Operating Systems – spring 2024
Tutorial-Assignment 5

Instructor: Hans P. Reiser

Submission Deadline: Tuesday, Februar 20th, 2024 – 23:59

A new assignment will be published every week. It must be completed before its submission deadline (late policy for programming assignments: up to two days, 10% penalty/day)

Lab Exercisess are theory and programming exercises discussed in the lab class. They are not graded, but should help you solve the graded questions and prepare for the final exam. Make sure to read and think about possible solutions before the lab class.

T-Questions are theory homework assignments and need to be answered directly on Canvas (quiz).

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

Topic of this assignment is dynamic memory allocation. This week, the tutorial part will be shorter, allowing for more individual help on the P assignment. Start coding early this week! A good solution will require just adding around 50 lines of code (not counting comments), this is not that much, but if things fail, finding bugs can be more tricky this time.

Lab 5.1: Memory allocation

a. Exercises on bit operations in C: Implement a C function that performs the following operation efficiently with low-level bit operations (and, or, xor, shift)

- Write a function to divide an integer by 4 using bit operations.
- Write a function to multiply an integer by 8 using bit operations.
- Write a function to round down a number to the nearest multiple of 1024 ($1024 = 2^{10}$)
- Write a function to round up a number to the nearest multiple of 1024
- Write a function to test if a number is a power of 2.
- Write a function to test if the two least significant bits of a number are noth set

b. Assume you have the following definition of a header for a memory allocator:

```
#define ALLOCATED_BLOCK_MAGIC (0xfeedcafecafe)

typedef struct _Block {
        /* The size of this block, including the header
         * Always a multiple of 8 bytes. */
        uint64_t size;

        /* For a free block: a pointer to the next free block (or NULL for the last block)
         * For an allocated block: a magic number (ALLOCATED_BLOCK_MAGIC) */
        union {
                struct _Block *next;
                intptr_t magic;
        };

        /* The data area of this block.
         * (usually application data; in case of a free block, this part is unused) */
        uint8_t data[];
} Block;
```

The memory (heap) is organized as a sequence of consecutive blocks. The header's `size` value indicates the total size of a block (including the header). Implement a function

```
static Block *_getNextBlockBySize(const Block *current) { ... }
```

that takes a `Block *` as input and returns the reference to the next block in memory using the `size` information. See also Fig. 2 in the P-assignment.

Note: Iterating over blocks using the size information gives you all blocks, whereas the `next` pointer has the purpose of providing direct links to free blocks, potentially skipping over allocated blocks).

c. Implement a function that iterates (a) over all blocks (free and allocated) using the size information and (b) over all free blocks using the next pointer, printing the address and size of each allocation.

T-Question 5.1: Memory Allocation

a. What is the difference between internal and external fragmentation in memory management? Select the most accurate explanation. **2 T-pt**

true	false	
☐	☐	Internal fragmentation occurs when the size of a memory block is smaller than the size of the data it should hold, while external fragmentation occurs when the size of a memory block is larger than the size of the data it should hold.
☐	☐	External fragmentation causes memory allocation requests to fail even if the sum of the sizes of all free blocks is sufficiently large (due to fragmentation into several, smaller blocks). Internal fragmentation causes memory allocation requests to fail due to a lack of available free blocks, even if there was enough unused space within allocated blocks, due to over-allocation of memory.
☐	☐	Internal fragmentation is memory fragmentation caused by memory allocations performed by your application (thus internal). External fragmentation is memory fragmentation caused by memory allocations performed by other applications in a separate address space (thus external).
☐	☐	Internal fragmentation occurs when there is not enough contiguous free memory to hold new data, while external fragmentation occurs when free memory is not used because it is too small to hold new data.

b. You cannot easily mitigate external fragmentation on the heap (dynamically allocated memory in a process) with compaction. Select the most appropriate answer that explains why! **2 T-pt**

true	false	
☐	☐	The heap is a memory area that does not suffer from external fragmentation, so there is no need for compaction.
☐	☐	Compaction can only be performed on memory blocks that are freed (moving free blocks such that they become contiguous and can be merged), but external fragmentation occurs because of the placement of allocated blocks, which cannot be fixed with compaction.
☐	☐	Compaction requires moving allocated memory blocks to different locations in order to remove fragmentation. Changing the location of data requires updating all references to that data (pointers in memory, CPU registers, etc.) to be updated to reflect the new location, and this is (usually) infeasible, because the memory allocator does not have information about where such references are stored.
☐	☐	Compaction is not effective in reducing external fragmentation on the heap because it can cause internal fragmentation, which can lead to further memory allocation failures.

c. Why would you prefer the allocation policy first fit over best fit? Chose the most appropriate answer! **1 T-pt**

true	false	
☐	☐	First fit is generally faster and simpler to implement, and usually (almost) as good as best fit.
☐	☐	First fit provides better memory utilization (less internal fragmentation) than best fit.
☐	☐	First fit has a higher run-time complexity than best fit, but the resulting allocation is, in general, superior because of reduced external fragmentation.
☐	☐	First fit uses less memory than best fit.

d. Assume you want to implement a memory allocator using a bitmap for managing allocated and free memory allocation units. Which piece of information cannot extracted from the bitmap, and thus needs to be managed separately? **1 T-pt**

true	false	
☐	☐	Bitmaps can be used to store which memory areas are free, but not which memory areas are allocated, so a bitmap always needs to be combined with a allocated blocks linked list.
☐	☐	Bitmaps can be used to store which memory areas are allocated, but not which memory areas are free, so a bitmap always needs to be combined with a free blocks linked list.
☐	☐	A bitmap does not store the length of an allocation: Multiple adjacent allocations cannot be distingushed from a single larger allocations. Thus, the length of allocations needs to be stored in an additional datastructure.
☐	☐	A bitmap does not store the length of free blocks: Multiple adjacent free blocks cannot be distingushed from a single larger free block. Thus, the length of free blocks needs to be stored in an additional datastructure.

e. What is the buddy allocator and what problem does it solve? Select the most accurate answer. **2 T-pt**

true	false	
☐	☐	The buddy allocator is a memory allocator that allocates memory in fixed size blocks. As all allocations are of identical size, external fragmentation cannot occur.
☐	☐	The buddy allocator is a memory allocator that splits blocks into smaller blocks when they are freed. This approach makes it more efficient to handle variable-size allocations.
☐	☐	The buddy allocator is a memory allocator that allocates memory in blocks of sizes that are powers of two. Compared to an allocator using arbitrary variable allocation sizes, this approach may result in higher internal fragmentation, but reduces external fragmentation.
☐	☐	The buddy allocator is a memory allocator that groups memory allocations for objects of a specific type. The advantage of the buddy allocator is a better cache locality.

f. Let's assume you have a memory allocator that uses a linked list of unallocated blocks as free-list. You have the following three blocks on free list (notation as on lecture slide 8 of part 06):

head → (addr:10, len=5) → (addr:0, len=10) → (addr:20, len=10) → NULL

This state was reached after free()-ing a block and re-inserting it into the free-list. To minimize fragmentation, adjacent free blocks should be merged. What is the best possible result of such merge operation? **2 T-pt**

true	false	
☐	☐	head → (addr:0, len=15) → (addr:20, len=10) → NULL
☐	☐	head → (addr:10, len=5) → (addr:0, len=10) → (addr:20, len=10) → NULL
☐	☐	head → (addr:0, len=30) → NULL
☐	☐	head → (addr:0, len=25) → NULL

P-Question 5.1: 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).

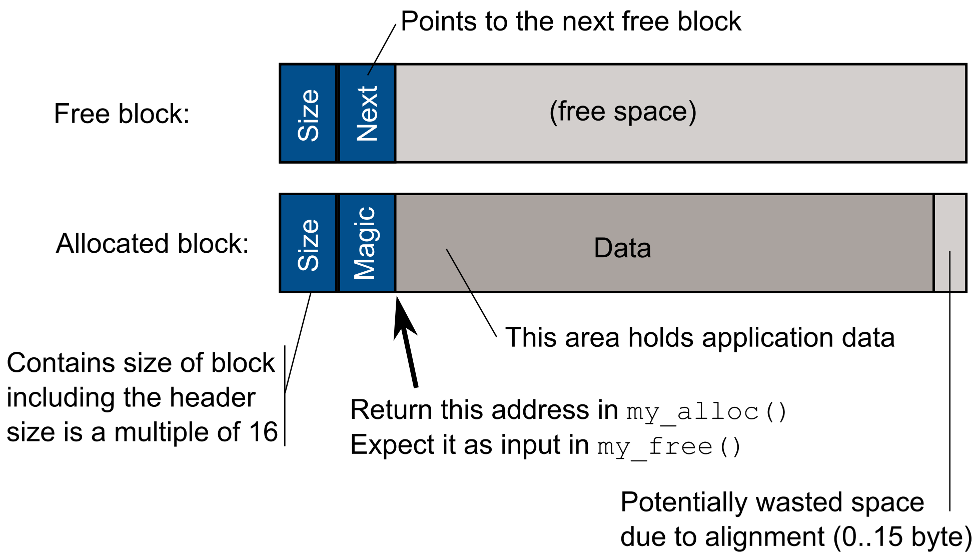
Points to the next free block
Free block:
Size
Next
(free space)
Allocated block:
Size
Magic
Data
This area holds application data
Contains size of block
including the header
size is a multiple of 16
Return this address in my_alloc()
Expect it as input in my_free()
Potentially wasted space
due to alignment (0..15 byte)

Figure 1: Heap block

Each block starts with a header (16 bytes), which describes the block. The header contains a `size` field as well as either a `magic` number (for allocated blocks) or a `next` pointer to the next free block (for free blocks). 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 allocator 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 (first fit)**. When a block is allocated, it is removed from the free-list, and when it is released, it is inserted into the free-list. The free-list is ordered by block address 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).

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.

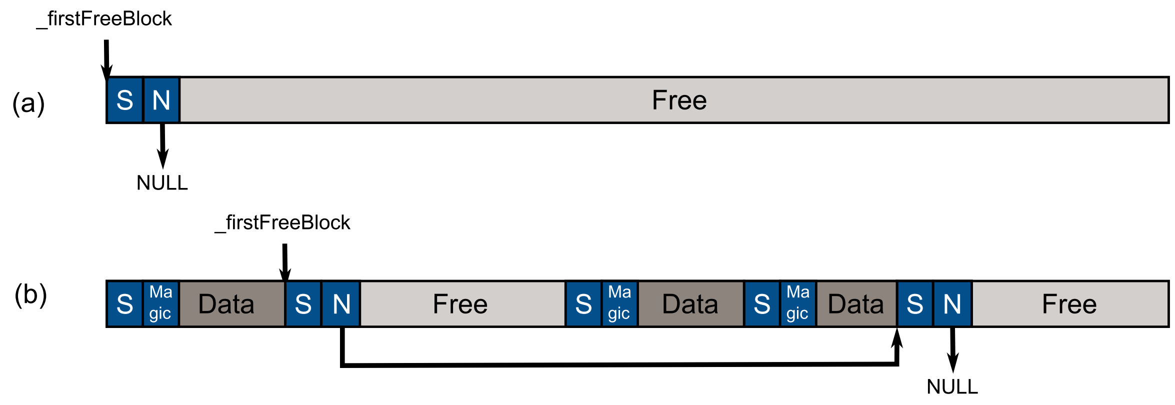

Figure 2: (a) Heap at the beginning. All memory is free. (b) Heap after some allocations and frees.

- 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`.
- If the free block is of exactly the requested size, removes the free block from the free-list, marks it with the magic number, 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. It is marked as used block with the magic number and returned.
  (2) One new free block, which holds the spare free space. Don't forget to put it into the free-list (instead of the original free block) by updating the next pointer of the previous free block (or `_firstFreeBlock`).
  Free blocks that are only large enough to hold the block header (i.e., 16 bytes) are valid blocks with a zero-length unused 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**

- If `address` is NULL, it does nothing and just returns.
- Otherwise, it derives the allocation block from the supplied address.
- It inserts the block into the free-list at the correct position according to the block's address.
- If possible, it 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