Project TA - Peter CHUNG (cspeter@cse.ust.hk): Handling questions before the deadline
Grader TA - Tianyi BAI (tbaiag@cse.ust.hk): Grading and appeal handling after the deadline

COMP3511 Operating System (Fall 2024)

PA3: Simplified Memory Management (smm)

Released on 10-Nov (Sun) Due on 30-Nov (Sat) at 23:59

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. `mm_print()`) are already implemented in the provided base code. Instead of reinventing the wheels, please read carefully the starter 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)

If the input line is for defragmentation, you should only see 1 special word:

```
combine_nearby_free
```

Here is an example containing one memory allocation and one memory deallocation (in total, 2 operations). It allocates `1000` bytes to a pointer `a`, and then free the pointer `a`

```
2
malloc a 1000
free a
```

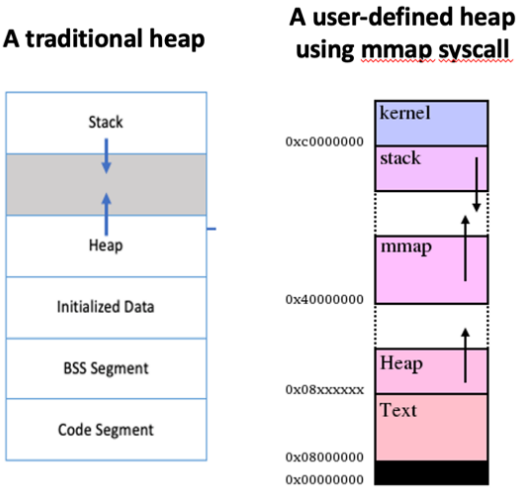
The Output Format

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
```

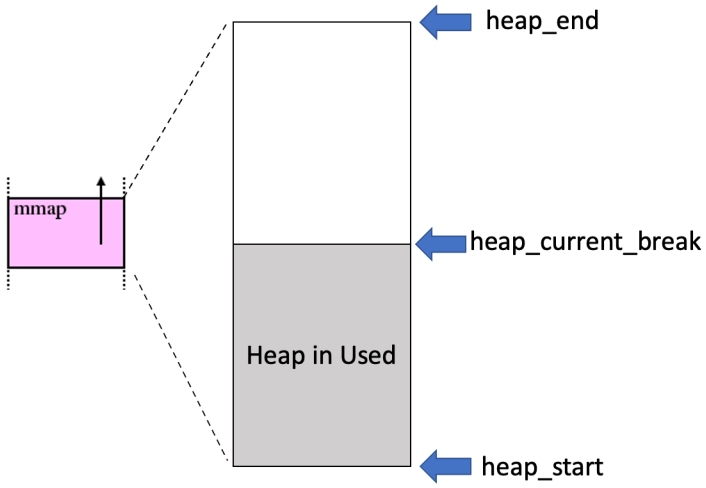
A Traditional Heap V.S. A User-defined Heap



Traditionally, each process has a heap region, and we can use a special system call (`sbrk`) to change the size of the heap. `sbrk` is still functional but deprecated. The main problem is that the `malloc/free` in C standard library makes changes to the heap segment at the same time.

The modern way to implement a customized memory allocator is to use `mmap/munmap` system call to create/remove a mapped region from the per-process memory space. In this project assignment, we adopt the modern approach, and implement a helper function `mm_sbrk` to simulate the original `sbrk` system call.

mm_sbrk() helper function



To get the current break address, you can invoke `mm_sbrk(0)`
You can use `mm_sbrk(size)`, where size is in bytes, to expand the heap region
You can use `mm_sbrk(size)` with a negative size to shrink the heap region

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
    MetaData
{
    size_t size; // 8 bytes (in 64-bit OS)
    char status; // 1 byte ('f' or 'o')
};

// calculate the meta data size and store as a constant (exactly 9 bytes)
const size_t meta_data_size = sizeof(struct MetaData);
```

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. A compiler directive is added to the struct to avoid the compiler padding extra bytes. In this project, we would like to be accurate (i.e., automatically packing bytes will mess up our data structure, and we need to ask the gcc compiler to stop that)

Thus, the meta data size is exactly equal to 9 bytes (i.e., without extra bytes added).

A Sample Memory Layout

```
// Data structure of MetaData
//
// The memory layout for this project assignment is:
//
// |--------------| <-- heap_current_break
// | Data N       |
// |--------------|
// | MetaData N   |
// |--------------|
// |    ...       |
// |    ...       |
// |--------------|
// | Data 1       |
// |--------------|
// | MetaData 1   |
// |--------------| <--- heap_start
```

The meta data block stores the information related to the following allocated memory block

- 'f' means the block is free
- 'o' means the block is occupied

As each MetaData block has exactly 9 bytes and we can use pointer arithmetic to traverse the region.

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.
Find the **first-fit free block** with the following situations:

- If no sufficiently large free block is found, we use `mm_sbrk` to allocate more space. After that, we fill in the details of a new block of meta data
- 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.
- Be-careful with the zero-size memory. For example, suppose the meta data size is 9 bytes, and we allocate 291 bytes from 300 bytes free block. We won't split because splitting will generate a zero-size memory (i.e., 9 bytes of meta data and 0 bytes for the memory block)

Implementation of mm_free

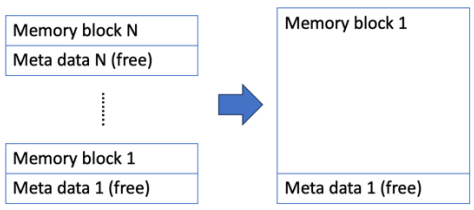
```
void mm_free(void *p);
```

Deallocate the input pointer, `p`, from the heap. In our algorithm, we iterate and compare the address of `p` with the address of the data block. If it matches, we mark the `free` attribute of `MetaData` 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).

Implementation of mm_combine_nearby_free

```
void mm_combine_nearby_free();
```

Our simple algorithm suffers from the fragmentation problem after several memory allocations and deallocations. A simple strategy is to combine nearby free blocks into a bigger one. Here is an example, and please read the PA3-related lab notes for more:



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

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 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)