Operating Systems

Tutorial 8

Fabian Klopfer

21. Januar 2021

Intro

- Pingo Polls
- No new exercise sheet this week
- Loaders: The broad context & Linking

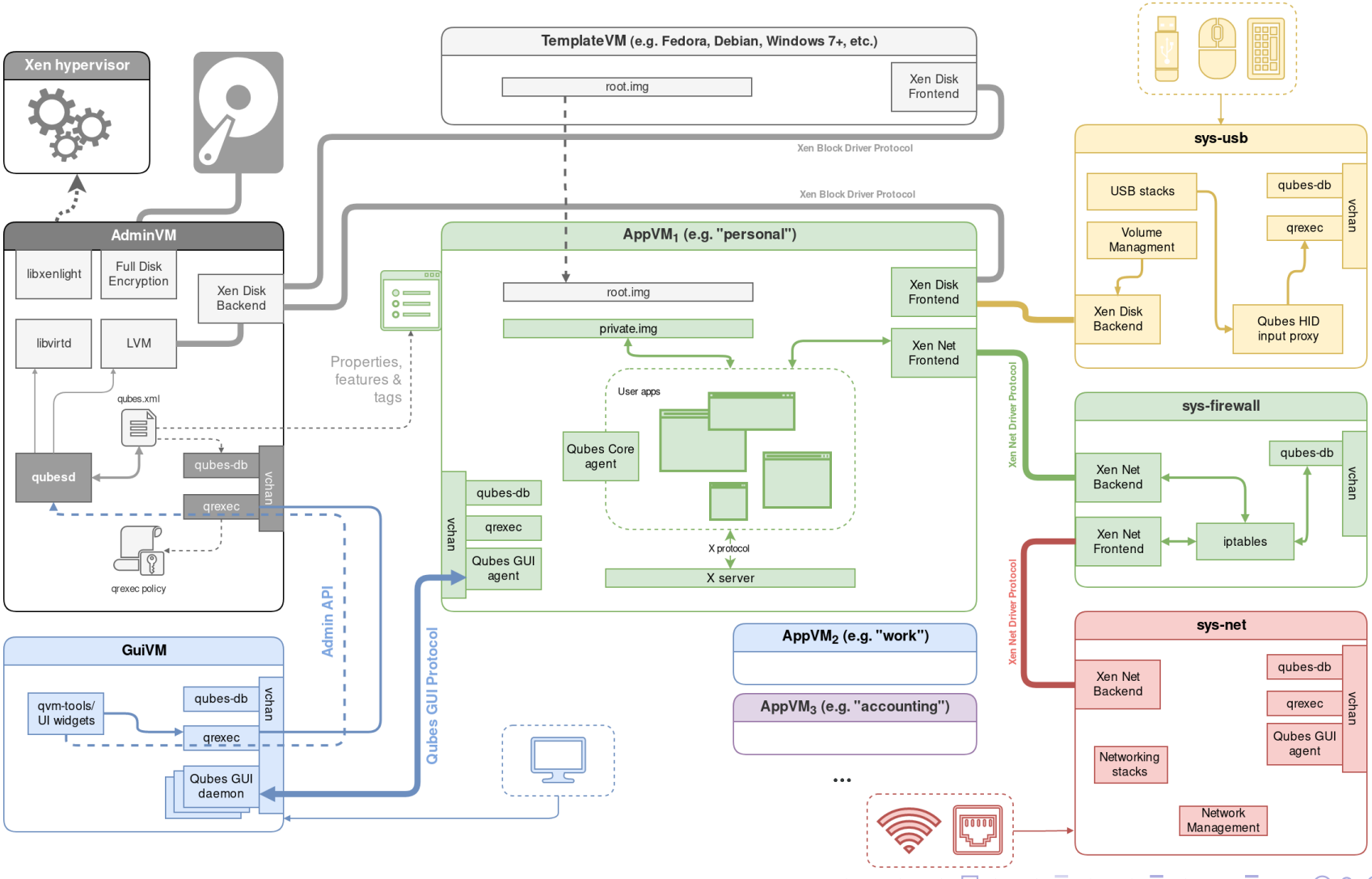
Exercise Sheet 7

Exercise 1

Exercise 1 I

1. How could virtualization on a desktop computer be useful to developers?
   - Testing
   - Avoid dual boot setups
   - Security through isolation: QubesOS

Exercise 1 II



Exercise 1 III

2. Which of the following instruction groups is not sensitive in the sense of Popek & Goldberg (1974)?
   - 2.1 Instructions that access the memory management unit (MMU),
   - 2.2 instructions that access the input/output MMU (I/O MMU),
   - 2.3 instructions that access the Arithmetic-Logical Unit (ALU).

   instructions that access the Arithmetic-Logical Unit (ALU).
3. What is a privileged instruction in the sense of Popek & Goldberg (1974)?
   Privileged instructions hand over the control to the OS via a trap.
4. Can a hypervisor virtualize more operating systems than there are (hard)disk partitions in the computing system? Why?
   Yes, hypervisors can create an arbitrary number of Subpartitions

Exercise 1 IV

5. Give an example in the context of virtualization where binary translation can be faster than the original code.
   Clear Interrupt instructions:
   - On HW: trap to hypervisor
   - Using BAT: Changing a flag in memory
6. What is the main problem with virtualization and memory?
   Multiple Guests allocate frame k at the same time $\Rightarrow$ Problem
   Solution: Another page table/mapping: Guest frame to Hypervisor/actual frame
7. What problem does the hypervisor face when it wants to evict a page of a virtualized operating system?
   May evict page that is frequently used by guest.

Exercise 1 V

8. What is the purpose of balloon drivers?
   Hypervisor does not know page usage of guests
   Workasrround: Balloon drivers in the guests.
   If hypervisor runs out of space the ballon driver increases its memory usage, forcing the guest to evict a page.

Exercise 2

Exercise 2 I

1. What does `int * const p1;` declare?

   Unchangeable pointer that can be used to change the pointed-to integer.

2. What does `int const * p2;` declare?

   A changable pointer that cannot be used to change the pointed to integer.
   Same as `const int* p2`

3. What does `int (*p3)[3];` declare?

   Pointer to an array of 3 integers.

4. What does `float f1(int const * p);` declare?

   A function that returns a float, taking a pointer to an unchangeable integer as argument.

Exercise 2 II

5. Is the function signature `void f2(int * const p);` meaningful? Why?

   As pointers are passed by value (the object they point to are passed by reference), a local variable within the scope of the function is created. By the const one cannot change where the pointer points to. The local variable can not be changed. Meaningful? Sort of, but not really.

6. What is the problem in the following code snippet?

```
void f(char *p) { *p = 'x'; }
...
f("Hello world!");
```

   "Hello World" is a compile time constant, i.e. it will be put into the binary/code segment of the process which is **read-only**.
   Changing it as in f() segfaults

Exercise 2 III

7. Declare a function that returns a pointer to an `int` array with 3 elements and accepts no arguments.

```
int (*f(void))[3];
```

8. Declare a pointer to this function.

```
int (*(*p4)(void))[3];
```

Exercise 3

Exercise 3 I

A *palindrome* is a string which is the same when read forwards or backwards:

$$\forall i = 1, \ldots, |s| : s_i = s_{|s|-i+1}$$

Where $|s|$ denotes the length of the string s. Write a function with the signature `int is_palindrome(char const* s)` that tests whether a string is a palindrome.

```
int is_palindrome(char const* s) {
    char const* p = s + strlen(s);

    while (s < p) {
        if (*s++ != *(--p)) {
            return 0;
        }
    }

    return 1;
}
```

Exercise 4

Christmas animation I

Write a program that outputs a seasonally appropriate animation to the text console, for example, a landscape with snow falling.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <time.h>

#define NCOLS 79
#define NROWS 25
#define SNOWFLAKE '*'

void run_simulation() {
    char world[NROWS * NCOLS];
    memset(world, ' ', NROWS * NCOLS);

    while (1) {
        print_frame(world);
        simulate_timestep(world);
        usleep(0.35 * 1000000);
    }
}

int main(void) {
    srand(time(NULL)); // seed PRNG with current time
    run_simulation();

    return 0;
}
```

Christmas animation II

```
void simulate_timestep(char* world) {
    // add one new snow flake
    int pos = rand() / ((RAND_MAX + 1u) / NCOLS);
    world[pos] = SNOWFLAKE;

    // float down snowflakes
    for(int row = NROWS - 2; row >= 0; --row) {
        for(int col = 0; col < NCOLS; ++col) {
            if (world[row * NCOLS + col] == SNOWFLAKE
              && world[(row + 1) * NCOLS + col] == ' ') {
                world[row * NCOLS + col] = ' ';
                world[(row + 1) * NCOLS + col] = SNOWFLAKE;
            }
        }
    }
}
```

Christmas animation III

```
void print_frame(const char* world) {
    // ANSI sequence clears screen and repositions cursor
    printf("\033[2J\033[1;1H");

    // print rows
    for(int row = 0; row < NROWS; ++row)
        printf("%.*s\n", NCOLS, world + row * NCOLS);

    printf("\nPress ctrl+c to abort...");
    fflush(stdout);  // flush to print last line
}
```

Christmas animation IV

Loader

We want to know what happens when we run code exactly.
First we write some code:

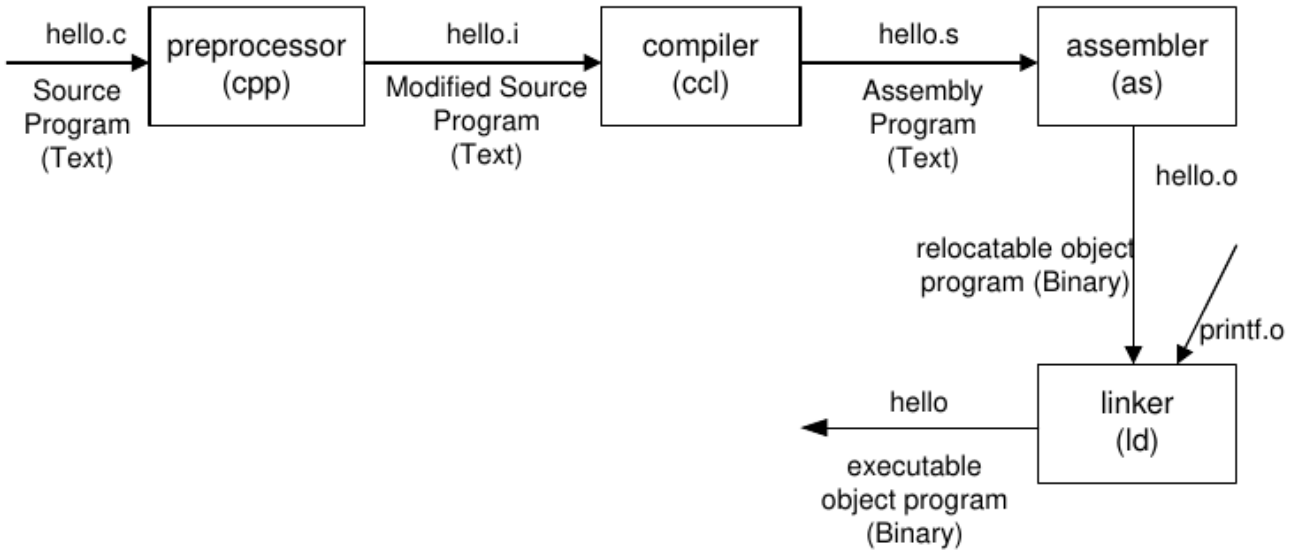
```c
#include <stdio.h>

int main()
{
        printf("Hello world!\n");
        return 0;
}
```

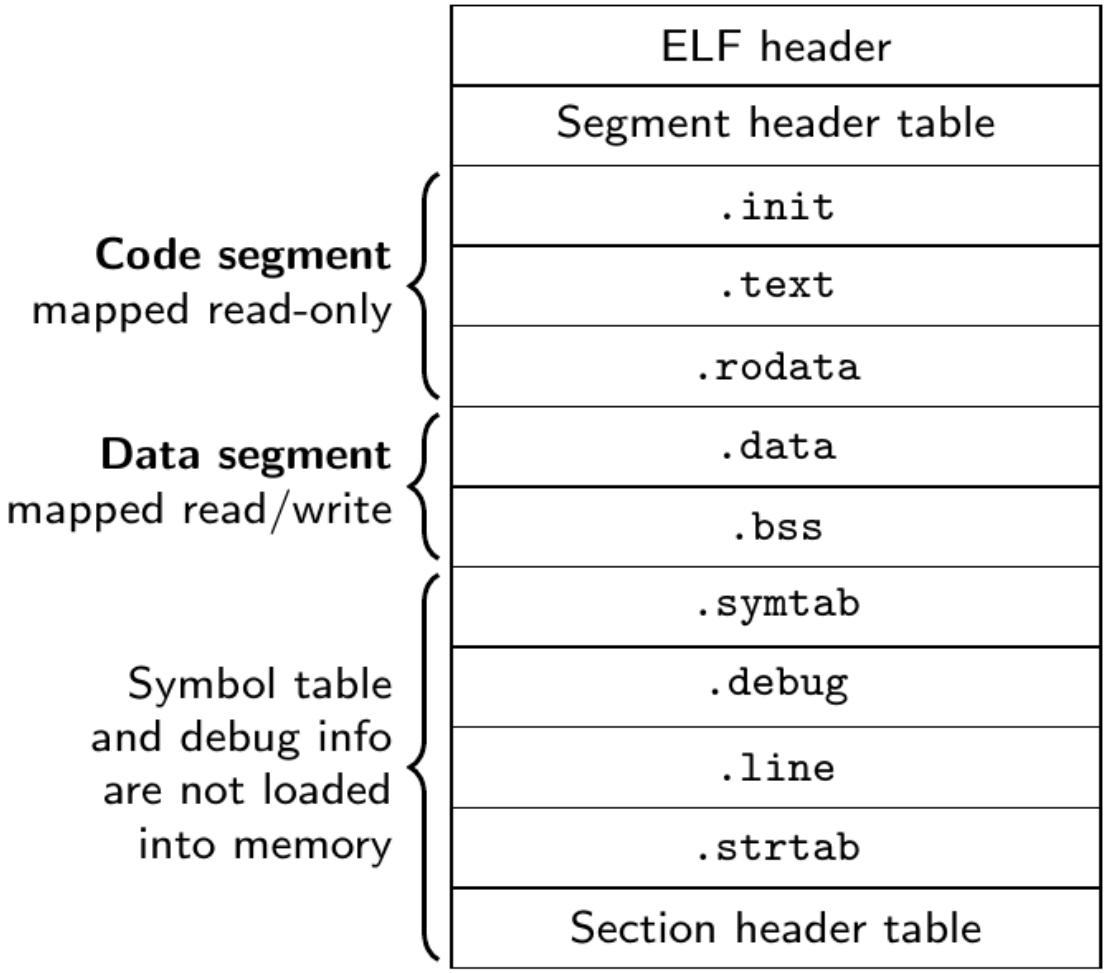
We compile it:

```
gcc hello_world.c -o hello
```

What actually happens:



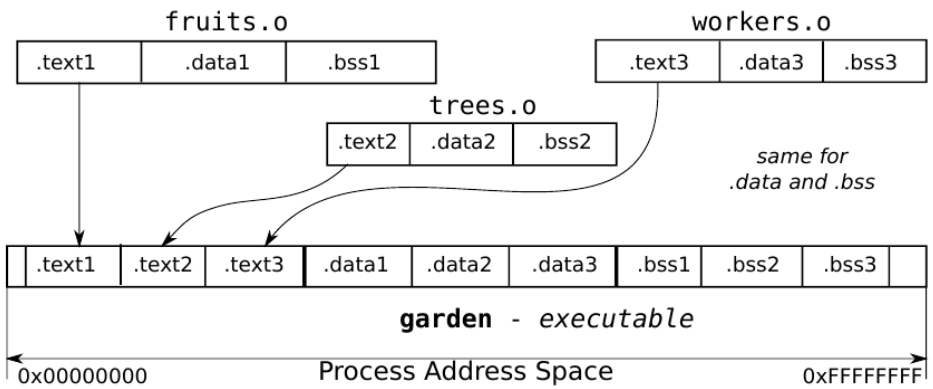
The assembler produces binary code in “Executable and Linking Format” (ELF) ...
Which looks like this:


ELF header
Segment header table
.init
.text
.rodata
.data
.bss
.symtab
.debug
.line
.strtab
Section header table
Code segment
mapped read-only
Data segment
mapped read/write
Symbol table
and debug info
are not loaded
into memory

What does the linker actually do?

Given: A file in ELF format with unresolved symbols and paths to libraries[1]
Desired: A file in ELF format with resolved symbols.
What it does: Scan the files for the appropiate symbol and replace it accordingly.

- So after resolving the symbols, the linker needs to put all the code from the individual object files' sections into the final program's sections:



[1] defaults: `/lib`, `/usr/lib`

Taking a closer look at (the relevant part of) the .text section:

```
0000000000001139 <main>:
    1139:        55                          push   %rbp
    113a:        48 89 e5                    mov    %rsp,%rbp
    113d:        48 8d 3d c0 0e 00 00        lea    0xec0(%rip),%rdi
    1144:        e8 e7 fe ff ff              callq  1030 <puts@plt>
    1149:        b8 00 00 00 00              mov    $0x0,%eax
    114e:        5d                          pop    %rbp
    114f:        c3                          retq
```

We can see that puts is called. If we look at the symbol table (.symtab) we notice:

```
50: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@@GLIBC_2.2.5
```

puts is undefined after linking!

But wait! We linked it and `puts` is still undefined?!

Yes it is and that's totaly fine:

Executables vs. Static libraries (`.a`) vs. shared libraries (`.so`)

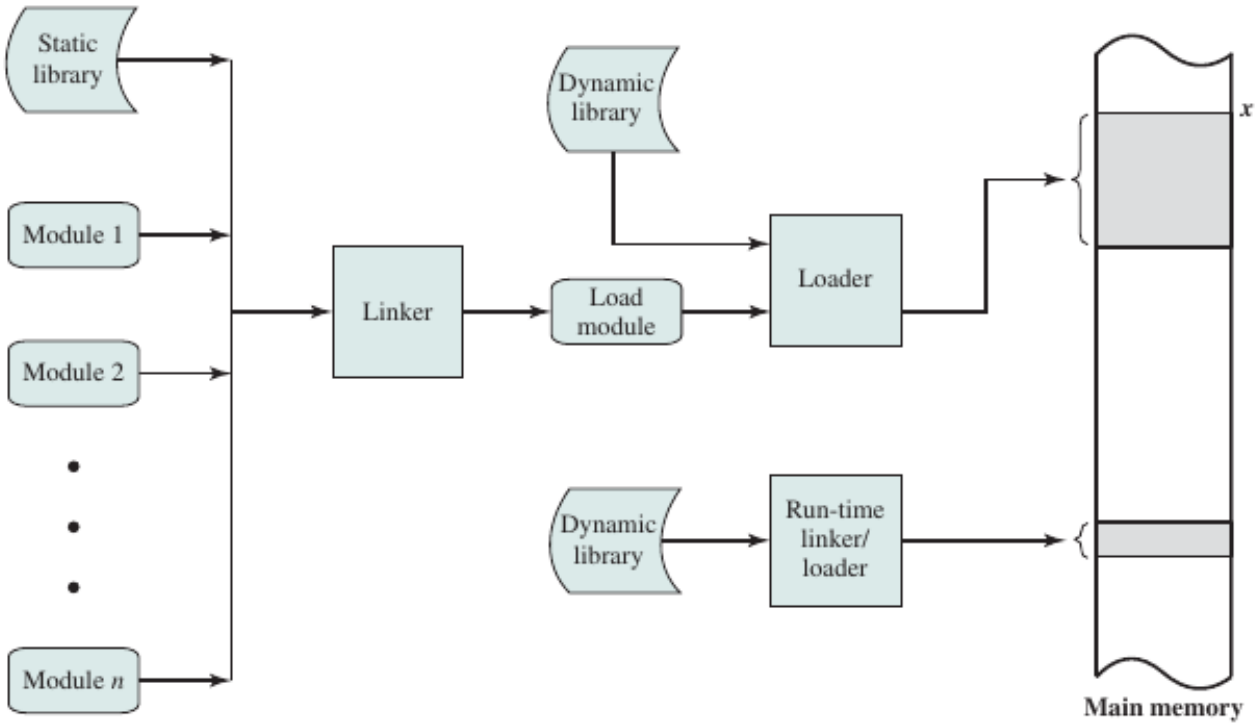


Figure 7.16 **A Linking and Loading Scenario**

More comprhensive linking description: Programming slides, last section
Thus distingiush between static and dynamic executables.

Our hello executable is a dynamic executable:

```
$ file hello
hello: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=39f3f7b380d05d9929a4bd46b87b1355c6a4ed1b,
for GNU/Linux 3.2.0, not stripped
```

The Linnker just marks what's to be done for the loader:

Linkers link static libraries and other source files.
Additionally they mark symbols from dynamic libraries for the loader.

The Loader links dynamic libraries/shared objects/dlls, previously marked by the linker

But what about the rest?

Lets execute our elf file: `./hello`
What happens under the hood of the shell is (pseudo code)

```
switch(pid = fork()) {
    case -1:
        exit(-1);

    case 0: // child
        execve("./hello", NULL, NULL);
        exit(0);

    default: // parent
        waitpid(pid)
}
```

I.e. copy the process image using fork and replace the relevant stuff using execve

The `execve` system call builds a an instanct of the `binprm` struct

Then iterates over available binary handlers using `search_binary_handler`.

Besides ELF there are handlers for e.g. scripts and jars
Finally `load_elf_binary(struct binprm* bin)` is called.

To be continued

References I

[1] W. Stallings, *Operating systems: internals and design principles*. Upper Saddle River, NJ: Pearson/Prentice Hall, 2009.
[2] A. Silberschatz, P. B. Galvin und G. Gagne, *Operating system principles*. John Wiley & Sons, 2006.
[3] A. S. Tanenbaum und H. Bos, *Modern operating systems*. Pearson, 2015.
[4] B. W. Kernighan und D. M. Ritchie, *The C programming language*. 2006.
[5] D. Trugman. (6. Jan. 2021), „ELF Loaders, Libraries and Executables on Linux | by Daniel Trugman | Medium“, Adresse: https://medium.com/@dtrugman/elf-loaders-libraries-and-executables-on-linux-e5cfce318f94 (besucht am 06. 01. 2021).
[6] (6. Jan. 2021). „ld.so(8): dynamic linker/loader - Linux man page“, Adresse: https://linux.die.net/man/8/ld.so (besucht am 06. 01. 2021).
[7] (6. Jan. 2021). „ld(1): GNU linker - Linux man page“, Adresse: https://linux.die.net/man/1/ld (besucht am 06. 01. 2021).
[8] (25. Nov. 2019). „How programs get run [LWN.net]“, Adresse: https://lwn.net/Articles/630727/ (besucht am 06. 01. 2021).
[9] (8. Sep. 2018). „How programs get run: ELF binaries [LWN.net]“, Adresse: https://lwn.net/Articles/631631/ (besucht am 06. 01. 2021).
[10] (30. Nov. 2020). „A Whirlwind Tutorial on Creating Really Teensy ELF Executables for Linux“, Adresse: http://www.muppetlabs.com/~breadbox/software/tiny/teensy.html (besucht am 06. 01. 2021).
[11] (6. Jan. 2021). „linux/binfmt_elf.c at fcadab740480e0e0e9fa9bd272acd409884d431a ů torvalds/linux ů GitHub“, Adresse: https://github.com/torvalds/linux/blob/fcadab740480e0e0e9fa9bd272acd409884d431a/fs/binfmt_elf.c (besucht am 06. 01. 2021).
[12] (21. Dez. 2020). „elf(5) - Linux manual page“, Adresse: https://man7.org/linux/man-pages/man5/elf.5.html (besucht am 06. 01. 2021).
[13] (6. Jan. 2021). „Linux Foundation Referenced Specifications“, Adresse: https://refspecs.linuxfoundation.org/ (besucht am 06. 01. 2021).