Betriebssysteme

Tutorium 3

Peter Bohner

21. November 2025

0.1 ABI

ABI

What does that acronym even mean?
Application **B**inary **I**nterface

And what does that specify?

- Interface of *binary* programs (i.e. *after* compilation)
- Instruction Set (e.g. x86, ARM)
- Calling convention (e.g. `cdecl` or `System V AMD64 ABI`)
- Basic data types and their size / alignment (`int`, `sizeof(int)`)
- How to perform System Calls

ABI

You might also know the term „ABI of a library“?
This is used when talking about *binary compatibility* of different library versions. ⇒ Do you need to recompile your code against the new version?

ABI

What's the usual calling convention on modern Linux?
`System V AMD64 ABI` https://godbolt.org/z/68xexn

- Integer arguments in `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`, then stack
- FP arguments in `xmm0` to `xmm7`
- Integer return value in `rax`, `rdx`

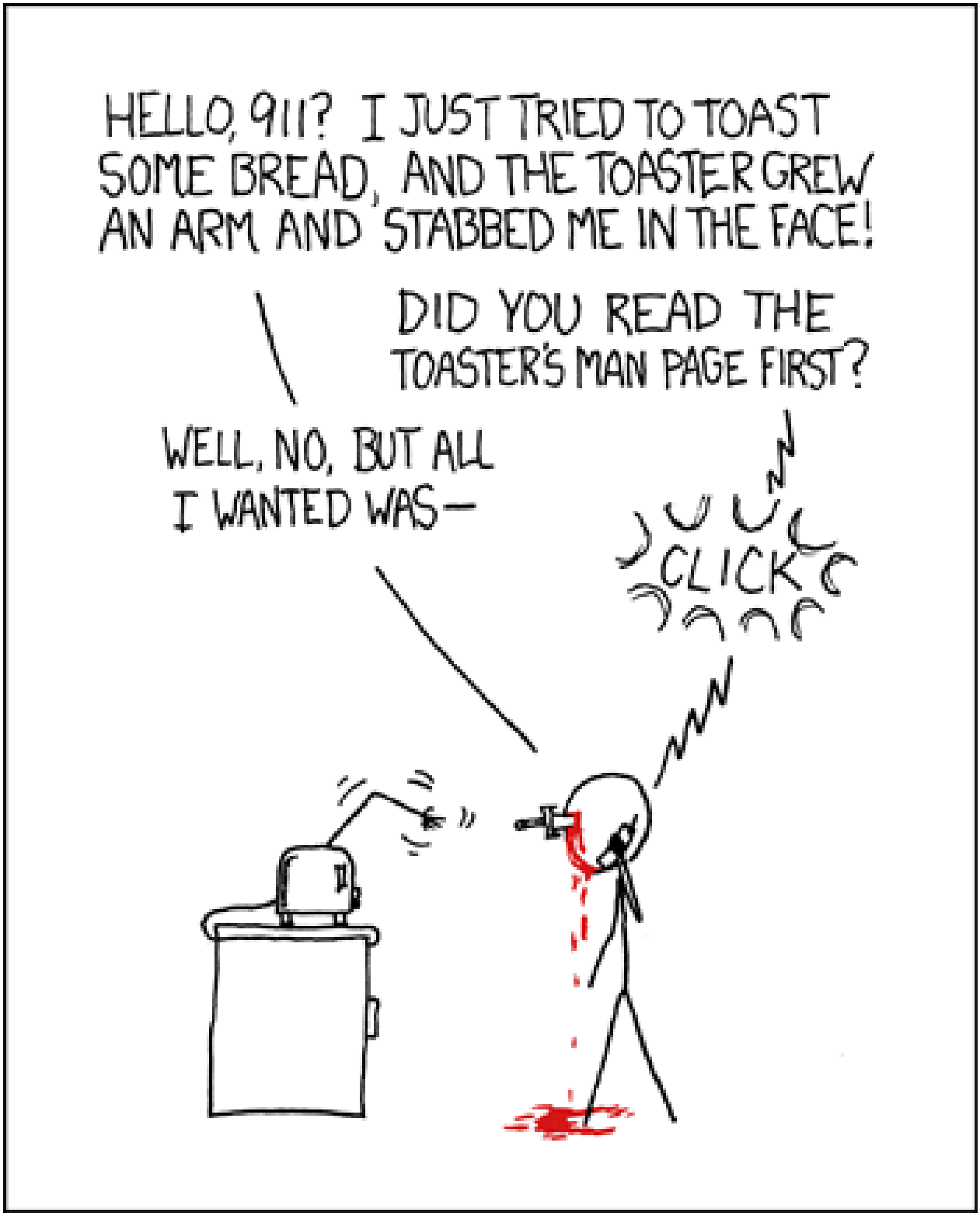
1 Fork

Fork

What does `fork` do?

- Creates a mostly identical child process
  - Same address space layout
  - Same data (as if it was all copied over)
  - Share open file descriptors
  - But the child has its own PID and address space
  - ... ⇒ `man fork`

~~Fork~~ *Knife*



Life is too short for man pages, and occasionally much too short without them. XKCD 293 - RTFM

Fork

Let's fork!

- Print `Fork failed` if the fork failed
- Print `I am the child` in the child
- Print `I am the parent of <child PID>` in the parent
- Wait for the child to die peacefully and print when it is dead in the parent

Fork

Let's fork! (More official solution)

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>

int main() {
    pid_t pid;
    switch( (pid = fork()) ) {
        case -1:
            printf( "Error. Fork failed\n" );
            break;
        case 0:
            printf( "I am the child!\n" );
            break;
        default: // pid > 0
            printf( "I am the parent!\nChild PID is %d\n", pid );
            wait( NULL ); // wait and ignore the result
            printf( "Child terminated\n" );
    }
    return 0;
}
```

Fork

Is `fork` sufficient to create a small shell?

- No, you also need to *load the other program* in the child

⇒ `exec(vp|v|...)`

2 Interrupts

Interrupts and friends

What three events lead to an invocation of the kernel?

- Interrupts
- Syscalls
- Exceptions

Interrupts and friends

What is an Interrupt?

- A signal generated *outside* the processor (typically from an I/O device)

What is an Exception?

- Caused by the CPU
- Inform the operating system of an error condition during execution
- E.g. `division by zero` or page faults

What is a System Call?

- Explicit call from user application into OS
- Used to interact with the OS (e.g. to read a file)

Interrupts and friends

A simple categorisation

	Voluntary	Involuntary
Sync	System call	Exception
Async	???	Interrupt

Gotta Catch em all

Can you think of an exception the OS can't repair?
i.e. it will terminate the process :(

- Divide by Zero
- The wrong kind of page fault (access forbidden or invalid memory)

Can you think of an exception the OS *can* repair?

- The right kind of page fault (e.g. access memory currently paged out to disk)
- When the OS is run in a virtual environment, it might try to access privileged instructions

⇒ CPU throws an exception

⇒ Virtual machine monitor handles that and executes the action on the real physical device (if allowed)

Looks like you fell into my little trap

How is the `trap` instruction related to system calls?

1. `trap` is executed and switches the CPU to kernel mode
2. The CPU continues execution at a hardware dependent location (where exactly depends on your architecture and a few other things, but the important part is that the operating system registered itself there)

⇒ `trap` can be seen as a low-level instruction making a higher-level mechanism (syscalls) possible

- You could also synchronously trap into the kernel using a division by zero or other fun stuff. That isn't normally done though...

Syscall numbers

What is a system call number? Why do you need them?

The user-level application can't just call a kernel function!

⇒ The kernel somehow needs to dispatch to the correct function

⇒ Array of function pointers with system call numbers as indices

Syscall - Parameter Passing

How can you pass parameters to the kernel when executing a syscall?

- Registers (used by Linux, see `man 2 syscall`)
- Stack (used by Windows 32bit)
- Main memory (and location in register)

Syscall - Turtles all the way down

How do syscalls relate to libraries?
Abstraction? How do you call them?

- Syscall mechanism is usually hidden behind library functions (`libc` on linux)
- The functions take care of preparing arguments, setting the syscall number and trapping in an efficient way
- Windows uses the `ntdll.dll` / `Win32 API`

Syscall - Security?

What do we need to keep an eye on when performing a system call in the kernel?

- Validate all parameters! *Twice*! *And a third time to be sure.*
- Otherwise you might have some arbitrary reads/writes into system memory

2.1 C - Insert

Write an insert function

Task

```
1 void insert(int *a, size_t *len, int z);
2
3 void main() {
4     int a[10] = {0, 1, 2, 3, 5, 6, 7, 8, 9};
5     size_t len = 9;
6     insert(a, &len, 4);
7     assert(a[4] == 4);
8     assert(len == 10);
9 }
```

What happens when we insert another?
We write over the array boundary!

What methods does C offer to copy memory?

`memcpy`

+ Potentially more efficient
- Can not handle overlaps

`memmove`

+ *Can* handle overlaps
- Potentially less efficient

Macros

Here will be demo!
Write `max(a, b)` as a function and as a macro.

- What differences in usage do you notice?
- What is more efficient?
- Any pitfalls you might see?

What happens with `max(a++, b)`?

2.2 Linking

Static and Dynamic Linking

What is the difference between static and dynamic linking?

- A static library is a collection of object files

⇒ The linked can treat it as normal code

- A dynamic library is loaded and linked *at runtime*

What are advantages of static / dynamic linking?

S+ Unused references can be elided

S+ Library calls just as fast as local ones

S+ No runtime overhead for loading and relocation

S- Library can not be shared ⇒ Memory overhead

D+ Library (Code segment) can be shared

2.2.1 PIC

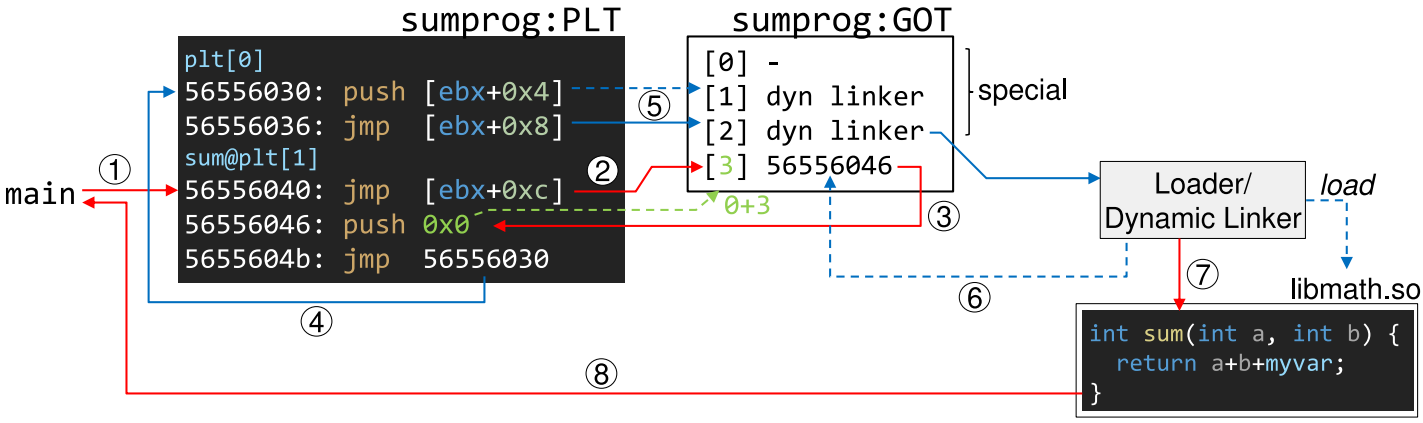
PIC

What does PIC stand for?
Position **I**ndependent **C**ode

Independent to *what*?
It's position in the process' address space.

- Shared libraries are loaded *somewhere* in the address space of a process
- PIC uses *relative* addresses (to what?)(to the instruction pointer)only – and can therefore be loaded anywhere
- How do you find *global* symbols then? ⇒ GOT, the `Global Offset Table`

PIC - Methodenaufrufe



Quelle: Vorlesungsfolien