Jessica Ackerman, Steven Barrios
README.pdf
pa3

Description:
Our program was designed to implement and enhance the processes of Malloc and Free. We began by creating a static array that would hold 5000 bytes to represent the amount of memory available for allocation. In order to have access to 5000 bytes, we made the array of type char since every char is 1 bytes.

We were able to use macro definitions to make any malloc and free calls go to our implementation of dynamic memory allocation:

```
    #define malloc( x ) mymalloc( x, __FILE__, __LINE__ )
    #define free( x ) myfree( x, __FILE__, __LINE__ )
```

When our program receives this data, x is the amount of bytes the user requests, __FILE__ is the file name from where the user requests, and __LINE__ is the line number.

In order to keep track of the memory being allocated in our array, we implemented a doubly linked list of nodes to hold any information that would be useful towards data management. This information includes: isFree, prev, next, and size. Due to these nodes also being inserted into our array, the total amount of data being allocated for the user can never be equal to 5000 bytes since there must be at least one node inserted and the size of a node is 40 bytes.

Our two main functions are mymalloc and myfree. Each one resembles the standard malloc and free functions respectively. Mymalloc first checks to see if the head of the DLL is NULL; if it is then we initialize the head to the appropriate values. After the head is initialized we can properly check if our array can properly allocate the requested amount of memory. This process is a simple list traversal. At each node we are checking for several criteria which include: is current node free, is current node size less than size requested, is current node's next not NULL and current node's size bigger than size requested, and is current node's size > size requested + size of a node. All of these checks are placed inside a while loop and will be performed while the current node is not NULL. If the program exits the while loop then we know that there was not enough space in the list to complete the request.

The next function is myfree which takes a pointer as its argument. Ideally, myfree takes a pointer that mymalloc has allocated just like the standard free. However, myfree can also take pointers that were not allocated by myfree but instead of crashing like the standard free, ours will return an error message describing the incident. The first couple of lines in myfree check to see if the array is empty and if the pointer is NULL. If any of these two conditions is met, the program will print an error message and exit. If this does not occur, we will take the address of the pointer and subtract the size of the node, which is 40, which will give us the assumed address of the node. Now this is a simple list traversal once again. If the address of our assumed node is less than current node then we know we have passed the pointer and it is an invalid pointer. If the address of current node is equal to address of the assumed node then we have found a valid pointer to free. We set all of the bytes that were allocated to 0 using memset. Here we decide to optimize freeing space by checking if either the previous node and the next node are free. If one or both are, then we merge the nodes together by a simple node deletion allowing for one bigger space to be available rather than several smaller spaces. If current node address is not equal to assumed node then we simply continue to the next node. If the current node is NULL and we exit the while loop, we know that the address is outside of our array and it was never allocated using our malloc.

Conclusion:

Our mymalloc and myfree functions run in at most O(N) time if the last available is the last node or the pointer to be freed is also the last node.