Custom Memory Allocator

Tsotne Chakhvadze

During previous two week, we had to implement custom memory allocator. One day before deadline I'm submitting my work.

In my allocator there is a implementation three different allocation policies: First Fit, Next Fit and Best Fit. Also my code is written in a general way and can deal every memory alignment values (it tested on 1,2,4,8,16,32,64).

General design principles what used in the memory allocator ares:

1. Memory blocks' (free or used) metadata are stored inside that blocks.
2. To work properly, we don't have to keep a list of allocated blocks.
3. Because we store metadata in the blocks itself, returned address to user is the address after that metadata and plus few bytes to be sure that our returned address would be aligned to our alignment value.
4. When user wants to free memory, he gives a same address as argument in free function, what we returned to him in allocation function.
5. when we recreate free blocks, we check if adjacent blocks are also free and merge them into one big free block.
6. When selected free block is bigger than user wants, we check and split that block, in the way that, reminder bytes should be enough to create free block. if they are not enough we don't split block.
7. As I mentioned, in the previous principle, we give a bigger space to user.
8. Minimum size of the blocks (used and free) are the size of their metadata, because our memory allocator also return valid address too for zero size request.

Speaking on the size of metadata, in my implementation they are

```
typedef struct mem_free_block{
   size_t size;
   struct mem_free_block * next;
   struct mem_free_block * prev;
} mem_free_block_t;

typedef struct mem_used_block{
   size_t size;
} mem_used_block_t;
```

Free one, with size of block and also links to previous and next free blocks. In used one, I store only size of allocated bytes.

For displaying state of heap I used ‘#’ symbol as used one and ‘.’ for free one (dash is more cool and looks much nicer, than uppercase ‘X’, it’s too bigger than point and looks bad).

My code is structured, have comments and compiles without any warning messages too. My work pass all the provides tests (in the test folder) and also ls and ps command test for any allocation policy and any alignment value combination. I added few test files in the folders, also I tested it manually over few thousand times and it handles 99% possible (de)allocations. (Maybe there is 1% cases, that I couldn't imagine, but yeah it works like a charm).

The implementation also includes sanity checks:

1. Forgetting to free memory – in the end if user forget to free memory, my allocator reminds to user every memory address what should be freed with iconic phrase from genial Georgian film: "What they were teaching you in the school?!" (The Eccentrics was filmed in 1974 by Eldar Shengelaia. In the original phrase instead of school, it used word Gymnasy, what was like schools, but I'm not sure there is that word in English with that meaning).
2. Calling free() incorrectly – I check if the given address in in the range of my managed memory, also I know if the address is not aligned with the alignment value, that address is not provided by my allocator. Also maybe user give as used block address, but that address is not the start of used block. And sanity checks in free function, prevents free already free memory (which never been used or used and freed already ("Double free")). And because our allocator return valid address for zero size request, there is a "cool" special case in that check. Returned address to user is start_used_block_addr+aligned_size(MIN_USED_BLOCK+0 (size of used block)) and because the size of that used block is 0, immediately in the same address as returned one to user, starts new free block. And how to check when user wants to free that address, is he trying to free free memory or is that special case?! And because I'm Tsotne, it handles that case properly (reading this, imagine me making dab move :D).
3. Corrupting the allocator metadata. Before every malloc or free call check_corruption function checks if everything ok – if not, something happened, even someone changed free or used blocks size in the metadata, or changed previous and next links in the free block metadata.

To show that sanity checks work, I provided for each case, testing code in the main function (it's commented and to test some case, just uncomment it).

And for bonus section I provided mesure_fragmentation function, which is called before every allocation and it analyzes external and internal fragmentation of current states (because I give to users sometimes bigger size block, to be aligned next address, I added size_without_aligment property in used block metadata (It could be done without it, but for this I should have modify code)).

External fragmentation is calculate like this (sum_free-max_free)/sum_free where sum_free is all available free memory sum and max_free largest free block.

Internal one, I calculate (full_used_space - full_payload) / full_used_space which shows how many percentage is wasted.

Also I count all possible free blocks which are enough for new request.

I tried few scenarios, and depending managed memory size, allocation request sizes, freeing it and its order, I got different results for different policies. In general, Worst-fit algorithm is the best placement algorithm with respect to fragmentation because it results in less amount of fragmentation, which haven't been in our work and Best Fit is worst one, if we consider its time too - for get result in BF we should scan full list.

Before that lab, I already knew how to use gdb, but now, I'm the expert to explore segmentation faults :3 :D

I can lie, but usually I don't, so I like OS classes, and in general, I think when you do something you should do it very good, otherwise you shouldn't do it. To sum up, I spent around 7 days to make this project and from them I spend 2 whole weekend (4 days) to test it properly. I hope, it done

good, and when I heard that if it would work on some machine it’s enough, I was kinda sad. But it’s ok, I love doing stuff almost perfectly, If I can.

So to explore and debug ps and ls, I use set exec-wrapper gdb command (I did not know it before). Also on the lab machine almost everyone got explicit_bzero error for ps. So, I knew that its declaration should be in string.h header, I checked

cat /usr/include/string.h |grep explicit

and it was not, so what I did (-_-): I just download whole libc package (link) and check it in that header file too. Of course there wasn’t.

So again I read carefully man of explicit_bzero and in the version section I found: “explicit_bzero() first appeared in glibc 2.25.” Hope someday they will update lab machines systems.