Computer Science 473 Project 2 Report

Tejas Desale tad5569

Nicholas Wanner njw9

Overview:

We decided the best way that we could implement the stack was to use a global binary tree. The highest root node represented the full stack of the mem_size that was passed into the setup(). To then finely control where memory would be stored whenever it was needed the root would be called with the helper function we made called split(). Which would denote the node as having been split and would give it two leaf children. Each leaf would have half the memory size of its parent node. Each node would be given a status to keep track of it, it could be FREE which meant it had no children and could be split or have memory allocated in it, it could be SPLIT which meant that it had children, or it could be OCCUPIED which denoted that it had memory allocated in that location. Whenever a node was freed it would have its node status changed back to FREE, and it would perform a check in the helper function combine() to see if it's both children of a parent were free. If both children nodes were free it would remove them from its parent and set the parents status from SPLIT to FREE.

The reason we opted to use the binary tree implementation was because we found it much easier to visualize the process of mallocing and freeing for buddy and slab. It was easier for us to draw a quick picture to figure out the necessary logic to work with a tree rather than the data structure the TA's had mentioned in their zoom call help session.

For the cases of slab malloc, we created a structure called slab. This had all the attributes that were described in class such as a type of memory that it would hold, a total size, and an array which contained only 1s, and 0s to denote which spots in the slab were occupied or not with 0 denoting an empty spot. The Slab descriptor table used to track all of the slabs was a global linked list with each slab being a node in the list. We decided to do this since we already had the code for a linked list which could be easily adjusted for this purpose.

Buddy Malloc:

In buddy(), the function for malloc in the case of a buddy input, we would first do a check to see if the given size was larger than the stack memory size. If it was, we could easily return a -1 void pointer for an error. Otherwise, we would do a for loop to find the smallest power of 2 that the given size could be allocated into. We also had a condition so that it would not go below a minimum of 1KB as was specified in the instructions. This was necessary since all the nodes in our binary tree hold powers of 2 for their size. We then called a helper function allocation_search() which would do a depth first search in the binary tree without changing it to see if there existed a free node that had exactly the memory size needed, if it was found it would return a pointer to that node. If there was it would be allocated into that node. If a perfect fit node could not be found we would call the second helper function allocation_split() which would also do a depth first search but would look for the lowest address in the stack that could be split. Once it found that node it would split it recursively until it had a node which would hold the size and return a pointer to it. After allocation_search() and allocation_split() were both called we would do a quick calculation to get the pointer that would need to be returned by adding 4 and the global start of the memory and cast it to a void pointer before returning it.

Buddy Free:

For freeing in the case of buddy we would first reverse calculate the pointer to the node by subtracting 4 and the global start of memory to find the pointer that we would need for our stack. Once we had the pointer we would then use a helper function dfs_free() which would do a recursive depth first search for a node which had a matching pointer. Once a node was found it would set that node's status to FREE. It would then call combine() on the parent to merge the children back into the parent if both children were free.

Slab Malloc:

In our function for malloc in the case of slab. We would first check if the size of a hypothetical slab would be too big to fit into the stack. If it was we would return a -1 pointer. Otherwise we ran a check to see if a new slab of that memory size would be able to fit into the stack, if not a -1 pointer would be returned. Then if the slab descriptor table was empty we would create a new slab, mark the first element in the stack pointer as occupied, and call buddy() to allocate that slab into the stack. If the slab descriptor was not empty then we would iterate through it to find if it existed in the table. If it did, we would find the first free element in that slab's stack pointer and update that to be occupied, increment the number of objects, and calculate and return a pointer. If it was not in the slab descriptor table we would create a new slab, mark the first element in the stack pointer as occupied, and malloc it with buddy() and then return the given address.

Slab Free

For out slab free function, we would first calculate the pointer by subtracting the global start of memory. Then we would iterate through the slab descriptor table and find the slab which contained the given address. We would then calculate which element in the slab pointer would need to be updated and change that element to a 0 and decrement the number of objects. We would also do a check if the slab was now empty. If it was we would remove it from the slab descriptor table and free the memory space using buddy_free().

Challenges:

We found this project to be easier to do than the scheduler project previously as we didn't have to struggle with mutex for this project. There was uncertainty about if we should be implementing the data structure the TA's mentioned in their Zoom help session or if we should implement a binary tree. As for the work load it was typically divided evenly. Most of the work was done together over call, with both of us working on it simultaneously. If we ran into a difficult error to diagnose or identify, we would each work independently work on debugging using gdb. Although the debugging was done individually, both of us were on the call still and informed the other person of their suspicions and then work accordingly. We continued this process until one person found a solution.