

Virtual memory allocation and semaphores

Recap

Lucas Yri

KØBENHAVNS UNIVERSITET



Plan for this recap

- Semaphores (10-15 minutes)
 - Quick explanation of semaphores
 - Semaphore exercise
 - Additional semaphore exercise (if time allows)
 - Heap allocation (The rest)
 - Explaining the basics
 - 2.3.1 from exam 21/22
 - 2.3.1 from exam 20/21
 - 2.3.1 from exam 22/23 (if time)
 - Why?
 - (Warning, made in paint)
-
- There is no specific time for questions, raise hands as needed

The exercises in question

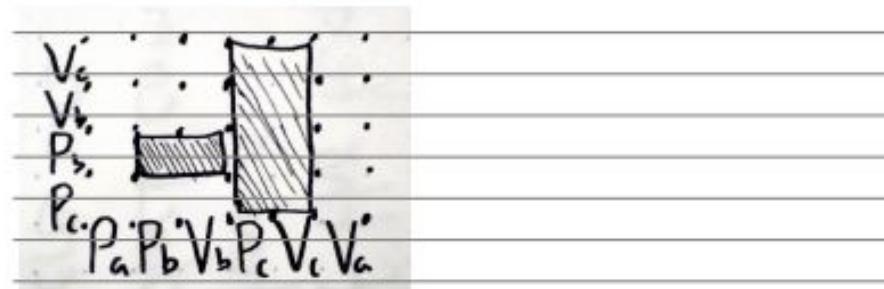
2.2 Short Questions (about 12 %)

Question 2.2.1: Consider two threads performing the following semaphore operations:

Initially: a = 1; b = 1; c = 1

| | |
|-----------|-----------|
| Thread 1: | Thread 2: |
| P(a); | P(c); |
| P(b); | P(b); |
| V(b); | V(b); |
| P(c); | V(c); |
| V(c); | |
| V(a); | |

Draw a process graph with thread 1 along the horizontal axis and the other threads along the vertical axis. Show the forbidden regions, and argue whether this means deadlocks are possible.



| Address | Original value | After realloc | After free |
|-----------|----------------|---------------|------------|
| 0x12c028 | 0x00000012 | [REDACTED] | [REDACTED] |
| 0x12c024 | 0x012c611c | 0x012c611c | 0x012c611c |
| 0x12c020 | 0x012c512c | 0x012c512c | 0x012c512c |
| 0x12c01c | 0x00000012 | [REDACTED] | [REDACTED] |
| 0x12c018 | 0x00000023 | [REDACTED] | [REDACTED] |
| 0x12c014 | 0x012c511c | 0x012c511c | 0x012c511c |
| 0x12c010 | 0x012c601c | 0x012c601c | 0x012c601c |
| 0x12c00c | 0x00000000 | [REDACTED] | [REDACTED] |
| 0x12c008 | 0x00000000 | [REDACTED] | [REDACTED] |
| 0x12c004 | 0x012c601c | 0x012c601c | 0x012c601c |
| 0x12c000 | 0x012c511c | 0x012c511c | 0x012c511c |
| 0x12bfffc | 0x00000023 | [REDACTED] | [REDACTED] |

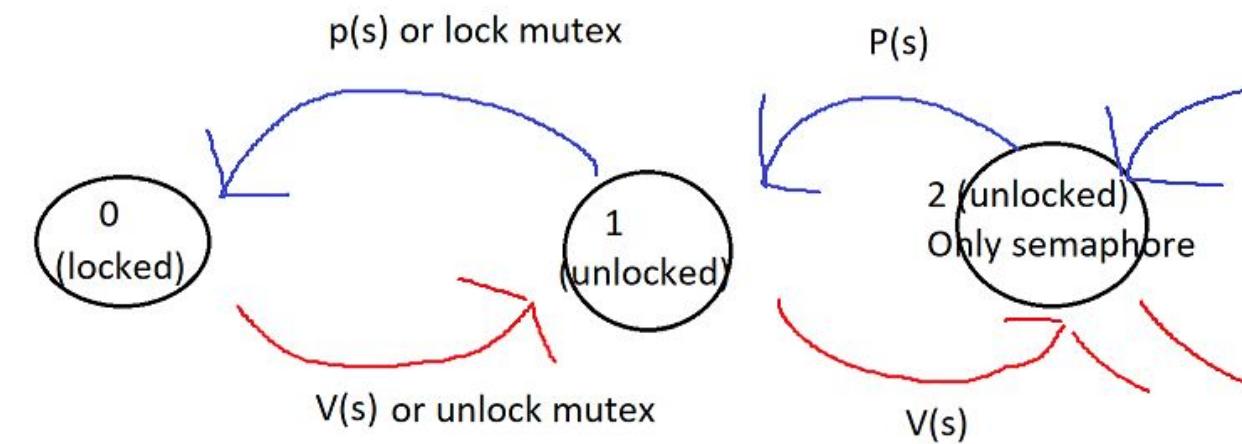
Semaphores

- Kind of like a combination of a mutex and a counter
- P(s) decrements
- V(s) increments
- Locked if 0 (memory is inaccessible)
- For example if the semaphore is 5, it would need 5 calls of P(s) to become locked.
- A mutex is a type of semaphore that is only 0 (locked) or 1 (unlocked).

```
wait(S) { // originally called P(S)  
    while ( S ≤ 0 ) do no-op;  
    S --;  
}
```

Generally implemented without busy waiting using *sleep()* and *wakeup()*.

```
signal(S) { // originally called V(S)  
    S++;  
}
```



Re-Exam 21/22 - how to solve?

2.2 Short Questions (about 12 %)

Question 2.2.1: Consider two threads performing the following semaphore operations.

Initially: $a = 1; b = 1; c = 1;$

Thread 1: Thread 2:

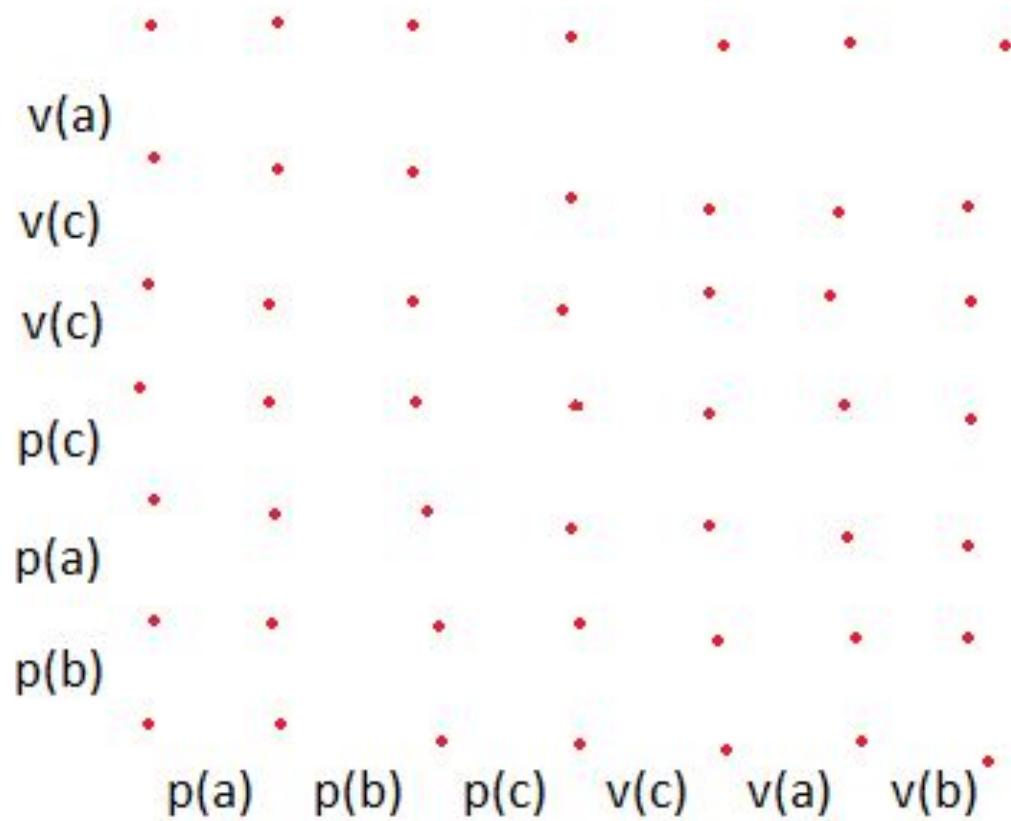
| | |
|-------|-------|
| P(a); | P(b); |
| P(b); | P(a); |
| P(c); | P(c); |
| V(c); | V(c); |
| V(a); | V(b); |
| V(b); | V(a); |

Draw a process graph with thread 1 along the horizontal axis and thread 2 along the vertical axis, show the forbidden regions, and argue whether this means deadlocks are possible or not. (You are welcome to attach the figure as an image in the handin.)

(Warning)

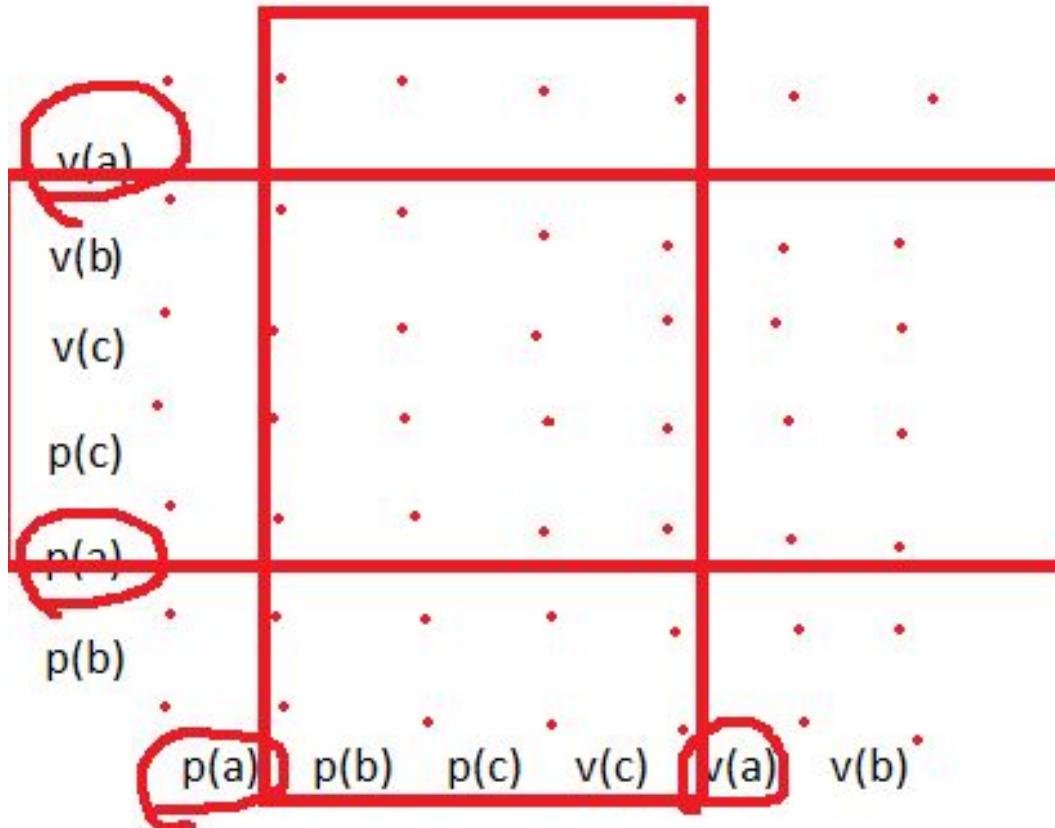
- I accidentally swapped thread 1 and thread 2 in the exercises I will show. It does not alter the correctness, just keep it in mind. You might also want to have it correct in the exam

Re-Exam 21/22 - how to solve?



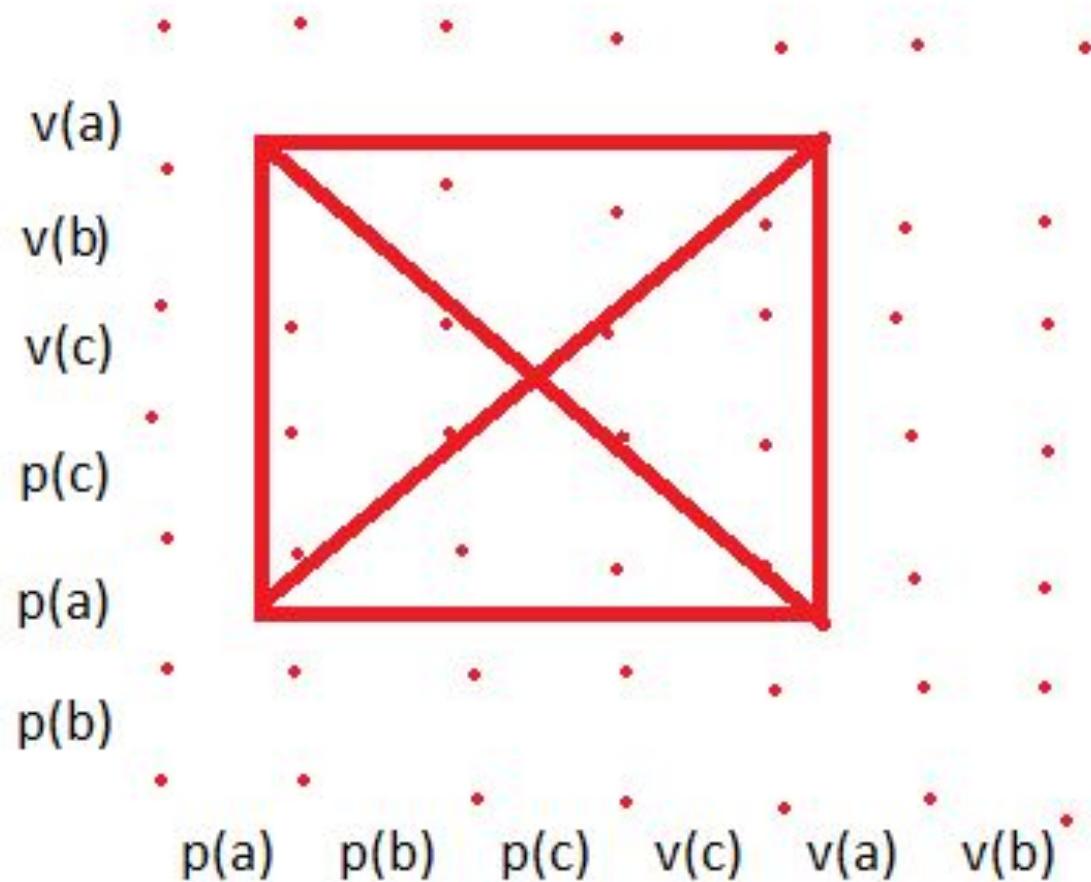
- One thread along the bottom
- The other thread along the top
- If moving right, one thread is progressing
- If moving up, the other thread is progressing
- Every p() or v() is a transition from 1 dot to another, where that action now happened

Re-Exam 21/22 - how to solve?



- Where is variable a locked?
- from p(a) until v(a)
- The place where both of them lock variable a is forbidden, simply put, we cannot go there on the graph as the semaphore does not allow 2 to access at the same time.

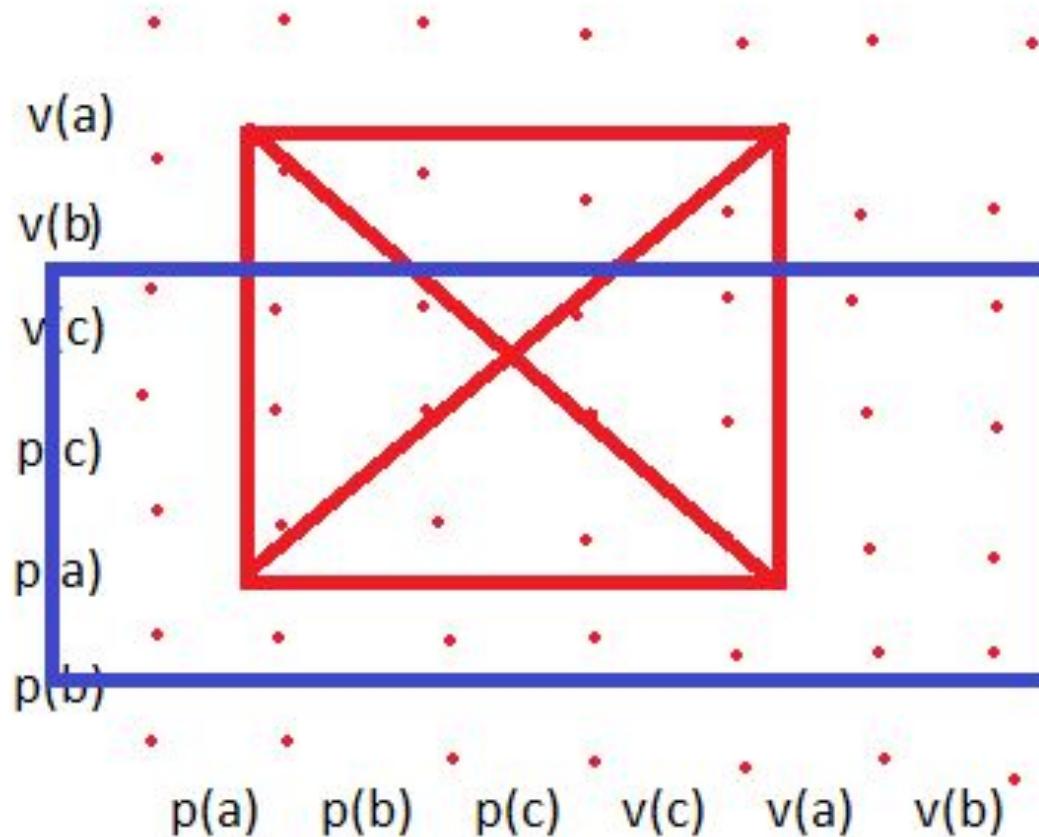
Re-Exam 21/22 - how to solve?



- Where is variable a locked?
- from $p(a)$ until $v(a)$
- The place where both of them lock variable a is forbidden, simply put, we cannot go there on the graph as the semaphore does not allow 2 to access at the same time.

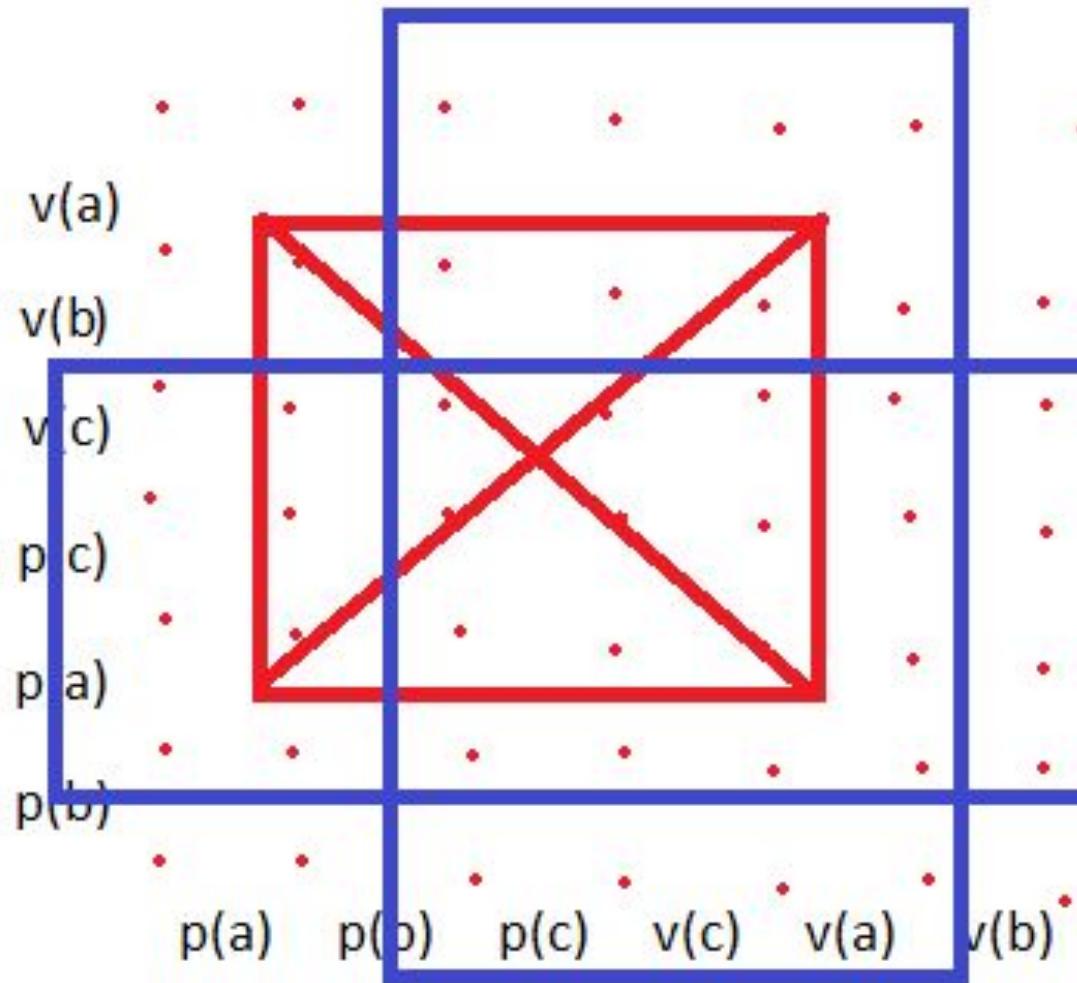
Re-Exam 21/22 - how to solve?

- Where is variable b locked?



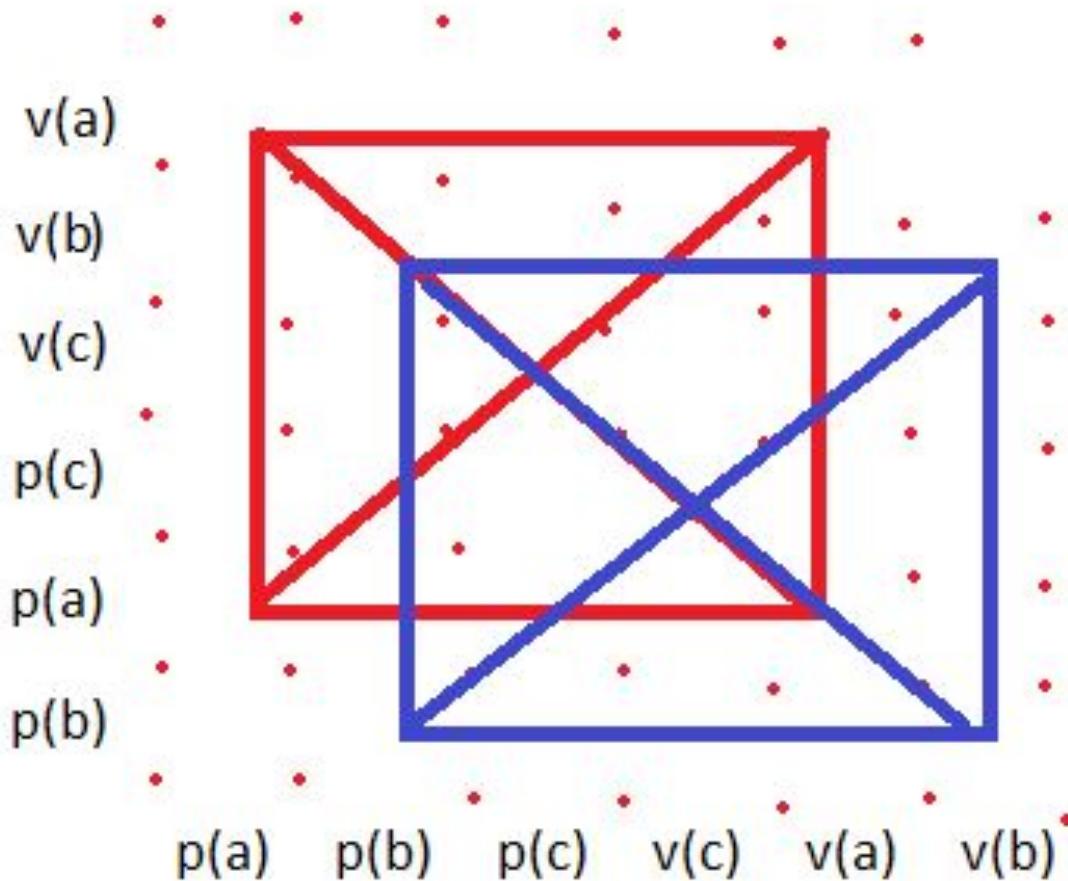
Re-Exam 21/22 - how to solve?

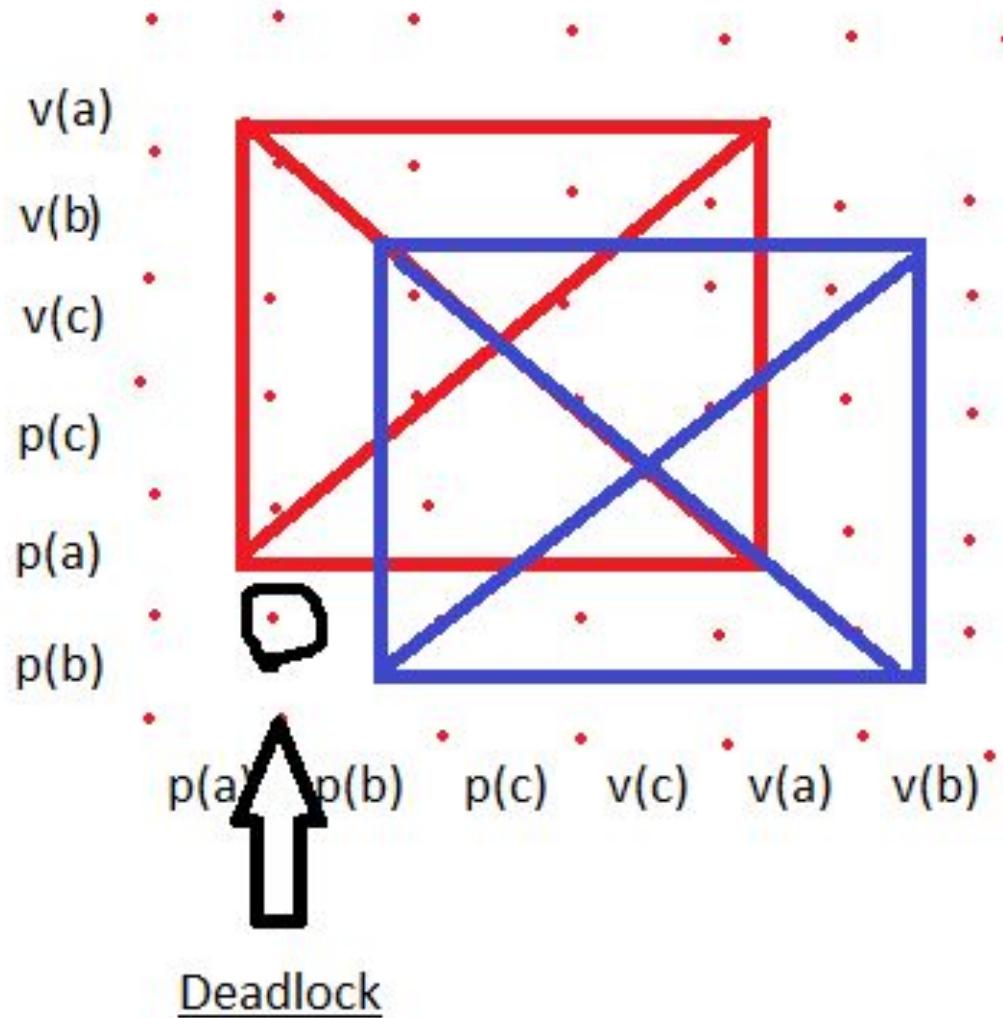
- Where is variable b locked?



Re-Exam 21/22 - how to solve?

- Where is variable b locked?

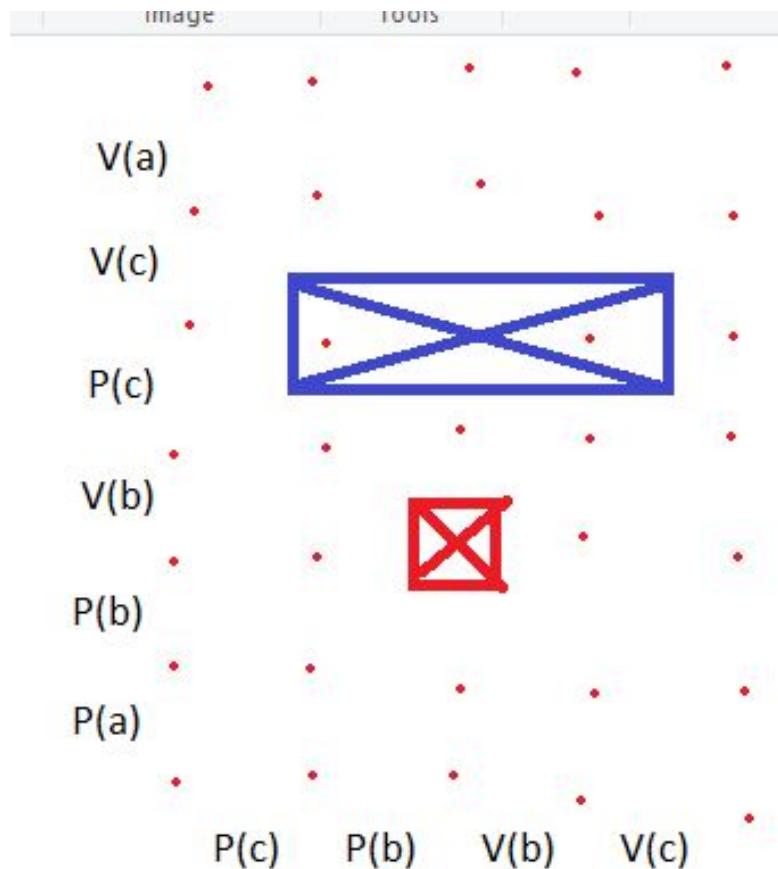




- The question was whether a deadlock existed
- Argumentation: “A deadlock is possible, as in this spot, you may go neither up nor right”
- (Checking c not necessary)

Another (quick) example from exam 21/22

2.2 Short Questions (about 12 %)



Question 2.2.1: Consider two threads per:

Initially: $a = 1; b = 1; c = 1;$

Thread 1:

$P(a);$

$P(b);$

$V(b);$

$P(c);$

$V(c);$

$V(a);$

Thread 2:

$P(c);$

$P(b);$

$V(b);$

$V(c);$

Can always proceed right

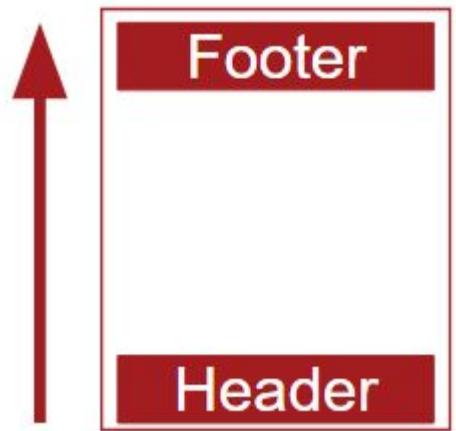
or up = no deadlock exists

(Again, I swapped the
threads, oops)

Now onto the heap

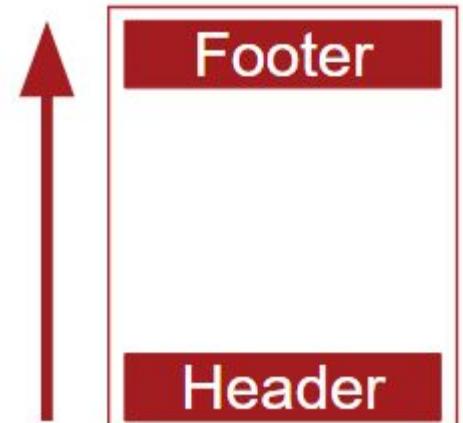
Firstly...

- The heap grows bottom-up



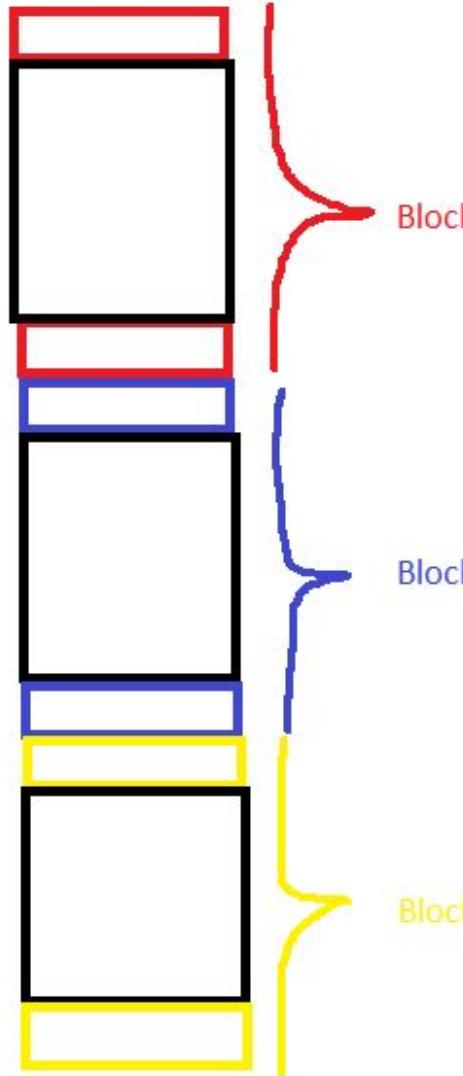
Firstly...

- The heap grows bottom-up
- The footer and header contain information about the block including its size. Since they represent the same block the footer and header have the same value
- The size of a block is always a multiple of 8
- Each address represents 4 bytes of space



A heap might look like this

Footer 2->



Header 2->
Footer 1->

Header 1->
Footer 0->

Header 0 ->

Address Original value

0x500c028 0x00000013

0x500c024 0x500c611c

0x500c020 0x500c512c

0x500c01c 0x00000013

0x500c018 0x00000013

0x500c014 0x500c511c

0x500c010 0x500c601c

0x500c00c 0x00000013

0x500c008 0x00000013

0x500c004 0x500c601c

0x500c000 0x500c511c

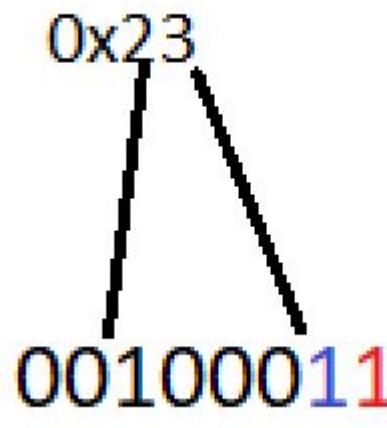
0x500bffc 0x00000013

What do the bits represent?

- (In general always check the rules, they might change)
- Bit 0 represents whether the current block is allocated
- Bit 1 represents whether the previous block is allocated
- Bit 2 is always 0
- Instant coalescing (explained more later)
- Example:

What do the bits represent?

- (In general always check the rules, they might change)
- Bit 0 represents whether the current block is allocated'
- Bit 1 represents whether the previous block is allocated
- Bit 2 is always 0
- Instant coalescing (explained more later)
- Example:

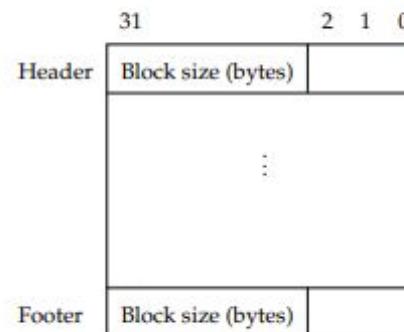


- Is allocated
- Previous is allocated
- Size is gotten by setting the 3 lower bits to 0:
 $00100011 \rightarrow 00100000 = 0x20$ or 32
- (Size is inclusive header and footer)

Let's get started (Exam 21/22)

2.3 Long Questions (about 14 %)

Question 2.3.1: Consider an allocator that uses an implicit free list and immediate coalescing of neighbouring free blocks. The layout of each allocated and free memory block is as follows, with one 32-bit word per row:



Each memory block, either allocated or free, has a size that is a multiple of eight bytes, rounding up allocations if necessary. Thus, only the 29 higher order bits in the header and footer are needed to record block size, which includes the header and footer. The usage of the remaining 3 lower order bits is as follows:

- bit 0 indicates the use of the current block: 1 for allocated, 0 for free.
- bit 1 indicates the use of the previous adjacent block: 1 for allocated, 0 for free.
- bit 2 is unused and is always set to be 0.

Important: We must *never* create blocks with zero payload (i.e. we must *never* create blocks with size 8).

Given the heap shown on the left, show the new heap contents after *consecutive* calls to

1. `realloc(0x12c000, 8)`. Assume the the return value is `0x12c000`, and that the existing allocation is resized to be as small as possible.
2. `free(0x12c000)`.

Your answers should be given as hex values. Note that the address grows from bottom up. Assume that the allocator uses immediate coalescing, that is, adjacent free blocks are merged immediately each time a block is freed.

- You are given a `realloc()` `free()` or `malloc()`
- Given this information, along with an initial heap, transform the heap accordingly

realloc(0x12c000, 8). free(0x12c000).

| Address | Original value | After realloc | After free |
|-----------|----------------|---------------|------------|
| 0x12c028 | 0x00000012 | | |
| 0x12c024 | 0x012c611c | | |
| 0x12c020 | 0x012c512c | | |
| 0x12c01c | 0x00000012 | | |
| 0x12c018 | 0x00000023 | | |
| 0x12c014 | 0x012c511c | | |
| 0x12c010 | 0x012c601c | | |
| 0x12c00c | 0x00000000 | | |
| 0x12c008 | 0x00000000 | | |
| 0x12c004 | 0x012c601c | | |
| 0x12c000 | 0x012c511c | | |
| 0x12bfffc | 0x00000023 | | |

First element



0x12c000

Then this must
be the header



0x12bfffc

realloc(0x12c000, 8). free(0x12c000).

| | Address | Original value | After realloc | After free |
|---------------------------------|-----------|----------------|---------------|------------|
| Footer? | 0x12c028 | 0x00000012 | | |
| | 0x12c024 | 0x012c611c | 0x012c611c | 0x012c611c |
| | 0x12c020 | 0x012c512c | 0x012c512c | 0x012c512c |
| Header? | 0x12c01c | 0x00000012 | | |
| Footer? | 0x12c018 | 0x00000023 | | |
| | 0x12c014 | 0x012c511c | 0x012c511c | 0x012c511c |
| | 0x12c010 | 0x012c601c | 0x012c601c | 0x012c601c |
| | 0x12c00c | 0x00000000 | | |
| | 0x12c008 | 0x00000000 | | |
| | 0x12c004 | 0x012c601c | 0x012c601c | 0x012c601c |
| First element | 0x12c000 | 0x012c511c | 0x012c511c | 0x012c511c |
| Then this must be the header | 0x12bfffc | 0x00000023 | | |

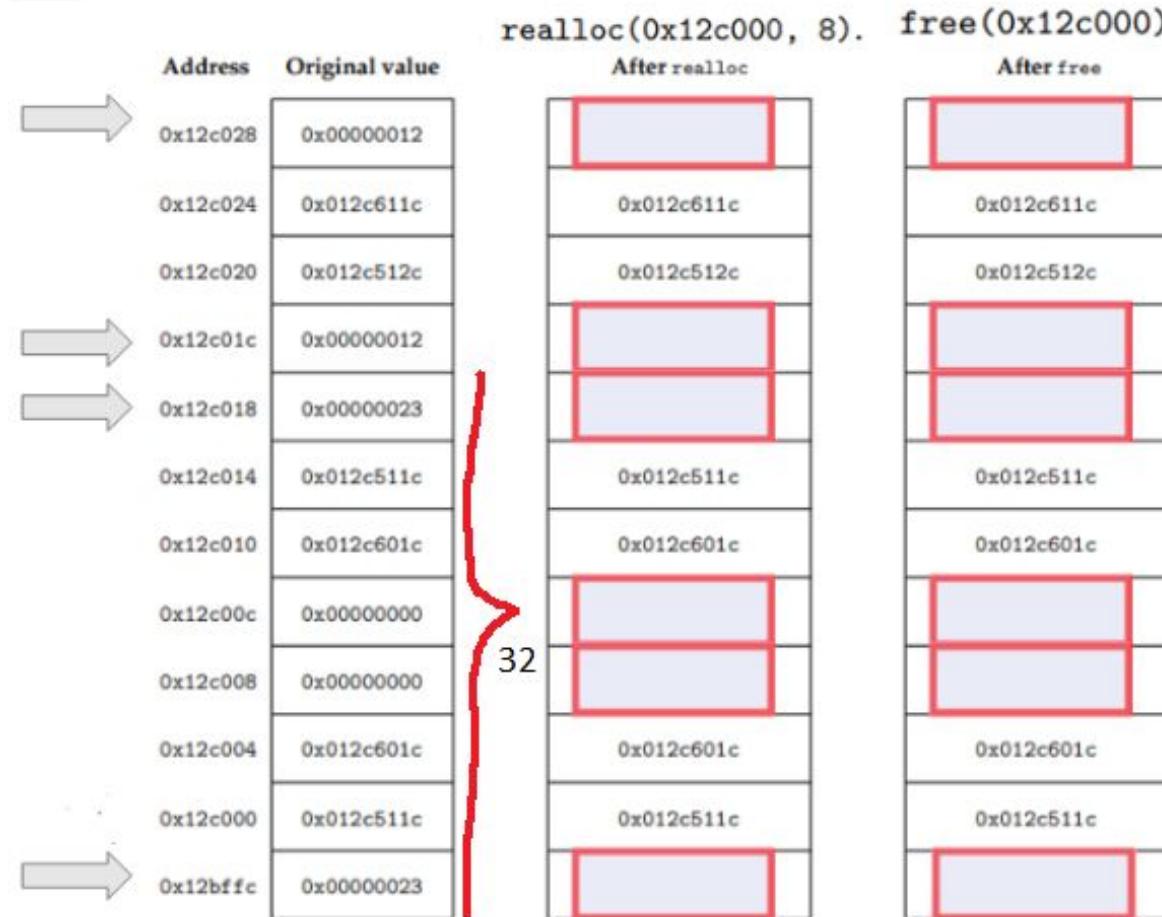
realloc(0x12c000, 8). free(0x12c000).

| | Address | Original value | After realloc | After free |
|---------------------------------|-----------|----------------|---------------|------------|
| Footer? | 0x12c028 | 0x00000012 | | |
| | 0x12c024 | 0x012c611c | | |
| | 0x12c020 | 0x012c512c | | |
| Header? | 0x12c01c | 0x00000012 | | |
| Footer? | 0x12c018 | 0x00000023 | | |
| | 0x12c014 | 0x012c511c | | |
| | 0x12c010 | 0x012c601c | | |
| | 0x12c00c | 0x00000000 | | |
| | 0x12c008 | 0x00000000 | | |
| | 0x12c004 | 0x012c601c | | |
| | 0x12c000 | 0x012c511c | | |
| First element | 0x12c000 | 0x00000023 | | |
| Then this must be the header | 0x12bfffc | | | |

| Address | Original value | After realloc | After free |
|-----------|----------------|---------------|------------|
| 0x12c028 | 0x00000012 | | |
| 0x12c024 | 0x012c611c | 0x012c611c | 0x012c611c |
| 0x12c020 | 0x012c512c | 0x012c512c | 0x012c512c |
| 0x12c01c | 0x00000012 | | |
| 0x12c018 | 0x00000023 | | |
| 0x12c014 | 0x012c511c | 0x012c511c | 0x012c511c |
| 0x12c010 | 0x012c601c | 0x012c601c | 0x012c601c |
| 0x12c00c | 0x00000000 | | |
| 0x12c008 | 0x00000000 | | |
| 0x12c004 | 0x012c601c | 0x012c601c | 0x012c601c |
| 0x12c000 | 0x012c511c | 0x012c511c | 0x012c511c |
| 0x12bfffc | 0x00000023 | | |

0x23
00100011

Is allocated
Previous is allocated
Size = 00100000 = 0x20 = 32



0x23
00100011

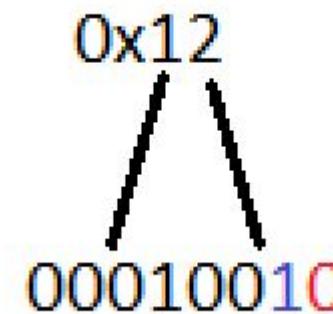
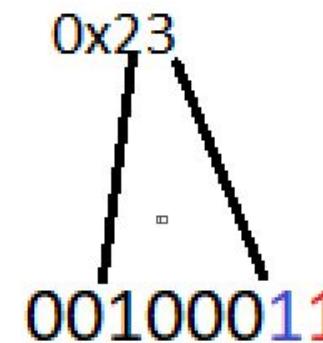
Is allocated
Previous is allocated
Size = 00100000 = 0x20 = 32

0x12
00010010

Is not allocated
Previous is allocated
Size = 00010000 = 0x10 = 16

| Address | Original value | After realloc |
|-----------|----------------|---------------|
| 0x12c028 | 0x00000012 | |
| 0x12c024 | 0x012c611c | 0x012c611c |
| 0x12c020 | 0x012c512c | 0x012c512c |
| 0x12c01c | 0x00000012 | |
| 0x12c018 | 0x00000023 | |
| 0x12c014 | 0x012c511c | 0x012c511c |
| 0x12c010 | 0x012c601c | 0x012c601c |
| 0x12c00c | 0x00000000 | |
| 0x12c008 | 0x00000000 | |
| 0x12c004 | 0x012c601c | 0x012c601c |
| 0x12c000 | 0x012c511c | 0x012c511c |
| 0x12bfffc | 0x00000023 | |

| Address | After realloc | After free |
|-----------|---------------|------------|
| 0x12c028 | | |
| 0x12c024 | 0x012c611c | 0x012c611c |
| 0x12c020 | 0x012c512c | 0x012c512c |
| 0x12c01c | | |
| 0x12c018 | | |
| 0x12c014 | 0x012c511c | 0x012c511c |
| 0x12c010 | 0x012c601c | 0x012c601c |
| 0x12c00c | | |
| 0x12c008 | | |
| 0x12c004 | 0x012c601c | 0x012c601c |
| 0x12c000 | 0x012c511c | 0x012c511c |
| 0x12bfffc | | |



Is allocated
Previous is allocated
Size = 00100000 = 0x20 = 32

Is not allocated
Previous is allocated
Size = 00010000 = 0x10 = 16

realloc(0x12c000, 8).

| Address | Original value |
|-----------|----------------|
| 0x12c028 | 0x00000012 |
| 0x12c024 | 0x012c611c |
| 0x12c020 | 0x012c512c |
| 0x12c01c | 0x00000012 |
| 0x12c018 | 0x00000023 |
| 0x12c014 | 0x012c511c |
| 0x12c010 | 0x012c601c |
| 0x12c00c | 0x00000000 |
| 0x12c008 | 0x00000000 |
| 0x12c004 | 0x012c601c |
| 0x12c000 | 0x012c511c |
| 0x12bfffc | 0x00000023 |

| Address | Original value |
|-----------|----------------|
| 0x12c028 | 0x00000012 |
| 0x12c024 | 0x012c611c |
| 0x12c020 | 0x012c512c |
| 0x12c01c | 0x00000012 |
| 0x12c018 | 0x00000023 |
| 0x12c014 | 0x012c511c |
| 0x12c010 | 0x012c601c |
| 0x12c00c | 0x00000000 |
| 0x12c008 | 0x00000000 |
| 0x12c004 | 0x012c601c |
| 0x12c000 | 0x012c511c |
| 0x12bfffc | 0x00000023 |

free(0x12c000).

0x23
00100011

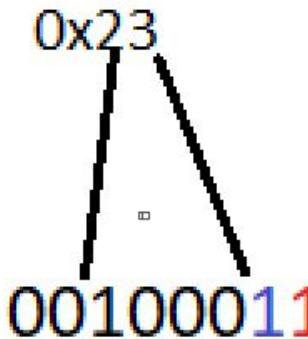
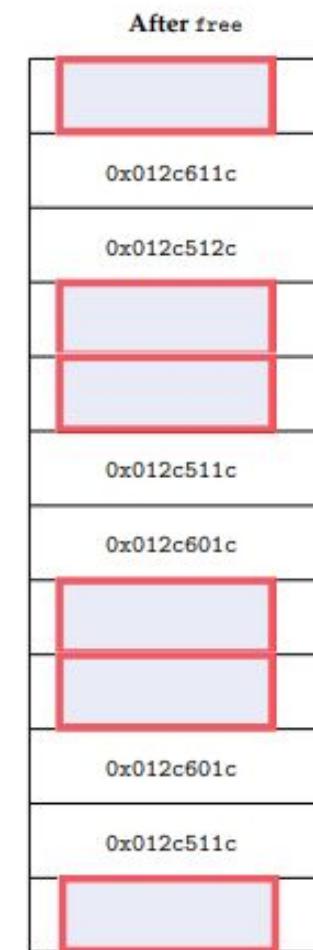
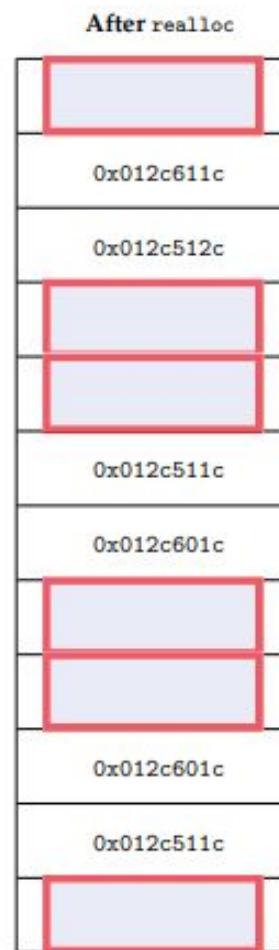
Is allocated
Previous is allocated
Size = 00100000 = 0x20 = 32

0x12
00010010

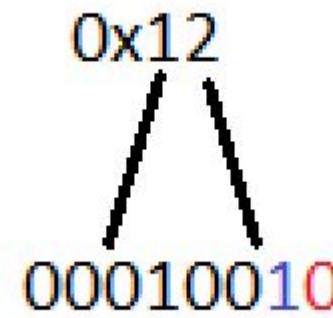
Is not allocated
Previous is allocated
Size = 00010000 = 0x10 = 16

realloc(0x12c000, 8).

| Address | Original value |
|-----------|----------------|
| 0x12c028 | 0x00000012 |
| 0x12c024 | 0x012c611c |
| 0x12c020 | 0x012c512c |
| 0x12c01c | 0x00000012 |
| 0x12c018 | 0x00000023 |
| 0x12c014 | 0x012c511c |
| 0x12c010 | 0x012c601c |
| 0x12c00c | 0x00000000 |
| 0x12c008 | 0x00000000 |
| 0x12c004 | 0x012c601c |
| 0x12c000 | 0x012c511c |
| 0x12bfffc | 0x00000023 |



Is allocated
Previous is allocated
Size = 00100000 = 0x20 = 32



Is not allocated
Previous is allocated
Size = 00010000 = 0x10 = 16

new size is 16, (8 + header/footer)
it will be allocated (via realloc())
The previous was allocated before, still will be

Size = 00010000, so: 00010011 = 0x13

realloc(0x12c000, 8).

free(0x12c000).

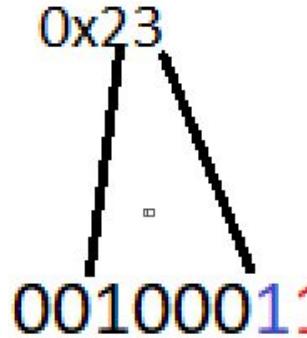
| Address | Original value | After realloc |
|-----------|----------------|---------------|
| 0x12c028 | 0x00000012 | |
| 0x12c024 | 0x012c611c | 0x012c611c |
| 0x12c020 | 0x012c512c | 0x012c512c |
| 0x12c01c | 0x00000012 | |
| 0x12c018 | 0x00000023 | |
| 0x12c014 | 0x012c511c | 0x012c511c |
| 0x12c010 | 0x012c601c | 0x012c601c |
| 0x12c00c | 0x00000000 | |
| 0x12c008 | 0x00000000 | 0x13 |
| 0x12c004 | 0x012c601c | 0x012c601c |
| 0x12c000 | 0x012c511c | 0x012c511c |
| 0x12bfffc | 0x00000023 | 0x13 |

Free

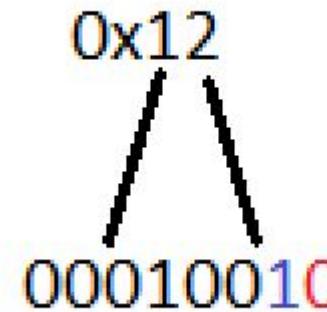
Now free

They must combine

After free



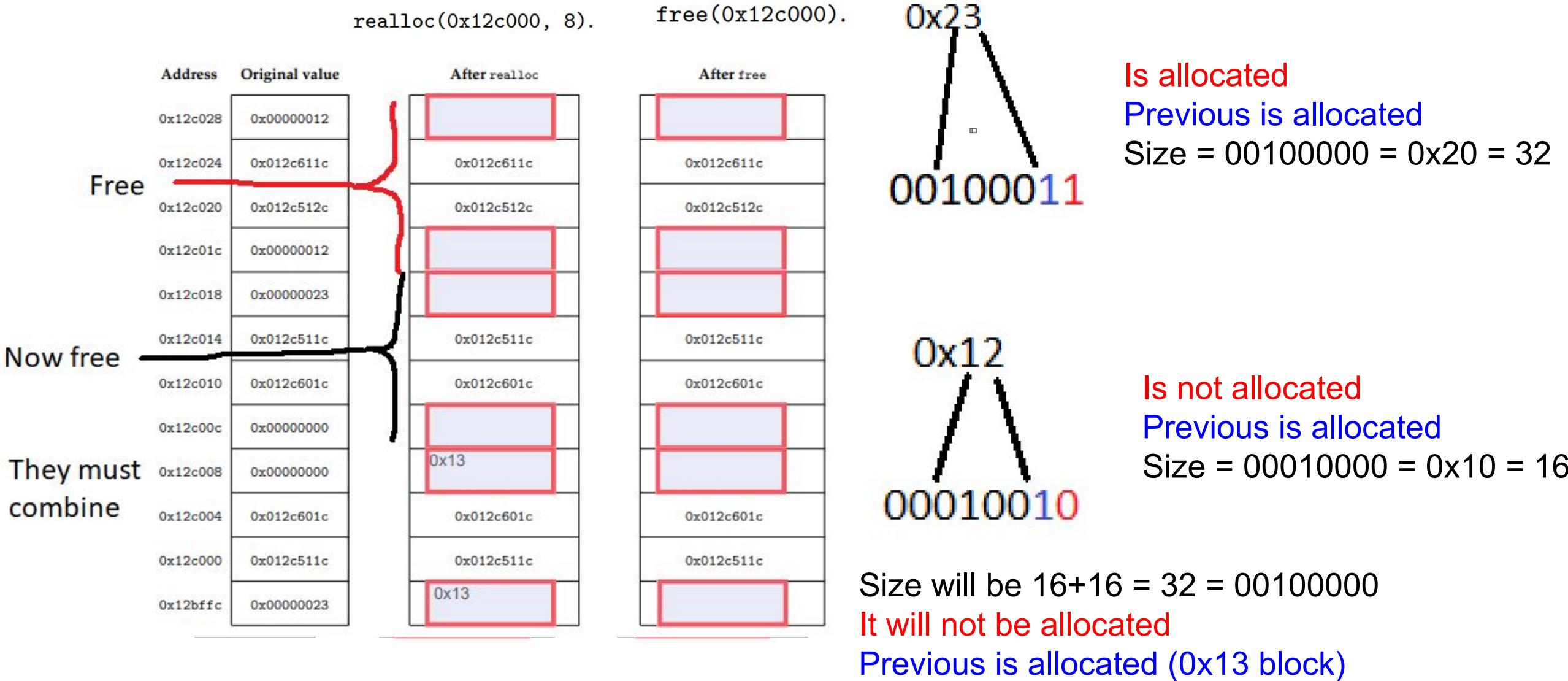
Is allocated
Previous is allocated
Size = 00100000 = 0x20 = 32



Is not allocated
Previous is allocated
Size = 00010000 = 0x10 = 16

new size is 16, (8 + header/footer)
it will be allocated (via realloc())
The previous was allocated before, still will be

Size = 00010000, so: 00010011 = 0x13



realloc(0x12c000, 8). free(0x12c000).

| Address | Original value | After realloc | After free |
|-----------|----------------|---------------|------------|
| 0x12c028 | 0x00000012 | 0x22 | |
| 0x12c024 | 0x012c611c | | 0x012c611c |
| 0x12c020 | 0x012c512c | | 0x012c512c |
| 0x12c01c | 0x00000012 | 0x12 | |
| 0x12c018 | 0x00000023 | 0x23 | |
| 0x12c014 | 0x012c511c | | 0x012c511c |
| 0x12c010 | 0x012c601c | | 0x012c601c |
| 0x12c00c | 0x00000000 | 0x22 | |
| 0x12c008 | 0x00000000 | 0x13 | |
| 0x12c004 | 0x012c601c | | 0x012c601c |
| 0x12c000 | 0x012c511c | | 0x012c511c |
| 0x12bfffc | 0x00000023 | 0x13 | |

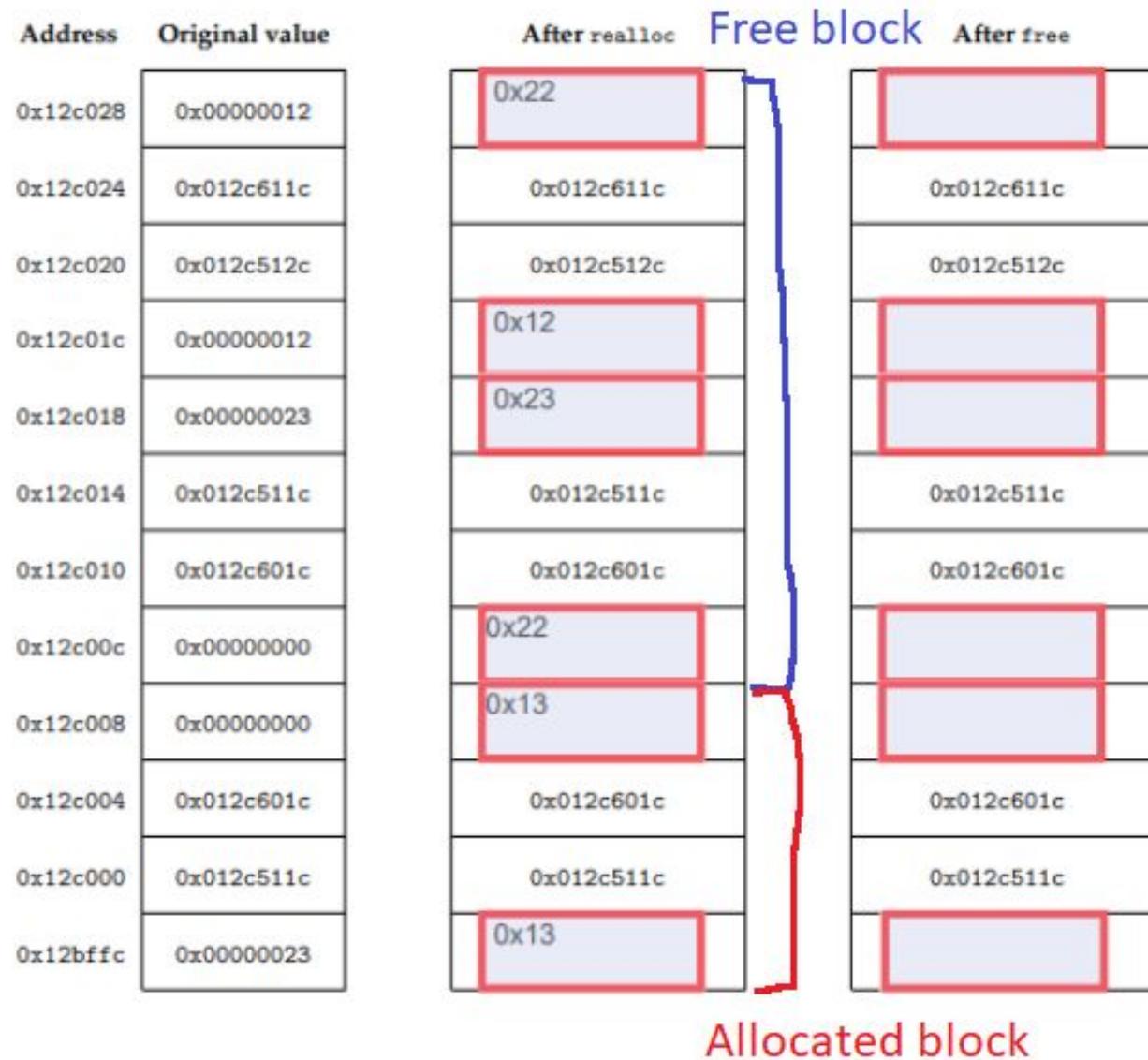
What about 0x12 and 0x23?

- They stay the same in c, we do not need to change them, so we do not. They are now essentially garbage values that are not allocated

Size will be $16+16 = 32 = 00100000$
It will not be allocated
 Previous is allocated (0x13 block)

so: $001000\textcolor{blue}{1}0 = 0x22$

realloc(0x12c000, 8). free(0x12c000).



Next: free(0x12c000)

realloc(0x12c000, 8). free(0x12c000).

| Address | Original value | After realloc |
|----------|----------------|---------------|
| 0x12c028 | 0x00000012 | 0x22 |
| 0x12c024 | 0x012c611c | 0x012c611c |
| 0x12c020 | 0x012c512c | 0x012c512c |
| 0x12c01c | 0x00000012 | 0x12 |
| 0x12c018 | 0x00000023 | 0x23 |
| 0x12c014 | 0x012c511c | 0x012c511c |
| 0x12c010 | 0x012c601c | 0x012c601c |
| 0x12c00c | 0x00000000 | 0x22 |
| 0x12c008 | 0x00000000 | 0x13 |
| 0x12c004 | 0x012c601c | 0x012c601c |
| 0x12c000 | 0x012c511c | 0x012c511c |
| 0x12bffc | 0x00000023 | 0x13 |

| Address | After free |
|----------|------------|
| 0x12c028 | |
| 0x12c024 | 0x012c611c |
| 0x12c020 | 0x012c512c |
| 0x12c01c | |
| 0x12c018 | |
| 0x12c014 | 0x012c511c |
| 0x12c010 | 0x012c601c |
| 0x12c00c | |
| 0x12c008 | |
| 0x12c004 | 0x012c601c |
| 0x12c000 | 0x012c511c |
| 0x12bffc | |

Next: free(0x12c000)

Free

Conclusion: Will be combined with instant coalescing

Will be free

realloc(0x12c000, 8).

free(0x12c000).

| Address | Original value | After realloc |
|-----------|----------------|---------------|
| 0x12c028 | 0x00000012 | 0x22 |
| 0x12c024 | 0x012c611c | 0x012c611c |
| 0x12c020 | 0x012c512c | 0x012c512c |
| 0x12c01c | 0x00000012 | 0x12 |
| 0x12c018 | 0x00000023 | 0x23 |
| 0x12c014 | 0x012c511c | 0x012c511c |
| 0x12c010 | 0x012c601c | 0x012c601c |
| 0x12c00c | 0x00000000 | 0x22 |
| 0x12c008 | 0x00000000 | 0x13 |
| 0x12c004 | 0x012c601c | 0x012c601c |
| 0x12c000 | 0x012c511c | 0x012c511c |
| 0x12bfffc | 0x00000023 | 0x13 |

| Address | After free |
|-----------|------------|
| 0x12c028 | |
| 0x12c024 | 0x012c611c |
| 0x12c020 | 0x012c512c |
| 0x12c01c | |
| 0x12c018 | |
| 0x12c014 | 0x012c511c |
| 0x12c010 | 0x012c601c |
| 0x12c00c | |
| 0x12c008 | |
| 0x12c004 | 0x012c601c |
| 0x12c000 | 0x012c511c |
| 0x12bfffc | |

size will be $32+16 = 00110000$ (48)

It will not be allocated

Previous is still allocated (off-screen)

32

so: 00110010 = 0x32

16

realloc(0x12c000, 8). free(0x12c000).

| Address | Original value | After realloc | After free |
|----------|----------------|---------------|------------|
| 0x12c028 | 0x00000012 | 0x22 | 0x32 |
| 0x12c024 | 0x012c611c | 0x012c611c | 0x012c611c |
| 0x12c020 | 0x012c512c | 0x012c512c | 0x012c512c |
| 0x12c01c | 0x00000012 | 0x12 | 0x12 |
| 0x12c018 | 0x00000023 | 0x23 | 0x23 |
| 0x12c014 | 0x012c511c | 0x012c511c | 0x012c511c |
| 0x12c010 | 0x012c601c | 0x012c601c | 0x012c601c |
| 0x12c00c | 0x00000000 | 0x22 | 0x22 |
| 0x12c008 | 0x00000000 | 0x13 | 0x13 |
| 0x12c004 | 0x012c601c | 0x012c601c | 0x012c601c |
| 0x12c000 | 0x012c511c | 0x012c511c | 0x012c511c |
| 0x12bffc | 0x00000023 | 0x13 | 0x32 |

size will be $32+16 = 00110000$ (48)

It will not be allocated

Previous is still allocated
(under, off-screen)

so: $001100\textcolor{blue}{1}0 = 0x32$

- Previous headers and footers in the middle (arrows) stay unchanged, as they do not need to change

`realloc(0x12c000, 8). free(0x12c000).`

| Address | Original value | After realloc | After free |
|-----------|----------------|---------------|------------|
| 0x12c028 | 0x00000012 | 0x22 | 0x32 |
| 0x12c024 | 0x012c611c | 0x012c611c | 0x012c611c |
| 0x12c020 | 0x012c512c | 0x012c512c | 0x012c512c |
| 0x12c01c | 0x00000012 | 0x12 | 0x12 |
| 0x12c018 | 0x00000023 | 0x23 | 0x23 |
| 0x12c014 | 0x012c511c | 0x012c511c | 0x012c511c |
| 0x12c010 | 0x012c601c | 0x012c601c | 0x012c601c |
| 0x12c00c | 0x00000000 | 0x22 | 0x22 |
| 0x12c008 | 0x00000000 | 0x13 | 0x13 |
| 0x12c004 | 0x012c601c | 0x012c601c | 0x012c601c |
| 0x12c000 | 0x012c511c | 0x012c511c | 0x012c511c |
| 0x12bfffc | 0x00000023 | 0x13 | 0x32 |

| Address | Original value | After realloc | After free |
|-----------|----------------|---------------|------------|
| 0x12c028 | 0x00000012 | 0x22 | 0x32 |
| 0x12c024 | 0x012c611c | 0x012c611c | 0x012c611c |
| 0x12c020 | 0x012c512c | 0x012c512c | 0x012c512c |
| 0x12c01c | 0x00000012 | 0x12 | 0x12 |
| 0x12c018 | 0x00000023 | 0x23 | 0x23 |
| 0x12c014 | 0x012c511c | 0x012c511c | 0x012c511c |
| 0x12c010 | 0x012c601c | 0x012c601c | 0x012c601c |
| 0x12c00c | 0x00000000 | 0x22 | 0x22 |
| 0x12c008 | 0x00000000 | 0x13 | 0x13 |
| 0x12c004 | 0x012c601c | 0x012c601c | 0x012c601c |
| 0x12c000 | 0x012c511c | 0x012c511c | 0x012c511c |
| 0x12bfffc | 0x00000023 | 0x13 | 0x32 |

Also

Also answer the following: Would it be possible for a free block to start at address 0x12c02C?

| Address | Original value | After realloc | After free |
|----------|----------------|---------------|------------|
| 0x12c028 | 0x00000012 | 0x22 | 0x32 |
| 0x12c024 | 0x012c611c | 0x012c611c | 0x012c611c |
| 0x12c020 | 0x012c512c | 0x012c512c | 0x012c512c |
| 0x12c01c | 0x00000012 | 0x12 | 0x12 |
| 0x12c018 | 0x00000023 | 0x23 | 0x23 |
| 0x12c014 | 0x012c511c | 0x012c511c | 0x012c511c |
| 0x12c010 | 0x012c601c | 0x012c601c | 0x012c601c |
| 0x12c00c | 0x00000000 | 0x22 | 0x22 |
| 0x12c008 | 0x00000000 | 0x13 | 0x13 |
| 0x12c004 | 0x012c601c | 0x012c601c | 0x012c601c |
| 0x12c000 | 0x012c511c | 0x012c511c | 0x012c511c |
| 0x12bffc | 0x00000023 | 0x13 | 0x32 |

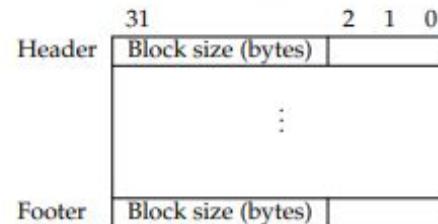
Answer:

- No, since then the uppermost block (0x12) was free. And if this was the case the block would initially have been combined with it due to instant coalescing

Exam 20/21

2.3 Long Questions (about 15 %)

Question 2.3.1: Consider an allocator that uses an implicit free list. The layout of each allocated and free memory block is as follows, with one 32-bit word per row:



Each memory block, either allocated or free, has a size that is a multiple of eight bytes, rounding up allocations if necessary. Thus, only the 29 higher order bits in the header and footer are needed to record block size, which includes the header and footer. The usage of the remaining 3 lower order bits is as follows:

- bit 0 indicates the use of the current block: 1 for allocated, 0 for free.
- bit 1 indicates the use of the previous adjacent block: 1 for allocated, 0 for free.
- bit 2 is unused and is always set to be 0.

Important: We must *never* create blocks with zero payload (i.e. we must *never* create blocks with size 8).

Given the heap shown on the left, show the new heap contents after consecutive calls to

- free(0x500c010).
- realloc(0x500c000, 12). Assume the return value is 0x500c000, meaning that the existing allocation is resized.

Your answers should be given as hex values. Note that the address grows from bottom up. Assume that the allocator uses immediate coalescing, that is, adjacent free blocks are merged immediately each time a block is freed.

| Address | Original value | After free | After realloc |
|-----------|----------------|------------|---------------|
| 0x500c028 | 0x00000013 | | |
| 0x500c024 | 0x500c611c | 0x500c611c | 0x500c611c |
| 0x500c020 | 0x500c512c | 0x500c512c | 0x500c512c |
| 0x500c01c | 0x00000013 | | |
| 0x500c018 | 0x00000013 | | |
| 0x500c014 | 0x500c511c | 0x500c511c | 0x500c511c |
| 0x500c010 | 0x500c601c | 0x500c601c | 0x500c601c |
| 0x500c00c | 0x00000013 | | |
| 0x500c008 | 0x00000013 | | |
| 0x500c004 | 0x500c601c | 0x500c601c | 0x500c601c |
| 0x500c000 | 0x500c511c | 0x500c511c | 0x500c511c |
| 0x500bfcc | 0x00000013 | | |

free(0x500c010). realloc(0x500c000, 12).

| Address | Original value | After free | After realloc |
|-----------------|----------------|------------|---------------|
| 0x500c028 | 0x00000013 | | |
| 0x500c024 | 0x500c611c | 0x500c611c | 0x500c611c |
| 0x500c020 | 0x500c512c | 0x500c512c | 0x500c512c |
| 0x500c01c | 0x00000013 | | |
| 0x500c018 | 0x00000013 | | |
| 0x500c014 | 0x500c511c | 0x500c511c | 0x500c511c |
| First element | 0x500c010 | 0x500c601c | 0x500c601c |
| Header of block | 0x500c00c | 0x00000013 | 0x500c601c |
| Footer of prev | 0x500c008 | 0x00000013 | 0x500c511c |
| | 0x500c004 | 0x500c601c | |
| | 0x500c000 | 0x500c511c | |
| | 0x500bff8 | 0x00000013 | |

free(0x500c010). realloc(0x500c000, 12).

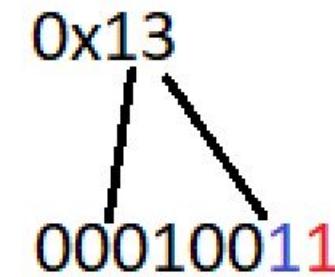
| Address | Original value | After free | After realloc |
|-----------|----------------|------------|---------------|
| 0x500c028 | 0x00000013 | | |
| 0x500c024 | 0x500c611c | 0x500c611c | 0x500c611c |
| 0x500c020 | 0x500c512c | 0x500c512c | 0x500c512c |
| 0x500c01c | 0x00000013 | | |
| 0x500c018 | 0x00000013 | | |
| 0x500c014 | 0x500c511c | 0x500c511c | 0x500c511c |
| 0x500c010 | 0x500c601c | 0x500c601c | 0x500c601c |
| 0x500c00c | 0x00000013 | | |
| 0x500c008 | 0x00000013 | | |
| 0x500c004 | 0x500c601c | 0x500c601c | 0x500c601c |
| 0x500c000 | 0x500c511c | 0x500c511c | 0x500c511c |
| 0x500bffc | 0x00000013 | | |

0x13
00010011

Is allocated
Previous is allocated
size is:
00010011 -> 00010000
= 0x10 or 16

free(0x500c010).realloc(0x500c000, 12)

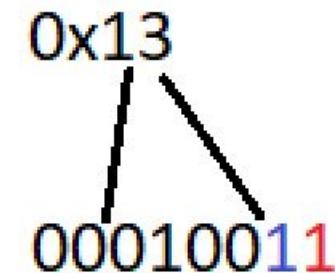
| Address | Original value | After free | After realloc |
|------------|----------------|------------|---------------|
| 0x500c028 | 0x00000013 | | |
| 0x500c024 | 0x500c611c | 0x500c611c | 0x500c611c |
| 0x500c020 | 0x500c512c | 0x500c512c | 0x500c512c |
| 0x500c01c | 0x00000013 | | |
| 0x500c018 | 0x00000013 | | |
| 0x500c014 | 0x500c511c | 0x500c511c | 0x500c511c |
| 0x500c010 | 0x500c601c | 0x500c601c | 0x500c601c |
| 0x500c00c | 0x00000013 | | |
| 0x500c008 | 0x00000013 | | |
| 0x500c004 | 0x500c601c | 0x500c601c | 0x500c601c |
| 0x500c000 | 0x500c511c | 0x500c511c | 0x500c511c |
| 0x500bfffc | 0x00000013 | | |



Is allocated
Previous is allocated
size is:
00010011 -> 00010000
= 0x10 or 16

```
free(0x500c010).realloc(0x500c000, 12) 2)
```

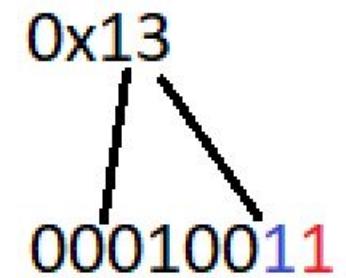
| Address | Original value | After free | After realloc |
|------------|----------------|------------|---------------|
| 0x500c028 | 0x00000013 | | |
| 0x500c024 | 0x500c611c | 0x500c611c | 0x500c611c |
| 0x500c020 | 0x500c512c | 0x500c512c | 0x500c512c |
| 0x500c01c | 0x00000013 | | |
| 0x500c018 | 0x00000013 | | |
| 0x500c014 | 0x500c511c | 0x500c511c | 0x500c511c |
| 0x500c010 | 0x500c601c | 0x500c601c | 0x500c601c |
| 0x500c00c | 0x00000013 | | |
| 0x500c008 | 0x00000013 | | |
| 0x500c004 | 0x500c601c | 0x500c601c | 0x500c601c |
| 0x500c000 | 0x500c511c | 0x500c511c | 0x500c511c |
| 0x500bfffc | 0x00000013 | | |



Is allocated
Previous is allocated
size is:
00010011 -> 00010000
= 0x10 or 16

`free(0x500c010).realloc(0x500c000, 12)`

| Address | Original value | After free | After realloc |
|-----------|----------------|------------|---------------|
| 0x500c028 | 0x00000013 | | |
| 0x500c024 | 0x500c611c | 0x500c611c | 0x500c611c |
| 0x500c020 | 0x500c512c | 0x500c512c | 0x500c512c |
| 0x500c01c | 0x00000013 | | |
| 0x500c018 | 0x00000013 | | |
| 0x500c014 | 0x500c511c | 0x500c511c | 0x500c511c |
| 0x500c010 | 0x500c601c | 0x500c601c | 0x500c601c |
| 0x500c00c | 0x00000013 | | |
| 0x500c008 | 0x00000013 | | |
| 0x500c004 | 0x500c601c | 0x500c601c | 0x500c601c |
| 0x500c000 | 0x500c511c | 0x500c511c | 0x500c511c |
| 0x500bffc | 0x00000013 | | |



Is allocated
Previous is allocated
size is:
 $00010011 \rightarrow 00010000$
 $= 0x10$ or 16

- Here, as no coalescing will occur this is simply changing **bit 0** which says that it is allocated
- Remember to also change **bit 1** of the next block

`free(0x500c010). realloc(0x500c000, 12).`

| Address | Original value | After free | After realloc |
|-----------|----------------|------------|---------------|
| 0x500c028 | 0x00000013 | 0x11 | |
| 0x500c024 | 0x500c611c | 0x500c611c | 0x500c611c |
| 0x500c020 | 0x500c512c | 0x500c512c | 0x500c512c |
| 0x500c01c | 0x00000013 | 0x11 | |
| 0x500c018 | 0x00000013 | 0x12 | |
| 0x500c014 | 0x500c511c | 0x500c511c | 0x500c511c |
| 0x500c010 | 0x500c601c | 0x500c601c | 0x500c601c |
| 0x500c00c | 0x00000013 | 0x12 | |
| 0x500c008 | 0x00000013 | 0x13 | |
| 0x500c004 | 0x500c601c | 0x500c601c | 0x500c601c |
| 0x500c000 | 0x500c511c | 0x500c511c | 0x500c511c |
| 0x500bffc | 0x00000013 | 0x13 | |

0x13
00010011

Is allocated
Previous is allocated
size is:
 $00010011 \rightarrow 00010000$
 $= 0x10$ or 16

- Here, as no coalescing will occur this is simply changing **bit 0** which says that it is allocated
- Remember to also change **bit 1** of the next block
- $000010010 = 0x12$
- $000010001 = 0x11$

free(0x500c010). realloc(0x500c000, 12).

| Address | Original value | After free | After realloc |
|-----------|----------------|------------|---------------|
| 0x500c028 | 0x00000013 | 0x11 | |
| 0x500c024 | 0x500c611c | 0x500c611c | 0x500c611c |
| 0x500c020 | 0x500c512c | 0x500c512c | 0x500c512c |
| 0x500c01c | 0x00000013 | 0x11 | |
| 0x500c018 | 0x00000013 | 0x12 | |
| 0x500c014 | 0x500c511c | 0x500c511c | 0x500c511c |
| 0x500c010 | 0x500c601c | 0x500c601c | 0x500c601c |
| 0x500c00c | 0x00000013 | 0x12 | |
| 0x500c008 | 0x00000013 | 0x13 | |
| 0x500c004 | 0x500c601c | 0x500c601c | 0x500c601c |
| 0x500c000 | 0x500c511c | 0x500c511c | 0x500c511c |
| 0x500bff8 | 0x00000013 | 0x13 | |

- Now must realloc(12)

Now
Free

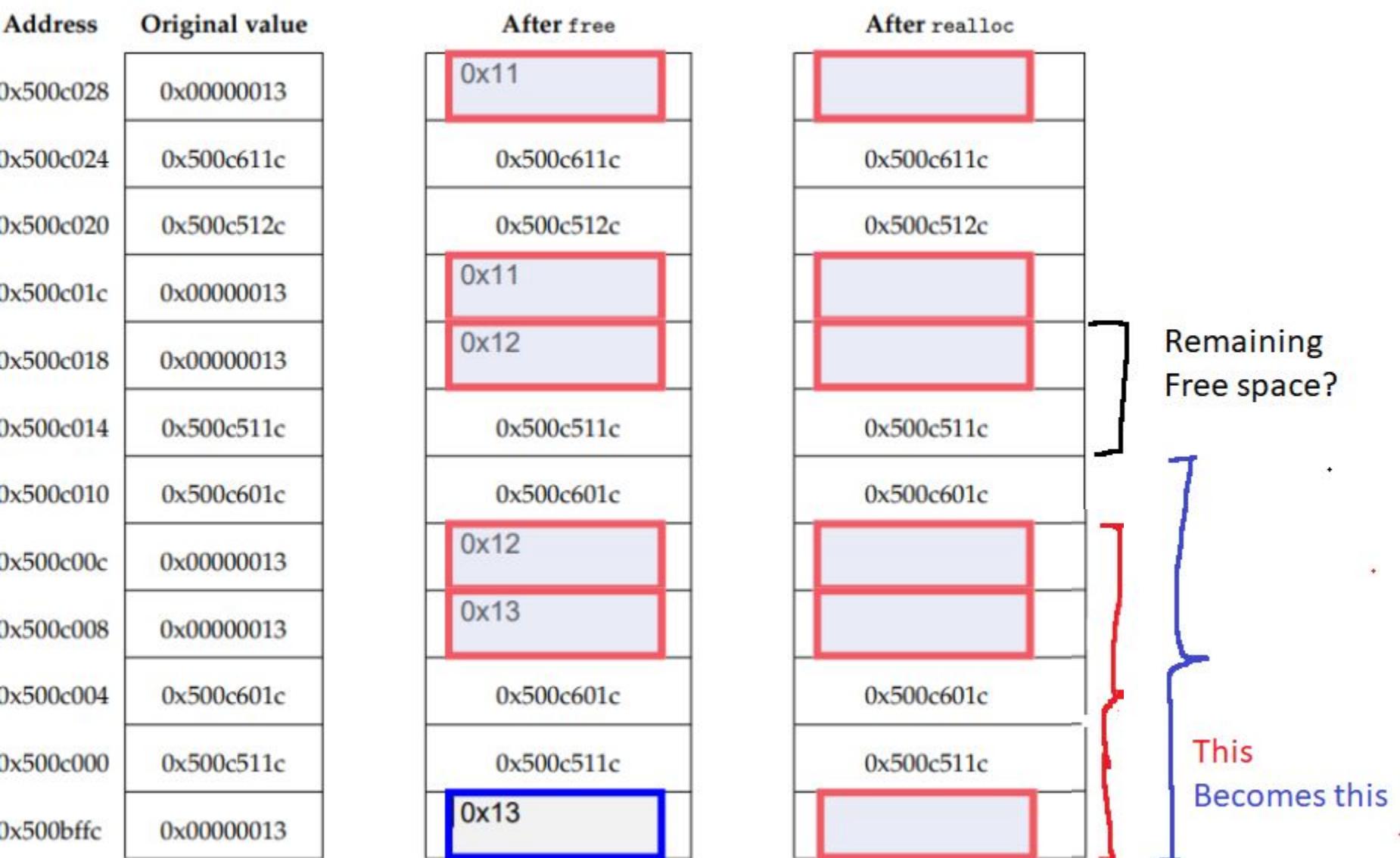
Realloc (on this block)

free(0x500c010). realloc(0x500c000, 12).

| Address | Original value | After free | After realloc |
|------------|----------------|------------|---------------|
| 0x500c028 | 0x00000013 | 0x11 | |
| 0x500c024 | 0x500c611c | 0x500c611c | 0x500c611c |
| 0x500c020 | 0x500c512c | 0x500c512c | 0x500c512c |
| 0x500c01c | 0x00000013 | 0x11 | |
| 0x500c018 | 0x00000013 | 0x12 | |
| 0x500c014 | 0x500c511c | 0x500c511c | 0x500c511c |
| 0x500c010 | 0x500c601c | 0x500c601c | 0x500c601c |
| 0x500c00c | 0x00000013 | 0x12 | |
| 0x500c008 | 0x00000013 | 0x13 | |
| 0x500c004 | 0x500c601c | 0x500c601c | 0x500c601c |
| 0x500c000 | 0x500c511c | 0x500c511c | 0x500c511c |
| 0x500bfffc | 0x00000013 | 0x13 | |

- Now must realloc(12)
- Problem is that 12 is not a multiple of 8, so we have to round up resulting realloc(16)
- Additionally, remember that 16 is the amount of space we want, excluding the header and footer

free(0x500c010). realloc(0x500c000, 12).



- New problem:
The alloc will leave the free
space as a block
of size 8.
- Remember the
rule?

Each memory block, either allocated or free, has a size that is a multiple of eight bytes, rounding up allocations if necessary. Thus, only the 29 higher order bits in the header and footer are needed to record block size, which includes the header and footer. The usage of the remaining 3 lower order bits is as follows:

- bit 0 indicates the use of the current block: 1 for allocated, 0 for free.
- bit 1 indicates the use of the previous adjacent block: 1 for allocated, 0 for free.
- bit 2 is unused and is always set to be 0.

Important: We must *never* create blocks with zero payload (i.e. we must *never* create blocks with size 8).

Given the heap shown on the left, show the new heap contents after *consecutive* calls to

1. `free(0x500c010)`.
2. `realloc(0x500c000, 12)`. Assume the return value is `0x500c000`, meaning that the existing allocation is resized.

Your answers should be given as hex values. Note that the address grows from bottom up. Assume that the allocator uses immediate coalescing, that is, adjacent free blocks are merged immediately each time a block is freed.

free(0x500c010). realloc(0x500c000, 12).

| Address | Original value | After free | After realloc |
|-----------|----------------|------------|---------------|
| 0x500c028 | 0x00000013 | 0x11 | |
| 0x500c024 | 0x500c611c | 0x500c611c | 0x500c611c |
| 0x500c020 | 0x500c512c | 0x500c512c | 0x500c512c |
| 0x500c01c | 0x00000013 | 0x11 | |
| 0x500c018 | 0x00000013 | 0x12 | |
| 0x500c014 | 0x500c511c | 0x500c511c | 0x500c511c |
| 0x500c010 | 0x500c601c | 0x500c601c | 0x500c601c |
| 0x500c00c | 0x00000013 | 0x12 | |
| 0x500c008 | 0x00000013 | 0x13 | |
| 0x500c004 | 0x500c601c | 0x500c601c | 0x500c601c |
| 0x500c000 | 0x500c511c | 0x500c511c | 0x500c511c |
| 0x500bffc | 0x00000013 | 0x13 | |

- Solution?
- Even more internal fragmentation
- Essentially, since the remaining free space is too small, the only choice is for this to be taken by the realloc aswell... so

Will-be block
with 0 payload

This
Becomes this

free(0x500c010). realloc(0x500c000, 12).

| Address | Original value | After free | After realloc |
|-----------|----------------|------------|---------------|
| 0x500c028 | 0x00000013 | 0x11 | |
| 0x500c024 | 0x500c611c | 0x500c611c | 0x500c611c |
| 0x500c020 | 0x500c512c | 0x500c512c | 0x500c512c |
| 0x500c01c | 0x00000013 | 0x11 | |
| 0x500c018 | 0x00000013 | 0x12 | |
| 0x500c014 | 0x500c511c | 0x500c511c | 0x500c511c |
| 0x500c010 | 0x500c601c | 0x500c601c | 0x500c601c |
| 0x500c00c | 0x00000013 | 0x12 | |
| 0x500c008 | 0x00000013 | 0x13 | |
| 0x500c004 | 0x500c601c | 0x500c601c | 0x500c601c |
| 0x500c000 | 0x500c511c | 0x500c511c | 0x500c511c |
| 0x500bffc | 0x00000013 | 0x13 | |

We must have this:

Now, after solving
the puzzle, we fill in
the fields

This
Becomes this

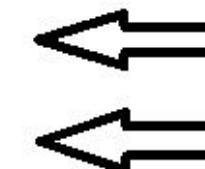
`free(0x500c010). realloc(0x500c000, 12)`

| Address | Original value | After free | After realloc |
|------------|----------------|------------|---------------|
| 0x500c028 | 0x00000013 | 0x11 | |
| 0x500c024 | 0x500c611c | 0x500c611c | 0x500c611c |
| 0x500c020 | 0x500c512c | 0x500c512c | 0x500c512c |
| 0x500c01c | 0x00000013 | 0x11 | |
| 0x500c018 | 0x00000013 | 0x12 | |
| 0x500c014 | 0x500c511c | 0x500c511c | 0x500c511c |
| 0x500c010 | 0x500c601c | 0x500c601c | 0x500c601c |
| 0x500c00c | 0x00000013 | 0x12 | |
| 0x500c008 | 0x00000013 | 0x13 | |
| 0x500c004 | 0x500c601c | 0x500c601c | 0x500c601c |
| 0x500c000 | 0x500c511c | 0x500c511c | 0x500c511c |
| 0x500bfffc | 0x00000013 | 0x13 | |

- What will the header of the block be?
- Will be allocated
- Previous is still allocated
- Size is $16+16 = 32 = 0010\ 0000$
- So we get:
 $0010\ 0011 = 0x23$

free(0x500c010). realloc(0x500c000, 12)

| Address | Original value | After free | After realloc |
|-----------|----------------|------------|---------------|
| 0x500c028 | 0x00000013 | 0x11 | |
| 0x500c024 | 0x500c611c | 0x500c611c | 0x500c611c |
| 0x500c020 | 0x500c512c | 0x500c512c | 0x500c512c |
| 0x500c01c | 0x00000013 | 0x11 | |
| 0x500c018 | 0x00000013 | 0x12 | 0x23 |
| 0x500c014 | 0x500c511c | 0x500c511c | 0x500c511c |
| 0x500c010 | 0x500c601c | 0x500c601c | 0x500c601c |
| 0x500c00c | 0x00000013 | 0x12 | 0x12 |
| 0x500c008 | 0x00000013 | 0x13 | 0x13 |
| 0x500c004 | 0x500c601c | 0x500c601c | 0x500c601c |
| 0x500c000 | 0x500c511c | 0x500c511c | 0x500c511c |
| 0x500bffc | 0x00000013 | 0x13 | 0x23 |



- Remember, these don't change due to instant coalescing

free(0x500c010). realloc(0x500c000, 12)

| Address | Original value | After free | After realloc |
|-----------|----------------|------------|---------------|
| 0x500c028 | 0x00000013 | 0x11 | |
| 0x500c024 | 0x500c611c | 0x500c611c | 0x500c611c |
| 0x500c020 | 0x500c512c | 0x500c512c | 0x500c512c |
| 0x500c01c | 0x00000013 | 0x11 | |
| 0x500c018 | 0x00000013 | 0x12 | 0x23 |
| 0x500c014 | 0x500c511c | 0x500c511c | 0x500c511c |
| 0x500c010 | 0x500c601c | 0x500c601c | 0x500c601c |
| 0x500c00c | 0x00000013 | 0x12 | 0x12 |
| 0x500c008 | 0x00000013 | 0x13 | 0x13 |
| 0x500c004 | 0x500c601c | 0x500c601c | 0x500c601c |
| 0x500c000 | 0x500c511c | 0x500c511c | 0x500c511c |
| 0x500bffc | 0x00000013 | 0x13 | 0x23 |

- And of the last block?
- Not much changes.
Except now the previous is allocated again, so **bit 1** changes back to 1.
- $0x11 = 0001\ 00\textcolor{blue}{0}1$
->
 $0001\ 00\textcolor{red}{1}1 = 0x13$
- We also see this is exactly the same header as the original state

`free(0x500c010). realloc(0x500c000, 12)`

| Address | Original value | After free | After realloc |
|------------|----------------|------------|---------------|
| 0x500c028 | 0x00000013 | 0x11 | 0x13 |
| 0x500c024 | 0x500c611c | 0x500c611c | 0x500c611c |
| 0x500c020 | 0x500c512c | 0x500c512c | 0x500c512c |
| 0x500c01c | 0x00000013 | 0x11 | 0x13 |
| 0x500c018 | 0x00000013 | 0x12 | 0x23 |
| 0x500c014 | 0x500c511c | 0x500c511c | 0x500c511c |
| 0x500c010 | 0x500c601c | 0x500c601c | 0x500c601c |
| 0x500c00c | 0x00000013 | 0x12 | 0x12 |
| 0x500c008 | 0x00000013 | 0x13 | 0x13 |
| 0x500c004 | 0x500c601c | 0x500c601c | 0x500c601c |
| 0x500c000 | 0x500c511c | 0x500c511c | 0x500c511c |
| 0x500bfffc | 0x00000013 | 0x13 | 0x23 |

- And of the last block?
- Not much changes. Except now the previous is allocated again, so **bit 1** changes back to 1.
- $0x11 = 0001\ 00\textcolor{blue}{0}1$
->
 $0001\ 00\textcolor{red}{1}1 = 0x13$
- We also see this is exactly the same header as the original state

Correct?

| Address | Original value | After free | After realloc | Address | Original value | After free | After realloc |
|-----------|----------------|------------|---------------|-----------|----------------|------------|---------------|
| 0x500c028 | 0x00000013 | 0x11 | 0x13 | 0x500c028 | 0x00000013 | 0x11 | 0x13 |
| 0x500c024 | 0x500c611c | 0x500c611c | 0x500c611c | 0x500c024 | 0x500c611c | 0x500c611c | 0x500c611c |
| 0x500c020 | 0x500c512c | 0x500c512c | 0x500c512c | 0x500c020 | 0x500c512c | 0x500c512c | 0x500c512c |
| 0x500c01c | 0x00000013 | 0x11 | 0x13 | 0x500c01c | 0x00000013 | 0x11 | 0x13 |
| 0x500c018 | 0x00000013 | 0x12 | 0x23 | 0x500c018 | 0x00000013 | 0x12 | 0x23 |
| 0x500c014 | 0x500c511c | 0x500c511c | 0x500c511c | 0x500c014 | 0x500c511c | 0x500c511c | 0x500c511c |
| 0x500c010 | 0x500c601c | 0x500c601c | 0x500c601c | 0x500c010 | 0x500c601c | 0x500c601c | 0x500c601c |
| 0x500c00c | 0x00000013 | 0x12 | 0x12 | 0x500c00c | 0x00000013 | 0x12 | 0x12 |
| 0x500c008 | 0x00000013 | 0x13 | 0x13 | 0x500c008 | 0x00000013 | 0x13 | 0x13 |
| 0x500c004 | 0x500c601c | 0x500c601c | 0x500c601c | 0x500c004 | 0x500c601c | 0x500c601c | 0x500c601c |
| 0x500c000 | 0x500c511c | 0x500c511c | 0x500c511c | 0x500c000 | 0x500c511c | 0x500c511c | 0x500c511c |
| 0x500bffc | 0x00000013 | 0x13 | 0x23 | 0x500bffc | 0x00000013 | 0x13 | 0x23 |

Extra question (if time allows)

After realloc

| |
|------------|
| 0x13 |
| 0x500c611c |
| 0x500c512c |
| 0x13 |
| 0x23 |
| 0x500c511c |
| 0x500c601c |
| 0x12 |
| 0x13 |
| 0x500c601c |
| 0x500c511c |
| 0x23 |

Also answer the following: Does the resulting heap exhibit internal or external fragmentation? Explain your answer.

Yes, the block whose payload starts at 0x500c000 was asked to contain 12 bytes of payload, but ended up being a 32-byte block due to the 8 bytes of headers/footers (necessary), but also having to round up the allocation to be a multiple of 8 bytes, and then further expanding to avoid a following zero-payload block.

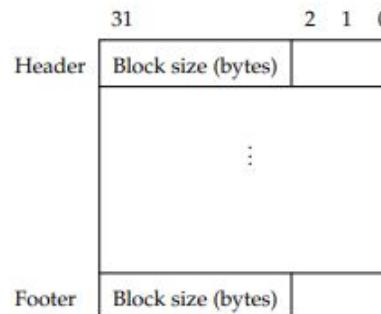
Essentially:

Wasted space
(internal fragmentation)

What we really wanted

Exam 22/23

Question 2.3.1: Consider an allocator that uses an implicit free list and immediate coalescing of neighbouring free blocks. The layout of each allocated and free memory block is as follows, with one 32-bit word per row:



Each memory block, either allocated or free, has a size that is a multiple of eight bytes, rounding up allocations if necessary. Thus, only the 29 higher order bits in the header and footer are needed to record block size, which includes the header and footer. The minimum block size is 8. The usage of the remaining 3 lower order bits is as follows:

- Bit 0 indicates the use of the current block: 1 for allocated, 0 for free.
- Bit 1 indicates the use of the previous adjacent block: 1 for allocated, 0 for free.
- Bit 2 is unused and is always set to be 0.

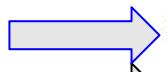
Given the partial contents of the heap shown on the left, show the new contents of the heap after a call to `malloc(8)` is executed that returns `0xd1c008` (in the middle column), followed by a call `free(0xd1c028)` (rightmost column).

- All numbers are hexadecimal, and so should your answers be.
- Note that the address grows from bottom up.
- Some parts of the heap may lie outside the area shown.
- Assume that the allocator uses immediate coalescing, that is, adjacent free blocks are merged immediately each time a block is freed.
- Perform the minimum number of memory changes required.

| Address | Original value | After malloc | After free |
|----------|----------------|--------------|------------|
| 00d1c04c | 00000100 | 00000100 | 00000100 |
| 00d1c048 | 00000020 | 00000020 | 00000020 |
| 00d1c044 | 00000042 | 00000042 | 00000042 |
| 00d1c040 | 00000021 | 00000021 | 00000021 |
| 00d1c03c | 0000001d | 0000001d | 0000001d |
| 00d1c038 | 00000018 | 00000018 | 00000018 |
| 00d1c034 | 00000000 | 00000000 | 00000000 |
| 00d1c030 | 00000000 | 00000000 | 00000000 |
| 00d1c02c | 00000000 | 00000000 | 00000000 |
| 00d1c028 | 0000001d | 0000001d | 0000001d |
| 00d1c024 | 00000021 | 00000021 | 00000021 |
| 00d1c020 | 00000022 | 00000022 | 00000022 |
| 00d1c01c | 00000000 | 00000000 | 00000000 |
| 00d1c018 | 00000000 | 00000000 | 00000000 |
| 00d1c014 | 00000000 | 00000000 | 00000000 |
| 00d1c010 | 00000000 | 00000000 | 00000000 |
| 00d1c00c | 00000000 | 00000000 | 00000000 |
| 00d1c008 | 00000000 | 00000000 | 00000000 |
| 00d1c004 | 00000022 | 00000022 | 00000022 |
| 00d1c000 | 0000000b | 0000000b | 0000000b |
| 00d1bfff | 0000000b | 0000000b | 0000000b |

Larger heap
= smaller text
(sorry)

| Address | Original value | 0xd1c008 After malloc | free(0xd1c028) After free |
|----------|----------------|--------------------------|------------------------------|
| 00d1c04c | 00000100 | 00000100 | 00000100 |
| 00d1c048 | 00000020 | 00000020 | 00000020 |
| 00d1c044 | 00000042 | | |
| 00d1c040 | 00000021 | | |
| 00d1c03c | 0000001d | | |
| 00d1c038 | 00000018 | 00000018 | 00000018 |
| 00d1c034 | 00000000 | 00000000 | 00000000 |
| 00d1c030 | 00000000 | 00000000 | 00000000 |
| 00d1c02c | 00000000 | 00000000 | 00000000 |
| 00d1c028 | 0000001d | 0000001d | 0000001d |
| 00d1c024 | 00000021 | | |
| 00d1c020 | 00000022 | | |
| 00d1c01c | 00000000 | 00000000 | 00000000 |
| 00d1c018 | 00000000 | 00000000 | 00000000 |
| 00d1c014 | 00000000 | | |
| 00d1c010 | 00000000 | | |
| 00d1c00c | 00000000 | 00000000 | 00000000 |
| 00d1c008 | 00000000 | 00000000 | 00000000 |
| 00d1c004 | 00000022 | | |
| 00d1c000 | 0000000b | | |
| 00d1bfec | 0000000b | | |

First element 

Header 

Footer 

Same as always - analyse the initial state of the heap

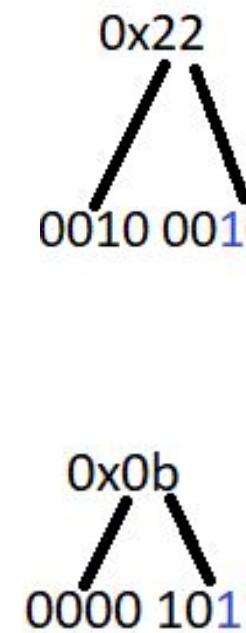
- The header is once again behind first element
- The footer of the previous block is behind that.
- We start analysing

| Address | Original value | 0xd1c008 After malloc | free(0xd1c028) After free |
|---------------|----------------|--------------------------|------------------------------|
| 00d1c04c | 00000100 | 00000100 | 00000100 |
| 00d1c048 | 00000020 | 00000020 | 00000020 |
| 00d1c044 | 00000042 | | |
| 00d1c040 | 00000021 | | |
| 00d1c03c | 0000001d | | |
| 00d1c038 | 00000018 | 00000018 | 00000018 |
| 00d1c034 | 00000000 | 00000000 | 00000000 |
| 00d1c030 | 00000000 | 00000000 | 00000000 |
| 00d1c02c | 00000000 | 00000000 | 00000000 |
| 00d1c028 | 0000001d | 0000001d | 0000001d |
| 00d1c024 | 00000021 | | |
| 00d1c020 | 00000022 | | |
| 00d1c01c | 00000000 | 00000000 | 00000000 |
| 00d1c018 | 00000000 | 00000000 | 00000000 |
| 00d1c014 | 00000000 | | |
| 00d1c010 | 00000000 | | |
| 00d1c00c | 00000000 | 00000000 | 00000000 |
| First element | 00d1c008 | 00000000 | 00000000 |
| Header | 00d1c004 | 00000022 | 00000000 |
| Footer | 00d1c000 | 0000000b | 0000000b |
| | 00d1bfcc | 0000000b | 0000000b |

0x22
0010 0010

- Not allocated (makes sense, since we will malloc it)
- Previous allocated
- size = 0010 0010 -> 0010 0000 = 0x20 = 32

| Address | Original value | 0xd1c008 After malloc | free(0xd1c028) After free |
|---------------|----------------|--------------------------|------------------------------|
| 00d1c04c | 00000100 | 00000100 | 00000100 |
| 00d1c048 | 00000020 | 00000020 | 00000020 |
| 00d1c044 | 00000042 | (red box) | (red box) |
| 00d1c040 | 00000021 | (red box) | (red box) |
| 00d1c03c | 0000001d | (red box) | (red box) |
| 00d1c038 | 00000018 | 00000018 | 00000018 |
| 00d1c034 | 00000000 | 00000000 | 00000000 |
| 00d1c030 | 00000000 | 00000000 | 00000000 |
| 00d1c02c | 00000000 | 00000000 | 00000000 |
| 00d1c028 | 0000001d | 0000001d | 0000001d |
| 00d1c024 | 00000021 | (red box) | (red box) |
| 00d1c020 | 00000022 | (red box) | (red box) |
| 00d1c01c | 00000000 | 00000000 | 00000000 |
| 00d1c018 | 00000000 | 00000000 | 00000000 |
| 00d1c014 | 00000000 | (red box) | (red box) |
| 00d1c010 | 00000000 | (red box) | (red box) |
| 00d1c00c | 00000000 | 00000000 | 00000000 |
| First element | 00d1c008 | 00000000 | 00000000 |
| Header | 00d1c004 | 00000022 | 00000000 |
| Footer | 00d1c000 | 0000000b | 0000000b |
| | 00d1bfcc | 0000000b | 0000000b |



- Not allocated (makes sense, since we will malloc it)
- Previous allocated
- size = 0010 0010 -> 0010 0000 = 0x20 = 32
- Allocated
- Previous is allocated
- size = 0000 1011 -> 0000 1000 = 0x08 = 8

But that's not allowed? Right?

Be careful! Rules change

Each memory block, either allocated or free, has a size that is a multiple of eight bytes, rounding up allocations if necessary. Thus, only the 29 higher order bits in the header and footer are needed to record block size, which includes the header and footer. The minimum block size is 8. The usage of the remaining 3 lower order bits is as follows:

- Bit 0 indicates the use of the current block: 1 for allocated, 0 for free.
- Bit 1 indicates the use of the previous adjacent block: 1 for allocated, 0 for free.
- Bit 2 is unused and is always set to be 0.

Given the partial contents of the heap shown on the left, show the new contents of the heap after a call to `malloc(8)` is executed that returns `0xd1c008` (in the middle column), followed by a call `free(0xd1c028)` (rightmost column).

- All numbers are hexadecimal, and so should your answers be.
- Note that the address grows from bottom up.
- Some parts of the heap may lie outside the area shown.
- Assume that the allocator uses immediate coalescing, that is, adjacent free blocks are merged immediately each time a block is freed.
- Perform the minimum number of memory changes required.

Rule stating:

“Important: We must never create blocks with zero payload (i.e. we must never create blocks with size 8).”

Is no longer present. It was changed.

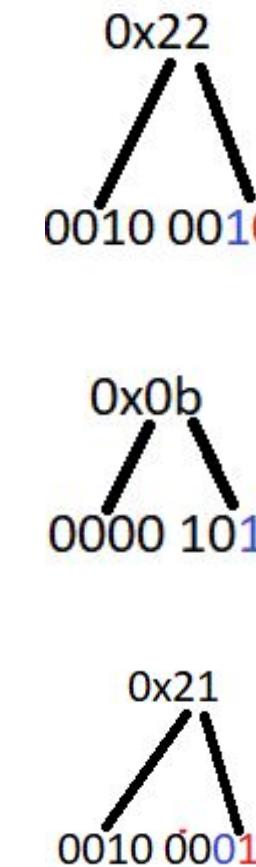
| Address | Original value | 0xd1c008 After malloc | free(0xd1c028) After free |
|---------------|----------------|--------------------------|------------------------------|
| 00d1c04c | 00000100 | 00000100 | 00000100 |
| 00d1c048 | 00000020 | 00000020 | 00000020 |
| 00d1c044 | 00000042 | (red box) | (red box) |
| 00d1c040 | 00000021 | (red box) | (red box) |
| 00d1c03c | 0000001d | (red box) | (red box) |
| 00d1c038 | 00000018 | 00000018 | 00000018 |
| 00d1c034 | 00000000 | 00000000 | 00000000 |
| 00d1c030 | 00000000 | 00000000 | 00000000 |
| 00d1c02c | 00000000 | 00000000 | 00000000 |
| 00d1c028 | 0000001d | 0000001d | 0000001d |
| 00d1c024 | 00000021 | (red box) | (red box) |
| 00d1c020 | 00000022 | (red box) | (red box) |
| 00d1c01c | 00000000 | 00000000 | 00000000 |
| 00d1c018 | 00000000 | 00000000 | 00000000 |
| 00d1c014 | 00000000 | (red box) | (red box) |
| 00d1c010 | 00000000 | (red box) | (red box) |
| 00d1c00c | 00000000 | 00000000 | 00000000 |
| First element | 00d1c008 | 00000000 | 00000000 |
| Header | 00d1c004 | 00000022 | 00000000 |
| Footer | 00d1c000 | 0000000b | 0000000b |
| | 00d1bfcc | 0000000b | 0000000b |

0x22
0010 0010

0x0b
0000 1011

- Not allocated (makes sense, since we will malloc it)
- Previous allocated
- size = 0010 0010 -> 0010 0000 = 0x20 = 32
- Allocated
- Previous is allocated
- size = 0000 1011 -> 0000 1000 = 0x08 = 8
- Also, last year it appears they tried to make just guessing the headers / footers a bit harder

| Address | Original value | 0xd1c008 After malloc | free(0xd1c028) After free |
|----------|----------------|--------------------------|------------------------------|
| 00d1c04c | 000000100 | 000000100 | 000000100 |
| 00d1c048 | 000000020 | 000000020 | 000000020 |
| 00d1c044 | 000000042 | | |
| 00d1c040 | 000000021 | | |
| 00d1c03c | 00000001d | | |
| 00d1c038 | 000000018 | 000000018 | 000000018 |
| 00d1c034 | 000000000 | 000000000 | 000000000 |
| 00d1c030 | 000000000 | 000000000 | 000000000 |
| 00d1c02c | 000000000 | 000000000 | 000000000 |
| 00d1c028 | 00000001d | 00000001d | 00000001d |
| 00d1c024 | 000000021 | 000000021 | 000000021 |
| 00d1c020 | 000000022 | 000000022 | 000000022 |
| 00d1c01c | 000000000 | 000000000 | 000000000 |
| 00d1c018 | 000000000 | 000000000 | 000000000 |
| 00d1c014 | 000000000 | 000000000 | 000000000 |
| 00d1c010 | 000000000 | 000000000 | 000000000 |
| 00d1c00c | 000000000 | 000000000 | 000000000 |
| 00d1c008 | 000000000 | 000000000 | 000000000 |
| 00d1c004 | 000000022 | | |
| 00d1c000 | 00000000b | | |
| 00d1bfff | 00000000b | | |

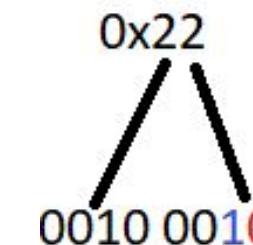


- Not allocated (makes sense, since we will malloc it)
- Previous allocated
- size = 0010 0010 -> 0010 0000 = 0x20 = 32

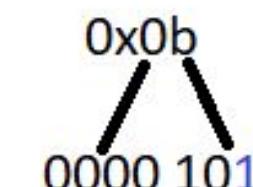
- Allocated
- Previous is allocated
- size = 0000 1011 -> 0000 1000 = 0x08 = 8

- Allocated
- Previous is not allocated
- size = 0010 0001 -> 0010 0000 = 0x20 = 32

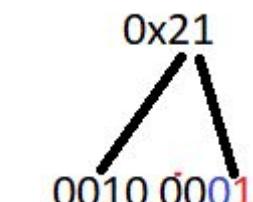
| Address | Original value | 0xd1c008 | free(0xd1c028) |
|----------|----------------|--------------------|----------------|
| | | After malloc | After free |
| 00d1c04c | 00000100 | 00000100 | 00000100 |
| 00d1c048 | 00000020 | 00000020 | 00000020 |
| 00d1c044 | 00000042 | Header 00000042 | 00000042 |
| 00d1c040 | 00000021 | 00000021 | 00000021 |
| 00d1c03c | 0000001d | 0000001d | 0000001d |
| 00d1c038 | 00000018 | 00000018 | 00000018 |
| 00d1c034 | 00000000 | 00000000 | 00000000 |
| 00d1c030 | 00000000 | 00000000 | 00000000 |
| 00d1c02c | 00000000 | 00000000 | 00000000 |
| 00d1c028 | 0000001d | 0000001d | 0000001d |
| 00d1c024 | 00000021 | 00000021 | 00000021 |
| 00d1c020 | 00000022 | 00000022 | 00000022 |
| 00d1c01c | 00000000 | 00000000 | 00000000 |
| 00d1c018 | 00000000 | 00000000 | 00000000 |
| 00d1c014 | 00000000 | 00000000 | 00000000 |
| 00d1c010 | 00000000 | 00000000 | 00000000 |
| 00d1c00c | 00000000 | 00000000 | 00000000 |
| 00d1c008 | 00000000 | 00000000 | 00000000 |
| 00d1c004 | 00000022 | 00000022 | 00000022 |
| 00d1c000 | 0000000b | 0000000b | 0000000b |
| 00d1bfcc | 0000000b | 0000000b | 0000000b |



32



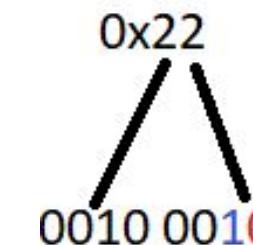
32



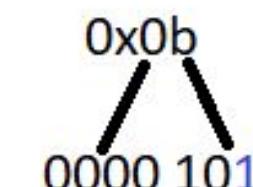
32

That's quite a large number, feels like it will go beyond the visible heap

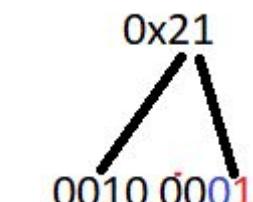
- Some parts of the heap may lie outside the area shown.



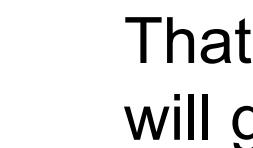
32



32



32



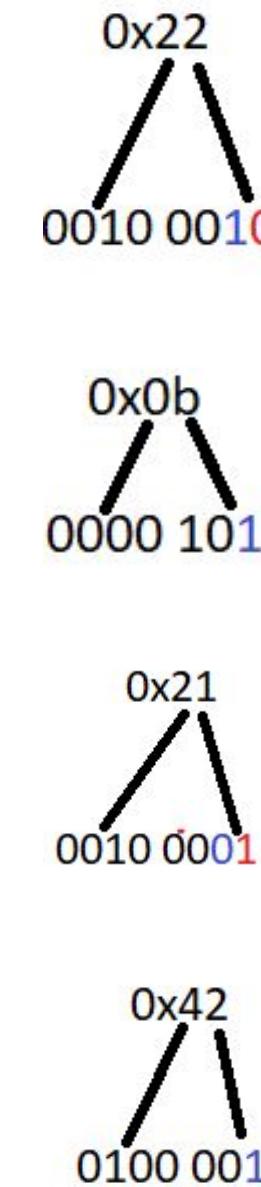
32

- Not allocated (makes sense, since we will malloc it)
- Previous allocated
- size = 0010 0010 -> 0010 0000 = 0x20 = 32

- Allocated
- Previous is allocated
- size = 0000 1011 -> 0000 1000 = 0x08 = 8

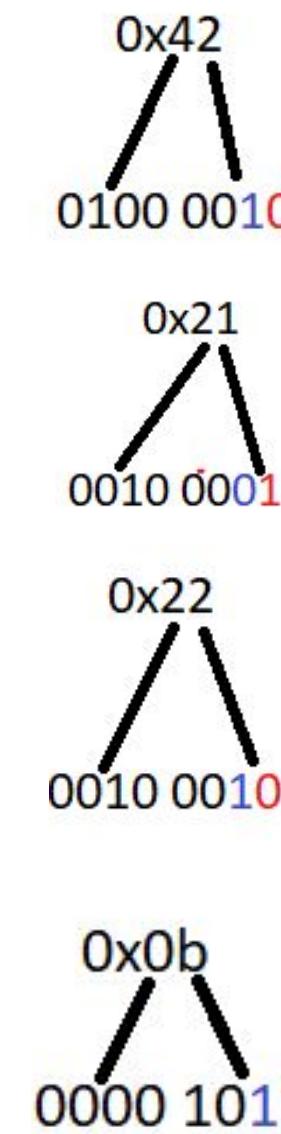
- Allocated
- Previous is not allocated
- size = 0010 0001 -> 0010 0000 = 0x20 = 32

| Address | Original value | 0xd1c008 After malloc | free(0xd1c028) After free |
|----------|----------------|--------------------------|------------------------------|
| 00d1c04c | 00000100 | 00000100 | 00000100 |
| 00d1c048 | 00000020 | 00000020 | 00000020 |
| 00d1c044 | 00000042 | Header 00000042 | 00000042 |
| 00d1c040 | 00000021 | | |
| 00d1c03c | 0000001d | | |
| 00d1c038 | 00000018 | 00000018 | 00000018 |
| 00d1c034 | 00000000 | 00000000 | 00000000 |
| 00d1c030 | 00000000 | 00000000 | 00000000 |
| 00d1c02c | 00000000 | 00000000 | 00000000 |
| 00d1c028 | 0000001d | 0000001d | 0000001d |
| 00d1c024 | 00000021 | Header 00000021 | 00000021 |
| 00d1c020 | 00000022 | 00000022 | 00000022 |
| 00d1c01c | 00000000 | 00000000 | 00000000 |
| 00d1c018 | 00000000 | 00000000 | 00000000 |
| 00d1c014 | 00000000 | Header 00000000 | 00000000 |
| 00d1c010 | 00000000 | 00000000 | 00000000 |
| 00d1c00c | 00000000 | 00000000 | 00000000 |
| 00d1c008 | 00000000 | 00000000 | 00000000 |
| 00d1c004 | 00000022 | Header 00000022 | 00000022 |
| 00d1c000 | 0000000b | 0000000b | 0000000b |
| 00d1bfcc | 0000000b | 0000000b | 0000000b |



- Not allocated (makes sense, since we will malloc it)
- Previous allocated
- size = 0010 0010 -> 0010 0000 = 0x20 = 32
- Allocated
- Previous is allocated
- size = 0000 1011 -> 0000 1000 = 0x08 = 8
- Allocated
- Previous is not allocated
- size = 0010 0001 -> 0010 0000 = 0x20 = 32
- Not allocated
- Previous is allocated
- size = 0100 0010 -> 0100 0000 = 0x40 = 64
- Way outside the heap, important later

| Address | Original value | 0xd1c008 | free(0xd1c028) |
|----------|----------------|--------------|----------------|
| | | After malloc | After free |
| 00d1c04c | 00000100 | 00000100 | 00000100 |
| 00d1c048 | 00000020 | + 00000020 | 00000020 |
| 00d1c044 | 00000042 | | |
| 00d1c040 | 00000021 | | |
| 00d1c03c | 0000001d | | |
| 00d1c038 | 00000018 | 00000018 | 00000018 |
| 00d1c034 | 00000000 | 00000000 | 00000000 |
| 00d1c030 | 00000000 | 00000000 | 00000000 |
| 00d1c02c | 00000000 | 00000000 | 00000000 |
| 00d1c028 | 0000001d | 0000001d | 0000001d |
| 00d1c024 | 00000021 | | |
| 00d1c020 | 00000022 | | |
| 00d1c01c | 00000000 | 00000000 | 00000000 |
| 00d1c018 | 00000000 | 00000000 | 00000000 |
| 00d1c014 | 00000000 | | |
| 00d1c010 | 00000000 | | |
| 00d1c00c | 00000000 | 00000000 | 00000000 |
| 00d1c008 | 00000000 | 00000000 | 00000000 |
| 00d1c004 | 00000022 | | |
| 00d1c000 | 0000000b | | |
| 00d1bfec | 0000000b | | |
| 00d1bfff | 0000000b | | |



- Not allocated
- Previous is allocated
- size = 0100 0010 -> 0100 0000 = 0x40 = 64
- Way outside the heap, important later
- Allocated
- Previous is not allocated
- size = 0010 0001 -> 0010 0000 = 0x20 = 32
- Not allocated (makes sense, since we will malloc it)
- Previous allocated
- size = 0010 0010 -> 0010 0000 = 0x20 = 32
- Allocated
- Previous is allocated
- size = 0000 1011 -> 0000 1000 = 0x08 = 8

| Address | Original value | 0xd1c008 | free(0xd1c028) |
|----------|----------------|------------------------|----------------|
| | | After malloc | After free |
| 00d1c04c | 00000100 | 00000100 | 00000100 |
| 00d1c048 | 00000020 | 00000020 + 00000020 | 00000020 |
| 00d1c044 | 00000042 | | |
| 00d1c040 | 00000021 | | |
| 00d1c03c | 0000001d | | |
| 00d1c038 | 00000018 | 00000018 | 00000018 |
| 00d1c034 | 00000000 | 00000000 | 00000000 |
| 00d1c030 | 00000000 | 00000000 | 00000000 |
| 00d1c02c | 00000000 | 00000000 | 00000000 |
| 00d1c028 | 0000001d | 0000001d | 0000001d |
| 00d1c024 | 00000021 | | |
| 00d1c020 | 00000022 | | |
| 00d1c01c | 00000000 | 00000000 | 00000000 |
| 00d1c018 | 00000000 | 00000000 | 00000000 |
| 00d1c014 | 00000000 | | |
| 00d1c010 | 00000000 | | |
| 00d1c00c | 00000000 | 00000000 | 00000000 |
| 00d1c008 | 00000000 | 00000000 | 00000000 |
| 00d1c004 | 00000022 | | |
| 00d1c000 | 0000000b | | |
| 00d1bfec | 0000000b | | |

Sorry I need space, try to remember the rest

We malloc(8) here, meaning 8 + header + footer = 16

- It will be allocated
- Previous will be allocated (0xb)
- Size = 16 = 0001 0000
- So -> 0001 0011 -> 0x13

0xd1c008 free(0xd1c028)

| Address | Original value | After malloc | After free |
|----------|----------------|--------------|------------|
| 00d1c04c | 00000100 | 00000100 | 00000100 |
| 00d1c048 | 00000020 | 00000020 | 00000020 |
| 00d1c044 | 00000042 | | |
| 00d1c040 | 00000021 | | |
| 00d1c03c | 0000001d | | |
| 00d1c038 | 00000018 | 00000018 | 00000018 |
| 00d1c034 | 00000000 | 00000000 | 00000000 |
| 00d1c030 | 00000000 | 00000000 | 00000000 |
| 00d1c02c | 00000000 | 00000000 | 00000000 |
| 00d1c028 | 0000001d | 0000001d | 0000001d |
| 00d1c024 | 00000021 | | |
| 00d1c020 | 00000022 | | |
| 00d1c01c | 00000000 | 00000000 | 00000000 |
| 00d1c018 | 00000000 | 00000000 | 00000000 |
| 00d1c014 | 00000000 | | |
| 00d1c010 | 00000000 | | |
| 00d1c00c | 00000000 | 00000000 | 00000000 |
| 00d1c008 | 00000000 | 00000000 | 00000000 |
| 00d1c004 | 00000022 | 0x13 | |
| 00d1c000 | 0000000b | 0xb | |
| 00d1bffc | 0000000b | 0xb | |

Sorry I need space, try to remember the rest

We malloc(8) here, meaning $8 + \text{header} + \text{footer} = 16$

- It will be allocated
- Previous will be allocated (0xb)
- Size = 16 = 0001 0000
- So -> 0001 0011 -> 0x13

Unaffected by the change

0xd1c008 free(0xd1c028)

| Address | Original value | After malloc | After free |
|----------|----------------|--------------|------------|
| 00d1c04c | 00000100 | 00000100 | 00000100 |
| 00d1c048 | 00000020 | 00000020 | 00000020 |
| 00d1c044 | 00000042 | | |
| 00d1c040 | 00000021 | | |
| 00d1c03c | 0000001d | | |
| 00d1c038 | 00000018 | 00000018 | 00000018 |
| 00d1c034 | 00000000 | 00000000 | 00000000 |
| 00d1c030 | 00000000 | 00000000 | 00000000 |
| 00d1c02c | 00000000 | 00000000 | 00000000 |
| 00d1c028 | 0000001d | 0000001d | 0000001d |
| 00d1c024 | 00000021 | | |
| 00d1c020 | 00000022 | | |
| 00d1c01c | 00000000 | 00000000 | 00000000 |
| 00d1c018 | 00000000 | 00000000 | 00000000 |
| 00d1c014 | 00000000 | | |
| 00d1c010 | 00000000 | | |
| 00d1c00c | 00000000 | 00000000 | 00000000 |
| 00d1c008 | 00000000 | 00000000 | 00000000 |
| 00d1c004 | 00000022 | 0x13 | 0xb |
| 00d1c000 | 0000000b | | 0xb |
| 00d1bffc | 0000000b | | |

Sorry I need space, try to remember the rest

We malloc(8) here, meaning $8 + \text{header} + \text{footer} = 16$

- It will be allocated
- Previous will be allocated (0xb)
- Size = 16 = 0001 0000
- So -> 0001 0011 -> 0x13

What about the rest of the free block? (just above)

- It will not be be allocated
- Previous will be allocated
- Size = $32 - 16 = 16 = 0001 0000$
- So -> 0001 0010 -> 0x12

| | 0xd1c008 | free(0xd1c028) | |
|----------|----------------|----------------|------------|
| Address | Original value | After malloc | After free |
| 00d1c04c | 00000100 | 00000100 | 00000100 |
| 00d1c048 | 00000020 | 00000020 | 00000020 |
| 00d1c044 | 00000042 | | |
| 00d1c040 | 00000021 | | |
| 00d1c03c | 0000001d | | |
| 00d1c038 | 00000018 | 00000018 | 00000018 |
| 00d1c034 | 00000000 | 00000000 | 00000000 |
| 00d1c030 | 00000000 | 00000000 | 00000000 |
| 00d1c02c | 00000000 | 00000000 | 00000000 |
| 00d1c028 | 0000001d | 0000001d | 0000001d |
| 00d1c024 | 00000021 | | |
| 00d1c020 | 00000022 | 0x12 | |
| 00d1c01c | 00000000 | 00000000 | 00000000 |
| 00d1c018 | 00000000 | 00000000 | 00000000 |
| 00d1c014 | 00000000 | 0x12 | |
| 00d1c010 | 00000000 | 0x13 | |
| 00d1c00c | 00000000 | 00000000 | 00000000 |
| 00d1c008 | 00000000 | 00000000 | 00000000 |
| 00d1c004 | 00000022 | 0x13 | |
| 00d1c000 | 0000000b | 0xb | |
| 00d1bffc | 0000000b | 0xb | |

Sorry I need space, try to remember the rest

We malloc(8) here, meaning $8 + \text{header} + \text{footer} = 16$

- It will be allocated
- Previous will be allocated (0xb)
- Size = 16 = 0001 0000
- So -> 0001 0011 -> 0x13

What about the rest of the free block? (just above)

- It will not be be allocated
- Previous will be allocated
- Size = $32 - 16 = 16 = 0001 0000$
- So -> 0001 0010 -> 0x12

| | 0xd1c008 | free(0xd1c028) | |
|----------|----------------|----------------|------------|
| Address | Original value | After malloc | After free |
| 00d1c04c | 00000100 | 00000100 | 00000100 |
| 00d1c048 | 00000020 | 00000020 | 00000020 |
| 00d1c044 | 00000042 | 0x42 | |
| 00d1c040 | 00000021 | 0x21 | |
| 00d1c03c | 0000001d | 0x1d | |
| 00d1c038 | 00000018 | 00000018 | 00000018 |
| 00d1c034 | 00000000 | 00000000 | 00000000 |
| 00d1c030 | 00000000 | 00000000 | 00000000 |
| 00d1c02c | 00000000 | 00000000 | 00000000 |
| 00d1c028 | 0000001d | 0000001d | 0000001d |
| 00d1c024 | 00000021 | 0x21 | |
| 00d1c020 | 00000022 | 0x12 | |
| 00d1c01c | 00000000 | 00000000 | 00000000 |
| 00d1c018 | 00000000 | 00000000 | 00000000 |
| 00d1c014 | 00000000 | 0x12 | |
| 00d1c010 | 00000000 | 0x13 | |
| 00d1c00c | 00000000 | 00000000 | 00000000 |
| 00d1c008 | 00000000 | 00000000 | 00000000 |
| 00d1c004 | 00000022 | 0x13 | |
| 00d1c000 | 0000000b | 0xb | |
| 00d1bfcc | 0000000b | 0xb | |



(not a real header.
Nothing changes)

Size unchanged, previous still
unallocated. stays the same

0xd1c008 free(0xd1c028)

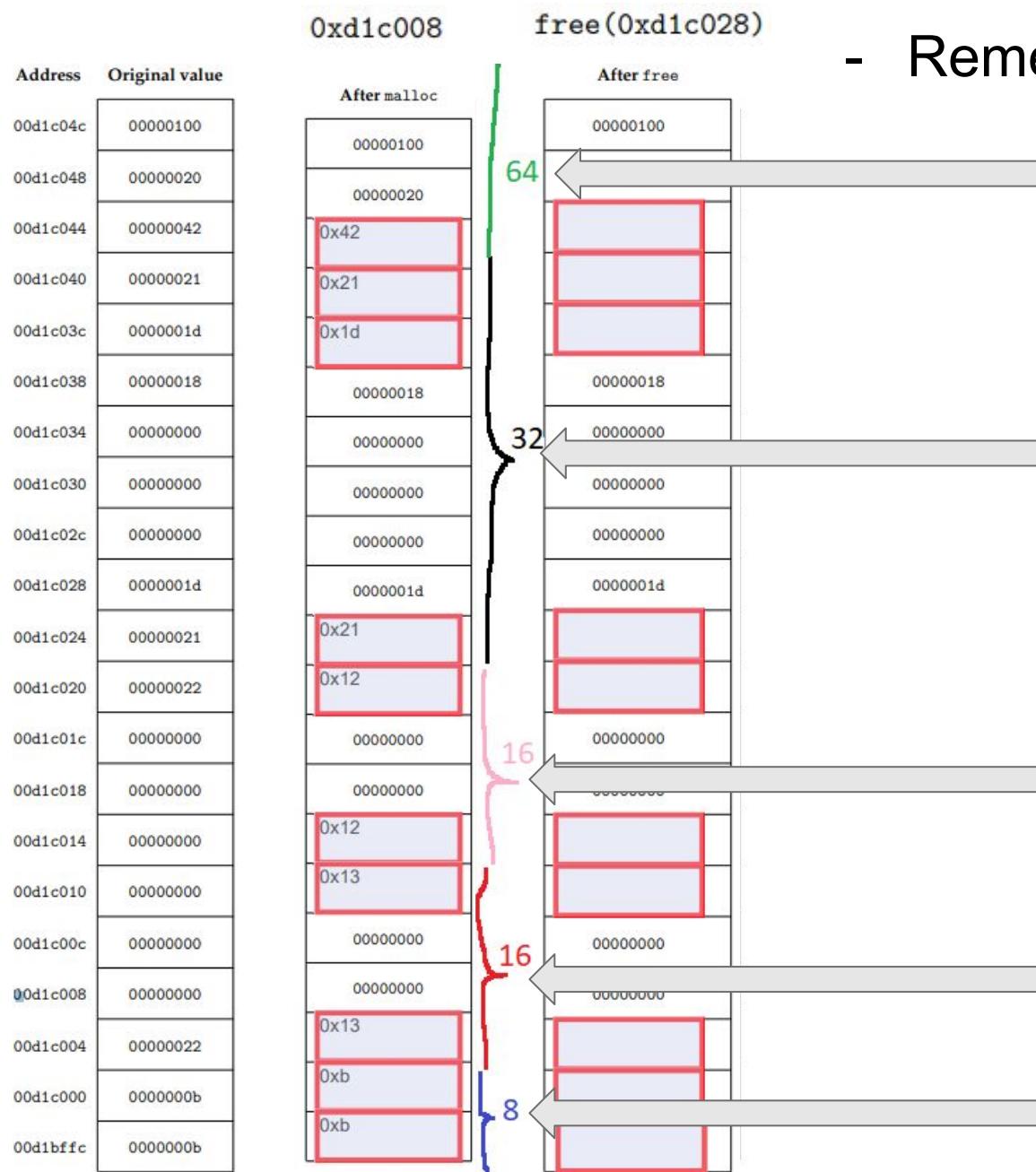
| Address | Original value | After malloc | After free |
|----------|----------------|--------------|------------|
| 00d1c04c | 00000100 | 00000100 | 00000100 |
| 00d1c048 | 00000020 | 00000020 | 00000020 |
| 00d1c044 | 00000042 | 0x42 | |
| 00d1c040 | 00000021 | 0x21 | |
| 00d1c03c | 0000001d | 0x1d | |
| 00d1c038 | 00000018 | 00000018 | 00000018 |
| 00d1c034 | 00000000 | 00000000 | 00000000 |
| 00d1c030 | 00000000 | 00000000 | 00000000 |
| 00d1c02c | 00000000 | 00000000 | 00000000 |
| 00d1c028 | 0000001d | 0000001d | 0000001d |
| 00d1c024 | 00000021 | 0x21 | |
| 00d1c020 | 00000022 | 0x12 | |
| 00d1c01c | 00000000 | 00000000 | 00000000 |
| 00d1c018 | 00000000 | 00000000 | 00000000 |
| 00d1c014 | 00000000 | 0x12 | |
| 00d1c010 | 00000000 | 0x13 | |
| 00d1c00c | 00000000 | 00000000 | 00000000 |
| 00d1c008 | 00000000 | 00000000 | 00000000 |
| 00d1c004 | 00000022 | 0x13 | |
| 00d1c000 | 0000000b | 0xb | |
| 00d1bfcc | 0000000b | 0xb | |

Stays the same

0x21 Footer

(not a real header.
Nothing changes)

Size unchanged, previous still
unallocated. Stays the same



- Remember:

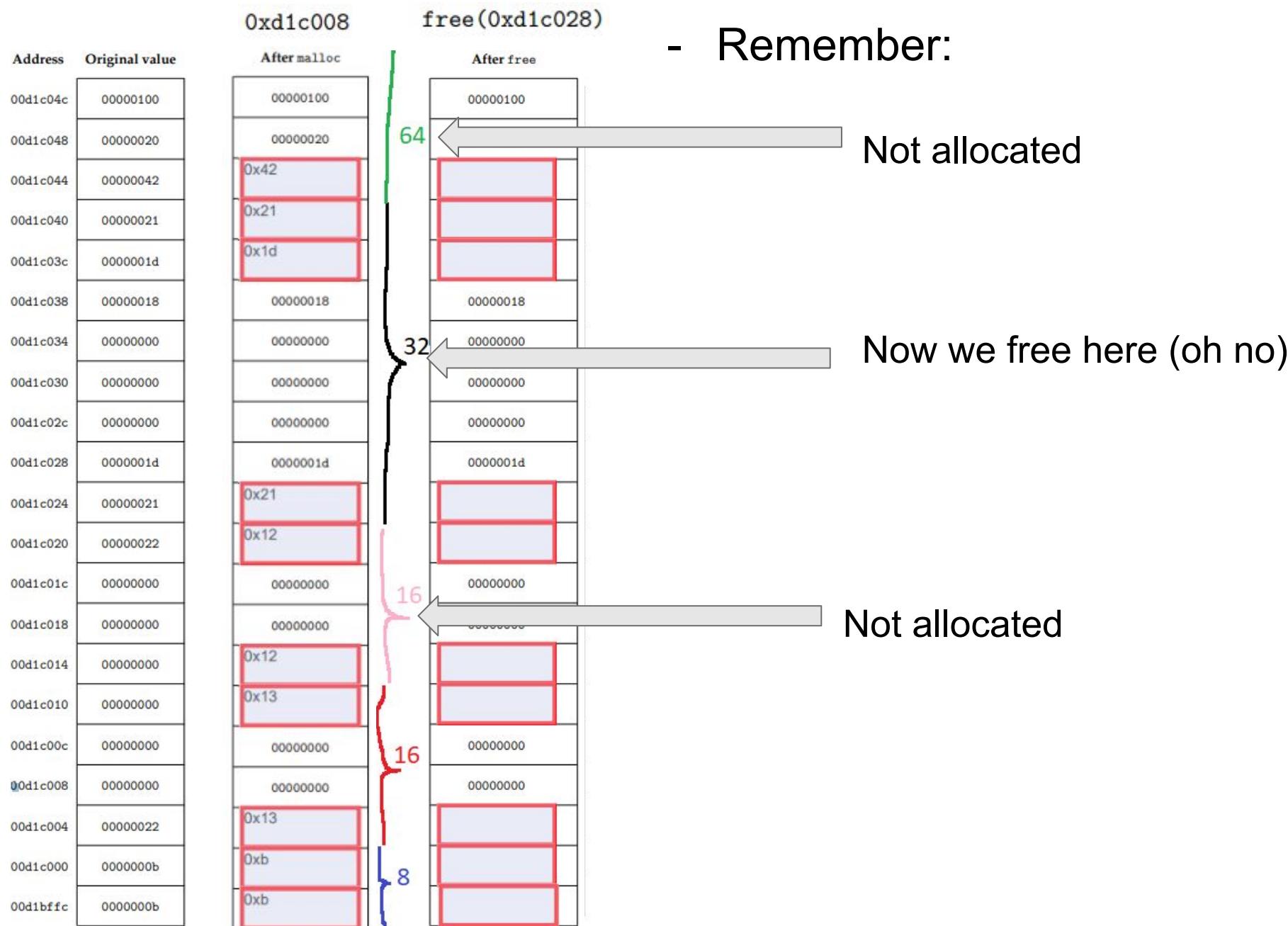
Not allocated

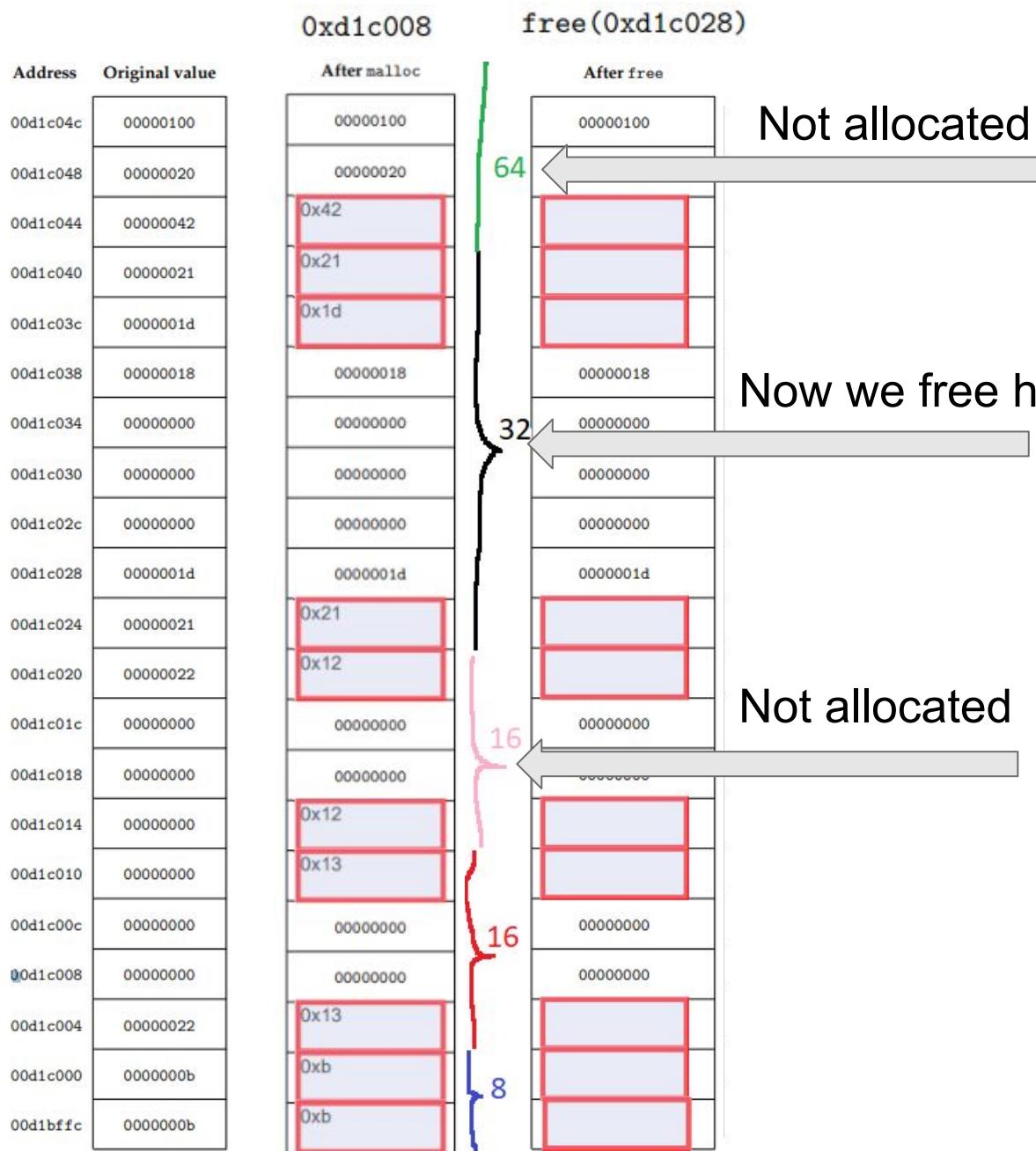
Allocated

Not allocated

Allocated (now)

Allocated





- But how? The **block above** is not even on the heap.
- Twofold: We solve the size using simple math, trusting the bits even if we cannot see the **footer**. And we do not change the footer, since it is not possible.

Simply, the heap we see must be correct for the exercise. We are not concerned with the non-visible heap

| Address | Original value | 0xd1c008 | free(0xd1c028) |
|----------|----------------|--------------|----------------|
| | | After malloc | After free |
| 00d1c04c | 00000100 | 00000100 | 00000100 |
| 00d1c048 | 00000020 | 00000020 | 00000020 |
| 00d1c044 | 00000042 | 0x42 | |
| 00d1c040 | 00000021 | 0x21 | |
| 00d1c03c | 0000001d | 0x1d | |
| 00d1c038 | 00000018 | 00000018 | 00000018 |
| 00d1c034 | 00000000 | 00000000 | 00000000 |
| 00d1c030 | 00000000 | 00000000 | 00000000 |
| 00d1c02c | 00000000 | 00000000 | 00000000 |
| 00d1c028 | 0000001d | 0000001d | 0000001d |
| 00d1c024 | 00000021 | 0x21 | |
| 00d1c020 | 00000022 | 0x12 | |
| 00d1c01c | 00000000 | 00000000 | 00000000 |
| 00d1c018 | 00000000 | 00000000 | 00000000 |
| 00d1c014 | 00000000 | 0x12 | |
| 00d1c010 | 00000000 | 0x13 | |
| 00d1c00c | 00000000 | 00000000 | 00000000 |
| 00d1c008 | 00000000 | 00000000 | 00000000 |
| 00d1c004 | 00000022 | 0x13 | |
| 00d1c000 | 0000000b | 0xb | |
| 00d1bffc | 0000000b | 0xb | |

It will not be allocated
 Its neighbour will be allocated
 Size = 16+32+64 = 0111 0000
 So: 0111 0010 = 0x72

| Address | Original value | After malloc | After free |
|----------|----------------|--------------|------------|
| 00d1c04c | 00000100 | 00000100 | 00000100 |
| 00d1c048 | 00000020 | 00000020 | 00000020 |
| 00d1c044 | 00000042 | 0x42 | |
| 00d1c040 | 00000021 | 0x21 | |
| 00d1c03c | 0000001d | 0x1d | |
| 00d1c038 | 00000018 | 00000018 | 00000018 |
| 00d1c034 | 00000000 | 00000000 | 00000000 |
| 00d1c030 | 00000000 | 00000000 | 00000000 |
| 00d1c02c | 00000000 | 00000000 | 00000000 |
| 00d1c028 | 0000001d | 0000001d | 0000001d |
| 00d1c024 | 00000021 | 0x21 | |
| 00d1c020 | 00000022 | 0x12 | |
| 00d1c01c | 00000000 | 00000000 | 00000000 |
| 00d1c018 | 00000000 | 00000000 | 00000000 |
| 00d1c014 | 00000000 | 0x12 | |
| 00d1c010 | 00000000 | 0x13 | |
| 00d1c00c | 00000000 | 00000000 | 00000000 |
| 00d1c008 | 00000000 | 00000000 | 00000000 |
| 00d1c004 | 00000022 | 0x13 | |
| 00d1c000 | 0000000b | 0xb | |
| 00d1bffc | 0000000b | 0xb | |

It will not be allocated
 Its neighbour will be allocated
 Size = 16+32+64 = 0111 0000
 So: 0111 0010 = 0x72

| Address | Original value | 0xd1c008 | free(0xd1c028) |
|-----------|----------------|--------------|----------------|
| | | After malloc | After free |
| 00d1c04c | 000000100 | 000000100 | 000000100 |
| 00d1c048 | 000000020 | 000000020 | 000000020 |
| 00d1c044 | 000000042 | 0x42 | 0x42 |
| 00d1c040 | 000000021 | 0x21 | 0x21 |
| 00d1c03c | 00000001d | 0x1d | 0x1d |
| 00d1c038 | 000000018 | 000000018 | 000000018 |
| 00d1c034 | 000000000 | 000000000 | 000000000 |
| 00d1c030 | 000000000 | 000000000 | 000000000 |
| 00d1c02c | 000000000 | 000000000 | 000000000 |
| 00d1c028 | 00000001d | 00000001d | 00000001d |
| 00d1c024 | 000000021 | 0x21 | 0x21 |
| 00d1c020 | 000000022 | 0x12 | 0x12 |
| 00d1c01c | 000000000 | 000000000 | 000000000 |
| 00d1c018 | 000000000 | 000000000 | 000000000 |
| 00d1c014 | 000000000 | 0x12 | 0x72 |
| 00d1c010 | 000000000 | 0x13 | 0x13 |
| 00d1c00c | 000000000 | 000000000 | 000000000 |
| 00d1c008 | 000000000 | 000000000 | 000000000 |
| 00d1c004 | 000000022 | 0x13 | 0x13 |
| 00d1c000 | 0000000b | 0xb | 0xb |
| 00d1bfffc | 0000000b | 0xb | 0xb |

Unchanged, again, because of immediate coalescing. They do not need to be changed so they just remain the same value, but now unallocated.

There was even a hint/specification for this, this time:

- Perform the minimum number of memory changes required.

Unchanged, unaffected by changes in any way

Correct?

| Address | Original value | After malloc | After free | Address | Original value | After malloc | After free |
|----------|----------------|--------------|------------|----------|----------------|--------------|------------|
| 00dic04c | 00000100 | 00000100 | 00000100 | 00d1c04c | 00000100 | 00000100 | 00000100 |
| 00dic048 | 00000020 | 00000020 | 00000020 | 00d1c048 | 00000020 | 00000020 | 00000020 |
| 00dic044 | 00000042 | 0x42 | 0x42 | 00d1c044 | 00000042 | 00000042 | 00000042 |
| 00dic040 | 00000021 | 0x21 | 0x21 | 00d1c040 | 00000021 | 00000021 | 00000021 |
| 00dic03c | 0000001d | 0x1d | 0x1d | 00d1c03c | 0000001d | 0000001d | 0000001d |
| 00dic038 | 00000018 | 00000018 | 00000018 | 00d1c038 | 00000018 | 00000018 | 00000018 |
| 00dic034 | 00000000 | 00000000 | 00000000 | 00d1c034 | 00000000 | 00000000 | 00000000 |
| 00dic030 | 00000000 | 00000000 | 00000000 | 00d1c030 | 00000000 | 00000000 | 00000000 |
| 00dic02c | 00000000 | 00000000 | 00000000 | 00d1c02c | 00000000 | 00000000 | 00000000 |
| 00dic028 | 0000001d | 0000001d | 0000001d | 00d1c028 | 0000001d | 0000001d | 0000001d |
| 00dic024 | 00000021 | 0x21 | 0x21 | 00d1c024 | 00000021 | 00000021 | 00000021 |
| 00dic020 | 00000022 | 0x12 | 0x12 | 00d1c020 | 00000022 | 00000012 | 00000012 |
| 00dic01c | 00000000 | 00000000 | 00000000 | 00d1c01c | 00000000 | 00000000 | 00000000 |
| 00dic018 | 00000000 | 00000000 | 00000000 | 00d1c018 | 00000000 | 00000000 | 00000000 |
| 00dic014 | 00000000 | 0x12 | 0x72 | 00d1c014 | 00000000 | 00000012 | 00000072 |
| 00dic010 | 00000000 | 0x13 | 0x13 | 00d1c010 | 00000000 | 00000013 | 00000013 |
| 00dic00c | 00000000 | 00000000 | 00000000 | 00d1c00c | 00000000 | 00000000 | 00000000 |
| 00dic008 | 00000000 | 00000000 | 00000000 | 00d1c008 | 00000000 | 00000000 | 00000000 |
| 00dic004 | 00000022 | 0x13 | 0x13 | 00d1c004 | 00000022 | 00000013 | 00000013 |
| 00dic000 | 0000000b | 0xb | 0xb | 00d1c000 | 0000000b | 0000000b | 0000000b |
| 00d1bfff | 0000000b | 0xb | 0xb | 00d1bfff | 0000000b | 0000000b | 0000000b |

No bonus question

Likely due to a change in the format

From a 8-hour take home exam to a 4 hour written exam

Extra

Either for reading or in case I have extra extra time

General Heap tips

- There will almost always be some trick, such that you actually have to understand the heap to be able to complete the exercise
- Even with that, this becomes easily manageable with enough practice.
- At the first step, always analyse the entire initial heap, before doing any of the heap functions
- Even if it is recommended to analyse the length of a block to see where the next block starts, in case of time trouble, and/or having good intuition of block lengths (hexadecimal is not that far from decimal, especially in low numbers) you may be able to guess which are headers and which are not
- Read. The. Stated. Rules

Semaphore tips

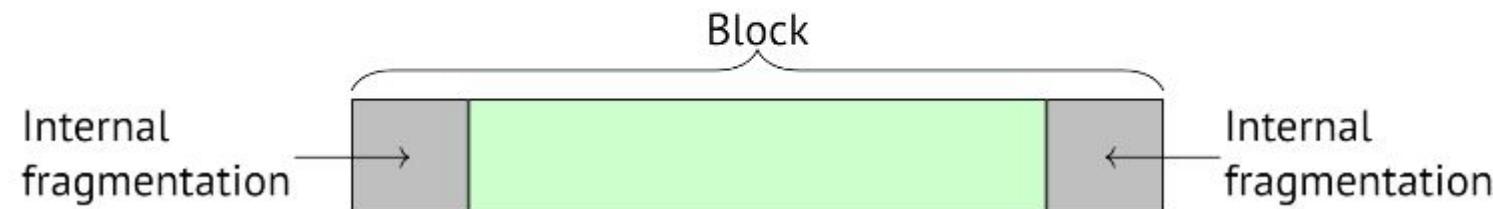
- If both lock same place, this is no-go zone
- Draw no-go zones
- If at any point can go neither up nor right. This is a deadlock
- Else not deadlock

Internal fragmentation

- Internal fragmentation we saw in our second example, and is the result of standards (such as a block being a multiple of 8 bytes or needing to eat up a block that cannot exist on its own)
- The fragmentation is, then, the extra space taken up that is not needed

Internal fragmentation occurs when the payload is smaller than the block size.

*(Shamelessly stolen from Troels slides
Virtual memory ii)*



External fragmentation

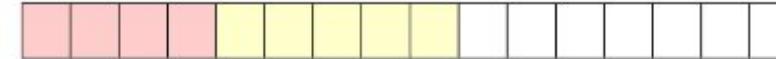
- External fragmentation is when a heap with a lot of allocating and deallocating, after long time will be a mess. What was once a heap with blocks neatly adjacent is now a lot of blocks everywhere with inconvenient space in between.

Occurs when there is enough aggregate heap memory, but no single free block is large enough.

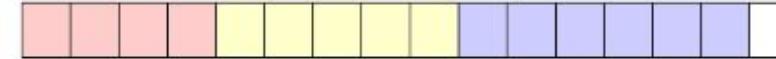
```
p1 = malloc(4*4);
```



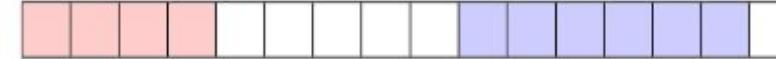
```
p2 = malloc(5*4);
```



```
p3 = malloc(6*4);
```



```
free(p2);
```



```
p4 = malloc(6*4);
```

No free block with room for six words.

Depends on future requests, so difficult to measure.

(Shamelessly stolen from Troels slides
- Virtual memory ii)

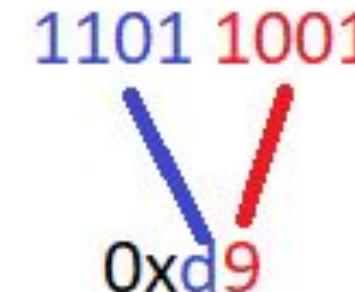
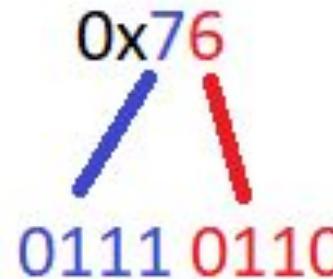
← We kind of wish the blue block could easily be moved 5 to the left for some nice adjacency.

Fragmentation Intuition

- Internal and external fragmentation are called so, because internal fragmentation exists within blocks and external fragmentation happens outside (between) blocks. This can be a way to remember them.

How to convert between hexadecimal, binary, and decimal

- https://www.youtube.com/watch?v=D_YC6DSPpQE
- No, yeah, there already exists a great resource I won't be trying to beat.
- But essentially the big thing is that since a hexadecimal digit can represent 16 different values, and the same is the case with 4 binary digits (since $2^4 = 16$) then translating between them is really easy, just from least to most take the 4 binary digits and each is a hexadecimal digit, and reversed. For example:



Fin.