

Operating Systems

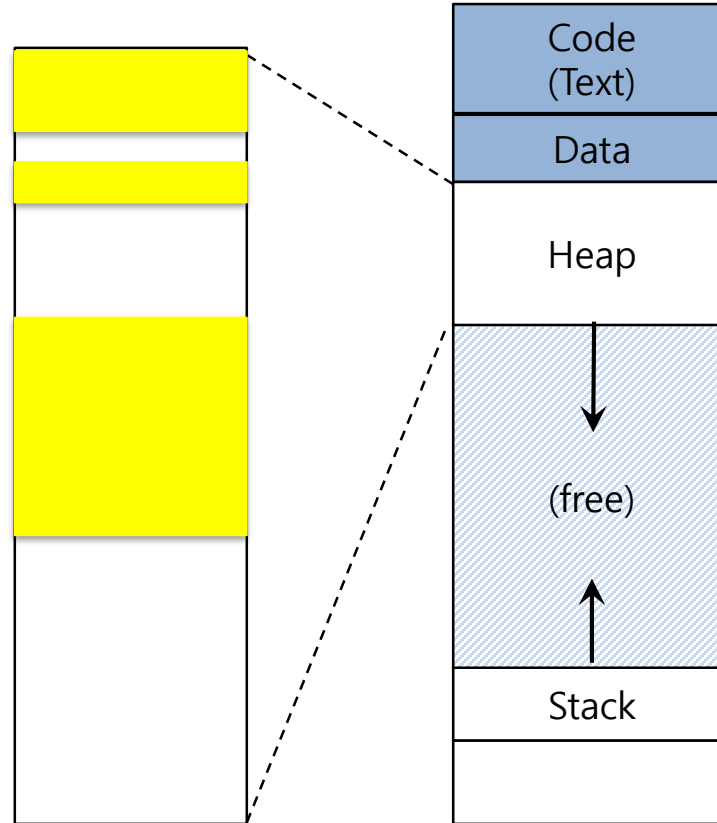


17. Free-Space Management

Managing heap

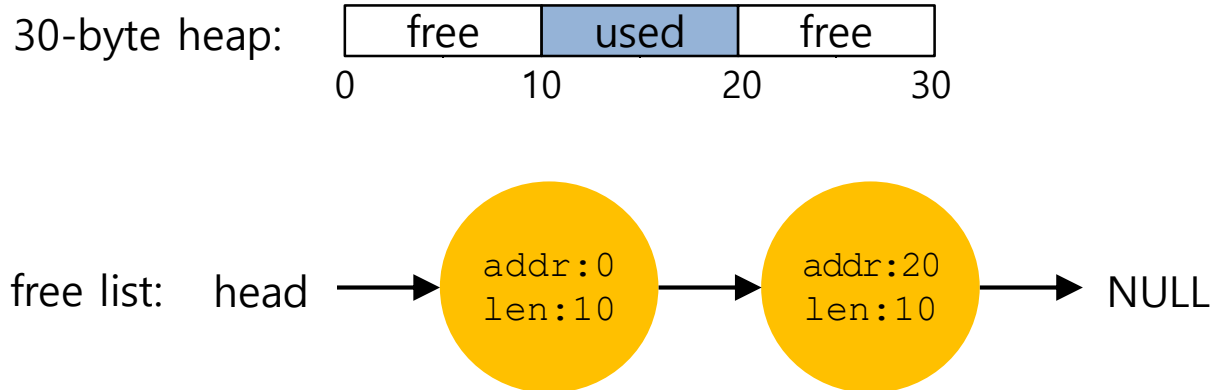
`malloc()`
`free()`
in **libc**

To manage the
memory in a
heap



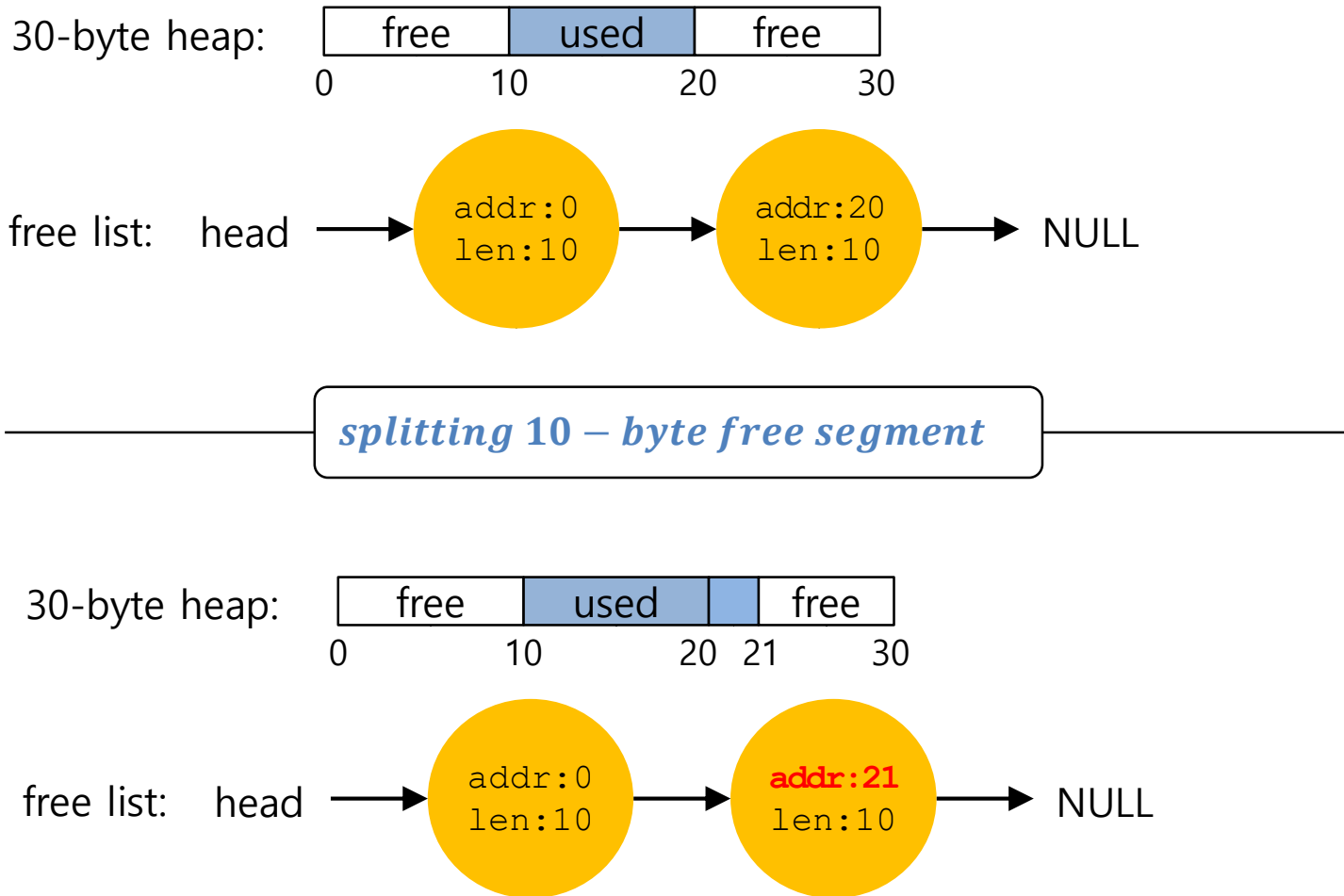
Splitting

- ▣ 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.



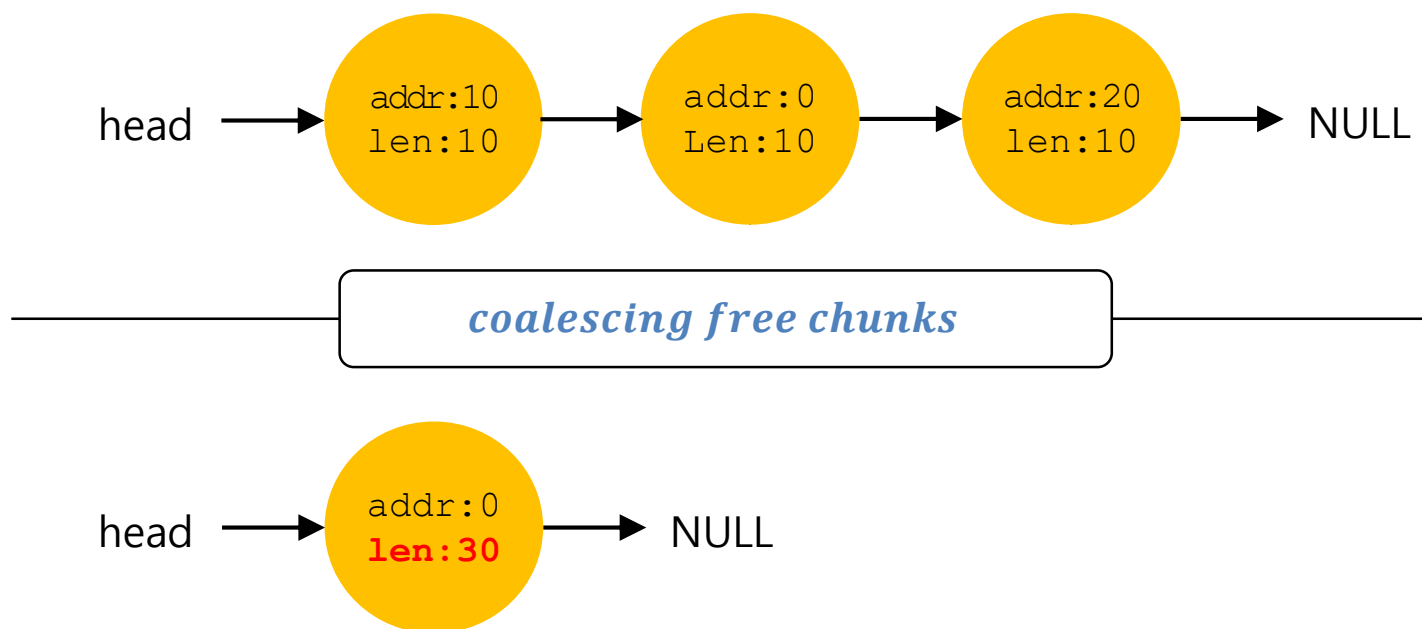
Splitting(Cont.)

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



Coalescing

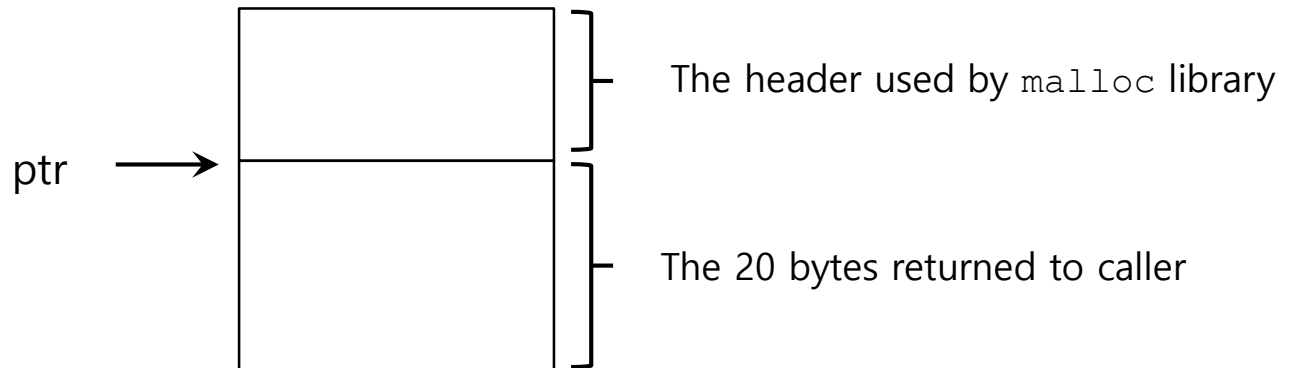
- ❑ If a user requests memory that is **bigger than free chunk size**, the list will **not find** such a free chunk.
- ❑ Coalescing: **Merge** returning a free chunk with existing chunks into a large single free chunk if **addresses** of them are **nearby**.



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?

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

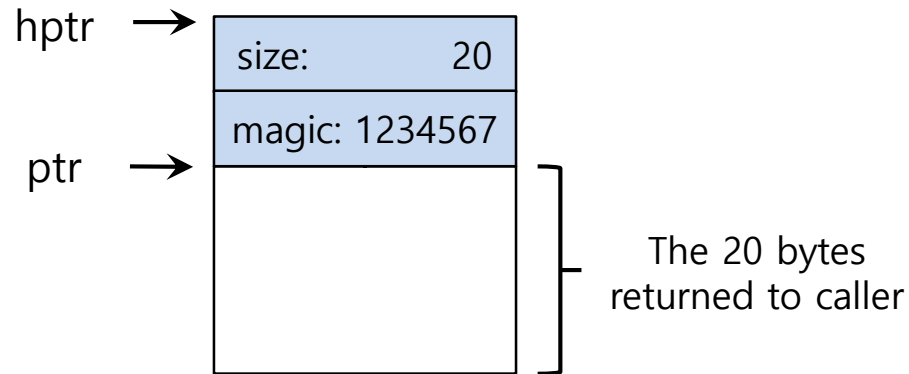


An Allocated Region Plus Header

The Header of Allocated Memory Chunk

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

Actual chunk size of malloc(N) = N + size of header
Here, 28 Byte



The Header of Allocated Memory Chunk(Cont.)

```
void free(void *ptr) {  
    header_t *hptr = (void *)ptr - sizeof(header_t) ;  
    ...  
    assert(hptr->magic==1234567) ;  
    ...  
}
```

Embedding A Free List

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

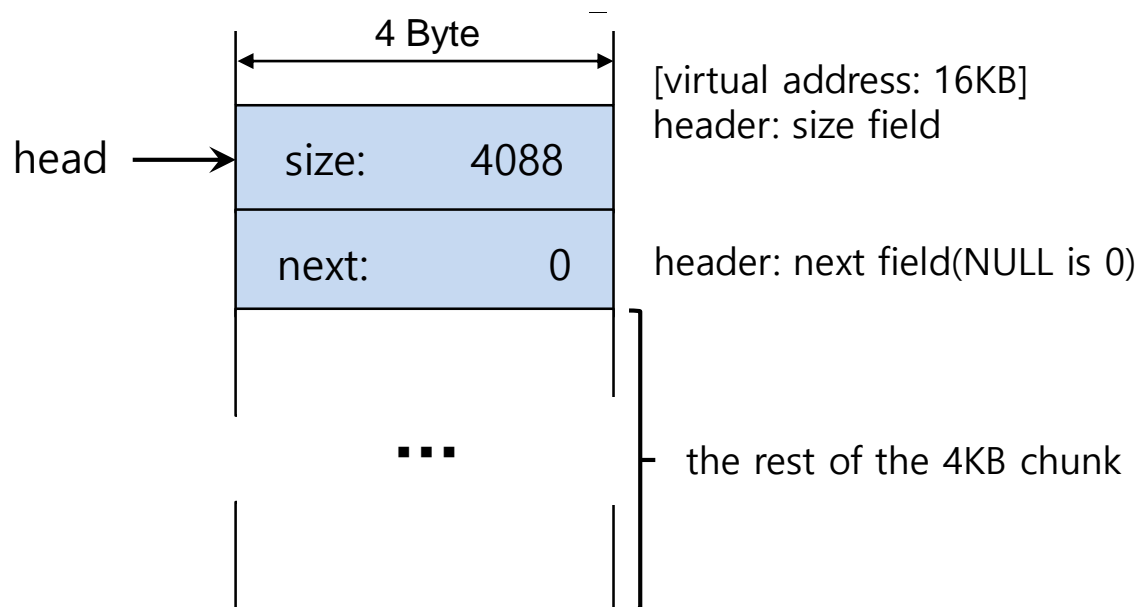
Heap Initialization

// 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;
```



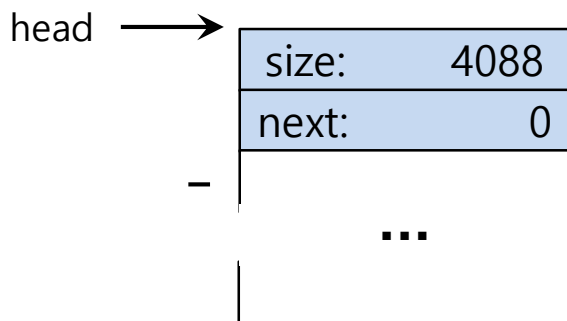
Embedding A Free List: Allocation

- ❑ 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.

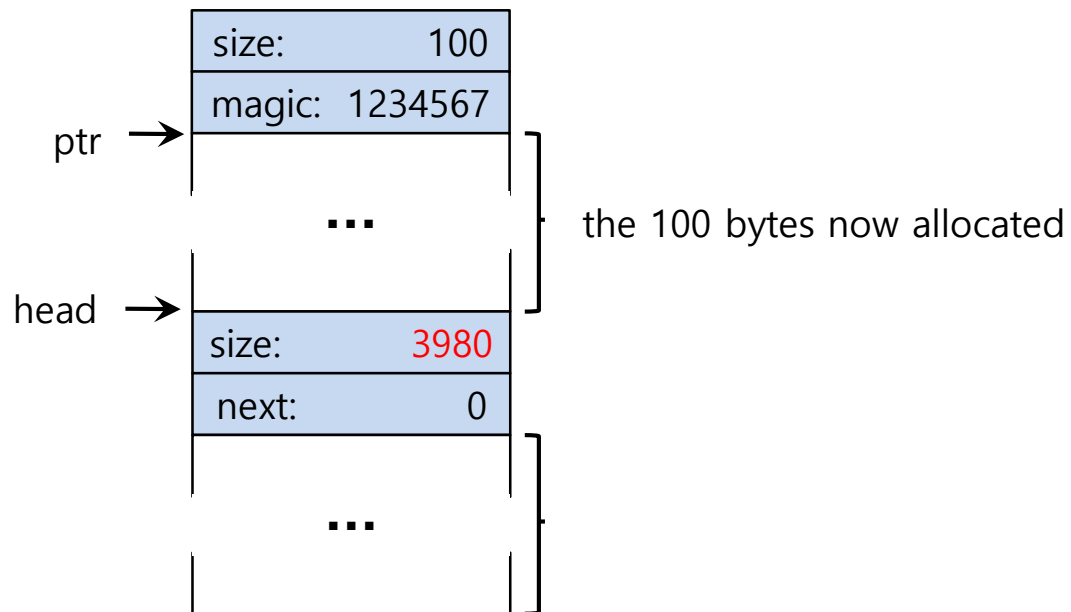
Embedding A Free List: Allocation(Cont.)

- Example: a request for 100 bytes by `ptr = malloc(100)`
→ 108 byte is returned.

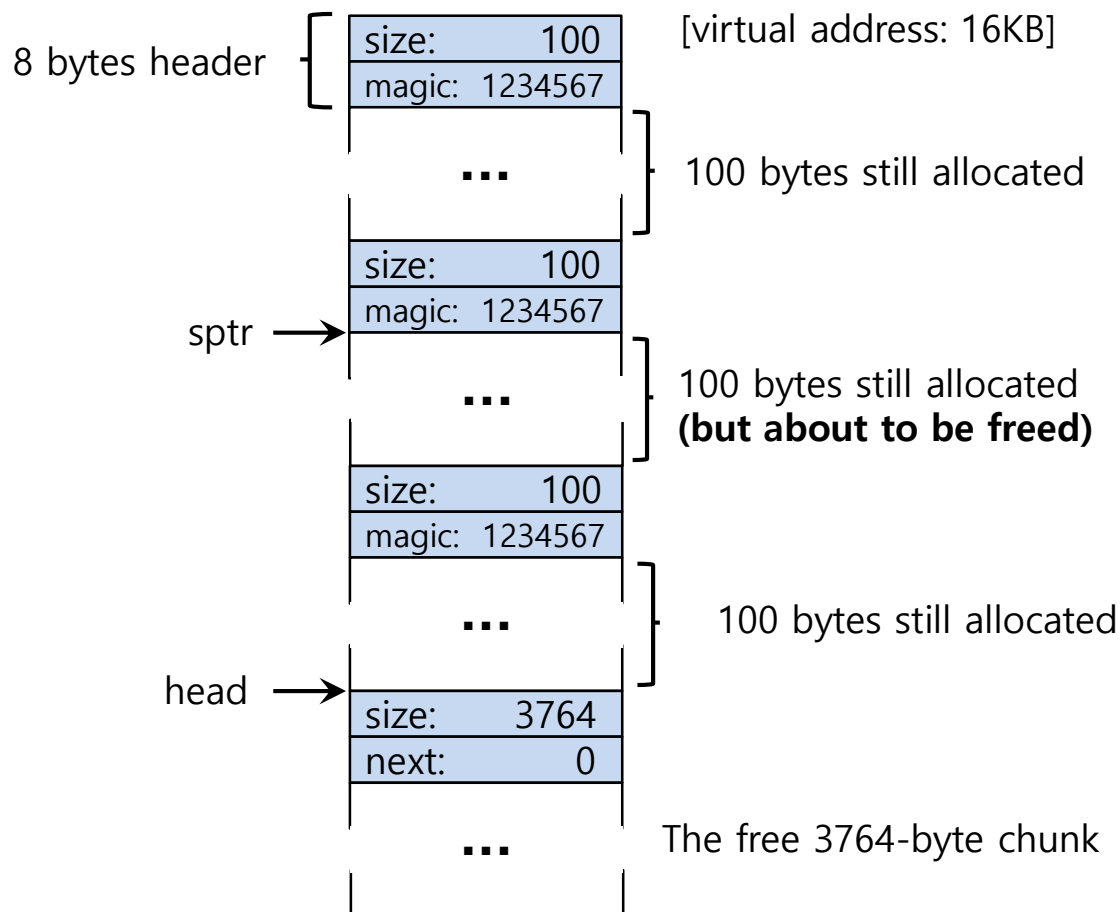
A 4KB Heap With One Free Chunk



A Heap : After One Allocation



Free Space With Chunks Allocated



Free Space With Three Chunks Allocated

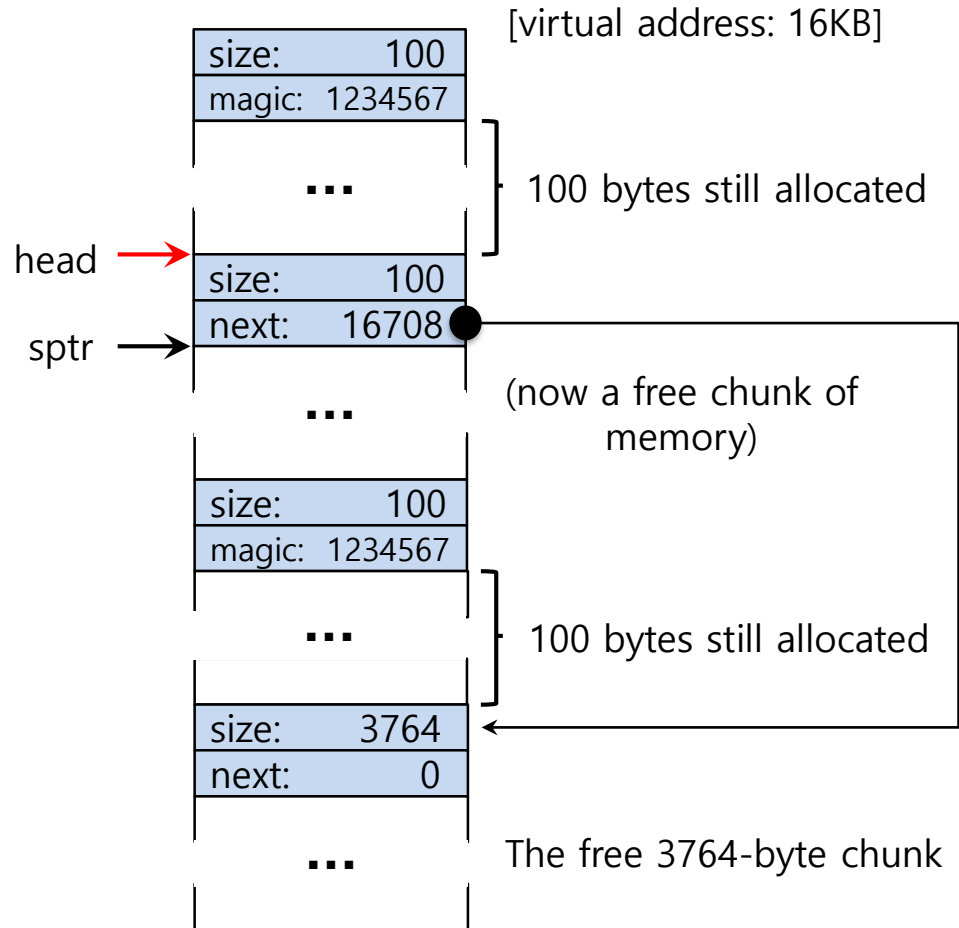
Free Space With `free()`

▣ `free(sptr)`

```
void* tmp = head ;
```

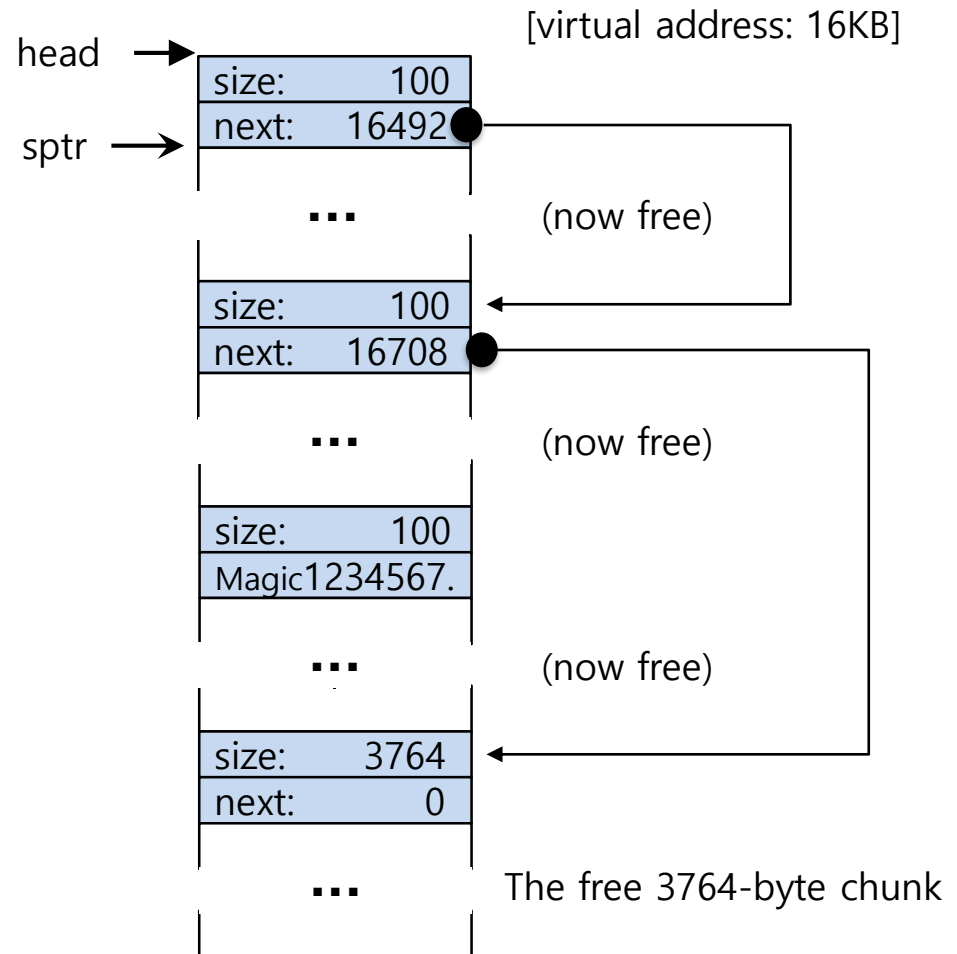
```
head = sptr ;
```

```
head->next = tmp ;
```



Free Space With Freed Chunks

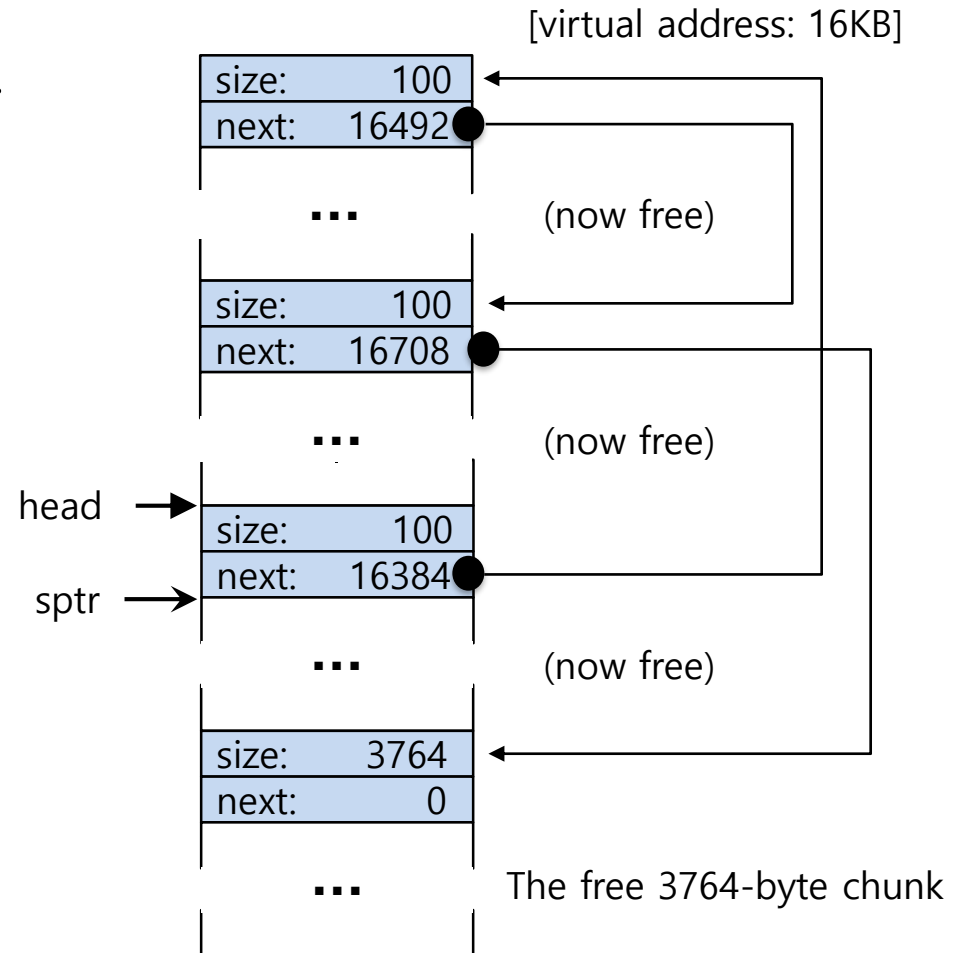
▣ `free(sptr)`



Free Space With Freed Chunks

- free(sptr)

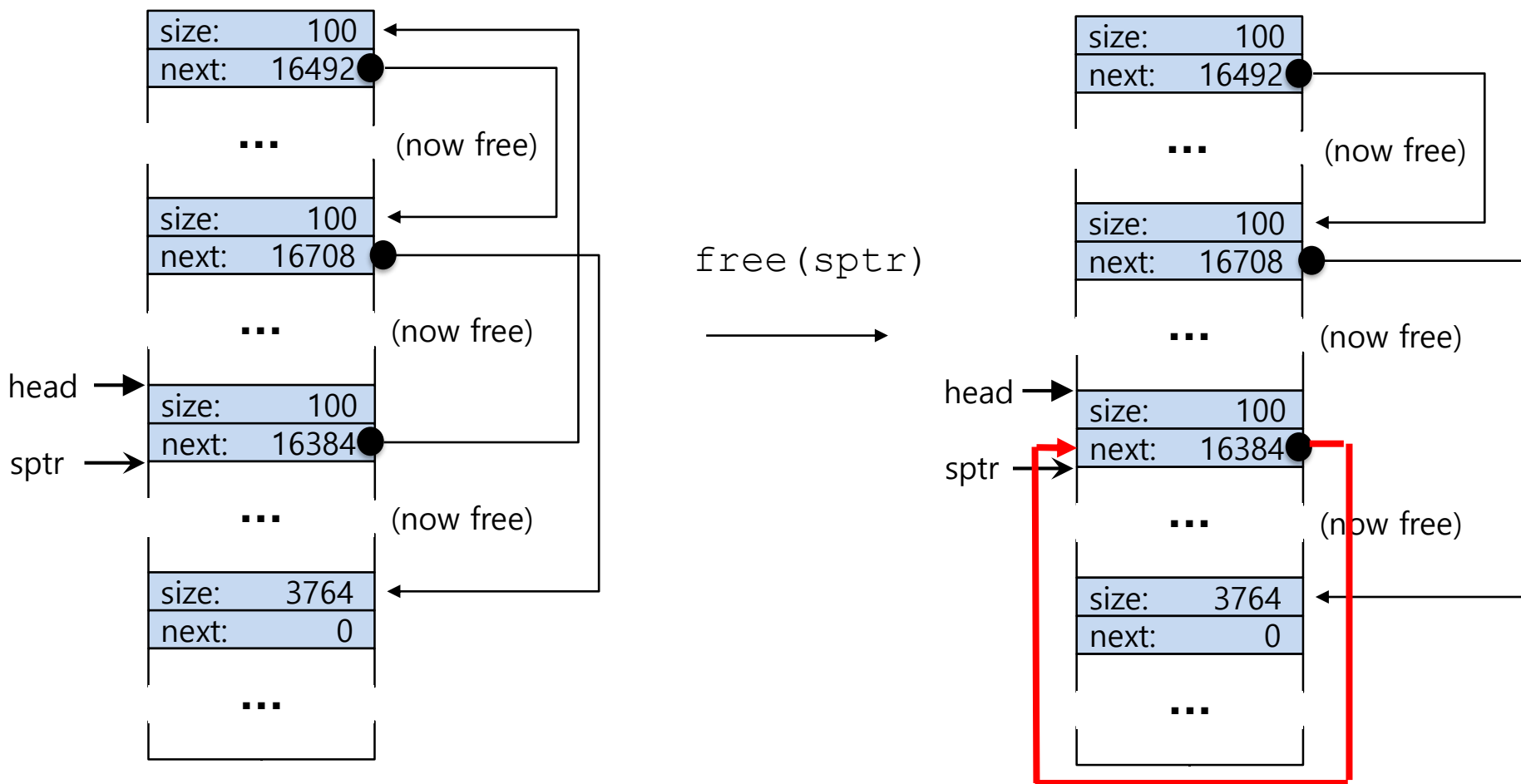
- Coalescing** is needed in the list.



Double free

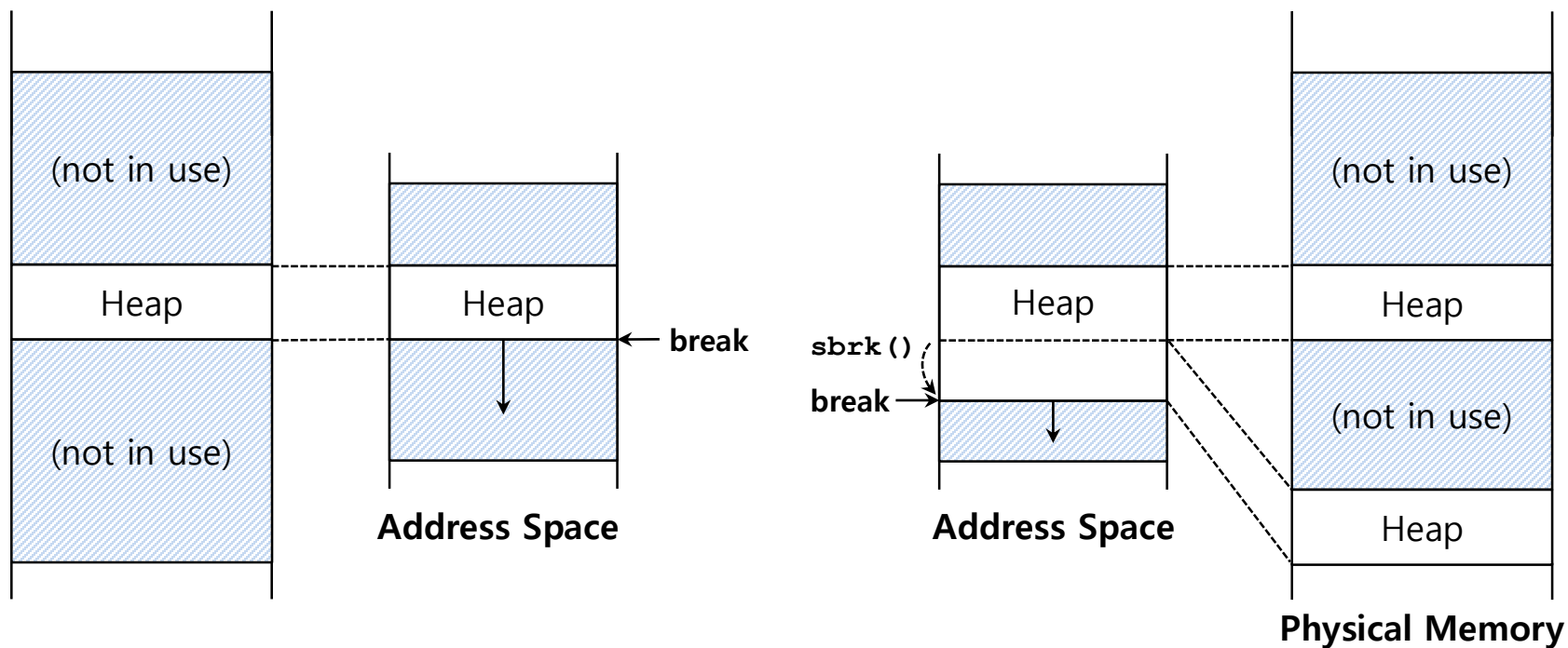
Free(sptr): dangling reference

[virtual address: 16KB]



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

□ Best Fit:

- ◆ Finding free chunks that are **big or bigger than the request**
- ◆ Returning the **one of smallest** in the chunks **in the group** of candidates

□ Worst Fit:

- ◆ Finding the **largest free chunks** and allocation the amount of the request
- ◆ **Keeping the remaining chunk** on the free list.

Managing Free Space: Basic Strategies(Cont.)

▣ First Fit:

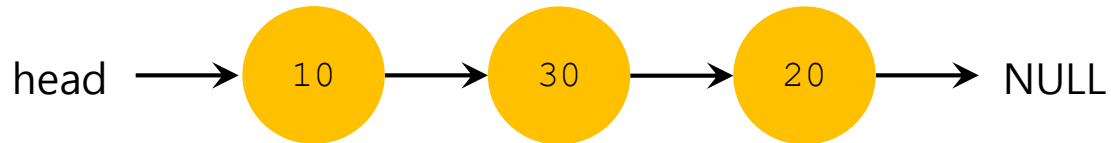
- ◆ Finding the **first chunk** that is **big enough** for the request
- ◆ Returning the requested amount and remaining the rest of the chunk.

▣ Next Fit:

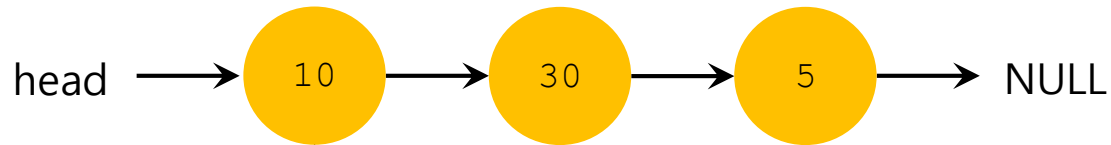
- ◆ Finding the first chunk that is big enough for the request.
- ◆ Searching at **where one was looking** at instead of the beginning of the list.

Examples of Basic Strategies

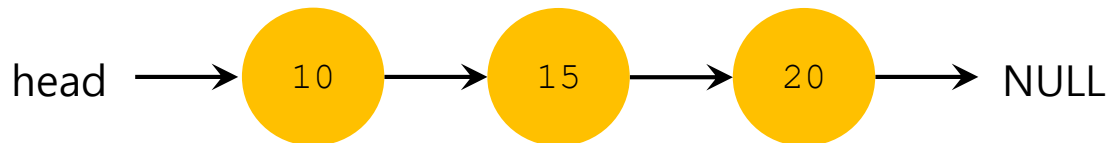
- Allocation Request Size 15



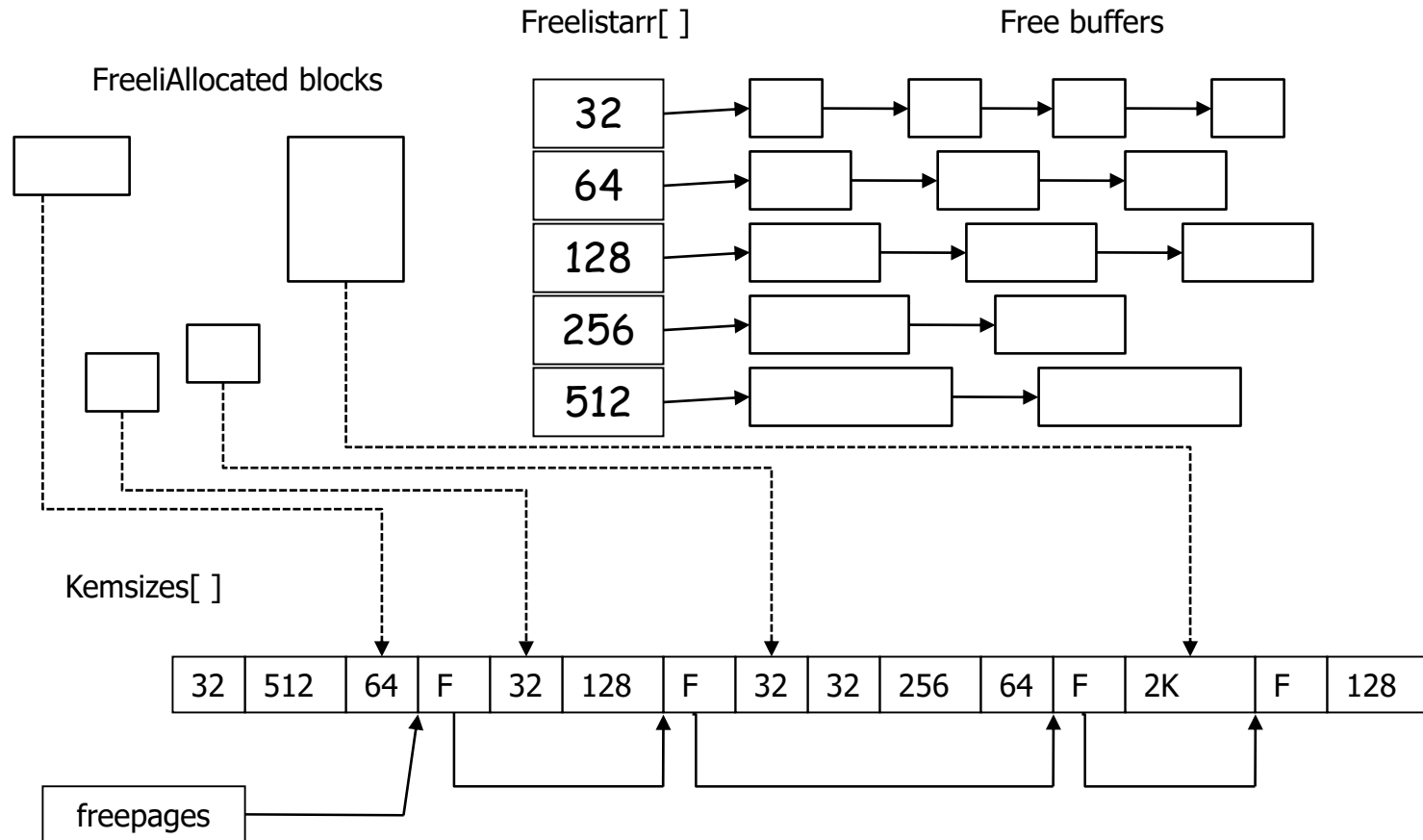
- Result of Best-fit



- Result of Worst-fit



Segregated List: McKusick-Karels Allocator (4.3 BSD)



McKusick-Karels Allocator

- ▣ Used in 4.3BSD and Digital UNIX.
- ▣ Management
 - ◆ Contiguous set of pages
 - ◆ Same buffer size within a page
- ▣ Manage pages by page usage array(`kmemsizes[]`).
 - ◆ Element of `kmemsizes`: free | size of buffer
- ▣ `malloc()` allocates the requested sized buffer from the free list.
- ▣ `malloc()` invoke a routine that acquires free page and splits by a requested size if free list is empty.
- ▣ How much memory should dedicate to the pool of memory that serves specialized requests of a given size?

Buddy System

- ❑ Create the small buffers by repeatedly halving a large buffer and coalesce the adjacent free buffers.
- ❑ When a buffer is split, each half is called the buddy of the other.
- ❑ Minimum size is 32byte.
- ❑ Use a bitmap to monitor each 32-byte chunk of the block.

Detailed step

- ▣ Allocate 256 byte
 - ◆ Split the block into A and A'. Puts A' in the 512 byte free list.
 - ◆ Split A into B and B'. Put B' in the 256 byte free list.
 - ◆ Return B to the client
- ▣ Allocate 128 Byte
 - ◆ Split B' into C and C'. Put C' in the free list.
 - ◆ Return C to the client.
- ▣ Allocate 64 byte.
 - ◆ Split c' into D and D'. Put D' in the free list.
 - ◆ Return D to the client.
- ▣ Allocate 128
 - ◆ Split A' into E and E'. Put E' in the free list of 256 byte
 - ◆ Split E into F and F'. Put F' in the free list of 128 byte
 - ◆ Allocate F.

Example

- Maintain freelist for every buffer size(32-512).

bitmap

1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

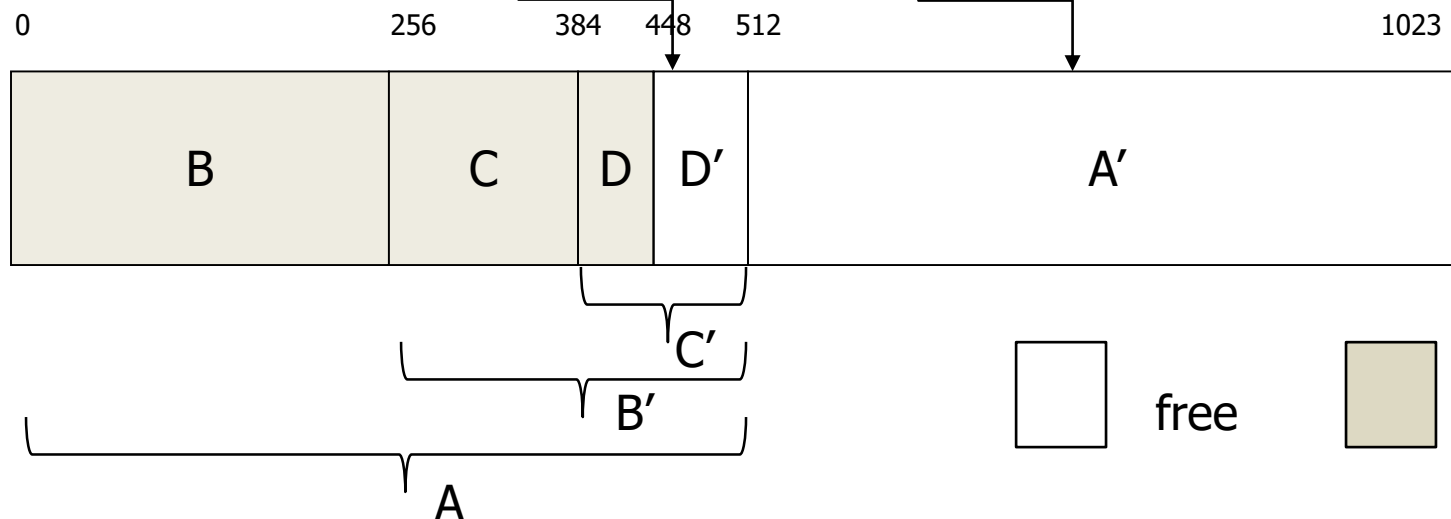
Free list headers

32	64	128	256	512
----	----	-----	-----	-----

Allocate(256)

Allocate(128)

Allocate(64)

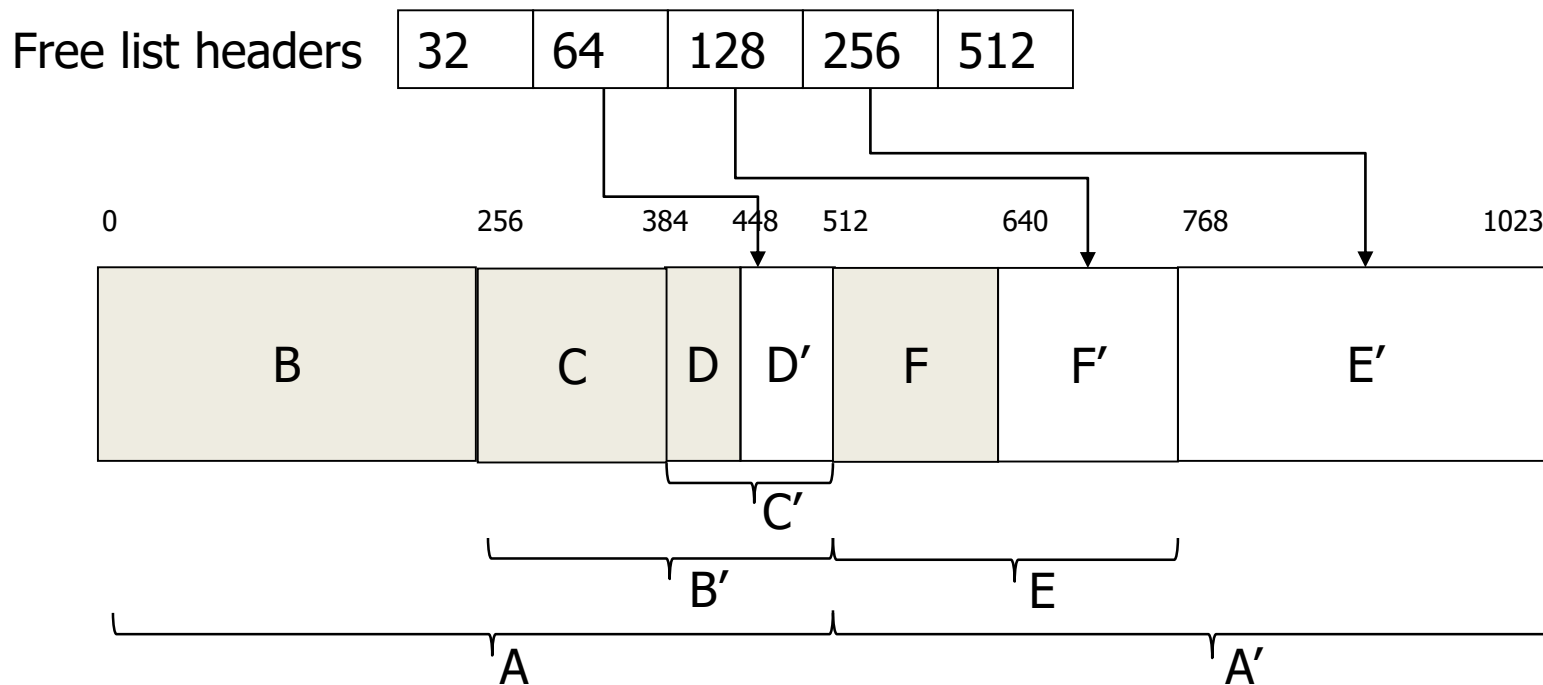


Example

bitmap

1	1	1	1	1	1	1	1	1	0	0	0	0	1	1	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Allocate(128)

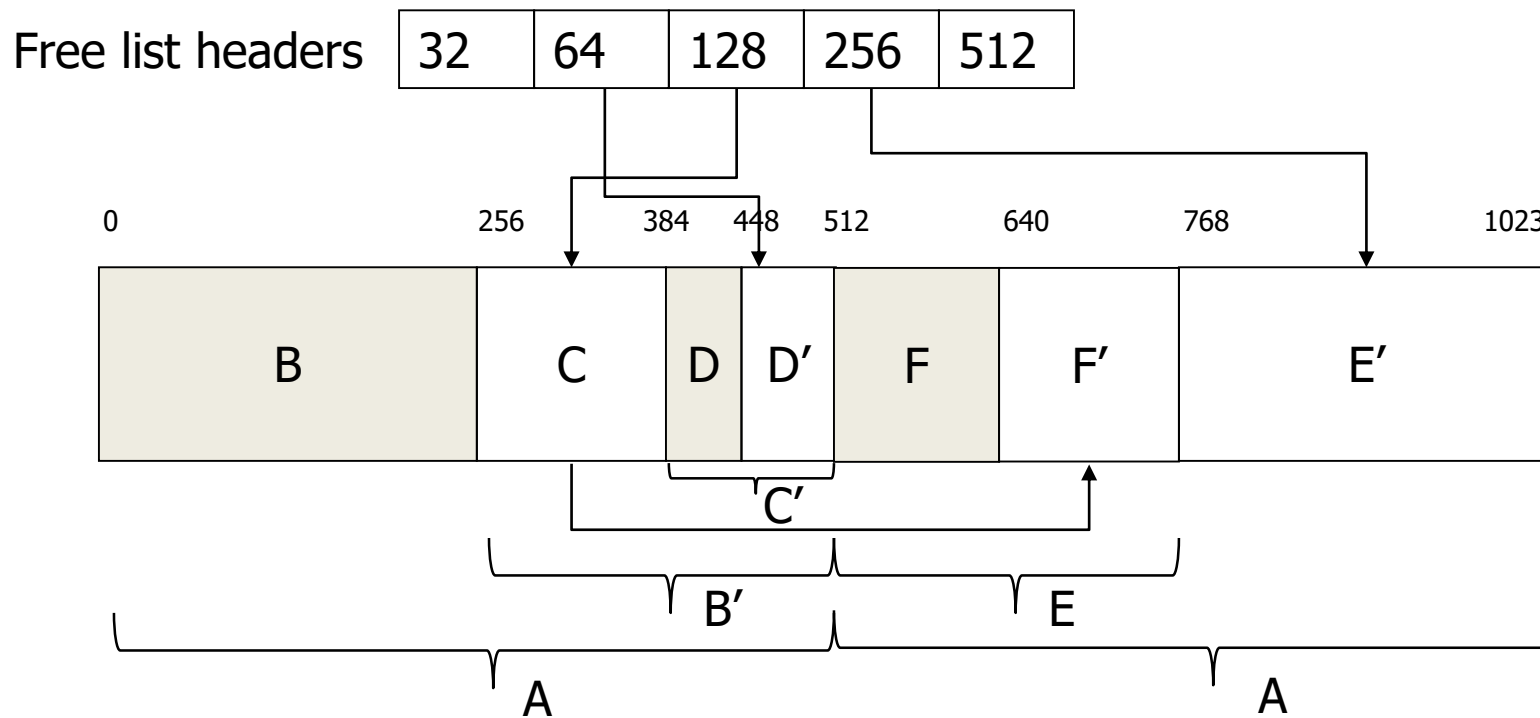


Example

bitmap

1	1	1	1	1	1	1	1	1	0	0	0	0	1	1	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Release(C, 128)

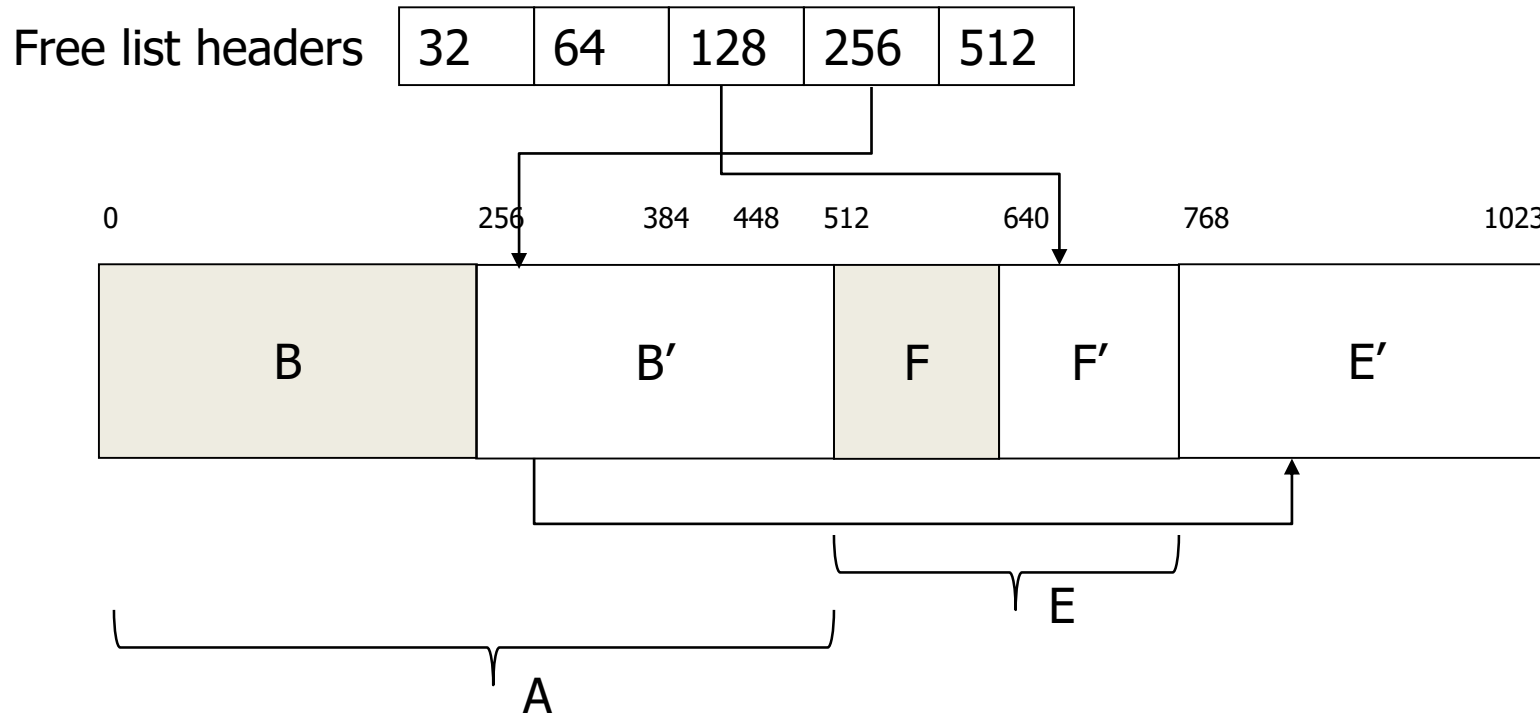


Example

bitmap

1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Release(D, 64)



Analysis

- Characteristic
 - Internal fragmentation by power-of-two allocation
 - Easy to find a buddy of a buffer by address and size
 - Use bitmap for coalescing.
- Advantage
 - Does a good job of coalescing adjacent free buffers.
 - Easy exchange of memory between the allocator and the paging system
- Disadvantage
 - Performance degrade: every time a buffer is released, the allocator tries to coalesce as much as possible.
 - Release routine needs both the address and size of the buffer.
 - Partial release is insufficient.

Slab allocator

- ❑ Advanced form of segregated list
- ❑ Slab: a set of kernel pages
 - ◆ Consists of same objects, e.g. inodes, locks, sockets
 - ◆ They are all initialized before allocation.
- ❑ Using object
 - ◆ Allocate memory and Construct the object
 - ◆ Use the object.
 - ◆ Deconstruct it, Free the memory.
- ❑ Construction
 - ◆ Initialization of various fields
- ❑ Object reuse
 - ◆ It is better to reuse possible data structures rather than release

Memory allocation

- ▣ libc
 - ◆ Based on linked list
- ▣ kernel
 - ◆ Buddy: allocating memory to process
 - ◆ Slab: allocating memory for small kernel objects (proc structure, inode, socket and etc.)

