MP2 Report

Garrett Haynes

Ryan Walters

10/11/15

Introduction:

In this machine problem, we design a memory allocator that has the functions my_malloc() and my_free(). This memory allocator initializes an initial block size "M", and has a basic block size "b" using malloc(). The goal of this allocator is to keep the free memory grouped together in the largest blocks possible. Using a buddy system, when my_malloc() is called the free memory breaks into smaller blocks until the smallest block size that has enough memory needed is made and given to the user. When blocks are free, their buddies are checked and merged together recursively to keep the free memory in the largest blocks possible. The point of the machine problem is to learn a new memory allocation technique.

Procedure:

We began by designing our node structures to have a header and a memory space. The header consists of a pointer to the next available free block of equivalent size, and an integer of the size in bytes of the block. Also in order to get rid of the free char (used to determine whether the block is used or not) we made our size Int consist of size + 1 if in use, and just the size if not. Next, we created an array of node pointers that points to the beginning of the list of free nodes for each size available (this array was named headers). Therefore the headers[0] points to the beginning of the list of free nodes that are of the basic block size. And headers[log2(M/b)] points to the beginning of the list of free nodes that are of the largest size possible.

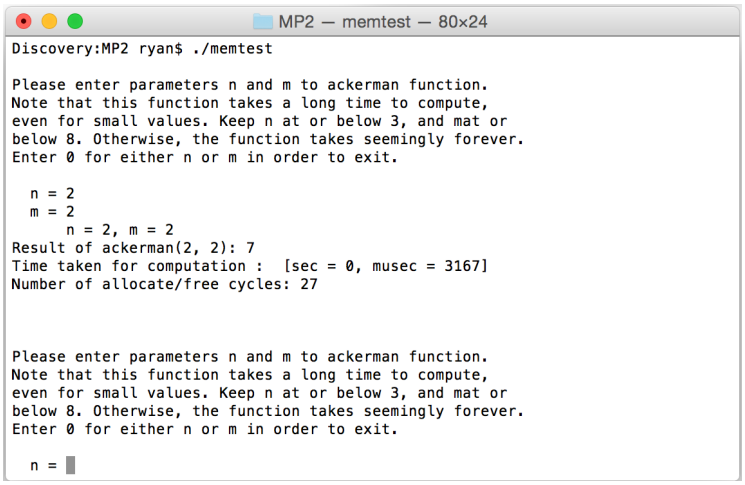
Next we began to design the my_malloc function. This required a recursive function we called choose_index(unsigned int _length) where the _length is the size of memory that the user wants to allocate. What this function does is look through the headers array to find the first block of free memory that is large enough to fit the requested _length. When a free block is found, if it is of block size "b" it returns the first index of the array, otherwise it checks the previous index's block size to see if it is large enough and a split is necessary. If a split is necessary the block is split, and the headers are adjusted accordingly, then the function is recursively called. If a split is not necessary, then the current index is returned. If no space is found, the index -1 is returned. Now that the memory blocks are adjusted accordingly, and the proper index "x" is found, we simply return the address of the block at headers[x] plus the header size. We set the block size to the size plus one to mark that it is in use, then we adjust the headers array appropriately.

We then designed the my_free function. What this function does is clears the memory of the block that is being freed, and adjusts the size of the block to size-1, then it calls the combine(Addr _a) function. This function checks the buddy node by XORing the address of the block with its size to get it's address. If the buddy node is free, the two nodes are combined, then the function is recursively called until the buddy node is in use, or the largest block size is created. The entire time, the headers array is updated accordingly. The last convenience function we created was a print_list function that prints the list of free nodes for each index of the array.

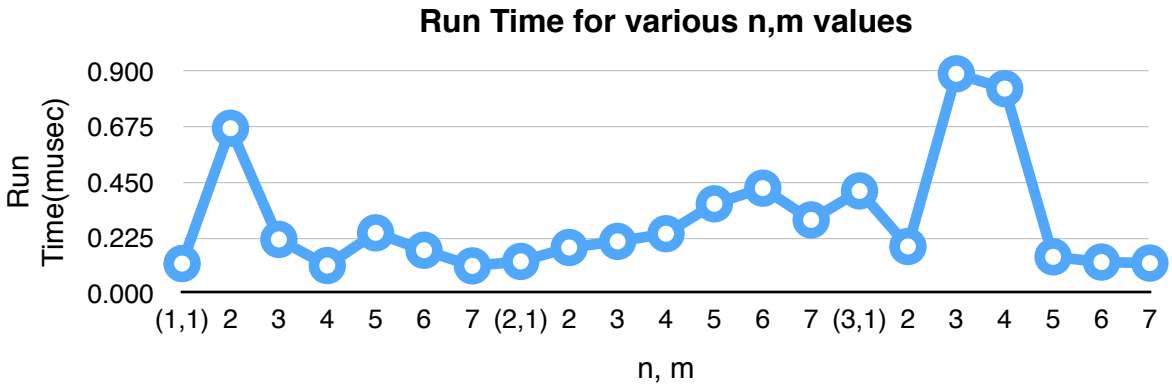
Work separation:

Team member Garrett Haynes created the Init function as well as the my_malloc function. While team member Ryan Walters created the my_free function as well as combining everything to work accordingly with the Ackerman function. The report was worked on between both members.

Result:



In this Terminal window, we can see the result of running the application with the input values $n = 2$ and $m = 2$. The first line contains the instruction to run the application. This also takes arguments in the form of '-b [basic block size] -s [memory to allocate]'. These use the *getopt* library introduced in the last Machine Problem. After the input lines, the result of the ackerman is shown along with the time it took to process the result (including the number of allocate/free cycles). The user is prompted to continue if desired, or enter a 0 for any value to stop the program.



Ackerman Output Values

N	M	Value Returned	Run Time *(sec.musec)*
1	1	3	0.119
1	2	4	0.668
1	3	5	0.217
1	4	6	0.1110
1	5	7	0.2437
1	6	8	0.1731
1	7	9	0.1097
2	1	5	0.1278
2	2	7	0.1843
2	3	9	0.2090
2	4	11	0.2401
2	5	13	0.3615
2	6	15	0.4250
2	7	17	0.2961
3	1	13	0.4137
3	2	29	0.1884
3	3	61	0.8889
3	4	125	0.8287
3	5	253	0.14634
3	6	509	0.12517
3	7	1021	0.12099

Analysis:

Upon testing, we noticed the default options do not provide enough memory for functions right higher *n* and *m* values than *1,5* and above(up to *n=2*) and then *2,2* and above. We also noticed that doubling the memory size and running *2,5* again gave us a roughly doubled run time. This was the same for other values of *n,m* as well. One poor implementation of our code is in the my_malloc. We made the function check to find the first available free node that is large enough, then split if it could also fit in the previous index, it then recursively is called. This is not as efficient as possible, we could have also mathematically figured out how small the block needed to be, then when we found the first block we could have in a loop split the blocks to that size. This would be more efficient do to deleting the need recursively check the free blocks.

Conclusion:

NO MACHINE PROBLEM QUESTIONS