



CS 240

Foundations of Computer Systems



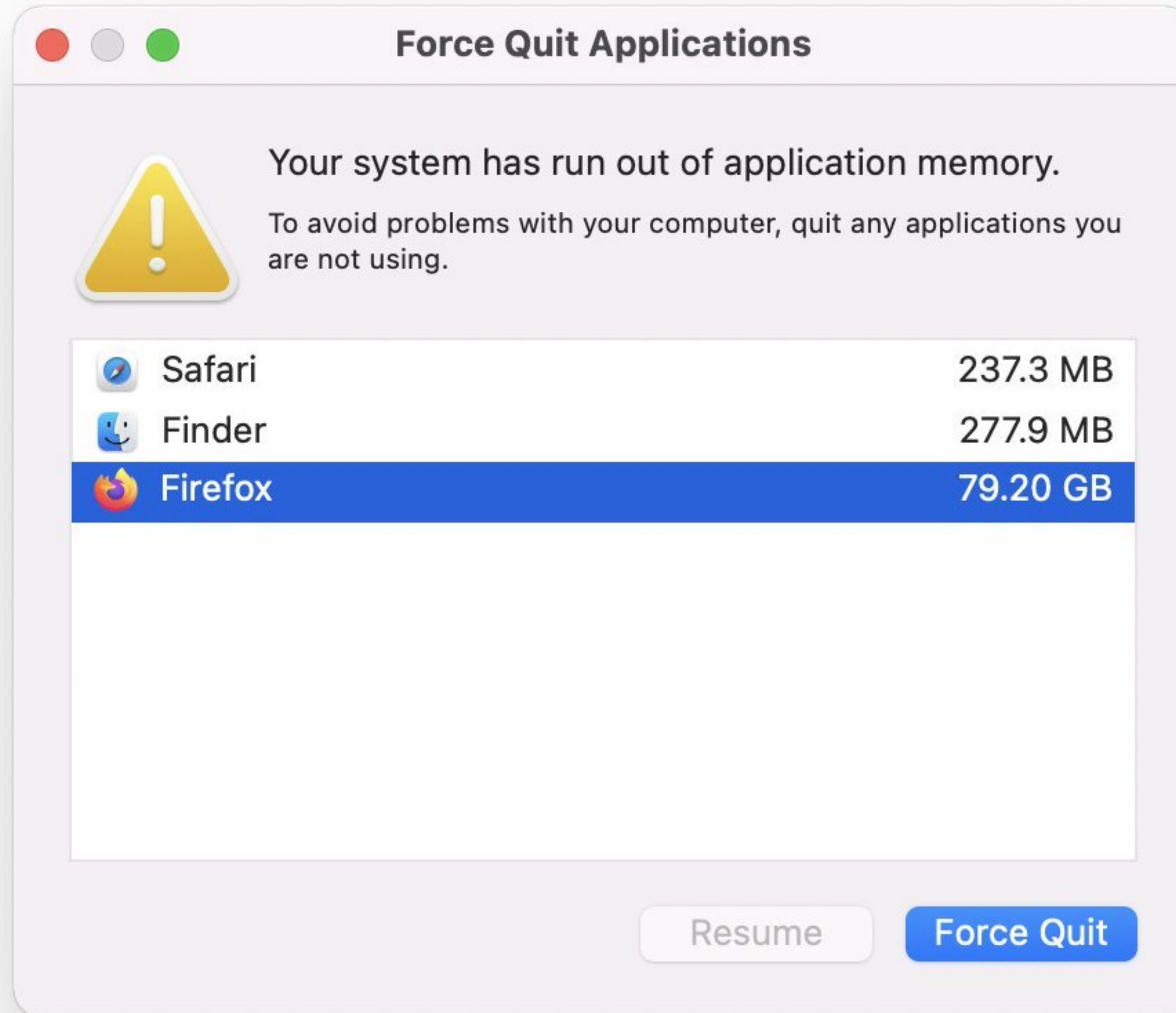
Dynamic Memory Allocation in the Heap

Explicit allocators

Manual memory management

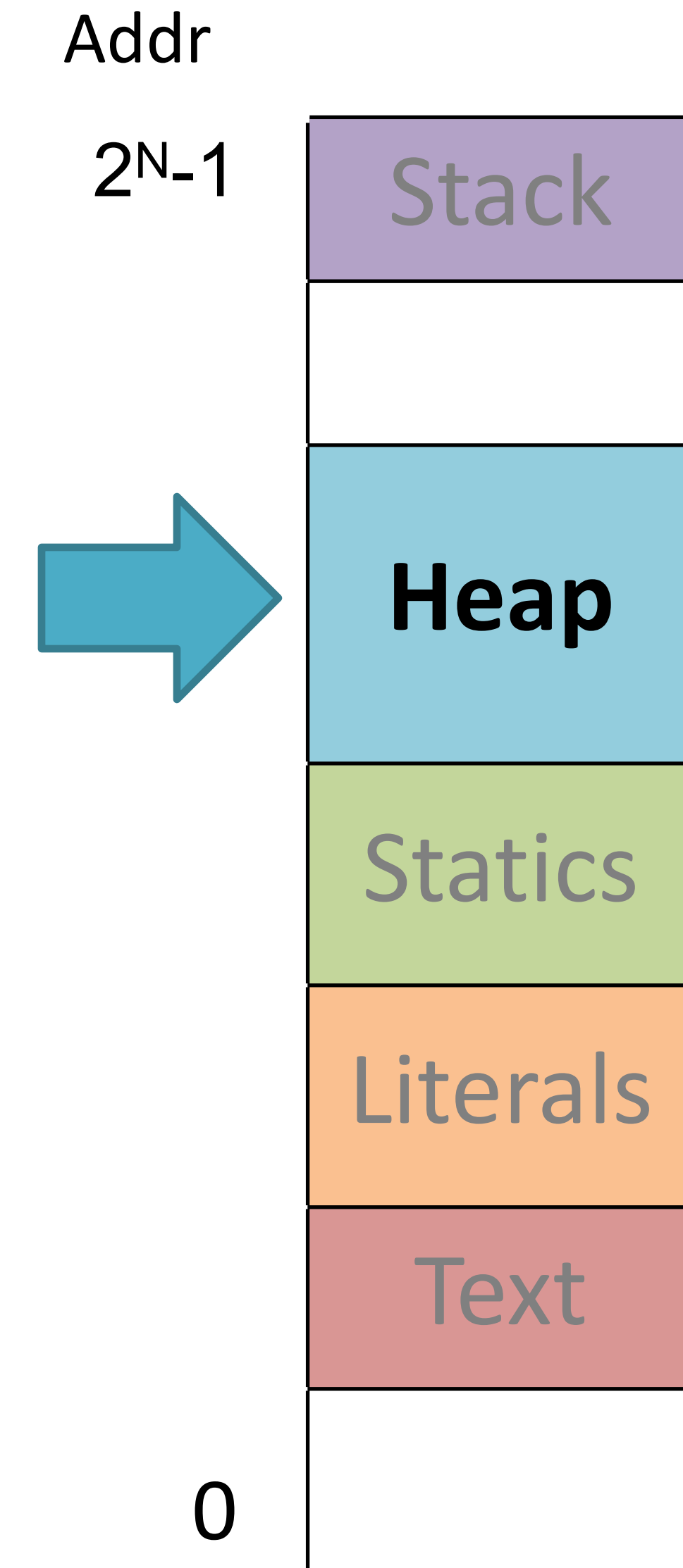
C: implementing malloc and free

Big picture: how is dynamic memory implemented in C?

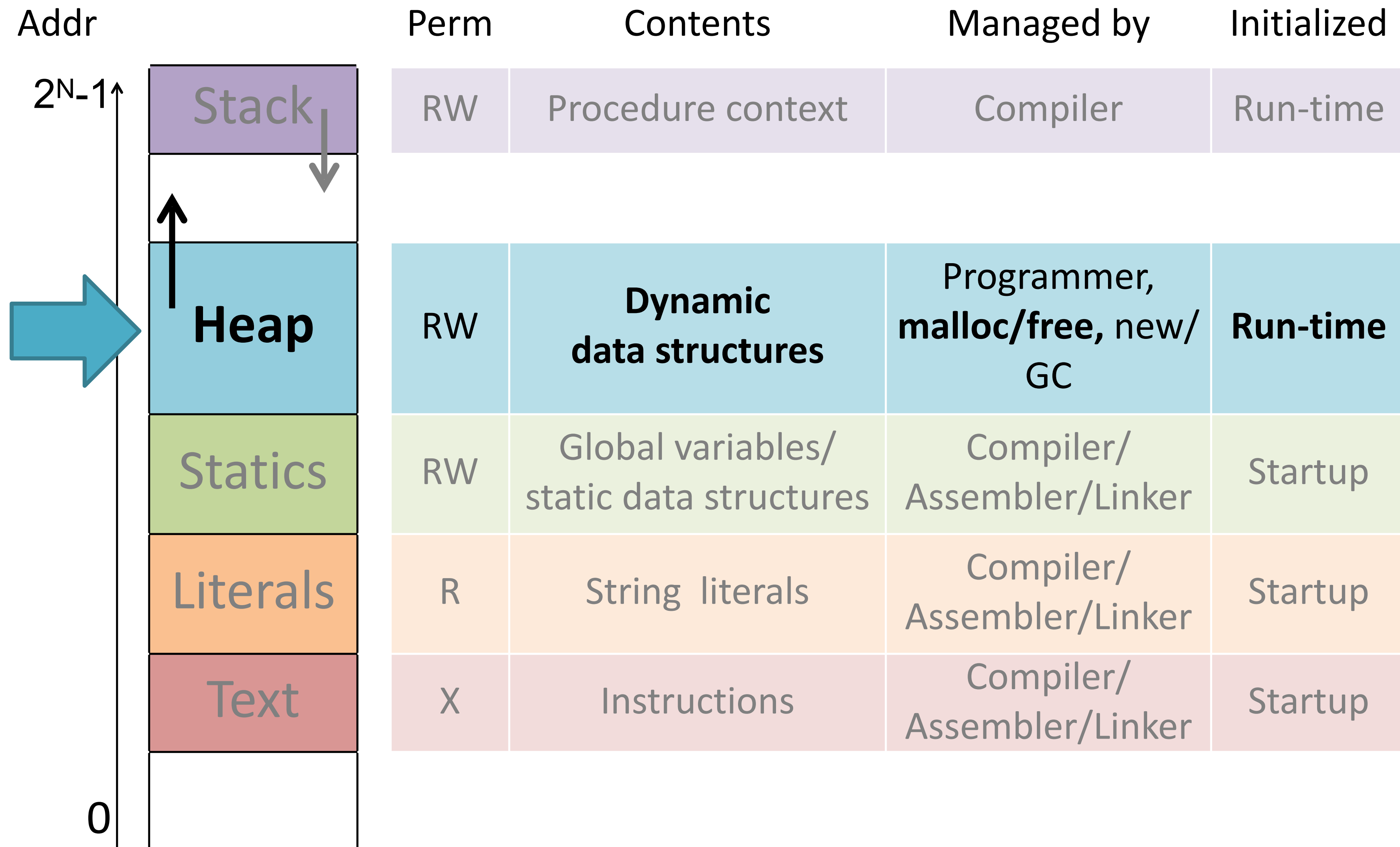


Outline

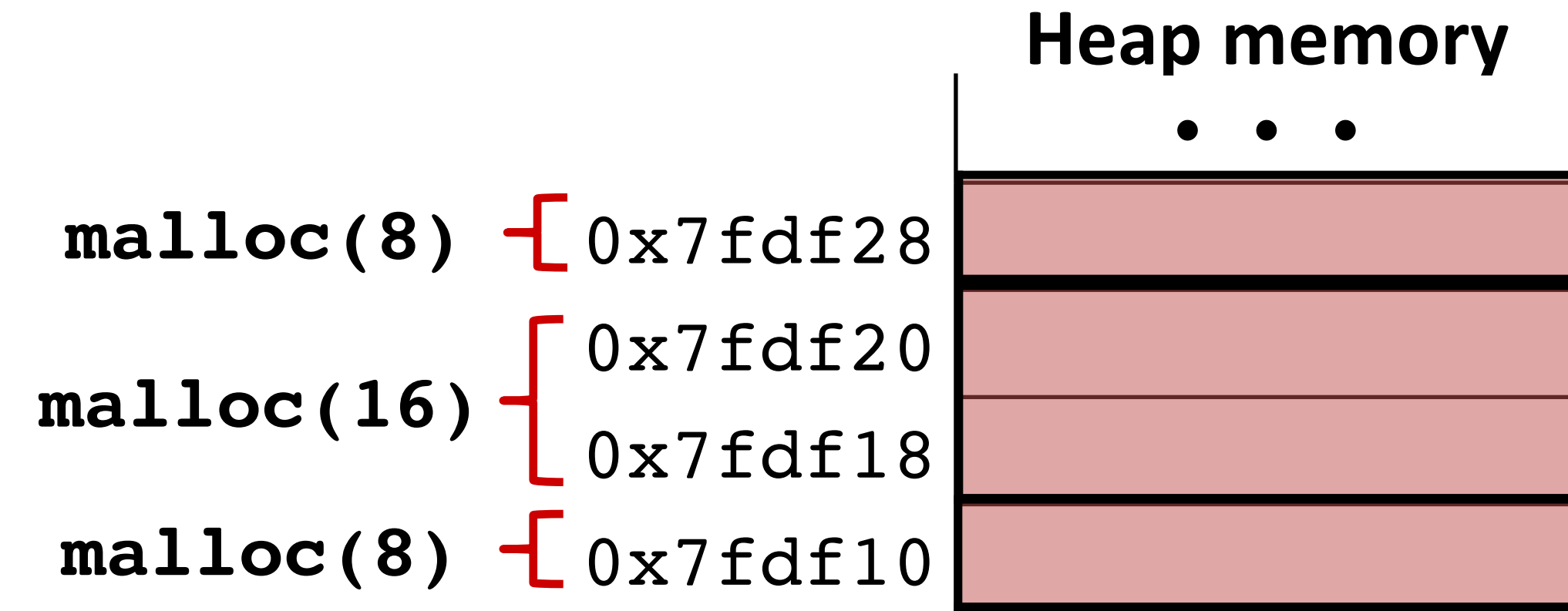
- Motivation/alternatives
- Design goals for a memory allocator
 - Utilization/fragmentation
- Implicit free list allocator
 - Tracking sizes
 - Allocating blocks
 - Coalescing blocks
- Explicit free lists
 - List vs. memory order
 - Freeing/coalescing



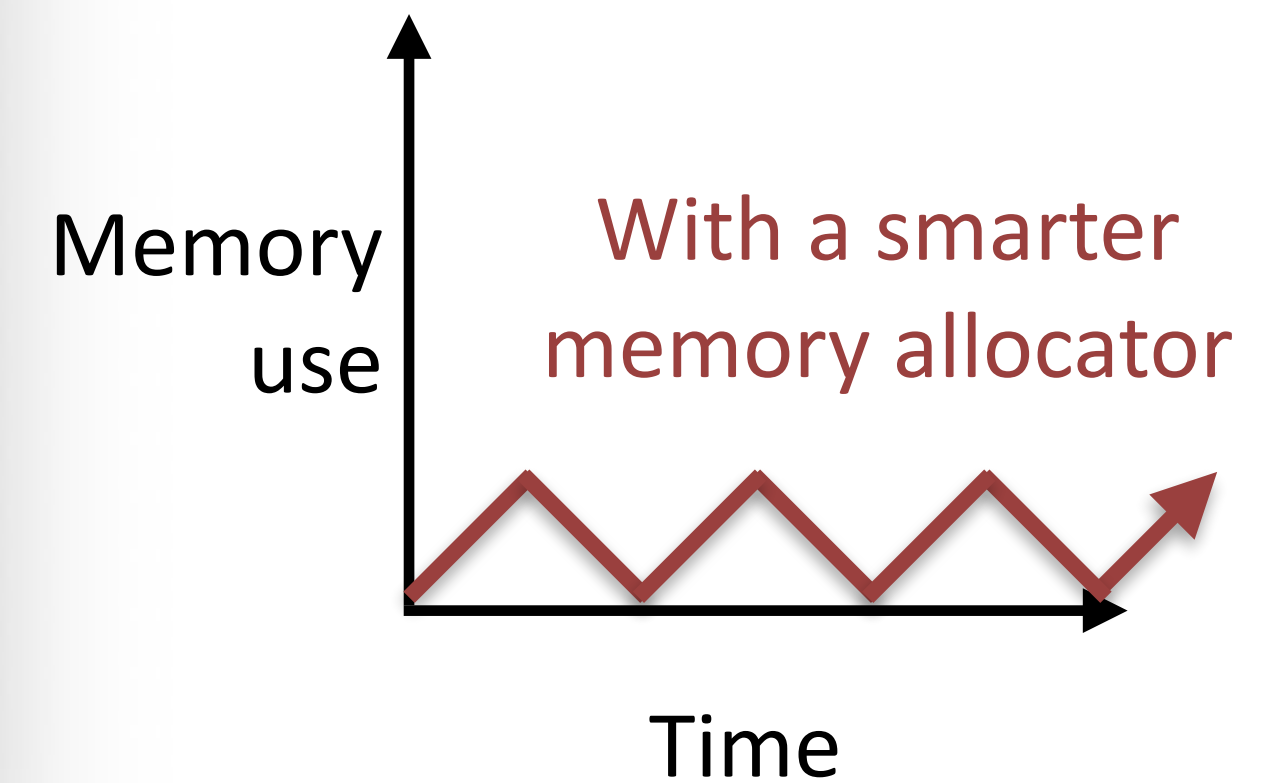
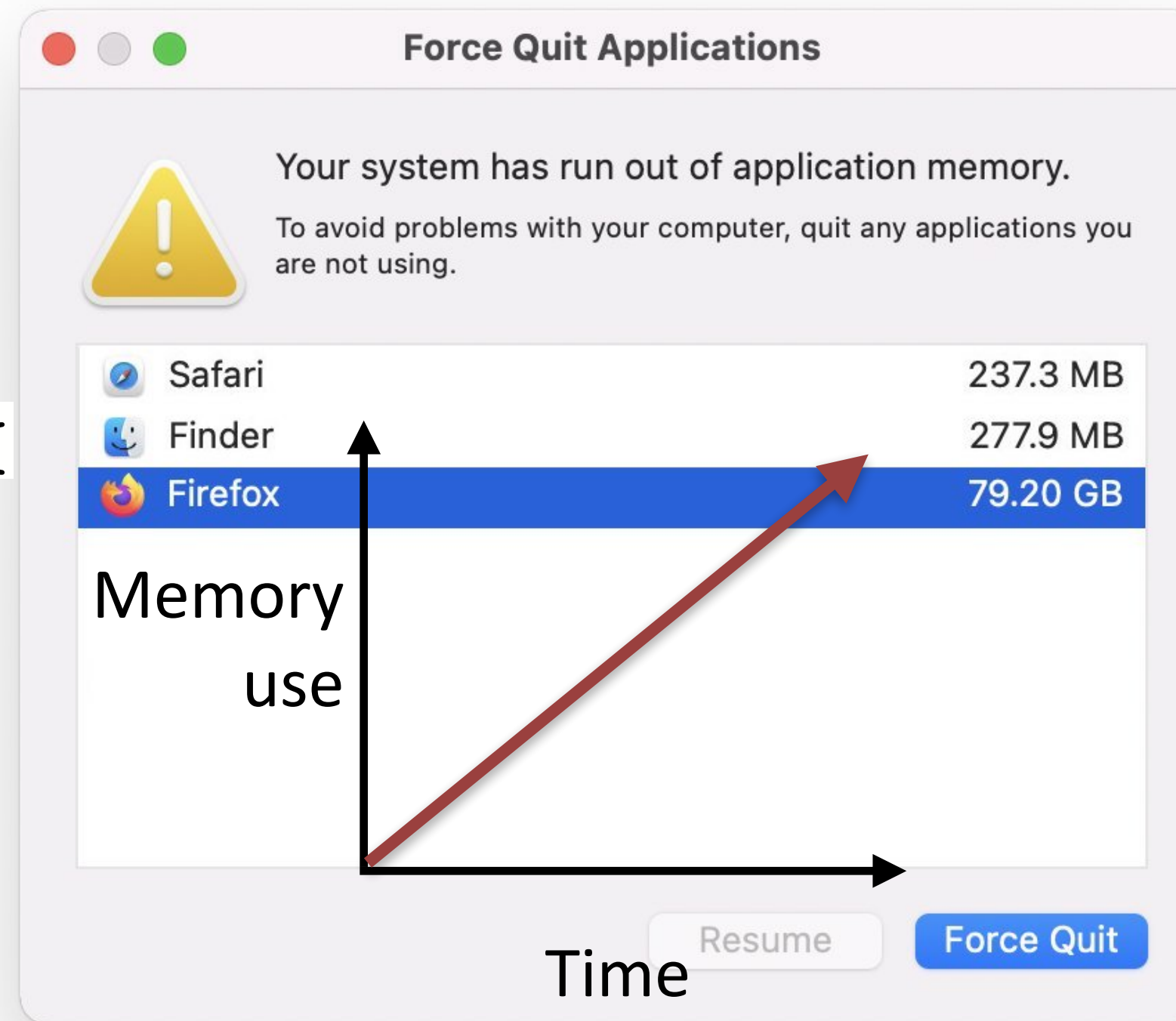
Heap Allocation



Motivation: why not just allocate in memory order?



```
void process_incoming_data(int data[]) {  
    // Build complicated data structures  
    // ...  
    print("%d", result);  
    // Don't need data or backing work!  
}
```



Motivation: what data do we need to track?

ex

Idea: given a page (4096 bytes), support these two functions

pointer to newly allocated block
of at least that size

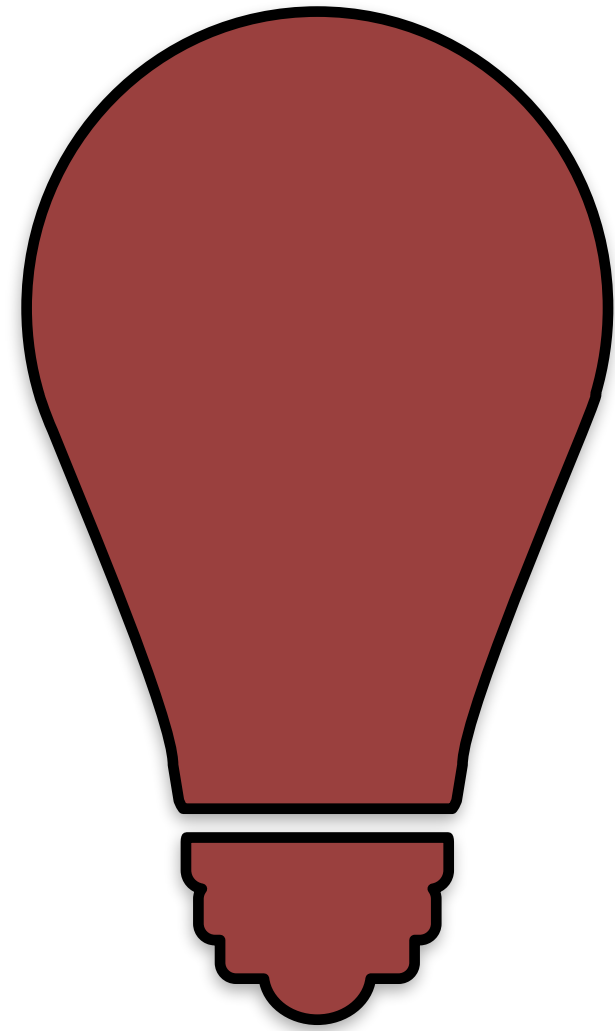
number of contiguous bytes required

 `void*` `malloc` (`size_t` `size`);

`void` `free` (`void*` `ptr`); 

What data structures could we use to track this?

Actual dynamic memory allocator design

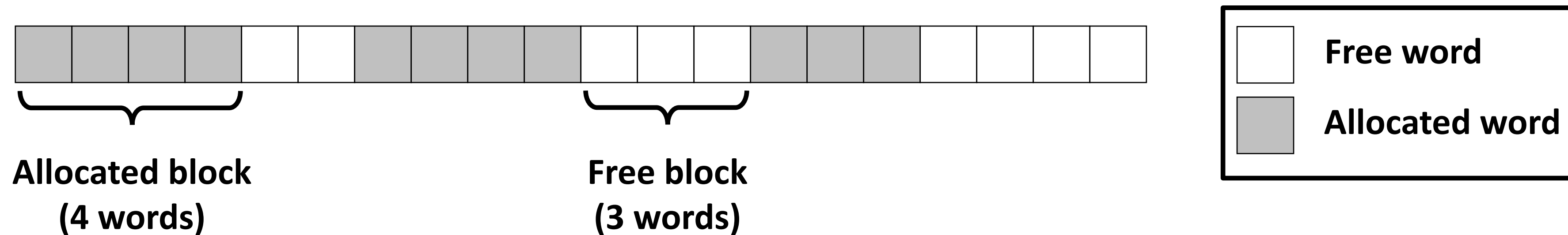


Design the allocator to store data
“inline” within the heap memory itself

- Space efficient: no need for much data “on the side”
- Use pointer arithmetic to calculate results
- Good use of caches/locality (we’ll cover more later)

Allocator basics

Pages (OS-provided) too coarse-grained for allocating individual objects.
Instead: flexible-sized, word-aligned blocks.



pointer to newly allocated block
of at least that size

void* malloc(size_t size);

number of contiguous bytes required

void free(void* ptr);

pointer to allocated block to free

Example (64-bit words)

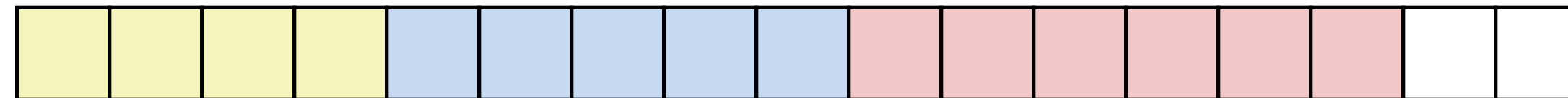
```
p1 = malloc(32);
```



```
p2 = malloc(40);
```



```
p3 = malloc(48);
```



```
free(p2);
```



```
p4 = malloc(16);
```



Allocator goals: malloc/free

1. Programmer does not decide locations of distinct objects.

Programmer decides: what size, when needed, when no longer needed

```
p = malloc(32);  
// ...  
free(p)
```

2. Fast allocation.

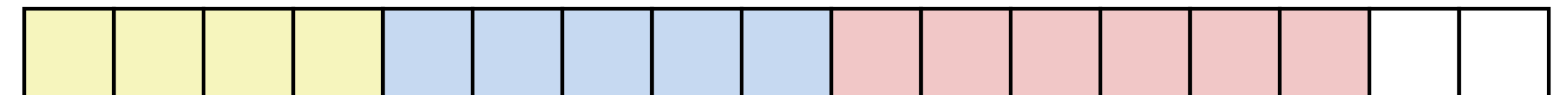
mallocs/second or bytes malloc'd/second



3. High memory utilization.

Most of heap contains necessary program data.

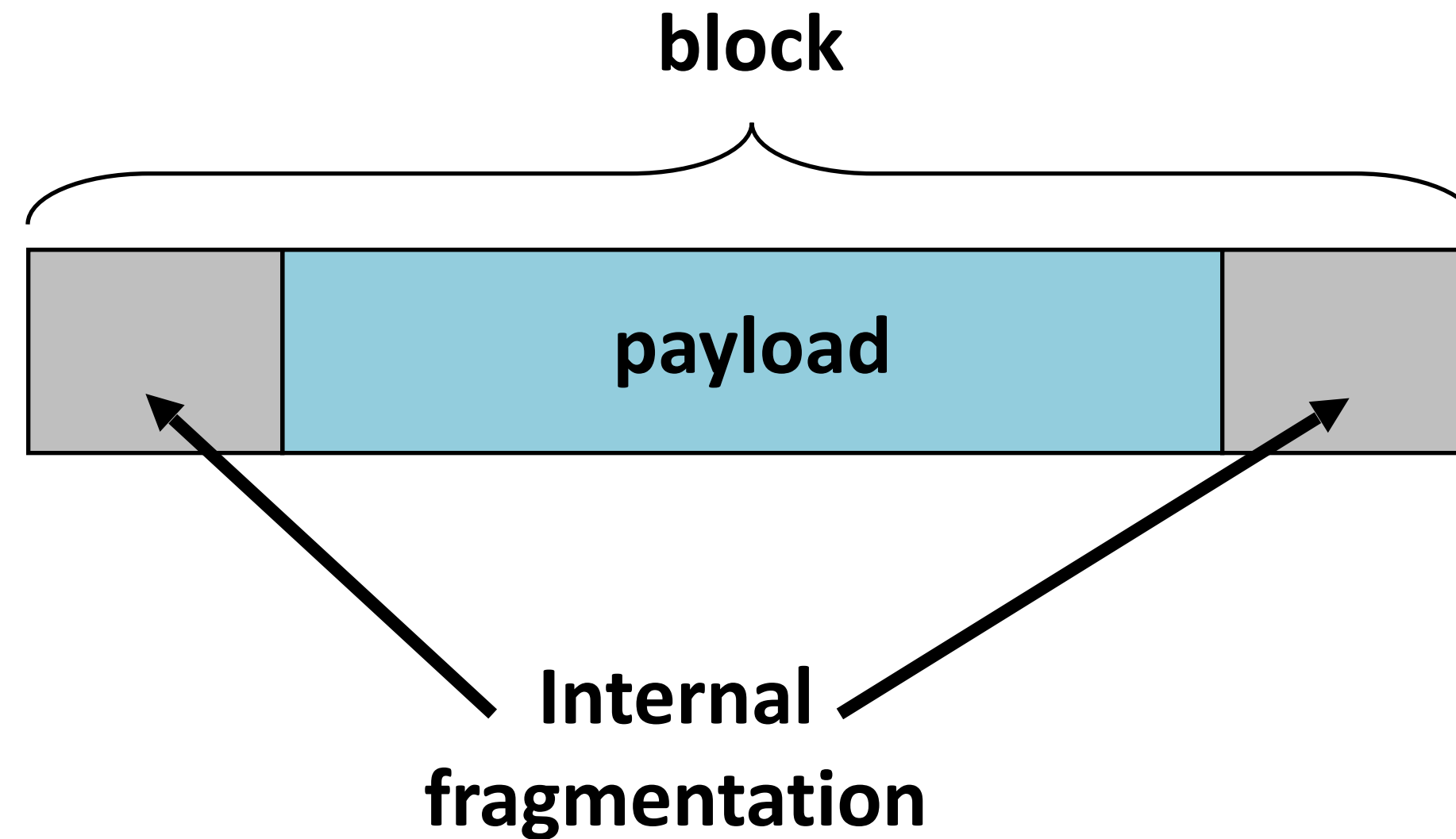
Little wasted space.



Enemy: **fragmentation** – unused memory that cannot be allocated.

Internal fragmentation

Payload smaller than block



Causes

- Metadata (bookkeeping)
- Alignment (8, 16, ...)
- Policy decisions

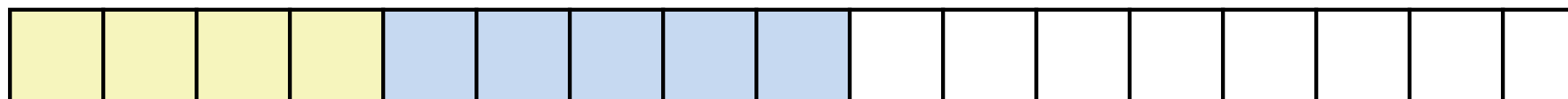
External fragmentation (64-bit words)

Total free space large enough, but no contiguous free block large enough!

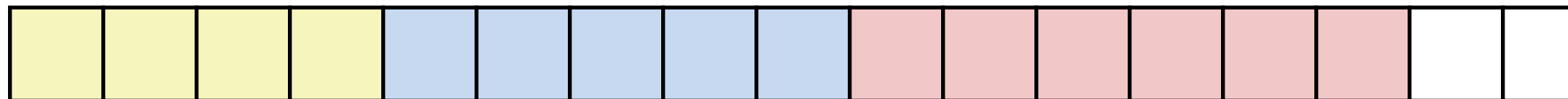
```
p1 = malloc(32) ;
```



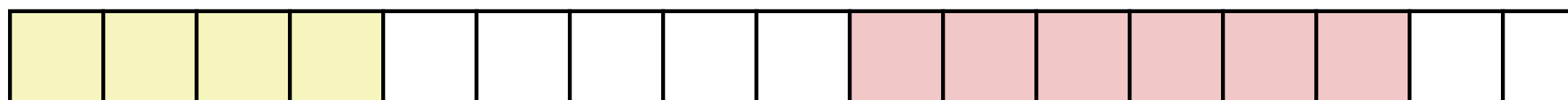
```
p2 = malloc(40) ;
```



```
p3 = malloc(48) ;
```



```
free(p2) ;
```



```
p4 = malloc(48) ;
```

Depends on the pattern of future requests.

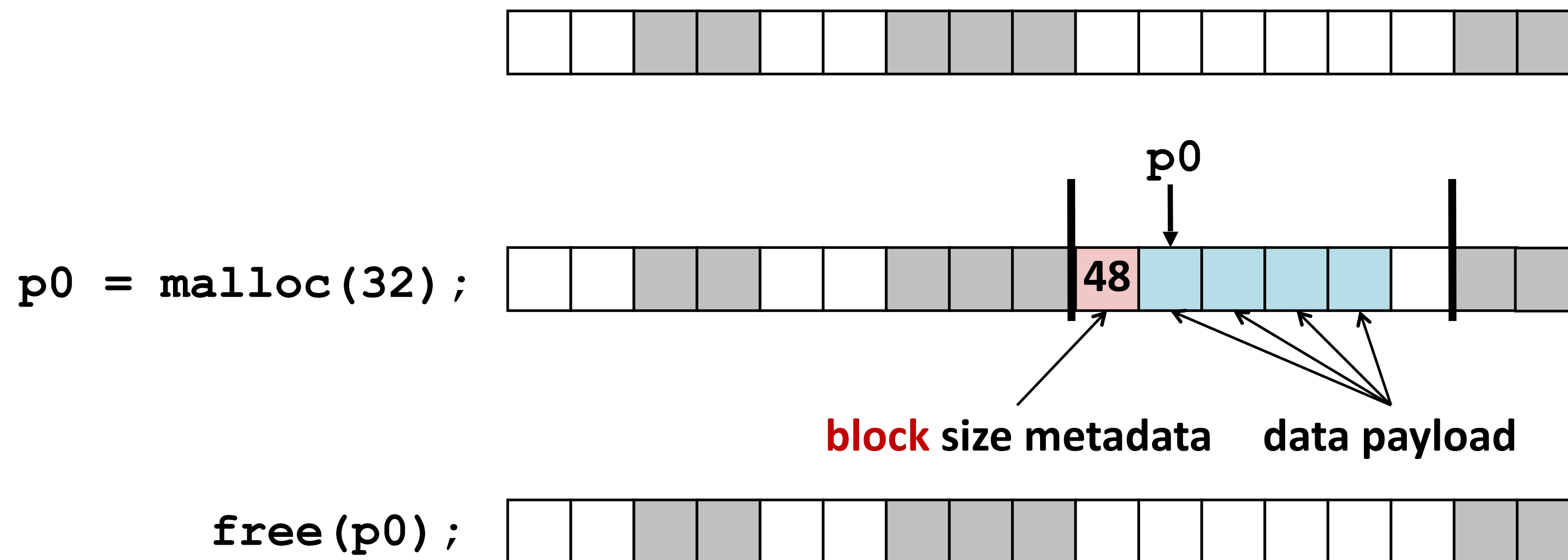
Implementation issues

1. Determine **how much** to free given just a pointer.
2. Keep track of **free blocks**.
3. **Pick** a block to allocate.
4. Choose what do with **extra space** when allocating a structure that is smaller than the free block used.
5. Make a **freed block available** for future reuse.

Knowing how much to free

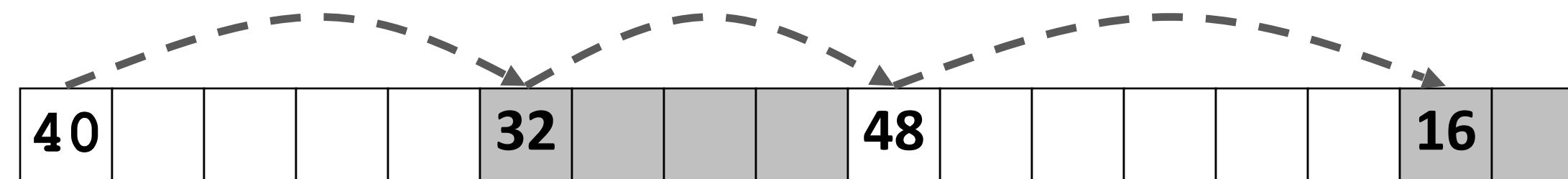
Keep length of block in *header* word preceding block

Takes extra space!

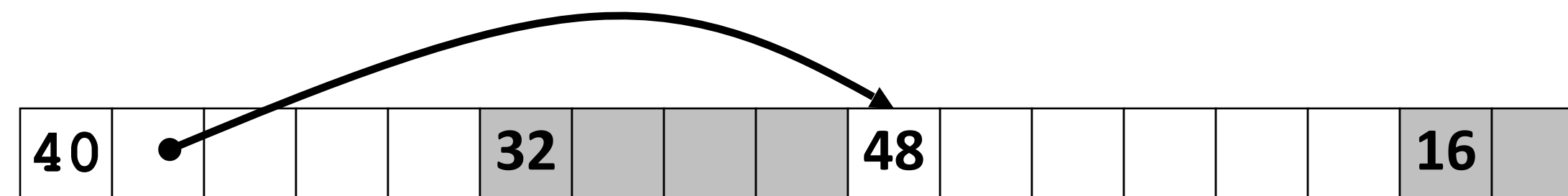


Keeping track of free blocks

Method 1: *Implicit free list* of all blocks using length



Method 2: *Explicit free list* of free blocks using pointers



Method 3: *Seglist*

Different free lists for different size blocks

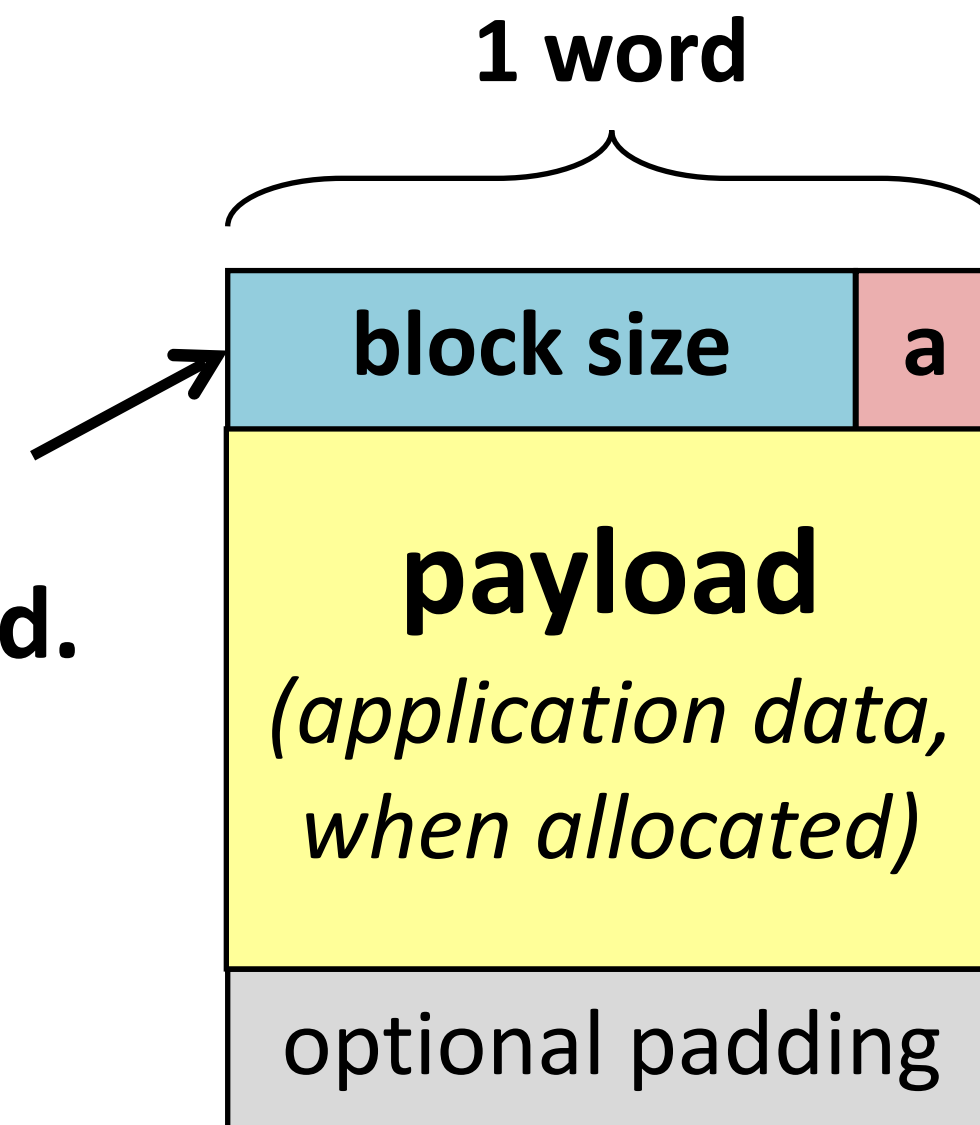
More methods that we will skip...

Implicit free list: block format

Block metadata:

1. Block size
2. Allocation status

Store in one header word.



Steal LSB for status flag.

LSB = 1: allocated

LSB = 0: free

16-byte aligned sizes have
4 zeroes in low-order bits

00000000

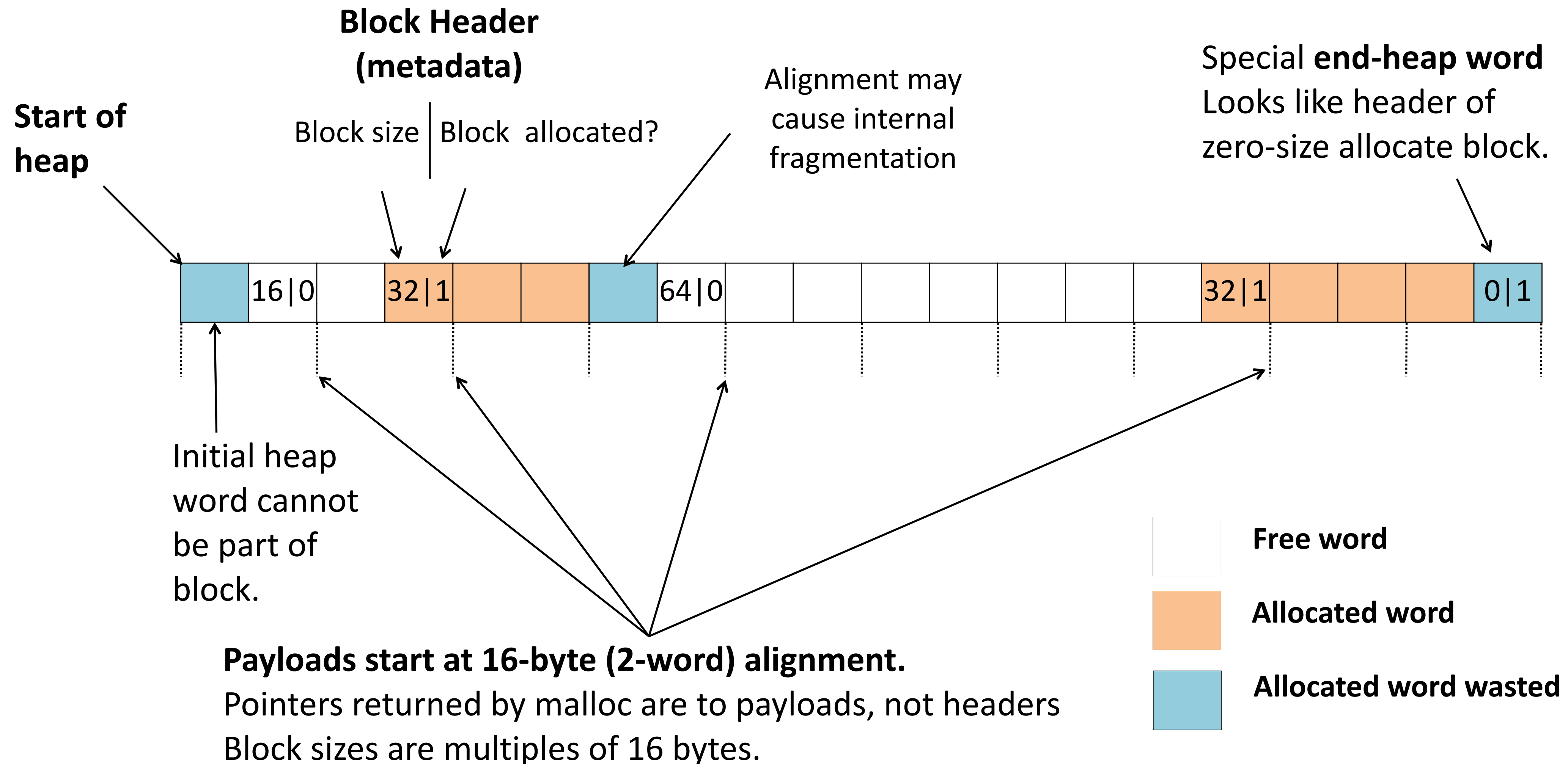
00010000

00100000

00110000

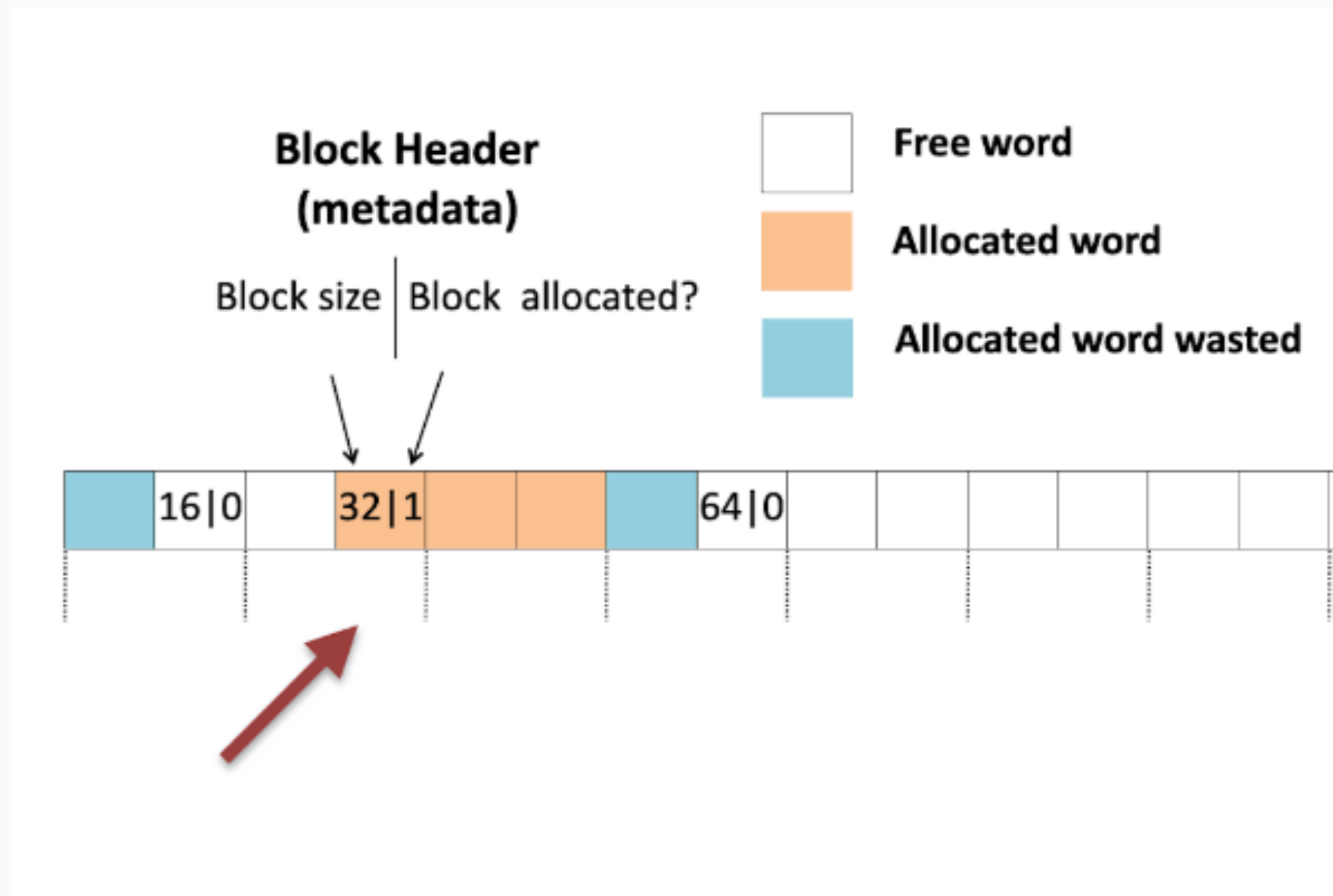
...

Implicit free list: heap layout



Recall: in this implicit free list heap, why does the block pointed to by the red arrow have size 32?

0



payload is 2 words, $2 \times 16 = 32$ (A)

payload is 2 words, header 2, $4 \times 8 = 32$ (B)

payload is 3 words, header 1, $4 \times 8 = 32$ (C)

payload 2 words, header 1, 1 wasted (align...) (D)

None of the above (E)

Implicit free list: **finding a free block**

First fit:

Search list from beginning, choose ***first*** free block that fits

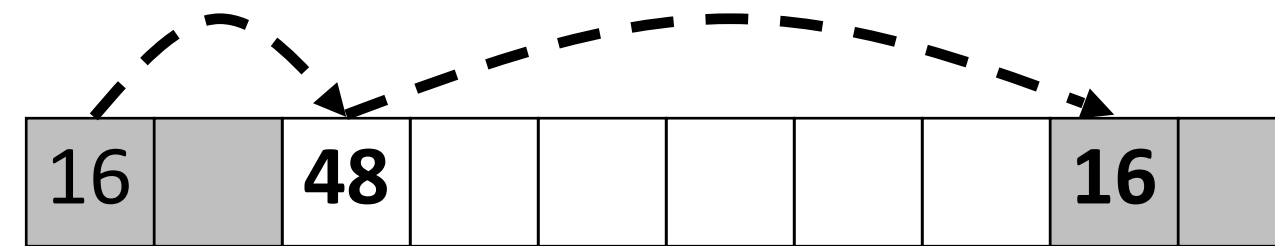
Next fit:

Do first-fit starting where previous search finished

Best fit:

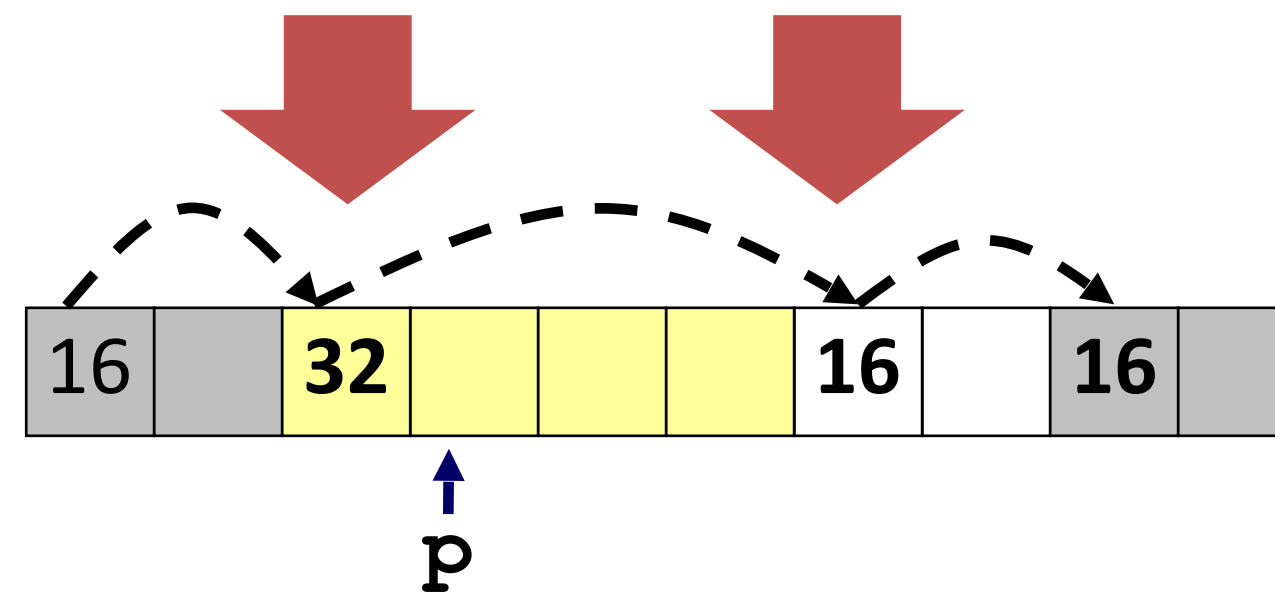
Search the list, choose the ***best*** free block: fits, with fewest bytes left over

Implicit free list: allocating a free block



```
p = malloc(24);
```

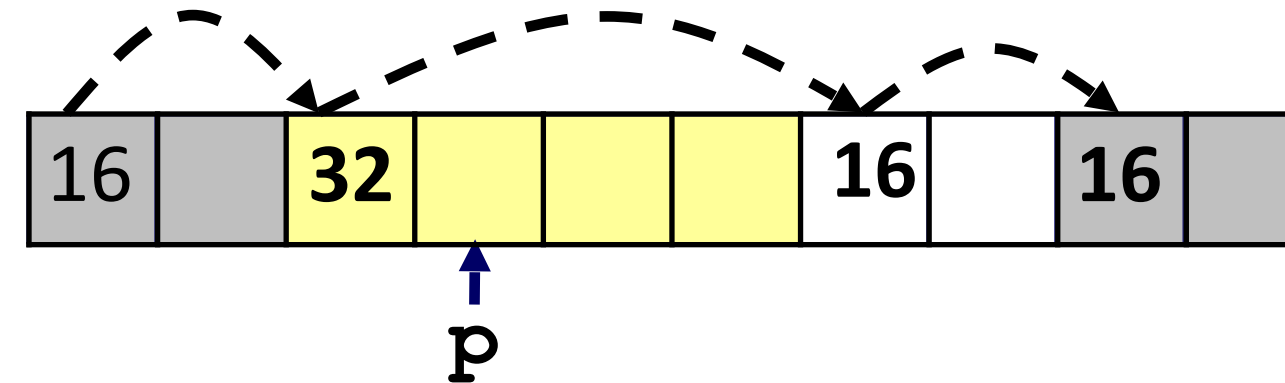
Allocated space $\leq$ free space.
Use it all? Split it up?



Block Splitting

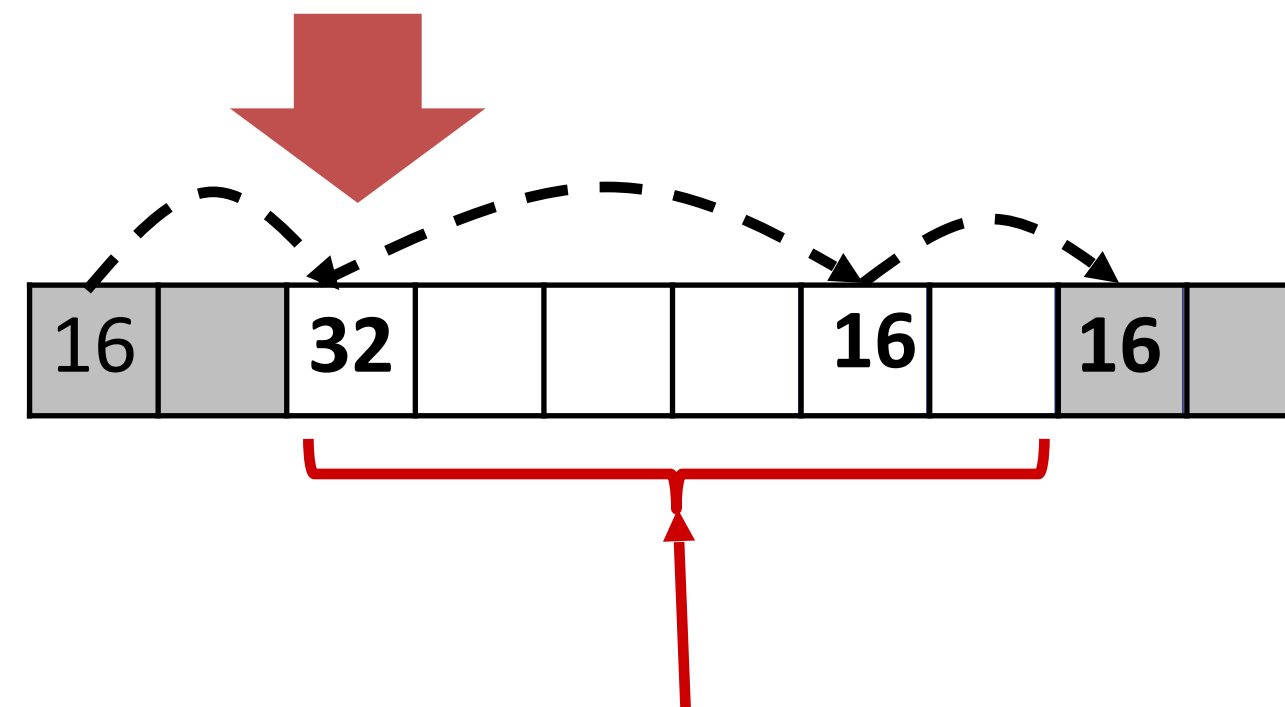
Now showing allocation status flag implicitly with shading.

Implicit free list: freeing an allocated block



`free(p) ;`

Clear *allocated* flag.



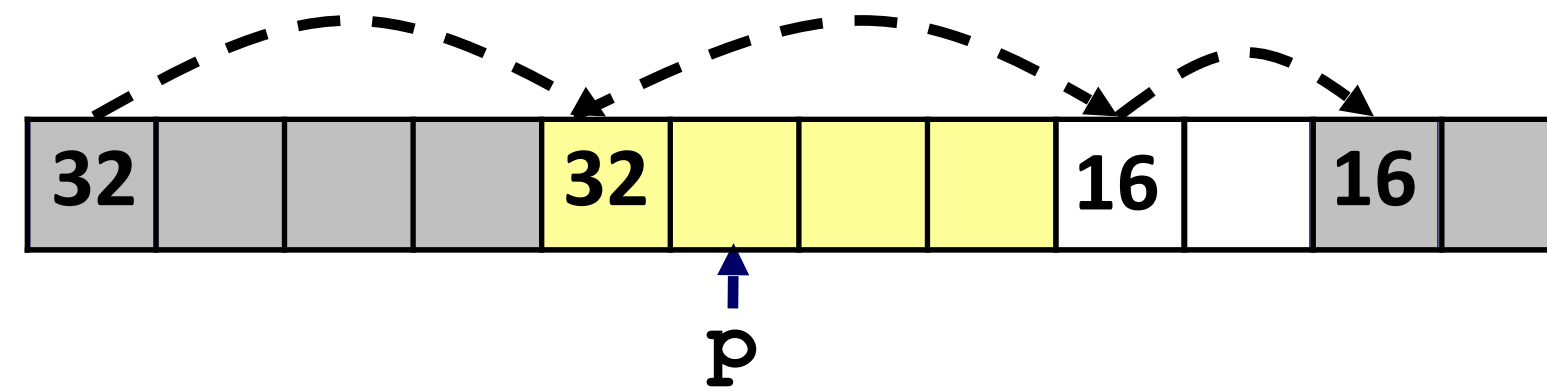
`malloc(40) ;`



External fragmentation!

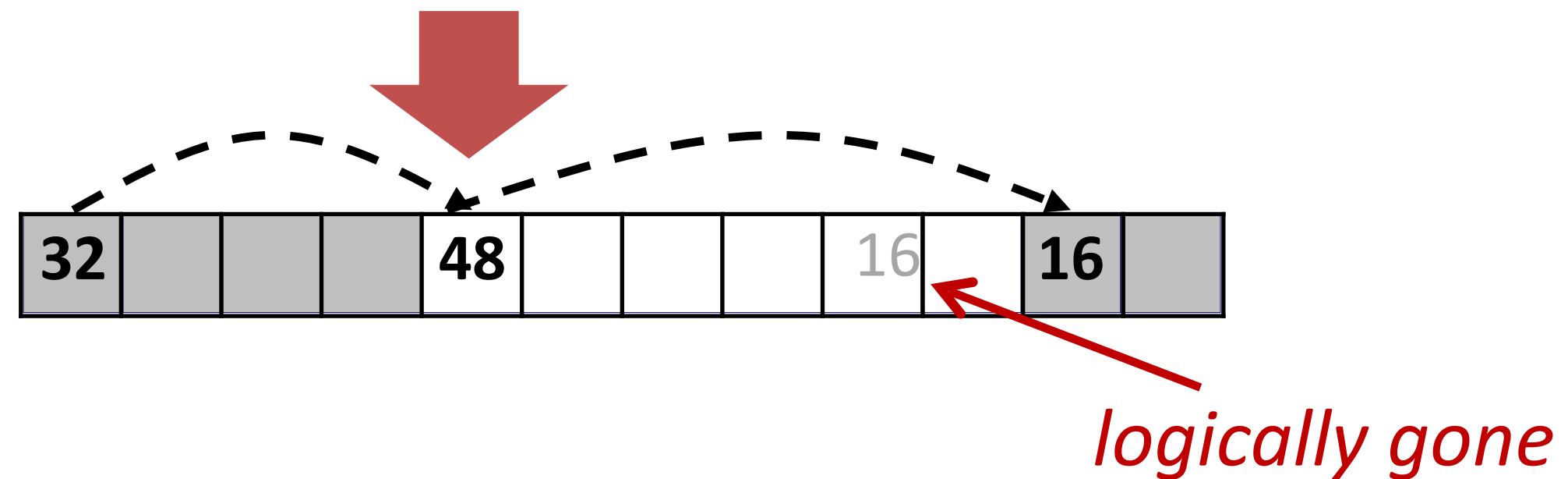
Enough space, not one block.

Coalescing free blocks



`free(p)`

Coalesce with following *free* block.

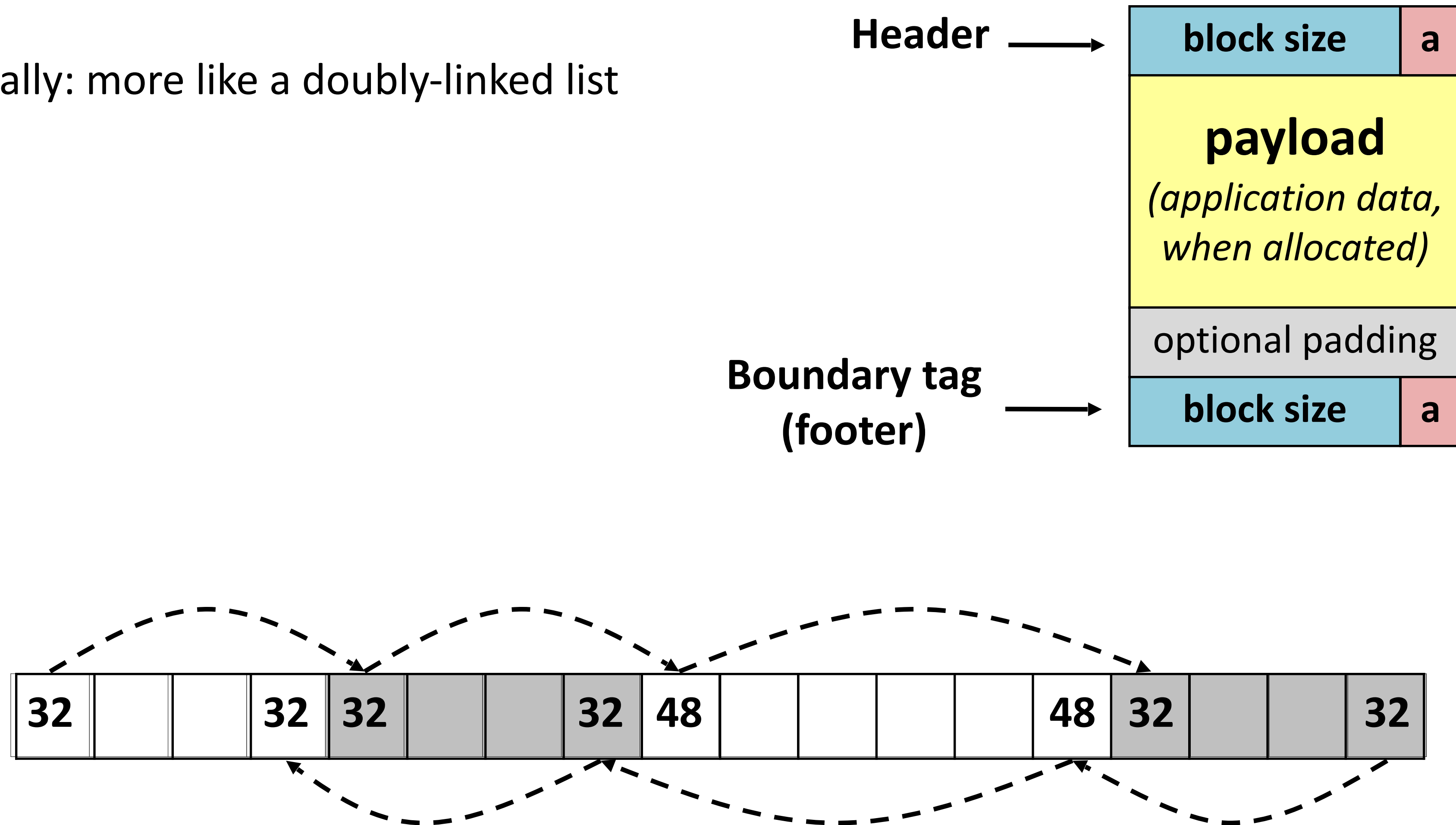


Coalesce with **preceding** *free* block?

Bidirectional coalescing: boundary tags

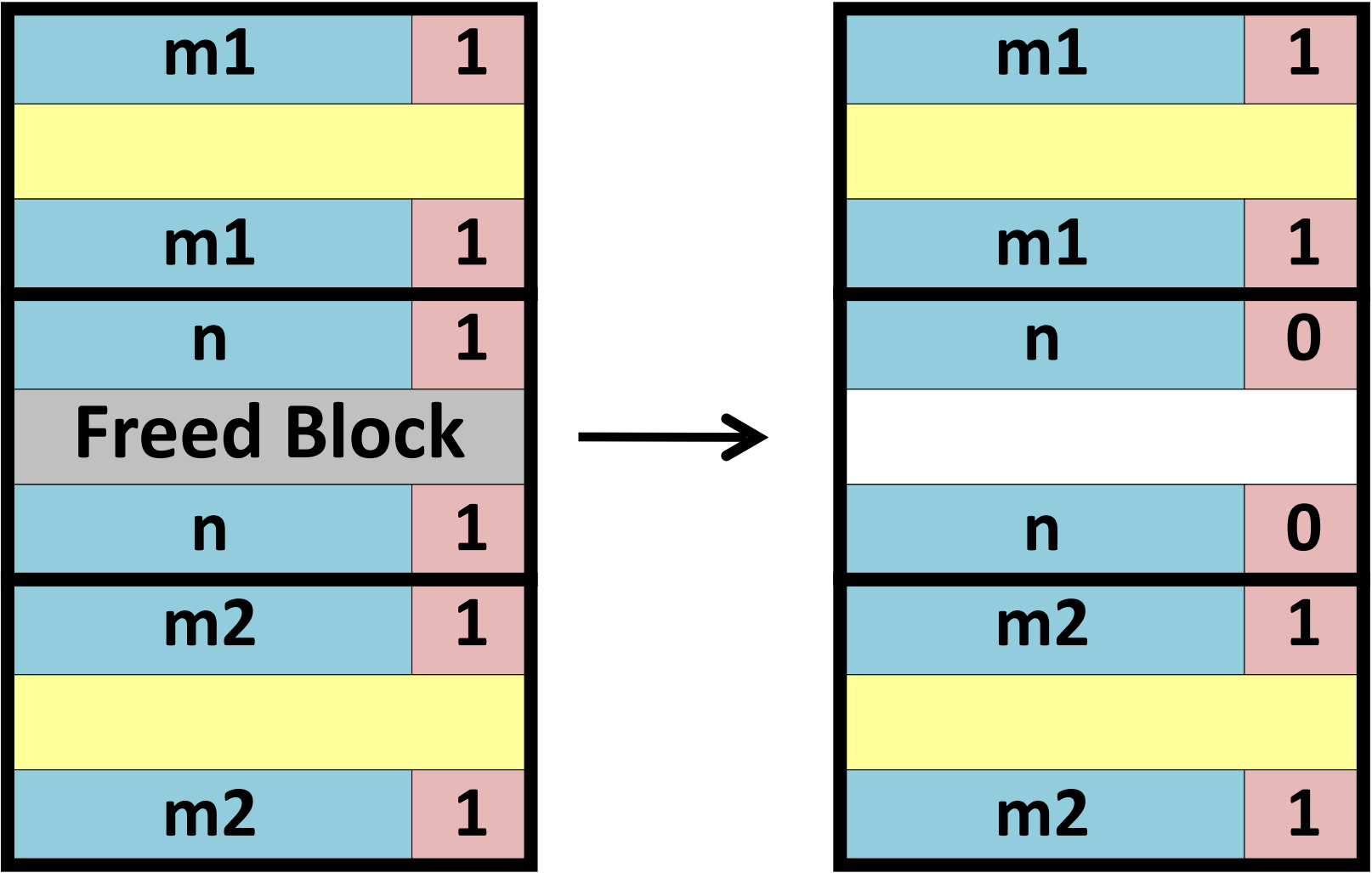
[Knuth73]

Conceptually: more like a doubly-linked list

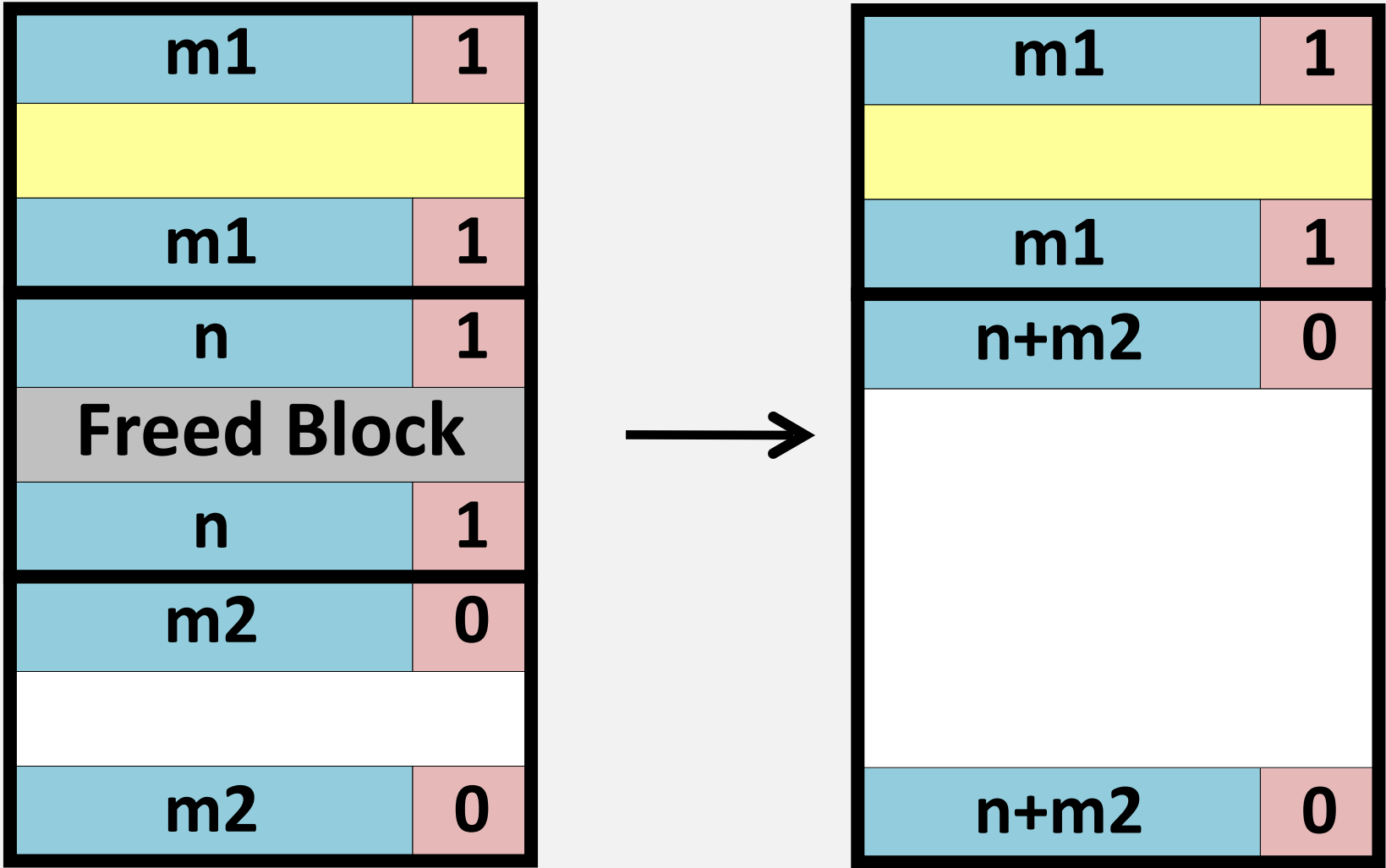


Constant-time $O(1)$ coalescing: 4 cases

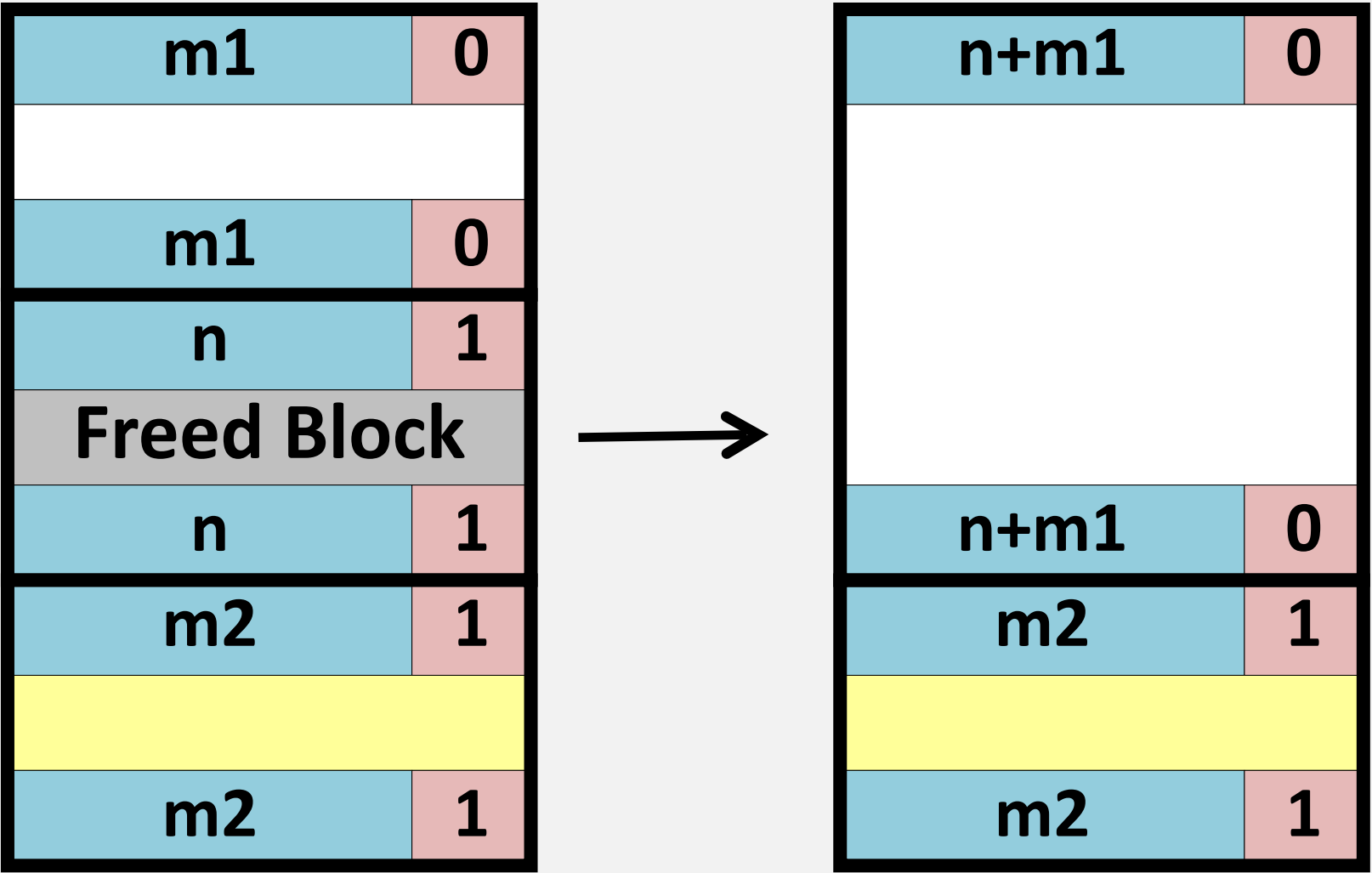
before: allocated
after: allocated



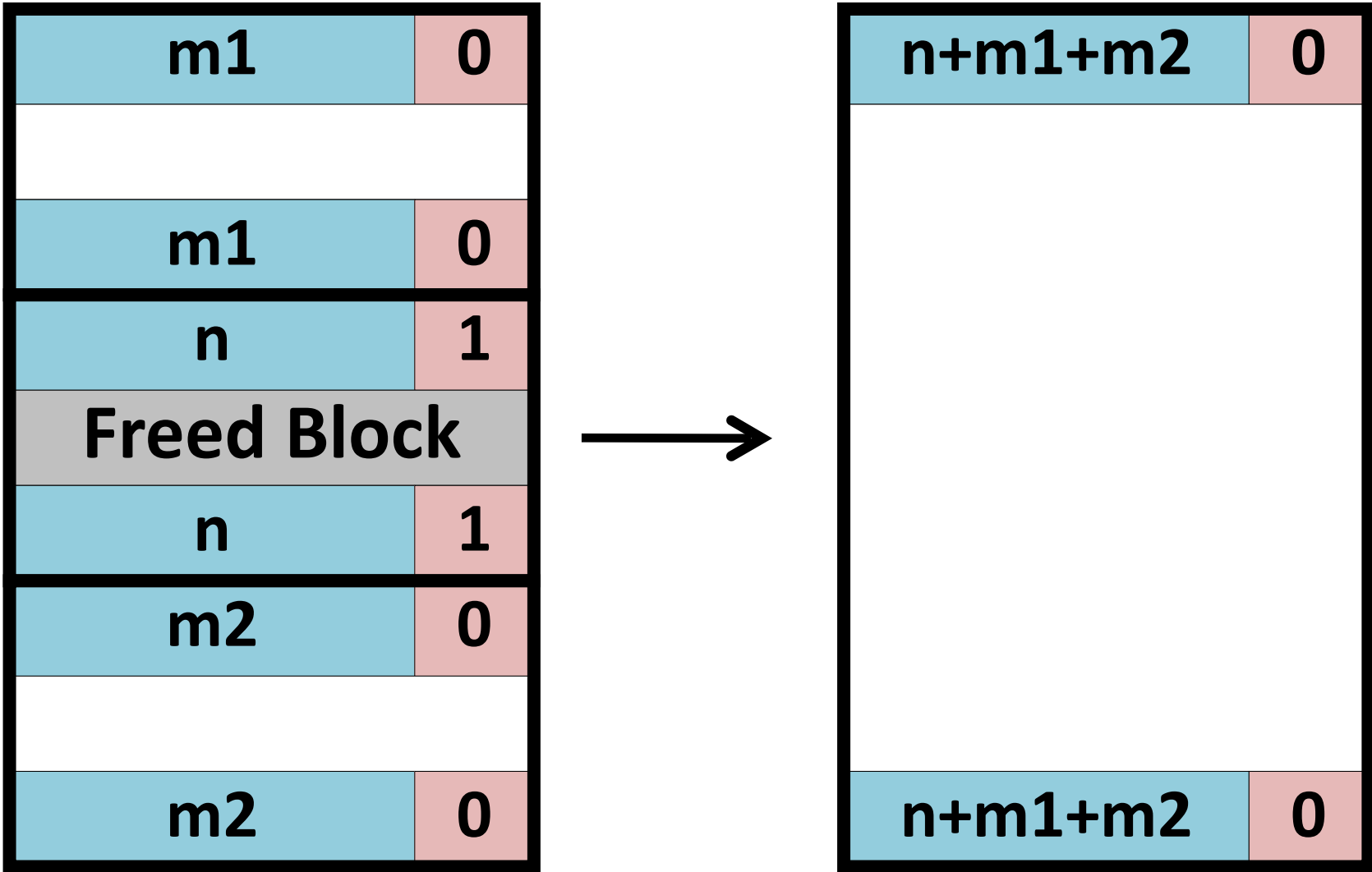
before: allocated
after: free



before: free
after: allocated

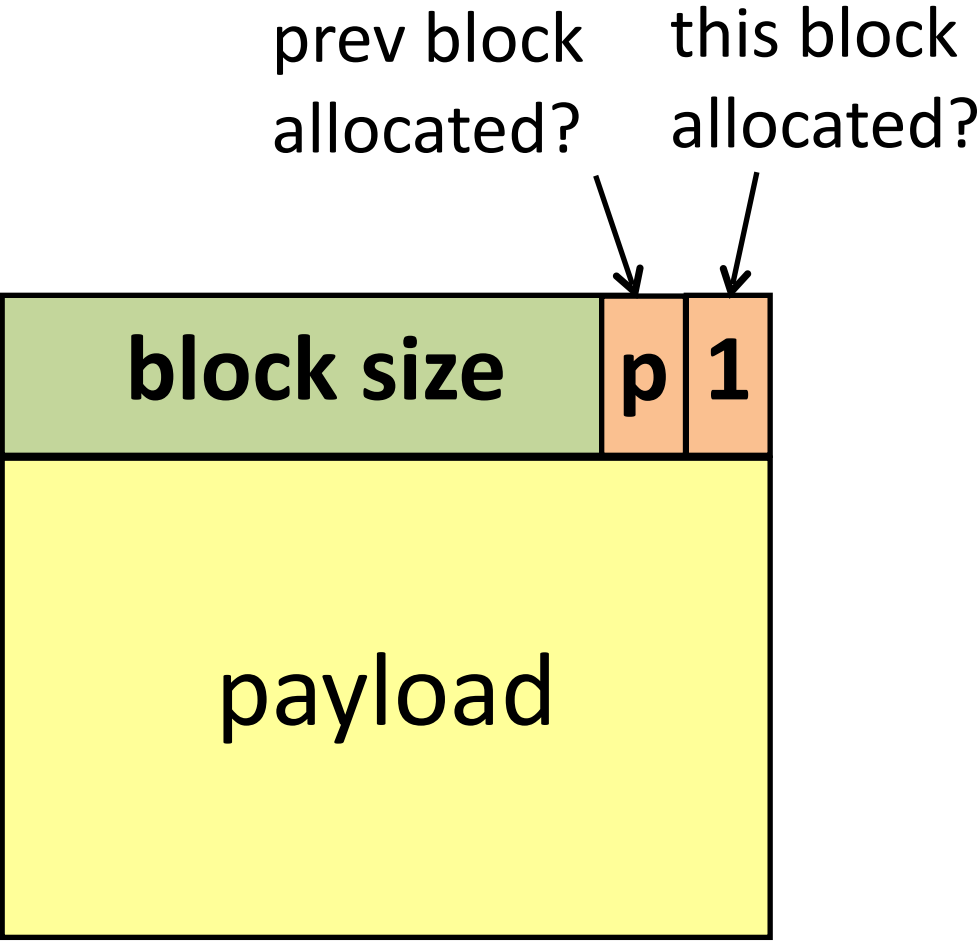


before: free
after: free

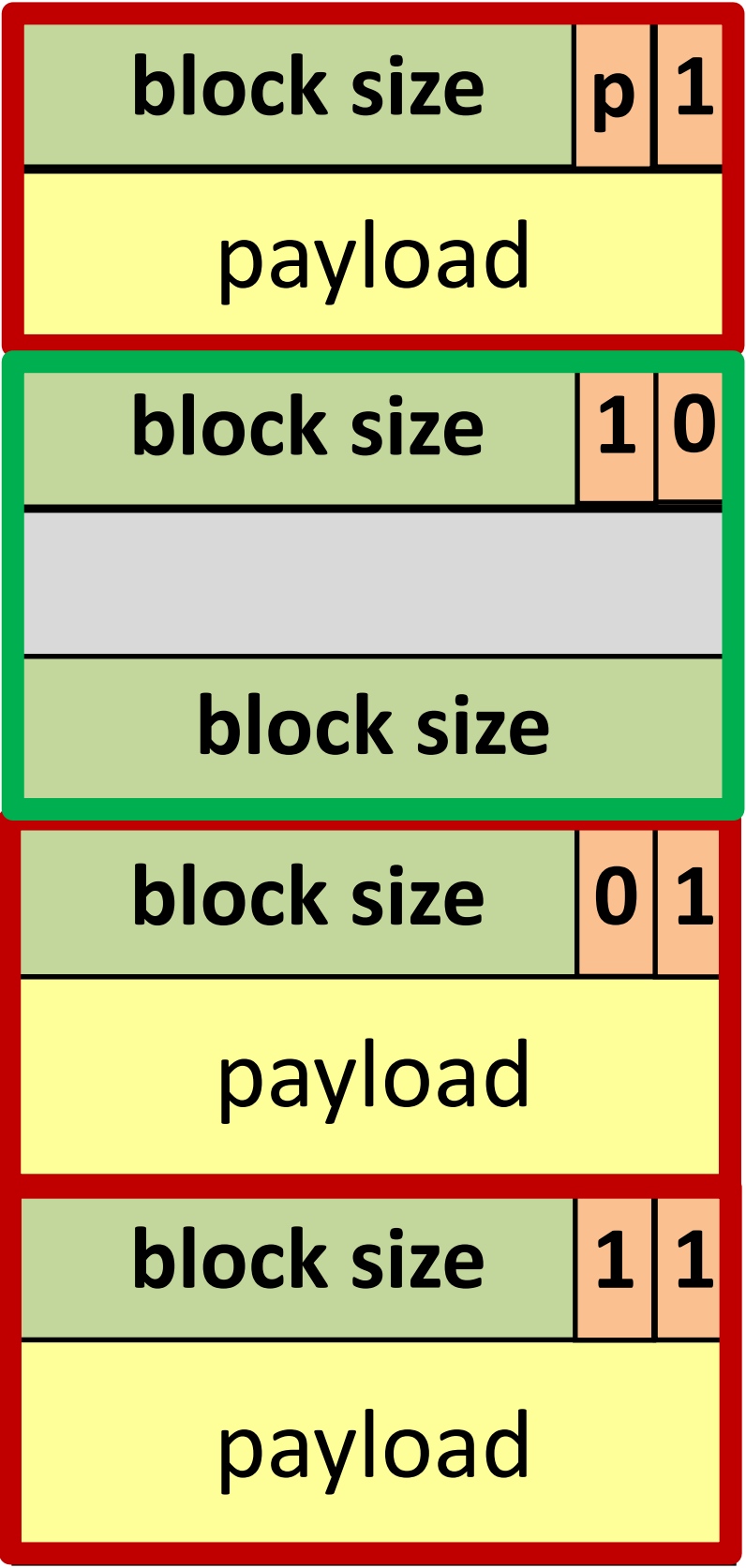
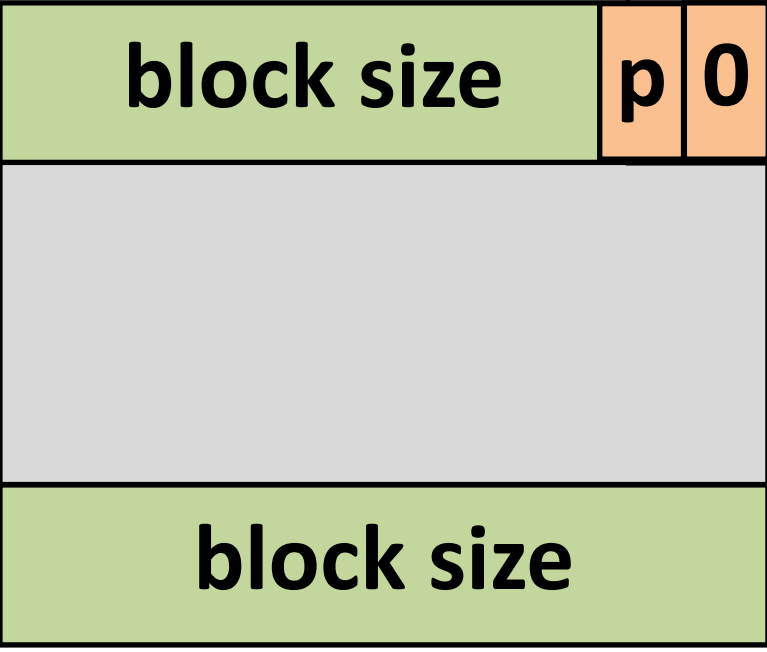


Improved block format for implicit free lists

Allocated block:



Free block:



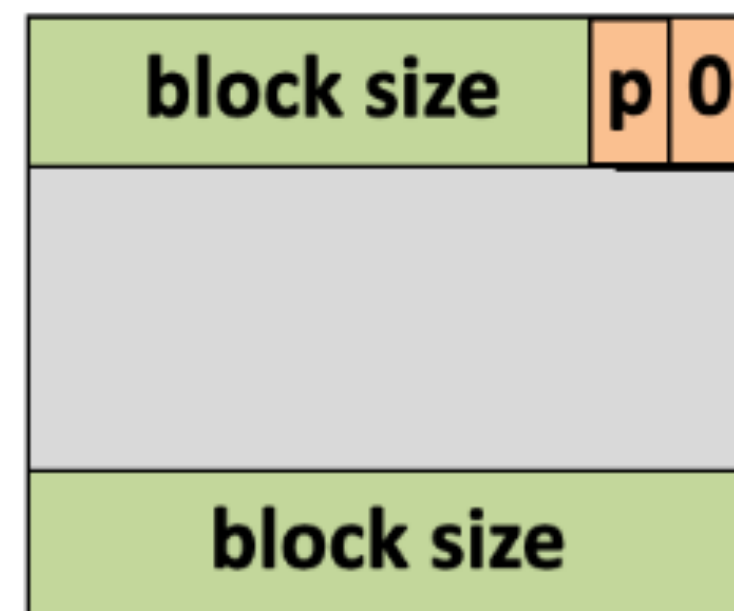
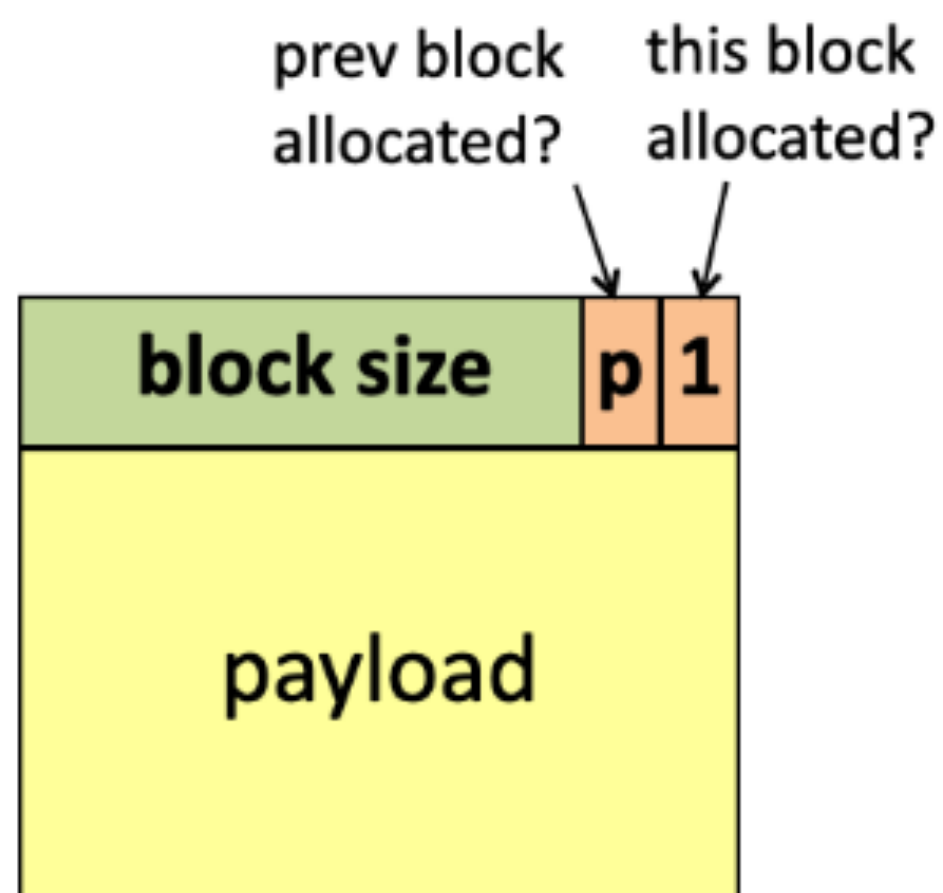
Update headers of 2 blocks on each malloc/free.

Minimum block size for implicit free list?

What is the minimum block size for an implicit free block (in bytes)?

Allocated block:

Free block:



8

16

24

32

None of the above

Summary: implicit free lists

Implementation: simple

$O(\dots)$ for allocate and free?

Allocate: $O(\text{blocks in heap})$

Free: $O(1)$

Memory utilization: depends on placement policy

Not as widely used in practice

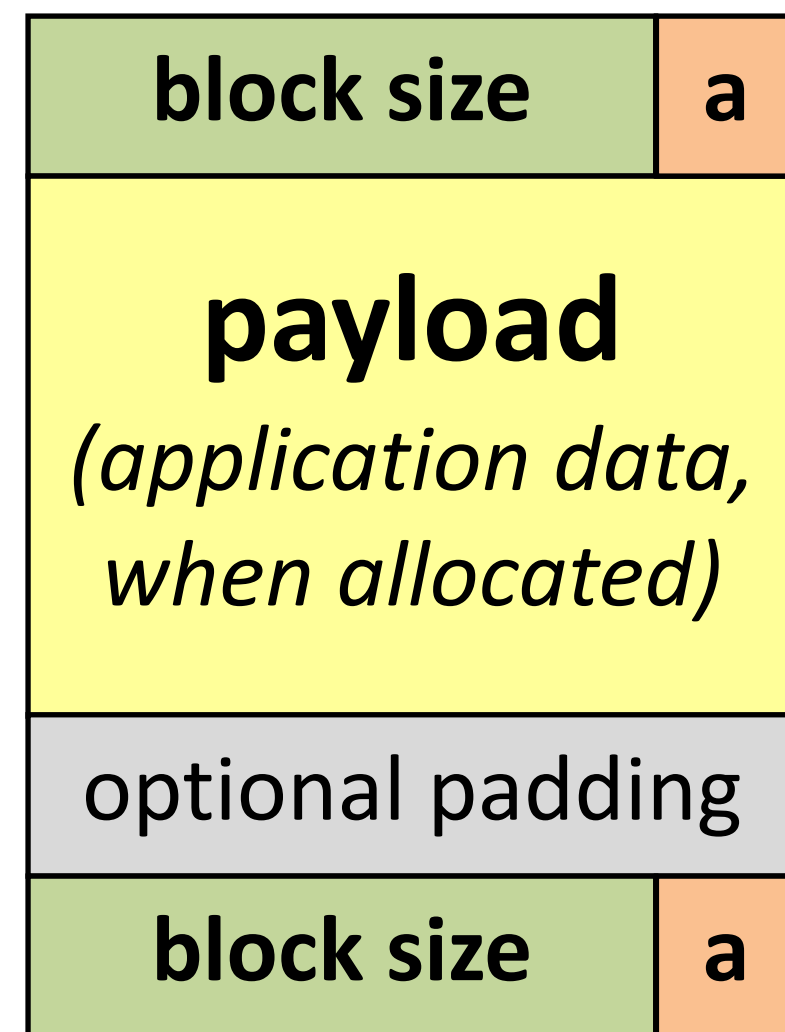
some special purpose applications

Splitting, boundary tags, coalescing are **general** to *all* allocators.

Explicit free list: block format

Explicit list of *free* blocks rather than implicit list of *all* blocks.

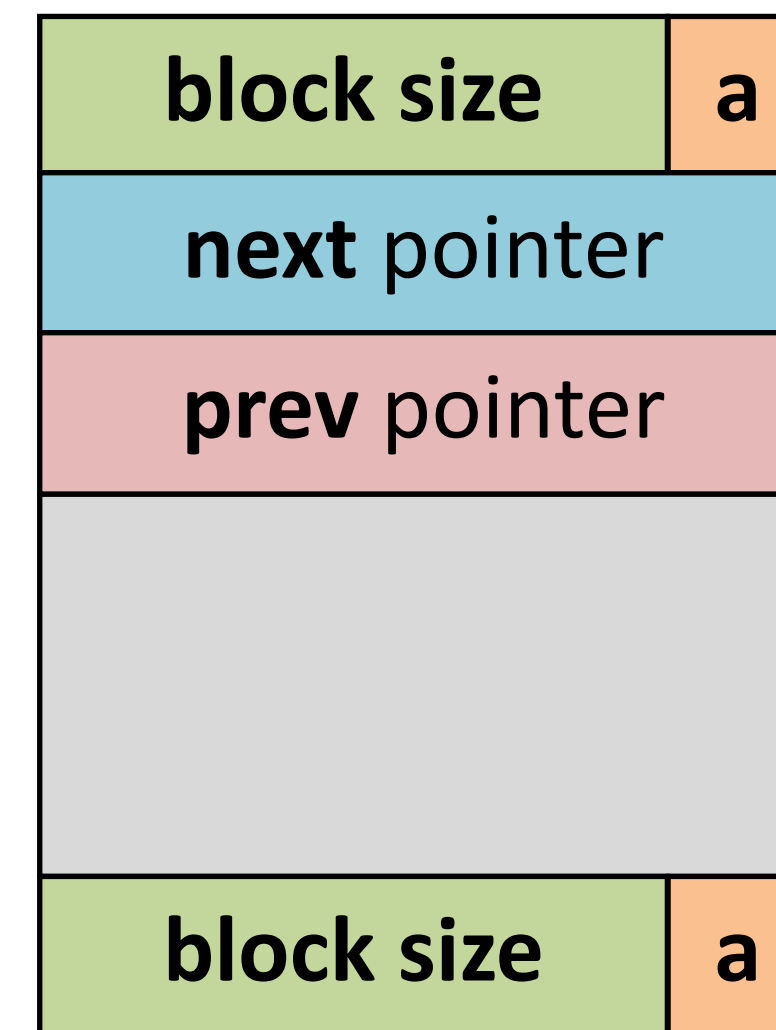
Allocated block:



Possible to omit footer

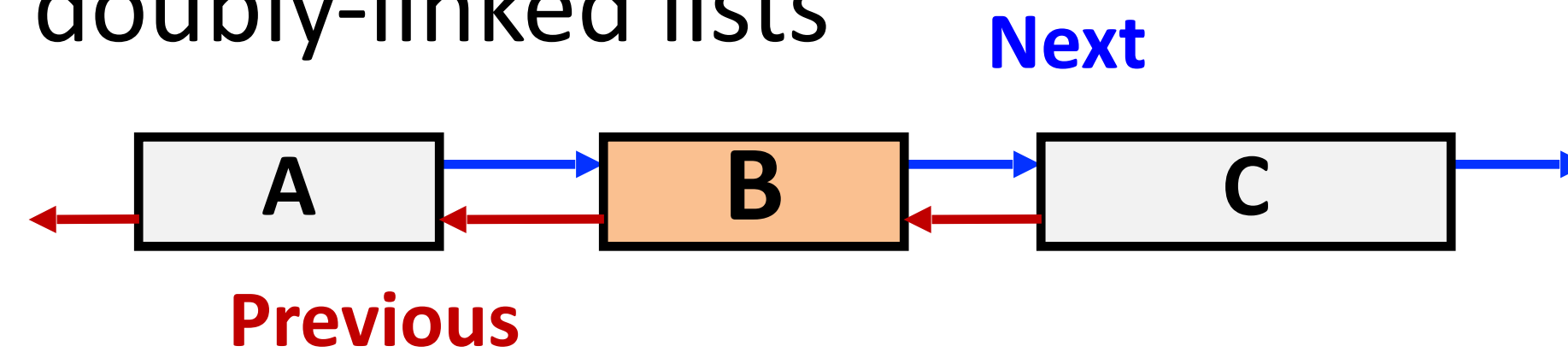
(same as implicit free list)

Free block:

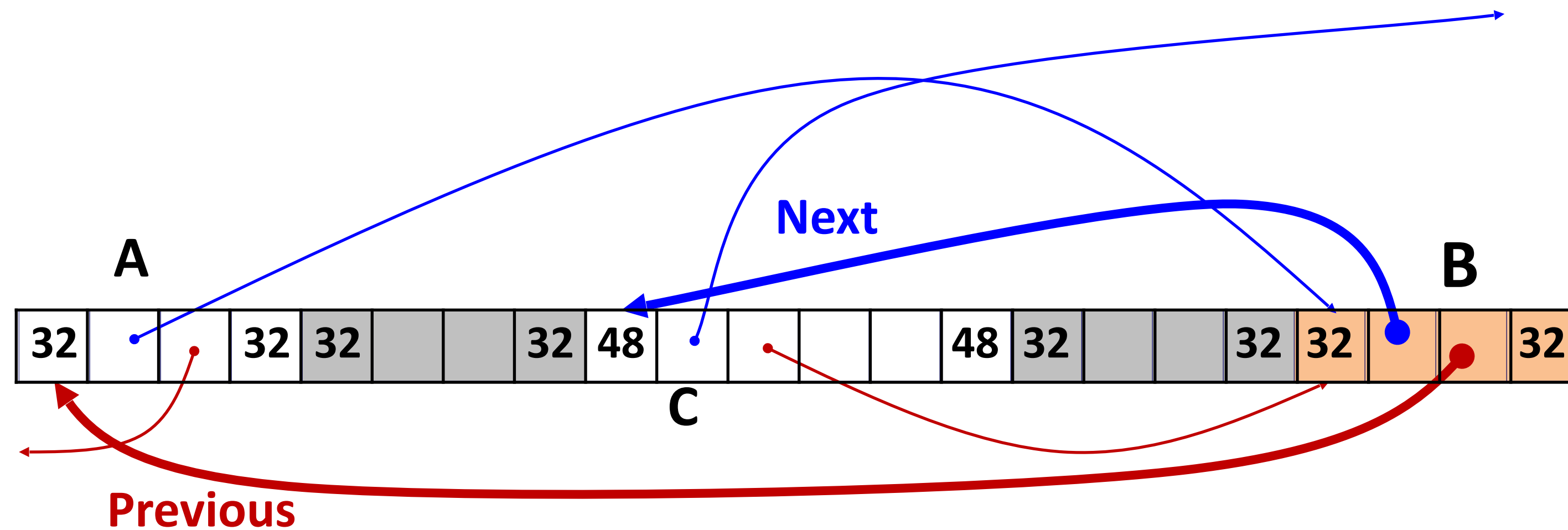


Explicit free list: list vs. memory order

Abstractly: doubly-linked lists



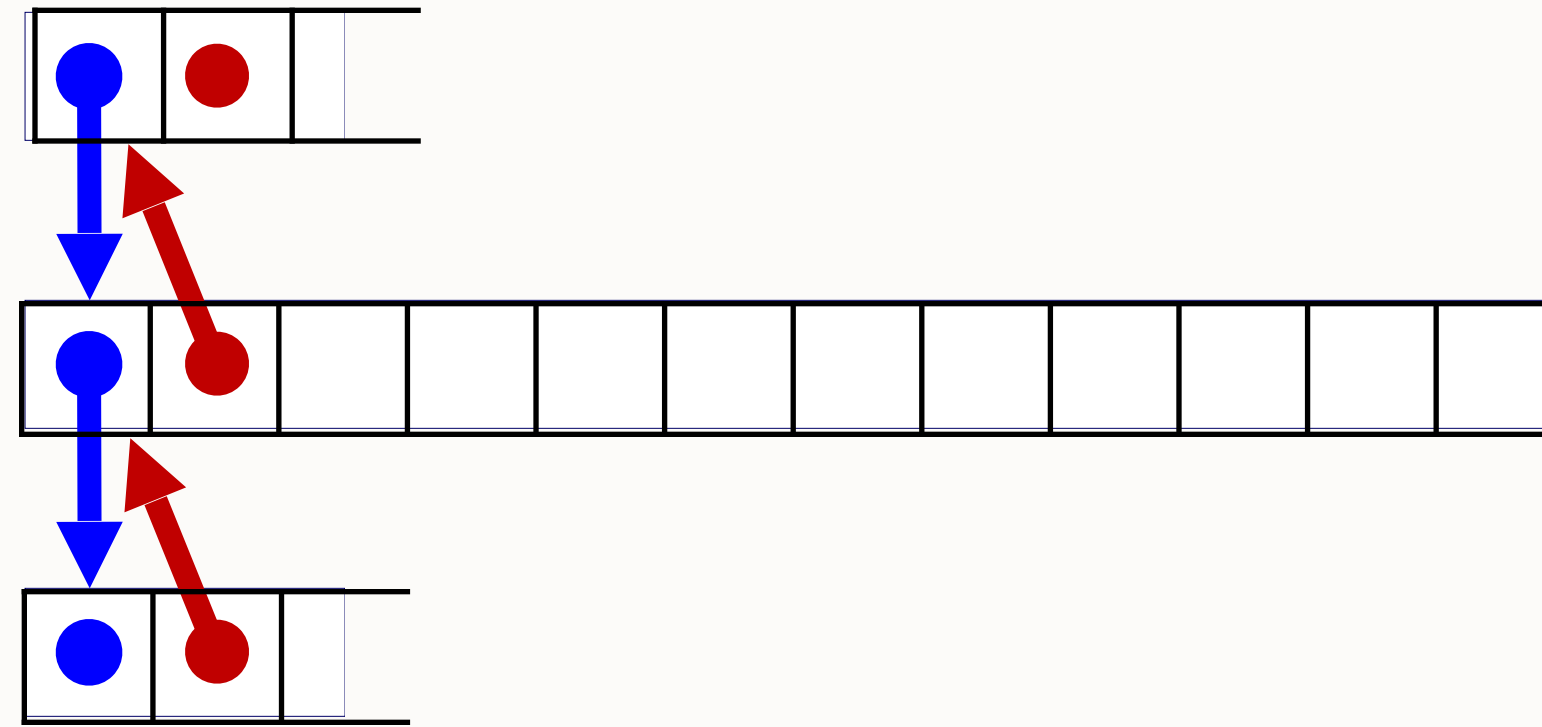
Concretely: free list blocks in any memory order



List Order $\neq$ Memory Order

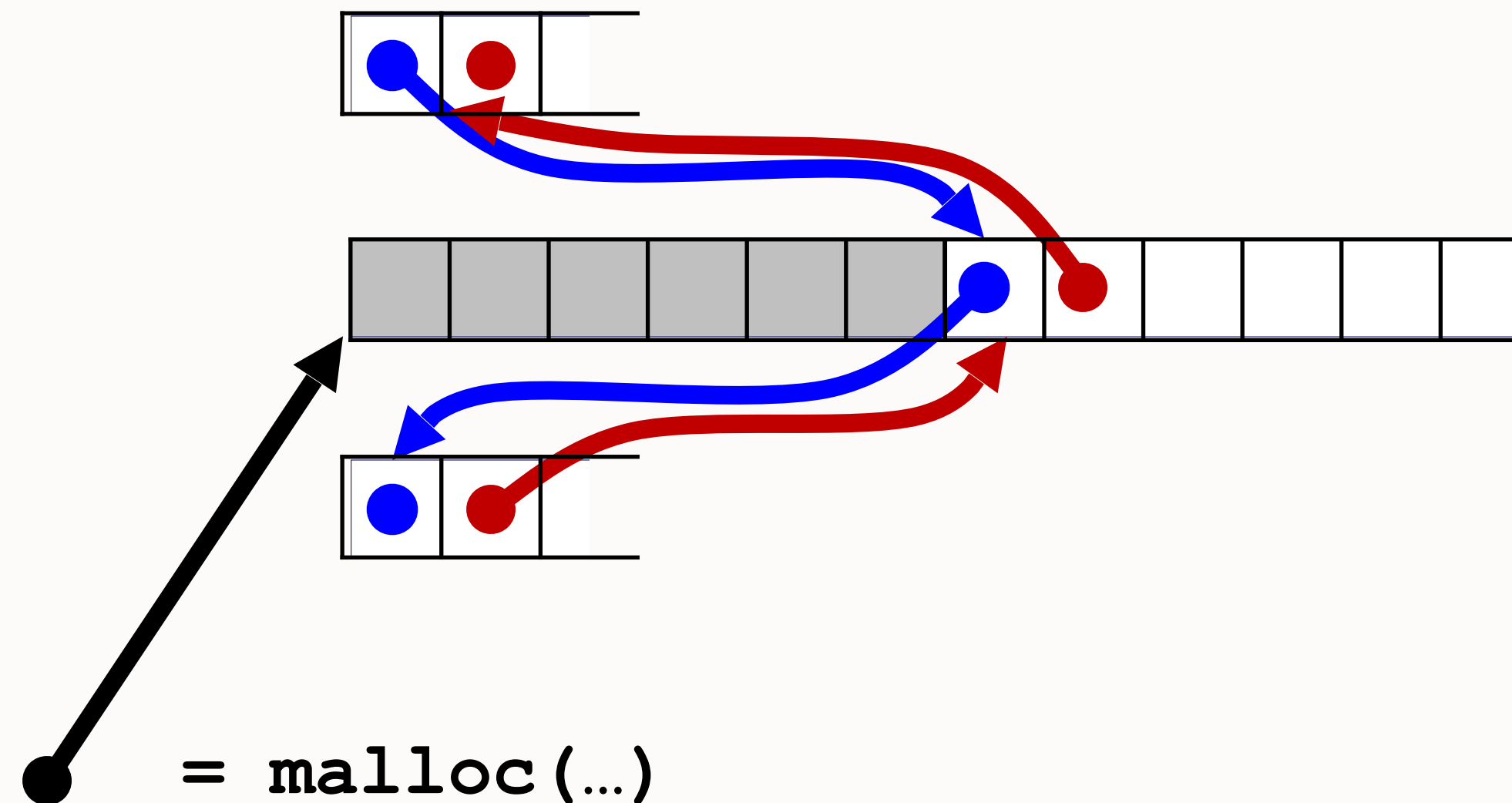
Explicit free list: allocating a free block

Before



After

(with splitting)



Explicit free list: **freeing a block**

Insertion policy: Where in the free list do you add a freed block?

LIFO (last-in-first-out) policy

Pro: simple and constant time

Con: studies suggest fragmentation is worse than address ordered

Address-ordered policy

Con: linear-time search to insert freed blocks

Pro: studies suggest fragmentation is lower than LIFO

LIFO Example: 4 cases of freed block neighbor status.

Freeing with LIFO policy: between allocated blocks



blue: next
red: prev
open: NULL

Imagine we used this struct for each block header:

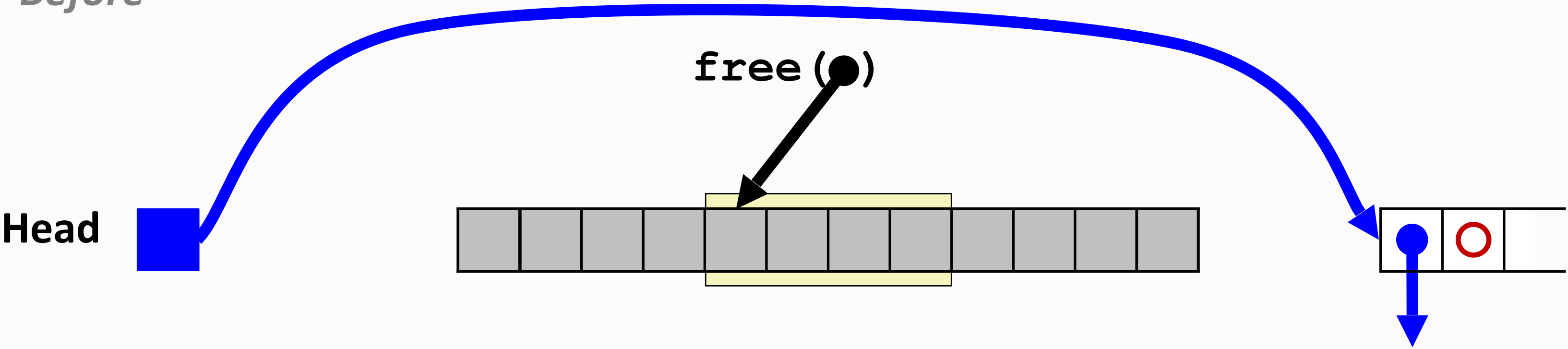
```
typedef struct Block {
    struct Block* next;
    struct Block* prev;
} Block;
```

Write code to insert the freed block at head of free list.

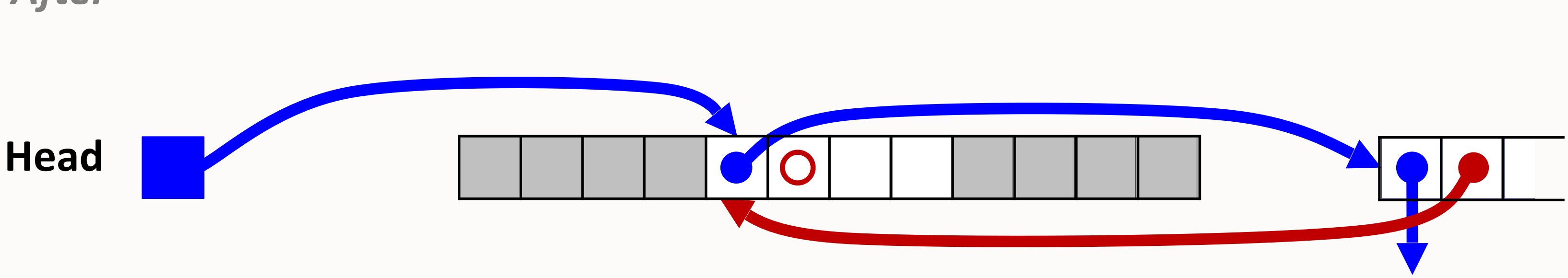
```
Block *head;

void insert1(Block* freed) {
    // ... fill in
}
```

Before



After

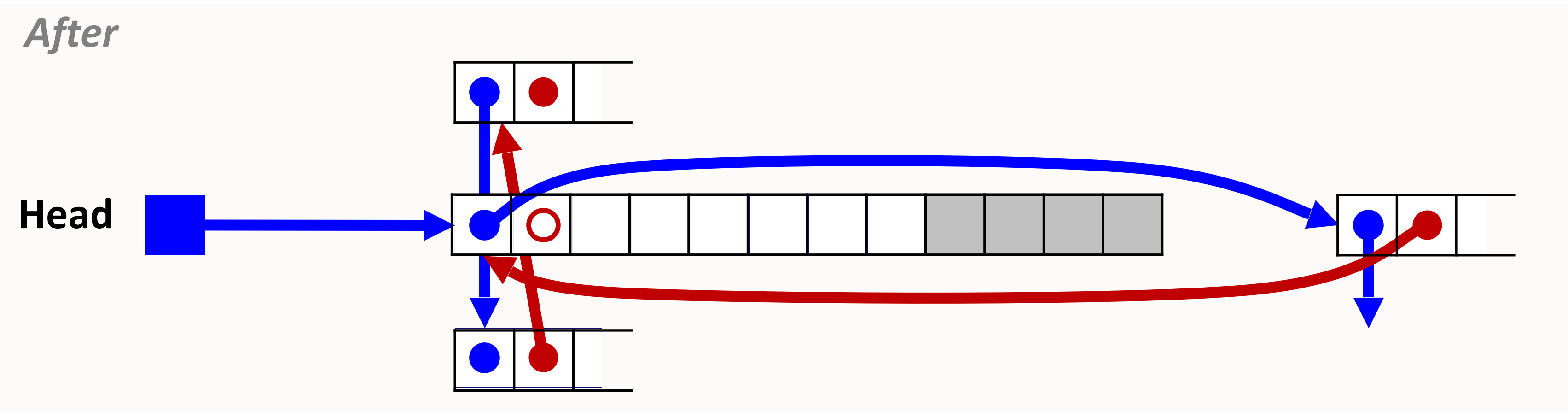
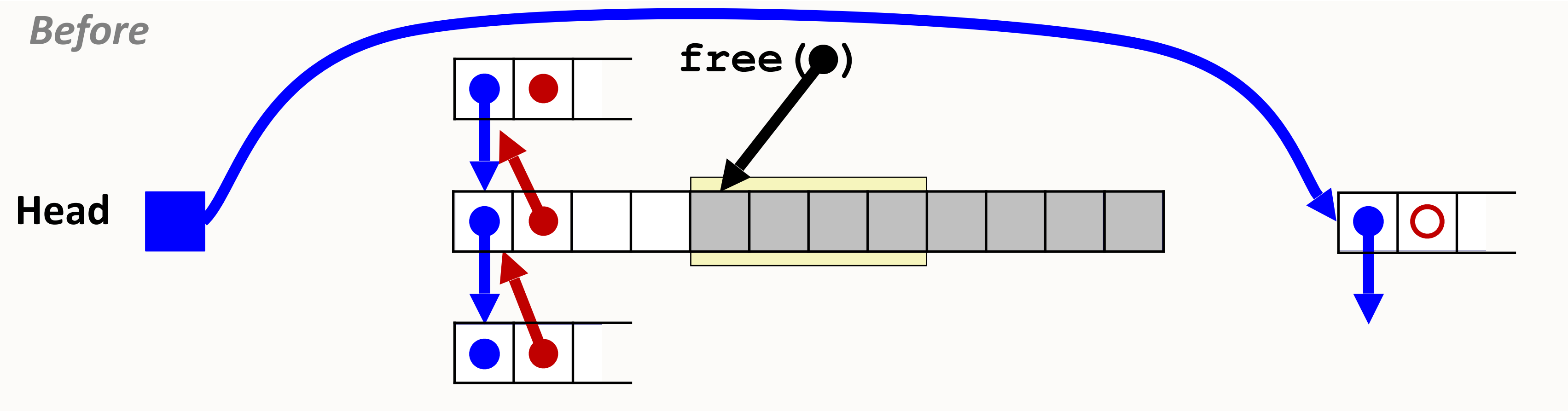


Freeing with LIFO policy: between free and allocated



blue: next
red: prev
open: NULL

Splice out predecessor block, coalesce both memory blocks, and insert the new block at the head of the free list.



```
typedef struct Block {
    struct Block* next;
    struct Block* prev;
} Block;

Block *head;

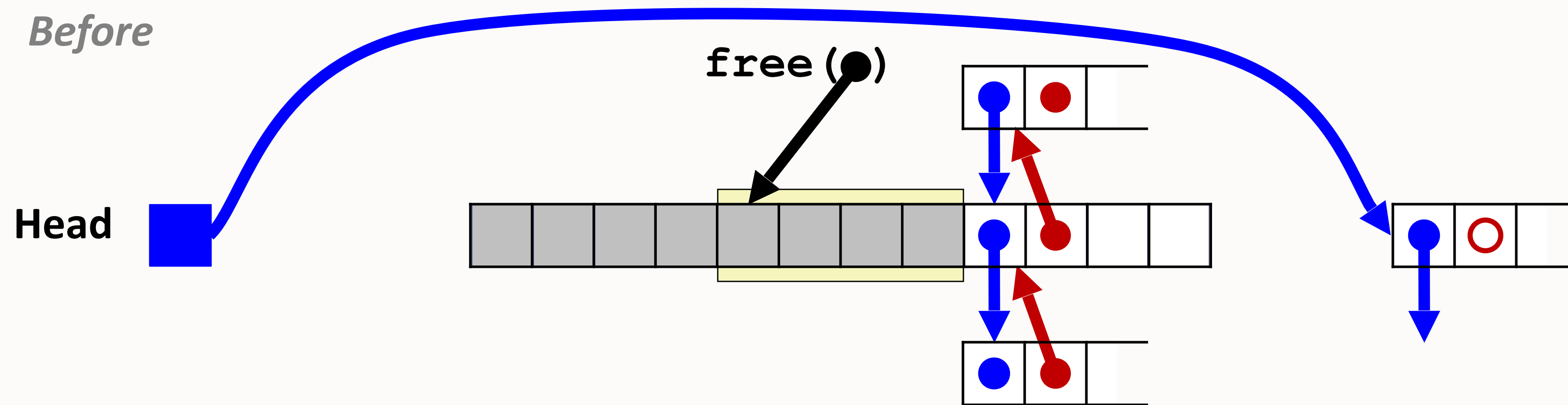
void insert2(Block* freed) {
    // ... fill in
}
```

Freeing with LIFO policy: between allocated and free

ex

Splice out successor block, coalesce both memory blocks and insert the new block at the head of the free list.

blue: next
red: prev
open: NULL

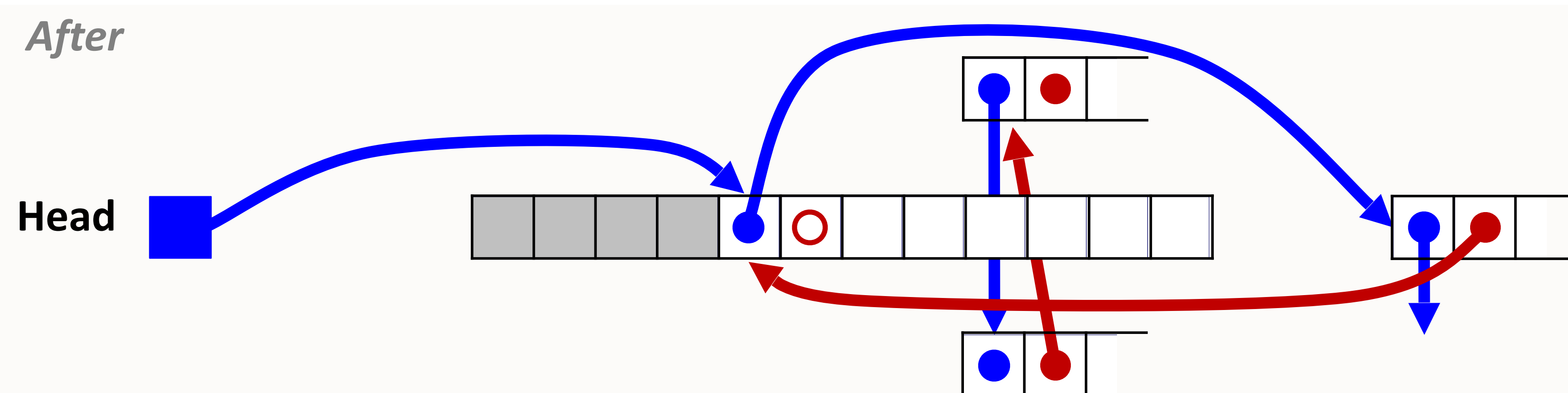


```
typedef struct Block {  
    struct Block* next;  
    struct Block* prev;  
} Block;
```

```
Block *head;
```

```
void insert3(Block* freed) {  
    // ... fill in
```

```
}
```

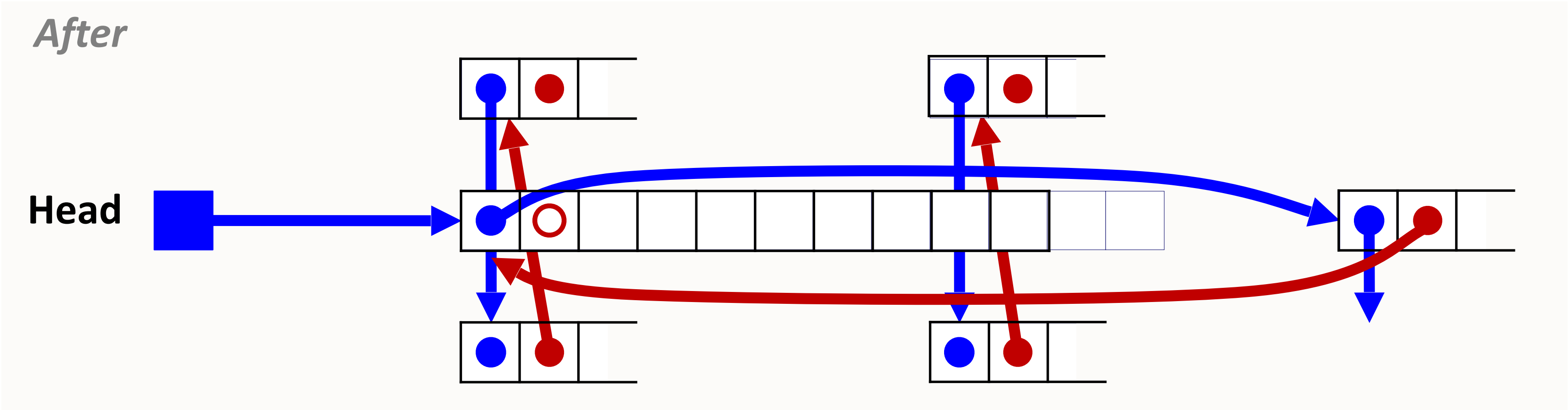
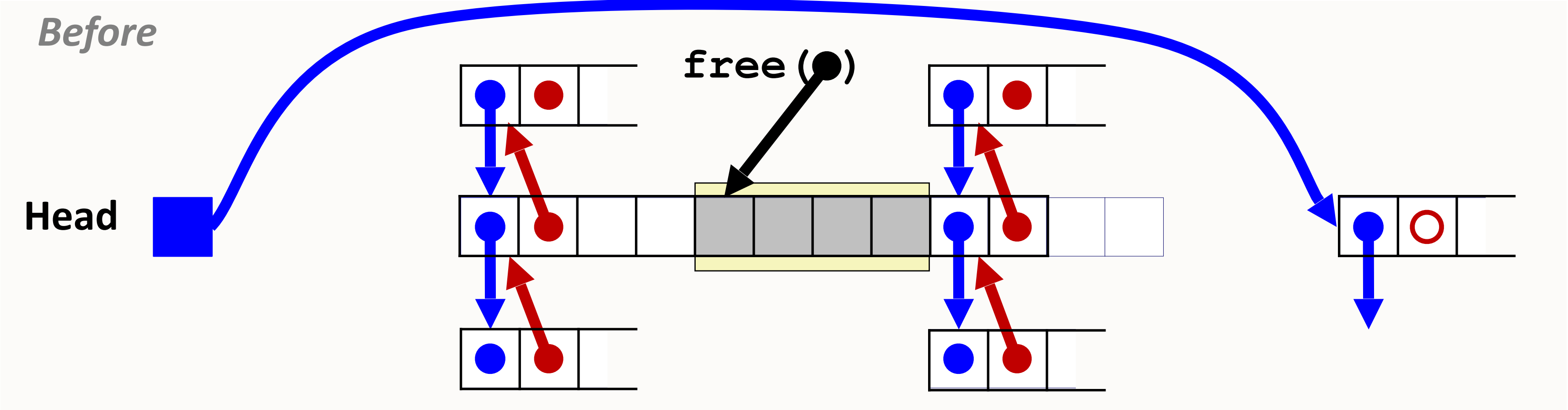


Freeing with LIFO policy: between free blocks



blue: next
red: prev
open: NULL

Splice out predecessor and successor blocks, coalesce all 3 memory blocks and insert the new block at the head of the list.



```
typedef struct Block {
    struct Block* next;
    struct Block* prev;
} Block;

Block *head;

void insert4(Block* freed) {
    // ... fill in
}
```

Summary: Explicit Free Lists

Implementation: fairly simple

Allocate: $O(\textit{free} \text{ blocks})$ vs. $O(\textit{all} \text{ blocks})$

Free: $O(1)$ vs. $O(1)$

Memory utilization:

depends on placement policy

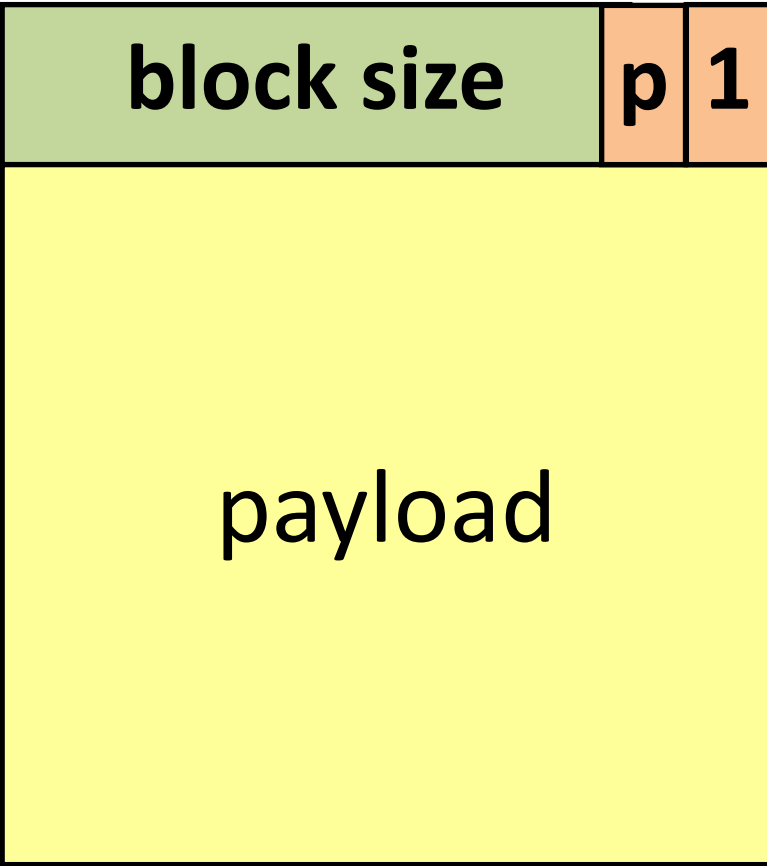
larger minimum block size (next/prev) vs. implicit list

Used widely in practice, often with more optimizations.

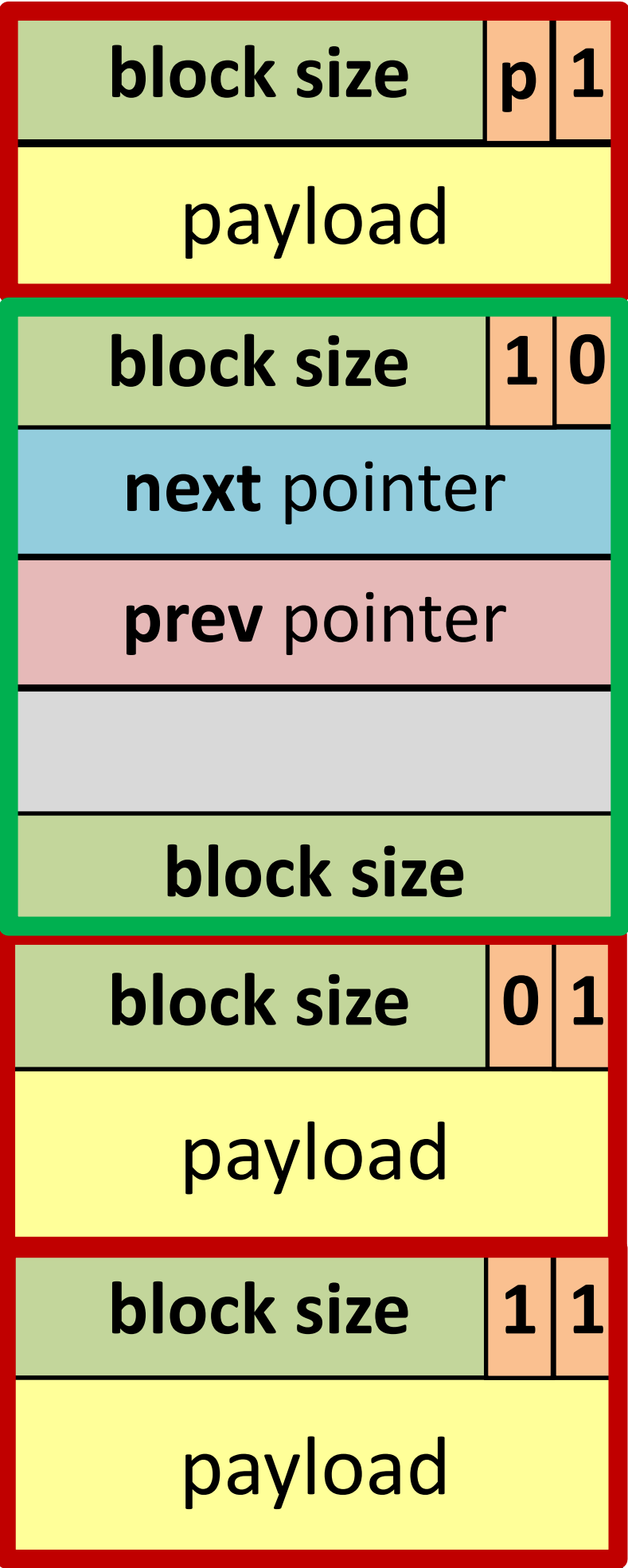
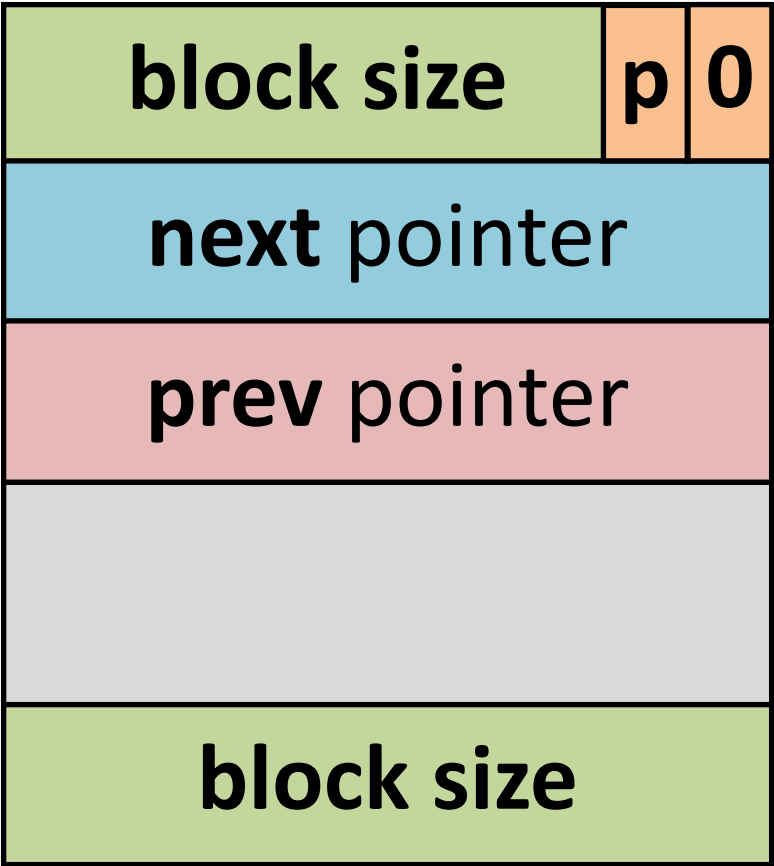
Splitting, boundary tags, coalescing are general to *all* allocators.

Improved block format for explicit free lists

Allocated block:



Free block:



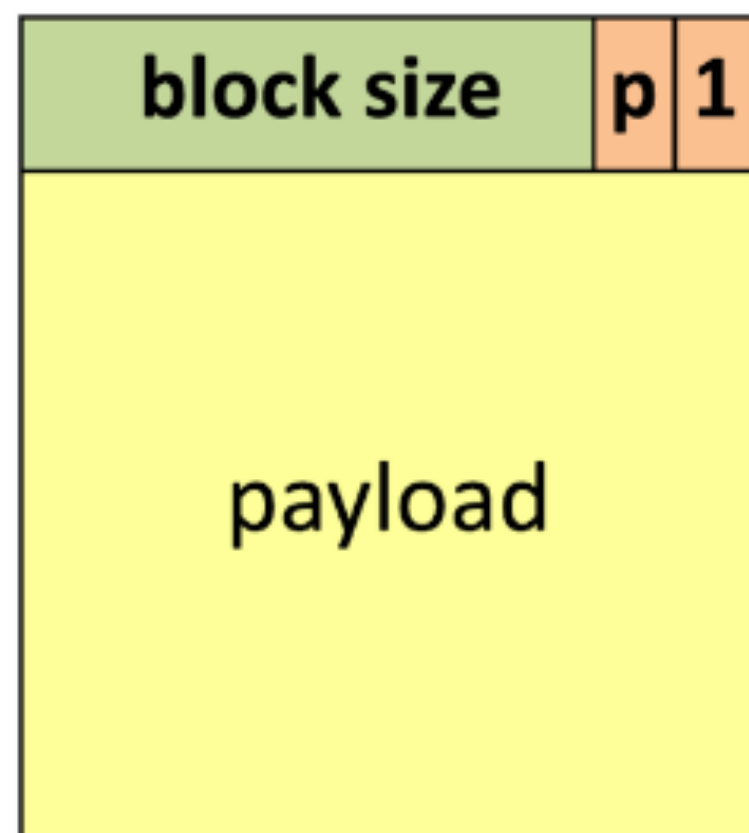
Update headers of 2 blocks on each malloc/free.

Minimum block size for explicit free list?

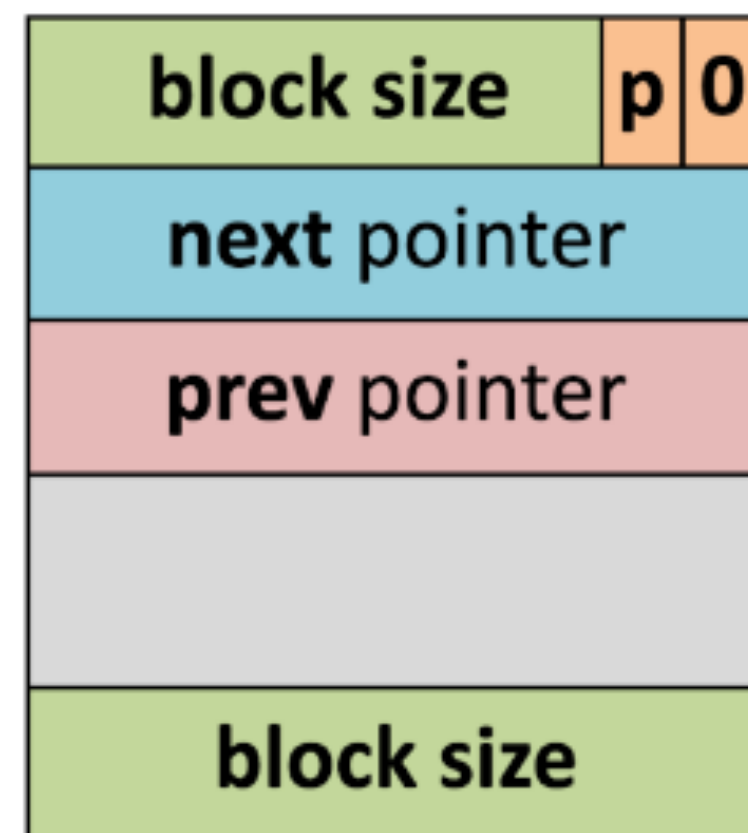
What is the minimum block size for an explicit free block (in bytes)?

0

Allocated block:



Free block:



8

16

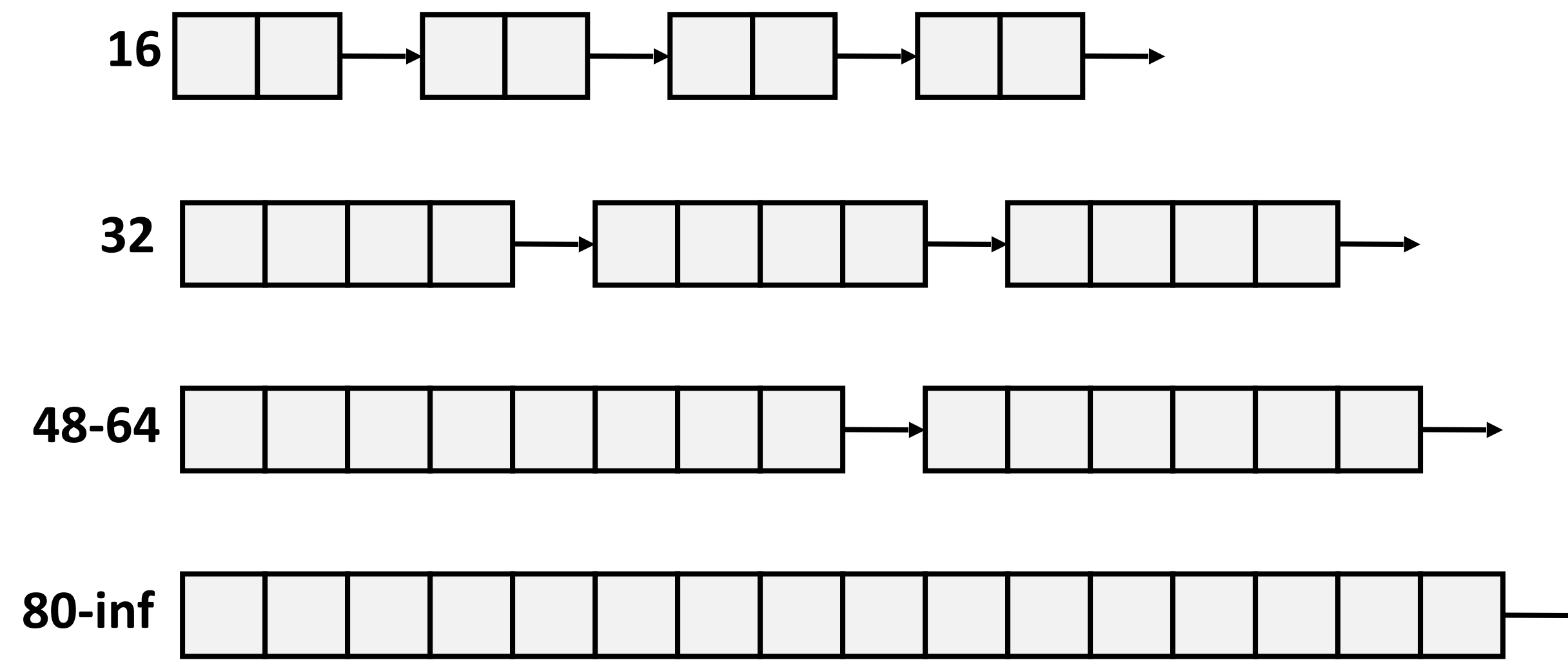
24

32

None of the above

Seglist allocators

Each *size bracket* has its own free list



Faster best-fit allocation...

Summary: allocator policies

All policies offer **trade-offs** in fragmentation and throughput.

Placement policy:

First-fit, next-fit, best-fit, etc.

Seglists approximate best-fit in low time

Splitting policy:

Always? Sometimes? Size bound?

Coalescing policy:

Immediate vs. deferred