Machine Problem One:

High Performance Linked List

Daniel Frazee and Edgardo Angel

Due: Feb. 15, 2015

Introduction

Machine Problem One: Part One is essentially the implementation of a Singly Linked List from scratch, in C language. This part consists of six major functions that basically allocate memory for the list, create the nodes, and insert or delete nodes from the list. Lastly, the list is printed with the print_list function. Each node consists of a pointer to the next node, a key for searching, an integer that represents the length of the value being stored, and finally the value.

Part Two implemented a tier system with *t* many linked lists making up the tiers. An array of pointers to the list heads of each linked list was essentially the way of keeping track of all the tiers. From this, keys are randomly generated and nodes are placed into their specified linked list. This implementation also came from scratch in C language.

Procedures

Part One: Much of the modularity of this part was given to us from the source files. The linked list consisted of linked_list .c and .h with a main.c for testing. The core functions `init`, `destroy`, `insert`, `delete`, `lookup`, and `print_list` made up the functionality of the list. `init` was the initialization of the list which took in arguments for the total size of the list as well as the size of each node. From these arguments, `init` used malloc() to allocate the proper amount of memory for the linked list. Next, `insert` takes arguments for the key, the value length, and the actual value, then the list is traversed until the end is reached and then the new node with the proper values is inserted at the end. From here, `delete` and `lookup` are pretty similar. `lookup` takes a key as an argument and traverses the list, checking each node until the key is found. If the key is not found, a message will display letting the user know. `delete` does essentially the same thing as lookup, but when the key is found, the node is deleted by skipping the pointer that pointed to that node, to the node that was one past it. `print_list` does exactly what it says; it simply prints the list by traversing through and printing the keys and the values. Finally, `destroy` frees up the memory taken by the list by feeding the free() function the pointer to the head of the list.

Part two: This section builds from what was developed in part one. When creating the memory pool, malloc was used to create the size of each tiered memory pool, and store the header pointer to that memory pool to an array of pointers. What was different in some of the functions was knowing what tier was being worked on. Since most of all the functions depended on the key, a helper function was developed. `getTier` is a function that helps other functions to know where the key is located at. `getTier` function works by receiving as the key, and then searching for the header pointer to its appropriate tier. Since the instructions to the assignment said that the tiered memory pool should contain a total of 2^{31} -1 values, once the memory pool size was received by the user, the value of 2^{31}-1 was divided by the amount of tiers used. Then the key was searched in between the values of the tiers and if found, would return the tier where the key was found. Knowing the tier, along with pointer arithmetics, using the array of pointers, the correct pointer was given back.

Results

Our results were positive in the sense the linked list worked as expected. The traversing of the lists through memory worked properly with little hiccups and storing the proper values in the right places was successful. Also, printing of the lists and proper output for each function is neat and easy to understand. To fully test the tier system, it was filled up with random generated numbers, to make sure the tier allocation system worked. Then adding, looking up and deleting existing and non existing elements were tested to make sure all functionality worked. Printing the tiered memory pool was always tested when completing the other functions

Conclusion

This machine problem taught us about allocating and moving through computer memory. We had to move bytes at a time through memory and keep pointers to the parts we found important. We found this troublesome at times due the wastage of memory when an item was deleted. Since a new node could not be inserted into a deleted node's space. We probably could have made that type of insertion possible which would saved a significant amount of memory, relative to our list size. But, we would have to be careful because only nodes of certain size would be able to fit; just because there is open memory does not necessarily mean there is room for an insertion. Also, the execution of creating the linked list was difficult at times, and finding the right pointer for the right situation was not easy. The maximum value for our pointers being just 8 bytes is 2^64. When implementing the tiered linked list, we tested it with signed values, that it from 0 to 2^{31}-1. By dividing this into *i* tiers, the general expression for the range of numbers that go into the *i*th tier is simply, (2^{31}-1)/i. By the end, we successfully implemented the tier system with Singly Linked Lists and gained a lot of knowledge about computer memory and accessing it with the C language.