Project 3 Final Write-up
Beshr Islam Bouli and Christian Teshome
11/10/24

1. Design Overview

In this design overview, we will first discuss the overall structure of our implementation and summarize our current performance, and then go into detail about how each of our main functions work, and the reasons we implemented them in this way.

We are using the standard binned free lists approach described in lecture / recitation where we have a free list for each power of 2. However, the size of the blocks in each free list doesn't need to be exactly a power of 2, the powers of 2 just define the bounds of what sizes should go into the corresponding free list. For example, the 512 bin can have blocks with sizes anywhere from 512 bytes to 1023 bytes. To make deleting from the lists easier, we implemented them as a doubly linked list with prev and next pointers.

Our blocks have both a header and a footer, and each of them are 4 bytes long and store the full size of the block. We are also using the footer to indicate whether a block is free or not, where we set it to -1 to indicate the block isn't free. We are also keeping track of the last block in our heap in order to do the "wilderness" optimization mentioned in the blog post linked in the project handout.

1.1 Current Performance [Beta & Final]

Right now, when we run our code on telerun, we see that the traces have around 87% utilization and 97-99% throughput, giving us a performance index around 90-91. On the real programs, we get a performance around 49-50, and it seems like treeadd is the main program contributing to the lower performance.

For our final submission, we see that our traces have roughly the same performance, with around 87% utilization and 94-96% throughput, giving us a performance index around 90=91. Our performance on the real programs improved significantly to around 87.

1.2 my_malloc()

In malloc, we first compute the aligned size taking into account our 4 bytes for the header and 4 bytes for the footer. If this aligned size is less than our min block size (set to 32), then we round up the aligned size to 32 to avoid fragmentation and also to make sure that we have enough space in the block to store the next and prev pointers of our free list nodes.

Then we check in our free lists to see if we already have a free block that will fit the requested size. We first check the lower bin and then check the higher bin if we couldn't find anything in

the lower bin. For example, if the requested aligned size was 640, we would first loop through the 512 bin and take the smallest block in there with a size >= 640. If there was no block there, we check the 1024 bin and then just take the first block since it's guaranteed to be >= 640 bytes. This happens in our best_fit function.

If we found a free block, if the difference in size between this free block and the requested aligned size is greater than our min block size, then we will break up the block and put the extra portion into our free list. For example, if the requested aligned size was 640 and our best fit was a 720 block, we would break up the 720 into 640 and 80, and we would give the 640 block to the user and put the extra 80 back into our free list.

If there was no free block, then we will have to call mem_sbrk to expand our heap. However, if the last block in our heap is free, then we don't need to call mem_sbrk with aligned_size, we only need to call it with the difference between the last block's size and the aligned_size. For example, if the last block in our heap happens to be free and has size 30, and the user's requested aligned size is 50, we only need to mem_sbrk by 20 bytes instead of 50 bytes. This happens in our new_sbrk function.

1.3 my_free()

In free, we implement both forwards and backwards coalescing. First, we check if the block to our right is free and if so, remove it from its free list and add its size to our total. Then, we check if the block to our left is free, and if so, remove it from its free list and add its size to our total. Then we create one new combined block and add it to our free lists. We also have some logic to make sure we are correctly updating our "last" pointer if we are coalescing.

For example, if we had a block of size 2, then a block of size 4, and then a block of size 8, called B2, B4, and B8 all next to each other, and we are freeing B4 and both B2 and B8 happen to be free, we would initialize our total size to 4, and then see that B8 is free and add 8, making our total 12. Then we would look to the left and see that B2 is also free (using the footer) and add 2, making our total 14. We delete both B2 and B8 from our free lists and insert a new size 14 block into our free lists where B2 was.

1.4 my_realloc()

In realloc, we first check if the size of the block they pass in is greater than the size they are requesting, if so, we don't need to do anything and we can just return the ptr they give us. But, if the difference is size is greater than our min block size, we will split the block and free the extra portion. For example, if they give us a block of size 480 and their requested size is 320, we will shorten their given block to size 320 and then free the extra 160.

Otherwise, if the pointer they give us happens to be the last block in the heap, then we decided to just mem_sbrk the additional amount instead of mallocing and copying the data over. For

example, if they give us a block of size 480 that happens to be the last block in the heap and their requested size is 640, we will just mem_sbrk 160 and return their original pointer.

We also do coalescing in realloc: If they are requesting a larger size, and the block to the right of their block happens to be free and the total of their sizes is larger than their requested size, then we will just combine the two blocks instead of mallocing and copying the data over. For example, if give us a block of size 480 and their requested size is 640, if there's a block of size 320 to the right of their block, we can combine them to make a block of size 800 and give that back to the user. In this case, we would also split the combined block into a 640 block for the user and a 160 block for us to free.

Also, when we are looking at the block to the right for the coalescing mentioned above, and if that block was free but its size wasn't big enough to use, we will check if that block to the right was the last block in the heap. If so, then we will expand the heap by whatever amount is missing instead of mallocing and copying the data over.

If none of these conditions are true, we use the default implementation where we malloc their requested size and copy the data over and then free the original block.

1.5 Other Notes

In init, we mem_sbrk 12 bytes that we don't use. This is because our header size is 4 bytes, so we need to offset our memory addresses by 12 in order for the address we return in my_malloc to be 16 aligned.

We use the following two macro definitions to read from and write to the header and footer locations of a block.

```
#define h(p) ((void*)((char*)p - SIZE_T_SIZE))
#define f(p,sz) ((void*)((char*)p + sz - 2*SIZE_T_SIZE))
```

h(p) takes in a pointer to the block (starting after the header) and returns a pointer to the start of the header for the block.
f(p) takes in a pointer to the block (starting after the header) and the size of the block, and returns a pointer to the start of the footer for that block.

1.6 Attempted Optimizations / Plans for Final

There were a few optimizations that we attempted but either weren't able to get working before the beta deadline, or didn't improve our performance significantly.

One of these was to try to keep our blocks cache aligned so that they weren't spread onto multiple cache lines unnecessarily. We feel that this would improve our performance on the real programs.

We also tried using a different scheme for our binned free lists instead of using the powers of 2. For example, we tried making the bins just increment by 16 up to 512, and then do powers of 2 from 512 onwards.

Also, we noticed that the treeadd benchmark program did a lot of 32 byte allocations, so to reduce the amount of mem_sbrk calls we make, we tried to add some extra amount every time we call mem_sbrk. For example, whenever we call mem_sbrk(size), instead call mem_sbrk(size + 320), and then put those extra 320 bytes in our free lists. This was able to improve our performance on the real programs significantly (up to ~70) and also give high performance on the traces. However, we couldn't actually implement this and have it be fully correct. This is because when we do mem_sbrk(size + 320), we need to first check if (size + 320) is less than the maximum size of the heap. We know that the max heap size for the traces is 50MB, but we don't know the max heap size for the real programs, so we can't guarantee correctness for this optimization so we didn't include it in our submission.

One other thing we tried was to implement a binary search tree to store the free blocks instead of binned free lists, but we weren't able to debug it in time. This would make it O(logn) to locate blocks in the tree, but would also require making sure the tree is balanced after every time we remove or add a block to the tree and might make coalescing a bit slower.

We plan to revisit all of these ideas for the final submission, and also try playing around with OpenTuner to see if we can adjust some parameters in our program.

1.7 Final Optimizations

For the final, our main priority was improving our performance on the real programs (specifically treeadd), while maintaining our performance on the traces from the beta. We first revisited the idea of calling mem_sbrk() with an extra amount each time. However, based on looking at some of the top beta submissions, we instead decided that, whenever we need to call mem_sbrk, we round up the amount to some minimum threshold size. This logic happens starting at line 233 in our code inside my_malloc(). Our minimum threshold size was called PERFECT_SIZE in the code. We experimented with a few different values for it but ultimately decided to set it to 4096.

1.8 Work Log

#	Date	Start time	Duration		Descriptions
			Beshr	Christian	
1	10/25	6:30 pm	3	3	Read handout, write team contract, implement validator
2	10/27	3:30 pm	2.5	2.5	Reread handout, implement basic free list, implement basic binned free lists, implement slightly better binned free lists (if free list empty, take one from bins above)
3	10/28	12:00pm	3	3	Implement splitting in malloc (ex. Looking for block 4 but find block 32 -> break up into block 16, block 8, block 4, and return block 4 to user)
4	10/29	4:00 pm	6	6	In free, we look to see if there's a free block adjacent to the right. If so, we combine and move into next bin. Also do left coalescing in same way. Improve realloc and adding the different cases Also make binned lists not store exact powers of 2
5	10/30	6:30	4.5	4.5	Implement and debug heap wilderness optimization. Keep track of last block in heap and modify malloc free and realloc to use it
6	10/31	5:15 pm	3.5	3.5	Try not powers of two scheme where its linear up to 512 and exponential after 512. Try binary search tree
7	11/1	5:00 pm	4	4	Try final optimizations: cache alignment, mem_sbrk(size+320), etc..
8	11/9	9:00 pm	2	2	Implement mem_sbrk(PERFECT_SIZE) change for final submission
Total			**28.5**	**28.5**	