CS107 Lecture 23: Two-Minute Etudes

Problem 1: Explicit Allocator Calisthenics

These Problem 1 etudes assume the explicit allocator from lecture.

- Each block begins with an eight-byte pre-block header that stores the payload size. The header's least significant bit is the allocation bit, where 1 means in use and 0 means free.
- Payload sizes are always multiples of 8, and the minimum payload size is 16 bytes.
- The first 16 bytes of a free block's payload store two pointers: the address of the previous free block's header, followed by the address of the next free block's header.
- The global **`free_list`** holds the address of the first free block's header. You can assume the list is last-in, first-out, so a newly freed block is always prepended to the list.
- **`free`** right-coalesces if the right neighbor is free, and **`realloc`** resizes in place if possible.

a. Assuming everything summarized above, answer the following questions:

   - What payload sizes result from calls to **`malloc(1)`**, **`malloc(17)`**, and **`malloc(24)`**?
   - Assuming a free block has a 48-byte payload and is chosen to service a **`malloc(24)`** request, what becomes of the excess payload?
   - What if the same free block were chosen to service a **`malloc(32)`** request instead?

b. A free block's two pointers reside in its payload, so reaching them requires some clever casting. Implement each of the two functions below, assuming **`hdr`** is the address of a free block's header. (Each one needs just a single line of code.)

```
size_t *get_prev(size_t *hdr) {


}
```

```
size_t *get_next(size_t *hdr) {


}
```

c. The figure below shows the entirety of a 96-byte heap. Assume that **`free_list`** is **`0x58`**, **`p`** is **`0x30`**, and **`q`** is **`0x48`**.

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58	0x60	0x68
16 Free	0x58	NULL	**16 Used**			**16 Used**			**16 Free**	NULL	0x10

Suppose the client calls **`free(p)`**.

   - Four of the eight-byte words in the heap change. Which ones, and what does each hold afterward?

- What's the new value of **`free_list`**?

d. Rewind to the original figure and assume the client calls **`free(q)`** instead.

- What's the payload size of the free block that results?
- Starting with **`free_list`**, list the header addresses on the free list, in order.

e. Assume that **`memory`** addresses a block with a 48-byte payload. Its right neighbor is free and carries a 24-byte payload, and the block after that one is in use. For each call below, taken on its own, decide whether **`realloc`** can resize in place, and give the payload size of **`memory`**'s block afterward.

- **`memory = realloc(memory, 45);`**
- **`memory = realloc(memory, 30);`**
- **`memory = realloc(memory, 20);`**
- **`memory = realloc(memory, 60);`**

Problem 2: A Bidirectional Implicit Allocator

These Problem 2 etudes assume an entirely different allocator design—one that's implicit at heart but has a different header structure than any of those presented in lecture.

- There's no free list. Each block starts with an eight-byte pre-block header, immediately followed by its payload, just as in our implicit allocator.
- The **lower** 32 bits of the header store the block's own payload size, which is always a multiple of 8. The least significant bit is the allocation bit, where 1 means in use and 0 means free.
- The **upper** 32 bits of the header store the payload size of the previous block, which is the one just before it in memory. The first block's upper 32 bits are all zeroes.
- **`free`** coalesces with the left neighbor and the right neighbor whenever they're free.

a. Headers now carry two sizes, so reading and writing them takes a little more care.

- A header holds **`0x0000002000000019`**. What's the block's payload size, is the block free or in use, and what's the previous block's payload size?
- What eight-byte value, in hex, belongs in the header of a free block with a 16-byte payload whose left neighbor has a 48-byte payload?

b. The **`get_size`** from lecture returned **`*hdr & ~0x7UL`**, and that no longer works, because the upper half of the header would come along for the ride. Implement **`get_size`** for this allocator and implement **`get_prev_size`** as well. Each is just one line of code.

```
size_t get_size(size_t *hdr) {

}
```

```
size_t get_prev_size(size_t *hdr) {

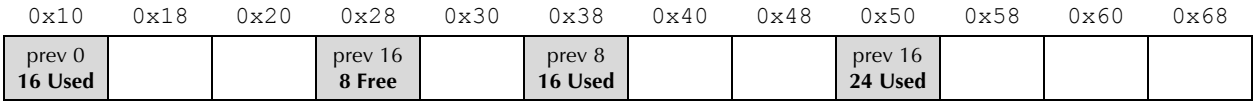
}
```

c. Implement **`get_prev_header`**, which returns the address of the header just before **`hdr`** in memory. You may assume **`hdr`** isn't the first header in the heap.

```
size_t *get_prev_header(size_t *hdr) {

}
```

d. The figure below shows all of a 96-byte heap. Each shaded rectangle is a header, and it shows the previous block's payload size above the block's own payload size and status.



Assume **`q`** is the address **`0x40`** and the client calls **`free(q)`**.

- Where's the header of the resulting free block, and what's that block's payload size?
- One other header needs to be updated. Which one, and why?

e. Assume that a free block's header is at **`0x100`**, that its payload is 40 bytes, and that it's followed by another block. The block is chosen to service a **`malloc(16)`** request, and the excess is split off as a new free block immediately to its right. Three headers need to be written or updated. For each one, give its address and say what changes.

f. Every design decision comes with some tradeoffs.

- What's the largest payload size a block can have now?
- Name one thing this allocator can do that the implicit allocator from lecture can't, and one new chore it takes on in exchange.

Problem 1 Answers:

a. There are three sub-questions here:
   - 16, 24, and 24 bytes. Sizes are rounded up to a multiple of 8, and no payload is smaller than 16.
   - The 24 leftover bytes become a new free block: 8 bytes of header and a 16-byte payload.
   - Only 16 bytes would be left over, which is 8 short of what a free block needs. They stay behind as padding, and the block keeps its 48-byte payload.

b. `size_t *get_prev(size_t *hdr) { return *(size_t **)(hdr + 1); }`
   `size_t *get_next(size_t *hdr) { return *(size_t **)(hdr + 2); }`
   Returning `(size_t *)hdr[1]` and `(size_t *)hdr[2]` works just as well.

c. There are two sub-questions here:
   - The header at `0x28` goes from `0x11` to `0x10`. The word at `0x30` becomes `NULL`, since the block is the new front of the list. The word at `0x38` becomes `0x58`, the old front. The word at `0x60` becomes `0x28`, since the old front's previous pointer needs to point back. The right neighbor is in use, so there's nothing to coalesce.
   - `0x28`.

d. There are two sub-questions here:
   - 40 bytes: 16 from each payload, plus the 8 bytes of the absorbed header at `0x58`.
   - `0x40`, then `0x10`. The block at `0x58` is spliced out of the list, and the merged block at `0x40` goes on the front.

e. There are four sub-questions here:
   - In place, and still 48 bytes. The request fits in the payload that's already there.
   - In place, and still 48 bytes. The request rounds up to 32, and the 16 bytes of excess are 8 short of what a free block needs.
   - In place, and 24 bytes. The 24 bytes of excess are exactly enough for a new free block with a 16-byte payload.
   - In place, and 80 bytes. The request rounds up to 64, and absorbing the right neighbor gives 48 + 8 + 24 = 80. The 16 bytes of excess are too few to split off.

Problem 2 Answers:

a. There are two sub-questions here:
   - The payload is 24 bytes and the block is in use, since the lower half is **`0x19`**. The previous block's payload is 32 bytes, since the upper half is **`0x20`**.
   - **`0x0000003000000010`**.

b.
```
size_t get_size(size_t *hdr) { return *hdr & 0xFFFFFFF8UL; }
size_t get_prev_size(size_t *hdr) { return *hdr >> 32; }
```
Masking with **`~0x7U`** works for **`get_size`** as well. That's the mask that was broken in the Lecture 22 etudes, and here its zeroed upper half is exactly what's needed.

c.
```
size_t *get_prev_header(size_t *hdr) {
   return (size_t *)((char *)hdr - sizeof(size_t) - get_prev_size(hdr));
}
```

d. There are two sub-questions here:
   - The header is at **`0x28`**, and the payload is 32 bytes: 8 from the left neighbor, 16 from **`q`**'s block, and 8 for **`q`**'s absorbed header at **`0x38`**.
   - The header at **`0x50`**. Its upper half goes from 16 to 32, since the block before it just grew.

e. Here are the three headers:
   - The header at **`0x100`** now records a 16-byte payload that's in use. Its upper half doesn't change.
   - A new header at **`0x118`** records a free block with a 16-byte payload, which is 40 - 16 - 8, and a previous size of 16.
   - The header at **`0x130`** belongs to the block that follows, and its upper half goes from 40 to 16.

f. There are two sub-questions here:
   - 2^{32} - 8 bytes, which is just shy of 4GB. A size has to fit in 32 bits and still be a multiple of 8.
   - It can find the block to its left, so it can walk the heap backward and coalesce in both directions. In exchange, every split, merge, or resize has to update the following block's header as well.