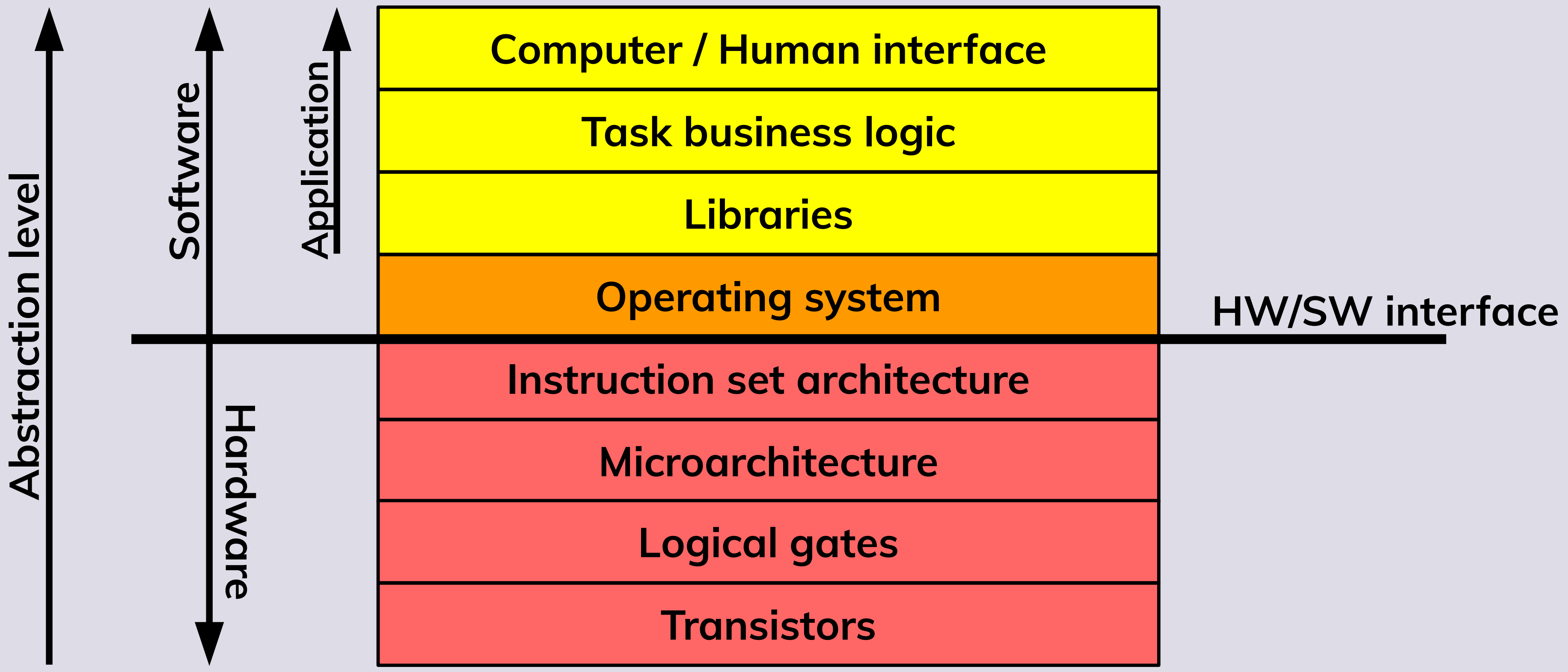
Advanced Operating Systems
Summer Semester 2023/2024

Martin Děcký

2

Programming Languages and Techniques

Levels of Abstraction



Operating Systems Specifics

- **Operating system as a whole**
  - In principle, a piece of software like any other
    - Potentially complex software architecture
      - Monolithic (unikernel), layered/stacked (monolithic kernel), component-based (microkernel), etc.
  - Properties that are less usual
    - Almost always an open-ended platform
      - Application Programming Interface (API)
      - Component life cycle management at run time
      - Potentially recursive (virtual machines)
    - Different criticality and/or privilege levels
      - Multiple address spaces
    - Application Binary Interface (ABI)

Operating Systems Specifics

- **Operating system kernel**
  - In principle, a program like any other
    - In the “steady state”, mostly following data-driven and event-driven model
      - Inputs, outputs, events, etc.
  - Properties that are less usual
    - Self-supporting its own run time environment
      - Chicken-and-egg problem especially during bootstrap
      - Peculiar shutdown
    - Direct interaction with hardware
    - Privileged mode (little to no “safety net” for errors)
      - Access into multiple address spaces

Requirements on the Programming Language

- **Versatility as a "platform builder"**
  - Interfacing with hardware and firmware
    - No limitations regarding the means of the communication (memory access patterns, special instructions, etc.)
  - Code self-modification
  - ABI malleability
  - Modularity
- **No excessive baggage, not standing in the way**
  - No complex external run time that would require its own major support
- **Safety**
  - Ideally fundamental or at least reasonably achievable
- **Predictability**
  - Straightforward mapping between language constructs and machine code
  - Also in the time domain (especially for real time)

Logo by DALL·E 3 via ChatGPT 4

Assembly Language

- **Language of symbolic machine code instructions**

```
swap:
      movslq %esi, %rsi                   0100100001100111111110110
      leaq (%rdi, %rsi, 4), %rdx          01001000100011010001010010110111
      leaq 4(%rdi, %rsi, 4), %rax    →    0100100010001101010001001011011100000100
      movl (%rdx), %ecx                   1000101100001010
      movl (%rax), %esi                   1000101101110000
      movl %esi, (%rdx)                   1000100101110010
      movl %ecx, (%rax)                   1000100100001000
      retq                                11000111
```

```
swap:
      sll $a1, $a1, 2                     00000000000001010010100010000000
      addu $a1, $a1, $a0                  00000000101001000010100000100001
      lw $v0, 0($a1)                      10001100101000100000000000000000
      lw $v1, 4($a1)                 →    10001100101000110000000000000100
      sw $v1, 0($a1)                      10101100101000100000000000000100
      sw $v0, 4($a1)                      10101100101000110000000000000000
      jr $ra                              00000011111000000000000000001000
```

Assembly Language

- **Language of symbolic machine code instructions**

instruction mnemonics

opcode

```
swap:
     movslq %esi, %rsi                 010010000110001111110110
     leaq (%rdi, %rsi, 4), %rdx        01001000100011010001010010110111
     leaq 4(%rdi, %rsi, 4), %rax       0100100010001101010001001011011100000100
     movl (%rdx), %ecx           →     1000101100001010
     movl (%rax), %esi                 1000101100110000
     movl %esi, (%rdx)                 1000100100110010
     movl %ecx, (%rax)                 1000100100001000
     retq                              11000111
```

dereference

register name

constant

```
swap:
     sll $a1, $a1, 2                   00000000000001010010100010000000
     addu $a1, $a1, $a0                00000000101001000010100000100001
     lw $v0, 0($a1)                    10001100101000100000000000000000
     lw $v1, 4($a1)              →     10001100101000110000000000000100
     sw $v1, 0($a1)                    10101100101000100000000000000100
     sw $v0, 4($a1)                    10101100101000110000000000000000
     jr $ra                            00000011111000000000000000001000
```

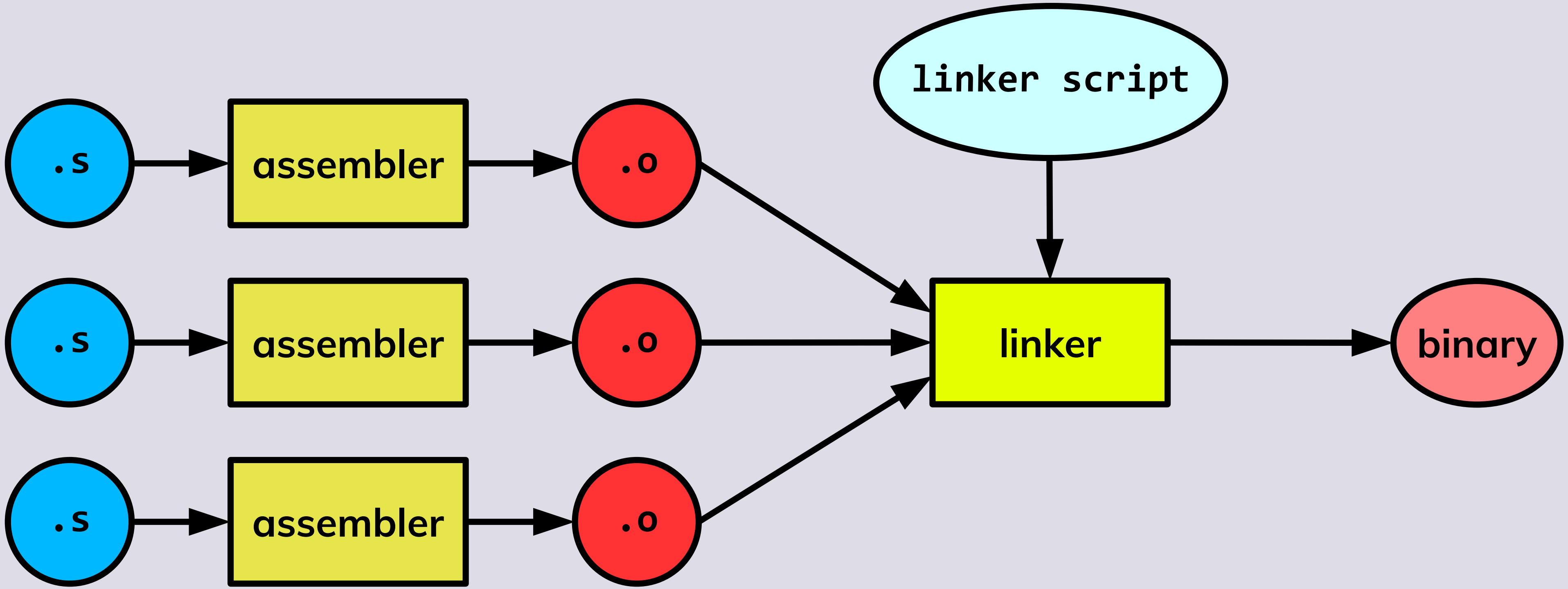
displacement

Assembly Language

- **Maximal versatility and almost no baggage**
  - Everything that a machine code can express can assembler express
    - Unknown instructions can be “typed in” as arbitrary bytes
- **Specific assembler implementations provide a relatively rich programming features**
  - Symbolic labels for memory locations
    - Usable as branch targets, variables, values in expressions, etc.
  - Synthetic instructions
  - Directives
    - Compiler configuration
    - Instruction and data modifiers
    - Modular compilation (sections, external labels, etc.)
  - Constants and (compile-time) expressions
  - Subroutines, macros (with compile-time control flow)
  - Comments

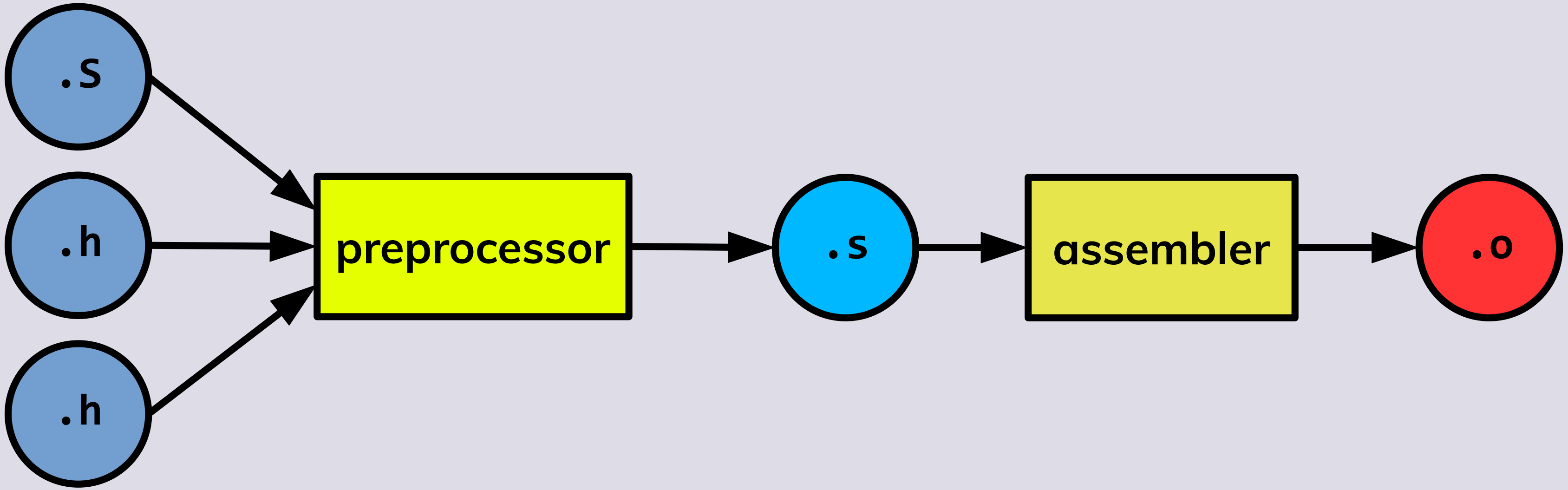
Assembly Language

- **Modularity**



Assembly Language

- **Meta-assembler**



Assembly Language

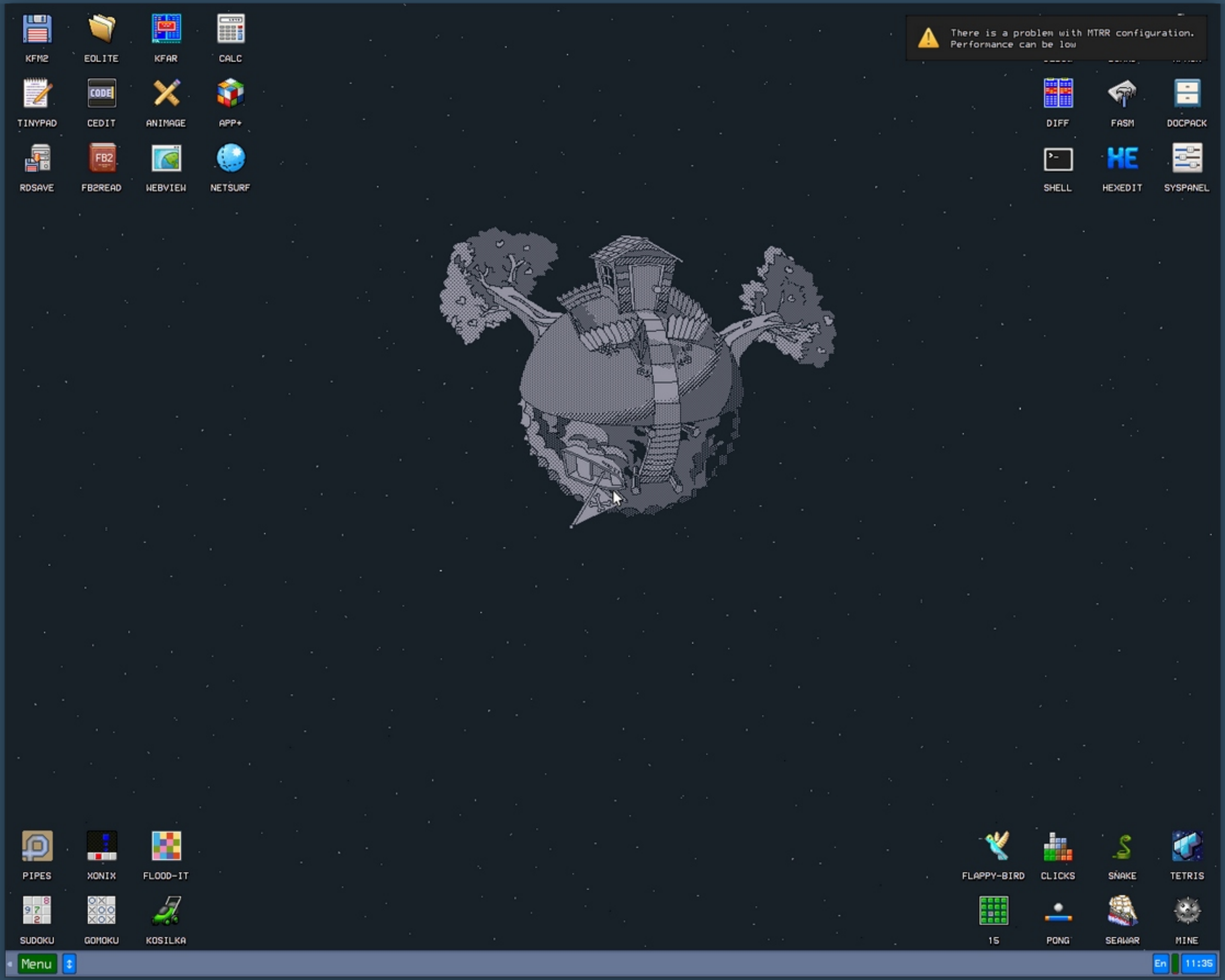
- **Limitations**
  - Typically single-pass compilation
    - Inability to modify already generated output
      - Output addresses within a module can only increment
      - Worked around by outputting into different sections
    - Undefined symbolic addresses are considered external (to be filled in by the linker)
      - Potentially pessimistic code due to unknown address sizes

Assembly Language

- **Drawbacks**
  - Very narrow portability
    - Assembly language is not just ISA-specific
      - There is little portability support even for ISA variants, CPU models, etc.
  - Verbosity
    - Especially on RISC architectures
  - Extremely poor maintainability
    - In principle, there could be code inspection, refactoring, completion, etc.
      - Writing code in assembly is a niche nowadays
      - Some advanced features are integrated in reverse engineering tools, but hardly in modern IDEs
  - Poor performance of larger pieces of code
    - On modern superscalar CPUs, humans outperform optimizing compilers only on specific small and tight routines (e.g. direct hardware manipulation, memory copying, etc.)

Assembly Nowadays

- **Demoscene**
  - 256 B, 4 KiB demos
- **Hobbyists**
  - MenuetOS, KolibriOS
- **Routines requiring tight hardware control**
  - Firmware DRAM initialization (code running in CPU cache only)
  - Bootstrap code with no usable stack
  - Kernel memory copying between address spaces
    - Fixups in case of a page fault require stack usage discipline
  - Code resilient to timing side channels
  - Context & mode switching routines
  - Low-level interrupt handling, virtualization, etc.
- **Substitution for missing compiler intrinsics (inline assembly)**
  - Atomics and synchronization
  - Tight inner loops, SIMD routines


KFM2
EOLITE
KFAR
CALC
TINYPAD
CEDIT
ANIMAGE
APP+
RDSAVE
FB2READ
WEBVIEW
NETSURF
There is a problem with MTRR configuration.
Performance can be low
DIFF
FASM
DOCPACK
SHELL
HEXEDIT
SYSPANEL
PIPES
XONIX
FLOOD-IT
SUDOKU
GOMOKU
KOSILKA
FLAPPY-BIRD
CLICKS
SNAKE
TETRIS
15
PONG
SEAWAR
MINE
Menu
En
11:35

Logo by DALL·E 3 via ChatGPT 4

D3S

C Language

- **Originally designed for implementing Unix utilities**
  - Later used to reimplement the Unix kernel
- **Key features**
  - A standalone C program requires very little run time support
    - Memory for the code
    - Memory for the static data (global variables)
    - Memory for the stack
    - Well-defined entry context
      - Instruction pointer, stack pointer and a few other platform-specific registers
  - In the `freestanding` environment, the existence of the standard C library in not assumed

C Language

- **Other properties**
  - Function arguments passed as values (generally on the stack or in registers)
  - Single lexical scope of functions
  - Pointer arithmetic, memory model (originally quite rudimentary)
  - Ad hoc run-time polymorphism
  - Basic modularity, conditional compilation and meta-programming
  - Abstract machine
    - Language constructs and operations
    - Static (but weakly enforced) type system
    - Maps in a straightforward way to most ISAs while providing solid portability
      - **Caution:** Definitively not a 1:1 mapping

C Language

- **Used to be synonymous for “the system programming language”**
  - Almost universally adopted in 1980s and early 1990s for system-level software (firmware, kernels, core OS components and libraries)
    - Gradually replaced assembler
  - One of the most popular programming languages in general
    - Actually used for the majority of non-system applications
      - Despite generally poor support for strings, generic data types, etc.
      - Almost universal (theoretical) portability
  - Not without adverse effects
    - Arguably a major cause of the dire state of safety and security of many software stacks

C Language

- **Problematic aspects**
  - C preprocessor
    - Header inclusion is a poor replacement for proper module support
      - Boilerplate include guards
    - Conditional compilation and macro expansion does not understand or respect the language syntax
      - Overuse of macros often leads to a “DSL from hell”
  - Obsoleted features / Should be obsoleted features
    - Functions without a declaration assume to have a variadic argument list and the `int` return type
    - Strange operator precedence (e.g. bitwise operators vs. comparison)
    - Bitfields with implementation-specific memory layout
  - Type safety of variadic functions
  - Misunderstanding of the `volatile` modifier (not usable as universal atomic)


And then they forgot to allocate the memory for the null terminator
Segfaaaaaault!
THE
C
PROGRAMMING
Dennis M. Ritchie

C Language

- **Undefined behavior**
  - **Caution:** Not “unspecified” or “implementation defined” behavior
    - *Abstract machine in an unknown state → Entire program behavior undefined*
    - Compiler is allowed to assume that well-formed code does not contain undefined behavior
      - Could be used to drive the compiler optimization (`assume` macro)
  - Examples
    - Accessing an uninitialized variable
    - Division by zero (or other mathematically undefined operation)
    - Signed integer overflow
    - Bitwise shifts larger than the type bit width (or negative)
    - Modifying an object between two sequence points more than once
    - Data race
    - Not returning a value from a non-void function

C Language

- **Undefined behavior**
  - Examples
    - Spatial memory safety violation
      - Out of bounds memory accesses
      - Dereferencing a NULL pointer
      - Modifying a string literal or constant object
    - Temporal memory safety violation
      - Accessing local variables outside their scope
      - Use-after-free, double free
    - Strict aliasing violation
    - Alignment violation
    - Infinite loop without a side-effect
      - Not just empty loop

C Language

- **Undefined behavior**

```
unsigned int add_inc(unsigned int *a, unsigned int *b)
{
    return (*a)++ + (*b)++;
}
```

C Language

- **Undefined behavior**

```
typedef struct {
    unsigned int uid;
} user_t;

int elevate(void)
{
    user_t *user = get_privileged_used();
    unsigned int uid = user->uid;

    if (user == NULL)
        return -EINVAL;

    grant_access(uid);
    return 0;
}
```

C Language

- **Undefined behavior**

```
#define SIZE 42

unsigned int data[SIZE];

bool present(unsigned int value)
{
    for (unsigned int i = 0; i <= SIZE; i++) {
        if (data[i] == value)
            return true;
    }

    return false;
}
```

Response to C Shortcomings

- **Coding guidelines & standards**
  - MISRA C
    - Motor Industry Software Reliability Association
    - De facto requirement for safety certifications
    - Set of mandatory, required and advisory guidelines
      - Each deviation from a required guideline must be documented with a rationale
    - Mixes genuinely useful rules with some rather questionable
      - Rule 15.5: *A function should have a single point of exit at the end*
    - Very hard to be applied to a dynamic operating system
      - Rule 17.2: *Functions shall not call themselves, either directly or indirectly*
      - Rule 21.3: *The memory allocation and deallocation functions shall not be used*

Response to C Shortcomings

- **Coding guidelines & standards**
  - CERT C
    - Computer Emergency Response Team Coordination Center (CERT/CC) at Software Engineering Institute (SEI)
    - https://wiki.sei.cmu.edu/confluence/display/c
    - Broader target than MISRA C
      - Some focus on security
    - Classification of rules
      - Severity, likelihood, remediation cost, priority, etc.
    - Assessment of detection tools

C++

D3S

C++ Language

- **Originally an object-oriented extension of C ("C with Classes")**
  - Easy interoparability with C (although not a strict superset)
  - Higher-level abstractions for existing C constructs
    - Pointers → References
    - Macros → Templates, constant expressions
    - Boolean as integer → Dedicated boolean type
    - Error return values → Exceptions
    - Manual encapsulation & polymorphism → Classes, overloading, default arguments
    - Function pointers → Lambda expressions
    - Dynamic memory management integrated into the language
    - Stricter type system, concepts
  - Goal of providing abstractions at reasonable run-time cost
    - Zero cost if the abstraction is not actually used

C++ Language

- **Some language aspects considered problematic for system-level use**
  - Unpredictable cost of abstractions (memory, time)
    - The `freestanding` mode still assumes the presence of the run-time library and a minimal standard library
      - Required for static constructors and destructors
    - Run-time type identification has memory overhead even if not used
      - Required for exceptions, `typeid`, `dynamic_cast`
    - Stack unwinding and dynamic memory allocation on throwing exceptions
    - STL considered bloated
  - Disabling or avoiding those aspects is possible by many compilers, but the resulting language might be non-conforming

C++ Language

- **Custom implementation of standard features**
  - Static constructors and destructors, deferred constructors
  - Smart pointers (`unique_ptr`)
  - Tagged dynamic casting (limited to upcasting)
  - Type traits
  - Containers
  - Replacement of virtual methods (inheritance) by compile-time composition of alternatives
- **Useful system-level constructs**
  - Guarded objects
  - Better type safety (e.g. type-safe integers)

Source: Matfyz memes that will make you ČVUT

C++ Language

- **Problematic aspects**
  - Templates are the new macros
    - Templates from hell
  - Operator overloading as an elegant obfuscation
  - Almost all C undefined behavior is still with us
    - Plus some more
      - `delete[]` on a single object, `delete` on an array
      - All sorts of class shenanigans (incorrect casting, calling methods before all base constructors, calling virtual methods from constructor)
      - Extending the `std` namespace
      - Infinite template recursion

Rust Language

- **Goal: System-level language akin to C, but designed in 2010s**
  - Key requirements
    - Relative simplicity (no inheritance, no template meta-programming)
    - Straightforward mapping to current hardware (two's complement, fixed byte size, etc.)
    - Lean run time (but good support for Unicode strings)
    - Explicit resource and memory management
  - Avoiding the major shortcomings of C / C++
    - No surprising undefined behavior
    - Compile-time memory, type and concurrency safety
  - Certainly not the first attempt on "modern C"
    - D, Nim, Go, V, etc.
  - **Novel approach:** Two languages in one (safe + unsafe)

Rust Language

- **Features**
  - Curly-bracket syntax with familiar control flow keywords and operators
  - Fixed-sized integer and float types
  - Unicode character and static strings built-in types
  - Tuple built-in type, bottom/never type (no-return functions)
  - Non-null references and raw (unsafe) pointers
  - Structures and tagged/disjoint unions with methods (memory layout is not predefined)
  - Pattern matching
  - Ranges
  - Statements as expressions (implicit function return)

Rust Language

- **Features**
  - Function argument type polymorphism
  - Ad hoc type polymorphism using traits
  - Immutable variables by default, type inference
  - Mandatory initialization
  - `Option` type (nullable) and `Return` type (error handling) as library constructs
  - Memory and data race safety via compile-type lifetime tracking
    - Every valid object has exactly one owner
    - References exist only for valid objects
    - A single mutable reference exists only if no immutable references exist
    - Destructors for resource management

Rust Language

- **Unsafe mode**
  - For implementing system-level features
    - Violating ownership rules
    - Dereferencing raw pointers
    - Type casting (punning)
    - Volatile memory access
    - Intrinsics, inline assembly
  - Assumptions of the safe mode hold after the unsafe block ends
    - Otherwise it is undefined behavior
- **Other cases of undefined behavior**
  - Typically diagnosed with a run-time panic

Rust Language

- **Macros**
  - Declarative macros
    - Expansion using pattern matching
    - Similar to other macro languages, but core language concept
  - Procedural macros
    - Compile-time modification of the input tokens
    - Code generation
- **Modularity and package management**
- **Language features versioning**
  - Still, ongoing language development and the approach to the supply chain can be problematic
- **C interoperability using `bindgen`**
- `no-std` **environment**
  - Still needs some unstable/custom run time parts (e.g. **`alloc`**)
    - Practically on a similar level as C++

Other System Languages

- **Forth**
  - OpenBoot, Open Firmware
- **C#, Spec#, Sing#, M#**
  - Singularity, Midori
- **Pascal, Modula(-2), Oberon**
  - Legacy Apple OSes, Oberon
- **Ada, SPARK**
  - Muen
- **(BBC) Basic**
  - Legacy RISC OS
- **Smalltalk, Objective-C**
- **Zig, Jakt, Hare, V**

Thank you!

Questions?