Team members: Shakeel Ahmad

Vaynee Sungeelee

- **List of the features implemented:**

1. The fast pools
2. The standard pools
3. Displaying the state of the heap

- **Tests that pass successfully:**

The program passes the automatic tests alloc1 to alloc8.

- **Limitations:**

The display function only gives information about the blocks allocated or free but do not give information about the number of allocated/free bytes.

- **Description of main design choices:**

Fast Pool

For the fast pools, we consider the pointers passed to the mem*_free_fast_pool* function to be the header pointers instead of the data pointers because the header pointers are returned in the call to *mem_alloc_fast_pool* .

Standard Pools

If splitting a block after allocation does not satisfy the minimum size for a free block (size of header + size of footer+ size of node previous + size of node next), the block is not split, resulting in an allocated region larger than the requested size.

Display function

The *print_mem_state* function has been implemented in such a way as to display the allocated or free blocks in each pool. '[.]' Has been used to represent a free block and a [X] to represent an allocated block.

For example a standard pool with 3 allocated blocks is represented as follows:

[X][X][X][.]

Fast pools: To display each block's state , we increment the address by fixed block sizes of the pools (64 for pool 0 , 256 for pool 1 and 1024 for pool2). Since the addresses are not necessarily sorted in the free list, we search each block's address in the free list and if it is found we display '[.]'. Otherwise '[X]' is displayed.

Standard pools: To display each block's state, we increment the address by the number of bytes allocated to each block (given by function *get_block_size*()). Since the free list is sorted for the standard pools, we only check if the block at each address is free and '[.]'. Otherwise '[X]' is displayed.

- **Explain why a LIFO policy for the free block allocation is interesting in the context of fast pools.**

LIFO implies that free nodes will be inserted or deleted from the head of the free list. In the case of fast pools, this is simple and fast because the whole list does not need to be parsed each time a node is inserted to or deleted from the list. The downside caused by internal fragmentation is less important here in the case of small blocks than in the case of large blocks because the lost space is smaller. Furthermore, the fast pools do not suffer from external fragmentation because the free blocks are not split after allocation.

- **Describe the main principles of the algorithm you use to insert a freed block back into the free list, and to apply coalescing if need be. You should try to make best use of the additional metadata included in blocks (footer, double linked list)**

Standard pool

In the function *mem_free_standard_pool*, when a block is about to be freed, we check whether coalescing can be done. We use the 'prev' and 'next' nodes of the double linked list to add or delete nodes from the free list. Inserting a node can be done at the head, between two other nodes or at the tail by connecting the relevant nodes together. To delete a node, we simply disconnect it from its previous node if it is the tail, from its next node if it is the head and from adjacent nodes if it is between two nodes.

For example:

To add a node *nodeToInsert* at the head , the next node *nNext* is connected to the node to be inserted.

nodeToInsert->next = nNext;

nNext->prev = nodeToInsert;

nodeToInsert->prev = NULL;

To delete a node *nodeToDel* at the head:

nodeToDel->next->prev = NULL;

There are 3 possible scenarios for coalescing:

1. Coalescing with a previous block
2. Coalescing with a next block
3. Coalescing with both previous and next blocks

Case 1. The previous block can be found by moving up to the previous block's footer (current block - size of the footer). We then check if the previous block is free (the highest bit of that block's footer is off). If the block is free and coalescing is possible, we calculate the new space (previous block size + previous block's footer + current block's header + allocated size) and free it. A pointer pointing to the address of the previous block is created and a new free node is inserted.

Case 2. For coalescing with a next free block, we check that that there exists a block whose address is larger than the current block's address and that it is free. If this is the case, we calculate the new space (current block's footer size + allocated size + next header's size + next block's size) and free it. The pointer stays at address of the current block and a new node is added in the free list.

Case 3. If a previous block is free, we immediately check if a free block is present just below the current block. In this case, we free the whole space and create a pointer at the address of the previous block. The new free node is then created in the free list.

If no free blocks are found above or below the current block, then, no coalescing is done and a new node with the size of the previously allocated region is created and inserted in the free list.

- **A feedback section including any suggestions you would have to improve the next version of this lab.**

This lab was useful as it helped us understand many concepts in memory allocation as well as improve our pointer skills in c.

Improvements to this lab would be to implement the remaining features which we did not have time to address in this version of the lab.