COSC 301: Operating Systems
Fall 2019
Project 4: Malloc library
Due: 12 November 2019, 11pm

Overview

In this project, you will complete three tasks to implement a memory management library for user applications. Your library will implement functions similar to the `malloc` and `free` functions that you have used in programs so far. (Instead of malloc and free, our functions will be named `xmalloc` and `xfree` to avoid clashing with the built-in function.)

The goals for this project are to get practice in managing the heap, and thus the "free list", and learning how malloc and free work.

As with prior homework projects, please work with a partner.

You may want to refer to the OSTEP chapter on managing memory freespace (http://pages.cs.wisc.edu/~remzi/OSTEP/vm-freespace.pdf) since your implementation will be almost identical to that described in the text.

Getting started

Click on the Github classroom link posted on Moodle to create a git repo for your work, then clone the repo on a department Linux server (`birds.cs.colgate.edu`). In the repo, you'll see that there 8 files. The implementation for `xmalloc` and `xfree` is in `forgetful.c`. This is the only file that you should modify for this project. You'll also notice 5 test files (named `memtest1.c` – `memtest5.c`). The tests contained within these files are described in detail later in this document. When you type "`make`", the memory management library is compiled, as are all 5 test files.

Your tasks

There is quite a bit of code already written in `forgetful.c` (it is described in the next section). You will need to build upon the existing code and complete the following tasks:

1. Implement worst fit block allocation. When an allocation request arrives via `xmalloc`, you will need to find the free block that, when split, will yield the largest left-over space. The code for implementing worst fit should go in the existing `find_free_block` function, starting around line 101 in `forgetful.c`.
2. When a block is chosen for allocation, it may or may not need to be *split*. For example, if

the left-over space from an allocation would be greater than the smallest allocation that can be made, then the block needs to be split and the free list updated. You'll need to implement the block-split within the existing `allocate_block` function, around line 135 in `forgetful.c`.

3. Lastly, you'll need to implement `xfree`, which should return a block to the free list and do any block coalescence, if necessary. I'd strongly suggest separating the two steps of "linking" the block back into the free list, and doing block coalescing.
    - First, link the block back into the free list *without* doing any coalescence. Just get the linked list modified in the correct way. The block should be added in order by memory address.
    - *After* the block has been linked back into the free list, then traverse the free list to do any coalescing.

Note 1: you may not use the built-in malloc and free functions anywhere in your code, for perhaps obvious reasons.

Note 2: do not call `dump_memory_map` more than once! This function has a side-effect that will cause problems if you call it more than once. If you replace the printf(...) calls in dump_memory_map to use fprintf(stderr, ...), you may be able to call dump_memory_map multiple times. Using gdb is another good way to debug and trace your code.

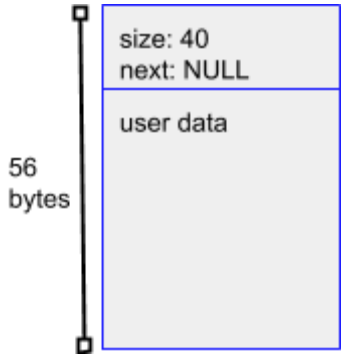
Description of Existing Code

As noted above, there is already quite a bit of code in `forgetful.c` to handle some basic tasks, such as "extending" the heap and doing basic first-fit block allocation. Before you dive in and write any code to complete the above tasks, you will need to familiarize yourself with the existing code in order to better understand how to go about making the changes you need to make.

Similar to how the OSTEP textbook implements its freelist, there is a C struct that defines the "header" at the beginning of each memory block:

```
struct block_header {
      uint64_t size;
      struct block_header *next;
};
```

Notice that the size of a `struct block_header` will be 16 bytes on a modern CPU (the size of a pointer is 64 bits, which is the same size as the `size` field). The size (just as in the OSTEP book) refers to the amount of user-available memory in the block, which does *not* include the header. For example, if a 40 byte block will actually consume 56 bytes in memory:



Note one other operation you will observe in the code: *casts*. Since the linked list that forms the free list is made *within the free blocks themselves*, but the blocks are otherwise "untyped" blocks of memory, there are numerous instances in which a `void *` pointer is cast to (`struct block_header*`) in order to treat the beginning of the block as having a header.

The allocation process is nearly complete (you will need to implement worst fit and block splitting, but the other parts are all there). For the existing allocation code, the following steps take place:

1. When `xmalloc` is called, the requested allocation size is inspected and potentially increased.
    - There are two constraints for an allocation size, enforced in `compute_request_size`. First, the allocation must be at least the size of a block header. Second, the allocation size must be a multiple of a pointer size (which on a 64-bit architecture is 8 bytes).
    - Note that because of the above, the smallest block that will ever be allocated by xmalloc is 32 bytes (including the header).
2. Next, `allocate_block` is called. This function does the following:
    - It searches the free list using `find_free_block` for an existing block to satisfy the request. Currently `find_free_block` implements the *first fit* policy. You will need to modify this function to implement *worst fit*.
    - If `find_free_block` returns `NULL`, `NULL` is immediately returned.
    - If a block on the free list is found, its size is checked to see whether it should be *split*. You will need to implement the block splitting functionality. Note that when you split the block in two, the first part will be eventually returned as the allocated block, while the second part will remain on the free list.
    - After the block is (possibly) split, the free list is updated to *remove* the allocated block.
    - Lastly, a pointer is computed to the start of the *user bytes* within the block (i.e., *after* the header). The pointer to the user data is what gets returned from `allocate_block.`
3. In `xmalloc`, if `allocate_block` returns NULL, then the function `extend_heap` is called to increase the size of the heap. This function is complete and does not need to be modified. It extends the heap by (at least) the requested number of bytes, and adds the

new block to the *end* of the free list. Once `extend_heap` returns, `allocate_block` is called *again* to allocate a block. Unless the computer has run out of memory, this second call to `allocate_block` should succeed since the heap size was just increased.

- You might find it helpful to read `extend_heap` in order to understand how the heap is actually increased (the `sbrk` system call does this; see the man page for details. Fair warning: this is another weird system call). Also helpful to see is that after the heap is extended, the free list is modified by adding the new block (created by extending the heap) to the end.

Lastly, note that there is a function called `dump_memory_map`, which prints a representation of the heap, including both free and allocated blocks. As you are modifying and testing your code, you may find it helpful to insert calls to this function to print out the contents of the heap. (Note that it does not attempt to determine the "break point" between two consecutive allocated blocks. It will simply indicate that a region of the heap is allocated.)

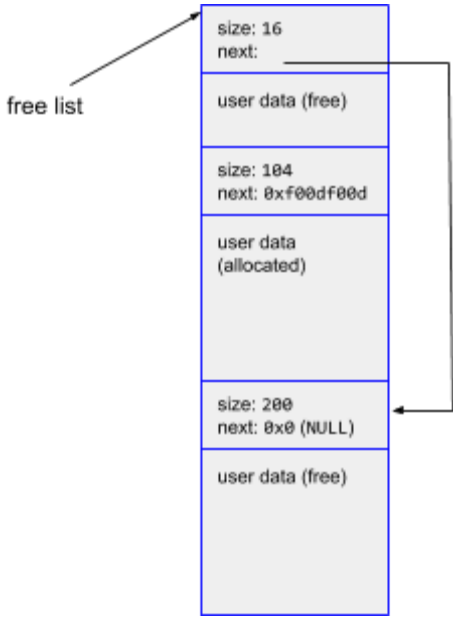
Description of Tests

There are five programs (`memtest1` – `memtest5`) that you can use for testing your code. Each one tests different aspects of how `xmalloc` and `xfree` should work. These tests are described in some detail below.

1. The first test (`memtest1`) should already pass—all it does is `xmalloc` four blocks of memory and do zero `xfrees`. This test is in place to verify that your code does not *regress* as you make modifications to it.

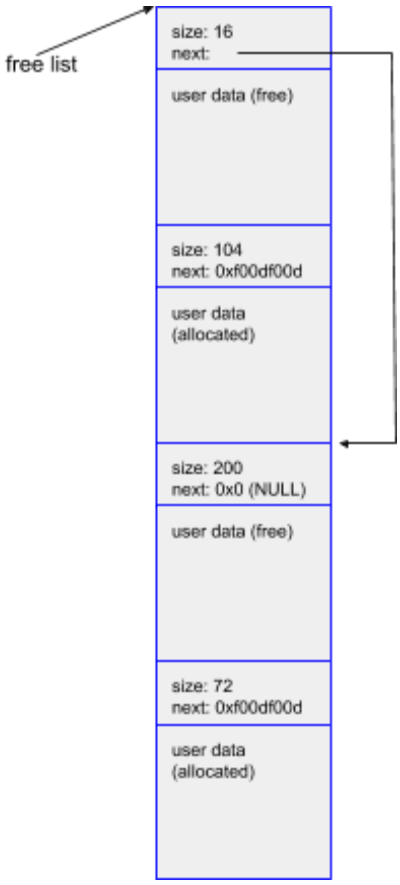
2. The second test (`memtest2`) does the following:
    - Calls `xmalloc(8)`, which results in a block of size 16 (excluding header) being allocated.
    - Calls `xmalloc(100)`, which results in a block of size 104 (excluding header) being allocated.
    - Calls `xmalloc(200)`, which results in a block of size 200 (excluding header) being allocated.
    - At this point, the free list is `NULL`.
    - `xfree` for the *first* call to `xmalloc`.
    - `xfree` for the *third* call to `xmalloc`.

    The heap should look like the following at the end of this test (`0xf00df00d` is our "MAGIC" value). Note that the user data portion is not drawn to scale.



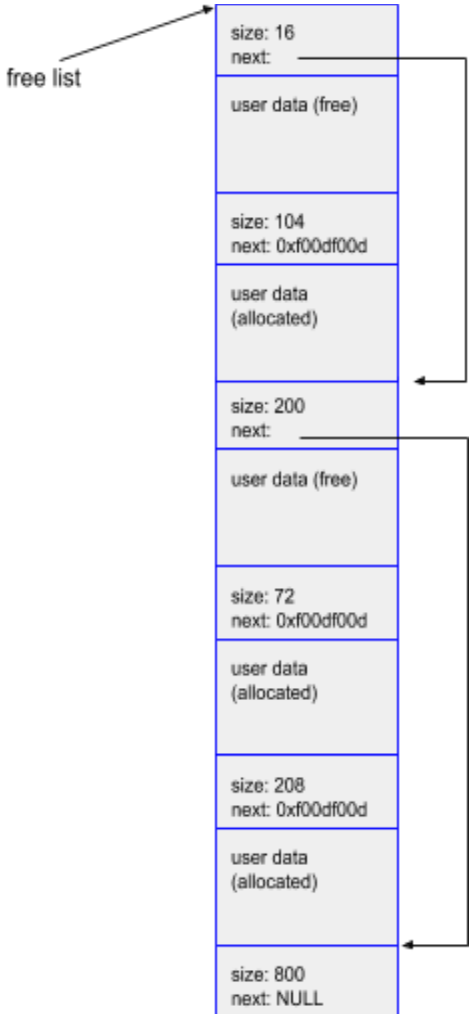
3. The third test (`memtest3`) makes the same three `xmalloc` calls (8, 100, 200), resulting in allocated blocks of size 16, 104, and 200 (excluding headers). What comes next is different:
    a. The second block is freed (it is now the only block on the free list).
    b. The first block is freed. There should still be one block on the free list, with one block allocated. The block on the free list should be of size 136 (16 + 104 + the size of one block header = 16 + 104 + 16 = 136).
    c. The third block is freed. All blocks are freed now, and the free list should consist of a single block of size 352 bytes (excluding the header).

4. The fourth test (`memtest4`) first has four xmalloc calls: `xmalloc(8)`, `xmalloc(100)`, `xmalloc(200)`, and `xmalloc(72)`. At this point the free list is `NULL`. What happens next is the following:
    a. The block allocated by `xmalloc(8)` is freed.
    b. The block allocated by `xmalloc(200)` is freed. At the end of this call to `xfree`, the heap should look like the following:



c. Lastly, the block allocated by `xmalloc(100)` is freed. At the end of this test program, the block allocated by `xmalloc(72)` is still used, so there should be one block on the freelist (of size 352, not including the header), and one block of 72 bytes (not including the header) allocated.

5. You may have noticed that none of the prior tests exercise the worst fit algorithm. The fifth test does that by:
    a. Calling `xmalloc(8)`, `xmalloc(100)`, `xmalloc(200)`, `xmalloc(72)`, and `xmalloc(1024)`.
    b. Freeing the blocks allocated by `xmalloc(8)`, `xmalloc(200)`, and `xmalloc(1024)`.
    c. `xmalloc(202)` is called, which should result in the last block on the free list being split. The contents of the heap should look like the following:



d. `xmalloc(4)` is called, which should again result in the last block on the free list being split.
e. Finally, the block allocated by `xmalloc(100)` is freed.

Submitting your work

Make sure all code is committed and pushed to Github. Also, answer the questions in `README.md`.