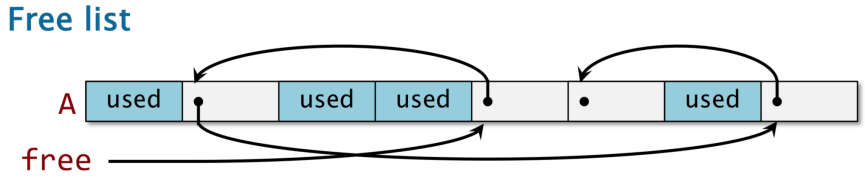
# 6.172 Project 3: Beta Write Up

Sam Ubellacker, Tyler Wasser
subella@mit.edu, twasser@mit.edu

October 2018

## 1 Introduction

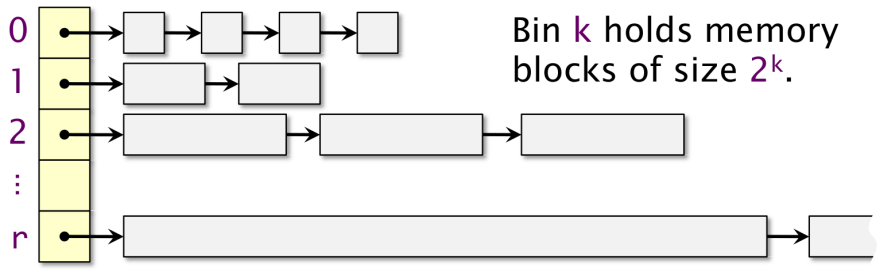
For project 3, we decided to start by creating an implementation that makes use of the binned free list data structure in order to allocate, free, and reallocate memory efficiently. Our free list is a linked list structure that is added to when we free memory. If a block of memory is freed, it is added to the free list, and then when an allocation for memory of the same size is called later, a pointer to this memory is given and the data is put there. Each entry in the free list then points to the next free block in memory.




## 2 Design Overview

### 2.1 Data Structures

Our implementation involves utilizing a binned list structure to minimize fragmentation. Essentially, we have an array of 32 free lists that each hold a size that is a power of 2 (Index 0 holds a size of 1 byte, index 2 2 bytes, index 3 8 bytes). With this structure, we can efficiently allocate memory for various sizes by rounding that size up to the nearest free list that can contain it. For example, if the client is trying to allocate 100 bytes, then we would check if there is a 128 byte free list block available to use.




Important metadata for each free list structure is contained in its header. As of now, we only contain the size of the data held in the free list block and a pointer to the next block in the same free list array. We plan to also hold a pointer to the previous block in the free list and the size of the previous block in memory to aid in coalescing blocks in the future. Our free list structure looks like this:

free_list_t:
- int32_t* size
- free_list_t* next

The bins_list array holds 32 of these free_list_t structs. This value was chosen so that there will be a free list available for the maximum allocation size.

## 2.2 Helper Functions

We have macros for heap alignment. ALIGN() returns the byte aligned size of the size passed in. ALIGN_BINS() returns the bin aligned size. This size is a power of 2 corresponding to which free list the size will be placed in. Passing in 26 returns a size of 32 for example.

get_bin_index takes in a size and returns the corresponding index in the bins_list that the size corresponds to. It does this by computing the log2 of the size rounded up to the nearest power of two. We use a look up table and DeBruijn's constant for fast calculation.

break_up_free_list divides a free list into smaller free lists, continuing until there is a free block available in the goal index. It divides the first block in bins_list[index] in half, storing one of the halves in the bins_list[index-1] free list. The other half is divided further until the goal size is reached. For example, if we are trying to break up a 256 byte free list into a 64 byte free list, first the 256 byte would be split into 2 128 byte blocks. One of those blocks will be stored in the 128 byte free list (bins_list[7]), and the other is divided into two 64 byte blocks. Both the 64 byte blocks are placed into the 64 byte free list.

add_to_free_list simply takes a block and adds it to the free list of the given

index.

### 2.3 Procedure

In the init function, we first set our bins_list to NULL. In my_malloc, we get the byte aligned size of the requested size plus the size of the header needed for storing metadata. We then get which index of the bins_list that this size corresponds to and check if we have a free block of that size available. If we do, then we can just use that block and remove it from the free list. If not, then we iterate over all the larger free lists and attempt to break them down into the smallest size needed to handle the request. We will break down the next smallest free list that is not null. If this also fails, then we have no choice but to extend the heap. We get the bin size needed by using the ALIGN_BINS macro. We then extend the heap by this size and return the appropriate pointer.

In my_free, we first cast the given ptr to a free list type by getting a pointer to the start of our header. We then read the header to determine the size that the block was holding. We add this new free block to the appropriate free list using our helper functions.

In my_realloc, we get the byte aligned size of the both the previous size it was holding and the new requested size. If both of these sizes lie in the same free list, then we do not need to allocate any more space, so we update the size in the header and return. If not, then we check if the block being reallocated is at the end of the heap. In this case, we can simply extend the heap by the difference in our current free list size and new free list size. If neither of these checks are true, then we are forced to allocate new memory using my_malloc.

## 3 Completeness, Performance, Bugs

Our implementation successfully passed the validator that we created, so we have good reason to believe that it is allocating properly. Performance wise, our code has 100% throughput on almost every trace file. Utilization is strong on some traces but weaker on others. Our final perfid is 74.4. There are not any bugs that we are aware of.

## 4 Future Plans

Due to timing constraints, we were unable to finish everything we have envisioned for our beta. We had code written to coalesce smaller blocks into larger blocks in our my_free function, but we were unable to debug this fully yet. The framework needed to do this is already written (although not shown in our beta submission), so with more time to debug we should be able to coalesce blocks as well as split them.