Operating Systems – spring 2022
Assignment 4

Instructor: Hans P. Reiser

Submission Deadline: Monday, Februar 14th, 2022 – 23:59

A new assignment will be published every week, right after (or a bit before) the last one was due. It must be completed before its submission deadline. (Hard deadlines, no extension.)

T-Questions are theory homework assignments. **The answers to the assignments must be uploaded to Canvas (in the quiz).**

P-Questions are programming assignments. Download the provided template from Canvas. Do not fiddle with the compiler flags. Submission instructions can also be found in first assignment.

Topics of this assignment are threads and segmentation, and you will implement a simple dynamic memory allocator.

T-Question 4.1: Threads and Thread Models

a. In the lecture we discussed two different basic approaches how to implement multiple threads for a user process, depending on where the thread management is implemented. What are these two types? Give a short explanation for each. **1 T-pt**

b. Which of the following objects are all shared by the threads of a process (i.e., managed by/stored in the PCB, not the individual TCB)? **1 T-pt**

true	not true	
☐	☐	Code, heap, open network connections
☐	☐	Instruction pointer, register contents, open files
☐	☐	Code and stack only
☐	☐	Code, stack, open network connections
☐	☐	Instruction pointer, code, register contents, stack

c. Explain (briefly, one sentence each) two disadvantages of the many-to-one thread model. **2 T-pt**

d. Do you agree with the statement "'Switching to a different process is usually much faster than switching to a different user level thread, because switching processes can very efficiently be handled by the CPU scheduler of the operating systems"? Justify! **2 T-pt**

T-Question 4.2: Segmentation

a. Assume a system with 16-bit virtual addresses that supports four different segments (the two most significant bits of the virtual address encode the segment number, the remaining 14 bits the offset within the segment). Assume we use the

following segment table: **4 T-pt**

Segment Number	Base	Limit
0	0x1000	0x15f3
1	0x8000	0x00FF
2	0x25f3	0x1000
3	0x4300	0x0300

Complete the following table.

Virtual Address	Segment Number	Offset	Valid?	Physical Address
0xC0DE				
	1	0x0200		
			yes	0x25f3

P-Question 4.1: Simple Heap Allocator

Download the template **p1** for this assignment from Canvas. You may only modify and upload the file `malloc.c`.

A process can request memory at runtime by employing a dynamic memory allocation facility. You already used such a facility in the last assignment by calling the user-space C heap allocator with `malloc()` and `free()`. In this programming question you will implement your own heap allocator. Your heap will organize free and allocated memory regions into blocks (see Figure 1).

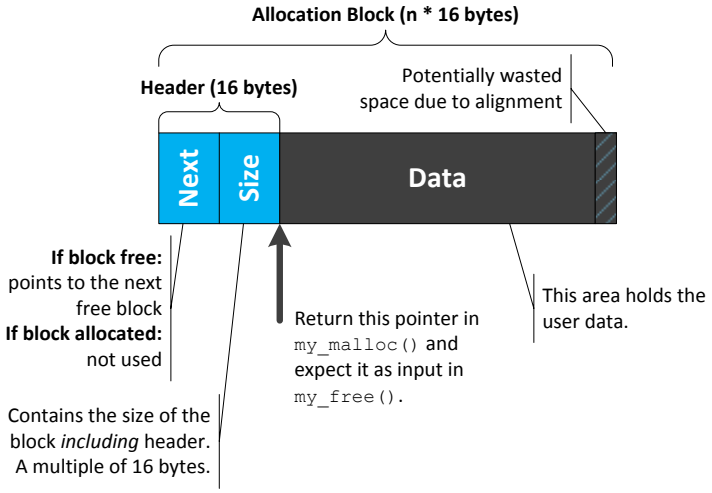

Figure 1: Heap block

Each block starts with a header (16 bytes), which describes the block. The header consists of a `next` pointer and a `size` field. The data region–the region which is returned to the user–follows the header. Its length depends on the requested size, but is always rounded up to a multiple of 16 bytes.

A heap must be able to quickly satisfy new memory requests. It is therefore important to quickly identify free blocks. Your heap will keep all free blocks in a linked list, connected by the `next` pointers and choose the first block that is large enough. When a block is allocated, it is removed from the free-list, and when it is released, it is inserted into the free-list. In any case, all blocks (used and free blocks) can be iterated by going from one block to the next using the `size` field. The heap allocator splits/merges blocks as appropriate (see Figure 2).

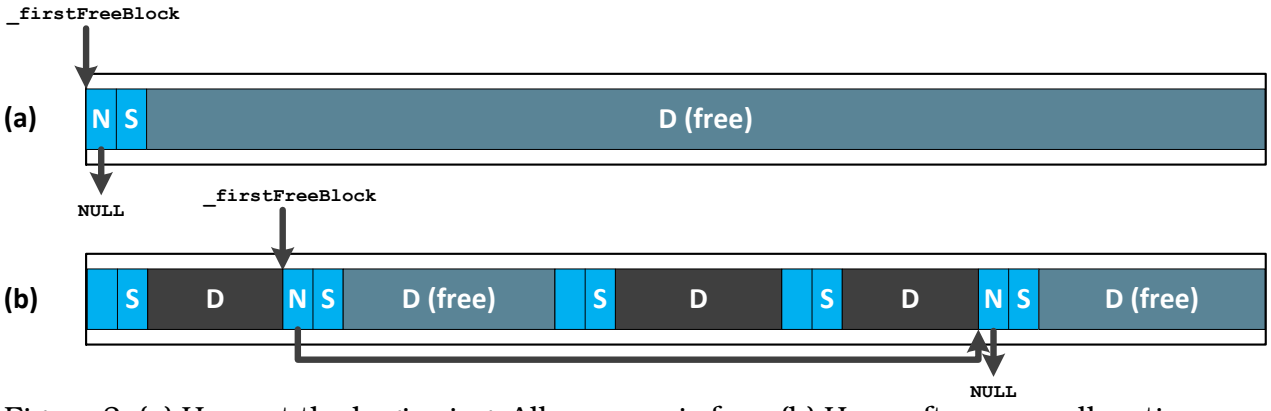

Figure 2: (a) Heap at the beginning. All memory is free. (b) Heap after some allocations and frees.

a. To ease the design, the heap allocator manages memory only at the granularity of 16 bytes. Write a function that rounds a given integer up to the next larger multiple of 16 (e.g., $6 \rightarrow 16$, $16 \rightarrow 16$, $17 \rightarrow 32$, etc.). **1 P-pt**

```
uint64_t roundUp(uint64_t n);
```

b. Implement the `my_malloc()` function that allocates memory from your heap. Do not use any external allocator. Your implementation shall implement the "first-fit" strategy (see lecture) and satisfy the following requirements: **5 P-pt**

- Uses your `roundUp()` to round the requested size up to the next larger multiple of 16.
- Searches the free-list to find the *first* free block of memory that is large enough to satisfy the request.
- If the free-list is empty or there is no block that is large enough, returns `NULL`.
- Otherwise, removes the free block from the free-list and returns a pointer to the beginning of the block's data area.
- If the free block is larger than the requested size, splits it to create two blocks: (1) One (at the lower address) with the requested size. This block is returned. (2) One new free block, which holds the spare free space. Don't forget to add it to the free-list by updating the next pointer of the previous free block. Free blocks that are only large enough to hold the header (i.e., 16 bytes) are valid blocks with a zero-length data region.

Hints: Adding or removing blocks from the free-list may require updating the free-list head pointer (`_firstFreeBlock`), which points to the first free block. If the list is empty, this pointer should be `NULL`. The last free block's `next` pointer should always be `NULL`. Before you submit your solution, experiment with different allocation patterns and print the heap layout with `dumpAllocator()`.

```
void *my_malloc(uint64_t size);
```

c. Implement the `my_free()` function that frees memory previously allocated with `my_malloc()`. Your implementation should satisfy the following requirements: **4 P-pt**

- Considers `my_free(NULL);` as valid and just returns.
- Otherwise, derives the allocation block from the supplied address.
- Inserts the block into the free-list at the correct position according to the block's address.
- If possible, merges the block with neighbor free blocks (before and after) to form a larger free block.

All hints of the previous question apply.

```
void my_free(void *address);
```

Total:
10 T-pt
10 P-pt