Operating Systems 2018/19
Assignment 3

Prof. Dr. Michael Beigl
Mathias Gottschlag, M.Sc.
Jonas Kiemel, M.Sc.

Submission Deadline: Sunday, November 18th, 2018 – 23:59

A new assignment will be published every week, right after the last one was due. It must be completed before its submission deadline.

The assignments must be filled out online in ILIAS. Handwritten solutions are no longer accepted. You will find the online version for each assignment in your tutorial's directory. **P-Questions** are programming assignments. Download the provided template from ILIAS. Do not fiddle with the compiler flags. Submission instructions can be found on the first assignment.

In this assignment you will get familiar with threads and different thread models as well as learn how to implement a dynamic memory allocator.

T-Question 3.1: Threads and Thread Models

a. What is the difference between a PCB and a TCB? **1 T-pt**

b. The lecture and tutorials introduced two fundamental types of threads, depending on where the thread is implemented and where it executes. What are these two types? Give a short explanation for each. **2 T-pt**

c. Give two disadvantages of the many-to-one thread model. **1 T-pt**

d. Explain the basic concept of a hybrid thread model. **2 T-pt**

e. Why does a switch to a thread of a different process normally take longer than a switch to a thread in the same process? **1 T-pt**

P-Question 3.1: Simple Heap Allocator

Download the template **p1** for this assignment from ILIAS. 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).

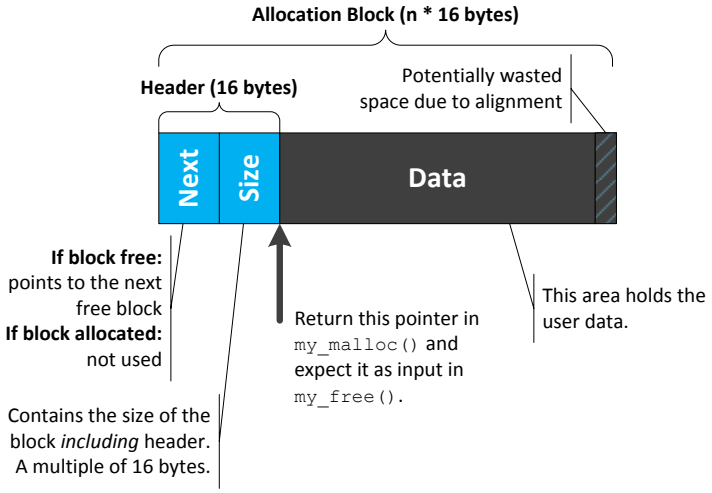

Abbildung 1: Heap block

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, which is large enough. When a block is allocated, it is removed from the free-list, when it is released, it is inserted into the free-list. In any case, 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).

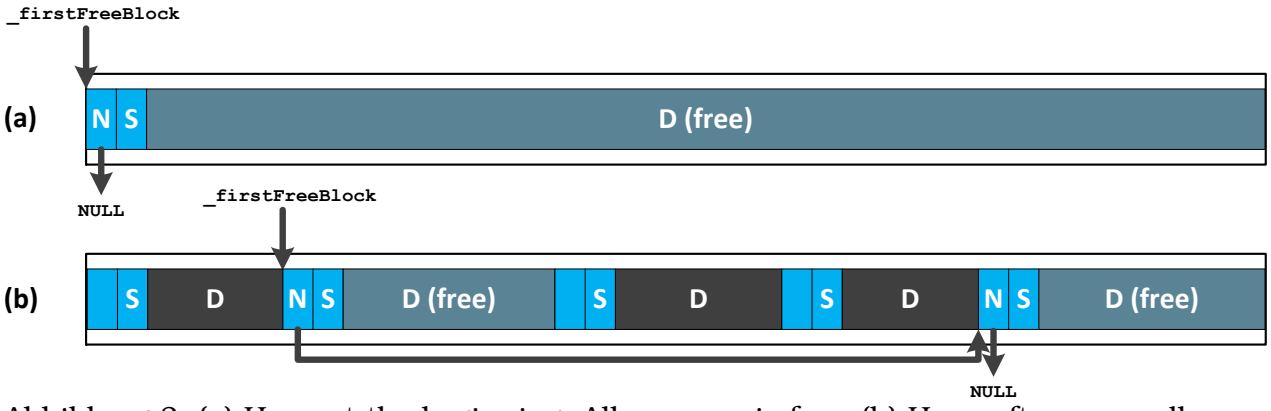

Abbildung 2: (a) Heap at the beginning. All memory is free. (b) Heap after some allocations and frees.

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 should satisfy the following requirements: **4 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: **3 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);
```

P-Question 3.2: User-Level Threads

Download the template **p2** for this assignment from ILIAS. You may only modify and upload the file `dispatcher.c`.
This question expects a x86-64 (64-bit) CPU, a 64-bit operating system and the GNU GCC toolchain. Do not change the compiler flags provided by the Makefile. You can use computers from the ATIS if necessary.

Threads are a fundamental abstraction of the CPU provided by the operating system. In this question you will write your own user-level thread dispatcher that will be able to create, start and switch between user-level threads.

a. Implement the `yield()` function that performs the switch from one user-level thread to another. Your implementation should satisfy the following requirements: **3 P-pt**

- Pushes the CPU registers `rbp`, `rbx`, and `r12 - r15` to the current thread's stack. *Note:* By the calling convention, these registers are callee-saved, which means that the function which modifies them, needs to save them. Since the switch with `yield()` is voluntary, only these registers need to be saved. All other registers are–because of the calling convention–saved (if in use) by the caller of `yield()`.
- Saves the current stack pointer (`rsp`) in the current thread control block (`_threads[_prevThread]`).
- Switches to the stack of the next thread by writing the `rsp` register.
- Restores the previously pushed registers from the stack of the next thread.

Hints: To implement the necessary actions you need to write GCC inline assembler. Use the assembler instructions `pushq <register>` and `popq <register>` to push/-pop a register to/from the stack, and `movq <src>, <dest>` to perform register/register or register/memory data transfers. Complete each line of inline assembler with `\n\t`. Example: `pushq %%rax\n\t`.

```
void yield();
```

b. Implement the `startThread()` function that initializes a new user-level thread and prepares its stack to start execution in the user supplied main function after returning from `yield()`. When the thread exits its main function, it should automatically execute `_parkThread()`. **3 P-pt**

Hints: Build the program with your implemented `yield()` function. Then call `objdump -Sd dispatcher` and locate the assembler code you have added to `yield()`. You will notice that the compiler added two more assembler instructions at the end of the function:

```
40090a:  c9  leaveq  ; Look in Intel Instruction Set Manual for LEAVE
40090b:  c3  retq    ; pops the return address and performs the jump
```

A fully functional stack for a new thread has to contain values for these instructions, a valid return address for the thread's main function, and values for the callee-saved registers. Some of these values can be left uninitialized. The resulting stack should look like Figure 3.

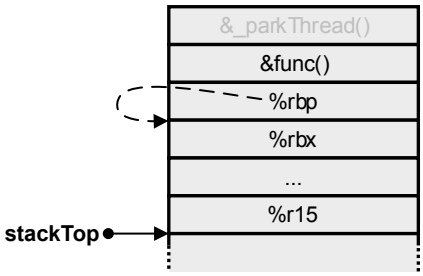

Abbildung 3: Stack layout of a new thread

Note that `rbp` is read from the stack twice: Once as part of the callee-saved registers, and once by `leaveq`. As the figure shows, one entry for `rbp` on the new stack is sufficient, though: Making the corresponding stack entry point at itself causes `leaveq` to read the same stack entry a second time.

```
int startThread(void (*func)(void));
```

Total:
7 T-pt
14 P-pt