CS107 Lecture 22: Two-Minute Etudes

Problem: Implicit Allocator Fluency

Every etude assumes the implicit allocator from lecture.

- Each block starts with an eight-byte pre-block header, followed immediately by its payload.
- The header stores the payload size, which is always a multiple of 8. The header's own eight bytes aren't counted.
- The least significant bit of the header is the allocation bit: 1 means in use, 0 means free.

You can rely on these three helpers from lecture whenever you'd like:

```
size_t get_size(size_t *hdr) {
   return *hdr & (~0x7UL);
}
```

```
bool is_free(size_t *hdr) {
   return (*hdr & 1UL) == 0;
}
```

```
void toggle_alloc(size_t *hdr) {
   *hdr ^= 1UL;
}
```

a. **`toggle_alloc`** flips the allocation bit no matter what it was. Implement the two functions below so that each leaves the bit in a known state, whatever it was beforehand, and leaves the size untouched. Your implementations should be at most two lines of code for each.

```
void mark_in_use(size_t *hdr) {


}
```

```
void mark_free(size_t *hdr) {


}
```

b. **`mymalloc`** finds a header but needs to return a payload address. **`myfree`** accepts a payload address but must update a header. Implement both conversions.

```
void *get_payload(size_t *hdr) {


}
```

```
size_t *get_header(void *payload) {


}
```

c. Here are three attempts at **`get_size`**, each of which differs only in the mask's suffix.

```
size_t get_size_one(size_t *hdr) { return *hdr & ~0x7; }
size_t get_size_two(size_t *hdr) { return *hdr & ~0x7U; }
size_t get_size_three(size_t *hdr) { return *hdr & ~0x7UL; }
```

Exactly one of them is broken. Which one, and for which payload sizes does it return the wrong answer?

d. **next_header** is designed to return the address of the header following **hdr** in the implicit list.

```
size_t *next_header(size_t *hdr) {
   return hdr + 1 + get_size(hdr);
}
```

- Suppose **hdr** is 0x1000 and the block's payload size is 16. What address does this version return?
- What address **should** it return?
- Rewrite the return statement so it's correct.

e. Implement **count_free_bytes**, which walks the implicit list and returns the total number of payload bytes sitting in free blocks. Assume **start** is the address of the first header, **end** is the address just past the last byte of the heap, and **next_header** from part d works properly.

```
size_t count_free_bytes(size_t *start, size_t *end) {




}
```

f. A program calls **malloc(1)** one thousand times and never frees anything.

- How many bytes of heap do those calls consume?
- Of the bytes consumed by each call, how many are header, how many are used for padding, and how many did the client actually ask for?

g. Our implicit allocator's **realloc** always calls **malloc** for a new block, **memcpy**s the data over, and **free**s the old block. Consider the following three-line code snippet.

```
char *memory = malloc(20);
memory = realloc(memory, 100);
memory = realloc(memory, 10);
```

- How many bytes should **memcpy** copy during the first **realloc**? The allocator has no idea the client originally asked for 20, since that 20 was bumped up to 24 and the original request was never recorded.
- How many bytes should **memcpy** copy during the second **realloc**?
- Come up with a single expression, in terms of the old block's payload size and the newly requested size, for the number of bytes to copy.

Answers:

a.
```
void mark_in_use(size_t *hdr) { *hdr |= 1UL; }
void mark_free(size_t *hdr) { *hdr &= ~1UL; }
```

b.
```
void *get_payload(size_t *hdr) { return hdr + 1; }
size_t *get_header(void *payload) {
   return (size_t *) payload - 1;
}
```

c. **get_size_two** is broken, and the others work out just fine. **~0x7U** is a 32-bit unsigned value, **0xFFFFFFF8**, and it's zero-extended to 64 bits before the **&**, so the mask wipes out the upper 32 bits of the header. Any payload size of 2^{32} or more comes back incorrectly. **get_size_one** works—almost by accident—because **~0x7** is a signed, four-byte -8, which is sign-extended to **0xFFFFFFFFFFFFFFF8**. **get_size_three** is the version I consider fully correct and transparent about what it's trying to accomplish.

d. There are three sub-questions here:
   - It returns **0x1088**. Adding 1 + 16 = 17 to a **size_t *** advances by 17 * 8 = 136 bytes, or **0x88**.
   - It should be returning **0x1018**: 8 bytes of header plus 16 bytes of payload beyond **0x1000**.
   - The size is in bytes, so it's best to force the arithmetic to be done in bytes, as with **(size_t *)((char *) hdr + sizeof(size_t) + get_size(hdr))**

e. This is some real code, and you'll want to cannibalize it for your own **assign6** submission once it goes out.
```
size_t count_free_bytes(size_t *start, size_t *end) {
   size_t total = 0;
   size_t *hdr = start;
   while (hdr < end) {
      if (is_free(hdr)) total += get_size(hdr);
      hdr = next_header(hdr);
   }
   return total;
}
```

f. There are two sub-questions here:
   - 16,000 bytes.
   - Each call consumes 16 bytes: 8 of header, 7 of padding, and the 1 byte the client asked for.

g. There are three sub-questions here:
   - 24 bytes, since the header only remembers the rounded-up payload size.
   - 10 bytes, because that's all the client is retaining, and it's the argument to the call itself. Copying 16, the new block's payload size, would also be okay, but copying all 104 bytes of the old payload would run well past the end of the new block.
   - **old_size < new_size ? old_size : new_size** is ideal, since you want the smaller of the two sizes.