6.172 Project 3

Adriano Hernandez & Nancy Vargas

Design Overview

Beta

We used a simple version of the binned free lists algorithm. In an array, we have thirty two "bins" (of which only the top twenty eight, roughly, are used) for each power of two between zero and thirty one. The largest allocation size we support is $2^{31} - 8$ since we had an eight byte header storing the size of the block as a logarithm in the lowest five bits (i.e. the index of the bin it belongs in). Each bin is a singly linked list of free (not necessarily consecutive) memory. In each bin (i.e. at index k) the size of the blocks is 2^{k}.

Malloc will try to find a block in the bin that is the lowest power of two above the requested size plus the header size. If it fails, it tries to find a larger bin and split it up into a sum of smaller powers of two (terminating on two instances of the desired power of two). This splitting is called "cascade." The unused fragments of the larger block during the cascade are inserted into their corresponding bins since they are all powers of two in size. If it fails to find anything, it simply uses *mem_sbrk* to create a new block.

Realloc uses malloc if necessary, but prefers to simply do nothing if the reallocated size is under the block size (which may be smaller than the size originally requested by the user). Free is similarly very simple: it just appends the block to the corresponding bin's linked list's head. Initialization simply sets all bins to point to *NULL* and ensures that the memory begins eight byte aligned. Since all powers of two above eight (we do not support smaller powers of two as sizes), when added to eight, are eight byte aligned, we may note that by induction the memory will remain thusly aligned.

We have many more details in the header file *allocator.h* with further expositions of our ideas outlined below (as well) in both *master* and *adriano/flexibins-coalesce*. You are encouraged to read there for a more full exposition including an explanation of the various helper methods such as "request" and "cascade" which implement the multiple steps of malloc and cascade respectively.

Final

We ended up using "flexibins" for our final. Because it was implemented in our beta, please read about it below in "Ideas Tried" for "Beta." The reason it was so slow for our beta is because we had a bug that we did not originally know of. In the *cascade* function which would, after finding a larger bin with an available block, split that block, storing the leftover into its corresponding bin and returning the requested size to the user, we were actually never splitting this block. That is because we were checking if the block left over was sufficiently large to keep in a bin (the smallest allowed is 24 bytes for a header and then two pointers inside the body).

However, we were not checking if the size was over 24 bytes (which is 8 + 2^4 bytes), but instead if the floor of the logarithm of the size (which in this case is the bin index in which it would fit, and must be at least 4) was over 24. Thus we were failing to split for any block whose leftover was less than 2^{24}, which in all likelihood limited all blocks whose original size was less than 2^{25} from being split, which was an unusual and very large number and probably was never reached for almost any traces (we inserted print statements and they were not printed).

Our mistake stemmed from the fact that we used a macro that had an ambiguous name and, in implementing this “flexibins” approach the day the beta was due, we had, in confusion, used it wrong.

We tried to also add coalescing to flexibins as was done by rank 0, but we did not succeed in time. More on that is below.

Example Scenario

Here is an example trace to explain our program’s behavior. Say we try to malloc 8 bytes. Then, our program will first increase the size to the minimum allowable of 16 bytes (because it needs to store a doubly linked-list inside). Then it will align it (in this case, keeping 16 bytes). Then (assuming our binned free lists are empty) our program will find that there is nothing in any bin and use *mem_sbrk(24)* to allocate a new block with an 8-byte header, inserting the size “16” into that header (a *uint64_t*) and returning the pointer at the end of the header.

Assume then they want to realloc this to 35 bytes. 35 is between 32 and 64, so it would go into bin 5 (as 32 is 2^5). However, the block the user is giving us is too small, so our program would try to find a replacement. Since the bins are still empty, it would find nothing and sbrk. However, it would check if the block given was the last block (by checking the header’s size and then adding that to the pointer and seeing if this was equal to *mem_heap_hi()*). Since in this case it is the last block, it would simply *mem_sbrk()* the remaining 19 bytes necessary to take 16 to 35 (from the point of view of the user. However, this would first be aligned to 24, so the program would *mem_sbrk(24)* and update the header to store the 8-byte aligned size of the block, 16 + 24 = 40.

From this scenario, note how we do not count the header size in the header. Note how the header stores the value of the block and not the size as requested by the user. The block’s actual size is obviously guaranteed to be larger than the user’s requested size. Note that bin indices are ranges for the floors of the logarithms of the sizes of blocks including their headers (not just the body).

Say we then realloc to a smaller size, say 24. Nothing is done here because the size is smaller.

Say we then free this block. Our program would find the total block size 40 + 8 = 48. Then it would find the floor of the logarithm of this value: in this case 5. Thus, it would insert it onto the head of the doubly linked-list in bin 5.

Say that subsequently we malloc and then free a bunch of blocks that all go into bin 5. Now we want to malloc something of size 32 (that is to say, at least size 32). Our program will first look into that list and find that it is non-empty. It will traverse the list and use a heuristic to see if any block it finds is a good fit. A good fit in our heuristic is defined such that the block which is being tested is not larger than the desired size plus halfway towards the next power of two. If no such block is found, but a block which is valid (between 31 and 64, exclusive) is found, we will return it at the end of the search (otherwise we'd have returned that "good fit" block earlier).

Say that after that we wish to malloc a block of size 15. This would be increased and aligned to 16. We would find an empty bin (bin 4) and so we would search upwards for a bin with elements in it. We would find bin 5 with a long linked-list. In this bin we would pop the head (assume it has 56 elements including the header, so 48 without the header). We would, noting that we only need 24 elements, "split" it (also called "cascade" in our code) into 24 elements in the beginning (overriding the header) and 32 elements in the back. The 32 elements we would find to be sufficiently large (larger than, or equal to, 24) so we would write to their first 8 bytes a header saying "24" (since 32 bytes for a block in total minus 8 bytes for its header is 24 bytes of body) and insert that into bin 5 since 5 is the floor of the logarithm of 32. That way we can reuse the "leftover" of the block we just fetched. Subsequently, we'd return that updated first 15-byte block (plus 8 bytes of header for a total of 24 bytes, though, of course the user can only see 16 of them) to the user.

Ideas Tried

Beta

We also tried binned free lists such that the bins held blocks between the power of two of the index and the next power of two instead of a fixed size. This ended up being slower than regular binned free lists even with coalescing in both directions. We call this the "flexibins" implementation and its in a correspondingly named branch.

In this case we allocated the size requested by the user plus a heuristically decided pad. We decided on no padding. However, we did check the corresponding bin when a user requested a size first. In that case we searched for the first block that was good enough according to a second heuristic. That second heuristic would only consider blocks such that the block's size was not above the halfway mark between the desired size and the next power of two. If no such block was found, but there was a block such that it was valid (i.e. longer than the desired size and passing a third heuristic, which was a dummy) it was used, and otherwise another block was *mem_sbrk*-ed.

Cascade was different. Instead of splitting into as set of blocks whose lengths were powers of two and their lengths' sums was a larger block's length, we split into a single block of the desired length and another of some leftover length which was placed in the correct bin by taking a floor of the logarithm of its length.

Free was roughly the same and so was initialization. Realloc was roughly the same, but would try to *mem_sbrk* the correct amount and return the original pointer when the given block was at the end of the heap (important for traces 9 *v0* and *v1*). This change to realloc was necessary to avoid running out of memory.

We also tried coalescing by storing pointers to the spatially (in memory) next and previous blocks. On each free, we would search for the leftmost and rightmost pointers around a given block by traversing the doubly linked list created by such a structure (stopping once we hit an unfree block). These blocks' lengths were summed and then that entire body was turned into a block which was inserted into the proper bin. We used a doubly linked list for the free lists and so could easily remove each of the component blocks of the entire body. We made sure to mark blocks as free or unfree using the upper thirty two bits in our eight byte header (the lower thirty two bits represented the length of the block). We also stored a pointer to the last block in the entire heap so we could increment the spatial doubly linked list on *mem_sbrk* calls. We made sure to update the bitfields in our eight byte header carefully as necessary. We did not rigorously test, but know that no errors were introduced. We are unsure as to whether we made a mistake and forgot to mark blocks as "free" or if coalescing was actually unhelpful. We passed tests, but did not have time to implement coalescing benefits checks. This requires further testing in the future.

Final

We gained some inspiration from rank 0 and tried to implement it. We used ranged bins as I have described immediately above, but instead of having a doubly linked list for the adjacent (in memory) blocks, we used footers instead. We shrunk our headers and footers to have a size of 4 bytes with the most significant bit representing whether the block was free or not and the other bits representing the length (as an unsigned integer). We offset the very first header from *mem_heap_lo()* by 4 bytes (from the earliest 8-byte alignment) and then aligned every size to be 8-byte aligned. That way, each pointer returned would be 8-byte aligned (since 4 bytes before it was 4-byte aligned, but not 8-byte aligned) and then the footer's end would be 8-byte aligned, meaning the next block (4 bytes later) would be 4-byte aligned but not 8-byte aligned just like the previous block (and so forth).

Coalescing involved only coalescing blocks that were immediately next to each other, since for any sequence of freed blocks, the last free block is next to two blocks which must have already coalesced with the blocks further away (think of it as a disjoint union), which simplified our code greatly from previously when we looped.

However, neither of these elegant implementations worked in time for our final, so we simply used regular flexibins with a single bug fix which boosted its performance by around 5.

We found a bug in our *cascade* method which was stopping it from ever splitting big blocks into smaller pieces, and instead it was using entire big blocks for cases that called for much smaller ones. We fixed it. Read more in "Final" above.

We did not have time to implement any more exotic ideas like bubble sorting online or using fancier data structures.

Ideas Not Tried

We also had various ideas ranging from the more concrete to the poetic. We will begin with the most concrete ones and continue onto the less practical ones. They are in a list format for easy digestion.

1. Bubble sort flexibins online as we search for a sufficiently good fitting block. Additionally, store a start and end pointer to the list. This way, if we are on the lower half of the bin we can start on the start of the list, and if we are on the upper half, we can start on the end of the list and more quickly find a good element. It's unclear how much overhead this may introduce, but is worth a shot. Alternatively, sort with insertion sort.
2. Instead of picking an arbitrary element that is valid in the flexibins bin search, pick the best such element even if no such elements were sufficiently good.
3. Improve coalescing by having fixed "atomic" block sizes (i.e. some small power of two) and having blocks of these atomic blocks. This way we simply do pointer arithmetic. Additionally, instead of storing pointers, store offsets (from heap low or high).
4. On realloc, make sure to cascade if the realloc size is smaller than the current size by a lot.
5. For the blocks, try out a balanced binary search tree like an AVL, Red-Black or random binary search tree (if such random binary trees are on average balanced: this would require a proof). Alternatively, explore skip-list structures.
6. Initialize with non-*NULL* bins intelligently.
7. Find a better splitting scheme than powers of two, such that it is also not ranged.
8. Use smart pointers by offsetting by multiples of eight with large or variable-size headers. This would allow us to encode (at least) boolean values into the pointer itself, and thus perhaps use it to find bins. Alternatively, we could introduce an additional set of bins that are special. These special bins would be such that within each bin all elements are within two pointer bounds. When an element is freed, if it lies within any pointer bounds, then it belongs to that free list, and thus no headers are necessary. However, we may run out of blocks in a given free list, in that case we can double the number of pre-generated blocks in that free list using a new consecutive range. The old blocks would go into a second free list equal to our best current free list implementation. To find blocks we always search the pointer bounded free lists first and the regular one second. There is no cascading in the pointer bounded one, but there may be cascading from that one to the normal one. The point here is to deal with powers of two or other such values, where

appending a header would lead to the use of excessive padding. The pointer bounded free list is meant to deal with the values which shouldn't be padded, while the regular free list is meant for the elements whose padding would be minimal.

9. Define a vector field (i.e. the gradient of an energy function, potentially stochastic and probably discrete) such that its sinks point towards beneficial locations to take. Basically, increase the heap along the (one dimensional) space, then decide how to allocate it by using this vector field to path towards places at which it would be beneficial to malloc from. For example, pick one or more random locations in the space of the heap. Follow the vector field towards the sincs and at some point decide to malloc from there. You can think of these vectors as forces or flows or likelihoods or suggestions of some kind. There is no need to relegate them to be one dimensional, it is possible to define a graph structure using data in the free blocks and then vectors along the edges. The original idea here was that it would be poetic to encourage coalescing through gravitation on the heap space.

Beta Results

Our original implementation had great throughput (100.0) but bad utilization (around 54). The total geometric average was around 74. It seems to have struggled on traces three, eight, and nine. Luckily, due to the power of two padding, it managed to survive trace nine since it only had to malloc (in the realloc calls) *O(logN)* times for *N* realloc calls, assuming the size increased linearly. We did not have time to analyze the other traces. Our coalesced range binned free lists (flexibins) implementation had a total geometric average of around 71. It also struggled on trace nine.

Final Results

We were able to fix a bug in our "flexibins" ranged bins implementation and that boosted that implementation from around 71 on average to 76 on average, which beat our default solution for powers of two. This was an improvement over our beta, though we weren't able to implement coalescing in time, which we think would have improved our runtime significantly.

Acknowledgements

Beta

We would like to thank some students and TAs/LAs for their help on the project. Hoang gave us the original idea for how to design binned free lists and Jeff gave us the suggestion to use *mem_sbrk* on the tail portion of blocks that were already on the end of the heap in *my_alloc*

and *my_realloc*, though we didn't end up submitting it. Sameer (as well as Philip and August) helped us debug our implementations. Lastly, we thank Piazza.

Final

We thank rank 0 and Stephen Otremba for verbally (or in the case of rank 0, literally) giving us suggestions on how to implement header/footer-based ranged bins. We thank our MITPOSSE for giving us the wise suggestion of writing an example/test scenario.

Work Logs

Beta

Adriano

Format is date (or other tag), followed by (start) times, amounts, and work done.

Thursday, Oct 21st	9:00AM	1	Read handout, brainstormed with Nancy
Monday, Oct 25th	8:00PM	2	Wrote broken code for binned free lists, set up (unused) python simulation
Tuesday, Oct 26th	5:00PM, 9:30PM	5	Implemented working binned free list in the simplest way I knew
Wednesday Oct 27th	12:00PM, 9:00PM	10	Fixed flexibins with Nancy, implemented coalescing for flexibins, wrote writeup
Beta Total	-	18	Worked primarily on the allocator

Nancy

Wednesday, Oct 20th	10:00PM	0.5	Read project 3 handout carefully
Monday, Oct 25th	9:00PM	2.5	Wrote untested validator
Tuesday, Oct 26th	6:00 PM	6	Refined validator: rejects bad malloc on all traces
Wednesday, Oct 27th	12:00AM, 5:30PM	10	Finished validator: accepts our malloc, debugged flexibins (span binned free list)
Beta total	-	19	Worked primarily on the validator and debugging the allocator

Final

Adriano

Wednesday, Nov 3rd	7:00-8:00, 9:00-10:00	2	Read Rank 0, peer programmed a little with Nancy, documented
Thursday, Nov 4th	2:00-2:30	0.5	Wrote a lot of boilerplate and pseudocode, documented
Friday, Nov 5th	5:00-12:00	7	Tried to get everything together, but failed
Final Total	-	9.5	Worked primarily on debugging, implementing new strategies, planning approach

Nancy

Wednesday, Nov 3rd	8:00-12:00	4	Tried to implement coalescing
Friday, Nov 5th	5:00-12:00	7	Tried to debug coalescing and ranged free lists
Final Total	-	11	Worked primarily on new implementations, debugging

Team Dynamics

We did a lot of peer programming. In fact, most of the work for this project was done together. We both were able to follow our commitments without issue, though we were also both busy and so weren’t able to work on the project perhaps as much as we needed to truly yield meaningful performance increases for the final. This was a fun project and we would love to learn more about these subjects in the future.