Seminar 2 Making malloc() and free() functions

Amirhossein Namazi
<anamazi@kth.se>
KTH Royal Institute of Technology

November 23, 2020

Introduction

The main purpose of this seminar was to understand how the **malloc()** and **free()** functions are actually implemented. How often do we make system calls and how our functions handle blocks of memory were the important parts of this seminar.

First Implementation

First part of the seminar consists of mostly helper functions and definitions of constants. What I needed to do was to follow the instruction and general strategy and make my own preliminary implementation. After implementing the functions, it was time to run a simple test to see if everything was working as intended. Most of the programming assignments do not work in the first run and this was not an exception. In the terminal, I received the Segmentation fault (core dumped) error. The next step was to figure out what is causing the problem so I started using some **printf()** statements for debugging purposes. I previously had my free list of blocks implemented as a double linked list. Then I decided it would make the life easier if I implemented it as a circular double linked list and I did so. After a few hours of debugging I manged to have the first implementation up and running.

One of the functions I created was the function that prints the blocks in the **flist**.

```
void printFlist(){
  int i = 0;
  struct head *first = flist;
  printf("First block ==> position %d size %d location %p\n", i,
  first->size, first);
  first = first-> next;
  while(first != flist){
    i++;
    printf("    Block: position %d size %d location %p\n", i,
    first->size, first);
    first = first->next;
```

```
    }
}
```

In addition to this I created another function that prints the total space used by the headers. I have a global variable that keeps track of the total space used by the headers and update it in functions such as **split()** and **merge()**(which we will get to in a few lines.)

Coalescing

The next step was to improve our first implementation. In the first implementation when ever we were asked to hand out a piece of memory to the user, we would split the memory into smaller block. After freeing that block, we would insert it into a list. After a few iterations we would end up with a lot of small blocks in that list and this would result in external fragmentation and causing us to lose a lot of memory. The solution is to Merge the blocks. We would do so by checking if the blocks before or after the block in question is free. If this is the case we will merge the blocks and fix the pointers, flags and size. One should note that by blocks before and after we do not mean the blocks in the **flist** but rather the neighbouring blocks in the actual memory layout.

```
struct head *merge( struct head *block){
    struct head *aft = after(block);
    struct head *bfr = before(block);
    int size = 0;

    if(block -> bfree){
      space -= HEAD;
      detach(bfr);
      size = (block -> bsize) + HEAD + block -> size ;
      block = bfr;
      block -> size = size;
    }

    if(aft -> free){
      space += HEAD;
      detach(aft);
      size = block -> size + HEAD + aft -> size ;
      block -> size = size ;
      block -> free = TRUE;
    }
  return block;
}
```

Optimization

The next step was to improve the implementation even further. We had the ability to choose between different suggestions and I decided to focus on the size of the headers.

Size of the head

In the first implementation we have a header of size 24 bytes for every block. If we re-think this design we figure out that two of the pointers in the current **head** structure that are ***next** and ***prev** are only needed when we want to link a block in the **flist**. In other words, if a block is not free or rather "taken" we can omit those two pointers and that would save us 16 bytes.

Let's name this structure **taken**. A **taken** will have a size of 8 bytes which is a really good improvement in terms of memory wasted on the headers. We should modify our previous implementation. We need to define a new structure as stated before, change the **HEAD**, **MIN()**, **MAGIC()** and **HIDE()** macros. A **taken** would be as follows:

```
struct taken {
  uint16_t bsize;
  uint16_t bfree;
  uint16_t size;
  uint16_t free;
};
```

and our macros as:

```
#define HEAD ( sizeof ( struct taken ) )
#define MIN(size) (((size) > (16))? (size) : (16))
#define MAGIC(memory) ( ( struct taken*)memory - 1 )
#define HIDE(block) ( void *) ( ( struct taken* ) block + 1 )
```

Benchmark

Benchmarks and testing was a difficult and tricky part of this assignment. Below I have the program I used for the testing.

```
int main(){

  void *memory;
  for(int i = 0; i < LOOPS; i++){
    for(int j = 0; j < BLOCKS; j++){
      memory = dalloc(rand()%MAX_SIZE);
      if(j%3 == 0){
```

```
            void *memory2 = dalloc(rand()%MAX_SIZE);
            void *memory3 = dalloc(rand()%MAX_SIZE);
            dfree(memory);
          }
      }
      printf("End of iteration i %d \n", i);
      printFlist();
    }
  printSpace();
return 0;
}
```

In my tests I used 10 LOOPS with 100 BLOCKS each loop and the maximum (randomly chosen) size of 500 bytes for each Block.

The first test I ran was on the first implementation which is without the **merge()**. There is 6240 bytes wasted memory space on headers and there is clearly external fragmentation since there are 33 free blocks at some point of the test.

The next test was to run the same test on the improveed version with **merge()**. There is 6144 bytes wasted memory on the headers in this case which is a little improvement but not much. As of fragmentation, the worst case is 28 free blocks.

The location of the arena: 0x7f68c9310000
made an arena with size 65488 for user
End of iteration i 0
First block ===¿ position 0 size 352 location 0x7f68c9316270
Block:position 1 size 128 location 0x7f68c9316888
Block:position 2 size 168 location 0x7f68c9316f38
Block:position 3 size 96 location 0x7f68c93172d0
Block:position 4 size 40 location 0x7f68c9317ef8
Block:position 5 size 72 location 0x7f68c9318668
Block:position 6 size 32 location 0x7f68c9318a90
Block:position 7 size 16 location 0x7f68c93197a0
Block:position 8 size 48 location 0x7f68c931a3e0
Block:position 9 size 40 location 0x7f68c931a320
Block:position 10 size 16 location 0x7f68c931b768
Block:position 11 size 56 location 0x7f68c931bc60
Block:position 12 size 8 location 0x7f68c931c780
Block:position 13 size 24 location 0x7f68c931c650
Block:position 14 size 64 location 0x7f68c931cc00
Block:position 15 size 24952 location 0x7f68c9310000
Block:position 16 size 56 location 0x7f68c931dce8
Block:position 17 size 136 location 0x7f68c931e2c0
Block:position 18 size 24 location 0x7f68c931eed0
Block:position 19 size 32 location 0x7f68c931f860
Block:position 20 size 64 location 0x7f68c931fe50
End of iteration i 1
First block ===¿ position 0 size 64 location 0x7f68c9315c58
Block:position 1 size 24 location 0x7f68c9310310

Block:position 2 size 16 location 0x7f68c9310b38
Block:position 3 size 80 location 0x7f68c9310000
Block:position 4 size 32 location 0x7f68c9311528
Block:position 5 size 16 location 0x7f68c9312760
Block:position 6 size 32 location 0x7f68c93129b8
Block:position 7 size 72 location 0x7f68c93135a0
Block:position 8 size 32 location 0x7f68c9314680
Block:position 9 size 32 location 0x7f68c9314a70
Block:position 10 size 72 location 0x7f68c9315638
Block:position 11 size 40 location 0x7f68c93163a8
Block:position 12 size 24 location 0x7f68c9316f38
Block:position 13 size 40 location 0x7f68c9317ef8
Block:position 14 size 72 location 0x7f68c9318668
Block:position 15 size 32 location 0x7f68c9318a90
Block:position 16 size 16 location 0x7f68c93197a0
Block:position 17 size 48 location 0x7f68c931a3e0
Block:position 18 size 40 location 0x7f68c931a320
Block:position 19 size 16 location 0x7f68c931b768
Block:position 20 size 56 location 0x7f68c931bc60
Block:position 21 size 8 location 0x7f68c931c780
Block:position 22 size 24 location 0x7f68c931c650
Block:position 23 size 64 location 0x7f68c931cc00
Block:position 24 size 56 location 0x7f68c931dce8
Block:position 25 size 24 location 0x7f68c931eed0
Block:position 26 size 32 location 0x7f68c931f860
Block:position 27 size 64 location 0x7f68c931fe50
End of iteration i 2
First block ===¿ position 0 size 64 location 0x7f68c931fe50
Block:position 1 size 16 location 0x7f68c9315638
Block:position 2 size 16 location 0x7f68c9310b38
Block:position 3 size 16 location 0x7f68c9312760
Block:position 4 size 32 location 0x7f68c9314a70
Block:position 5 size 24 location 0x7f68c9316f38
Block:position 6 size 32 location 0x7f68c9318a90
Block:position 7 size 16 location 0x7f68c93197a0
Block:position 8 size 40 location 0x7f68c931a320
Block:position 9 size 16 location 0x7f68c931b768
Block:position 10 size 8 location 0x7f68c931c780
Block:position 11 size 24 location 0x7f68c931c650
Block:position 12 size 24 location 0x7f68c931eed0
Block:position 13 size 32 location 0x7f68c931f860
End of iteration i 3
First block ===¿ position 0 size 16 location 0x7f68c9312760
Block:position 1 size 16 location 0x7f68c93197a0
Block:position 2 size 16 location 0x7f68c931b768
Block:position 3 size 8 location 0x7f68c931c780
End of iteration i 4
First block ===¿ position 0 size 8 location 0x7f68c931c780
End of iteration i 5
First block ===¿ position 0 size 8 location 0x7f68c931c780

End of iteration i 6
First block ===¿ position 0 size 8 location 0x7f68c931c780
End of iteration i 7
First block ===¿ position 0 size 8 location 0x7f68c931c780
End of iteration i 8
First block ===¿ position 0 size 8 location 0x7f68c931c780
End of iteration i 9
First block ===¿ position 0 size 8 location 0x7f68c931c780
space for HEADERS 6144

The next test was on the optimized version which clearly would improve the efficiency of our implementation.

The location of the arena: 0x7fec9eaf0000

space for HEADERS 2152

This shows that there is a significant improvement compared to the previous test regarding the wasted memory on headers.

With the current measures both test 2 and test 3 take same amount of time to execute which is around 0.097s.

Conclusion

This assignment helped me understand how **malloc()** and **free()** actually work. Debugging for this assignment was a tricky part and I believe benchmarking and testing was even harder.