

# Operating Systems

(Sistemas Operativos)

## Memory Virtualization Abstractions and Mechanisms

University of Minho  
2024-2025



# What will we learn?

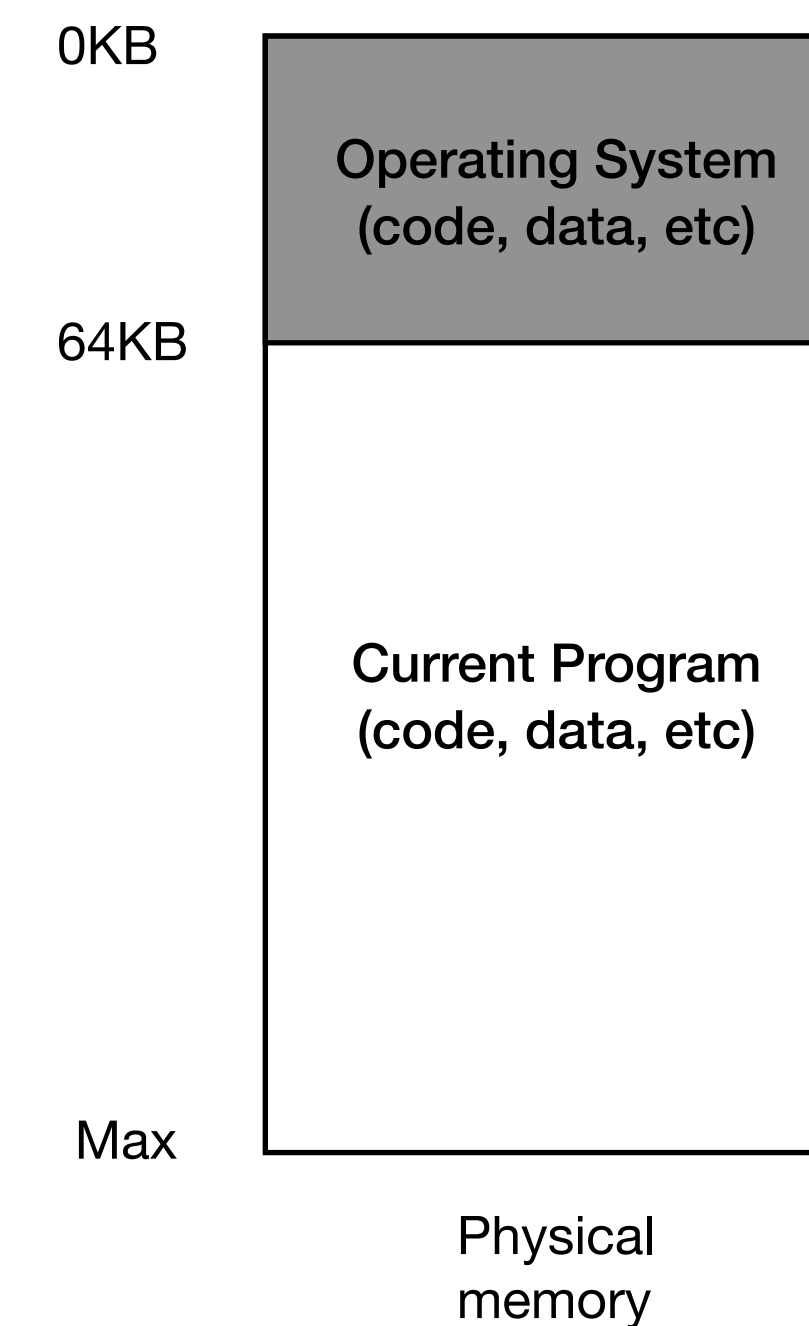
## Memory Management

- ◎ Physical memory can be seen as a simple array of bytes. Programs (running) need to store their instructions and data structures in memory
  - How does the OS abstract and manage memory accesses made by programs?
  - How is it possible to have multiple programs safely sharing the same physical memory?
  - How does the OS provide the illusion of having more memory, than the physically available one, for running programs?
- ◎ Let's study **abstractions, mechanisms** and **structures**!

# Memory Management

## Early systems...

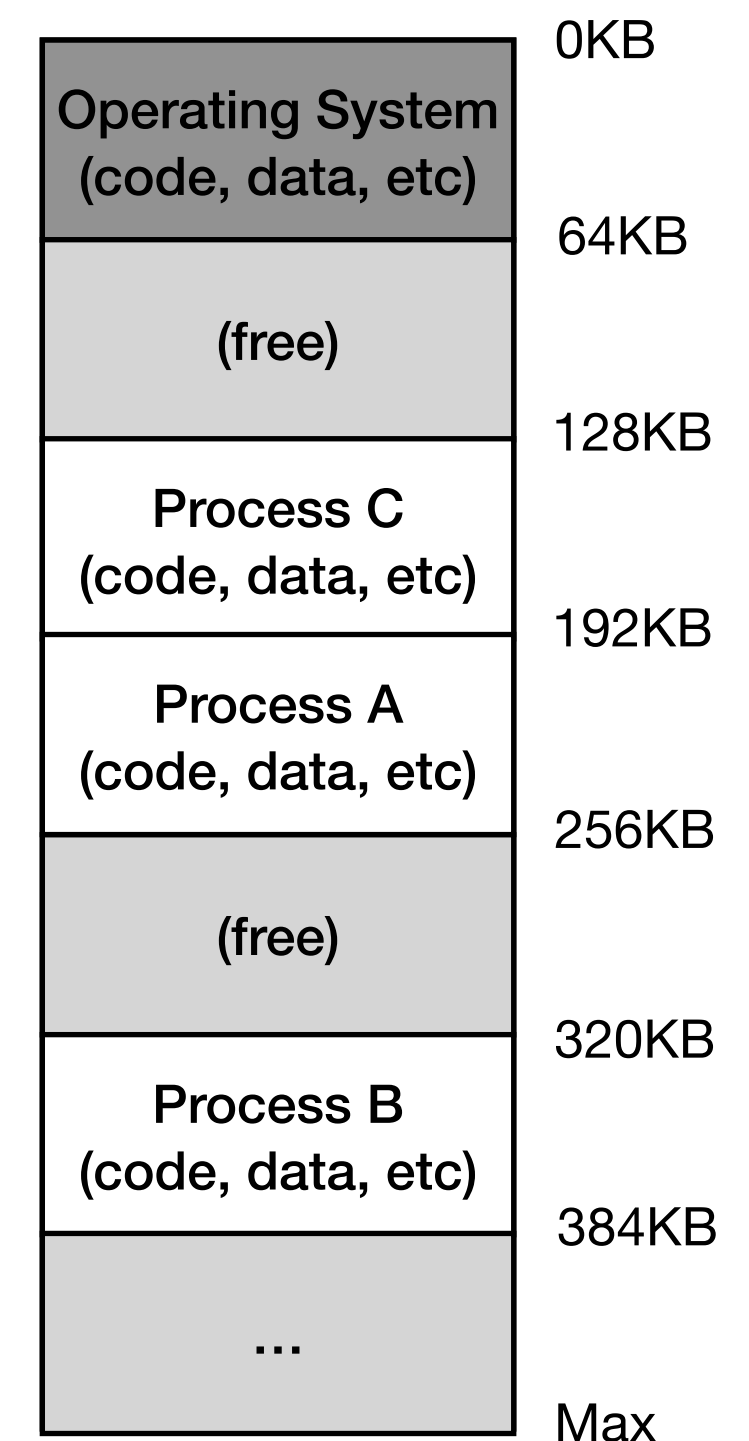
- The OS was a library (set of routines) placed at the beginning of the memory space (e.g., addresses 0 to 64KB)
- The rest of the memory was used to run the program (one program running at each time...)



# Memory Management

## The era of time sharing

- ◎ The era of **multiprogramming** and **time sharing** arrived
  - Processes are now swapped (e.g., while waiting for I/O, to make the system more interactive)
  - Process information must be loaded into CPU registers but doing it from disk is slow...
- ◎ **Solution:** Important information to run/resume processes that are *Ready* to execute is kept in main memory!



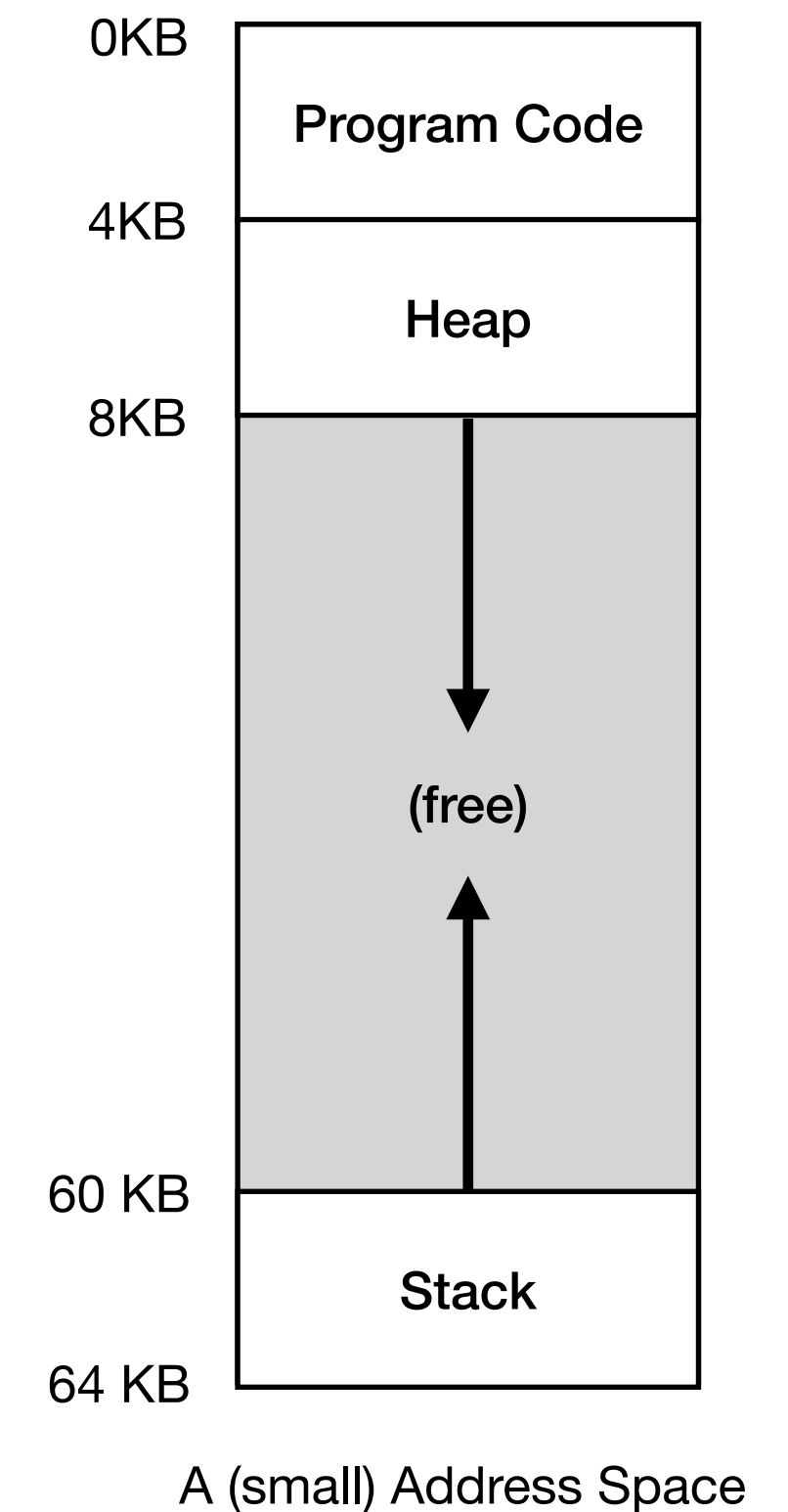
If these concepts seem confusing, look again at the *CPU Virtualization Mechanisms and Scheduling* slides!

Physical  
memory

# The Address Space

## Memory abstraction

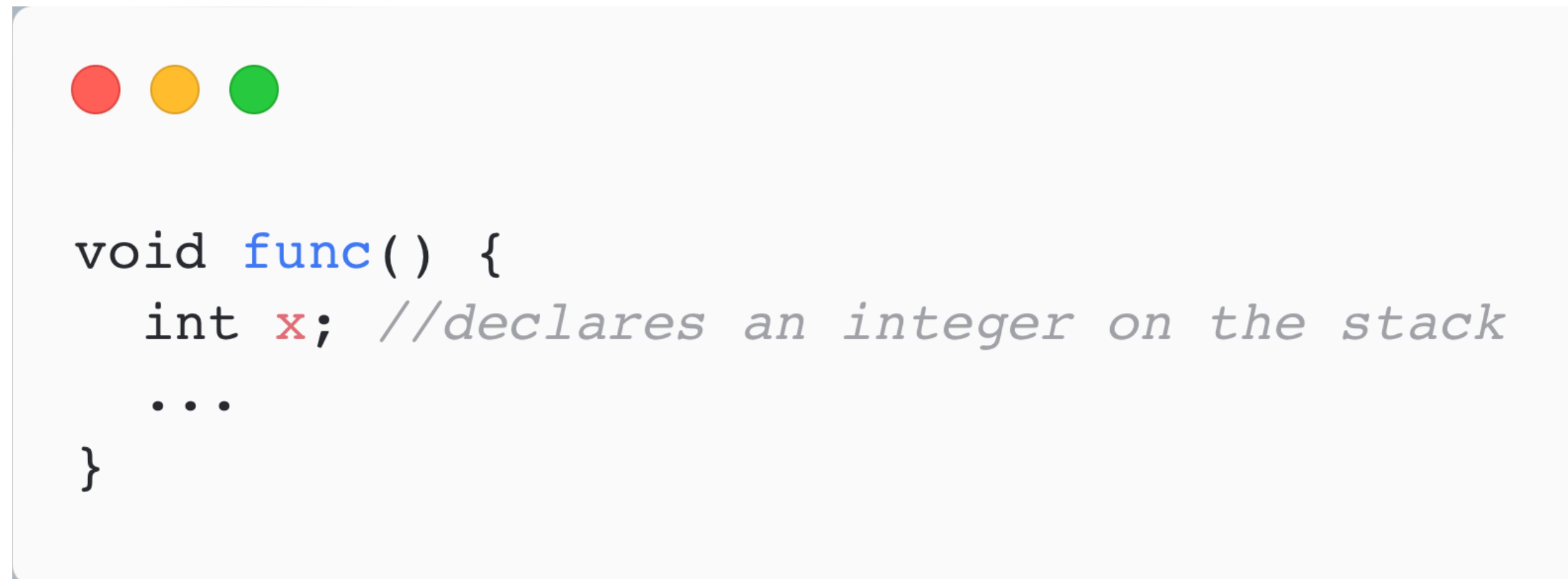
- The Address Space contains the memory state of a process
  - **Code:** the instructions of the program  
(the Code segment is also referred to as the *Text* in the literature)
  - **Stack:** local variables, arguments of routines, return values...
  - **Heap:** dynamically allocated memory (e.g., with *malloc()*)
- Heap and stack can grow while the program runs
- The address space may contain more segments other than the code, heap, and stack
  - E.g., the *Data* segment holds static data such as initialized (global) variables
  - But let's keep the address space simple for our examples next



# Memory API

## Types of Memory - Stack

- ◎ Stack memory (also known as automatic memory)
  - Allocations and deallocations are handled by the compiler
  - In the example below
    - when the *func()* is called, the compiler allocates space in the stack for variable *x*
    - The compiler deallocates the memory when the function returns



```
void func() {  
    int x; //declares an integer on the stack  
    ...  
}
```



# Memory API

## Types of Memory - Heap

- ◎ Heap memory, long-lived dynamic memory
  - Allocations and deallocations are handled by the programmer explicitly
  - In the example below both types of allocation occur
    - The compiler allocates stack memory for a pointer to an integer (*int \*x*)
    - When the program calls *malloc()*, it requests space for an integer at the heap



```
void func() {  
    int *x = (int *) malloc(sizeof(int));  
    ...  
}
```

# Memory API

## Malloc and Free call

● *void \*malloc(size\_t size);*

- Receives, as argument, the size (amount) of memory to allocate (in bytes)
- Returns a pointer to the newly allocated space (or NULL if the call fails)
- The *sizeof()* operator can be used to request the right amount of bytes for a given data type

● *void free(void \*ptr);*

- Used to free heap memory that is no longer being used
- Receives, as argument, the pointer to a memory region allocated with *malloc()*

● Note that *malloc()* and *free()* may be called in distinct functions



# Common Errors

## Malloc

- ◎ **Forgetting to allocate memory** may lead to a segmentation fault
  - We will learn why this error is called like this!
- ◎ **Not allocating enough memory** may lead to a buffer overflow error
  - Tricky error that may only manifest in some program executions
- ◎ **Forgetting to initialize allocated memory** may lead to program bugs
  - E.g., your program ends up reading strange values from an array of integers whose values were not explicitly initialized (e.g., zeroed)

# Common Errors

## Free

- ◎ **Freeing memory before being done with it** leads to a dangling pointer
  - The pointer is no longer associated with the allocated memory and, when accessing it, the program may crash or overwrite other valid memory allocated meanwhile
- ◎ **Freeing memory repeatedly** leads to a double free
  - The result of a double-free is undefined, meaning that the memory-allocation library may exhibit different erroneous behaviors
- ◎ **Calling free() incorrectly** may lead to invalid frees
  - E.g., freeing other allocated memory, program crashes, ...

# Memory API

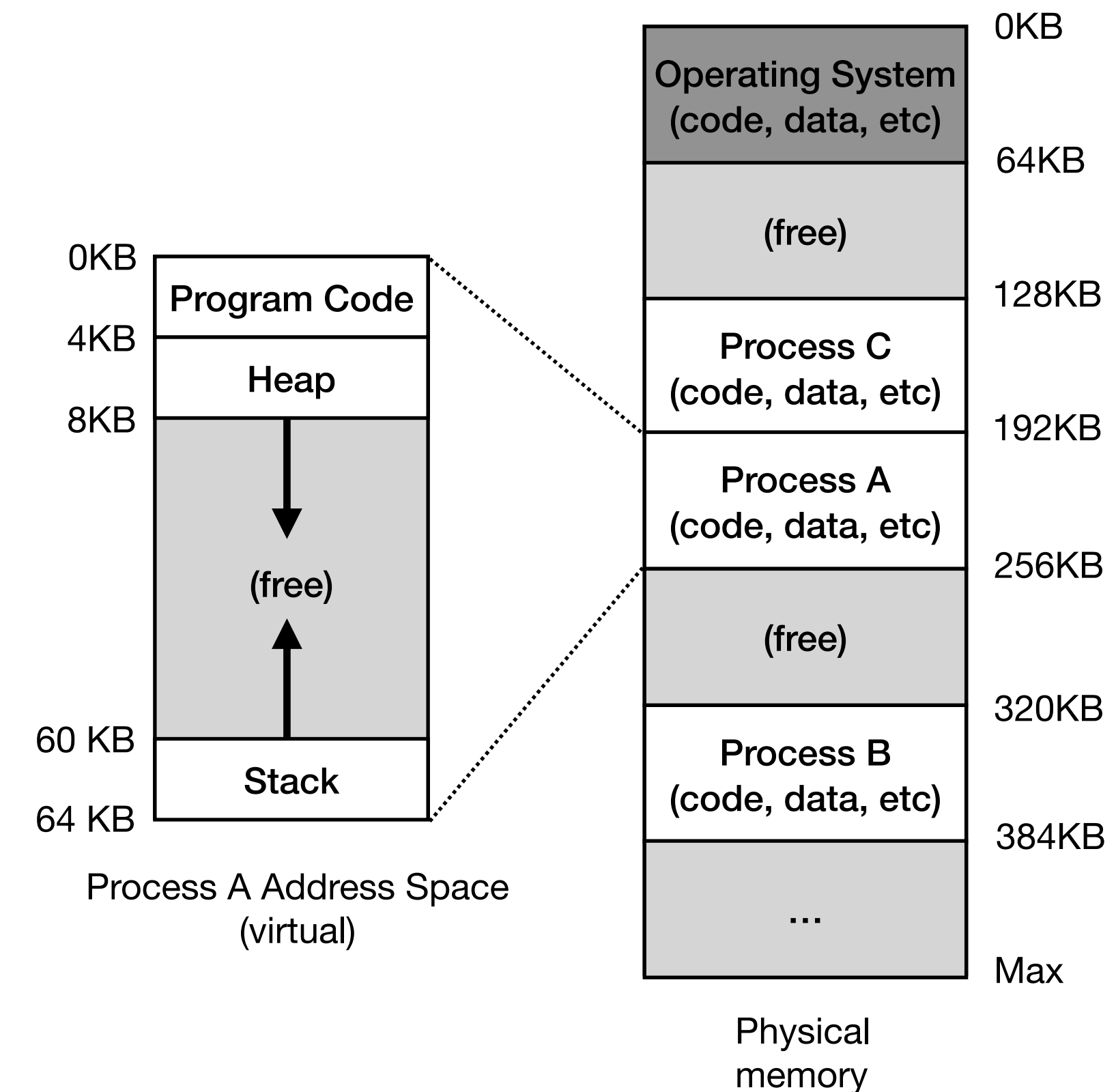
## System Calls

- ◎ *malloc()* and *free()* are not system calls, these are **library calls**
  - These are build on top of system calls like *brk* and *sbrk* that change the size of the heap (increase it or decrease it)
  - As a programmer **you should not use directly these system calls**, use *malloc()* and *free()* instead!
- ◎ *mmap()* is another call for obtaining memory from the OS
  - For now take a look at the man page, we will discuss this call later on...

# Virtualizing Memory

## Virtual and physical addresses

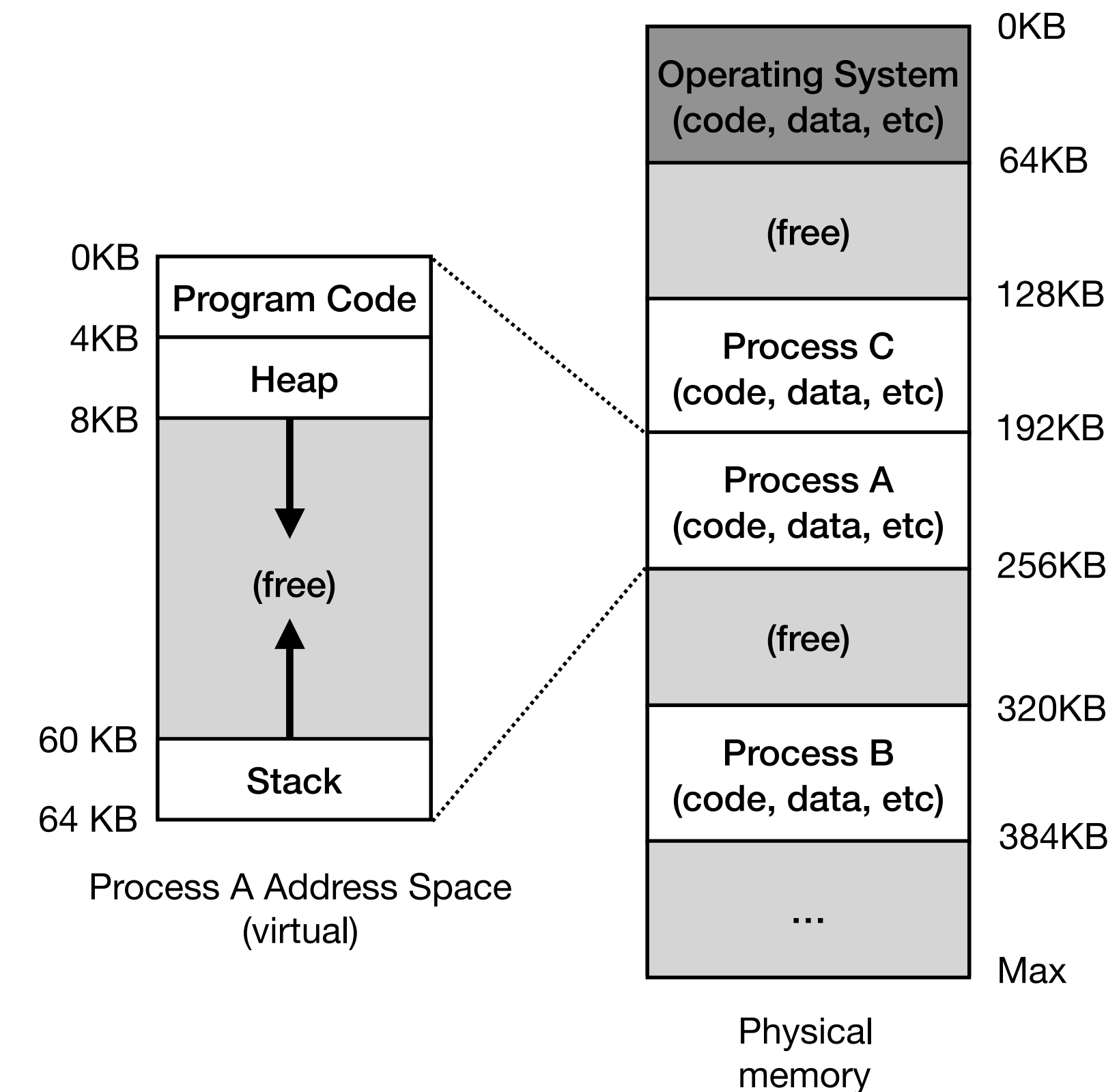
- Back to the address space... and how it is mapped to memory
- The address space of *Process A* is not really loaded into physical memory at addresses 0 to 64KB
  - When *Process A* accesses address 0 (**virtual address**), the request is mapped to address 192KB at main memory (**physical address**)
- **Note:** Address spaces are much larger in real systems
  - 32 bit ( $2^{32} = 4GB$ ) or 64 bits ( $2^{64} = 16EB$ )
  - Even larger than the physical memory at your computer...
- How does the OS build the abstraction of a **private**, **potentially large** address space for **multiple processes**?



# Virtualizing Memory

## Let's first define the OS Goals

- **Transparency<sup>1</sup>**: Processes are not aware that their memory is being virtualized. Each process behaves as if it has its **own private physical memory**
  - Addresses you see as a programmer (or user-level program) are virtual!
- **Efficiency**: Memory virtualization must be efficient in terms of **performance** (i.e., programs cannot run slower) and **space** (i.e., structures used by the OS cannot consume too much memory)
- **Protection**: Processes cannot access/modify memory regions from other processes or the OS (i.e., address spaces must be isolated)
  - **Isolation** is a key property for building reliable systems.



<sup>1</sup> **Transparent** in this context means that the illusion created by the OS should not be visible to running programs



# Address Translation

## Basic idea

- ◎ **Goal:** Create the **illusion** that each process has its own private memory
  - **Solution:** Each memory access (e.g., fetch, load, store) done by a process uses a **virtual address** that must be **translated**, by the hardware, into the corresponding **physical address** (i.e., actual location in memory)
- ◎ The hardware provides the low level mechanisms to efficiently translate addresses...
- ◎ The OS is responsible for instructing the hardware to do the correct translations, managing memory, and protecting processes from each other



# Address Translation

## Some initial assumptions...

- ◎ Let's define some assumptions to reason about memory virtualization
  1. A process **address space** is **placed contiguously in physical memory**
  2. The **size** of an **address space** is **smaller** than the **size of physical memory**
  3. Each **address space** has **exactly the same size**
  
- ◎ Most of these assumptions are unrealistic...  
...we will refined them as we move along

# Base and Bounds (aka Dynamic relocation)

## Base Register

- The processor (CPU) has two additional registers, the **base** and the **bounds**
  - The **Memory Management Unit (MMU)** is the component of the CPU responsible for holding these registers, and helping with address translation
- The **memory virtualization illusion** created by the OS consists of
  - Each program is written and compiled as if it is loaded at address 0
  - When the program starts running, the OS decides where to load it at physical memory i.e., sets the **base** register to the physical memory address
  - For each memory access made by the program, **virtual addresses** are translated into **physical ones**, by the processor, in the following way

$$\textit{Physical address} = \textit{Virtual address} + \textit{Base}$$

# Base and Bounds (aka Dynamic relocation)

## Bounds register

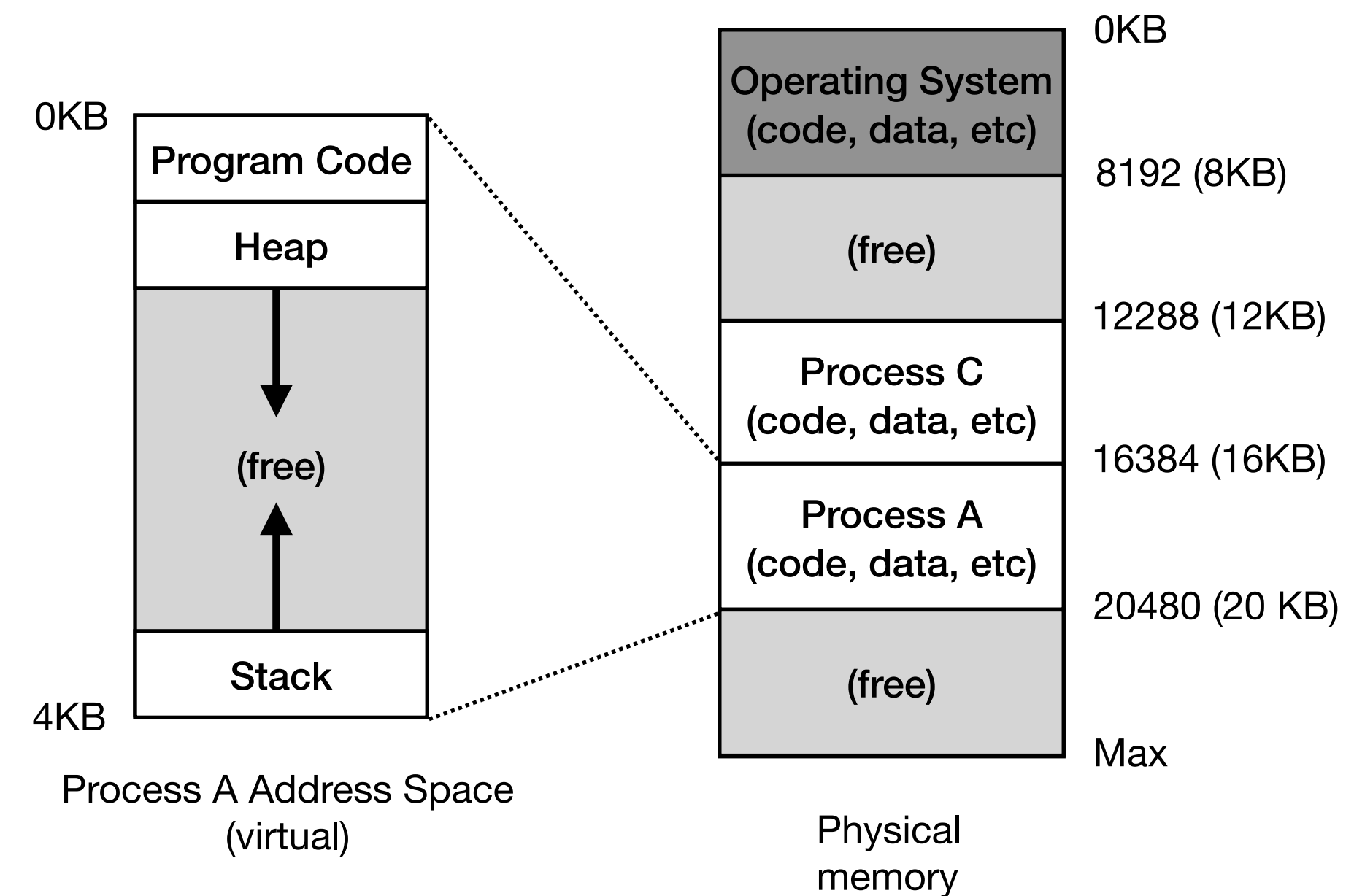
- ◎ The bounds register helps **protecting** the address space limits of each process
  - The processor first checks if the memory access is within **bounds**
  - If not, an **out of bounds exception** (trap!) is generated.
    - The OS must provide exception handlers for this trap
- ◎ Since *address translation* is done at **runtime**, a process's address space can be **dynamically relocated** (moved) while the process is running
  1. The OS must deschedule the process
  2. Copy its address space to another location at physical memory
  3. Update base and bounds registers
  4. Run the process
- ◎ Base and bounds registers must be **saved and changed** when **processes are switched**

# Base and bounds (aka Dynamic relocation)

## Example

- **Example:** *Process A* has an address space of 4096 bytes (4KB) loaded at physical address 16384 (16 KB)
- If the virtual address value is **higher than the bounds register or negative** (4096), the access is **out of bounds**
  - Otherwise, add the base (16384) to the virtual address to get the physical one

Virtual Address		Physical Address
0	→	16384 (16 KB)
1024	→	17408 (17 KB)
3000	→	19384
4400	→	FAULT (20784 IS OUT OF BOUNDS)



# Limited Direct Execution With Dynamic Relocation

## OS (Kernel Mode)

- create  $PCB_A$
- find a free memory slot at the **free list**<sup>1</sup>
- set base/bound registers
- return-from-trap

- handle trap
  - decide to terminate  $Process_A$
  - deallocate  $Process_A$  memory, add it to the free list
  - remove  $PCB_A$

## Hardware

- restore regs of  $Process_A$
- move to user mode
- jump to  $Process_A$  PC

- ensure address is in bounds
- translate virtual address into physical address

- Ensure address is in bounds (**out of bounds!**)
- move to kernel mode
- jump to trap handler

## Process (User Mode)

- run *main()* ...
- memory load/store instruction*

- Continue execution*
- bad memory load/store instruction*

If these concepts seem confusing, look again at the *CPU Virtualization Mechanisms* slides!

<sup>1</sup> The OS must track free slots at physical memory to allocate the address space of new processes. We call this structure a **free list**.



# Limited Direct Execution With Dynamic Relocation

## OS (Kernel Mode)

- create  $PCB_A$
- find a free memory slot at the free list
- set base/bound registers
- return-from-trap

- handle interrupt
- call *switch()* routine**
  - save  $PCB_A^1$
  - restore regs from  $PCB_B^1$
  - switch to kernel stack<sub>B</sub>

## Hardware

- restore regs of Process<sub>A</sub>
- move to user mode
- jump to Process<sub>A</sub> PC

- timer interrupt**
- save regs of Process<sub>A</sub><sup>1</sup>
- move to kernel mode
- jump to interrupt handler

- restore regs of Process<sub>B</sub>
- move to user mode
- jump to Process<sub>B</sub> PC

## Process (User Mode)

- run *main()* ...
- do stuff...*

- Process B*
- ...

If these concepts seem confusing, look again at the *CPU Virtualization Mechanisms* slides!

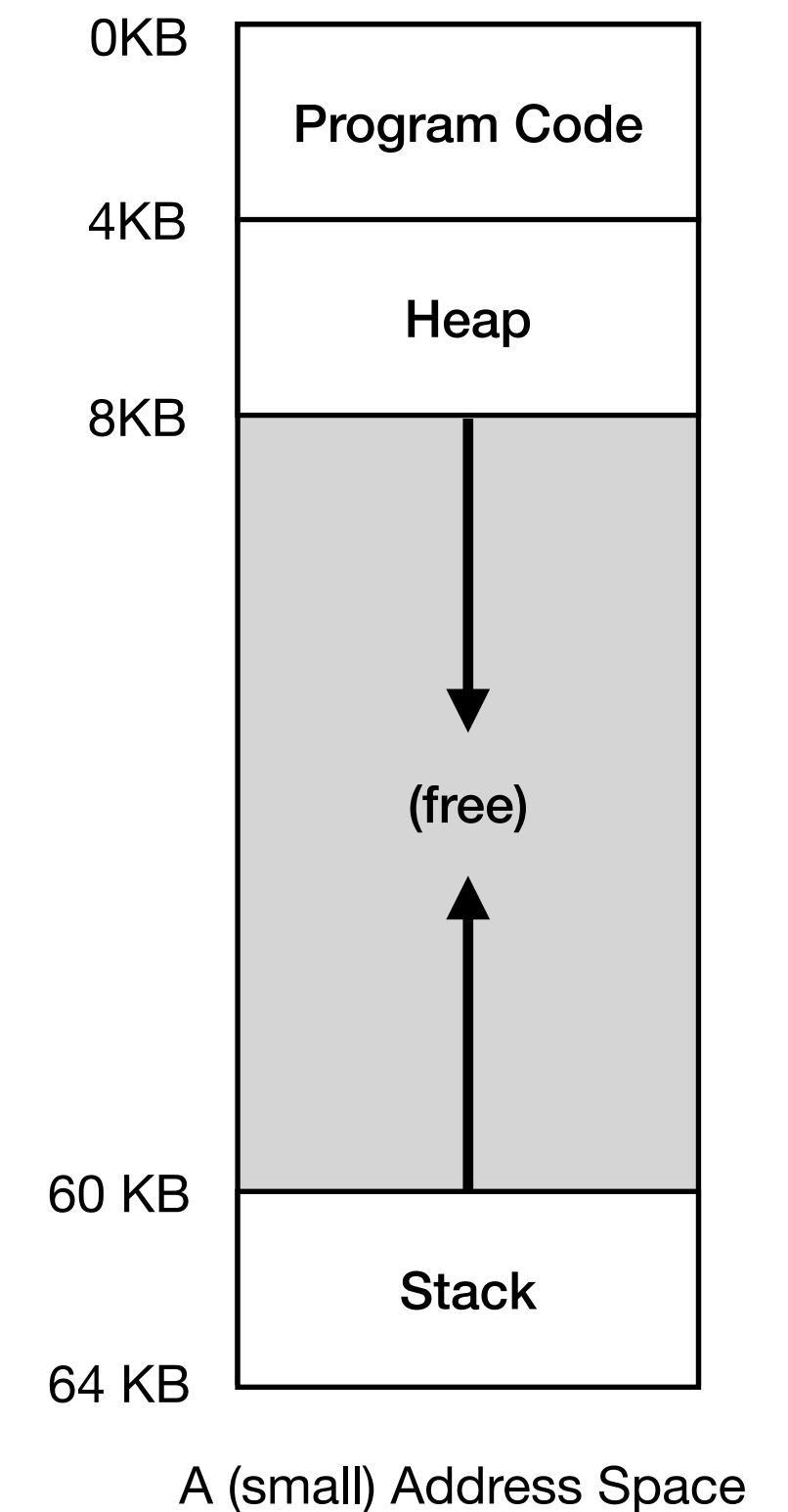
<sup>1</sup> Including base and bounds!



# Internal Fragmentation

## Heap and Stack

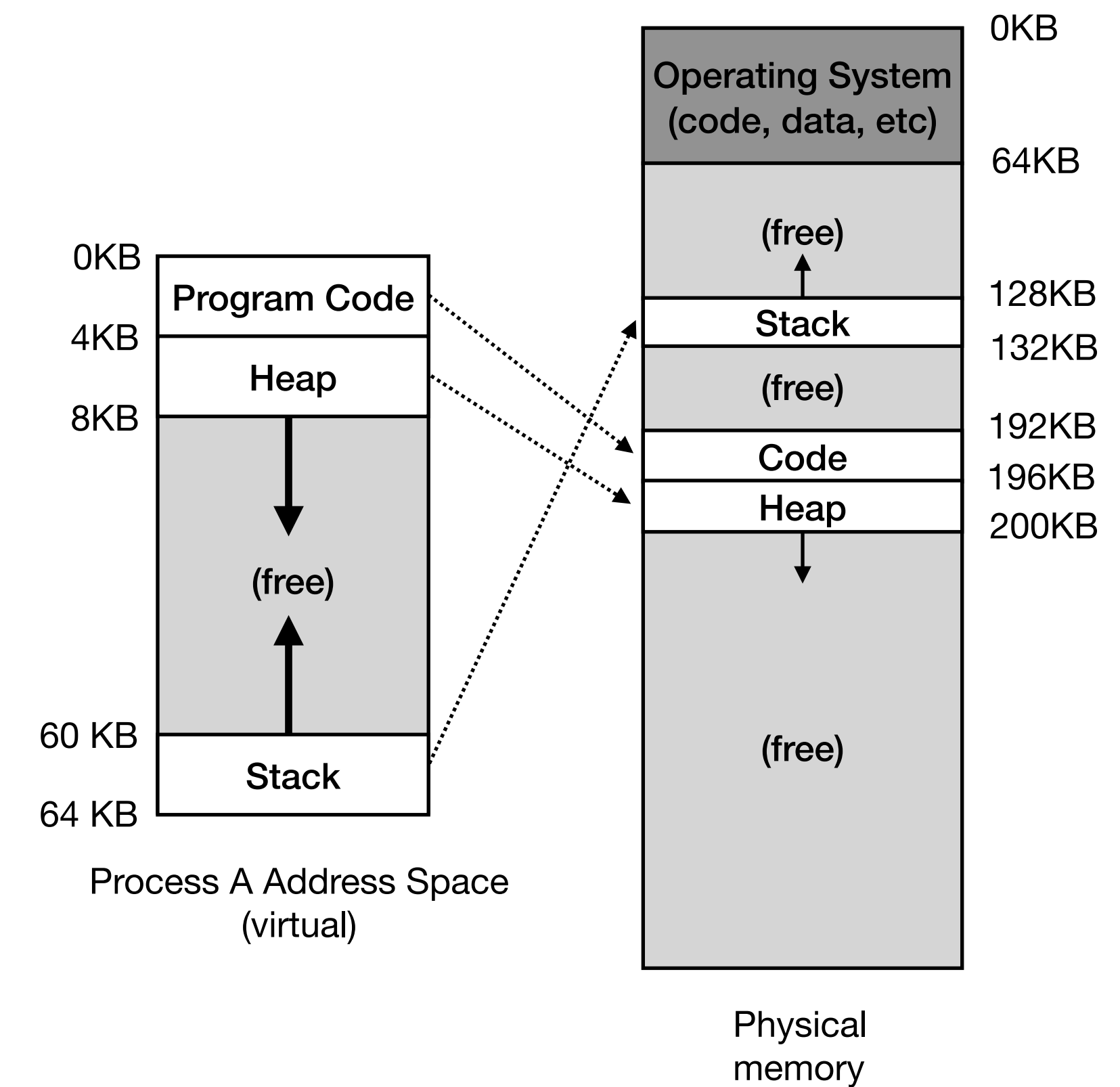
- © There is a potentially large **unused** memory chunk between the heap and stack that is still taking physical memory
  - E.g., a 32 bit address space requires 4 GB of main memory (while many programs may only actually use some MBs...)
- © In other words, placing the address space contiguously in physical memory leads to **internal fragmentation** i.e., wasted memory inside the address space of a process
- © How can the OS reduce this internal fragmentation?



# Segmentation

## Generalized Base/Bounds

- Let's relax *Assumption 1* - The process's **address space** is **placed contiguously in physical memory**
- Each logical **segment** of the address space (e.g., code, heap, stack) is placed at a different physical memory region
  - This technique, named **segmentation**, can be “easily” implemented if there are several base and bounds registers, one per logical segment
- Each segment can be **relocated** across memory as the system runs
- **Context switching** now requires saving and restoring all these registers



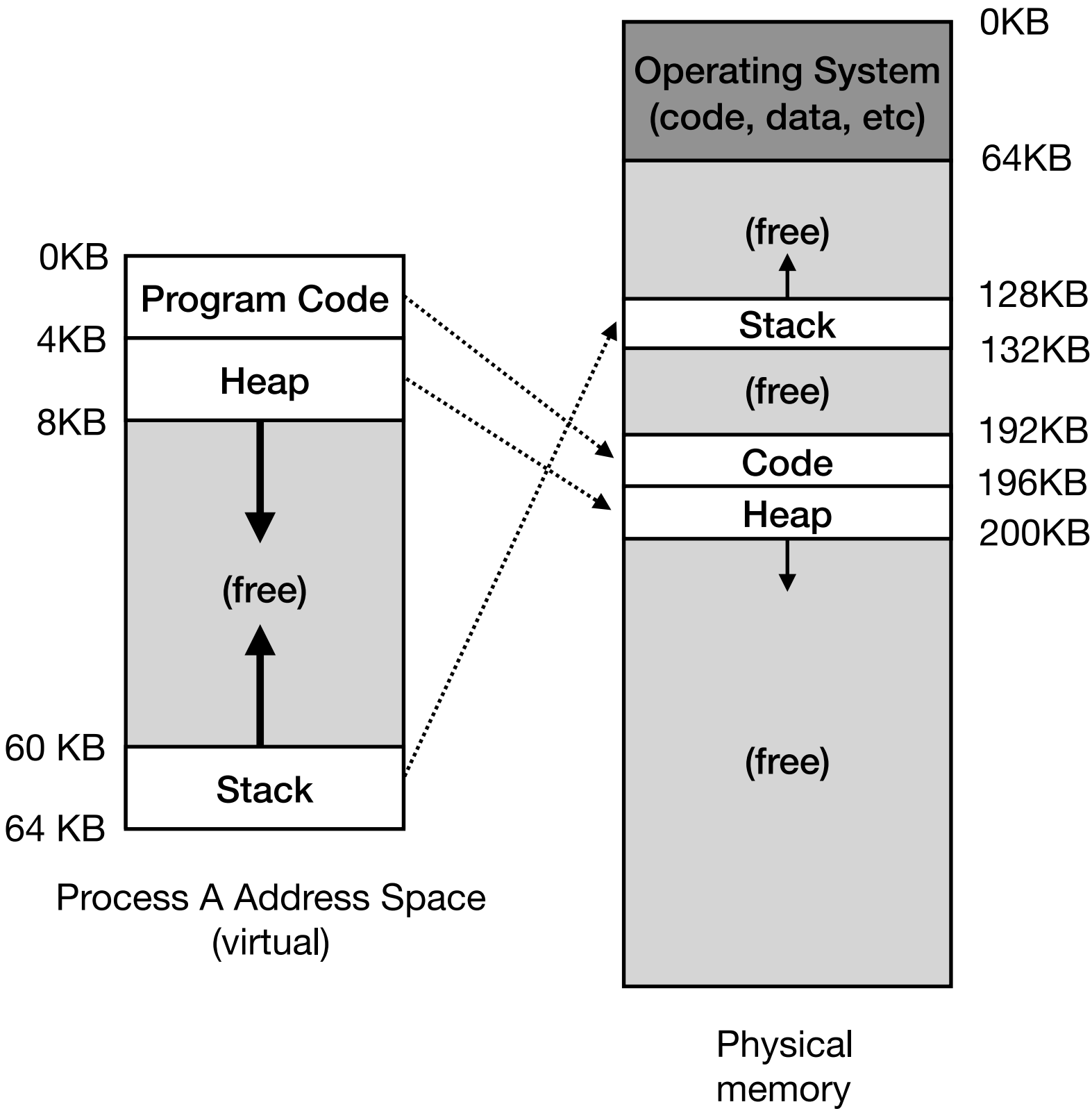
# Segmentation

## Examples

● **Example1:** Request to **virtual address** 100 (code segment)

*Physical address = 196 608 (base) + 100 (offset) = 196 708*

Segment	Base Register	Bounds Register
CODE	196 608 (192KB)	4096 (4KB)
HEAP	200 704 (196KB)	4096 (4KB)
STACK	135 168 (132KB)	4096 (4KB)



# Segmentation

## Examples

● **Example 1:** Request to **virtual address** 100 (code segment)

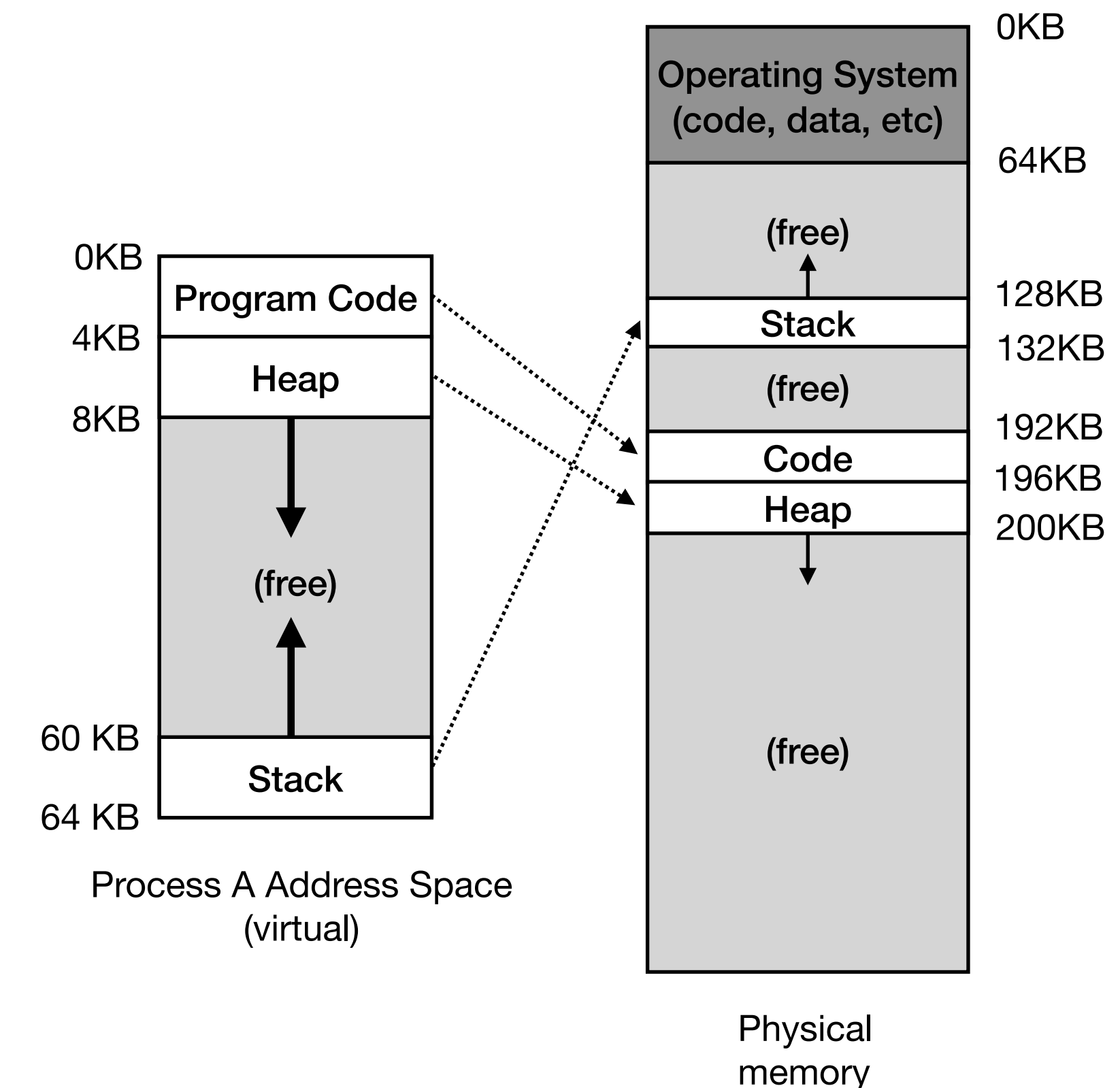
*Physical address = 196 608 (base) + 100 (offset) = 196 708*

● **Example 2:** Request to **virtual address** 5120 (heap segment)

- If we just add the virtual address to the base register of the heap one gets

*Physical address = 200 704 + 5120 = 205 824 (201KB - **wrong address!**)*

Segment	Base Register	Bounds Register
CODE	196 608 (192KB)	4096 (4KB)
HEAP	200 704 (196KB)	4096 (4KB)
STACK	135 168 (132KB)	4096 (4KB)



# Segmentation

## Examples

● **Example 1:** Request to **virtual address** 100 (code segment)

*Physical address = 196 608 (base) + 100 (offset) = 196 708*

● **Example 2:** Request to **virtual address** 5120 (heap segment)

- If we just add the virtual address to the base register of the heap one gets

*Physical address = 200 704 + 5120 = 205 824 (201KB - **wrong address!**)*

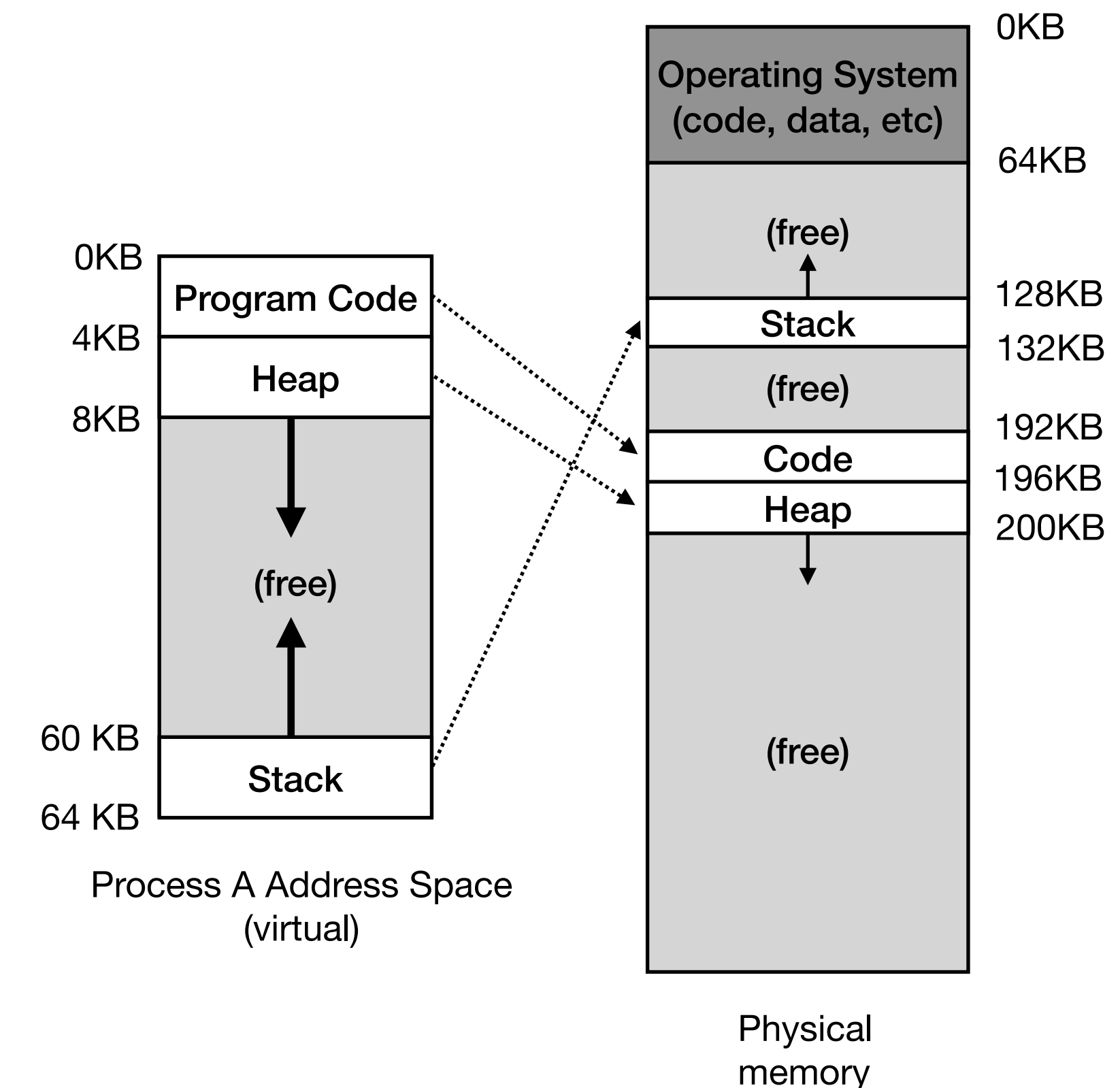
- First we extract the *offset* into the heap

*Offset = 5120 (virtual address) - 4096 (begin of heap's virtual address) = 1024*

- Then we calculate the physical address

*Physical address = 200 704 (base) + 1024 (offset) = 201 728 (197KB)*

Segment	Base Register	Bounds Register
CODE	196 608 (192KB)	4096 (4KB)
HEAP	200 704 (196KB)	4096 (4KB)
STACK	135 168 (132KB)	4096 (4KB)





# Segmentation

## Examples

● **Example 1:** Request to **virtual address** 100 (code segment)

*Physical address = 196 608 (base) + 100 (offset) = 196 708*

● **Example 2:** Request to **virtual address** 5120 (heap segment)

- If we just add the virtual address to the base register of the heap one gets

*Physical address = 200 704 + 5120 = 205 824 (201KB - **wrong address!**)*

- First we extract the *offset* into the heap

*Offset = 5120 (virtual address) - 4096 (begin of heap's virtual address) = 1024*

- Then we calculate the physical address

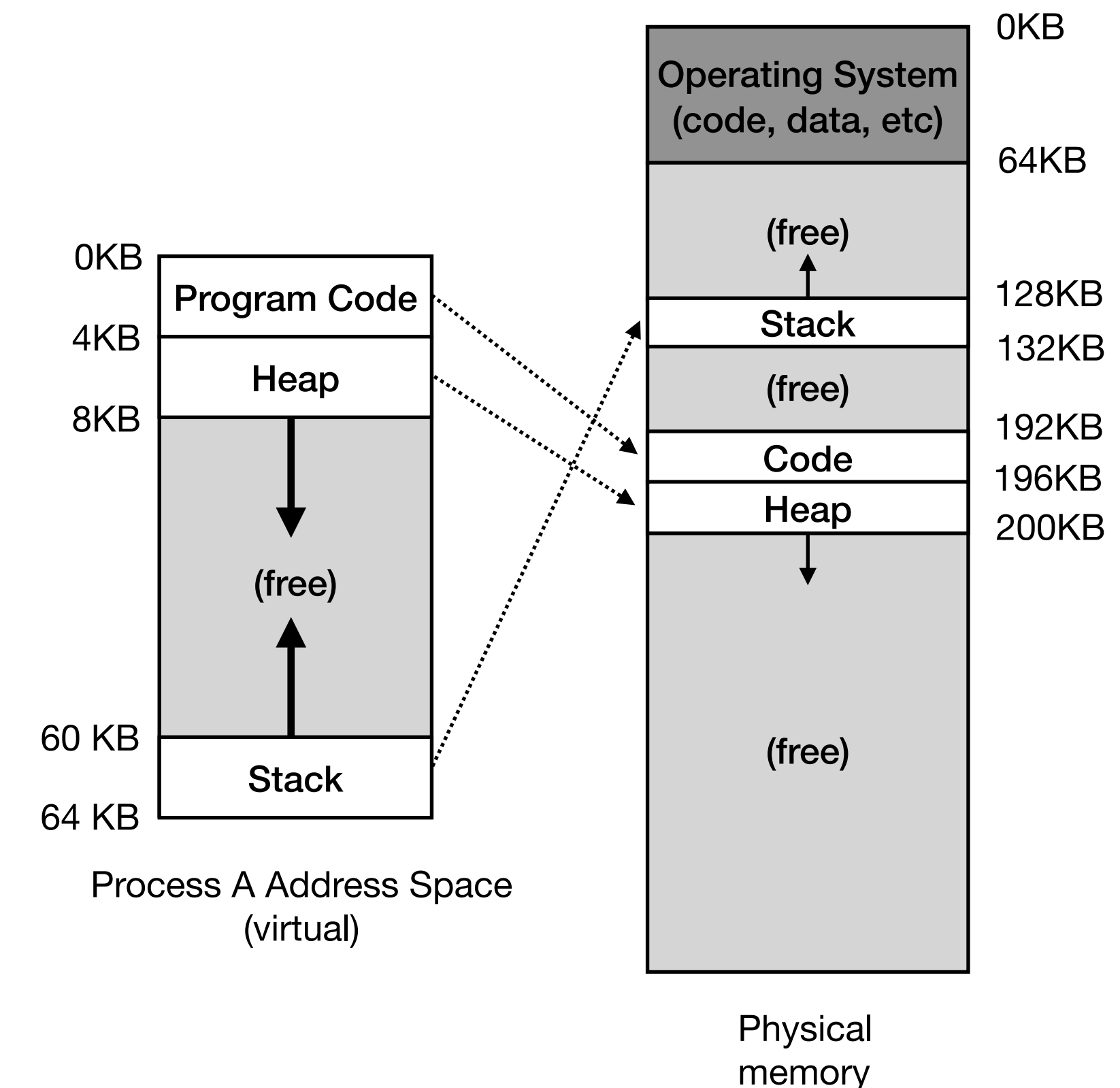
*Physical address = 200 704 (base) + 1024 (offset) = 201 728 (197KB)*

● **Example 3:** Request to virtual address 9216 (9KB) (heap segment)

*Offset = 9216 - 4096 = 5120 (higher than 4096 - **out of bounds!**)*

- The hardware detects the address is out of bounds, traps into the OS, the latter terminates the offending process
- Now you know the origin of a **segmentation fault!**

Segment	Base Register	Bounds Register
CODE	196 608 (192KB)	4096 (4KB)
HEAP	200 704 (196KB)	4096 (4KB)
STACK	135 168 (132KB)	4096 (4KB)





# Segmentation

## Hardware-translation

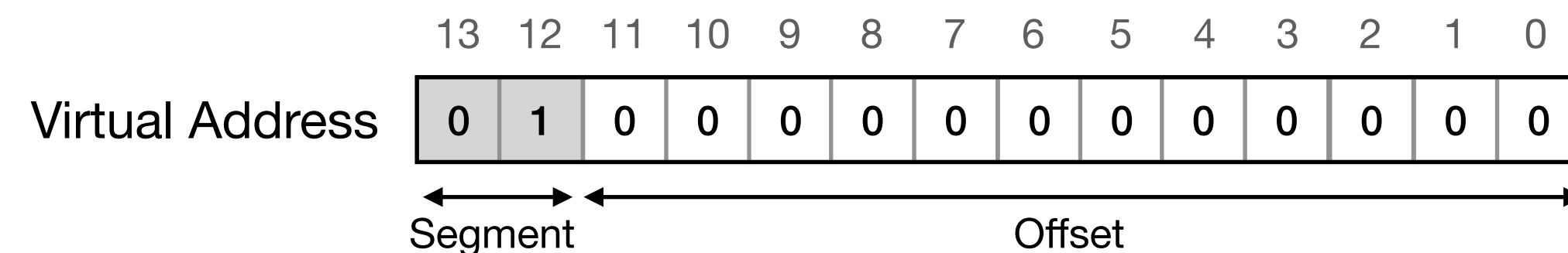
Segment	Base Register	Bounds Register
CODE (00)	196 608 (192KB)	4096 (4KB)
HEAP (01)	200 704 (196KB)	4096 (4KB)
STACK (10)	135 168 (132KB)	4096 (4KB)

● How does the hardware know the **segment** and **offset** of a virtual address?

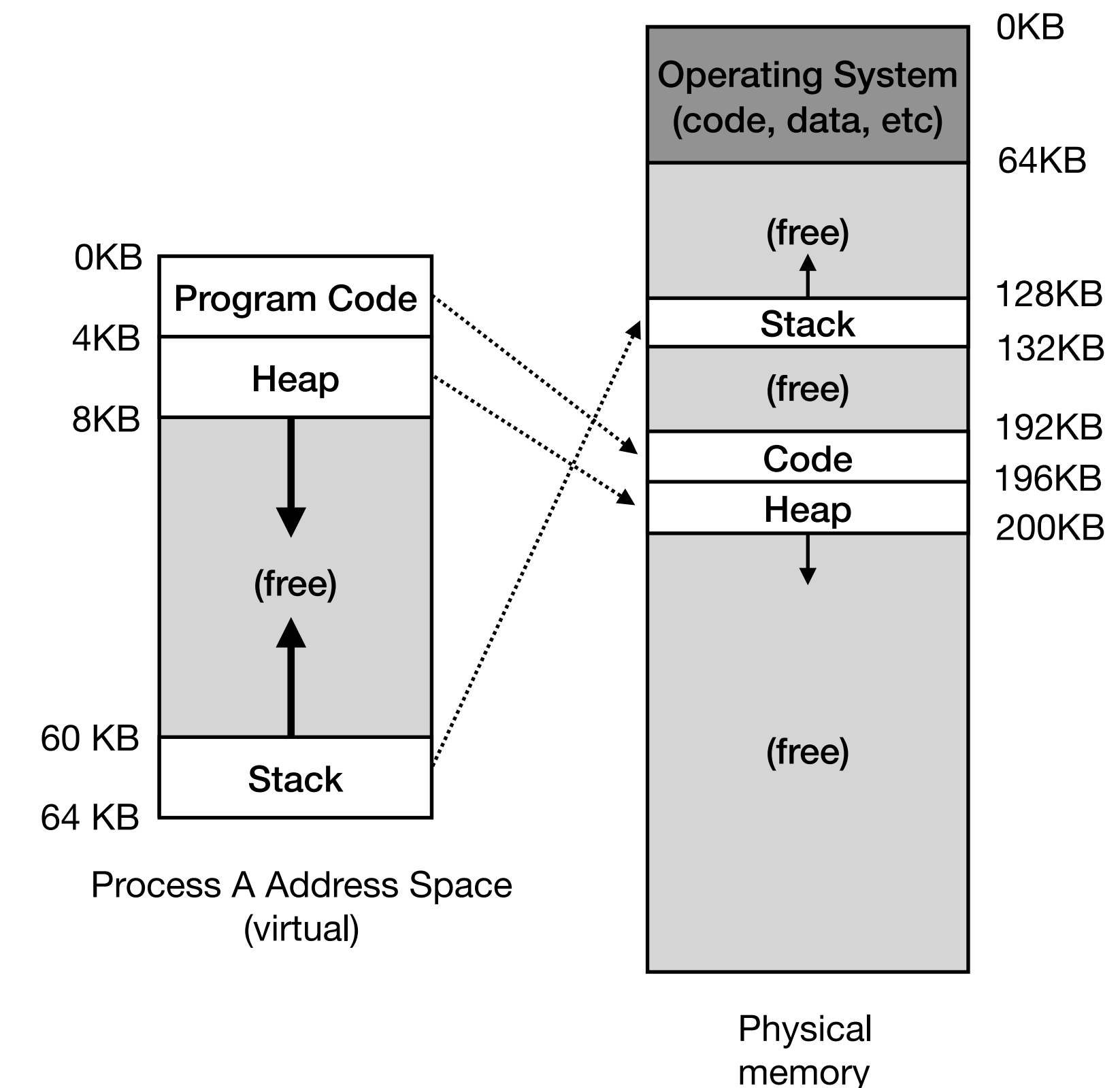
- **Solution:** use the top bits of the virtual address to identify the segment

● **Example:** considering a virtual address with 14 bits

- Access to the **heap** segment at **offset 0**



$$\text{Physical address} = 200\,704 \text{ (base)} + 0 \text{ (offset)} = 200\,704$$



# Segmentation

## Hardware-translation

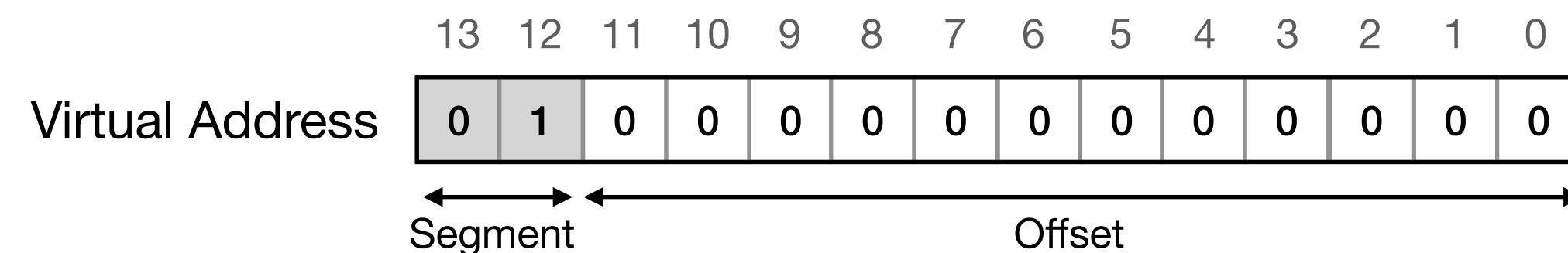
Segment	Base Register	Bounds Register
CODE (00)	196 608 (192KB)	4096 (4KB)
HEAP (01)	200 704 (196KB)	4096 (4KB)
STACK (10)	135 168 (132KB)	4096 (4KB)

● How does the hardware know the **segment** and **offset** of a virtual address?

- **Solution:** use the top bits of the virtual address to identify the segment

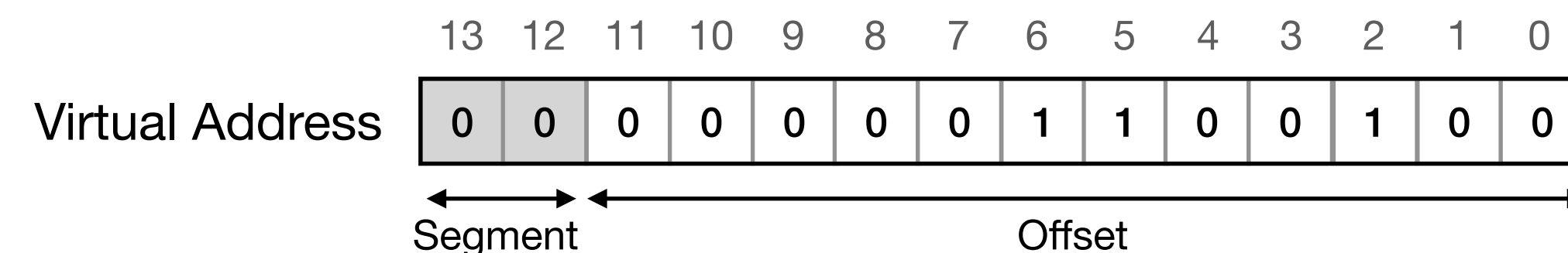
● **Example:** considering a virtual address with 14 bits

- Access to the **heap** segment at **offset 0**

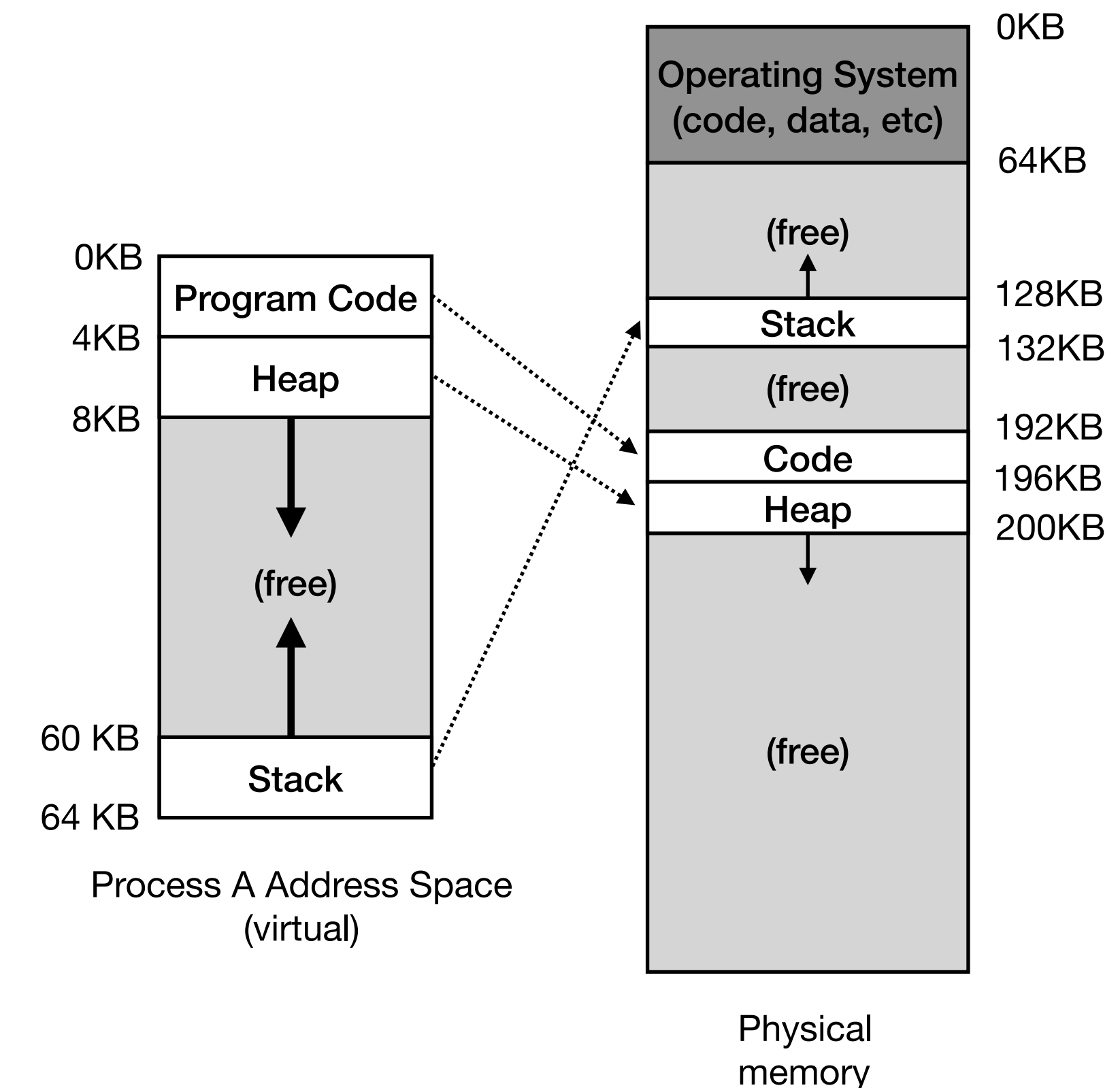


$$\text{Physical address} = 200\,704 \text{ (base)} + 0 \text{ (offset)} = 200\,704$$

● **Another example:** Access to the **code** segment at **offset 100**



$$\text{Physical address} = 196\,608 \text{ (base)} + 100 \text{ (offset)} = 196\,708$$



# Segmentation

## Stack translation

Segment	Base Register	Bounds Register	Grows Positive?
CODE (00)	192KB	4KB	1
HEAP (01)	196KB	4KB	1
STACK (10)	132KB	4KB	0

● Note that the **stack** starts at address 132 KB<sup>1</sup>

- The stack grows **backwards!**  
i.e., it goes from from physical address 132KB to 128KB in the example

● **Example:** Access to **virtual address 63KB**

- Let's calculate the offset by subtracting the base address to the virtual one

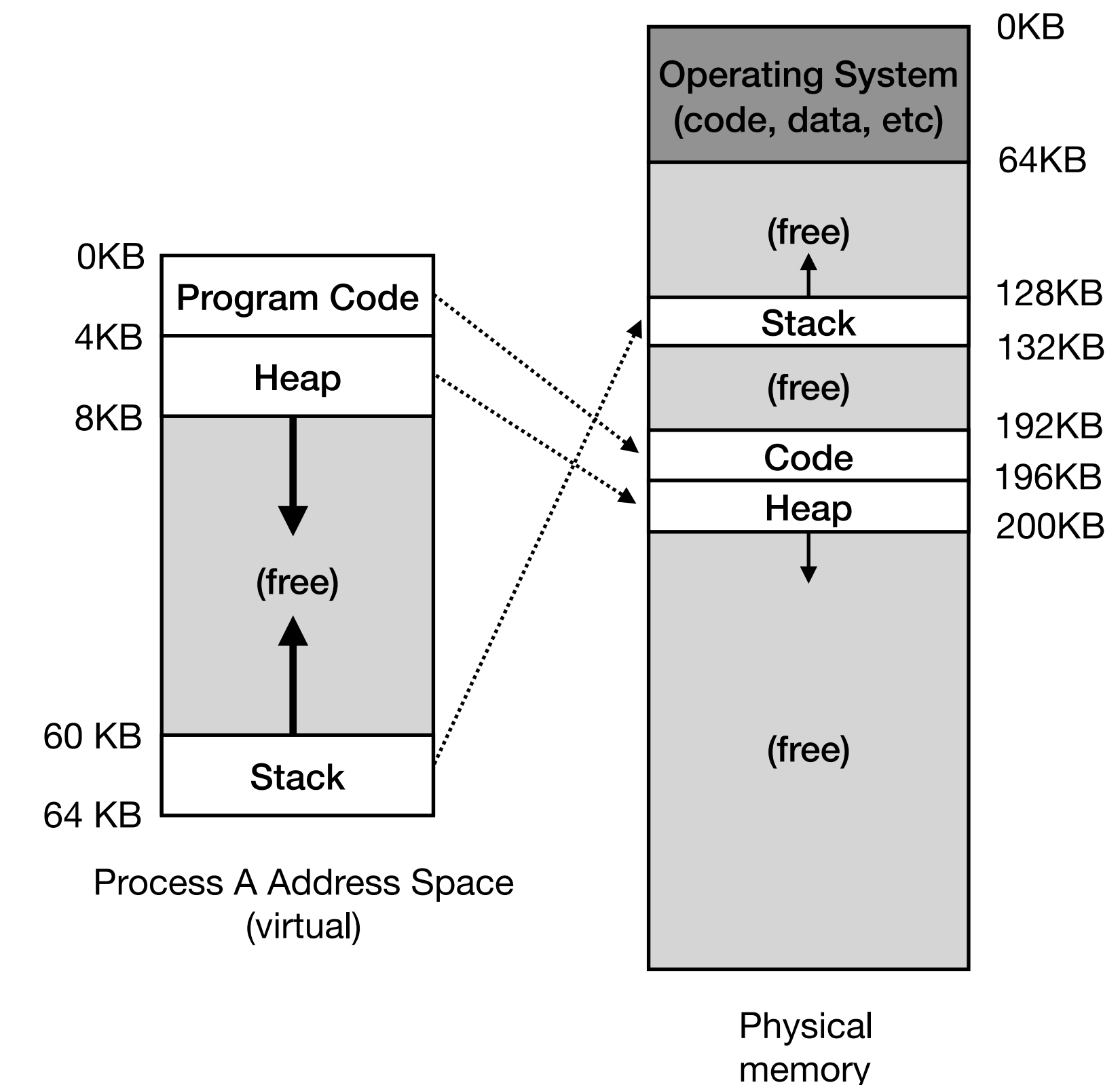
$$\text{Offset} = 63\text{KB (virtual address)} - 64\text{KB (begin of stack's virtual address)} = -1\text{KB}$$

- Now we get the physical address

$$\text{Physical address} = 132\text{KB (base)} - 1\text{KB (offset)} = 131\text{KB}$$

● The OS needs help from the hardware to translate stack addresses

- extra bit set to **1** if the segment grows in positive direction, **0** otherwise



<sup>1</sup> actually to be precise it starts at 132KB minus 1 byte

# Segmentation

## Sharing support

- With segmentation it becomes possible to **share segments** across multiple processes
  - E.g., one can share **code** segments from libraries used by multiple programs!
  - If the base registers for the code segments of *Processes A* and *B* point to the same physical address... one has a **shared segment**!
  - **Important:** Each process has its own registers! Sharing is attained by making these registers refer to the same physical address
- One needs **protection bits** (a little more support from the hardware)
  - The bits indicate what a process can do with each segment, i.e., read, write or execute (instructions)
  - By marking a segment as read-only, several processes can use it **without harming isolation**
- The hardware now also checks if the access to the segment has the right permissions, if not the hardware raises an exception and the OS deals with the offending process

Segment	Base Register	Bounds Register	Grows Positive?	Protection
CODE (00)	192KB	4KB	1	READ-EXECUTE
HEAP (01)	196KB	4KB	1	READ-WRITE
STACK (10)	132KB	4KB	0	READ-WRITE



# Segmentation

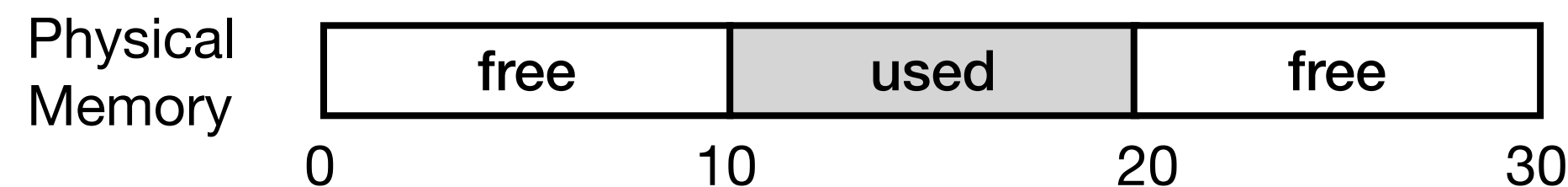
## Final remarks

- ◎ With segmentation we actually relaxed all our assumptions!
  1. The process **address space** is **placed contiguously in physical memory**
  2. The **size** of an **address space** is **smaller** than the **size of physical memory**
  3. Each **address space** has **exactly the same size**
- ◎ In our example, there are only three segments (**coarse-grained segmentation**)
  - It is possible to have smaller segments (**fine-grained segmentation**)
  - Such may require further support, such as a **segment table** (stored in memory), for translating the segments to their physical memory locations
- ◎ And one last issue
  - How does the OS **track and manage free physical memory slots** (chunks) to place segments?

# Managing Free Space

## External Fragmentation

- When a new address space is created, the OS must know where to place the corresponding segments (i.e., into the physical memory)
- Since segments vary in size, physical memory can quickly be filled with several little holes, leading to **external fragmentation**
  - **Example below:** if a process needs to allocate 20 bytes of memory, there isn't a contiguous chunk of free memory to do so. Memory is **fragmented externally** into two 10 byte chunks



- Stopping processes to reallocate and **compact** their memory is a costly strategy
  - The typical approach is instead to use a **free list management strategy**



# Managing Free Space

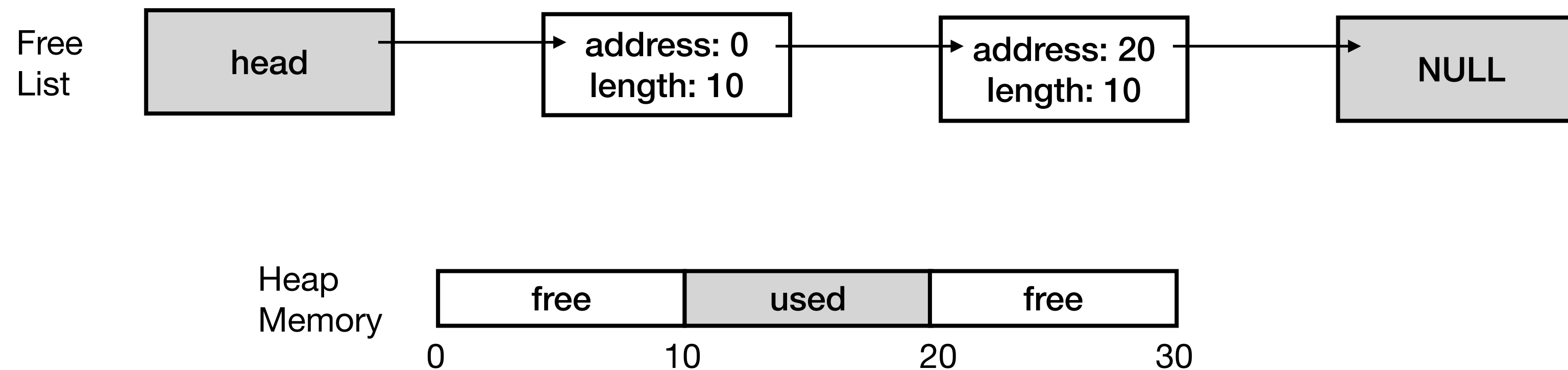
## Challenges

- Easy when the memory being managed is allocated with a fixed size
  - Stay tuned for our discussion on **paging**
- Harder, and more interesting, when one is using variable-sized units
  - Useful for **segmentation** (as discussed in the previous slide)
  - Also useful for user-level **heap** memory allocation with *malloc()* and *free()*  
*We will focus our next examples over this second use-case*
- **Challenges:**
  - Reducing external fragmentation
  - Efficiency (i.e., quickly finding a free chunk to allocate memory)

# Free List

## The structure

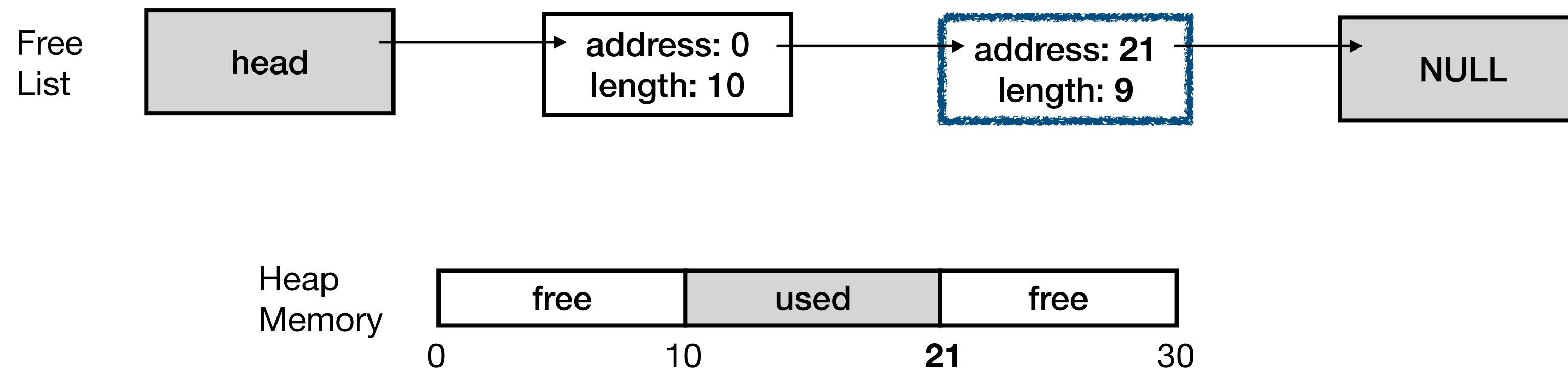
- A **free list** contains the **addresses** and **length** of free chunks at physical memory
- In the example below
  - A *malloc()* request bigger than 10 bytes cannot be served, returning *NULL* i.e., there isn't a contiguous chunk of memory larger than 10 bytes
  - A request for 10 bytes is served by one of the free chunks



# Free List

## Splitting mechanism

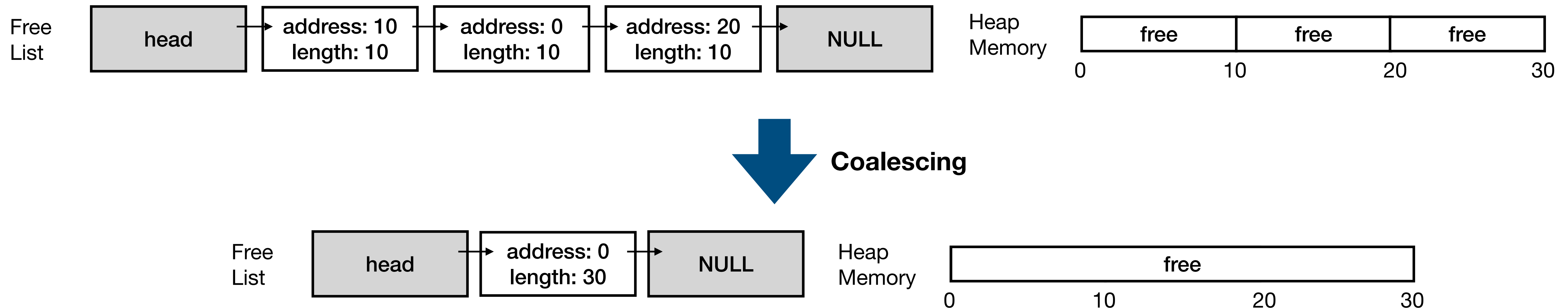
- What about a request to allocate 1 byte?
  - One of the chunks is split into two chunks  
That's why the technique is known as **splitting**...
- Lets assume that the **second free chunk is chosen**
  - *malloc()* returns the address 20, and the **second free chunk now starts at address 21**



# Free List

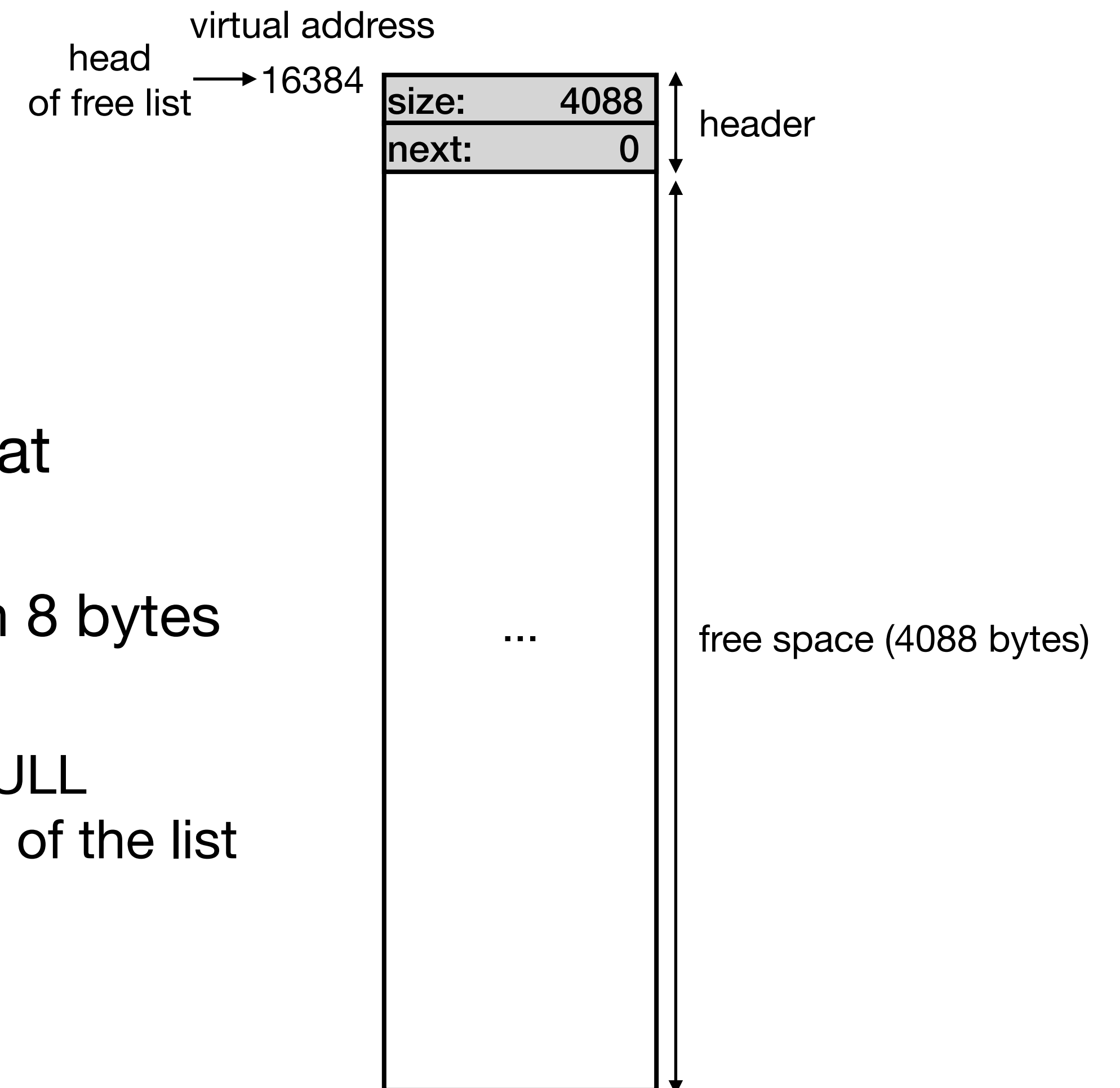
## Coalescing mechanism

- What if the program eventually frees all allocated memory and...
  - The free list ends up divided into three 10 byte chunks, not being able to serve *malloc()* requests larger than 10 bytes...
  - Another mechanism, named **coalescing**, is used to merge free contiguous chunks



# Embedding the Free List

## Into the heap space



● Where is the free list stored?

- Option 1: in **another physical memory region**
- Option 2: **embedded into the heap space**

● Let's assume an empty heap with 4KB, starting at **virtual address 16384 (16KB)**

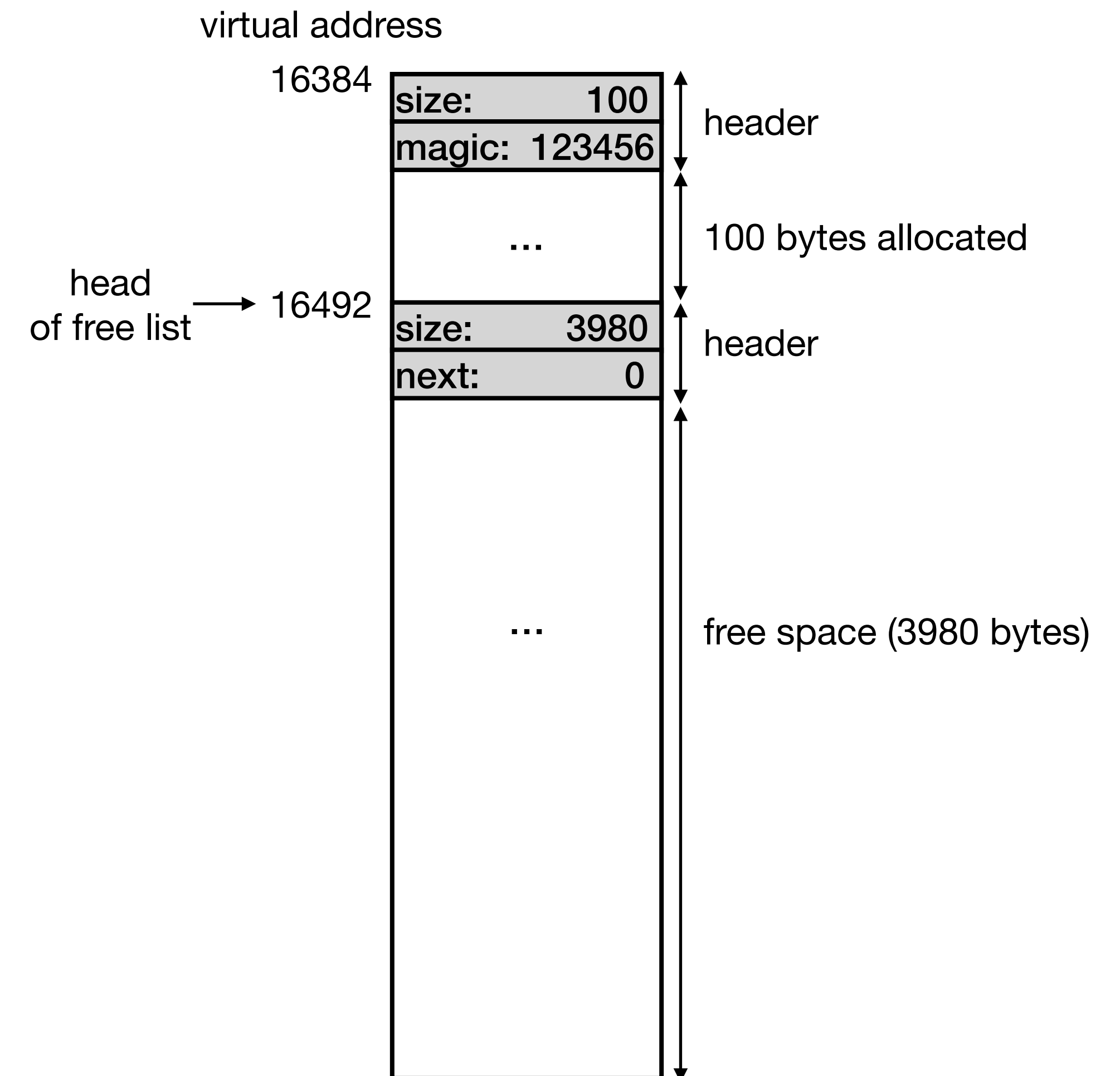
- We start with one chunk, including an header with 8 bytes
  - 4 bytes for the size of free space at the chunk
  - 4 bytes pointing to the next chunk. If the value is NULL (represented by the value 0), then we are at the end of the list
- Since the header also resides on heap space, the available free space is  $4096 - 8 = 4088$



# Embedding the Free List

## Into the heap space

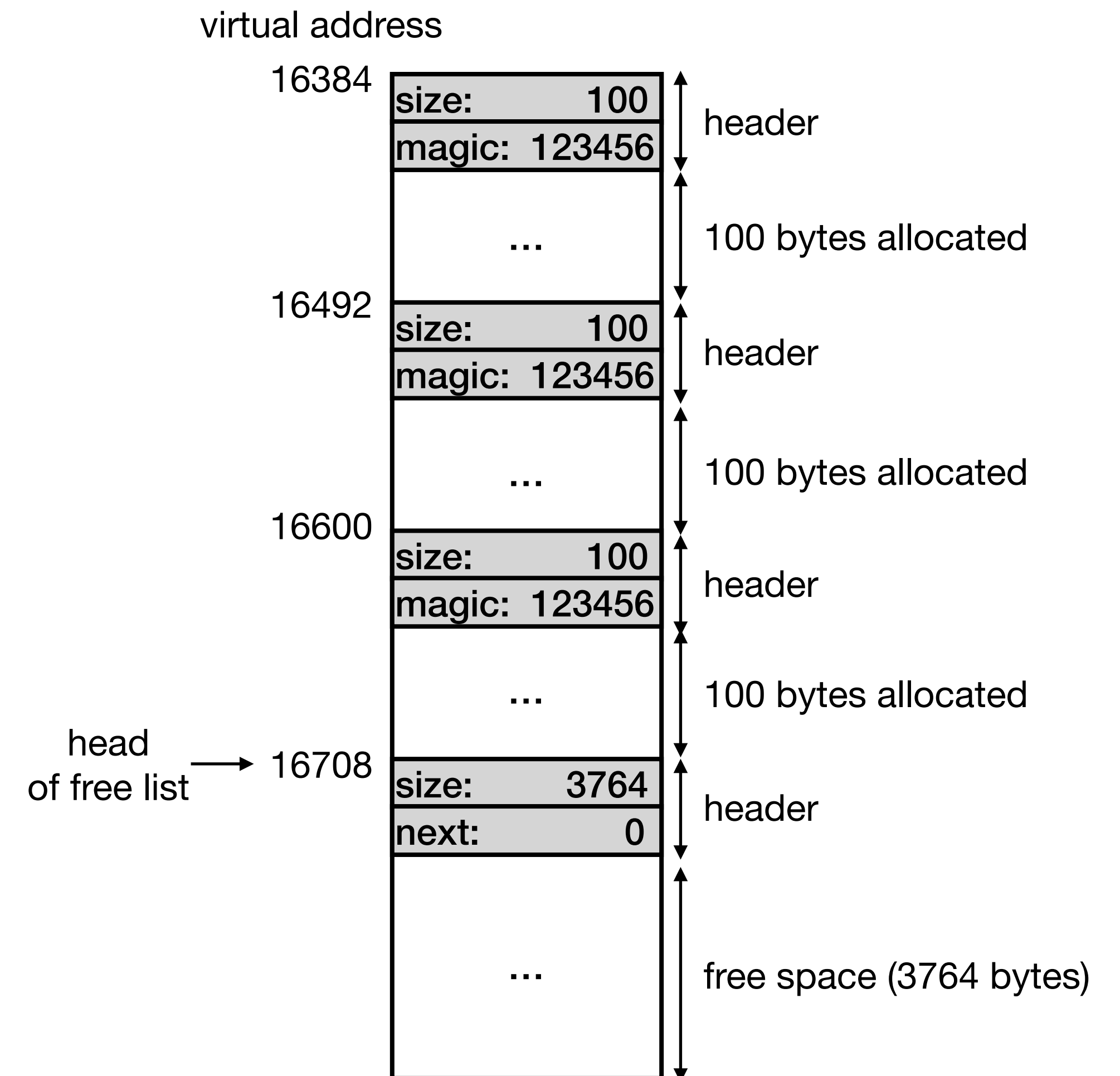
- A *malloc()* for 100 bytes is issued
- The original chunk is **split** into two
  - One chunk with 100 + 8 bytes (header)
    - Since the space is being used, the chunk is not indexed by the free list (no need for the **next** field)
    - The **magic number** field is used for integrity checking
  - Another chunk with 3980 + 8 bytes (header)
- The **head pointer** (beginning of the free list) is moved to the beginning of the second header



# Embedding the Free List

## Into the heap space

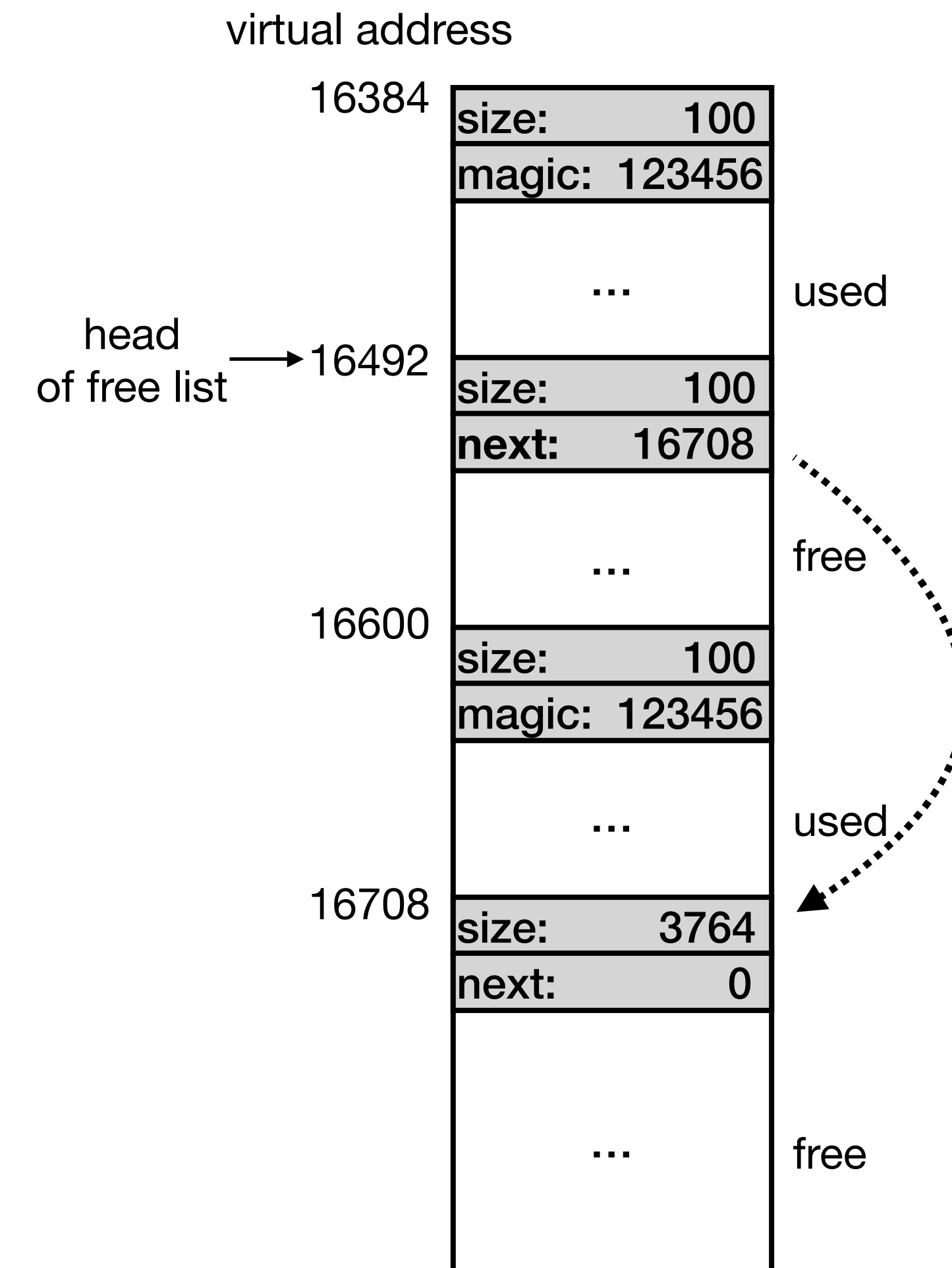
- Two more *malloc()* operations, each requesting 100 bytes, are issued
- Our heap ends up with **three allocated** 108 bytes chunks and a **single chunk** (3772 bytes) **with free space**
- What if *free()* is called for a given chunk?



# Embedding the Free List

## Into the heap space

- *free()* is called for address 16492 (second chunk)
  - **Note:** *free()* does not require the size of memory to be free as argument, because the information is already at the header :)
- Assuming one inserts free chunks at the **beginning** (head) of the **free list**
  - Head now points to the recently freed chunk (address 16492)
- What if *free()* is called for the other two chunks?



# Embedding the Free List

## Into the heap space

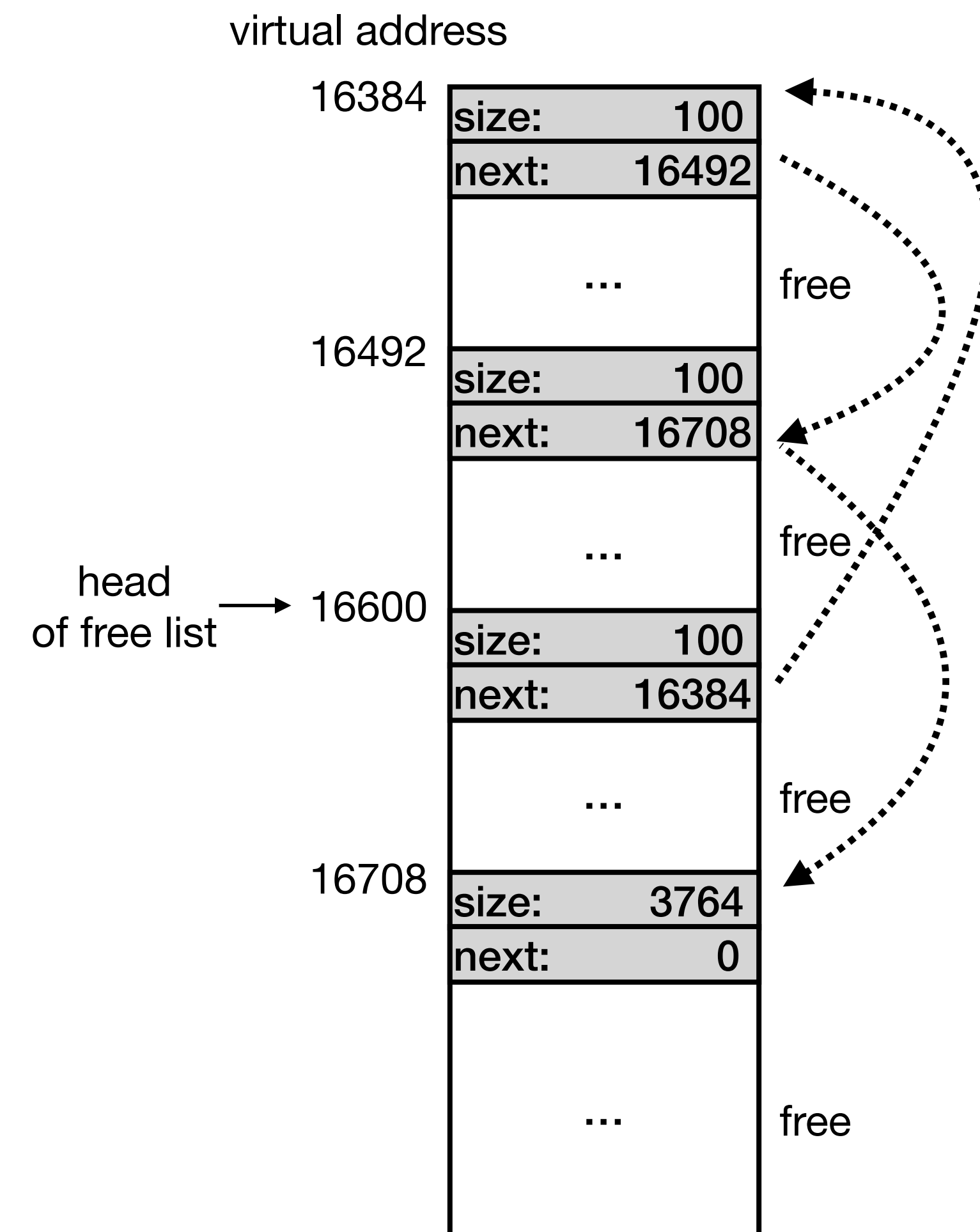
● *free()* is called first for address 16384 and then for address 16600

● All elements are now free

- The head now points to address 16600 (the last chunk added to the free list)

● **Issue:** The free list now has several chunk fragments

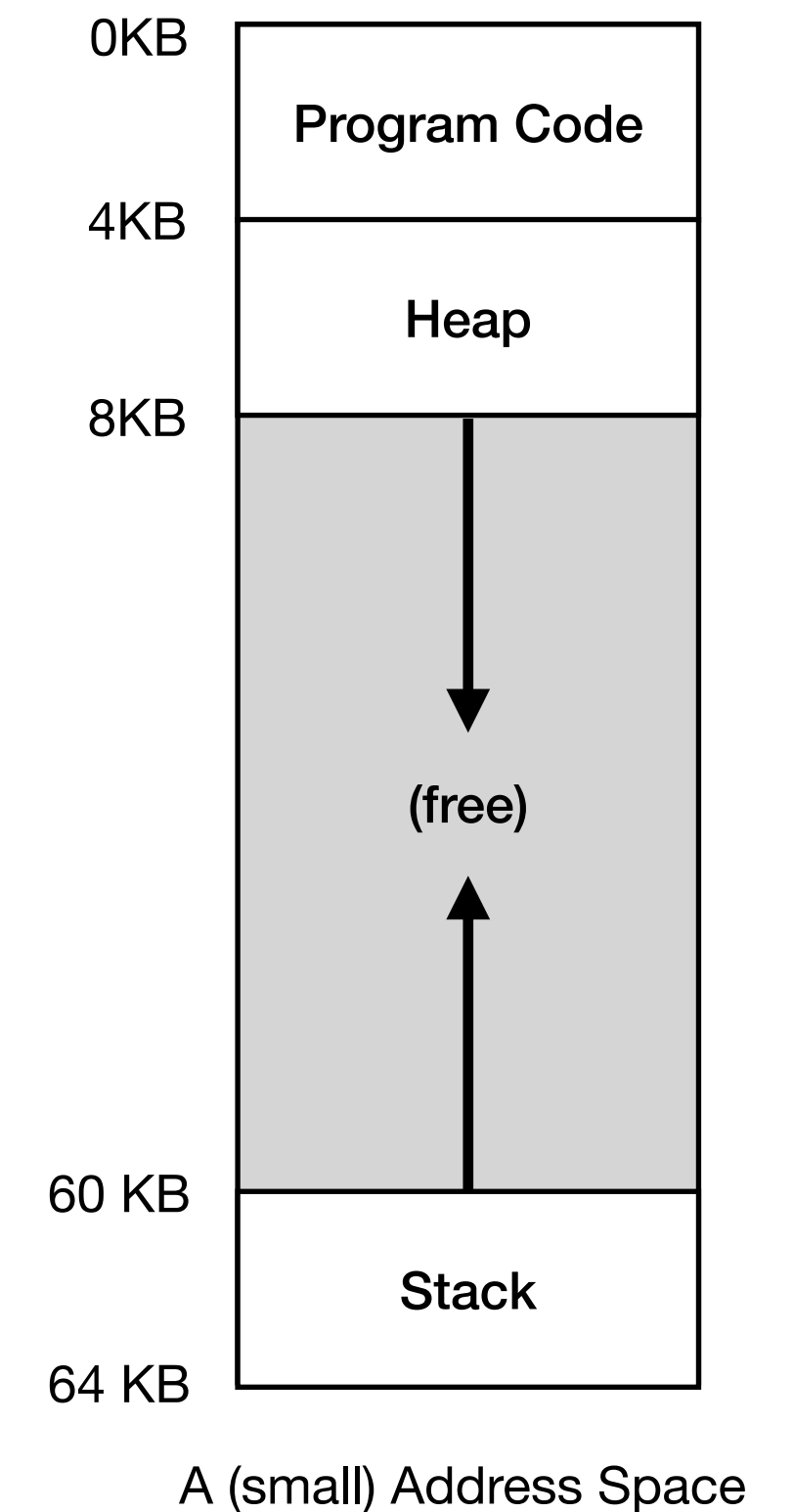
- Ideally one would **coalesce** the free list by merging adjacent free chunks into a single one
- When finished the heap would have again a single and large free chunk



# Managing Free Space

## Troubleshooting

- What do do if there is no space at the heap?
  - Option 1: the malloc **fails**...
  - Option 2: **grow the heap**  
(start with a small one and eventually increase its size...)
    - This is where the memory allocator does a system call (e.g., sbrk)

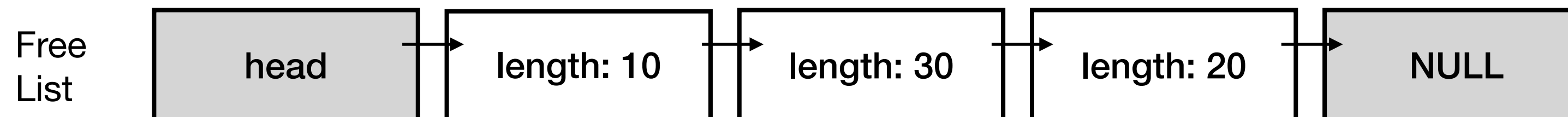




# Managing Free Space

## Strategies

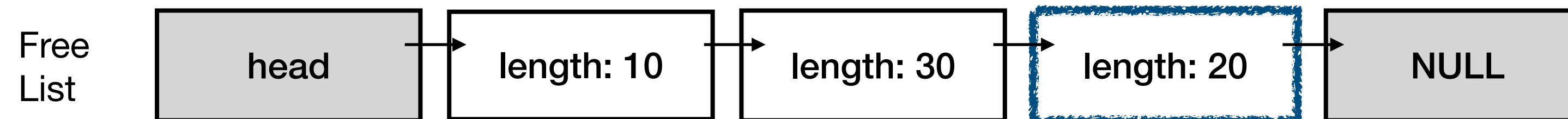
- **Challenge:** which element from the **free list** should the memory allocator choose to serve a **malloc()** for 15 bytes?
  - Let's just focus on the size available (length) at each element of the list
  - Can you think of strategies?



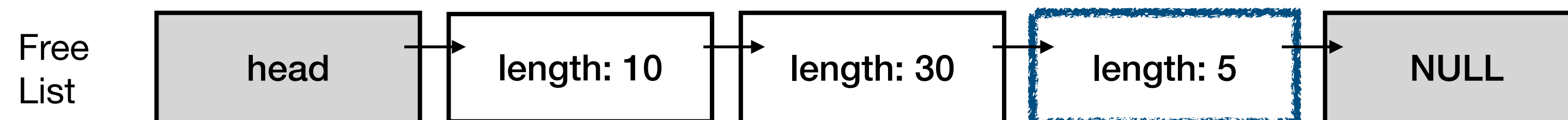
# Strategies

## Best fit

- ◎ Go through the list and find the **smallest chunk that fits the requested size**
  - The idea is simple, find the free chunk that best fits the user request to avoid waisting memory
  - **Example below:** the best fit strategy would go through the full list and choose length 20



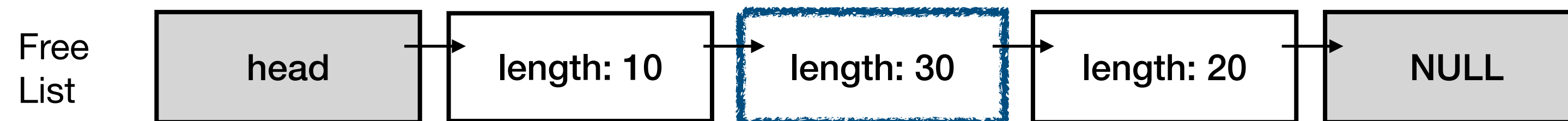
- ◎ The result after **splitting** would be



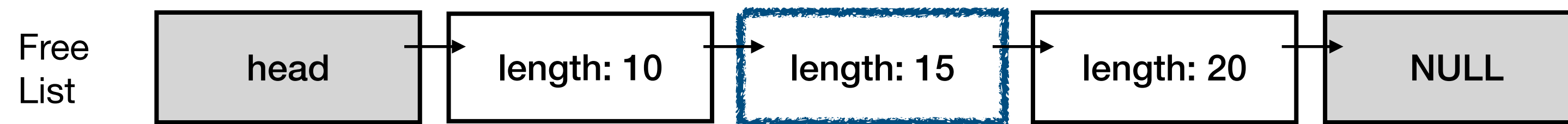
# Strategies

## Worst fit

- Go through the list and find the **largest chunk that fits the requested size**
  - Opposite idea to best fit. The goal is to use large chunks so that the remaining free space (after splitting) is still usable by other allocation requests.
  - Studies show that, in fact, the worst fit strategy leads to excess external fragmentation
  - **Example below:** the worst fit strategy would go through the full list and choose length 30



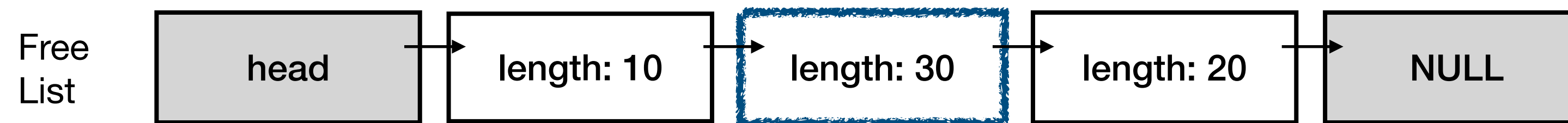
- The result after **splitting** would be



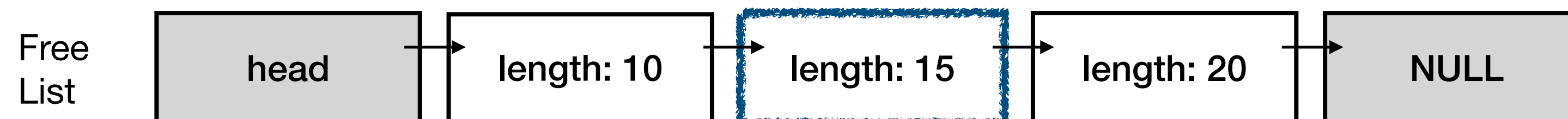
# Strategies

## First fit

- ◎ Best and Worst fit strategies requires going through the full list to find a candidate.  
This may take some time...
- ◎ **First fit** looks for the **first free chunk that is large enough to accommodate the request**
  - Faster to find a candidate... but can pollute the beginning of the free list with several small free chunks (leading to **severe external fragmentation** of the first chunks at the free list)
  - **Example below:** the first fit strategy would choose length 30, without going through the full list



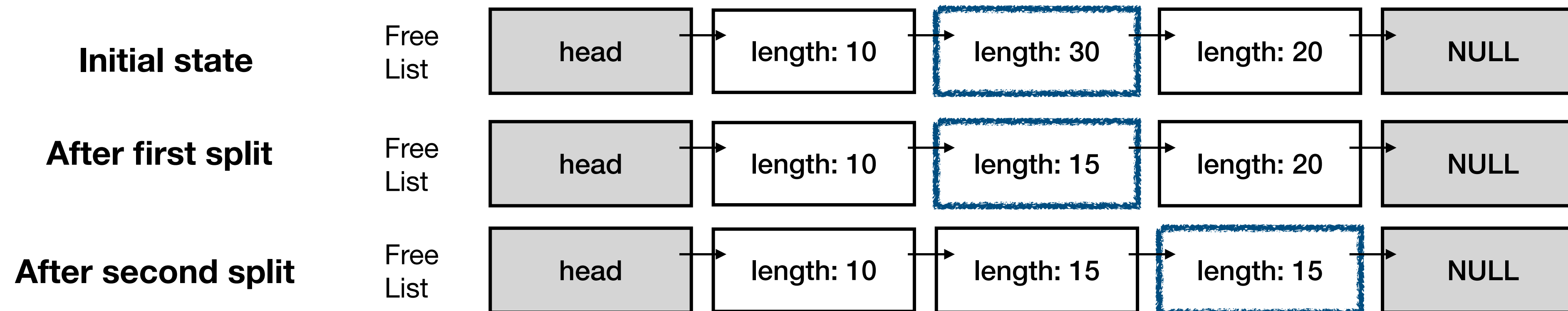
- ◎ The result after **splitting** would be



# Strategies

## Next fit

- **Next fit** looks for the **first free chunk that is large enough to accommodate the request**
  - Differently from first fit, it records the last chunk being split at the free list. When a new allocation is needed, the search starts from such position onwards.
  - Search for free chunks becomes more uniform across the free list, avoiding small fragments at the list's beginning
  - **Example below:** the next fit strategy would choose length 30, without going through the full list. If then another *request to allocate 5 bytes* would arrive, the search would start from the second chunk onwards





# Managing Free Space

## Homework and considerations

- ◎ We have just shown some simple examples for these policies
  - **Homework:** try with other examples of free lists to fully understand the benefits of each strategy
- ◎ These are basic policies, there are more sophisticated algorithms...

# Segregated Lists

## Slab allocator

● If a program has one (or a few) popular-sized memory allocation requests, keep separate (**segregated**) free lists to manage objects with those sizes

- Fragmentation is easy to manage
- Finding free space is very efficient

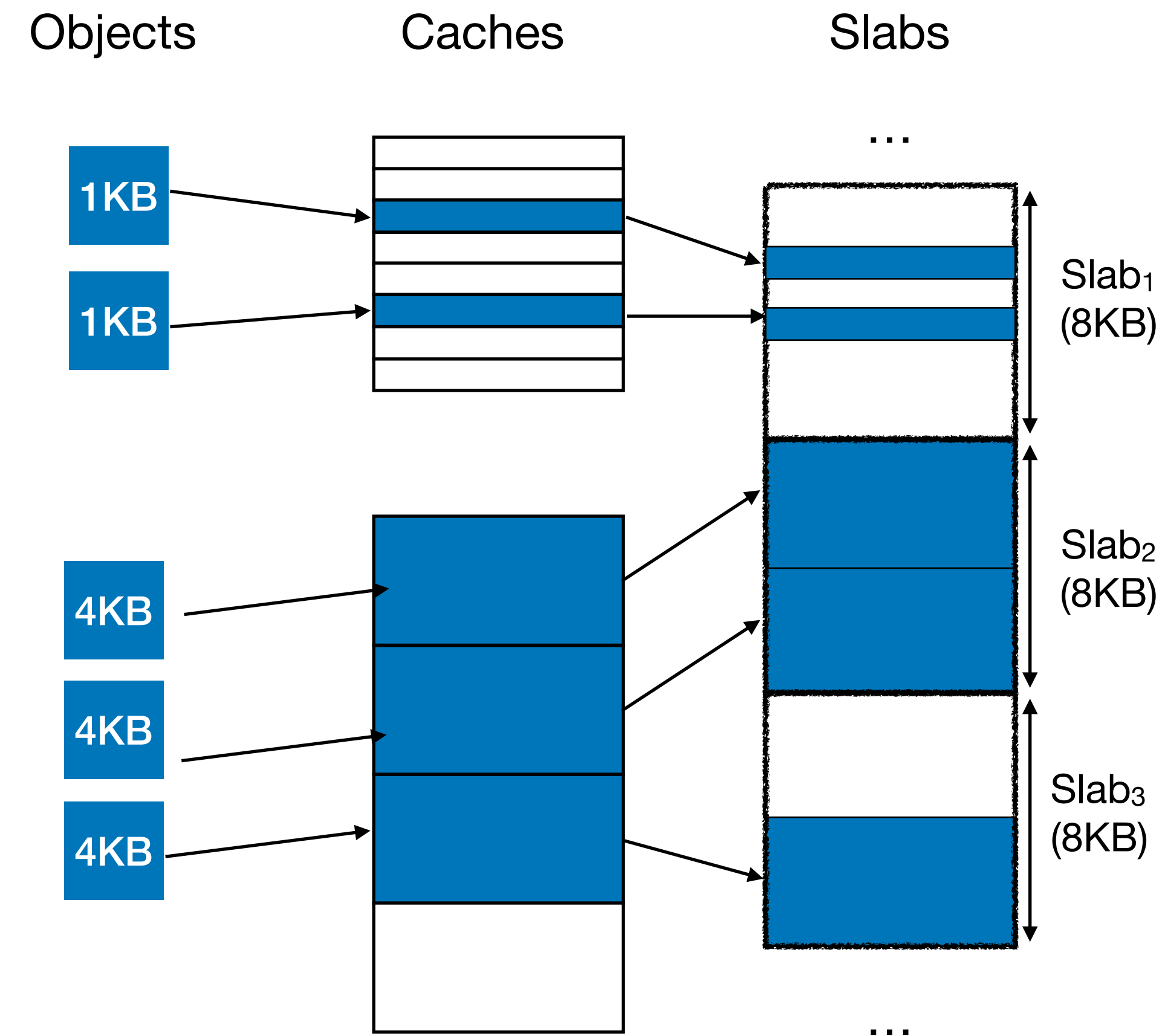
● Interesting for kernel objects (e.g., the **slab allocator**)

- Different object **caches** are allocated for specific kernel objects (e.g., locks, file system inodes, ...)
- Each object cache is a **segregated free list** with a given object size

● How much memory should one give to each cache?

- **Slabs**, managed by a more general memory allocator<sup>1</sup>, allow making this a dynamic decision by
  - Allocating a slab to a cache when the latter is low on available memory
  - Reallocating a slab when the cache is no longer using it

<sup>1</sup> We will talk about paging (page allocator) in the next class



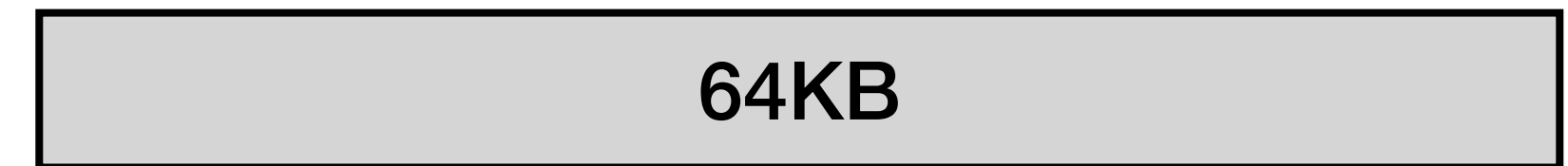
**Disclaimer:** illustrative example

# Buddy Allocation

## Splitting

- Free memory is managed as a big space of size  $2^N$ 
  - When a request for memory is made, **free space is recursively split by two** until reaching the **smallest chunk that can hold the request**
  - Chunks have a fixed size ( $2^N$ ), meaning that this scheme can suffer from **internal fragmentation!**

● **Example:** Allocate 5KB

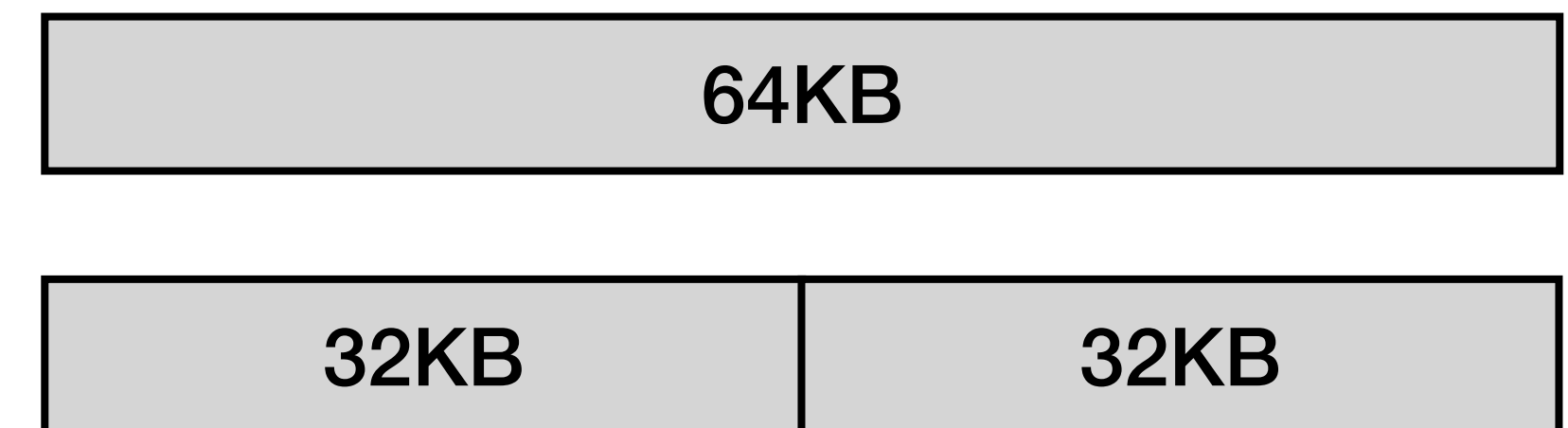


# Buddy Allocation

## Splitting

- Free memory is managed as a big space of size  $2^N$ 
  - When a request for memory is made, **free space is recursively split by two** until reaching the **smallest chunk that can hold the request**
  - Chunks have a fixed size ( $2^N$ ), meaning that this scheme can suffer from **internal fragmentation!**

● **Example:** Allocate 5KB

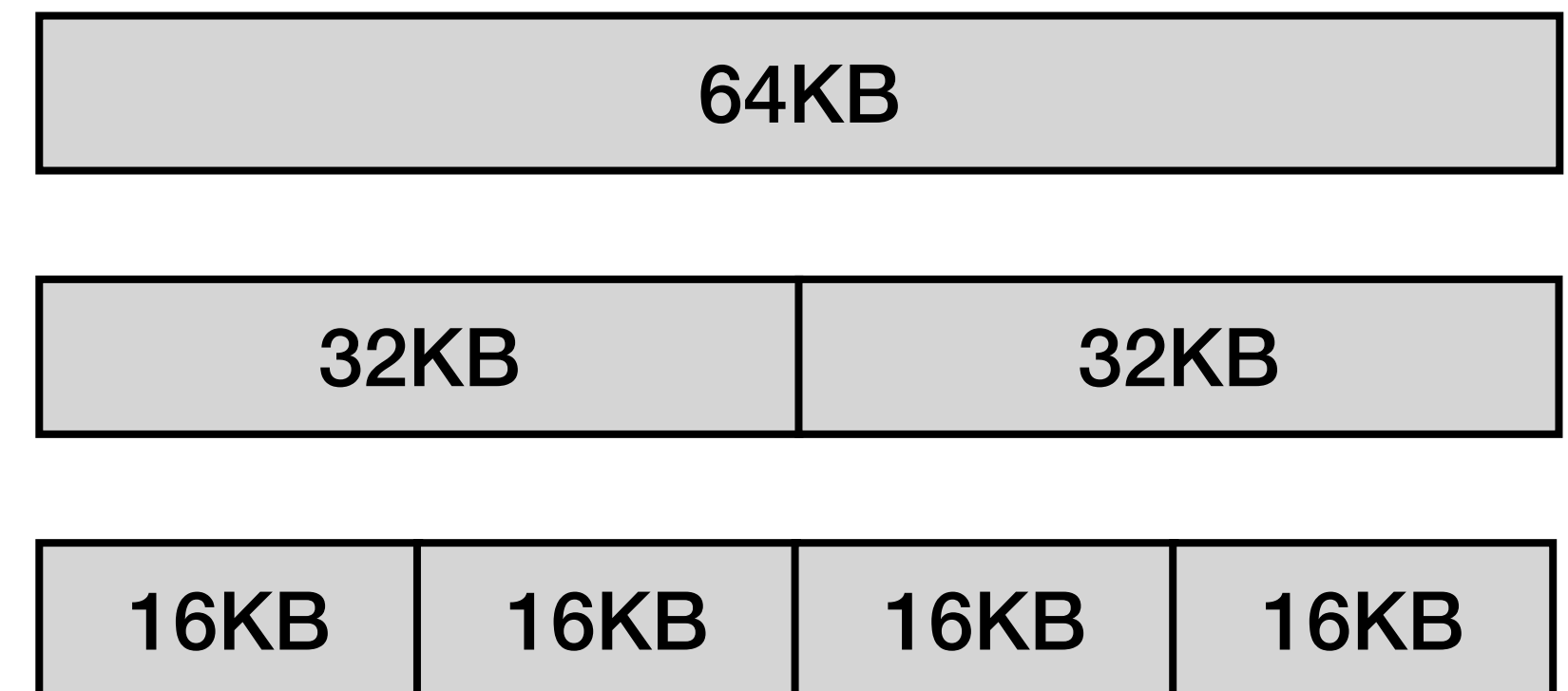


# Buddy Allocation

## Splitting

- Free memory is managed as a big space of size  $2^N$ 
  - When a request for memory is made, **free space is recursively split by two** until reaching the **smallest chunk that can hold the request**
  - Chunks have a fixed size ( $2^N$ ), meaning that this scheme can suffer from **internal fragmentation**!

● **Example:** Allocate 5KB



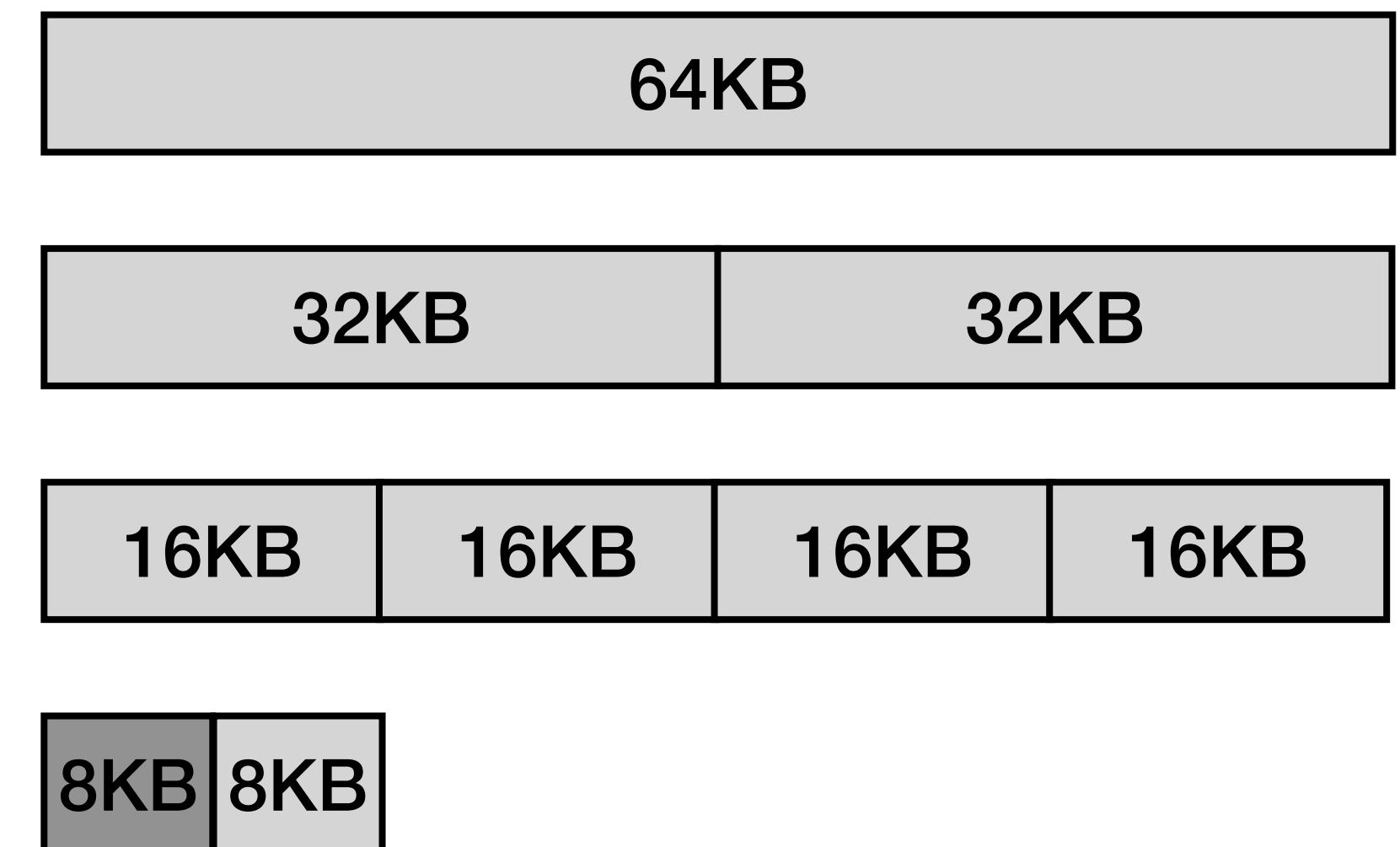


# Buddy Allocation

## Splitting

- Free memory is managed as a big space of size  $2^N$ 
  - When a request for memory is made, **free space is recursively split by two** until reaching the **smallest chunk that can hold the request**
  - Chunks have a fixed size ( $2^N$ ), meaning that this scheme can suffer from **internal fragmentation**!

● **Example:** Allocate 5KB

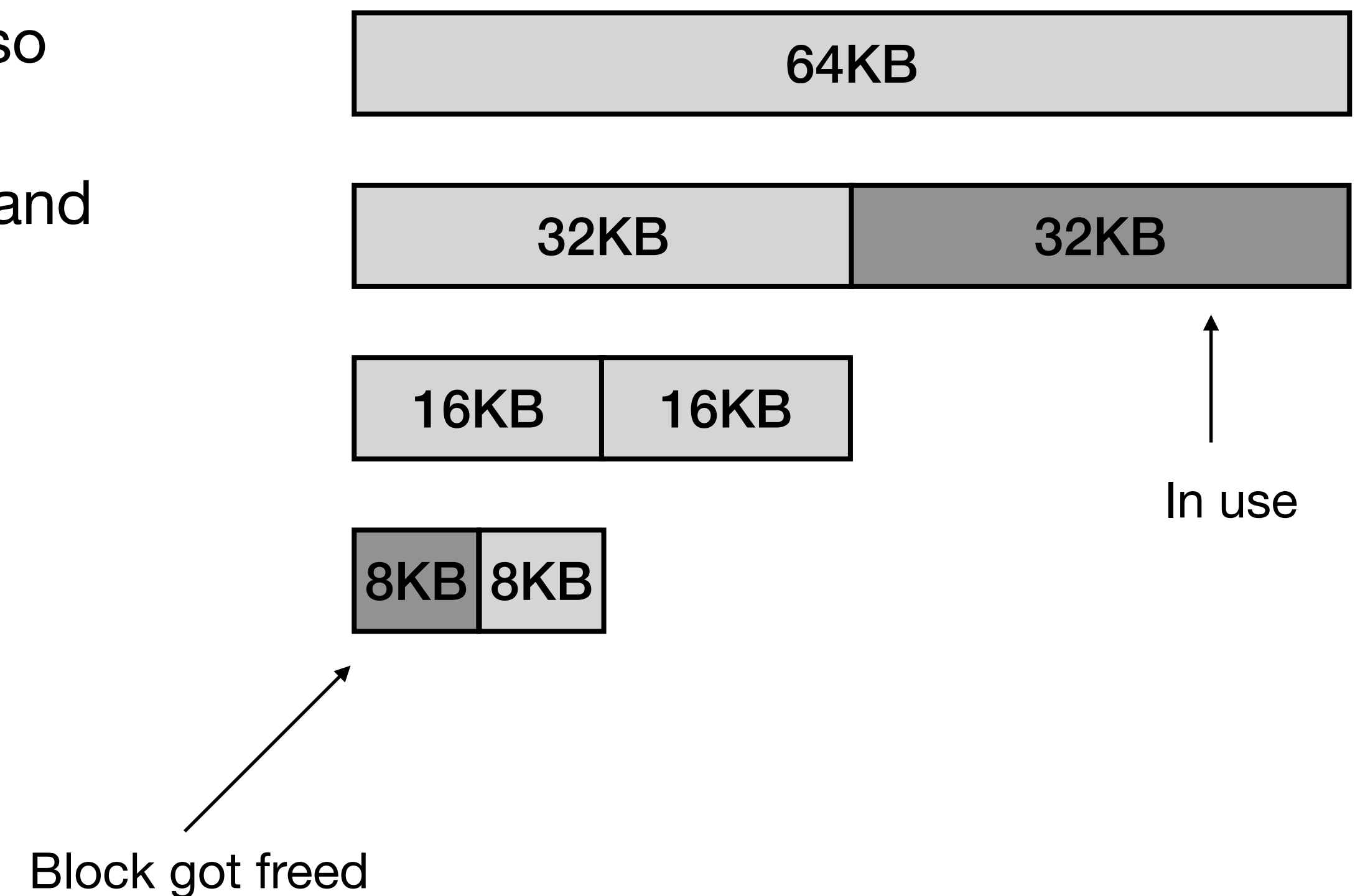


Allocation found!  
(but 3KB are being wasted...)

# Buddy Allocation

## Coalescing

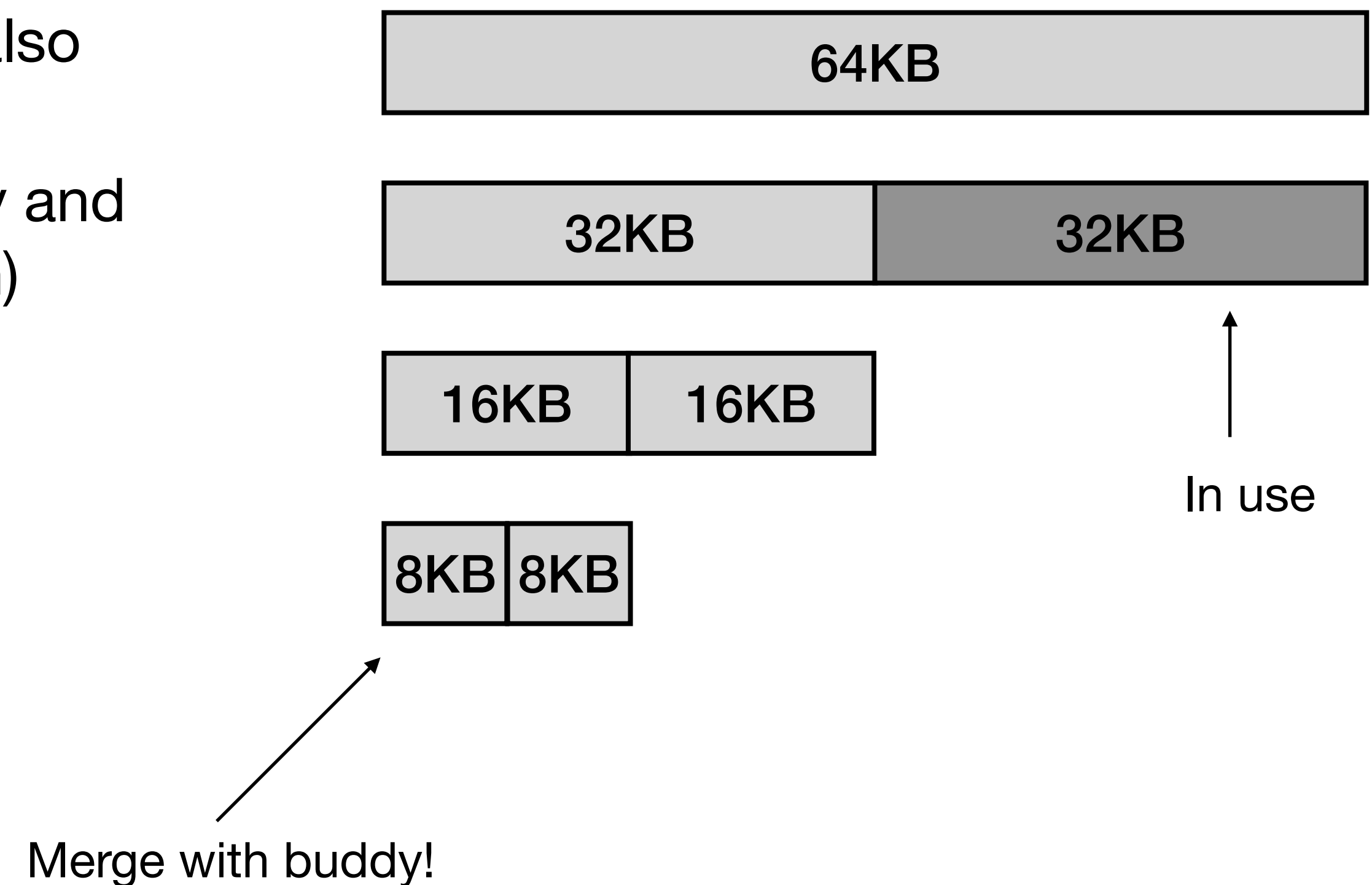
- ◎ The algorithm shines when coalescing blocks
  - When a block is free, and the buddy (neighbor) is also free, then **coalesce the two chunks**
  - The resulting chunk is then matched with its buddy and coalesced again if both are free (**recursive pattern**)
- ◎ **Example:** Free the first 8KB block



# Buddy Allocation

## Coalescing

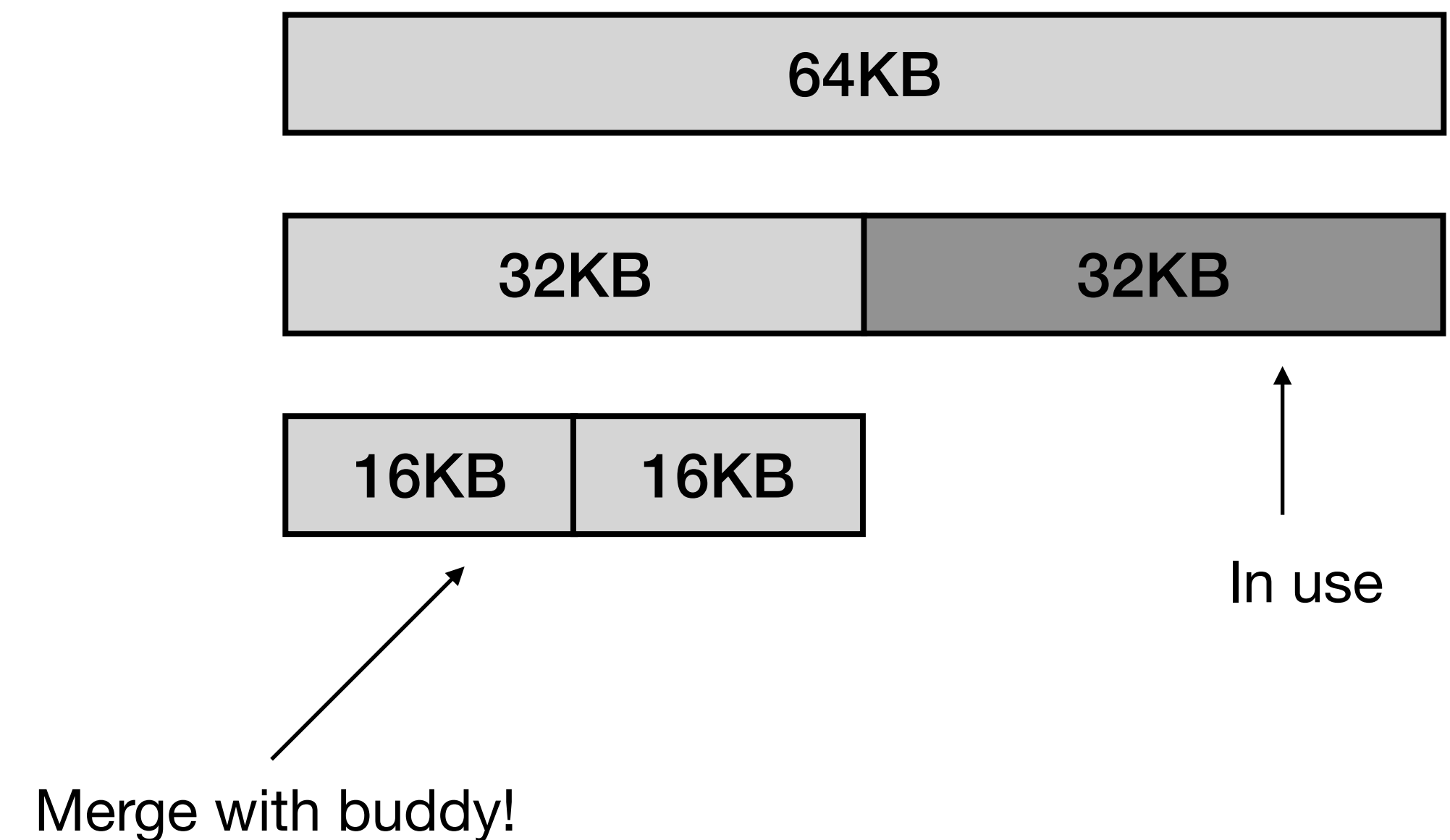
- ◎ The algorithm shines when coalescing blocks
  - When a block is free, and the buddy (neighbor) is also free, then **coalesce the two chunks**
  - The resulting chunk is then matched with its buddy and coalesced again if both are free (**recursive pattern**)
- ◎ **Example:** Free the first 8KB block



# Buddy Allocation

## Coalescing

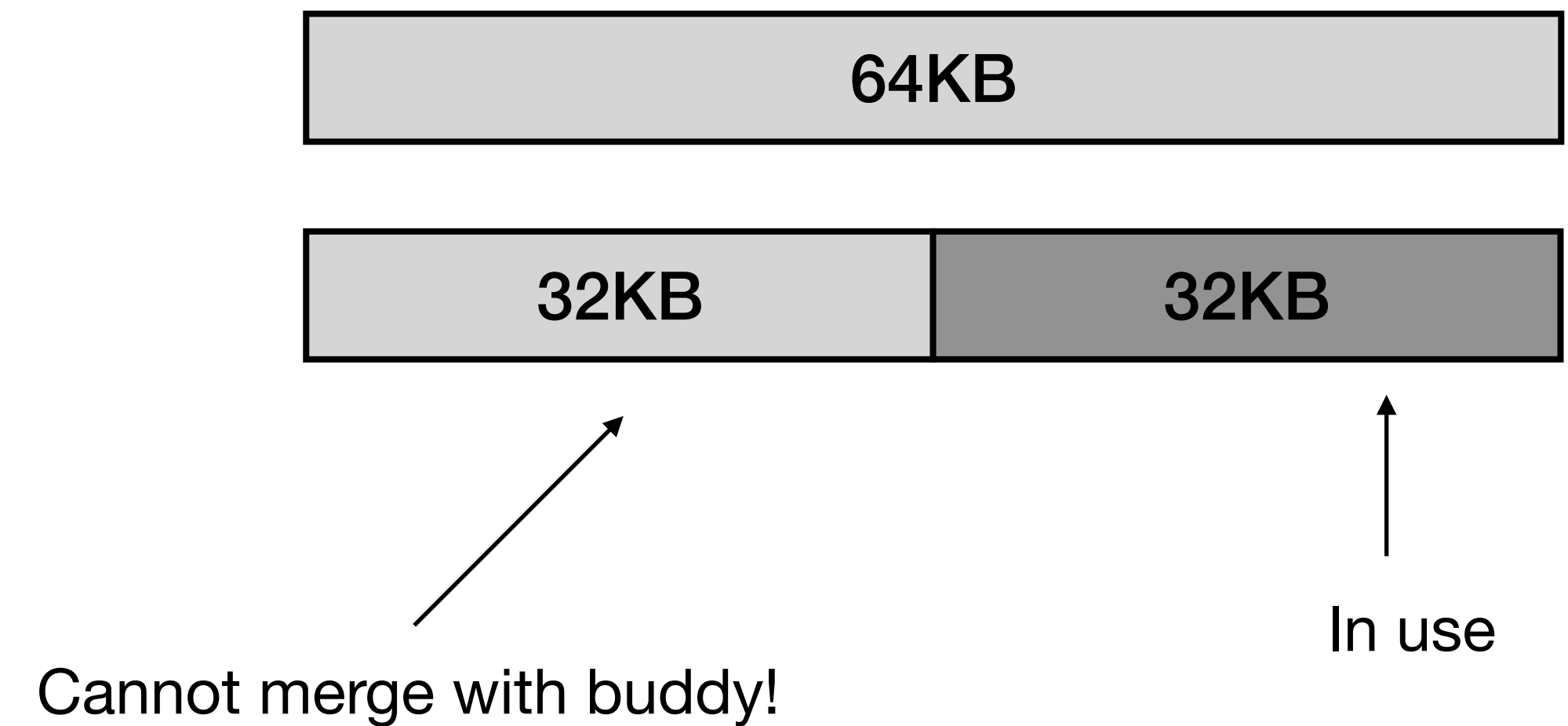
- ◎ The algorithm shines when coalescing blocks
  - When a block is free, and the buddy (neighbor) is also free, then **coalesce the two chunks**
  - The resulting chunk is then matched with its buddy and coalesced again if both are free (**recursive pattern**)
- ◎ **Example:** Free the first 8KB block



# Buddy Allocation

## Coalescing

- ◎ The algorithm shines when coalescing blocks
  - When a block is free, and the buddy (neighbor) is also free, then **coalesce the two chunks**
  - The resulting chunk is then matched with its buddy and coalesced again if both are free (**recursive pattern**)
- ◎ **Example:** Free the first 8KB block





# Managing Free Space

## Other allocators

- Using lists may not be the most efficient structure...
  - Advanced allocators use more complex structures such as trees
- Freeing memory for multi-threaded workloads and multi processor systems is also challenging for allocators
- Making a fast, space-efficient, and scalable allocator is still a on-going research challenge in our days :)

# More Information

- **Chapters 13 to 17** - Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau. **Operating Systems: Three Easy Pieces**. Arpaci-Dusseau Books, 2018.
- Avi Silberschatz, Peter Baer Galvin, Greg Gagne. **Operating System Concepts (10. ed)**. John Wiley & Sons, 2018.

# Questions?