Marshall Lowe (mhl5178)
Christopher Kurcz (cjk6056)
CMPSC 473
11/7/2022

Project 2 Design Report

Overall Design:

In order to implement the buddy system, we designed a binary tree data structure to organize our data about where our holes and blocks of memory are. Every node in the tree will always either be a leaf or have two child nodes. A leaf will always either be a hole in the memory, or allocated memory for the user (or a slab of memory chunks). Starting from the root node, which contains the entire size of the memory, each subsequent level would then manage half of the size of memory in the level above it (stopping at the minimum memory chunk size). Leaves are also organized left to right by their stored starting addresses, so the leftmost leaf's stored starting address will be the start of the memory given during initialization.

In order to implement the slab allocation system, we designed a slab descriptor table (SDT) data structure and a slab descriptor table entry data structure to maintain the data records for specific "types" of slabs. Type represents the size of each object in a given slab. As well as the type, the SDT entry structures maintains data about the size, total objects, objects used, next entry, and a linked list of data structures to represent the internal workings of a slab such as a slab's starting address and a bitmap representing the open chunks in a slab. This table in combination with the buddy tree is used to represent the state of our memory throughout the runtime of the program.

my_setup() Design:

The purpose of our my_setup() function is to store the given policy in a global variable and to initialize the primary data structures that we use throughout the program. The data structures that we initialized in this function are the buddy tree and the slab descriptor table.

my_malloc() Design:

The first thing that my_malloc() does is to check whether the policy is the buddy allocator or the slab allocator. It then will follow the steps necessary depending on the given policy.

Buddy Allocator: For the buddy allocator, first we determine the size of the chunk of memory that will be required to allocate memory for the user (which would be the size requested + the amount of header bytes all rounded up the the nearest power of two). Next, it makes a new leaf node in the tree containing the required chunk size. It does this by first finding the smallest leftmost hole that is greater than or equal to required size, which we call the "placement node". However, if the placement node is greater than the required size, it will split the node by creating two new child nodes, both of which will be a hole of half of its size. We recursively split the left child node until it reaches the required size. Then, all we do is store the size of the memory in the header and return the starting address that the newly allocated node stores + the header bytes.

Slab Allocator:The slab allocator utilizes a slab descriptor table (SDT) to keep track of slabs of memory chunks such that all the chunks in a single slab share a common size. We refer to this common size as the “type” of the slab. The size of the chunk, and therefore the type as we will be referring to it as this going forward, is a sum of the requested size and the size of a header. When a memory allocation request comes in, the first step is to check the SDT to see if there is already an entry for that type. If there is an entry, then we check all of the slabs that are related to that entry for the first open memory location by using an array of 0’s and 1’s representing holes and allocated memory regions respectively in the slab. If there is an open memory location (a 0), then the type of the chunk is stored in the header of the memory location and we return the address of start of the usable memory for the user. In the case where there is no open memory locations in any of the slabs that are already allocated for the given entry, a new slab is allocated using the buddy allocator with the size equal to (HEADER_SIZE + type * N_OBJS_PER_SLAB) rounded to the next highest power of two. This new slab is then added to the linked list of slabs corresponding to the entry. Since this is a fresh slab, the first chunk of memory in the slab can be used to satisfy the request; therefore, the type is stored in the header and the address of the start of the usable memory for the user is returned. In the case of an allocation of a type that is not yet in our SDT, a new entry is made and inserted into the SDT, and the same procedure as above is followed for allocating a slab for this new entry and returning the users starting memory address.

my_free() Design:

The first thing that my_free() does is to check whether the policy is the buddy allocator or the slab allocator. It then will follow the steps necessary depending on the given policy.

Buddy Allocator: For the buddy allocator, first we find the node in the tree that stores the starting address of the memory the user is trying to free. We do this simply by traversing through the tree until we find a match. Once we have the node, we simply change that leaf node from one containing memory to a hole lead node. Then, we recursively tell the parent of that new hole leaf node to check if it can merge its two children. If both of its children are holes, then it will remove both of its children from the tree and set itself as a hole leaf node, and repeat the process by telling its parent to check if it can merge.

Slab Allocator: In order to release the given memory region according to the slab allocator, we must figure out which slab the requested memory to be freed lies in as well as which chunk in the slab is the one to be freed. To do so, the first thing to be done is to retrieve the size of the memory that is requesting to be freed. To do this, we used pointer arithmetic to retrieve the size of the memory chunk that was stored in its header. The next step is to utilize the slab descriptor table (SDT) to figure out which slab the memory chunk lies in, as well as which index it corresponds with in the slab’s bitmap so that we can now mark it as a hole. To do so, we do a lookup in our table using the size that we retrieved from the header as the type. Following the acquisition of which entry the free corresponds to, we now traverse that entries slab linked list checking whether the memory address to be freed lies in range of the slabs memory addresses. Once the correct slab is found, we iterate through our slabs bitmap to find the index that directly corresponds with the address to be freed, and then we flip the 1 to a 0 to

represent it now as a hole. After this completes, we perform a check to see if the slab is now empty, and if so, we delete it from that entry's list of slabs and use the buddy tree functionality to update the memory representation to be correct. Following this, we do another check to see if there are no longer any corresponding slabs for that entry, and if there are none, we remove the entry from our table.

Distribution of Work:

Chris created most of the buddy system binary tree data structure, along with the buddy system parts of my_malloc and my_free. Marshall created most of the slab descriptor table along with the slab allocator parts of my_malloc and my_free. However, both of us together made the design decisions for the entire project, and we also both helped each other fix bugs and make minor changes to the code each of us wrote.