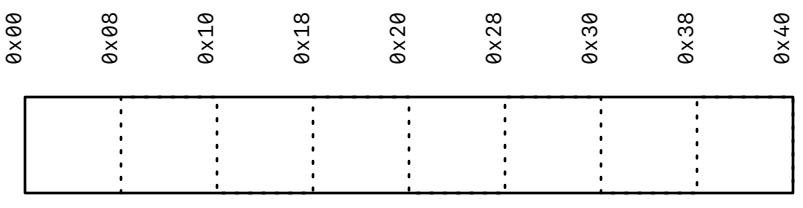
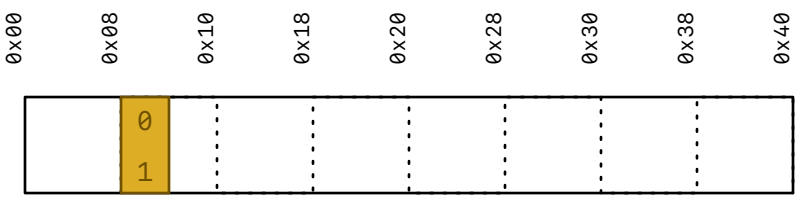
CSCI 380 — Memory Allocator Lab — Getting Started Visual

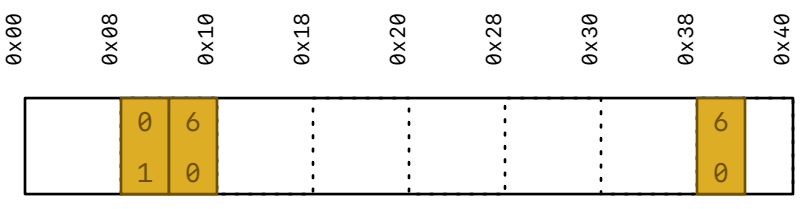
We first must grow/allocate the heap to some size.
I suggest `N * sizeof(word) * 2`. In this case, `N=4`



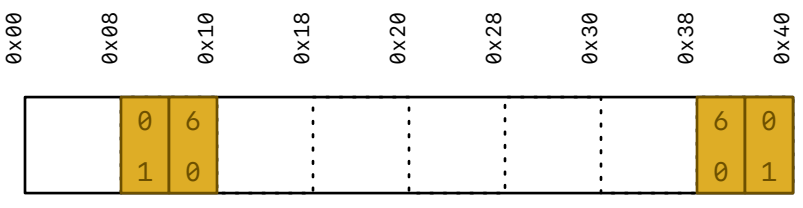
We now need to somehow "initialize" our heap to some default state. First we can place the false begin "footer"



Then we can place the initial free block
Remember the `N` from earlier?
The block size is `2 * (N - 1)` words



Finally, we will place the false end "header"



Potentially useful type aliases:

```
typedef uint64_t word;
typedef uint32_t tag;
typedef uint8_t  byte;
typedef byte*    address;
```

Potentially useful function prototypes:

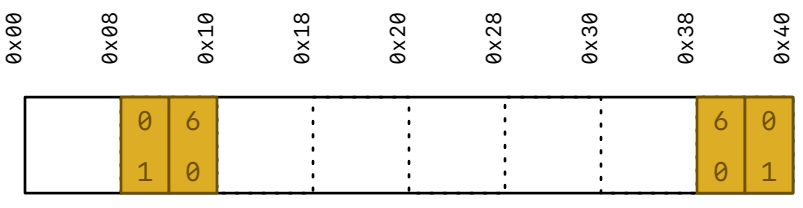
```
/* returns HDR address given basePtr */
tag*
header (address);

/* return true IFF block is allocated */
bool
isAllocated (address);

/* returns size of block (words) */
uint32_t
sizeOf (address);

/* returns FTR address given basePtr */
tag*
footer (address);

/* gives the basePtr of next block */
address
nextBlock (address);

/* gives the basePtr of prev block */
address
prevBlock (address);

/* basePtr, size, allocated */
void
makeBlock (address, uint32_t, bool);

/* basePtr — toggles alloced/free */
void
toggleBlock (address);
```

Note: `0x10` is the "base" of the heap.
You can start searching for blocks from here

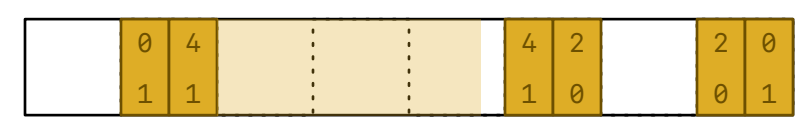
What happens when . . . `malloc(22)`

- The parameter value of 22 is in bytes!
- Our allocator "speaks" in words
- All blocks must be double-word aligned

```
uint32_t const OVERHEAD = 2 * sizeof(tag);                // 8
uint32_t const DWORD = 2 * sizeof(word);                  // 16
uint32_t const SIZE = space + OVERHEAD;                   // 30
uint32_t const DWORD_SIZE = (SIZE + (DWORD - 1)) / DWORD; // (30 + 15) / 2 = 2
uint32_t const WORD_SIZE = DWORD_SIZE * 2;                // 4
```



Search for a free block . . . found at `0x10`

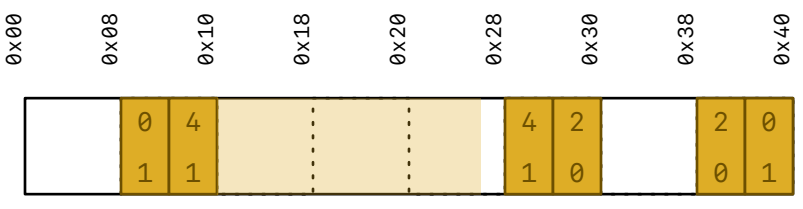


- Block found at 0x10 is 6 words
- 6 - 4 = 2 — we should split the block!
- The 4-word block should be marked as allocated
- The 2-word block shall be free

We ultimately return the address `0x10` after the call to `malloc(22)`

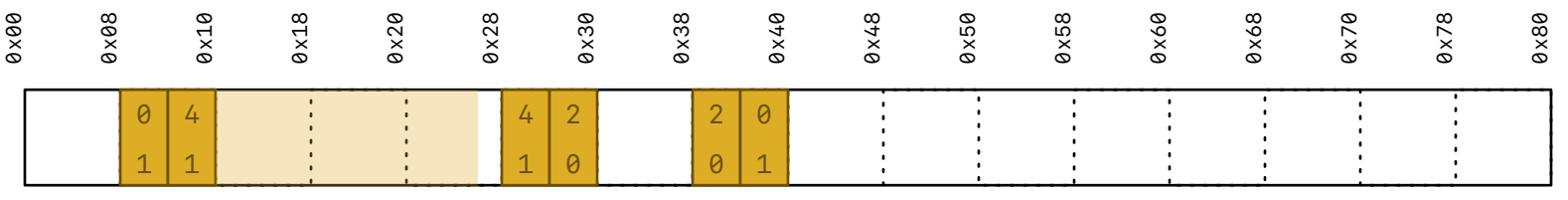
What happens when . . . `malloc(26)`

Resulting block size is 6 blocks . . . oof

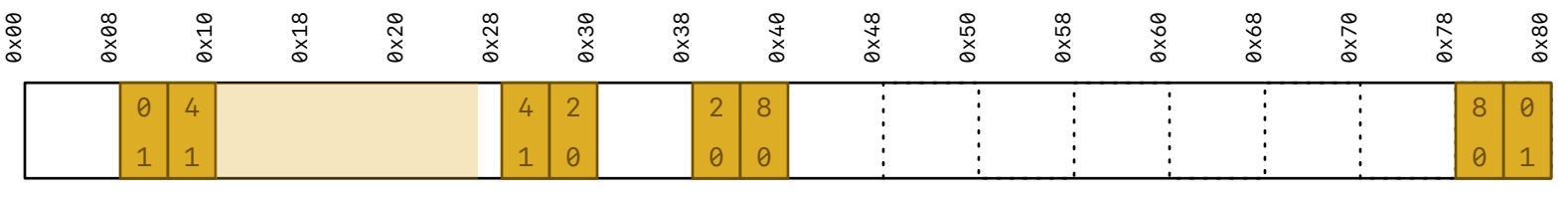


Search for a free block with size >= 6 . . . uh oh

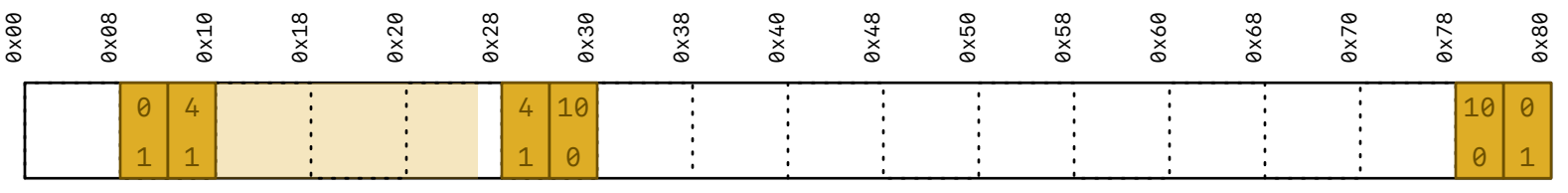
Extend heap by at least 4 words let's expand by 8 words



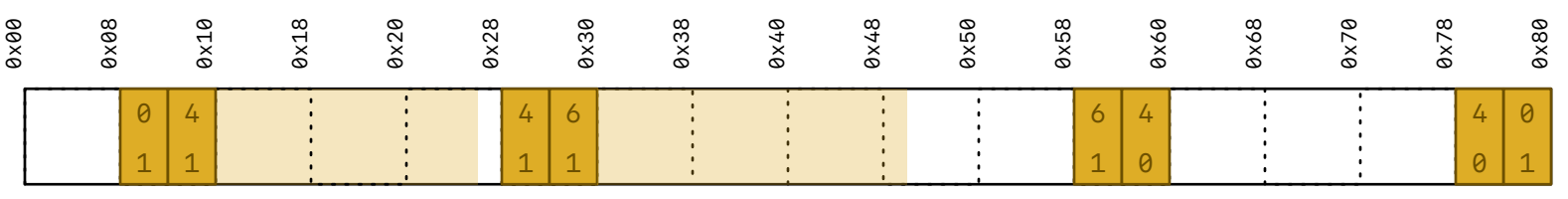
Now we must place the new "free" block and false end "header"



We have two adjacent free blocks — let's coalesce them!



Now we can finally split the free block (like the last malloc)

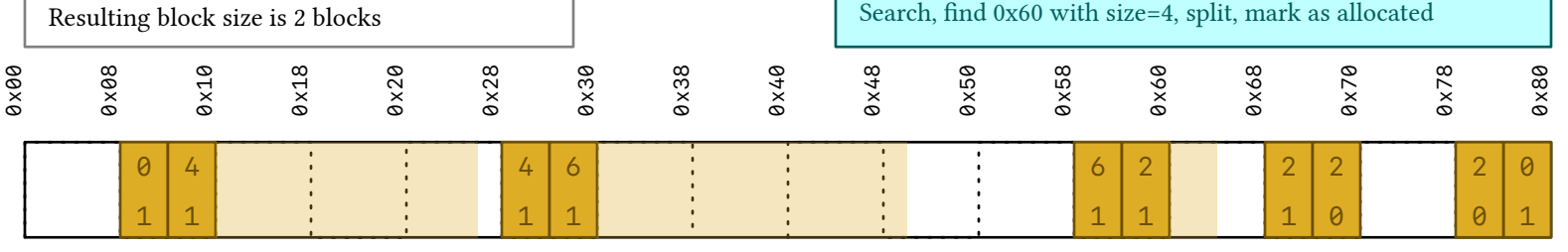


Return the address `0x30`

What happens when . . . `malloc(4)`

Resulting block size is 2 blocks

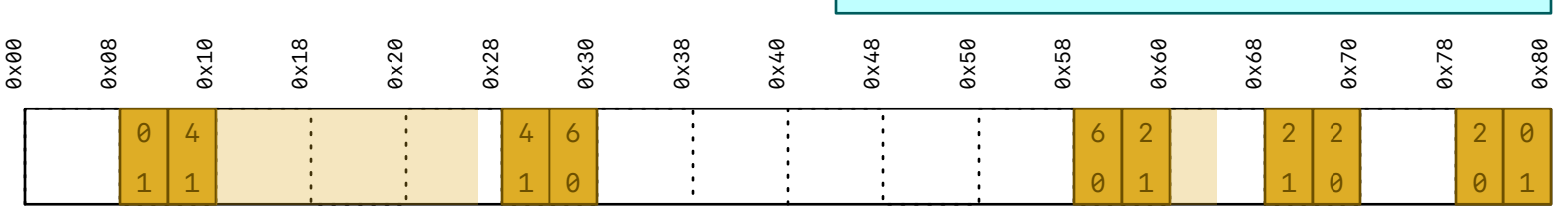
Search, find 0x60 with size=4, split, mark as allocated



Return the address `0x60`

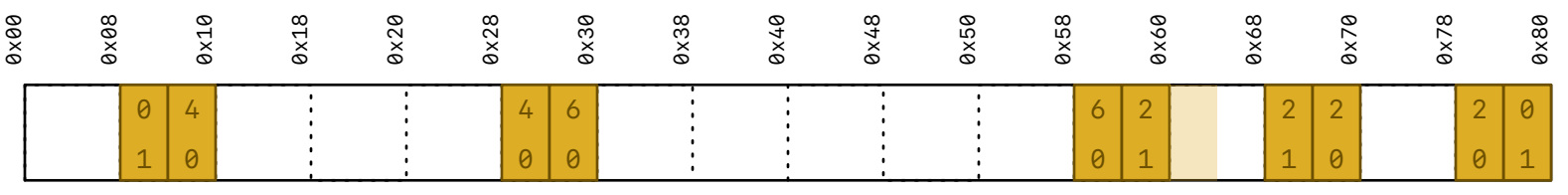
What happens when . . . `free(0x30)`

Toggle the block — check for adjacent free blocks

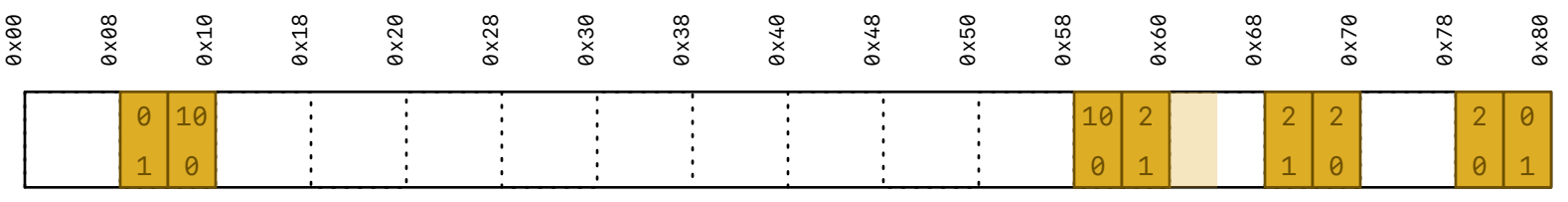


What happens when . . . `free(0x10)`

Toggle the block — check for adjacent free blocks

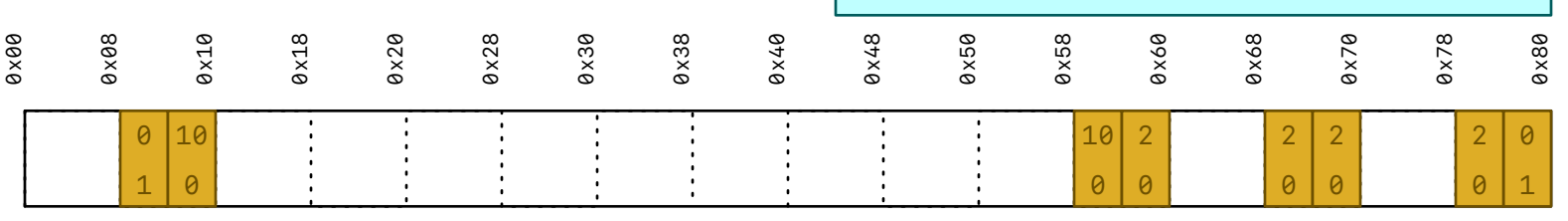


Coalesce the two adjacent free blocks



What happens when . . . `free(0x60)`

Toggle the block — check for adjacent free blocks



Coalesce the three adjacent free blocks