CS252 Lab 1: Memory Allocation

# CS252 Lab 1: Memory Allocation

Fall 2016

## Code Review Form

## Introduction -

As you no doubt know by now, a major part of C programming is the management of memory. You have used malloc() and its kin before, but now it is time to delve into how malloc() works. In this lab, you will implement a memory allocator, which allows users to malloc() and free() memory as needed. Your allocator will request bulk chunks of memory from the OS, and manage it efficiently.

## The Skeleton -

Login to data and type

```
cd
mkdir cs252
git clone /homes/cs252/sourcecontrol/work/$USER/lab1-src.git
cd lab1-src
```

**Important: You have to be in data when you run the git clone command.**
**Also, you will have to login to data or any of the lab machines to work on your lab.**

The skeleton code within MyMalloc.c contains a basic allocator that does not do much. It always satisfies *malloc()* calls by requesting memory from the OS (via *sbrk()*), and *free()* is a no-op (memory is always lost). Run ./testall to run the testing suite. All tests should fail, to start.

## The Allocator -

You will implement an allocator that uses a **free list** to manage memory. A free list is simply a linked list of memory blocks that are "free" (that is, not currently allocated). The list will begin as an empty list with a Sentinel (head) node. When a call to *malloc()* is made when the list is empty, you will need to request a chunk of memory from the OS (2MB) and add this large block to the free list. When you exhaust this memory (after satisfying *malloc()* calls), you will request more 2MB blocks as needed.

When a *malloc()* call is processed, you will search this free list for the first block large enough to satisfy the request, remove that block from the list, and return it. You will satisfy *malloc()* requests in a *first fit* manner. To accomplish this, if a block is larger than the request size, you will need to split it into two smaller blocks: one block which satisfies the request, and one which contains extra memory beyond the request size. To make things simpler, requests will only be satisfied in multiples of 8 bytes (round the requested size up to the next 8 byte boundary).

Every block in the free list will have a **header** at the beginning (it will play an important part in the *free()* process, which will be discussed later). The header will contain the size of the object (including the header), a flag for if the block is currently allocated, as well as pointers to the next and previous blocks in the free list. The header will also keep track of the size of the block preceding it in memory.

To allocate a block of memory, the algorithm is as follows:

1. Round up the requested size to the next 8 byte boundary.
2. Add the size of the block's header (i.e. real_size = roundup8(requested size) + sizeof(header)).

3. Traverse the free list from the beginning, and find the first block large enough to satisfy the request.
    a. If the block is large enough to be split (that is, the remainder is at least 8 bytes plus the size of the header), split the block in two. The second block (highest memory) should be removed from the free list and returned to satisfy the request (see the diagram below). Set the *_allocated* flag to true in the header. Update the proceeding block's *leftObjectSize* to the size of the allocated block.
    b. If the block is not large enough to be split, simply remove that block from the list and return it.
    c. If the list is empty, request a new 2MB block, insert the block into the free list, and repeat step 3.



** Split header 1 and return Header 2's memory to the user. This prevents having to remove header 1 from the list and insert header 2 into it, which would be the case if the first block was allocated.*

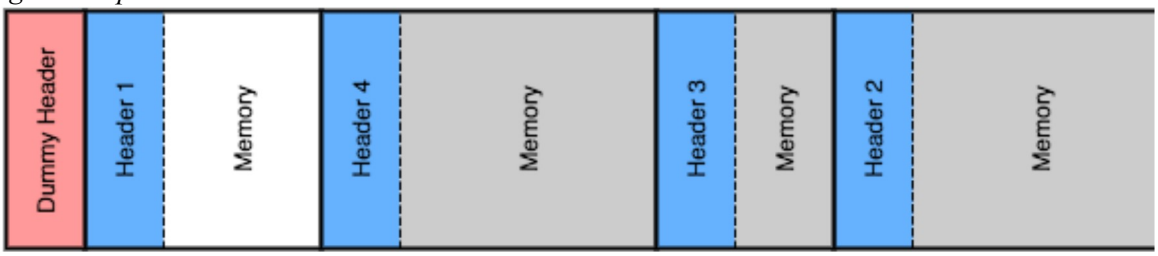
**Note:** It is important for determinism in testing that the free list be kept in a specific order. So all insertions into the free list should be made at the beginning of the list.

## Freeing Memory and Coalescing -

In a simple world, *free()*ing a block of memory would involve just inserting the block back into the free list; however, over time this can create external **fragmentation**, where the list is divided into many small blocks which are incapable of satisfying larger requests. To combat this, many allocators use what is called **coalescing**, where a block to be freed is merged with free memory immediately adjacent to the block. This way, larger blocks can be created to satisfy a broader range of requests in the future efficiently.

The header plays an integral part to this process. The basic layout of all blocks in memory should look something like this:

**Remember: when splitting blocks of memory, you will retrieve the bottom chunk, which will result in a numbering like above in cases of multiple mallocs in a row, with the first header the furthest left, then the rest in ascending order up to two. **




When freeing an object, you will check the header of the left neighbor (the block immediately before the object to be freed in memory) and the of the right neighbor (the block immediately after in memory) to see if those blocks are also free. If so, then you will merge one or the other (or both) with the block to be freed, and update the pointers accordingly. If both the left neighbor and the right neighbor are free, you are required to coalesce the left, the freed

block and the right block into a single block. The algorithm to free a block is as follows:

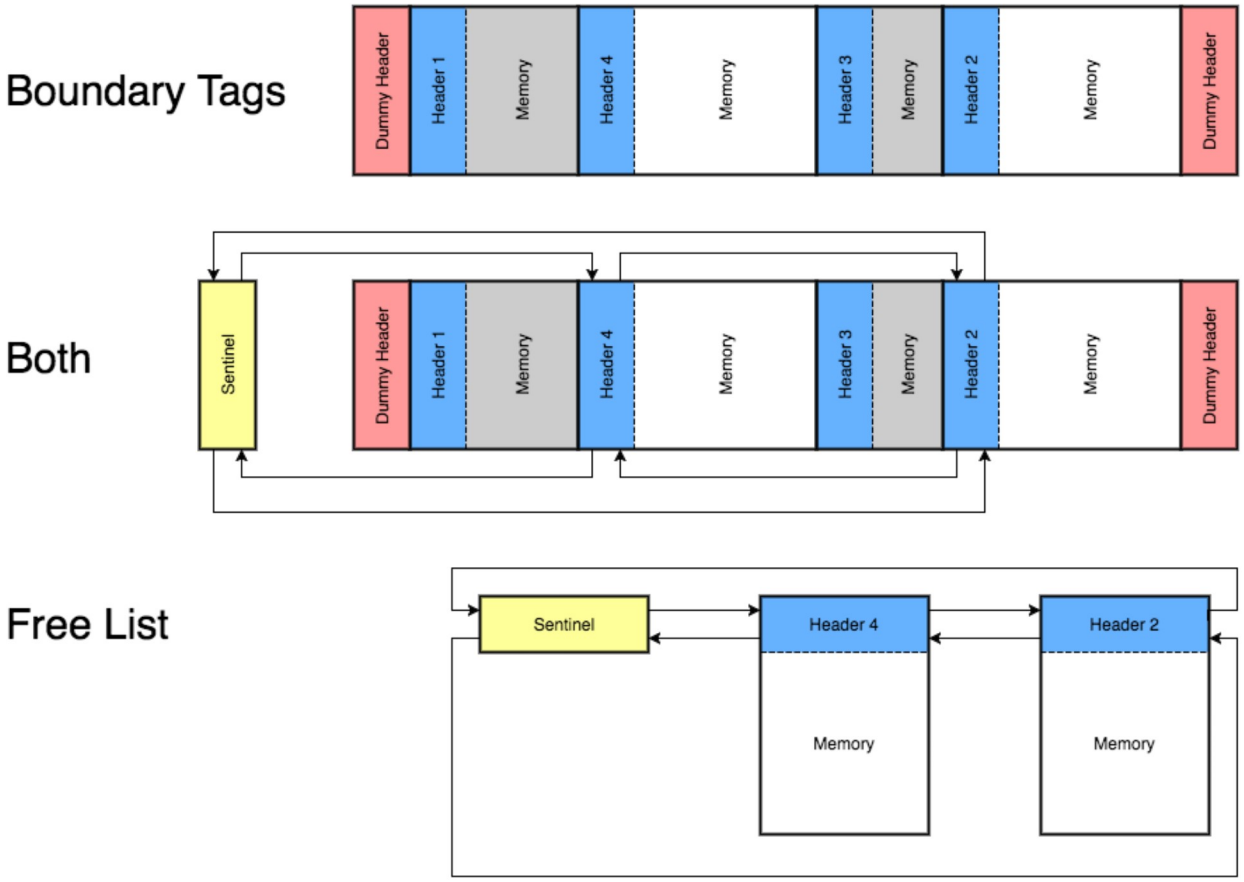
1. Check the if the header of one or both of the neighboring blocks are free. If they are free then coalesce the block being freed into the unallocated blocks.
2. If neither the left nor right neighbors are free, simply mark the block as free and insert it into the free list without any coalescing at the head of the free list.

Remember to insert all not coalesced blocks at the head of the free list.

## Fence Posts -

There is a corner case regarding coalescing that must be dealt with. If a block is either at the very beginning or the very end of the heap, accessing a header beyond the boundary of a block for the purposes of coalescing may result in a crash (from accessing memory you do not own). To combat this, every time you request a 2MB block of memory from the OS via *sbrk(),* you will add additional "dummy headers" to the beginning and end of the block. These should have the *_allocated* flag set permanently to 1, so that it warns that memory beyond these boundaries should not be coalesced. These are in addition to the meaningful header you attach to the block itself.

## Data Structures -




** Here is an example of how the two data structures look on their own or together. It is important to note that there are fields in the header for the boundary tags and other fields for the free list. **

The fields that belong to the boundary tags data structure are the *leftObjectSize,* and *allocated* fields. The fields that belong to the free list are the *listNext* and *listPrev* pointers. Both structures share the *objectSize* field. It is important to note that the free list fields are only maintained while the object is free and are no longer valid once the object has been allocated. However the boundary tags fields are maintained regardless of the allocation state of the object, and relate to

actual memory within the block. For example, in the diagram above, the *leftObjectSize* field for Header 2 will contain the size of the block attached to Header 3, but the *listPrev* field will point to Header 4, because block 3 is allocated, and therefore not in the *Free List*.

The purpose of the *Boundary Tags* is to enable coalescing so you can tell the allocation state of the objects around you.

The purpose of the *Free List* is to manage freed memory so a block of memory, which can satisfy the user's request, may be found.

## Thread Safety -

A memory allocator is a structure that is often used by many threads at once. As a result, the critical sections of your allocator need to be thread safe. Make sure this is the case using the locking mechanism of your choice (for example, pthread_mutex locks).

## Testing -

Implement your allocator in MyMalloc.c. Simply type "make" to build everything, including the tests. Run the ./testall script to run the testing suite. You can see the individual tests in test1.c, test2.c, etc. Be sure to write some tests of your own!

## Turning in your Project

You don't need to turn in your project. Your project is automatically pushed to the git repository every time you type "make" so your latest sources will be there.

The deadline for lab1 is 11:59pm Monday September 5th, 2016.