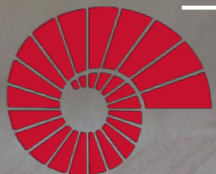# COMP201

Computer Systems & Programming

Lecture #24 – Managing The Heap

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Spring 2021

Recap

- Static Linking
- Symbol Resolution
- Relocation
- Static Libraries
- Shared Libraries
- Case study: Library interpositioning

Plan for Today

- The heap so far

- What is a heap allocator?

- Heap allocator requirements and goals

- Method 0: Bump Allocator

- Method 1: Implicit Free List Allocator

- Method 2: Explicit Free List Allocator

Disclaimer: Slides for this lecture were borrowed from

—Nick Troccoli's Stanford CS107 class

COMP201 Topic 8: How do the core `malloc/realloc/free` memory-allocation operations work?

How do `malloc`/`realloc`/`free` work?

Pulling together all our COMP201 topics this semester:

- Testing

- Efficiency

- Bit-level manipulation

- Memory management

- Pointers

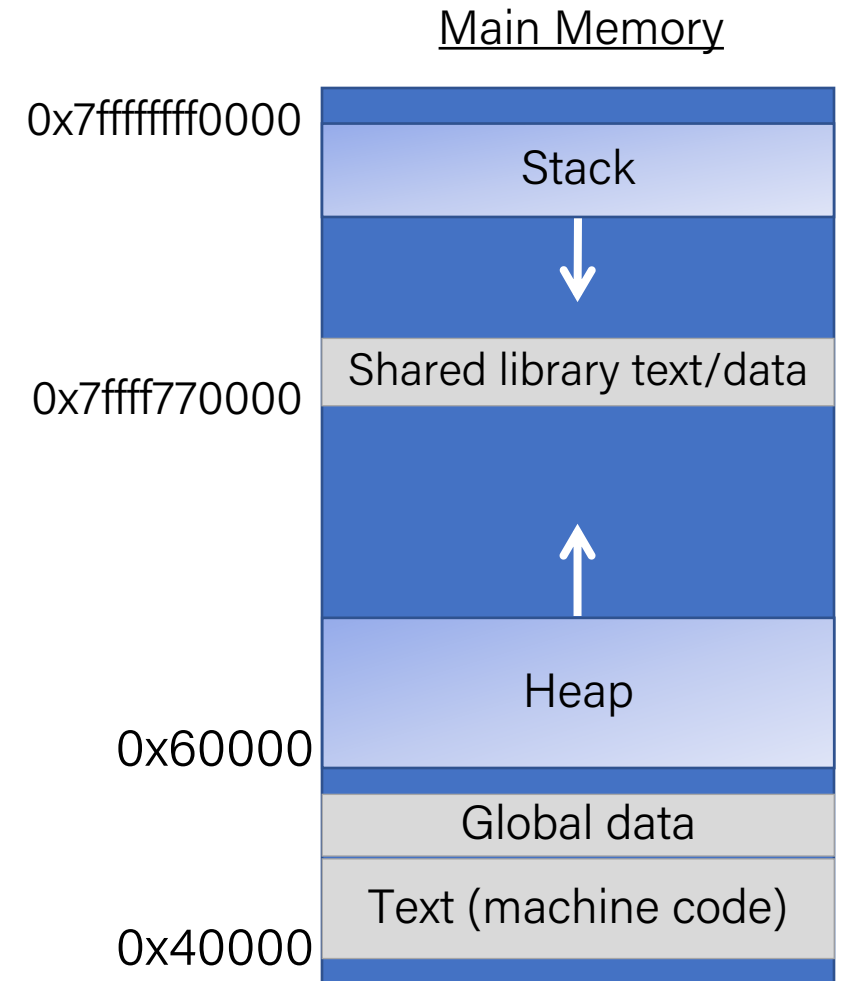- Generics

- Assembly

- And more...

Learning Goals

- Learn the restrictions, goals and assumptions of a heap allocator

- Understand the conflicting goals of utilization and throughput

- Learn about different ways to implement a heap allocator

Lecture Plan

- The heap so far
- What is a heap allocator?
- Heap allocator requirements and goals
- Method 0: Bump Allocator
- Method 1: Implicit Free List Allocator
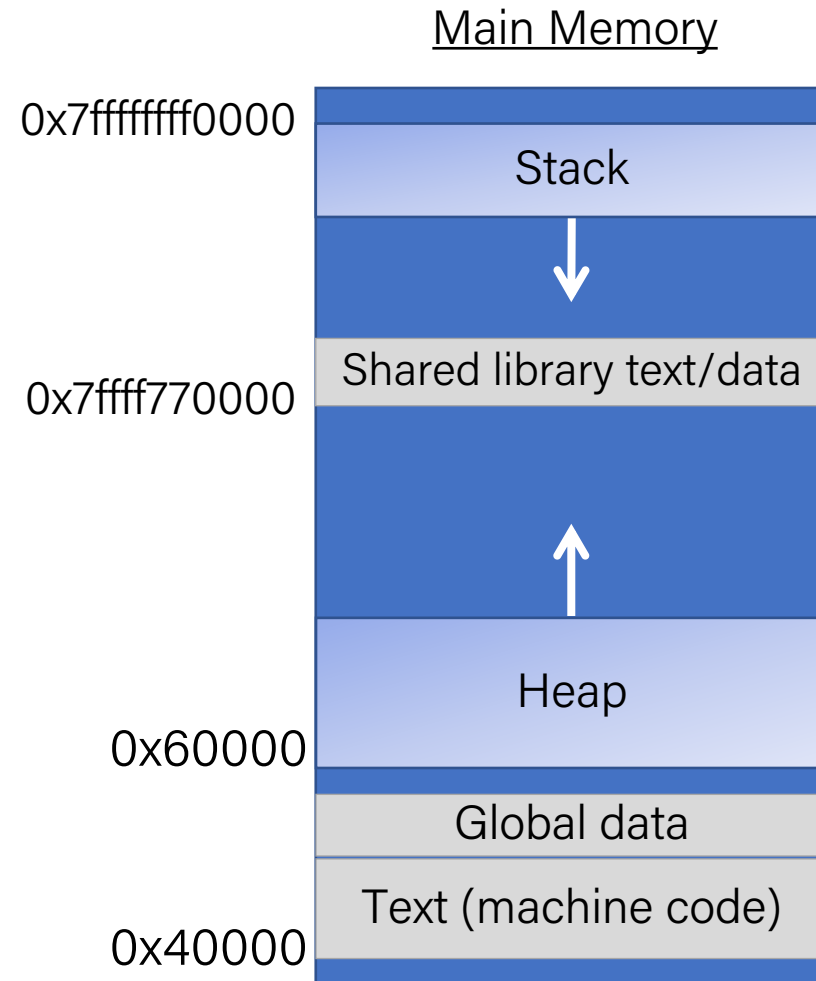- Method 2: Explicit Free List Allocator

Running a program

- **Creates new process**

- **Sets up address space/segments**

- **Read executable file, load instructions, global data**
  Mapped from file into gray segments

- **Libraries loaded on demand**

- **Set up stack**
  Reserve stack segment, init %rsp, call main

- **`malloc` written in C, will init self on use**
  Asks OS for large memory region, parcels out to service requests

Main Memory

0x7ffffffff0000

Stack

Shared library text/data

0x7ffff770000

Heap

0x60000

Global data

Text (machine code)

0x40000

8

The Stack Revisited

Main Memory

0x7ffffffff0000
Stack

Shared library text/data
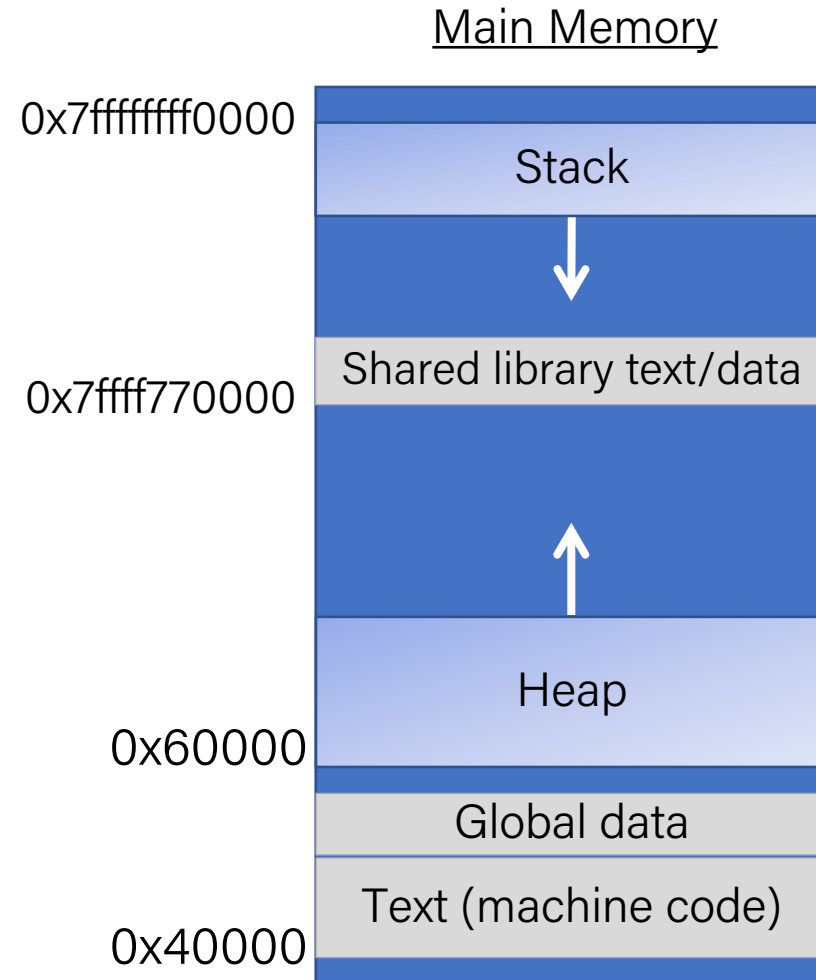0x7ffff770000

Heap
0x60000

Global data

Text (machine code)
0x40000

Stack memory "goes away" after function call ends.

Automatically managed at compile-time by gcc

Today: The Heap

Main Memory

0x7fffffff0000

Stack

Shared library text/data

0x7ffff770000

Heap

0x60000

Global data

Text (machine code)

0x40000

Heap memory persists until caller indicates it no longer needs it.

Managed by C standard library functions (`malloc`, `realloc`, `free`)

This lecture: How does heap management work?

Lecture Plan

- The heap so far

- **What is a heap allocator?**

- Heap allocator requirements and goals

- Method 0: Bump Allocator

- Method 1: Implicit Free List Allocator

- Method 2: Explicit Free List Allocator

Your role so far: Client

`void *malloc(size_t size);`

Returns a pointer to a block of heap memory of at least size bytes, or NULL if an error occurred.

`void free(void *ptr);`

Frees the heap-allocated block starting at the specified address.

`void *realloc(void *ptr, size_t size);`

Changes the size of the heap-allocated block starting at the specified address to be the new specified size. Returns the address of the new, larger allocated memory region.

Your role now: Heap Hotel Concierge

(aka **Heap Allocator**)

What is a heap allocator?

- A heap allocator is a set of functions that fulfills requests for heap memory.

- On initialization, a heap allocator is provided the starting address and size of a large contiguous block of memory (the heap).

0x10   0x11   0x12   0x13   0x14   0x15   0x16   0x17   0x18   0x19

AVAILABLE

What is a heap allocator?

- A heap allocator is a set of functions that fulfills requests for heap memory.

- On initialization, a heap allocator is provided the starting address and size of a large contiguous block of memory (the heap).

- A heap allocator must manage this memory as clients request or no l~~onger use~~ of it.

Request 1: Hi! May I please have 2 bytes of heap memory?

Allocator: Sure, I've given you address 0x10.

0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19
AVAILABLE									

What is a heap allocator?

- A heap allocator is a set of functions that fulfills requests for heap memory.

- On initialization, a heap allocator is provided the starting address and size of a large contiguous block of memory (the heap).

- A heap allocator must manage this memory as clients request or no longer use of it.

Request 1: Hi! May I please have 2 bytes of heap memory?

Allocator: Sure, I've given you address 0x10.

0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19
FOR REQUEST 1		AVAILABLE							

What is a heap allocator?

- A heap allocator is a set of functions that fulfills requests for heap memory.

- On initialization, a heap allocator is provided the starting address and size of a large contiguous block of memory (the heap).

- A heap allocator must manage this memory as clients request or no longer need of it.

Request 2: Howdy! May I please have 3 bytes of heap memory?

Allocator: Sure, I've given you address 0x12.

0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19
FOR REQUEST 1		AVAILABLE							

What is a heap allocator?

- A heap allocator is a set of functions that fulfills requests for heap memory.

- On initialization, a heap allocator is provided the starting address and size of a large contiguous block of memory (the heap).

- A heap allocator must manage this memory as clients request or no longer make use of it.

Request 2: Howdy! May I please have 3 bytes of heap memory?

Allocator: Sure, I've given you address 0x12.

0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19
FOR REQUEST 1		FOR REQUEST 2			AVAILABLE				

What is a heap allocator?

- A heap allocator is a set of functions that fulfills requests for heap memory.

- On initialization, a heap allocator is provided the starting address and size of a large contiguous block of memory (the heap).

- A heap allocator must manage this memory as clients request or no longer make use of it.

Request 1: I'm done with the memory I requested. Thank you!

Allocator: Thanks. Have a good day!

| 0x10 | 0x11 | 0x12 | 0x13 | 0x14 | 0x15 | 0x16 | 0x17 | 0x18 | 0x19 |

FOR REQUEST 1	FOR REQUEST 2	AVAILABLE

What is a heap allocator?

- A heap allocator is a set of functions that fulfills requests for heap memory.

- On initialization, a heap allocator is provided the starting address and size of a large contiguous block of memory (the heap).

- A heap allocator must manage this memory as clients request or no longer make use of it.

Request 1: I'm done with the memory I requested. Thank you!

Allocator: Thanks. Have a good day!

0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19
AVAILABLE		FOR REQUEST 2			AVAILABLE				

What is a heap allocator?

- A heap allocator is a set of functions that fulfills requests for heap memory.

- On initialization, a heap allocator is provided the starting address and size of a large contiguous block of memory (the heap).

- A heap allocator must manage this memory as clients request or no ~~longer need~~ of it.

Request 3: Hello there! I'd like to request 2 bytes of heap memory, please.

Allocator: Sure thing. I've given you address 0x10.

0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19
AVAILABLE		FOR REQUEST 2			AVAILABLE				

What is a heap allocator?

- A heap allocator is a set of functions that fulfills requests for heap memory.

- On initialization, a heap allocator is provided the starting address and size of a large contiguous block of memory (the heap).

- A heap allocator must manage this memory as clients request or no longer need it.

Request 3: Hello there! I'd like to request 2 bytes of heap memory, please.

Allocator: Sure thing. I've given you address 0x10.

0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19
FOR REQUEST 3		FOR REQUEST 2			AVAILABLE				

What is a heap allocator?

- A heap allocator is a set of functions that fulfills requests for heap memory.

- On initialization, a heap allocator is provided the starting address and size of a large contiguous block of memory (the heap).

- A heap allocator must manage this memory as clients request or no longer need it.

Request 3: Hi again! I'd like to request the region of memory at 0x10 be reallocated to 4 bytes.

Allocator: Sure thing. I've given you address 0x15.

0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19

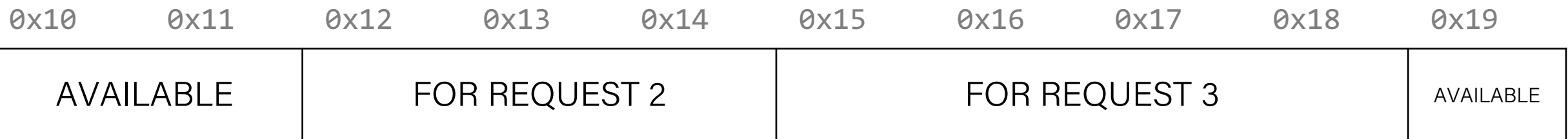| FOR REQUEST 3	FOR REQUEST 2	AVAILABLE

What is a heap allocator?

- A heap allocator is a set of functions that fulfills requests for heap memory.

- On initialization, a heap allocator is provided the starting address and size of a large contiguous block of memory (the heap).

- A heap allocator must manage this memory as clients request or no longer need it.

Request 3: Hi again! I'd like to request the region of memory at 0x10 be reallocated to 4 bytes.

Allocator: Sure thing. I've given you address 0x15.

0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19

AVAILABLE	FOR REQUEST 2	FOR REQUEST 3	AVAILABLE

Lecture Plan

- The heap so far

- What is a heap allocator?

- **Heap allocator requirements and goals**

- Method 0: Bump Allocator

- Method 1: Implicit Free List Allocator

- Method 2: Explicit Free List Allocator

Heap Allocator Functions

```
void *malloc(size_t size);

void free(void *ptr);

void *realloc(void *ptr, size_t size);
```

Heap Allocator Requirements

A heap allocator must…

1. Handle arbitrary request sequences of allocations and frees
2. Keep track of which memory is allocated and which is available
3. Decide which memory to provide to fulfill an allocation request
4. Immediately respond to requests without delay

Heap Allocator Requirements

A heap allocator must…

1. **Handle arbitrary request sequences of allocations and frees**

2. Keep track of which memory is allocated and which is available

3. Decide which memory to provide to fulfill an allocation request

4. Immediately respond to requests without delay

A heap allocator cannot assume anything about the order of allocation and free requests, or even that every allocation request is accompanied by a matching free request.

Heap Allocator Requirements

A heap allocator must...

1. Handle arbitrary request sequences of allocations and frees
2. **Keep track of which memory is allocated and which is available**
3. Decide which memory to provide to fulfill an allocation request
4. Immediately respond to requests without delay

A heap allocator marks memory regions as **allocated** or **available**. It must remember which is which to properly provide memory to clients.

Heap Allocator Requirements

A heap allocator must...

1. Handle arbitrary request sequences of allocations and frees
2. Keep track of which memory is allocated and which is available
3. **Decide which memory to provide to fulfill an allocation request**
4. Immediately respond to requests without delay

A heap allocator may have options for which memory to use to fulfill an allocation request. It must decide this based on a variety of factors.

Heap Allocator Requirements

A heap allocator must…

1. Handle arbitrary request sequences of allocations and frees
2. Keep track of which memory is allocated and which is available
3. Decide which memory to provide to fulfill an allocation request
4. **Immediately respond to requests without delay**

A heap allocator must respond immediately to allocation requests and should not e.g. prioritize or reorder certain requests to improve performance.

Heap Allocator Requirements

A heap allocator must…

1. Handle arbitrary request sequences of allocations and frees

2. Keep track of which memory is allocated and which is available

3. Decide which memory to provide to fulfill an allocation request

4. Immediately respond to requests without delay

5. **Return addresses that are 8-byte-aligned (must be multiples of 8).**
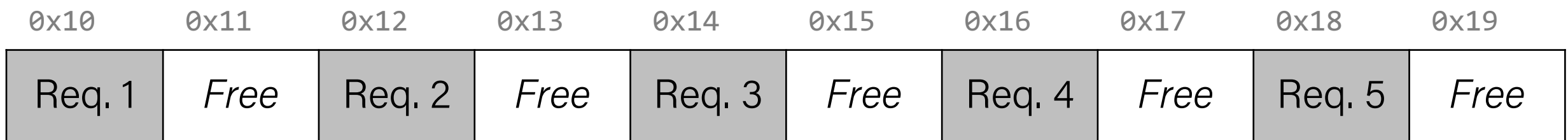
Heap Allocator Goals

- Goal 1: Maximize **throughput**, or the number of requests completed per unit time. This means minimizing the average time to satisfy a request.

- Goal 2: Maximize memory **utilization**, or how efficiently we make use of the limited heap memory to satisfy requests.

Utilization

- The primary cause of poor utilization is **fragmentation**. **Fragmentation** occurs when otherwise unused memory is not available to satisfy allocation requests.

- In this example, there is enough aggregate free memory to satisfy the request, but no single free block is large enough to handle the request.

- **In general:** we want the largest address used to be as low as possible.

Request 6: Hi! May I please have 4 bytes of heap memory?

Allocator: I'm sorry, I don't have a 4 byte block available...

0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19
Req. 1	*Free*	Req. 2	*Free*	Req. 3	*Free*	Req. 4	*Free*	Req. 5	*Free*

Utilization

Question: what if we shifted these blocks down to make more space? Can we do this?

A. YES, great idea!

B. YES, it can be done, but not a good idea for some reason (e.g. not efficient use of time)

C. NO, it can't be done!

| 0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19
Req. 1	Req. 2	Req. 3	Req. 4	Req. 5	*Free*				

Utilization

Question: what if we shifted these blocks down to make more space? Can we do this?

- **No** - we have already guaranteed these addresses to the client. We cannot move allocated memory around, since this will mean the client will now have incorrect pointers to their memory!

0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19

Req. 1	Req. 2	Req. 3	Req. 4	Req. 5	*Free*
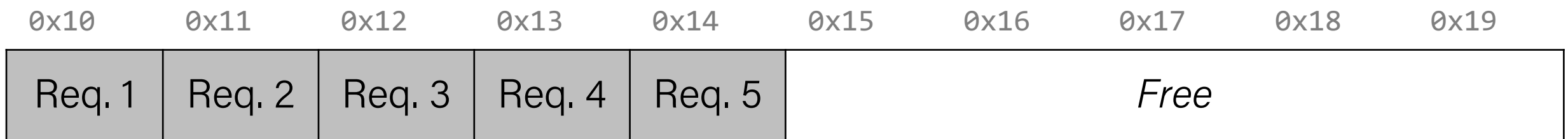
Fragmentation

- **Internal Fragmentation:** an allocated block is larger than what is needed (e.g. due to minimum block size)

- **External Fragmentation:** no single block is large enough to satisfy an allocation request, even though enough aggregate free memory is available

Heap Allocator Goals

- Goal 1: Maximize **throughput**, or the number of requests completed per unit time. This means minimizing the average time to satisfy a request.

- Goal 2: Maximize memory **utilization**, or how efficiently we make use of the limited heap memory to satisfy requests.

These are seemingly conflicting goals – for instance, it may take longer to better plan out heap memory use for each request.

Heap allocators must find an appropriate balance between these two goals!

Heap Allocator Goals

- Goal 1: Maximize **throughput**, or the number of requests completed per unit time. This means minimizing the average time to satisfy a request.

- Goal 2: Maximize memory **utilization**, or how efficiently we make use of the limited heap memory to satisfy requests.

Other desirable goals:
Locality ("similar" blocks allocated close in space)
Robust (handle client errors)
**Ease of implementation/maintenance**

Lecture Plan

- The heap so far

- What is a heap allocator?

- Heap allocator requirements and goals

- **Method 0: Bump Allocator**

- Method 1: Implicit Free List Allocator

- Method 2: Explicit Free List Allocator

Bump Allocator

Let's say we want to entirely prioritize throughput, and do not care about utilization at all. This means we do not care about reusing memory. How could we do this?

1. Utilization

😱
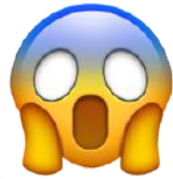
Never reuses memory

2. Throughput

⭐

Ultra fast, short routines

Bump Allocator

- A **bump allocator** is a heap allocator design that simply allocates the next available memory address upon an allocate request and does nothing on a free request.

- Throughput: each `malloc` and `free` execute only a handful of instructions:
  - It is easy to find the next location to use
  - Free does nothing!

- Utilization: we use each memory block at most once. No freeing at all, so no memory is ever reused. ☹

Bump Allocator

```
void *a = malloc(8);

void *b = malloc(4);

void *c = malloc(24);

free(b);

void *d = malloc(8);
```

0x10	0x14	0x18	0x1c	0x20	0x24	0x28	0x2c	0x30	0x34

AVAILABLE

Bump Allocator

void *a = malloc(8);

void *b = malloc(4);

void *c = malloc(24);

free(b);

void *d = malloc(8);

Variable	Value
a	0x10

0x10	0x14	0x18	0x1c	0x20	0x24	0x28	0x2c	0x30	0x34

a	AVAILABLE

Bump Allocator

```
void *a = malloc(8);
void *b = malloc(4);
void *c = malloc(24);
free(b);
void *d = malloc(8);
```

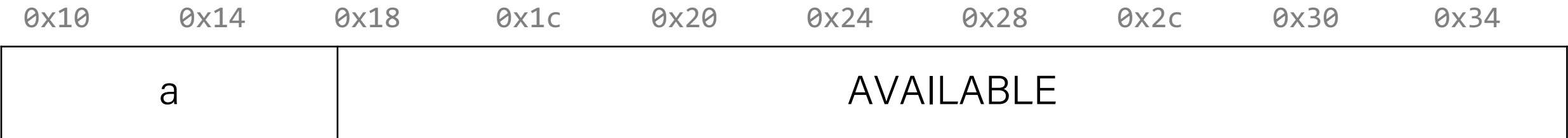| Variable	Value
a	0x10
b	0x18

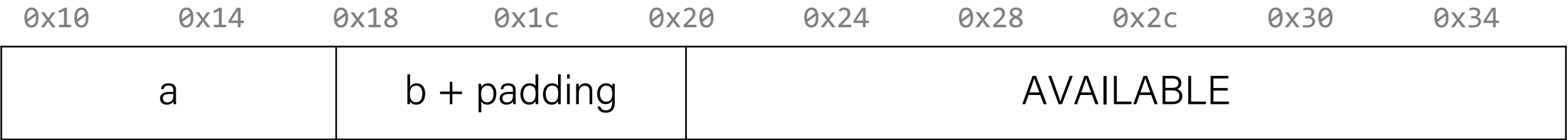| 0x10	0x14	0x18	0x1c	0x20	0x24	0x28	0x2c	0x30	0x34

a	b + padding	AVAILABLE

Bump Allocator

```
void *a = malloc(8);

void *b = malloc(4);

void *c = malloc(24);
free(b);

void *d = malloc(8);
```

Variable	Value
a	0x10
b	0x18
c	0x20

0x10   0x14   0x18   0x1c   0x20   0x24   0x28   0x2c   0x30   0x34

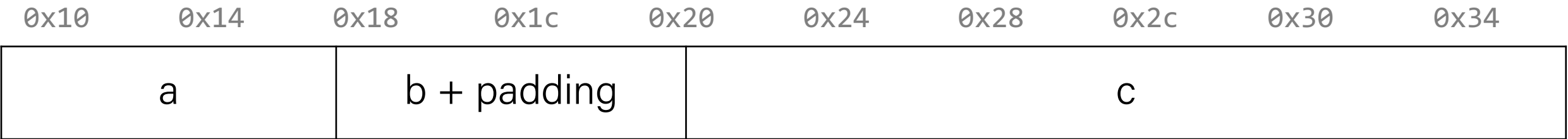| a	b + padding	c

Bump Allocator

```
void *a = malloc(8);
void *b = malloc(4);
void *c = malloc(24);
free(b);
void *d = malloc(8);
```

Variable	Value
a	0x10
b	0x18
c	0x20

0x10	0x14	0x18	0x1c	0x20	0x24	0x28	0x2c	0x30	0x34

a	b + padding	c

Bump Allocator

```
void *a = malloc(8);
void *b = malloc(4);
void *c = malloc(24);
free(b);
void *d = malloc(8);
```

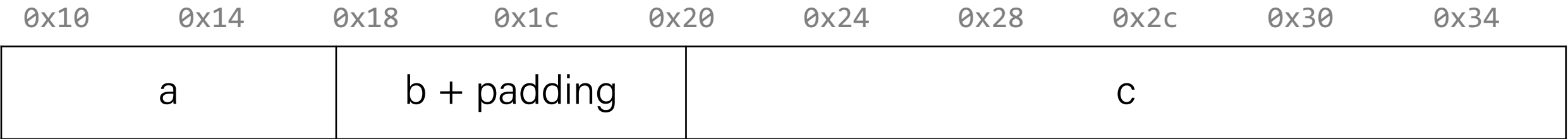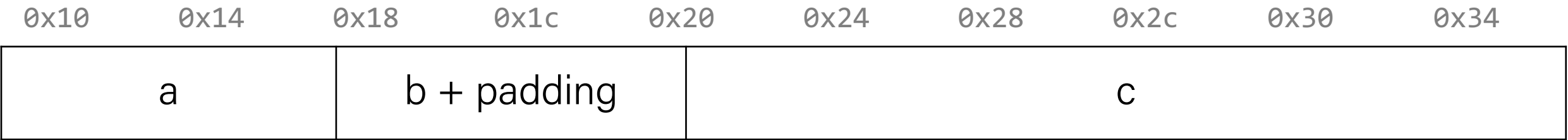| Variable	Value
a	0x10
b	0x18
c	0x20
d	NULL

0x10	0x14	0x18	0x1c	0x20	0x24	0x28	0x2c	0x30	0x34

a	b + padding	c

Summary: Bump Allocator

- A bump allocator is an extreme heap allocator – it optimizes only for **throughput**, not **utilization**.

- Better allocators strike a more reasonable balance. How can we do this?

Questions to consider:

1. How do we keep track of free blocks?

2. How do we choose an appropriate free block in which to place a newly allocated block?

3. After we place a newly allocated block in some free block, what do we do with the remainder of the free block?

4. What do we do with a block that has just been freed?

Lecture Plan

- The heap so far

- What is a heap allocator?

- Heap allocator requirements and goals

- Method 0: Bump Allocator

- **Method 1: Implicit Free List Allocator**

- Method 2: Explicit Free List Allocator

Implicit Free List Allocator

- **Key idea:** in order to reuse blocks, we need a way to track which blocks are allocated and which are free.

- We could store this information in a separate global data structure, but this is inefficient.

- Instead: let's allocate extra space before each block for a **header** storing its payload size and whether it is allocated or free.

- When we allocate a block, we look through the blocks to find a free one, and we update its header to reflect its allocated size and that it is now allocated.

- When we free a block, we update its header to reflect it is now free.

- The header should be 8 bytes (or larger).

- By storing the block size of each block, we *implicitly* have a *list* of free blocks.

Implicit Free List Allocator

- **Key idea:** in order to reuse blocks, we need a way to track which blocks are allocated and which are free.

- We could store this information in a separate global data structure, but this is inefficient.

- Instead: let's allocate extra space before each block for a **header** storing its payload size and whether it is allocated o

- When we allocate a block, we look throug
  we update its header to reflect its allocate

- When we free a block, we update its hea

- The header should be 8 bytes (or larger).

- By storing the block size of each block, we *implicitly* have a *list* of free blocks.

This is larger than the 4 byte headers specified in the book, as this makes it easier to satisfy the alignment constraint and store information!.

Implicit Free List Allocator

```
void *a = malloc(4);

void *b = malloc(8);

void *c = malloc(4);

free(b);

void *d = malloc(8);

free(a);

void *e = malloc(24);
```

0x10   0x18   0x20   0x28   0x30   0x38   0x40   0x48   0x50   0x58

72
Free

Implicit Free List Allocator

void *a = malloc(4);

void *b = malloc(8);

void *c = malloc(4);

free(b);

void *d = malloc(8);

free(a);

void *e = malloc(24);

Variable	Value
a	0x18

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58

8 Used	a + pad	56 Free	

Implicit Free List Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```
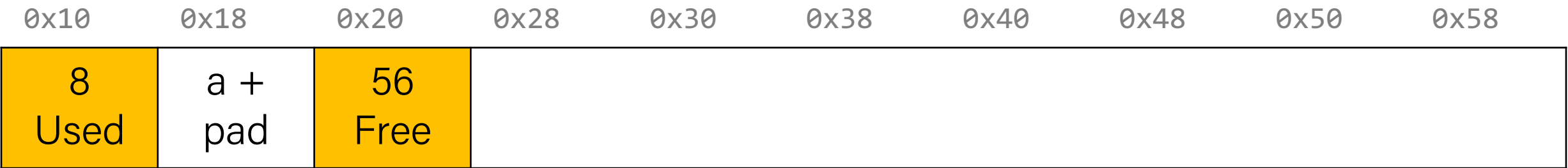
Variable	Value
a	0x18
b	0x28

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58

8 Used	a + pad	8 Used	b	40 Free		

Implicit Free List Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```
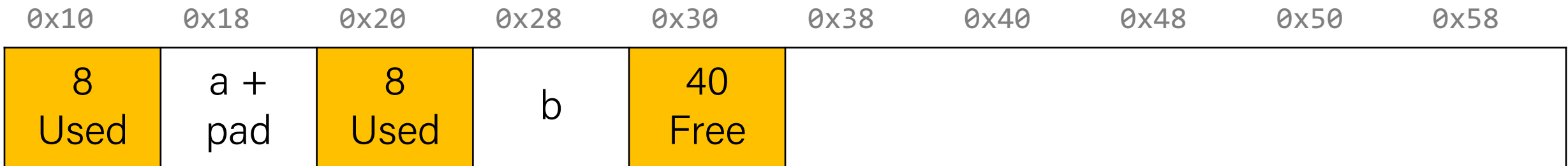
Variable	Value
a	0x18
b	0x28
c	0x38

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58

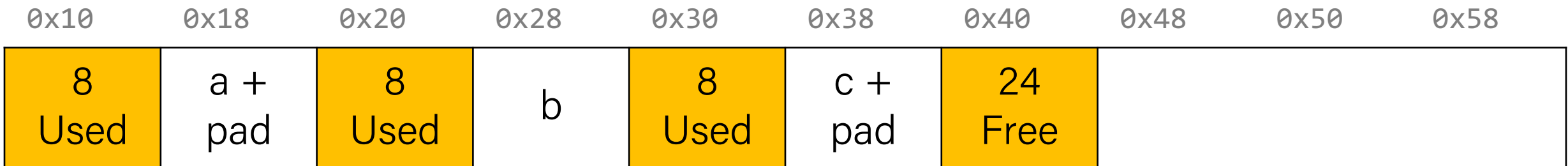| 8 Used | a + pad | 8 Used | b | 8 Used | c + pad | 24 Free | | | |

Implicit Free List Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```

Variable	Value
a	0x18
b	0x28
c	0x38

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58

| 8 Used | a + pad | 8 Free | b | 8 Used | c + pad | 24 Free | | | |

Implicit Free List Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```
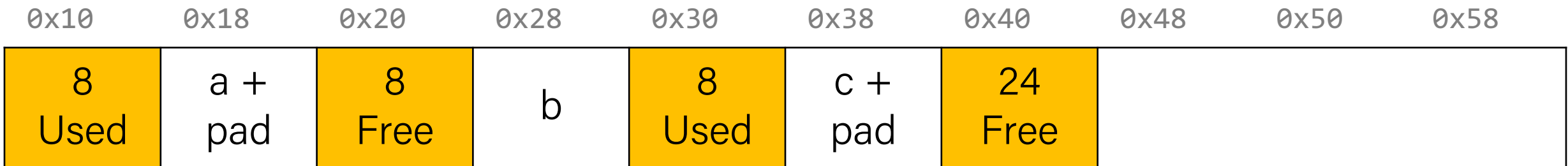
Variable	Value
a	0x18
b	0x28
c	0x38
d	0x28

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58

| 8 Used | a + pad | 8 Used | d | 8 Used | c + pad | 24 Free | | | |

Implicit Free List Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```
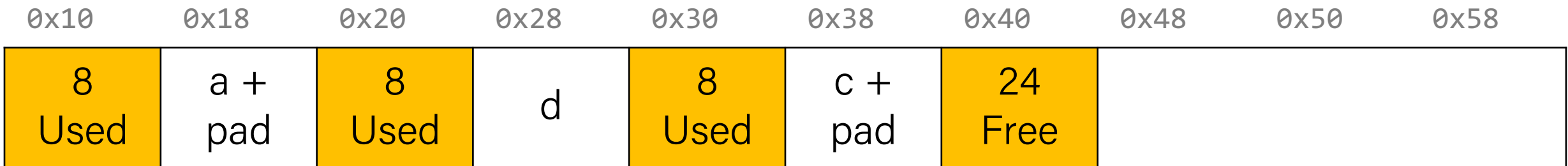
Variable	Value
a	0x18
b	0x28
c	0x38
d	0x28

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58

| 8 Free | a + pad | 8 Used | d | 8 Used | c + pad | 24 Free | | | |

Implicit Free List Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```
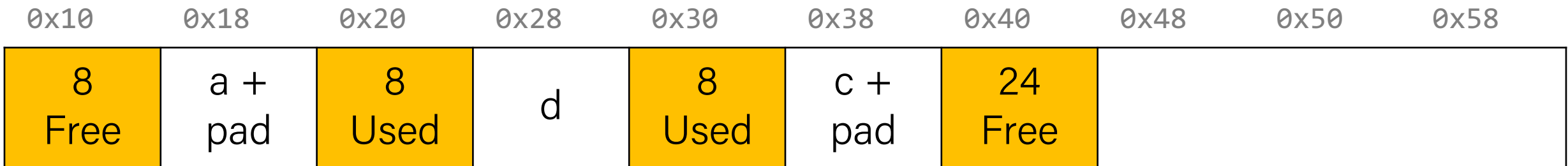
Variable	Value
a	0x18
b	0x28
c	0x38
d	0x28
e	0x48

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58

8 Free	a + pad	8 Used	d	8 Used	c + pad	24 Used	e

Implicit Free List Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```
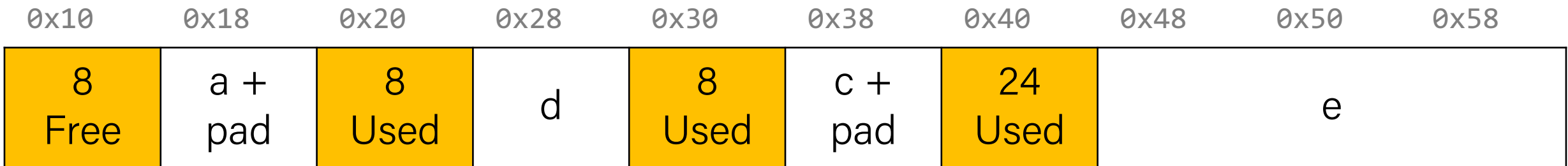
Variable	Value
a	0x18
b	0x28
c	0x38
d	0x28
e	0x48

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58

8 Free	a + pad	8 Used	d	8 Used	c + pad	24 Used	e

Representing Headers

How can we store both a size and a status (Free/Allocated) in 8 bytes?

`int` for size, `int` for status? **no! malloc/realloc use size_t for sizes!**

Key idea: block sizes will *always be multiples of 8*. (Why?)

• Least-significant 3 bits will be unused!

• *Solution:* use one of the 3 least-significant bits to store free/allocated status

Implicit Free List Allocator

How can we choose a free block to use for an allocation request?

- **First fit:** search the list from beginning each time and choose first free block that fits.

- **Next fit:** instead of starting at the beginning, continue where previous search left off.

- **Best fit:** examine every free block and choose the one with the smallest size that fits.

- First fit/next fit easier to implement

- What are the pros/cons of each approach?

Practice 1

- For the following heap layout, what would the heap look like after the following request is made, assuming we are using an **implicit** free list allocator with a **first-fit** approach?

[24 byte payload, free] [16 byte payload, free] [8 byte payload, allocated for A]

```
void *b = malloc(8);
```

[8 byte payload, allocated for B] [8 byte payload, free] [16 byte payload, free]
[8 byte payload, allocated for A]

Practice 2

- For the following heap layout, what would the heap look like after the following request is made, assuming we are using an **implicit** free list allocator with a **best-fit** approach?

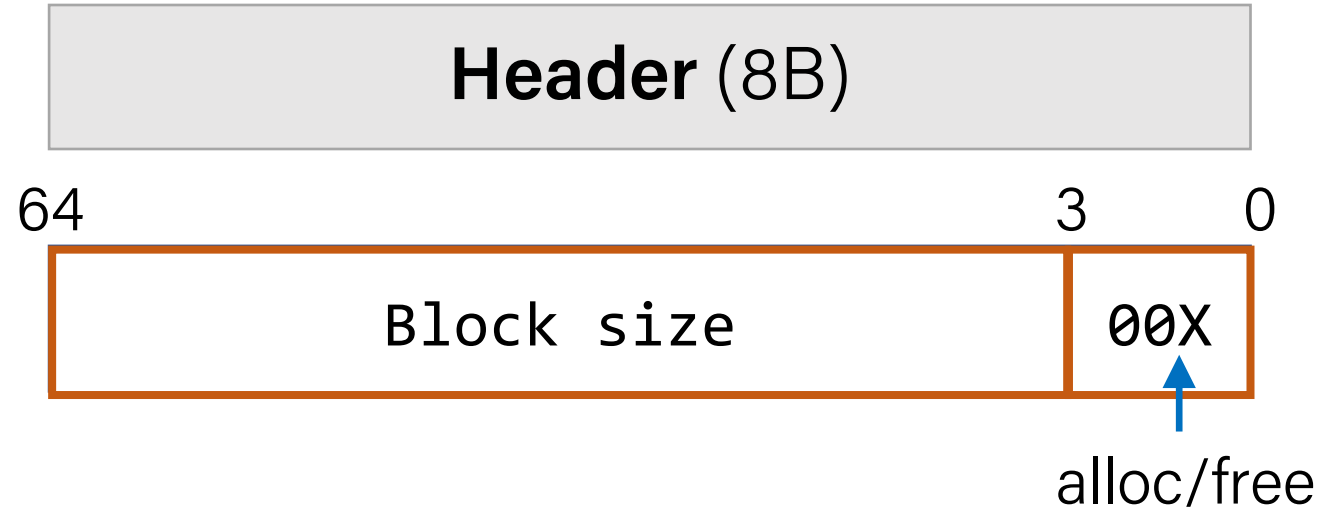[24 byte payload, free] [8 byte payload, free] [8 byte payload, allocated for A]

```
void *b = malloc(8);
```

[24 byte payload, free] [8 byte payload, allocated for B] [8 byte payload, allocated for A]

Implicit Free List Summary

For **all blocks**,

- Have a header that stores size and status.

- Our list links *all* blocks, allocated (A) and free (F).

Header (8B)

64   3   0
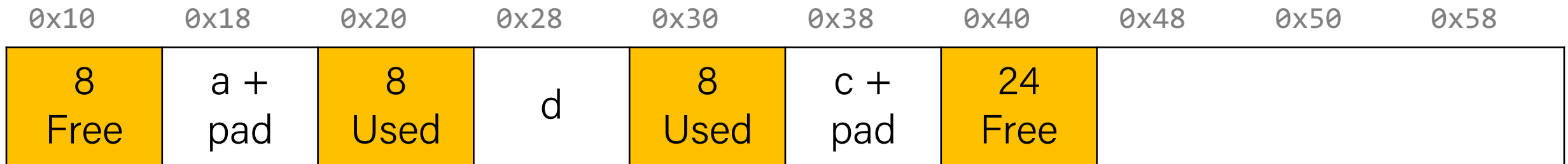
| Block size | 00X |

alloc/free

Keeping track of free blocks:

- Improves memory utilization (vs bump allocator)

- Decreases throughput (worst case allocation request has O(A + F) time)

- Increases design complexity ☺

Splitting Policy

```
...
void *e = malloc(16);
```

So far, we have seen that a reasonable allocation request splits a free block into an allocated block and a free block with remaining space. **What about edge cases?**

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58

8 Free	a + pad	8 Used	d	8 Used	c + pad	24 Free			

Splitting Policy

```
...
void *e = malloc(16);
```

So far, we have seen that a reasonable allocation request splits a free block into an allocated block and a free block with remaining space. **What about edge cases?**
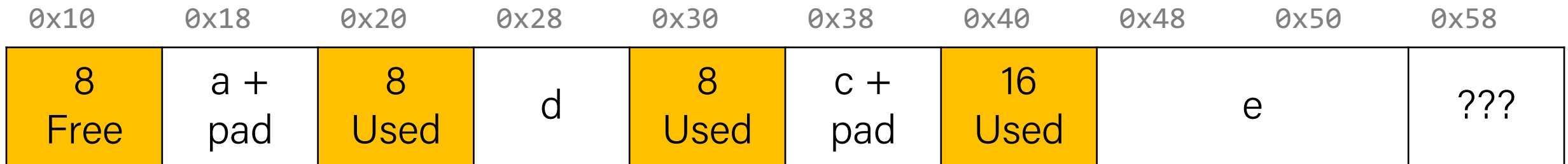
0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58

8 Free	a + pad	8 Used	d	8 Used	c + pad	16 Used	e	???

Splitting Policy

```
...
void *e = malloc(16);
```

So far, we have seen that a reasonable allocation request splits a free block into an allocated block and a free block with remaining space. **What about edge cases?**
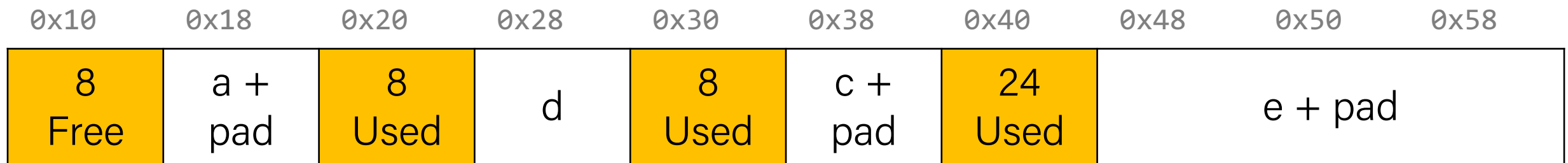
A. Throw into allocation for e **as extra padding?** *Internal fragmentation – unused bytes because of padding*

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58

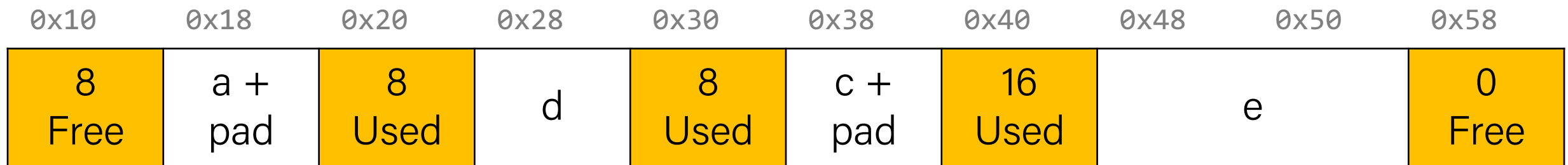| 8 Free	a + pad	8 Used	d	8 Used	c + pad	24 Used	e + pad

Splitting Policy

```
...
void *e = malloc(16);
```

So far, we have seen that a reasonable allocation request splits a free block into an allocated block and a free block with remaining space. **What about edge cases?**

A. Throw into allocation for e as extra padding?
B. Make a "zero-byte free block"? *External fragmentation – unused free blocks*

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58

8 Free	a + pad	8 Used	d	8 Used	c + pad	16 Used	e		0 Free

Revisiting Our Goals

Questions we considered:

1. How do we keep track of free blocks? **Using headers!**

2. How do we choose an appropriate free block in which to place a newly allocated block? **Iterate through all blocks.**

3. After we place a newly allocated block in some free block, what do we do with the remainder of the free block? **Try to make the most of it!**

4. What do we do with a block that has just been freed? **Update its header!**

Coalescing

```
void *e = malloc(24);   // returns NULL!
```

We do not need to worry about this problem for the implicit allocator, but investigate this for the *explicit* allocator! (More about this later).

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58

8 Free		8 Free		8 Free		24 Used			
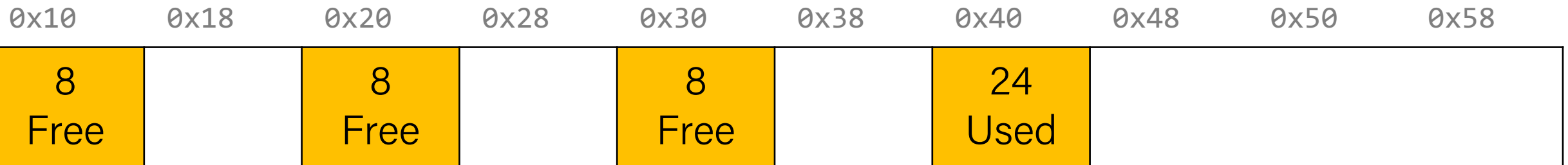
In-Place Realloc

```
void *a = malloc(4);

void *b = realloc(a, 8);
```
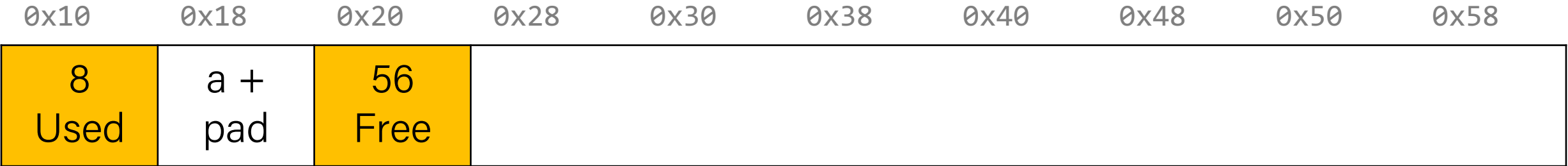
0x10   0x18   0x20   0x28   0x30   0x38   0x40   0x48   0x50   0x58

72 Free

In-Place Realloc

**void *a = malloc(4);**

void *b = realloc(a, 8);

Variable	Value
a	0x10

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58

8 Used	a + pad	56 Free	

In-Place Realloc

```
void *a = malloc(4);
void *b = realloc(a, 8);
```

Variable	Value
a	0x10
b	0x28

The implicit allocator can always move memory to a new location for a `realloc` request. The *explicit* allocator must support in-place `realloc` (more on this later).

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58

8 Free	a + pad	8 Used	b	40 Free	
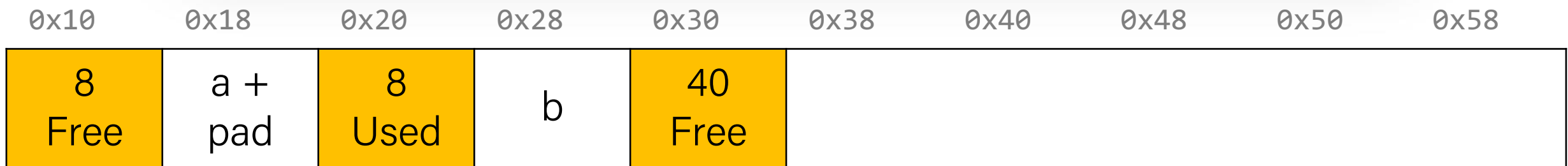
Summary: Implicit Allocator

- An implicit allocator is a more efficient implementation that has reasonable **throughput** and **utilization** due to its recycling of blocks.
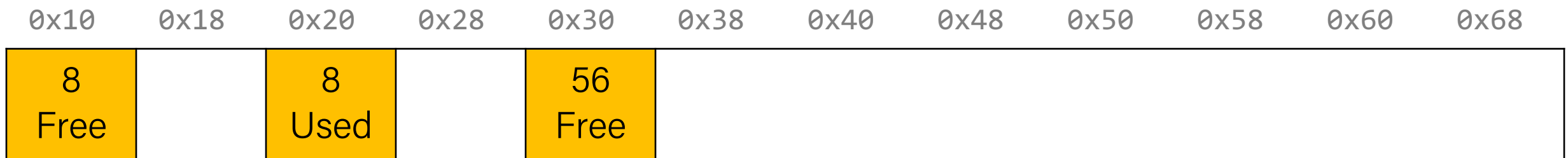
Can we do better?

1. Can we avoid searching all blocks for free blocks to reuse?

2. Can we merge adjacent free blocks to keep large spaces available?

3. Can we avoid always copying/moving data during `realloc`?

Lecture Plan

- The heap so far

- What is a heap allocator?

- Heap allocator requirements and goals

- Method 0: Bump Allocator

- Method 1: Implicit Free List Allocator

- **Method 2: Explicit Free List Allocator**
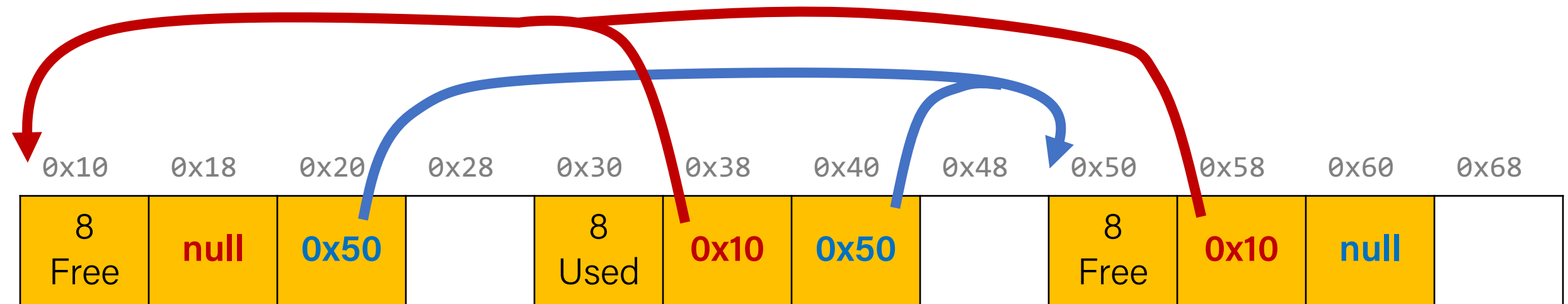 - **Explicit Allocator**
  - Coalescing
  - In-place `realloc`

Can We Do Better?

- It would be nice if we could jump *just between free blocks*, rather than all blocks, to find a block to reuse.

- **Idea:** let's modify each header to add a pointer to the previous free block and a pointer to the next free block.

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58	0x60	0x68

8 Free		8 Used		56 Free							

Can We Do Better?

- It would be nice if we could jump *just between free blocks*, rather than all blocks, to find a block to reuse.

- **Idea:** let's modify each header to add a pointer to the **previous** free block and a pointer to the **next** free block.

Can We Do Better?

- It would be nice if we could jump *just between free blocks*, rather than all blocks, to find a block to reuse.

- **Idea:** let's modify each header to add a pointer to the **previous** free block and a pointer to the **next** free block.

> This is inefficient – it triples the size of *every* header, when we just need to jump from one <u>free</u> block to another. And even if we just made free headers bigger, it's complicated to have two different header sizes.

| 0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58	0x60	0x68
8 Free	null	0x50		8 Used	0x10	0x50		8 Free	0x10	null	

Can We Do Better?

- It would be nice if we could jump *just between free blocks*, rather than all blocks, to find a block to reuse.

- **Idea:** let's modify each header to add a pointer to the previous free block and a pointer to the next free block.

Can We Do Better?

- It would be nice if we could jump *just between free blocks*, rather than all blocks, to find a block to reuse.

- **Idea:** let's modify each header to add a pointer to the previous free block and a pointer to the next free block. *This is inefficient / complicated.*

- **Where can we put these pointers to the next/previous free block?**

- **Idea:** In a separate data structure?

Can We Do Better?

- It would be nice if we could jump *just between free blocks*, rather than all blocks, to find a block to reuse.

- **Idea:** let's modify each header to add a pointer to the previous free block and a pointer to the next free block. *This is inefficient / complicated.*

- **Where can we put these pointers to the next/previous free block?**

- **Idea:** In a separate data structure? *More difficult to access in a separate place – prefer storing near blocks on the heap itself.*
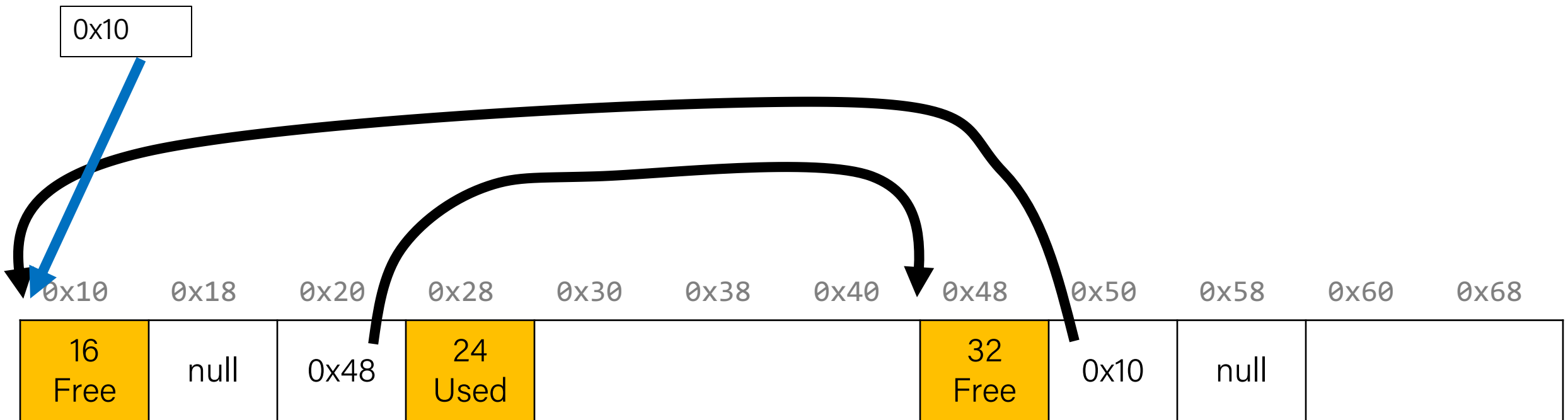
Can We Do Better?

- **Key Insight:** the payloads of the free blocks aren't being used, because they're free.

- **Idea:** since we only need to store these pointers for free blocks, let's store them in the <u>first 16 bytes of each free block's payload!</u>

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58	0x60	0x68

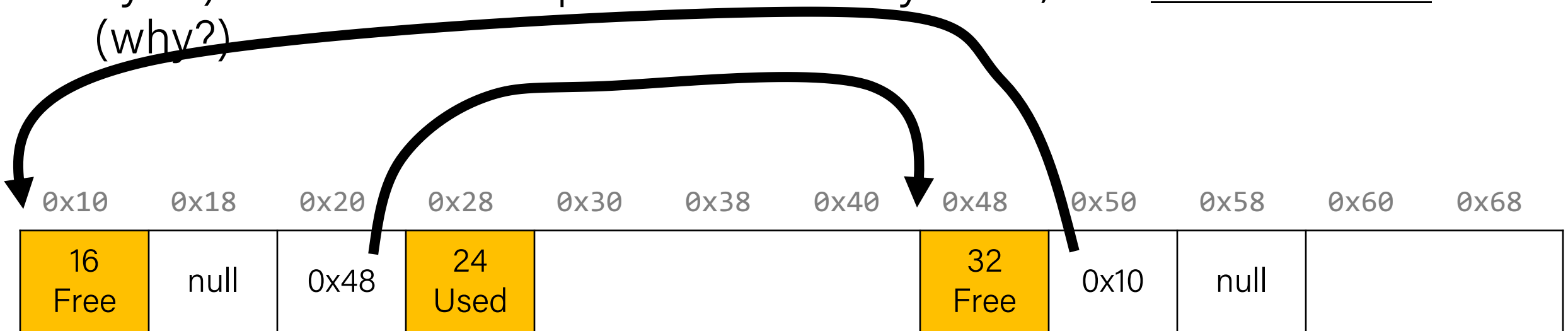| 16 Free		24 Used			32 Free	
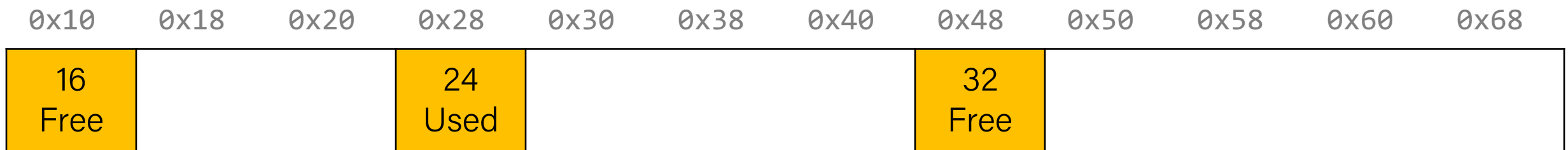
Can We Do Better?

- **Key Insight:** the payloads of the free blocks aren't being used, because they're free.

- **Idea:** since we only need to store these pointers for free blocks, let's store them in the <u>first 16 bytes of each free block's payload!</u>

First free block

0x10

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58	0x60	0x68

16 Free	null	0x48	24 Used				32 Free	0x10	null	

Can We Do Better?

- **Key Insight:** the payloads of the free blocks aren't being used, because they're free.

- **Idea:** since we only need to store these pointers for free blocks, let's store them in the <u>first 16 bytes of each free block's payload!</u>

- This means each payload must be big enough to store 2 pointers (16 bytes). So we must require that for every block, free <u>and allocated.</u> (why?)

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58	0x60	0x68

16 Free	null	0x48	24 Used				32 Free	0x10	null	

Explicit Free List Allocator

- This design builds on the implicit allocator, but also stores pointers to the next and previous free block inside each free block's payload.

- When we allocate a block, we look through <u>just the free blocks</u> using our linked list to find a free one, and we update its header <u>and the linked list</u> to reflect its allocated size and that it is now allocated.

- When we free a block, we update its header to reflect it is now free and <u>update the linked list</u>.

This **explicit** list of free blocks increases request throughput, with some costs (design and internal fragmentation)
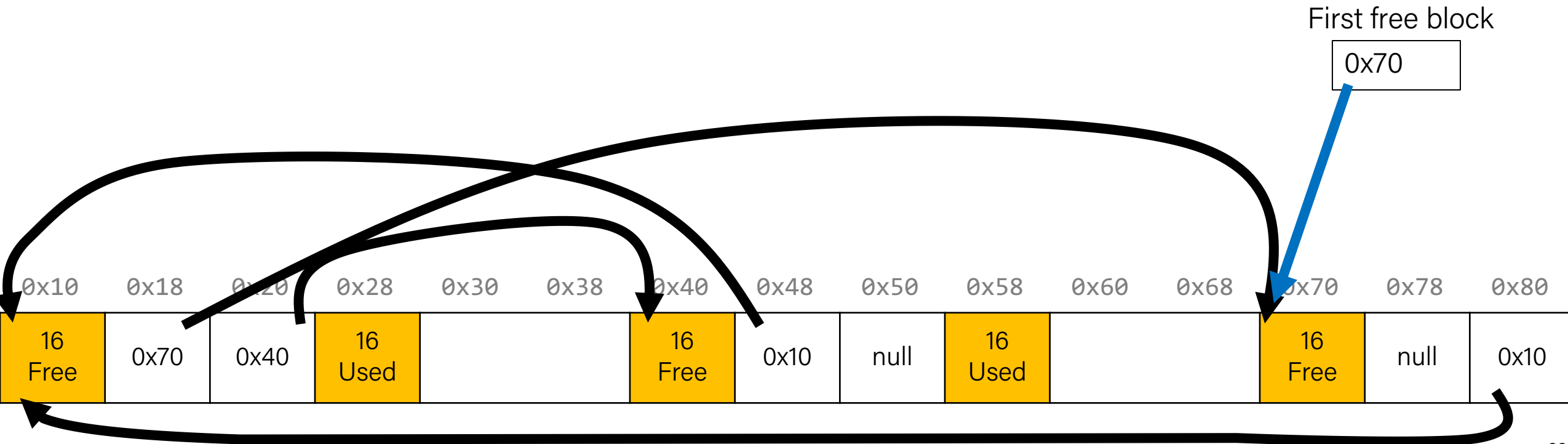
Explicit Free List: List Design

How do you want to organize your explicit free list?
(compare utilization/throughput)

A. Address-order (each block's address
   is less than successor block's address)

Better memory util,
Linear free

B. Last-in first-out (LIFO)/like a stack, where
   newly freed blocks are at the beginning of the list

Constant free (push
recent block onto stack)

C. Other (e.g., by size, etc.)

(more at end of lecture)

Explicit Free List: List Design

Note that the doubly-linked list *does not have to be in address order*.

First free block

0x70

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58	0x60	0x68	0x70	0x78	0x80
16 Free	0x70	0x40	16 Used			16 Free	0x10	null	16 Used			16 Free	null	0x10

Implicit vs. Explicit: So Far

Implicit Free List

- 8B header for size + alloc/free status

- Allocation requests are worst-case linear in total number of blocks

- Implicitly address-order

Explicit Free List

- 8B header for size + alloc/free status

- Free block payloads store prev/next free block pointers

- Allocation requests are worst-case linear in number of free blocks

- Can choose block ordering
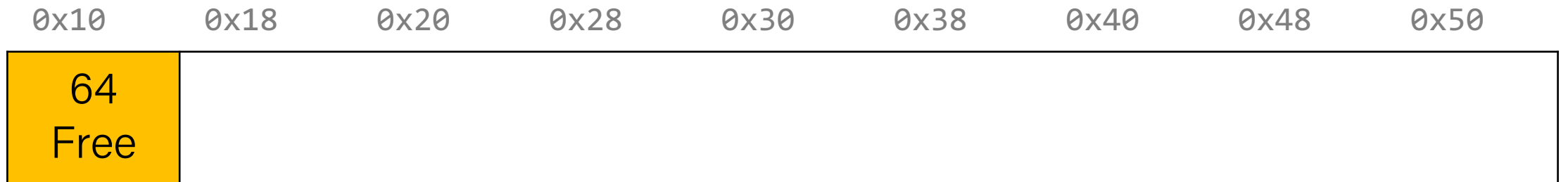
Revisiting Our Goals

Can we do better?

1. Can we avoid searching all blocks for free blocks to reuse? **Yes! We can use a doubly-linked list.**

2. Can we merge adjacent free blocks to keep large spaces available?

3. Can we avoid always copying/moving data during `realloc`?

Revisiting Our Goals

Can we do better?

1. Can we avoid searching all blocks for free blocks to reuse? **Yes! We can use a doubly-linked list.**

2. Can we merge adjacent free blocks to keep large spaces available?

3. Can we avoid always copying/moving data during `realloc`?

Lecture Plan

- The heap so far

- What is a heap allocator?

- Heap allocator requirements and goals

- Method 0: Bump Allocator

- Method 1: Implicit Free List Allocator

- **Method 2: Explicit Free List Allocator**
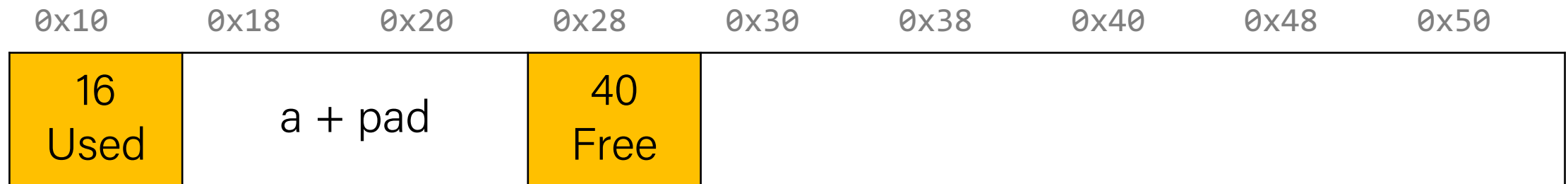 - Explicit Allocator
  - **Coalescing**
  - In-place `realloc`

Coalescing

```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```
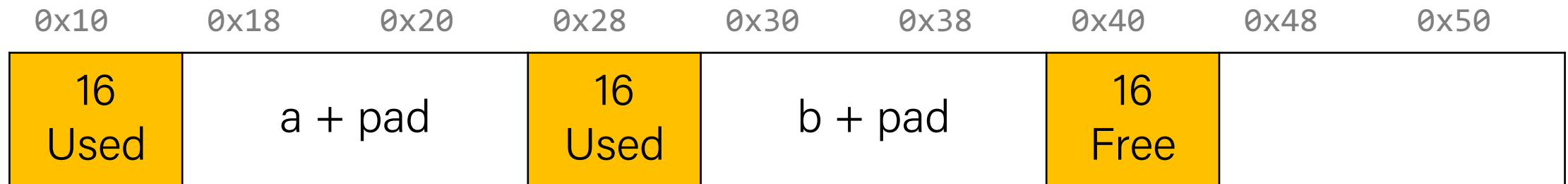
0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50

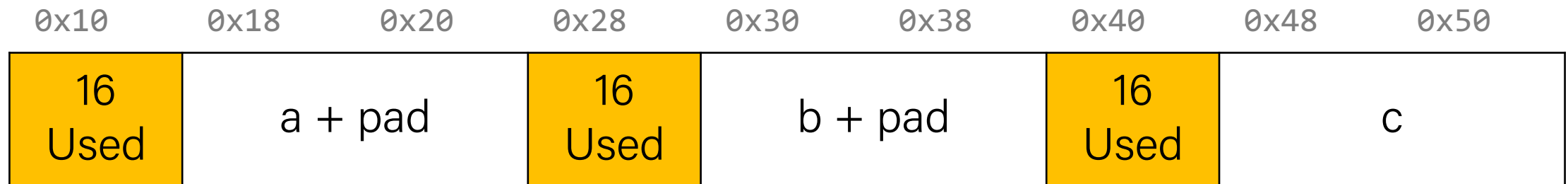| 64 Free	

Coalescing

```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```

```
0x10        0x18        0x20        0x28        0x30        0x38        0x40        0x48        0x50
```
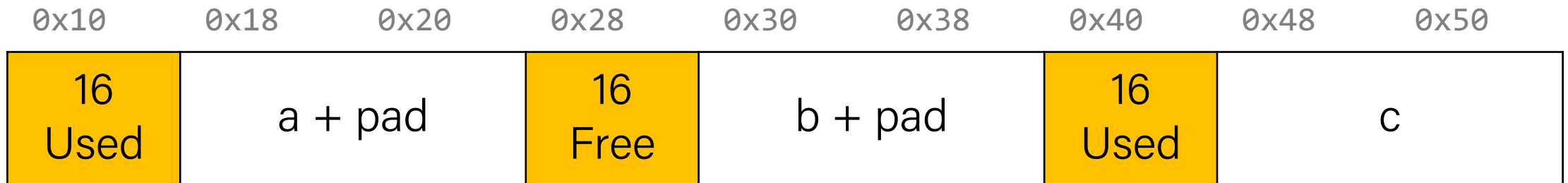
16 Used	a + pad	40 Free	

Coalescing

```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50

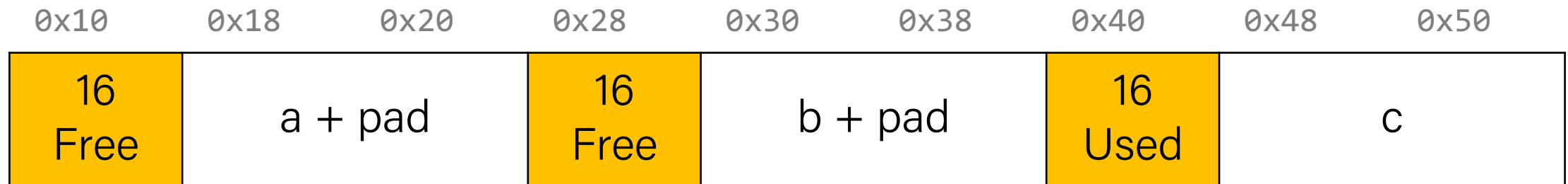| 16 Used	a + pad	16 Used	b + pad	16 Free	

Coalescing

```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50

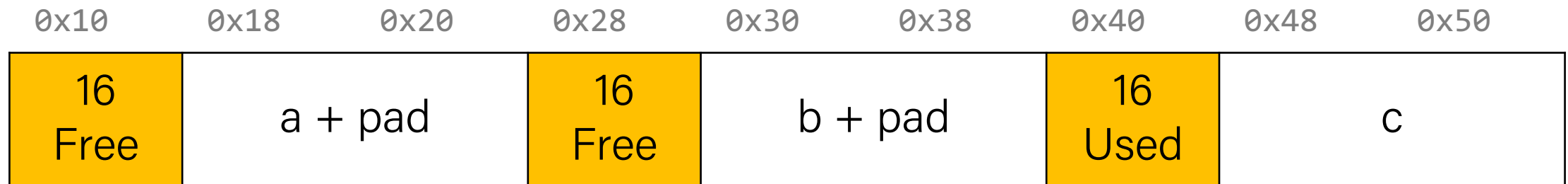| 16 Used	a + pad	16 Used	b + pad	16 Used	c

Coalescing

```
void *a = malloc(8);

void *b = malloc(8);

void *c = malloc(16);

free(b);

free(a);

void *d = malloc(32);
```

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50

16 Used	a + pad	16 Free	b + pad	16 Used	c

Coalescing

```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50

16 Free	a + pad	16 Free	b + pad	16 Used	c

Coalescing

```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```

We have enough memory space, but it is fragmented into free blocks sized from earlier requests!

We'd like to be able to merge adjacent free blocks back together. How can we do this?

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50

16 Free	a + pad	16 Free	b + pad	16 Used	c

Coalescing

```
void *a = malloc(8);

void *b = malloc(8);

void *c = malloc(16);

free(b);

free(a);

void *d = malloc(32);
```

Hey, look! I have a free neighbor. Let's be friends! ☺

| 0x10 | 0x18 | 0x20 | 0x28 | 0x30 | 0x38 | 0x40 | 0x48 | 0x50 |

| 16 Free | a + pad | 16 Free | b + pad | 16 Used | c |

Coalescing

```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```

Hey, look! I have a free neighbor. Let's be friends! ☺

0x10   0x18   0x20   0x28   0x30   0x38   0x40   0x48   0x50
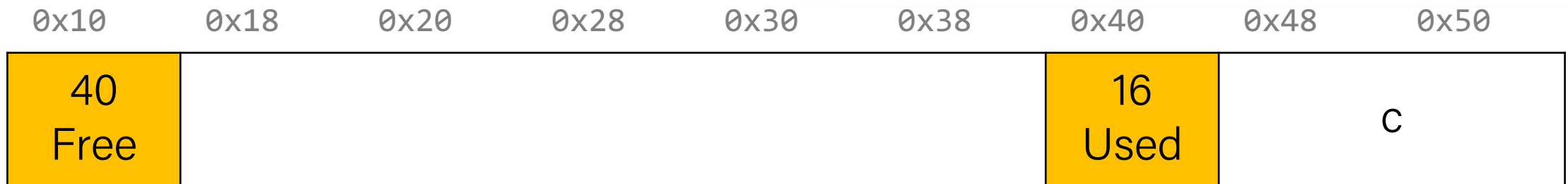
| 40 Free | | 16 Used | c |
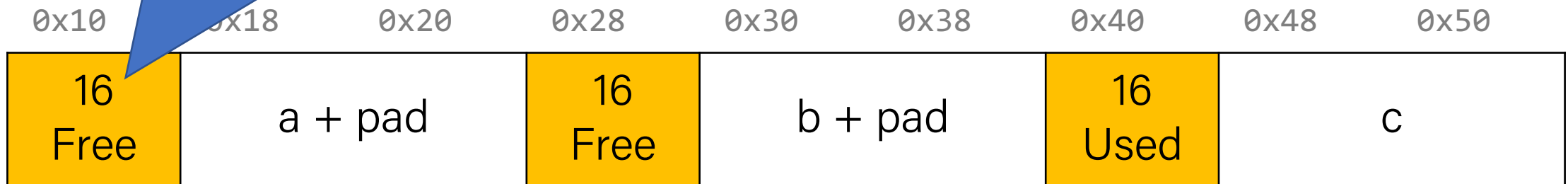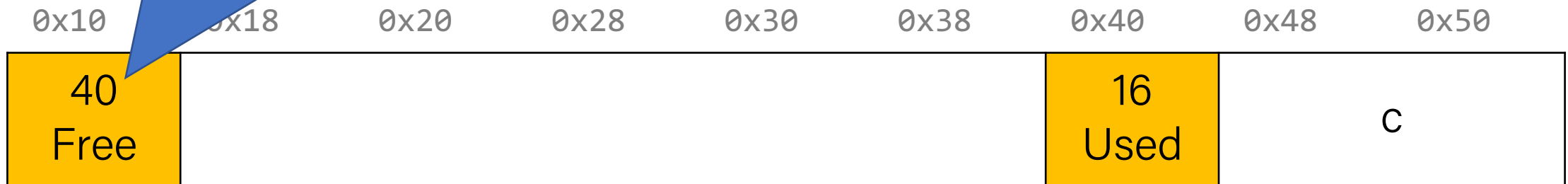
Coalescing

```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```

The process of combining adjacent free blocks is called coalescing.

For your explicit heap allocator, you should coalesce if possible when a block is freed. **You only need to coalesce the most immediate right neighbor.**

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50

| 40 Free | | | | | | 16 Used | c | |

Revisiting Our Goals

Can we do better?

1. Can we avoid searching all blocks for free blocks to reuse? **Yes! We can use a doubly-linked list.**

2. Can we merge adjacent free blocks to keep large spaces available? **Yes! We can coalesce on** `free()`.

3. Can we avoid always copying/moving data during `realloc`?

Revisiting Our Goals

Can we do better?

1. Can we avoid searching all blocks for free blocks to reuse? **Yes! We can use a doubly-linked list.**

2. Can we merge adjacent free blocks to keep large spaces available? **Yes! We can coalesce on** `free()`.

3. Can we avoid always copying/moving data during `realloc`?

Lecture Plan

- The heap so far

- What is a heap allocator?

- Heap allocator requirements and goals

- Method 0: Bump Allocator

- Method 1: Implicit Free List Allocator

- **Method 2: Explicit Free List Allocator**
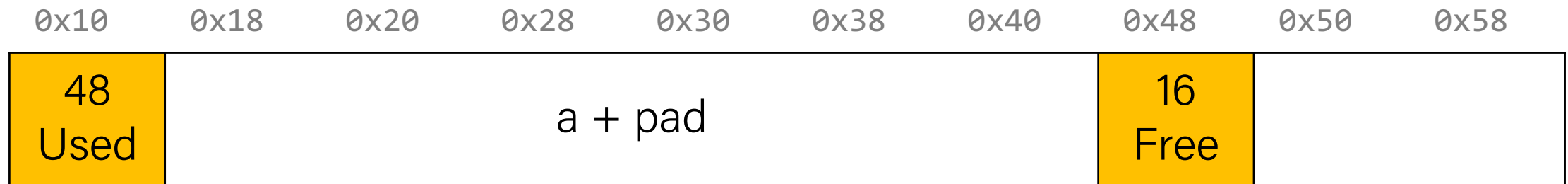 - Explicit Allocator
  - Coalescing
  - In-place `realloc`

`realloc`

- For the implicit allocator, we didn't worry too much about `realloc`. We always moved data when they requested a different amount of space.
  - Note: `realloc` can grow *or* shrink the data size.
- But sometimes we may be able to keep the data in the same place. How?
  - **Case 1:** size is growing, but we added padding to the block and can use that
  - **Case 2:** size is shrinking, so we can use the existing block
  - **Case 3:** size is growing, and current block isn't big enough, but adjacent blocks are free.

realloc: Growing In Place

```
void *a = malloc(42);
...
void *b = realloc(a, 48);
```

a's earlier request was too small, so we added padding. Now they are requesting a larger size we can satisfy with that padding! So `realloc` can return the same address.
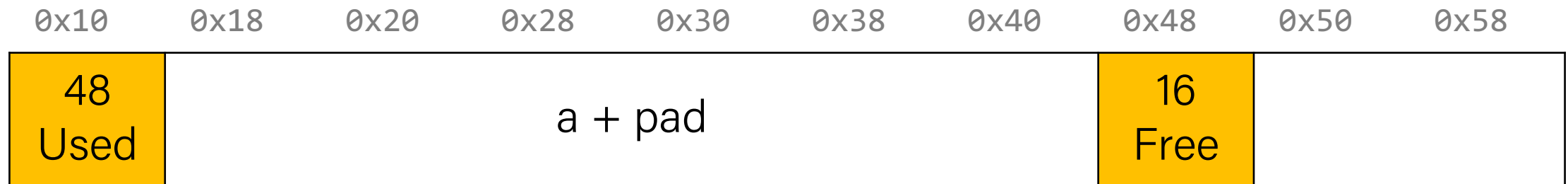
0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58

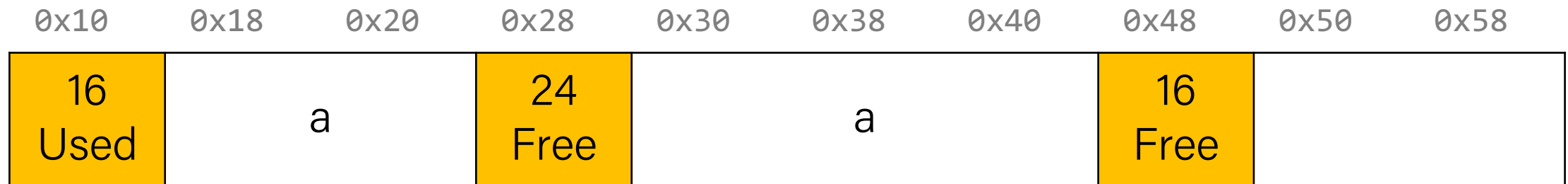| 48 Used	a + pad	16 Free	

realloc: Growing In Place

```
void *a = malloc(42);
...
void *b = realloc(a, 16);
```

If a **realloc** is requesting to shrink, we can still use the same starting address.

If we can, we should try to recycle the now-freed memory into another freed block.

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58

48 Used	a + pad	16 Free	

realloc: Growing In Place

```
void *a = malloc(42);
...
void *b = realloc(a, 16);
```

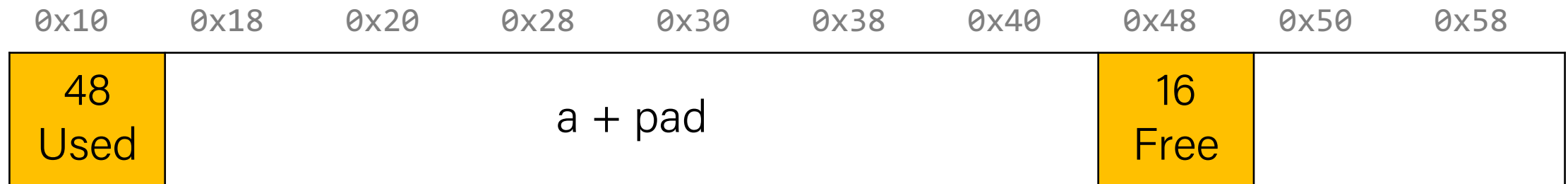If a `realloc` is requesting to shrink, we can still use the same starting address.

If we can, we should try to recycle the now-freed memory into another freed block.

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58

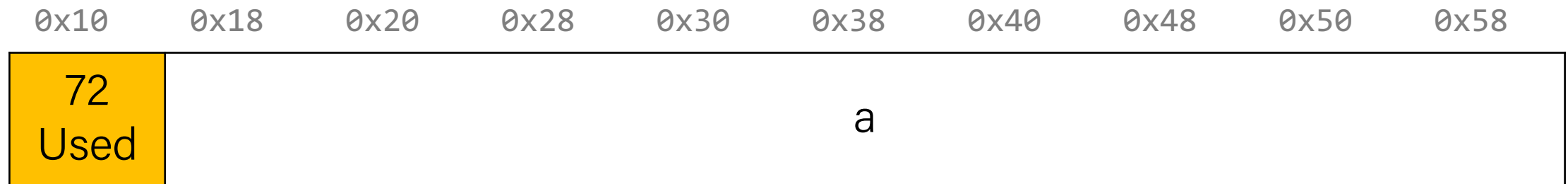| 16 Used	a	24 Free	a	16 Free	

realloc: Growing In Place

```
void *a = malloc(42);
...
void *b = realloc(a, 72);
```

Even with the padding, we don't have enough space to satisfy the larger size. But we have an adjacent neighbor that is free – let's team up!

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58

48 Used	a + pad	16 Free	

realloc: Growing In Place

```
void *a = malloc(42);
...
void *b = realloc(a, 72);
```

Even with the padding, we don't have enough space to satisfy the larger size. But we have an adjacent neighbor that is free – let's team up!
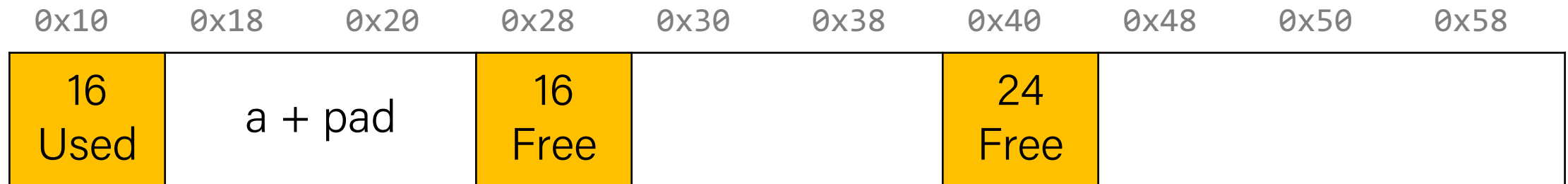
Now we can still return the same address.

| 0x10 | 0x18 | 0x20 | 0x28 | 0x30 | 0x38 | 0x40 | 0x48 | 0x50 | 0x58 |

72
Used

a

realloc: Growing In Place

```
void *a = malloc(8);
...
void *b = realloc(a, 72);
```

Here, you should combine with your *right* neighbors as much as possible until we get enough space, or until we know we cannot get enough space.
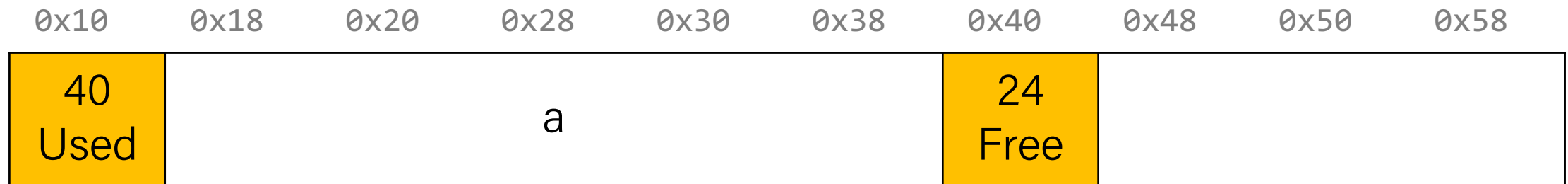
| 0x10 | 0x18 | 0x20 | 0x28 | 0x30 | 0x38 | 0x40 | 0x48 | 0x50 | 0x58 |

| 16 Used | a + pad | | 16 Free | | | 24 Free | | | |

realloc: Growing In Place

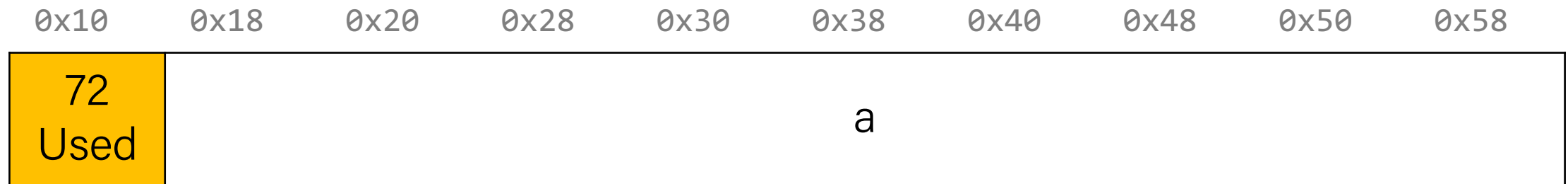```
void *a = malloc(8);
...
void *b = realloc(a, 72);
```

Here, you should combine with your *right* neighbors as much as possible until we get enough space, or until we know we cannot get enough space.

| 0x10 | 0x18 | 0x20 | 0x28 | 0x30 | 0x38 | 0x40 | 0x48 | 0x50 | 0x58 |
|------|------|------|------|------|------|------|------|------|------|

| 40 Used | a | 24 Free | |
|---------|---|---------|--|

# realloc: Growing In Place

```
void *a = malloc(8);
...
void *b = realloc(a, 72);
```

Here, you should combine with your *right* neighbors as much as possible until we get enough space, or until we know we cannot get enough space.

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58

72
Used

a

`realloc`

- For the implicit allocator, we didn't worry too much about `realloc`. We always moved data when they requested a different amount of space.
  - Note: `realloc` can grow *or* shrink the data size.
- But sometimes we may be able to keep the data in the same place. How?
  - **Case 1:** size is growing, but we added padding to the block and can use that
  - **Case 2:** size is shrinking, so we can use the existing block
  - **Case 3:** size is growing, and current block isn't big enough, but adjacent blocks are free.
- If you can't do an in-place `realloc`, then you should move the data elsewhere.

Practice 3

- For the following heap layout, what would the heap look like after the following request is made, assuming we are using an **explicit** free list allocator with a **first-fit** approach and **coalesce on free + realloc in-place**?

[24 byte payload, allocated for B] [16 byte payload, free] [16 byte payload, allocated for A]

```
free(B);
```

[48 byte payload, free] [16 byte payload, allocated for A]

Practice 4

- For the following heap layout, what would the heap look like after the following request is made, assuming we are using an **explicit** free list allocator with a **first-fit** approach and **coalesce on free + realloc in-place**?
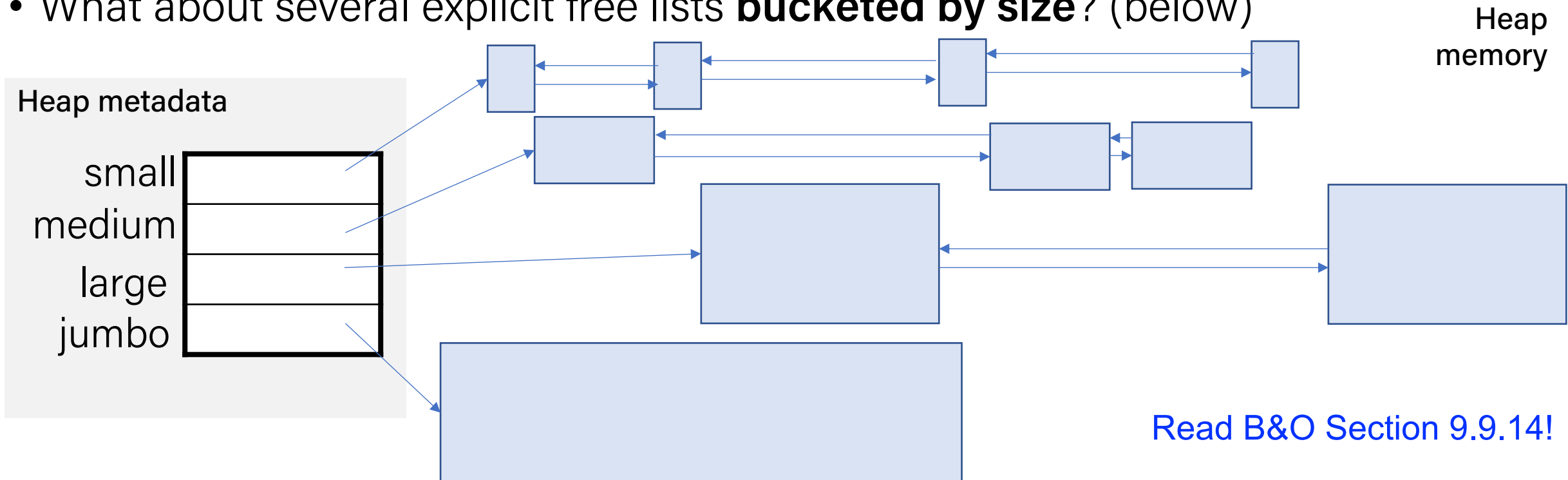
[16 byte payload, allocated for A] [32 byte payload, free] [16 byte payload, allocated for B]

```
realloc(A, 24);
```

[24 byte payload, allocated for A] [24 byte payload, free] [16 byte payload, allocated for B]

Going beyond: Explicit list w/size buckets

- Explicit lists are much faster than implicit lists.

- However, a first-fit placement policy is still linear in total # of free blocks.

- What about an explicit free list **sorted by size** (e.g., as a tree)?

- What about several explicit free lists **bucketed by size**? (below)

Heap memory

Heap metadata

small
medium
large
jumbo

Read B&O Section 9.9.14!

Recap

- The heap so far

- What is a heap allocator?

- Heap allocator requirements and goals

- Method 0: Bump Allocator

- Method 1: Implicit Free List Allocator

- Method 2: Explicit Free List Allocator