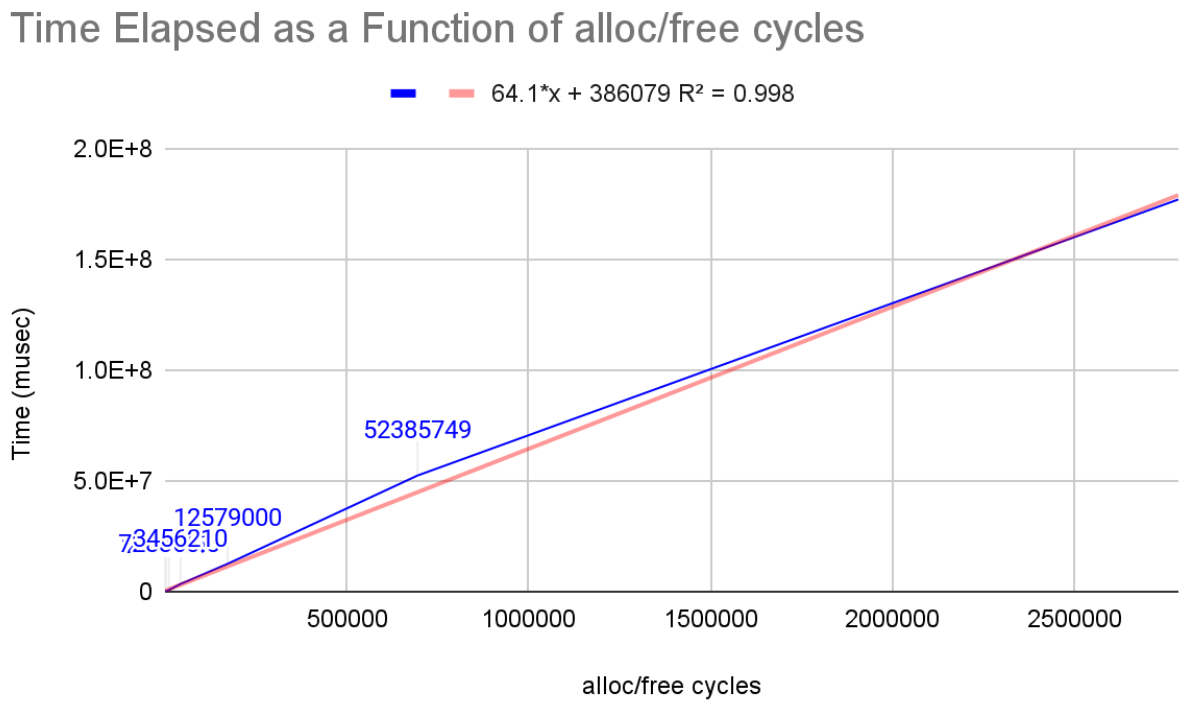
Raul Escobar
328003859

Report PA1



This was done with basic block size at 128 and total memory size at 128MB

Cycles	Time
4	80
6	665
8	1315
10	969
12	1937
14	1666
16	1616
18	1574
27	2802
44	4633
65	5492
90	9832
106	14627
119	11675

Raul Escobar
328003859

152	22677
189	24058
541	79819
2432	244405
10307	728361
42438	3456210
172233	12579000
693964	52385749
2785999	176969000

Bottleneck Analysis

Above is a graph of time elapsed as a function of alloc/free cycles while running the ackermann function. Perhaps this is not the best function to use as it has a random component to it while allocating memory. Not only is there a random element to the function, but the granularity in which we can use it to test the speed of alloc/free cycles is very bad. As can be seen in the table above, the first half of the alloc/free cycles are in the order of 0-100, but then it very rapidly increases to 2,000,000. These two things combined make it difficult to extract useful data. This may weaken the claim that our alloc/free methods are performed in linear time, but there is still some credibility to it.

Alloc

Lets trace a simple alloc/free cycle to determine its time complexity and where it can be improved.

When an alloc() method is called, first we must determine the blocksize to return and index in which it belongs in the FreeList, which is a simple math calculation O(1), then we check if it is available on the FreeList. Worst case, we want the smallest block, but only the largest block is available which means we have to iteratively search for the available block and split it until we get the desired size (splits are O(1) time complexity). The LinkedList methods are all O(1) time since we are simply pushing and popping for alloc.

So far the sum total time complexity of the alloc call is O(log(M/B)) where M is total memory size and B is the smallest unit of basic block size.

After we find the blockhead to return, we perform some bookkeeping which takes O(1) time, and return the writable part of the block which results in the entire thing being O(log(M/B)) time complexity.

Free

Raul Escobar
328003859

When the free method is called, first we must check that input is not a nullptr and typecast the memory to a block header which is constant time. Then we iteratively get our buddy and check that he is free. Then worst case, if he is free we have to remove him from the FreeList. Since we are removing in a single LinkedList, this takes O(n) time complexity. Then we merge our block and its buddy and return 0.

The total time complexity of this method is O(n) time complexity where n is the length of the LinkedList in the FreeList; however, since the maximum amount of block headers we can have in a LinkedList is M/2*B, the time complexity can be rewritten as O(M / B).

Total

In total, the time complexity of a cycle is O(M / B) , where M is total memory size and B is the smallest unit of basic block size. This means that the time complexity of a single alloc/free cycle is dependent entirely on memory allocated and smallest memory unit specified. This agrees with our time graphs because since we kept our total memory size and basic block size constant, we should see that as the cycles increase, the total time should increase linearly. We can interpret the slope of the graph to be the best fit time taken for one cycle which is 64 microseconds.

The bottleneck in our code seems to be the remove method of our LinkedList. This is because, since it is singly linked, even though we have the address of the blockheader to remove, we cannot point the blockheader before it to the blockheader after the one we are trying to remove which leads to memory leaks. The work-around was to iterate over the entire list one pointer searching for our blockheader to remove, and another one that is lagging which is then used to point to the blockheader after the one that is to be removed. This results in linear time.

The solution would be to make the LinkedList a doubly linked list, this allow us to go backwards from the blockheader to remove and have the previous blockheader point to the blockheader after the one we are removing. This reduces the time complexity to constant time, but increases the space complexity because we have to add a linear amount of extra pointers.

Another solution would be to decrease the size of M, or increase the size of B, but this would make the buddy system impractical so a balance must be found to optimize time and practicality.