CS107 Lecture 23
Managing The Heap, Take III

Reading: None

This document is copyright (C) Stanford Computer Science, licensed under Creative Commons Attribution 2.5 License. All rights reserved.
Based on slides created by Cynthia Lee, Chris Gregg, Jerry Cain, Lisa Yan, Nick Troccoli, and others.

Explicit Allocators and In-Place `realloc`

For our **implicit** allocator, **`realloc`** isn't even remotely clever. We always **`malloc`** a new block, **`memcpy`** the data from old to new, and **`free`** the old block before returning the new payload address.

Sometimes we can **resize in place**. Your **explicit allocator** should try this:

- **Case 1:** size is **bigger**, but we added **padding to the block**, and we can just use that.
- **Case 2:** size is **smaller**, so we can use the **existing block** and perhaps even **clip off the excess** to create another free node.
- **Case 3:** size is **bigger**, and the current block **isn't big enough**, but adjacent blocks to the right are **free**.

Explicit Allocators and In-Place `realloc`

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

Case 1:

a's initial request for 42 bytes **was bumped up to 48 bytes**, because all payloads are required to be **a multiple of 8 bytes**.

Now the client is requesting a larger size that **can be accommodated with that padding**! **`realloc`** can return the **same address** without doing anything else.



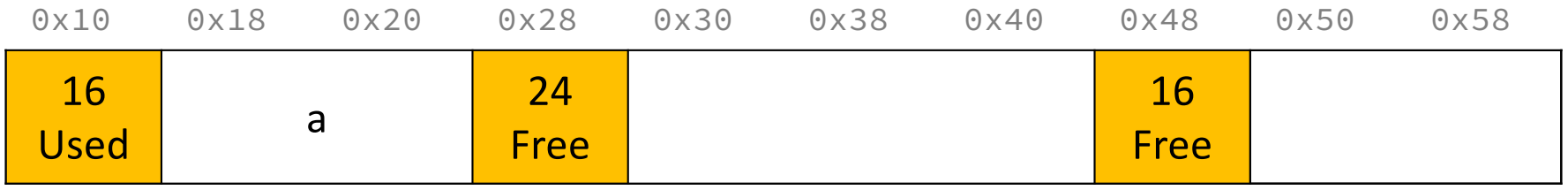
Explicit Allocators and In-Place `realloc`

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

Case 2:

If **`realloc`** is being asked to **reduce the size of the block**, we can always **return the same address**.

If the new size is **at least 24 bytes** less than the old size, the **excess payload** can be **cordoned off into another block**.
Woo!



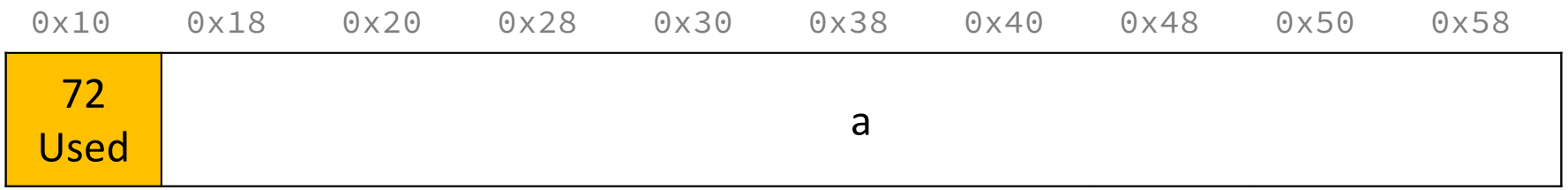
Explicit Allocators and In-Place `realloc`

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

Case 3:

Even with the padding, **there isn't enough space** within the original block to accommodate the larger size.

The **right neighbor is free**, so **`realloc`** can **absorb** it into the **original**. (Had there been 24 or more bytes of **excess payload**, we'd have **clipped off a new free block**.)



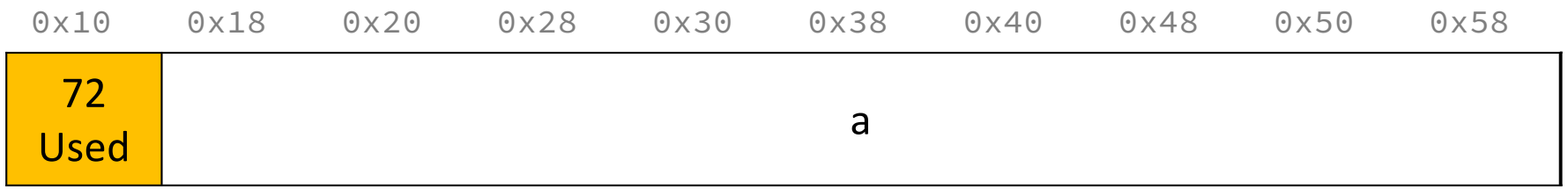
Explicit Allocators and In-Place `realloc`

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

Case 3 Generalized:

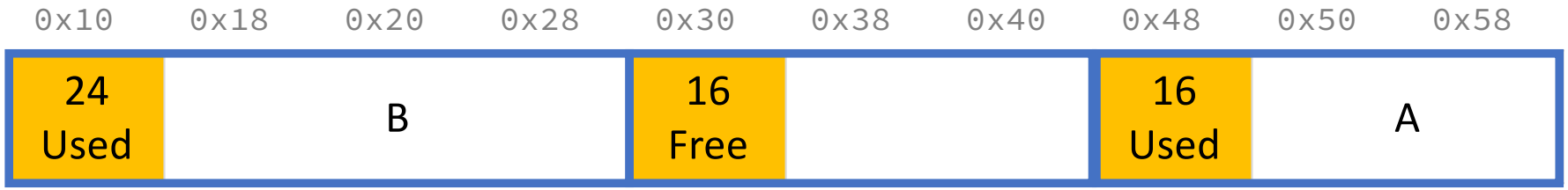
For **`assign6`**, you should **repeatedly combine with your right neighbors** until there's enough space, or until you **run out of free blocks** and **resort to a move**.

(If you **run out of free blocks** and still can't **`realloc`** in place, you **don't need to revert or restore the original blocks**.)



Explicit Allocation and Coalescing Etude 1

For the heap layout below, 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`**?

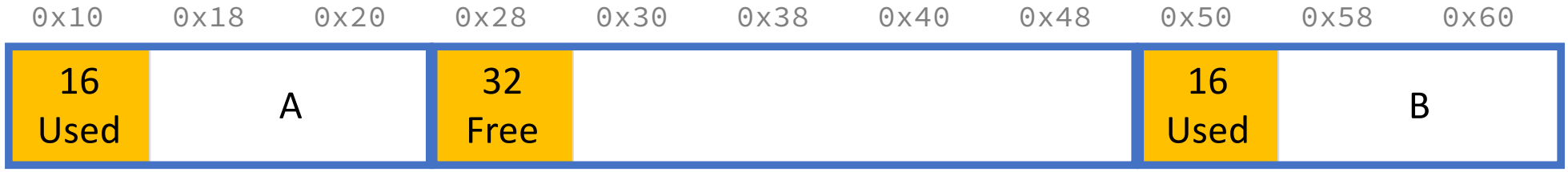



```
free(b);
```

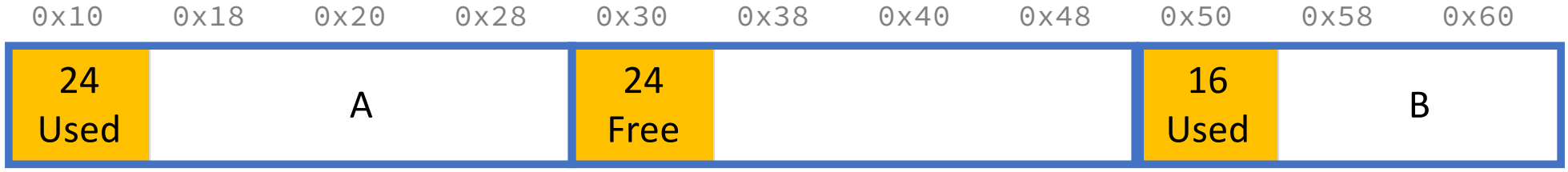


Explicit Allocation and Coalescing Etude 2

For the heap layout below, 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`** + **in-place `realloc`**?

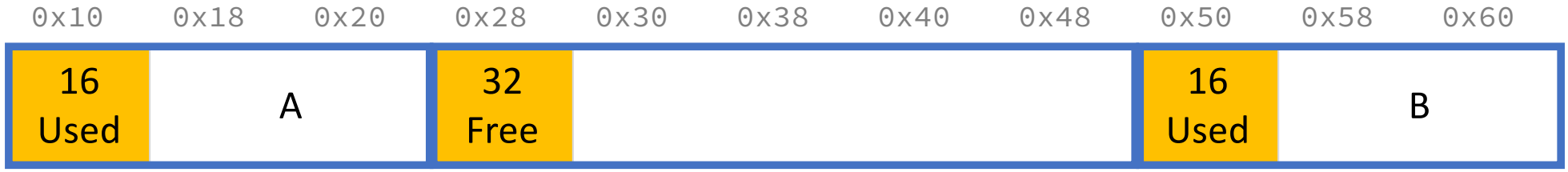



```
a = realloc(a, 24);
```

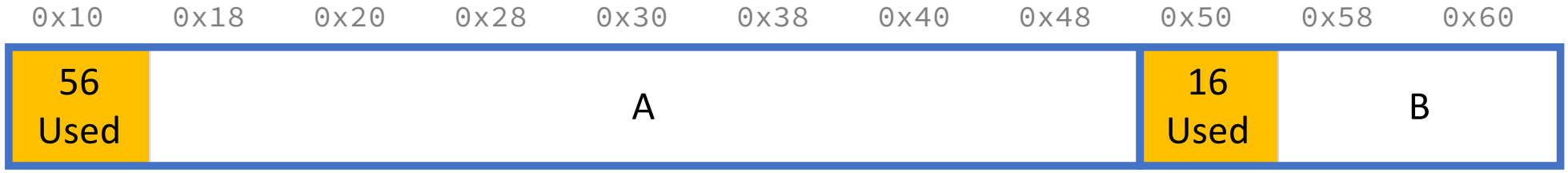


Explicit Allocation and Coalescing Etude 3

For the heap layout below, 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`** + **in-place `realloc`**?

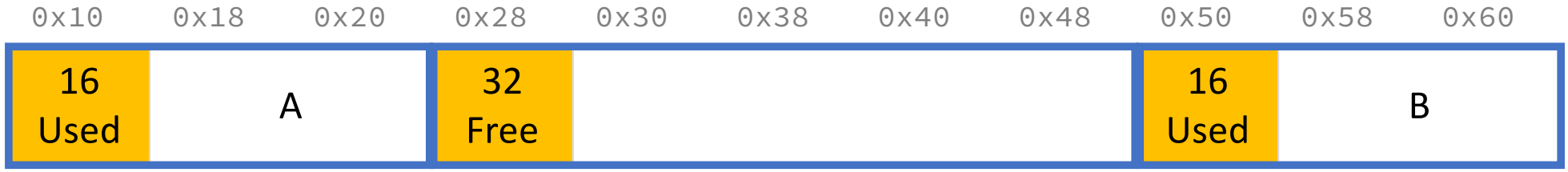



```
a = realloc(a, 56);
```



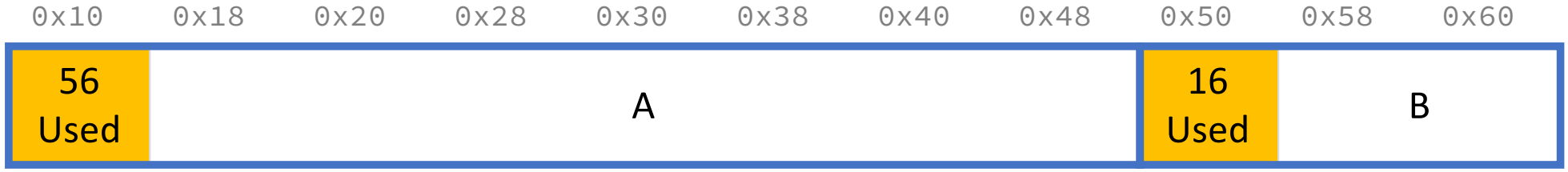
Explicit Allocation and Coalescing Etude 4

For the heap layout below, 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`** + **in-place `realloc`**?




```
a = realloc(a, 48);
```

For our explicit allocator, **we can't have a payload with fewer than 16 bytes**. The only sensible option for the remaining 8 bytes is to **add them in as padding**.



Final Assignment: Explicit Allocator

- **Must have** headers that track block information as with implicit (size, status in use or free). You should **copy your implicit version** to get started.
- **Must have** an explicit free list managed as a **doubly-linked list**, using the first 16 bytes of each free block's payload for **`next`** and **`prev`** pointers.
- **Must have** a **`malloc`** implementation that **searches the explicit list** of free blocks, thereby **ignoring blocks marked as in use**.
- **`free`** **must right-coalesce** its neighbor **whenever possible**. You may, though you're not required to, right-coalesce more than one if you'd like.
- **Must support** in-place **`realloc`** **whenever possible**. When in-place **`realloc`** is **not possible**, you should still **absorb each free block to the right** until you **exhaustively absorb all of them**.