CSE251: System Programming

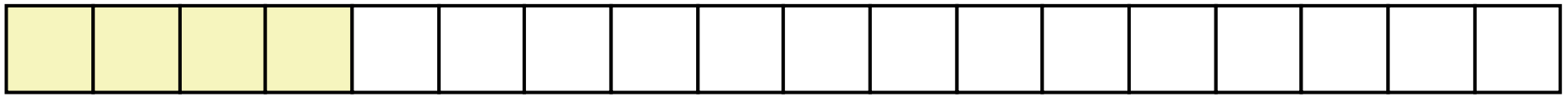
18. Dynamic Memory Allocation (2)

Seongil Wi

Department of Computer Science and Engineering

Recap: Fragmentation


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



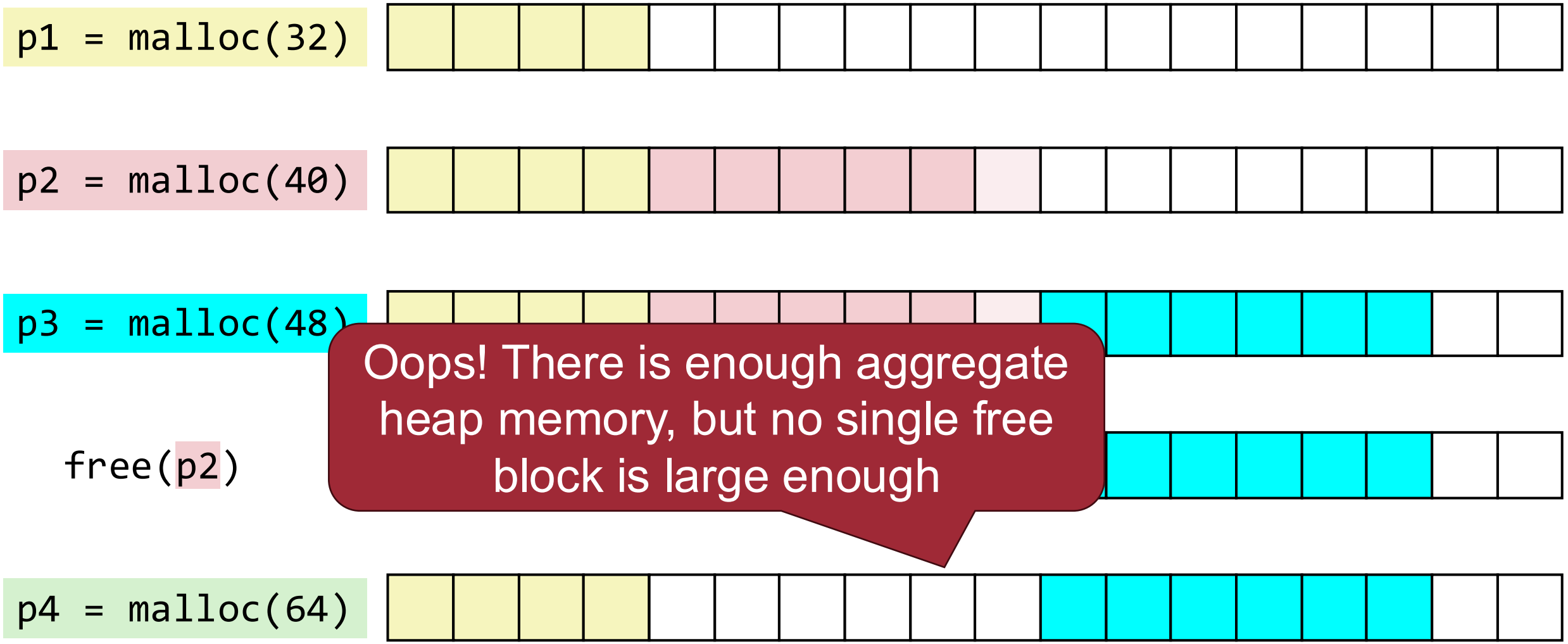
```
p2 = malloc(40)
```

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

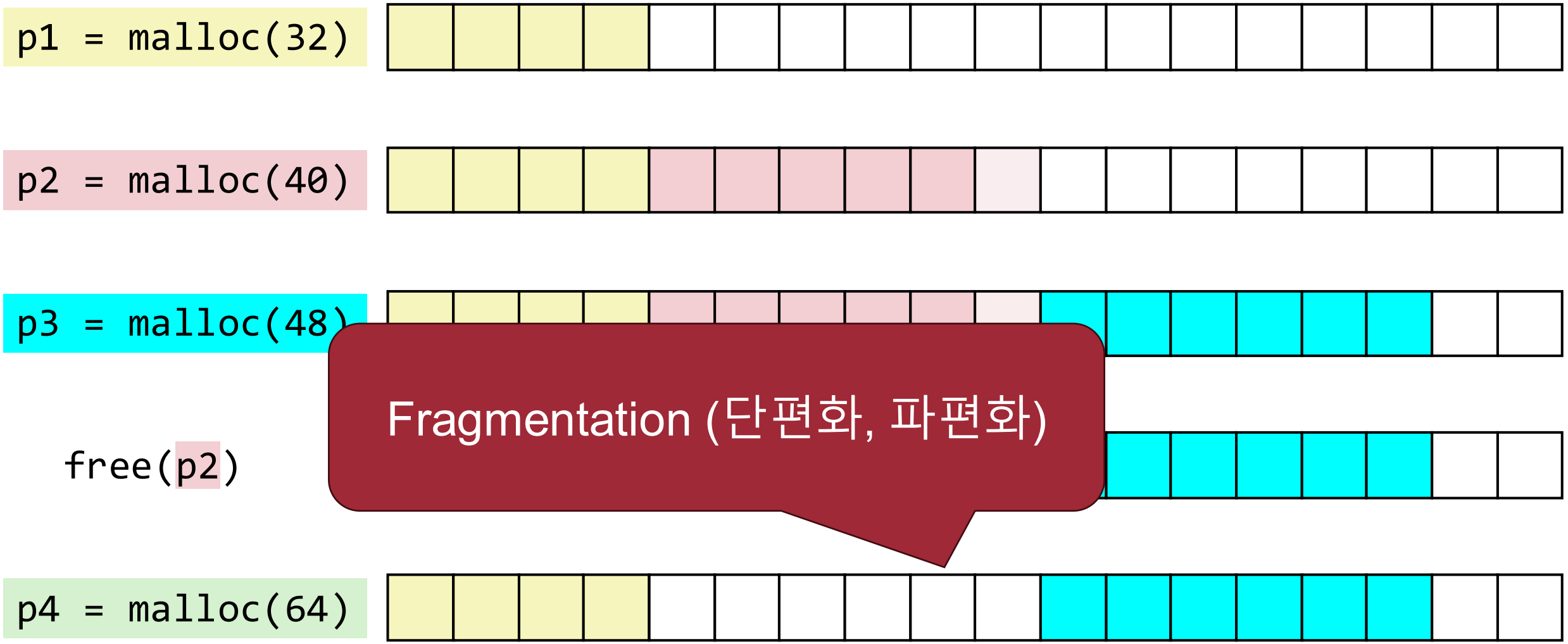
```
free(p2)
```

```
p4 = malloc(64)
```

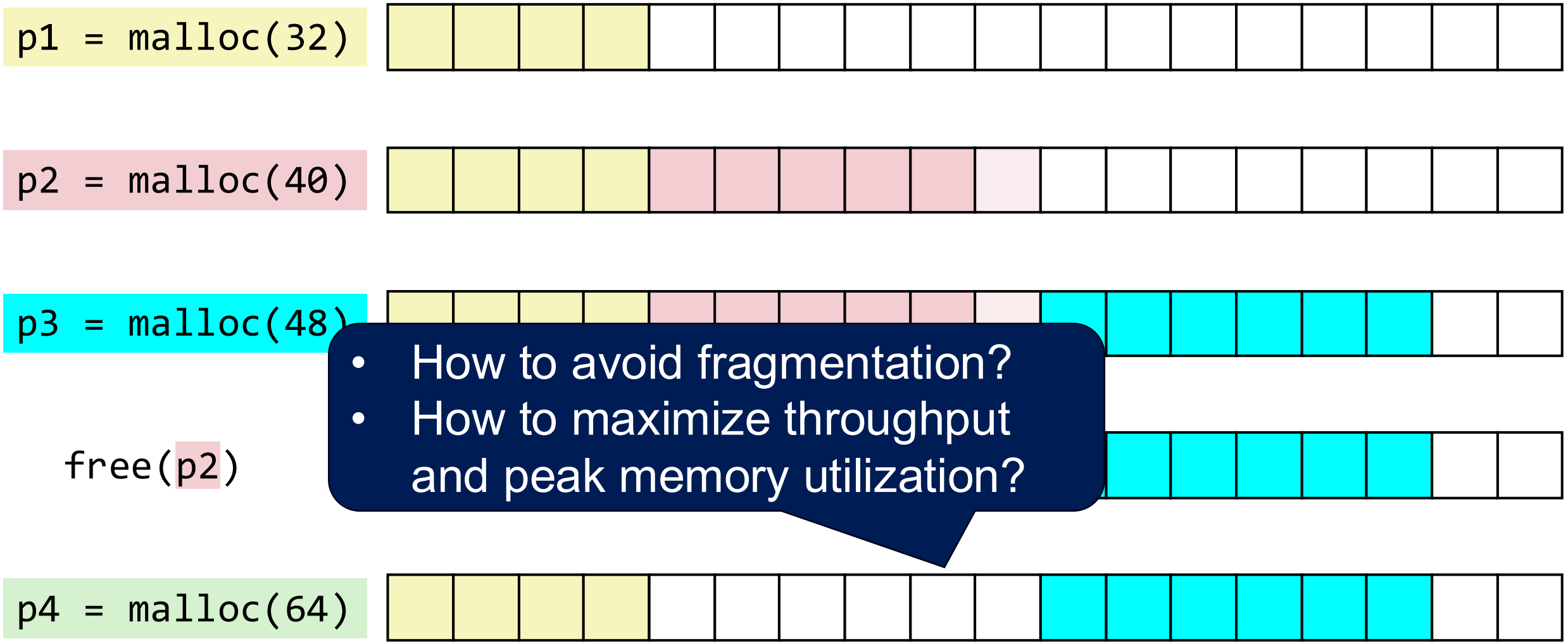
Recap: Fragmentation



Recap: Fragmentation



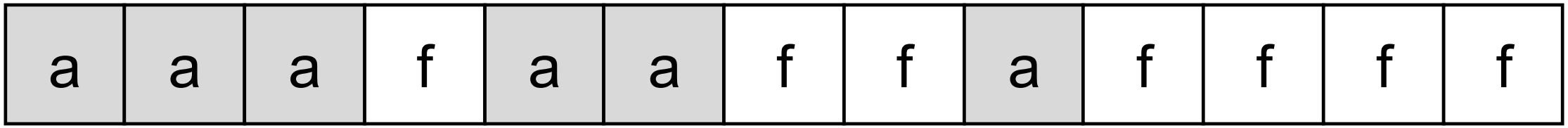
Motivating Question



Today's Topic: Managing Free Blocks

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

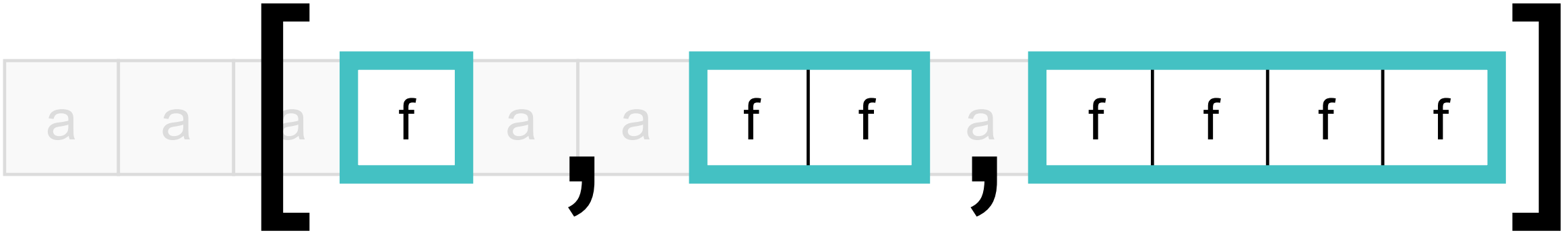
- We need to efficiently track free blocks and select one for allocation



Today's Topic: Managing Free Blocks

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

- We need to efficiently **track free blocks** and select one for allocation

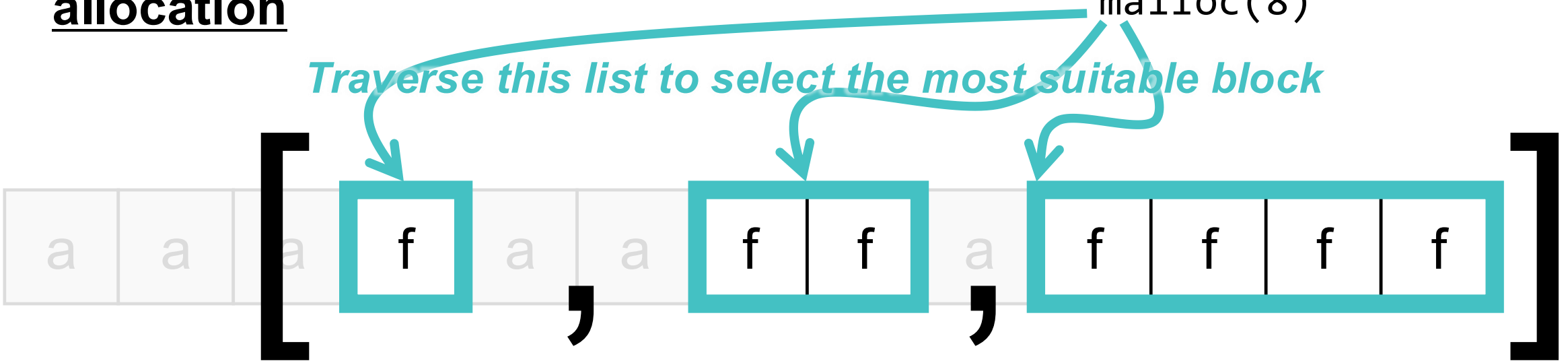


Keep track of free blocks as the list (e.g., linked list)

Today's Topic: Managing Free Blocks

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

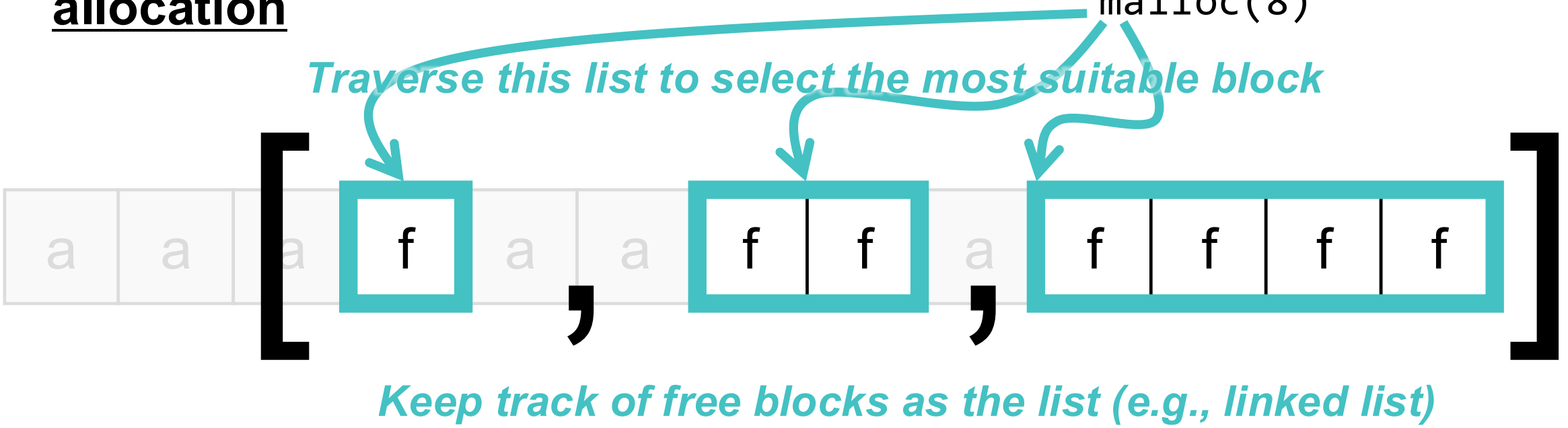
- We need to efficiently **track free blocks** and **select one for allocation**



Today's Topic: Managing Free Blocks

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

- We need to efficiently **track free blocks** and **select one for allocation**



How we implement this one?
What implementation issues should we consider?

Today's Topic

Allocator Implementation Issues

1. How do we know how much memory to free given just a pointer?

Today's Topic

Allocator Implementation Issues

1. How do we know how much memory to free given just a pointer?

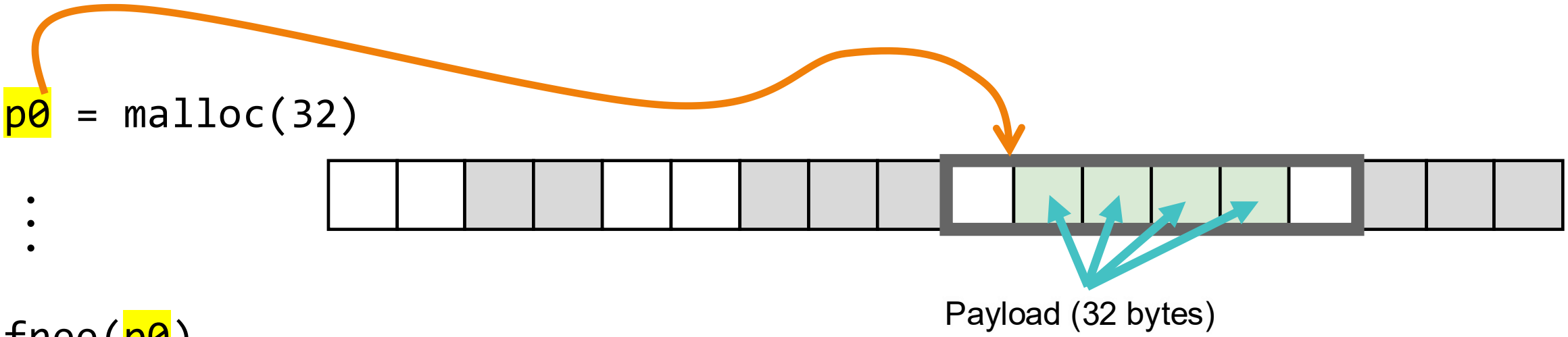
```
p0 = malloc(32)
 .
 .
 .
free(p0)
```

How does `free` know the size of the block pointed to by `p0`?

Introduction to Header Field

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

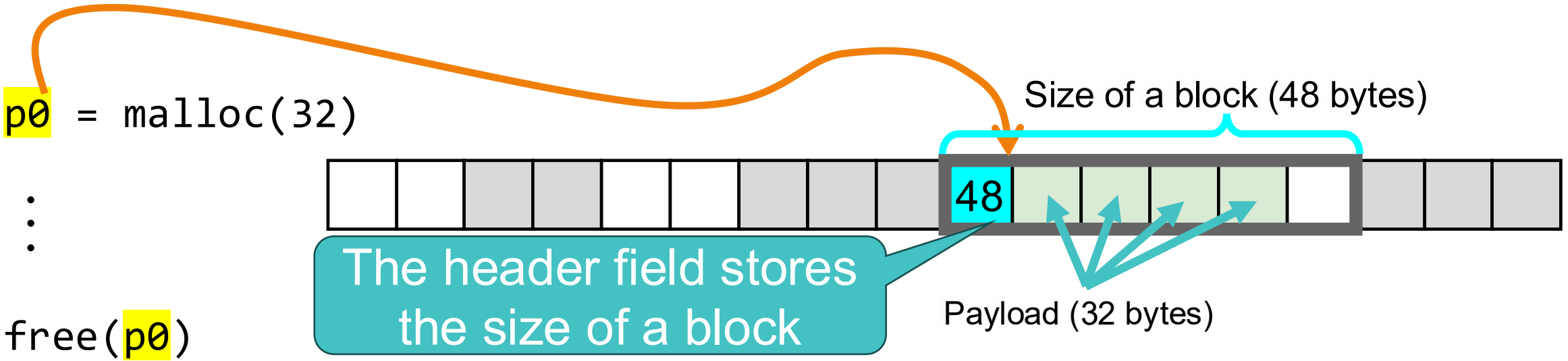
- Store the block size, in bytes, in the first word of each block
  - This word is often called the ***header field*** or ***header***
  - The block size includes the header



Introduction to Header Field

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

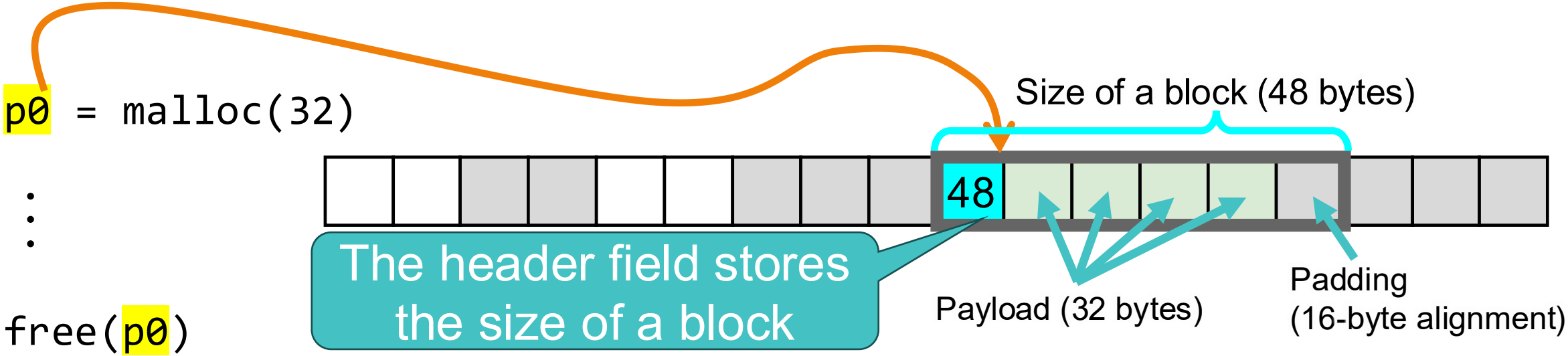
- Store the block size, in bytes, in the first word of each block
  - This word is often called the ***header field*** or ***header***
  - The block size includes the header



Introduction to Header Field

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

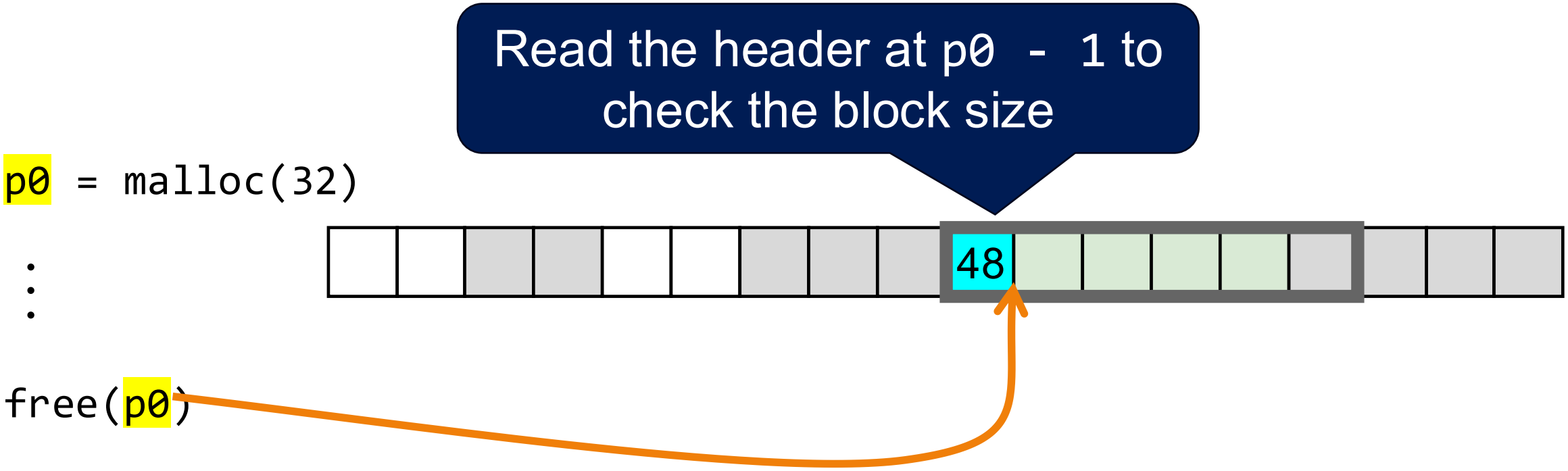
- Store the block size, in bytes, in the first word of each block
  - This word is often called the ***header field*** or ***header***
  - The block size includes the header



Introduction to Header Field

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

- Store the block size, in bytes, in the first word of each block
  - This word is often called the ***header field*** or ***header***
  - The block size includes the header

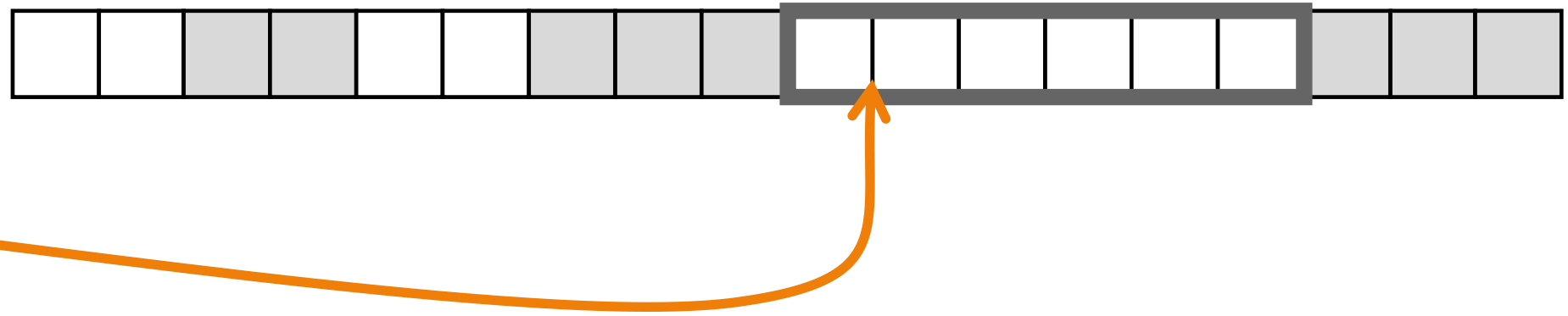


Introduction to Header Field

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

- Store the block size, in bytes, in the first word of each block
  - This word is often called the ***header field*** or ***header***
  - The block size includes the header

```
p0 = malloc(32)
.
.
.
free(p0)
```



Today's Topic

Allocator Implementation Issues

1. How do we know how much memory to free given just a pointer?

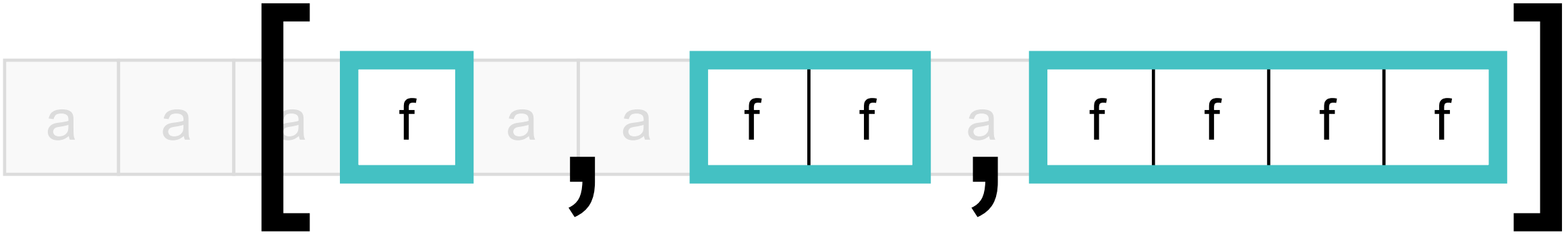
   → ***The header field stores the size of a block***

Today's Topic

Allocator Implementation Issues

1. How do we know how much memory to free given just a pointer?
   → ***The header field stores the size of a block***

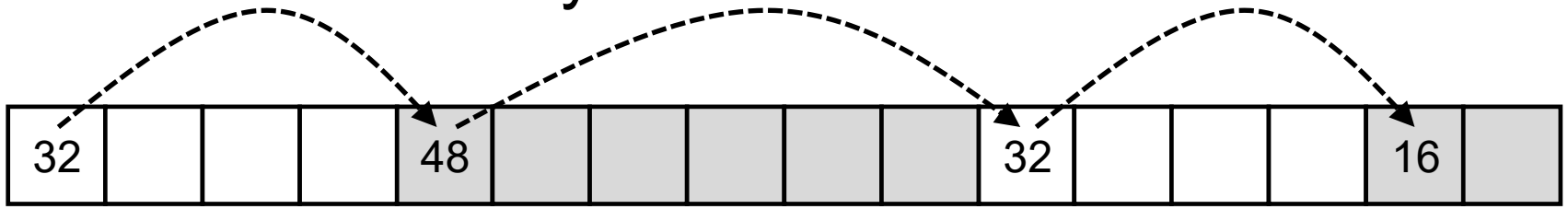
2. How do we keep track of the free blocks?



Keeping Track of Free Blocks

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

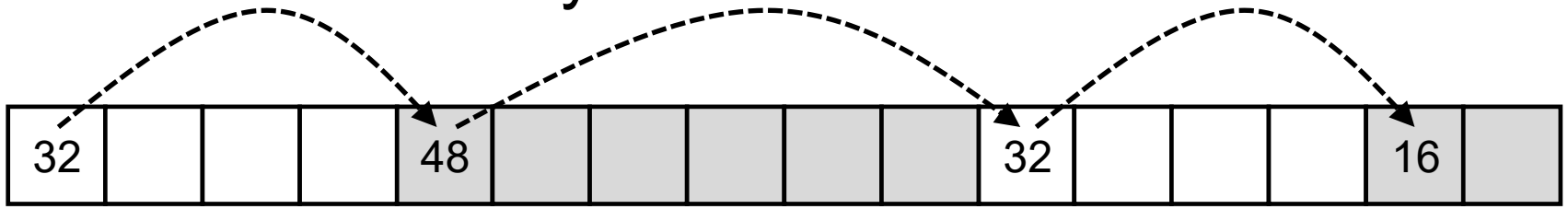
- Method #1: ***Implicit list*** using length. It links all blocks, regardless of whether they are allocated or free



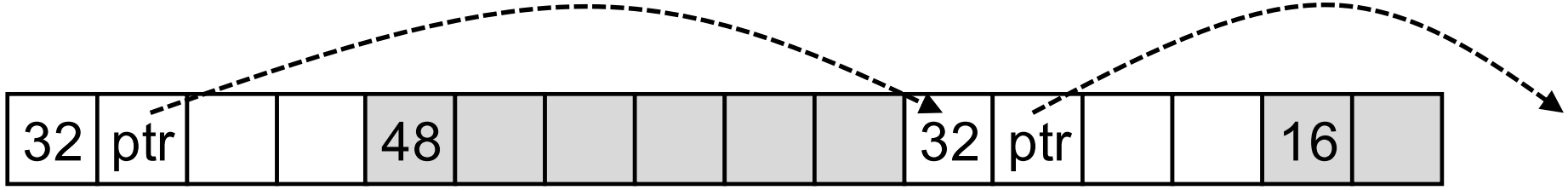
Keeping Track of Free Blocks

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

- Method #1: ***Implicit list*** using length. It links all blocks, regardless of whether they are allocated or free

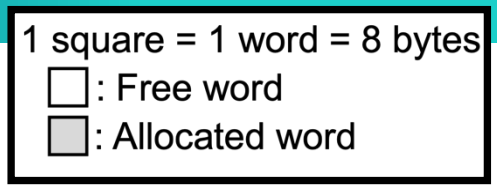


- Method #2: ***Explicit list*** among the free blocks using pointers

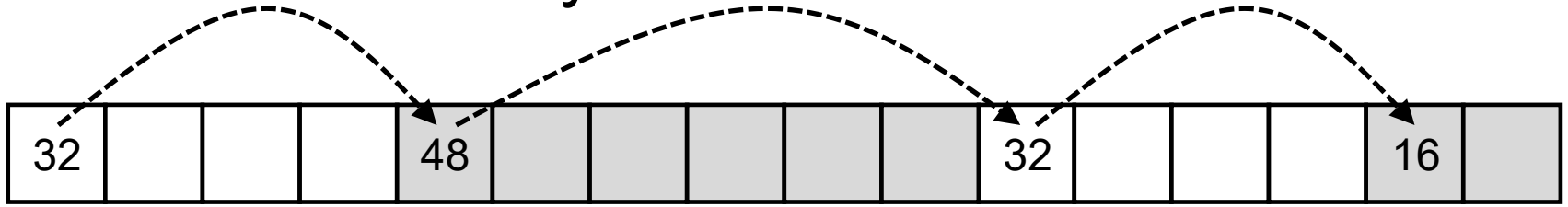


Q. In terms of throughput, which one is good?
Q. Which one is more likely to cause internal fragmentation?

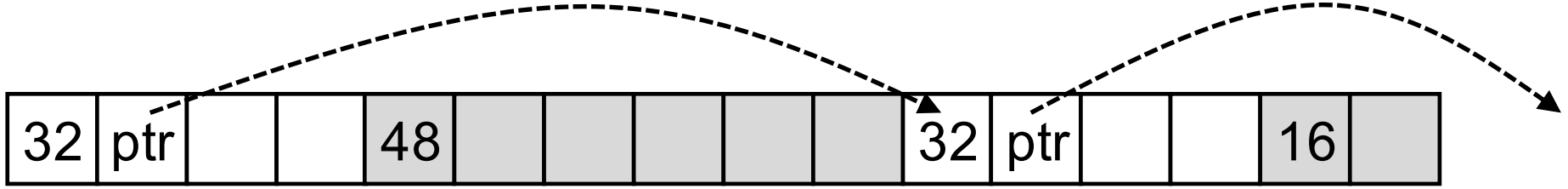
Keeping Track of Free Blocks



- Method #1: ***Implicit list*** using length. It links all blocks, regardless of whether they are allocated or free



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



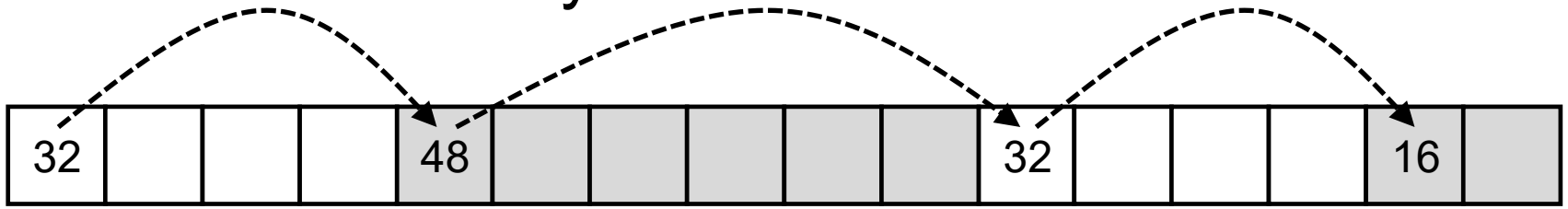
- Method #3: ***Segregated free list***
  - Different free lists for different size classes

Keeping Track of Free Blocks

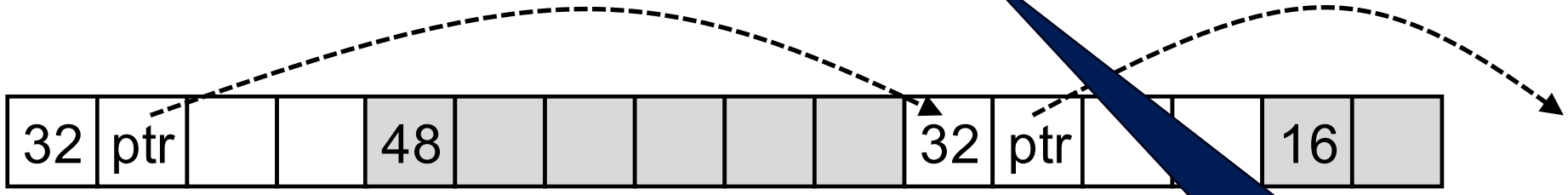
1 square = 1 word = 8 bytes
: Free word
: Allocated word

- Method #1: ***Implicit list*** using length. It links all blo
  regardless of whether they are allocated or free

Today, we focus on implicit lists



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



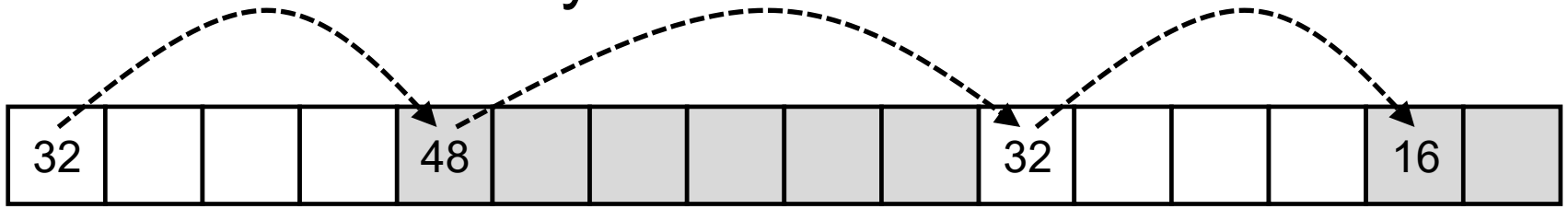
- Method #3: ***Segregated free list***
  - Different free lists for different size classes

We will cover these later

Overview of The Implicit Free List

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

- Method #1: ***Implicit list*** using length. It links all blocks, regardless of whether they are allocated or free

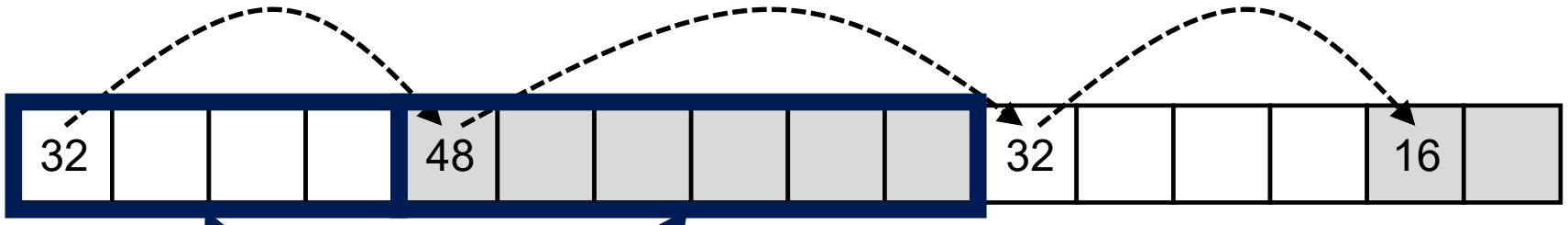


How to traverse this list?

```
block = start;
while (block != end) {
    ...
    block_size = p[0]
    offset = block_size / 8
    block = block + offset
}
```

Move to the next block

Overview of The Implicit Free List

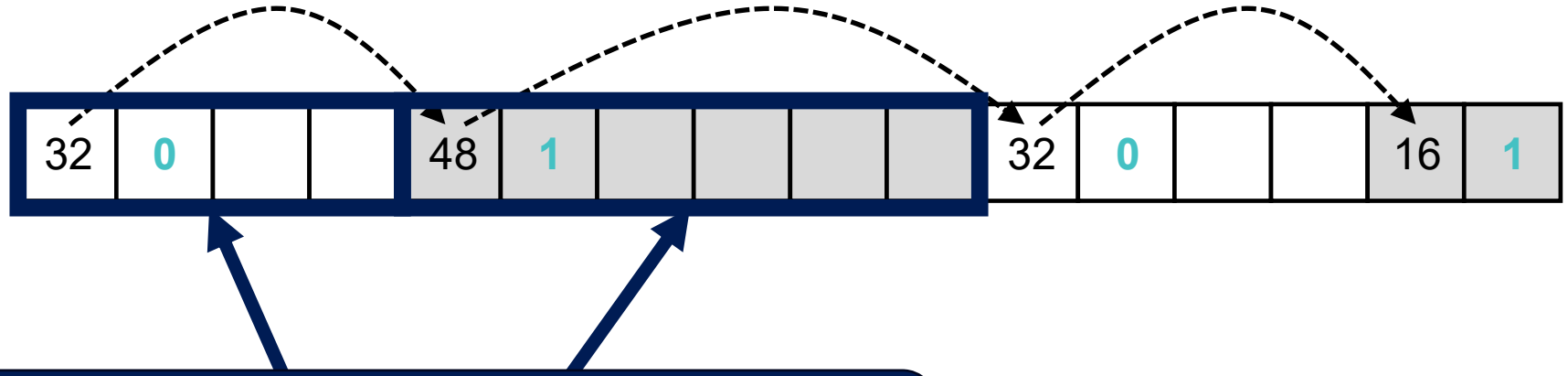


We need to know that whether the block is **allocated** or **free**

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

Overview of The Implicit Free List

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



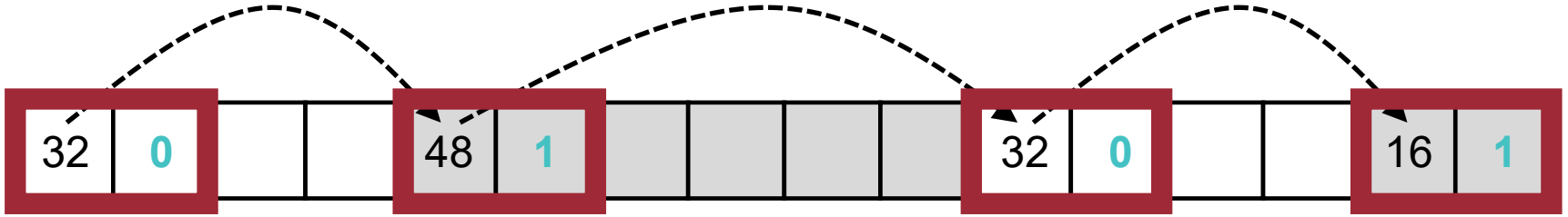
We need to know that whether the block is **allocated** or **free**

Introduce the tag field!

- **1**: Allocated block
- **0**: Free block

Overview of The Implicit Free List

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



We need to know that whether the block is **allocated** or **free**

Introduce the tag field!

- **1**: Allocated block
- **0**: Free block

Any problems?

For each block, we need both size and allocation status

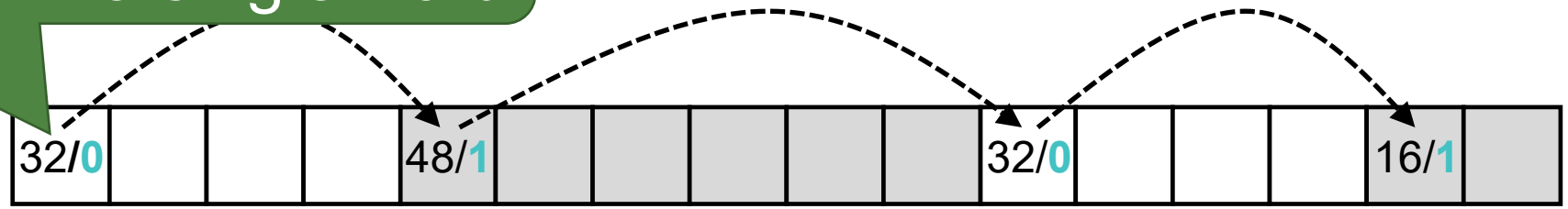
Memory consumption

(internal fragmentation – 2 words per block)!

Standard Trick

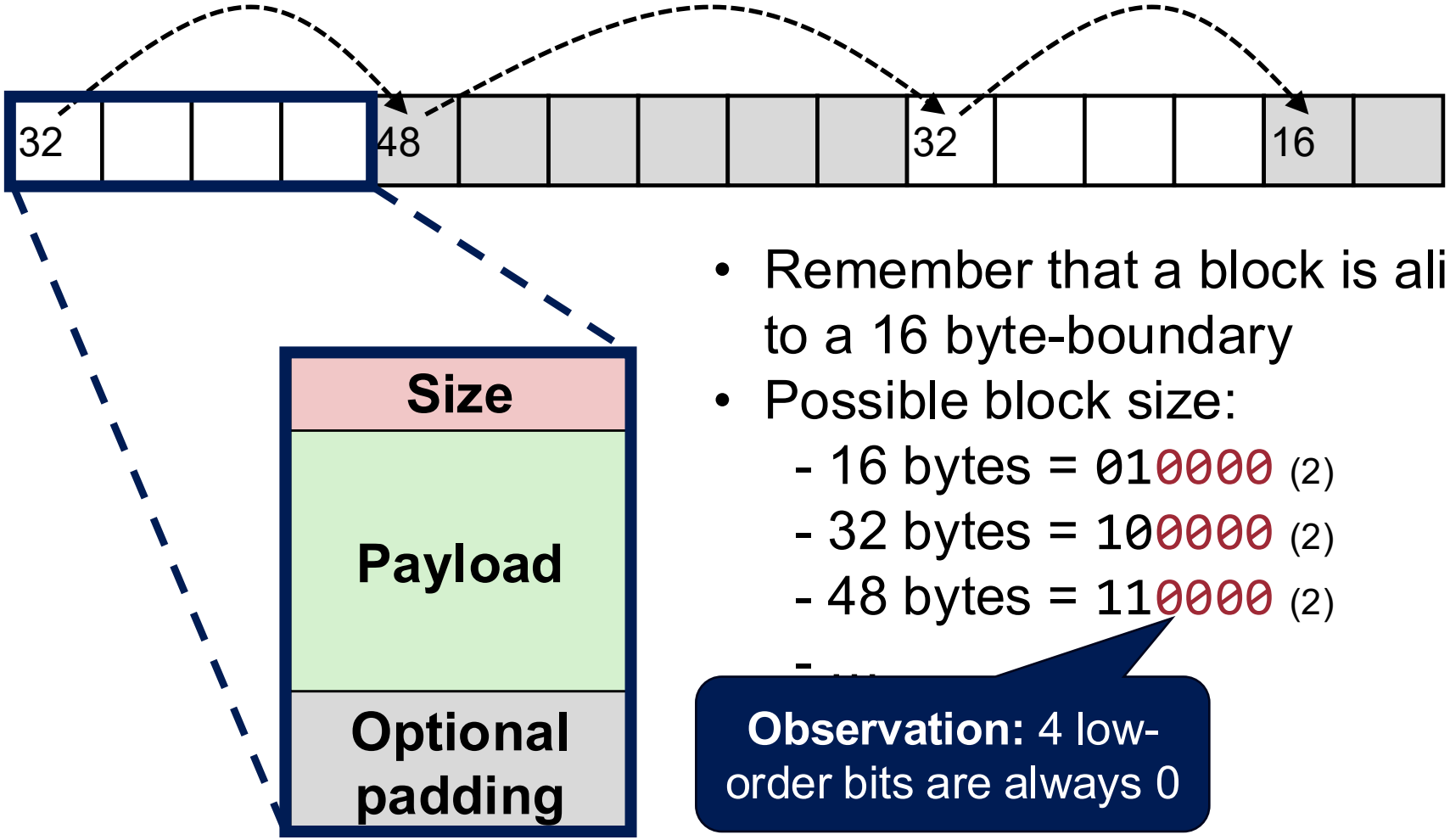
1 square = 1 word = 8 bytes
: Free word
: Allocated word

Solution: Express both meanings in a single word



How It Can Be Possible?

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

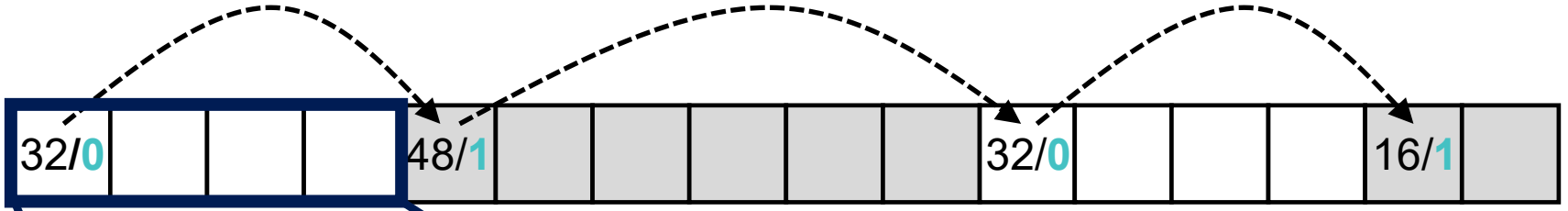


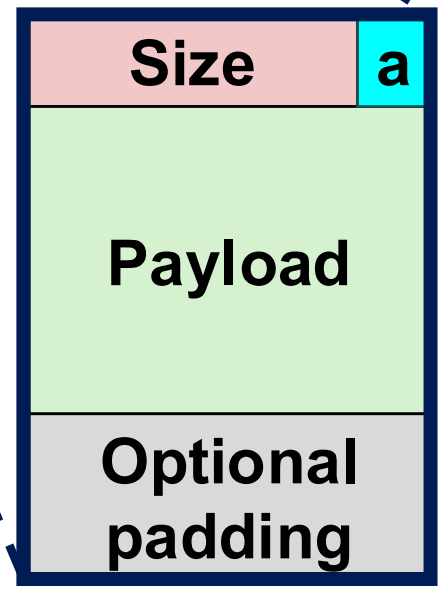
- Remember that a block is aligned to a 16 byte-boundary
- Possible block size:
  - 16 bytes = 010000 (2)
  - 32 bytes = 100000 (2)
  - 48 bytes = 110000 (2)
  - ...

Observation: 4 low-order bits are always 0

How It Can Be Possible?

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





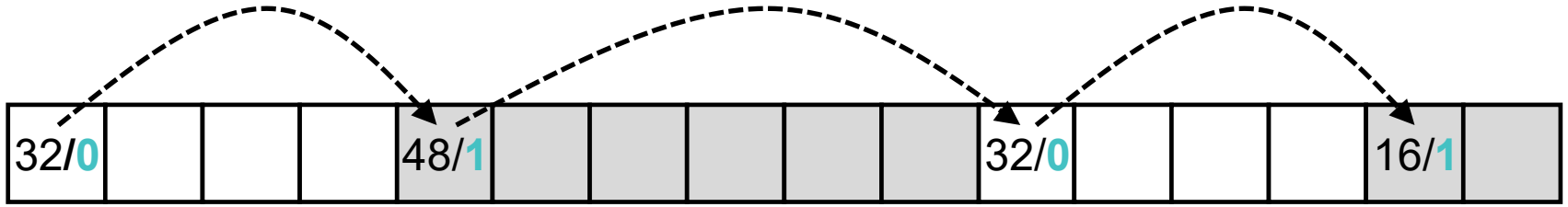
- Remember that a block is aligned to a 16 byte-boundary
- Possible block size:
  - 16 bytes = 010000 (2)
  - 32 bytes = 100000 (2)
  - 48 bytes = 110000 (2)
  - ...

Observation: 4 low-order bits are always 0

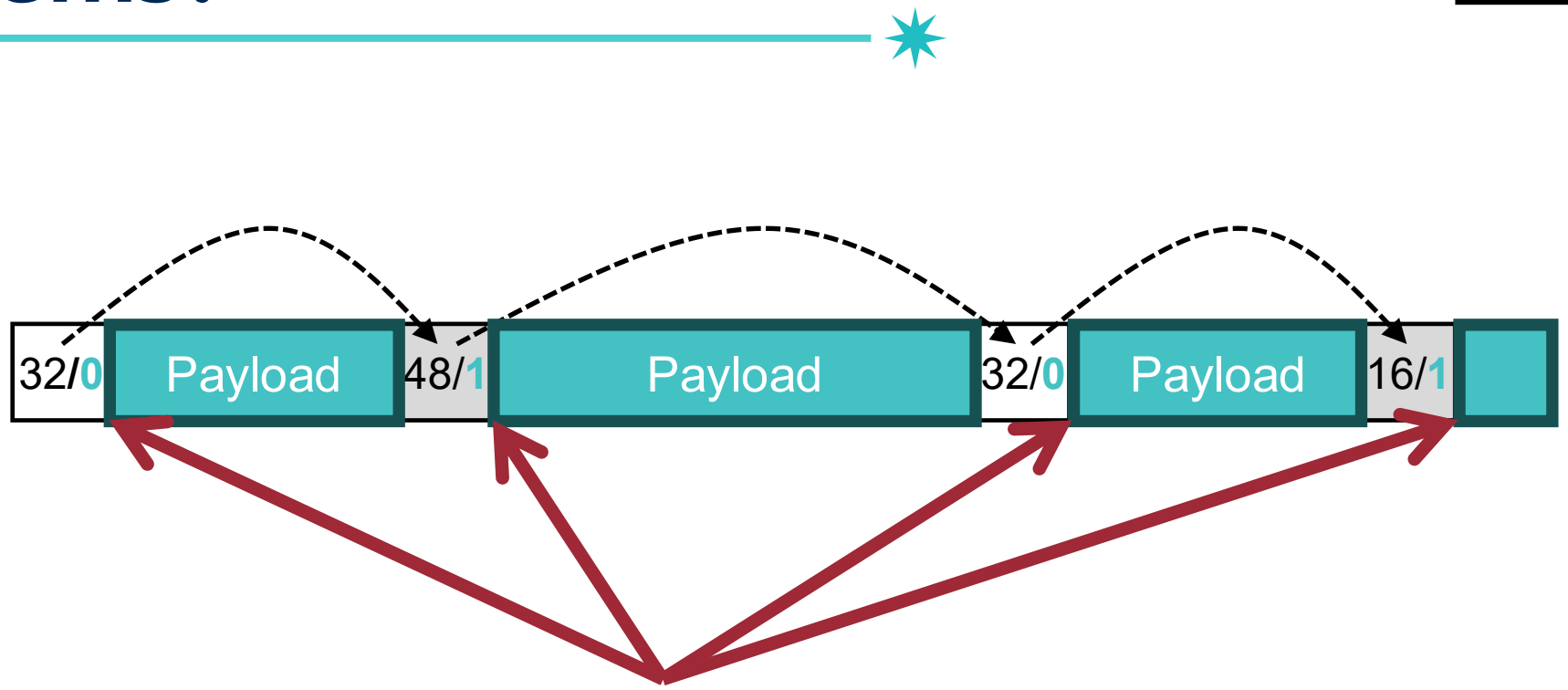
Idea: use this bit as 'flag'
- **1**: Allocated block
- **0**: Free block

Problems?

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



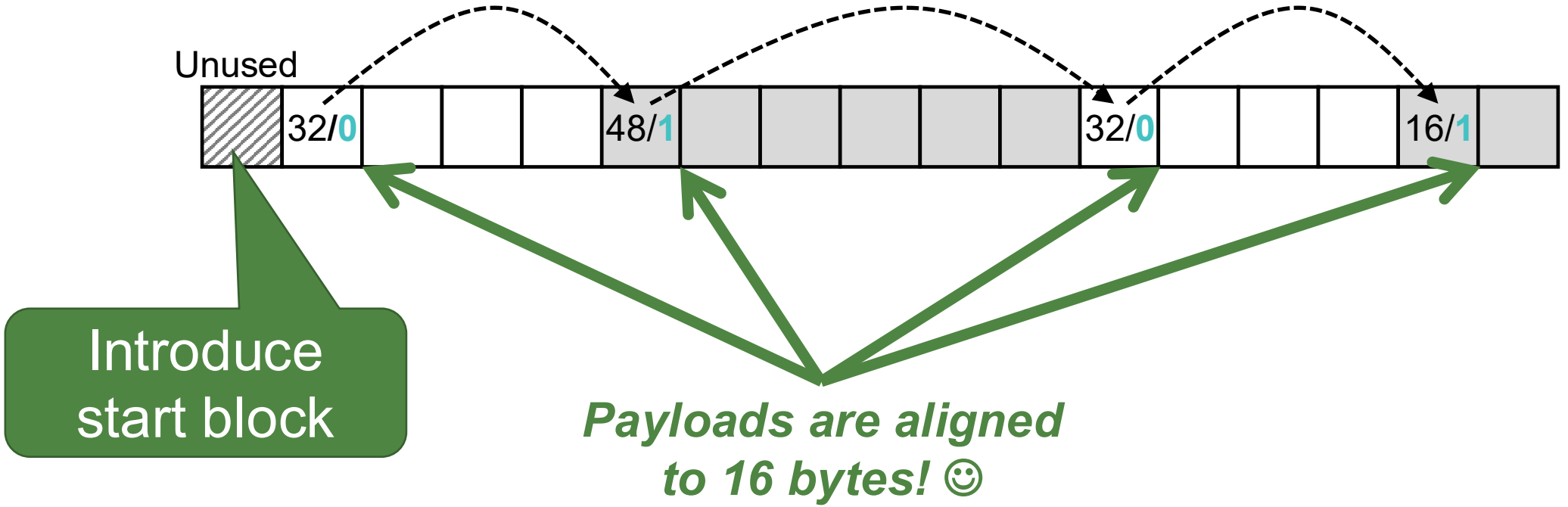
Problems?



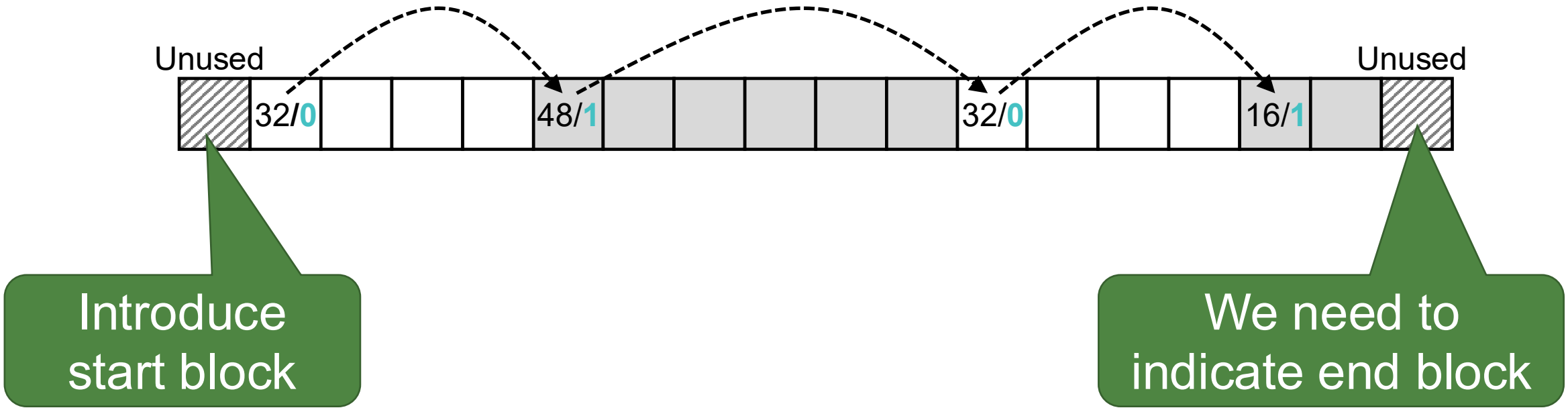
Address of the payload (malloc-returned pointer) is not aligned to 16 bytes ☹

On x86-64, malloc payloads are typically aligned to 16 bytes because this satisfies data-type alignment requirements and makes block management simpler

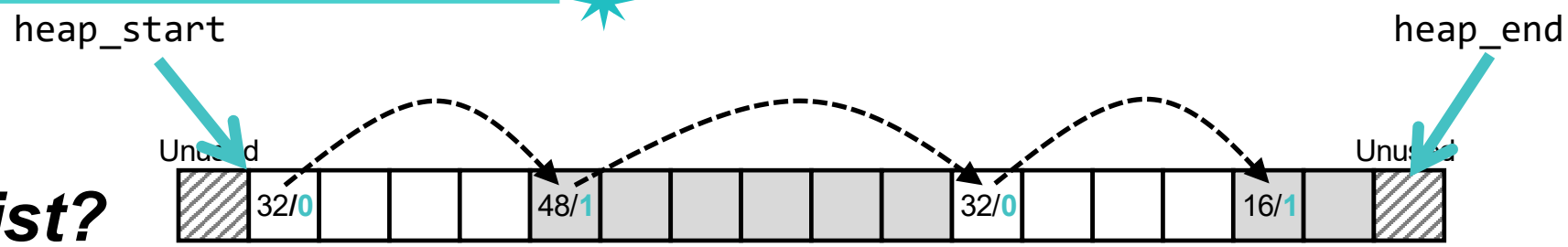
Deployment of Unused Blocks



Deployment of Unused Blocks



Final Version of The Implicit List



How to traverse this list?

```
block = heap_start;

while (block != heap_end) {
    header = block[0];

    block_size = header & ~0xF;
    allocated = header & 0x1;

    if (allocated == 0) {
        ...        // Free block!
    }

    block = block + block_size / 8;
}
```

Today's Topic

Allocator Implementation Issues

1. How do we know how much memory to free given just a pointer?

   → ***The header field stores the size of a block***

2. How do we keep track of the free blocks?

   → ***Managed by the linked list*** *(implicit list, explicit list, segregated free list)*

Today's Topic

Allocator Implementation Issues

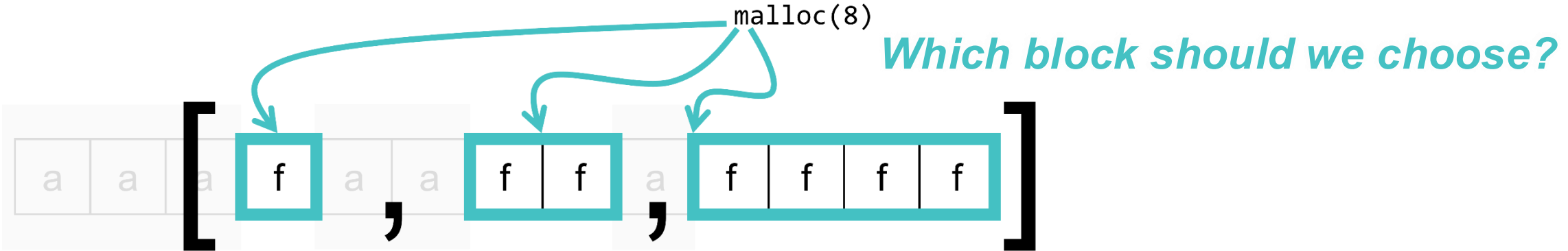
1. How do we know how much memory to free given just a pointer?

   → ***The header field stores the size of a block***

2. How do we keep track of the free blocks?

   → ***Managed by the linked list*** *(implicit list, explicit list, segregated free list)*

3. How do we select one of the free block for allocation?



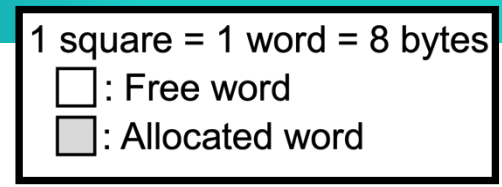
Placement Policy

- First fit
- Next fit
- Best fit

Placement Policy

- **First fit**
  - Search list from beginning, choose ***first*** free block that fits
- **Next fit**
- **Best fit**

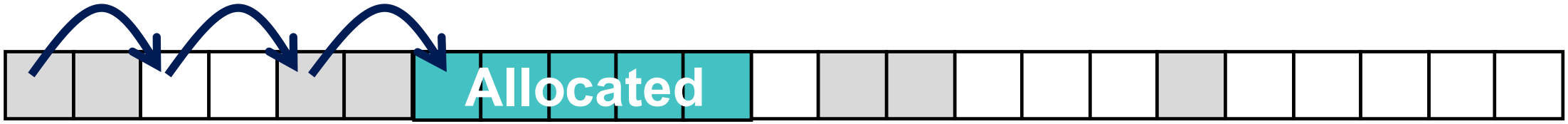
Placement Policy – First Fit

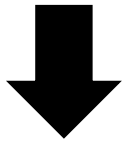


- Search list from beginning, choose ***first*** free block that fits

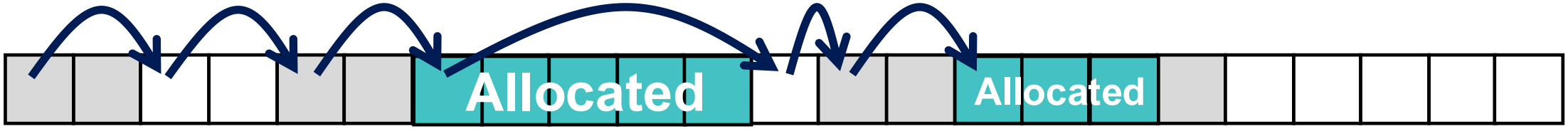
Assume a simplified heap (no headers, no end…)

`malloc(40)`
→ 5 words

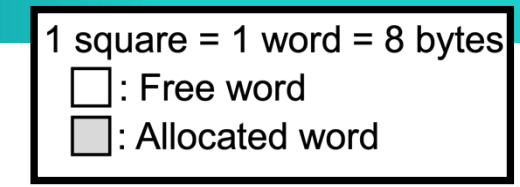




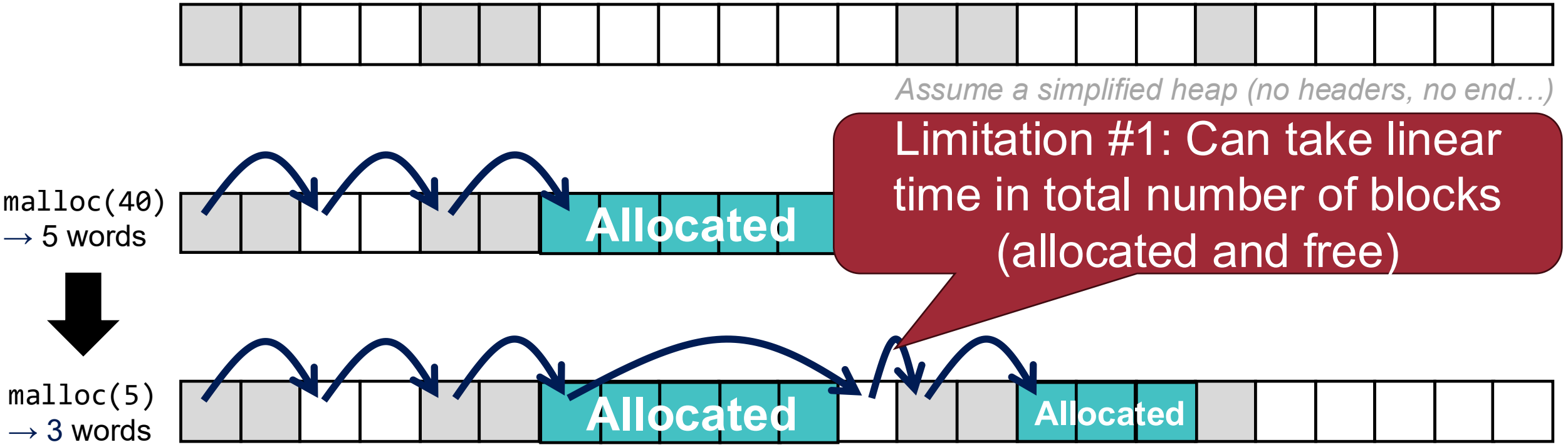
`malloc(5)`
→ 3 words



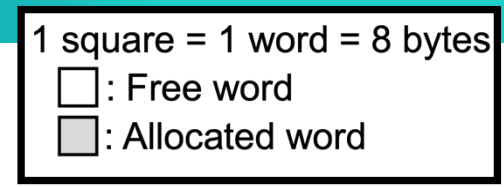
Placement Policy – First Fit



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



Placement Policy – First Fit



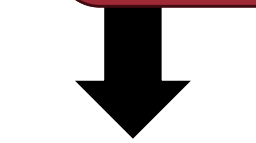
- Search list from beginning, choose ***first*** free block that fits

Assume a simplified heap (no headers, no end…)

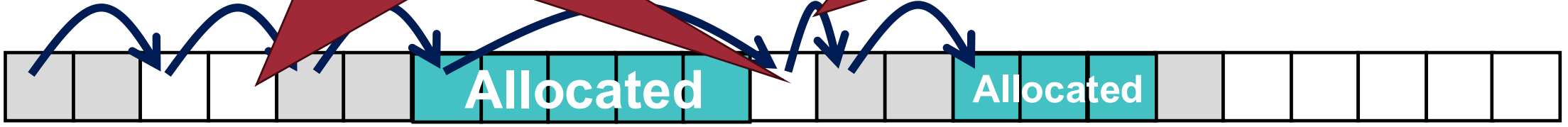
Limitation #2: Can cause many small free block at beginning

Limitation #1: Can take linear time in total number of blocks (allocated and free)

ma
→



malloc(5)
→ 3 words



Placement Policy

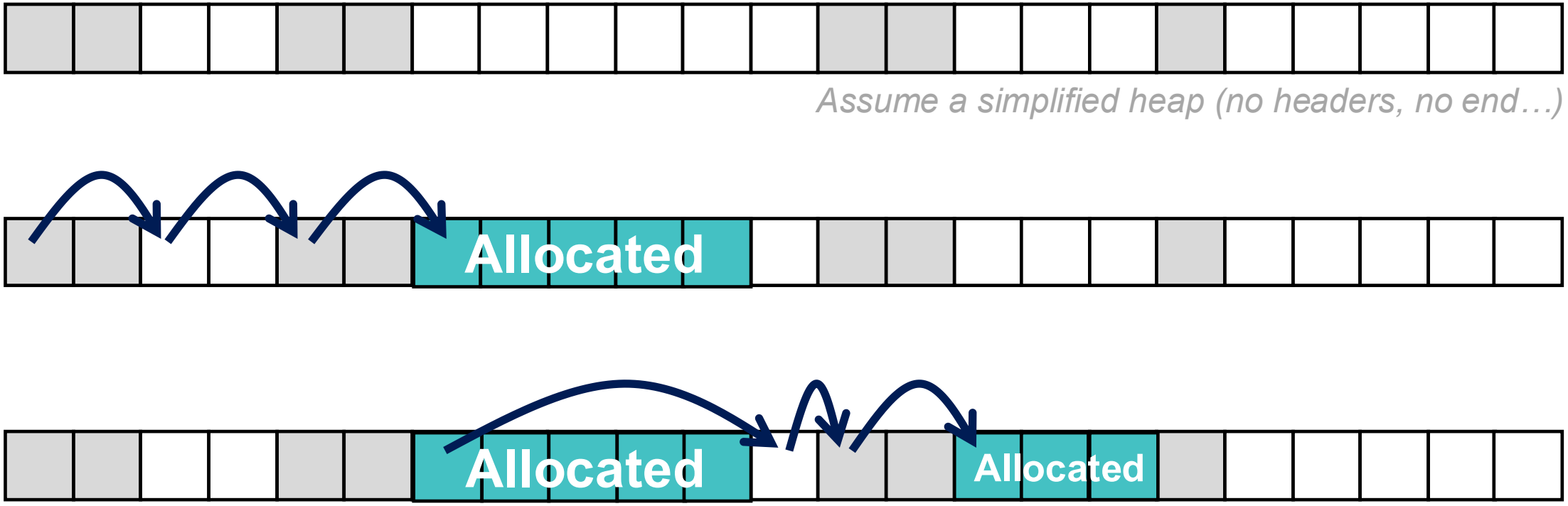
- **First fit**
  - Search list from beginning, choose ***first*** free block that fits
- **Next fit**
- **Best fit**

Placement Policy

- **First fit**
  - Search list from beginning, choose ***first*** free block that fits
- **Next fit**
  - Like first fit, but search list ***starting where previous search finished***
- **Best fit**

Placement Policy – Next Fit

- Like first fit, but search list ***starting where previous search finished***



Placement Policy – Next Fit

- Like first fit, but search list ***starting where previous search finished***

Assume a simplified heap (no headers, no end…)

Limitation: Some research suggests that fragmentation is worse

Advantage: Often be faster than first fit (avoids rescanning unhelpful blocks)

ma
→

malloc(5)
→ 3 words

Placement Policy

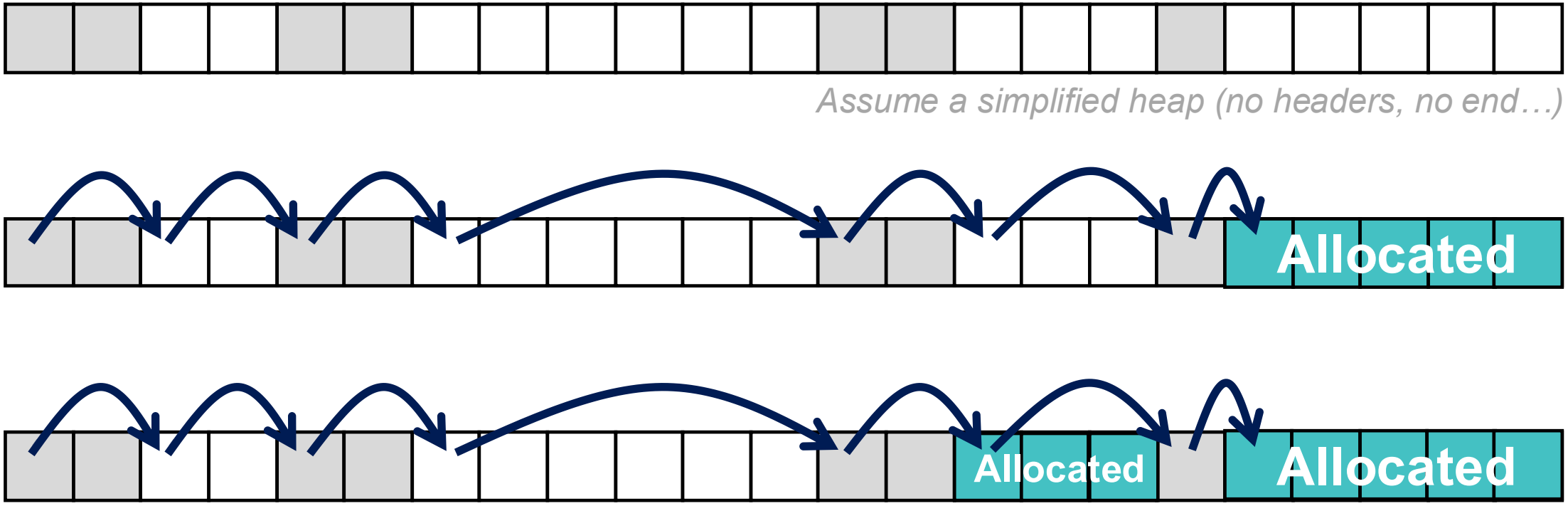
- **First fit**
  - Search list from beginning, choose ***first*** free block that fits
- **Next fit**
  - Like first fit, but search list starting where previous search finished
- **Best fit**

Placement Policy

- **First fit**
  - Search list from beginning, choose ***first*** free block that fits
- **Next fit**
  - Like first fit, but search list starting where previous search finished
- **Best fit**
  - Search the list, choose the ***best*** free block: fits, with fewest bytes left over

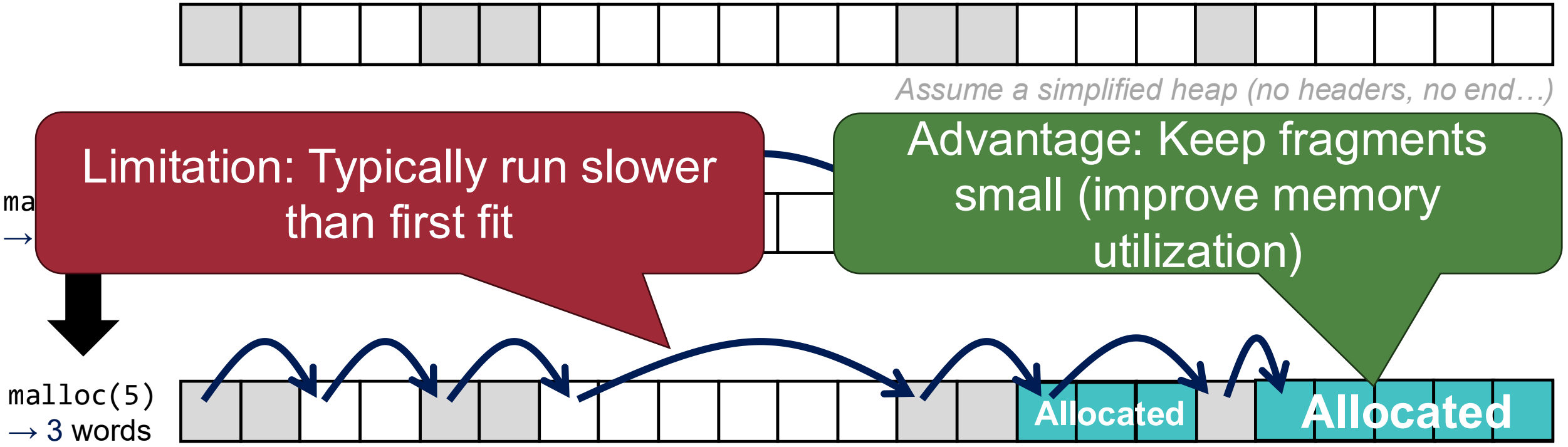
Placement Policy – Best Fit

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



Placement Policy – Best Fit

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



Placement Policy

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

Which one is better?

Each strategy has its own trade-off.

There is no universally best placement policy.

The final choice should be made based on research-based results.

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

Summary

1. How do we know how much memory to free given just a pointer?

   → ***The header field stores the size of a block***

2. How do we keep track of the free blocks?

   → ***Managed by the linked list*** *(implicit list, explicit list, segregated free list)*

3. How do we select one of the free block for allocation?

   → ***Placement policy*** *(first fit, next fit, best fit)*

Question?