Aly Ghallab

CMPSC 473 P2 Report

I worked on this Project alone hence all the work was done by me. I started by implementing structs in the header files, which served as the foundational data structures for this project. A general overview of the data structure is that I used a primary linked list, and each node in that linked list is the head of its own linked list. One can think of it as an array of linked lists, with each index representing a power of 2 up until 2^{23} = MAX_MEM hence there are 24 lists in the array. Each of those lists can represent a free list that keeps track of the free nodes when using a buddy or a cache which is a collection of slabs of a specific size when using the slab allocator. I also created a struct for the allocator to neatly initialize the global variables and keep track of other parameters passed in from main.c.

Now to interface.c , my_setup() is where I initialized all my global variable and structures that were to be used throughout the program's life. It is also where I create the array of lists and assign each index a power of 2. My_malloc allocates *size* bytes of memory from the mem_size chunk of memory that was assigned to us and which we were given a pointer to the start of that contiguous chunk of memory that was passed through main.c via the setup function. The function returns a pointer with an 8-byte Header offset factored in, which means this is the location where the next memory allocation can occur.

When using buddy as the allocation type, I have a helper function, malloc_buddy that allocates a portion of memory of a specific size passed in through the parameter into a buddy node then removes the node from the free_list (as it is no longer free). Malloc_buddy uses a first-

fit allocation policy, hence it picks the lowest memory address that fits the allocation. Slab allocation also uses malloc_buddy to allocate new caches (list of slabs) when there is not one available.

When slab allocation is used, my_malloc Iterate through caches until the end of the list of caches, or it finds an available cache with slabs of the requested size. If those conditions are not met, it will create a new cache with the desired slab size using malloc_buddy, as previously mentioned. This is where most of the challenges came from when it came to this project, as it requires substantial pointer arithmetic and an understanding of a complex data structure whilst accounting for other properties such as the Header size (8 bytes) and slab utilization which was how many objects are in that given slab.

my_free also presented similar challenges to my_malloc due to the pointer arithmetic but it was an easier feat. When using buddy allocation, my free calls a helper function I wrote that frees buddy nodes called free_buddy. Free_buddy was one of the hardest helper functions to implement as I needed to implement the conditions required for the coalescence of two free buddy nodes into a larger free segment of memory. Free_buddy takes a pointer to a buddy node and deallocates it from memory while coalescing with other free nodes if possible.

When my_free() is called when using slab allocation, it iterates through caches until it finds one with an appropriate slab size, then it iterates through that cache until it finds the slab referenced by the pointer that was passed in as an argument. It then checks if the slab is already deallocated, so it does not commit a double free. It empties the slab and adjusts the bitmaps accordingly before calling a helper function that frees slabs from memory, free_slab(). Overall, the project was challenging yet insightful, and the levels of abstraction for this assignment made

this assignment easier from a labor perspective. Still, it took time to wrap my head around the possible implementation while referencing the concepts we learned in class.