INF113: Segmentation and Free Space

Kirill Simonov

03.10.2025

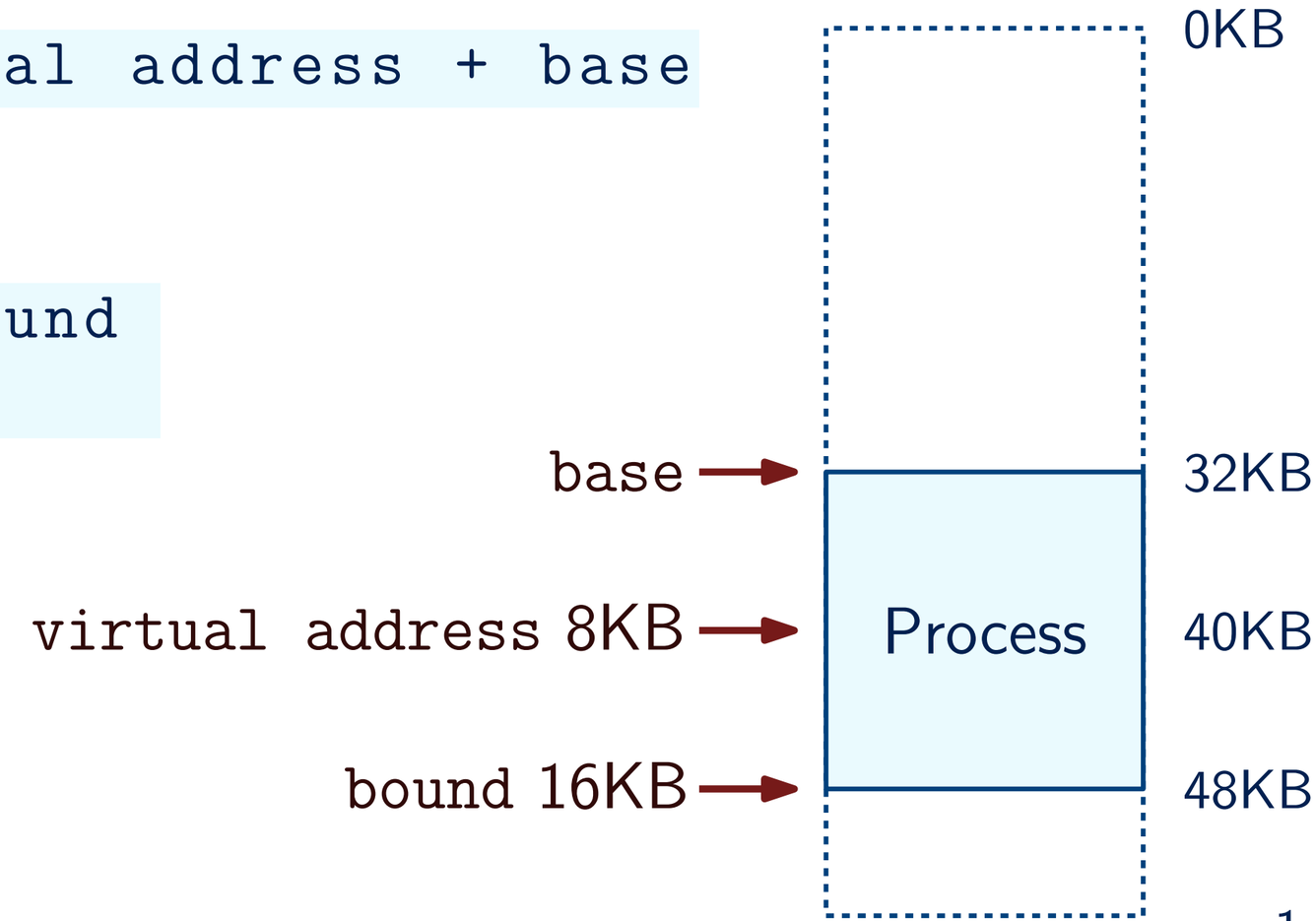
Base and bound: Reminder

- **Mechanism:** Store boundaries of process space in registers
- Translating addresses:

```
physical address = virtual address + base
```

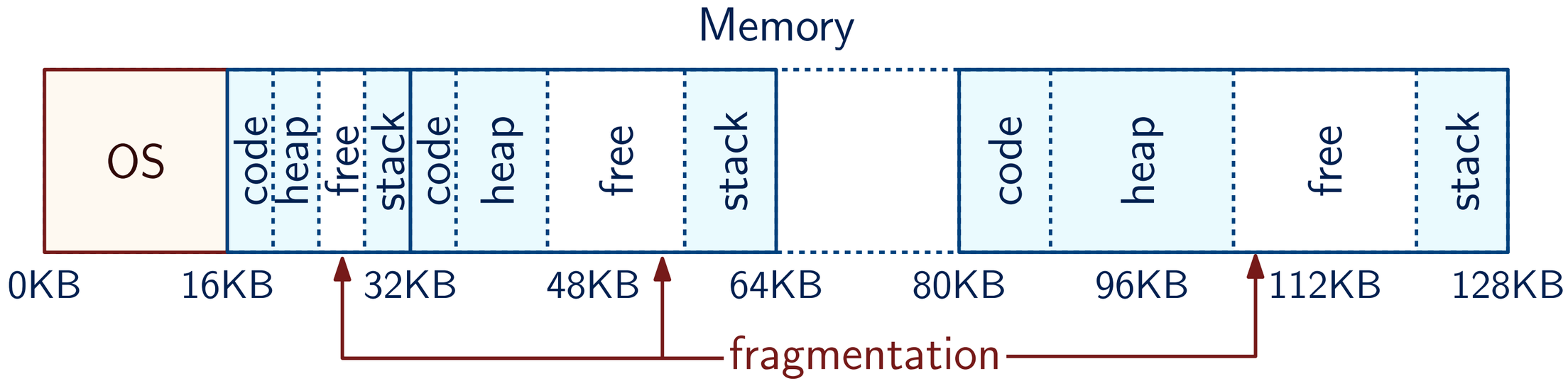
- Checking validity:

```
if virtual address >= bound
  fail
```

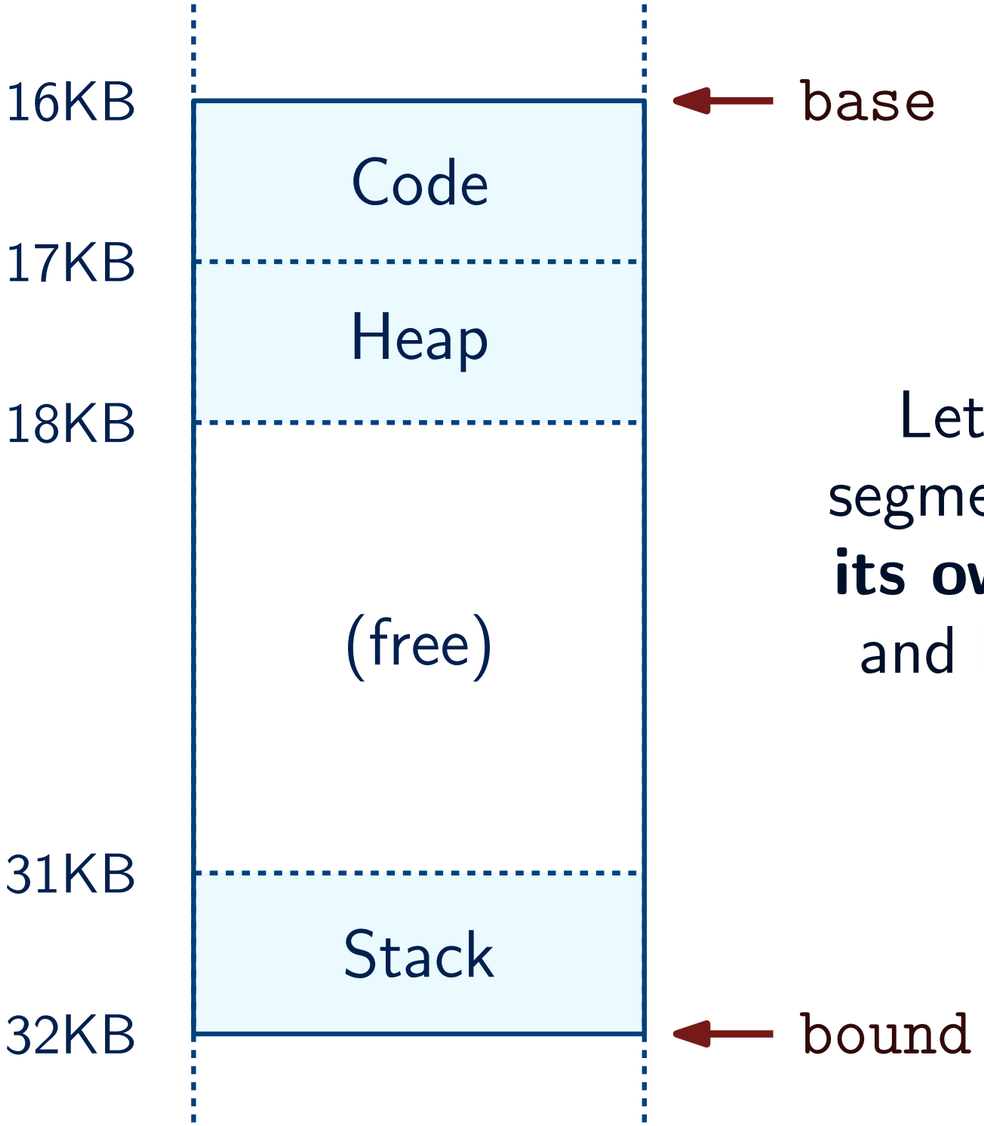


Base and bound: Issues

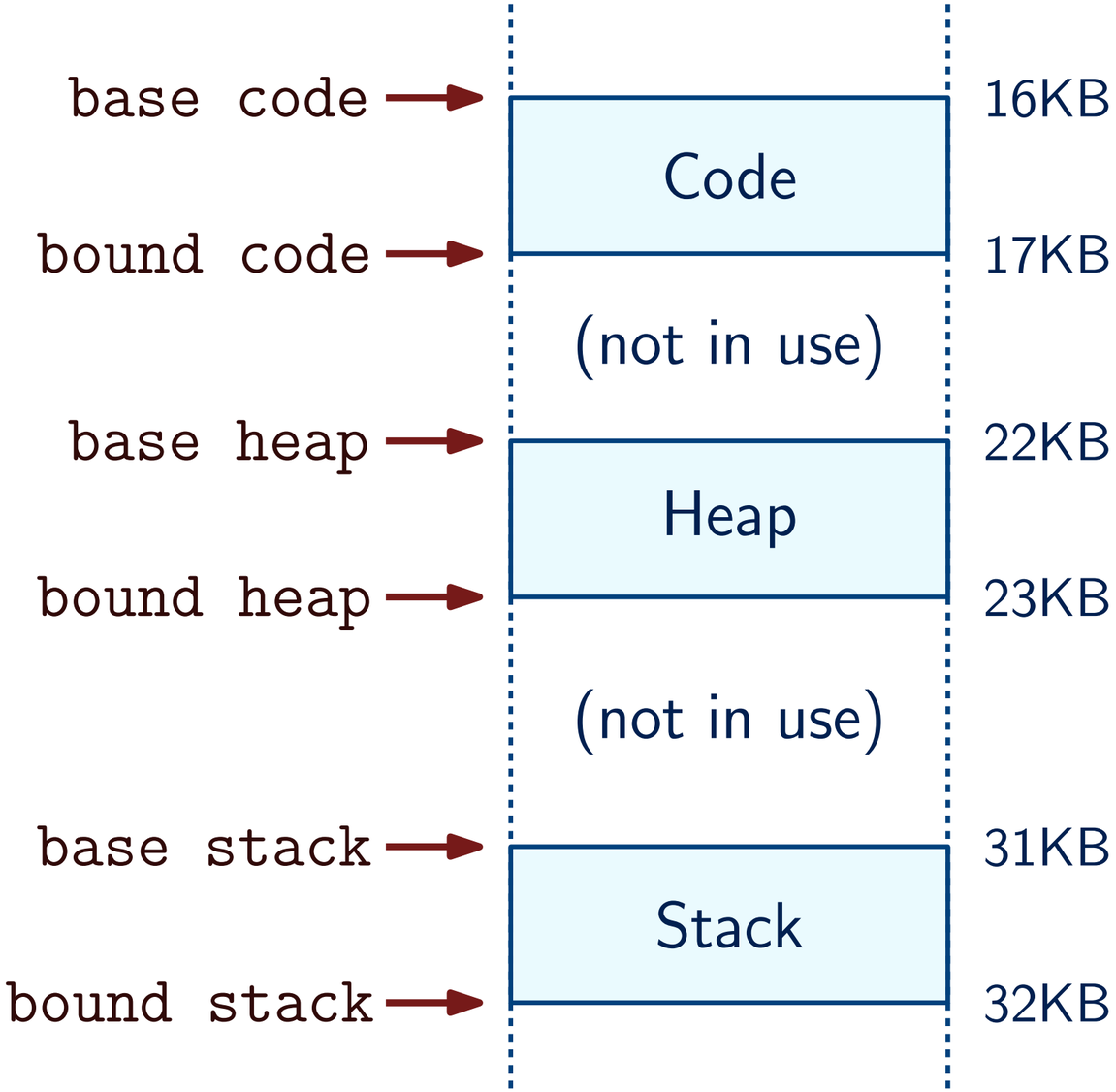
- Issues with same-sized address spaces:
  - Each address space is small
  - Most processes will not use most of their free memory—**fragmentation**
- If we made address spaces different-sized but still contiguous?
- Each process should have a large chunk of free memory within
  - If e.g. heap needs more memory, whole process has to be relocated



Segmentation



Let each segment have **its own** base and bound!



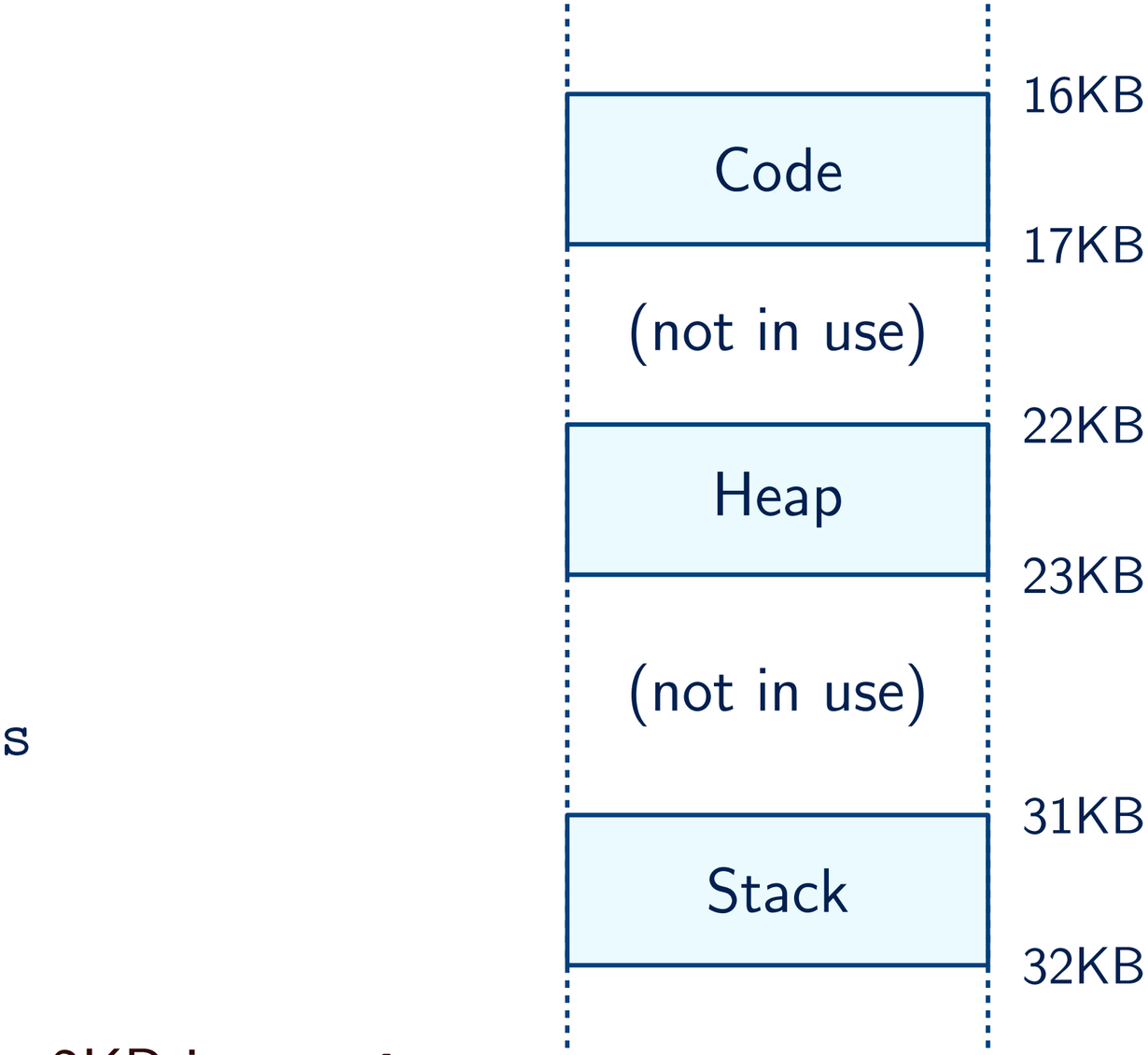
Implementing segmentation

- Store the segment table on registers:

	base	bound
code	16KB	1KB
heap	22KB	1KB
→ stack	31KB	1KB

- To translate a virtual address, use the bounds for the respective segment

virtual address		physical address
0.5KB in heap	→	22.5KB
0.5KB in code	→	16.5KB
2KB in stack	→	fail



How to know the segment?

- Virtual addresses should still be simple

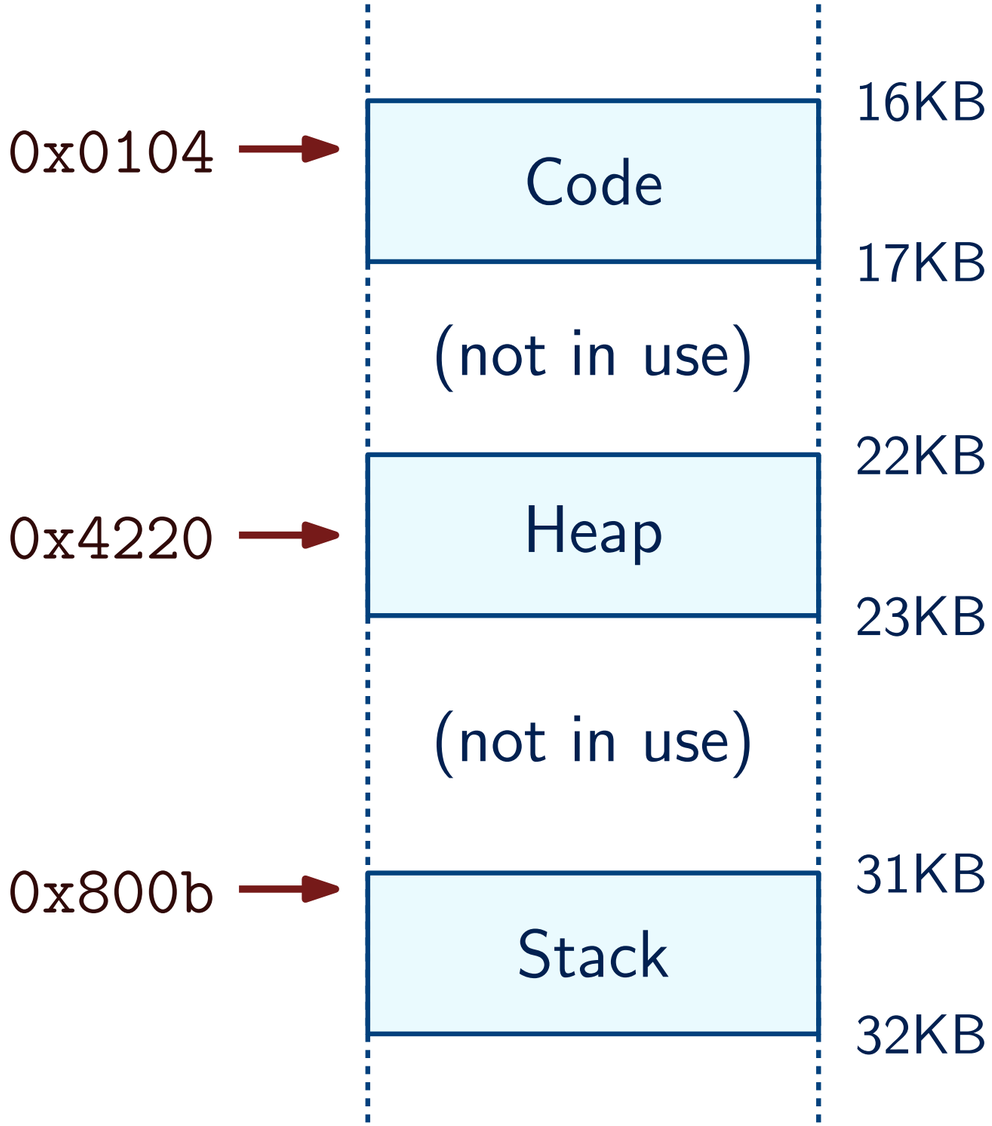
get value at `0x014f` **vs** get value at `(code, 0x014f)`

- Possible solution: Use leading bits to describe the segment

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	1	0	1	1

segment offset

00: `code`	260	`0x0104`
01: `heap`	544	`0x4220`
11: `stack`	11	`0x800b`

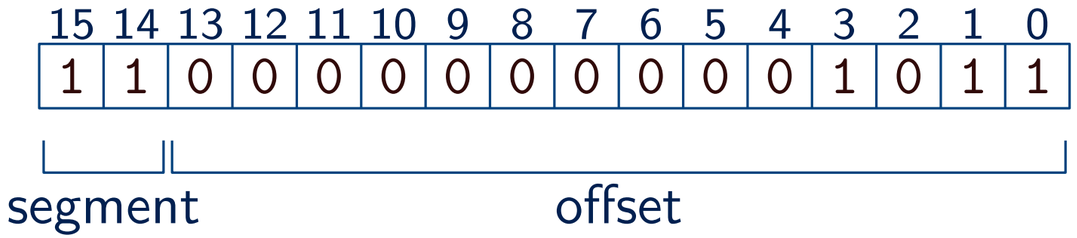


How to know the segment?

- Virtual addresses should still be simple

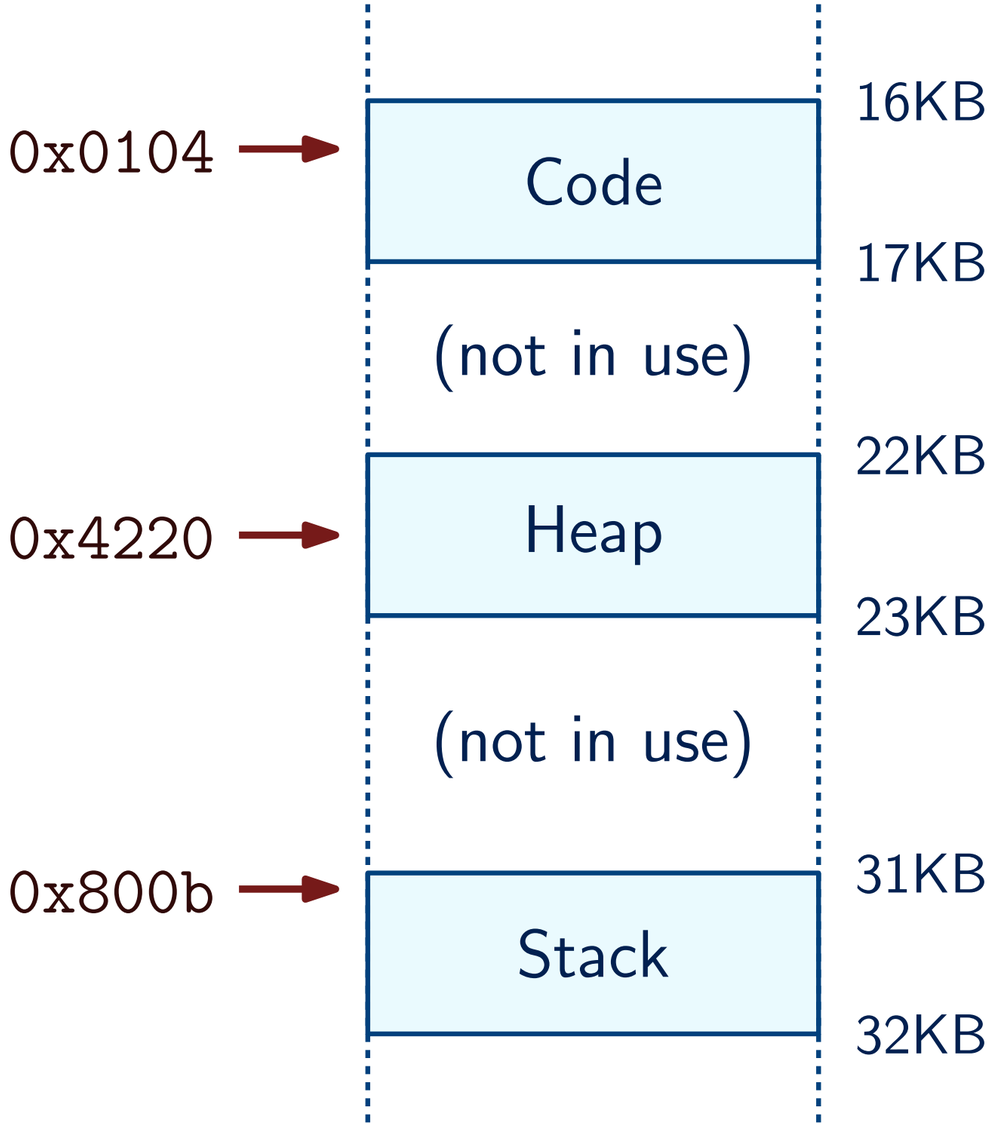
get value at `0x014f` **vs** get value at `(code, 0x014f)`

- Possible solution: Use leading bits to describe the segment



- MMU can quickly compute the physical address

```
segment = address >> 14;
offset = address & 0x3fff;
if (offset >= bounds[segment])
  fail;
phys_address = base[segment] + offset;
```

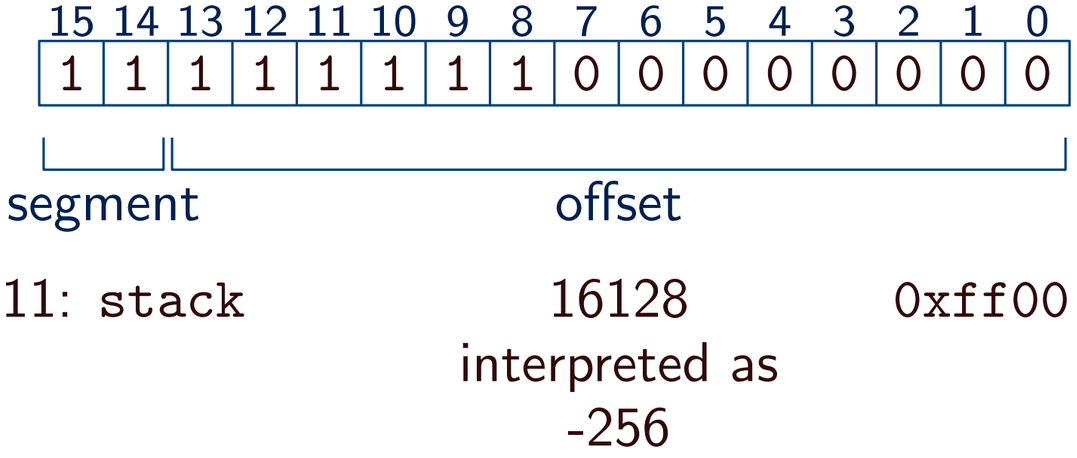


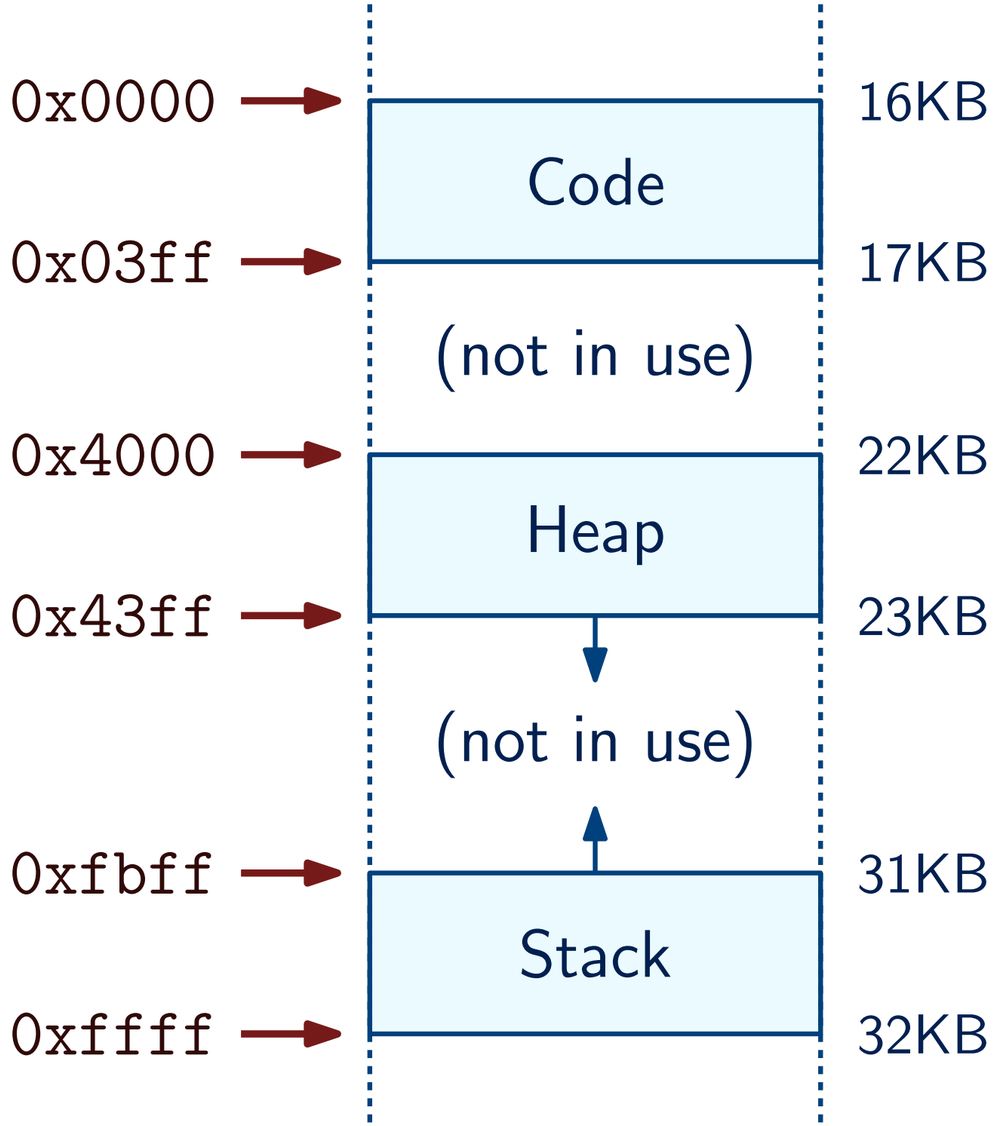
What about stack?

- Store direction of growth

	base	bound	dir
code	16KB	1KB	1
heap	22KB	1KB	1
stack	32KB	1KB	0

- Intepret offset as a negative number





Segment protection

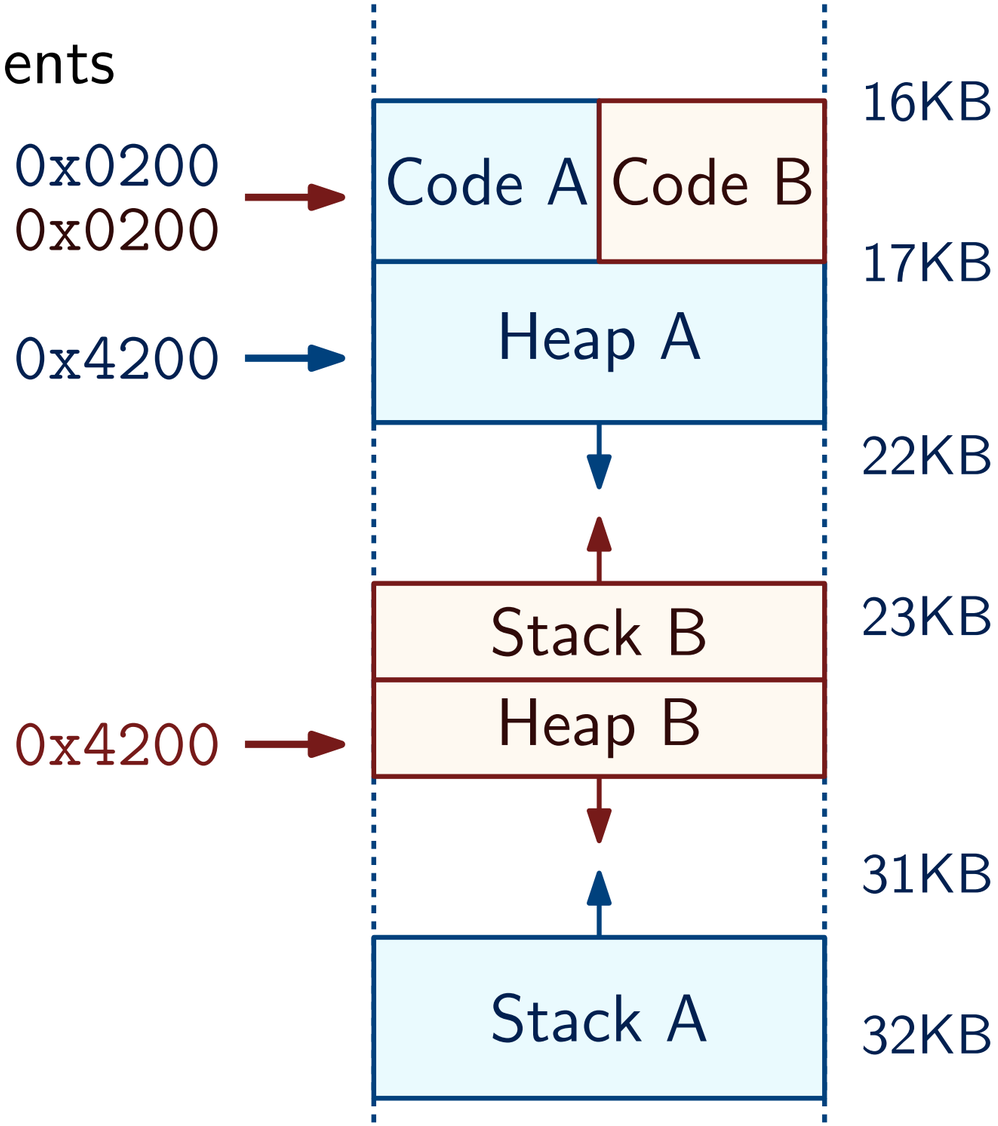
- Specify which operations are allowed on the segments

	base	bound	dir	rights
code	16KB	1KB	1	r-x
heap	22KB	1KB	1	rw-
stack	32KB	1KB	0	rw-

- MMU additionally checks operation type

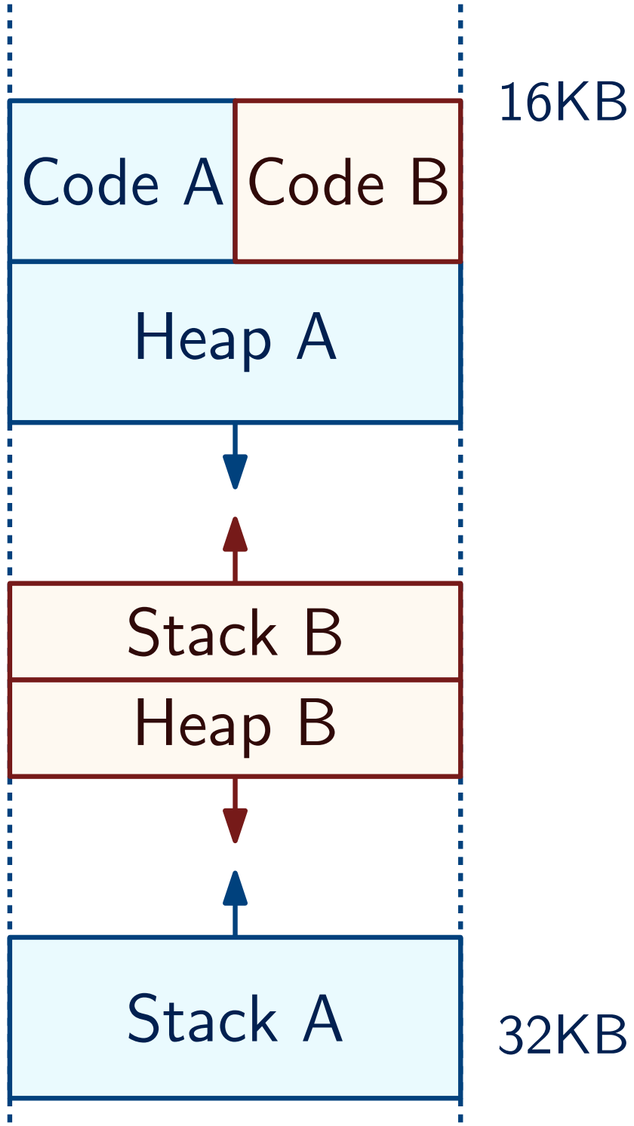
```
...
if (op & rights == 0
  || offset >= bounds[segment])
  fail;
...
```

- Read-only segments can be shared!



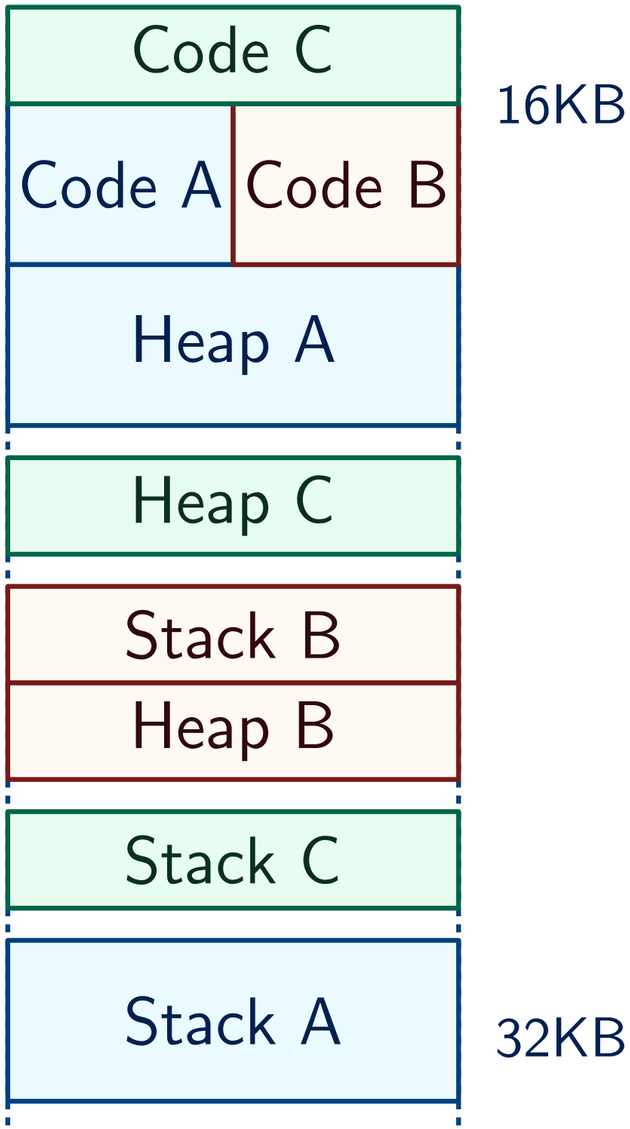
Why segmentation?

- No "reserved" gaps within individual address spaces
- More flexibility $\rightarrow$ filling gaps more easily
- Can relocate segments individually $\rightarrow$ less overhead
- Additional savings via sharing segments



Why segmentation?

- No "reserved" gaps within individual address spaces
- More flexibility $\rightarrow$ filling gaps more easily
- Can relocate segments individually $\rightarrow$ less overhead
- Additional savings via sharing segments
- Still, lots of room for fragmentation...
- Real solution next week: **Paging**



Segmentation fault

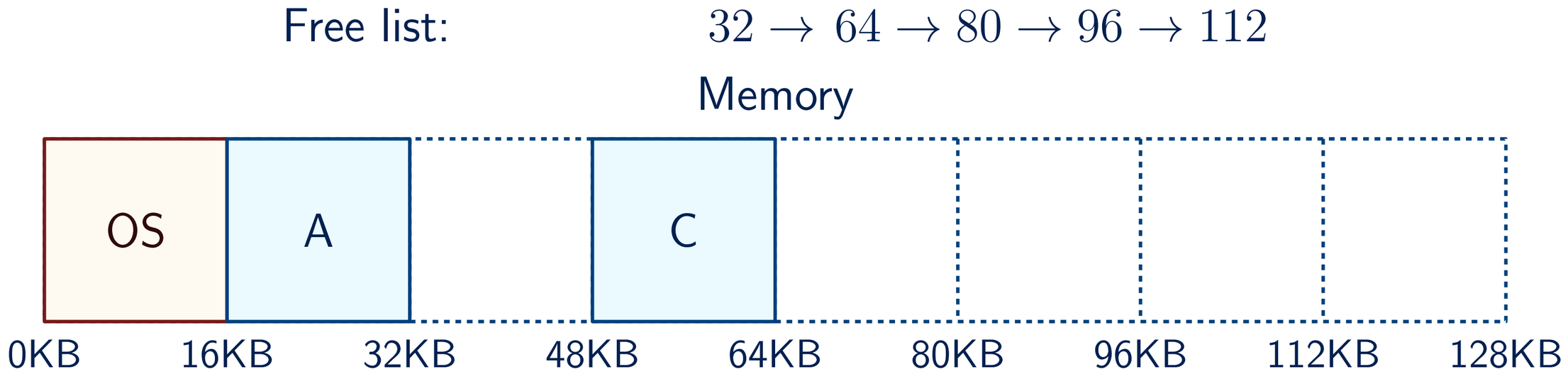
- The scariest thing a programmer can see...

- Simply means that the segmentation table is violated

	base	bound	dir	rights
code	16KB	1KB	1	r-x
heap	22KB	1KB	1	rw-
stack	32KB	1KB	0	rw-

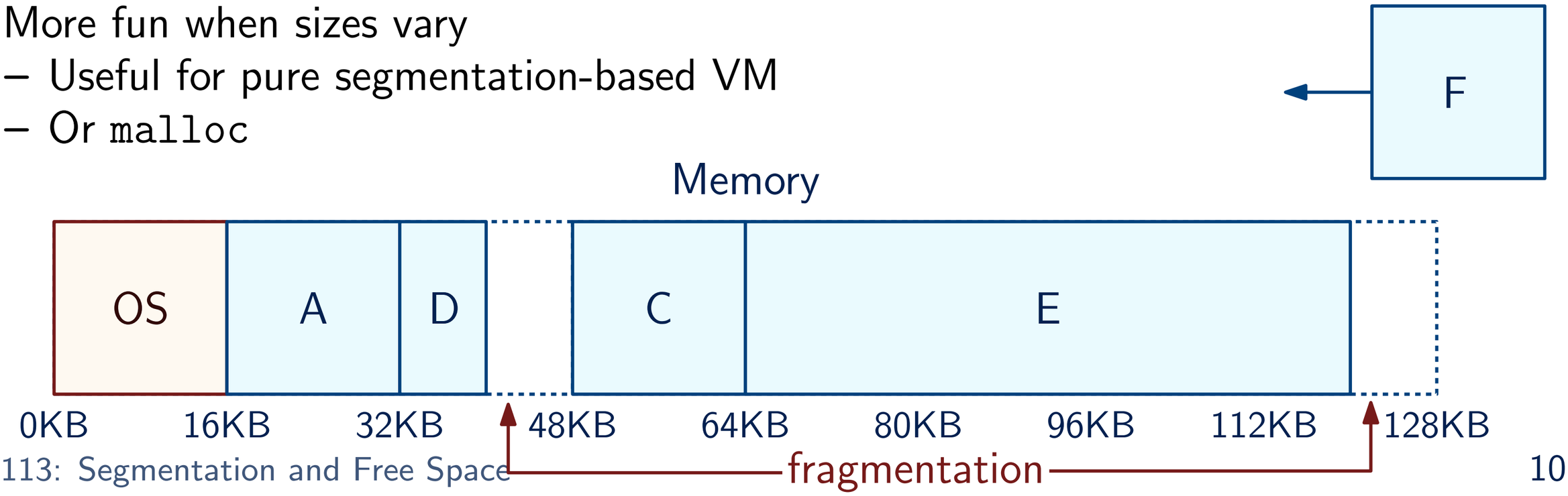
Free space management

- How to manage the memory well?
  - Find a place for the next process/segment efficiently
  - Avoid fragmentation
- Easy when all requests have same size
  - Also holds for paging-based VM in OS



Free space management

- How to manage the memory well?
  - Find a place for the next process/segment efficiently
  - Avoid fragmentation
- Easy when all requests have same size
  - Also holds for paging-based VM in OS
- More fun when sizes vary
  - Useful for pure segmentation-based VM
  - Or `malloc`

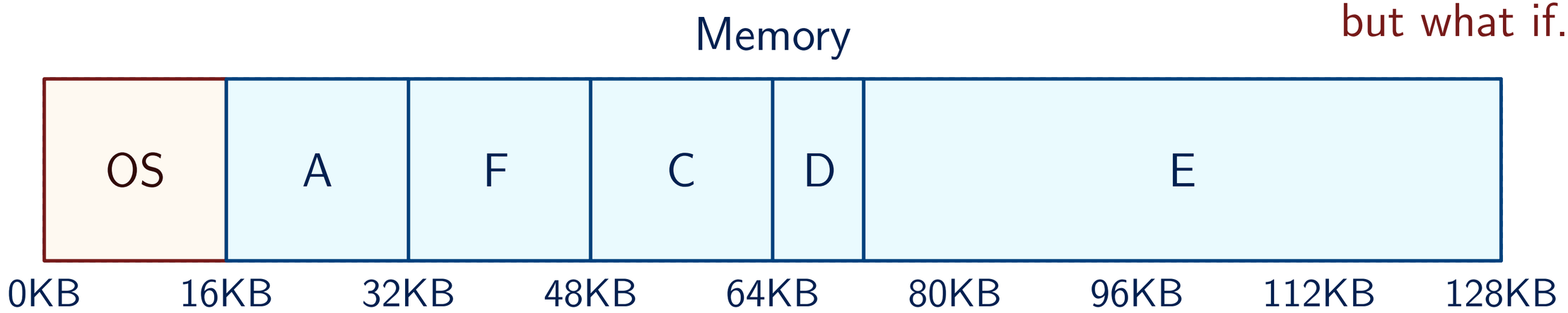


Free space management

- How to manage the memory well?
  - Find a place for the next process/segment efficiently
  - Avoid fragmentation
- Easy when all requests have same size
  - Also holds for paging-based VM in OS
- More fun when sizes vary
  - Useful for pure segmentation-based VM
  - Or `malloc`

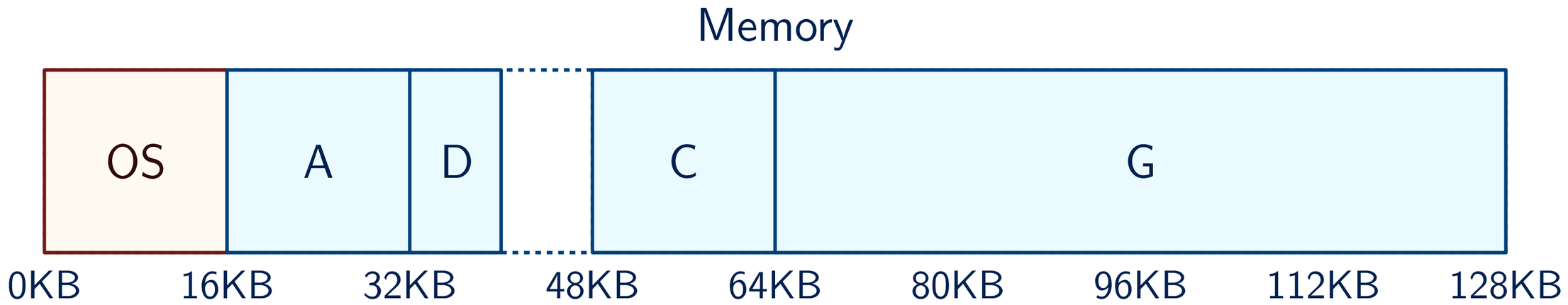
perfect!

but what if. . .



Free space management

- How to manage the memory well?
  - Find a place for the next process/segment efficiently
  - Avoid fragmentation
- Easy when all requests have same size
  - Also holds for paging-based VM in OS
- More fun when sizes vary
  - Useful for pure segmentation-based VM
  - Or `malloc`

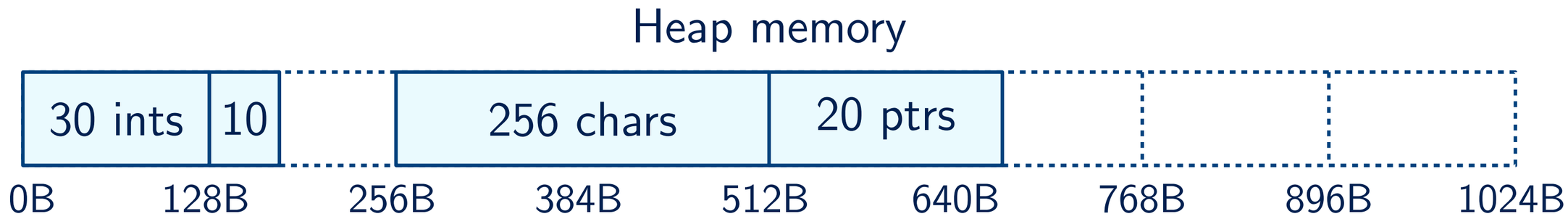


Malloc

- What happens then?

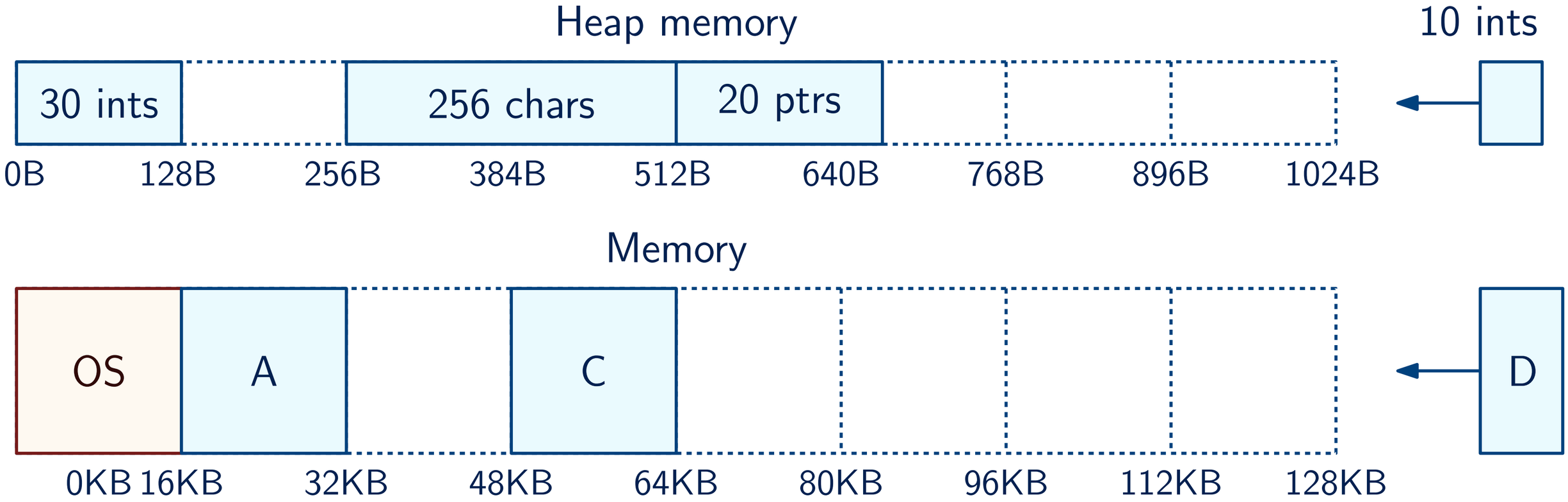
```
int* a = malloc(10 * sizeof(int));
```

- `malloc` does not just go to the OS to ask for memory—syscall/context switch is too slow!
- Instead, `malloc` will try to fit the request into already allocated memory
- Only if this fails, `malloc` will ask the OS to extend the heap via e.g. `sbrk`
- If this fails too, `malloc` will return `NULL`—watch out!



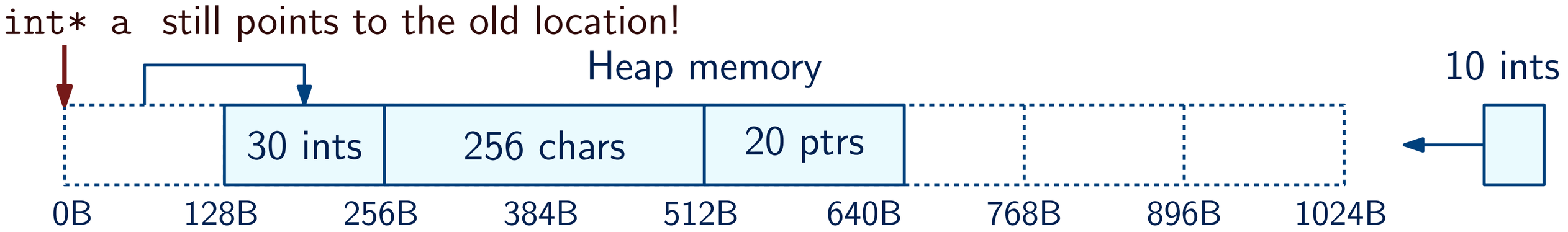
Malloc vs OS VM

- Both need memory management logic
  - Where to place the next request?
  - How to keep track of free/occupied memory?



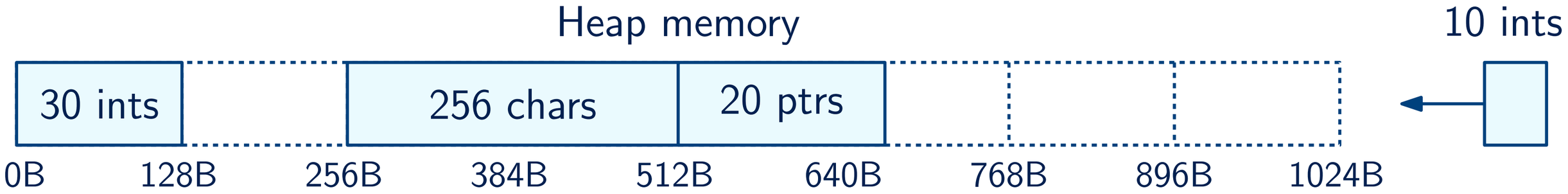
Malloc specifics

- `malloc` cannot relocate segments
  - There are pointers in the code, and they will be invalidated



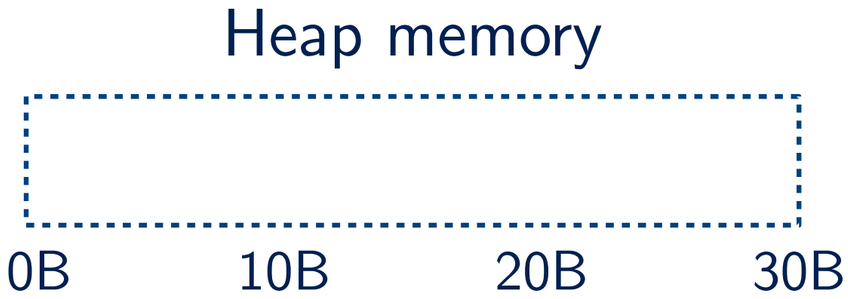
Malloc specifics

- `malloc` cannot relocate segments
  - There are pointers in the code, and they will be invalidated
- `malloc` can ask for more memory
  - But it should only ask if absolutely necessary
- `malloc` cannot call `malloc`!
  - The internal `malloc` data structure has to be allocated explicitly

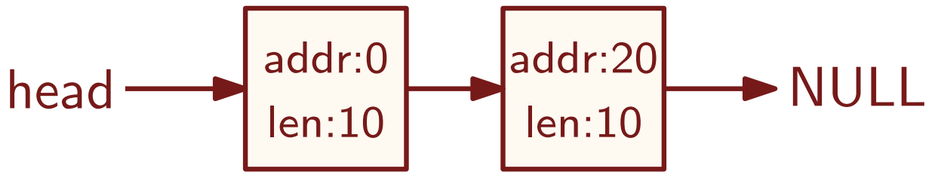


Free list

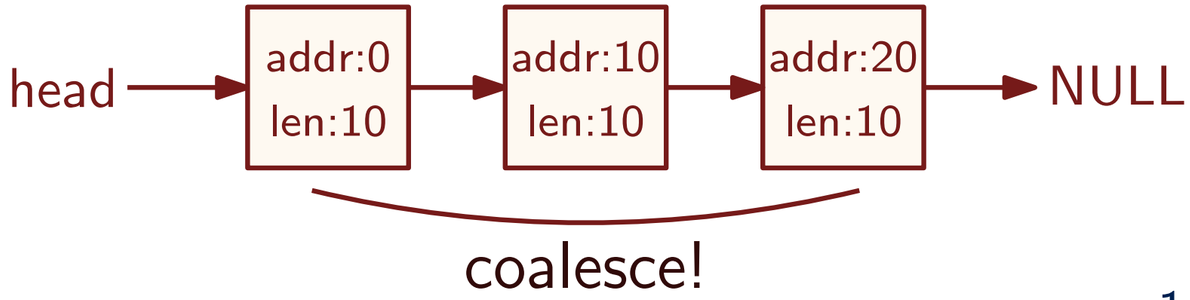
- Let us store all free ranges in the list—**free list**



- **Split:** Allocating a segment splits the range



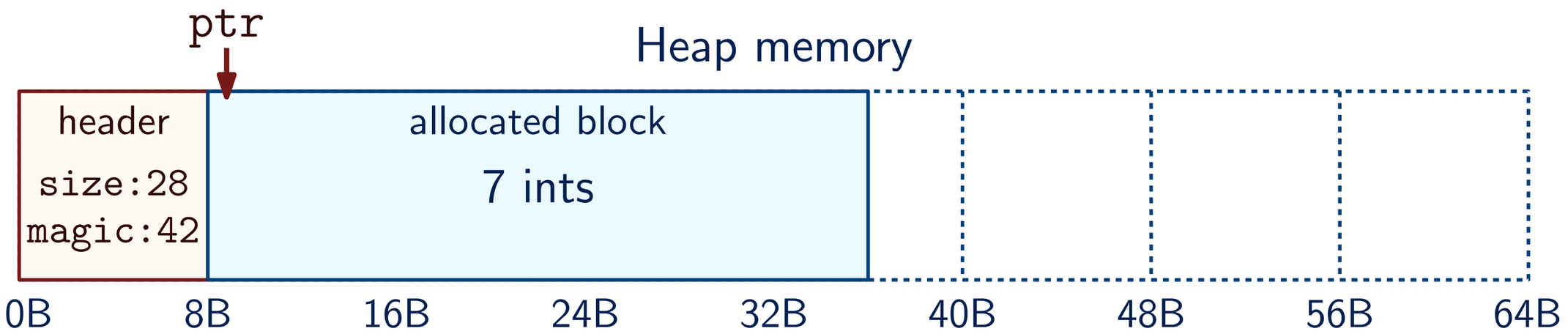
- **Coalesce:** Once a segment is freed, merge adjacent free ranges



Header blocks

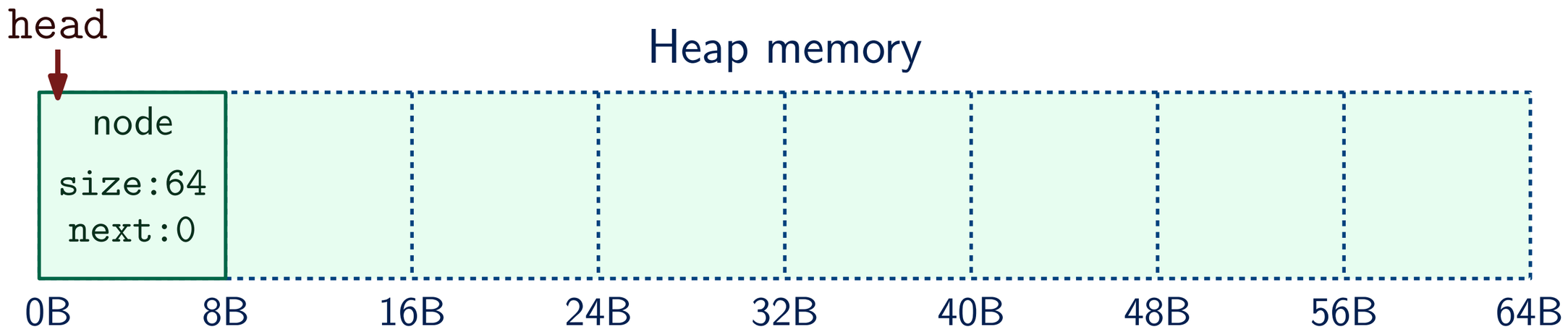
- When we call `free(10)`, we only supply the address—how to determine the size?
- Store information directly in the memory!
- Each allocated block comes with a header
  - Size—to know what is occupied
  - Magic number—to make sure the header is actually a header

```
int* ptr = malloc(7 * sizeof(int));
```



Embedding free list

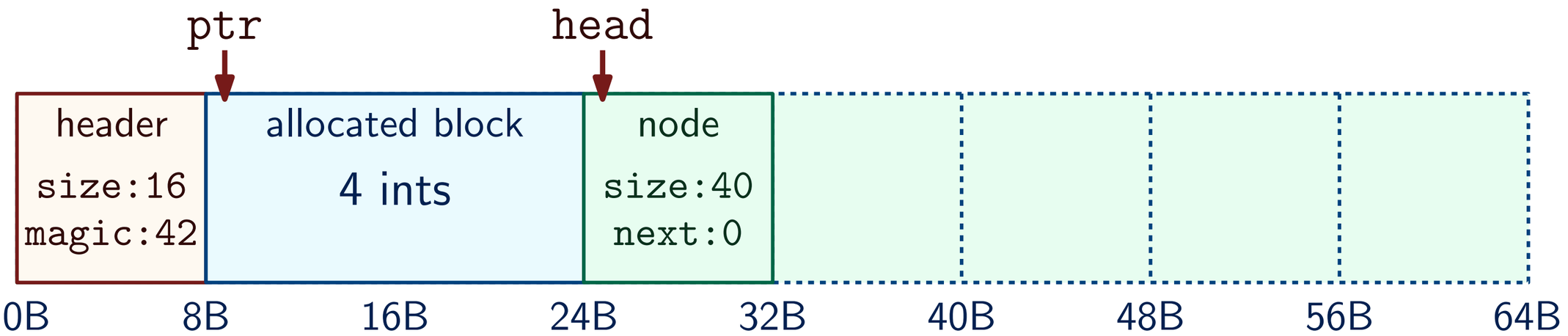
- Apply the same trick for the free list
  Store free range info at the start of the range
- When a range splits, insert new node



Embedding free list

- Apply the same trick for the free list
  Store free range info at the start of the range
- When a range splits, insert new node

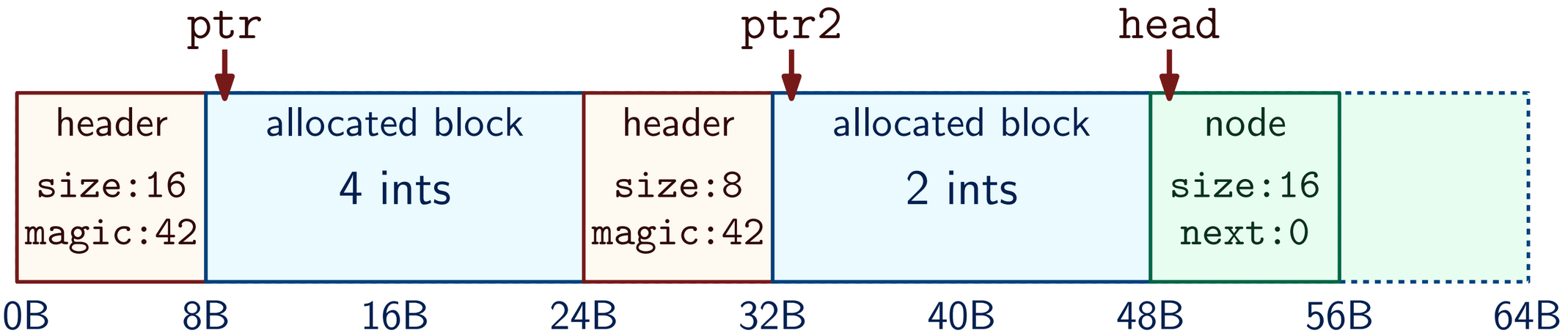
```
int* ptr = malloc(4 * sizeof(int));
```



Embedding free list

- Apply the same trick for the free list
  Store free range info at the start of the range
- When a range splits, insert new node

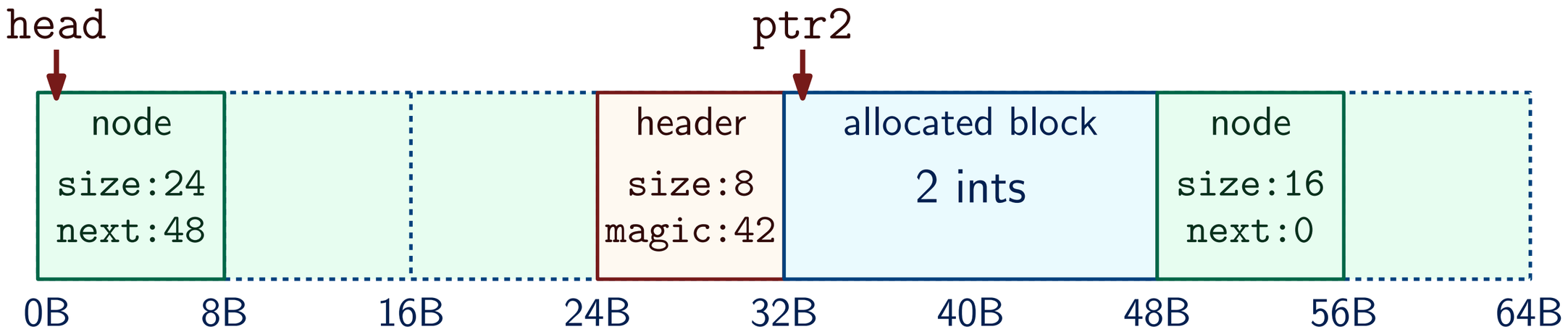
```
int* ptr = malloc(4 * sizeof(int));
int* ptr2 = malloc(2 * sizeof(int));
```



Embedding free list

- Apply the same trick for the free list
  Store free range info at the start of the range
- When a range splits, insert new node

```
int* ptr = malloc(4 * sizeof(int));
int* ptr2 = malloc(2 * sizeof(int));
free(ptr);
```

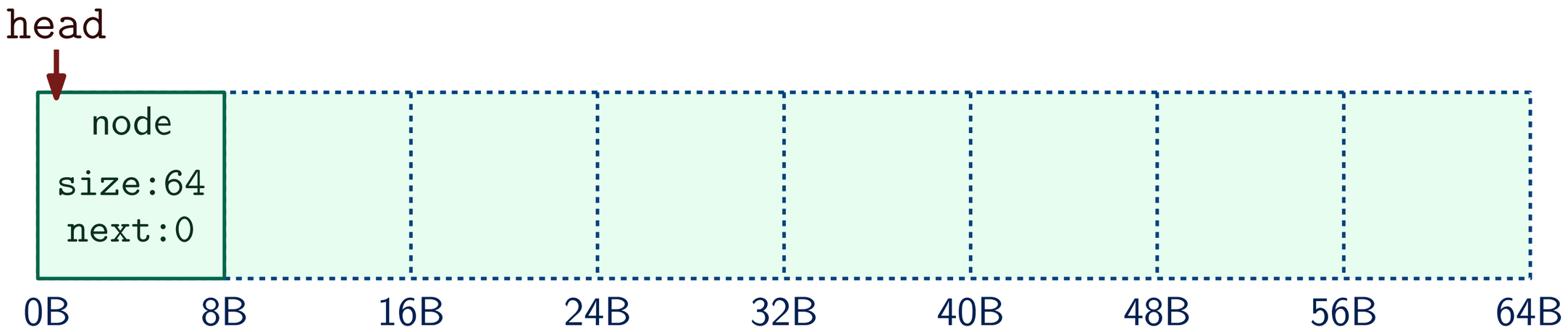


Embedding free list

- Apply the same trick for the free list
  Store free range info at the start of the range
- When a range splits, insert new node

```
int* ptr = malloc(4 * sizeof(int));
int* ptr2 = malloc(2 * sizeof(int));
free(ptr);
free(ptr2);
```

- Coalesce when a block is freed

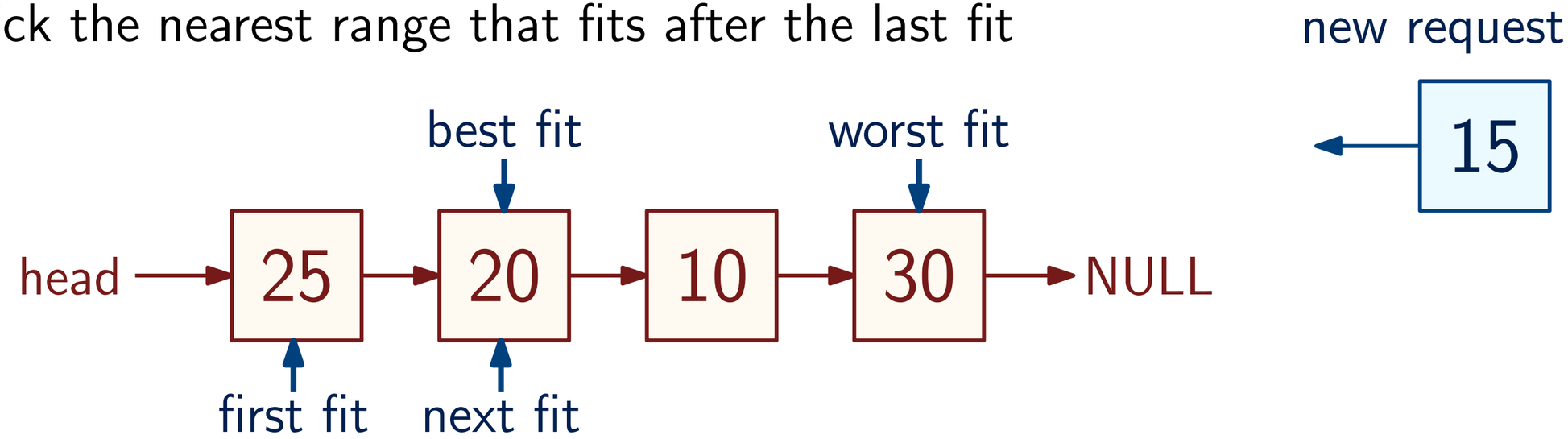


Modifying unallocated memory may break `malloc`!

Basic strategies

How to pick the free range?

- **Best fit**: Pick the smallest range that fits
- **Worst fit**: Pick the largest range that fits
- **First fit**: Pick the first range that fits
- **Next fit**: Pick the nearest range that fits after the last fit



Summary

- Segmentation: split address space into logical segments
  - The idea is heavily used in OS, but there's more
- Filling memory with arbitrary-length requests is a complicated task with various strategies
- Basic idea behind how `malloc` manages free/occupied ranges
  - The actual implementation is much more complicated
  - Focus on efficiency and specific fitting strategies

`sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/`

- **Next time:** Paging
- **Homework:** After Chapters 15 and 16