

A decorative graphic consisting of a thin yellow circle and a horizontal bar with a yellow-to-white gradient. A large black left square bracket is on the left side of the bar, and a large yellow right square bracket is on the right side.

Thanks for the memory

CS 241
Heap allocation

[Why is malloc not easy?]

- **Must be fast**

- Can only perform relatively simple computation
- Should avoid too many system calls (`sbrk()`)

- **Must be memory-efficient**

- Can't predict what or when the user will malloc/free
- Even if we knew sizes in advance, packing the requests into memory optimally is NP-complete, i.e., a provably hard problem!

- **Must work!**

- Easy to make mistakes with pointer & bit manipulation



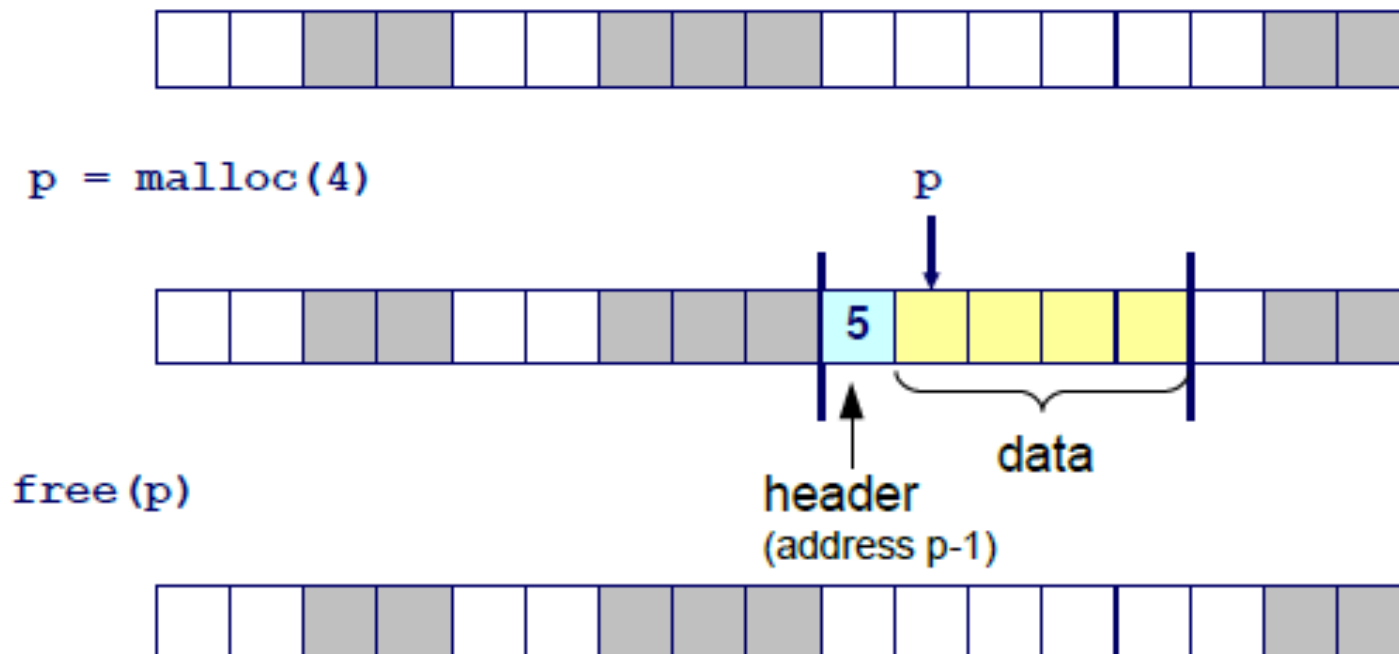
[Implementation Issues]

- How do we know how much memory to free just given a pointer?
- How do we keep track of the free blocks?
- What do we do with the extra space when allocating a memory block that is smaller than the free block it is placed in?
- How do we pick which free block to use for allocation?



[Knowing how much to free]

- Standard method
 - Keep the length of the block in the header preceding the block
 - Requires an extra word for every allocated block



[Keeping Track of Free Blocks]

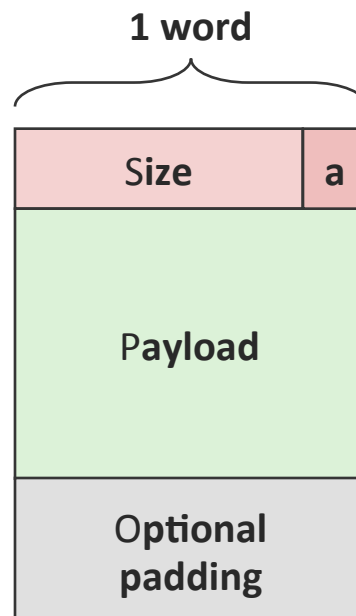
- One of the biggest jobs of an allocator is knowing where the free memory is
- The allocator's approach to this problem affects:
 - Throughput – time to complete a malloc() or free()
 - Space utilization – amount of extra metadata used to track location of free memory
- There are many approaches to free space management
 - Next, we will talk about one: **Implicit free lists.**



[Implicit free list]

- For each block we need both size and allocation status
 - Could store this information in two words: wasteful!
- Standard trick
 - If blocks are aligned, low-order address bits are always 0
 - Why store an always-0 bit? Use it as allocated/free flag!
 - When reading size word, must mask out this bit

*Format of
allocated and
free blocks*



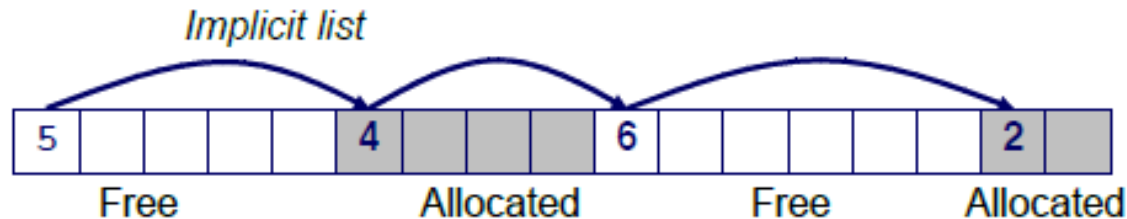
a = 1: Allocated block
a = 0: Free block

Size: block size

**Payload: application data
(allocated blocks only)**



[Implicit free list]

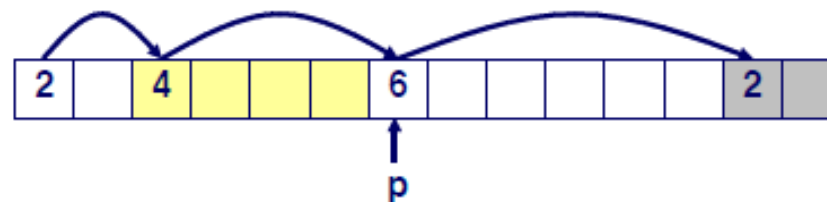


- No **explicit** structure tracking location of free/allocated blocks.
 - Rather, the size word (and allocated bit) in each block form an **implicit** “block list”
- How do we find a free block in the heap?
 - Start scanning from the beginning of the heap.
 - Traverse each block until (a) we find a free block and (b) the block is large enough to handle the request.
 - This is called the **first fit** strategy.
 - Could also use **next fit**, **best fit**, etc



[Implicit list: Allocating a Block]

- Splitting free blocks
 - Since allocated space might be smaller than free space, we may need to split the free block that we're allocating within

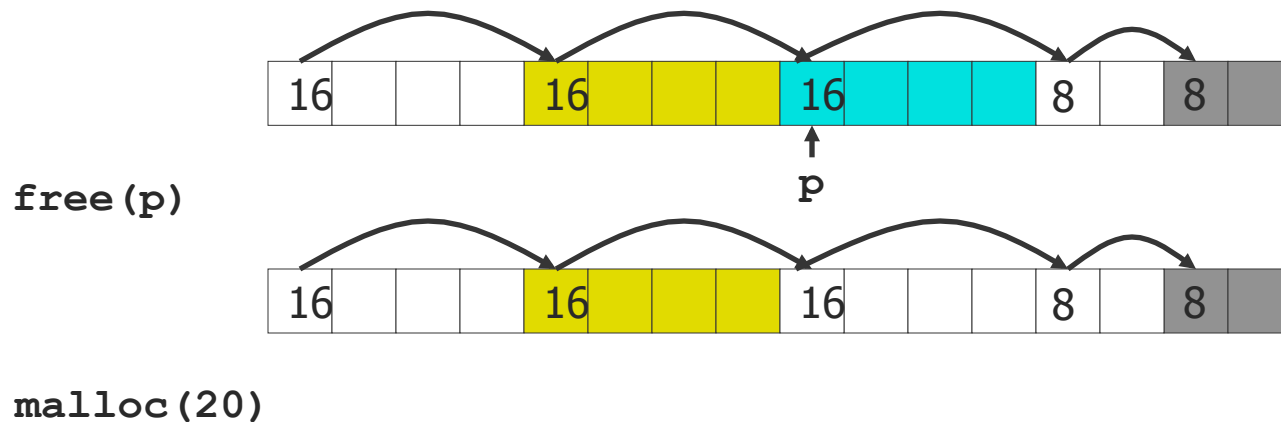


`addblock(p, 4)`



[Implicit List: Freeing a Block]

- Simplest implementation:
 - Only need to clear allocated flag
 - `void free_block(ptr p) { *p = *p & ~1; }`
- But can lead to “false fragmentation”



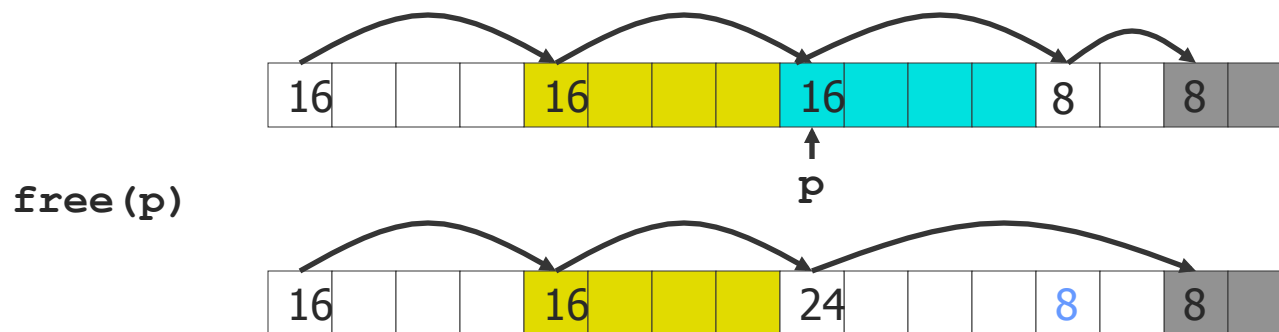
Oops!

- There's enough free space, but allocator won't find it!



[Implicit List: Coalescing]

- Join (coalesce) with next and previous block if they are free
 - Coalescing with next block

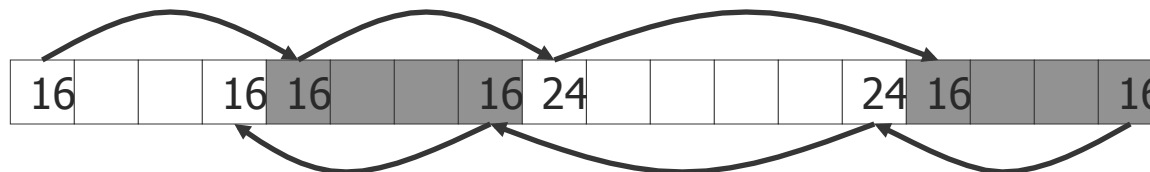
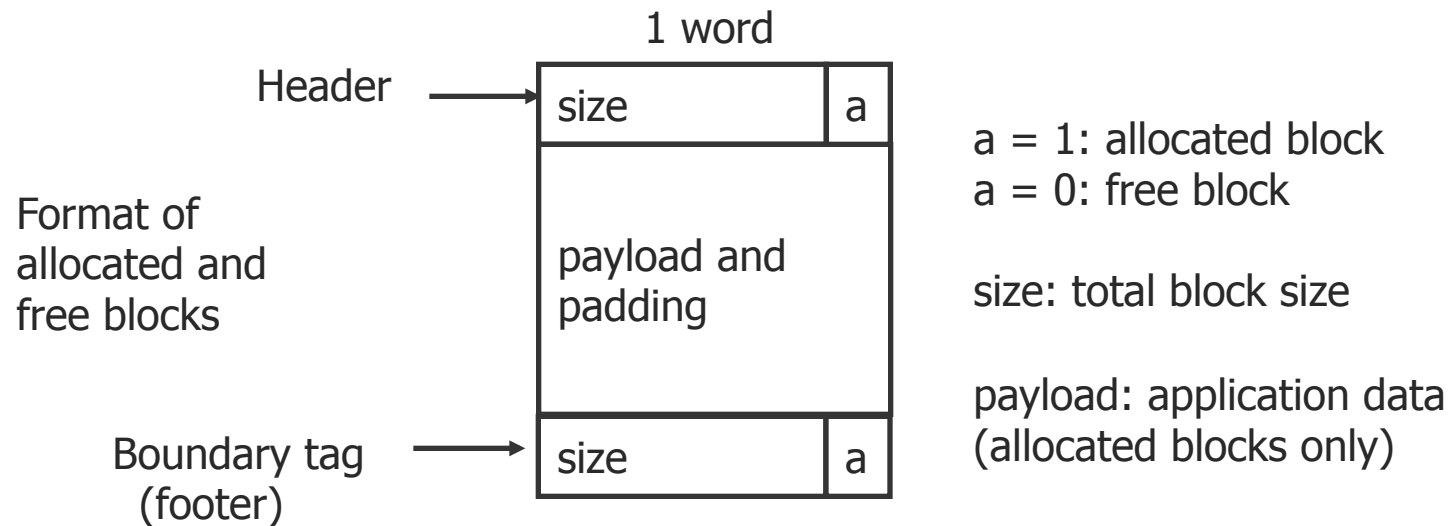


- But how do we coalesce with previous block?



[Implicit List: Bidirectional Coalescing]

- **Boundary tags** [Knuth73]
 - Replicate size/allocated word at tail end of all blocks
 - Lets us traverse list backwards, but needs extra space
 - General technique: doubly linked list



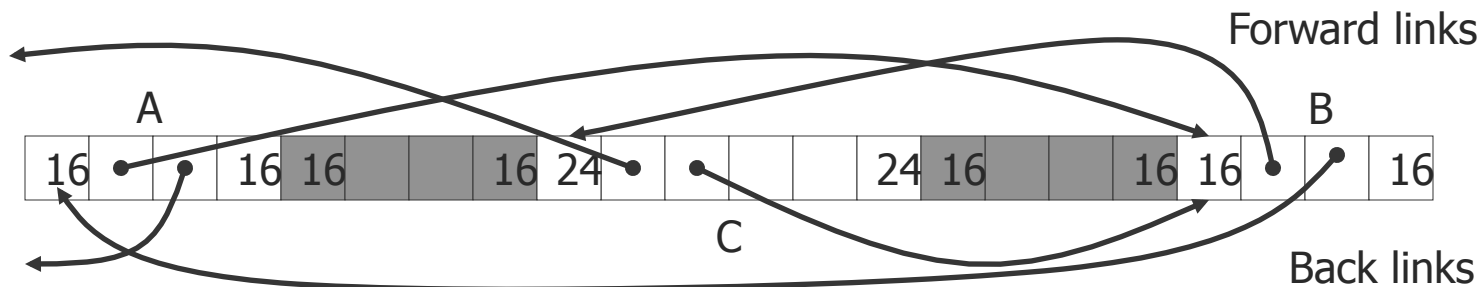
[Implicit Lists: Summary]

- **Implementation:** very simple
- **Allocate:** linear-time worst case
- **Free:** constant-time worst case—even with coalescing
- **Memory usage:** will depend on placement policy
 - First, next, or best fit
- Not used in practice for malloc/free because of linear-time allocate, but used in some special-purpose applications
- However, concepts of splitting and boundary tag coalescing are general to *all* allocators



[Alternative: Explicit Free Lists]

- Linked list among free blocks
- Use data space for link pointers
 - Typically doubly linked
 - Still need boundary tags for coalescing



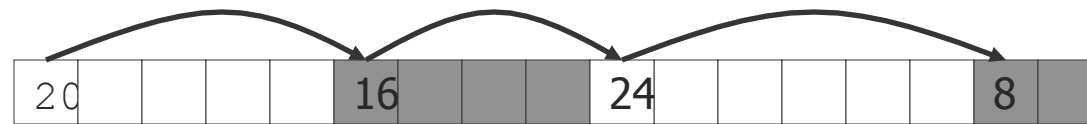
[Freeing with Explicit Free Lists]

- **Insertion policy**: Where in free list to put newly freed block?
 - LIFO (last-in-first-out) policy
 - Insert freed block at beginning of free list
 - Pro: simple, and constant-time
 - Con: studies suggest fragmentation is worse than address-ordered
 - Address-ordered policy
 - Insert freed blocks so list is always in address order
 - i.e. $\text{addr}(\text{pred}) < \text{addr}(\text{curr}) < \text{addr}(\text{succ})$
 - Con: requires search (using boundary tags); slow!
 - Pro: studies suggest fragmentation is better than LIFO

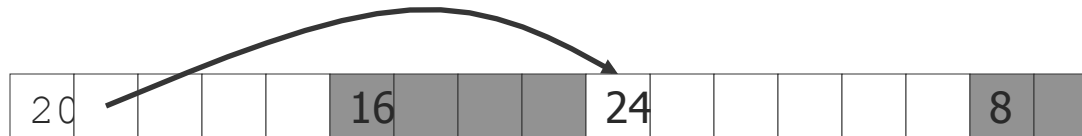


[Summary: tracking free blocks]

- Method 1: **Implicit list** using lengths -- links all blocks



- Method 2: **Explicit list** among the free blocks using pointers within the free blocks

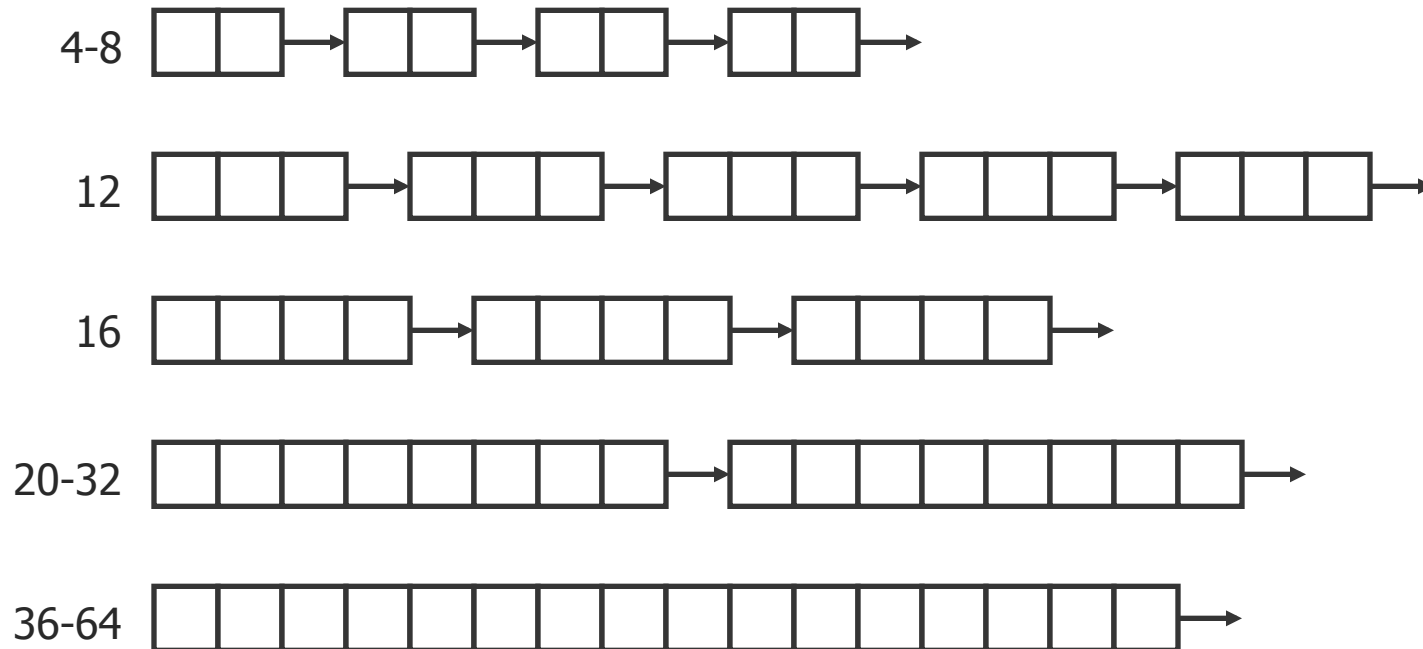


- Method 3: **Segregated free list**
 - Different free lists for different size classes
 - We'll talk about this one next



[Segregated free lists]

- Each **size class** has its own collection of blocks



- Often separate size class for every small size (8, 12, 16, ...)
- For larger, typically have size class for each power of 2
- What is the point of having separate lists?



[Buddy Allocators]

- Special case of segregated free lists
- Basic idea:
 - Limited to power-of-two sizes
 - Can only coalesce with "buddy", who is other half of next-higher power of two
- Clever use of low address bits to find buddies
- See http://en.wikipedia.org/wiki/Buddy_memory_allocation
- Problem: large powers of two result in large internal fragmentation (e.g., what if you want to allocate 65537 bytes?)



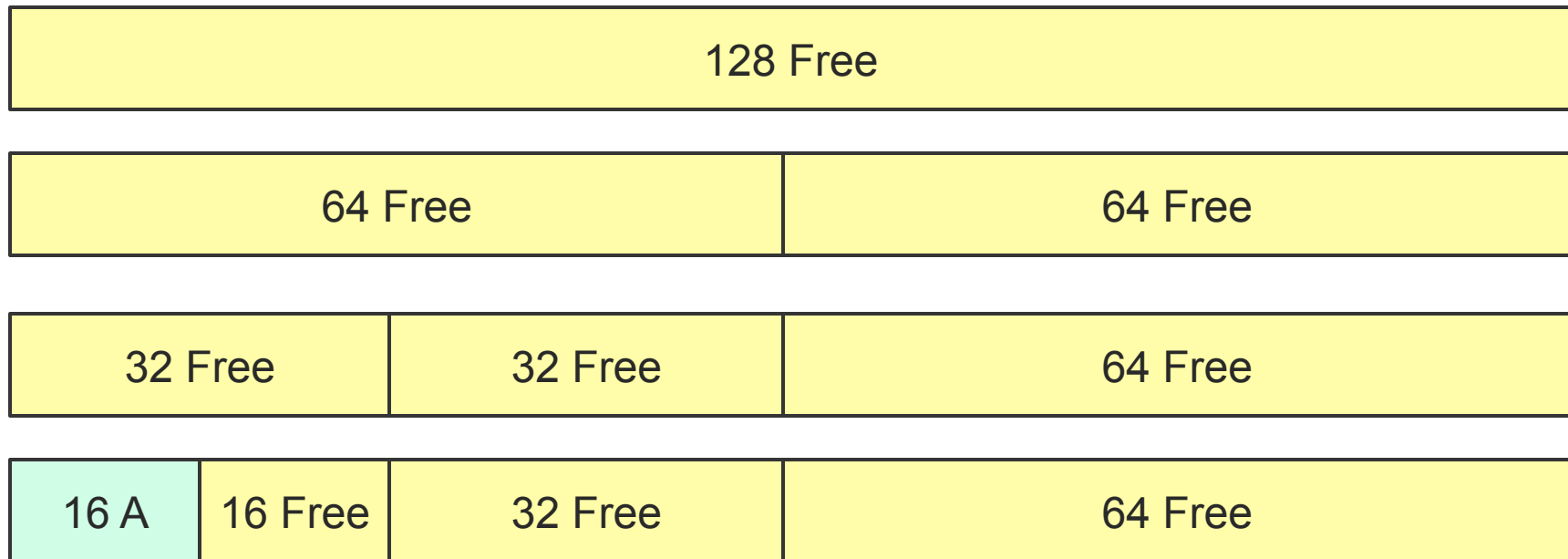
[Buddy System Example]

128 Free



[Buddy System Example]

pA= malloc(16)



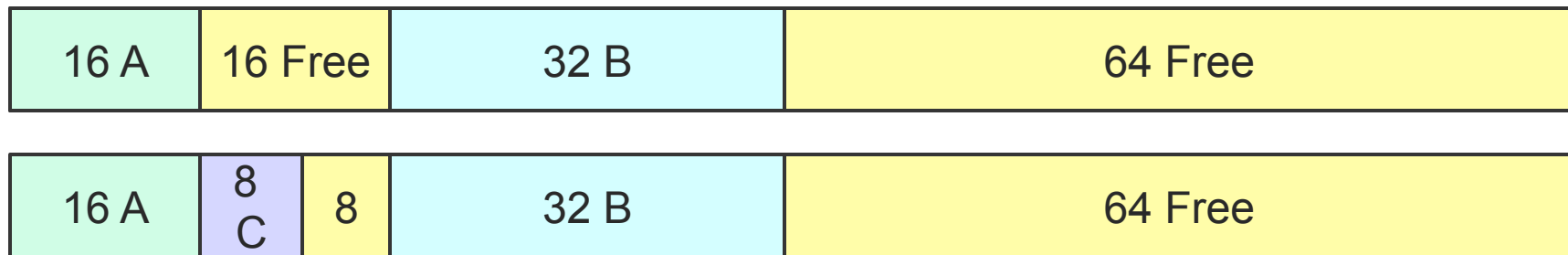
[Buddy System Example]

pB = malloc(32)



[Buddy System Example]

pC = malloc(8)



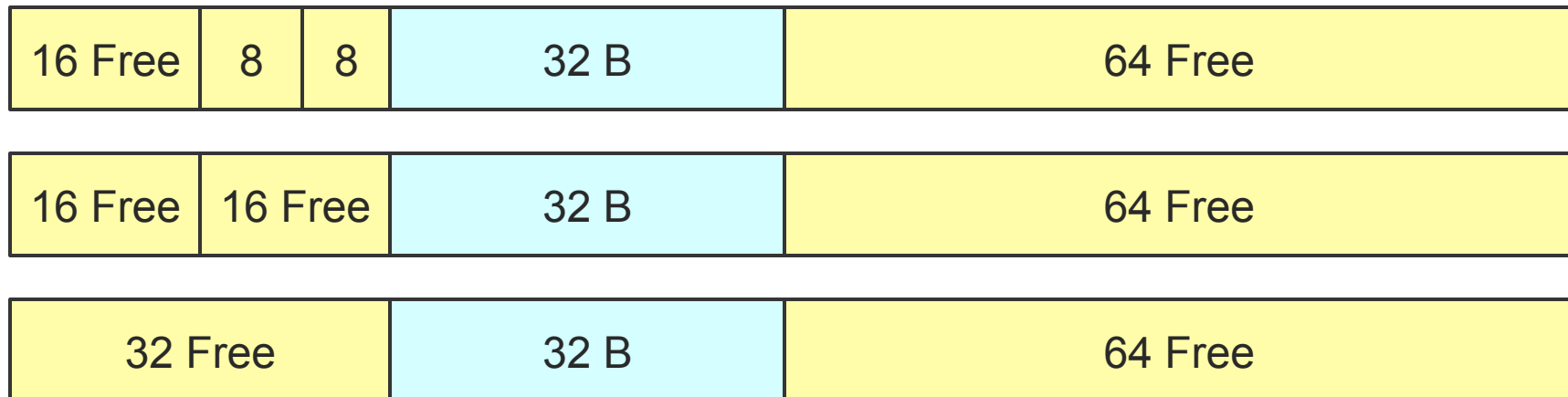
[Buddy System Example]

free(pA)



[Buddy System Example]

free(pC)



- **Advantage:** Low external fragmentation
- **Disadvantage:** Internal frag if not 2^n -sized request
- More info: wikipedia : [Buddy memory allocation](#)





[So what should I do for MP?]

- Designs sketched here are reasonable
- Many other possible designs
- Implement anything you want!

