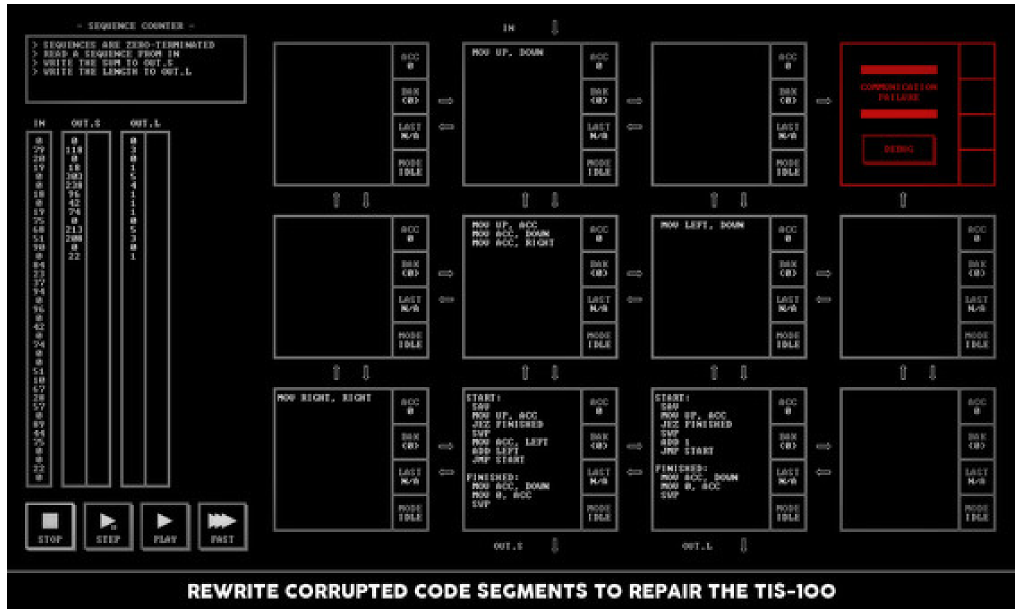
INF113: Linker and Syscalls

Kirill Simonov

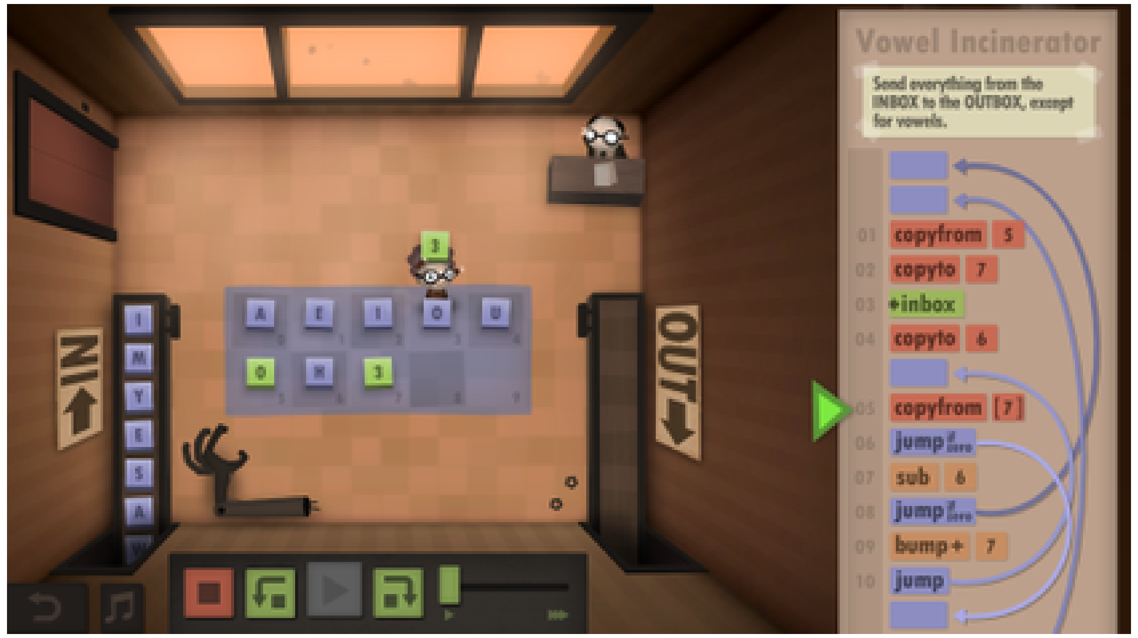
29.08.2025

Feel the assembly

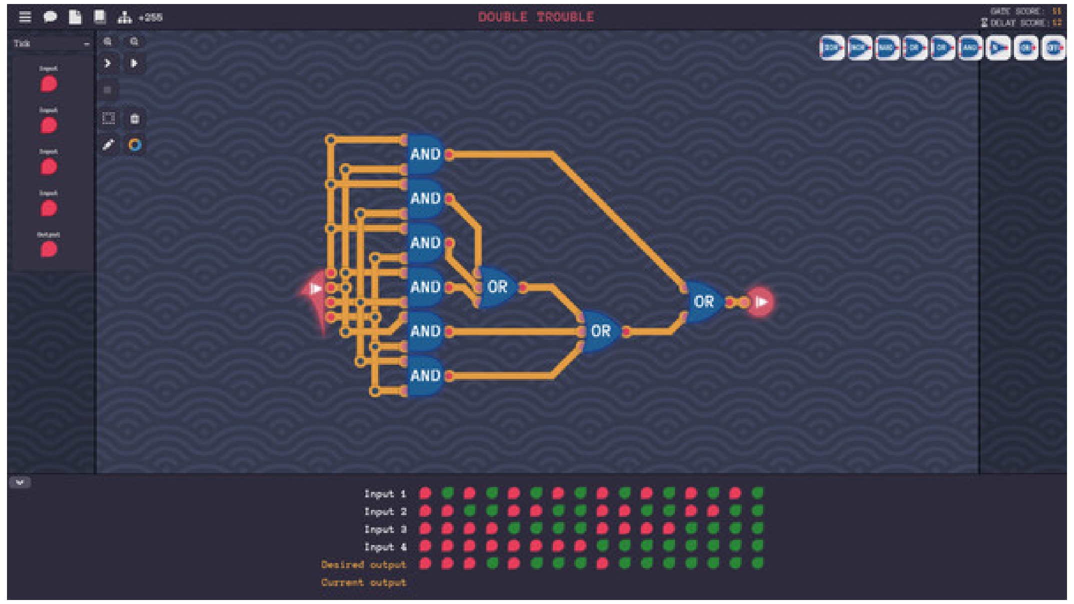
- Learn how old computers used to work
  - Example: PDP-11 from 1970s, watch **PDP11.mp4** in **Modules**
  - No OS, no processes
  - Setup the program directly in memory, and hit “run”
- Videogames*



TIS-100



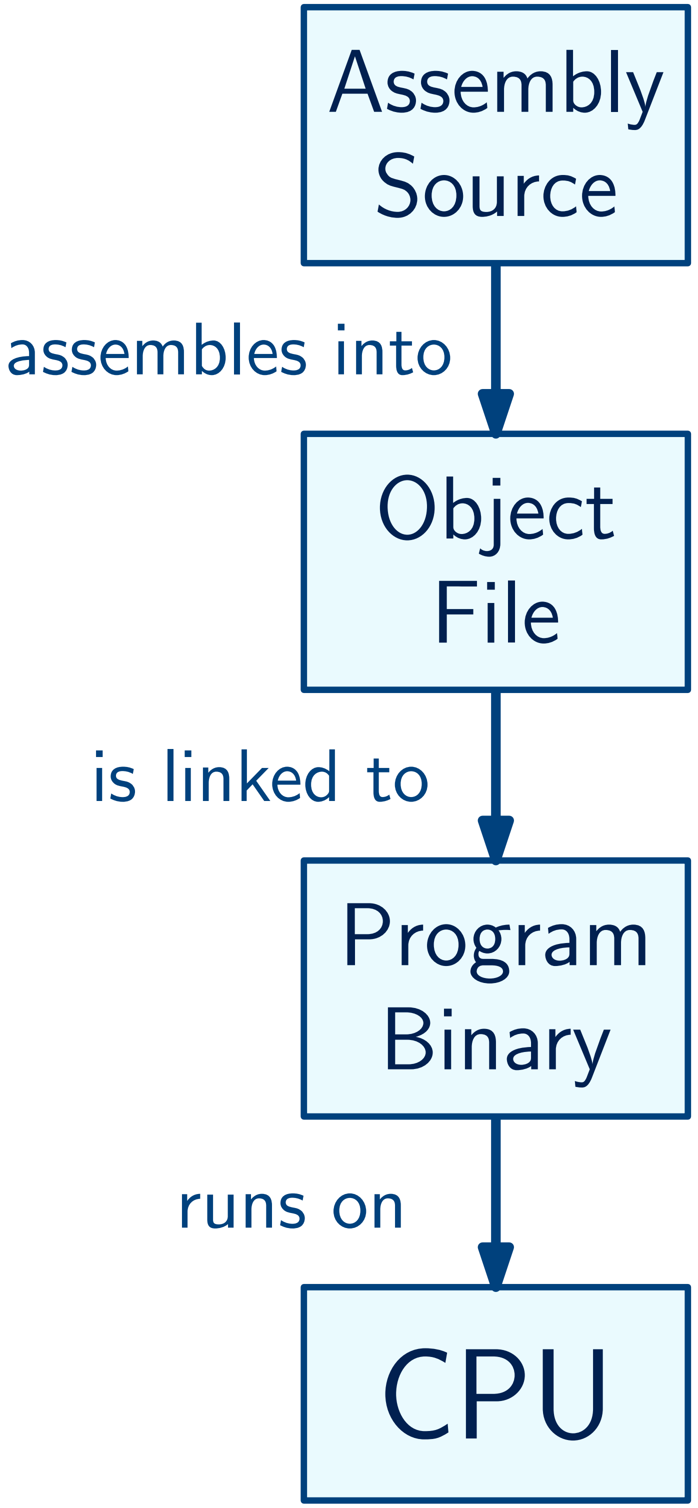
Human Resource Machine



Turing Complete

* not a paid advertisement

From assembly to binary




```
as -o program.o program.s
```

```
ld -o program program.o [...]
```

also external binaries

```
./program
```

Binaries

- Assembler converts mnemonic commands into machine codes

but there's more...

- Sets the file up in the ELF format, adding certain headers/metadata
  - Architecture/sizing information
  - Memory information
  - Sections and their offsets

Executable and **L**inkable **F**ormat (**ELF**)

a.k.a. the standard binary file format on Unix-like systems

`https://en.wikipedia.org/wiki/Executable_and_Linkable_Format`

- Linker then adjusts metadata to make a standalone executable
  - Also "connects" with other binaries—will see later today!

How to read ELFish

- hexdump outputs a binary file byte-by-byte in hex

executables are ≥ 4kB...

```
$ cat hello.txt
Hello, world!
$ hexdump -C hello.txt
00000000  48 65 6c 6c 6f 2c 20 77  6f 72 6c 64 21 0a
          |Hello, world!.|
0000000e
```

- objdump parses the ELF structure

actual code only!

```
$ objdump -d program
b:     file format elf64-x86-64
Disassembly of section .text:
0000000000401000 <_start>:
  401000: 48 c7 c7 23 00 00 00   mov    $0x23,%rdi
  401007: 48 c7 c0 3c 00 00 00   mov    $0x3c,%rax
  40100e: 0f 05                  syscall
```

Linker

- The object file has “unset” adresses

```
$ nm program.o
0000000000000000 T _start
$ objdump -s program.o
b.o:     file format elf64-x86-64
Contents of section .text:
 0000 48c7c723 ...
```

- The linker adjusts the address where instructions start

```
$ nm program
...
0000000000401000 T _start
$ objdump -s program
b:     file format elf64-x86-64
Contents of section .text:
 401000 48c7c724 ...
```

0x401000 is the default address for Instruction Counter to start

Linking multiple files

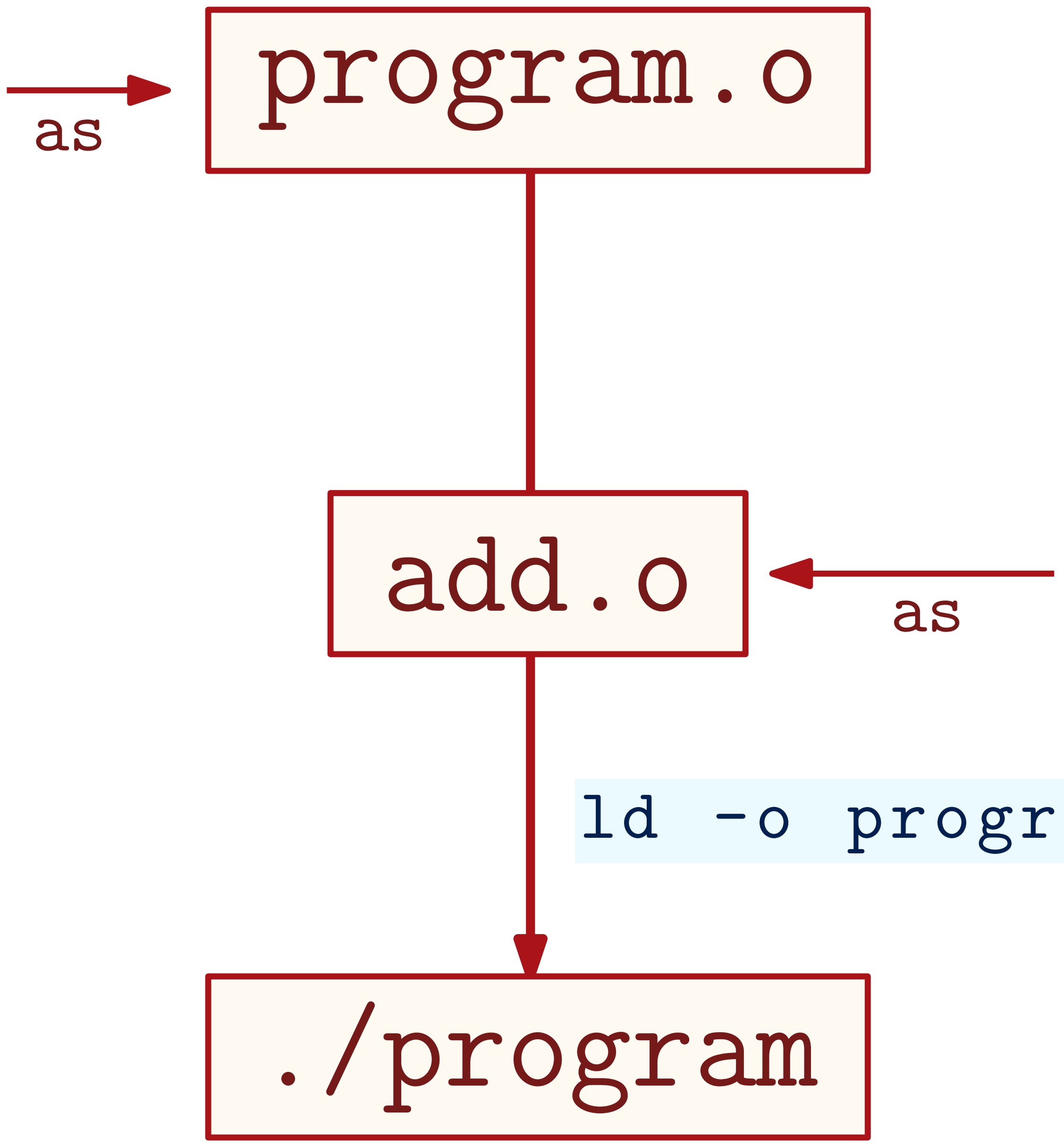
```
.code64

.section .text
.global _start

_start:
    mov $19, %rax
    mov $16, %rbx

    call _add

    mov %rbx, %rdi
    mov $60, %rax
    syscall
```




```
.code64

.section .text
.global _add

_add:
    add %rax, %rbx
    ret
```

```
ld -o program program.o add.o
```

Linking addresses

`ld`: `program.o` — `add.o` → `./program`

```
$ nm program.o
                 U _add
0000000000000000 T _start
```

```
$ nm add.o
0000000000000000 T _add
```

each object file has an entry point at zero

```
$ nm program
000000000040101f T _add
0000000000402000 T __bss_start
0000000000402000 T _edata
0000000000402000 T _end
0000000000401000 T _start
```

linker arranges all records

Linking calls

ld: program.o — add.o → ./program

```
$ objdump -d program.o

0000000000000000 <_start>:
   0: 48 c7 c0 13 00 00 00
   7: 48 c7 c3 10 00 00 00
   e: e8 00 00 00 00
  13: 48 89 df
  16: 48 c7 c0 3c 00 00 00
  1d: 0f 05
```

```
$ objdump -d program

0000000000401000 <_start>:
  401000: 48 c7 c0 13 00 00 00   mov    $0x13,%rax
  401007: 48 c7 c3 10 00 00 00   mov    $0x10,%rbx
  40100e: e8 0c 00 00 00         call   40101f <_add>
  401013: 48 89 df               mov    %rbx,%rdi
  401016: 48 c7 c0 3c 00 00 00   mov    $0x3c,%rax
  40101d: 0f 05                  syscall

000000000040101f <_add>:
  40101f: 48 01 c3               add    %rax,%rbx
  401022: c3                     ret
```

$\underbrace{\texttt{0x13}}_{\text{old IC}} + \texttt{0x0c} = \underbrace{\texttt{0x1f}}_{\text{new IC}}$

- Linker puts the correct address for each function call

System calls

- The programs are run in a "sandbox" environment
  - No I/O
  - No external memory
  - No setting exit code
  - ...
- To perform these actions, the program makes a `syscall`:
  1. Write what exactly needs to be done
  2. Pass the control back to the OS
  3. OS checks that instructions left by the program are ok
  4. OS completes the action and passes control back

```
.code64

.section .text
.global _start

_start:
    mov $19, %rax
    mov $16, %rbx
    add %rax, %rbx

    mov %rbx, %rdi
    mov $60, %rax
    syscall
```

Here: the value 60 in `%rax` is interpreted as the `exit` command
OS expects to find the `exit` code in `%rdi`

How to find syscalls

- Whenever `syscall` is invoked, the OS checks the operation code in `%rax`, e.g.:
  - 0: read
  - 1: write
  - 60: exit
- Look for codes
  - Locally in `/usr/include/x86_64-linux-gnu/asm/unistd_64.h`
  - Or in the source repo
    `https://github.com/torvalds/linux/blob/master/arch/x86/entry/syscalls/syscall_64.tbl`
  - Or in the guides, here also which argument goes to which register:
    `https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/`
- OS will expect arguments in certain order:

register mapping for system call invocation using `syscall`

system call number	1st parameter	2nd parameter	3rd parameter	4th parameter	5th parameter	6th parameter	result
`rax`	`rdi`	`rsi`	`rdx`	`r10`	`r8`	`r9`	`rax`

`https://en.wikibooks.org/wiki/X86_Assembly/Interfacing_with_Linux`

Hello, world!

- `write` has code 1, and expects arguments:
  1. stream in `%rdi`,
  2. address of the string in `%rsi`,
  3. length of the string in `%rdx`
- Pre-defined values have a separate section, here in read-only (`.rodata`)
- We can use assembly expressions to get the length

```
.code64
.section .rodata
msg:
    .ascii "Hello, world!\n"
    .set msglen, (. - msg)
.section .text
.global _start
_start:
    mov $1, %rax
    mov $1, %rdi
    lea msg, %rsi
    mov $msglen, %rdx
    syscall

    mov $60, %rax
    xor %rdi, %rdi
    syscall
```

System calls in C

- System calls have the same principle for C programs
- System calls themselves are implemented as C functions
  - Look up in the source repo

    https://github.com/torvalds/linux/blob/master/arch/x86/entry/syscalls/syscall_64.tbl
  - Also local manuals:

```
$ man 2 write
SYNOPSIS
       #include <unistd.h>

       ssize_t write(int fd, const void buf[.count], size_t count);
```

- Compiling with `gcc` both generates the object file, and calls the linker
  - The result is always linked with `libc`, hence calling library function works