Machine Problem 1: A High Performance Linked List

Introduction:

In a traditional implementation of a linked list, memory is allocated for each newly inserted item. When an item on this list is deleted, the memory allocated for it is freed and given back to the operating system. Consequently, each allocation/de-allocation request for insertions or deletions respectively involves interacting with the operating system's memory manager. System calls have a large overhead associated with them. The processor must stop the execution of the user process, switch to executing system code, spend time performing the requested system function, and as such, the calls to the memory manager hinder the performance of the linked list. In this assignment, we will explore a solution to this problem which will produce a high performance and efficient implementation of a linked list.

In this assignment, the program you create will serve as the memory manager instead of relying on the operating system to do the dirty work. Your program should obtain/reserve a fixed amount of memory from the operating system's memory manager during initialization (this is done by calling either malloc or new). After initially acquiring memory from the system, your program should use this memory to manage the linked list throughout its execution. Consequently, extraneous and expensive calls to the system's memory manager will no longer be necessary.

Each element of the linked list occupies a specific amount of space, known as the basic block size and denoted by the variable b. The memory size as a whole is determined by the parameter m. As such, there can be at most m/b elements in the list. Any insertion requests that would attempt to insert more than m/b elements into the list should be immediately rejected.

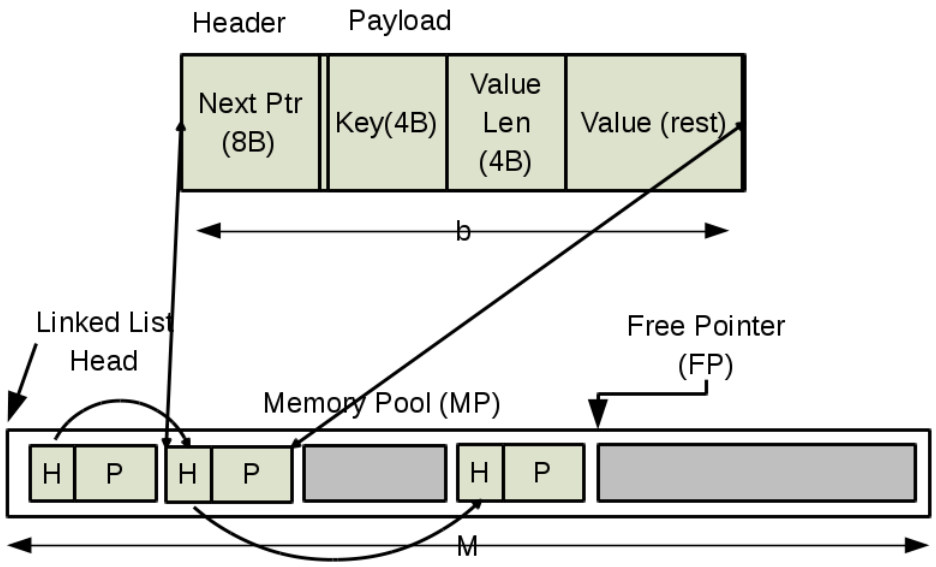
The list should be managed through a Head Pointer (HP) and a Free Pointer (FP). The former points to the head of the list, and the latter points to where the next insertion should happen.

Each linked list item can be separated into two sections: a header and a payload. The header contains necessary information for maintaining the list (i.e. the next pointer, the previous pointer (for doubly linked lists), other metadata, etc...). The payload portion consists of a key-value pair. The key is a 4 byte integer, and the value is of variable length, but has a maximum size that is determined by the header and key size. In addition, each header contains a value length that denotes the size of the current data item. Since the value is variable length, there's no way, as is, to know where the end is. Some languages solve this problem by ending strings with a null terminating byte that is never found in normal text (represented as \0). When you find the null character, you know that

you have reached the end of the string. In this implementation, we will instead store the length of the payload in an integer named value length. This is less efficient than a null terminator since integers consume at least 4 bytes (compared to 1 byte with the null terminator). However, its use here is more pedagogical and will help you understand how implementations are set up behind the scenes.

Figure #1 visually demonstrates how a singly linked list should be organized. In the top of the figure, a linked list item is shown in detail. Note that the size of pointers depend on the machine/OS type (i.e. in a 64-bit machine, pointers are 8-bytes as opposed to the 4-byte pointers present in 32-bit machines).

Figure #1: Structural view of a linked list in memory



Assignment:

Part 1 - Singly-Linked List, Due 2/3/17

You are to implement a singly-linked list with the mentioned features in either C or C++.

- There should be three files in your program: main.c, linked_list.h, and linked_list.c. Sample code containing bare bones versions of each of these programs will be given to you.
- Your implementation should define, in linked_list.c, all of the functions declared in linked_list.h. The implementation of each declared function is up to you. However, do not change any of the declarations from the ones that are given to you. A grading script is used to test your project and to assign grades based on those tests. If you change the API from what the script is expecting, the script will not be able to grade your assignment. As an unfortunate consequence, you'll get a 0.

- Use the getopt() C library function to parse the command line for arguments. The usage for your program should be as follows:

  testlist [-b <blocksize>] [-s <memsize>]

-b <blocksize>	Defines the basic block size, b, in bytes. Default = 128 bytes
-s <memsize>	Defines size of memory allocated in bytes. Default = 512 kB

- Make sure that your program does not crash in any case. Here are the scenarios that your program should account for. In these cases, your program should simply print an error, skip the given instruction, and continue to work (i.e. do not exit for any reason).
  - Deleting non-existent keys from the list.
  - Trying to insert keys after the given memory is full.
  - Trying to insert values that do not fit in the payload section.

Part 2 - Stratified Linked List, Due 2/10/17

Modify the linked list you implemented in part 1 to make a multi-tiered list that groups keys into a number of disjoint intervals and keeps numbers from those intervals in separate lists. Take another integer, t, add input that indicates how many levels/tiers you will have in the tiered linked list. Total memory, indicated by M, stays the same as in Part 1, which means that every tier now will contain M/t bytes of memory. Each of these regions will act as an independent linked list as in Part 1.

In order to distribute numbers onto these separate linked lists, divide the integer number space (i.e. [0, 2^{31} - 1] or [0, INT_MAX] for signed integers (Side note, the value INT_MAX is contained in the header file: climits), you can safely assume that the input keys to your program are all non-negative) equally into t regions. Note that you are only dividing the entire number space equally, and this division should not depend on the particular input array you are working with. Therefore, it is possible that a particular input sequence could be placed into only one out of the t tiers simply because all the numbers in the sequence map to that tier. However, if, in general, there is a uniform sample of input keys in the range of [0, INT_MAX], then the tiers should contain roughly the same number of keys.

Figure 2 (shown below) demonstrates the organization of a 4-tier list. All numbers in tier i should be less than those in tier $i + 1$. You are NOT allowed to use the modulus operation on an input to determine its tier. Instead, you should use either division or bit shift operations. The following are requirements of your program (Remember, like part 1, part 2 must be implemented in C or C++):

- There should be three files in your program: main.c, linked_list2.h, and linked_list2.c/cpp. Similar to part1, a sample main file will be provided that includes some test cases for you to experiment with.

- Provide implementations for all of the functions listed in part 1. Note that those functions may work differently in the tiered design. In some cases, the function arguments will also change. For example, the Init(M, b) function will change to Init(M, b, t). Furthermore, the PrintList function will change as well. Do not print empty tiers (i.e. where no insertions have occurred).
- Name your program testlist2, and this time, your program should accept the following command line arguments:

testlist2 [-b <blocksize>] [-s <memsize>] [-t <tiers>]

Figure #2: Organization of a tiered linked list

Report Documentation

Provide a PDF report describing your findings in both Parts 1 and 2. You only have to write one report! Do not write two separate reports for the two parts of this machine problem. For part 1, do you notice any wastage of memory when items are deleted? If so, can your program avoid such wastage? How would you do so? Can you think of a scenario where there is space in the memory but no insertion is possible? What is the maximum size of the value when the pointers are 8 bytes? For Part 2, derive a general expression for the range of numbers that go into the i-th tier of the list.

Submission Instructions

Submit a zipped folder that contains two folders named MP1part1 and MP1part2, and a PDF report named MP1.pdf. Both folders (MP1part1 and MP1part2) should contain 3 c/cpp/h files. Demonstrate your work during lab meetings. Make sure that your program runs on the CSE department's linux server (linux2.cse.tamu.edu). Remember that we are using a new platform Vocareum (available at vocareum.com) for submitting machine problems. Please register as a student on this website, play around with it, and become

familiar with it so that there will be no issues before the assignment's deadline. Please don't be afraid to talk to your TA or instructor if you have any issues.

Grading Rubric:

Points will be assigned according to the following rubric. Each of the two parts of the assignment are worth 80 points for a total of 160 points. The report portion of the assignment is worth 40 points, bringing the overall assignment total to 200 possible points.

Basic things to note: do your best to account for every single edge/corner case that you can think of. The grading script will specifically test these cases to ensure that your program is completely correct. Also, your program should not produce segmentation faults. If your program does segfault while testing a function, you will receive a zero for that function and any other related functions. The grading script will still try to give you as much credit as possible in the event of segmentation faults.

- Part1 & Part2:
  - Init: 16 points, Full credit will be awarded for correctly initializing variables and setting up the list (i.e. your code should completely set up the list of nodes starting at the head pointer and connect them all into one big list using the next pointers. The nodes will be uninitialized when you set them up and should contain no payload).
  - Destroy: 6 points, Full credit will be awarded for correctly deleting every element in the list and then returning any used heap memory to the operating system using either free or delete.
  - Insert: 16 points, Full credit will be awarded for correctly inserting elements into the list that can be looked up afterwards. Inserts into a full list should fail.
  - Delete: 16 points, Full credit will be awarded for correctly removing elements from the list. Ensure that your delete function does not mess up the order of pointers or result in any segmentation faults.
  - Lookup: 8 points, Full credit will be awarded for correctly finding elements that are in the list and returning NULL when the element is not in the list.
  - PrintList: 8 points, This function is hard to grade based on a wide variety of formatting variations. You will be graded solely on whether or not your function returns something that is slightly coherent. However, it is strongly recommended that you put time into developing this function so that you can debug your program as errors come up.
  - Bonus: 10 points, Full credit will be awarded for correctly reclaiming a block to be used again in a future call to Insert.