CS 3410 Preliminary Design Document

Project 6: Malloc

Matthew Fusco(mjf328)   Maxwell Loetscher(mjl258)

May 8, 2020

1 Introduction

Project 6 deals with implementing our own version of malloc, or dynamic memory allocation. The purpose of this document is to help organize and communicate our thoughts and ideas.

References: We consulted the course lecture notes, textbook (Computer Organization and Design RISC-V edition), and other class resources for several parts of our design.

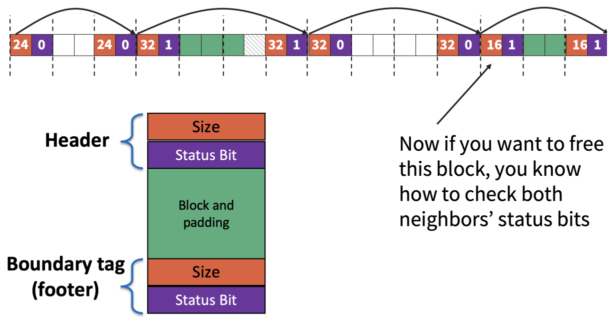
2 General Plans for Implementing Malloc

We plan to use an implicit list setup to implement malloc, using the "best fit" method to find a free block. (Search the list to find the free block with the least number of bits left over.)

2.1 Headers

We plan to store information about block size and allocation status in sections at both the start (header) and end (footer) of the block of memory we allocate in the heap. The block size will encompass the amount of space it takes to store the block, including the space for the header and footer and any necessary padding to make our address 8-byte aligned. (Block size = size(header) + size(block) + size(footer).) The allocation status will be stored in the header (and footer) directly to the right of the block size, with a 1 corresponding to a memory block that is currently allocated and a 0 corresponding to a block that is free. By having both the header and footer store information about block size and allocation status, we will be able to traverse the list forwards and backwards easily and it will be easier to coalesce free blocks. (Will talk more about freeing blocks in the next section.) The diagram below (from the course powerpoint for getting started on P6) isn't exactly how we plan to implement our list, but still shows the general logic behind the design.

Figure 1: Diagram of how we plan to use headers/footers from P6 "getting started" slides

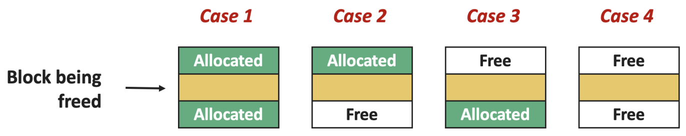


We plan to implement our header/footer by creating structs to store information about things such as heap size, block size, whether or not the block is in use, etc. We will update this header as we allocate (and free) memory in our heap, being sure it still accurately represents the state of things. (e.g. Switching the "in_use" field to be '0' or 'false', for a block of memory that was recently freed.)

2.2 Freeing Blocks

As was explained in the P6 videos, freeing blocks requires us to consider four representative situations, which are shown in the diagram below.

Figure 2: Diagram of the four coalescing cases from P6 "getting started" slides



Ultimately, what's most important with freeing blocks beyond updating the header/footer's allocation status is to ensure that we are properly combining consecutive free blocks. This is the logic behind having footers in addition to headers in that it allows us to traverse the list in both directions, giving us information about the block sizes of all adjacent segments of memory. This means we can more effectively combine adjacent and free blocks and prevent fragmentation.

2.3 Resize

Our general idea for resize is to copy/store the contents of the desired block (the one being resized) and after storing all of this block's data, to then free this block from memory. After successfully freeing the block, we then plan to update the block size to be whatever the new size value is. Finally, with this updated block size, we will then simply allocate the block to memory as we normally would.

Overall, our plan to implement resize is to (1) copy/store the information on the old block to a new block, (2) free the old block, (3) update the new block's size to be the desired size, and (4) allocate the new block to memory.

3 Implementation Details

To implement what we explained above, we plan to create a structs for the head of the heap as well as for each block (in a similar way as in the 'heaplame.c' file, with some slight differences). Every time an operation is performed that changes any of the blocks in the heap, we will update our headers (and footers) accordingly.

For example, in order to allocate a new block, we will first look at the head pointer, and see if the very first block is already allocated. If not, we also must check if that block has enough space for the data we are attempting to pass in. If these two conditions are not met, we will traverse down the heap to each header, and determine whether the conditions are right for a memory allocation there. If a large enough free block is found, we will allocate the whole block, adjusting the necessary headers/footers. As of right now we are leaning towards implementing "best fit."