Assignment 1 Report

Explicit Free List

Block Design

A free block in an explicit free list consists of one header, next block pointer, prev block pointer and one footer. Both header and footer are of type `Meta` which only contains one member `metadata` of type `size_t` . The footer is a boundary tag used for constant time coalescing. The next and prev block pointers point to the logically linked blocks.

A lot of utility functions are implemented for manipulating blocks and pointer arithmetics. This is inspired from Lab 6 and CSAPP.

Fenceposts

Dummy node, aka fencepost, is a common trick for implementing linked-list like data structures. In my implementation, I put a head fencepost at the head of the free list and a tail fencepost at the tail of the free list. Upon initialization, the allocation status of both fenceposts are set to allocated. This makes them never get coalesced and the traversing, insertion and deletion on the free list becomes easier in that you do not have to consider many creepy corner cases.

Constant time coalescing

The contant time coalescing is driven by boundary tags, i.e. footers in the physically adjacent left blocks. When inspecting the left block, we first use pointer arithmetics to compute the footer address of the left block. If the footer says the left block is free, we then extract the size field from it and do another round of pointer arithmetics to get the start address of the left block. Once we have the left block, we can coalesce it with the current block (and the right block if it's also free) in constant time.

Multiple Free List

Interleaving blocks differing in size too much might degrade the performance of the allocator, so I updated my explicit free list to a multiple free list, aka. segregated free list. This upgrade is relatively easy and direct. A global variable `SegFreeList` is allocated upon program start. It's an array of size `N_LISTS` in which elements are of type `FreeList` which is simply a wrapper of all stuff related to one free list, such as `heap_start` , `heap_end` , `head_fencepost` and `tail_fencepost` . Only functions involving the above variables need to be modified and the modification is very direct. I've designed a helper function `which_list` to decide which free list shall serve the given allocation size. The `i-`

`th` free list in the array serves allocation size `8*(i+1)`. If an allocation size goes beyond the size the last free list could serve, it's dispatched to the last free list.

Two cases worth noting:

- If a free block is splitted into a smaller free block which makes its size class changed, it's deleted from the old free list and inserted into a new free list of lower class.
- If a free block is coalesced into a bigger free block which makes its size class changed, it's deleted from the old free list and inserted into a new free list of highr class.

Attempted Optimizations

All optimizations required for a `D` grade are attempted and implemented.

- Use bit manipulations to set or get `size` and `status` fields in the header or footer.
- Reuse the space for storing the next and prev block pointers to store payload.
- Use boundary tags, i,.e. footers, to implement constant time coalescing.
- Use mulitple free lists to serve allocation sizes of difference size classes.

Challenges

- Setting and getting the next and prev block pointer. I compute the start address of the block pointers by adding appropriate offset on the start address of a block. In my first attempt, I simply treat these start addresses as the block pointers and undoubtly the tests failed. After a round of thinking and debugging, I've designed a cumbersome type conversion scheme to dereference the addresses to get the corresponding blocks.
- How to clear and toggle a given bit or a set of bits. It's solved by searching for help in the Internet >_<.

Observations

- The benching program hangs on the Lab 6 implicit free list implementation. Doubt there're some flaws or the implementation is way too slow.
- My multiple free list implementation has worse performance than the single explicit free list implementation. It slows over 5 times.