Advanced Operating Systems
Summer Semester 2022/2023

Martin Děcký
Languages and Run Times
Levels of Abstraction

- User interface
- Application core
- Libraries
- Operating system
- Instruction set architecture
- Microarchitecture
- Logical gates
- Transistors

Abstraction level

Software

Hardware

HW/SW interface
Operating Systems Specifics

- **Operating system**
  - A (potentially complex) piece of software like any other (in principle)
    - Various possible software architectures
      - Monolithic, layered, componentized, etc.
    - Usually several internal layers of abstraction
  - **Specifics**
    - Almost always an open-ended platform
      - Frequently component life cycle management at run time
    - Different criticality and privilege levels
    - Application Binary Interface (ABI) besides just Application Programming Interface (API)
Operating Systems Specifics

• Operating system kernel
  - A (potentially complex) program like any other (in principle)
    • At least in the “steady state” when already running
      - Inputs, outputs, events, etc.
  - Specifics
    • Self-supporting its own run time environment
      - Peculiar especially during bootstrap and shutdown
    • Limited protection from its own bugs
Requirements on the Programming Language

• **Sufficiently versatile as a “platform builder”**
  - Enable the interfacing with hardware / firmware
    • Especially no limitations regarding the means of the communication
  - Not in conflict with self-modifications
    • Supporting the open-endedness
    • Supporting the malleability of the ABI
  - Reasonably modular

• **Not carrying excessive baggage, not standing in the way**
  - No aspects that would require their own major support
    • Avoiding the chicken-and-egg problem

• **Safe?**
Assembly Language

- Language for symbolic machine code instructions

```
swap:
movslq %esi, %rsi
leaq (%rdi, %rsi, 4), %rdx
lea 4(%rdi, %rsi, 4), %rax
movl (%rdx), %ecx
movl (%rax), %esi
movl %esi, (%rdx)
movl %ecx, (%rax)
retq
```

```
swap:
sll $a1, $a1, 2
addu $a1, $a1, $a0
lw $v0, 0($a1)
lw $v1, 4($a1)
sw $v1, 0($a1)
sw $v0, 4($a1)
jr $ra
```
Assembly Language

- Language for symbolic machine code instructions

```assembly
swap:
movslq %esi, %rsi
leaq (%rdi, %rsi, 4), %rdx
leaq 4(%rdi, %rsi, 4), %rax
movl (%rdx), %ecx
movl (%rax), %esi
movl %esi, (%rdx)
movl %ecx, (%rax)
retq
```

```assembly
swap:
sll $a1, $a1, 2
addu $a1, $a1, $a0
lw $v0, 0($a1)
lw $v1, 4($a1)
sw $v1, 0($a1)
sw $v0, 4($a1)
jr $ra
```
Assembly Language

- **Maximal versatility and almost no baggage**
  - Everything that can be expressed in the machine code can be expressed in assembly
    - Even unknown instruction mnemonics can be just “typed in” as arbitrary bits/bytes

- **Specific assemblers 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

Diagram:
- Source files (.s) are assembled by an assembler to produce object files (.o).
- Multiple object files are linked together by a linker.
- The linker uses a linker script.
- The final output is a binary file.

linker script

binary
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
  - Not just among ISAs, but also among ISA variants, CPU modes, 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, thus such features are mostly integrated in reverse engineering tools rather than in IDEs
- Poor performance of larger pieces of code
  - On modern superscalar CPUs, humans outperform optimizing high-level compilers only on specific tight routines (e.g. direct hardware manipulation, memory copying, etc.)
Assembly in Today’s World

- **Hobbyists, demoscene**
  - 256 B, 4 KiB demos
  - MenuetOS, KolibriOS

- **Routines requiring tight hardware control**
  - Firmware DRAM initialization (only CPU cache usable)
  - Bootstrap code with no usable stack
  - Kernel memory copying between address spaces (with fixups in case of a page fault)
  - Code resilient to timing side channels
  - Context & mode switch routines

- **Substitution for missing compiler intrinsics (inline assembly)**
  - Atomics and other synchronization
  - Tight inner loops
  - SIMD, interrupts, virtualization, etc.
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 with the code
    - Memory with 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 relevant 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:** Defintively not a 1:1 mapping
C Language

- **Synonymous for “system programming language”**
  - Almost universally adopted in 1980s and early 1990s for system-level software (firmware, kernels, core OS components and libraries)
  - One of the most popular programming languages in general
  - Not without adverse effects
    - Arguably a major cause of the dire state of safety and security of many software stacks
C Language

- **Some problematic aspects**
  - C preprocessor
    - Header inclusion is a poor replacement for proper module support
      - Needs to be augmented by 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)
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 undefined behavior never happens
  - 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**
  - 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
C Language

- **Undefined behavior**

```c
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**

```c
#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 many 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](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++ Language

- **Originally an OOP 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
    - Booleans 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
  - Goal of providing abstractions at reasonable (preferably zero) run-time cost
C++ Language

- Many system-level use cases disable/avoid entire language aspects
  - There is the freestanding mode, but it assumes the existence of the run-time library and a minimal standard library
    - Run-time type identification (exceptions, typeid, dynamic_cast)
    - Static constructors and destructors
    - Stack unwinding (exceptions)
  - STL is mostly considered too bloated
C++ Language

- **Custom implementation of standard features**
  - Static constructors and destructors, deferred constructors
  - Smart pointers (unique_ptr)
  - Limited dynamic casting
  - Type traits
  - Containers
  - Replacement of virtual methods by compile-time composition of alternatives

- **Other useful features**
  - Guarded objects
  - Better type safety (e.g. type-safe integers)
C++ Language

• **Some problematic aspects**
  - Templates are the new macros
  - 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

- **What if C was designed in 2010s?**
  - Actual benefits ...
    - Relative simplicity
    - Straightforward mapping to hardware
    - Lean run time
    - Explicit resource and memory management
  - ... without the shortcomings
    - Undefined behavior
    - No guarantees for memory, type and concurrency safety
  - Certainly not the first attempt on “modern C”
    - D, Nim, Go, etc.
  - **Novel approach:** Two languages in one (safe & unsafe)
Rust Language

**Feature overview**
- 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

- **Feature overview**
  - 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**
  - Low level code
    - 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

- **bindgen** for C interoperability
- **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**
Thank you!

Questions?