
YTU Computer Engineering

# Malloc, Realloc & Free Implementations

## Operating Systems – Homework 2

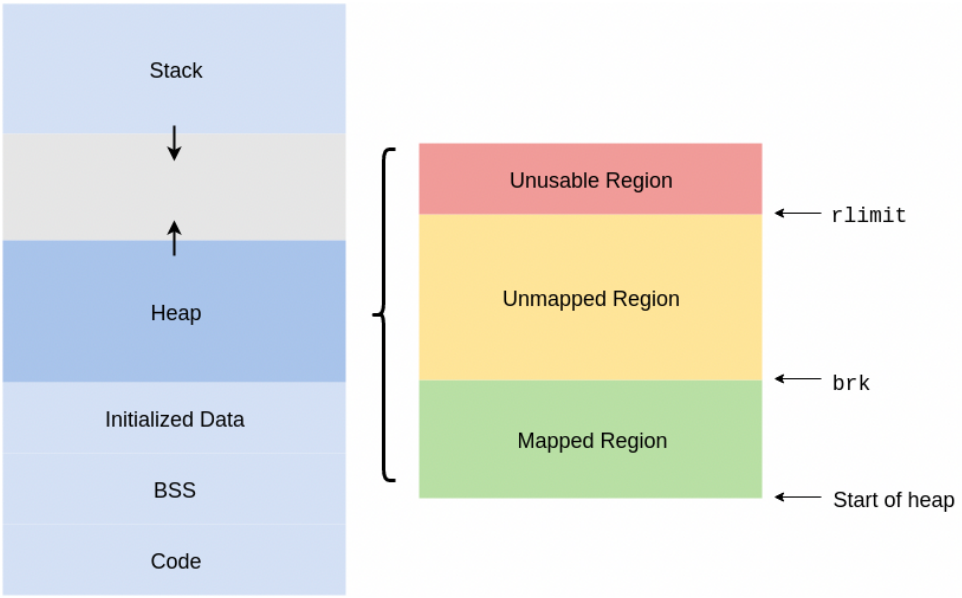
Student Name: Berk Sudan
Student ID: 16011601

Submission Date: 29/12/2017

# The Intro

We use malloc, realloc and free functions in <stdlib.h> C library everywhere when we write C code. They are, simply, used in order to partition the memory and give the appropriate place which we can use. We use these functions but don't know what's inside them and what kind of process included in these functions. The purpose of this homework is to find out how memory allocation transactions really work.

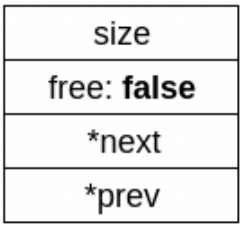
When some process running, operating system gives user an area to use. But operating system doesn't define what kind of data structure will be used in this area or what kind of find algorithm will be managed. This area is so called **heap**. Each process has its own virtual address space and these virtual adresses linked to physical memory. Operation system links the virtual addresses to physical adresses. Heap area is not monolithic and has 3 parts: unusable region, unmapped region and mapped region. Heap and the other parts of the memory is shown below:




Malloc, realloc and free functions use the mapped region and move the break(brk) appropriately in the mapped region expansion.

## Data Structure

In this implementation, we use classic linked list data structure. In linked list data structure each element points to the next element.  If we use double linked list then we can imply that each element points to the previous element. Also each element must have the *free* parameter that indicates whether this element is free or not. A free element doesn't mean it is not there or there is nothing in this area. A struct can be easily build to have these values. Thus we can build a header structure shown in below:




```c
typedef struct MEM_Block *Block_Ptr;
struct MEM_Block {
  size_t size;
  int free;
  Block_Ptr next;
  Block_Ptr prev;
};
```

# The Implementation

The struct(as data structure) is defined in the previous section. We have 3 functions to implement: malloc, realloc and free. The implementation of these functions and which operations they contain is listed below.

- **void *mm_malloc(size_t size)**
  - If size is equal to zero return NULL.
  - Align the requested size to four.
  - If root(the first element) is initialized:
    - Find a free block with **first fit algorithm**
    - If we found one:
      - If this block's size >= requested size + block size + 4(Minimum Unit Size)
        - Split the block
      - Make the free parameter as 0.
    - If not found:
      - Create more space and a block
      - If more space cannot be created return NULL.
  - If root(the first element) is NOT initialized:
      - Create more space and a block
      - If more space cannot be created return NULL.
      - Make new block's previous parameter NULL
      - Assign new block's address to the root address.
  - Fill the block with zero.
  - Return header address + block size (i.e. the address of the chunk of the free memory )
- **void *mm_realloc(void *ptr, size_t size)**
  - If ptr is NULL:
    - Call mm_malloc(size).
    - Return the return value of the mm_malloc.
  - If ptr is not NULL and size = 0 :
    - Call mm_free(ptr).
    - Return NULL.
  - Align the requested size to four.
  - If block's size >= requested size:
    - If this block can be split, split in two.
    - Return the parameter ptr.
  - mem_block->size < requested size:
    - If next block is not null and next block is free:
      - if current block size  + next block size + block size >= requested size:
        - Fill the next block with zero.
        - Coalesce with next free block.
        - If this block's size >= requested size + block size + 4(Minimum Unit Size)
          - Split the block.
    - If next block is  null or next block is not free:
      - Call mm_malloc(size), If mm_malloc return value is NULL, then return NULL
      - Get reallocated block address.
      - Call mm_free(ptr) in order to free the old block
      - Call memcpy(adr_reallocedBlock, ptr, oldSize) and return the  adress of the reallocated block's data.

- **void mm_free(void *ptr)**
  - If ptr is NULL:
    - Do nothing.
  - Get block address.
  - Mark block's free as 1.
  - If previous block is free and not NULL :
    - Coalesce previous block with this block.
  - If next block is free and not NULL :
    - Coalesce this block with next block.
  - If next block is NULL:
    - If previous block is NULL:
      - Assign root as NULL(i.e. this is the last block)
    - If previous block is not NULL:
      - Set previous block's next parameter as NULL.
    - Set break to the beginning of the block.

# The Analysis

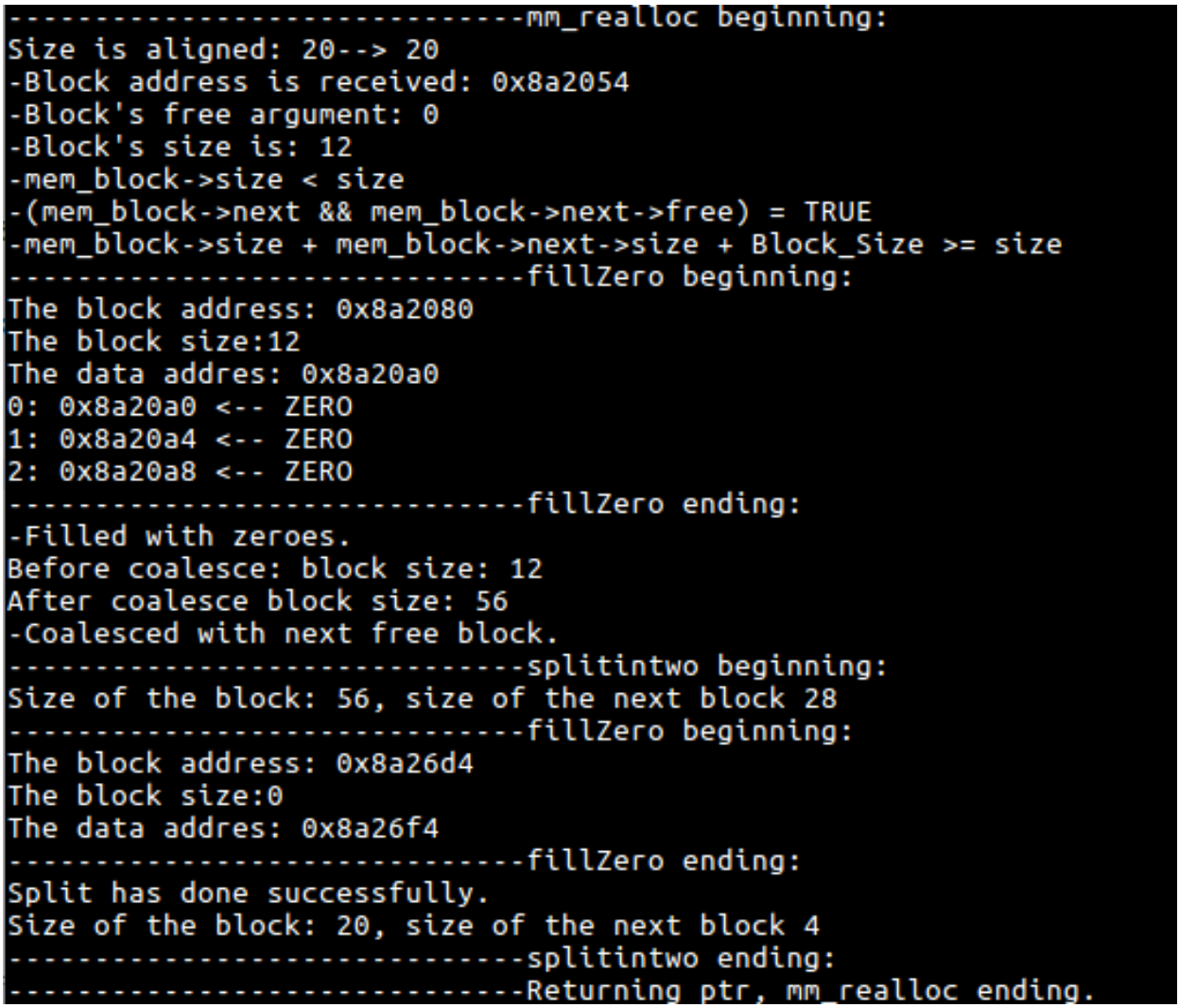
Block size is equal to 32 in decimal and 10H in hexadecimal.

- Test Case1: mm_realloc
  - Test Code

```
int *adata = (int*) mm_malloc(sizeof(int));
long double *bdata = (long double*) mm_malloc(sizeof(long double));
int *cdata = (int*) mm_malloc(sizeof(int) * 3);
int *ddata = (int*) mm_malloc(sizeof(int) * 3);
int *edata = (int*) mm_malloc(sizeof(int) * 7);
int *fdata = (int*) mm_malloc(sizeof(int));

mm_free(ddata);
cdata = (int*) mm_realloc(cdata, sizeof(int) * 5);
```

  - Test Result for mm_realloc function



- Test Case2: mm_realloc
  - Test Code

```
int *adata = (int*) mm_malloc(sizeof(int));
long double *bdata = (long double*) mm_malloc(sizeof(long double));
int *cdata = (int*) mm_malloc(sizeof(int) * 3);
int *ddata = (int*) mm_malloc(sizeof(int));
int *edata = (int*) mm_malloc(sizeof(int) * 7);
int *fdata = (int*) mm_malloc(sizeof(int));

mm_free(ddata);
cdata = (int*) mm_realloc(cdata, sizeof(int) * 5);
```

  - Test Result for mm_realloc function

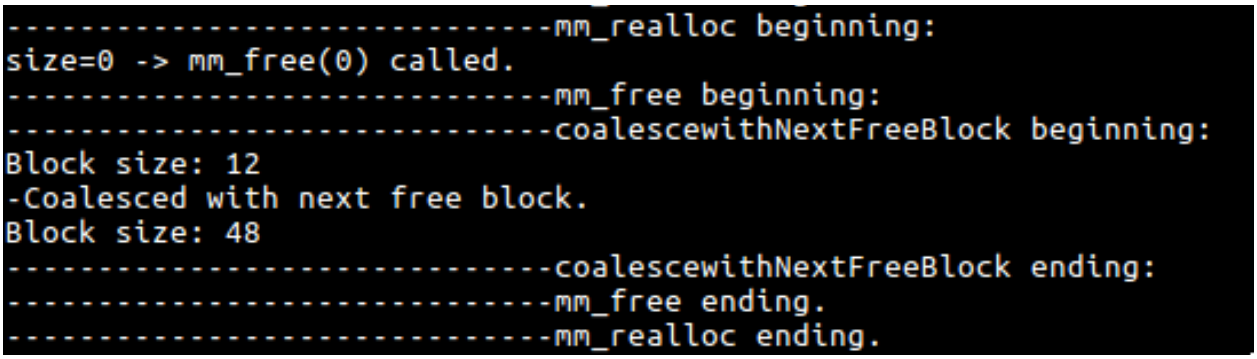
```
------------------------------mm_realloc beginning:
Size is aligned: 20--> 20
-Block address is received: 0xffa054
-Block's free argument: 0
-Block's size is: 12
-mem_block->size < size
-(mem_block->next && mem_block->next->free) = TRUE
-mem_block->size + mem_block->next->size + Block_Size >= size
------------------------------fillZero beginning:
The block address: 0xffa080
The block size:4
The data addres: 0xffa0a0
0: 0xffa0a0 <-- ZERO
------------------------------fillZero ending:
-Filled with zeroes.
------------------------------coalescewithNextFreeBlock beginning:
Block size: 12
-Coalesced with next free block.
Block size: 48
------------------------------coalescewithNextFreeBlock ending:
------------------------------Returning ptr, mm_realloc ending.
```

- Test Case3: mm_realloc
  - Test Code

```
int *adata = (int*) mm_malloc(sizeof(int));
long double *bdata = (long double*) mm_malloc(sizeof(long double));
int *cdata = (int*) mm_malloc(sizeof(int) * 3);
int *ddata = (int*) mm_malloc(sizeof(int));
int *edata = (int*) mm_malloc(sizeof(int) * 7);
int *fdata = (int*) mm_malloc(sizeof(int));

mm_free(ddata);
cdata = (int*) mm_realloc(cdata, 0);
```

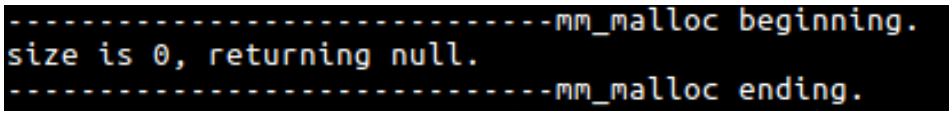
  - Test Result for mm_realloc function




- Test Case4: mm_malloc
  - Test Code

```
int *adata = (int*) mm_malloc(0);
```

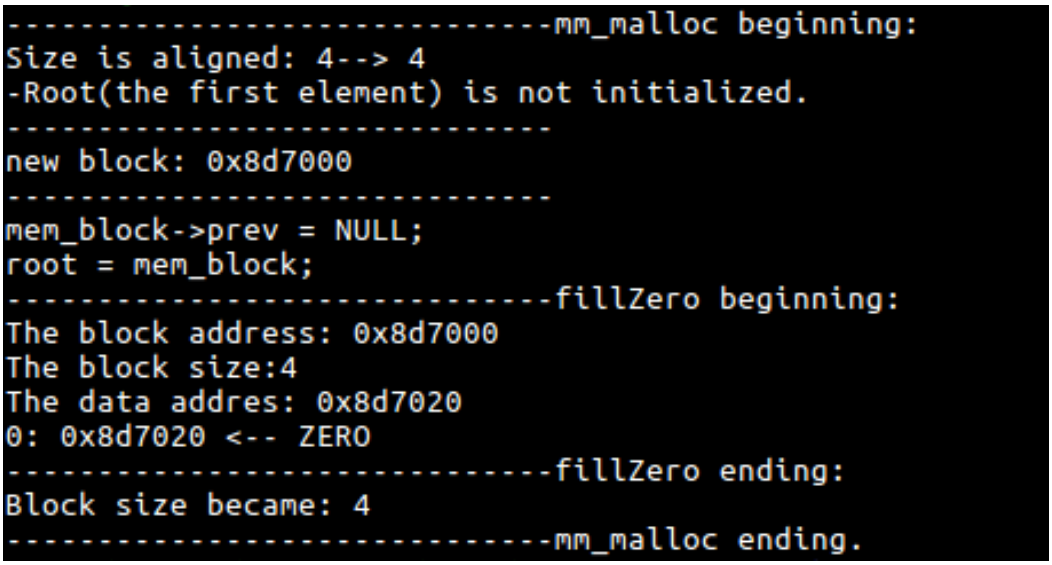
  - Test Result for mm_malloc function



- Test Case5: mm_malloc
  - Test Code

```
int *adata = (int*) mm_malloc(sizeof(int));
```

  - Test Result for mm_malloc function



- Test Case6: mm_free
  - Test Code

```
int *adata = (int*) mm_malloc(sizeof(int));
   long double *bdata = (long double*) mm_malloc(sizeof(long double));
   int *cdata = (int*) mm_malloc(sizeof(int) * 3);
   int *ddata = (int*) mm_malloc(sizeof(int) * 3);

   printf("mm_free(adata):\n");
   mm_free(adata);
   printf("mm_free(cdata):\n");
   mm_free(cdata);
   printf("mm_free(bdata):\n");
   mm_free(bdata);
   printf("mm_free(ddata):\n");
   mm_free(bdata);
```

  - Test Result for 4 mm_free functions

```
mm_free(adata):
------------------------------mm_free beginning:
Getting block address
mem_block->free <- 1
mem_block->next is not NULL
------------------------------mm_free ending.
mm_free(cdata):
------------------------------mm_free beginning:
Getting block address
mem_block->free <- 1
mem_block->next is not NULL
------------------------------mm_free ending.
mm_free(bdata):
------------------------------mm_free beginning:
Getting block address
mem_block->free <- 1
------------------------------coalescewithNextFreeBlock beginning:
Block size: 16
-Coalesced with next free block.
Block size: 60
------------------------------coalescewithNextFreeBlock ending:
------------------------------coalescewithNextFreeBlock beginning:
Block size: 4
-Coalesced with next free block.
Block size: 96
------------------------------coalescewithNextFreeBlock ending:
mem_block->next is not NULL
------------------------------mm_free ending.
mm_free(ddata):
------------------------------mm_free beginning:
Getting block address
mem_block->free <- 1
------------------------------coalescewithNextFreeBlock beginning:
Block size: 96
-Coalesced with next free block.
Block size: 96
------------------------------coalescewithNextFreeBlock ending:
mem_block->next is not NULL
------------------------------mm_free ending.
```

# The Conclusion

Memory allocation functions malloc, realloc, free can be implemented easily. There's lots of way to do it but one of the simplest method is using linked list as data structure and using first fitting algorithm for finding process. These functions seem very complicated at the first glance but they are not complicated at all and can be implemented by anyone who knows C-language.

In the solution we use headers to identify the data stored in the specific address(es). Headers must be invisible to the user who uses these function. It means when user uses mm_realloc or mm_malloc the returned address must show the data address but not the head. So user may not think there is header before or not. User just requests the size and gets what s/he wants. Abstraction is the key of the solution. On the other hand, mm_free function doesn't release the memory except the specific cases(e.g. the block is the last block or there is just 1 block left). mm_free function's main job is mark the related block's free parameter as 1. So, it makes this data overwritable. When first fit algorithm trying to find the first appropriate data, it looks if free parameter of a block is zero or one. If free parameter is one then new data can be written the data of this block. As result, free function doesn't really release resource.

Hence, we accomplished and solved the problem.