Tejas Marimuthu
Ayman Awal

CMPSC 473 P2 Report

Buddy

For Buddy we implemented an array of arraylists. We created a struct called element which holds data for size, count and an instance of another struct called chunk. The struct chunk contains a variable to hold the starting address as well as an instance of itself (chunk). This is done to create the linked list.

Also, the 0th index of the array is where the size of memory, 8MB, has been assigned.

my_malloc()

When my_malloc is called, if malloc_type is 0 (indicating it is buddy), we add 8 bytes to the requested size which accounts for the size of the header. Since the minimum size of the chunk given to us is 512 bytes, we compare the requested size to it. If it is less than 512, we round it up to 512, otherwise we use the pow function from the math library to ceil it up to the next power of 2.

We created a **findIndex(int size)** function which takes that rounded up size as an argument and traverses through the array and looks for the size. If the linked list at that size's index is not empty, meaning memory has been allocated and a chunk exists at that level, we will just return the index. Else, we will traverse backwards until we find the first non empty linked list. We will call **splitChunk(int size, int index)** function and return the index.

This function recursively keeps on splitting the chunks into halves, deletes the chunk that just got split using **deleteChunk(int index)**, adds chunks on to the next index using the **addChunk(int index, long address)**. We call addChunk() function twice since we are splitting one chunk and creating two chunks out of it. Our splitChunk() will stop running when we have reached our desired index with the given size.

Going back to the interface.c file, once we have received the index value, as long as there is memory available in the list (indicated by the value being not -1), we call the function **getAddress(int index)** to get the starting address of the head of the list at that particular index and store it in a long variable. We add the mem to the starting address and call deleteChunk() to remove it from our free (holes) list. Now we have successfully allocated the proper hole.

my_free()

When my_free is called, if malloc_type is 0 (indicating it is buddy), we typecast the ptr argument into a long variable to ease our calculations. We will negate the pointer by 8 bytes which corresponds to the header size. Using the line **size = ((int*)ptr)[0]** we get the corresponding size the ptr points to. We add 8 bytes to it which is the header size,

Since the minimum size of the chunk given to us is 512 bytes, we compare the requested size to it. If it is less than 512, we round it up to 512, otherwise we use the pow function from the math library to ceil it up to the next power of 2. We will use **findFreeIndex(int size)** which iterates through our array and checks if the size at each index is equal to the requested size and returns the index. We will call addChunk(int index, long address) to add a chunk to the corresponding index.

We have a **getCount(int index)** function which returns the number of chunks we have at each index of the array. If we have more than one chunk we have to combine them using the **combineChunks(int index)** function. In order for this function to combine chunks, it checks whether the chunks are buddies or not in the first place. They do so by dividing the starting addresses of each chunk by the size of memory at that particular index. After dividing the values, we get a series of even and odd numbers. We made it so that the left chunk has an even number and the right chunk has an odd number associated with. With the left chunk having an even number and the right one an odd number, we know for a fact these two are buddy chunks. We will delete those two chunks by using **deleteSpecific(int index, long addr).** We are going to call this function twice since we are deleting two chunks. We keep doing this recursively on the previous index as well by calling **combineChunks(index - 1**).

We reset the memory by calling getCount(0). If the returned value is 1, we delete the chunk at index 0 using deleteChunk(0) and reinitialize the array using **init_array(MEMORY_SIZE)**. We then add a chunk at index 0 and address 0 using addChunk(0,0).

Slab

For slab we implemented a slab descriptor table in the form of a linked list with each element being a struct called Slab, containing data such as the type (size of request), used bits, integer bitMap array of size 64, starting address and next pointer which is an instance of a slab.

my_malloc()

We take the requested size, add 8 bytes to it and multiply it by 64, which is the number of slab objects. We created a **checkExistence(int type)** function which checks whether or not the size of the request has already been allocated in the slab descriptor table or not by iterating through

the list. If it exists it returns 0, else 1. If the function returns 0, we must call buddy malloc to allocate the memory, which returns a pointer.

If the starting address is empty we return NULL, indicating there is no space available to allocate. Otherwise we will call **addSlab(int type, void *address)** which dynamically allocates memory for a new slab and creates the bitMap array in it containing all 0's except for the first bit being 1. One thing to mention, in this case when we addSlab() we add 8 bytes to the address to account for the slab header. After that we return the pointer to the address. Also we have a sortSlab() function which automatically gets called when we addSlab(). This sorts the linked list in order of increasing type. However when we have slabs of the same type, in that case we sort them based on their starting addresses.

my_free()

In the slab my_free function, we call **deleteBit(void *address)**. In this function we typecast the argument into a long variable called addr for more convenience. For each slab that we iterate through in the linked list, typecast the address of that slab into the bitAddr variable. If we find the addr is the same as bitAddr we know we should change this bit from 1 to 0; this is our way of deleting bits. We will decrement the used variable for the slab and return the address, which will break out of the function.

If there remains no more 1's in the bitMap for the slab, we call **deleteSlab(int type, void *address)** function to delete it since the slab has been freed up. Also deleteBit() returns NULL if there is no need to free the slab.

If deleteBit() returns a specified address, we will call **buddy my_free** and pass the returned address from deleteBit() while subtracting 8 from it to account for the header size.