TECHNION
Israel Institute
of Technology

Computer
Science
Department

Edited by: Ilya Kaikov

Operating Systems – 234123

Taken from: Homework Exercise 4 – Wet (August 2024)

Notes: malloc_3.cpp is my suggested implementation of the buddy allocator that is described in detail below.

This assignment was written by the course staff, and is provided here for better context of the requirements and functionality of the allocator, as well as its limits in performance and external fragmentation.

1. **`void* smalloc(size_t size):`**
   - Searches for a free block with at least *'size'* bytes or allocates (**`sbrk()`**) one if none are found.
   - Return value:
     i. Success – returns pointer to the first byte in the allocated block (excluding the meta-data of course)
     ii. Failure –
        a. If size is 0 returns NULL.
        b. If '*size'* is more than 10^8, return NULL.
        c. If sbrk fails in allocating the needed space, return NULL.

2. **`void* scalloc(size_t num, size_t size):`**
   - Searches for a free block of at least '*num'* elements, each *'size'* bytes that are all set to 0 or allocates if none are found. In other words, find/allocate *size * num* bytes and set all bytes to 0.
   - Return value:
     i. Success - returns pointer to the first byte in the allocated block.
     ii. Failure –
        a. If size or num is 0 returns NULL.
        b. If '*size * num'* is more than 10^8, return NULL.
        c. If sbrk fails in allocating the needed space, return NULL.

3. **`void sfree(void* p):`**
   - Releases the usage of the block that starts with the pointer *'p'*.
   - If *'p' is* NULL or already released, simply returns.
   - Presume that all pointers 'p' truly points to the beginning of an allocated block.

4. **`void* srealloc(void* oldp, size_t size):`**
   - If '*size'* is smaller than or equal to the current block's size, reuses the same block. Otherwise, finds/allocates *'size'* bytes for a new space, copies content of oldp into the new allocated space and frees the oldp.
   - Return value:
     i. Success –
        a. Returns pointer to the first byte in the (newly) allocated space.
        b. If *'oldp'* is NULL, allocates space for 'size' bytes and returns a pointer to it.
     ii. Failure –
        a. If size is 0 returns NULL.
        b. If '*size'* if more than 10^8, return NULL.
        c. If sbrk fails in allocating the needed space, return NULL.
        d. Do not free *'oldp'* if `srealloc()` fails.

On top of the memory allocation functions that you are defining, you are also required to define the following stats methods.

5. **`size_t _num_free_blocks():`**
   - Returns the number of allocated blocks in the heap that are currently free.

6. **`size_t _num_free_bytes():`**
   - Returns the number of **bytes** in all allocated blocks in the heap that are currently free, excluding the bytes used by the meta-data structs.
7. **`size_t _num_allocated_blocks():`**
   - Returns the overall (**free and used**) number of allocated blocks in the heap.

8. **`size_t _num_allocated_bytes():`**
   - Returns the overall number (**free and used**) of allocated **bytes** in the heap, excluding the bytes used by the meta-data structs.

9. **`size_t _num_meta_data_bytes();`**
   - Returns the overall number of meta-data bytes currently in the heap.

10. **`size_t _size_meta_data():`**
    - Returns the number of bytes of a single meta-data structure in your system.

Important Notes:

1. Note that once **size** field in the metadata is set for a block in this section in the metadata, it's not going to change.
2. You should always search for empty blocks in an **ascending manner**. This means if there are two free (and large enough) pre-allocated blocks at 0x1000 and at 0x2000, you should choose the block that starts at 0x1000. **Hint:** you should **maintain a sorted list** of all the allocations in the system, as described in the proposed solution. You can use large freed blocks for small allocations. This might cause fragmentation but **ignore it for now.**
3. An initial underline in function names means "hidden" or "private" functions in programmer lingo - these are not meant to be called directly by the user. We will use these in our testing, and so should you.
4. Wrong usage of `sfree()` and `srealloc()` (e.g. sending bad pointers) is not your responsibility, it is the library user's problem. Therefore, such action is undefined and there's no need to check for it. In other words, assume that the pointers sent to these functions are legal pointers that point to the first allocated byte, the same ones that are returned by the allocation functions.
5. In this part we will not look at optimizations other than reusing pre-allocated areas. If you come up with optimization ideas, keep them up for the next parts.
6. You should use `std::memmove` for copying data in `srealloc()`.
   a. Self-reading: read about 'std::memmove' and `std::memcopy`, what is the difference? Could you have used 'std::memcopy' instead? Why not?
7. You should use `std::memset` for setting values to 0 in your `scalloc()`.
8. You are **NOT PERMITTED** to use **STL** data structures for this part (e.g. std::vector or std::list). Use only primitive arrays or linked lists that you implemented by yourselves.
9. A "block" in the context above means **both** the meta-data structure and the usable memory next to it.
10. You should not count un-allocated space that's not been added to the heap by `sbrk()`.
11. You are not required to "narrow down" the heap (e.g. use `sbrk()` with a negative value).
12. If your algorithm chooses a large block (e.g. 1000 bytes) for a small allocation (e.g. 10 bytes), you should mark the entire block as used. This means that if the system had "X free bytes" before such allocation, it should have "$X - 1000$" free bytes after the allocation.

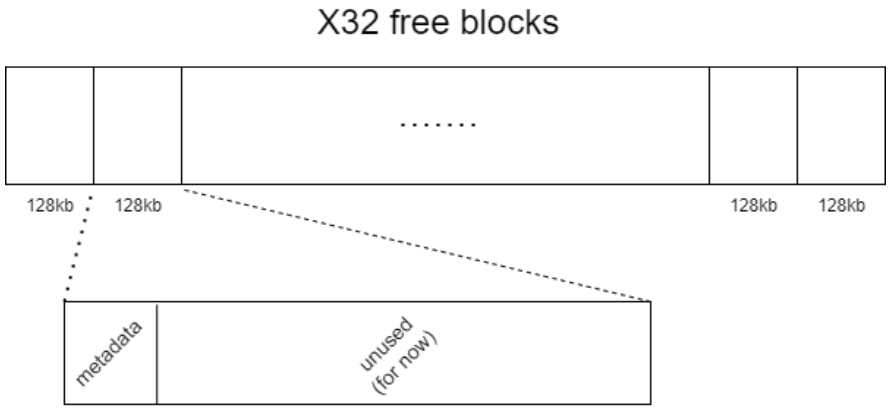
    This should be reflected in your `_num_free_bytes()` function.
13. You should not perform any alignments in this part.

Part 3 – Better Malloc

Our current implementation has a few **fragmentation** issues. Below are some which you might have noticed while working on the previous section (with their solutions). In this section you will work on solutions for some of those issues.

To solve these issues, you will implement a "buddy memory" allocator. This technique divides memory into partitions of sizes that are **always** powers of 2, to try and satisfy a memory request as suitably as possible. The basic concept behind it is as follows - memory is broken up into large blocks where each block is a power of two number of bytes. Upon allocation request, if a block of the desired size is not available, a large block is broken up in half and the two blocks are said to be **buddies** to each other. One half is used for the allocation and the other is free. The blocks are continuously halved as necessary until a block of the desired size is available. When a block is later freed, its buddy is examined and the two are merged if it is also free.

Your buddy allocator should initially increase the program break to obtain the memory region it will use for future allocations. writing your buddy allocator, you should initially have **32 free blocks** of size **128kb** (kb = 1024 bytes), so the heap's layout will look like this:



After having these 32 blocks, you should not use sbrk() again (as opposed to the previous section).

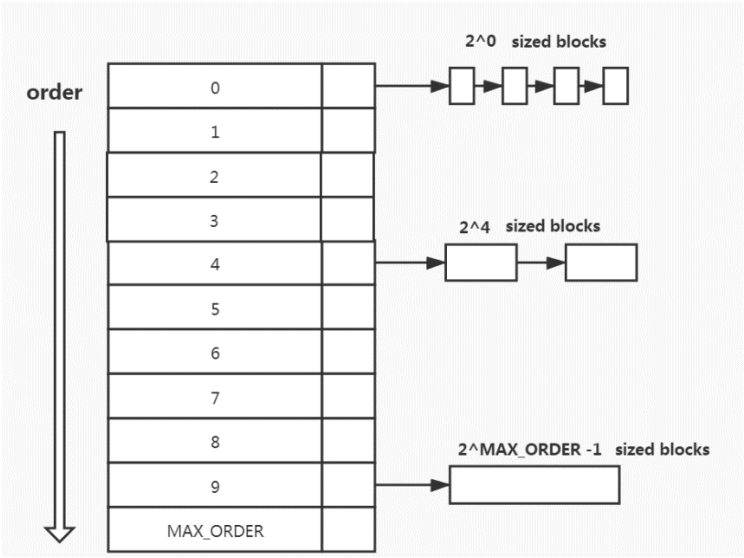
We will use the notation of "orders" to talk about the sizes of our memory blocks. The smallest possible order is 0 for blocks of size 128 bytes, the next order is 1 for blocks of size 256 bytes, and so on. The biggest blocks will be of order MAX_ORDER=10 and of size **128kb.** Sizes of blocks include the size of your metadata structure. For example if your metadata size is 28 bytes, when using blocks of order 0 we can serve user requests of up to 128-28=100 bytes.

Before beginning to implement the buddy allocator, please read challenges 0-2 and the notes at the end of the section and make sure that you understand how the buddy allocator works. You may also search the web to gain intuition about this allocator.

Open a new file, call it **malloc_3.cpp**, copy the content from malloc_2.cpp into it, and in it implement the following changes:

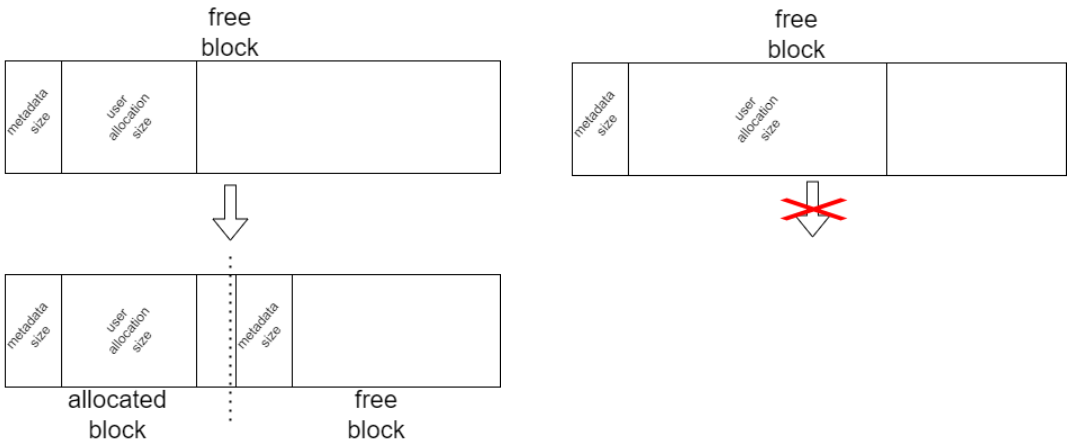
- **Challenge 0** (Memory utilization):
  As mentioned before, searching for an empty block in ascending order will cause internal fragmentation. To mitigate this problem, we shall allocate the smallest block possible that fits the requested memory, that way the internal fragmentation caused by this allocation will be as small as possible.

Solution: change your current implementation, such that allocations will use the 'tightest' fit possible i.e. the **free** memory block with the **minimal size** that can fit the requested allocation. If there are multiple blocks with minimal size, choose the one with minimal memory address.
You should use the following data structure to maintain the **free** blocks:



i.e., an array of lists such that cell i holds a linked list of all **free** memory blocks of order i.
- each linked list should be ordered by the blocks' memory addresses.
- You should use **doubly** linked lists for easy removals.
- You can use the "prev, next" pointers in each blocks' metadata to maintain these lists.
- this data structure needs to be valid after every operation (e.g. malloc(), free(), …).

- **Challenge 1** (Memory utilization):
  If we reuse freed memory sectors with bigger sizes than required, we'll be wasting memory (internal fragmentation).
  Solution: Implement a function that `smalloc()` will use, such that if a pre-allocated block is reused and is ***large enough***, the function will cut the block in **half** to two blocks (buddies) with two separate meta-data structs. One will serve the current allocation, and the other will remain unused for later (marked free and added to the free blocks data structure). This process should be done iteratively until the allocated block is no longer "large enough".
  Definition of "large enough": The allocated block is large enough if it is of order > 0, **and** if after splitting it to two **equal** sized blocks, the requested user allocation is small enough to fit **entirely** inside the first block, so the second block will be free.
  illustration of a large enough block: illustration of a block which isn't large enough:



- **Challenge 2** (Memory utilization):
  Many allocations and de-allocations might cause two **buddy** blocks to be free, but separate.
  Solution: Implement a function that `sfree()` will use, such that if we free a block which has a **free** buddy block, the function will automatically combine both free buddy blocks (the current one and the adjacent one) into one large free block. If after merging we find out that the new free block also has a free buddy block, we should iteratively merge blocks until no two buddy blocks are free. (note that we should **not** merge free blocks of order MAX_ORDER)

- **Challenge 3** (Large allocations):
  Recall from our first discussion that modern dynamic memory managers not only use `sbrk()` but also `mmap()`. This process helps reduce the negative effects of memory fragmentation when large blocks of memory are freed but locked by smaller, more recently allocated blocks lying between them and the end of the allocated space. In this case, had the block been allocated with `sbrk()`, it would have probably remained unused by the system for some time (or at least most of it).
  Solution: Change your current implementation, by looking up how you can use `mmap()` and `munmap()` instead of `sbrk()` for your memory allocation unit. Use this **only** for allocations that require **128kb space or more (128*1024 B).**

Note: You are not requested to "narrow" down the heap anywhere in this section. The only exception for allowing free memory to go back to the system is in challenge 3, when using `munmap()`.

Note: As opposed to the previous section, the 'size' field in the metadata for blocks here changes.

Note: please check that your metadata struct size isn't bigger than 64 bytes, otherwise you will not be able to pass the automatic tests.

Note: you should allocate the initial 32 free blocks of size 128kb the first time malloc() is called.

Recommended trick:

If you align the initial 32 blocks of size 128 kb in memory so that their starting address is a multiple of 32*128kb, you can find the address of a buddy block for any given block by XOR'ing its address with its size. This nice trick works because we only use blocks that have sizes that are powers of 2.

You are allowed to call once sbrk() before allocating the 32 initial blocks to make sure that they are properly aligned, thus making only two calls to sbrk() in this section. If you choose to use this trick, the first call to sbrk() for the purposes of alignment should not affect the statistics functions' results.

Notes about `srealloc()`:

`srealloc()` requires some edge-case treatment now. Use the following guidelines:

1. If `srealloc()` is called on a block and you find that this block and its buddy block are large enough to contain the request, merge and use them. Prioritize as follows:
    a. Try to reuse the current block without any merging.
    b. Check if by Iteratively merging with buddy blocks, we can obtain a large enough block for the allocation (but don't merge yet).

        • stop at the first point when the obtained free block is large enough for the allocation, merge all the blocks and reuse this obtained free block.
    c. find a different block that's large enough to contain the request (don't forget that you need to free the current block, therefore you should, if possible, merge it as described in challenge 1 before proceeding).
2. You can assume that we will not test cases where we will reallocate an mmap() allocated block to be resized to a block (excluding the meta-data) that's less than 128kb.
3. You can assume that we will not test cases where we will reallocate a normally allocated block to be resized to a block (excluding the meta-data) that's more than 128kb.
4. When srealloc() is called on an mmaped block, you are never to re-use previous blocks, meaning that a new block must be allocated (unless old_size==new_size).

Notes about `mmap()`:

1. It is recommended to have another list for `mmap()` allocated blocks, separate from the list of other allocations. (this list isn't required to be sorted)
2. To find whether the block was allocated with `mmap()` or regularly, you can either add a new field to the meta-data, or simply check if the 'size' field is greater than 128kb or not.
3. Remember to add support for your debug functions (function 5-10). Note that functions 5-6 should not consider `munmapp()`'ed areas as free.