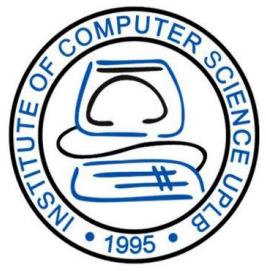
CMSC 125: Operating Systems

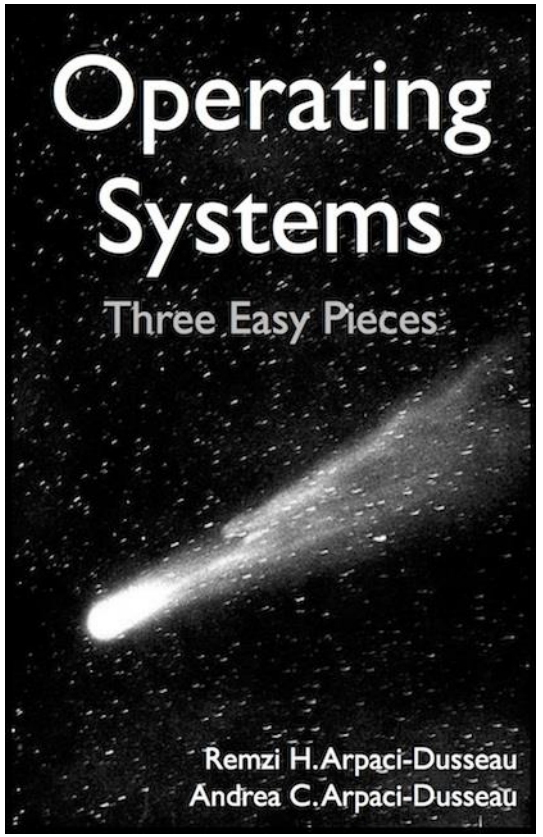
- Instructor: **Joseph Anthony C. Hermocilla**
- Email: jchermocilla@up.edu.ph
- Web: https://jachermocilla.org



Resources

Book: https://pages.cs.wisc.edu/~remzi/OSTEP/

Slides Template:
https://pages.cs.wisc.edu/~remzi/OSTEP/Educators-Slides/Youjip/



Acknowledgement

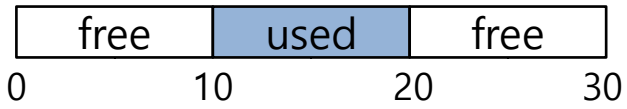
- This lecture slide set was initially developed for Operating System course in Computer Science Dept. at Hanyang University. This lecture slide set is for OSTEP book written by Remzi and Andrea at University of Wisconsin.

17. Free-Space Management

Operating System: Three Easy Pieces

Free-Space Management

- Free space must be managed to avoid fragmentation
- Easy if free space are in fixed-sizes (if using Paging)
- Hard when free space are in variable sizes (if using Segmentation for example)
  - Segmentation suffers from external fragmentation
  - Consider requesting 15 bytes given the configuration below. Can this request be satisfied?



Assumptions

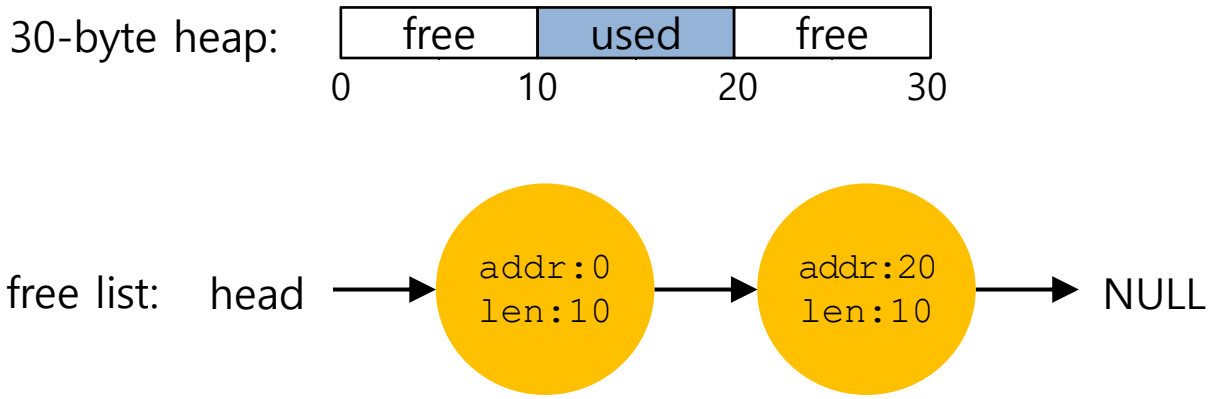
1. Focus is on user-level memory-allocation libraries
2. Interface follows malloc() and free()
   - `void *malloc(size t size)`
   - `void free(void *ptr)`
3. We focus on solving external fragmentation
4. Once memory from free list is allocated to a requester/client, it cannot be relocated to another location
   - No compaction of memory in free space
5. The memory being managed by the allocator is of fixed size and contiguous

Low-level Mechanisms

1. Splitting and Coalescing
2. Tracking the size of allocated regions
3. Building a simple list inside the free space to keep track free space

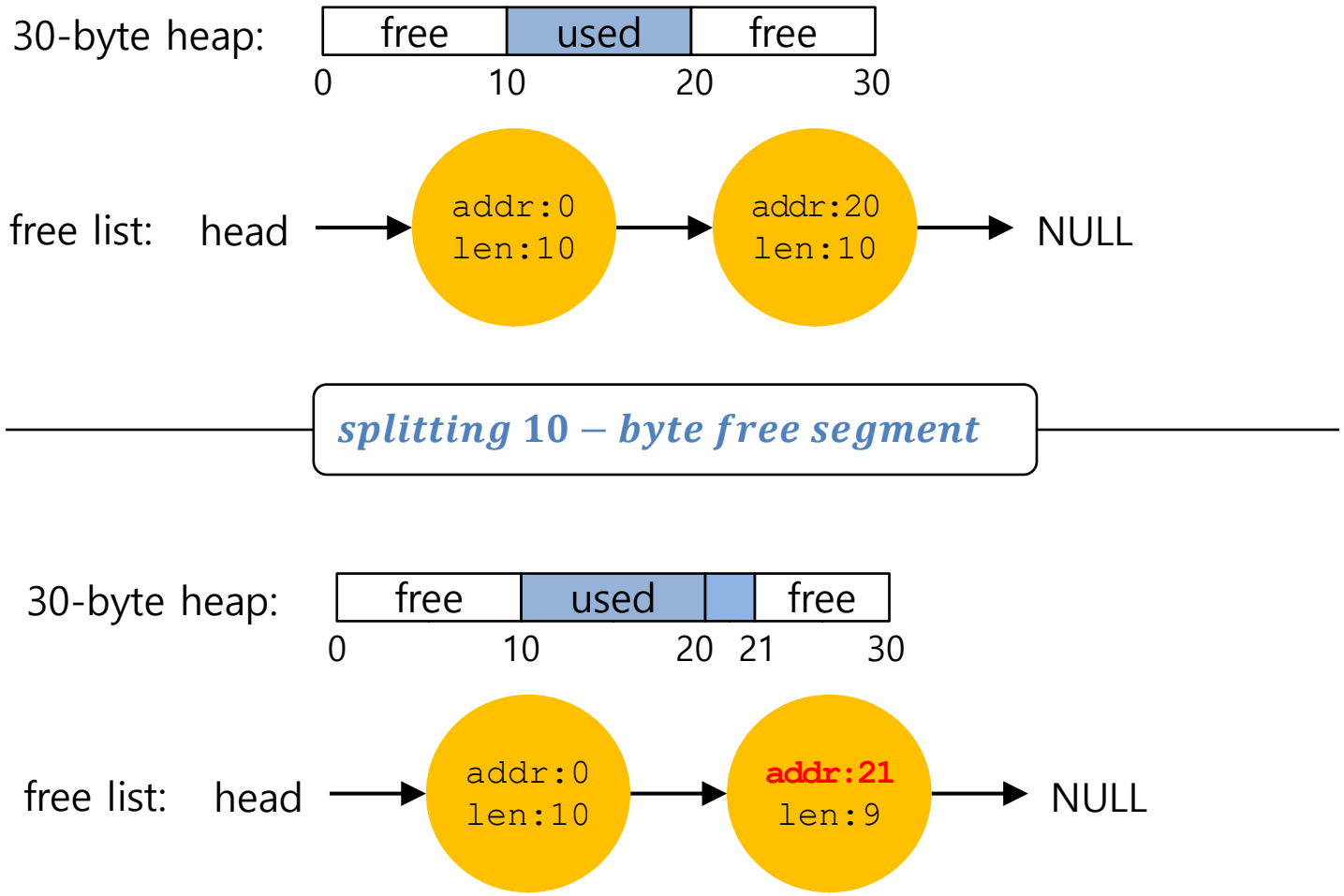
Splitting

- Finding a free chunk of memory that can satisfy the request and splitting it into two
  - When request for memory allocation is **smaller** than the size of free chunks



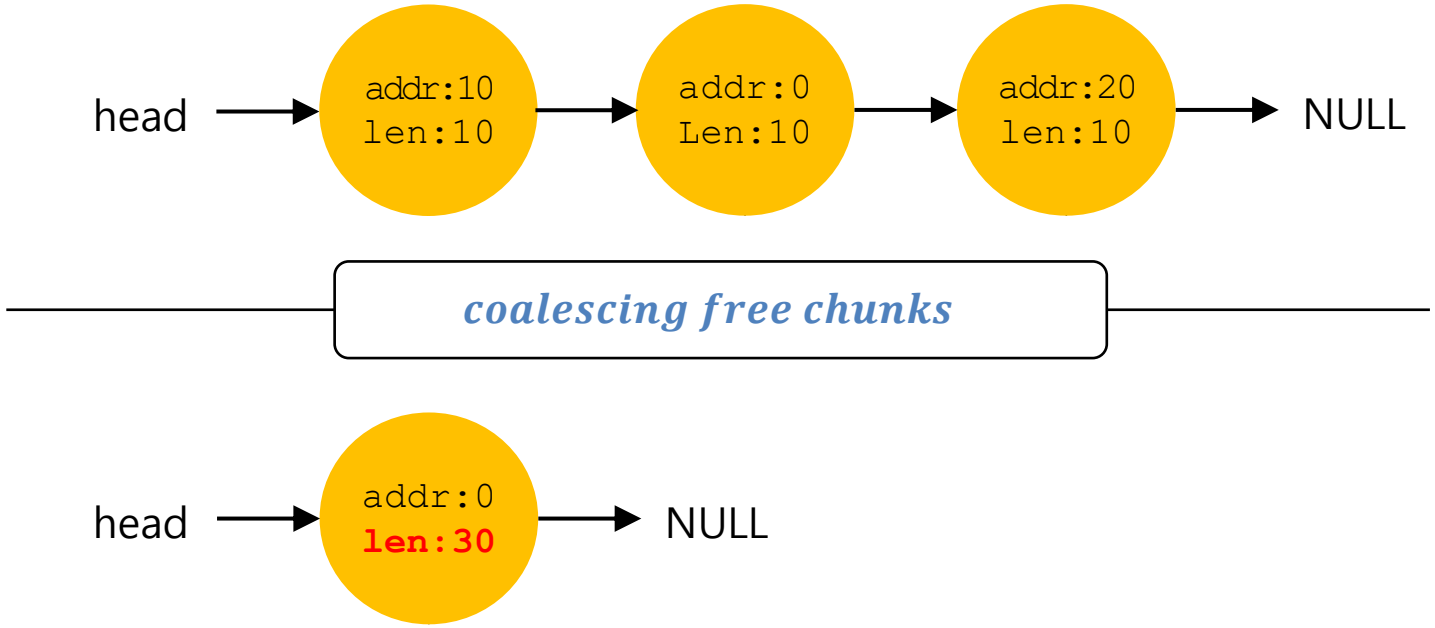
Splitting(Cont.)

- Two 10-bytes free segment with **1-byte request**



Coalescing

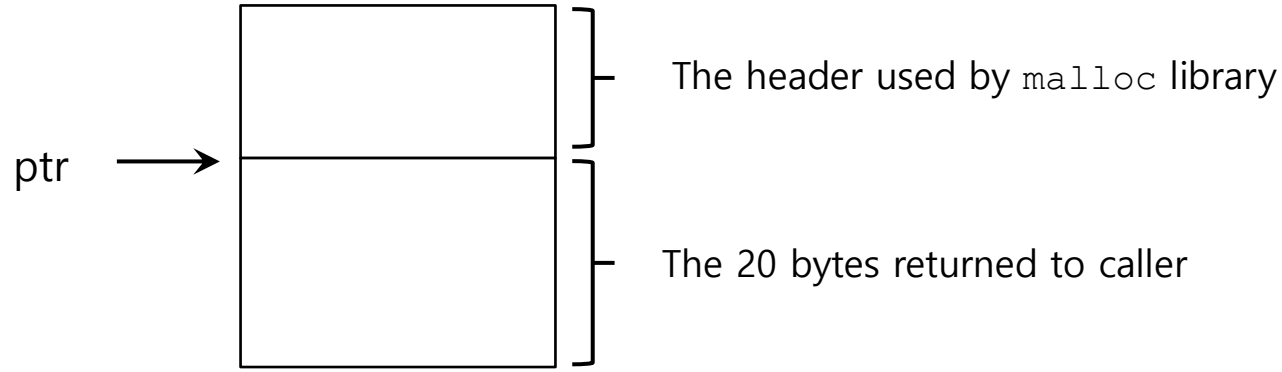
- If a user requests memory that is **bigger than free chunk size**, the list will **not find** such a free chunk
- Coalescing: Merging a free chunk with existing chunks into a large single free chunk if **addresses** of them are **nearby**



Tracking The Size of Allocated Regions

- The interface to `free(void *ptr)` does **not take** a **size parameter**
  - How does the library **know the size** of memory region that will be back **into free list**?
- Most allocators store **extra information** in a **header block**

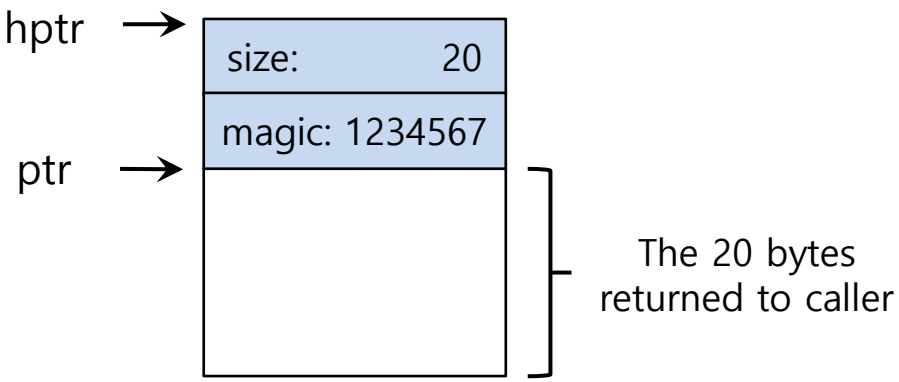
```
ptr = malloc(20);
```



An Allocated Region Plus Header

The Header of Allocated Memory Chunk

- The header minimally **contains the size** of the allocated memory region
- The header may also contain
  - Additional pointers to speed up de-allocation
  - A magic number for integrity checking



Specific Contents Of The Header

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

A Simple Header

The Header of Allocated Memory Chunk(Cont.)

- The **size** for free region is the **size of the header plus the size of the space** allocated to the user
  - If a user **request N bytes**, the library searches for a free chunk of **size N plus the size of the header**

- Simple pointer arithmetic to find the header pointer.

```
void free(void *ptr) {
        header_t *hptr = (void *)ptr - sizeof(header_t);
}
```

Embedding A Free List

- The memory allocation library **initializes** the heap and **puts** the first element of **the free list** in the **free space**
  - The library **can't use** `malloc()` to build a list **within itself**

Embedding A Free List(Cont.)

- Description of a node of the list

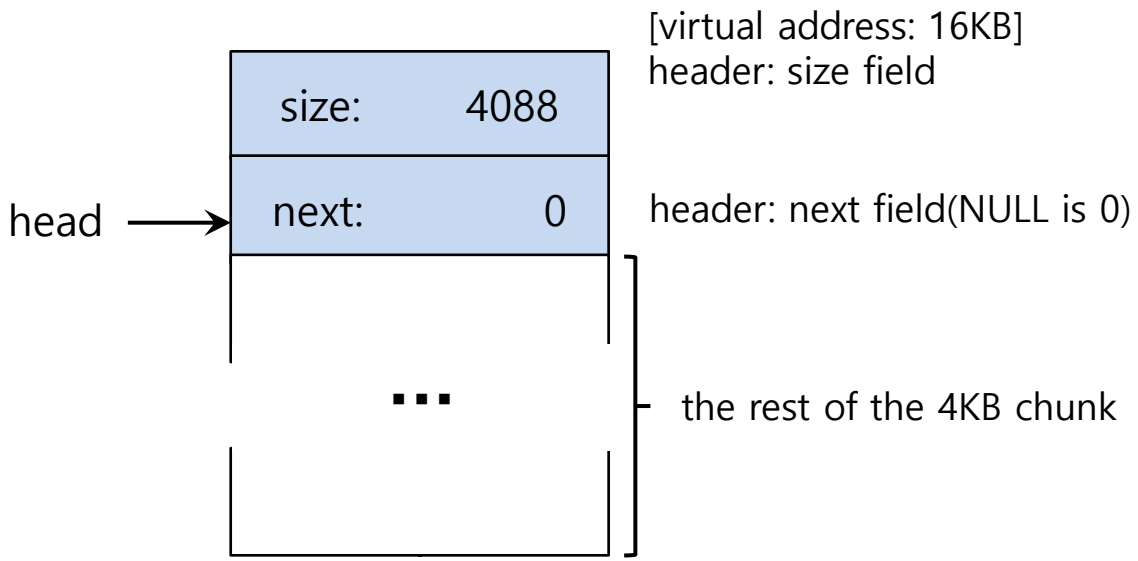
```
typedef struct __node_t {
        int size;
        struct __node_t *next;
} nodet_t;
```

- Building heap and putting a free list
  - Assume that the heap is built with `mmap()` system call.

```
// mmap() returns a pointer to a chunk of free space
node_t *head = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
                                MAP_ANON|MAP_PRIVATE, -1, 0);
head->size = 4096 - sizeof(node_t);
head->next = NULL;
```

A Heap With One Free Chunk

```
// mmap() returns a pointer to a chunk of free space
node_t *head = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
                                MAP_ANON|MAP_PRIVATE, -1, 0);
head->size = 4096 - sizeof(node_t);
head->next = NULL;
```

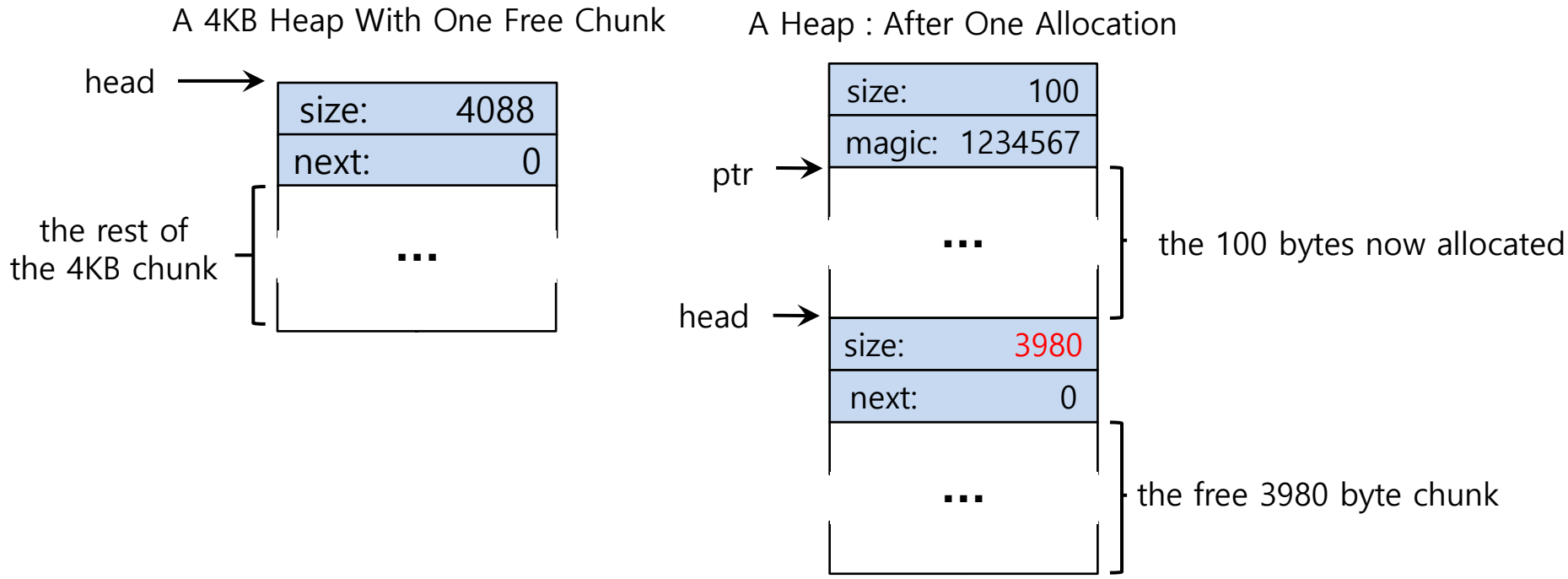


Embedding A Free List: Allocation

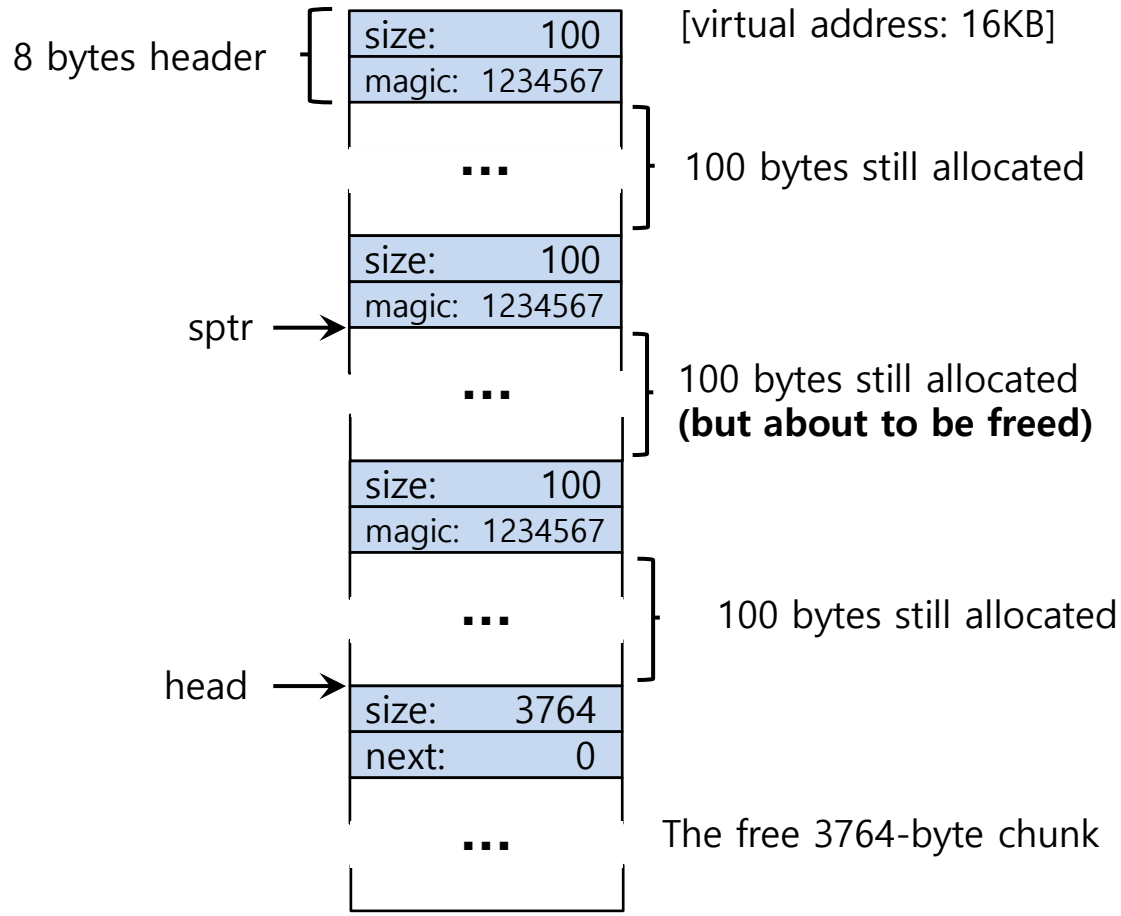
- If a chunk of memory is requested, the library **will first find** a chunk that is **large enough** to accommodate the request
- The library will
  - **Split** the large free chunk into two
    - **One** for the **request** and the **remaining** free chunk
  - **Shrink** the size of free chunk in the list

Embedding A Free List: Allocation(Cont.)

- Example: a request for 100 bytes by `ptr = malloc(100);`
  - Allocating 108 bytes out of the existing one free chunk (note 8 bytes for the header)
  - Shrinking the one free chunk to 3980 bytes (4088 minus 108)



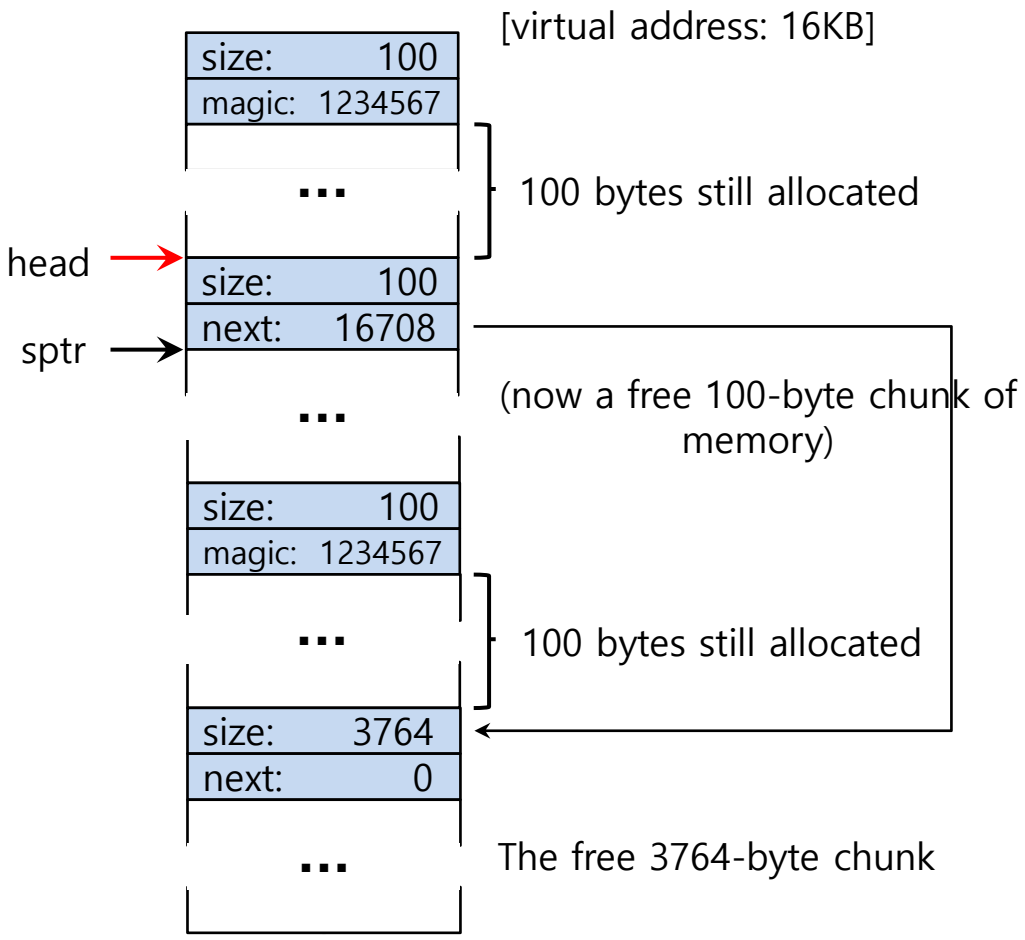
Free Space With Chunks Allocated



Free Space With Three Chunks Allocated

Free Space With `free()`

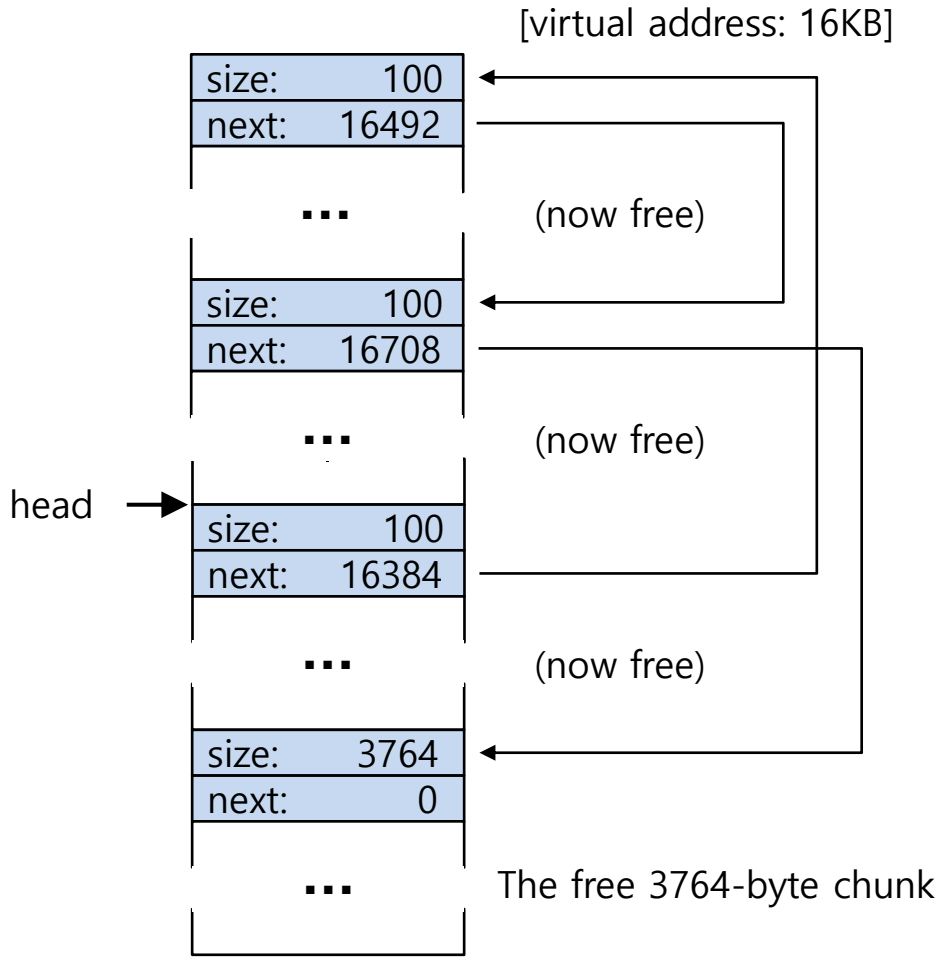
- Example: `free(sptr);`
  - Equivalent to `free(16500)`
    - `16384+108+8`
  - The 100 bytes chunks is **back into** the free list
  - The free list will **start** with **a small chunk**
    - The list header will point the small chunk



Free Space With Freed Chunks

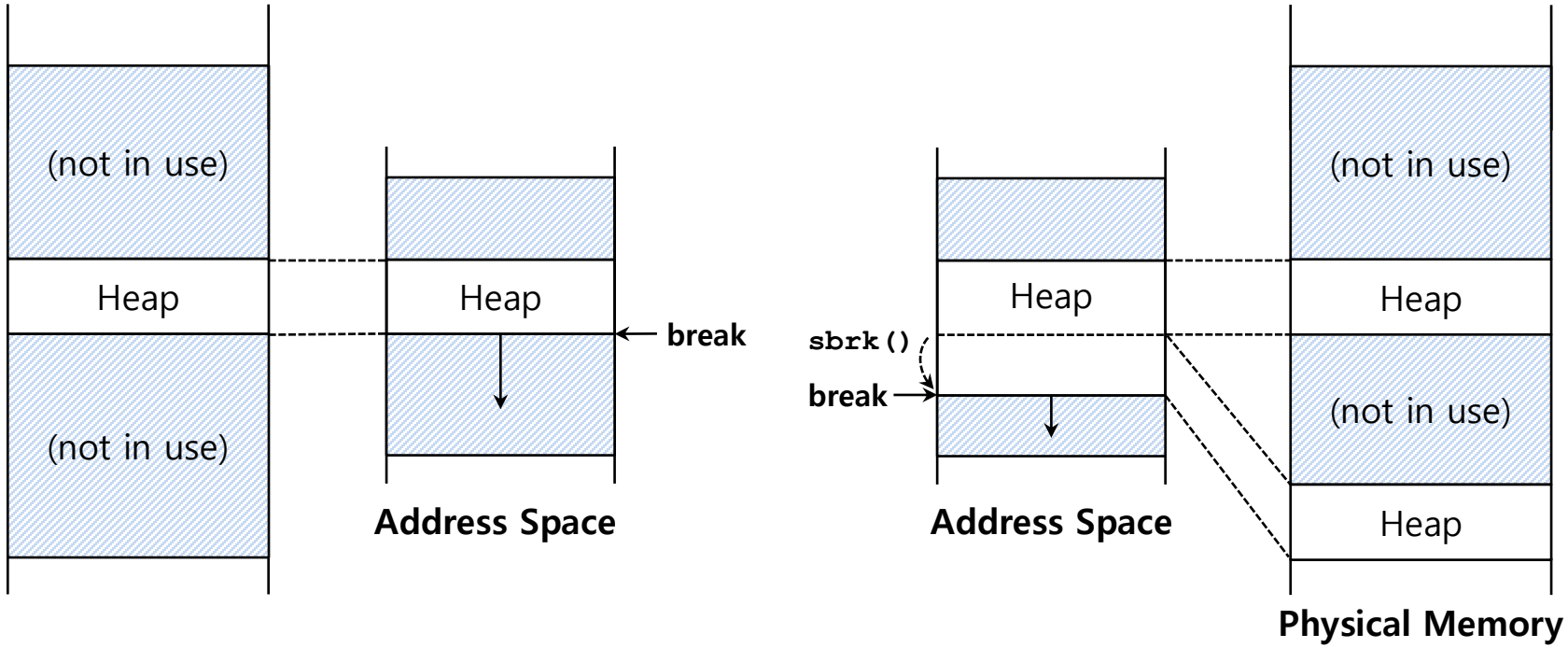
- Let's assume that the last two in-use chunks are freed

- **External Fragmentation** occurs
  - **Coalescing** is needed in the list



Growing The Heap

- Most allocators **start** with **a small-sized heap** and then **request more** memory from the OS when they run out
  - **e.g.,** `sbrk()`, `brk()` **in most UNIX systems**



Managing Free Space: Basic Strategies

- Ideal allocator is both fast and minimizes fragmentation
- **Best Fit**
  - Finding free chunks that are **big or bigger than the request**
  - Returning the **one that is the smallest** chunk **in the group** of candidates
- **Worst Fit**
  - Finding the **largest free chunks** and allocate the requested amount
  - **Keeping the remaining chunk** on the free list

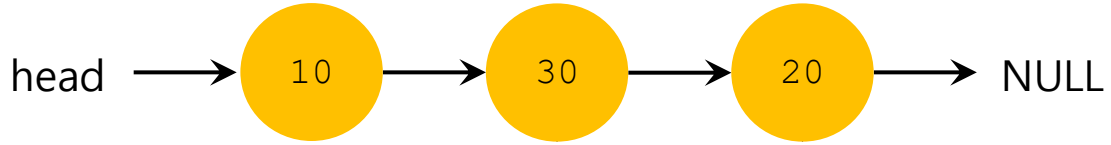
Managing Free Space: Basic Strategies(Cont.)

- **First Fit**
  - Finding the **first chunk** that is **big enough** for the request
  - Returning the requested amount and the remaining free space is kept free for other requests

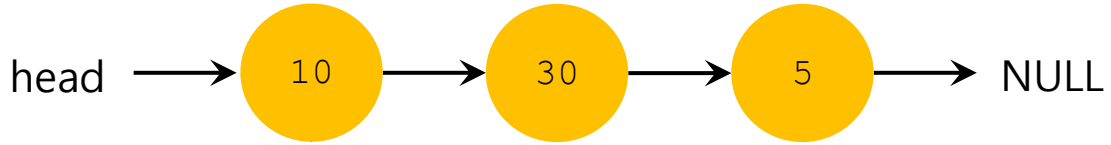
- **Next Fit**
  - Finding the first chunk that is big enough for the request
  - Searching at **where one was looking** at instead of the beginning of the list

Examples of Basic Strategies

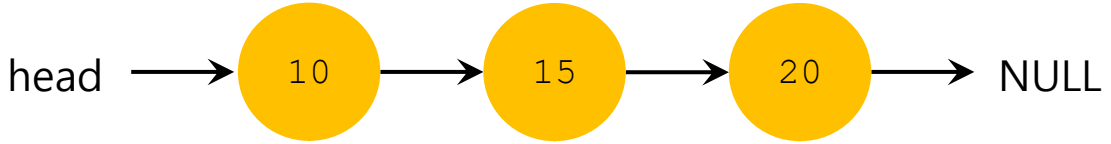
- Allocation request size is 15 bytes



- Result of Best Fit



- Result of Worst Fit



- What about First Fit and Best Fit?

Other Approaches: Segregated Lists

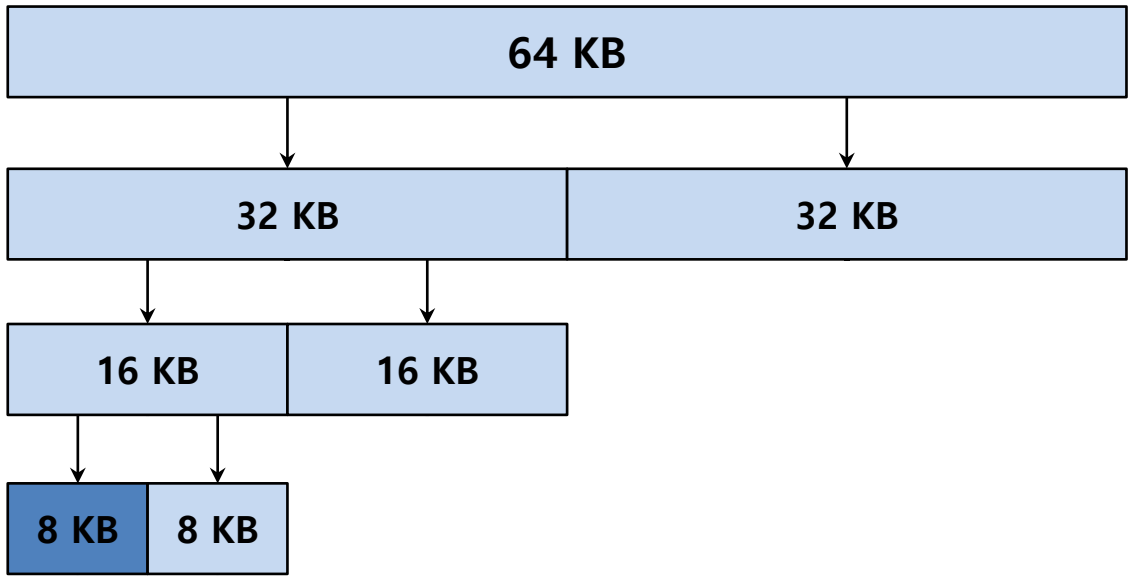
- Segregated Lists
  - Keep dedicated lists for popular request sizes, others are served by the general memory allocator
    - Examples: kernel data structures such as locks, inodes, etc.
  - New complication arises
    - **How much** memory should dedicate to **the pool of memory** that serves **specialized requests** of a given size?
  - **Slab allocator** handles this issue

Other Approaches: Segregated List(Cont.)

- **Slab Allocator**
  - Allocate a number of object caches
    - The objects are likely to be requested frequently
    - e.g., locks, file-system inodes, etc.
  - **Request some memory** from a more general memory allocator when **a given cache is running low** on free space

Other Approaches: Buddy Allocation

- Binary Buddy Allocation – optimized for coalescing
  - When a request is made, search for free space by recursively dividing the total free space(size is power of 2) until a block that is big enough is found
  - The allocator **divides free space** by two **until a block** that is big enough to accommodate the request is **found**



64KB free space for 7KB request

Other Approaches: Buddy Allocation(Cont.)

- Buddy allocation can suffer from **internal fragmentation**
- Buddy system makes **coalescing** simple
  - **Coalescing** two blocks into the next level of block
- Other approaches use more advanced data structures to address scaling and multiprocessor systems