6.172 Project 3

Adriano Hernandez & Nancy Vargas

Design Overview

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.

Ideas Tried

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.

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 Release Plans

We plan to explore the first four ideas we mention in "Ideas Not Tried." Beyond that we plan on seeing what the top teams do and brainstorming with friends and amongst ourselves. It is harder to figure out how to get good performance in this lab than we expected, so we will definitely need to think a little bit. However, it's likely that a combination of simple and intelligent heuristics will be effective, and therefore we will try to seek out such heuristics and combine them. We have tried to have well-documented and easy to expand code, and hope to keep it that way as we try new and different heuristics effectively.

Acknowledgements

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.

Work Logs

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