**System Software Crash Course***Draft  
Version of Sept 5, 2018*

**Overall amount of lectures/labs: 19.  
Starting date: February 2019**

**Block A: CPU Architecture and Low-level Programming**

Lectures: E. Zouev, D. Botcharnikov  
Labs: E. Zouev, D. Botcharnikov  
Approx. number of lectures: 3

* Common hardware organization: von Neumann vs Harvard. RISC architecture. Pipeline: benefits and hazards. Load/store organization and operand addressing modes. Branch prediction.
* Superscalar architecture & VLIW. Out-of-order execution. Vector units, GPU, multicore and manycore processors. Layered memory organization. Memory cache architecture and policies. Peripherals access: mapped or separated.
* ARM processors: common architecture. Memory, registers, instruction set overview. ARM assembly language.

Assemblers. Elements of assembler language: directives, instructions, literals. User-defined labels and types. Operand addressing modes. Assembler design: one-pass and multi-pass assemblers. Sections and memory control. Assembler macro facilities. Interfaces to OS and high level languages: platform ABI and calling convention.

**Block B: Basics of C**

Lectures: E. Zouev  
Labs: E. Zouev  
Approx. number of lectures: 4

* Review of the language; why you should know C basics. Common program structure: translation units and interface model (#include directive).  
  C memory model: code, heap and stack. Program execution: low-level model. Function calls, the notion of stack and stackframe, static & dynamic chains.
* Fundamental language notions: type, variable, function, expression, statement. Basic C syntax: declarations & definitions, the notion of scope. Fundamental types; type conversions. Basic compound types: Arrays & structures.
* Pointers; pointer arithmetics. Typical C problems: null & dangling pointers, unreachable objects, dangerous type conversions. Storage classes: static, extern, auto.
* Expressions: common syntax, operators, operator precedence. Statements. Preprocessing: macros & macro expansion; conditional compilation; header files.

**Block C: Compiler Construction**

Lectures: E. Zouev, S. Ignatov  
Labs: E. Zouev  
Approx. number of lectures: 6

* Introduction to compilers & compiler construction. Programming languages & compilers. Compilation tasks & aims. Literature on compiler construction. Phases of compilation. Canonical compilation scheme and its improvement. Compiler development technologies.

Typical compiler architecture. Examples of real compiler architectures. The evolution of compiler architecture: from the black box approach to the open collection of communicating components and resources. Compilation in narrow sense versus compilation in wide sense.

* Lexical analysis. Tokens & lexemes. Examples of lexical structure of programming languages. Regular grammars, finite state machines & regular expressions. Theory & implementation.
* Syntax analysis and program intermediate representation. Internal program representation. Symbol tables. The notion of Abstract Syntax Tree (AST). Hash tables and hash functions. Annotated AST (AAST). Recursive descent parsing. Internal representation of a language type system.
* Compiler construction technologies & tools. T-notation and bootstrapping technology. Bottom-up parsing: the basics. Automatic parser generation. Yacc/Bison parser generators and their clones. An extended example: creating Yacc grammar for the real language.
* Semantic analysis. Syntactic and semantic program correctness. Static and dynamic semantics. Semantics of declarations, statements and expressions. Type analysis. Standard and user-defined type conversions. Hidden semantics. A quick introduction to program static analysis. Semantic analysis for classes, abstract methods and polymorphism. Type inference for dynamic languages. Abstract interpretation.
* Program optimizations. Resources/aspects to be optimized: efficiency (speeding up) and/or memory consumption. Optimization categories & kinds. Source code and low-level optimizations. Examples of typical optimizing conversions.

Code generation: aims & techniques. Control flow graph (CFG) and single static assignment (SSA) as basic notions in code generation approaches. From AST to CFG. The SSA implementation in the LLVM infrastructure.

**Block D: Programming Languages and Execution Environments**

Lectures: E. Zouev, S. Ignatov, A. Soldatov  
Labs: S. Ignatov, A. Soldatov  
Approx. number of lectures: 3

* Conventional approaches for building programs: separate & independent compilation, static & dynamic linking.
* Virtual memory model. Process memory map. Memory types: data, bss, stack, heap. Approaches to OOP implementation: VTBL.
* Current trends and latest news in programming languages’ design, compiler construction and related areas. Quick review on modern programming languages: Go, Rust, Swift.
* Advanced C++: multiple & virtual inheritance, templates. The latest additions to the language: lambda expressions, advanced template mechanisms.
* Abstract machines: the notion and the implementation. From P-code to JVM/.NET and then to LLVM. Interpretation, compilation & JIT-compilation.
* Java Virtual Machine, Java class file and bytecode. Principles of managed execution. .NET MSIL code and the notion of assembly. The notion of metadata. LLVM framework and LLVM intermediate language (bit-code). Garbage collection: strategies and tools.
* Core development utilities. Toolchain essentials. Methods, techniques and tools for static and dynamic program analysis.

**Block E: Software development**

Lectures: D. Botcharnikov  
Labs: D. Botcharnikov  
Approx. number of lectures: 3

* Linkers. Memory map of process in modern OS. Separate compilation: benefits and drawbacks. Linker responsibilities and architecture. Static and dynamic linking: benefits and drawbacks, mechanisms of dynamic linking. Linking and loading on bare-bone systems. Object code file formats.
* Profilers & debuggers. Build tools (make/cmake, MS Build). Version control systems. Git and GitHub repositories. Testing strategies and approaches (regressions tests, unit tests, text coverage).
* Processing experimental data. Software quality & assurance.