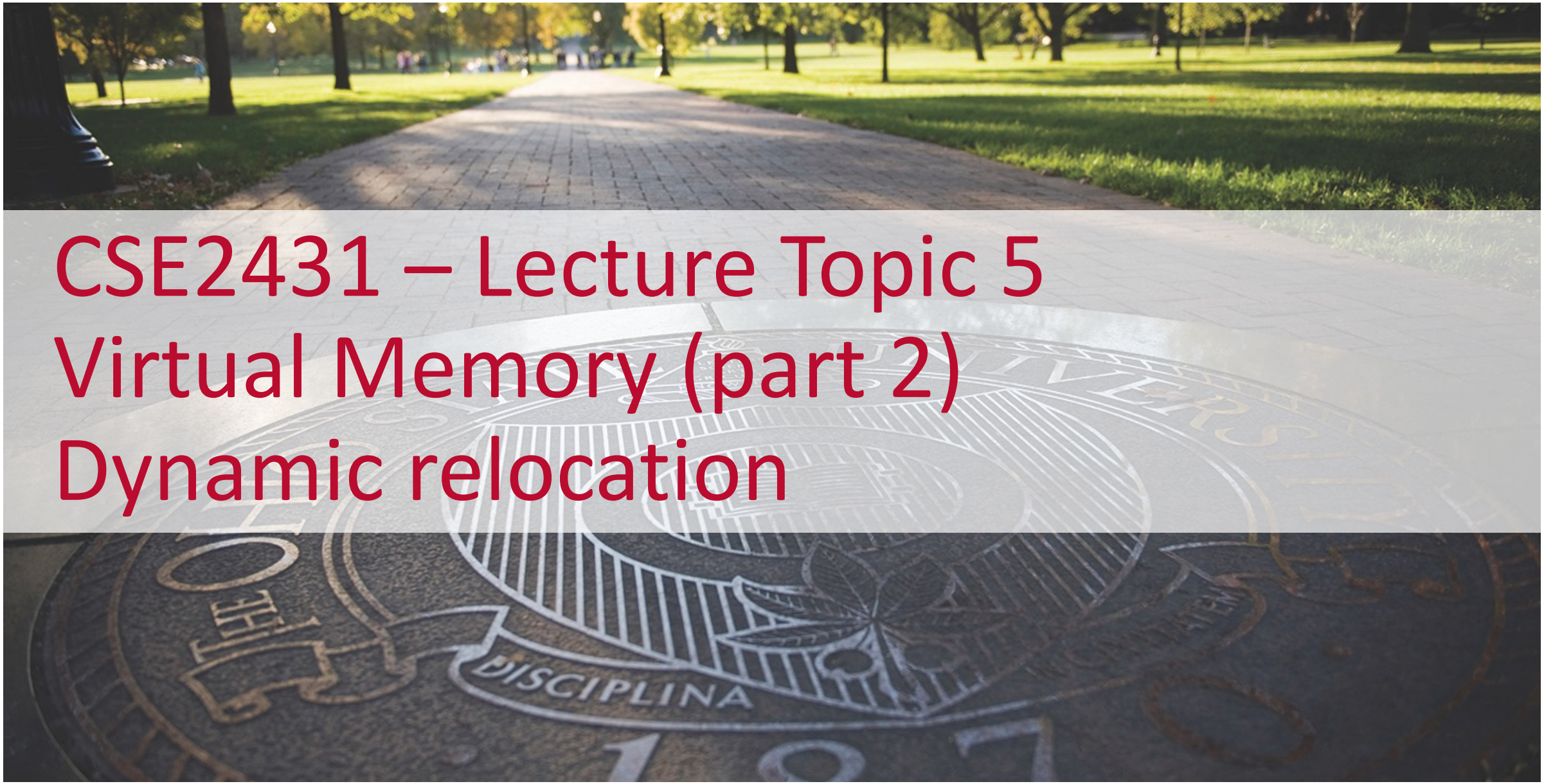


CSE2431 – Lecture Topic 5
Virtual Memory (part 2)
Dynamic relocation

THE OHIO STATE UNIVERSITY
COLLEGE OF ENGINEERING

Virtual Memory (pt 2) Dynamic Relocation

Instructor: Luan Duong, Ph.D.

CSE 2431: Introduction to Operating Systems

Reading: Chapter 13-17 in Required Textbook

Previous lecture

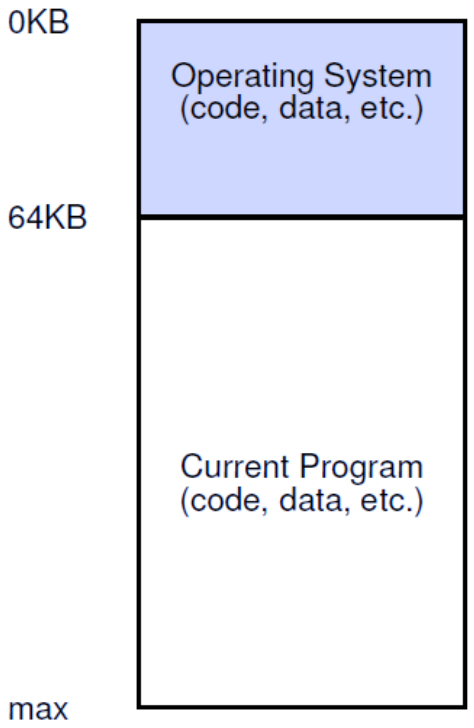
- Memory overview (part 1)
  - Overview of memory
  - Memory and bit operations
  - Virtual memory in Operating System
  - Virtual memory usage
- Virtual memory procedures (part 2)
  - Dynamic relocation
  - Paging
  - Swaping

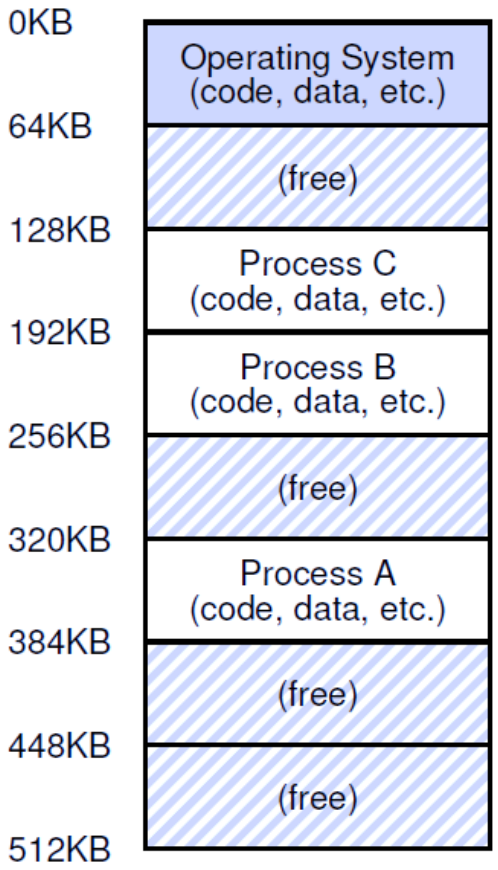
Outline: Virtual Memory (Part 2)

- Memory overview (part 1)
  - Overview of memory
  - Memory and bit operations
  - Virtual memory in Operating System
  - Virtual memory usage
- Virtual memory procedures (part 2)
  - Dynamic relocation
  - Paging
  - Swaping

Virtual memory in Modern OS

- OS provides an illusion to each process that memory is like this:





But in fact, memory is divided like this!

Let’s go back to operating system

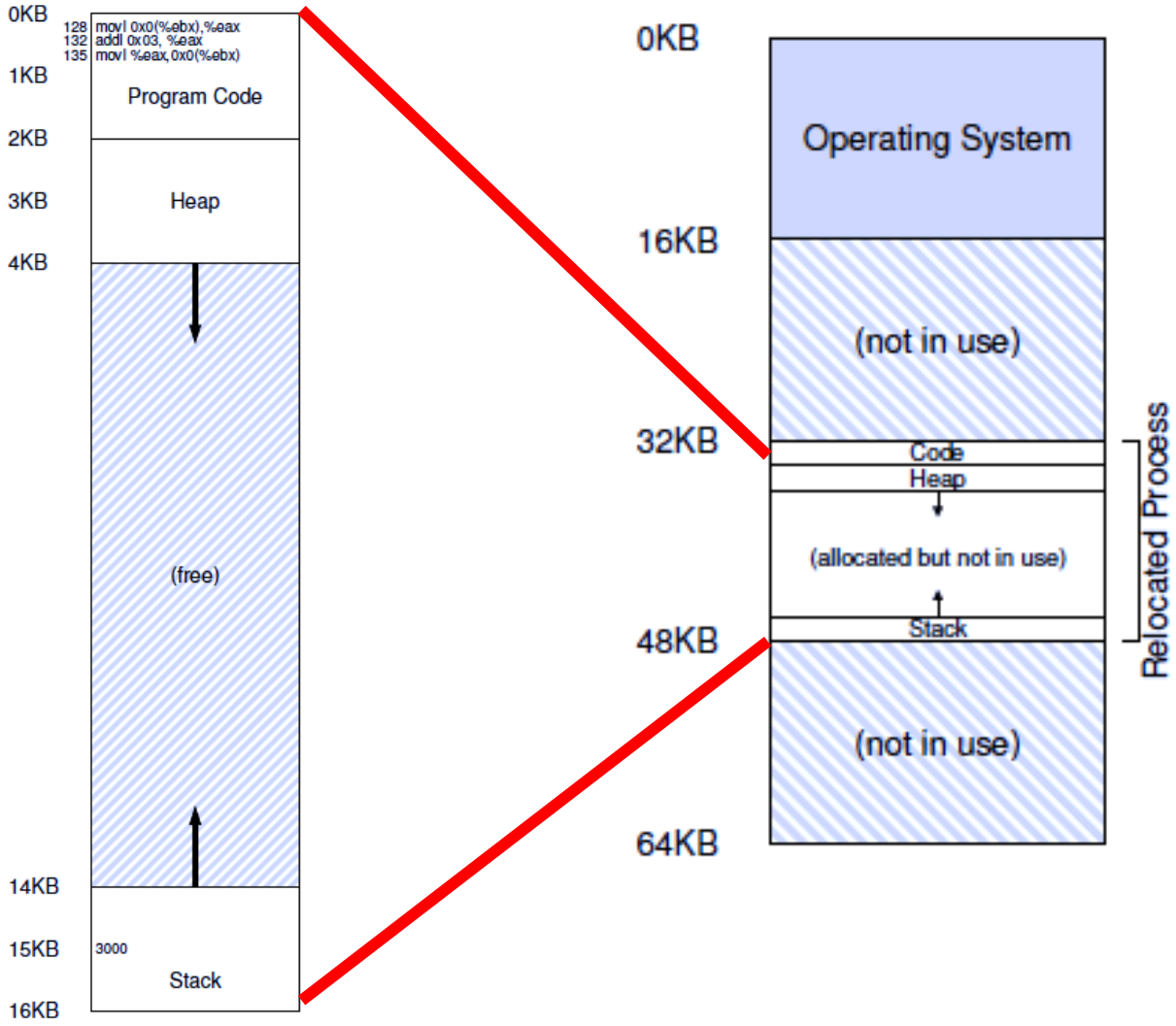
- Question: how does an OS provide an illusion to a program that its address space is from 0 to MAX while it is actually not

- Brief answer: **address translation**
  - **Support from CPU:** whenever CPU sees a **virtual memory address**, it can perform some computation to translate it into the **physical memory address**
  - **Support from OS:** OS tells the CPU how to perform the computation

Address Translation (simple)

- First, let's assume a process' address space is contiguously allocated in physical memory

- Example
  - Virtual address: 0KB-16KB
  - Physical address: 32KB-48KB



Dynamic Relocation

- CPU has two registers: **base** register and **bounds** register

- When an OS loads a process into memory, OS sets these two registers
  - In previous example, **base=32KB** and **bounds=16KB**

Dynamic Relocation

- Whenever the program accesses a (virtual) memory address, the CPU performs the following calculation
  - if virtual address >= bounds, report error
  - else physical address = virtual address + base

- Examples (base=32KB, bounds=16KB):
  - virtual address 0 => physical address? 32KB
  - virtual address 2KB => physical address? 34KB
  - virtual address 17KB => physical address? error

Dynamic Relocation

- Another example:
  - A process with an address space of size 4KB (this is unrealistically small). Assume it has been loaded at physical address 16 KB.
  - virtual address 0 => physical address? 16 KB
  - virtual address 1KB => physical address? 17 KB
  - virtual address 3000 => physical address? 19384 (16384 + 3000)
  - virtual address 4400 => physical address? Error (out of bound)

Details of dynamic relocation -- hardware

- Memory management unit (MMU)
  - Base and bounds registers
  - Logic to check bounds and add base to virtual address
- Special instructions to set/get base and bounds registers
  - How are they different from load/store instructions for normal registers?
  - These instructions should be restricted operations. Otherwise, a process may access arbitrary physical memory address by using these instructions.
- Error handling
  - Raise exceptions when virtual address is out of bound
  - Allow OS to register exception handlers: registering should also be restricted

Details of dynamic relocation -- OS

- When creating a process
  - The OS needs to find an empty memory space that is big enough to hold the new process. We will discuss details later.
  - Then OS needs to set base and bounds registers.
- When terminating a process
  - The OS needs to release the process' memory space
- When switching from process A to process B during a context switch
  - The OS saves the base and bounds registers of A and loads registers of B
- The OS needs to register exception handlers
  - A common approach is to kill the process

Dynamic Relocation -- Limitation

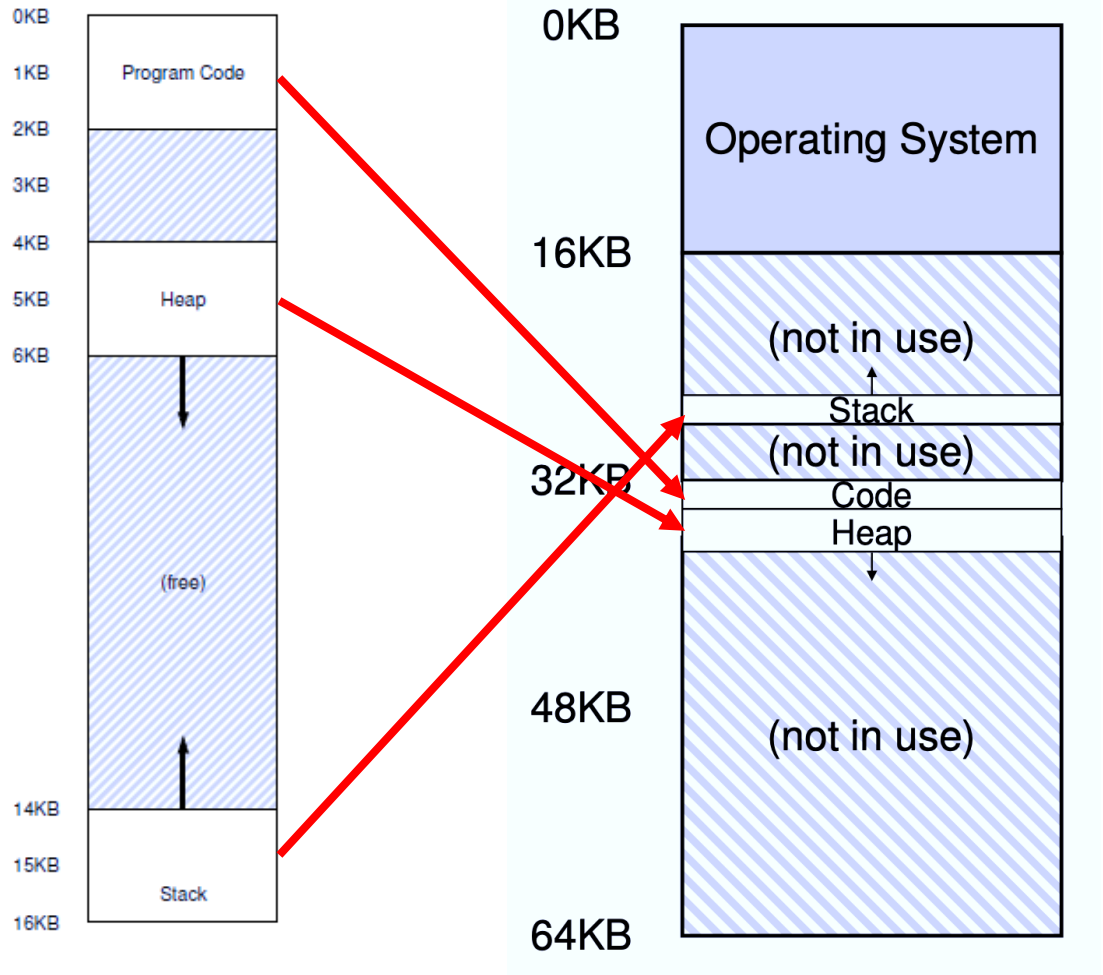
- When creating a process, OS has to allocate the max possible size of memory for the process
  - It will be a waste of memory if the process does not actually use all memory ---internal fragmentation

- When creating a process, OS has to find an empty memory space that is big enough to hold the new process
  - If OS has two 10KB empty spaces, it cannot load a 16KB process --- external fragmentation.

Segmentation – Improved version

- Motivations:
  - It's harder to find an empty memory slot for bigger processes
  - Multiple processes often share code. E.g. fork, libraries, …
- Segmentation: divide a process' memory into segments
  - Common segments: code, stack, heap, etc
  - Use a pair of base and bounds registers for each segment

Segmentation: Example



Segment	Base	Size
Code	32K	2K
Heap	34K	3K
Stack	28K	2K

Size segment is the **Bound registers** introduced previously

What is the physical address of virtual address:

100 In Code segment, so 32KB + 100

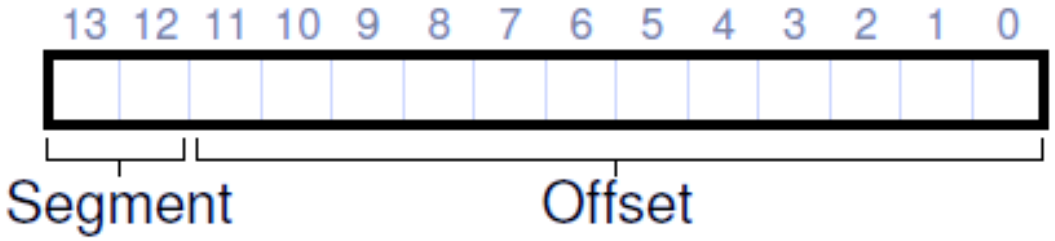
4200 In Heap segment, so 34KB + 4200 - 4K = 39016 – 4096 = 34920

7KB Not in any segment, so **invalid** (segmentation fault)

Determine the Correct Segment

- Question: given a virtual address, how to know which segment it belongs to (and which base and bounds registers to use)

- An explicit approach: use the first few bits of the virtual address to determine the segment



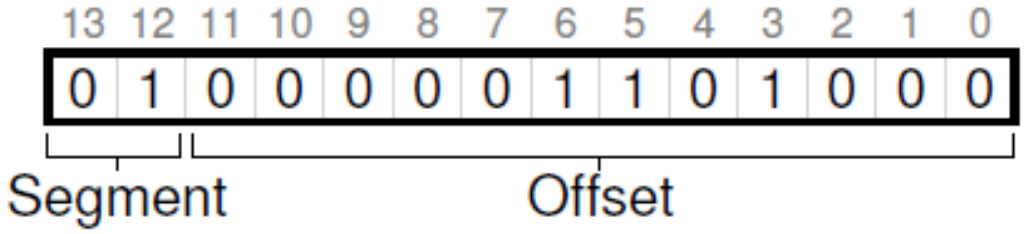
00 code
01 heap
10 stack

Cons: the max size of a segment becomes smaller.

Determine the Correct Segment

Virtual address 4200 = 01000001101000

00 code
01 heap
10 stack



heap

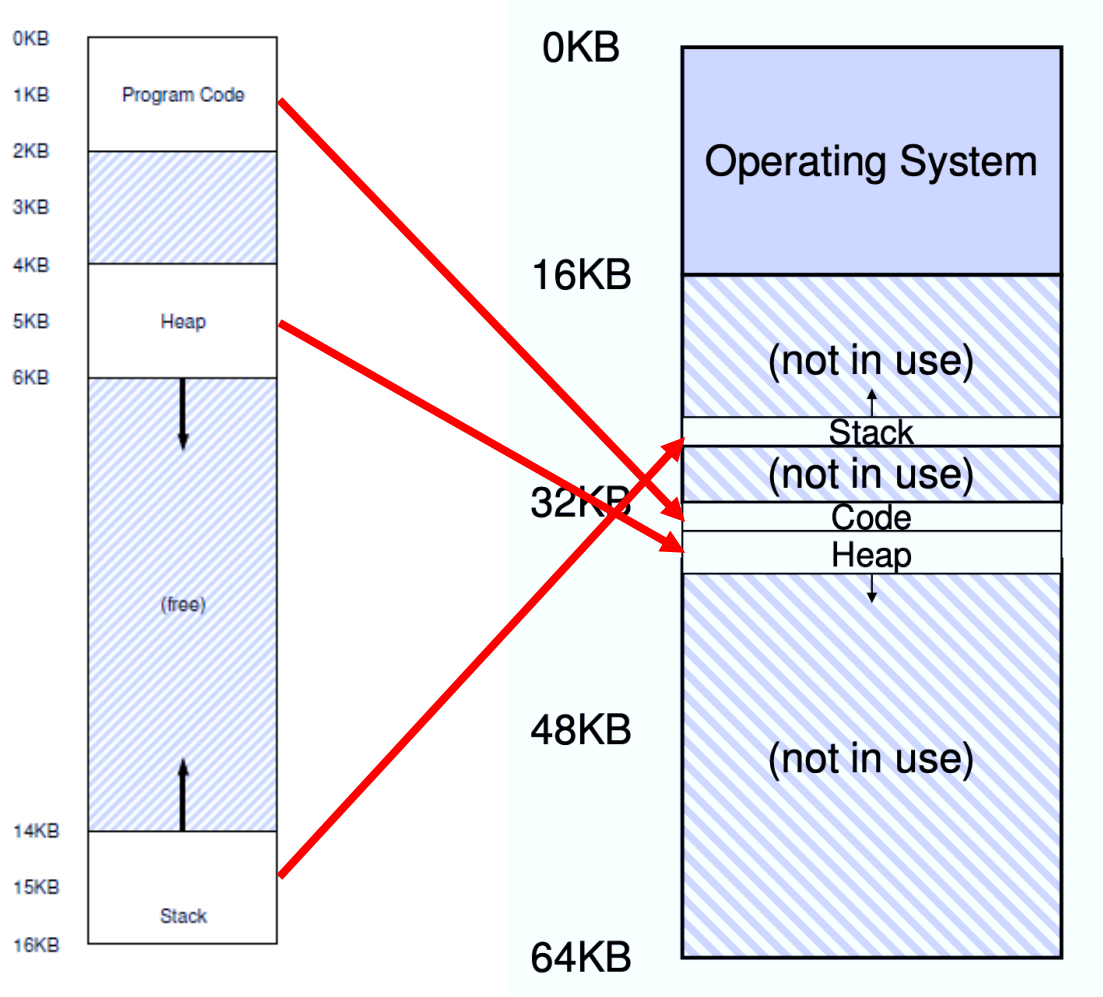
Segment	Base	Size
Code	32K	2K
Heap	34K	3K
Stack	28K	2K

Offset < 2K bound, so OK.
Physical address = offset + base
= 34K + 000001101000b

Note the allocation of this approach is different from the previous example.

Did you notice anything about the offset? In decimal it is indeed 104 (which is indeed 4200-4096)

How about the Stack?



Segment	Base	Size (max 4K)	Grows Positive?
$Code_{00}$	32K	2K	1
$Heap_{01}$	34K	3K	1
$Stack_{11}$	28K	2K	0

Example: we wish to access virtual address of 15KB.

15KB Binary: **11** . | **1100 0000 0000**
Stack segment | 3KB offset

Maximum size: 4K = 4096
So, we need to subtract the maximum size from 3KB (which results in -1KB); and then add it to the base, which is 28K.
Physical address is: 28K – 1KB = 27KB.

Determine the Correct Segment

- Any problem here?
- Using two bits for segments, the virtual address space is very limited. The **V.A. space then becomes 4KB maximum**! (instead of 16KB, because it is divided by 4)
- When we use the top two bits, we only have three segments (code, heap, stack). One segment of the address spaces is **useless**!

Support for sharing

- **Code segment** is often shared by multiple processes while heap and stack segments are usually not shared
  - If for example, the address was generated from the P.C. (instruction fetch) → address is within the **code segment**
  - If the address is based off the stack or base pointer, it must be in the **stack segment**
  - Any other address must be in **the heap.**

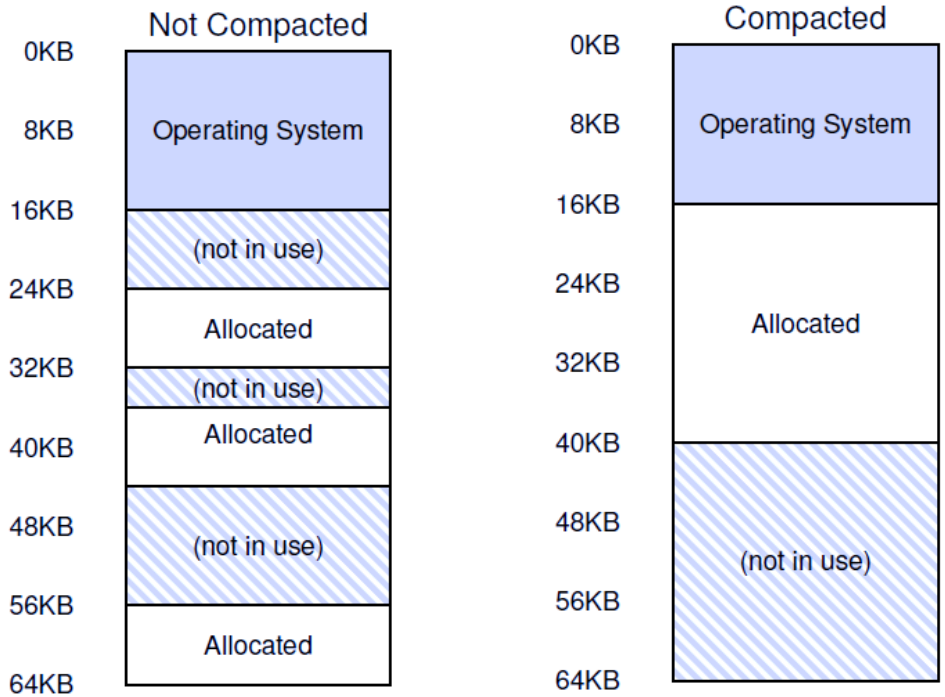
Support for sharing

- Problem 2: for security reason, a process should not be able to modify shared code
- CPU and OS have a mechanism to mark segments as read-only (implicitly)
  - Any write operation to the segment causes an error/exception

Segment	Base	Size (max 4K)	Grows Positive?	Protection
$Code_{00}$	32K	2K	1	Read-Execute
$Heap_{01}$	34K	3K	1	Read-Write
$Stack_{11}$	28K	2K	0	Read-Write

Find an empty slot for a segment

- **Problem:** memory **may not be allocated contiguously**
  - May not find a big enough empty slot even if the total free space is enough



One solution is to reorganize memory layout (called compaction)

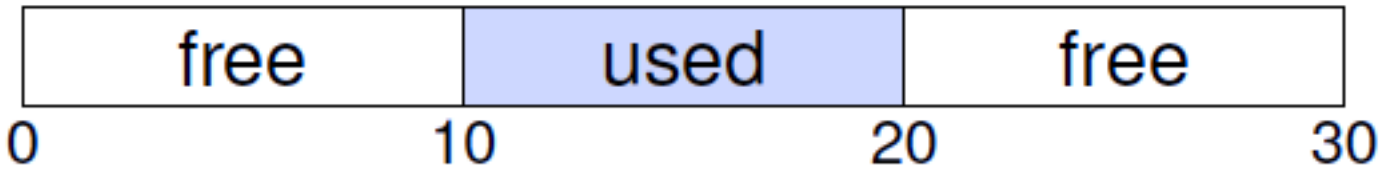
But obviously compaction is an expensive operation

The following discussion assumes no compaction, which means if a space is allocated, we cannot move it to another location in memory.

How to manage free space?

- How to manage variable-sized memory objects?
  - Segments in physical memory
  - malloc/free in heap segment

- Problem: external fragmentation
  - Free space gets chopped.
  - A memory allocation **may fail** even if the **total free space** is enough.



Questions of free-space management

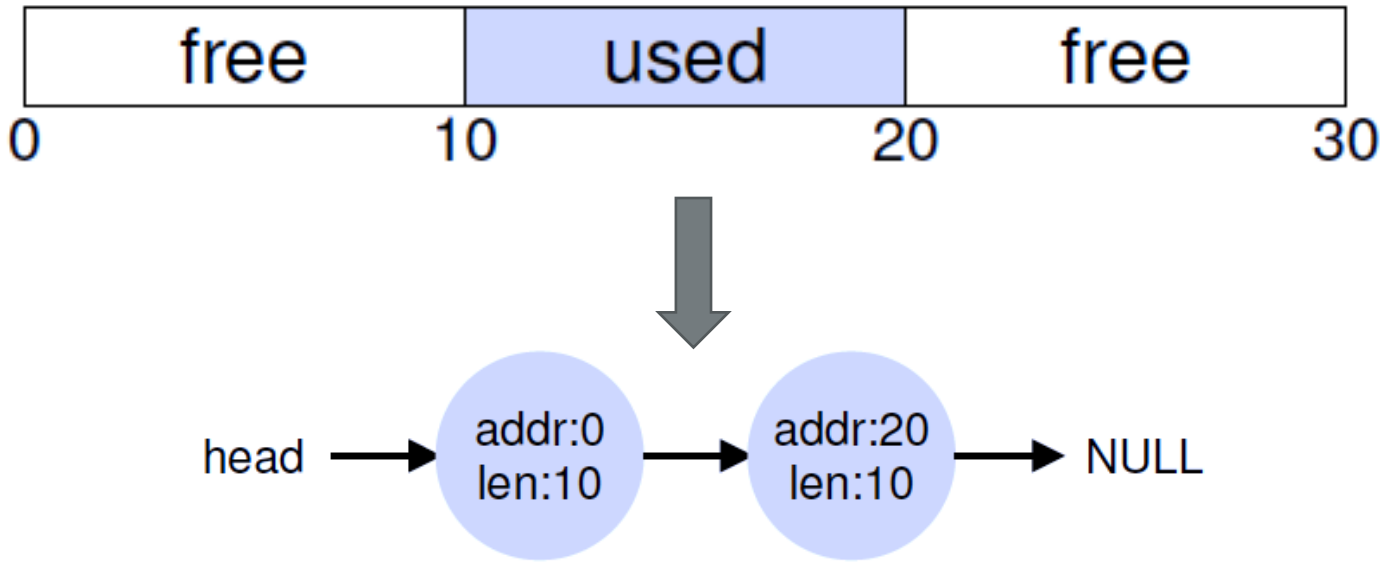
- How to find a big enough slot for a variable-sized request?
- How to minimize fragmentation?
- What are the overheads?

Mechanism and Policy

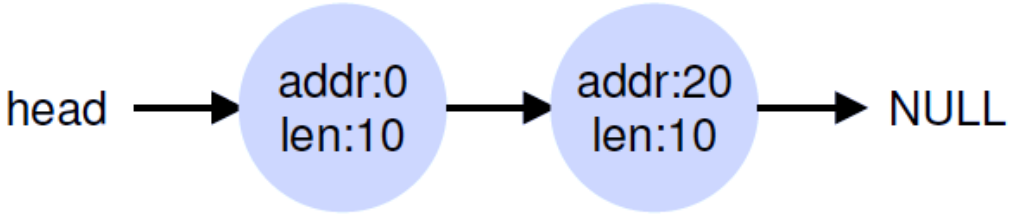
- Mechanism
  - How to keep track of used and empty slots?
- Policy
  - Which empty slot to use?
- Assumption:
  - The following discussion assumes no compaction, which means if a space is allocated, we cannot move it to another location in memory.

Mechanism: splitting and coalescing

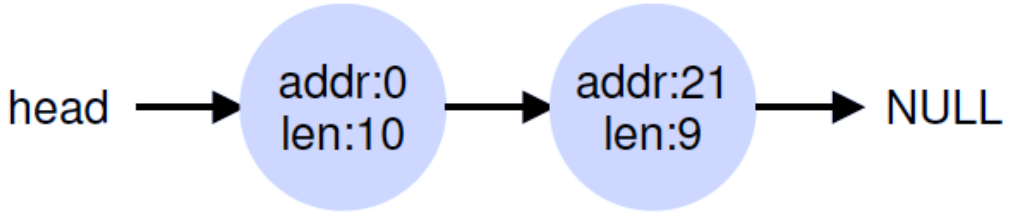
- What is the right data structure to keep track of free slots?
  - Linked list (free list)



Mechanism: splitting and coalescing

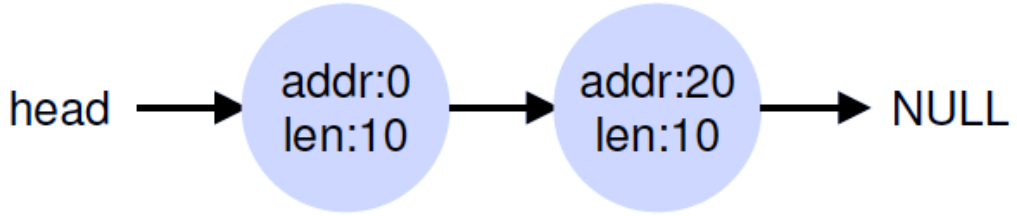


- User call `malloc(1)`



- This is called **Splitting**

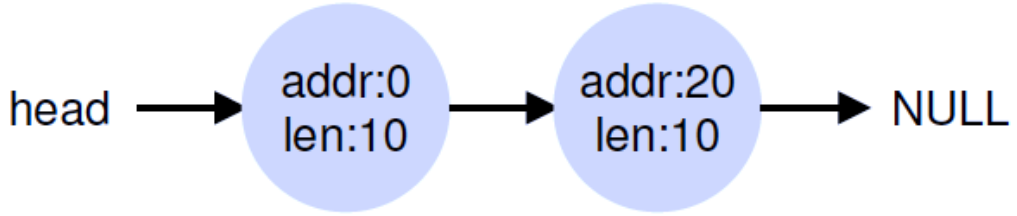
Mechanism: splitting and coalescing



- User call `free(ptr)`. ptr points to the used 10-byte slot

- Although entire heap is now FREE, but the memory still got divided into three chunks of 10 bytes each. Thus when the user requests 20 bytes, a simple list traversal will not find a free chunk!

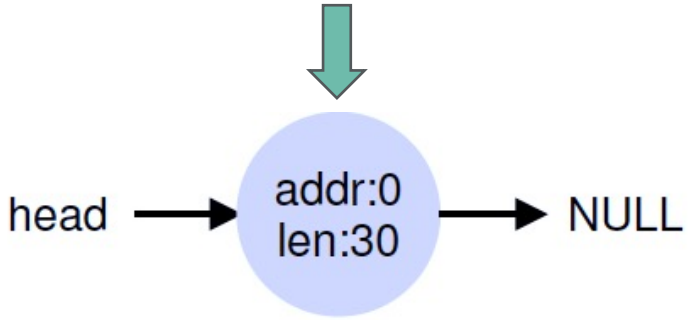
Mechanism: splitting and coalescing



- User call free(ptr). ptr points to the used 10-byte slot

- Coalesce free space

  Meaning: come together to form one mass or a whole

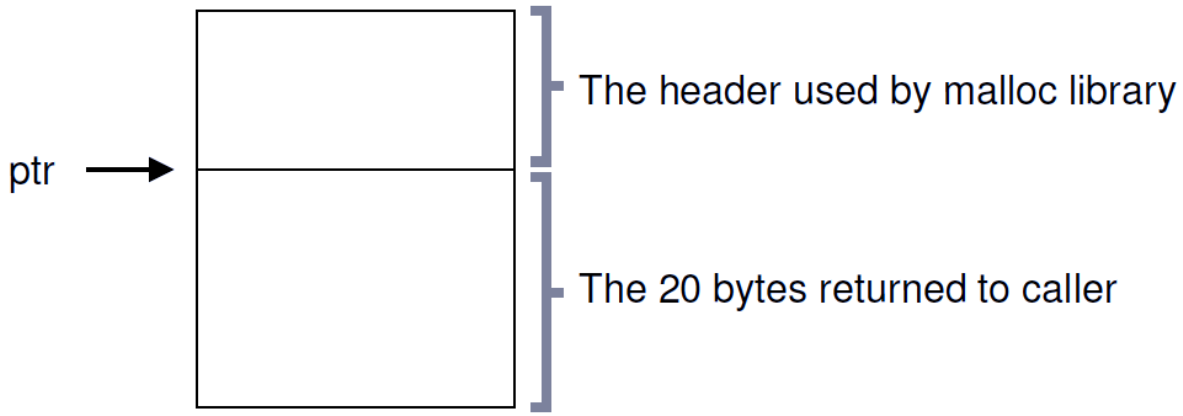


Implementing the mechanism

- So far you should get a basic idea of what are splitting and coalescing
- In the next slides, we will discuss how to implement them
- More specifically, suppose you are given a contiguous block of memory, how to implement malloc and free on it?
- Note: since we are discussing how to implement malloc, we cannot use a Linked List data structure directly, because a typical implementation needs to call malloc.

Layout

- Each used/empty slot has a hidden header
  - The header contains the size of the slot and other useful information
  - The header is created by `malloc`, but is not exposed to the user.
  - When you call `malloc(size)`, system actually needs to allocate **`size+header`**

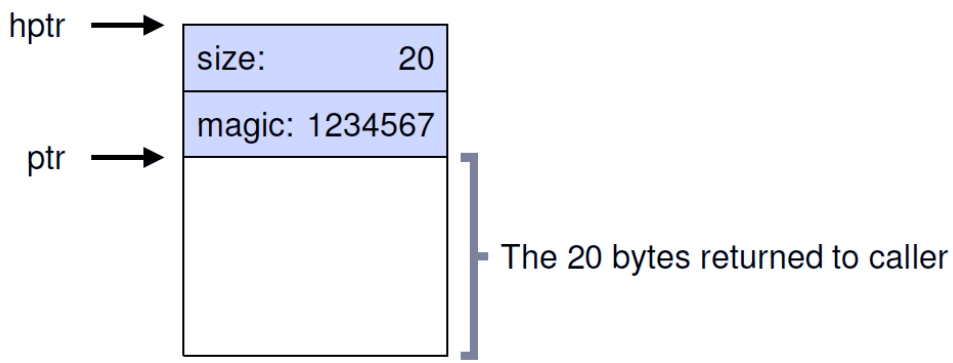


Header for a used slot

- For a **used slot**, header is defined as

```
typedef struct __header_t {
    int size;
    int magic;
} header_t;
```

- "Size" is the size of the used slot.
- "**Magic**" is a constant number for sanity check.



Header for an empty slot

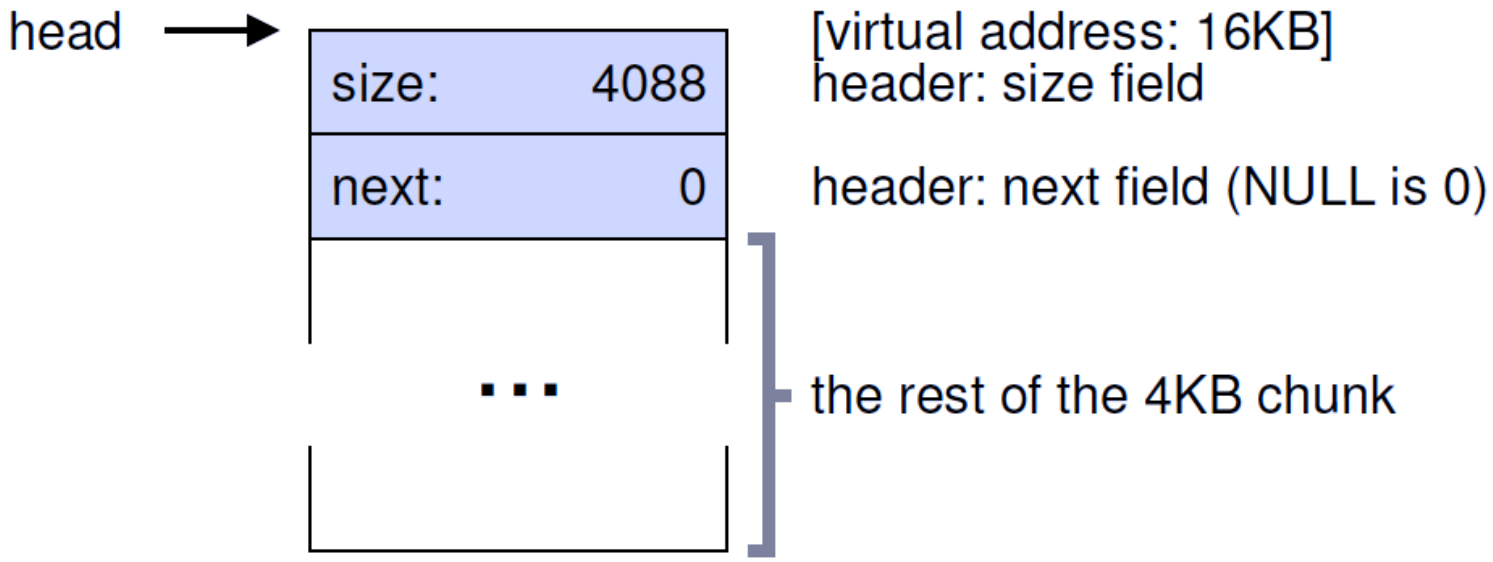
- For **an empty slot**, header is defined as

```
typedef struct __node_t {
    int                 size;
    struct __node_t *next;
} node_t;
```

- "Size" is the size of the empty slot
- "Next" identifies the address of the **next empty slot** (or 0 meaning end of linked list)

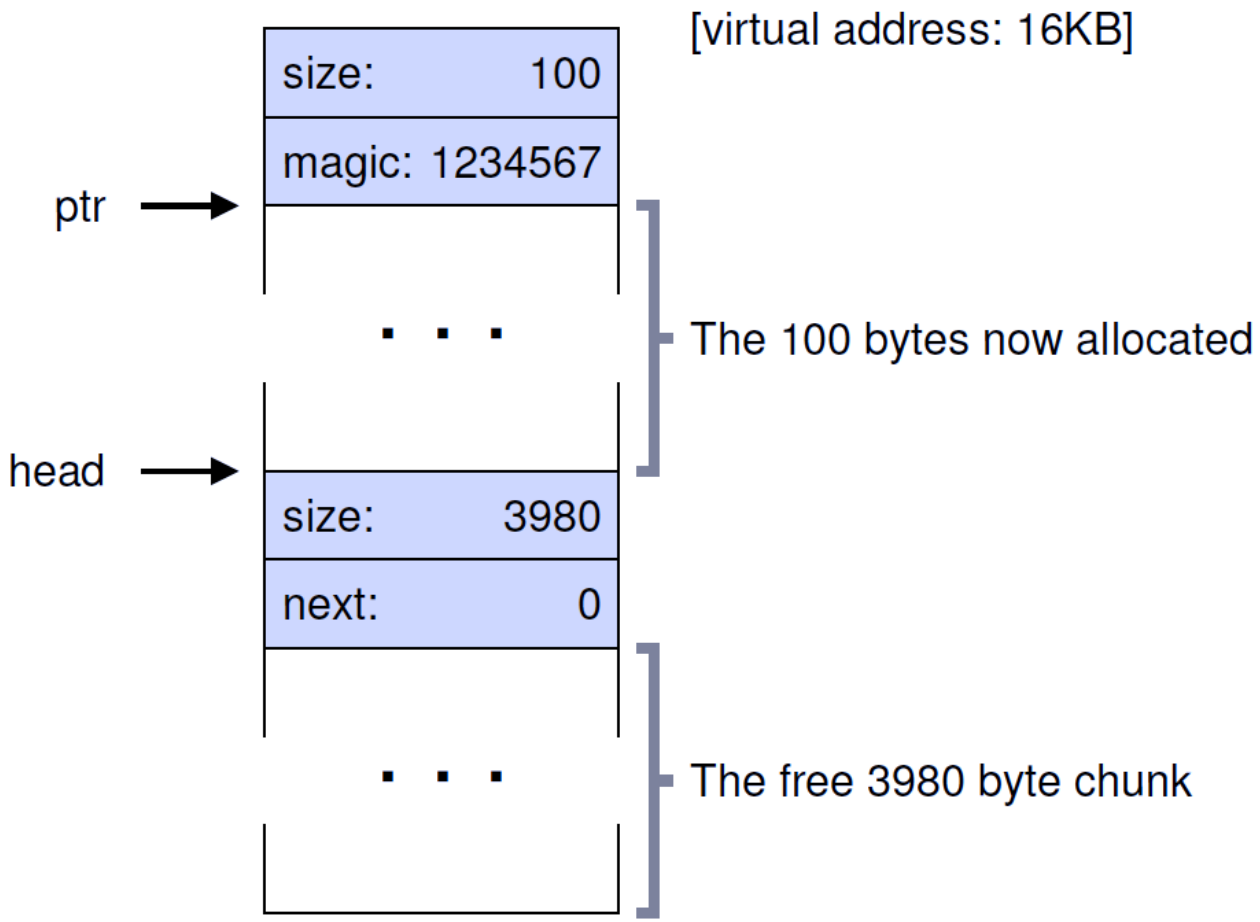
Example

- Suppose we have 4KB of empty memory space. A header takes 8 bytes.



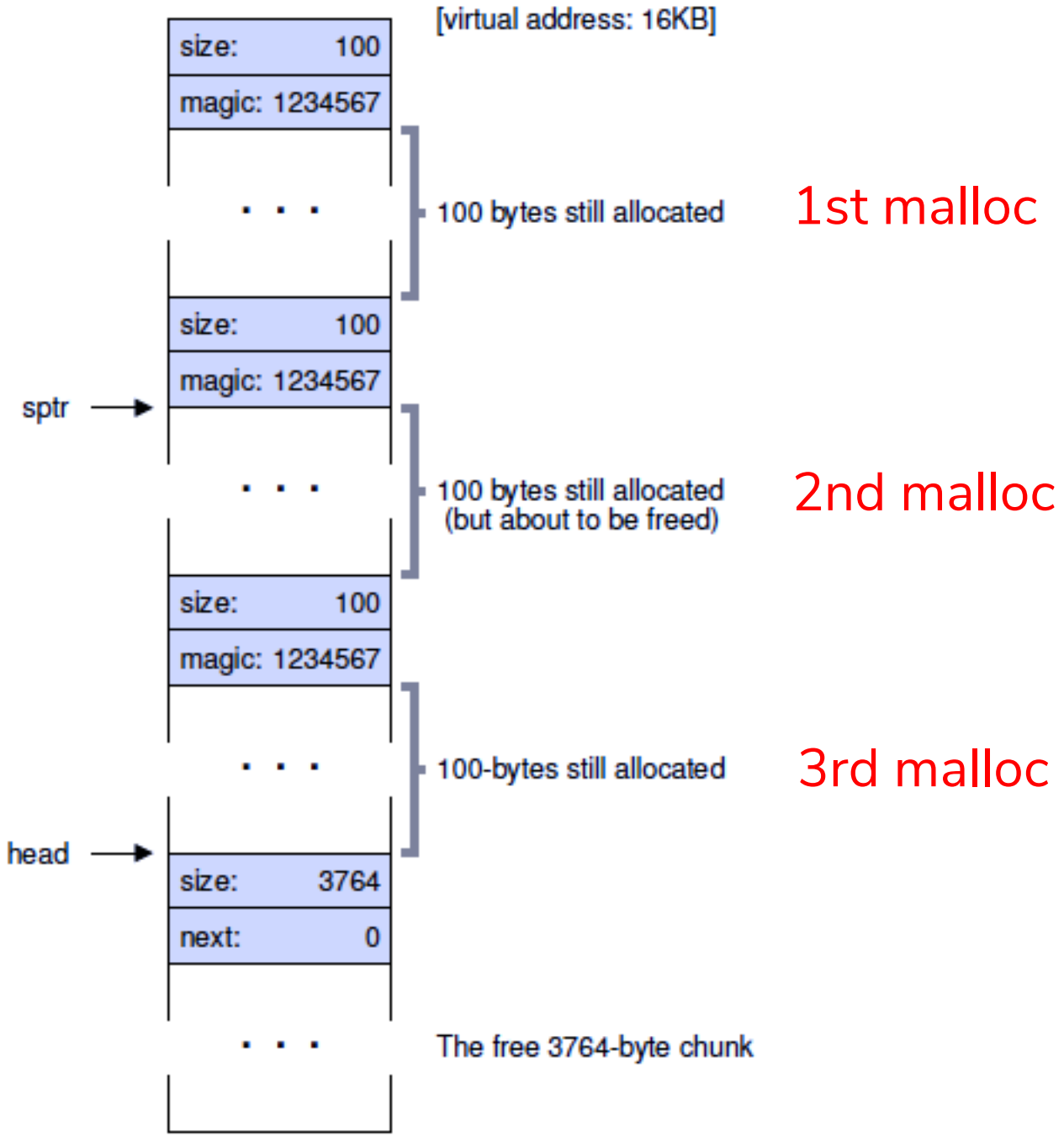
Example

- Suppose the user calls malloc(100)



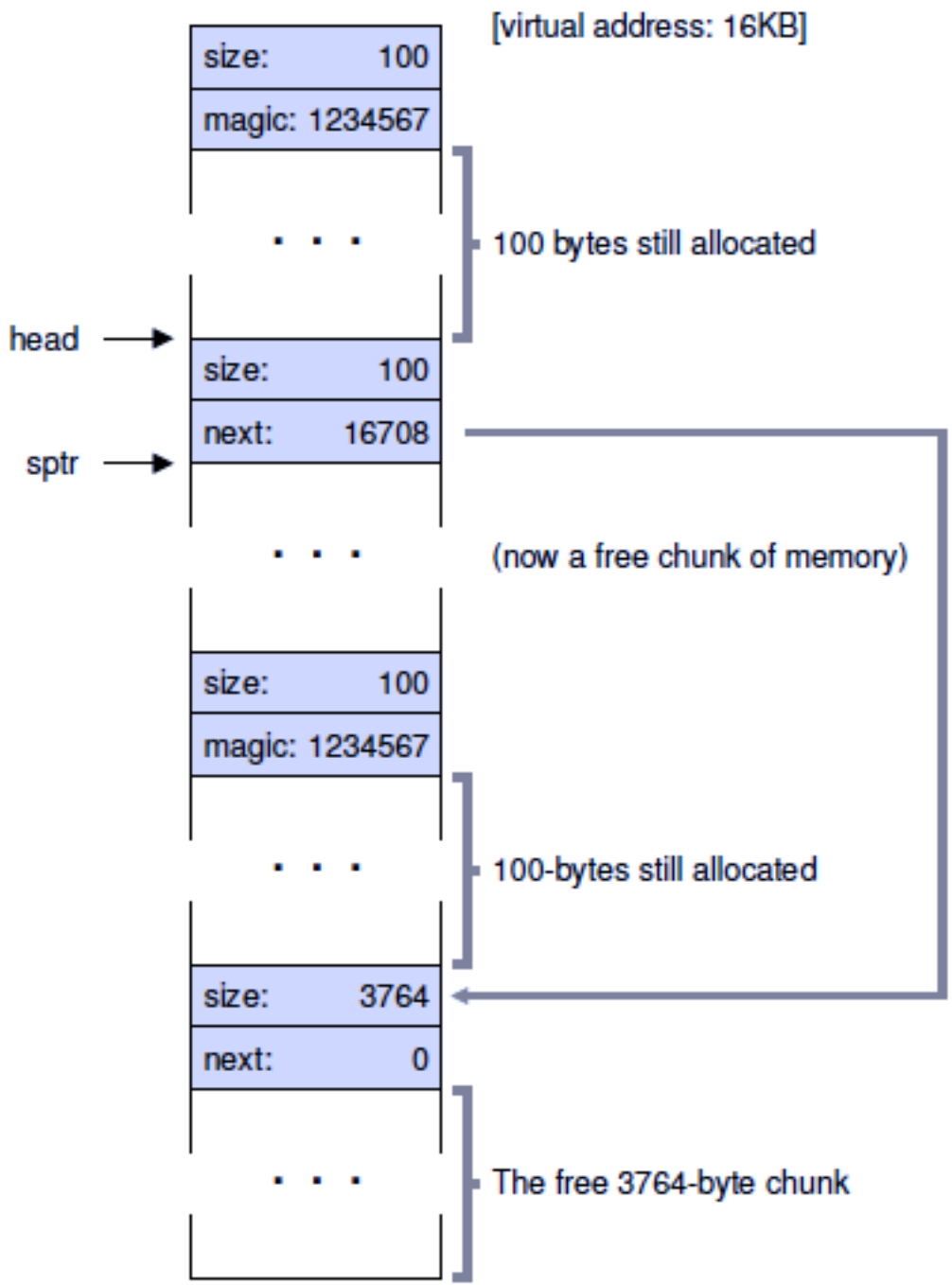
Example

- Suppose the user calls two more malloc(100)



Example

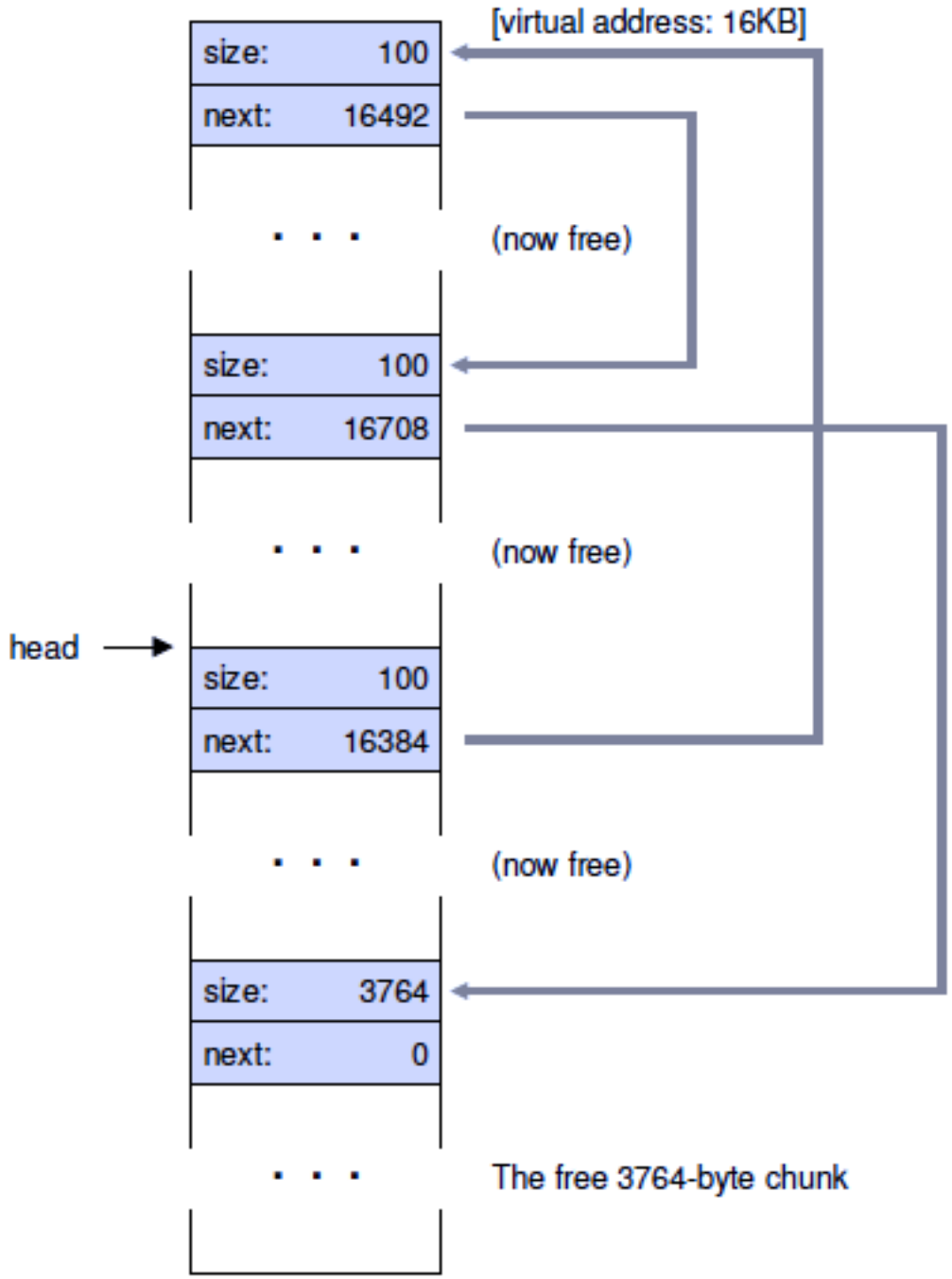
- Suppose the user calls free on the second used slot



Example

- Suppose the user calls free on all used slots

- We need some algorithm to merge empty slots



Policy

- Now we have a mechanism to keep track of used and empty slots
- Next question: when a user calls malloc(size), which empty slot to use?

Policy – Best fit

- Best fit:
  - Find all slots that are as big or bigger than requested size
  - Choose the smallest one among them
- Pros: reduce external fragmentation
- Cons: need to scan all empty slots

Policy – Worst fit

- Worst fit:
  - Find all slots that are as big or bigger than requested size
  - Choose the biggest one among them
- Pros: leave big chunks
- Cons: need to scan all empty slots; severe external fragmentation

Policy – First fit

- First fit:
  - Find the first empty slot that is big enough
- Pros: usually do not need to scan all empty slots
- Cons: may create many objects at the beginning of free list

Policy – Next fit

- Next fit:
  - Find the next empty slot that is big enough

- Pros: usually do not need to scan all empty slots; spread searches across the free list

Examples

- User calls malloc(15)
- Best fit

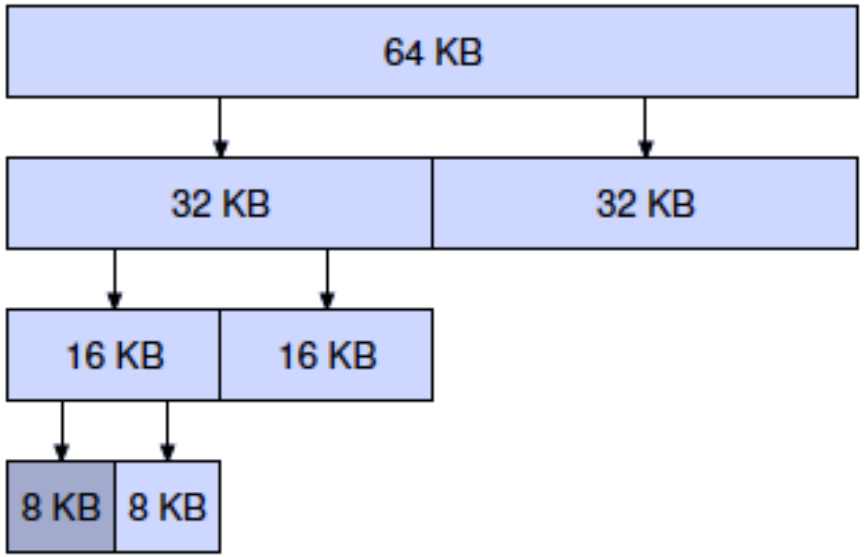
- First fit

Optimizations – Segregated Lists

- If one or a few fixed-sized objects are frequently allocated, we can keep a separate list just to manage those objects
- Pros: management of fixed-sized objects is significantly easier
- Examples of such objects: locks; file-system inodes; …

Optimizations – Buddy Allocation

- The allocator always allocates space of size 2^N.
- When user calls malloc, recursively divides free space by two until a block that is big enough to accommodate the request is found
- Example: malloc(7K)
- Pros: coalescing is simple
- Cons: internal fragmentation



Optimizations – Scaling

- Previous approaches need to scan free list
- On modern multi-core processors, if multiple processes call malloc simultaneously, one has to wait another.
- Could we allow concurrent search?

Summary: Dynamic Relocation

- Dynamic Relocation
  - How could we do address translation
  - How to determine the correct segment for translation
  - Splitting and Coalescing Mechanism, Implementation and Optimization
- Virtual memory procedures (part 2)
  - Paging
  - Swaping