● It’s hard
● Not all schools put you through the gauntlet like this
● You learned a ton in the past few weeks
● If you can stumble your way through assembly, you have met the learning goals as far as I’m concerned
assign6a/6: Heap Allocator

- Walkthrough already posted
- Quick tour
assign6a/6: Heap Allocator

Estimate time spent on assign6: implicit list heap allocator

10 responses

Estimate time spent on assign6: explicit list heap allocator

10 responses
Agenda

- What’s a Heap Allocator?
- Implicit List Terminology
- Headers
- Blocks
- Implicit List
What’s a Heap Allocator?
malloc and free

- Remember the heap?
- Every call to malloc involves reserving part of it (“allocating it”) and returning a pointer to it to the caller
- Every call to free involves “unreserving” (“deallocating”) part of it
- The heap starts off as a huge chunk of free memory, and becomes segmented into blocks of allocated and freed memory
- Heap allocators manage all of this - they are the behind-the-scenes of malloc and free
Implicit List Terminology
The heap is made up of an implicit list of blocks

<table>
<thead>
<tr>
<th>Req. 1</th>
<th>Req. 2</th>
<th>Req. 3</th>
<th>Req. 4</th>
<th>Req. 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free</td>
<td>Free</td>
<td>Free</td>
<td>Free</td>
<td>Free</td>
</tr>
</tbody>
</table>
Each block consists of a header and a payload

<table>
<thead>
<tr>
<th>0x10</th>
<th>0x11</th>
<th>0x12</th>
<th>0x13</th>
<th>0x14</th>
<th>0x15</th>
<th>0x16</th>
<th>0x17</th>
<th>0x18</th>
<th>0x19</th>
</tr>
</thead>
<tbody>
<tr>
<td>Req. 1</td>
<td>Free</td>
<td>Req. 2</td>
<td>Free</td>
<td>Req. 3</td>
<td>Free</td>
<td>Req. 4</td>
<td>Free</td>
<td>Req. 5</td>
<td>Free</td>
</tr>
</tbody>
</table>

- **24 Used**
- **e**
Each header consists of the size and the allocated bit

<table>
<thead>
<tr>
<th>0x10</th>
<th>0x11</th>
<th>0x12</th>
<th>0x13</th>
<th>0x14</th>
<th>0x15</th>
<th>0x16</th>
<th>0x17</th>
<th>0x18</th>
<th>0x19</th>
</tr>
</thead>
<tbody>
<tr>
<td>Req. 1</td>
<td>Free</td>
<td>Req. 2</td>
<td>Free</td>
<td>Req. 3</td>
<td>Free</td>
<td>Req. 4</td>
<td>Free</td>
<td>Req. 5</td>
<td>Free</td>
</tr>
</tbody>
</table>

0bxxxxxxxx 0bxxxxxxxx 0bxxxxxxxx 0bxxxxxxxx
0bxxxxxxxx 0bxxxxxxxx 0bxxxxxxxx 0bxxxxxxxx00a
Implicit List operations work at 3 levels

<table>
<thead>
<tr>
<th>0x10</th>
<th>0x11</th>
<th>0x12</th>
<th>0x13</th>
<th>0x14</th>
<th>0x15</th>
<th>0x16</th>
<th>0x17</th>
<th>0x18</th>
<th>0x19</th>
</tr>
</thead>
<tbody>
<tr>
<td>Req. 1</td>
<td><em>Free</em></td>
<td>Req. 2</td>
<td><em>Free</em></td>
<td>Req. 3</td>
<td><em>Free</em></td>
<td>Req. 4</td>
<td><em>Free</em></td>
<td>Req. 5</td>
<td><em>Free</em></td>
</tr>
</tbody>
</table>

- Work within the implicit list
- Work within a block
- Work within a header
Headers
Headers

- Required by the handout:
  - 8 bytes, and must be able to hold any size `malloc` takes in
  - So what type should a header be?
Headers

- Required by the handout:
  - 8 bytes, and must be able to hold any size `malloc` takes in
  - `typedef size_t header_t;`
Headers

- Required by the handout:
  - 8 bytes, and must be able to hold any size `malloc` takes in
  - `typedef size_t header_t;`
  - Sizes must be a multiple of 8 (use the `ALIGNMENT` constant)
  - An “allocated bit” must be stored within as well
  - Wait, we have to hold 64 bits for the size, and also 1 bit for whether it’s allocated - do we need 65 bits?
Some size_t numbers that are multiples of 8

<table>
<thead>
<tr>
<th></th>
<th>0b00000000</th>
<th>0b00000000</th>
<th>0b00000000</th>
<th>0b00000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0b00000000</td>
<td>0b00000000</td>
<td>0b00000000</td>
<td>0b00000000</td>
</tr>
<tr>
<td>8</td>
<td>0b00000000</td>
<td>0b00000000</td>
<td>0b00000000</td>
<td>0b00000000</td>
</tr>
<tr>
<td>16</td>
<td>0b00000000</td>
<td>0b00000000</td>
<td>0b00000000</td>
<td>0b00000000</td>
</tr>
<tr>
<td>24</td>
<td>0b00000000</td>
<td>0b00000000</td>
<td>0b00000000</td>
<td>0b00000000</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>8</td>
<td>16</td>
<td>24</td>
</tr>
<tr>
<td>---</td>
<td>------------------------</td>
<td>------------------------</td>
<td>------------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>0</td>
<td>0b000000000</td>
<td>0b000000000</td>
<td>0b000000000</td>
<td>0b000000000</td>
</tr>
<tr>
<td></td>
<td>0b000000000</td>
<td>0b000000000</td>
<td>0b000000000</td>
<td>0b000000000</td>
</tr>
<tr>
<td>8</td>
<td>0b000000000</td>
<td>0b000000000</td>
<td>0b000000000</td>
<td>0b000000000</td>
</tr>
<tr>
<td></td>
<td>0b000000000</td>
<td>0b000000000</td>
<td>0b000000000</td>
<td>0b000000000</td>
</tr>
<tr>
<td>16</td>
<td>0b000000000</td>
<td>0b000000000</td>
<td>0b000000000</td>
<td>0b000000000</td>
</tr>
<tr>
<td></td>
<td>0b000000000</td>
<td>0b000000000</td>
<td>0b000000000</td>
<td>0b000000000</td>
</tr>
<tr>
<td>24</td>
<td>0b000000000</td>
<td>0b000000000</td>
<td>0b000000000</td>
<td>0b000000000</td>
</tr>
<tr>
<td></td>
<td>0b000000000</td>
<td>0b000000000</td>
<td>0b000000000</td>
<td>0b000000000</td>
</tr>
</tbody>
</table>
Use the LSB as the allocated bit

- Arbitrarily decide that 0 = FREE, 1 = ALLOC (you can reverse this if you like, as long as you always follow the rule!)

- This has an actual value of 25. This value is meaningless.
- We know that when we extract out the size, the LSB should be 0.
- We can extract the actual size by zeroing out the LSB.
- We can extract the allocated bit by reading only the LSB.

25
0b00000000 0b00000000 0b00000000 0b00000000
0b00000000 0b00000000 0b00000000 0b00011001

24
0b00000000 0b00000000 0b00000000 0b00000000
0b00000000 0b00000000 0b00000000 0b00011000

(allocated bit = ALLOC)
Header-level Operations

- **Write:**
  - `bool is_free(header_t *header);`(group)
  - `void set_header(header_t *header, size_t size, int status);`(group)
  - `size_t get_size(header_t *header);`(partner)

- **You may assume:**
  - `typedef size_t header_t;`
  - `the constants FREE and ALLOC are defined (one of them is 1, the other is 0)`
Blocks
Pointers within a Block

- Header pointer (same as block pointer)
- Payload pointer (only used to return to user - you will never write to the payload for the implicit list!)
- Next block pointer
Block-level Operations

- Write:
  - `void *header2payload(header_t *header);` (group)
  - `header_t *payload2header(void *payload);` (partner)
  - `header_t *next_block(header_t *header);` (partner)
  - Is it possible to write `prev_block`?

- You may assume:
  - The existence of the header-level functions we wrote earlier (don’t examine the values of headers all over again!)
Implicit List
The implicit list must fit the entire heap exactly

<table>
<thead>
<tr>
<th></th>
<th>40, free</th>
<th>72, allocated</th>
<th>24, free</th>
<th>96, allocated</th>
</tr>
</thead>
</table>
Note: at the very beginning, the heap is just one big giant free block
This is after a single malloc(30) (Sizes include headers)

40, allocated

160, free
This is after we’ve done some mallocs and frees

- 40, free
- 72, allocated
- 24, free
- 96, allocated
Overlaps NOT allowed! (header pointer + size must point to next block)
The last block must take up exactly the heap remainder (last block pointer + size must be end of heap)

40, free

72, allocated

24, free

96, allocated
Don’t treat these slides as a list of validity characteristics, by the way. Sizes must be a multiple of 8, too, for example.

| 40, free | 72, allocated | 24, free | 96, allocated |
next\_block, which we already wrote, helps us iterate through blocks

- 40, free
- 72, allocated
- 24, free
- 96, allocated
Implicit list-level Operations

- Write:
  - `size_t count_blocks(header_t *start);` (group)
  - `header_t *find_block_of_64(header_t *start);` (partner)

- You may assume:
  - The existence of the block-level functions we wrote earlier (don’t parse pointers all over again!)