6.172 Project 3

Sabrina Tseng
Arkadiusz Balata

Beta version, October 29, 2020

1 Overview of Design

1.1 freelist_item struct

We represent each node in our freelist as a freelist_item, which we define as a structure. In order to maximize memory utilization, we define our struct with three sets of 4 bytes. The first bit of the first four bytes is a "free bit". This bit is set to one if the block is currently free. The next two bits are always empty. The rest of the first four bytes is dedicated to the size of the block. Because each block has to be byte aligned to 8, we know that all of the block sizes are multiples of 8. So, to save on memory utilization, we simply store the actual size of the block divided by 8. This way, we do not need all 4 bytes for the size - only 29 bits, which is the remainder of the first 4 bytes. The next four bytes is the memory address offset of the previous node in the freelist. Because we are only storing the offset from the start of the heap, we require only 4 bytes, not the usual 8 expected for a memory address. The final 4 bytes consists of the memory address offset for the next node in the freelist. Each free block also contains a footer which is exactly the same as the header, but this is not included in the struct since it is located at the right edge of the block. This structure is incredibly compact, so as to improve memory utilization.

1.2 Binned Free List

Our memory allocator uses binned free lists, where each bin is a linked list and bin i stores free blocks where $2^{i+3} \leq size < 2^{i+4}$. We don't need bins for blocks of size < 8 since we are aligning to 8 bytes, so an allocated block will always be at least 8 bytes. Each free block stores a header and a footer which includes the size of that block and whether or not is free (explained in more detail in the Coalescing section), and pointers/offsets to the next and previous free blocks in the linked list.

In malloc, we will allocate a block slightly larger than what the user asks for in order to satisfy alignment and leave space for a header and footer. Then, if we want a block of size s, we search in bin $\lfloor \log_2 s - 3 \rfloor$ for a free block. We split off the amount needed and then put the rest of the free block back into the freelist in its correct bin.

1.3 Coalescing

When we free blocks, we look for any opportunity to coalesce them with their left or right neighbor. To keep throughput high, we do this in O(1) time using a header and a footer. The header and the footer store the size of the block that they are in. However, because we stored the size divided by 8, we were able to use the leftmost bit of the size to say whether the associated block is free or not. When we free a block, we can easily check whether its adjacent blocks are free by accessing the footer of the previous block and the header of the next block (by decrementing or incrementing the start memory address by the footer size and the size of the current block respectively). By checking the first bit of the size against 1 (with a bithack - size & 1 « 31), we coalesce the current block we are freeing with adjacent blocks that have this bit set. Actually coalescing simply involves removing all the individual blocks from the free list, and inserting the new block with a larger size into the free list (or not if we are freeing from the end of the heap, explained in section 1.4.3).

1.4 Other Optimizations

1.4.1 Packing pointers/Offsets

Within each freelist item, we store pointers to the previous and next items. This makes it fast (constant time) to remove a block from the freelist, which we do in malloc and also in free when coalescing. However, instead of storing the actual 8-byte pointers, we store only the 4-byte offset from the start of the heap, since we know that the max allocation size is $2^{31} - 1$ so the heap should never grow beyond $2^{32} - 1$, the max size for a 4-byte unsigned int. This allows us to pack 2 "pointers" into an 8-byte region.

1.4.2 Extending heap in realloc

We made a few optimizations in realloc. We can get the actual size of the block that the user gives by looking in the header. If the new requested size is smaller than the actual size that we allocated for the block, then we do nothing and return the same pointer, since there is already enough space.

Another optimization is that if the user wants to expand a block, and that block happens to be at the right edge of the heap, rather than having to possibly reallocate an entire new block to return, we can just extend the edge of the heap by the amount needed and return the same pointer back to the user.

For the final, we may be able to optimize realloc further by actually making the block smaller if the new size requested is much smaller than the existing block, to avoid wasting memory. We could also check if there are free blocks to the left and right if a block needs to be expanded.

1.4.3 sbrk wrapper and local heap pointer

If we free a block at the right edge of the heap, rather than adding it back to the freelist, we can just pretend that the heap ends at the start of that block. We implement this using a local pointer to the end of the heap called `sp`, which can either point to the actual end of the heap, or a smaller address as long as none of the memory after `sp` is in use. Then, we create a wrapper around `mem_sbrk` called `my_sbrk`, which first checks if there is available memory between `sp` and the actual end of the heap that can be reused, and only calls `mem_sbrk` if more memory is required.

2 Correctness

2.1 Validator

To make sure that the allocator that we wrote worked correctly, we simply asserted the invariants that the staff told us:

- We checked that NULL was never returned from allocating or reallocating.
- We used the given helper macro to ensure that the allocator was 8-byte aligned.

- We simply checked the range of each block against the bounds of the heap to ensure that all blocks were in the heap.
- We iterated through all of the ranges of allocated blocks and made sure that none overlapped with any new block.
- Finally, when reallocating memory, we ensured that the original data was intact by filling each memory block with its own memory address. Then, when we reallocated, we checked that the original memory address was still written in the block.

2.2 Testing the Validator

To test our implementation of the validator, we wrote different versions of the bad allocator that we expect to fail on different invariants in the validator. These different versions are controlled using `#ifdefs`:

- `BAD_ALIGNMENT` does not align the sizes
- `BAD_SIZE` overwrites the size that the user requests, so it often fails on the payload being within the heap. Also if used in conjunction with `BAD_ALIGNMENT`, it overwrites everything with a non-aligned size.
- `BAD_OVERLAP` stores a previously returned pointer and returns that as a pointer to the next malloc call, so that payloads overlap.
- Using no flags will malloc correctly but in realloc, does not copy data from the old block, so it will fail the realloc invariant for that trace.

3 Performance

As an index for tracking performance, we recorded the total score output using the `traces/` folder for each change.

1. The initial Binned Free List implementation with power-of-2 block sizes and no coalescing yielded a total score of 75 (60 util, 100 tput).
2. Coalescing blocks on the right only did not improve performance significantly by itself because many of the traces free from left to right, or in other more complex patterns, and we could only coalesce when both blocks were the same size, since we were only supporting power-of-2 block sizes.
3. Supporting arbitrary size blocks at first dropped both the throughput and the util (both around 30), because coalescing was still very slow at this point (we had to walk through the linked list to remove freelist items), and only coalescing to the right even though we were splitting a lot wasted a lot of memory.
4. Adding footers and allowing blocks to coalesce left as well brought utilization up to about 80, since coalescing on both sides is optimal / results in all free blocks being as large as possible, and throughput was also around 80, since removing freelist items was still a bottleneck.
5. Packing the pointers/Using heap offsets gave us our final score of around 91. This allowed us to remove an item from the freelist in constant time. It kept the memory utilization at around 81, and increased throughout to around the 97-100 mark.

In general, for our beta submission we expect that our code is correct and achieves a performance index of about 89-91 on all of the test cases.

4 Project Log

Date	Start Time	Duration(hours)		Description
		Arkadiusz	Sabrina	
Oct 21	16:00	1	1	Read handout carefully + Partner Contract
Oct 24	16:30	1.5	2.5	Basically finished validator
Oct 25	17:00	2	3.5	Tested validator, finished basic Bin Free List
Oct 27	16:30	4	4	Coalesced, added free flag
Oct 28	00:00	0.5	2	Local sp
Oct 28	11:00	3.5	3.5	Arbitrary size blocks + footers
Oct 28	16:30	4	4	pack pointers/offsets and cleanup
		16.5	20.5	