CSE 351 Winter 2018 Welcome back to section ☺

Section 10 Worksheet: Memory Allocation

To understand the computer, you must become the computer.

For this problem, you will use a heap simulator to answer questions about heap memory allocation in C using `malloc` and `free`. First, access the heap simulator, either by going to the Lab 5 page on the course website, or by following the link:

`https://sarangjo.github.io/cse351-heap/`

Read through the information on the page so you can understand how the simulator works. If you check "simulation mode" it will walk you through each step in the process of allocating a block of memory on the heap. Notice that each header (and footer) contains three fields. They correspond to:

`block size : previous block allocated? : block allocated?`

where `1` means "allocated" and `0` means "unallocated." Remember that block size **includes** the size of the headers and footers. In Lab 5, all of this will be stored within 8 bytes using by reserving the last few bits of the size (see the spec for more information).

Now, answer the following questions:

1. Try allocating a 23 byte block with "simulation mode" enabled.
   (a) How much total space is allocated for the block, including for metadata?
   32 bytes
   (b) What is the header for the newly allocated block?
   `32 : 1 : 1` (total size is 32 bytes, previous block is allocated/unavailable), block is allocated)
   (c) What is the header for the block immediately following the allocated block?
   `96 : 1 : 0` (total size is 96 bytes, previous block is allocated/unavailable, block is unallocated)
   (d) Does the newly allocated block have a footer tag? If not, why doesn't it need one?
   No. If freeing the block to the right of it, we can check that block's `previous block allocated?` field to see if we should coalesce. If the field indicates that the block is allocated, then we don't check it.
   (e) What pointer is returned to the client?
   8 (the first 8 bytes are used by the header)

2. Now, armed with your newfound knowledge about this mysterious memory allocator, figure out some of its properties. You may add and/or remove whichever blocks you want to help answer these questions.
   (a) What is the **alignment** for this memory allocator?
   8. Notice that anytime we allocate a block, it always allocates in multiples of 8. For instance, suppose we ask for 44 bytes. Allocated blocks need a header (which is 8 bytes), so that brings us to $44 + 8 = 52$ bytes. But the allocator sets aside 56 bytes total, and adds internal 4 bytes of internal fragmentation to the block to maintain alignment.
   (b) What is the minimum block size? Why? What does the block size include?
   32 bytes. An unallocated block must store a header (8 bytes), a prev pointer and next pointer (8 bytes each), and a footer (8 bytes).

3. Starting with an empty heap (you can empty the heap by refreshing the page), "execute" the following code:

```
malloc(30)
malloc(40)
malloc(70)
```

   (a) What pointer is returned if we execute another `malloc` now?
   176 (the beginning of the free list + 8 bytes for the header.)

   (b) Which block(s) could you free that would cause fragmentation in the heap?
   Just the block with the pointer 48. Neither of the other blocks have an allocated block on both sides.

   (c) Which block(s) could you free that would cause coalescing to occur?
   Just the block wit the pointer 96. It's the only block bordered by an unallocated block.

   (d) Suppose `free(48)` is run immediately after `malloc(70)`. Draw a diagram of what the free list looks like afterwards.

   ⇔ `48 : 1 : 0` ⇔ `88 : 1 : 0` ⇔

   (e) Immediately after running `free(48)`, what is the maximum sized **payload** that we can allocate (i.e. the argument to `malloc`) such that we are returned a pointer to the address `48`?
   40. We have 48 bytes of free space starting at address 40 (which will return a pointer to 48 when allocated). The header consumes 8 bytes; so we have at most 40 bytes left over that we can use as a client.

4. What are the minimum and maximum arguments to `malloc` such that we must allocate a block of size N bytes? The first one has been filled out for you.

N (bytes)	Minimum Argument	Maximum Argument
32	1	24
48	33	40
72	57	64
88	73	80
128	113	120

5. In Lab 5, we will implement a memory management system that uses an explicit free list. Each block has pointers to the next and previous blocks. This is the block struct we will use:

```
struct BlockInfo {
    // Size of the block (in the high bits) and tags for whether the
    // block and its predecessor in memory are in use.
    size_t sizeAndTags;

    struct BlockInfo* next;
    struct BlockInfo* prev;
};
typedef struct BlockInfo BlockInfo;
```

(a) Implement the following method. Try using bitwise operators to access the tags in sizeAndTags.

```
// Bit masks used to retrieve tags from sizeAndTags.
#define TAG_USED 1
#define TAG_PRECEDING_USED 2
// SIZE(blockInfo->sizeAndTags) extracts the size of a 'sizeAndTags' field.

// Copies the tags (TAG_PRECEDING_USED and TAG_USED) from blockToCopy
// to originalBlock. Leaves the size of originalBlock unchanged.
void copyTags(BlockInfo* originalBlock, BlockInfo* blockToCopy) {
    size_t copyUsed = (blockToCopy->sizeAndTags) & TAG_USED;
    size_t copyPrecedingUsed = (blockToCopy->sizeAndTags)
                               & TAG_PRECEDING_USED;
    originalBlock->sizeAndTags = SIZE(originalBlock->sizeAndTags)
                                 | precedingUsed
                                 | used;
}
```

(b) Now, some practice with doubly-linked lists. Implement the following method.

```
BlockInfo *FREE_LIST_HEAD;

// Removes a block from the free list.
void removeFreeBlock(BlockInfo* freeBlock) {
    BlockInfo *nextFree, *prevFree;

    nextFree = freeBlock->next;
    prevFree = freeBlock->prev;

    // If the next block is not null, patch its prev pointer.
    if (nextFree != NULL) {
        nextFree->prev = prevFree;
    }

    // If we're removing the head of the free list, set the head to be
    // the next block, otherwise patch the previous block's next pointer.
    if (freeBlock == FREE_LIST_HEAD) {
        FREE_LIST_HEAD = nextFree;
    } else {
        prevFree->next = nextFree;
    }
}
```