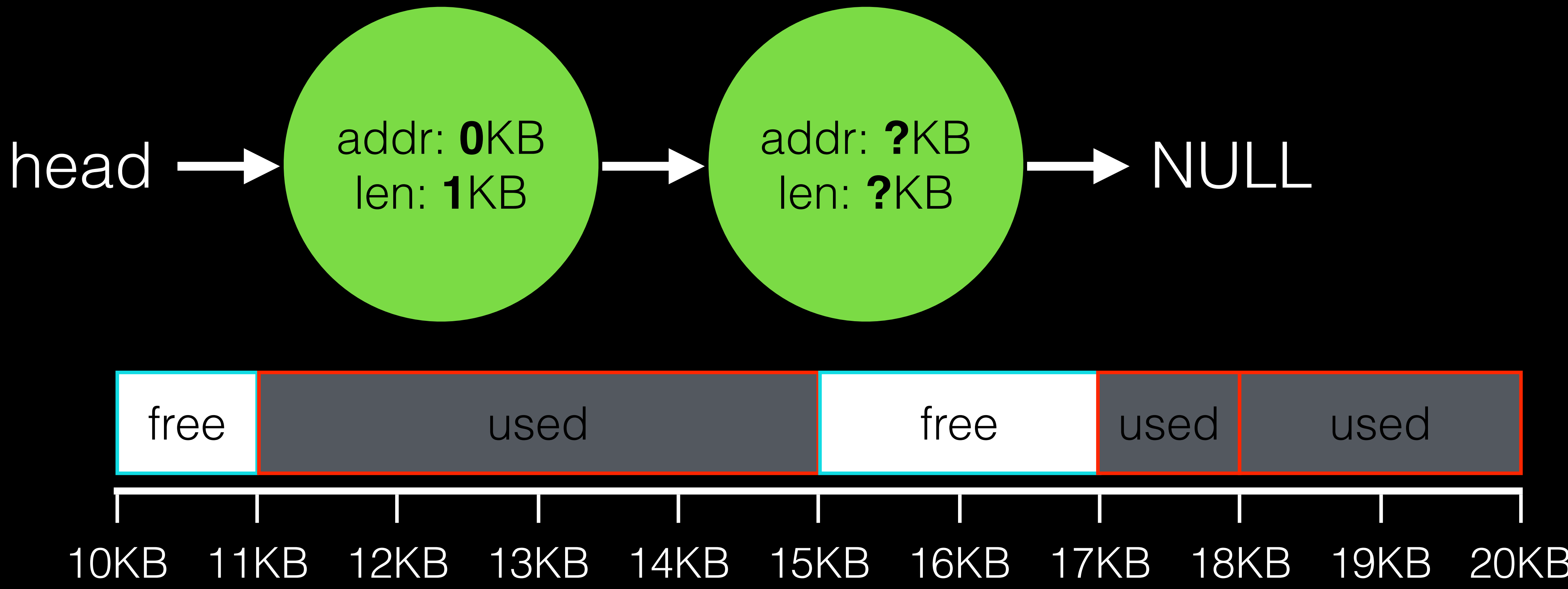

head
addr: **0**KB
len: **1**KB
addr: **15**KB
len: **3**KB
NULL
free
used
free
used
10KB
11KB
12KB
13KB
14KB
15KB
16KB
17KB
18KB
19KB
20KB

Start Over



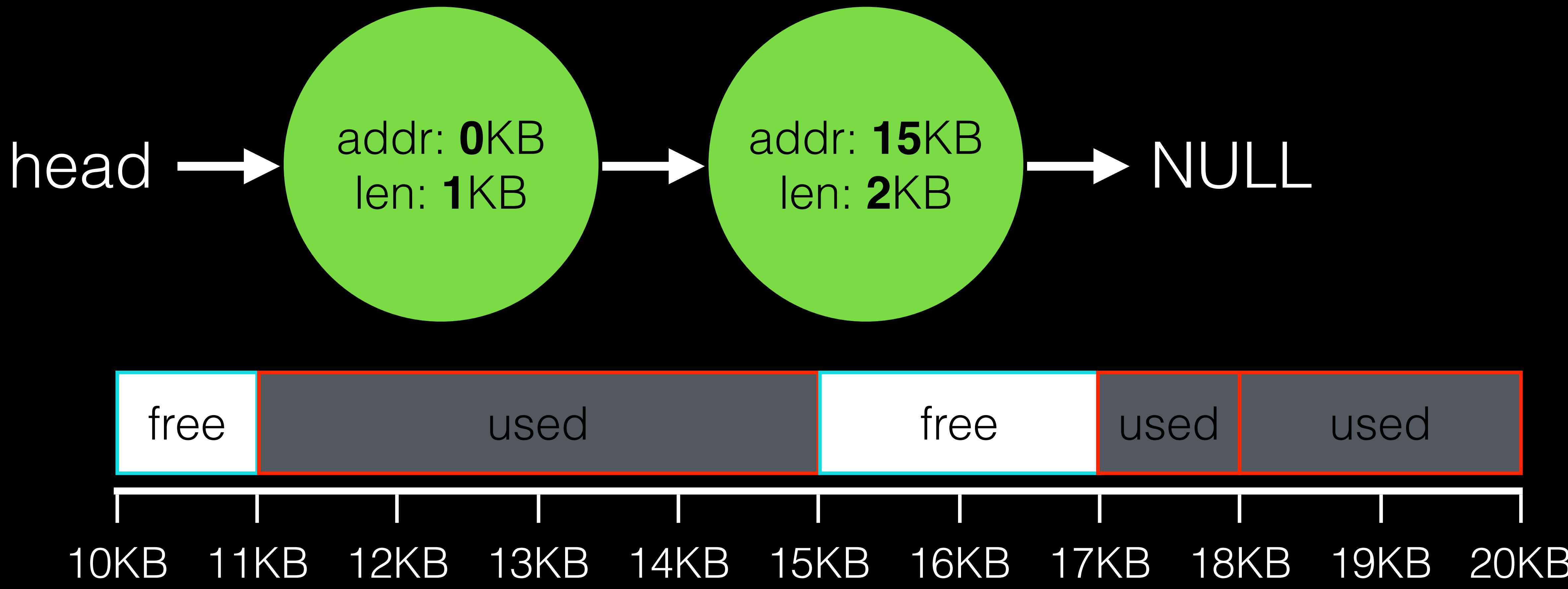
`malloc(1KB)`

Free List: malloc

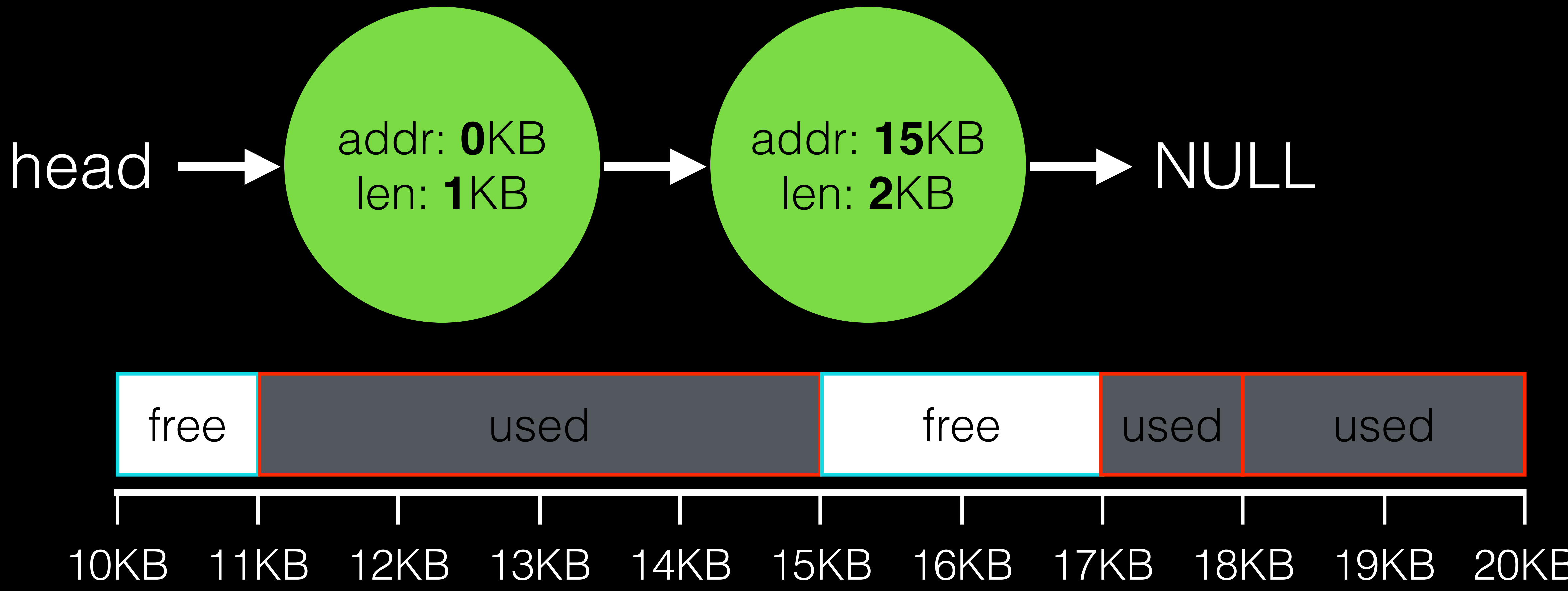


`malloc(1KB) = 17KB`

Free List: malloc

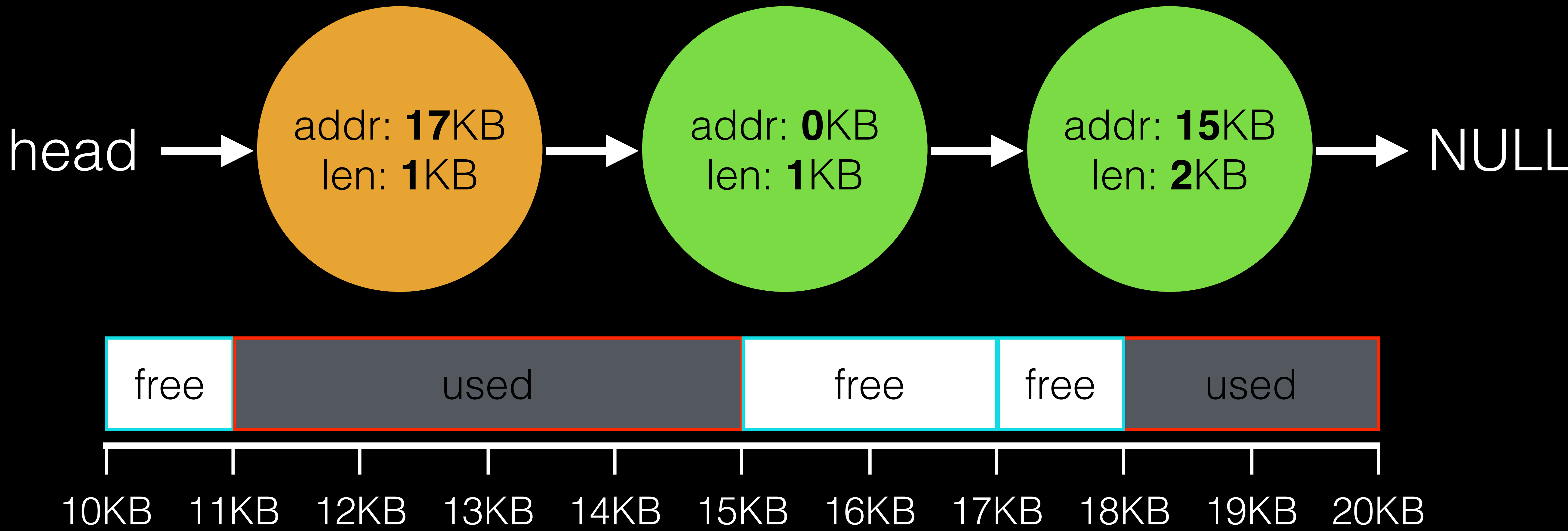


Free List: free



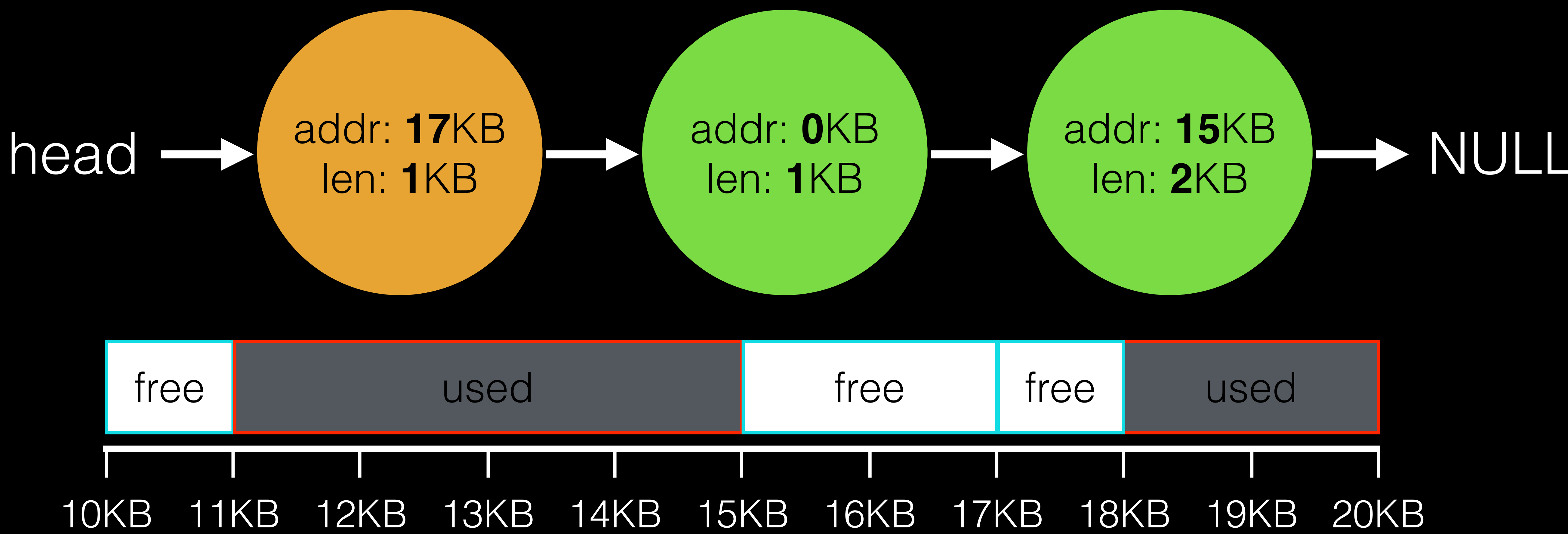
`free(17KB)`

Free List: free

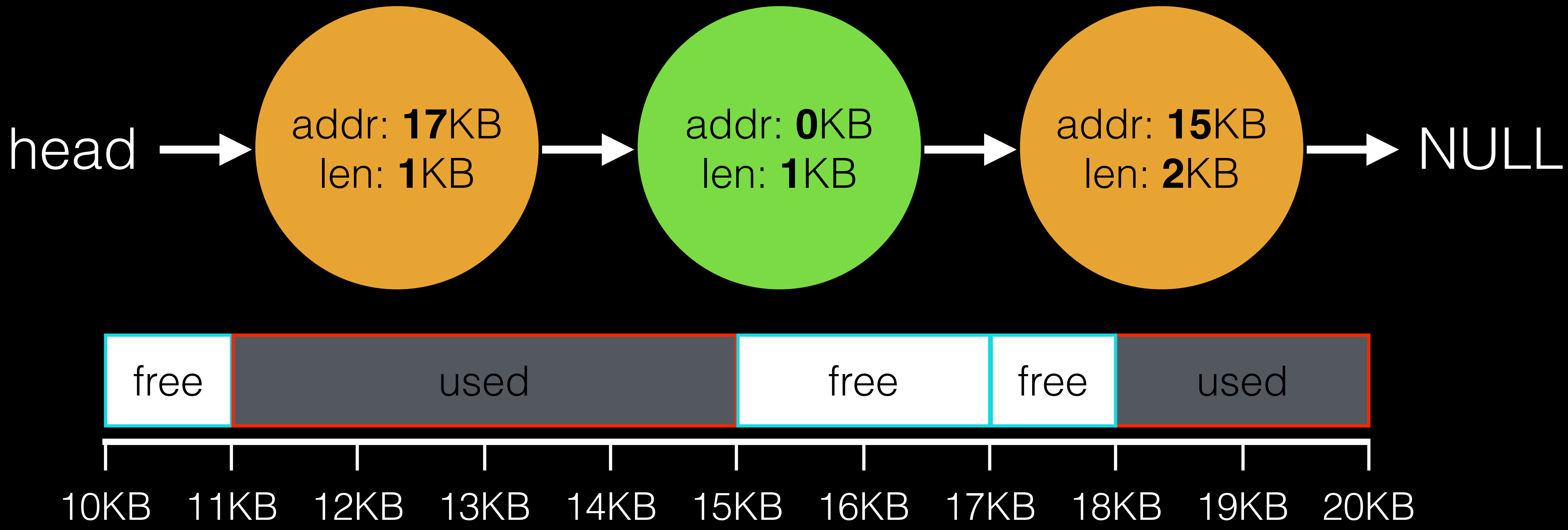


`free(17KB)`

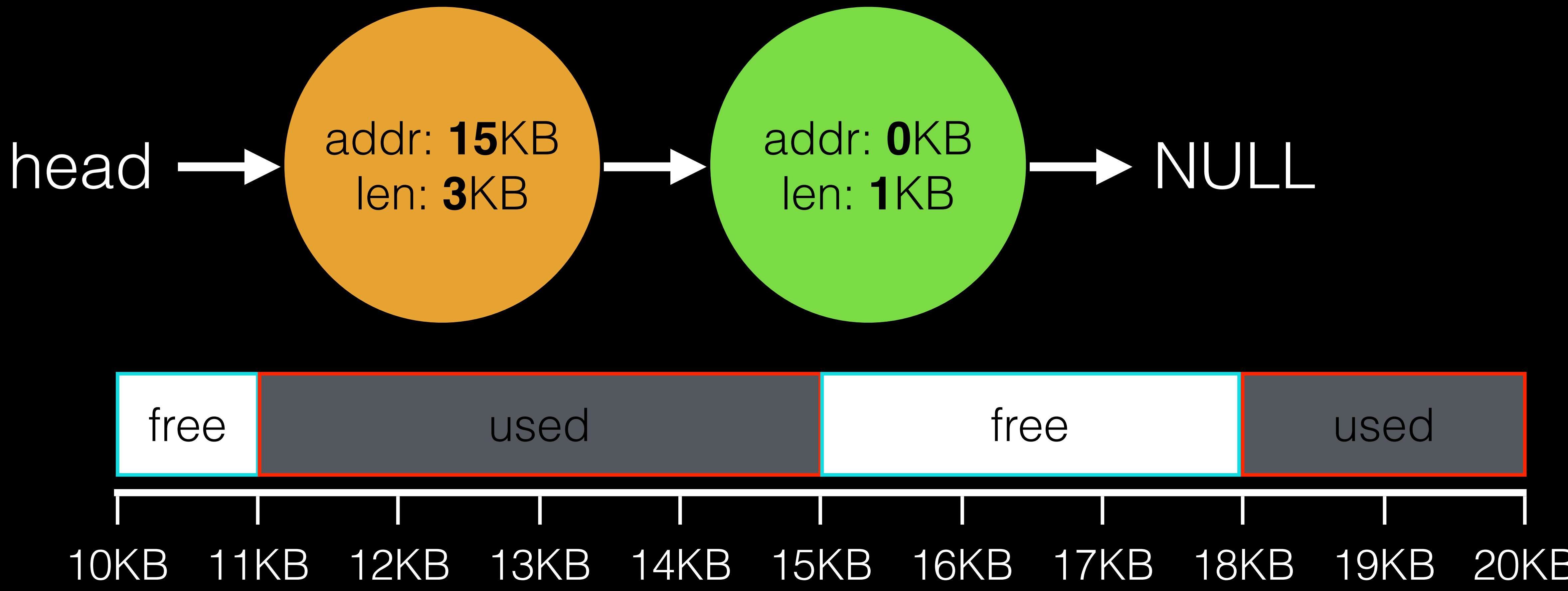
Free List: free



Free List: coalesce



Free List: coalesce



When coalescing is even trickier

Do we ever have to coalesce multiple areas?

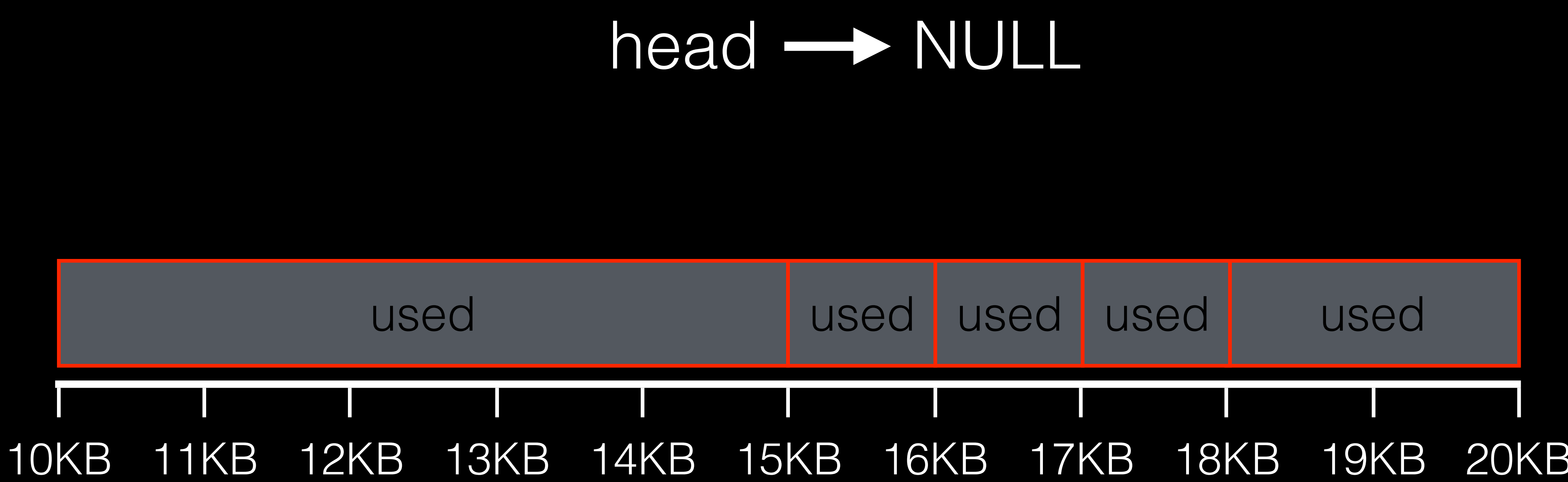
When coalescing is even trickier

Do we ever have to coalesce multiple areas?

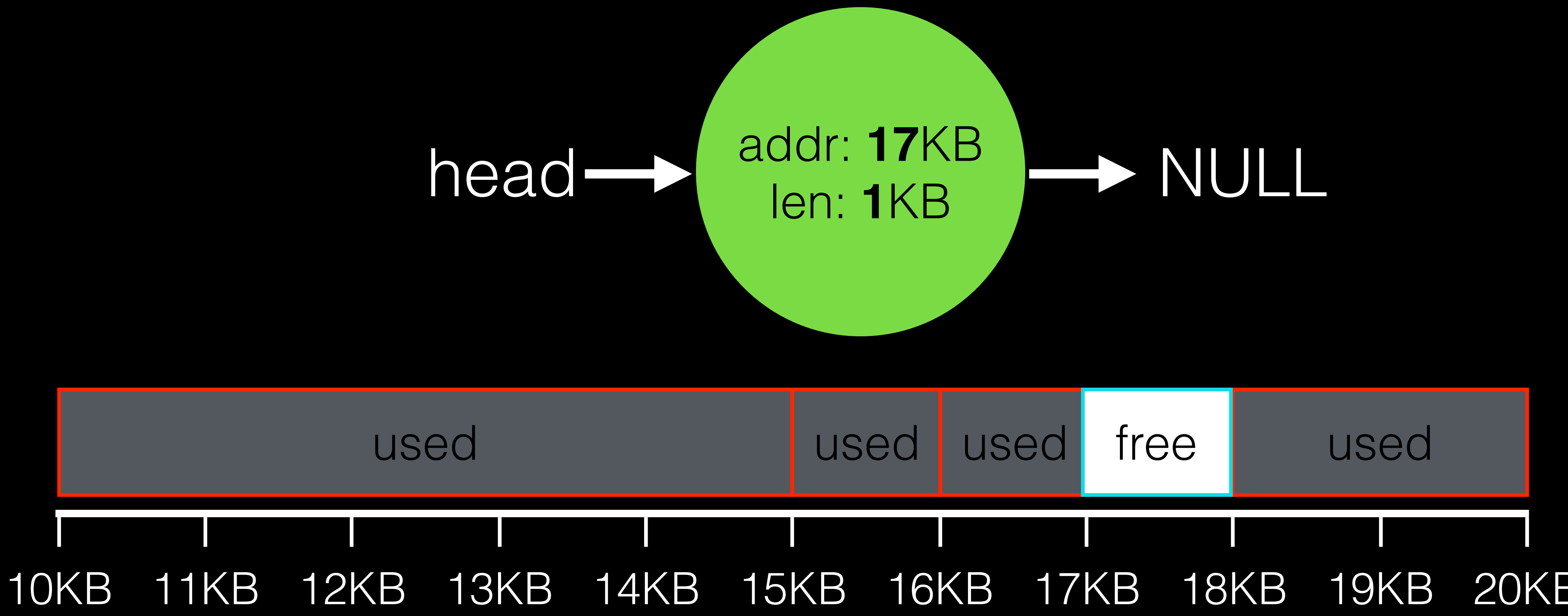
free order may be arbitrary:

```
free(17KB);
free(15KB);
free(16KB)
```

Double Coalesce

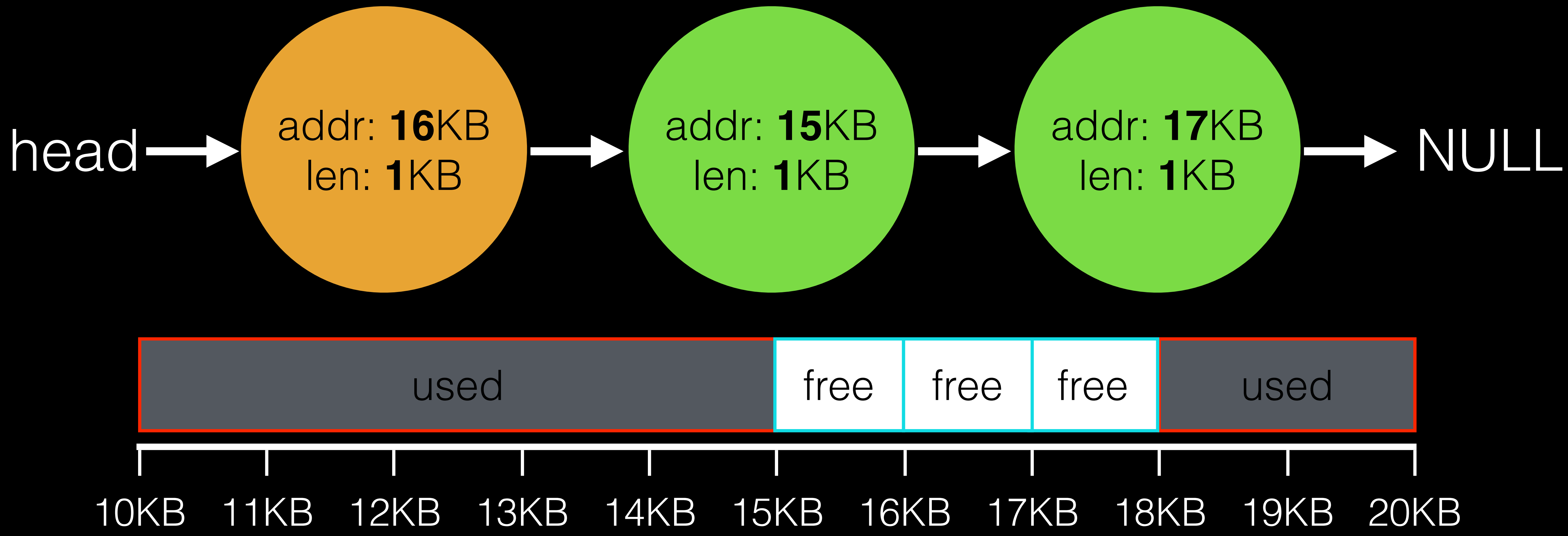


Double Coalesce



Double Coalesce

Double Coalesce



Bookkeeping: free list

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

Bookkeeping: free list

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

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

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?

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 **in** free space!

Bookkeeping: free list

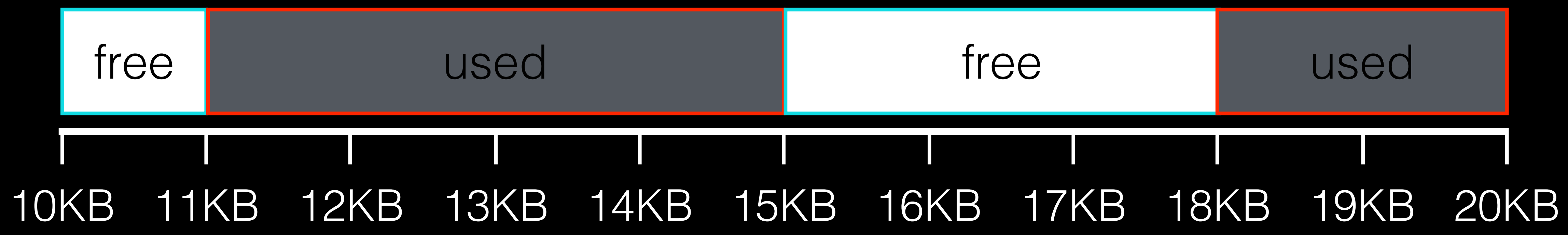
What fields do we need for each node in linked list?

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

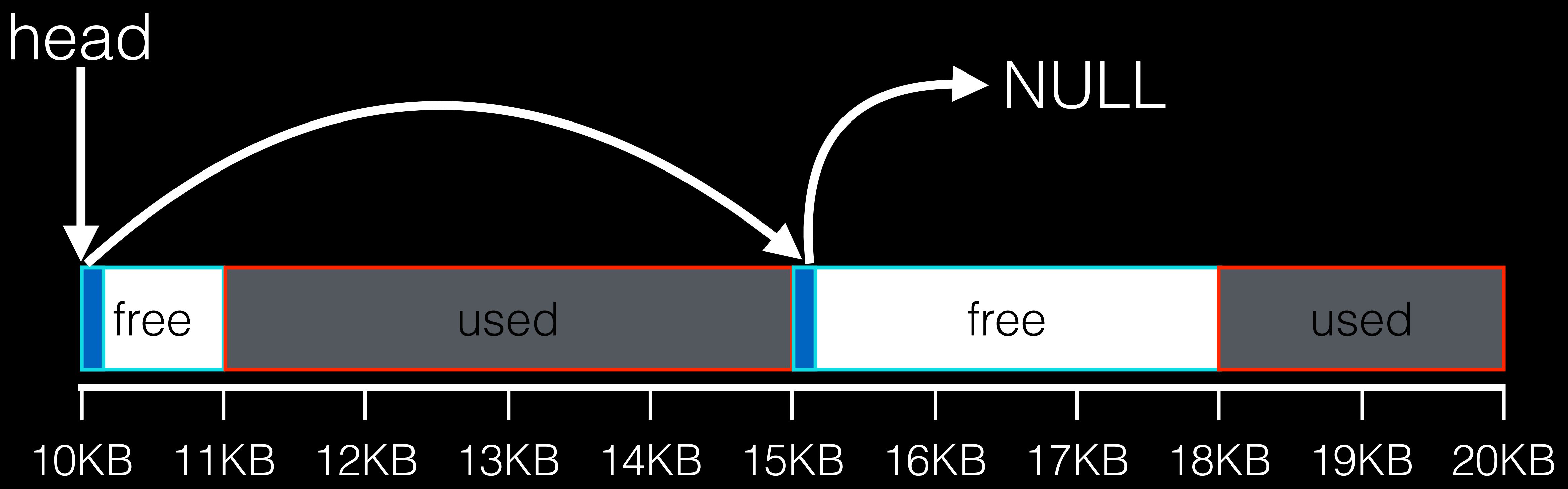
How do we allocate memory for new nodes?

- store them **in** 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()

Bookkeeping

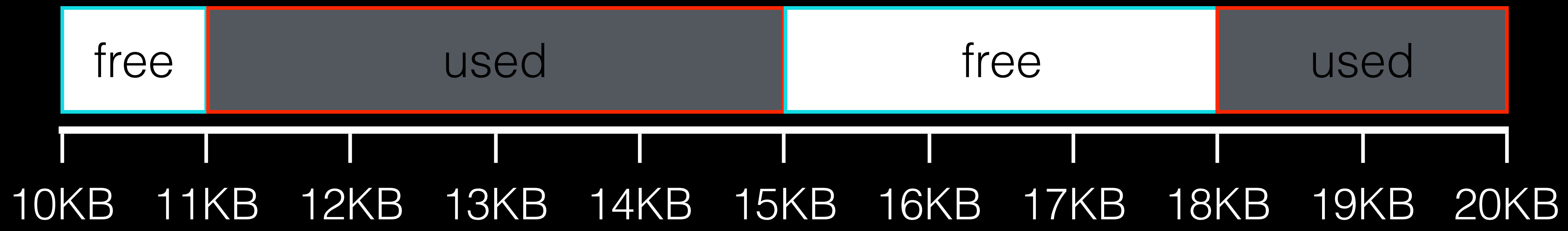
Need to know size+location of free spaces

- for malloc()

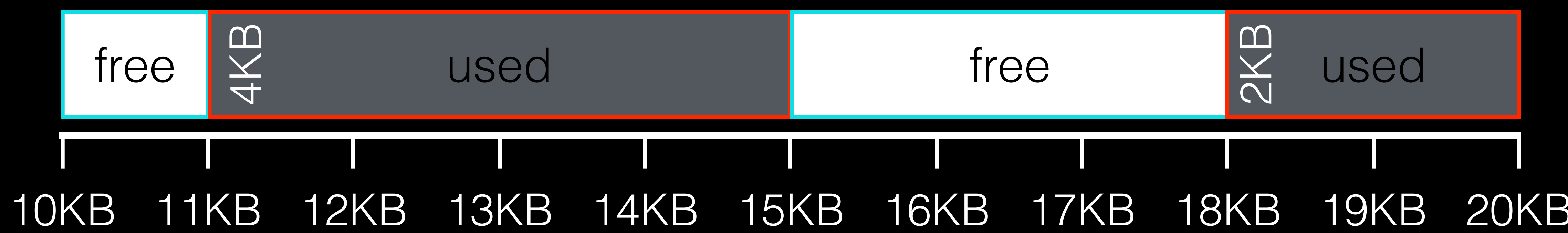
Need to know size of used spaces

- for free()

Used Size


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

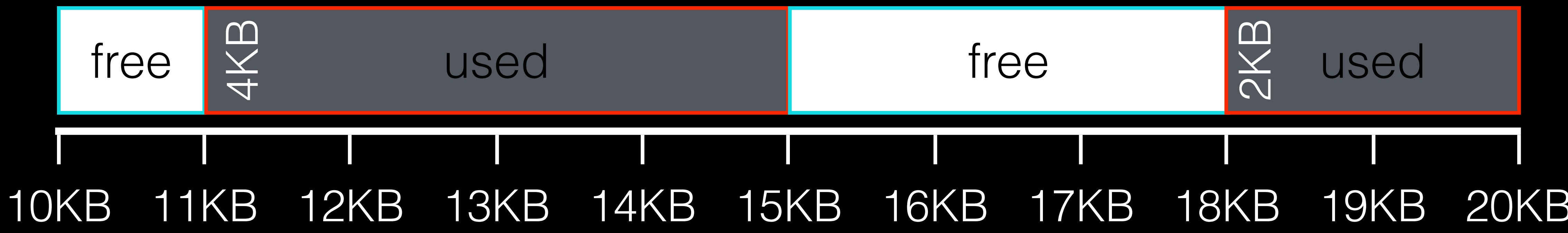
Used Size



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

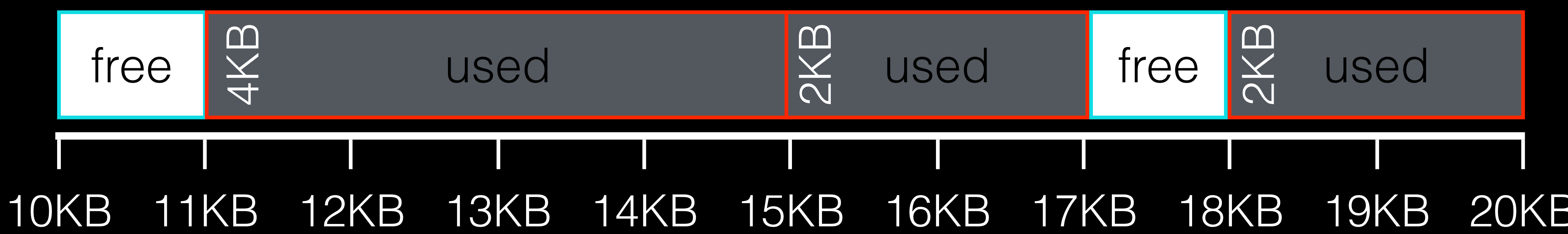
Used Size

`malloc(2KB)`



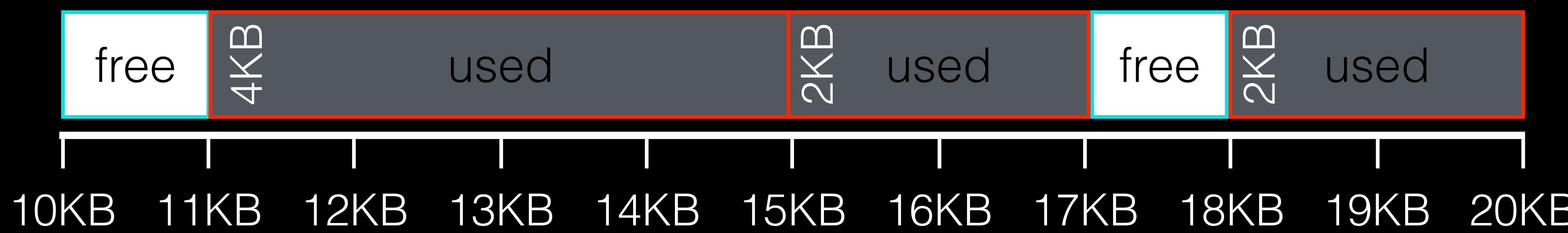
Used Size

```
malloc(2KB) = ?
```



Used Size

`malloc(3KB) = 15KB + sizeof(int)`



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?

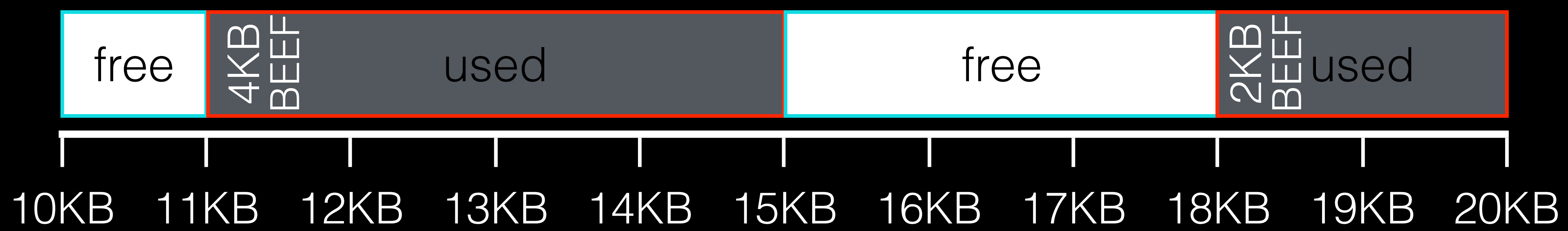
Magic Numbers

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

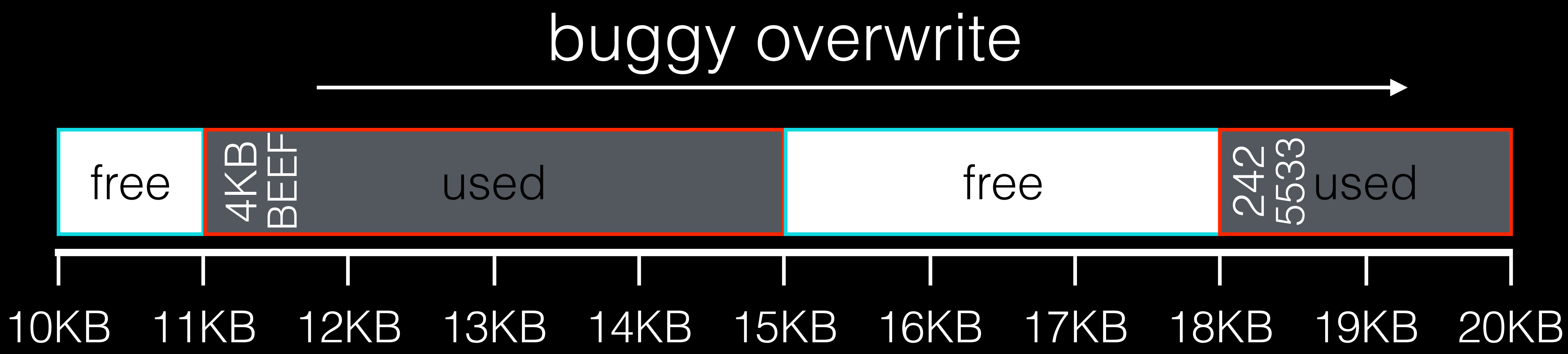
Add magic number to each allocated segment, if it is overwritten, there's a bug!

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

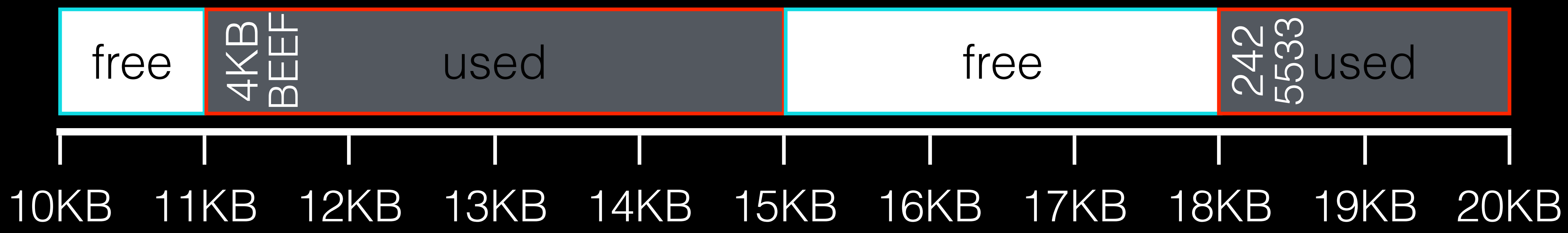
Magic



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



Allocator Policy

Which free space to consider?
Of those considered, which to use?

No perfect solutions!

Where to allocate?

Workload 1
1) need: 2KB

Where to allocate?

Workload 1
1) need: 2KB

Where to allocate?

Workload 1
1) need: 2KB
2) need: 3KB

Where to allocate?

Workload 1
1) need: 2KB
2) need: 3KB

Where to allocate?

Workload 1
1) need: 2KB
2) need: 3KB
3) need: 2KB

Where to allocate?

Workload 1
1) need: 2KB
2) need: 3KB
3) need: 2KB
(fail)

Where to allocate?

Workload 1
1) need: 2KB
2) need: 3KB
3) need: 2KB
(fail)

Where to allocate?

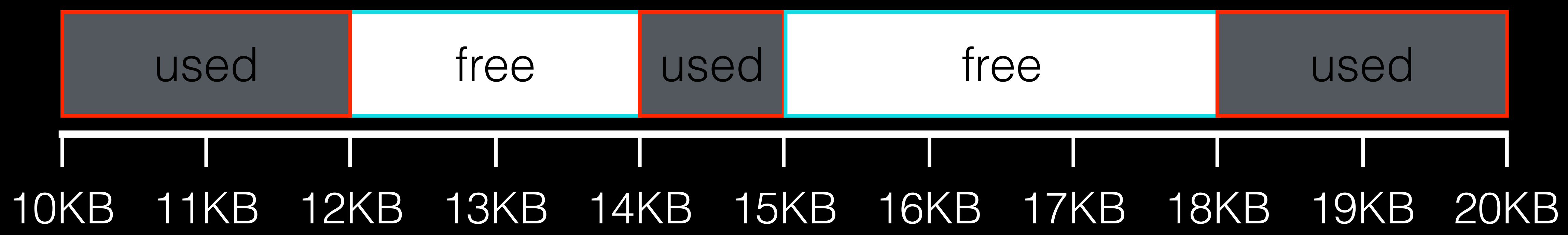
Workload 1
1) need: 2KB
2) need: 3KB
3) need: 2KB
(fail)

Workload 2
1) need: 2KB

Where to allocate?

Workload 1
1) need: 2KB
2) need: 3KB
3) need: 2KB
(fail)

Workload 2
1) need: 2KB



Where to allocate?

Workload 1
1) need: 2KB
2) need: 3KB
3) need: 2KB
(fail)

Workload 2
1) need: 2KB
2) need: 4KB

Where to allocate?

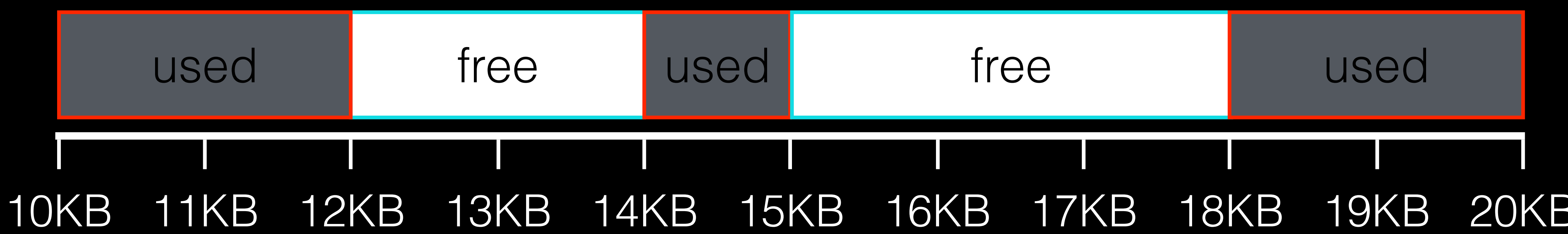
Workload 1
1) need: 2KB
2) need: 3KB
3) need: 2KB
(fail)

Workload 2
1) need: 2KB
2) need: 4KB
(fail)

Where to allocate?

Workload 1	Workload 2
1) need: 2KB	1) need: 2KB
2) need: 3KB	2) need: 4KB
3) need: 2KB	(fail)
(fail)	

no right answer



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:

$$f(\mathbf{W}, \mathbf{S}) = \mathbf{M}$$

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:

$$f(\mathbf{W}, \mathbf{A}) = \mathbf{M}$$

Allocator Basics

Workload:
- operations
- addresses
- sizes

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

Metrics:
- internal fragmentation
- external fragmentation
- search time

Allocator Basics

Workload:
- operations
- addresses
- sizes

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

Metrics:
- internal fragmentation
- external fragmentation
- search time

read more in OSTEP

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

P2: Visit 537 site