Memory allocator

TABLE OF CONTENTS

There are many ways to structure a memory allocator. In this part of the homework, you will be implementing a memory allocator using a linked list of memory blocks, as described in the previous section. In this section, we'll describe how allocation, deallocation, and reallocation should work in this scheme. To make your implementation succeed, you will need to modify `mm_alloc.c`.

Allocation

```
void* mm_malloc(size_t size);
```

The user will pass in the requested allocation `size`. Make sure the returned pointer is pointing to the beginning of the allocated space, not your metadata header. One simple algorithm for finding available memory is called **first fit**. When your memory allocator is called to allocate some memory, it iterates through its blocks until it finds a sufficiently large free block of memory.

Here are some implementation details to be aware of.

- If no sufficiently large free block is found, use `sbrk` to create more space on the heap.
- If the first block of memory you find is so large that it can accommodate both the newly allocated block and another block in addition, then the large block is split in two; one block to hold the newly allocated block, the other to be a residual free block.
- If the first block of memory you find is only a bit larger than what you need, but not large enough for a new block (i.e. it's not big enough to hold the metadata of a new block), be aware that you will have some unused space at the end of the newly allocated block.

- Return `NULL` if you cannot allocate the new requested size.
- Return `NULL` if the requested size is 0.
- For grading purposes, please **zero-fill your allocated memory** before returning a pointer to it.

Deallocation

```
void mm_free(void* ptr);
```

When a user is done using their memory, they'll call upon your memory allocator to free their memory, passing in the pointer ptr that they received from `mm_malloc`. Note that deallocating doesn't mean you have to release the memory back to the OS; you just have be able to allocate that block for future use now.

Here are some implementation details to be aware of.

- As a side-effect of splitting blocks in your allocation procedure, you might run into issues of **fragmentation**: when your blocks become too small for large allocation requests, even though you have a sufficiently large section of free memory. To solve this, you must **coalesce** consecutive free blocks upon freeing a block that is adjacent to other free block(s).
- Your deallocation function should do nothing if passed a `NULL` pointer.

Reallocation

```
void* mm_realloc(void* ptr, size_t size);
```

Reallocation should resize the allocated block at `ptr` to `size`. A suggested implementation is to first `mm_malloc` a block of the specified size, `memcpy` the old data to the new block, and then call `mm_free(ptr)` at the very end. Make sure you handle the following edge cases.

- Return `NULL` if you cannot allocate the new requested size. In this case, do not modify the original block.
- `mm_realloc(ptr, 0)` is equivalent to calling `mm_free(ptr)` and returning `NULL`.
- `mm_realloc(NULL, n)` is equivalent to calling `mm_malloc(n)`.
- `mm_realloc(NULL, 0)` is equivalent to calling `mm_malloc(0)`, which should just return `NULL`.
- Make sure you handle the case where size is less than the original size.