COMP3511 Operating System (Spring 2020)

Project 2 – A simplified malloc/free

Release on 15-Apr (Wednesday) Due on ~~06-May~~ 13-May (Wednesday) 23:59

Introduction

The aim of this project is to help students understand **virtual memory management** in Linux operating system. Upon completion of the project, students should be able to write their own memory management functions: `mm_malloc()` and `mm_free()`, and get familiar with a number of related Linux system calls, such as `sbrk()`.

Overview

You need to implement a system program named as `vmm` to simulate a number of memory allocation and deallocation, without using the corresponding C standard library functions (in other words, writing your own `malloc` and `free`). Here is a sample usage of `vmm`:

```
$> ./vmm < input.txt > output.txt
```

`$>` represents the shell prompt.
< means input redirection. > means output redirection. Thus, you can easily use the given test cases to test your program and use the `diff` command to compare the output files.

Getting Started

`vmm_skeleton.c` is provided. You should rename the file as `vmm.c`

Necessary data structures, variables and several helper functions (e.g. linked list operations and `mm_print()`) are already implemented in the provided base code. Instead of reinventing the wheels, please read carefully the provide base codes.

Restrictions

Please note that you **CANNOT** invoke any dynamic memory allocation functions in the C standard library (`<stdlib.h>`), such as `malloc()`, `calloc()`, `realloc()`, `free()`, because these library functions will change the heap and will affect our own memory management implementation. The grader TA will check your codes.

Zero marks will be given if any of the above dynamic memory allocation function is used.

Development Environment

CS Lab 2 is the development environment. Please use one of the following machines (`csl2wk`**`XX`**`.cse.ust.hk`), where **`XX`**=`01…50`. The grader TA will use the same platform to grade all submissions.

In other words, *"my program works on my own laptop/desktop computer, but not in one of the CS Lab 2 machines"* is an invalid appeal reason. Please test your program on our development environment (not on your own desktop/laptop) thoughtfully

The Input Format

An input file stores a number of memory allocation (`malloc`) and memory deallocation (`free`), one operation per line. You can assume the input sequence is valid.

If the input line is for memory allocation, it should have 2 input parameters:

```
malloc [name:char] [size:int]
```

- `name` is a character ranging from `a` to `z` (lower-case).
- `size` is a positive integer indicating the number of bytes to be allocated

If the input line is for memory deallocation, it should follow with 1 input parameter:

```
free [name:char]
```

- `name` is a character ranging from `a` to `z` (lowercase)

Here is an example containing one memory allocation and one memory deallocation. It allocates `1000` bytes to a pointer `a`, and then free the pointer `a`

```
malloc a 1000
free a
```

The Output Format

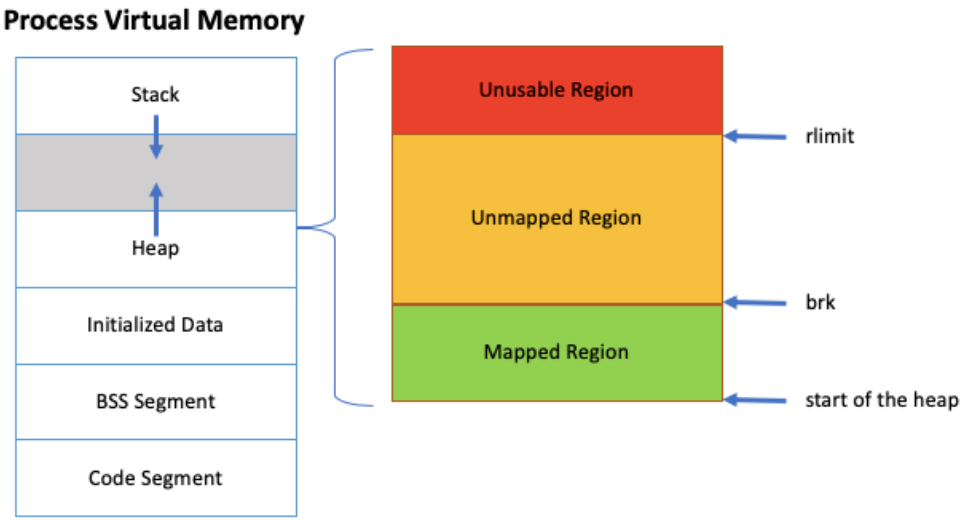
After each step, you should print out the current memory layout with the help of the given `mm_print` function.

Here is the sample output of the above 2 operations:

```
=== malloc a 1000 ===
Block 01: [OCCP] size = 1000 bytes
=== free a ===
Block 01: [FREE] size = 1000 bytes
```

Virtual Memory Address Space

Each process has its own virtual memory address space. In this assignment, we are focusing on the virtual memory address space. In order to build our own memory allocator, we need to understand how different parts of a process (e.g. heap, stack, …) are being mapped in the virtual address space.



sbrk() system call

In general, the heap is a continuous virtual memory address space with 3 main regions.

We only focus on the mapped region (i.e. the green region in the above figure):

- The mapped region is defined by 2 pointers:
  - the start of the heap
  - the current break (`brk`)
- The current break can be adjusted using system calls such as `sbrk()`
- To get the current break address, you can invoke `sbrk(0)`

Heap Data Structure

The following data structure is given in the base code. Please **DON'T** make any changes:

```
struct
  __attribute__((__packed__)) // compiler directive, avoid "gcc" padding bytes
  meta_data {
    size_t size;             // 8 bytes (in 64-bit OS)
    char free;               // 1 byte ('f' or 'o')
    struct meta_data *next;  // 8 bytes (in 64-bit OS)
    struct meta_data *prev;  // 8 bytes (in 64-bit OS)
};
// calculate the meta data size and store as a constant (exactly 25 bytes)
const size_t meta_data_size = sizeof(struct meta_data);
```

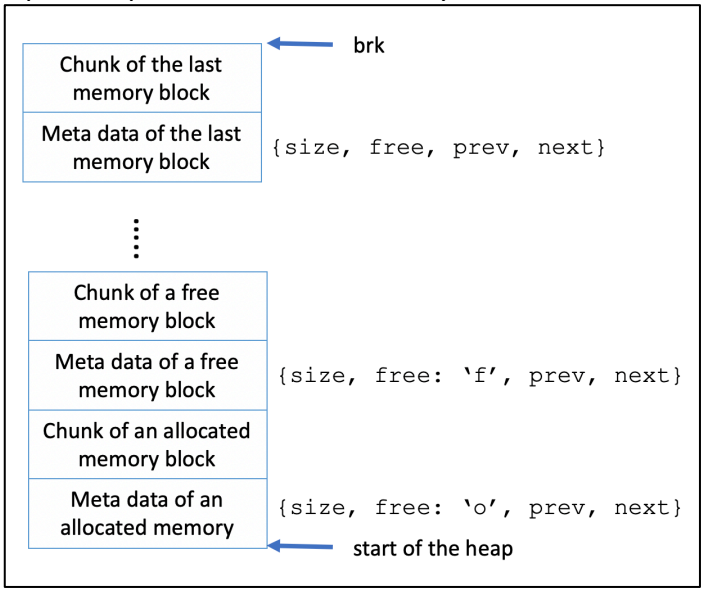
By default, gcc addes extra bytes to a struct (e.g. 32 bytes, instead of 25 bytes, will be used for the above struct). A compiler directive is added to the struct definition to avoid the compiler padding extra bytes to the structure. Padding bytes is a technique to scarifies some bytes to make copying operations consistent and efficient. We would like to be accurate. Thus, the meta data size is exactly equal to 25 bytes in a 64-bit OS.

Linked List Data Structure

In this assignment, we need to implement a linked list to keep track of the memory allocation. When we allocate a memory block, we need to first allocate the meta data and then fill in the details of the meta data. For each block, we should include the followings:

- `size`: the number of bytes for the allocated memory
- `free`: `'f'` means the block is free, and `'o'` means the block is occupied
- `prev, next`: pointers to the previous and the next block

Here is the graphical illustration. Please note that the linked list is an in-place linked list. You need to be familiar with pointer arithmetic and type casting in C programming language to calculate the correct pointer position of each memory block.



In the base code, a doubly linked list with a dummy head pointer and several helper functions of linked list are implemented:

```
// Global variables
void *start_heap = NULL;
// pointing to the start of the heap, initialize in main()
struct meta_data dummy_head_node;
// dummy head node of a doubly linked list, initialize in main()
struct meta_data *head = &dummy_head_node;

// The implementation of the following functions are given:
void list_add(struct meta_data *new, struct meta_data *prev, struct meta_data *next);
void list_add_tail(struct meta_data *new, struct meta_data *head);
void init_list(struct meta_data *list);
```

The Starting Pointing

You only need to complete the following missing parts:

```
int main() {
    start_heap = sbrk(0);
    init_list(head);
    // Assume there are at most 26 different memory allocation/deallocation
    // Here is the mapping rule
    // a=>0, b=>1, ..., z=>25
    void *pointers[26] = {NULL};

    // TODO: Complete the main function here
    return 0;
}

void *mm_malloc(size_t size) {

    // TODO: Complete mm_malloc here
    return NULL;

}

void mm_free(void *p) {
    // TODO: Complete mm_free here
}
```

Implementation of mm_malloc

```
void *mm_malloc(size_t size);
```

The input argument, `size`, is the number of bytes to be allocated from the heap. You can assume `size` is positive. Please ensure that the returned pointer is pointing to the beginning of the allocated space, not the start address of the meta data block.

Iterating the linked list to find the **first-fit free block**, we may have the following situations:

- If no sufficiently large free block is found, we use `sbrk` to allocate more space. After that, we fill in the details of a new block of meta data and then update the linked list
- If the first free block is big enough to be split, we split it into two blocks: one block to hold the newly allocated memory block, and a residual free block.
- If the first free block is not big enough to be split, occupy the whole free block and don't split. Some memory will be wasted (i.e. internal fragmentation). In this project, we don't need to handle internal fragmentation.

Implementation of mm_free

```
void mm_free(void *p);
```

Deallocate the input pointer, `p`, from the heap. In our algorithm, we iterate the linked list and compare the address of p with the address of the data block. If it matches, we mark the `free` attribute of `struct meta_data` from `'o'` (`OCCP`) to `'f'` (`FREE`) and return.

To simplify the requirements of this project, we don't need to release the actual memory back to the operating system (i.e. you don't need to decrease the current break of the heap).

In addition, we don't need to consider the problem of memory fragmentation, which may be a serious issue if we allocate/deallocate small memory blocks for many times.

Compilation

The following command can be used to compile the program

```
$> gcc -std=c99 -o vmm vmm.c
```

The option `c99` is used to provide a more flexible coding style (e.g. you can define an integer variable anywhere within a function)

Sample test cases

9 pair of test cases are provided (i.e. in**X**.txt and out**X**.txt, where **X=1-9**). We don't have other hidden test cases.

The grader TA will probably write a grading script to mark the test cases. Please use the Linux `diff` command to compare your output with the sample output. For example:

```
$> diff --side-by-side your-outX.txt  sample-outX.txt
```

Marking Scheme

1. (20%) Explanation of `mm_malloc()` on the comment block of this function. You should use point form to explain how you implement this function
2. (10%) Explanation of `mm_free()` on the comment block of this function. You should use point form to explain how you implement this function.
3. (70%) Correctness of the 9 pairs of test cases (sum up and scale to 70%)

Zero marks will be given if a student tries to fake the grader TA without implementing the algorithm. For example, a student may detect the input pattern (because all test cases are released) and print the corresponding output. The grader TA will check the codes (and it is not hard to judge)

Zero marks will be given if `malloc()`, `calloc()`, `realloc()`, `free()`, or any other related dynamic memory allocation techniques are used. It can be easily detected by first removing the comment lines, and then search for the above keywords using regular expression.

For example, the following shell command can check whether you use `malloc()`:

```
$> gcc -fpreprocessed -dD -E -P vmm.c | grep -n -E 'malloc[ \t]*\('
| grep -v 'mm_malloc'
```

If yes, the line(s) using `malloc()` will be shown. Otherwise, nothing will be shown.

Zero marks will be given for the plagiarism cases

Plagiarism: Both parties (i.e. a student providing the codes and a student copying the codes) will receive 0 marks.

Zero marks will be given if the file cannot be graded.

The grader TA cannot grade any submission with compilation errors in our CS Lab 2 machine. In the past semesters, some students submitted the executable file instead of the source file. The grader TA cannot grade the executable file.

Submission

File to submit: **`vmm.c`**

Please check carefully you submit the correct file.

You are not required to submit other files such as the input test cases.

You only need to submit the file via CASS on /before the due day mentioned on the course web page.