Project TA - Peter CHUNG (cspeter@cse.ust.hk): Handling questions before the project deadline
Grader TA - Lingyun YANG (lyangbk@cse.ust.hk): Grading and appeal handling after the deadline

COMP3511 Operating System (Fall 2023)

PA3: Simplified Memory Management (smm)

Released on 11-Nov-2023 (Sat) Due on 02-Dec-2023 (Sat) at 23:59 (Note: during the study break)

Introduction

The aim of this project is to help students understand **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()`.

Program Usage

You need to implement a system program named as `smm` to simulate several memory allocations and deallocations without using the corresponding C standard library functions. In other words, you are going to write your own version of `malloc` and `free`.
Here is a sample usage of `smm`:

```
$> ./smm < 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

`smm_skeleton.c` is provided. You should rename the file as `smm.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 starter code.

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.

Zero marks will be given if any of the above dynamic memory allocation function is used. The grader will check your code.

The Input Format

An input file stores several memory allocations and memory deallocations, 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` (lowercase, inclusive)
- `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, inclusive)

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

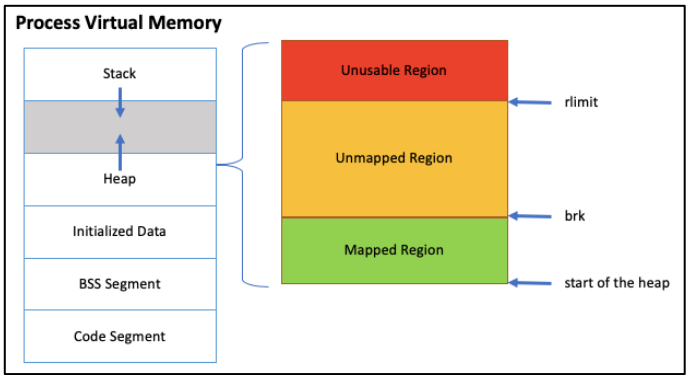
After each step, you should print out the current memory layout with the help of the given `mm_print` function.

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
```

Per-Process Memory Address Space

Each process has its own per-process memory address space (Process Virtual Memory). 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 per-process address space. The layout is:



sbrk() system call

The heap is a continuous 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 call: `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 to struct
    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);
```

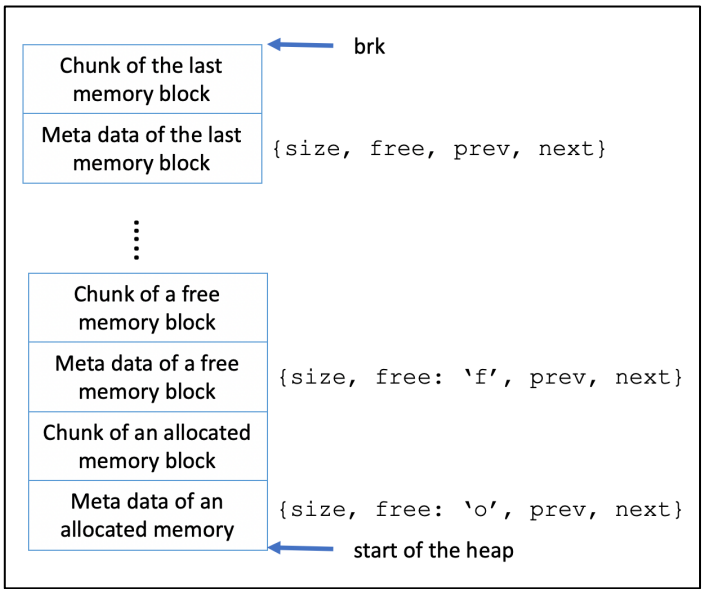
Padding bytes (in the closest power of 2) is a technique to scarify some bytes to make copying operations consistent and efficient. By default, the gcc compiler adds extra bytes to a struct (e.g., 32 bytes, instead of 25 bytes, will be used for the above struct). A compiler directive is added to the struct to avoid the compiler padding extra bytes. In this project, we would like to be accurate. Thus, the meta data size is exactly equal to 25 bytes.

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** to calculate the correct pointer position of each memory block.



A doubly linked list with a dummy head node and several helper functions of linked list are implemented and provided in the skeleton code:

```
// Global variables

// pointing to the start of the heap, initialize in main()
void *start_heap = NULL;

// dummy head node of a doubly linked list, initialize in main()
struct meta_data dummy_head_node;
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:

```
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** with 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 the internal fragmentation problem.

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. Don't need to run any memory leak checker for this assignment, this assignment has memory leak due to the above simplified requirements.

Compilation

The following command can be used to compile the program

```
$> gcc -std=c99 -o smm smm.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)

Test Cases

5 pair of test cases are provided (i.e., in**X**.txt and out**X**.txt, where **X=1-5**).

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
```

In addition to the given test cases, we have some hidden test cases.

Development Environment

CS Lab 2 is the development environment. Please use one of the following machines (`csl2wk`**`XX`**`.cse.ust.hk`), where **`XX`**`=01…40`. 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

Submission

File to submit:

`smm.c`

Please check carefully you submit the correct file. In the past semesters, some students submitted the executable file instead of the source file. Zero marks will be given as the grader cannot grade the executable file. You are not required to submit other files, such as the input test cases.

Marking Scheme

1. (50%) Correctness of the **5** given test cases.
    a. We will check the source codes to avoid students hard coding the test cases in their programs.
    b. Example of hard coding: a student may simply detect the input and then display the corresponding output without implementing the program. 0 marks will be given if hard coding is confirmed, even all 5 given test cases are passed.
2. (50%) Correctness of the **5** hidden test cases
    a. The hidden test cases are useful to detect hard coding (e.g., the chance of hard coding is high if a student can pass all given test cases but fail in all hidden test cases.
3. Make sure you use the Linux diff command to check the output format.
4. Make sure to test your program in one of our CS Lab 2 machines (not your own desktop/laptop computer)
5. Please fill in your name, ITSC email, and declare that you do not copy from others. A template is already provided near the top of the source file.

Zero marks will be given if `malloc(), calloc(), realloc(), free()`, or any other related dynamic memory allocation techniques are used. The reason is that it is meaningless to write your own malloc using an existing malloc function. During the grading, the usage of malloc can be easily detected by first removing the comment lines in your source file, and then search for the above keywords using regular expression.

Zero marks will be given for the plagiarism cases

***Plagiarism**: Both parties (i.e., students providing the codes and students copying the codes) will receive 0 marks. Near the end of the semester, a plagiarism detection software (JPlag) will be used to identify cheating cases. **DON'T** do any cheating!*

Late Submission

For late submission, please submit it via email to the grader TA.
A 10% deduction, and only 1 day late is allowed (Reference: Chapter 1 of the lecture notes)