David Rohweder
473 Project 2
11/7/2022

OS Memory Allocation Project 2

When designing the memory allocation program, the first aspect of the program was to build the buddy allocator. This is because the buddy allocator was the foundation for the whole memory allocation scheme and was a basis for allocating the slab allocation scheme as well. Both the buddy and slab allocator mechanisms were abstracted away from the interface by using a switch statement on allocation type in my_memory. The memory request function returned the address that was returned either for the slab memory request or buddy request functions. After the address was returned the header size and the start of memory were added and returned.

While at first, I considered the binary tree data structure for the buddy allocation scheme and using DFS on it to find holes, I found it more practical to maintain a linked list of free holes. I stored the lists of holes according to the power and size of the 2^{power}. At the beginning of the execution when initializing the data structures, I found the highest power given the memory size by taking the $\log_2 total\_memory$ and found the lowest by taking the $\log_2 \min\ \_chunk\_size$. Then, I malloced an array of powers from 0 to highestPower – lowestPower. Each of the indices had a power that organized the buddy holes and the size of the hole for addressing buddy purposes. Each hole also maintained the start address, end address, information about its split buddy, a state of occupied or filled, and a pointer to the next empty hole of the same power.

When implementing the buddy algorithm, I did sort of a divide and conquer approach where I searched in increasing order of powers for holes and once a free hole was found at a power I broke it down and kept splitting the holes down the powers until the smallest, containable hole was created at the lowest possible power. When the splitting occurred, it rendered the hole at the current power occupied then added two holes from the current hole start and end address at the next power level down. If for instance at power 2 there was a hole from 0 to 10 then at power 1 two holes from 0 to 5 and 5 to 10 would be created and marked as empty. When implementing the free buddy algorithm, I implemented the coalescing approach of first finding the hole in the powers list. This was done by iterating over every power and every free hole in that power until the free address was equal to the holes start address. Once a hole was found it was marked as empty. After being marked as empty the coalescing algorithm kicked in and the freed holes buddy was checked if it was also free. If the buddy was also free then the two child nodes were removed from the current power and the parent hole was changed from occupied to empty again. This process kept iterating until the holes buddy was occupied in which case the function would return.

The slab data structure was a doubly linked list that comprised the slab descriptor table. Firstly, there was a list structure containing both a head and tail pointer for O(1) insertions and contained the offset for each element. The list pointed to a slab-type structure which was used to organize the same types into its own linked list. Each of these slab-type structures held the organizing type, size of the table (N_OBJS_PER_SLAB * (type + offset)), the head and tail pointer of the slabs within that slab type, and a pointer to the next slab type. Each slab type contained slabs of that type. This means that if one slab of a type was filled up then another buddy request would be submitted for another slab of that type and then the first slab would point to the new one. Each of these inner slabs contained a buddy address pointer, an array of entries marking elements as occupied or empty, and the number of used entries in the slab.

For the slab algorithm, a slab size request would be submitted, and the first step was to make sure that the slab type was in the slab type list. If the slab type was not in the list, then it was added, then the algorithm would check if there were any slabs within the type. If there were no slabs within a slab type, then a new slab would be created of that type. This was done by first making a buddy request of the size of the table, then adding the slab with the returned buddy address to the list. From here once there is a valid slab table with empty entries then each memory request would set sequentially each entry to occupied. Upon a free request, each slab type would be iterated over and each slab within that type to see if the freed address is within the bounds of the slab table. If the target free address is within the bounds of the slab, then an offset would be calculated to find the correct target address to be set to empty. Each time a free request occurs the number of occupied entries is decremented and if the number of occupied entries is zero then the slab would be removed from the list of types. If once that slab of type is removed and there are no other slabs of that type left, then the whole slab type would be removed from the slab types of the outer linked list.

All the test cases were implemented for both the buddy allocation and the slab allocation systems. I learned even more about linked lists and pointers in this assignment compared to the first assignment. Also, this work was done all by me as this was a solo project. If given more time for the project I would have probably tried the binary tree and DFS implementation for the project for more DSA experience. I tried to keep the time complexity as fast as possible with little storage as possible.