COMP3511 Operating System (Fall 2019)

Project 2 – A Simplified malloc/free

Submission due date: 29/11/2019 23:59 via CASS

Introduction

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

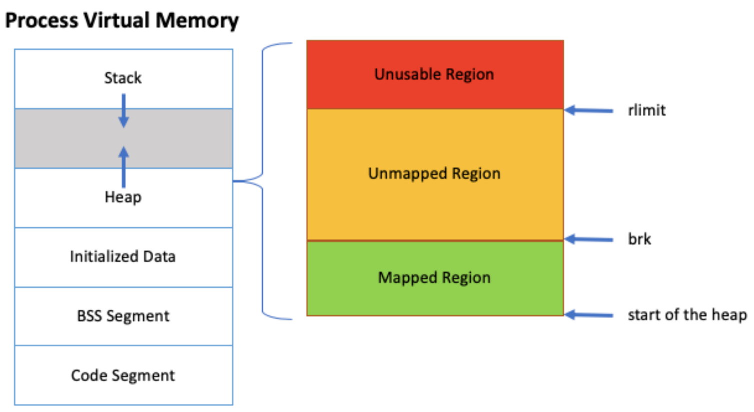
Getting Started

`pa2_skeleton.c` is provided. Necessary data structures, variables and several helper functions (e.g. `mm_print()`) are already implemented in the provided base code.

The `main()` function contains 8 test cases. Please **DON'T** make any changes on the provided `main()` function. In each case, either `mm_malloc()` or `mm_free` is used to change the memory states. After each step, `mm_print()` is invoked to print out the current memory states.

Virtual Memory Address Space

Each process has its own virtual memory address space. The operating system maps the virtual memory to physical memory through address translation. 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:

- The mapped region is defined by 2 pointers: the start of the heap and the current break. The current break can be adjusted using system calls such as `brk()` and `sbrk()`
- The unmapped region is defined by 2 pointers: the current break and the region limit. The region limit is the hard limit of the heap, which the break cannot surpass. The hard limit can be retrieved or adjusted using system calls such as `getrlimit` and `setrlimit`
- The unusable region is defined by the `rlimit` pointer and the stack segment of the program.
- To get the current break address, you can invoke `sbrk(0)`

sbrk_fail() helper function

In this assignment, you don't need to worry about the unusable region. After invoking `sbrk()`, you should use the provided helper function, `sbrk_fail()` to check whether the `sbrk()` function call is valid.

Here is a sample usage of `sbrk_fail()` in the skeleton code:

```
void *p = sbrk(size); // "size" (in bytes) to increase
if ( !sbrk_fail(p) ) {
      // okay!
} else {
      // failed
}
```

Heap Data Structure

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

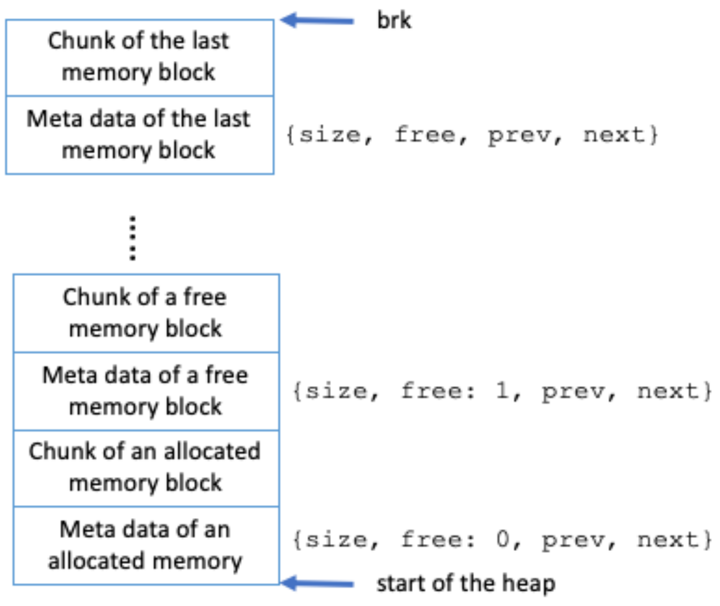
```
/**
 * Data structure of meta_data (32 bytes in 64-bit Linux OS)
 */
struct meta_data {
    size_t size;              // 8 bytes
    int free;                 // 8 bytes
    struct meta_data *next;   // 8 bytes
    struct meta_data *prev;   // 8 bytes
};
// calculate the meta data size and store as a constant
const size_t meta_data_size = sizeof(struct meta_data);
```

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`: `1` means the block is free, and `0` means the block is occupied
- `prev, next`: pointers to the previous and the next block

Here is the graphical depiction of a sample memory layout. Please note that the linked list is an in-place linked list. Please note that 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.


brk
Chunk of the last memory block
Meta data of the last memory block
{size, free, prev, next}
Chunk of a free memory block
Meta data of a free memory block
{size, free: 1, prev, next}
Chunk of an allocated memory block
Meta data of an allocated memory
{size, free: 0, prev, next}
start of the heap

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

```
void *start_heap = NULL;
struct meta_data dummy_head_node;
struct meta_data *head = &dummy_head_node;

// The implementation of the following functions are given:
struct meta_data *find_free_block(struct meta_data *head,
size_t size);
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 2 functions in the skeleton code:

```
void *mm_malloc(size_t size)
{
    /* TODO: Implement mm_malloc */
    return NULL;
}
void mm_free(void *p)
{
    /* TODO: Implement mm_free */
}
```

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. Please ensure that the returned pointer is pointing to the beginning of the allocated space, not the start address of the meta data block.

In this assignment, we iterate 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.
- Special case 1: Return `NULL` if we cannot allocate the new requested size
- Special case 2: Return `NULL` if the request size is 0

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 0 (`OCCP`) to 1 (`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). We also 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.

Expected Output

Use the following commands (marked in RED) to compile and run the program. Please note that the addresses of `heap` and `brk` may be different every time you run your program.

```
$> gcc -o pa2 pa2.c
$> ./pa2
=== After step 1 ===
start_heap = 0x158e000
>>>
Block 01: [OCCP] size = 1000 bytes
>>>
brk = 0x158e408
=== After step 2 ===
start_heap = 0x158e000
>>>
Block 01: [FREE] size = 1000 bytes
>>>
brk = 0x158e408
=== After step 3 ===
start_heap = 0x158e000
>>>
Block 01: [OCCP] size = 500 bytes
Block 02: [FREE] size = 468 bytes
>>>
brk = 0x158e408
=== After step 4 ===
start_heap = 0x158e000
>>>
Block 01: [OCCP] size = 500 bytes
Block 02: [OCCP] size = 468 bytes
>>>
brk = 0x158e408
=== After step 5 ===
start_heap = 0x158e000
>>>
Block 01: [OCCP] size = 500 bytes
Block 02: [OCCP] size = 468 bytes
Block 03: [OCCP] size = 5000 bytes
>>>
brk = 0x158f7b0
=== After step 6 ===
start_heap = 0x158e000
>>>
Block 01: [OCCP] size = 500 bytes
Block 02: [FREE] size = 468 bytes
Block 03: [OCCP] size = 5000 bytes
>>>
brk = 0x158f7b0
=== After step 7 ===
start_heap = 0x158e000
>>>
Block 01: [FREE] size = 500 bytes
Block 02: [FREE] size = 468 bytes
Block 03: [OCCP] size = 5000 bytes
>>>
brk = 0x158f7b0
=== After step 8 ===
start_heap = 0x158e000
>>>
Block 01: [FREE] size = 500 bytes
Block 02: [FREE] size = 468 bytes
Block 03: [FREE] size = 5000 bytes
>>>
brk = 0x158f7b0
```

Explanation of the expected output

In step 1, we use `mm_malloc` to allocate 1000 bytes to pointer p. Thus, 1000 bytes (+32 bytes for meta data) is required. The total size of the heap is `408(hex)` (Note: it is the difference between the start of the heap and the current break). `408 (hex)` is equal to `1032 (decimal)` bytes.

In step 2, we use `mm_free` to free the pointer `p` in step 1. Thus, the meta data of block 01 is set to free (`FREE`), which is different from occupied (`OCCP`) in step 1.

In step 3, we use `mm_malloc` to allocate 500 bytes to pointer q. Based on our algorithm, we find the first fit (a 1000-byte free block). As it is large enough to be split, we split the block to an occupied block (500 bytes) and a new free block. The new free block is 468 bytes (instead of 500 bytes) because meta data is required for the new free block.

In step 4, we use `mm_malloc` to allocate 462 bytes to pointer r. Based on our algorithm, we find the first fit (a 468-byte free block). However, it is not large enough to be split, so the whole block is occupied (468 bytes).

In step 5, we use `mm_malloc` to allocate 5000 bytes to pointer s. Based on our algorithm, we need to expand the heap. You should also notice that the current break is adjusted after the step 5.

In step 6, we use `mm_free` to free the pointer r (in step 4). Block 02 is marked as FREE.
In step 7, we use `mm_free` to free the pointer q (in step 3). Block 01 is marked as FREE.
In step 8, we use `mm_free` to free the pointer s (in step 5). Block 03 is marked as FREE.

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 program. Besides from the given test cases in the `main()` function, the TA may have another set of test case to test some boundary conditions (such as error handling). Please read carefully the problem description and handle the boundary cases as well.

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

Submission

You only need to submit **pa2.c** via CASS on /before the due day mentioned on the course web page. Please use one of the following machines (csl2wk**XX**.cse.ust.hk), where **XX** = 01 to 50.