Project 2: Memory Allocator

Benjamin Zydney & Yifan Lu
School of Electrical Engineering and Computer Science, Pennsylvania State University
CMPSC 473: Operating Systems Design & Construction
November 7, 2022

Design of the Scheduler + Implementation Choices / Restrictions:

When we began this project, we initially tried to implement the buddy allocator and `my_malloc()` with a list of free holes, as outlined in the slides. However, we found this to be rather difficult. After doing some research and stumbling upon the Wikipedia article on buddy memory allocation, we learned that buddy allocators are often implemented using a binary tree to represent used or unused split memory blocks.

As such, we decided to have the framework of our memory allocator be a binary tree, with each node representing a memory chunk that could be implemented with the buddy allocator. This node included pointers to each of its children in the tree, as well as data on the size of the memory chunk, the status of the memory chunk (`FREE`, `ALLOCATED`, or `SPLIT`), and the offset of the memory chunk from the start of memory. We initialized nodes with the `init_node()` function.

For the buddy allocator, we implemented several helper functions: `split()`, `combine()`, `allocate()`, and `free_helper()`.

The `split()` function was implemented recursively, starting from the root of the binary tree, and performing breadth-first search through the tree, with an additional parameter `desired_size` that gave the desired size of the block we wanted. If space was available with the desired block size, we returned a pointer to the offset of the block; otherwise, we recursively split the tree down the left and the right until we could find a block of desired size. If no block was available, we returned a pointer to `-1`.

The `combine()` function was a fairly straightforward implementation. We simply checked if the node's children were able to be combined, and if they were, we combined them.

The `allocate()` function was implemented with essentially the same code as the `split()` function, but the only difference was that we did not split any blocks; the function simply checks if the block is available, and if not, just recursively checks the left and right subtrees.

Finally, the `free_helper()` function was a depth-first search freeing function that called the `combine()` function to combine newly-emptied blocks.

With these helper functions, buddy allocating and freeing was simple to implement. For allocating, we simply checked that a block of given size wasn't larger than `MEMORY_SIZE`, found the smallest power of 2 greater than the size, tried to allocate the block immediately using `allocate()`, and if that failed, then called `split()` with a `while` loop on

`desired_size` to find the smallest block larger than `alloc_size`. If no space was found, we returned a pointer to `-1`. For freeing, we simply set `pointer_with_offset` to be equal to the actual start of the block (including header), and then called the `free_helper()` function to free the block, returning a pointer to `-1` if we couldn't free the block (i.e. the block didn't exist).

For the slab allocator, we initialized a slab descriptor table as a singly-linked list of slabs, which are a data type with information about slab size, status (`FREE`, `PARTIALLY_ALLOCATED`, or `FULLY_ALLOCATED`), offset, type (size of each chunk in the slab), number of chunks in the slab, an array of size `N_OBJS_PER_SLAB` that keeps track of which chunks in the slab are taken, and a slab identifier that helps with freeing.

Slab allocator and free implementation was fairly simple. For allocating, we first performed the necessary edge checks, and then if there was a chunk available of the necessary size, we just allocated it in the slab descriptor table; if not, we used the buddy allocator to allocate a block for `slab_size`. For freeing, we just iterated through the slabs to find the necessary slab and removed it from the chunks array, and then if all chunks were free, we freed the slab using `buddy_free()`.

Unimplemented Issues / Problems with Implementation:

After testing the program with the sample inputs and with some custom test cases that we created, which included multiple corner cases, we were unable to find any unimplemented issues. However, there were a few problems that we encountered in the process of designing the project, but have been fixed since then.

We found that this project was a good bit easier than the previous one, so our main problem had to deal with implementing the edge cases correctly. For example, one problem we had was finding the smallest power of 2 that was greater than the desired size. Initially, we just took `MEMORY_SIZE` and kept dividing it by 2 until it was less than the desired size, but we forgot to consider the corner cases.

The only other non-trivial issue we had was with `input_11` on the buddy allocator; specifically, why it gave us the wrong output when all of our other inputs worked. To fix this, we manually drew out the allocating binary tree, and found that we were allocating the *leftmost* block larger than the desired size, rather than the *smallest* block larger than the desired size. We were then able to fix this by setting a new parameter `desired_size = alloc_size * 2`, and then multiplying it by 2 each time we couldn't find a block of that size to allocate. This ensured that we found the *smallest* block larger than `alloc_size`.

Things Learned:

Coming into this project, neither of us were super familiar or well-versed with buddy or slab allocators. We had attended lectures and looked over the slides, but our understanding of how exactly they worked was very elementary. As such, we had some difficulty beginning this project, and so we used the buddy allocation Wikipedia page linked above, as well as a GeeksforGeeks article on slab allocation to enhance our background. After reading these articles and having to implement the project, we feel like we have a very strong understanding on both buddy and slab allocation, but are curious if the time complexity of freeing and allocating would be different if we had implemented it with a list of free holes, as opposed to our implementation with a binary tree.

Distribution of Workload:

The workload for this project was divided fairly evenly. Most of the work was done either together in-person or over text, which ensured that only one of us was committing work at a time so that no work would be lost or overridden by a merge. When we ran into errors that we needed to debug, such as the ones mentioned above, we would both talk about the potential causes of the errors and possible solutions. We would then both try to fix the errors, and if either of the proposed fixes worked, we would commit that change.