Free-Space Management

Prof. Yongtae Kim

Computer Science and Engineering
Kyungpook National University

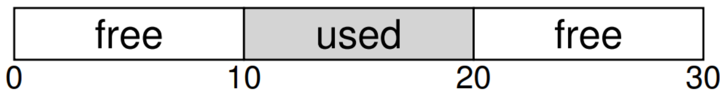
External Fragmentation

→ 가변 size의 메모리를 할당할 경우.

- **Free-space management becomes more difficult when the free space consists of variable-sized units and this arises in:**
  - A user-level memory-allocation library (`malloc()` & `free()`) heap 영역 (항상 10KB를 할당하는게 X.)
  - an OS managing physical memory when using segmentation for virtual memory

- **In either case, the problem is external fragmentation**
  - The external fragmentation is that the free space get chopped into little pieces of different sizes and is thus fragmented
  - Subsequent request may fail because there is no single contiguous space that can satisfy the request, although total free space exceeds the requested size

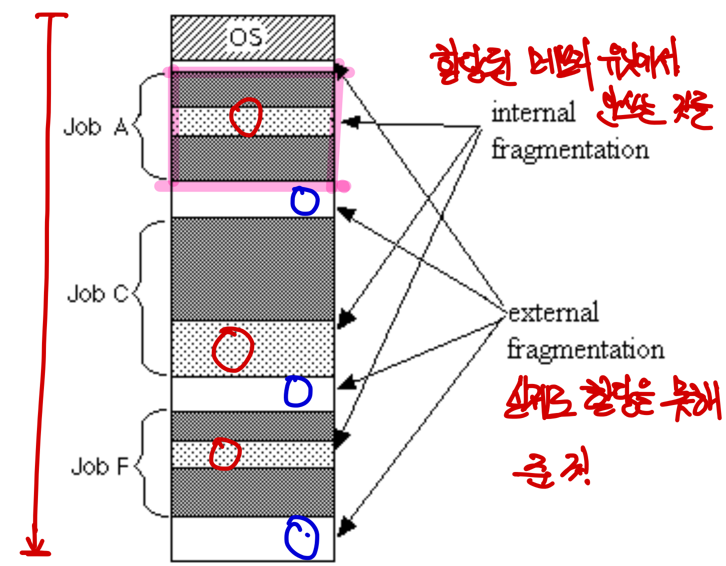
→ 15B를 할당이 불가능한 상황



  - e.g.) total space in above figure is 20 bytes; unfortunately, it is fragmented into two chunks of size 10 bytes each
    ➔ As a result, a request for 15 bytes will fail despite 20 bytes free

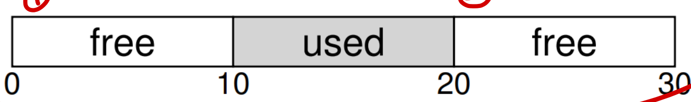
Assumptions

- **We assume a basic interface of `malloc()` & `free()`**
  - The space that the library manages is known historically as heap
  - linked-list로 관리
  - The generic data structure to manage free space in heap is a kind of free list
- **We assume that our primary concern is external fragmentation**
  - 차이
  - Allocators could also have the problem of internal fragmentation
  - Any unasked space in an allocated chunk is considered internal fragmentation; the waste occurs inside the allocated unit; which is another space waste
- **We assume that once memory is handed out to a client, it "cannot be" relocated to another location memory**
  - No compaction of free space is possible
- **We assume that allocator manages a contiguous region of bytes**
  - The region is a single fixed size → 변하지 않음.



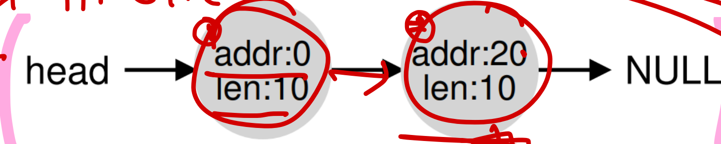
Low-Level Mechanisms: Splitting

- **A free list contains a set of elements that describe the free space remaining in the heap; let's consider a 30-byte heap**



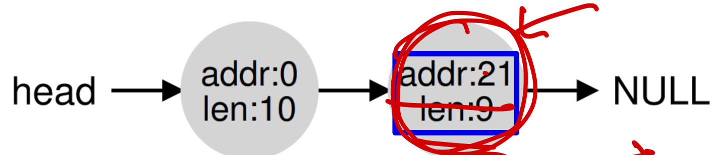
malloc 라이브러리가 관리해줌

2개



  – A request for ≤ 10 bytes will succeed; otherwise, it will fail (returning **NULL**)

- **What happens if the request is for something < 10 bytes?**
  – The allocator will perform splitting; it will find a free chunk and split it into "two"
  – The first chunk will return to the caller and the second will remain on the list
  – If a request for 1 byte were made, the call to **malloc()** would return the address 20 and the list will be:

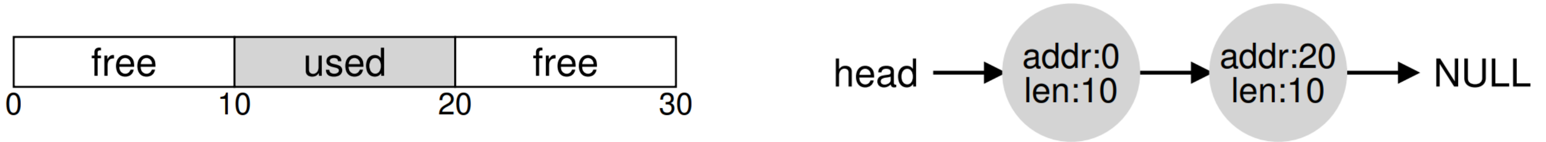


  – As can be seen, the list basically stays intact; the only change is that the free region now starts at 21, instead of 20, and the length is now 9

Low-Level Mechanisms: Coalescing

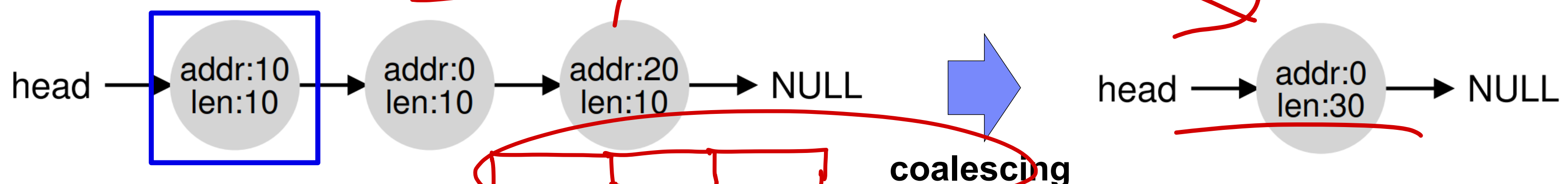
합치.

- **A corollary mechanism is known as coalescing of free space**
  - Coalescing is to merge returning a free chunk with existing chunks into a larger single free chunk if addresses of them are nearby.
  - Let's consider the same example



- **What happens if an application calls `free()` to free the chunk?**
  - If we simply add this free space back into the list, you will see the list with three chunks (left)
  - To avoid this problem, the allocator coalesces free space when a chunk of memory is freed (right)

15KB 할당 불가.

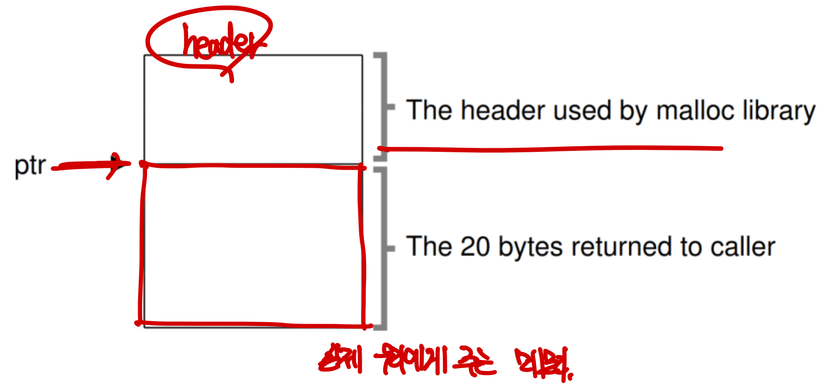


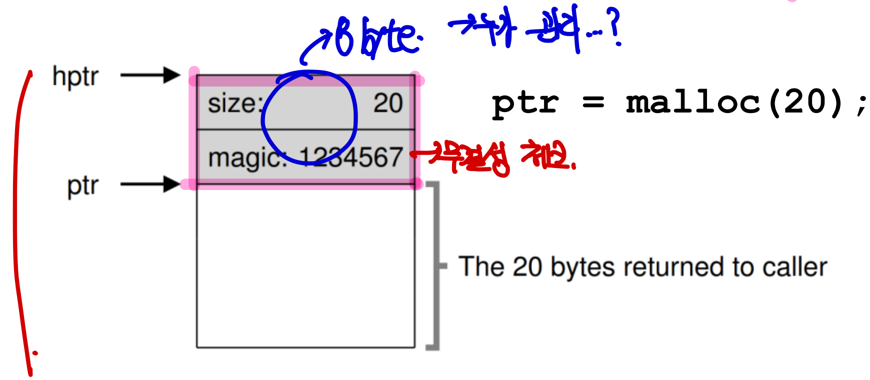
Tracking the Size of Allocated Regions

int *p = malloc(10)
free(p) # 10byte free

누군가가 size 정보를 가지고 있다는 뜻.

- **Inference to `free(void *ptr)` does not take size parameter**
  - The library can quickly determine the size of the region of memory being freed
  - To accomplish this, allocators store a little bit of extra information in header






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

  - The header contains the size of the allocated region and a magic number to offer additional integrity checking, and other information

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

void free(void *ptr) {
    header_t *hptr = (header_t *) ptr - 1;
    ...
    assert(hptr->magic == 1234567)
```

  - Therefore, when a user request (N) bytes of memory, the library searches for a free chunk of size (N + header size)

실제 메모리는 header size도 추가 할당.

Embedding a Free List: Initialization

4kB chunk
virtual memory 상 16kB

- **How do we build such a list inside the free space itself?**
  - Just **`malloc()`**? ➔ No! you can't do this

free-list
4096

- **Assume we have a 4096-byte chunk of memory to manage**
  - To manage this as a free list, we first have to initialize the list
  - The list should have only one entry of size 4096 (minus the header size)
  - We assume that the heap is built via a call to the system call **`mmap()`**

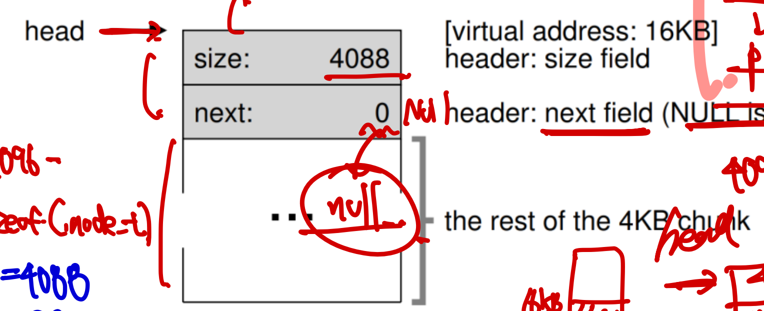
4kB heap의 mmap이 주어진

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

// 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;
```

4096 - sizeof(node_t) =4088

4kB - sizeof(node_t)



malloc() → mmap() → OS

4096 - 8
4088
next 0

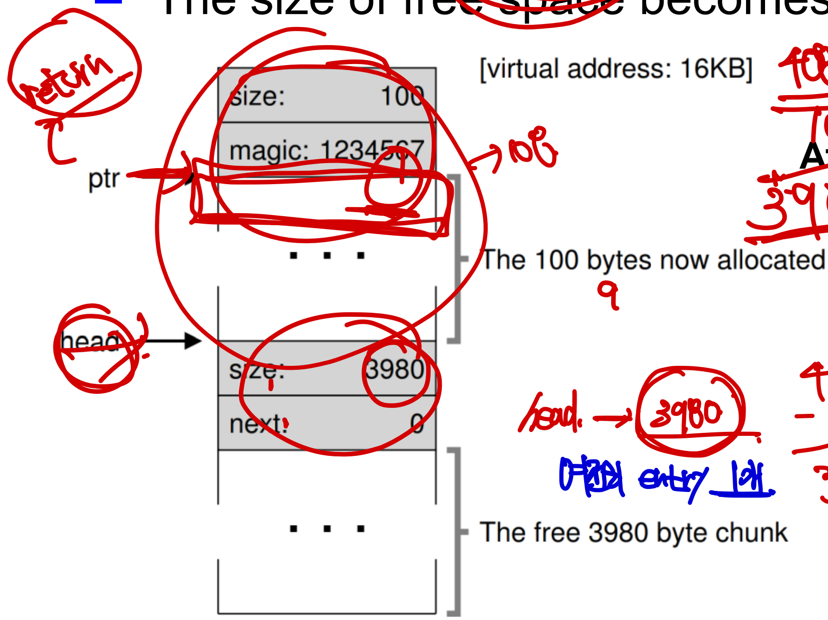
- Once initialization, the actual heap size is 4088 bytes and the **head** pointer contains the beginning address of this range (e.g. virtual address 16KB)

Embedding a Free List: Allocation

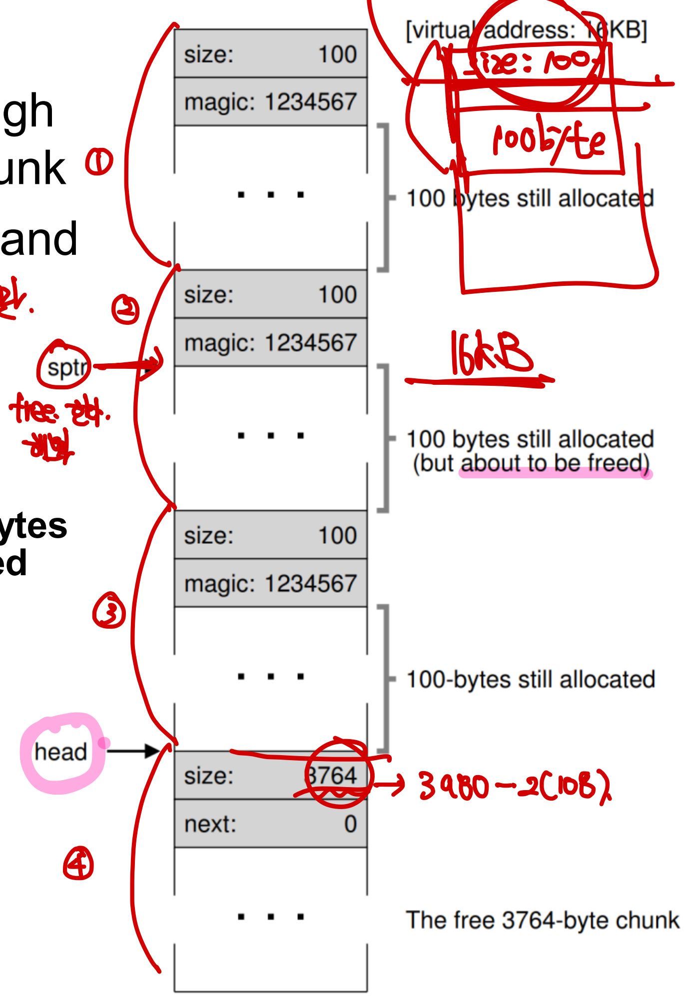
head → 4088 → Null

- **If a chunk of memory is request, the library will first find a chunk that is large enough to accommodate the request**
  - The chunk will be chosen as we have only one
  - Then, it will be split into two: one chunk big enough to service the request and the remaining free chunk ①
  - The library allocates 108 bytes (8 bytes header) and returns a pointer `ptr`
    ↳ 실제 사용자에게는 100 bytes를 주게됨.
  - The size of free space becomes 3980 bytes

4088 − 108 = 3980

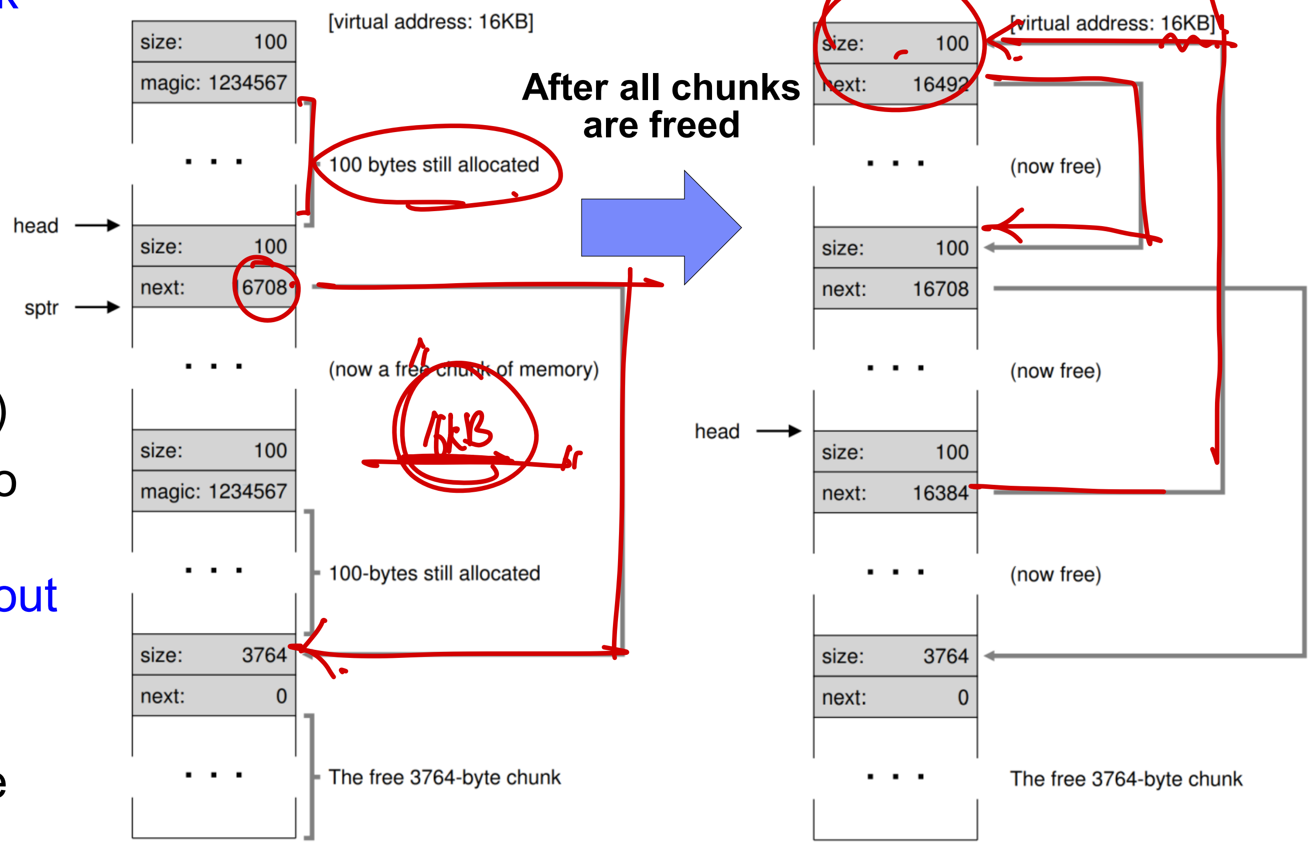


After Two more 100 bytes chunks are allocated



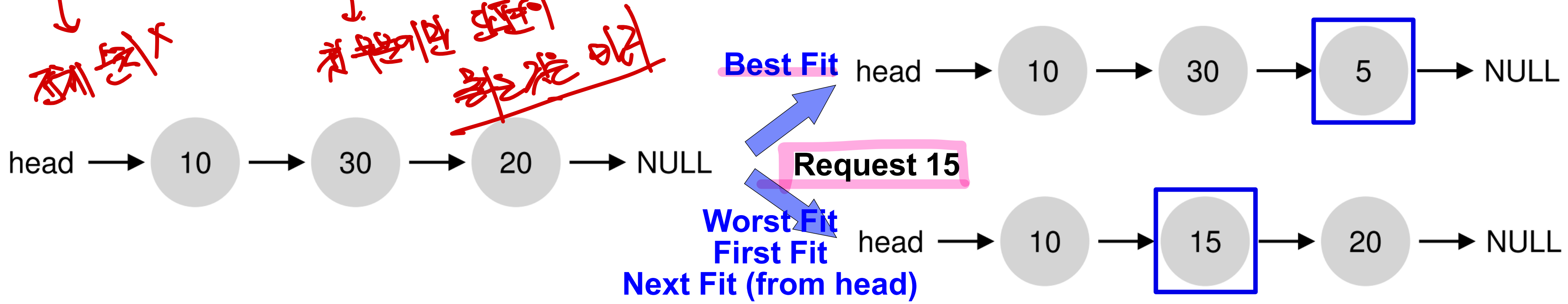
Embedding a Free List: Free

- **What happens when the program returns memory via `free()`?**
  - The application returns the middle chunk of memory by calling `free(16500)`. 16500 is derived by 16384+108+8 (`sptr`)
  - The library immediately figures out the size of the free region, and then adds the free chunk back onto the free list
  - Then, we have a list that starts with a small free chunk (100 bytes, `head`) and a large free chunk (3764 bytes)
  - Freeing the last two chunks will lead to fragmentation without coalescing
  - Merge neighboring chunks to coalesce



Basic Strategies for Managing Free Space

Strategy	Procedure
Best Fit	1) search through the free list and find chunks of free memory that are as big or bigger than the requested size 2) return the one that is the smallest in that group of candidates; this is the so called best-fit chunk 큰 것 중에 가장 작은.
Worst Fit	1) find the largest chunk and return the requested amount 2) keep the remaining (large) chunk on the free list 제일 큰 걸 줌.
First Fit	1) find the first block that is big enough 2) return the requested amount to the user 크기 넘으면 바로 줌.
Next Fit	1) keep an extra pointer to the location within the list where one was looking last 2) return the next big enough chunk of the pointer on the free list



Other Approaches: Segregated Lists

- **The basic idea of the segregated list is that:**
  - If a particular application has one or a few popular (fixed) sized request, it keeps a separate list to manage objects of that size

자주쓰는 메모리 크기 .
예를들면 3KB, 4KB 만 요청 한다.
fixed된

fragmentation이 줄 수 있음.

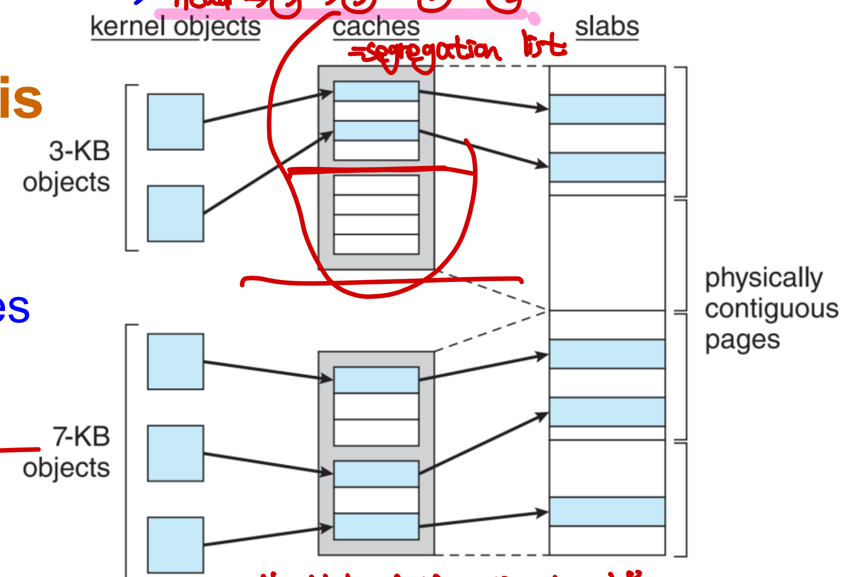
- **The benefit of this approach are obvious**
  - Fragmentation is much less of a concern
  - Allocation and free requests can be served quite quickly

linked-list에서 찾는 필요X

head → 100 → 10 → 15 → 20

head → 3 → 3 → 3 → 3

- **Slab allocator is an advanced form of segregated list adopted in Solaris**
  - A slab is one or more physically contiguous pages in memory
  - This scheme allocates some object caches for kernel object that are likely to be requested frequently (e.g., lock, i-node)
  - Then, the object caches are each segregated free list



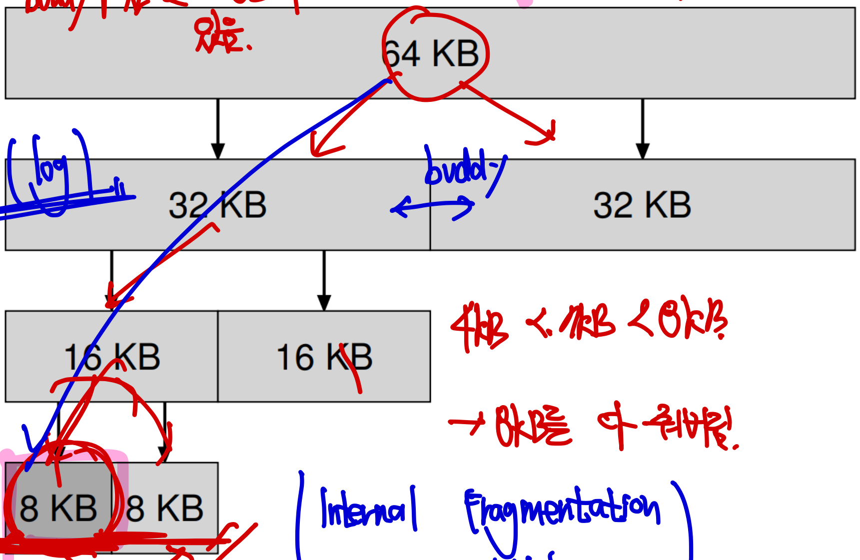
빨리빨리 할당을 해줄 수 있음

Other Approaches: Buddy Allocation

head → 64KB
7KB

- **The binary buddy allocator makes coalescing simple**
  coalescing
  - In this system, free memory is first conceptually through of as of one big space of size of 2^N $4096 = 2^{12}$
  - Each memory request, the search for free space recursively divides free space by two until a big enough block (best fit) to accommodate the request is found
  - e.g.) 7KB request of 64KB memory: 64KB ➔ 32KB ➔ 16KB ➔ 8KB (allocate)
  - This scheme suffers from internal fragmentation

- **The beauty of buddy allocation is found when the block is freed**
  - When 8KB block is freed, the allocator checks if "buddy" 8KB is free O(log)
  - If so, it coalesces the two blocks into 16KB and checks again if buddy of 16KB is free and coalesces if it is free
  - This recursive coalescing process continues up the tree



4KB < 7KB < 8KB
→ 8KB를 다 줘야됨.
(Internal fragmentation 문제가 발생함.)
4KB