CS107, Lecture 21
Managing The Heap, Take I

Reading: B&O 9.9 and 9.11

Ed Discussion

<u>CS107 Topic 6</u>: How do the core `malloc/realloc/free` memory-allocation operations work?

Heap Allocator Requirements

A heap allocator must…

1. Handle arbitrary request sequences of allocations and frees
2. Keep track of what memory has been allocated and what memory is free
3. Decide which memory to use when fulfilling an allocation request
4. Respond to requests as quickly as possible

Heap Allocator Requirements

A heap allocator must…

1. **Handle arbitrary request sequences of allocations and frees**
2. Keep track of what memory has been allocated and what memory is free
3. Decide which memory to use when fulfilling an allocation request
4. Respond to requests as quickly as possible

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 what memory has been allocated and what memory is free**
3. Decide which memory to use when fulfilling an allocation request
4. Respond to requests as quickly as possible

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 what memory has been allocated and what memory is free
3. **Decide which memory to use when fulfilling an allocation request**
4. Respond to requests as quickly as possible

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 what memory has been allocated and what memory is free
3. Decide which memory to use when fulfilling an allocation request
4. **Respond to requests as quickly as possible**

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…

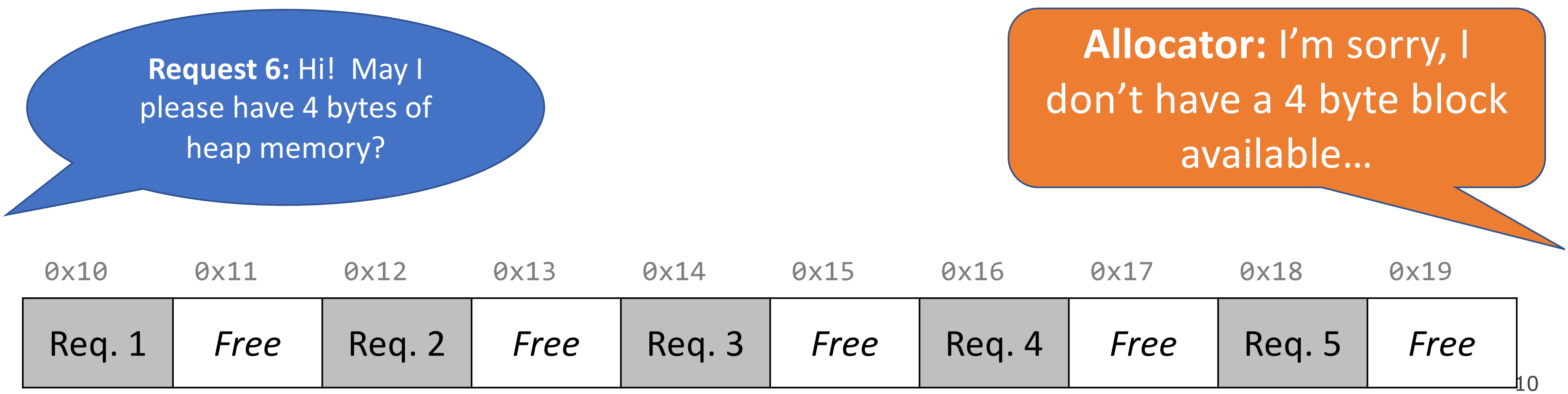
1. Handle arbitrary request sequences of allocations and frees
2. Keep track of what memory has been allocated and what memory is free
3. Decide which memory to use when fulfilling an allocation request
4. Respond to requests as quickly as possible
5. **Return addresses that are 8-byte-aligned (must be multiples of 8).**

Heap Allocator Goals

- Goal 1: Maximize **throughput**, or the number of requests completed per unit of time. This means minimizing the average time to satisfy a request.
- Goal 2: Maximize **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 memory to satisfy the request, but no single free block is large enough to satisfy it.
- In general: we want the largest address used to be as low as possible.



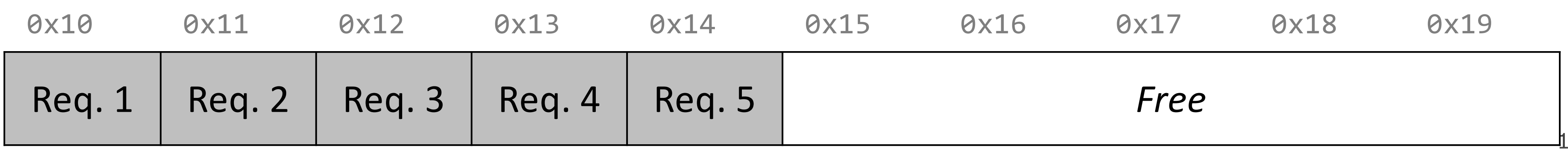
Utilization

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)

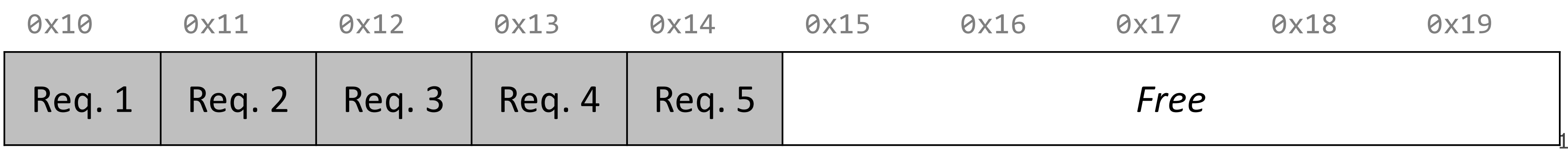
C. NO, it can't be done!



Utilization

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

- **No -** we have already shared these addresses to the client. We cannot move allocated memory around, since doing so would invalidate the pointers held by the client!



Fragmentation

- **Internal Fragmentation**: an allocated block is larger than what's 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 of 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 – i.e., it may take longer to better plan out heap memory use for each request.

Heap allocators must strike the right balance between the two.

Heap Allocator Goals

- Goal 1: Maximize **throughput**, or the number of requests completed per unit of 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 to each other)
Robust (handle client errors)
Ease of implementation/maintenance

Bump Allocator

Let’s say we want to prioritize throughput at all cost and not care about utilization even one bit. This might even mean we not care about reusing memory. How could we do this?

Bump Allocator Performance

1. Utilization

😱

Never reuses memory

2. Throughput

⭐

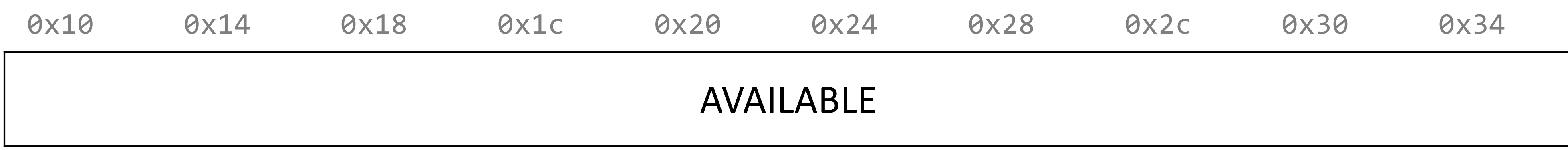
Ultra fast, short routines

Bump Allocator

- A **bump allocator** is an allocator that simply allocates the next available memory address in response to an allocation request and does **nothing** in response to **free**.
- Throughput: each **malloc** and **free** executes 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. ☹
- We provide a bump allocator implementation as part of the final assignment as a code reading exercise.

Bump Allocator

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



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

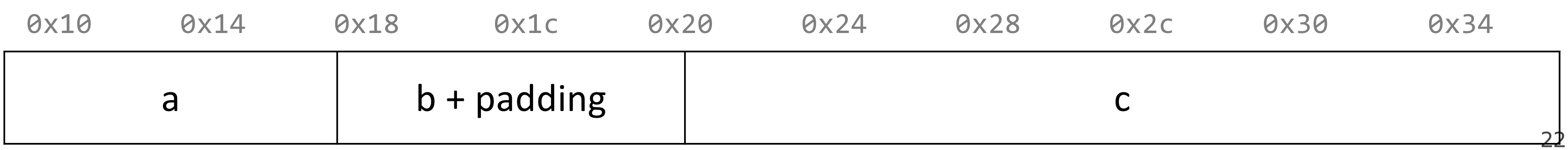
0x10 0x14 0x18 0x1c 0x20 0x24 0x28 0x2c 0x30 0x34

a	b + padding	AVAILABLE

Bump Allocator

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

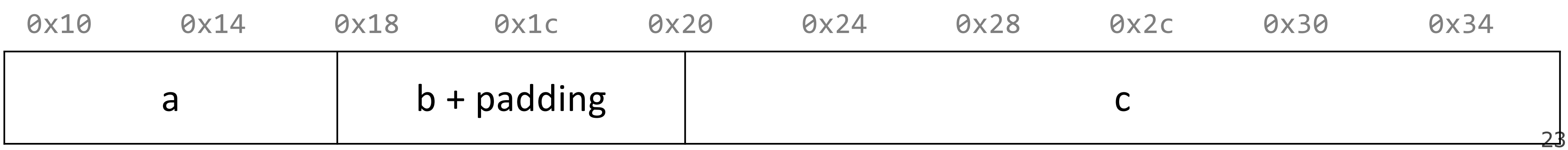
Variable	Value
a	0x10
b	0x18
c	0x20



Bump Allocator

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

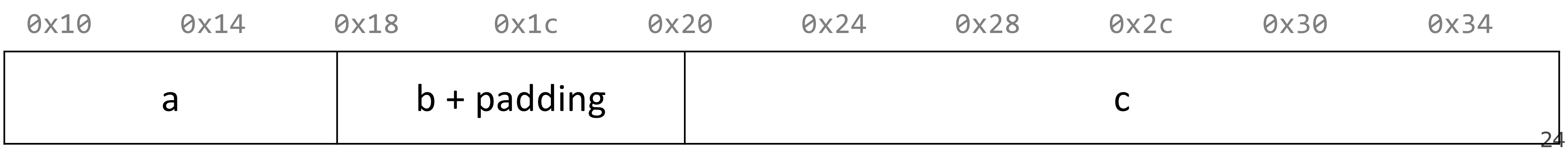
Variable	Value
a	0x10
b	0x18
c	0x20



Bump Allocator

```c
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



Summary: Bump Allocator

- A bump allocator is extreme–it optimizes only for **throughput**, not **utilization**.
- Better allocators strike a more reasonable balance to achieve acceptable and even admirable levels for both. But how?

Questions to consider:

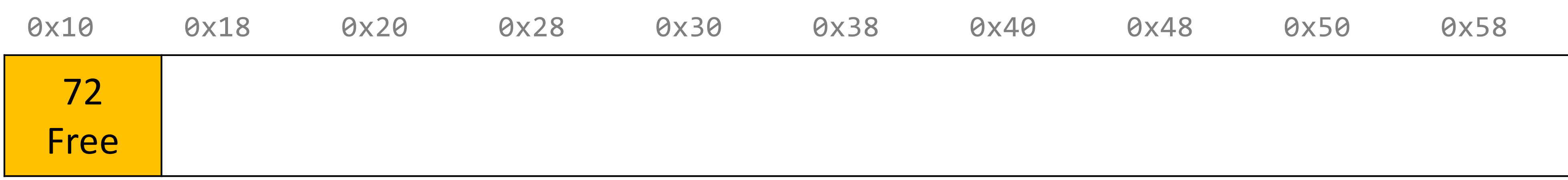
1. How do we keep track of free blocks?
2. How do we choose which free block to use to help satisfy an allocation request?
3. After we choose an appropriate free block, what do we do with any excess that isn't needed?
4. What do we do with a block as it's being freed?

Implicit Free List Allocator

- **Key idea:** in order to reuse blocks, we need a way to track which blocks are allocated and which ones are free.
- We could store this information in a separate global data structure, but this is, in general, inefficient and requires substantial overhead.
- Instead: let's allocate extra space before each block for a **header** storing its payload size and whether it's free or in use.
- When we allocate a block, we look through all blocks to find a free one and update its header to reflect its allocation size and status.
- When we free a block, we update its header to be clear it's now free.
- The header should be 8 bytes (or larger).
- By storing header information, we're **implicitly** maintaining a **list** of free blocks.

Implicit Free List Allocator

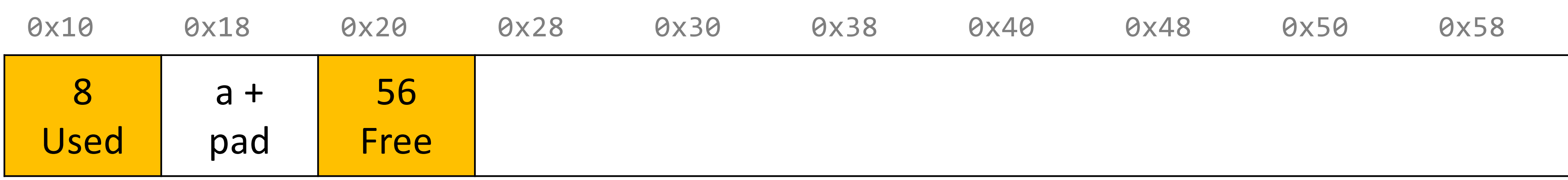
```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```



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);
```

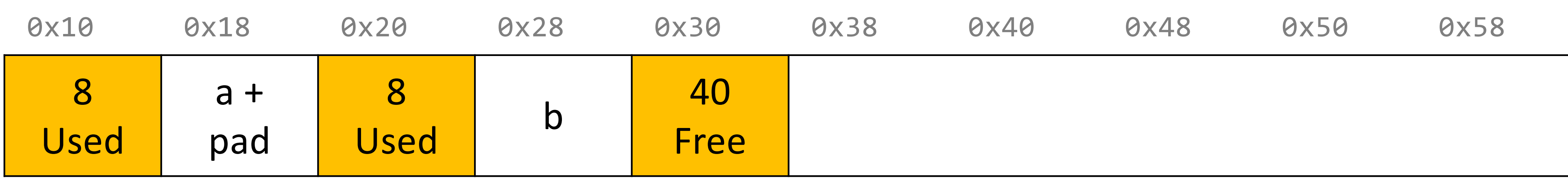
Variable	Value
a	0x18



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



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 versus 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*.

- Least-significant 3 bits aren't really needed to represent block size if they're assumed to always be zeroes!
- *Solution:* use one of the 3 least-significant bits to store free/allocated status

Implicit Free List Allocator

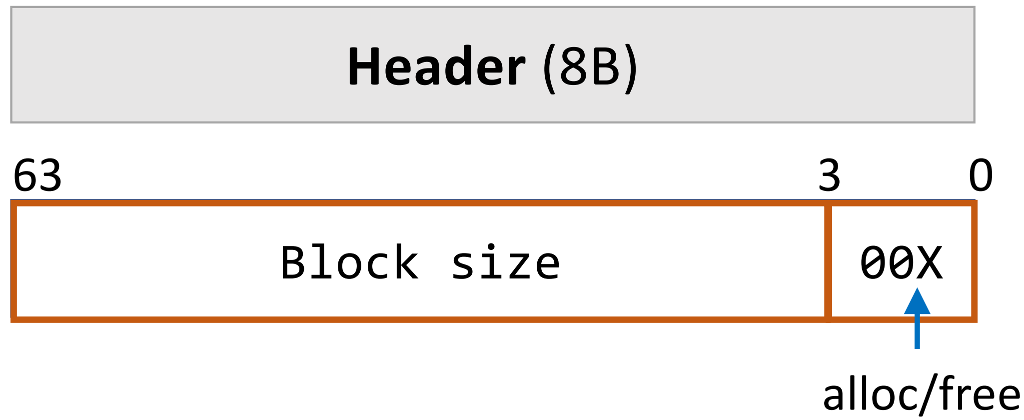
- How can we choose a free block to use for an allocation request?
  - **First fit:** search the list from beginning each time and choose first free block that fits.
  - **Next fit:** instead of starting at the beginning, continue where previous search left off.
  - **Best fit:** examine every free block 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?

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).



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 ☺ (but compared to bump, it's worth it)

Up to you! Implicit free list header design

Should we store the **block size** as
(A) payload size, or
(B) header + payload size?

Up to you! Your decision affects how you traverse the list (but be careful of off-by-one errors)

Up to you!

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?**

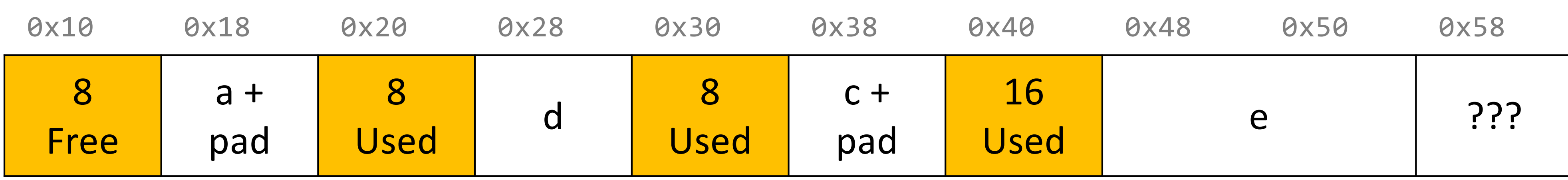
0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58
8 Free	a + pad	8 Used	d	8 Used	c + pad	24 Free			

Up to you!

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?**



Up to you!

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		

Up to you!

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

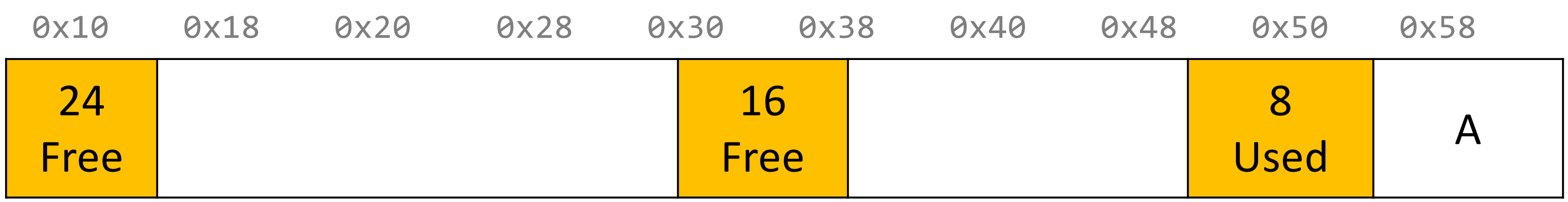
Revisiting Our Goals

Questions we considered:

1. How do we keep track of free blocks? **We use headers!**
2. How do we choose an appropriate free block in which to place a newly allocated block? **We iterate through all blocks!**
3. After we place a newly allocated block in some free block, what do we do with the rest of the free block? **We try to make the most of it!**
4. What do we do with a block that has just been freed? **We update its header!**

Practice 1: Implicit (first-fit)

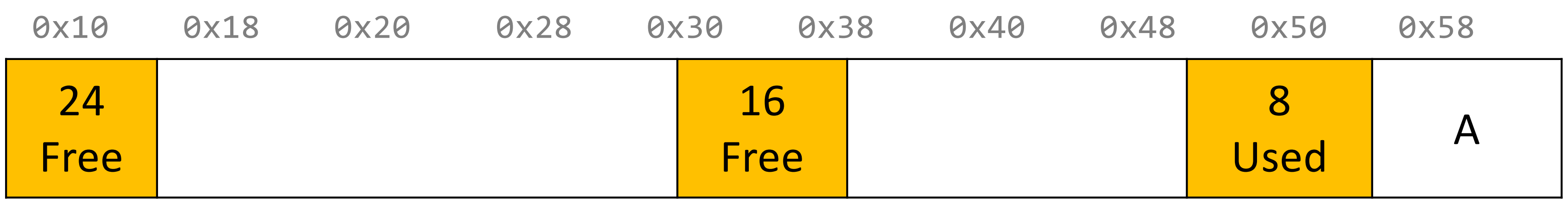
For the following heap layout, what would the heap look like after the following request is made, assuming we are using an **implicit** free list allocator with a **first-fit** approach?



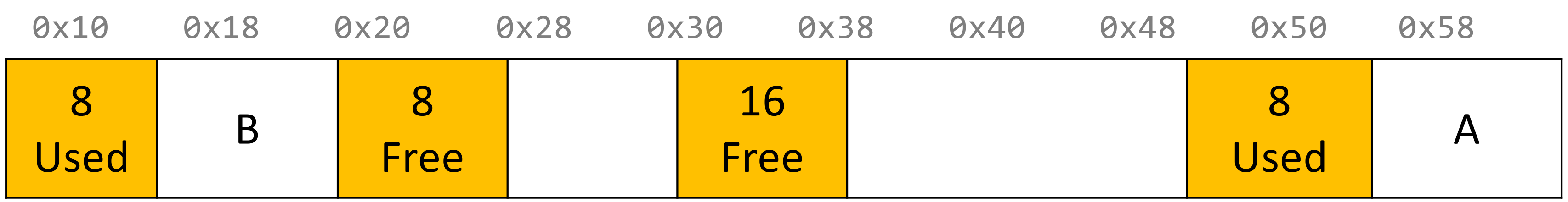

```
void *b = malloc(8);
```

Practice 1: Implicit (first-fit)

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?

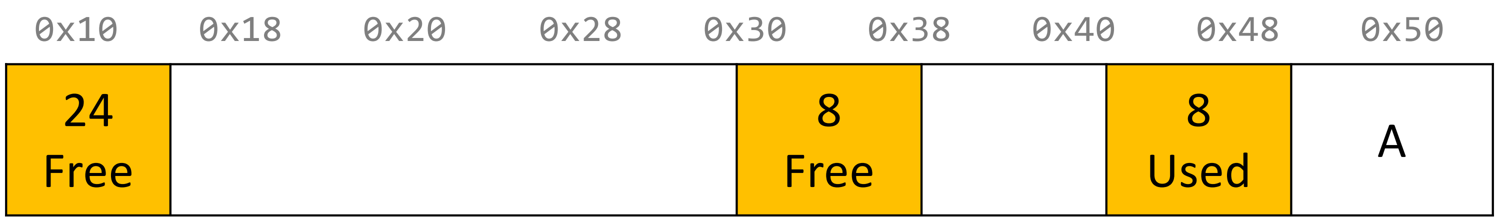



```
void *b = malloc(8);
```



Practice 2: Implicit (best-fit)

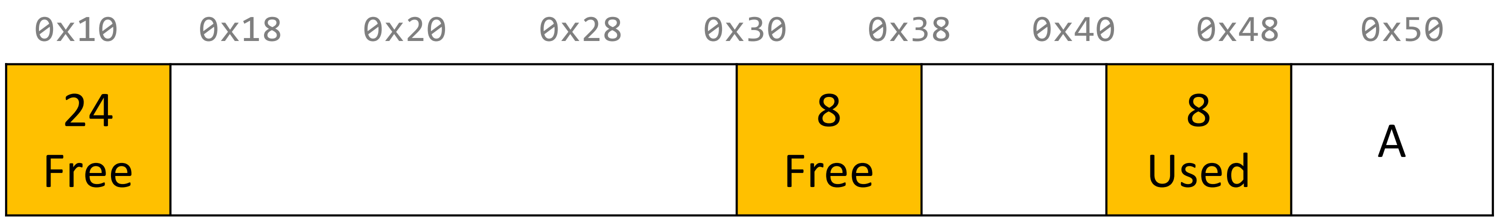
For the following heap layout, what would the heap look like after the following request is made, assuming we are using an **implicit** free list allocator with a **best-fit** approach?



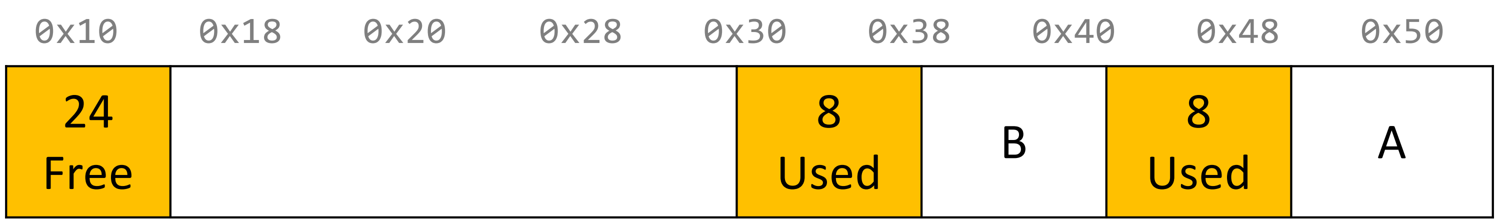

```
void *b = malloc(8);
```

Practice 2: Implicit (best-fit)

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?




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



Final Assignment: Implicit Allocator

- **Must have** headers that track block information (size, status in-use or free) – you must use the 8 byte header size, storing the status using the free bits (this is larger than the 4 byte headers used in the textbook, as this makes it easier to satisfy alignment constraints and store information in 64-bit systems).
- **Must allow,** when possible, free blocks to be recycled and reused for subsequent **malloc** requests
- **Must have** a **malloc** implementation that searches the heap for free blocks via its implicit list (i.e., traverses block-by-block).

- **Does not need to** coalesce free blocks.
- **Does not need to** support in-place **realloc**.

Coalescing

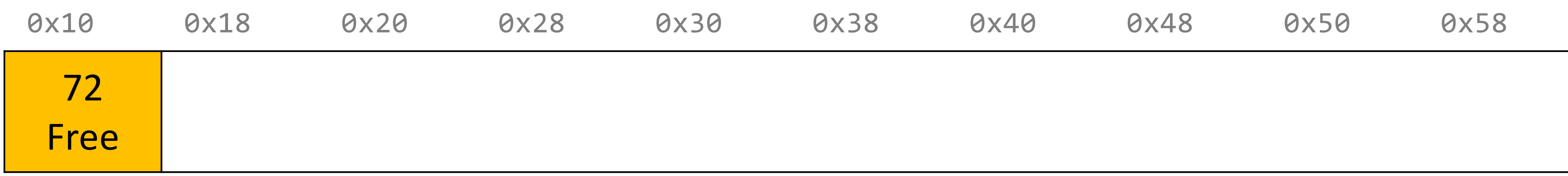
```
void *e = malloc(24);   // returns NULL!
```

You do not need to worry about this problem for the implicit allocator, but this is a requirement 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			

Supporting In-Place Realloc

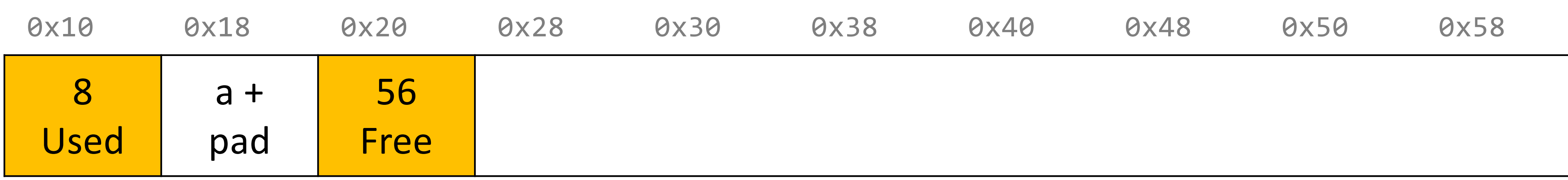
```
void *a = malloc(4);
void *b = realloc(a, 8);
```



Supporting In-Place Realloc

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

Variable	Value
a	0x18



Supporting 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**.

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 **ra**?