CSE251: System Programming

19. Dynamic Memory Allocation (3)

Seongil Wi

Department of Computer Science and Engineering

Overview of the Today's Topics

Q. How to allocate in free block?

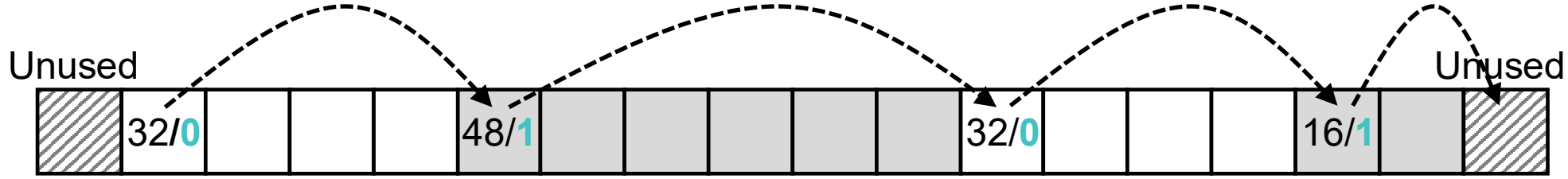
→ ***Splitting***

Q. How to free?

→ ***Coalescing***

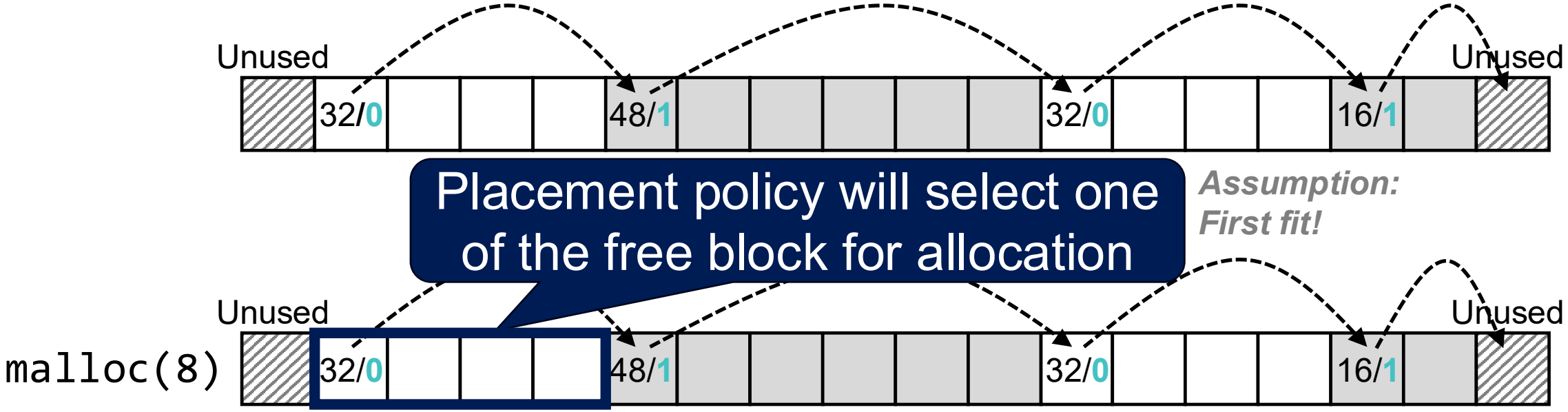
Let's study each one,
including its motivation and how it works!

Let's Think about Malloc



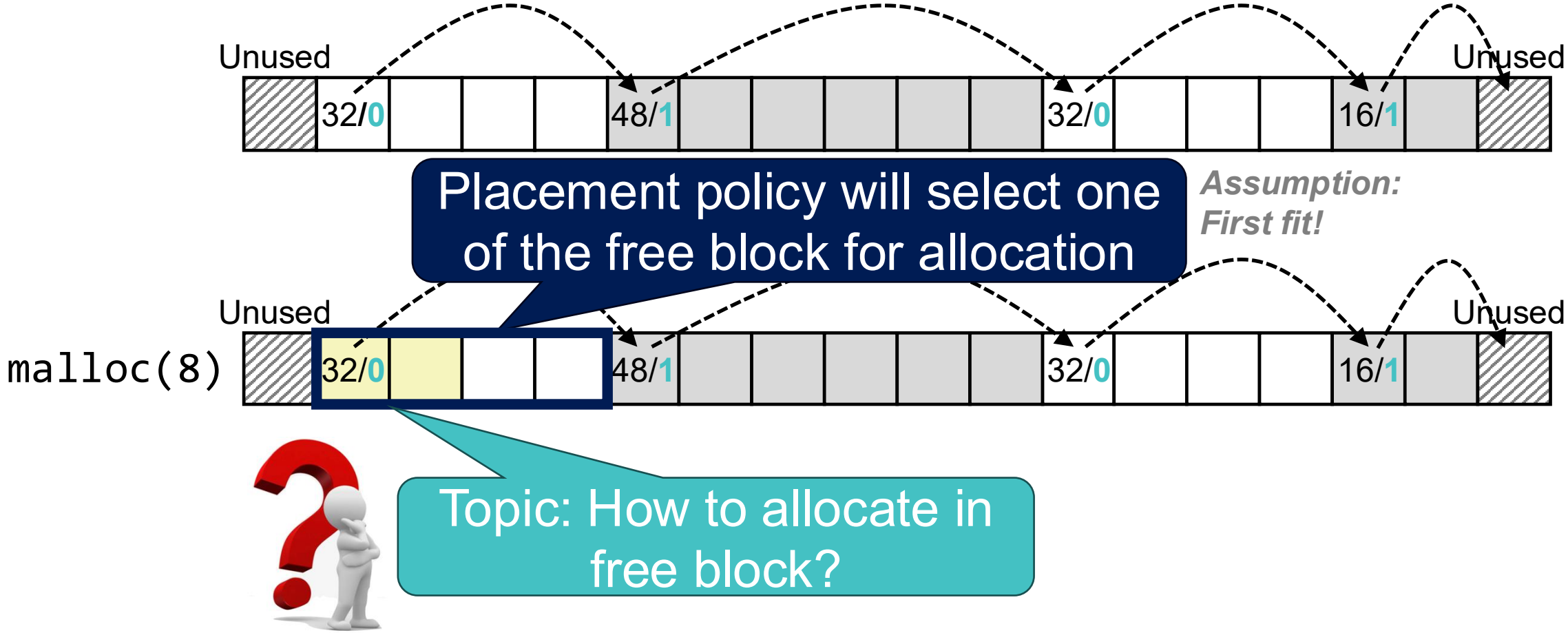

```
malloc(8)
```

Let's Think about Malloc



Let's Think about Malloc

1 square = 1 word = 8 bytes
: Free word
: Allocated word

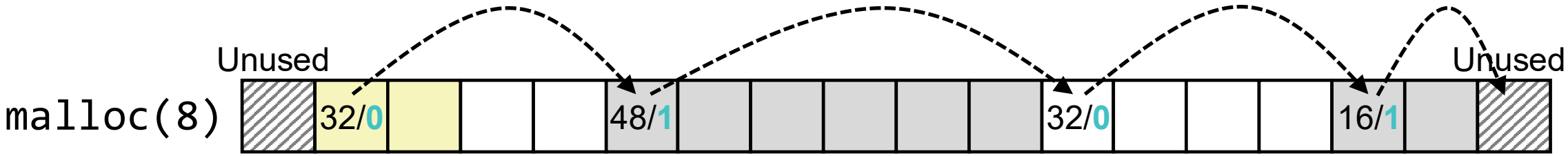


Splitting

Allocating in a Free Block: Splitting

1 square = 1 word = 8 bytes
: Free word
: Allocated word

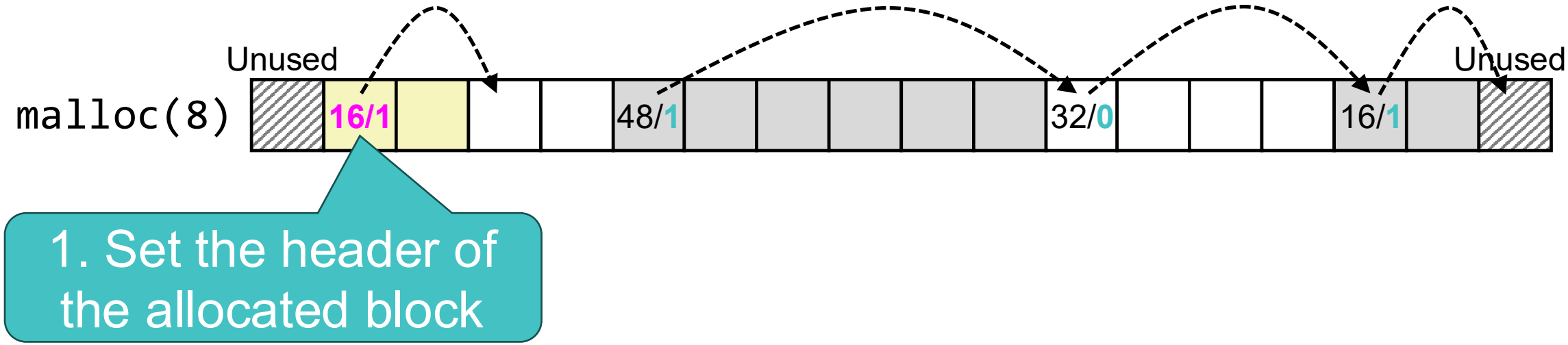
- Since allocated space might be *smaller than free space*, we might want to **split the block**



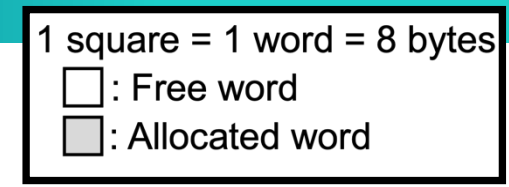
Allocating in a Free Block: Splitting

1 square = 1 word = 8 bytes
: Free word
: Allocated word

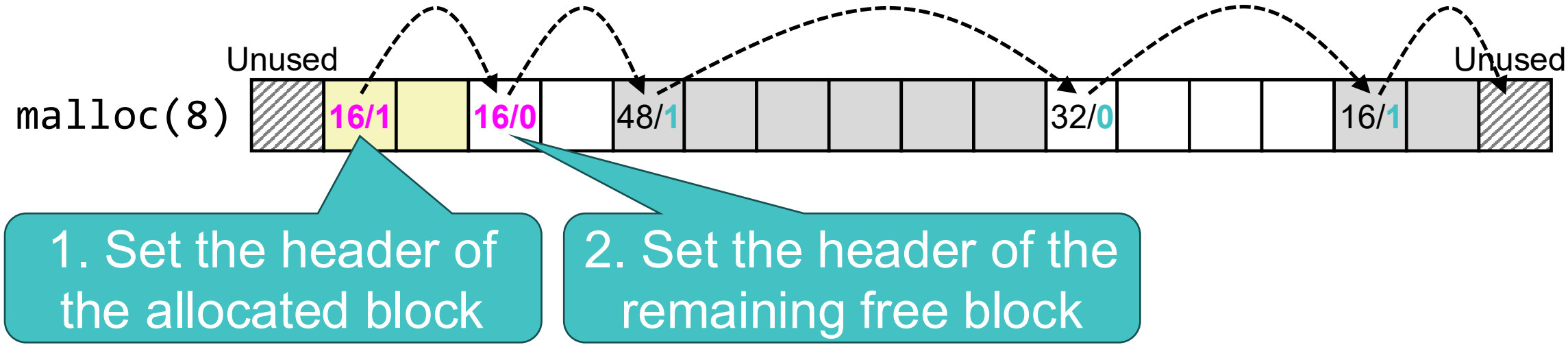
- Since allocated space might be *smaller than free space*, we might want to **split the block**



Allocating in a Free Block: Splitting



- Since allocated space might be *smaller than free space*, we might want to **split the block**

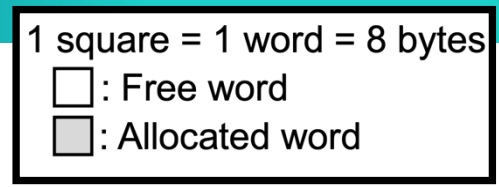


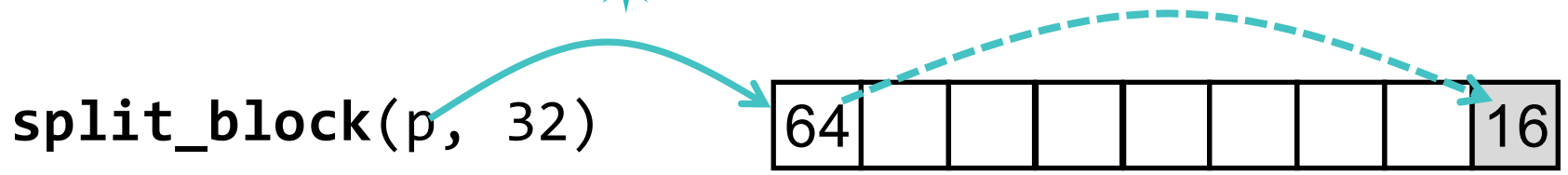
Allocating in a Free Block: Splitting

1 square = 1 word = 8 bytes
: Free word
: Allocated word

```
void split_block(ptr *block, int payload_size){
  int block_size = get_size(block);

  if ((block_size - payload_size) >= min_block_size) {
    write_header(block, payload_size, true);
    ptr *block_next = find_next(block);
    write_header(block_next, block_size - payload_size, false);
  }
}
```

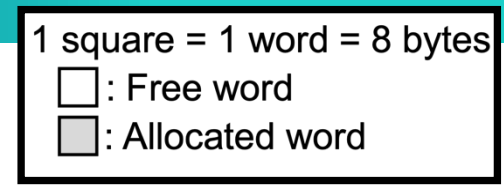
Allocating in a Free Block: Splitting



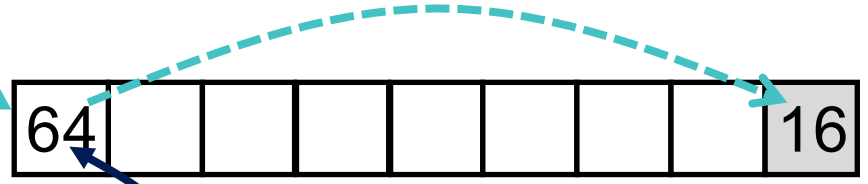



```
void split_block(ptr *block, int payload_size){
  int block_size = get_size(block);

  if ((block_size - payload_size) >= min_block_size) {
    write_header(block, payload_size, true);
    ptr *block_next = find_next(block);
    write_header(block_next, block_size - payload_size, false);
  }
}
```

Allocating in a Free Block: Splitting



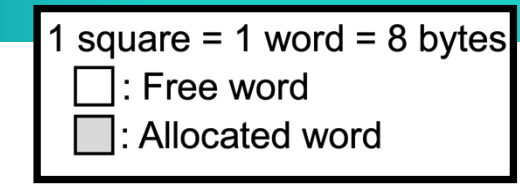
split_block(p, 32)

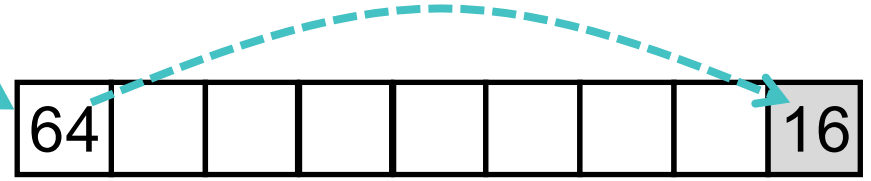



```
void split_block(ptr *block, int payload_size){
  int block_size = get_size(block);

  if ((block_size - payload_size) >= min_block_size) {
    write_header(block, payload_size, true);
    ptr *block_next = find_next(block);
    write_header(block_next, block_size - payload_size, false);
  }
}
```

64

Allocating in a Free Block: Splitting



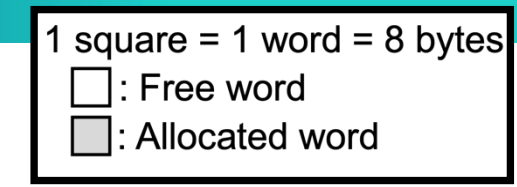


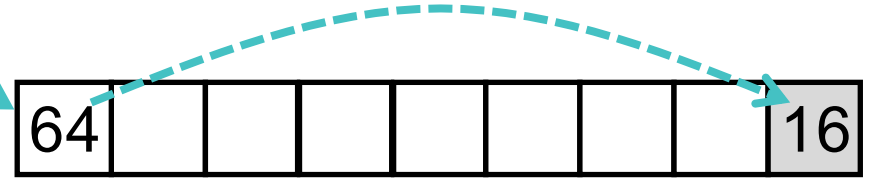

```
void split_block(ptr *block, int payload_size){
  int block_size = get_size(block);

  if ((block_size - payload_size) >= min_block_size) {
    write_header(block, payload_size, true);
    ptr *block_next = find_next(block);
    write_header(block_next, block_size - payload_size, false);
  }
}
```

64

32

Allocating in a Free Block: Splitting





Remaining free block's size (32)

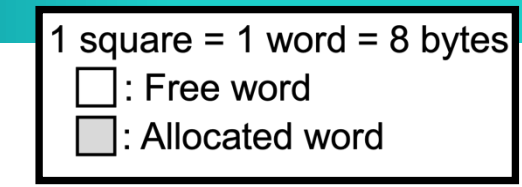
64

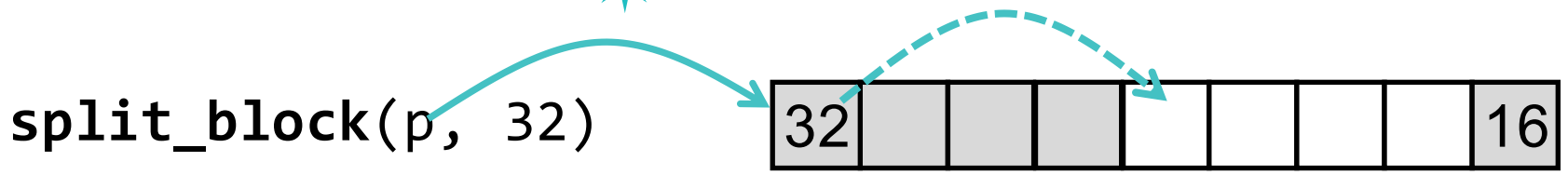
32

```
void split_block(ptr *block, int payload_size){
  int block_size = get_size(block);

  if ((block_size - payload_size) >= min_block_size) {
    write_header(block, payload_size, true);
    ptr *block_next = find_next(block);
    write_header(block_next, block_size - payload_size, false);
  }
}
```

`lock(p, 32)`

Allocating in a Free Block: Splitting

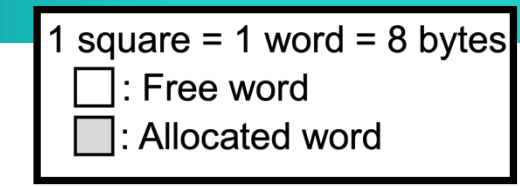




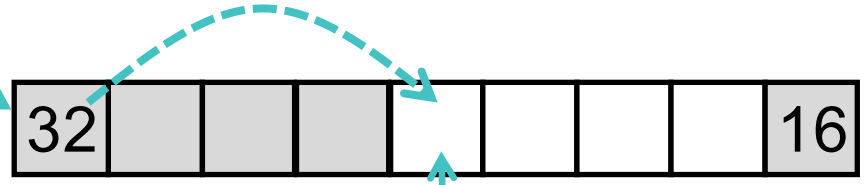

```
void split_block(ptr *block, int payload_size){
  int block_size = get_size(block);

  if ((block_size - payload_size) >= min_block_size) {
    write_header(block, payload_size, true);
    ptr *block_next = find_next(block);
    write_header(block_next, block_size - payload_size, false);
  }
}
```

Allocated

Allocating in a Free Block: Splitting

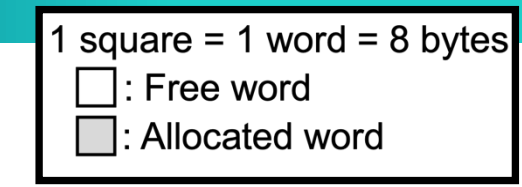


split_block(p, 32)

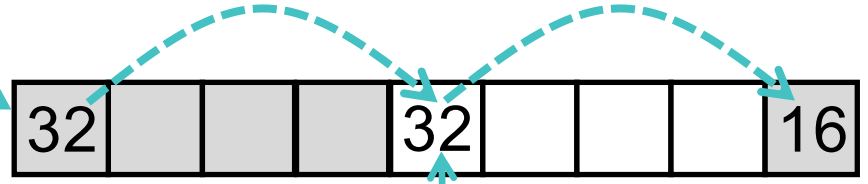



```
void split_block(ptr *block, int payload_size){
  int block_size = get_size(block);

  if ((block_size - payload_size) >= min_block_size) {
    write_header(block, payload_size, true);
    ptr *block_next = find_next(block);
    write_header(block_next, block_size - payload_size, false);
  }
}
```

Allocating in a Free Block: Splitting



split_block(p, 32)

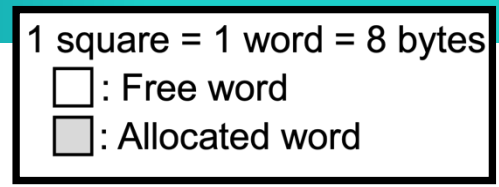


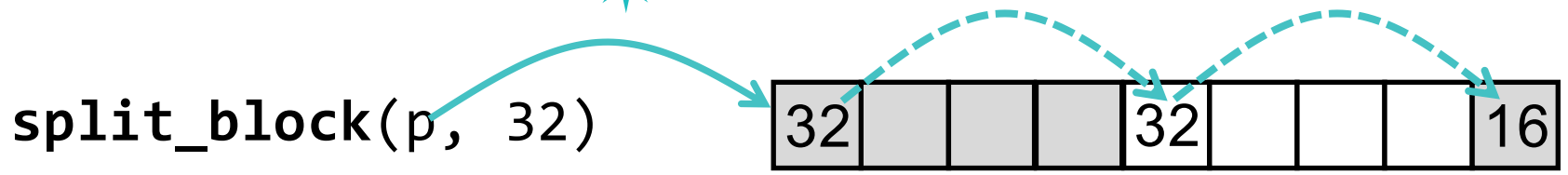

```
void split_block(ptr *block, int payload_size){
  int block_size = get_size(block);

  if ((block_size - payload_size) >= min_block_size) {
    write_header(block, payload_size, true);
    ptr *block_next = find_next(block);
    write_header(block_next, block_size - payload_size, false);
  }
}
```

Remaining free block's size (32)

Free

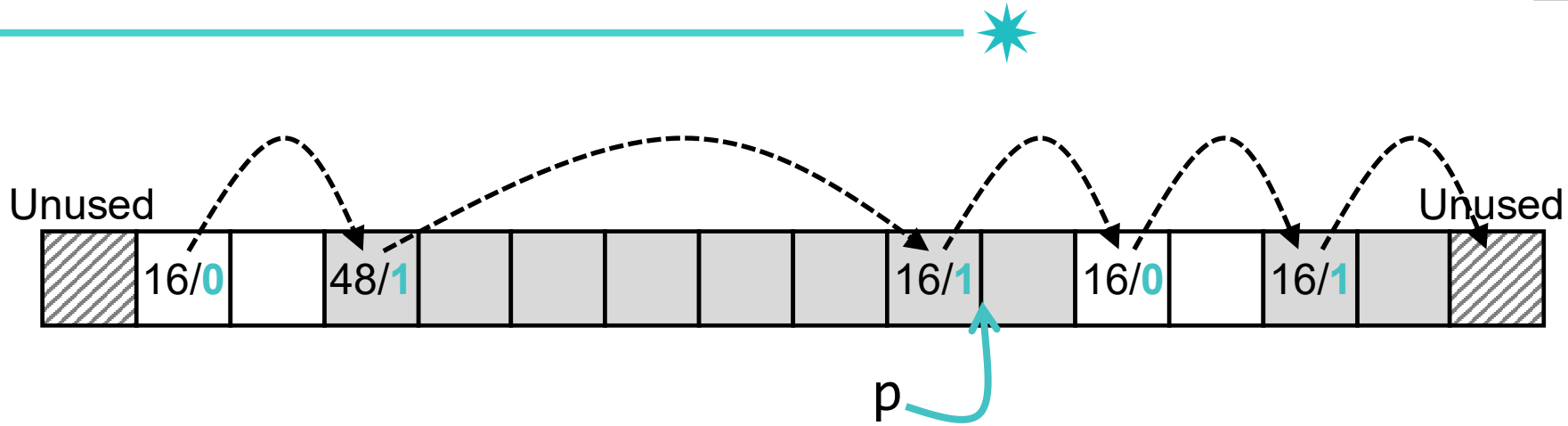
Allocating in a Free Block: Splitting





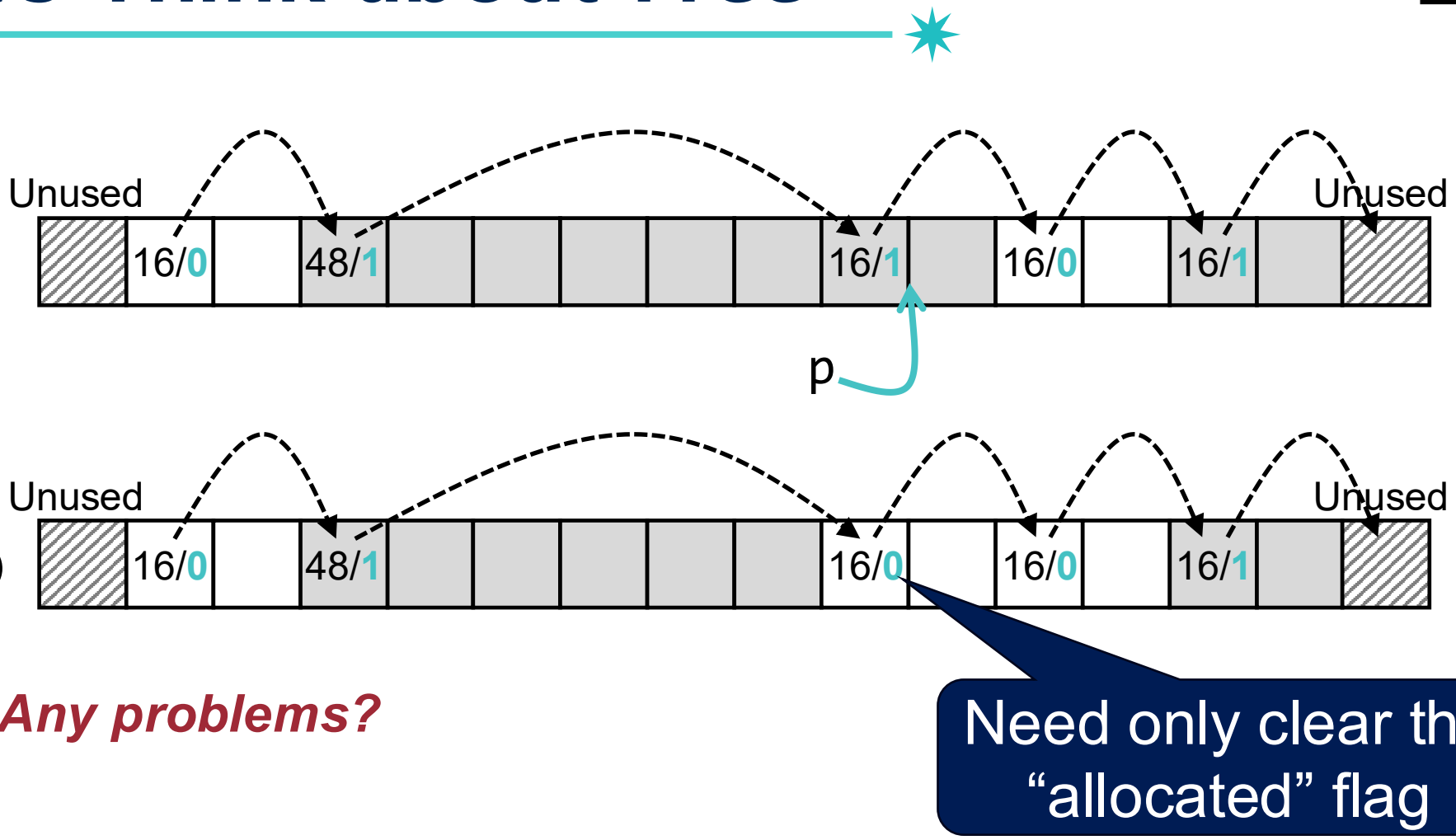

```
void split_block(ptr *block, int payload_size){
  int block_size = get_size(block);

  if ((block_size - payload_size) >= min_block_size) {
    write_header(block, payload_size, true);
    ptr *block_next = find_next(block);
    write_header(block_next, block_size - payload_size, false);
  }
}
```

Let's Think about Free




```
free(p)
```

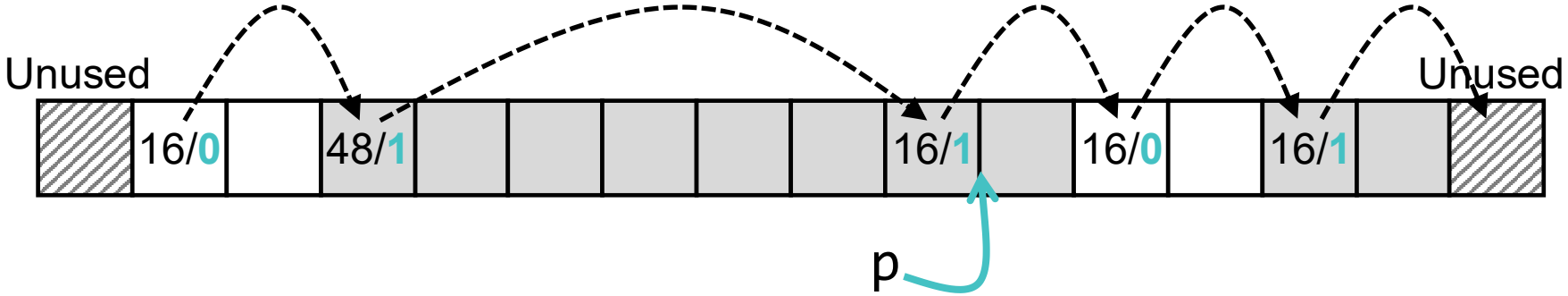
Let's Think about Free

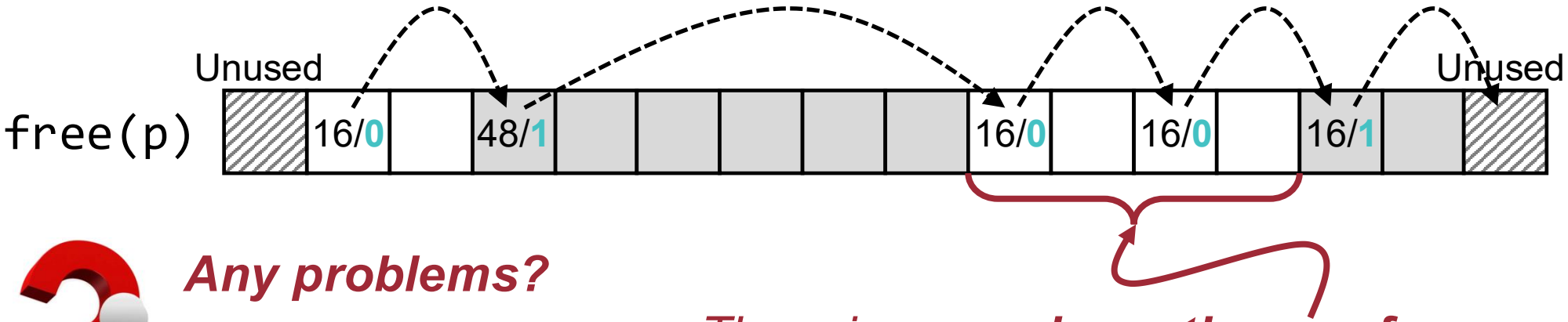


Any problems?

Let's Think about Free

1 square = 1 word = 8 bytes
: Free word
: Allocated word





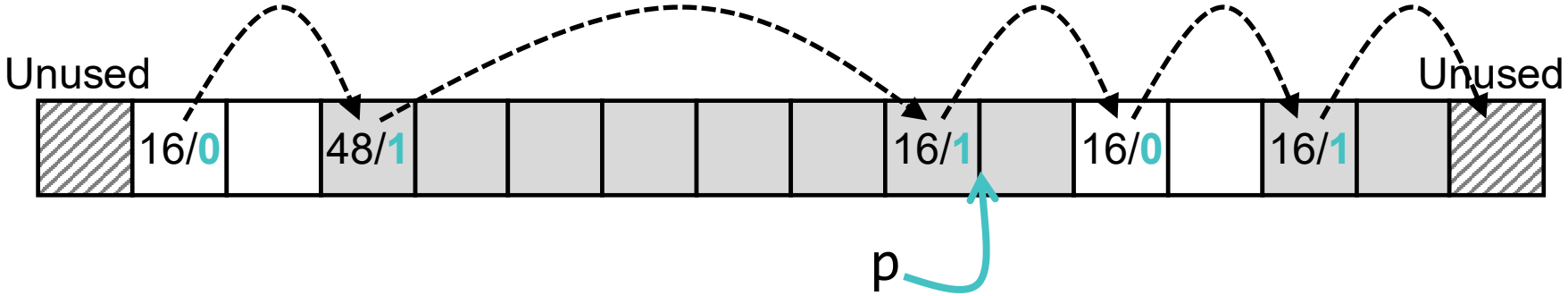
Any problems?

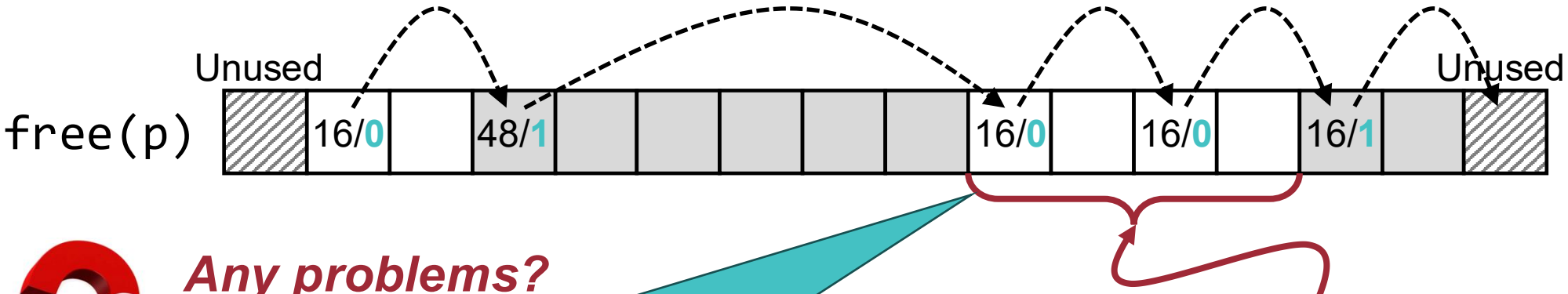
`malloc(32)` → *There is **enough contiguous free space**, but the **allocator won't be able to find it***

Lead to "false fragmentation" ☹

Let's Think about Free

1 square = 1 word = 8 bytes
: Free word
: Allocated word





Any problems?

Topic: How to free?

nough contiguous free space,
ocator won't be able to find it

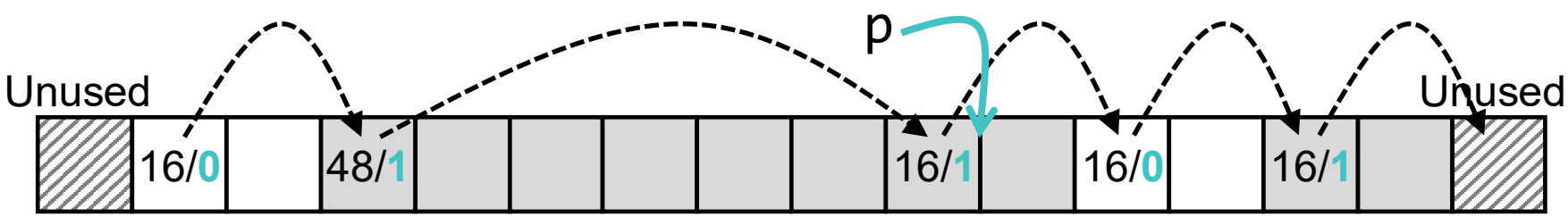
Lead to "false fragmentation" ☹

Coalescing

Joining with Adjacent Blocks: Coalescing

1 square = 1 word = 8 bytes
: Free word
: Allocated word

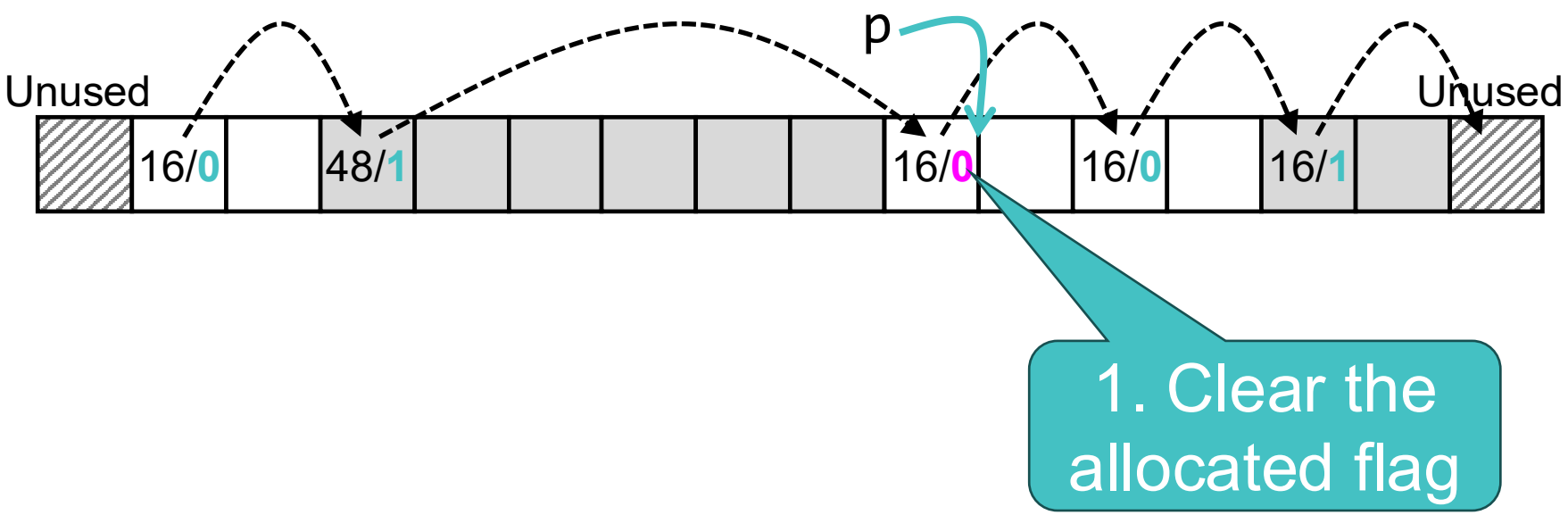
- Join (coalesce) with next/previous blocks, if they are free



Joining with Adjacent Blocks: Coalescing

1 square = 1 word = 8 bytes
: Free word
: Allocated word

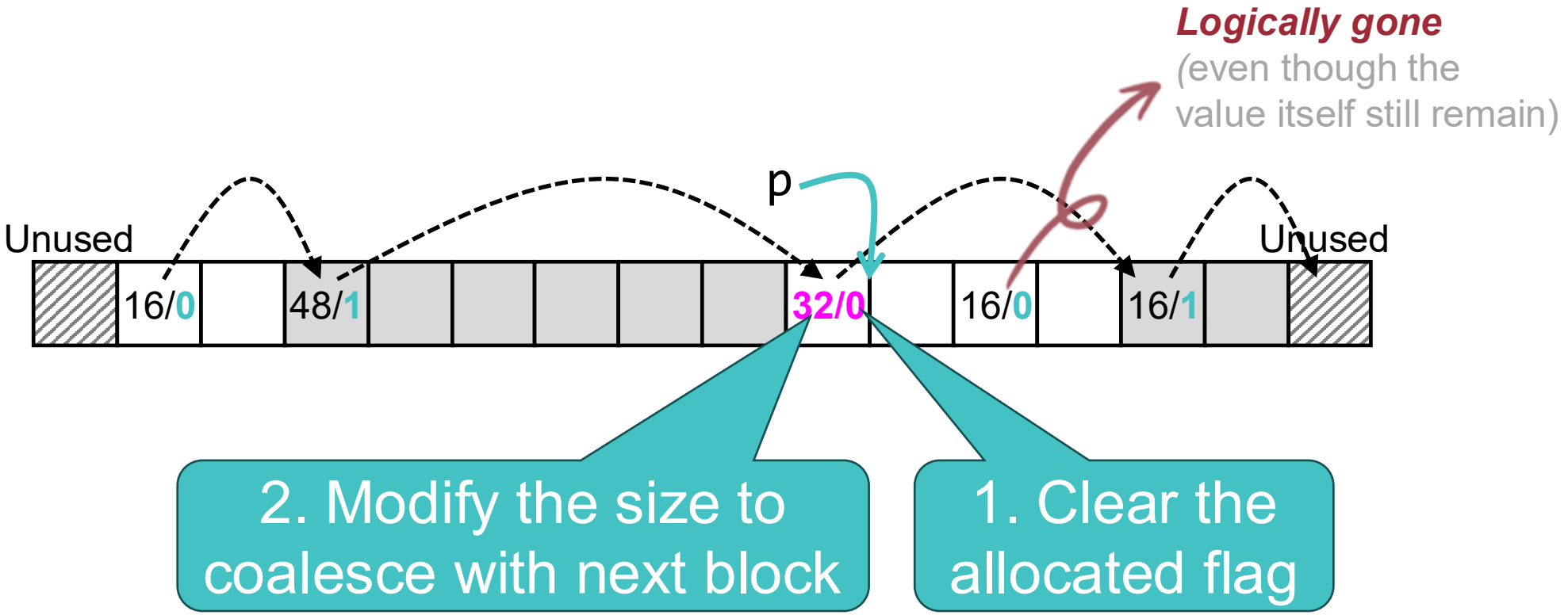
- Join (coalesce) with next/previous blocks, if they are free



Joining with Adjacent Blocks: Coalescing

1 square = 1 word = 8 bytes
: Free word
: Allocated word

- Join (coalesce) with next/previous blocks, if they are free



Resolve “false fragmentation” ☺

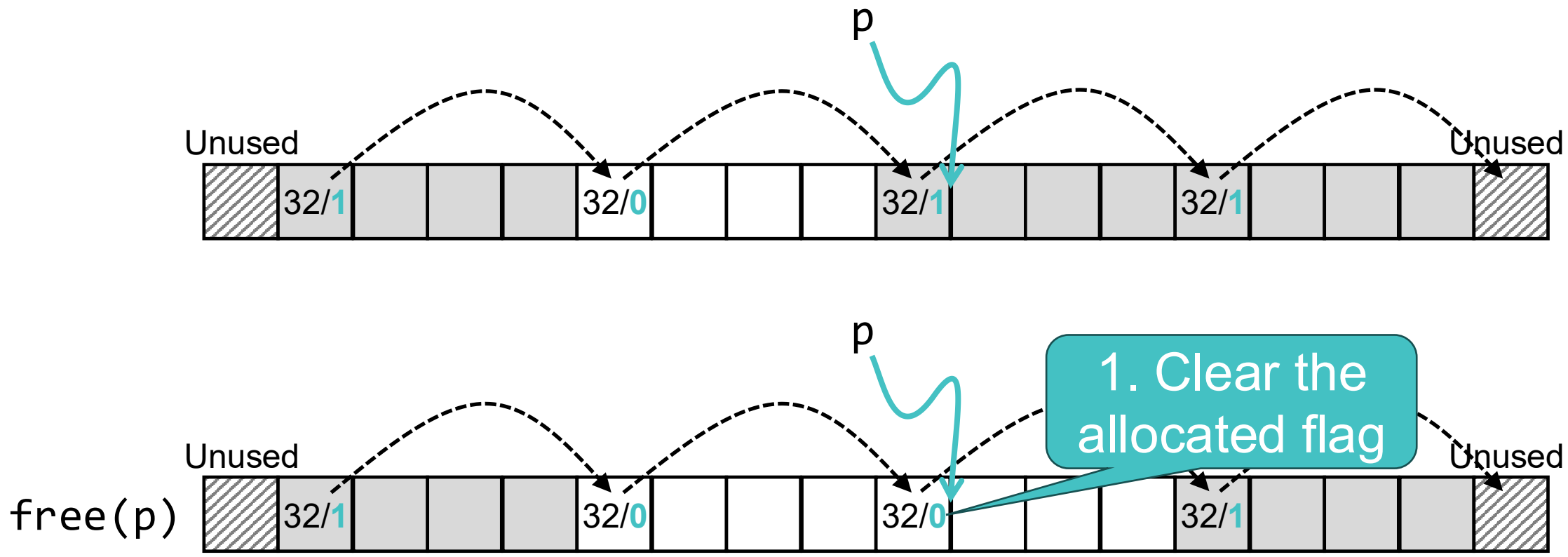
Problems of the Current Coalescing

- Coalescing **with the next block is easy**
- **But**, how about coalescing **with the previous block?**

Problems of the Current Coalescing

1 square = 1 word = 8 bytes
: Free word
: Allocated word

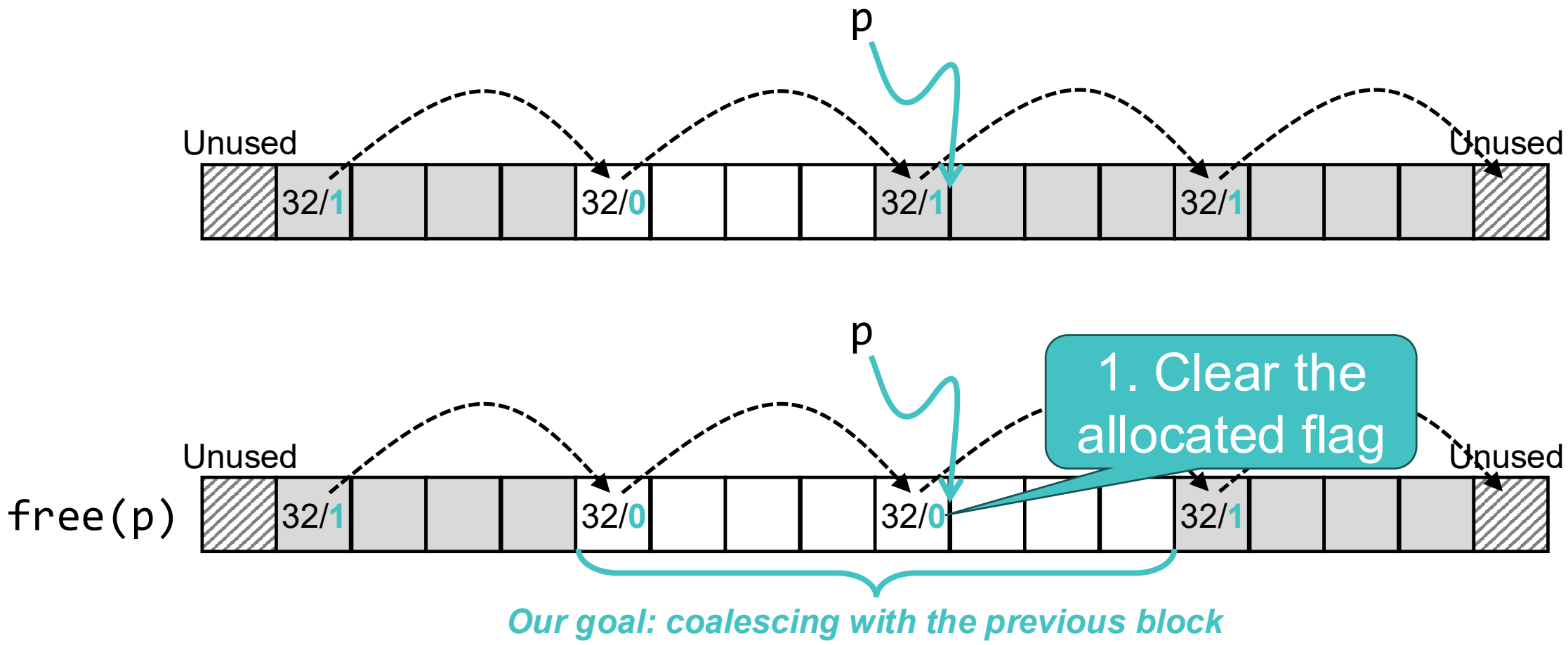
- Coalescing **with the next block is easy**
- **But**, how about coalescing **with the previous block?**



Problems of the Current Coalescing

1 square = 1 word = 8 bytes
: Free word
: Allocated word

- Coalescing **with the next block is easy**
- **But**, how about coalescing **with the previous block?**

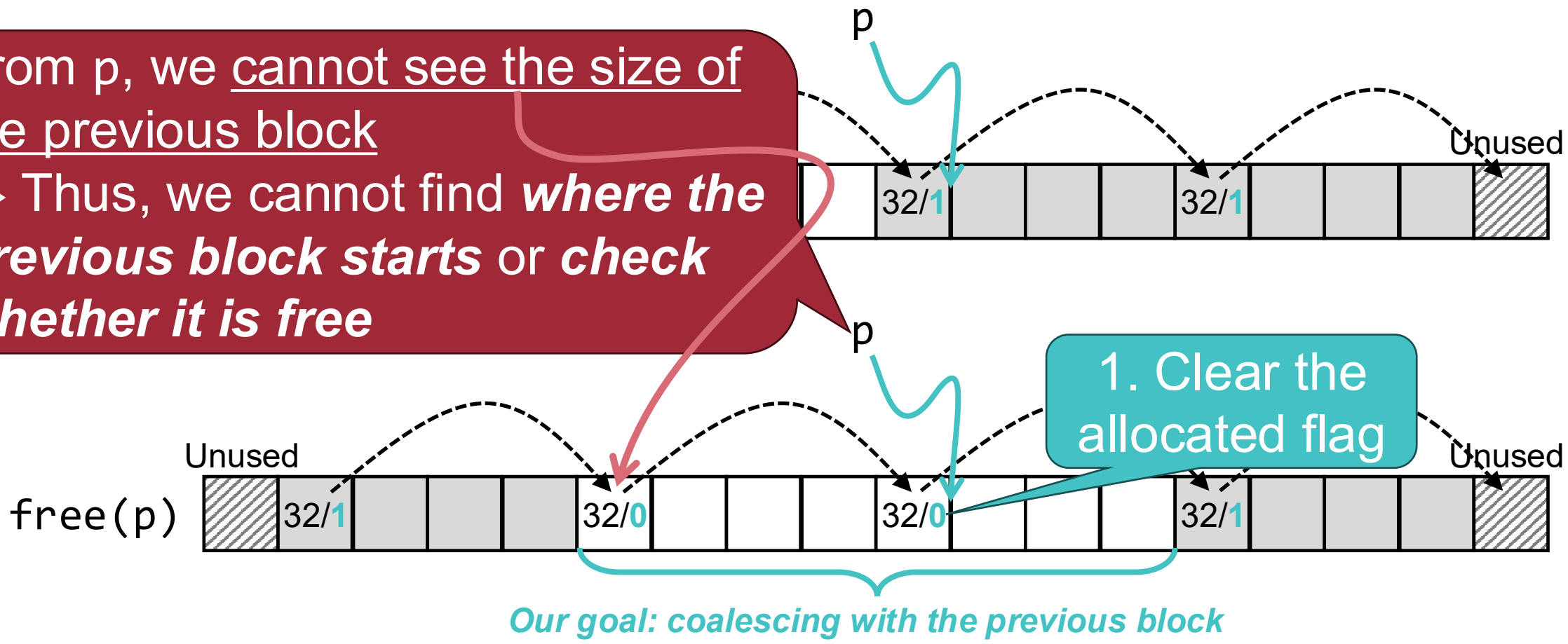


Problems of the Current Coalescing

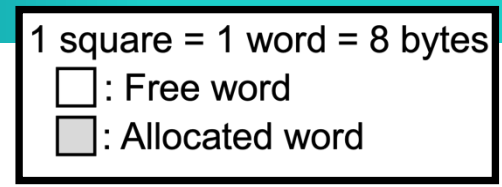
1 square = 1 word = 8 bytes
□: Free word
▨: Allocated word

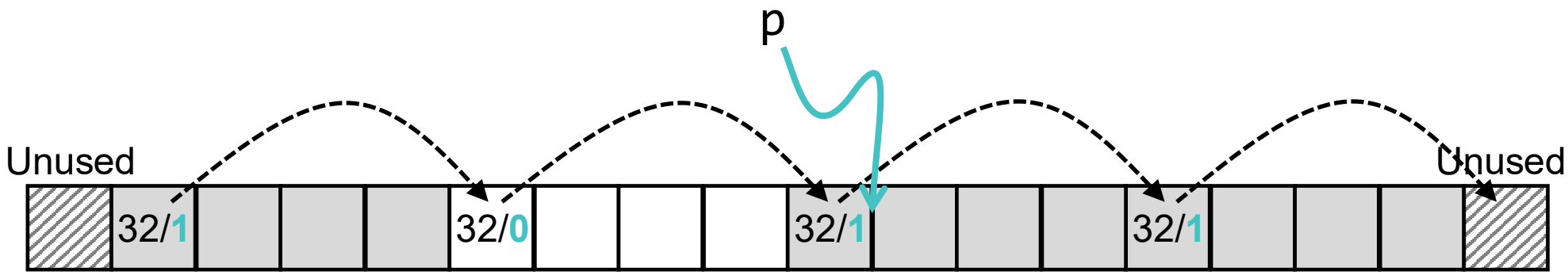
- Coalescing **with the next block is easy**
- **But**, how about coalescing **with the previous block?**

From p, we cannot see the size of the previous block
→ Thus, we cannot find ***where the previous block starts*** or ***check whether it is free***



How to Resolve This Problem?

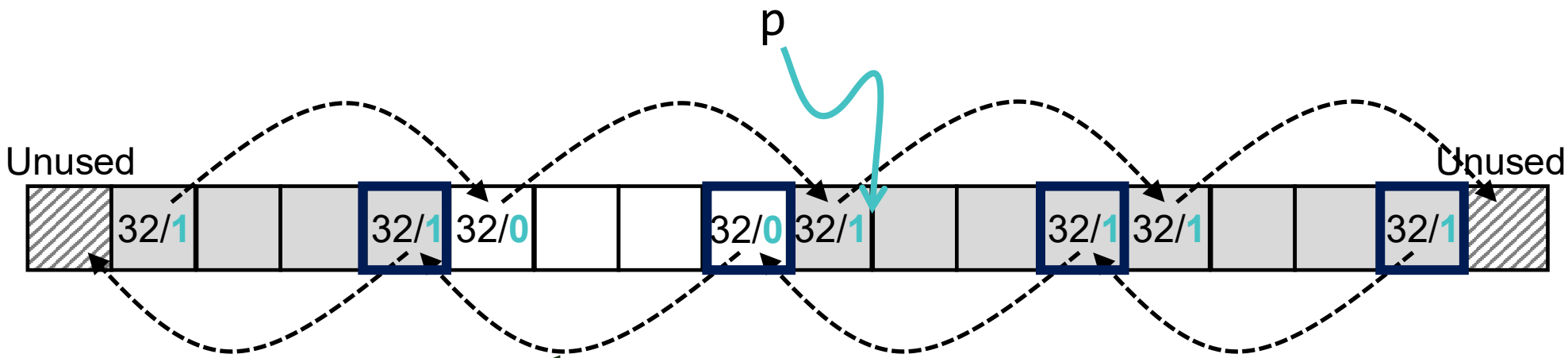




Idea: Bidirectional Coalescing

1 square = 1 word = 8 bytes
: Free word
: Allocated word

Boundary tags: replicate size/allocated word at *"bottom"* (end) of blocks



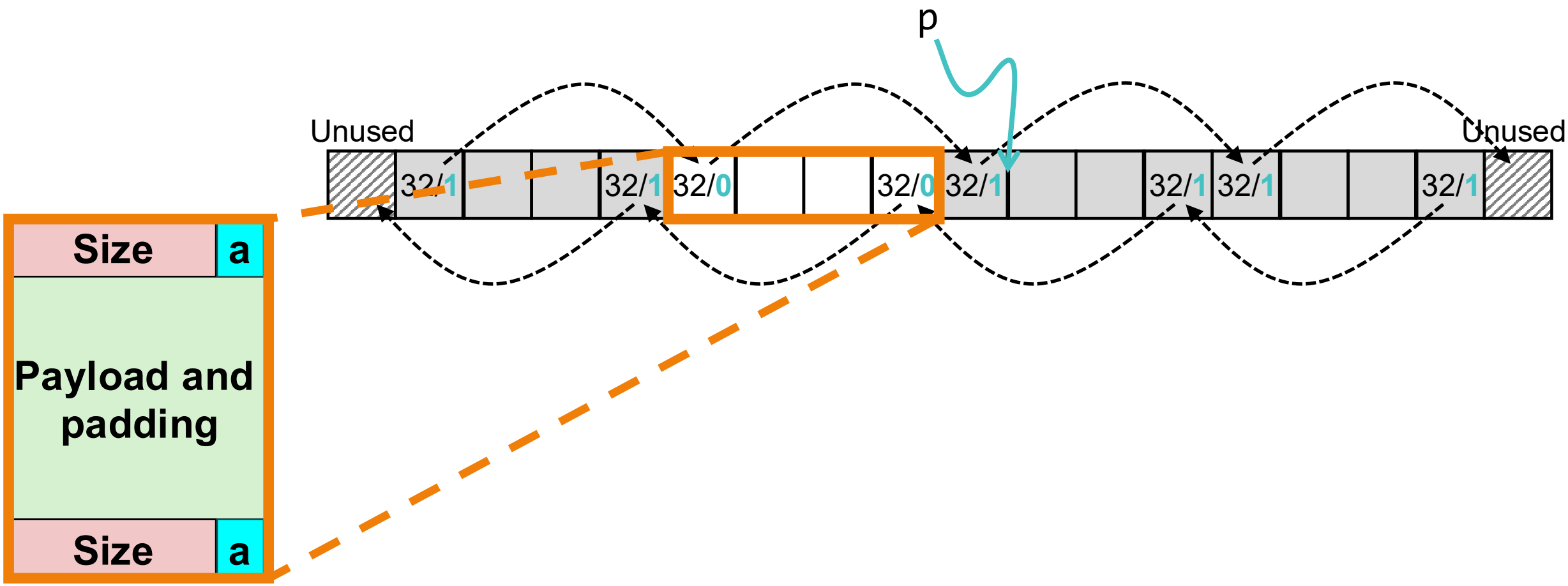
Allows us to traverse the "list" ***backwards***, but requires extra space

Donald E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison-Wesley.

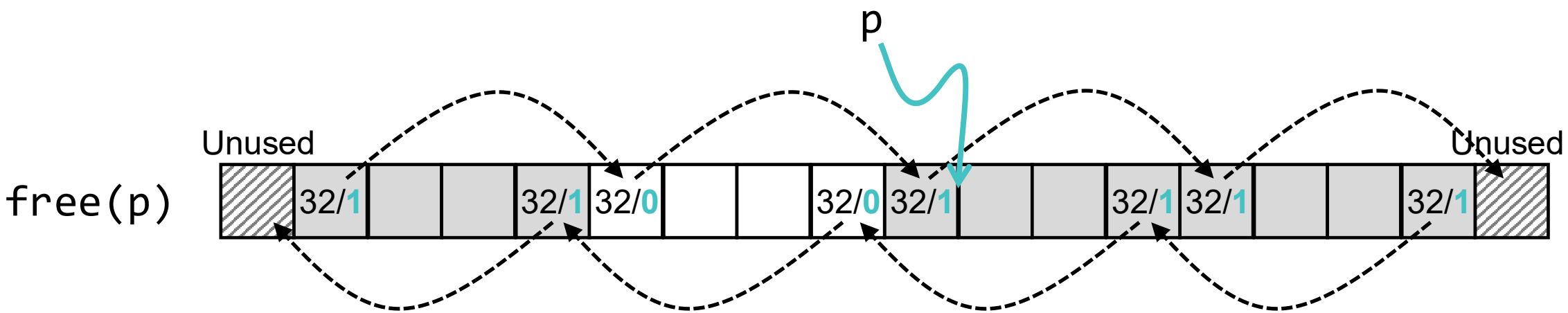
Idea: Bidirectional Coalescing

1 square = 1 word = 8 bytes
: Free word
: Allocated word

Boundary tags: replicate size/allocated word at *"bottom"* (end) of blocks



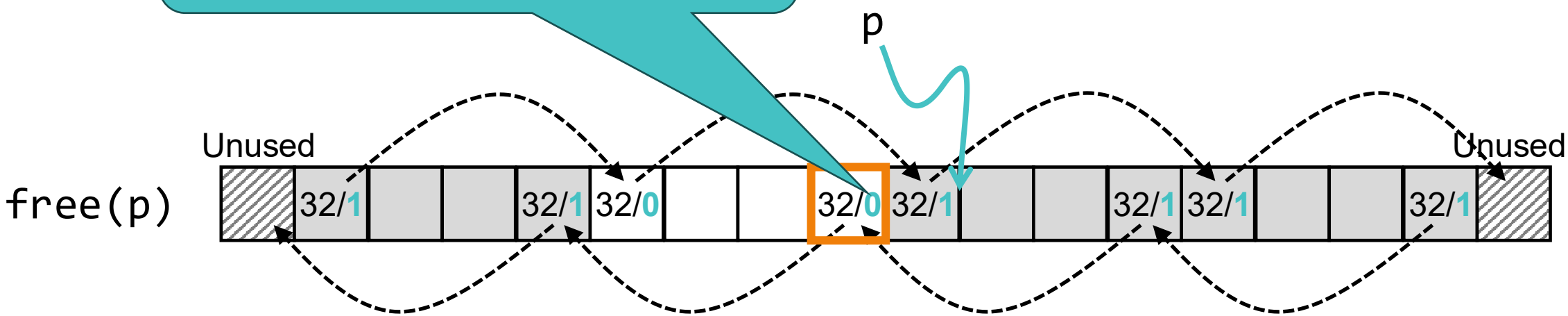
Coalescing with the Previous Block



Coalescing with the Previous Block

1 square = 1 word = 8 bytes
: Free word
: Allocated word

1. Check the **previous block's footer** `*(p-2)` to see if it is free

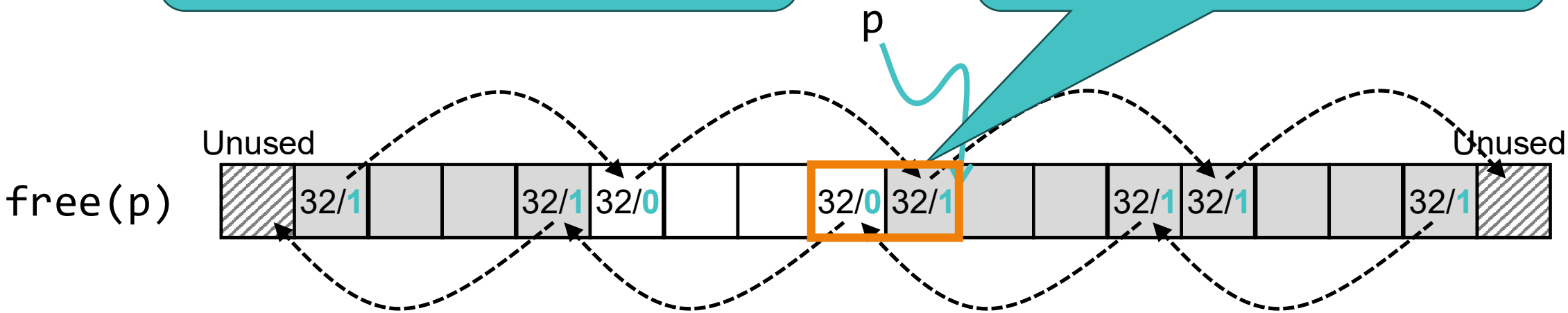


Coalescing with the Previous Block

1 square = 1 word = 8 bytes
: Free word
: Allocated word

1. Check the **previous block's footer** `*(p-2)` to see if it is free

2. Calculate **the size of the coalesced block** (=64)

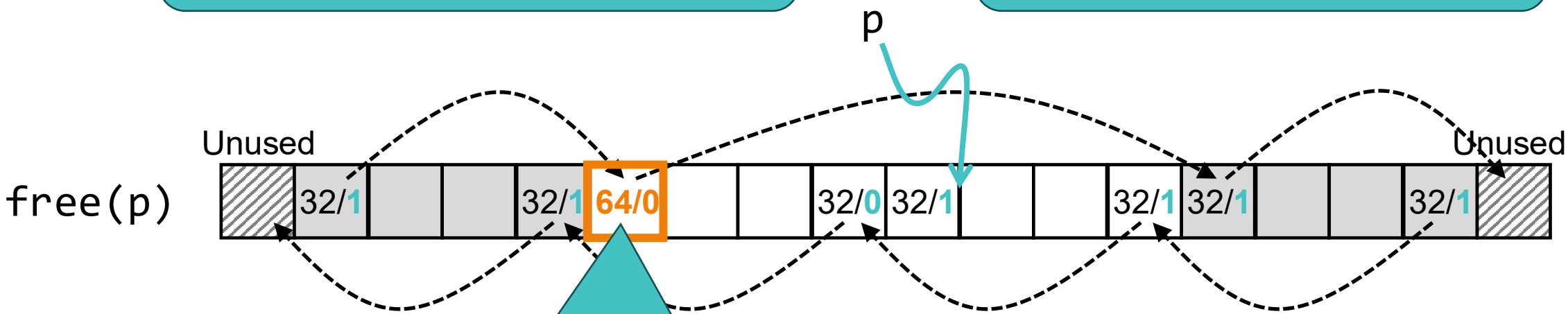


Coalescing with the Previous Block

1 square = 1 word = 8 bytes
: Free word
: Allocated word

1. Check the **previous block's footer** `*(p-2)` to see if it is free

2. Calculate **the size of the coalesced block** (=64)



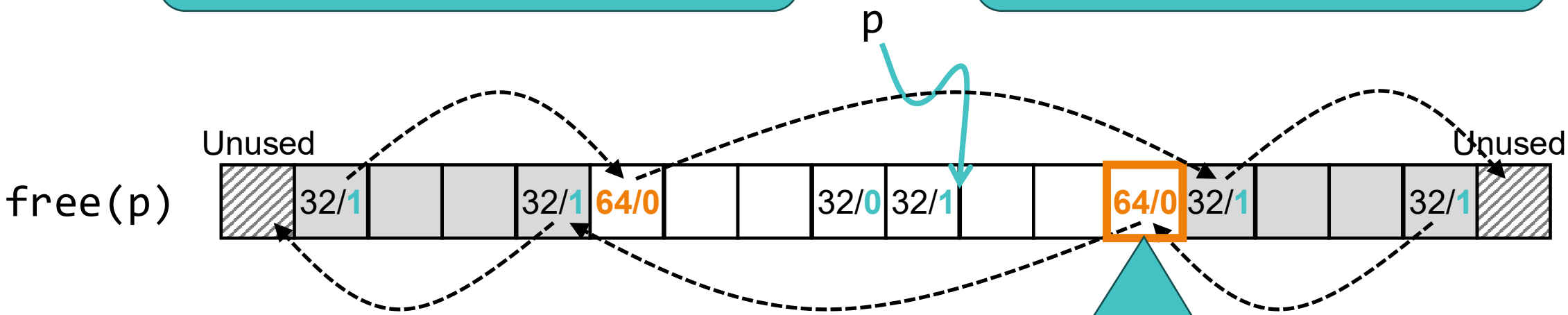
3. Write **the coalesced block's header**

Coalescing with the Previous Block

1 square = 1 word = 8 bytes
: Free word
: Allocated word

1. Check the **previous block's footer** *(p-2) to see if it is free

2. Calculate **the size of the coalesced block** (=64)

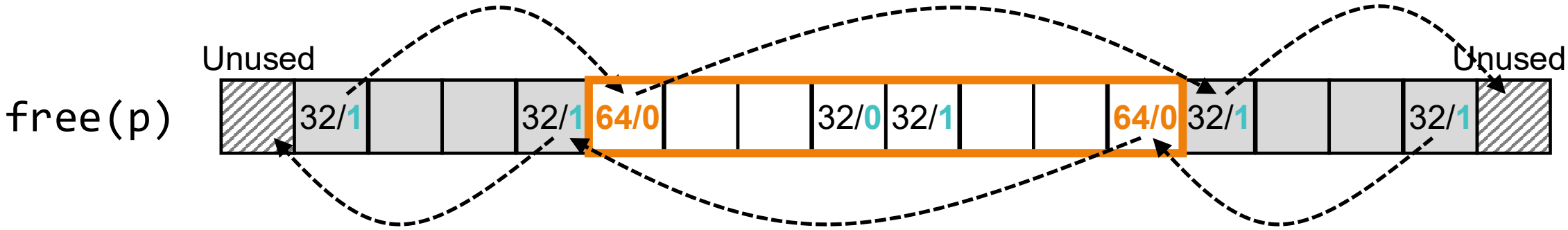


3. Write **the coalesced block's header**

4. Write **the coalesced block's footer**

Coalescing with the Previous Block

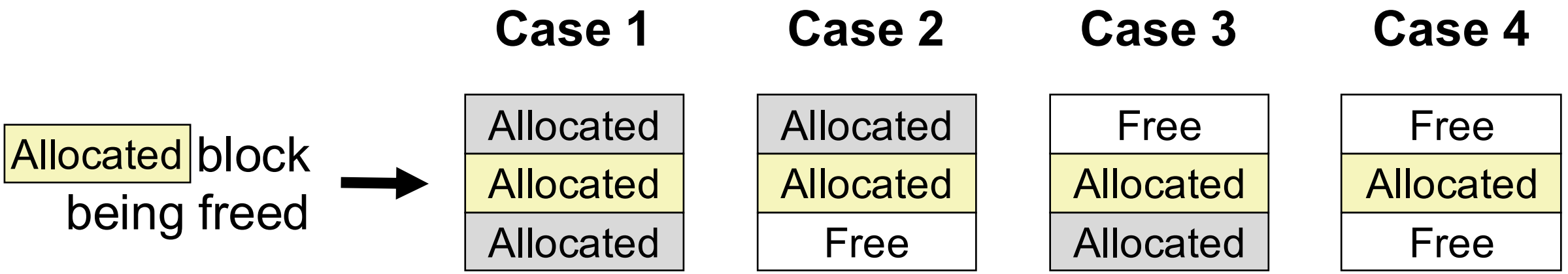
1 square = 1 word = 8 bytes
: Free word
: Allocated word



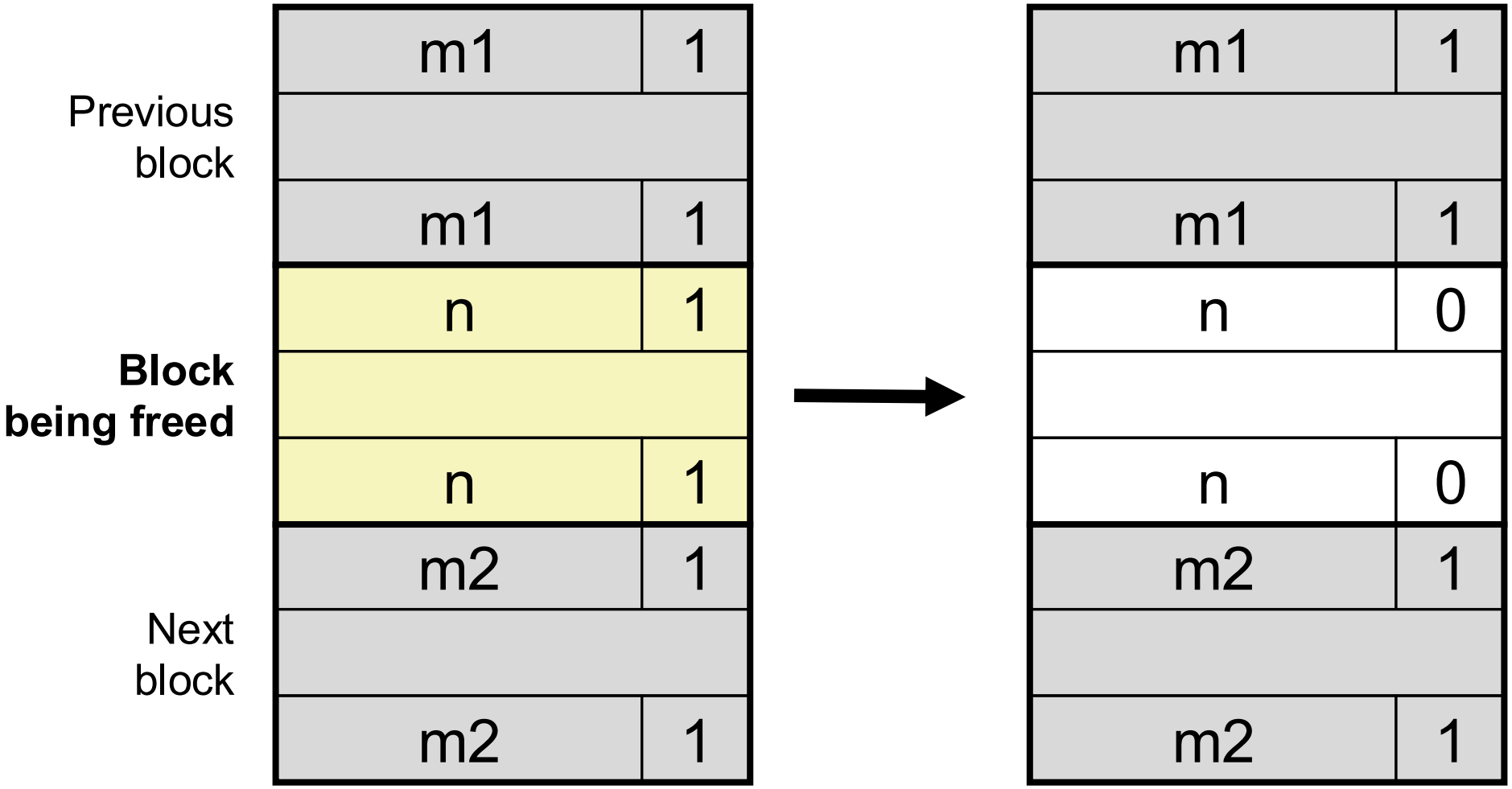
Coalescing can be done in *constant time* without traversing the entire list, which would take linear time!

Let's omit the code, as it can be easily derived from the algorithm described above

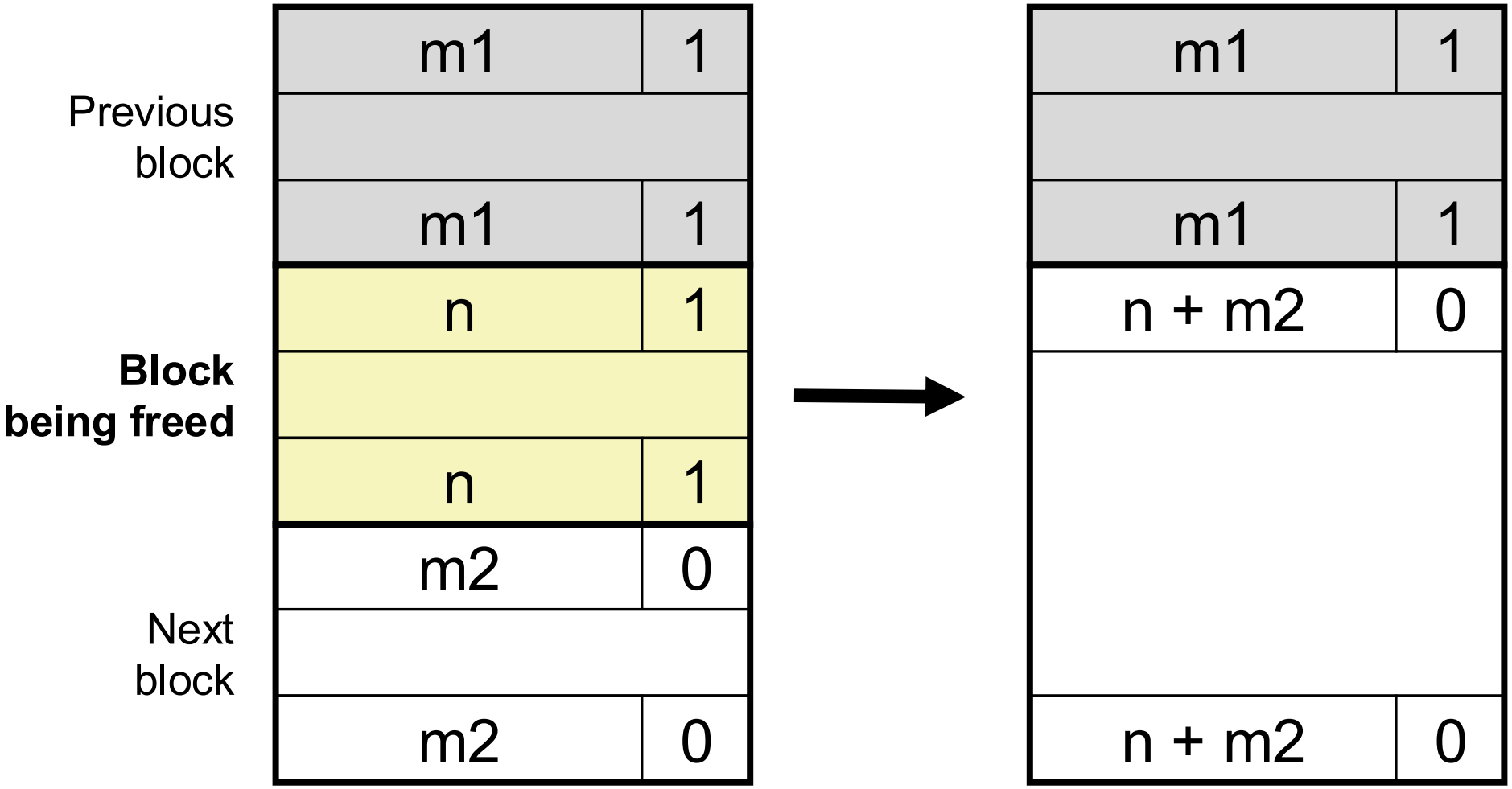
Coalescing Case Study



Coalescing Case #1

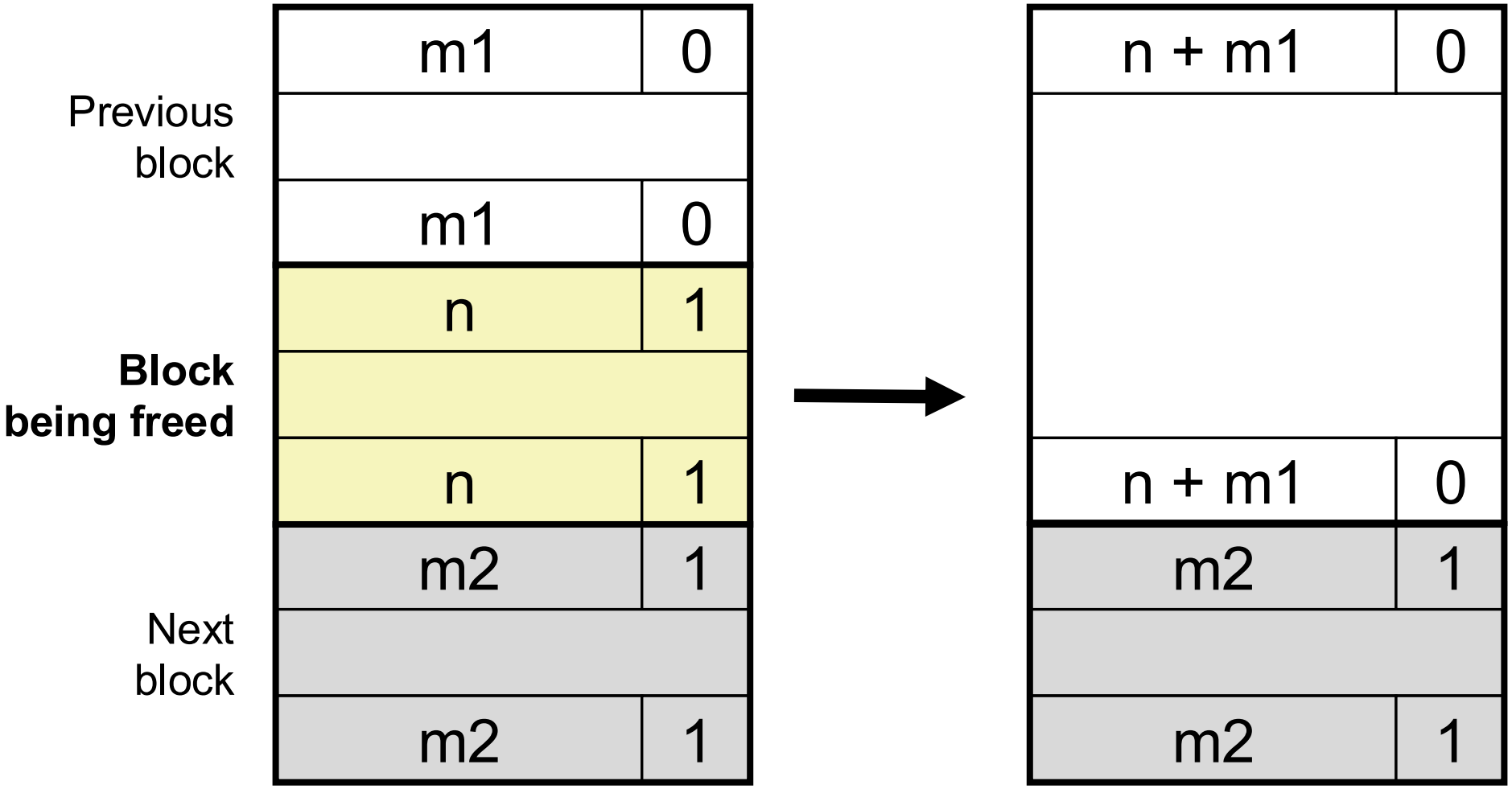


Coalescing Case #2

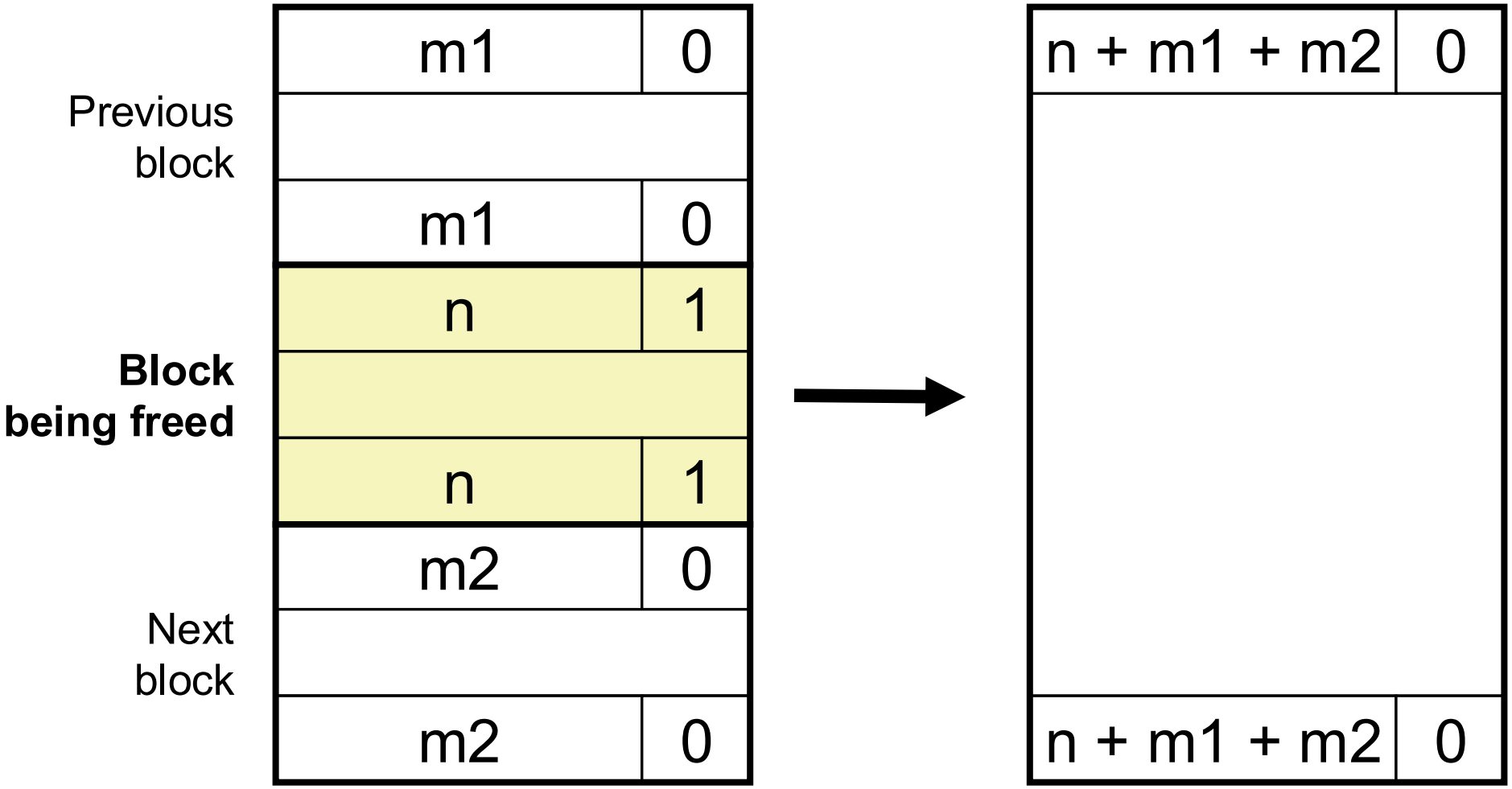


Allocated | Allocated | Free

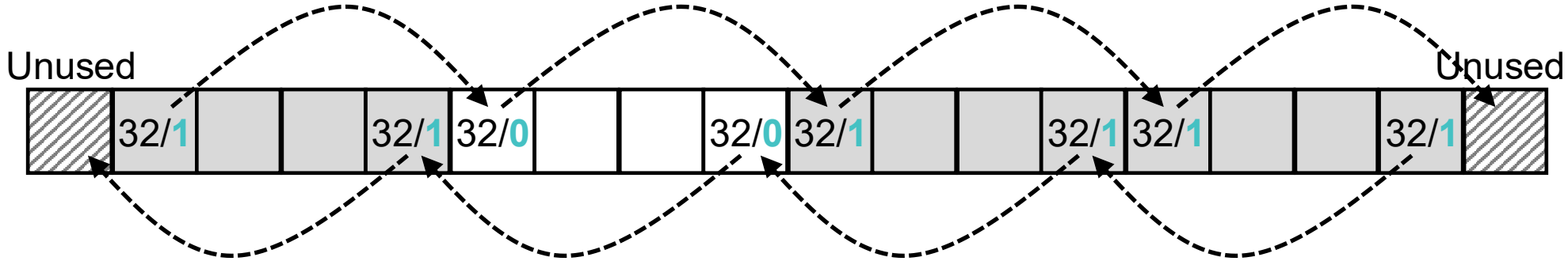
Coalescing Case #3



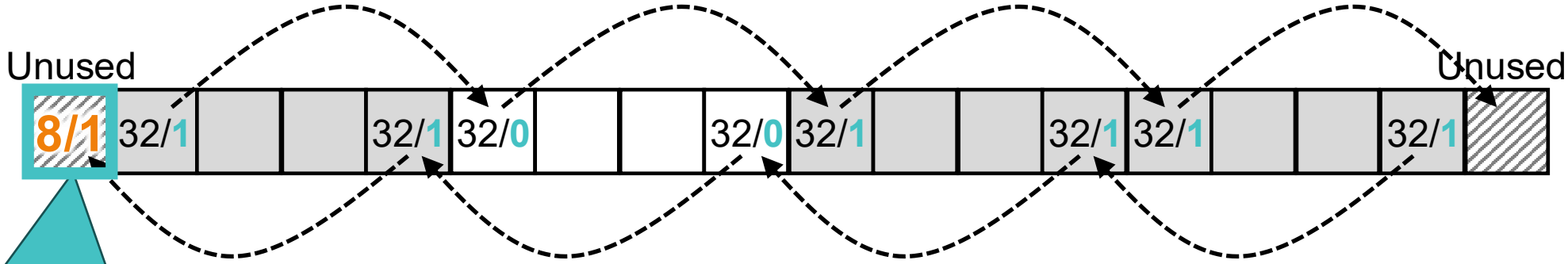
Coalescing Case #4



Header and Footer



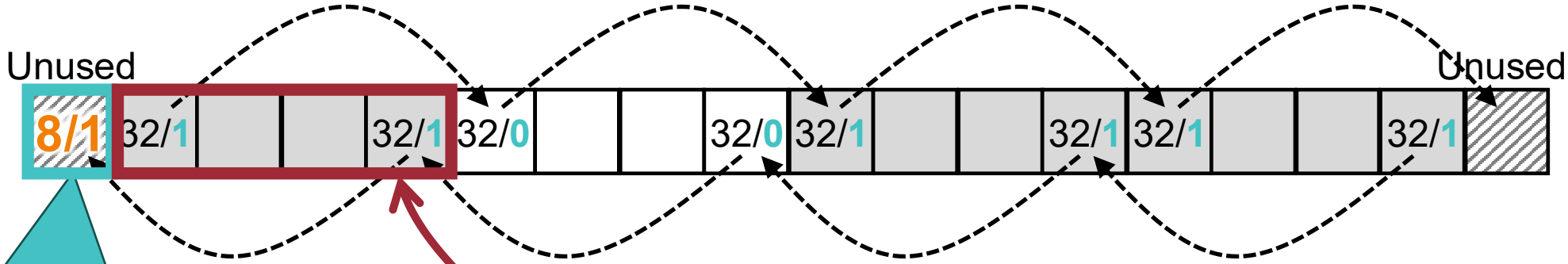
Header and Footer



Unused block: **dummy footer** before first header

- Marked as **allocated**
- **Prevents accidental coalescing** when freeing first block

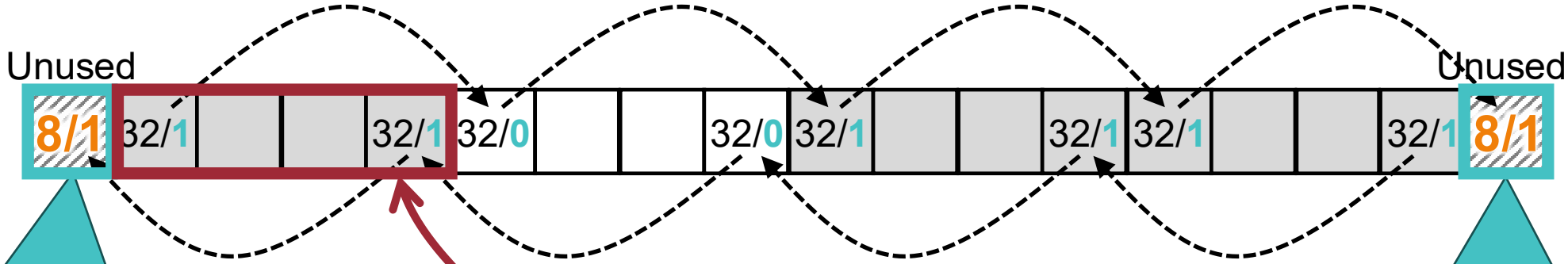
Header and Footer



Unused block: **dummy footer** before first header

- Marked as **allocated**
- **Prevents accidental coalescing** when freeing first block

Header and Footer



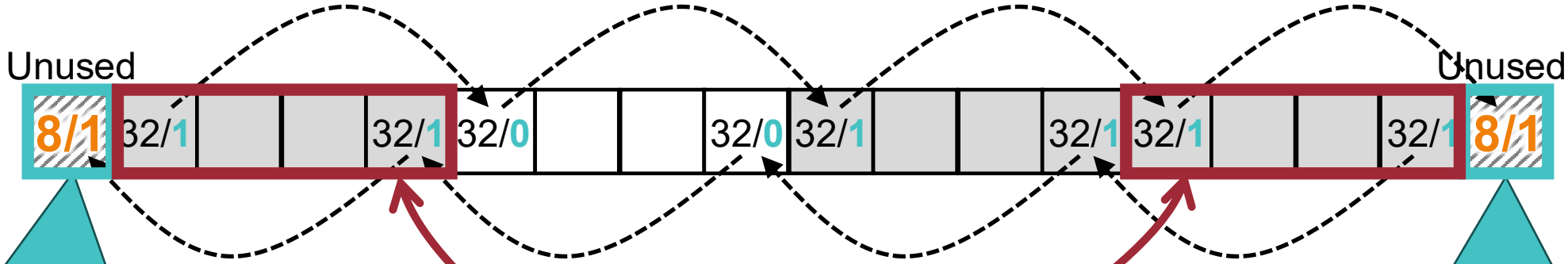
Unused block: **dummy footer** before first header

- Marked as **allocated**
- **Prevents accidental coalescing** when freeing first block

Unused block: **dummy header** after last footer

- Marked as **allocated**
- **Prevents accidental coalescing** when freeing final block

Header and Footer



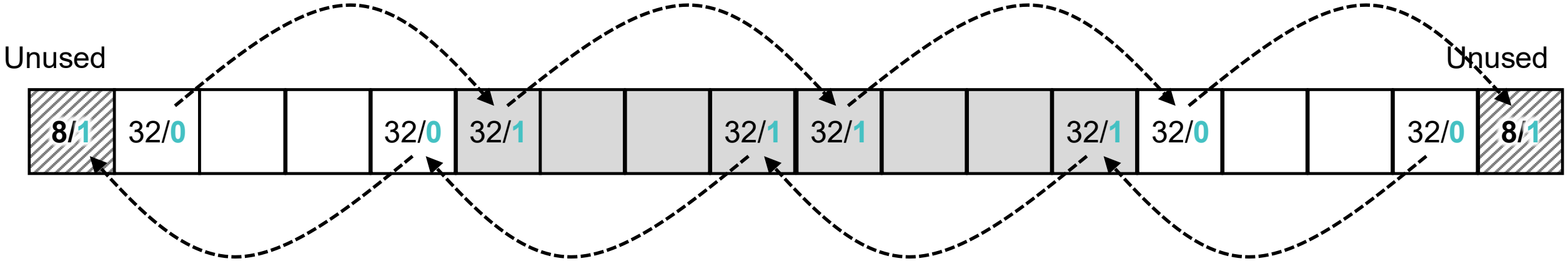
Unused block: **dummy footer** before first header

- Marked as **allocated**
- **Prevents accidental coalescing** when freeing first block

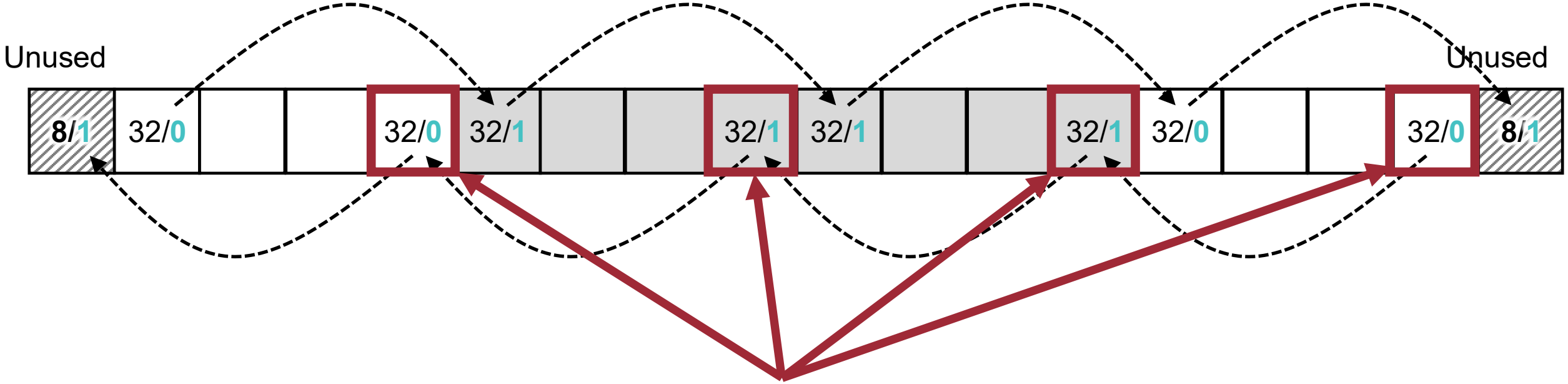
Unused block: **dummy header** after last footer

- Marked as **allocated**
- **Prevents accidental coalescing** when freeing final block

Disadvantages of Boundary Tags



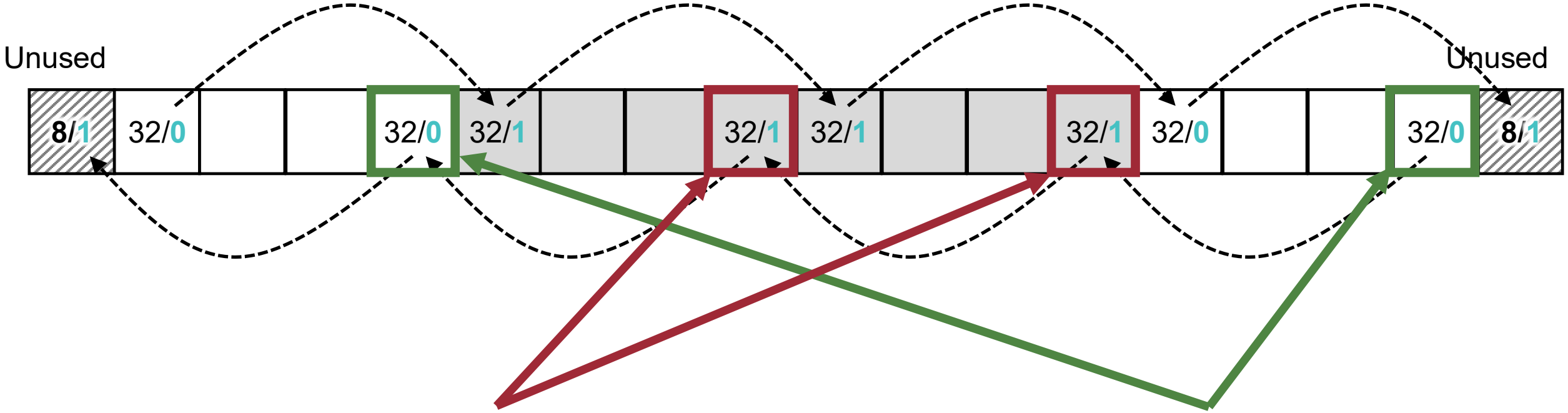
Disadvantages of Boundary Tags



Internal fragmentation

How we address this problem?

Observation: Which Blocks Need the Footer Tag?

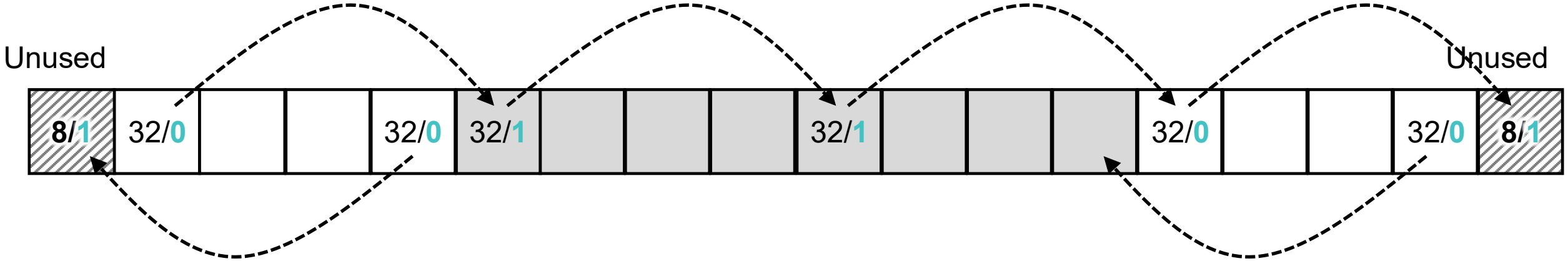


Allocated blocks do not need footer tags

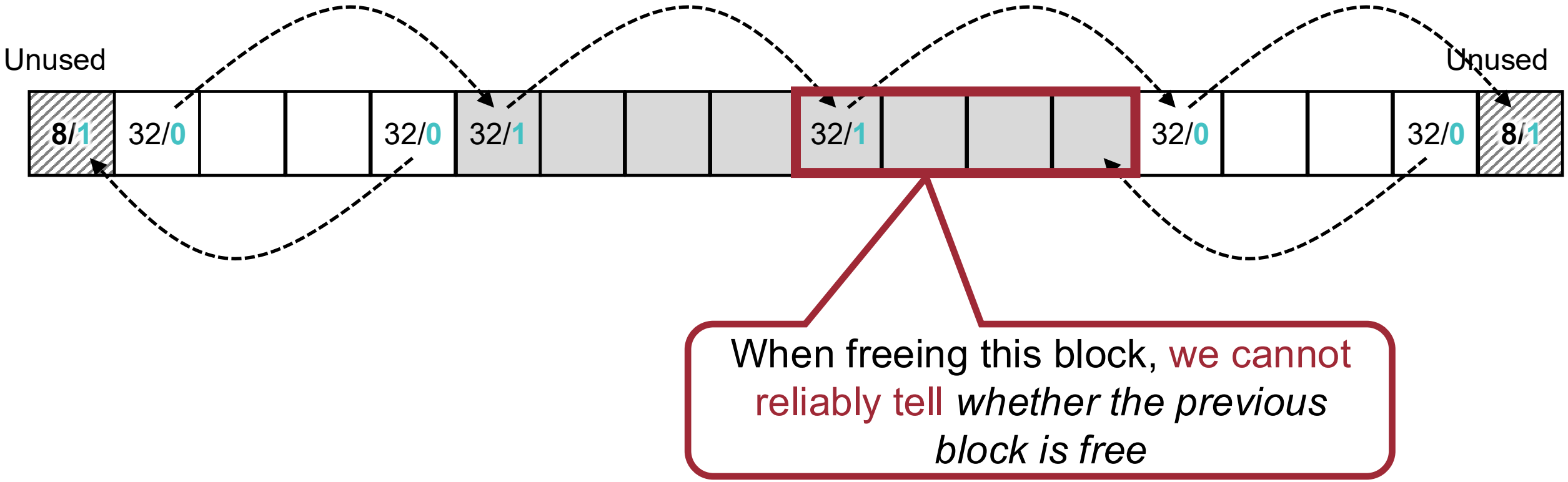
Free blocks always need the footer tag

Why not remove this one to mitigate internal fragmentation? ☺

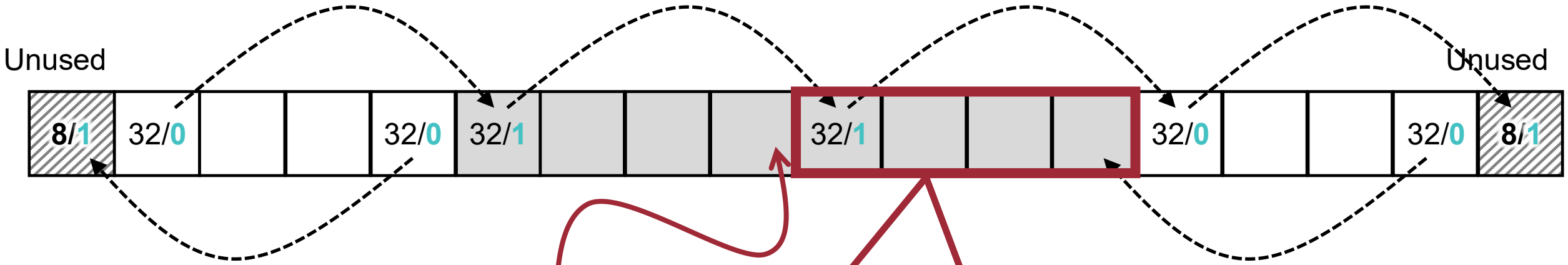
Solution: No Boundary Tag for Allocated Blocks



Any Problems?



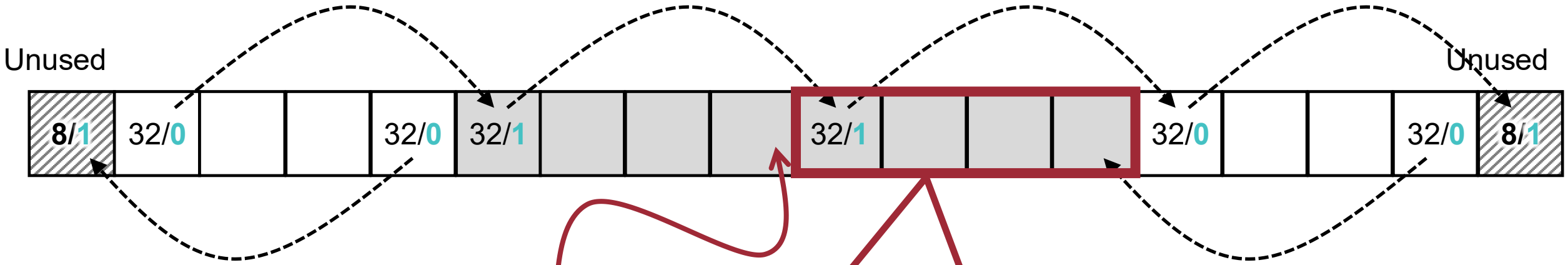
Any Problems?



If the previous block is allocated, it has no footer. Therefore, the word before the current block may just be payload data, and its lowest bit may accidentally be *0*

When freeing this block, we cannot reliably tell *whether the previous block is free*

Any Problems?



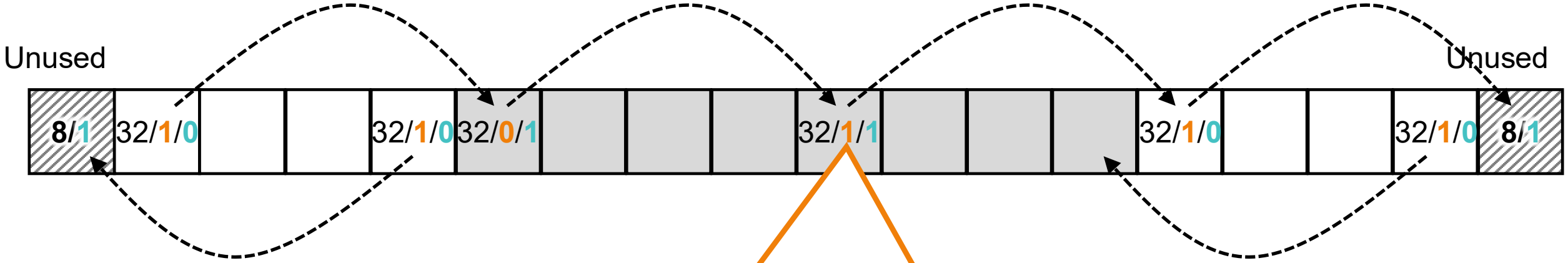
If the previous block is allocated, it has no footer. Therefore, the word before the current block may just be payload data, and its lowest bit may accidentally be *0*

When freeing this block, we cannot reliably tell *whether the previous block is free*

Thus, we have no way to know whether we should coalesce with the previous block.

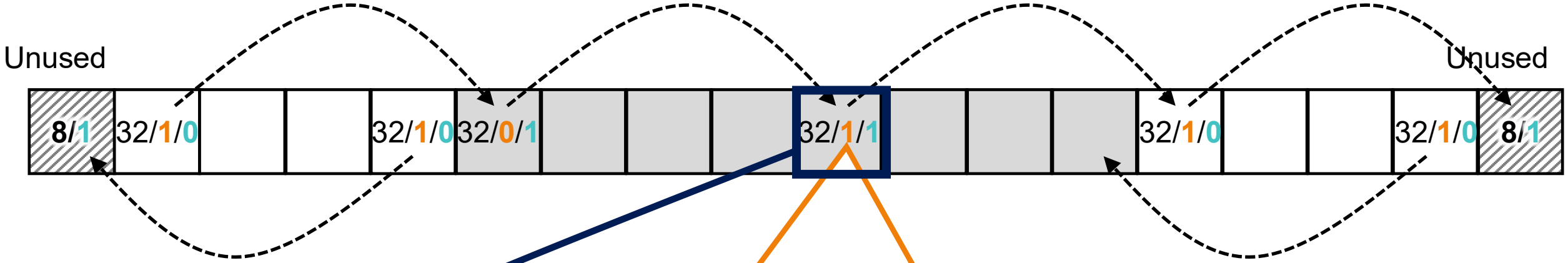
How can we solve this problem?

Optimization Idea: Additional Field



Additional field for coalescing: whether the previous block is allocated (1) or free (0)

Optimization Idea: Additional Field

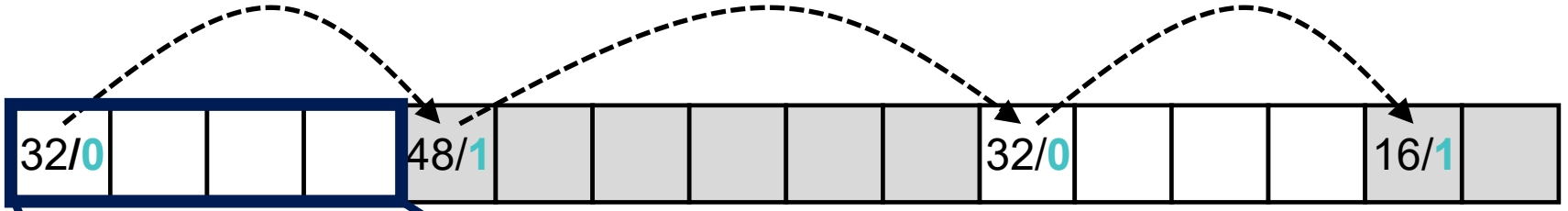


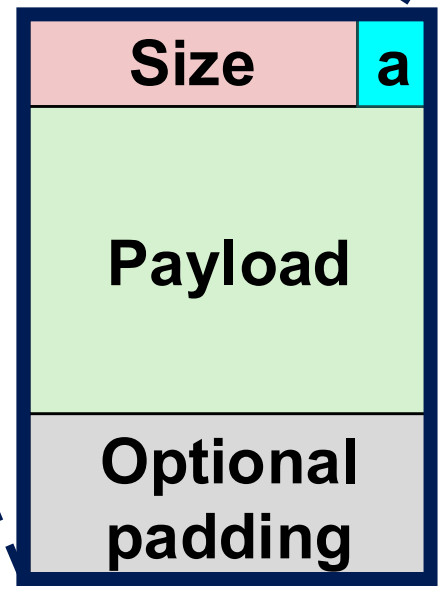
Additional field for coalescing: whether the previous block is allocated (1) or free (0)

We want to express multiple meanings
(**block size**, **previous block's allocation status**, **current block's allocation status**)
in a single word. How it can be possible?

Recap: Standard Track

1 square = 1 word = 8 bytes
: Free word
: Allocated word





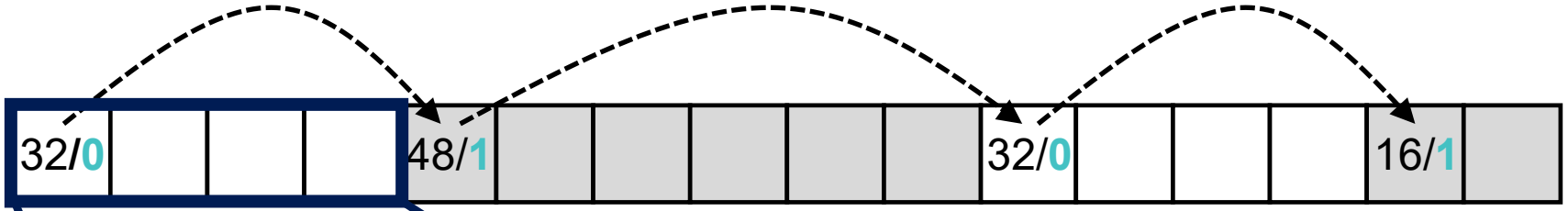
- Remember that a block is aligned to a 16 byte-boundary
- Possible block size:
  - 16 bytes = 010000 (2)
  - 32 bytes = 100000 (2)
  - 48 bytes = 110000 (2)
  - ...

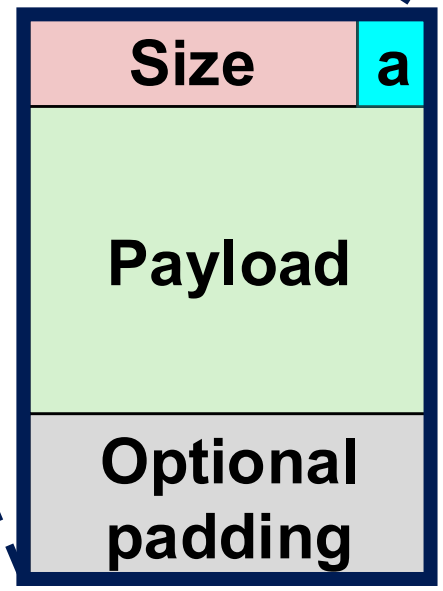
Observation: 4 low-order bits are always 0

Idea: use this bit as 'flag'
- **1**: Allocated block
- **0**: Free block

Recap: Standard Track

1 square = 1 word = 8 bytes
: Free word
: Allocated word





- ligned
- Possible block siz
  - 16 bytes = 010000 (2)
  - 32 bytes = 100000 (2)
  - 48 bytes = 110000 (2)
  -

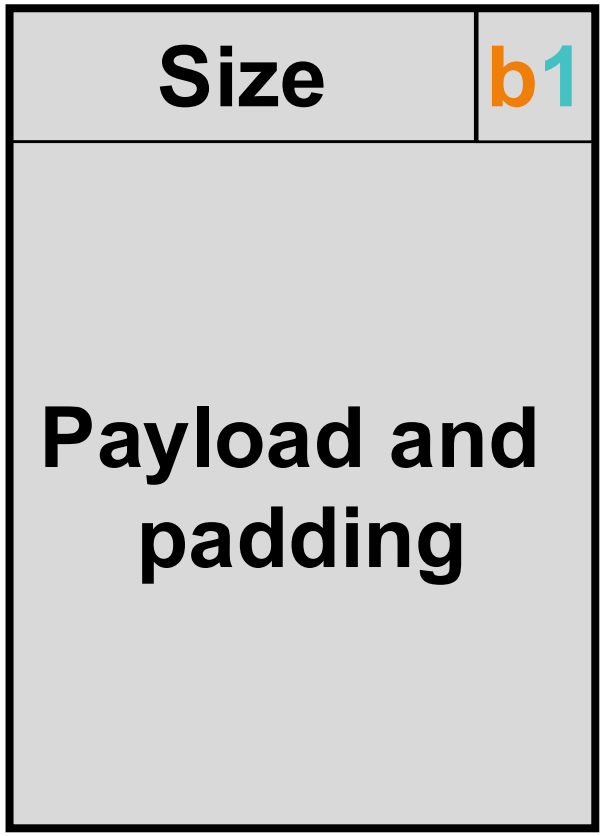
Let's use this field!
- **1**: Previous block is allocated
- **0**: Previous block is free

Observation: 4 low-order bits are always 0

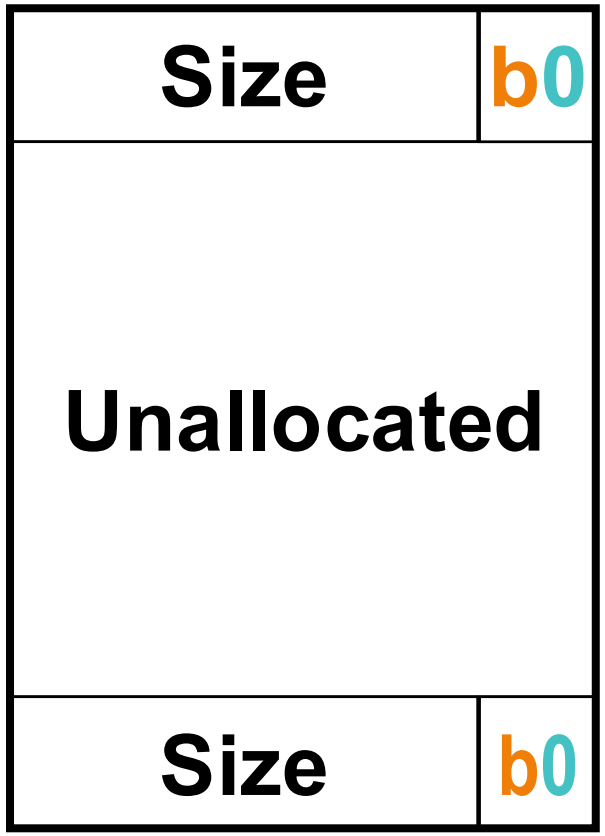
Idea: use this bit as 'flag'
- **1**: Allocated block
- **0**: Free block

Optimization Idea: Additional Field

- Whether the previous block is allocated (1) or free (0)



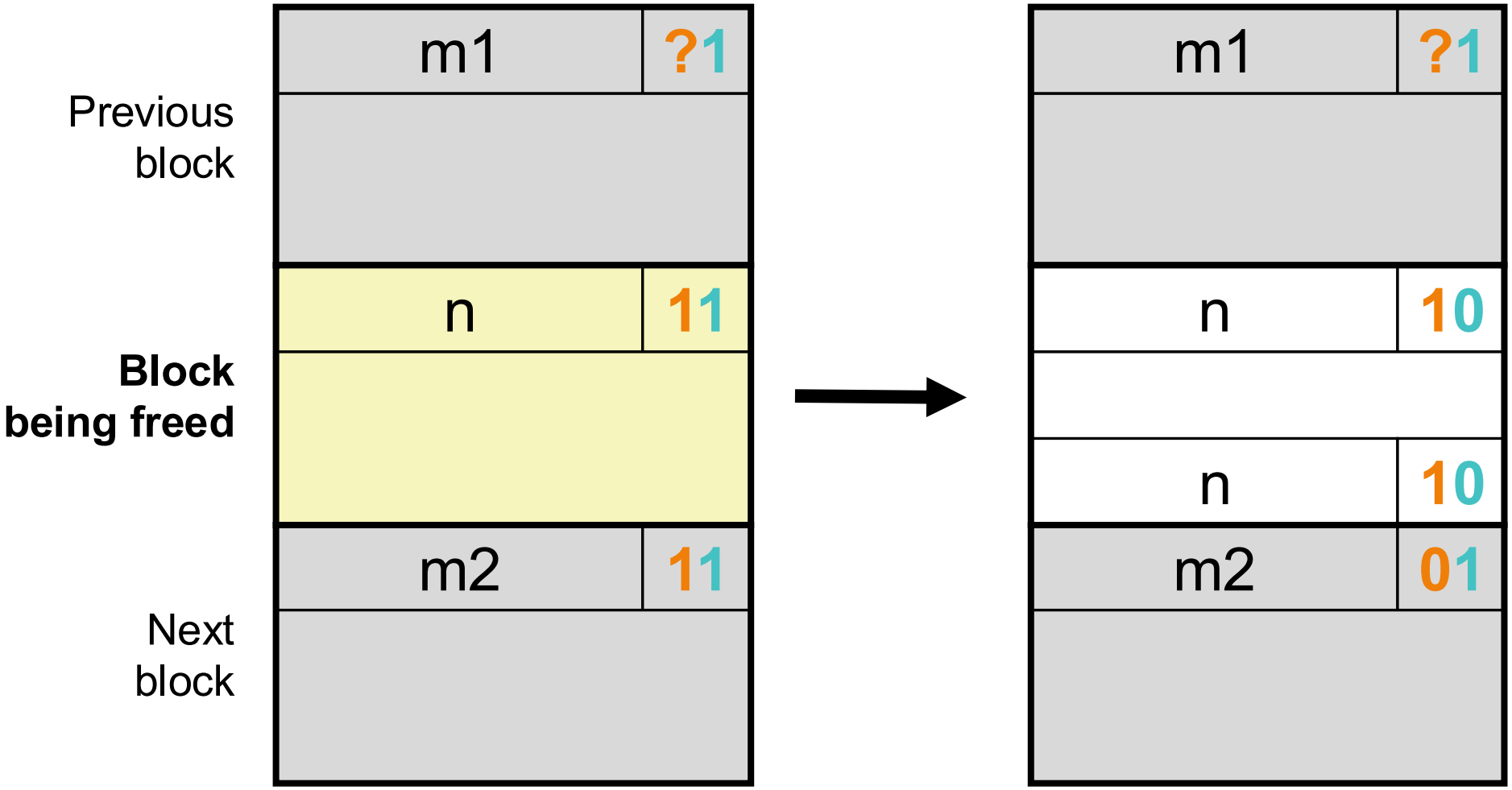
Allocated block



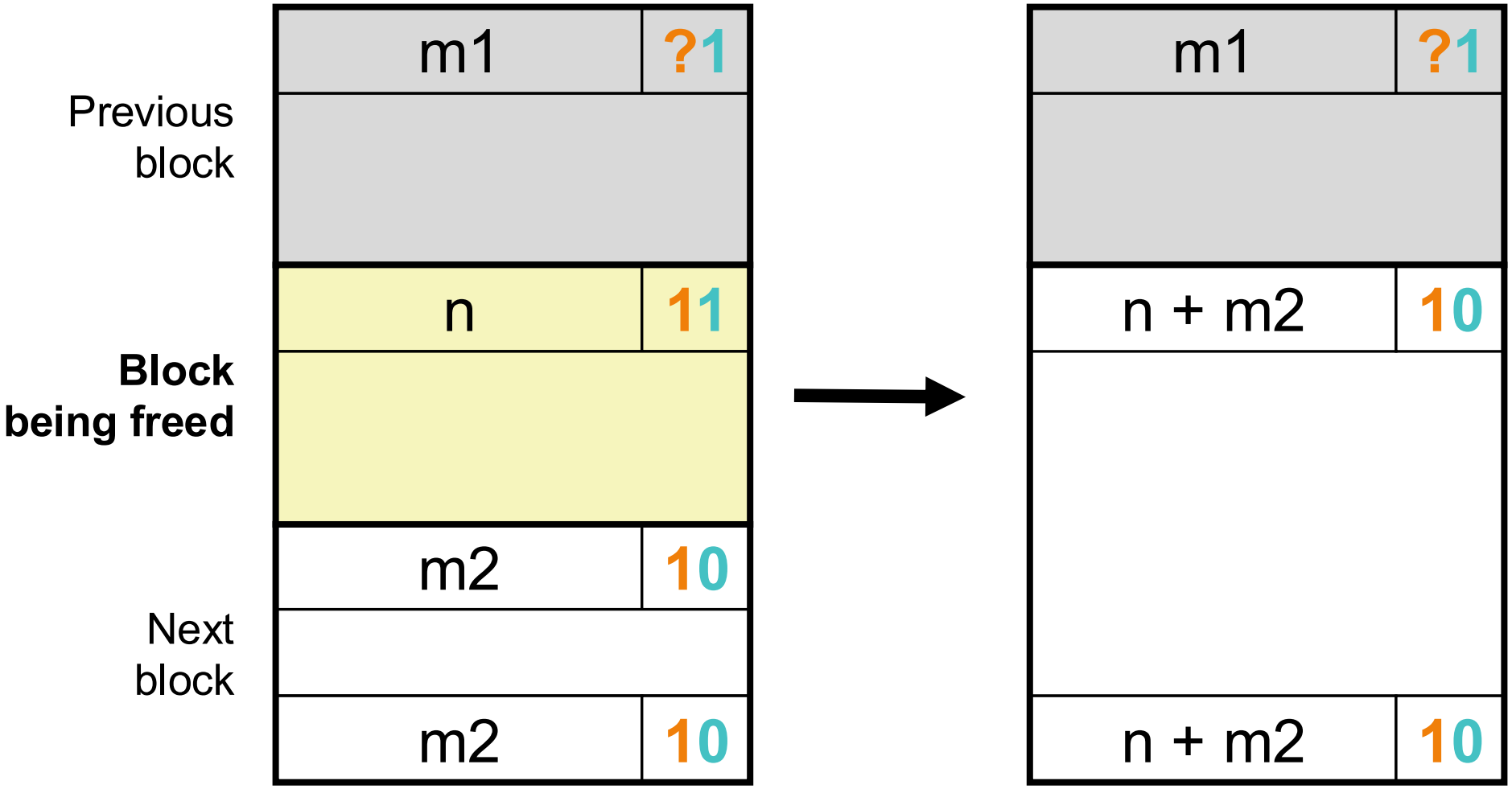
Free block

b = 1: Previous block is allocated
b = 0: Previous block is free

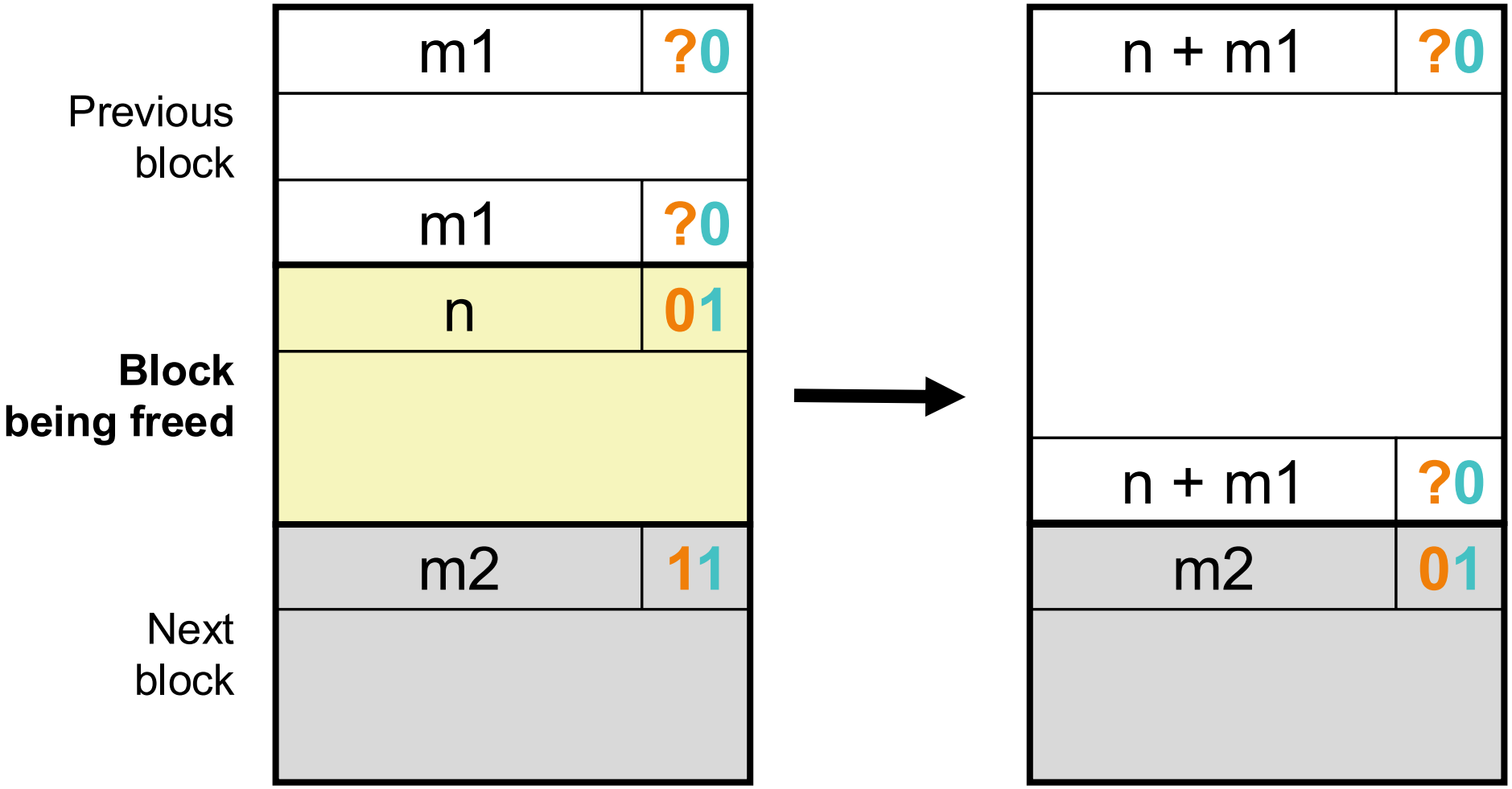
Coalescing (No Boundary Tag for Allocated Blocks) Case #1



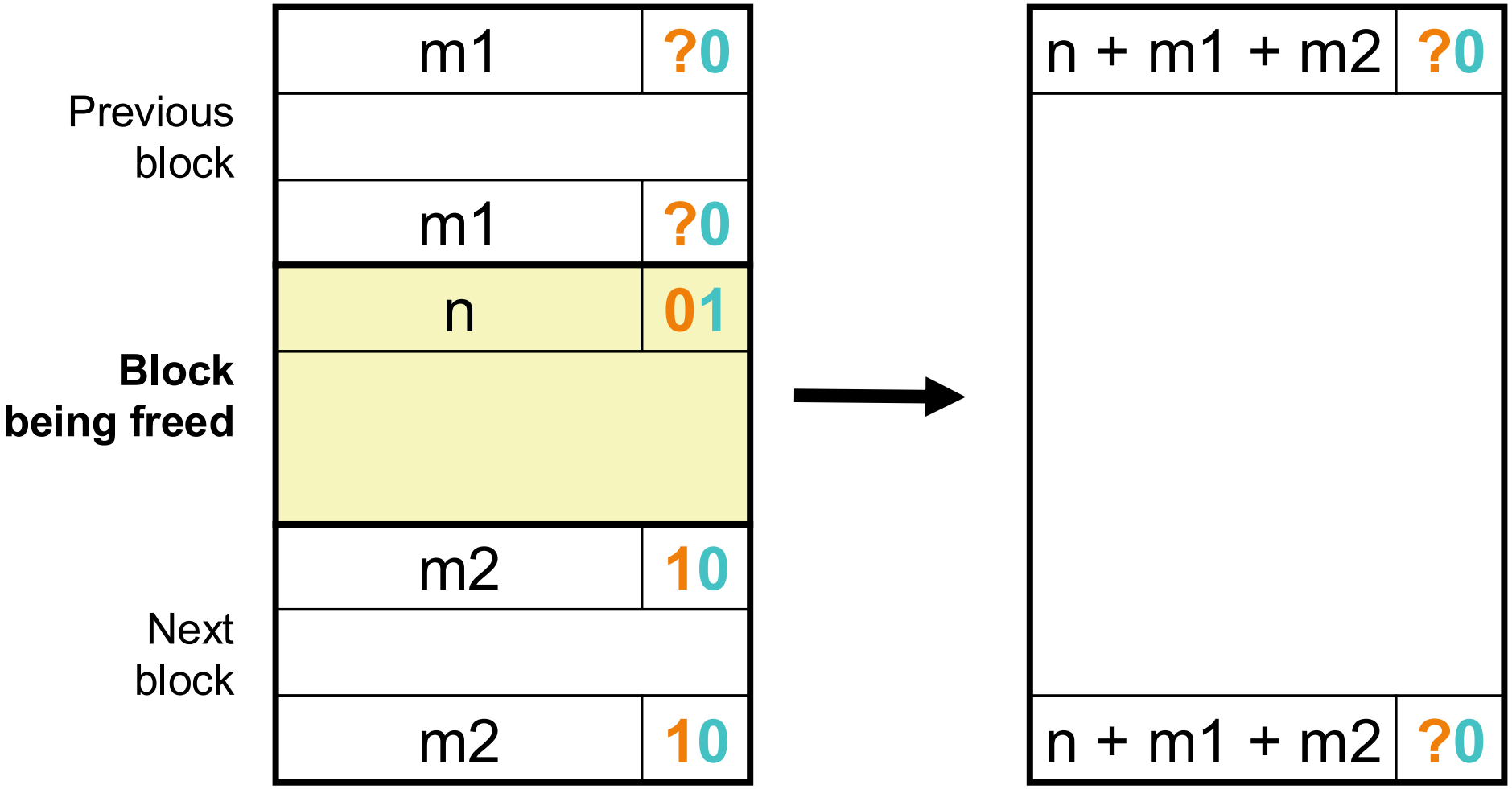
Coalescing (No Boundary Tag for Allocated Blocks) Case #2



Coalescing (No Boundary Tag for Allocated Blocks) Case #3



Coalescing (No Boundary Tag for Allocated Blocks) Case #4



Coalescing Policy

- **When should coalescing be performed?** There are two coalescing policies:
  - ***Immediate coalescing:*** coalesce each time `free` is called
  - ***Deferred coalescing:*** try to improve performance of `free` by deferring coalescing until needed
    - For example, when external fragmentation exceeds a certain threshold

(Summary) Key Allocator Policies

- **Placement policy:**
  - First-fit, next-fit, best-fit, etc.
  - Trades off lower throughput for less fragmentation
- **Splitting policy:**
  - When do we go ahead and split free blocks?
  - How much internal fragmentation are we willing to tolerate?
- **Coalescing policy:**
  - ***Immediate coalescing:*** coalesce each time `free` is called
  - ***Deferred coalescing:*** try to improve performance of `free` by deferring coalescing until needed

(Summary) Implicit Lists

- Implementation: very simple
- Allocate cost: linear time worst case
- Free cost: constant time worst case
  - even with coalescing
- Memory overhead
  - will depend on placement policy
  - First-fit, next-fit or best-fit
- Not used in practice for `malloc`/`free` because of linear-time allocation
- However, the concepts of splitting and boundary tag coalescing are general to *all* allocators

Question?