Making new malloc() and free() functions

Calin Capitanu
<capitanu@kth.se>
KTH Royal Institute of Technology

November 16, 2020

Introduction

The purpose of this work was to research and better understand how the functions **malloc()** and **free()** are implemented on the "unseen" side. That is, how do they manage system calls? How do they structure the blocks that they are given and how to serve them to the user. The way I implemented **malloc()** as of this assignment is a manager that hands out and receives back blocks of memory. The main issue is making sure that this is done efficiently with as small external fragmentation as possible and in a timely manner.

Implementation

Helper functions and definitions

First part of the implementation consist of mostly helper functions and definitions of constants. This is always a good way of implementing abstraction and modularity. Replacing blocks of code with simple and suggestive words that represent functions is always helpful for future simplicity. Besides the ones that I have already been asked for, I chose to implement a few more for displaying specific parts of the output, in order to make it easy to follow such an abstract and detailed subject.

```
#define TRUE 1
#define FALSE 0
#define HEAD sizeof(struct head)

#define MIN(size) (((size) > (16)) ? (size):(16))
#define LIMIT(size) (MIN(0) + HEAD + size)
#define MAGIC(memory) ((struct head*)memory - 1)
#define HIDE(block) (void*)((struct head*)block + 1)

#define ALIGN 8
#define ARENA (64*1024)

#define PRINT_FREE 0
#define PRINT_USED 0
```

Beside this, another helper function that would sit inside of the “dlmall.c” file that contains the malloc implementation is “print_flist()”. This goes through the double link list of free blocks and prints them. Helps debugging.

```
void print_flist(){
  int i = 0;
  struct head *n = flist;
  printf("Block␣at␣position␣%d␣has␣size␣%d␣and␣it
␣␣located␣at:␣%p\n", i, n->size,n);
  n = n->next;
  while(n != flist){
    i++;
    sleep(1);
    printf("Block␣at␣position␣%d␣has␣size␣%d␣and
␣␣␣␣is␣located␣at:␣%p\n", i, n->size,n);
    n = n->next;
  }
}
```

Data structures

In order to talk about the malloc() (here called “dalloc()”), I would need to describe how it works: there is a list of free blocks (initiated with only one big block - here called “arena”) in the form of a double linked list. Whenever a user requests a block, this list is consulted and a block from this list is chosen and given to the user.

```
struct head *find(int size){
  struct head *first = flist;
  if(first->size >= size){
    if(first->size > LIMIT(size)){
      return split(first, size);
    }
    detach(first);
    first->free = FALSE;
    struct head *aft = after(first);
    aft->bfree = FALSE;
    return first;
  }
  else {
    struct head *n = first->next;
    while(n != flist){
      if(n->size >= size){
        if(n->size > LIMIT(size)){
          return split(n,size);
        }
        struct head *aft = after(n);
        aft->bfree = FALSE;
```

```
            n->free = FALSE;
            detach(n);
            return n;
        }
        n = n->next;
    }
    return NULL;
  }
}
```

Before and After

If it is not obvious by now, giving away a block is not efficient at all, and it might fail at the second request with the implementation described above. But instead, the block is chopped into a piece that is required by the user and that piece is given to the user instead of the initial piece (that might produce internal fragmentation otherwise).

Now that you have a general image of what is going on, let us get to the more complicated (or rather abstract) concepts: the headers and the orders of the blocks. Headers are used by my implementation of "dalloc()" in order to easily manipulate and always have relevant information on either given or received blocks. They contain information about the status of the block (used or unused), the size of the current block (in Bytes), status of the before block, size of the before block as well as two pointers for the free blocks list. The reason I underlined "before" in the previous paragraph will be more clear now. There is a concept that is initially hard to grasp in this implementation, and that is spaciality. There are two data structures that handle "blocks" in here, but the difference is that one handles **ALL** the blocks and the other only handles **free** blocks. Why is that? Memory is laid down in a sequential manner inside the process' user-space. When I say that I give out blocks to the user with our function, I mean that I give parts of the segment that I have requested from the Operating System at the initiation of the process using **mmap()**. All these are laid in (as mentioned before) a sequential order. The "free" blocks however, are order in their receiving order and placed in a double circular linked list. The most confusing part of this assignment was exactly this part: understanding that when talking about a "previous" block in the linked list, that does not mean that that block is also "previous" in the memory layout.

```
// BEFORE AND AFTER FOR MEMORY LAYOUT
struct head *after(struct head *block){
  return (struct head*)((char*)block + HEAD +
  block->size);
}

struct head *before(struct head *block){
  return (struct head*)((char*)block
  - block->bsize - HEAD);
}

// BEFORE AND AFTER FOR LINKED LIST
// these functions show how adding or removing
// something from the linked list was done
void detach(struct head *block){
  if(block->next == NULL || block->prev == NULL){
    printf("block is not tied in the list\n");
    exit(1);
  }
  struct head *next_blk = block->next;
  struct head *prev_blk = block->prev;
  prev_blk->next = next_blk;
  next_blk->prev = prev_blk;
  if(flist == block){
    flist = next_blk;
  }
}

void insert(struct head *block){
  struct head *prev_blk = flist->prev;
  block->next = flist;
  block->prev = flist->prev;
  prev_blk->next = block;
  flist->prev = block;
  flist = block;
}
```

Merging/Coalescing

The last problem here was to concatenate blocks of free space back together. But “Why?” are you asking. Imagine that we have 6000 bytes and we are asked for a 1 byte block for 5000 times and then they are all freed. In our list of free blocks, we now have a 1000 bytes block and 5000 blocks of 1 byte. But what if we are asked for a 2000 bytes block? We can’t offer one as we don’t have any, and here coalescing comes in place.
Whenever the user returns a block of memory, we try to “glue” it back together to his neighbours, but remember, this neighbours are not neighbours in the linked list, but instead they must be neighbours in the actual memory layout.

```
struct head *merge(struct head *block){
  struct head *aft = after(block);
  struct head *bfr = before(block);
  int new_size;
  if(bfr != NULL && block->bfree){
    detach(bfr);
    new_size = HEAD + block->size + block->bsize;
    block = bfr;
    block->size = new_size;
  }
  if(aft != NULL && aft->free){
    detach(aft);
    new_size = HEAD + block->size + aft->size;
    block->size = new_size;
    block->free = TRUE;
  }
  return block;
}
```

Size of the head

Now talking about optimization, the first thing that I tried to tackle was the size of the head structure that came with every given/received block. As we wanted to be easy to access and manipulate blocks, a lot of information was needed about every block. All this made up to 24 bytes of data, for each header, but it turns out we only use parts of the data when we allocate memory and some other parts when we free it, thus making it a little more modular would come in handy!

This lead me to making the header NOT contain information about the "next" and "prev" pointers to other blocks in the list of free blocks UNLESS I needed to add one in the list.

Some typecasting away, the header of all blocks has been changed to only include flags for size, status, size of the before block and status of the before flag. This has reduced quite a significant bit of memory usage, but we will discuss more about this in the benchmarks and results section.

Order

Ordering the list of free blocks was another form of optimization. How did it help? Besides time and fragmentation issues, nothing much, but I feel like those are important enough. How does it work? When the user asks for some space, the start of the free-blocks list is traversed. As soon as a fitting block is found, it is offered to the user, but if that block is larger than what has been asked, it is split in a smaller block.

Now we get back to the previous problem with merging: imagine blocks are now given from different parts of the memory and big blocks are split in pieces to be given to the user. We can end up in the same situation where the blocks are small and sparse across the memory layout. Here, sorting comes in and helps! Giving the BEST fitting block to whatever is asked is one issue that is solved by ordering the list. We will also discuss more about this in the benchmarks. (spoiler: it was hard to benchmark this!)

More lists

When talking about lots of memory and lots of requests, things can get complicated. Let us take the simple example where the free-block list is ordered and there are 2000 entries in the linked list. If we want a block that is large enough to be located somewhere towards the end, we will have to traverse through the whole list in order to get there. So? You see where things get complicated? Benchmarks will show the results!

Results

Benchmarks

Benchmarks and testing in general was one of the trickiest parts of this assignment. The metrics that I chose were not that hard to pick: time, space used and fragmentation - by fragmentation I mean the number of free blocks that are left at each step.

```
void *test;
for(int i = 0; i < LOOPS; i++){
  for(int j = 0; j < NR_BLOCKS; j++){
    test = dalloc(rand()%MAX_SIZE);
    if(j%3 == 0){
      void *test2 = dalloc(rand()%MAX_SIZE);
      void *test3 = dalloc(rand()%MAX_SIZE);
      void *test4 = dalloc(rand()%MAX_SIZE);
      dfree(test);
    }
  }
  printf("Iteration %d\n", i);
  print_flist();
}
```

Finding the right type of usage (as closely as possible to normal usage), while also exploiting edge-cases for different optimization sections was the hardest part in here. Under normal conditions: 10 LOOPS with 100 blocks each loop and the maximum (randomly chosen) size of each block being 400 bytes, the operations had no problem and the whole program ran for 0.032 seconds and worst fragmentation was having 11 blocks, which finally at the last iteration ended up being only 1 big block and a total of 7152 bytes of wasted space on headers, which is clearly not good.
The location of the arena: 0x7f95381de000

```
Iteration 0
Position 0 has size 336 and it located at: 0x7f95381e4668
Position 1 has size 32 and is located at: 0x7f95381e4aa8
Position 2 has size 104 and is located at: 0x7f95381e4d58
Position 3 has size 24 and is located at: 0x7f95381e5968
Position 4 has size 48 and is located at: 0x7f95381e8790
Position 5 has size 88 and is located at: 0x7f95381e86e8
Position 6 has size 40 and is located at: 0x7f95381e95c0
Position 7 has size 88 and is located at: 0x7f95381ea0f8
Position 8 has size 48 and is located at: 0x7f95381ea910
Position 9 has size 32 and is located at: 0x7f95381ecda8
Position 10 has size 40 and is located at: 0x7f95381ecff0
Position 11 has size 25424, located at: 0x7f95381de000
Iteration 1
Position 0 has size 24 and it located at: 0x7f95381e3a30
Position 1 has size 24 and is located at: 0x7f95381e5968
Position 2 has size 32 and is located at: 0x7f95381ecda8
Position 3 has size 40 and is located at: 0x7f95381ecff0
Iteration 2
Position 0 has size 24 and it located at: 0x7f95381e5968
Iteration 3
Position 0 has size 24 and it located at: 0x7f95381e5968
Iteration 4
Position 0 has size 24 and it located at: 0x7f95381e5968
Iteration 5
Position 0 has size 24 and it located at: 0x7f95381e5968
Iteration 6
Position 0 has size 24 and it located at: 0x7f95381e5968
Iteration 7
Position 0 has size 24 and it located at: 0x7f95381e5968
Iteration 8
Position 0 has size 24 and it located at: 0x7f95381e5968
Iteration 9
Position 0 has size 24 and it located at: 0x7f95381e5968
```

However, chaging the header size to not fit in the pointers to next and previous blocks helped significantly in the amount of space for headers. This second run using this smaller headers only used 2424 bytes of headers memory.

```
The location of the arena: 0x7f3da2797000
Headers used 2424 bytes of memory in this run.
```

Finally, the last optimization issue was to deal with ordering the list of freed blocks. This was also tricky to benchmark as I needed to increase the amount of memory in the arena to get a observable difference in time when searching for the right block to be allocated.

Time comparison for similar tests (10 000 loops of 10000 blocks of max size 40 000):
Without ordering: 3.45 seconds
With ordering: 1.78 seconds
This shows a definite improvement overall in time using an ordered list of freed blocks!

Conclusion

To sum up, this project helped me better understand how **malloc()** and **free()** implementations work. Understanding what is going on under the abstraction layer that C provides helped me better use this functions. Implementing optimization features on these functions was not an easy task, but even harder was testing and benchmarking them.