Operating Systems 2016/17
Tutorial-Assignment 4

Prof. Dr. Frank Bellosa
Dipl.-Inform. Marc Rittinghaus

Question 4.1: Understanding Assignment P3.1

a. Recap how you can round up a given integer to the next larger multiple of a predefined power of two (e.g., 16) in C with simple integer and binary arithmetic only.

Solution:
We can round up x to a multiple of n, where n is a power of two, by (1) adding $n-1$ to x and (2) masking out the bits 0 to $(\log_2 n)-1$. In C, we can express this operation as:
`x = (x + n - 1) & ~(n - 1);`

This will perform the following steps, if $x = 13$ and $n = 16$:

```
      x          0000 1101 = 13
(+)   n - 1      0000 1111 = 15
      ------------------
                 0001 1100 = 28
(&)  ~(n - 1)    1111 0000
      ------------------
                 0001 0000 = 16
```

b. Look at the declaration of `Block`. Why is the `padding` field required? How does it work?

```
typedef struct _Block {
    struct _Block *next;
    uint8_t padding[8 - sizeof(void*)];
    uint64_t size;
    uint8_t data[];
} Block;
```

Solution:
The programming assignment requires that the header has a fixed size of 16 bytes, irrespective of the platform (x86 or x86-64) on which the code is built. The `next` field, however, is a pointer and its size depends on the platform (32 bits = 4 bytes or 64 bits = 8 bytes). The structure size therefore also depend on the platform. You can easily validate this by removing `padding` and printing the value of `sizeof(Block)` with `printf` or a debugger. Note: You can build a 32 bit program on a 64 bit system with `gcc` by using the `-m32` switch as shown in the lecture.

On 32 bit platforms `padding` declares an array of 4 bytes. On the 64 bit platforms, the array has 0 elements and thus occupies no space. `next` and `padding` always occupy together 8 bytes. As a result, `size` is always at the same memory offset within the structure.

c. What are the major steps for allocating memory as requested by the assignment? Where can you find the respective operations in the solution?

Solution:
The major steps are:

(a) *Round up the requested size up to a multiple of 16 (line 153).*

(b) *Search the list of free blocks for a block that is large enough (lines 157 - 166).*

(c) *Return NULL if no (properly sized) free block can be found (lines 169).*

(d) *If the free block's size exactly matches, simply remove it from the free list (lines 125, optionally 142).*

(e) *Otherwise, split the free block (lines 127 - 138).*

(f) *Return a pointer to the allocated data space (line 144).*

d. During allocation, a free block might be split into two blocks: one for the allocated space and one for the remaining free space. Can you think of a reason, why the assignment prefers returning the first block, although the implementation might be easier the opposite way?

Solution:
The operating system usually backs virtual memory with physical memory only after the first access. This concept will be later introduced in the lecture as demand-paging. When comparing both variants (see Figure d), we can see that using the first block will lead to better reuse of memory. This is because the assignment dictates that always the first large-enough block should be chosen.

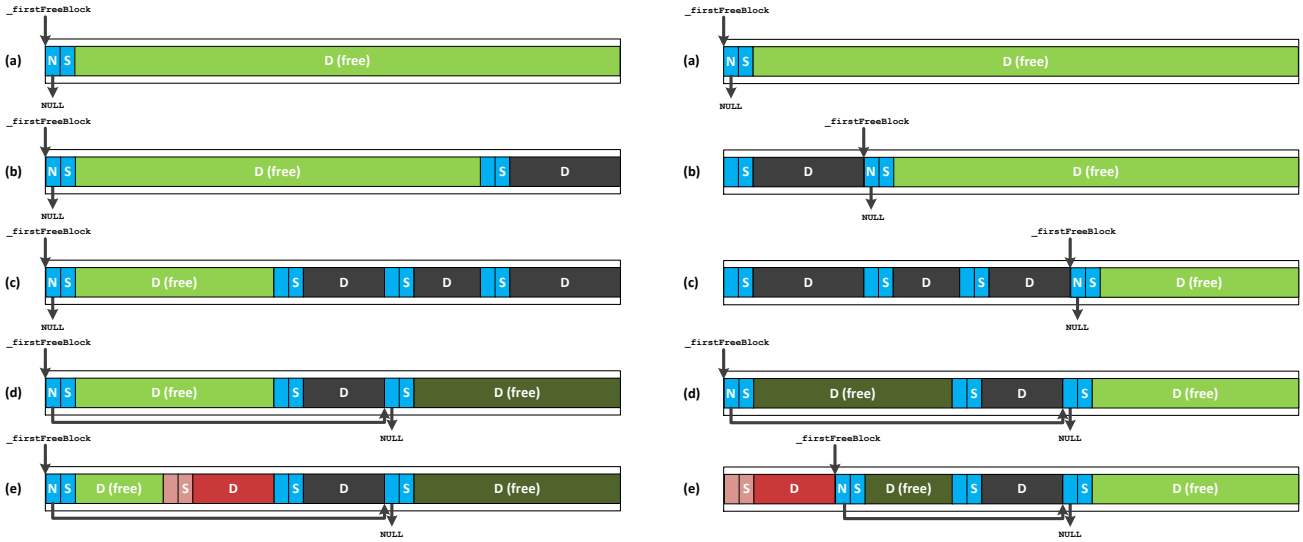

Abbildung 1: Left side shows example allocations and frees with second block return, right side shows same sequence with first block return. The last allocation is marked in red. Untouched free space is marked in light green, already touched space in dark green. First block return makes better reuse of memory.

e. What are the major steps for freeing memory as requested by the assignment? Where can you find the respective operations in the solution?

Solution:
The major steps are:

(a) Check if the caller supplied `NULL` and return (line 200 - 202).

(b) The caller provided an address to the data space of a block. To free the block, we need to get its header (line 210).

(c) Find the right position in the free list (lines 218, 226 - 228).

(d) Insert the block in the free list (lines 220,221,231,232).

(e) Merge the new free block with neighbor blocks if possible (lines 175 - 196, 223, 234, 235).

f. In the solution you will find many `assert` statements. What is their purpose? Why doesn't the solution use regular `if` statements instead?

Solution:
Assertions are designed for debugging purposes and aid in finding errors during the development phase of a software. Assertions should express the expectations that the developer had, when writing the code. Consequently, they usually check against conditions that should never happen. Assertions can therefore be used to check parameters and conditions within internal functions. `if` statements on the other side should be used to check for conditions that might happen. They are thus suited for example to perform parameter checking in publicly accessible functions (API functions).

An exception from the rule are interfaces that state undefined behavior for invalid input (e.g., see `free` in `man malloc`). Although this is bad practice, many functions in C use this approach. Some implementations of the C library add assertions to help identifying illegal parameters, because triggering undefined behavior often leads to hardly detectable bugs. An example is given in the solution with the first two assertions in `my_free`, lines 204 and 205.

In release builds assertions are typically deactivated, which means the compiler omits the checks.

Question 4.2: Understanding Assignment P3.2

a. Consider the following code from `dispatcher.c`, lines 122 to 136. Explain what components make up the assembly fragment.

```
__asm__ __volatile__ (

    "nop" // No-operation

    : [prevSp] "=m" (_threads[prevThread].currentSP)
    : [newSp] "m" (_threads[_currentThread].currentSP)
    : "cc", "memory"
);
```

Solution:
The code fragment uses the inline assembly feature of the GCC compiler. It gives the developer the ultimate control on what instructions are generated. This way CPU registers can be directly accessed. Inline assembly is also sometimes used to gain access to special CPU instructions that are generally not generated by the compiler. An example is the `sysenter` instruction as trap.

The first part of the fragment provides space for the instructions. After the first colon, a comma-separated list of output operands (i.e., variables modified by the instructions) is defined.

The next colon introduces a comma-separated list of input operands. In contrast to the output operands, input operands can be regular C expressions. Since we want to exchange the stack pointer register by storing the current value in the currently running thread's TCB (`prevThread`), and reading the new value from the next thread's TCB `_currentThread`.

The last list makes the compiler aware of changes to entries other than the output operands. This is important, because the assembly will be inlined with code generated by the compiler. The compiler therefore needs hints from the developer on what state is changed by the assembly so it can adjust code before and after the assembly accordingly (e.g., save/restore registers).

The `cc` and `memory` clobber arguments inform the compiler that the flags register may be changed and that the assembly may write to memory locations other than specified by the output operands (e.g., pushing the registers of the current thread onto its stack). The arguments prevent the compiler from performing optimization such as moving memory accesses that appear in the source code before the assembly, after the assembly.

The `__volatile__` statement at the beginning of the code fragment further disables all other potential optimization by the compiler, such as moving or omitting the assembly.

For more details see `https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html`.

b. Describe how the stack develops during the execution of the `yield` function. Visualize the stack contents for interesting points.

Solution:
See the document `assignment03-stack.pdf`.