Memory Allocation & Heap

Memory allocation within a process

- Stack data structure
  - Function calls follow LIFO semantics
  - So we can use a stack data structure to represent the process's stack – no fragmentation!
- Heap: `malloc, free`

MP2!

  - This is a much harder problem
  - Need to deal with **fragmentation**

`malloc` Constraints

- Applications
  - Can issue arbitrary sequence of `malloc` and `free` requests
  - `free` request must be to a `malloc`'d block

`malloc` Constraints

- Allocators
  - Can’t control number or size of allocated blocks
  - Must respond immediately to `malloc` requests
    - *i.e.*, can’t reorder or buffer requests
  - Must allocate blocks from free memory
  - Must align blocks so they satisfy all requirements
    - 8 byte alignment for `libc` `malloc` on Linux boxes
  - Can manipulate and modify only free memory
  - Can’t move the allocated blocks once they are `malloc`’d
    - *i.e.*, compaction is not allowed (why not?)

Goal 1: Speed

- Allocate fast!
  - Minimize overhead for both allocation and deallocation
- Maximize throughput
  - Number of completed `malloc` or `free` requests per unit time
  - Example
    - 5,000 `malloc` calls and 5,000 `free` calls in 10 seconds

Goal 1: Speed

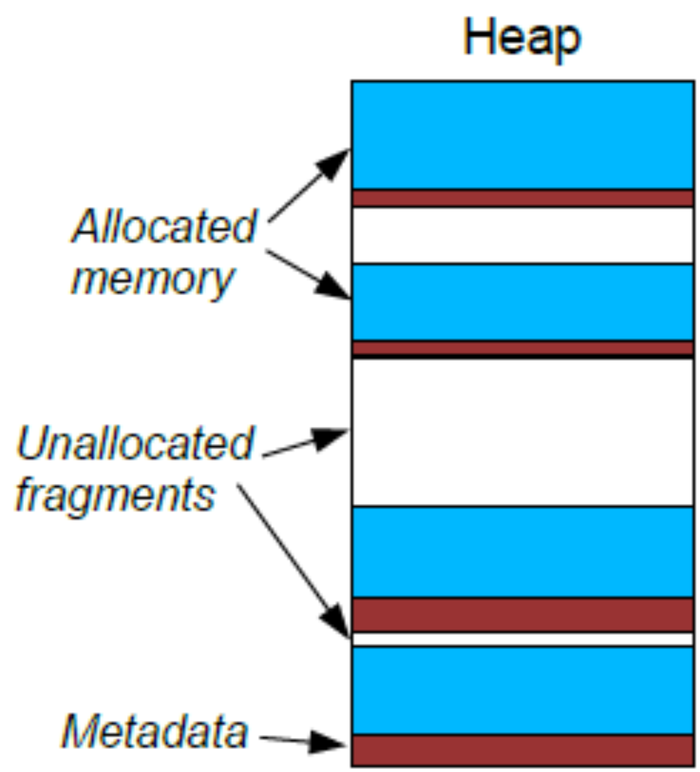
- Allocate fast!
  - Minimize overhead for both allocation and deallocation
- Maximize throughput
  - Number of completed `malloc` or `free` requests per unit time
  - Example
    - 5,000 `malloc` calls and 5,000 `free` calls in 10 seconds
    - Throughput is 1,000 operations/second

Goal 1: Speed

- BUT
  - A fast allocator may not be efficient in terms of memory utilization
  - Faster allocators tend to be “sloppier”
    - Example: don’t look through every free block to find the perfect fit

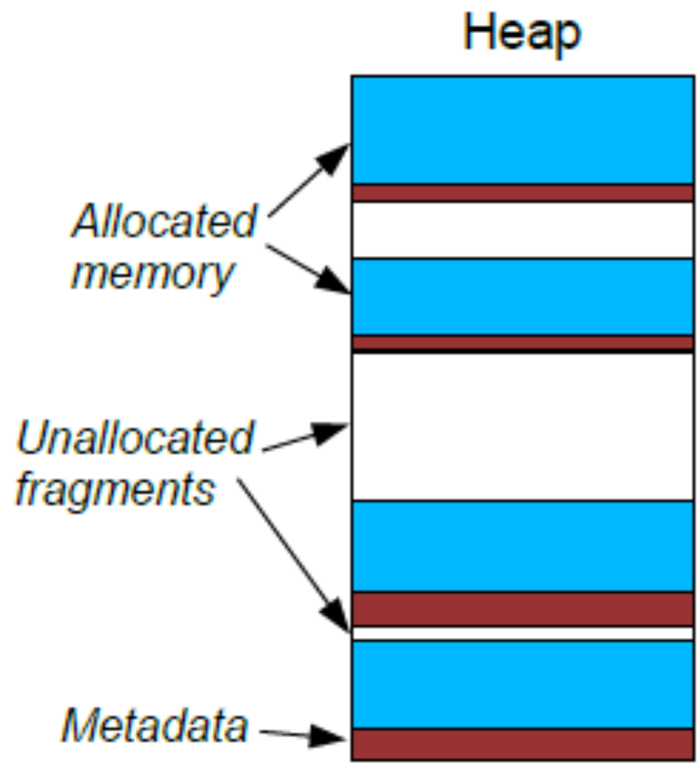
Goal 2: Memory Utilization

- Allocators usually waste some memory
  - Extra metadata or internal structures used by the allocator itself
    - Example: keeping track of where free memory is located
  - Chunks of heap memory that are unallocated (fragments)



Goal 2: Memory Utilization

- Memory utilization =
  - The total amount of memory allocated to the application divided by the total heap size
- Ideal
  - utilization = 100%
- In practice
  - try to get close to 100%



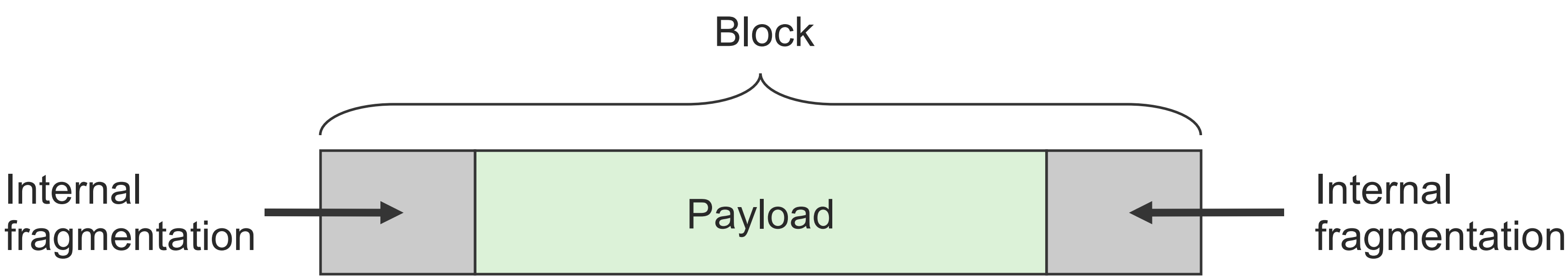
Fragmentation

- Poor memory utilization caused by unallocatable memory
  - internal fragmentation
  - external fragmentation

- `malloc` fragmentation
  - When allocating memory to applications

Internal fragmentation

- Payload is smaller than block size



- Caused by
  - Overhead of maintaining heap data structures
  - Padding for alignment purposes
  - Explicit policy decisions
    (e.g., to return a big block to satisfy a small request)

Experiment

- Does `libc`'s `malloc` have internal fragmentation? How much?
- How would you test this?

Run Example

fragtest

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
   char* a = (char*) malloc(1);
   char* b = (char*) malloc(1);
   char* c = (char*) malloc(100);
   char* d = (char*) malloc(100);

   printf("a = %p\n b = %p\n c = %p\n d = %p\n",
   a,b,c,d);
}
```

What output would you expect?

fragtest - Output

```
a = malloc(1); b = malloc(1);
c = malloc(100); d = malloc(100);
```

```
a = 0xdb64010
                0x20 = 32 ≠ 1
b = 0xdb64030
                0x20 = 32 ≠ 1
c = 0xdb64050
                0x70 = 112 ≠ 100
d = 0xdb640c0
```

External Fragmentation

- There is enough aggregate heap memory, but no single free block is large enough

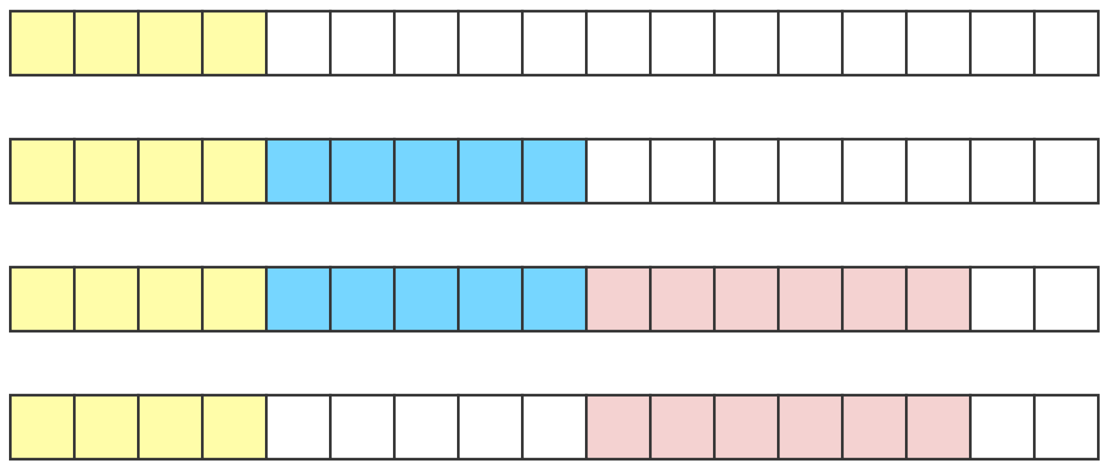
```
p1 = malloc(4)

p2 = malloc(5)

p3 = malloc(6)

free(p2)

p4 = malloc(6)
```



Oops! (what would happen now?)

Depends on the pattern of future requests
Difficult to plan for

Conflicting performance goals

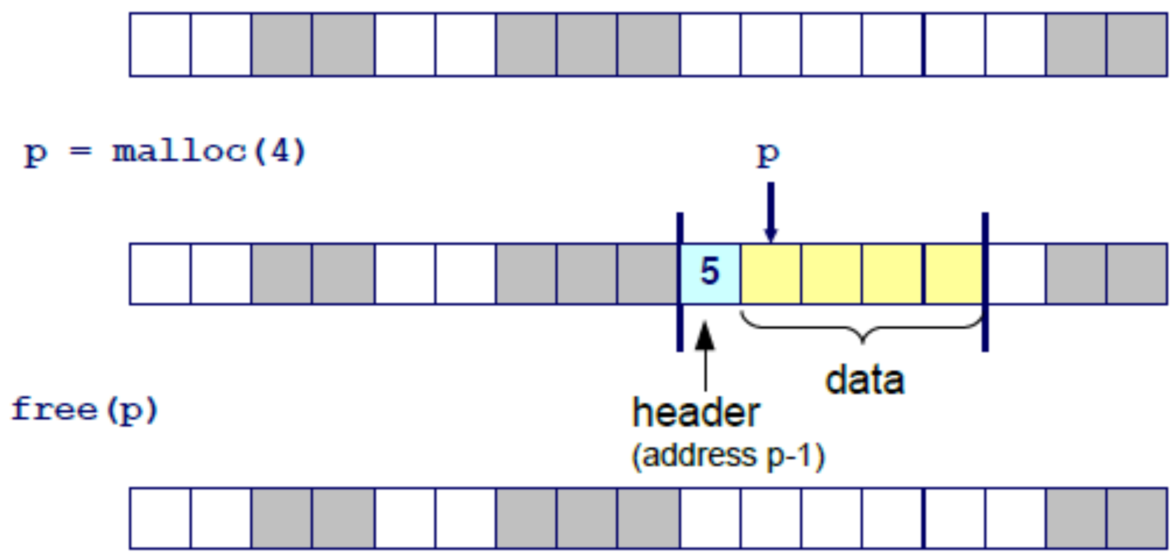
- Throughput vs. Utilization
  - Difficult to achieve simultaneously
  - Faster allocators tend to be “sloppier” with memory usage
  - Space-efficient allocators may not be very fast
    - Tracking fragments to avoid waste generally results in longer allocation times

Implementation issues you need to solve!

- How do I know how much memory to free just given a pointer?

Keep the length of the block in the header preceding the block

Requires an extra word for every allocated block

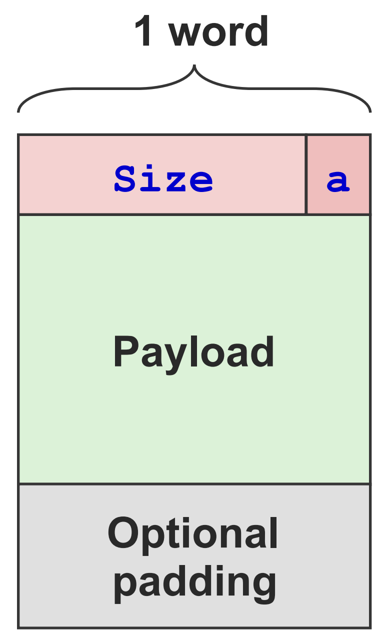


Keeping Track of Free Blocks

- 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

Implicit Free Lists

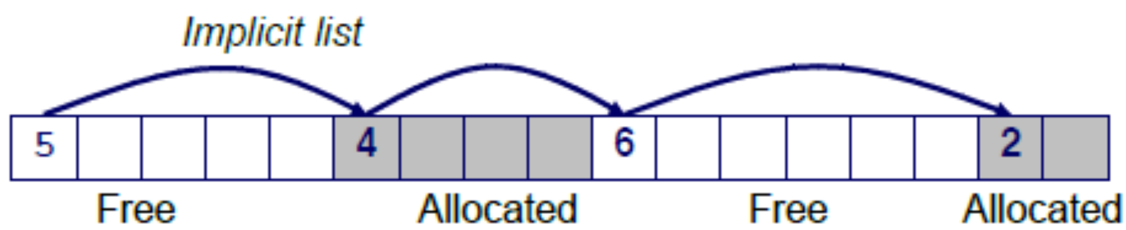
- For each block we need both size and allocation status
  - Could store this information in two words: wasteful!
- Standard trick
  - If blocks are aligned, low-order address bits are always 0
  - Why store an always-0 bit? Use it as allocated/free flag!
  - When reading size word, must mask out this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



`a = 1`: Allocated block
`a = 0`: Free block
`Size`: block size

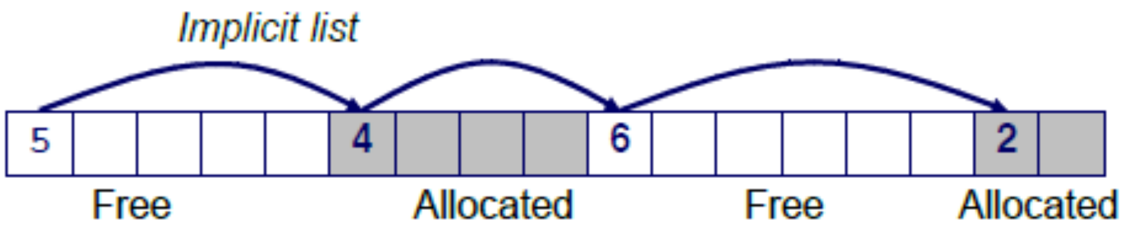
Payload: application data (allocated blocks only)

Implicit Free Lists



- No explicit structure tracking location of free/ allocated blocks.
  - Rather, the size word (and allocated bit) in each block form an implicit "block list"

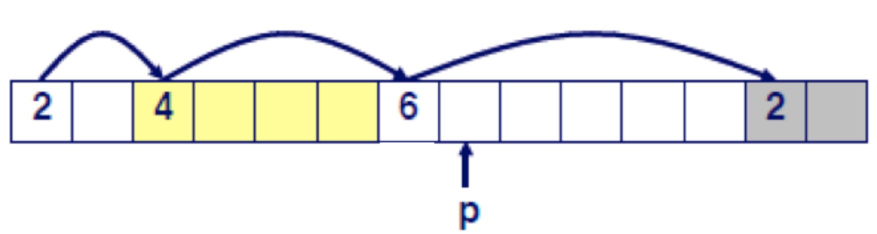
Implicit Free Lists: Free Blocks



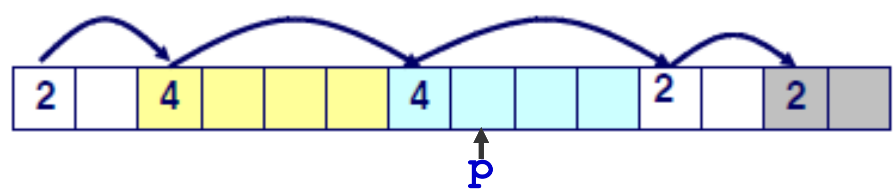
- How do we find a free block in the heap?
  - Start scanning from the beginning of the heap.
  - Traverse each block until (a) we find a free block and (b) the block is large enough to handle the request.
  - This is called the first fit strategy
    - Could also use best fit, etc

Implicit Free Lists: Allocating Blocks

- What if the allocated space is smaller than free space?
- Split free blocks




```
addblock(p, 4)
```

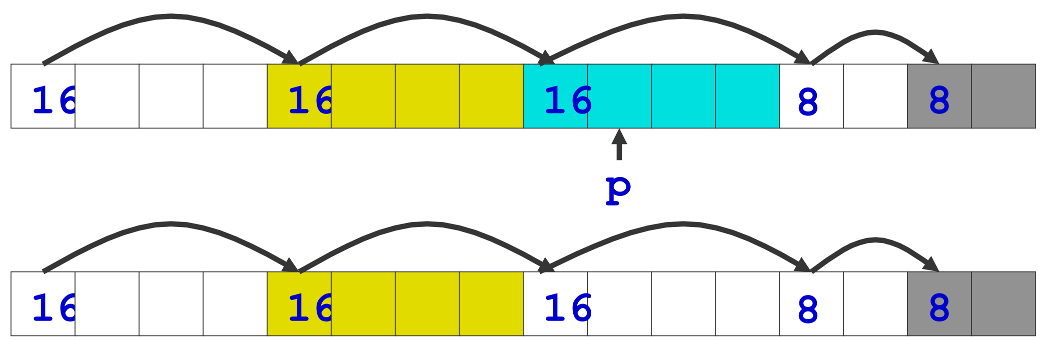


Implicit Free Lists: Freeing a Block

- How do you free a block?
- Simplest implementation:
  - Only need to clear allocated flag
- Problem?

Implicit Free Lists: Freeing a Block

- Only need to clear allocated flag
- Problem?
  - False fragmentation



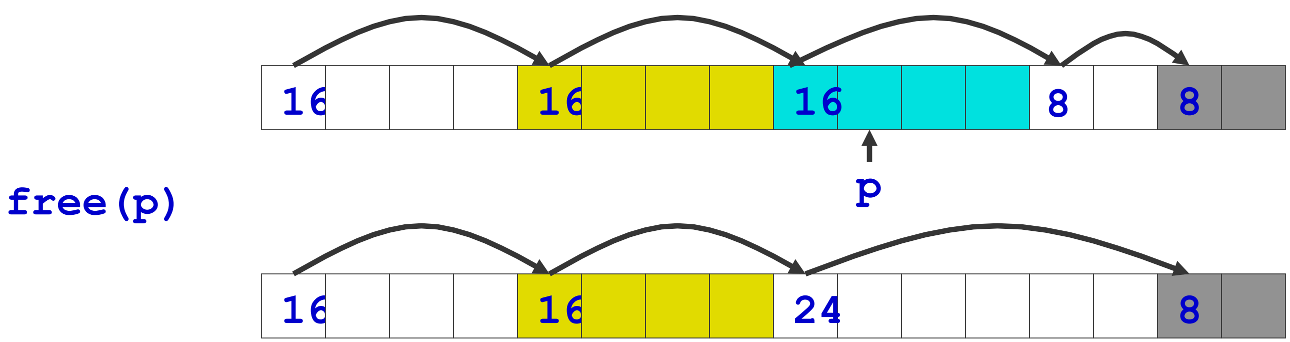
`free(p)`

`malloc(20)` **Oops!**

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

Implicit Free Lists: Coalescing Blocks

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



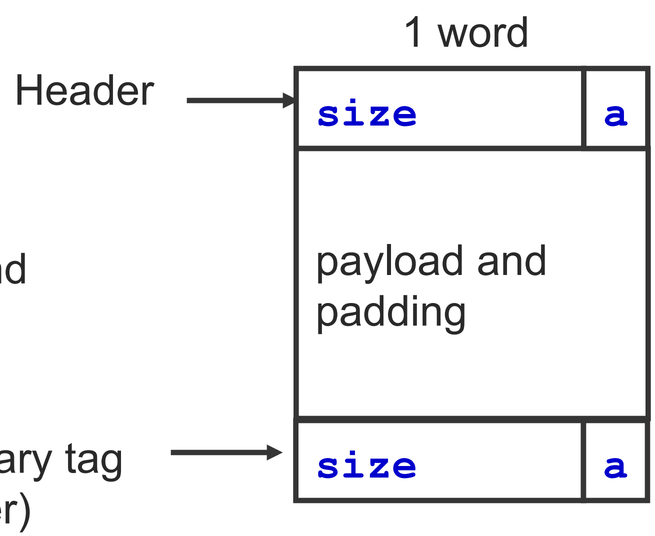
But how do we coalesce with previous block?

Implicit Free Lists: 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

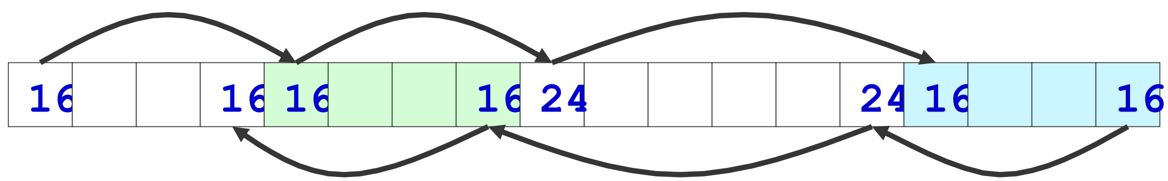
Implicit Free Lists: Bidirectional Coalescing

- Boundary tags [Knuth73]



`a = 1`: Allocated block
`a = 0`: Free block
`Size`: block size

payload: application data (allocated blocks only)



Implicit Free Lists: Summary

- Implementation
  - Very simple
- Allocation
  - linear-time worst case
- Free
  - Constant-time worst case—even with coalescing
- Memory usage
  - Will depend on placement policy
  - First fit, best fit,…

Implicit Free Lists: Summary

- Not used in practice for `malloc/free`
  - linear-time allocation is actually slow!
  - But used in some special-purpose applications

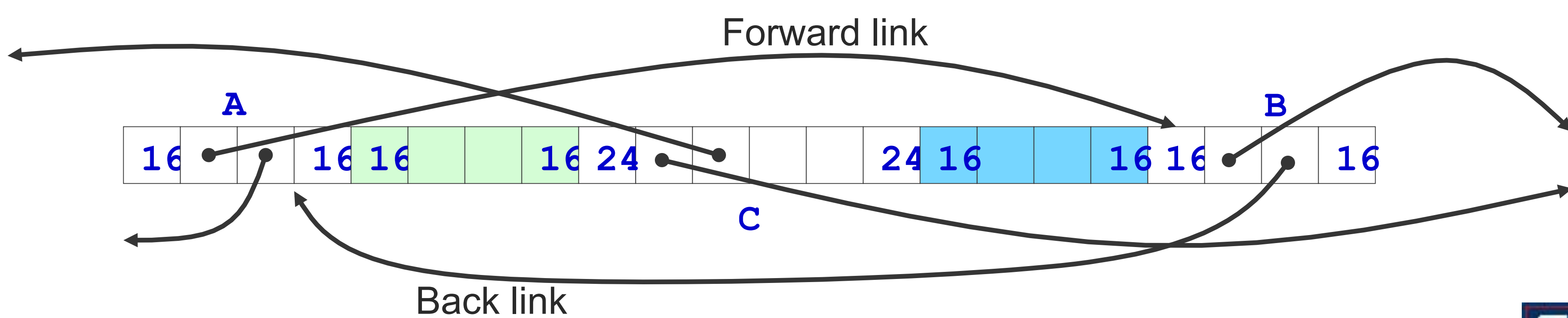
- However, concepts of splitting and boundary tag coalescing are general to all allocators

Alternative Approaches

- Explicit Free List
- Segregated Free Lists
  - Buddy allocators

Explicit Free List

- Linked list among free blocks
- Use data space for link pointers
  - Typically doubly linked
  - Still need boundary tags for coalescing
  - Links aren't necessarily in address order!



Explicit Free List: Inserting Free Blocks

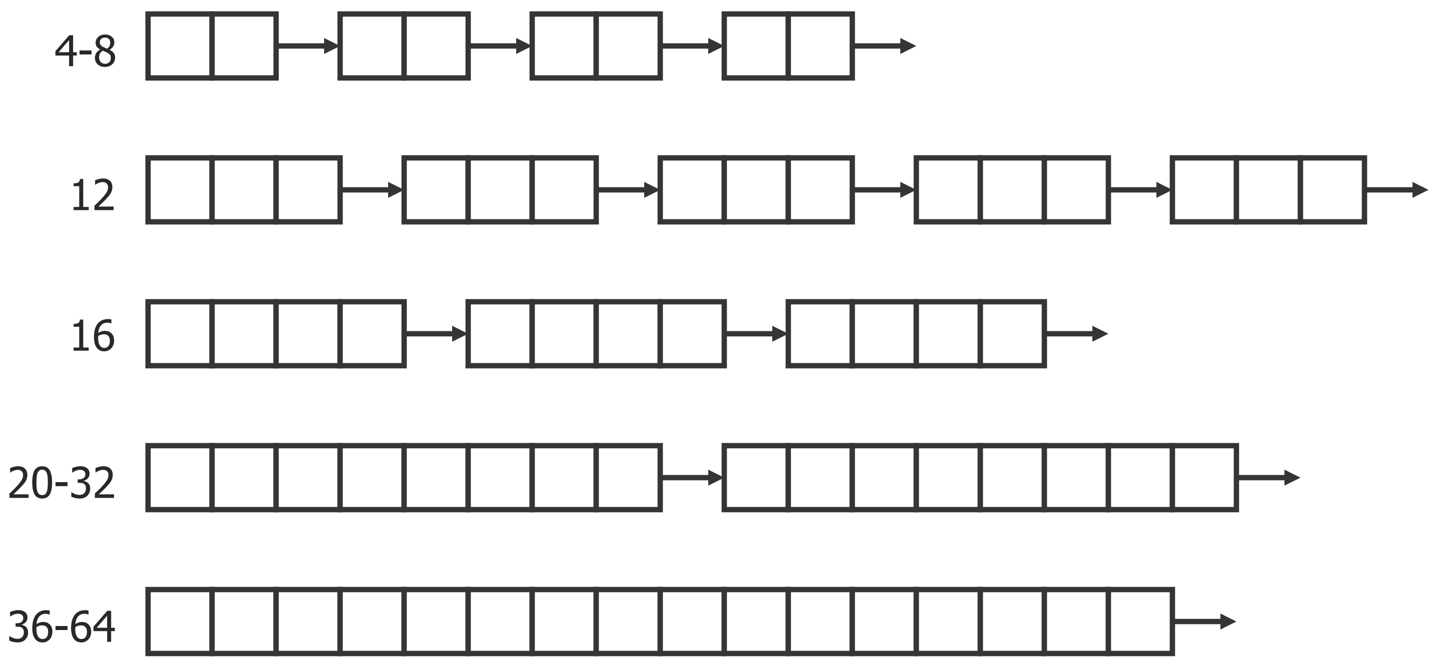
- Where should you put the newly freed block?
  - LIFO (last-in-first-out) policy
    - Insert freed block at beginning of free list
    - Pro
      - Simple, and constant-time
    - Con
      - Studies suggest fragmentation is high

Explicit Free List: Inserting Free Blocks

- Where should you put the newly freed block?
  - Address-ordered policy
    - Insert so list is always in address order
      - i.e. `addr(pred) < addr(curr) < addr(succ)`
    - Con
      - Requires search (using boundary tags); slow!
    - Pro
      - studies suggest fragmentation is better than LIFO

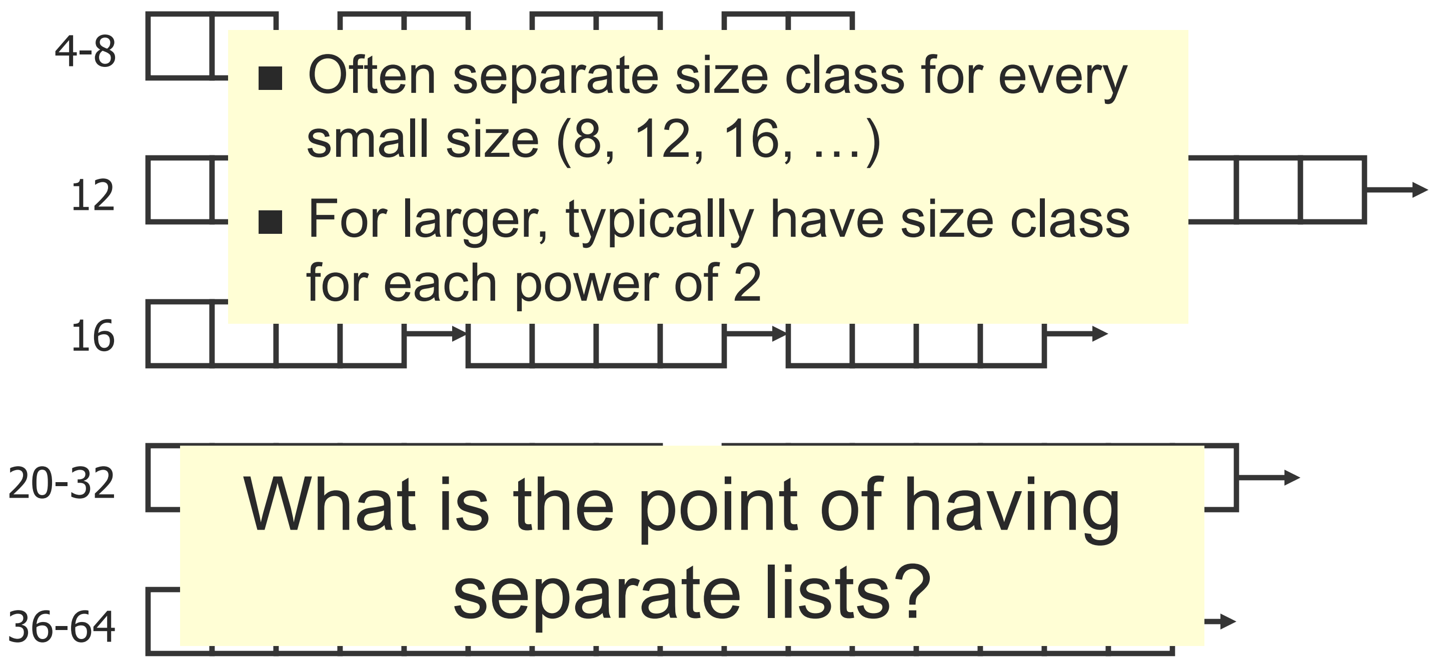
Segregated Free Lists

- Each size class has its own collection of blocks



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
  - Limit allocations 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
- Problem
  - Large powers of two result in large internal fragmentation (e.g., what if you want to allocate 65537 bytes?)

Buddy System

- Approach
  - Minimum allocation size = smallest frame
  - Maintain freelist for each possible frame size
    - Power of 2 frame sizes from min to max
  - Initially one block = entire buffer
  - If two neighboring frames (“buddies”) are free, combine them and add to next larger freelist

Buddy System Example

128 Free

Buddy System Example

Request A: 16

128 Free

64 Free	64 Free

32 Free	32 Free	64 Free

16 A	16 Free	32 Free	64 Free

Buddy System Example

Request B: 32

16 A	16 Free	32 B	64 Free

Buddy System Example

Request C: 8

16 A	16 Free	32 B	64 Free

16 A	8 C	8	32 B	64 Free

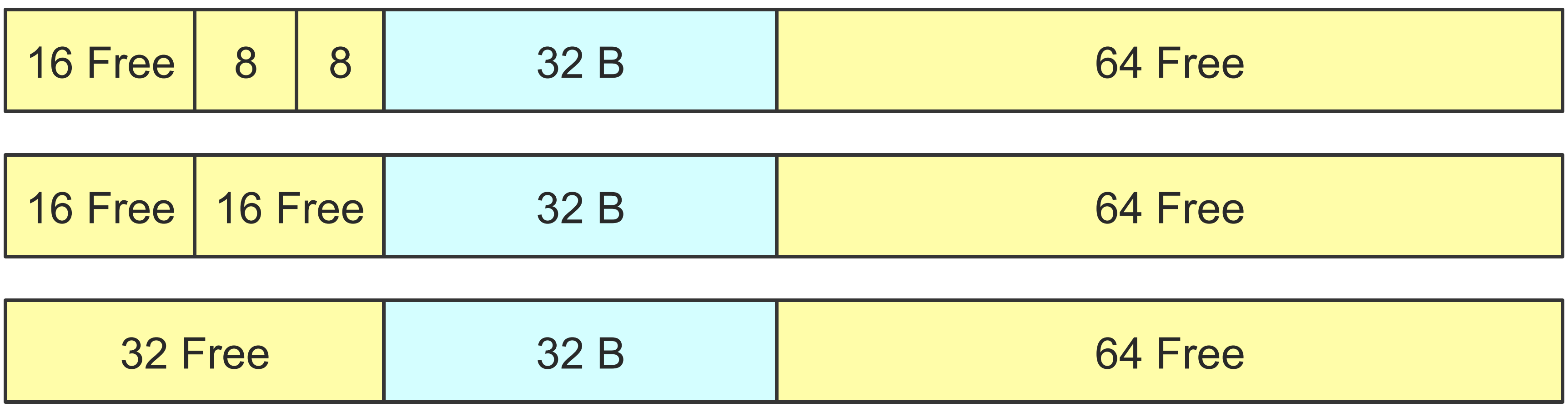
Buddy System Example

Request A frees

16 Free	8 C	8	32 B	64 Free

Buddy System Example

Request C frees



- Advantage
  - Minimizes external fragmentation
- Disadvantage
  - Internal fragmentation when not 2^n request

So what should I do for MP2?

- Designs sketched here are all reasonable
- But, there are many other possible designs
- So, implement anything you want!
- Suggestion:

➔ Before you start coding, **REALLY** spend time thinking about 1) your mem. manag. design; 2) its correctness; 3) the assumptions your code relies on; 4) performance trade-offs

Happy coding!