

Memory Allocation: Malloc Lab

The following questions are designed to help you navigate and understand the started code provided to you in your Lab 06 repositories. We have included some code snippets, but you should refer to your `mm.c` file for the full range of helper functions and macros that are available.

Allocating Blocks

The function `static void * search_free_list(size_t req_size)` traverses the explicit free list in order to find a suitable free block to satisfy an allocation request. The starter code is shown below:

```
/* Find a free block of the requested size in the free list. Returns  
   NULL if no free block is large enough. */  
static void * search_free_list(size_t req_size) {  
    struct BlockInfo* free_block;  
  
    free_block = FREE_LIST_HEAD;  
    while (free_block != NULL){  
        if (SIZE(free_block->size_and_tags) >= req_size) {  
            return free_block;  
        } else {  
            free_block = free_block->next;  
        }  
    }  
    return NULL;  
}
```

Of the three allocation policies discussed in class (best fit, next fit, first fit), which allocation policy does the above function implement?

First fit. The function scans through the free block list, always starting at the head of the free block list, and returns the first free block with a size that exceeds the target request size.

Now, rewrite `static void * search_free_list(size_t req_size)` so that it instead implements the best fit algorithm.

```
/**
 * Implements the best fit algorithm. Scan through the entire list,
 * noting the smallest free block that exceeds the target request size.
 */
static void * search_free_list(size_t req_size) {
    struct BlockInfo* free_block = FREE_LIST_HEAD;
    struct BlockInfo* best_so_far = NULL;

    while (free_block != NULL){
        // all candidates must exceed the minimum size
        if (SIZE(free_block->size_and_tags) >= req_size) {
            if (best_so_far == NULL) {
                // we havent see ANY blocks, so this one is the best so far
                best_so_far = free_block;
            } else if(SIZE(free_block->size_and_tags) < SIZE(best_so_far->size_and_tags)) {
                // this block beats the best, so it becomes the best
                best_so_far = free_block;
            }
            free_block = free_block->next;
        }
    }
    return best_so_far;
}
```

Coalescing Blocks

We saw in lecture that coalescing free blocks reduces "false fragmentation" and consequently plays a very important role in our allocator's performance. One challenge when coalescing blocks in an explicit free list allocator is that the order of blocks in the linked list of free blocks is not tied to the physical order of the blocks in memory. Below is the complete code for coalescing.

```
static void coalesce_free_block(struct BlockInfo* old_block) {
    struct BlockInfo *block_cursor;
    struct BlockInfo *new_block;
    struct BlockInfo *free_block;
    // size of old block
    size_t old_size = SIZE(old_block->size_and_tags);
    // running sum to be size of final coalesced block
    size_t new_size = old_size;

    // Coalesce with any preceding free block
    block_cursor = old_block;
    while ((block_cursor->size_and_tags & TAG_PRECEDING_USED)==0) { // (1)
        // While the block preceding this one in memory (not the
        // prev. block in the free list) is free:

        // Get the size of the previous block from its boundary tag.
        size_t size = SIZE(*(size_t*) UNSCALED_POINTER_SUB(block_cursor, WORD_SIZE));
        // Use this size to find the block info for that block.
        free_block = (struct BlockInfo*) UNSCALED_POINTER_SUB(block_cursor, size);
        // Remove that block from free list.
        remove_free_block(free_block); // (2)

        // Count that block's size and update the current block pointer.
        new_size += size;
        block_cursor = free_block;
    }
    new_block = block_cursor;

    // Coalesce with any following free block.
    // Start with the block following this one in memory
    block_cursor = (struct BlockInfo*) UNSCALED_POINTER_ADD(old_block, old_size);
    while ((block_cursor->size_and_tags & TAG_USED)==0) { // (1)
        // While the block is free:

        size_t size = SIZE(block_cursor->size_and_tags);
        // Remove it from the free list.
        remove_free_block(block_cursor); // (2)
        // Count its size and step to the following block.
        new_size += size;
        block_cursor = (struct BlockInfo*) UNSCALED_POINTER_ADD(block_cursor, size);
    }

    // If the block actually grew, remove the old entry from the free
    // list and add the new entry.
    if (new_size != old_size) {
        // Remove the original block from the free list
        remove_free_block(old_block); // (2)

        // Save the new size in the block info and in the boundary tag
        // and tag it to show the preceding block is used (otherwise, it
        // would have become part of this one!).
        new_block->size_and_tags = new_size | TAG_PRECEDING_USED; // (3) (5)
        // The boundary tag of the preceding block is the word immediately
        // preceding block in memory where we left off advancing block_cursor.
        *(size_t*) UNSCALED_POINTER_SUB(block_cursor, WORD_SIZE) = new_size | TAG_PRECEDING_USED; // (5)

        // Put the new block in the free list.
        insert_free_block(new_block); // (4)
    }
    return;
}
```

Please annotate the code above to label any lines of code that perform the following actions:

1. Checks the allocation status of the *memory-adjacent* neighbors of `old_block` .
2. Removes free blocks from the *explicit linked list* of free blocks.
3. Combines *memory-adjacent* free blocks into a larger block.
4. Inserts a free block into the *explicit linked list* of free blocks.
5. Updates the allocation size of a block.
6. Updates the allocation status of a block.

When coalescing blocks, be careful about the function's preconditions!!! Should the block's allocation status be allocated or free? Should the block be in the free list or not? The starter code works *if* you call the function with arguments that satisfy the preconditions.

End-to-end Allocation

In plain English, write the steps that you would take in order to satisfy a request for memory when a user calls `malloc(size)` .

.

.

End-to-end Freeing

In plain English, write the steps that you would take in order to satisfy a request for memory when a user calls `free(ptr)` .

.

.