CS107, Lecture 22

Managing The Heap, Take I

Reading: B&O 9.9 and 9.11

Ed Discussion: https://edstem.org/us/courses/28214/discussion/2149691

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.

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

How do malloc/realloc/free work?

Pulling together all our CS107 topics this quarter:

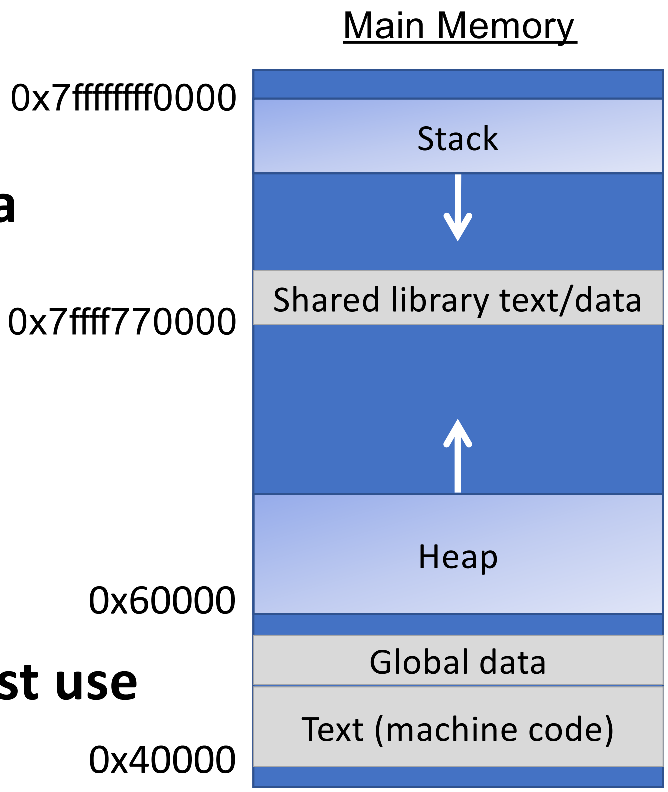
- Testing
- Efficiency
- Bit-level manipulation
- Memory management
- Pointers
- Generics
- Assembly
- And more…

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

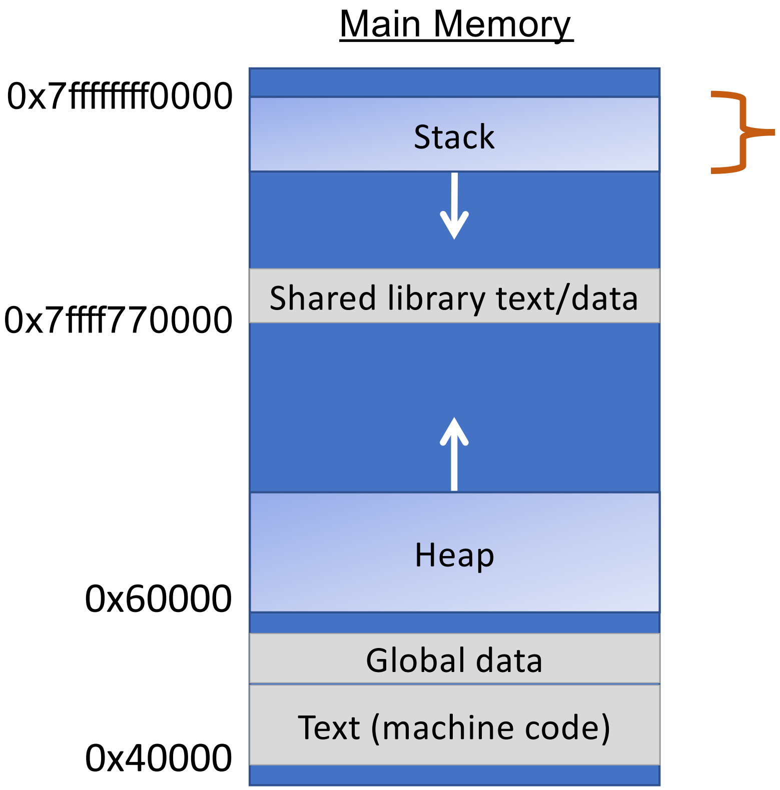
Running a program

- **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, initialize **%rsp**, **callq main**
- **malloc written in C, heap will initialize itself on first use**
  Asks OS for large memory region,
  parcels out to service requests



The Stack

Review

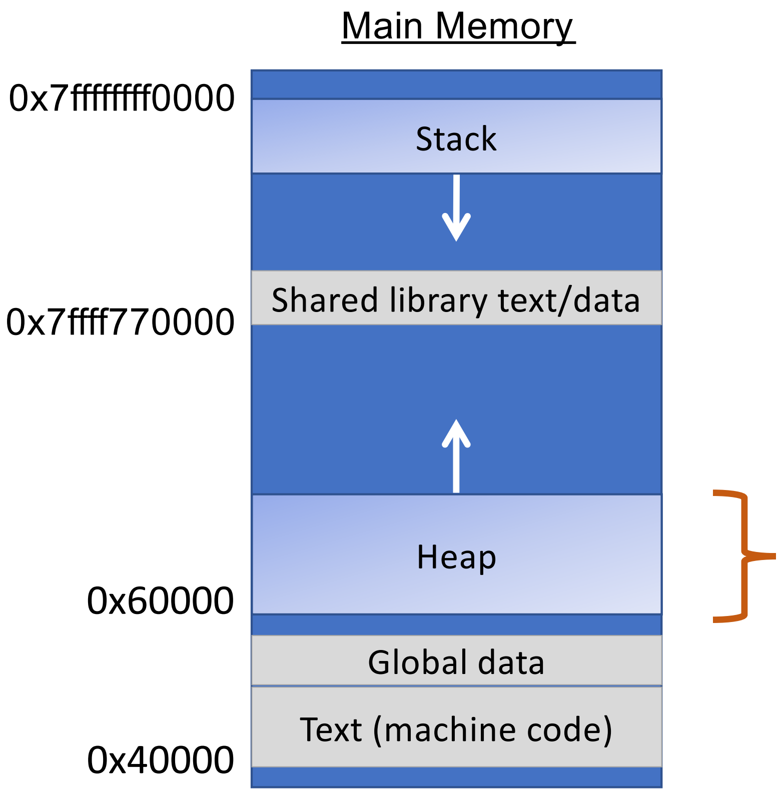


Stack memory "goes away" after function call ends.

Automatically managed at compile-time by gcc

From Assembly:
Stack management amounts to moving **%rsp** up and down (pushq, popq, mov)

Today: The Heap



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?

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);`

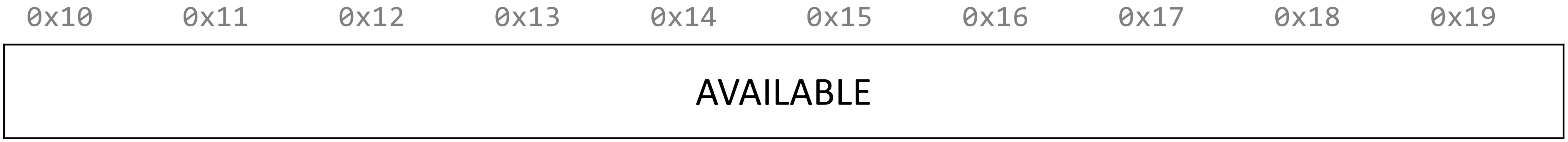
Frees the heap-allocated block starting at the specified address.

`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 address of new, larger allocated memory region. `realloc(NULL, size)` -> `malloc(size)`

What is a heap allocator?

- A heap allocator is a suite of functions that cooperatively fulfill requests for dynamically allocated memory.
- When initialized, a heap allocator tracks the base address and the size of a large contiguous block of memory. That block of memory is the heap.



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

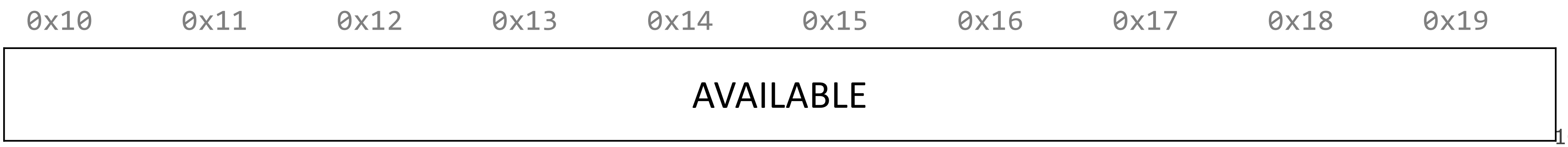
AVAILABLE

What is a heap allocator?

- A heap allocator is a suite of functions that cooperatively fulfill requests for dynamically allocated memory.
- When initialized, a heap allocator tracks the base address and the size of a large contiguous block of memory. That block of memory is the heap.
- The allocator manages the heap as clients request or donate back pieces of it.

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

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

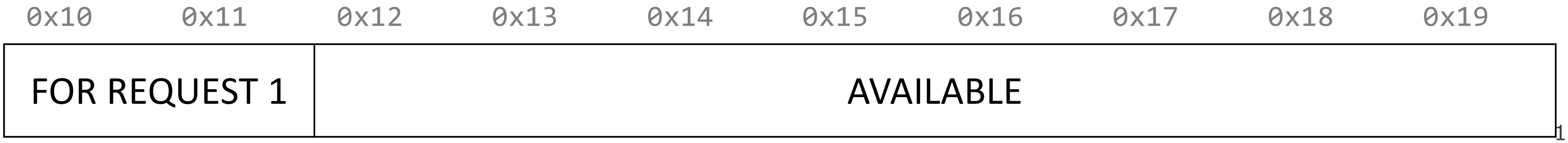


What is a heap allocator?

- A heap allocator is a suite of functions that cooperatively fulfill requests for dynamically allocated memory.
- When initialized, a heap allocator tracks the base address and the size of a large contiguous block of memory. That block of memory is the heap.
- The allocator manages the heap as clients request or donate back pieces of it.

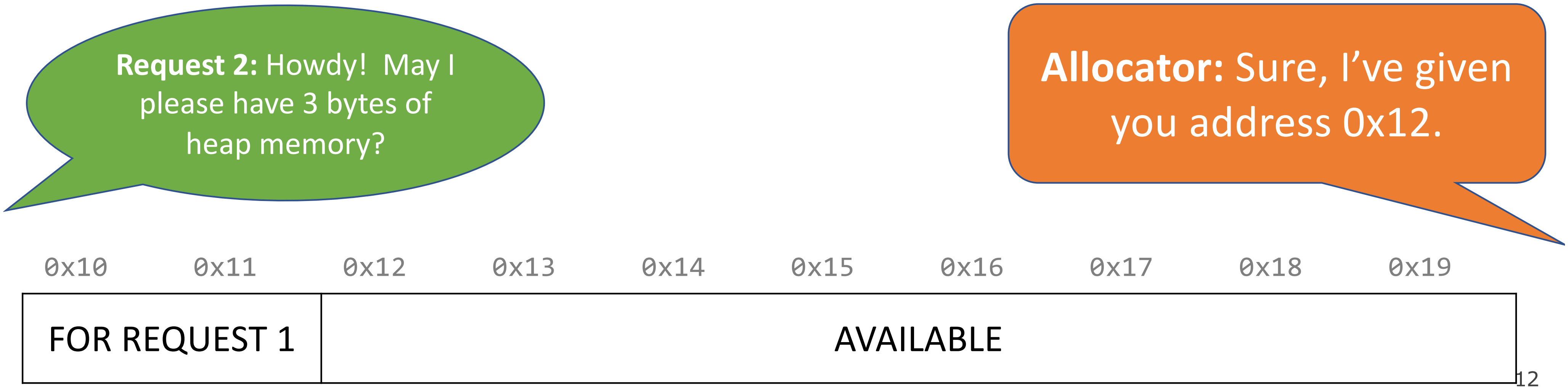
Request 1: Hi! May I please have 2 bytes of heap memory?

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



What is a heap allocator?

- A heap allocator is a suite of functions that cooperatively fulfill requests for dynamically allocated memory.
- When initialized, a heap allocator tracks the base address and the size of a large contiguous block of memory. That block of memory is the heap.
- The allocator manages the heap as clients request or donate back pieces of it.



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

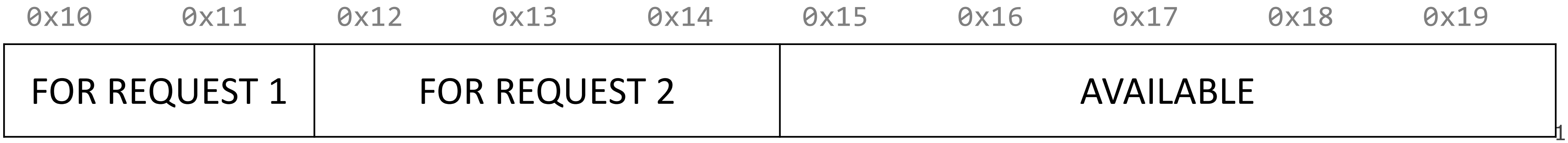
Allocator: Sure, I’ve given you address 0x12.

What is a heap allocator?

- A heap allocator is a suite of functions that cooperatively fulfill requests for dynamically allocated memory.
- When initialized, a heap allocator tracks the base address and the size of a large contiguous block of memory. That block of memory is the heap.
- The allocator manages the heap as clients request or donate back pieces of it.

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

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

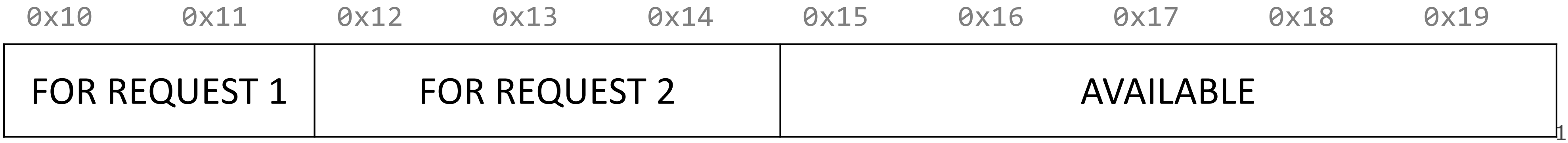


What is a heap allocator?

- A heap allocator is a suite of functions that cooperatively fulfill requests for dynamically allocated memory.
- When initialized, a heap allocator tracks the base address and the size of a large contiguous block of memory. That block of memory is the heap.
- The allocator manages the heap as clients request or donate back pieces of it.

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

Allocator: Thanks. Have a good day!



What is a heap allocator?

- A heap allocator is a suite of functions that cooperatively fulfill requests for dynamically allocated memory.
- When initialized, a heap allocator tracks the base address and the size of a large contiguous block of memory. That block of memory is the heap.
- The allocator manages the heap as clients request or donate back pieces 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 suite of functions that cooperatively fulfill requests for dynamically allocated memory.
- When initialized, a heap allocator tracks the base address and the size of a large contiguous block of memory. That block of memory is the heap.
- The allocator manages the heap as clients request or donate back pieces of it.

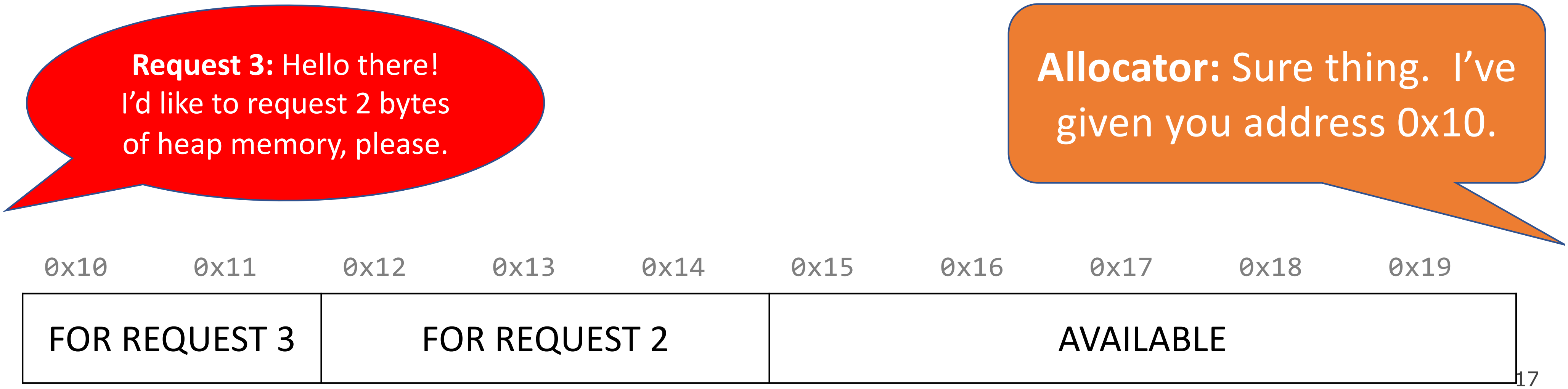
Request 3: Hello there! I'd like to request 2 bytes of heap memory, please.

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 suite of functions that cooperatively fulfill requests for dynamically allocated memory.
- When initialized, a heap allocator tracks the base address and the size of a large contiguous block of memory. That block of memory is the heap.
- The allocator manages the heap as clients request or donate back pieces of it.

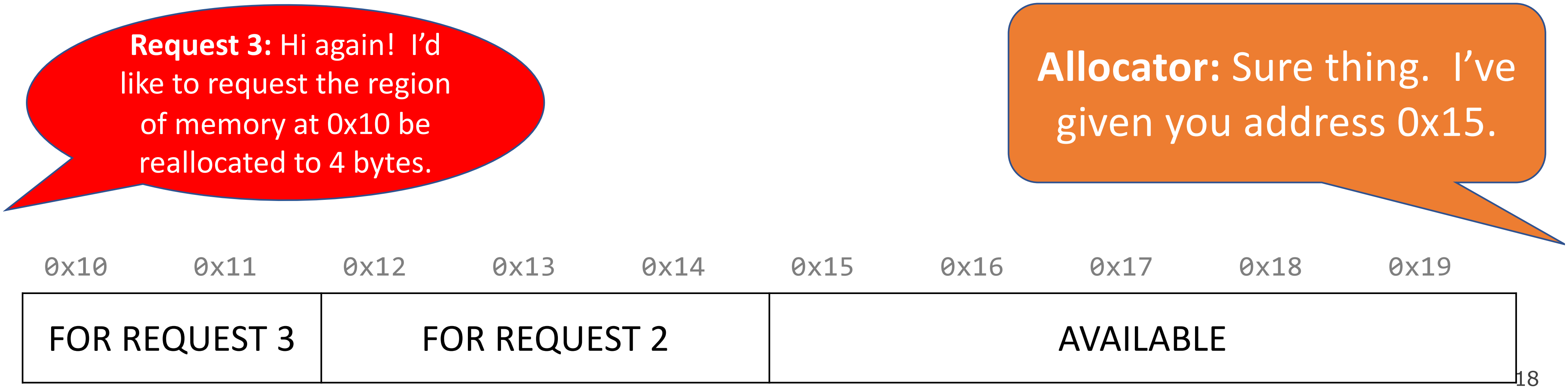


Request 3: Hello there! I'd like to request 2 bytes of heap memory, please.

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

What is a heap allocator?

- A heap allocator is a suite of functions that cooperatively fulfill requests for dynamically allocated memory.
- When initialized, a heap allocator tracks the base address and the size of a large contiguous block of memory. That block of memory is the heap.
- The allocator manages the heap as clients request or donate back pieces of it.



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

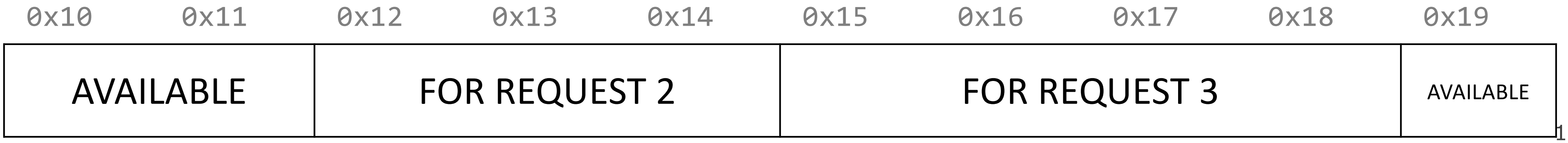
Allocator: Sure thing. I've given you address 0x15.

What is a heap allocator?

- A heap allocator is a suite of functions that cooperatively fulfill requests for dynamically allocated memory.
- When initialized, a heap allocator tracks the base address and the size of a large contiguous block of memory. That block of memory is the heap.
- The allocator manages the heap as clients request or donate back pieces of 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.



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 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.

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 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)

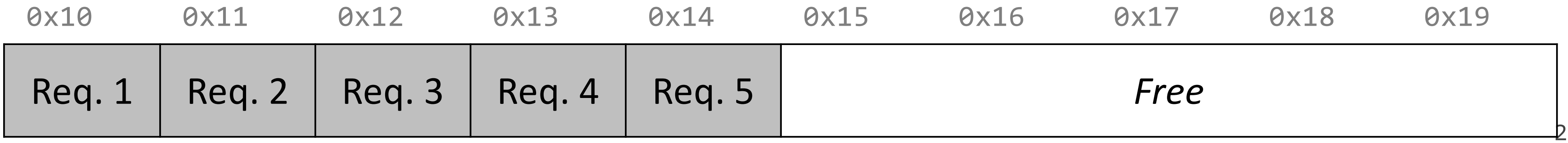
C. NO, it can't be done!

`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’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 these two goals!

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 means we do not care about reusing memory. How could we do this?

Bump Allocator Performance

1. Utilization

😱

Never reuses memory

2. Throughput

⭐

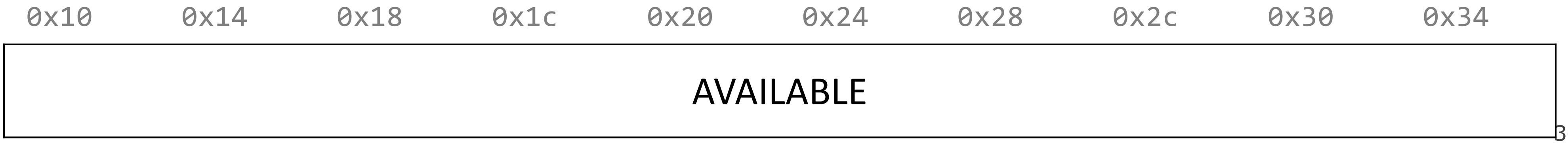
Ultra fast, short routines

Bump Allocator

- A **bump allocator** is 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** 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);
```



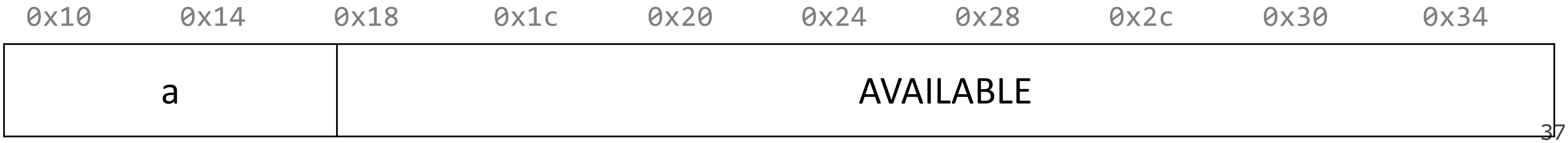
0x10 0x14 0x18 0x1c 0x20 0x24 0x28 0x2c 0x30 0x34

AVAILABLE

Bump Allocator

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

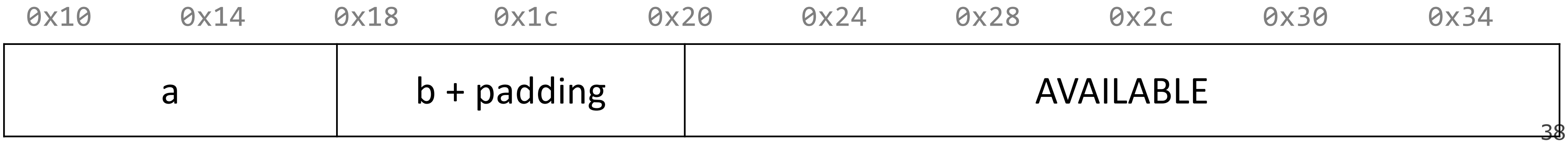
Variable	Value
a	0x10



Bump Allocator

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

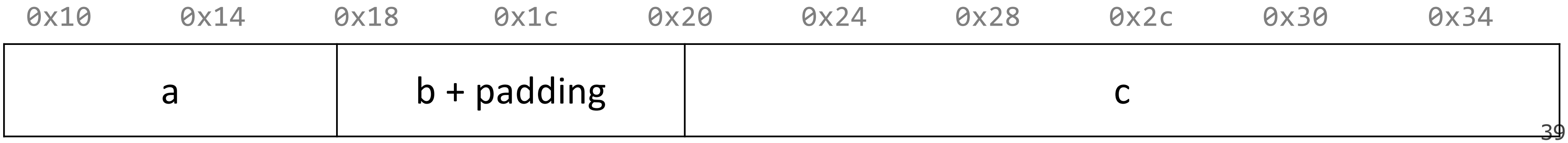
Variable	Value
a	0x10
b	0x18



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`



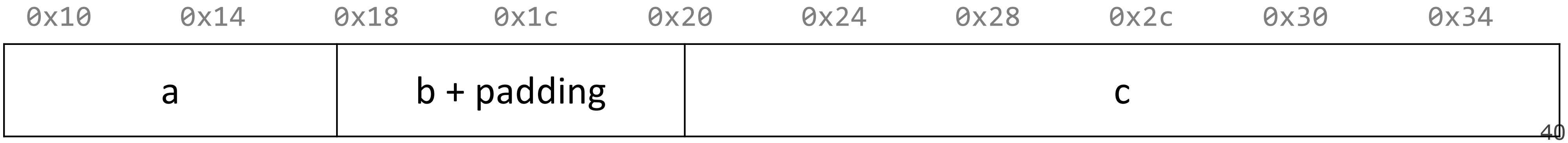
`0x10` `0x14` `0x18` `0x1c` `0x20` `0x24` `0x28` `0x2c` `0x30` `0x34`

a | b + padding | c

Bump Allocator

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

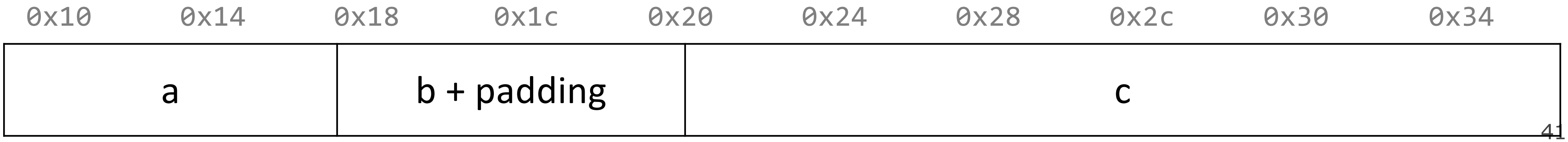
Variable	Value
a	0x10
b	0x18
c	0x20



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`



Summary: Bump Allocator

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

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?

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 and requires additional 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 reflect it is 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

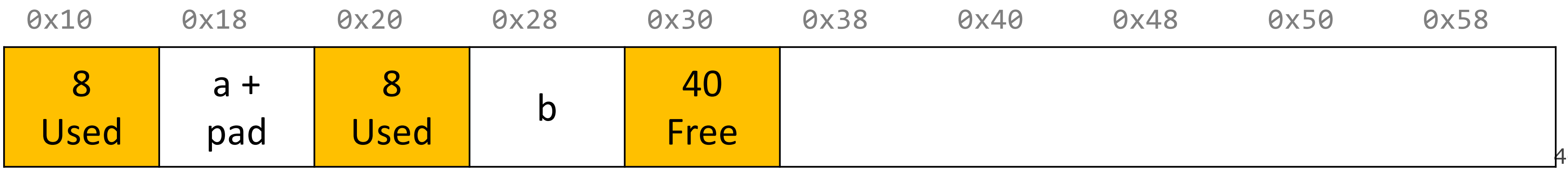
```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```

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

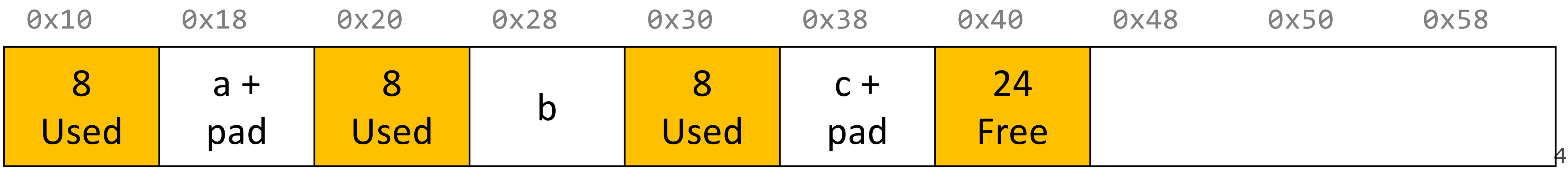


0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58
8 Used	a + pad	8 Used	b	40 Free					

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

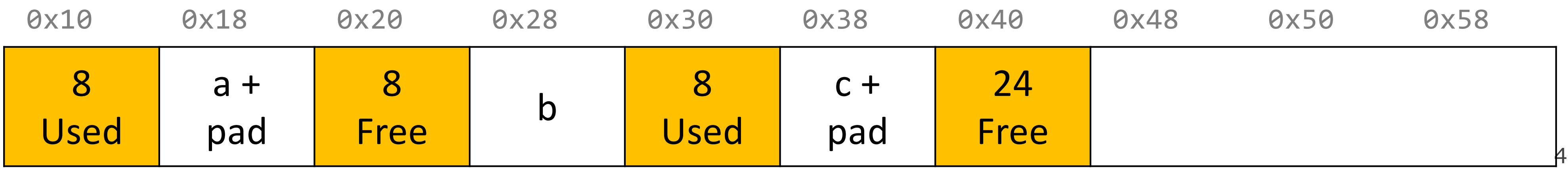


0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58
8 Used	a + pad	8 Used	b	8 Used	c + pad	24 Free			

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

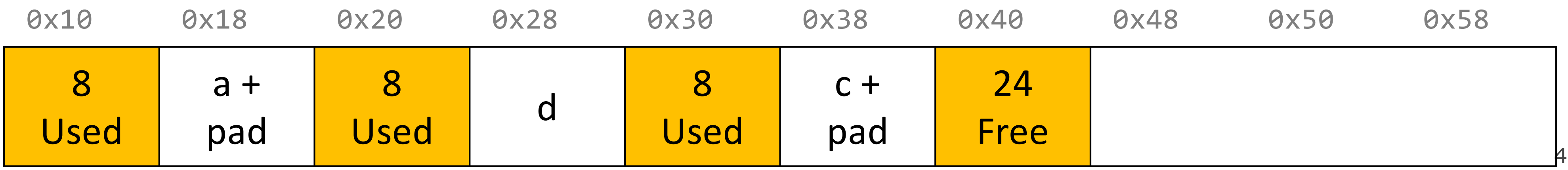


0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58
8 Used	a + pad	8 Free	b	8 Used	c + pad	24 Free			

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

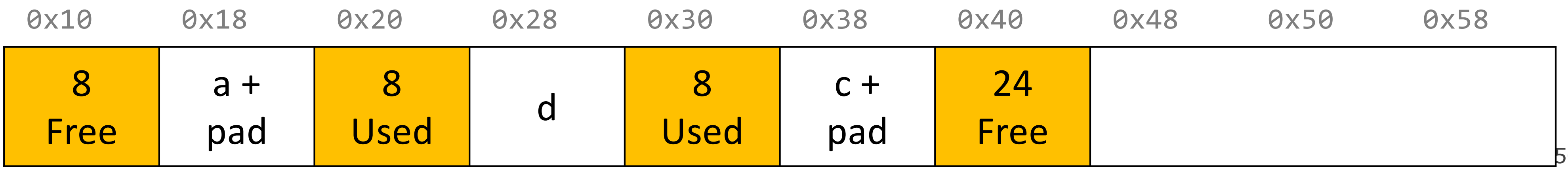


0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58
8 Used	a + pad	8 Used	d	8 Used	c + pad	24 Free			

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

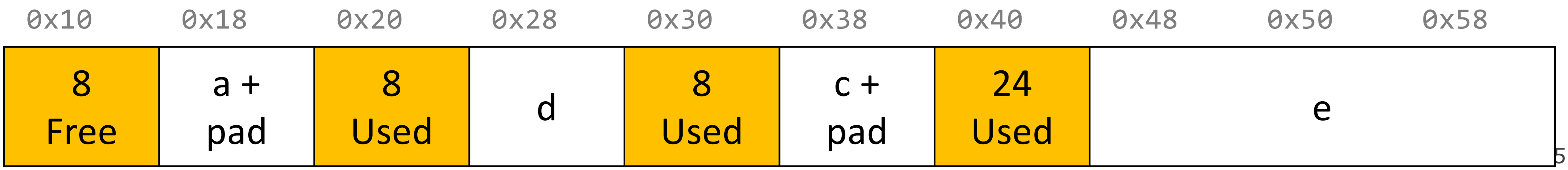


0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58
8 Free	a + pad	8 Used	d	8 Used	c + pad	24 Free			

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

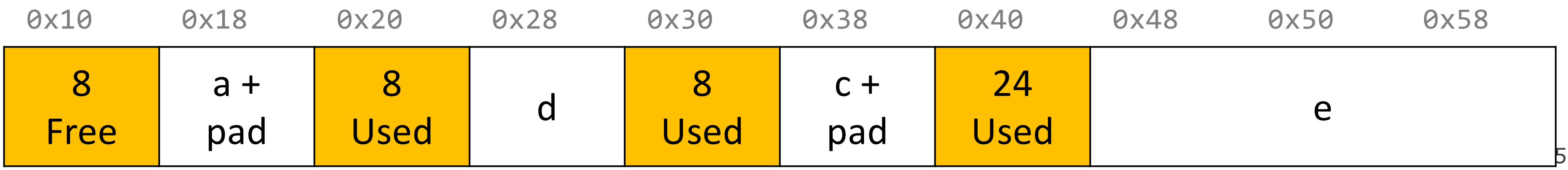


0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58
8 Free	a + pad	8 Used	d	8 Used	c + pad	24 Used	e		

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



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 will be unused!
- *Solution:* use one of the 3 least-significant bits to store free/allocated status

Implicit Free List Allocator

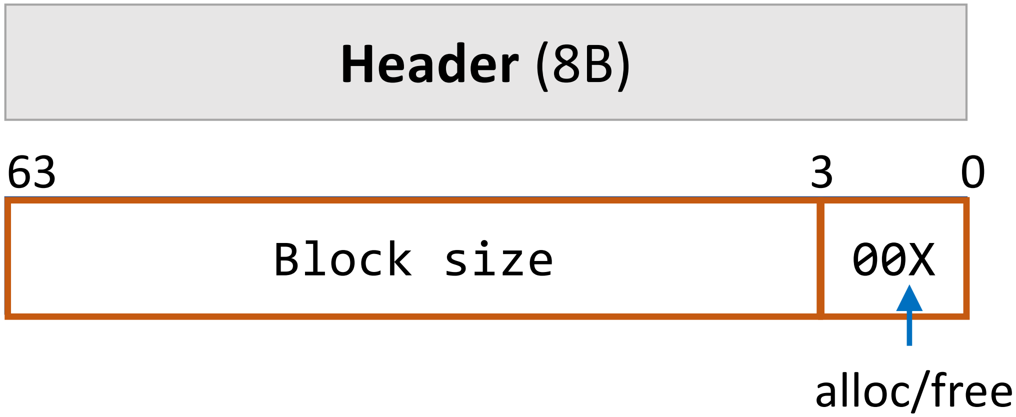
- How can we choose a free block to use for an allocation request?
  - **First fit:** search the list from beginning each time and choose first free block that fits.
  - **Next fit:** instead of starting at the beginning, continue where previous search left off.
  - **Best fit:** examine every free block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

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 ☺

Up to you!

Implicit free list header design

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

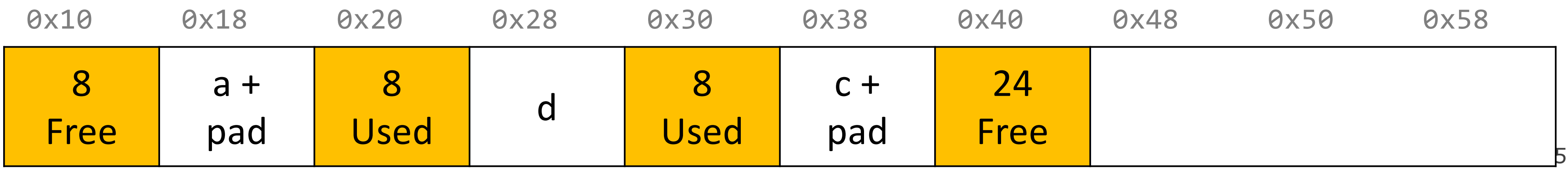
Up to you! Your decision affects how you traverse the list (be careful of off-by-one)

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?**

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58
8 Free	a + pad	8 Used	d	8 Used	c + pad	16 Used	e		???

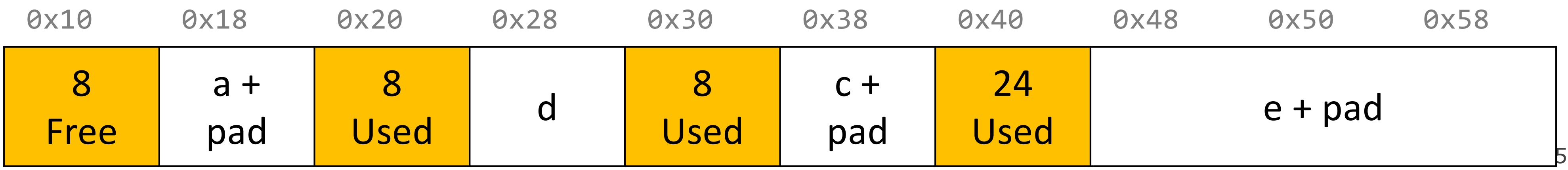
Up to you!

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!

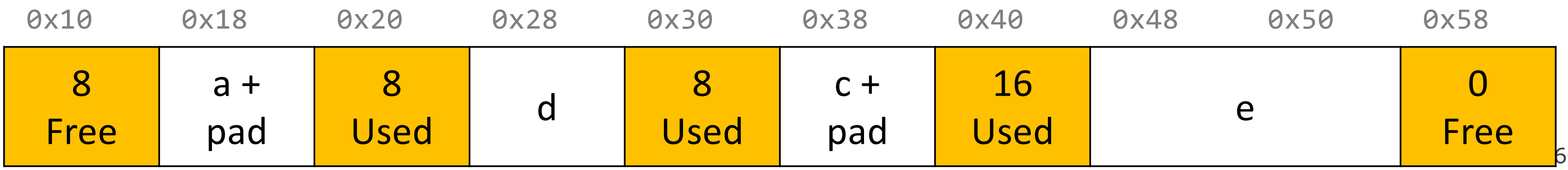
Splitting Policy

```
...
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

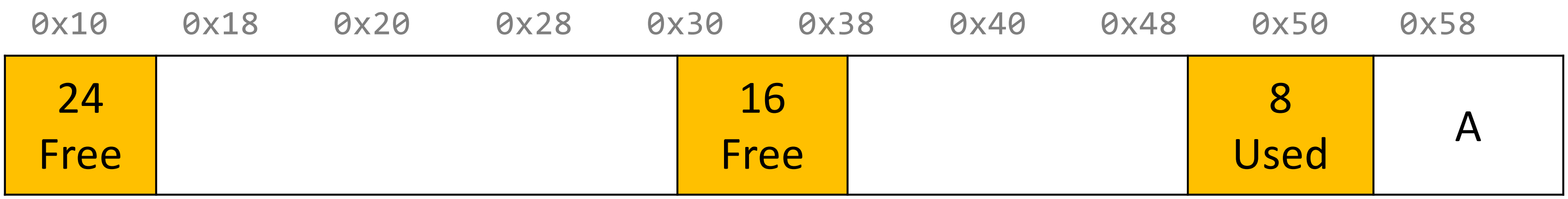
Revisiting Our Goals

Questions we considered:

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

Practice 1: Implicit (first-fit)

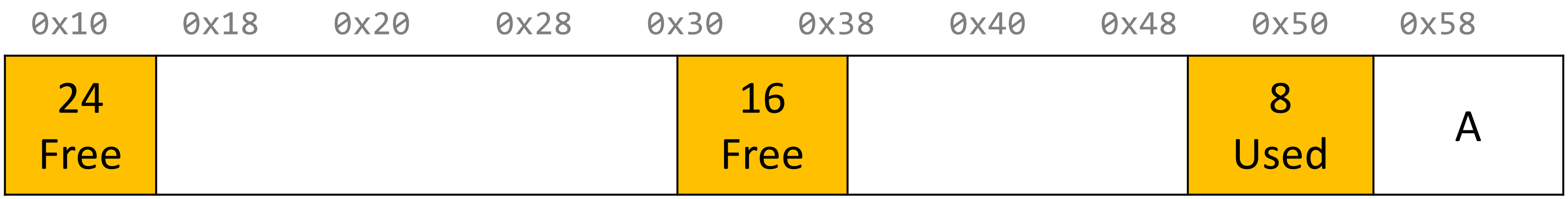
For the following heap layout, what would the heap look like after the following request is made, assuming we are using an **implicit** free list allocator with a **first-fit** approach?



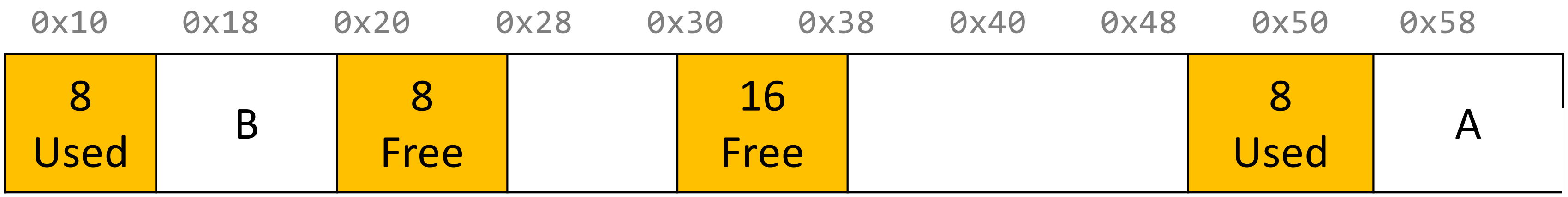
```
void *b = malloc(8);
```

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);
```



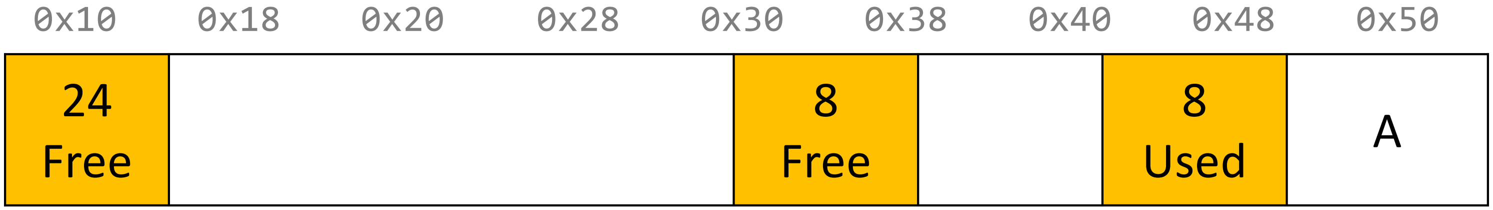
Practice 2: Implicit (best-fit)

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?

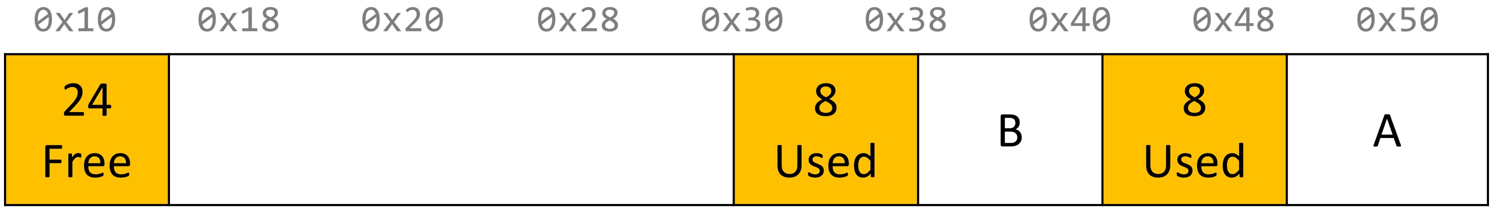
```
void *b = malloc(8);
```

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 specified in the book, as this makes it easier to satisfy the alignment constraint and store information).
- **Must have** free blocks that are recycled and reused for subsequent malloc requests if possible
- **Must have** a malloc implementation that searches the heap for free blocks via an implicit list (i.e. traverses block-by-block).

- **Does not need to** have coalescing of 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			

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	0x18

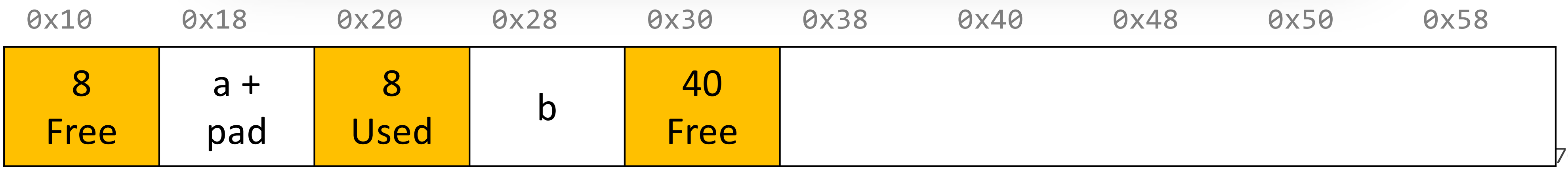
0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58
8 Used	a + pad	56 Free							

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?