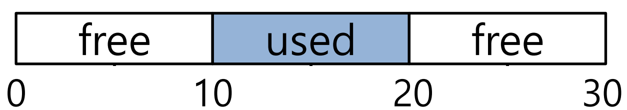
17. Free-Space Management

Operating System: Three Easy Pieces

Variable sized allocation

- Free space management is easier when the memory is divided into fixed size units
- For variable-sized units, the problem that exists is known as **external fragmentation**
  - Arises in a user-level memory-allocation library (as in malloc() and free())
  - And in an OS managing physical memory using segmentation
  - A request of 15 bytes will fail here even though total 20 bytes is free

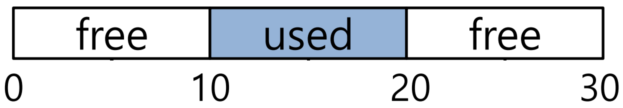


Given a block of memory, how do we allocate variable sized memory allocation requests to minimize fragmentation and overheads ?

Common Low Level Mechanism

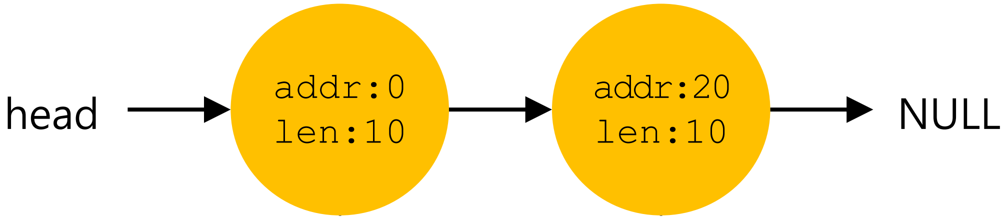
- **Free List:** generic data structure used by the library to manage free space in the heap
  - Assume the following 30 byte heap

30-byte heap:



  - The free list for this could be

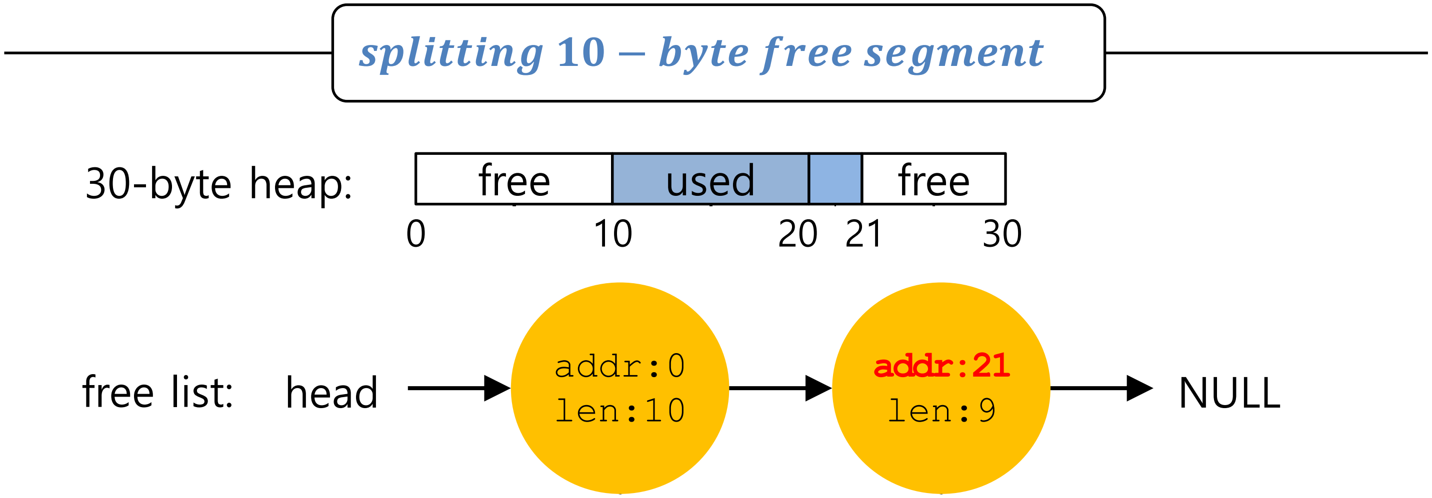
free list:



  - Any request = 10 bytes could be satisfied by either
  - What happens if the request < 10 bytes ?

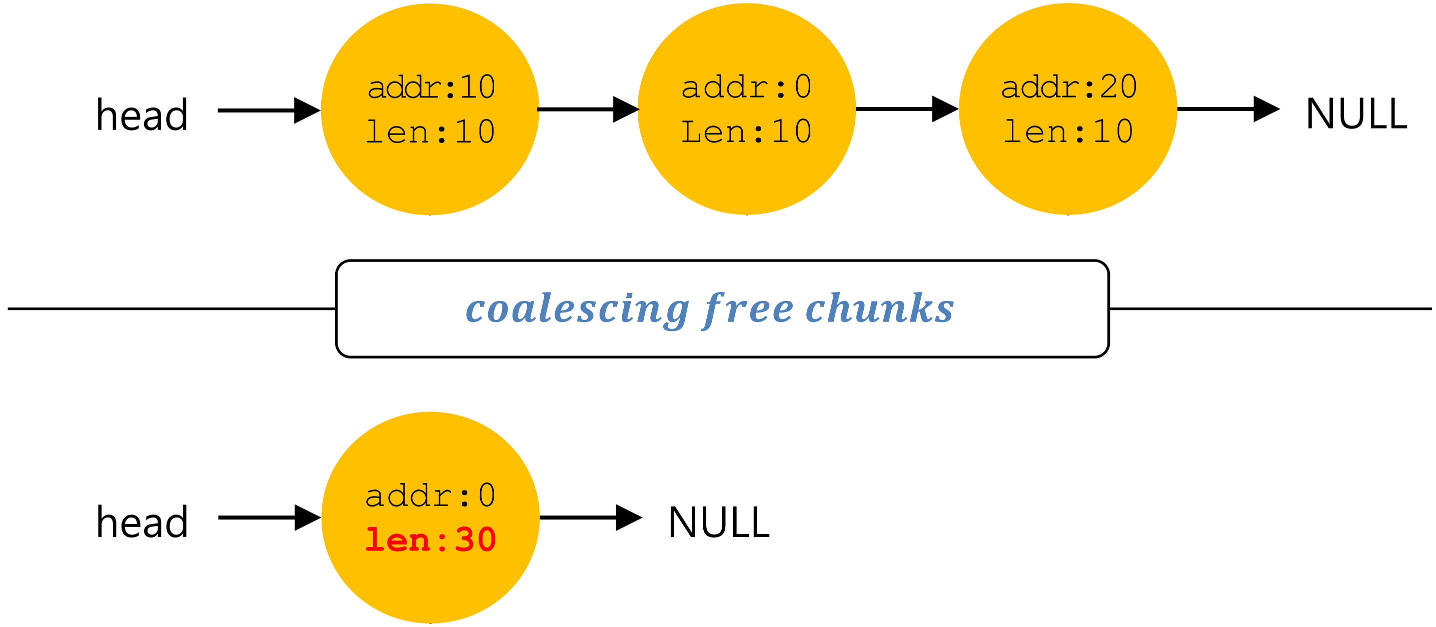
Common Low Level Mechanism

- **Splitting**: Finding a free chunk of memory that can satisfy the request and splitting it into two.
  - In previous example if we have **1-byte request** and lets say the allocator decided to use the second of the two elements on the list
  - That chunk would be split into two, returning the first chunk of 1-byte allo -cated region (with the address 20) and the second chunk would remain in list.



Common Low Level Mechanism - Coalescing

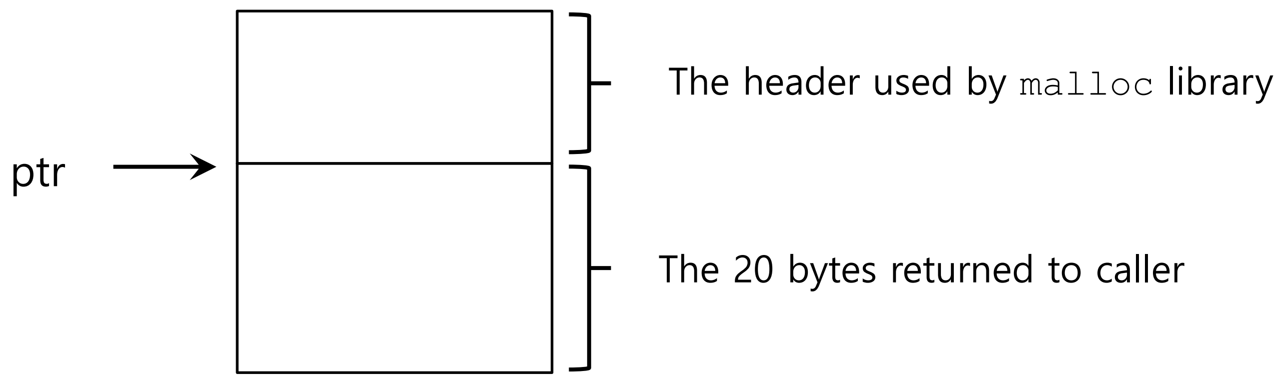
- If a user requests memory that is **bigger than free chunk size**, the list will **not find** such a free chunk.
- **Coalescing**: **Merge** existing free chunks into a large single free chunk if **addresses** of them are **nearby**. Ex. If the application calls free(10) returning the middle space



Tracking The Size of Allocated Regions

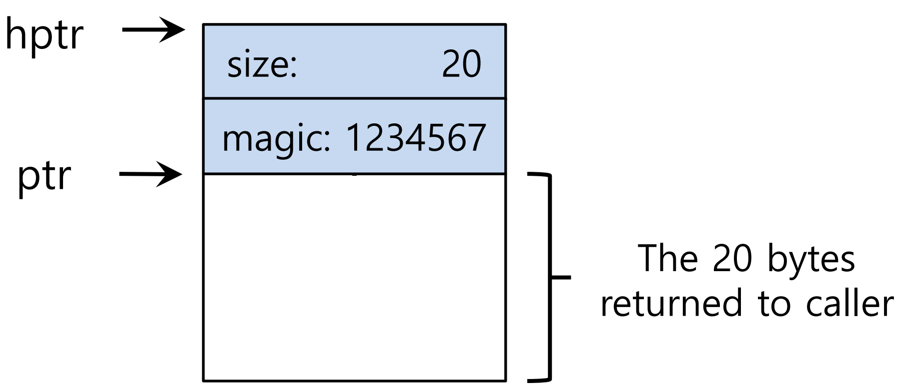
- The interface to `free(void *ptr)` does **not take** a **size parameter**.
  - How does the library **know the size** of memory region that will be back **into free list**?
- Most allocators store **extra information** in a **header** block.

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



The Header of Allocated Memory Chunk

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



Specific Contents Of The Header

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

A Simple Header

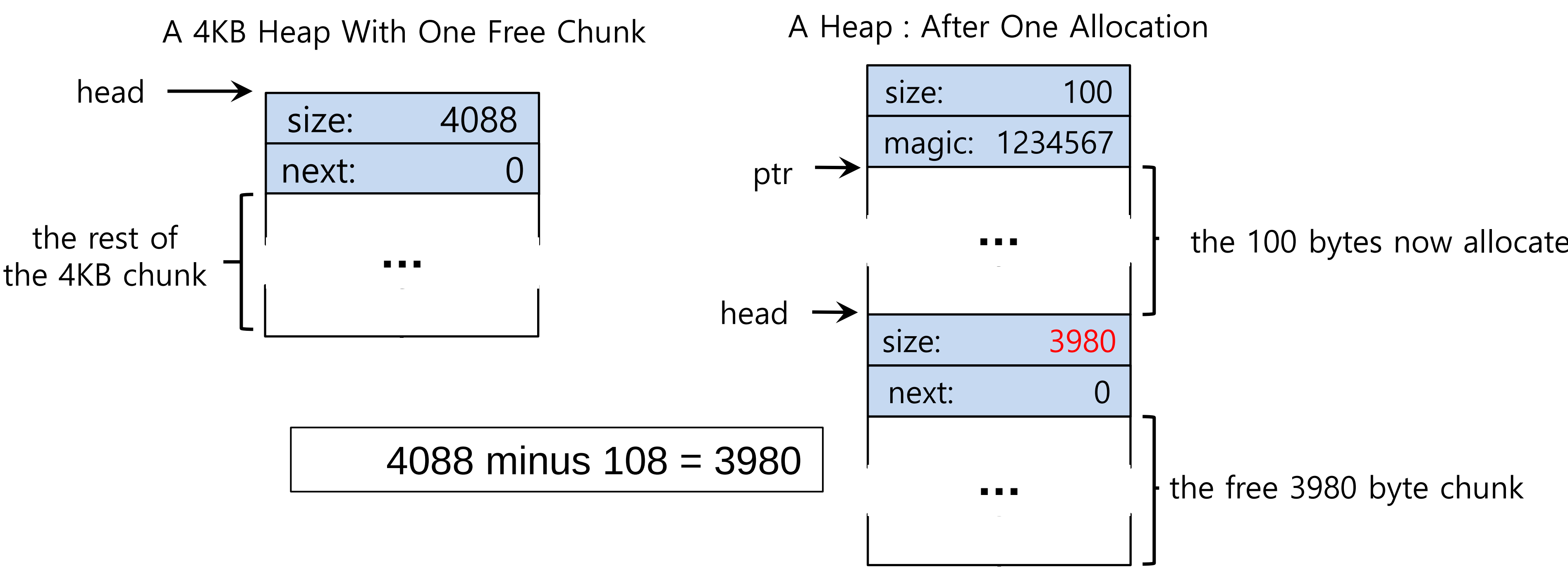
The Header of Allocated Memory Chunk(Cont.)

- If a user **request `N` bytes**, the library searches for a free chunk of **size `N` plus the size of the header**
- When user calls **free (ptr)**, library uses simple pointer arithmetic to find the header pointer.

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

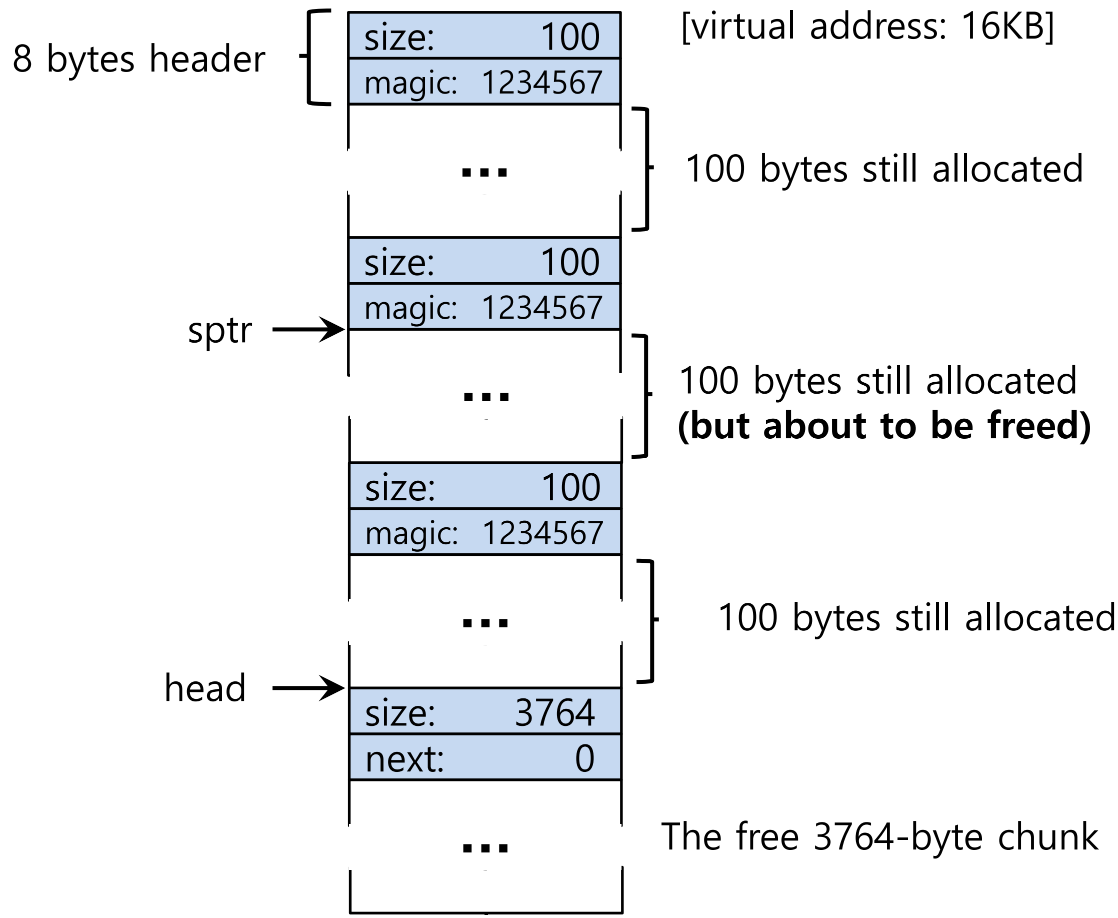
Embedding a Free List

- Free space managed as a linked list
  - Pointer to the next free chunk is embedded within the free chunk
- The library tracks the head of the list
  - Allocations happen from the head



Free Space With Chunks Allocated

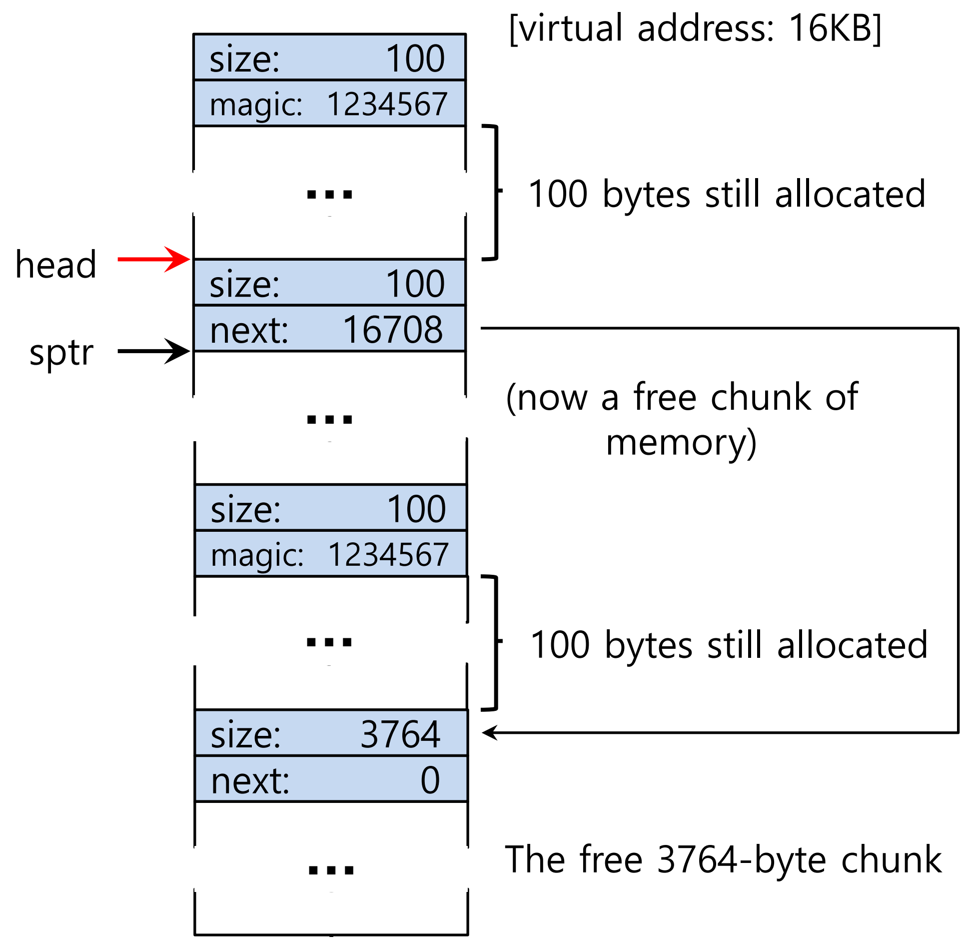
- Suppose 3 allocations of size 100 bytes each happen
- Then, the middle chunk pointed to by sptr is freed



Free Space With Three Chunks Allocated

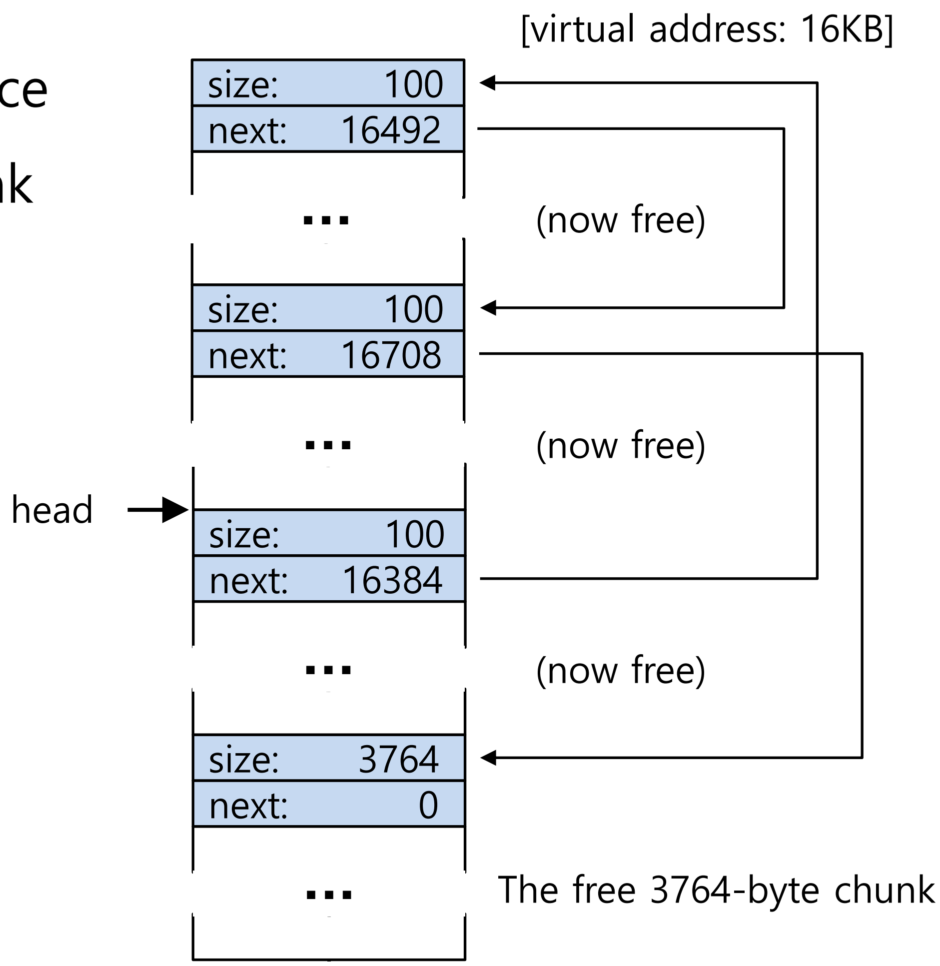
Free Space With `free()`

- Example: `free(sptr)`
  - The 100 bytes chunks is **back into** the free list.
  - The free list header will point the small chunk
  - It now has two non- contiguous elements
  - Free space may be scattered aro -und due to **fragmentation**
  - Cannot satisfy a request for 3800 bytes even though we have the free space



Free Space With Freed Chunks

- Let’s assume that the remaining two in-use chunks are also freed.
- A smart algorithm should coalesce them all into a bigger free chunk

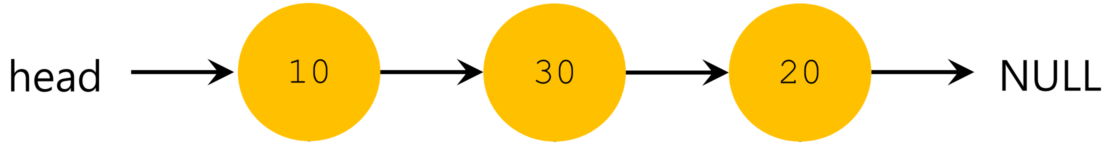


Managing Free Space: Basic Strategies

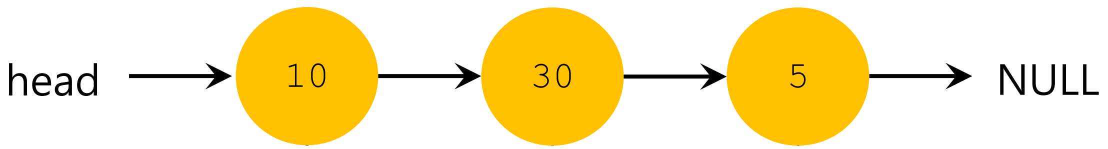
- First Fit: allocate the **first chunk** that is **big enough** for the request
- Best Fit: allocate free chunk that is closest in size
- Worst Fit: allocate free chunk that is farthest in size
- In all these strategies, the requested amount is returned while keeping the **remaining chunk** on the free list.

Examples of Basic Strategies

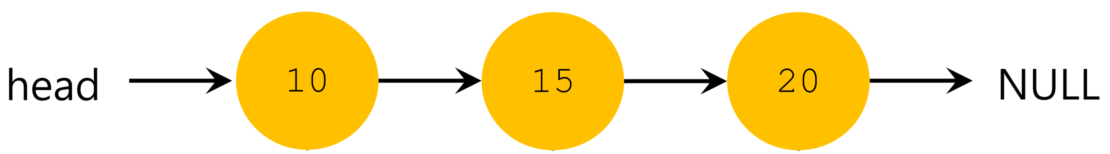
- Allocation Request Size 15



- Result of Best-fit: allocate the 20-byte chunk

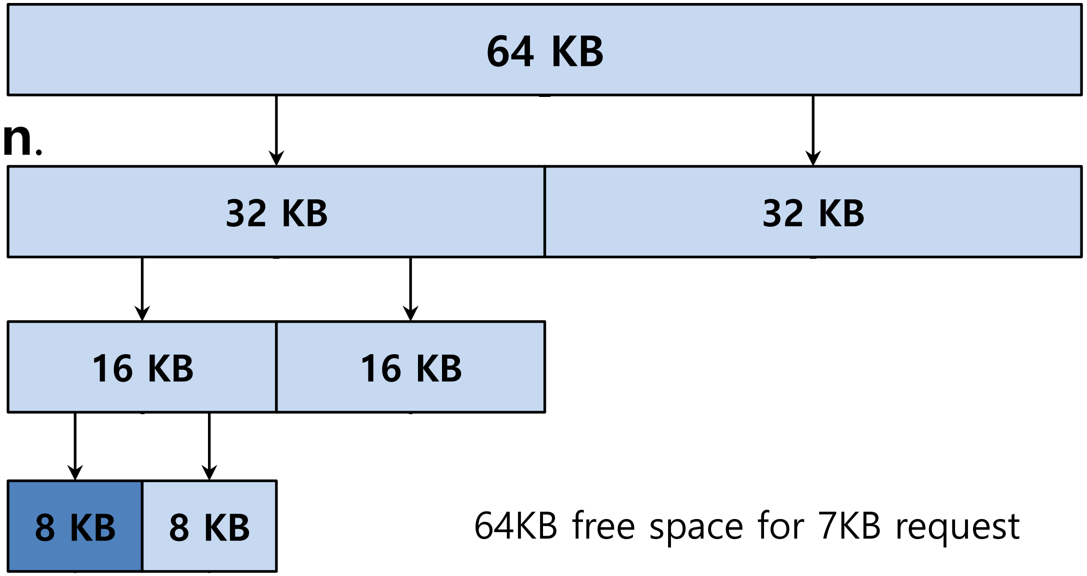


- Result of Worst-fit: : remaining chunk is bigger and more usable



Other Approaches: Buddy Allocation

- Binary Buddy Allocation
  - The allocator **divides free space** by two **until a block** that is big enough to accommodate the request is **found**.
  - E.g., for a request of 7000 bytes, allocate 8 KB chunk
- Pros: Easy coalescing - if 8KB block and its "buddy" (adjacent block) are free, they can form a 16KB chunk
- Cons: **internal fragmentation**.



64KB free space for 7KB request