COMP3511

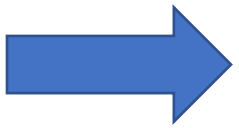
Project Introduction: A simplified malloc/free

Overview

- Implement a system program to simulate a sequence of memory allocation and deallocation
- Example:

```
malloc a 1000
free a
malloc b 300
malloc c 200
free c
free b
```

Sample Input



```
=== malloc a 1000 ===
Block 01: [OCCP] size = 1000 bytes
=== free a ===
Block 01: [FREE] size = 1000 bytes
=== malloc b 300 ===
Block 01: [OCCP] size = 300 bytes
Block 02: [FREE] size = 675 bytes
=== malloc c 200 ===
Block 01: [OCCP] size = 300 bytes
Block 02: [OCCP] size = 200 bytes
Block 03: [FREE] size = 450 bytes
=== free c ===
Block 01: [OCCP] size = 300 bytes
Block 02: [FREE] size = 200 bytes
Block 03: [FREE] size = 450 bytes
=== free b ===
Block 01: [FREE] size = 300 bytes
Block 02: [FREE] size = 200 bytes
Block 03: [FREE] size = 450 bytes
```

Sample Output

malloc and free in <stdlib.h>

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

- The malloc function requests a block of memory from the heap
- If the request is granted, the operating system will reserve the amount of memory (in bytes)
- The function returns the pointer pointing to the start of the memory address

```
void free(void *ptr);
```

- The free function frees the memory space pointed to by `ptr`
- `ptr` must point to a previously allocated memory from `malloc` (or variants of `malloc`)
- If `ptr` is NULL, no operation is performed.

Project Goal

- The aim of this project is to help students understand **virtual memory management** in an operating system
  - In other words, we are going to develop our own `malloc/free` functions
- Upon completion of the project, students should be able to write their own simplified memory management functions: `mm_malloc` and `mm_free`

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 implicitly and will affect our own memory management implementation
- **Zero** marks will be given if the above functions are invoked anywhere in your code
  - For example, the grader can check whether you use `malloc()` by using regular expression

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

Getting Started

- A base code (`vmm_skeleton.c`) is provided
  - Necessary data structures, variables and several helper functions (e.g. `mm_print`) are already implemented in the provided base code
- We will take a closer look at the base code together in the later slides

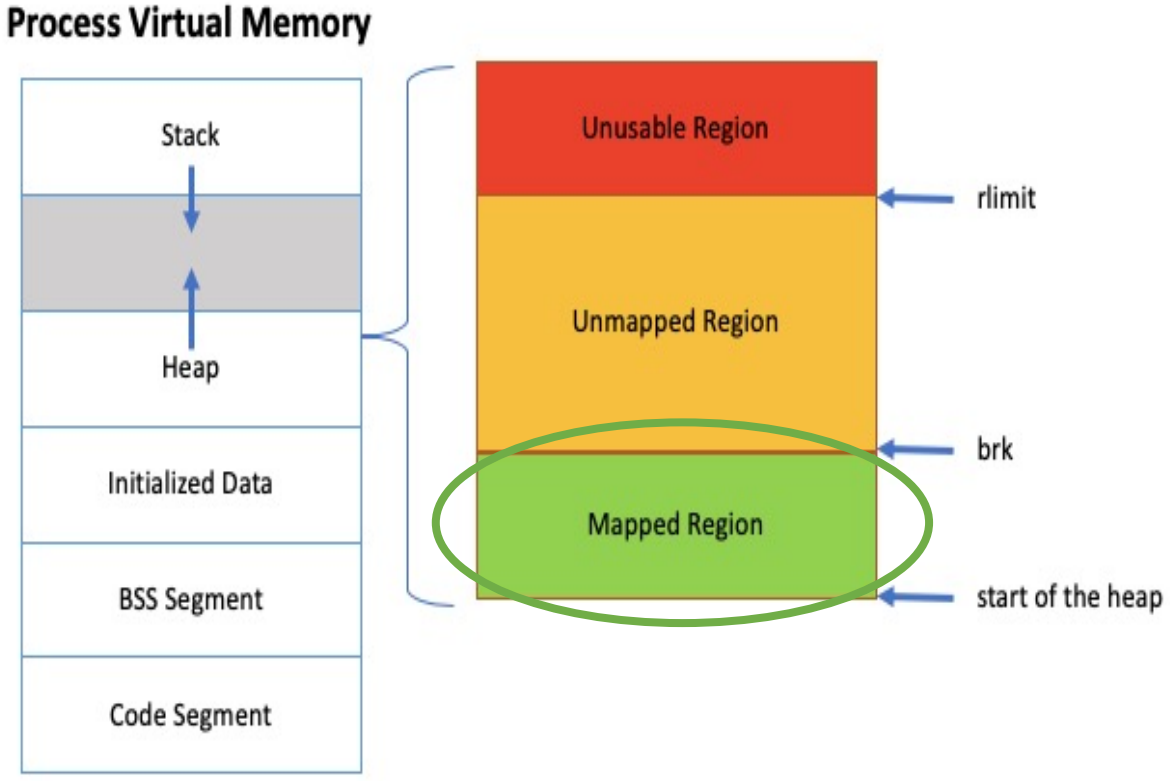
The starting point

- 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
}
```

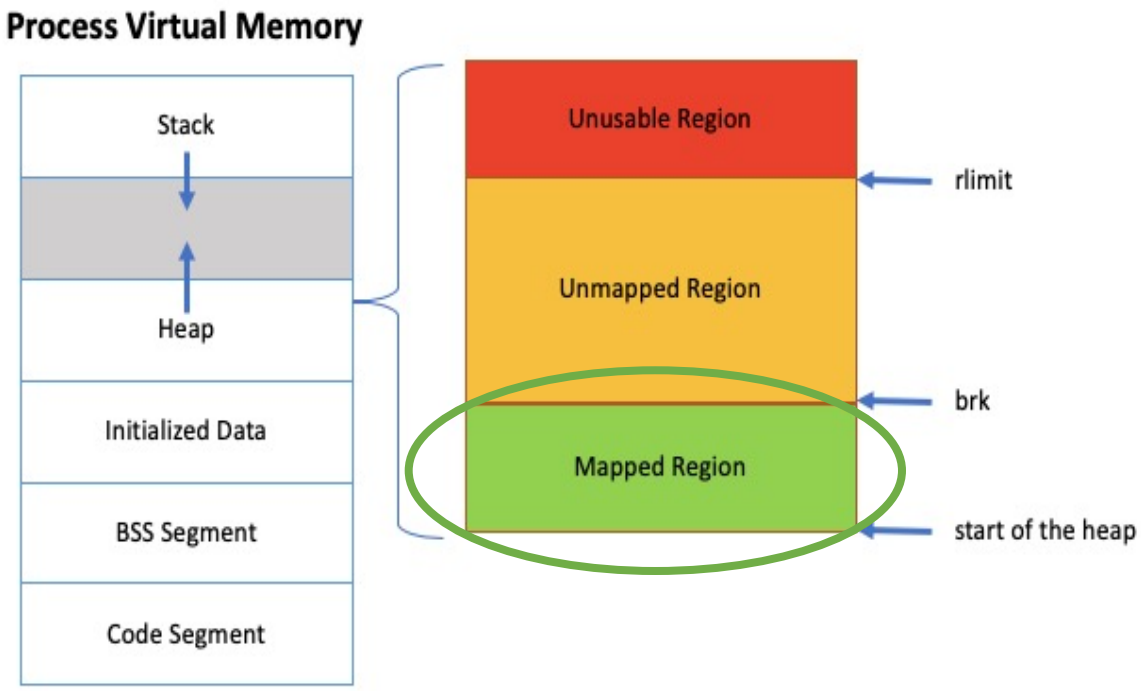
Virtual Memory Address Space

- Each process has its own 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
- In PA2, we only focus on the mapped region (i.e. the green region)



Using `sbrk()`

- To get the current break address, you can invoke `sbrk(0)`
- You can use `sbrk(size)`, where size is in bytes, to expand the mapped region
- You can use `sbrk(size)`, with a negative size value, to shrink the mapped region
  - However, it is not necessary in our project assignment



Data Structure Specification

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

Data Structure Implementation

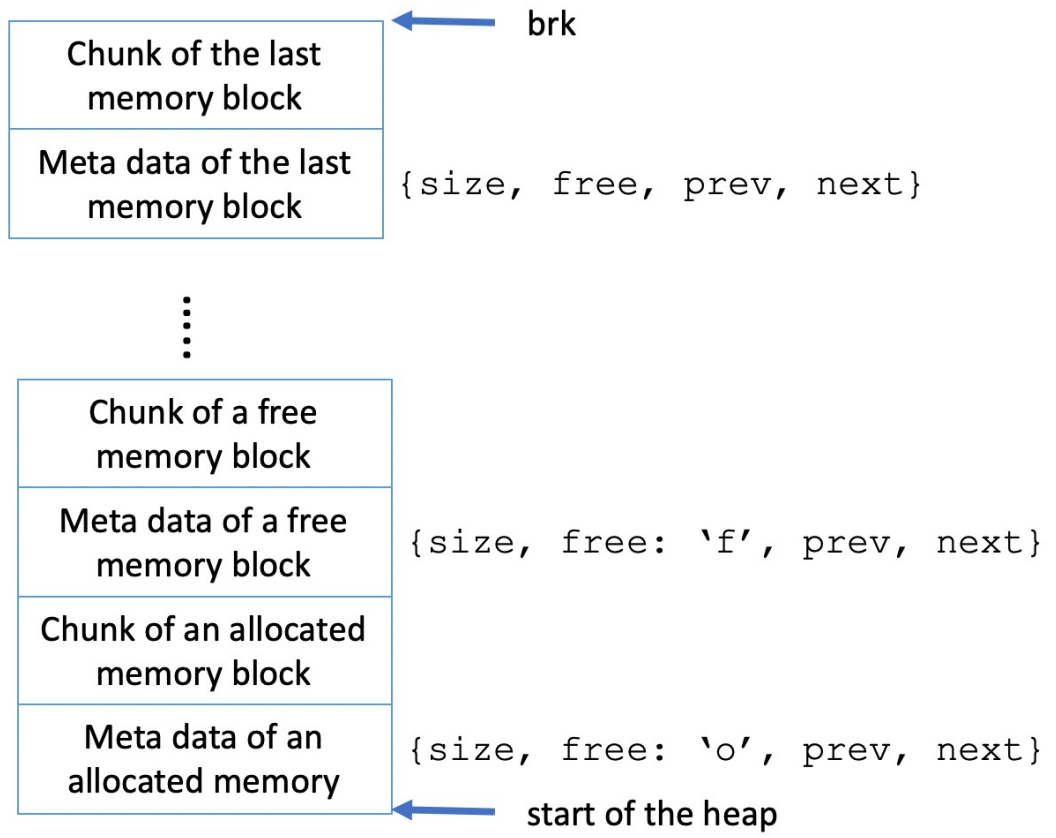
- The following data structure is given in the base code
- Please **DON'T** make any changes on this data structure

```
// Data structure of meta_data
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);
```

A sample memory layout

- This example shows at least 3 memory blocks in the linked list
- 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



Do we need to implement a linked list?

- No, **the TA already implemented**
  - A circular doubly-linked list with a dummy head node
  - Necessary functions to initialize and insert a node to the linked list
  - Why there is no linked list item deletion?
    - It is because the delete operation is not required in this project

```
// 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
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);
```

`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

Algorithm for `mm_malloc`

- 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
    - Use `sbrk` to allocate more space
    - After that, we fill in the meta data of the new block and then update the linked list
  - If the first free block is big enough to be split, we split it into 2 blocks:
    - One block holding the newly allocated memory block
    - The remaining bytes are assigned to 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
    - A few bytes will be wasted (i.e., internal fragmentation)
    - In this project, we don't need to handle internal fragmentation

`void mm_free(void *p);`

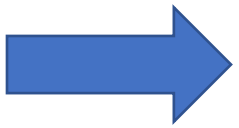
- Deallocate the input pointer `p` from the heap
- Algorithm:
  - we iterate the linked list and compare the address of `p` with the address of the data block
  - If it matches, we mark the free attribute of struct `meta_data` from `o(OCCP)` to `f(FREE)` and return
- To simplify the requirements of this 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

A Step-by-Step Illustration

- We use the following test case for a step-by-step illustration
- Example:

```
malloc a 1000
free a
malloc b 300
malloc c 200
free c
free b
```

Sample Input



```
=== malloc a 1000 ===
Block 01: [OCCP] size = 1000 bytes
=== free a ===
Block 01: [FREE] size = 1000 bytes
=== malloc b 300 ===
Block 01: [OCCP] size = 300 bytes
Block 02: [FREE] size = 675 bytes
=== malloc c 200 ===
Block 01: [OCCP] size = 300 bytes
Block 02: [OCCP] size = 200 bytes
Block 03: [FREE] size = 450 bytes
=== free c ===
Block 01: [OCCP] size = 300 bytes
Block 02: [FREE] size = 200 bytes
Block 03: [FREE] size = 450 bytes
=== free b ===
Block 01: [FREE] size = 300 bytes
Block 02: [FREE] size = 200 bytes
Block 03: [FREE] size = 450 bytes
```

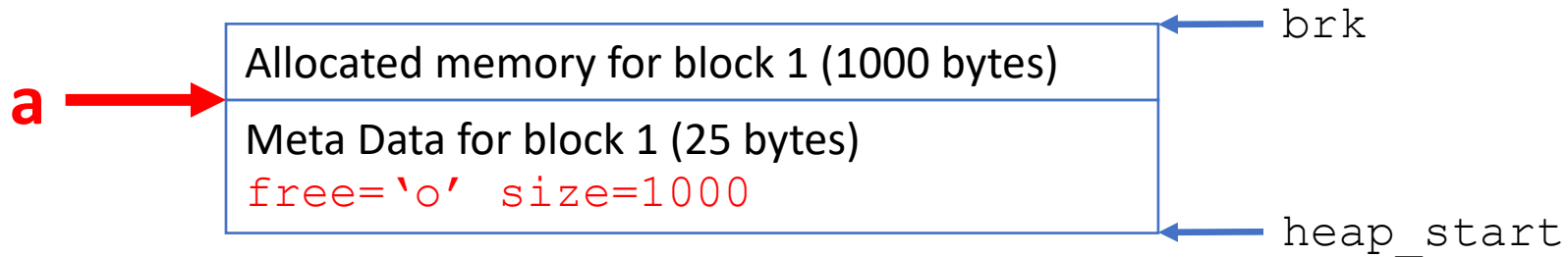
Sample Output

Step 1: allocate 1000 bytes

`malloc a 1000`

Reason: no sufficiently large free block is found

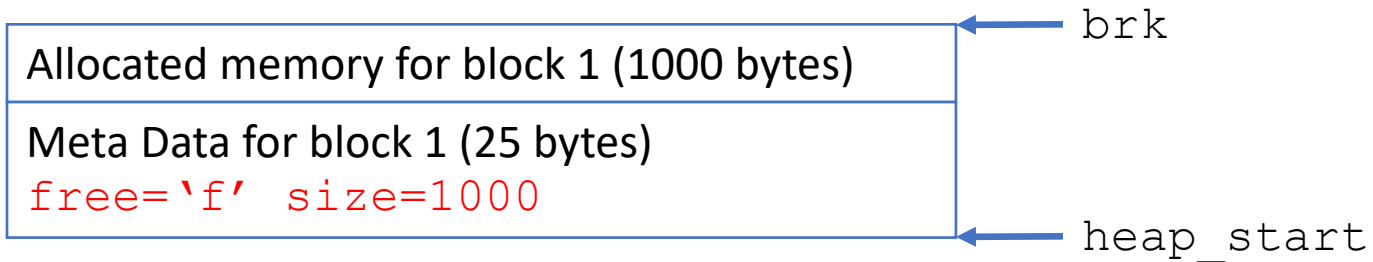
Please note that the pointer should point to the allocated memory block (not the meta data block)
*Otherwise, we will erase the meta data if we updating the data (e.g. *a = new value)*



Step 2: free 1000 bytes

`free a`

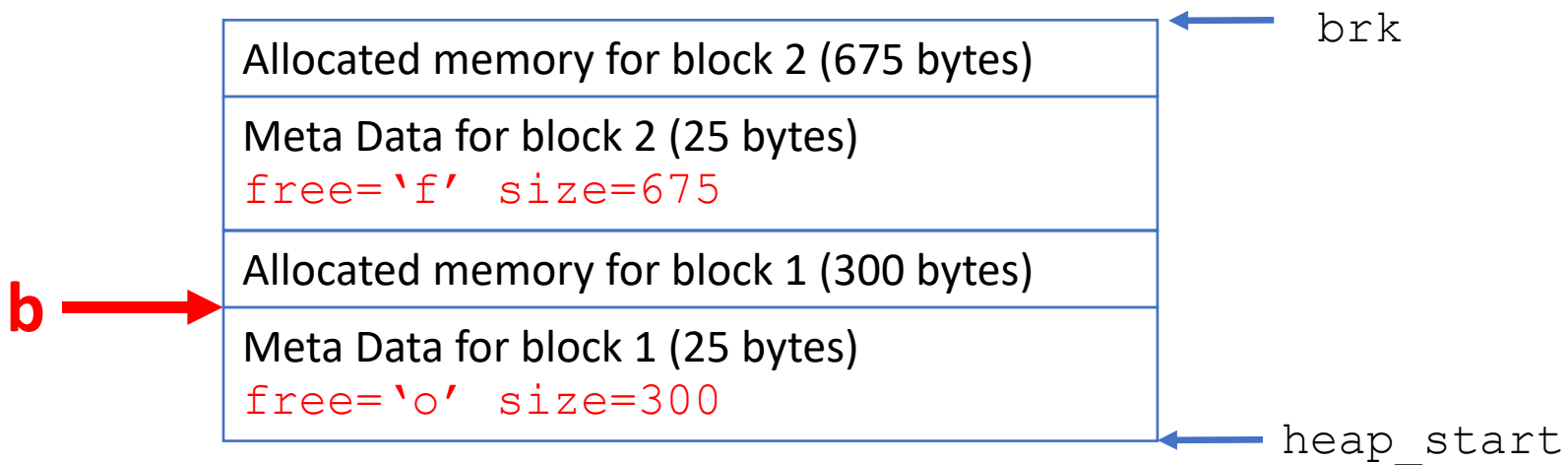
Reason: free an occupied block from the heap



Step 3: allocate 300 bytes

`malloc b 300`

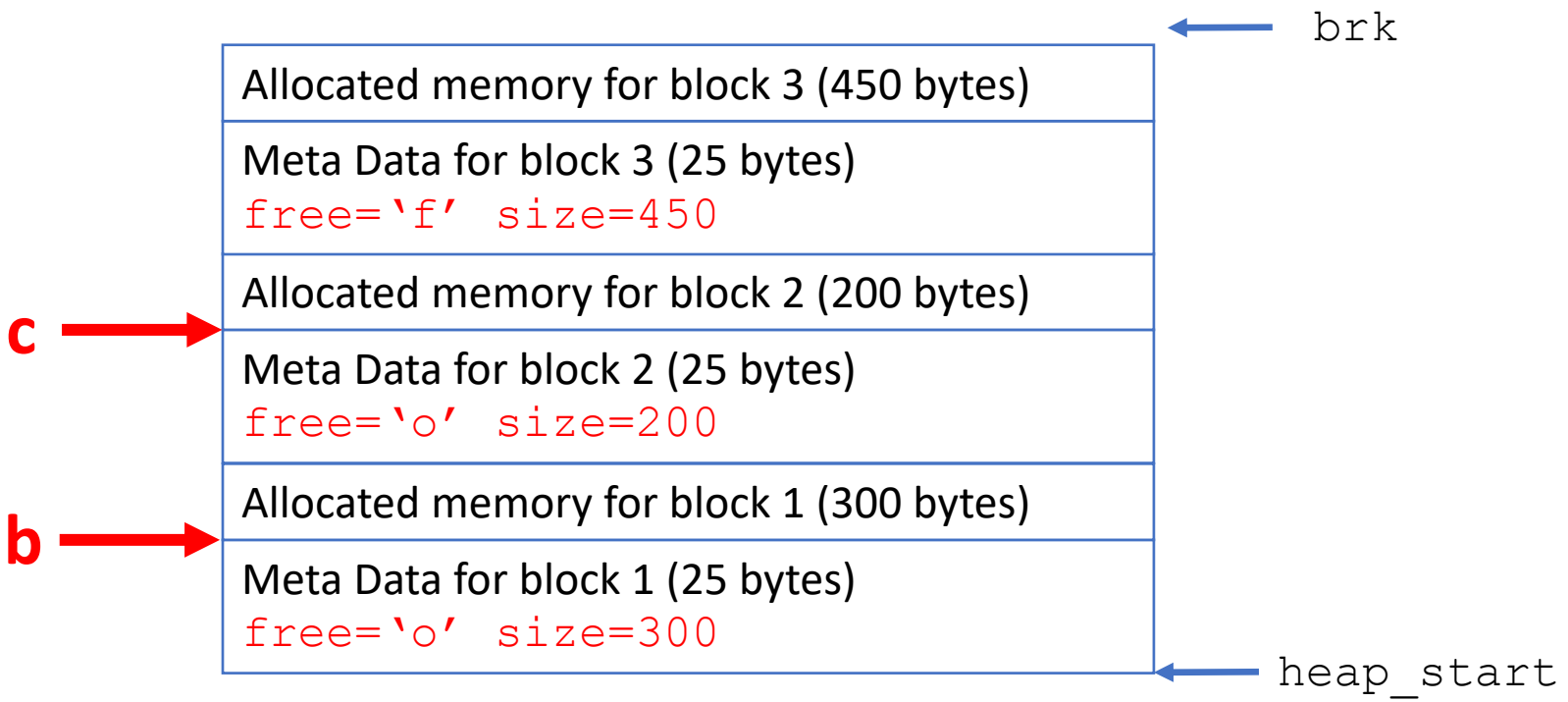
Reason: first free block is big enough to be split



Step 4: allocate 200 bytes

`malloc c 200`

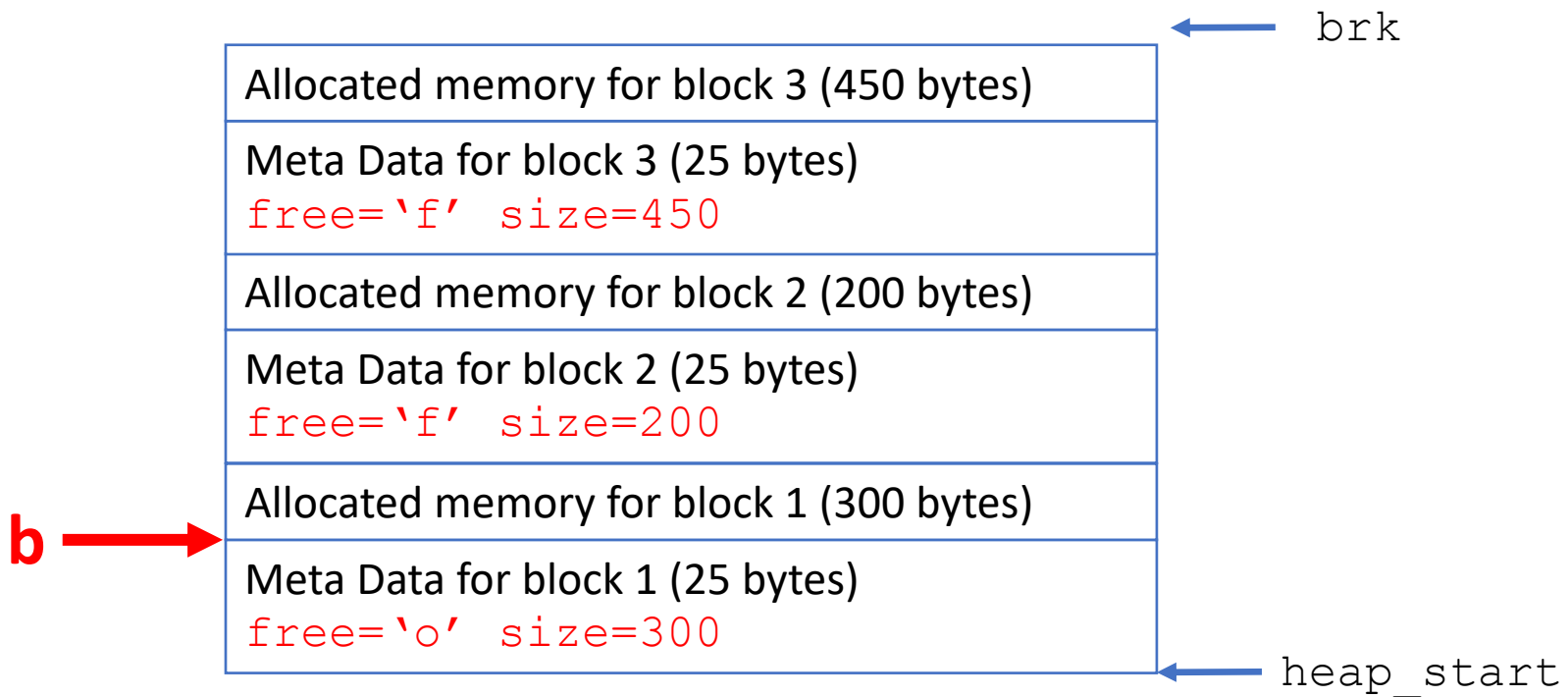
Reason: first free block is big enough to be split



Step 5: free pointer c

`free c`

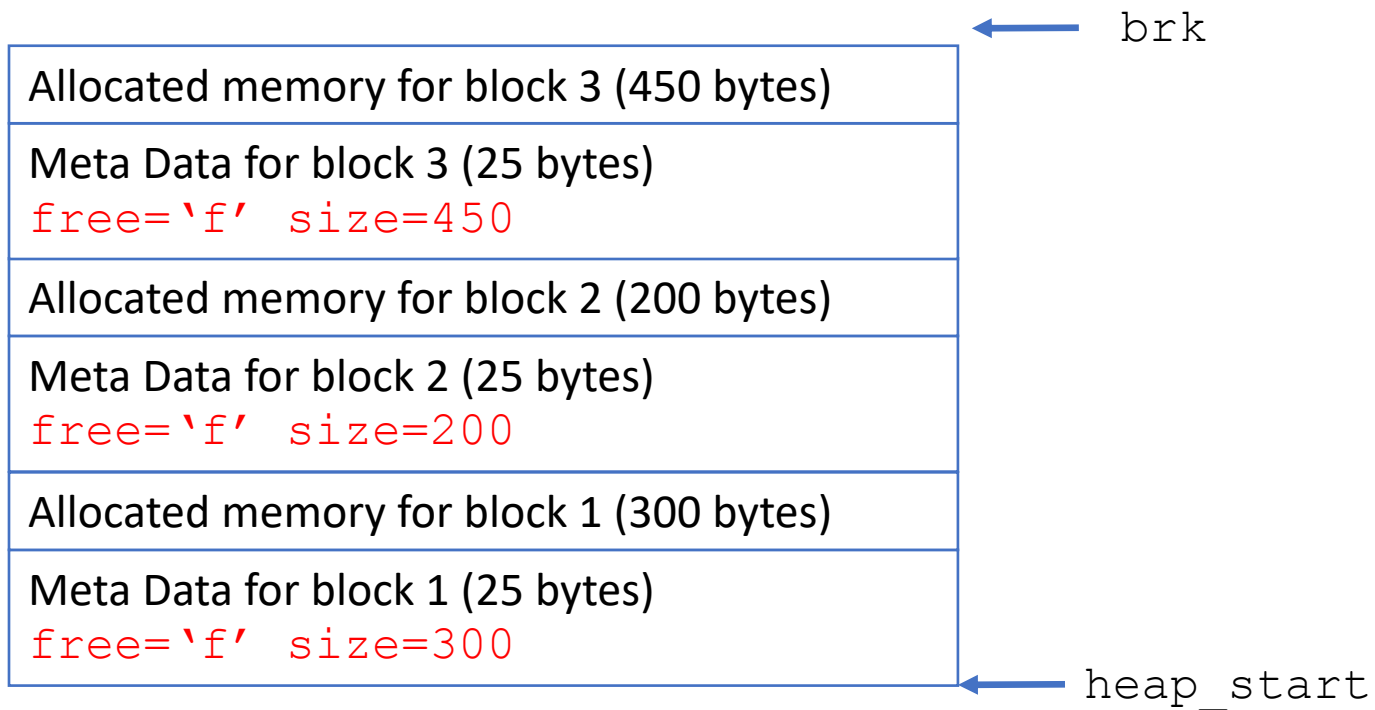
Reason: free an occupied block from the heap



Step 6: free pointer b

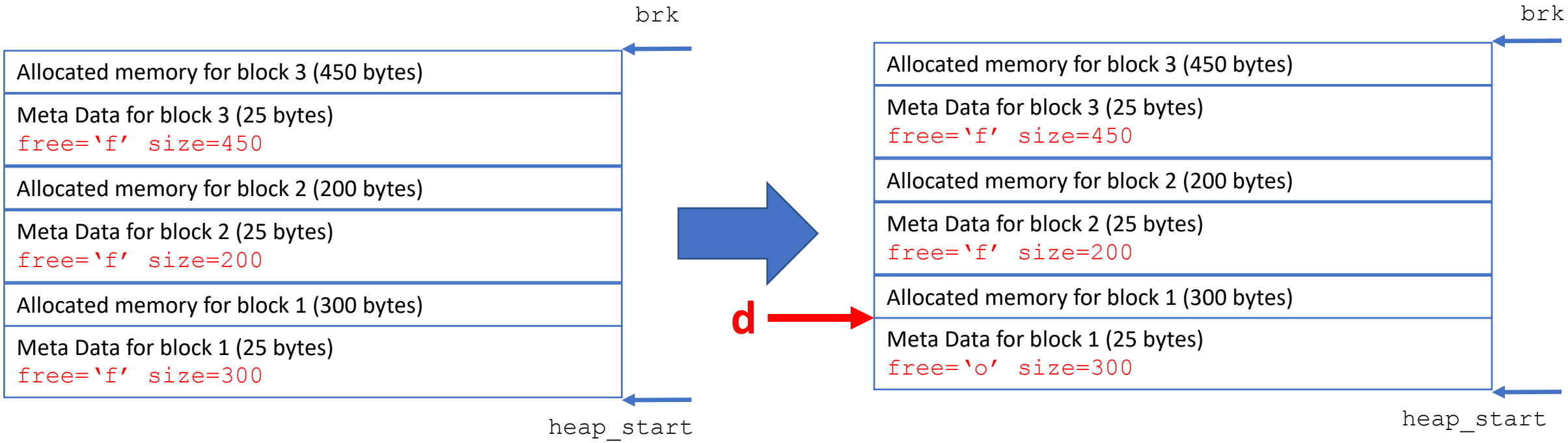
`free b`

Reason: free an occupied block from the heap



What if the first free block is not big enough to split?

- After the previous step, we allocate 299 bytes to pointer ’d’
  - `malloc d 299`
  - 299 bytes can be fitted in the first free block (300 bytes)
  - It is not enough to split to 2 blocks. Thus, the whole free block is occupied



Sample test cases

- 10 pair of test cases are provided
  - (i.e. in**X**.txt and out**X**.txt, where **X=1-10**)
- The grader TA will probably write a grading script to mark the test cases
  - The helper print function (i.e. `mm_print`) is given in the base code to avoid text formatting problems
  - Please use the Linux diff command to compare your output with the sample output

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

Summary

- **Think** carefully before you type **ANY** line of code
  - Good C programmers never do trial-and-error
  - A program that can compile does not mean that it can execute correctly
  - Check carefully to avoid runtime errors (i.e., Segmentation fault)
- Read carefully the provided base code
- Make sure you understand how to use the provided linked list related functions

Live Demo

The skeleton code

The sample Linux executable program