Memory Management Project

Goal: Implement Malloc/Free functions in Linux environment

Criteria

1- Use C language
2- No memory leaks tested on valgrind
3- Use doubly linked list
4- No error(s) tested on g++ -Wextra -Wall -Werror
5- Obviously, use of malloc/free is not allowed

Why this project?

Memory management is “the must-know” subject for all programmer.

Bad memory management means bad program/programmer.

Tools/functions used

IDE: CLION

Libraries:

1- stdbool.h for to check weather a block is free or not
2- zconf.h for use of sbrk function
3- stdlib.h for use of NULL and size_t

Compiling/git related:

1- g++ with -Wextra -Wall -Werror flags
2- Makefile(command) Cmake(Clion)
3- .gitignore

Struct used:

```
typedef struct s_block t_block;

struct s_block
{
    size_t size;
    t_block* next_block;
    t_block* prev_block;
    void* data;
    bool free;
};
```

1- size_t size = size of memory used.
2- t_block* next_block/prev_block = Doubly linked list!
3- void* data = this is where we store the data, it's void pointer because we don't know which type will be used.
4- bool free = check weather a block is freed or not

Chapter 1: small story

1- You need space? SBRK PLEASE!

"The sbrk subroutine adds to the program break value the number of bytes contained in the increment parameter and changes the amount of available space accordingly"[1]

Let's say you want to open a parking lot. You have prepared all the paper works, the money, the business name, etc. You are on your way to meet the city service for the final approval. You finally meet the city service and explain your goals, what you want to do and how you want to start your business. After hearing all about your project, the city service asks you this question: "I understand what you want to do and how you want to do it, but sir, you don't have any land to start your business"

You realized that, first, you need to burrow the land (Purchase more likely but think about the context please…) from the city.

We need to apply the same concept to "burrow" the memory from the computer. "I want to do something with the program so, would you kindly burrow some of your memory?"

2- SBRK and HEAP!

Let's go back to your parking lot business.

So, you decided to burrow some land from the city and the city asks you to specify the size that you would want to burrow. Let's say you asked for size of 100 smalls cars. The city service checks if there is enough of space for the asked size and luckily they found it. "0 ~ 200 Heap street will be burrowed to you" replied the city.

```
void *sbrk(intptr_t increment);
```

You input how much of memory you would want to borrow from the computer.

When you use SBRK function to allocate the memory, the return value will be the 'address" of memory(heap).

If the computer cannot give you any memory, it will return -1.

Chapter 2: MALLOC

```
//Malloc_perso
void* allocate_memory(size_t p_size);
```

In my malloc implementation, the use is as same as the regular malloc.

```
void* allocate_memory(size_t p_size)
{
    if(p_size == 0)
        return NULL;

    t_block* prev = NULL;
    t_block* temp = NULL;
    size_t aligned_size = align(p_size);

    if(!first_block_address)
    {
        temp = extend_heap(aligned_size);
        first_block_address = temp;
    }
    else if(first_block_address)
    {
        prev = (t_block*)first_block_address;
        temp = find_block(p_size);
        if(temp)
        {
            if((temp->size - aligned_size) >= (sizeof(t_block) + sizeof(size_t)))
            {
                split_block(temp, aligned_size);
            }
        }
        else
        {
            while(prev->next_block != NULL)
            {
                prev = prev->next_block;
            }
            temp = extend_heap(aligned_size);
            prev->next_block = temp;
            temp->prev_block = prev;
        }
        temp->free = false;
    }
    return temp->data;
}
```

If allocating size is 0, the address should return NULL. (burrowing 0 land = No land)

Since the memory space is allocated by multiple of 4(in 32x) or 8(in 64x), I align the size that I want to allocate.

```
size_t align(size_t p_size)
{
    if(p_size == 0)
        return 0;

    return ((p_size + (sizeof(size_t) - 1)) & ~(sizeof(size_t) - 1));
}
```

If the size is smaller than 4 (or 8), aligned size will be 4 or 8. If the size of bigger than 4 (or 8), aligned size will be 8 (or 16). And on and on. With this aligned size, we check whether it's the first time malloc is called.

CASE 1: First time Malloc

```
if(!first_block_address)
{
    temp = extend_heap(aligned_size);
    first_block_address = temp;
}
```

If the first block address doesn't exist, then logically, the memory was never allocated. So, we allocate the memory using extend_heap function.

```
t_block* extend_heap(size_t p_size)
{
    if(p_size == 0)
        return NULL;

    t_block* temp = NULL;
    temp = (t_block*)sbrk(0);
    if(sbrk(p_size + sizeof(t_block)) == (void*) -1)
        return NULL;

    initialize_block(temp);
    temp->size = p_size;
    temp->data = temp + 1;
    return temp;
}
```

We call sbrk (0) to return "the end" of the address of memory that is allocated until now (we call this a " break").

We extend the address of the break to aligned size + the meta-data size (t_block in this case). If memory is allocated, we set the size of block to aligned size, data to the address after the meta_data and return the allocated block.

We pass set this block's address as the first_block_adress so that we can keep the track of heap address for further use or malloc fuction.

CASE 2: !First time Malloc

```
else if(first_block_address)
{
    prev = (t_block*)first_block_address;
    temp = find_block(p_size);
    if(temp)
    {
        if((temp->size - aligned_size) >= (sizeof(t_block) + sizeof(size_t)))
        {
            split_block(temp, aligned_size);
        }
    }
    else
    {
        while(prev->next_block != NULL)
        {
            prev = prev->next_block;
        }
        temp = extend_heap(aligned_size);
        prev->next_block = temp;
        temp->prev_block = prev;
    }
    temp->free = false;
}
return temp->data;
```

What if the first_block_address already exists? This means the malloc function is already called before (once or many, who knows, it's not important). First, we need to check if there is a free block available.

If there is a free block, we need to check whether asked size will fit into this free block.

```
t_block* find_block(size_t p_size)
{
    t_block* temp = (t_block*)first_block_address;
    while(temp)
    {
        if(temp->free && temp->size >= p_size)
        {
            return temp;
        }
        temp = temp->next_block;
    }
    return NULL;
}
```

Even if there is a free block available, if the size doesn't fit, you need to burrow the memory again (using extend heap).

CASE 2a: Size does fit

```
if(temp)
{
    if((temp->size - aligned_size) >= (sizeof(t_block) + sizeof(size_t)))
    {
        split_block(temp, aligned_size);
    }
}
```

If the size does fit, we can reuse the free block. But, what if you don't need all the space? If the space is too big, what will you do?

In programming, optimizing the memory space is critical. You are not allowed (you are allowed but… you know what I mean) to have more space than you needed. To solve this issue, we should split the block into two and keep only the needed space.

```
void split_block(t_block* p_block, size_t p_size)
{
    if(!p_block || p_size == 0)
        return;

    t_block* to_split = NULL;
    to_split = (t_block*)((char*)p_block->data + p_size);
    to_split->size = p_block->size - p_size - sizeof(t_block);
    to_split->next_block = p_block->next_block;
    to_split->prev_block = p_block;
    to_split->free = true;
    to_split->data = to_split + 1;
    p_block->size = p_size;
    p_block->next_block = to_split;

    if(to_split->next_block)
        to_split->next_block->prev_block = to_split;
}
```

First, check if the split size is 0 or t_block pointer is NULL. If it's one of those two cases, you cannot split the block.

If you passed the safety test, first get the address of the "spot" where you need to split. Original block's size will be reduced to the size needed, the new block will keep the rest of the size minus the size of meta-data. Since we use doubly linked list, we must insert the new block in between the original block and next block of the original one.

If the next block exists, we link the next block to the new one too.

CASE 2b: Size doesn't fit

```
else
{
    while(prev->next_block != NULL)
    {
        prev = prev->next_block;
    }
    temp = extend_heap(aligned_size);
    prev->next_block = temp;
    temp->prev_block = prev;
}
temp->free = false;
```

If the size doesn't fit, we have no choice but to extend_heap and create another block.

Chapter 3: FREE

```
void free_memory(void* p_address)
{
    if (!p_address)
        return;

    t_block* temp = NULL;
    t_block* free_manager = NULL;
    if(first_block_address)
    {
        if (p_address > first_block_address && p_address < sbrk(0))
        {
            free_manager = (t_block*)((char*) p_address - sizeof(t_block));
            temp = free_manager;
            free_manager->free = true;
            if (free_manager->prev_block && free_manager->prev_block->free)
            {
                free_manager = try_to_fuse(free_manager->prev_block);
            }
            if (free_manager->next_block)
            {
                try_to_fuse(free_manager);
            }
            reset_memory(p_address, temp->size);
        }
    }
}
```

First, we need to if the address that you want to free is pointing at something. If it's NULL, then you cannot free something that doesn't exist. After passing the safety test, check if the first_block_address exists then check if the given address is in between the first_block_address and the break address. If so, the address must be valid and we can process to free it.

Change the status of block to free = true (this can be done later order is not important).

Check if previous block is available, if so, check if it's free. If you fall in this case, you can fuse previous block with the current one. This will optimize the process of malloc in the future. You'll have more chance to reuse the fused free block (it will have bigger size) than recreating the new block and extending the heap.

This optimized the memory management.

```
t_block* try_to_fuse(t_block *p_block)
{
    if(!p_block)
        return NULL;

    if(p_block->next_block && p_block->next_block->free)
    {
        p_block->size += sizeof(t_block) + p_block->next_block->size;
        p_block->next_block = p_block->next_block->next_block;
        if(p_block->next_block)
            p_block->next_block->prev_block = p_block;
    }
    return p_block;
}
```

To fuse the block, you add the size of next block (including meta data size) with the current block. Cut the link of list in the next block and relink with next-next block.

Lastly, you need to reset all the used data to NULL

```
void reset_memory(void* p_address, size_t p_size)
{
    for (unsigned int i = 0; i < (unsigned int)(p_size); ++i)
    {
        ((char*)p_address)[i] = 0;
    }
}
```

We travel the data array from the start to end and fill each case with NULL.

CHAPTER 4: CALLOC/REALLOC

1. Calloc

"Allocates a block of memory for an array of *num* elements, each of them *size* bytes long, and initializes all its bits to zero."[2]

```
6
7  void* c_allocate_memory(size_t p_items, size_t p_size)
8  {
9      if(p_items == 0 || p_size == 0)
10         return NULL;
11
12     void* temp = NULL;
13     temp = allocate_memory(p_items * p_size);
14     if(temp)
15     {
16         size_t align_size = align(p_items * p_size);
17         reset_memory(temp, align_size);
18         return temp;
19     }
20     return NULL;
21 }
```

Multiply the size and the item number.

Align the result.

Malloc with the aligned size and reset the memory, it's this simple.

2. Realloc

"Changes the size of the memory block pointed to by *ptr*.
The function may move the memory block to a new location (whose address is returned by the function)."[3]

Realloc is complicated.

```
void* reallocate_memory(void* p_address, size_t p_size)
{
    if(!p_address)
        return allocate_memory(p_size);

    if(p_address > first_block_address && p_address < sbrk(0))
    {
        size_t aligned_size = align(p_size);
        t_block* resize_block = (t_block*)((char*) p_address - sizeof(t_block));
        if(resize_block->size >= aligned_size)
        {
            if(resize_block->size - aligned_size >= sizeof(t_block) + sizeof(size_t))
            {
                for(size_t i = aligned_size; i < resize_block->size; ++i)
                {
                    ((char*)resize_block->data)[i] = 0;
                }
                split_block(resize_block, aligned_size);
            }
        }
```

First, if the given pointer is NULL, then no memory is allocated. We cannot "re" allocate what is not allocated so we use the malloc.

If the address has something, we need to check if the address is valid (same as Free).

If it's valid, then check if asked size for realloc is smaller or equal to the current size. If so, check whether we can spilt the block. If not, just use the current block since the asked size is not small or big enough to reallocate. If asked size is small enough to split the block, split the block and reset all the data after the asked size.

```
else
{
    if(resize_block->next_block && resize_block->next_block->free
      && (resize_block->size + sizeof(t_block) + resize_block->next_block->size)
       >= aligned_size)
    {
        try_to_fuse(resize_block);
        if(resize_block->size - aligned_size >= (sizeof(t_block) + sizeof(size_t)))
            split_block(resize_block, aligned_size);
    }
```

If the asked size is bigger than the current block's size, check these conditions below

1- Next block exists?
2- Is next block free?
3- Is aligned asked size is smaller or equal to the sum of size of the current block + size of the meta-data + size of the next block.

CASE 1: ALL CONDITIONS ARE TRUE

If all these conditions are true, we can fuse the current block with the next one.

After fusing the blocks, check if you are occupying too much of space, if it's the case, then split the block.

CASE2: IF NOT

```
else
{
    t_block* to_reallocate = NULL;
    void* reallocate_data = allocate_memory(aligned_size);
    if(reallocate_data)
    {
        to_reallocate = (t_block*)((char*)reallocate_data - sizeof(t_block));
        copy_data(resize_block, to_reallocate);
        free_memory(p_address);
        return reallocate_data;
    }
    else
    {
        return NULL;
    }
```

Remember the definition of realloc?

"The function may move the memory block to a new location"

This is what we need to do now. Allocate memory to extend the heap. Copy the data of the original block to the new allocated block and free the original block.

1. https://en.wikipedia.org/wiki/Sbrk
2. http://www.cplusplus.com/reference/cstdlib/calloc/
3. http://www.cplusplus.com/reference/cstdlib/realloc/