Czech Technical University
Faculty of Electrical Engineering

ESW EXAM

Author:
Jakub Marek

September 3, 2019

Contents

In the following section will be terms and their description.

1 Lecture 1

List of terms from the lecture:

- **Source code**
  Code written in some higher programing language like C or JAVA. It is human readeable. It is not directly executable at the given CPU architecture. It has to be compiled (C) into native language or translated into bytecode (JAVA) which can be furher interpreted or just-in-time compiled into native code as well.

- **Assembly code**
  It is human readable source code which represents machine language instructions and data. It is specific to one CPU architecture.

- **Native/machine code**
  Consists of machine instructions directly executable on existing CPU/hardware. Is not human readeable. It is produced by compiler or assembler from assembly code. It is executable only at one CPU architecture.

- **C Compiler**
  Compiler is a program which translates the source code into the native code which can be executed on given CPU architecture, therefore we end up with the executable. If the target CPU architecture is different from the one on which the compilation is invoked, then we speak about the cross-compilation. C compiler works in 4 logical phases:

  1. Pre-processing
     This phase consists of the *removal of the comments*, *expansion of macros*, *expansion of the included files* and *conditional compilation*. Conditional compilation is the process during which the compiler directives are defined to determine which parts of code will be compiled and which will be ignored. It is very usefull during the cross-platform development. The result of the first phase is the intermediate representation which is stored in .i file.
  2. Compilation
     In this phase, the .i file is translated into assembly code, which the assembler is able to understand and further translate. The result is .s file.
  3. Assembly
     In this phase the .s file is translated into .o file which contains machine instructions. The function calls like printf are not resolved in this step.
  4. Linking
     In this phase the function calls are linked with their definitions and final executable file is created. Linker can also make some optimizations. If we write own linker we can make it to put hot functions together to avoid cache eviction or remove unused functions or inline some functions, etc..

- **Java**
  Java is a general-purpose computer programming language that is concurrent, class-based, object-oriented, and specifically designed to have as few implementation dependencies as possible. It is intended to let application developers “write once, run anywhere” (WORA), meaning that compiled Java code can run on all platforms that support Java without the need for recompilation. For example, you can write and compile a Java program on UNIX and run it on Microsoft Windows, Macintosh, or UNIX machine without any modifications to the source code. WORA is achieved by compiling a Java program into an intermediate language called bytecode. The format of bytecode is platform-independent. A virtual machine, called the Java Virtual Machine (JVM), is used to run the bytecode on each platform.

- **Java compiler**
  Because the Java program is platform independent, the compilation is done in two steps. First one involves the translation of java source code into the bytecode, which is platform independent intermediate representation. Java bytecode is then interpreted by the Java Virtual Machine (JVM) which is custom built for each platform.

  1. Compilation
     Java code is translated into the platform-independent bytecode by program called javac. Content of each class contained in the original source code is stored in separate .class file. The conversion from source code to bytecode follows these steps:

- Parse
  Maps the *.java source files into Abstract Syntax Tree (AST).
- Enter
  Enters symbols for definitions into symbol table.
- Process annotations
  If Requested, processes annotations found in the specifed compilation units.
- Attribute
  Attributes the Syntax trees. This step includes name resolution, type checking and constant folding.
- Flow
  Performs dataflow analysis on the trees from the previous step. This includes checks for assignments and reachability.
- Desugar
  Rewrites the AST and translates away some syntactic sugar.
- Generate
  Generates '.Class' files.

2. Execution
   Bytecode can be either interpreted or Just-in-time compiled (JIT compilation) into native code. First, the class loader is called and *.class files are loaded into memory. Then bytecode verifier is called to check that the instructions are not performing any damaging actions and more. Lastly, the code is either interpreted or JIT compiled.

When does java interpret the bytecode and when does it compile it?
The application code is initially interpreted, but the JVM monitors which sequences of bytecode are frequently executed and translates them to machine code for direct execution on the hardware. For bytecode which is executed only a few times, this saves the compilation time and reduces the initial latency; for frequently executed bytecode, JIT compilation is used to run at high speed, after an initial phase of slow interpretation. Additionally, since a program spends most time executing a minority of its code, the reduced compilation time is significant. Finally, during the initial code interpretation, execution statistics can be collected before compilation, which helps to perform better optimization.

- **JVM - Java Virtual Machine**
  JVM(Java Virtual Machine) acts as a run-time engine to run Java applications. JVM is the one that actually calls the main method present in a java code. JVM is a part of JRE(Java Runtime Environment).

  Java applications are called WORA (Write Once Run Anywhere). This means a programmer can develop Java code on one system and can expect it to run on any other Java enabled system without any adjustment. This is all possible because of JVM.

  When we compile a .java file, .class files(contains byte-code) with the same class names present in .java file are generated by the Java compiler. This .class file goes into various steps when we run it. These steps together describe the whole JVM.

- **JIT - Just in time compilation**
  The Just-In-Time (JIT) compiler is a component of the Java Runtime Environment that improves the performance of Java applications at run time. Java programs consists of classes, which contain platform neutral bytecode that can be interpreted by a JVM on many different computer architectures. At run time, the JVM loads the class files, determines the semantics of each individual bytecode, and performs the appropriate computation. The additional processor and memory usage during interpretation means that a Java application performs more slowly than a native application. The JIT compiler helps improve the performance of Java programs by compiling bytecode into native machine code at run time. The JIT compiler is enabled by default, and is activated when a Java method is called. The JIT compiler compiles the bytecode of that method into native machine code, compiling it "just in time" to run. When a method has been compiled, the JVM calls the compiled code of that method directly instead of interpreting it. Theoretically, if compilation did not require processor time and memory usage, compiling every method could allow the speed of the Java program to approach that of a native application. JIT compilation does require processor time and memory usage. When the JVM first starts up, thousands of methods are called. Compiling all of these methods can significantly affect startup time, even if the program eventually achieves very good peak performance.

- **JNI - Java native libraries**
  "Native Library" generally means a non-Java library that's used by the system (so C/C++, etc). Java can load these native libraries through JNI.

- **Processor register**
  In computer architecture, a processor register is a quickly accessible location available to a computer's central processing unit (CPU). Registers usually consist of a small amount of fast storage, although some registers have specific hardware functions, and may be read-only or write-only. It is in the top of the memory hierarchy, therefore it is very fast, but small and expensive. Bellow the registers are caches, RAM, hard drives and tapes.

- **CPU Cache**
  A CPU cache is a hardware cache used by the central processing unit (CPU) of a computer to reduce the average cost (time or energy) to access data from the main memory. A cache is a smaller, faster memory, closer to a processor core, which stores copies of the data from frequently used main memory locations. Most CPUs have different independent caches, including instruction and data caches, where the data cache is usually organized as a hierarchy of more cache levels (L1, L2, L3, L4, etc.).

- **Cache coherence**
  In a shared memory multiprocessor system with a separate cache memory for each processor, it is possible to have many copies of shared data: one copy in the main memory and one in the local cache of each processor that requested it. When one of the copies of data is changed, the other copies must reflect that change. Cache coherence is the discipline which ensures that the changes in the values of shared operands(data) are propagated throughout the system. To ensure the cache coherency, the following protocols are used:

  1. Write through with update protocol
     When a processor writes a new value into its cache, the new value is also written into the memory module that holds the cache block being changed. Some copies of this block may exist in other caches, these copies must be updated to reflect the change caused by the write operation. We update the other cache copies by doing a broadcast with the updated data to all processor modules in the system. Each processor module receives the broadcast data, it updates the contents of the affected cache block if this block is present in its cache.
  2. Write through with invalidation of copies
     When a processor writes a new value into its cache, this value is written into the memory and all other copies in other caches are invalidated. This also done by broadcasting the invalidation request through the system. All caches receives this invalidation request and the cache which contains the updated data flushes its cache line.

- **RAM - Random Access Memory**
  Random-access memory is a form of computer memory that can be read and changed in any order, typically used to store working data and machine code. A random-access memory device allows data items to be read or written in almost the same amount of time irrespective of the physical location of data inside the memory. RAM contains multiplexing and demultiplexing circuitry, to connect the data lines to the addressed storage for reading or writing the entry. Usually more than one bit of storage is accessed by the same address.

- **Pipelining**
  Instruction pipelining is a technique for implementing instruction-level parallelism within a single processor. In each clock cycle, the processor (architecure dependent) can perform following operations at once:

  1. Instruction fetch
  2. Instruction decode and register fetch
  3. Execute
  4. Memory access
  5. Register write back

  If the pipeline of the instructions is prepared correctly, each instruction can go through one of the phases above, therefore we can achieve the parallelism.

- **Branch predictor**
  branch predictor is a digital circuit that tries to guess which way a branch (e.g. an if–then–else structure) will go before this is known definitively. The purpose of the branch predictor is to improve the flow in the instruction pipeline. Branch predictors play a critical role in achieving high effective performance in many modern pipelined microprocessor architectures such as x86.

  Example of 4-stage pipeline. The colored boxes represent instructions independent of each other. Two-way branching is usually implemented with a conditional jump instruction. A conditional jump can either be "not taken" and continue execution with the first branch of code which follows immediately after the conditional jump, or it can be "taken" and jump to a different place in program memory where the second branch of code is stored. It is not known for certain whether a conditional jump will be taken or not taken until the condition has been calculated and the conditional jump has passed the execution stage in the instruction pipeline.

  Without branch prediction, the processor would have to wait until the conditional jump instruction has passed the execute stage before the next instruction can enter the fetch stage in the pipeline. The branch predictor attempts to avoid this waste of time by trying to guess whether the conditional jump is most likely to be taken or not taken. The branch that is guessed to be the most likely is then fetched and speculatively executed. If it is later detected that the guess was wrong then the speculatively executed or partially executed instructions are discarded and the pipeline starts over with the correct branch, incurring a delay.

  The time that is wasted in case of a branch misprediction is equal to the number of stages in the pipeline from the fetch stage to the execute stage. Modern microprocessors tend to have quite long pipelines so that the misprediction delay is between 10 and 20 clock cycles. As a result, making a pipeline longer increases the need for a more advanced branch predictor.

  The first time a conditional jump instruction is encountered, there is not much information to base a prediction on. But the branch predictor keeps records of whether branches are taken or not taken. When it encounters a conditional jump that has been seen several times before then it can base the prediction on the history. The branch predictor may, for example, recognize that the conditional jump is taken more often than not, or that it is taken every second time.

- **Instruction parallelism - Superscalar CPU**
  A superscalar processor is a CPU that implements a form of parallelism called instruction-level parallelism within a single processor. In contrast to a scalar processor that can execute at most one single instruction per clock cycle, a superscalar processor can execute more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to different execution units on the processor. It therefore allows for more throughput (the number of instructions that can be executed in a unit of time) than would otherwise be possible at a given clock rate. Each execution unit is not a separate processor (or a core if the processor is a multi-core processor), but an execution resource within a single CPU such as an arithmetic logic unit.

- **Task parallelism:**

  – Multicore & Multisocket CPUs

    1. Multicore
       A multi-core processor is a computer processor integrated circuit with two or more separate processing units, called cores, each of which reads and executes program instructions, as if the computer had several processors. The instructions are ordinary CPU instructions (such as add, move data, and branch) but the single processor can run instructions on separate cores at the same time, increasing overall speed for programs that support multithreading or other parallel computing techniques. Manufacturers typically integrate the cores onto a single integrated circuit die (known as a chip multiprocessor or CMP) or onto multiple dies in a single chip package. The microprocessors currently used in almost all personal computers are multi-core. A multi-core processor implements multiprocessing in a single physical package. Designers may couple cores in a multi-core device tightly or loosely. For example, cores may or may not share caches, and they may implement message passing or shared-memory inter-core communication methods. Common network topologies to interconnect cores include bus, ring, two-dimensional mesh, and crossbar. Homogeneous multi-core systems include only identical cores; heterogeneous multi-core systems have cores that are not identical. Just as with single-processor systems, cores in multi-core systems may implement architectures such as VLIW, superscalar, vector, or multithreading.

2. Multi-socket
   Before hyper-threading and multi-core CPUs came around, people attempted to add additional processing power to computers by adding additional CPUs. This requires a motherboard with multiple CPU sockets. The motherboard also needs additional hardware to connect those CPU sockets to the RAM and other resources. There's a lot of overhead in this kind of setup. There's additional latency if the CPUs need to communicate with each other, systems with multiple CPUs consume more power, and the motherboard needs more sockets and hardware.

- Hyper-threaded CPU
  Hyper-threading works by duplicating certain sections of the processor—those that store the architectural state—but not duplicating the main execution resources. This allows a hyper-threading processor to appear as the usual "physical" processor and an extra "logical" processor to the host operating system (HTT-unaware operating systems see two "physical" processors), allowing the operating system to schedule two threads or processes simultaneously and appropriately. When execution resources would not be used by the current task in a processor without hyper-threading, and especially when the processor is stalled, a hyper-threading equipped processor can use those execution resources to execute another scheduled task. (The processor may stall due to a cache miss, branch misprediction, or data dependency.)
- NUMA - non-uniform memory access
  NUMA architecture is intended to increase the available bandwidth to the memory and for which it uses multiple memory controllers. It combines numerous machine cores into “nodes” where each core has a memory controller. To access the local memory in a NUMA machine the core retrieves the memory managed by the memory controller by its node. While to access the remote memory which is handled by the other memory controller, the core sends the memory request through the interconnection links. It is overall faster than UMA.
- Out-of-order execution
  The key concept of OoO processing is to allow the processor to avoid a class of delays (termed: "stalls ") that occur when the data needed to perform an operation are unavailable. OoO processors fill these "slots" in time with other instructions that are ready, then re-order the results at the end to make it appear that the instructions were processed as normal.
- Embedded Heterogeneous Systems
  An embedded system or a computing platform would be termed heterogeneous if there are multiple processing elements but of fundamentally different types. For instance, an embedded system that has a CPU and a FPGA would be termed a heterogeneous embedded system because CPU and FPGA are fundamentally different in architecture. Similarly, CPU + DSP Processor would also be termed a heterogeneous system because a DSP processor is different in architecture from a general purpose or embedded CPU architecture. If the CPU itself is a multi-core architecture but the cores are different in their capabilities i.e. at least two cores are different in their architecture, such a multi-core architecture is also termed heterogeneous multi-core and any embedded system based on it would be termed heterogeneous embedded system. Homogeneous multi-core would mean that all cores are identical. Any combination of CPU, GPU, DSP, FPGA, CGRA etc. would be termed a heterogeneous platform. This distinction and focus on heterogeneous architectures is gaining more importance these days because there is an increasing focus on selecting those processing elements which suit best the different tasks at hand in a complex application.

- **Instrumentation** (modification of the code in order to perform a measurements)
  Instrumentation refers to an ability to monitor or measure the level of a product's performance, to diagnose errors, and to write trace information. Programmers implement instrumentation in the form of code instructions that monitor specific components in a system (for example, instructions may output logging information to appear on the screen). When an application contains instrumentation code, it can be managed by using a management tool. Instrumentation is necessary to review the performance of the application.

  - Manual instrumentation
    Manual change of code, for example with use of printf methods in order to find the bottlenecks.
  - Static instrumentation
    Static code analysis is a method of debugging by examining source code before a program is run. It's done by analyzing a set of code against a set (or multiple sets) of coding rules. Static code analysis and static analysis are often used interchangeably, along with source code analysis. This type of

analysis addresses weaknesses in source code that might lead to vulnerabilities. Of course, this may also be achieved through manual code reviews. But using automated tools is much more effective.

  - Dynamic instrumentation
    Dynamic Binary Instrumentation (DBI) is a method of analyzing the behavior of a binary application at runtime through the injection of instrumentation code. This instrumentation code executes as part of the normal instruction stream after being injected. In most cases, the instrumentation code will be entirely transparent to the application that it's been injected to. Analyzing an application at runtime makes it possible to gain insight into the behavior and state of an application at various points in execution. This highlights one of the key differences between static binary analysis and dynamic binary analysis. Rather than considering what may occur, dynamic binary analysis has the benefit of operating on what actually does occur. This is by no means exhaustive in terms of exercising all code paths in the application, but it makes up for this by providing detailed insight into an application's concrete execution state.

- **Performance counters**
  Hardware performance counters, or hardware counters are a set of special-purpose registers built into modern microprocessors to store the counts of hardware-related activities within computer systems. The number of available hardware counters in a processor is limited while each CPU model might have a lot of different events that a developer might like to measure. Each counter can be programmed with the index of an event type to be monitored, like a L1 cache miss or a branch misprediction. Compared to software profilers, hardware counters provide low-overhead access to a wealth of detailed performance information related to CPU's functional units, caches and main memory etc. Another benefit of using them is that no source code modifications are needed in general. However, the types and meanings of hardware counters vary from one kind of architecture to another due to the variation in hardware organizations. There can be difficulties correlating the low level performance metrics back to source code. The limited number of registers to store the counters often force users to conduct multiple measurements to collect all desired performance metrics.

- **Heap profiling**
  Helps to determine the footprint of our program in the memory.

- **Event sampling**
  When the interesting event occurs, like cache miss, branch-prediction miss, page fault or similar event, we look where the program executes. The result is histogram with the adresses and event counts.

- **PERF**
  Perf is a performance analyzing tool in Linux. Userspace controlling utility, named perf, is accessed from the command line and provides a number of subcommands; it is capable of statistical profiling of the entire system (both kernel and userland code). It supports hardware performance counters, tracepoints, software performance counters, and dynamic probes.

List of extra terms from the lecture:

- Microcode
- ISA
- Virtual memory
- MMU
- Bus
- Arbiter
- OS Kernel
- Language runtime
- Application framework

Sources:

- https://www.quora.com/What-is-the-difference-between-bytecode-native-code-machine-code-and-assembly-code

- https://www.geeksforgeeks.org/compiling-a-c-program-behind-the-scenes/
- https://www.quora.com/Why-use-conditional-compilation-in-c-programming
- https://howtodoinjava.com/java/basics/what-is-java-programming-language/
- https://www.geeksforgeeks.org/compilation-execution-java-program/
- https://www.geeksforgeeks.org/compilation-execution-java-program/
- https://stackoverflow.com/questions/1326071/is-java-a-compiled-or-an-interpreted-programming-language
- https://www.geeksforgeeks.org/jvm-works-jvm-architecture/
- https://aboullaite.me/understanding-jit-compiler-just-in-time-compiler/
- https://stackoverflow.com/questions/2860642/java-difference-between-library-and-native-library
- https://en.wikipedia.org/wiki/Processor_register
- https://en.wikipedia.org/wiki/Memory_hierarchy
- https://en.wikipedia.org/wiki/CPU_cache
- https://en.wikipedia.org/wiki/Cache_coherence
- https://medium.com/@TechExpertise/cache-coherence-problem-and-approaches-a18cdd48ee0e
- https://en.wikipedia.org/wiki/Random-access_memory
- https://en.wikipedia.org/wiki/Instruction_pipelining
- https://en.wikipedia.org/wiki/Branch_predictor
- http://web.ist.utl.pt/luis.tarrataca/classes/computer_architecture/Chapter16-InstructionLevelParallelismAndSupersc
- https://en.wikipedia.org/wiki/Superscalar_processor
- https://en.wikipedia.org/wiki/Multi-core_processor
- https://www.howtogeek.com/194756/cpu-basics-multiple-cpus-cores-and-hyper-threading-explained/
- https://en.wikipedia.org/wiki/Hyper-threading
- https://techdifferences.com/difference-between-uma-and-numa.html
- https://simple.wikipedia.org/wiki/Out-of-order_execution
- https://www.quora.com/What-is-a-heterogeneous-embedded-system
- https://en.wikipedia.org/wiki/Instrumentation_(computer_programming)
- https://www.perforce.com/blog/sca/what-static-code-analysis
- https://en.wikipedia.org/wiki/Hardware_performance_counter
- https://courses.cs.washington.edu/courses/cse326/05wi/valgrind-doc/ms_main.html
- https://en.wikipedia.org/wiki/Perf_(Linux)

2 Lecture 2

- **JVM**
  See 1
- **JVM Bytecode**
  See 1. Each bytecode is composed of one byte that represents the opcode, along with zero or more bytes for operands.
  - OPcode
  - OPcode parameters
- **Assembly code**
  See 1.
- **JVM - memory layout**
  Memory in Java is separated into thread specific part and shared memory by all threads. First mentioned contains the stack for each thread. Everytime the thread invokes a method, the frame for the given method is stored to the threads stack in thread specific stack. The heap contains all the objects which are created during the runtime.
- **Stack**
  Java Stack memory is used for execution of a thread. They contain method specific values that are short-lived and references to other objects in the heap that are getting referred from the method. Stack memory is always referenced in LIFO (Last-In-First-Out) order. Whenever a method is invoked, a new block is created in the stack memory for the method to hold local primitive values and reference to other objects in the method. As soon as method ends, the block becomes unused and become available for next method. Stack memory size is very less compared to Heap memory.
- **Heap**
  Java Heap space is used by java runtime to allocate memory to Objects and JRE classes. Whenever we create any object, it's always created in the Heap space. Garbage Collection runs on the heap memory to free the memory used by objects that doesn't have any reference. Any object created in the heap space has global access and can be referenced from anywhere of the application.
- **Frame**
  Each thread has a stack with the frames. The frame is created each time the method is invoked.
  - Interpreted frame
    Per exactly one method.
  - Compiled frame
    Also includes all in-lined methods.
- **OOP - Ordinary Object Pointers**
  That is how are called the references in JVM.
- **Class file**
  A Java class file is a file (with the .class filename extension) containing Java bytecode that can be executed on the Java Virtual Machine (JVM). A Java class file is usually produced by a Java compiler from Java programming language source files (.java files) containing Java classes. If a source file has more than one class, each class is compiled into a separate class file. There are 10 basic sections to the Java Class File structure:
  1. Magic Number
  2. Version of Class File Format
  3. Constant pool
  4. Access Flags
  5. This Class
  6. Super Class
  7. Interfaces
  8. Fields

9. Methods
10. Attributes

- **JAVAP - disassembler**
  Disassembler is a computer program that translates machine language into assembly language.
- **Decompiler**
  Decompiler is a computer program that takes an executable file as input, and attempts to create a high level source file which can be recompiled successfully.
- **StackMapTable**
  StackMapTable is an attribute in classes compiled with Java 6 or higher. It is used by the JVM during the process of verification by type checking. Basically, a stack map frame defines the expected types of local variables and the operand stack (i.e. the state of the frame) of a method during its execution. During run-time, The JVM throws the VerifyError if expected and actual types are incompatible.
- **Obfuscation**
  Code obfuscation is the act of deliberately obscuring source code, making it very difficult for humans to understand, and making it useless to hackers who may have ulterior motives. it may also be used to deter the reverse-engineering of software.
- **JIT**
  See 1.
- **Hot spots**
  Frequently executed parts of code. Should be target of the heavy optimization.
- **Execution profile**
  Basically the data which shows the usage of code during the runtime or multiple executions.
- **Dynamic re-compilation**
  JIT includes also the adaptive optimization. Depending on current execution profile, JIT compilator can invoke the dynamic re-compilation to better optimize the code and speed it up.
- **JIT compilers:**
  There are two basic types of Just-In-Time Java compilers: Client and Server. Traditionally, the Client and the Server compilers are called C1 and C2 respectively. The main difference between the Client and the Server compilers is the aggressiveness in the way they compile code. The Client compiler is optimized to make an application start up faster, whereas the Server compiler gives better performance in the long run. As you might have guessed, the Client compiler is dedicated to any type of client application (usually GUI-based), whereas the Server compiler is designed for long-running server side applications. Why is the Server compiler faster eventually? It's because the Server compiler observes and analyzes the code for a longer period of time, and that knowledge allows the Server compiler to make better optimizations in the compiled code. On the contrary, the Client compiler tries to optimize and compile code as soon as possible, which lowers the start-up time.
  - C1 compiler (CLIENT)
    - simplified inlining, using CPU registers
    - window-based optimization over small set of instructions
    - intrinsic functions with vector optimizations SIMD
  - C2 compiler (SERVER)
    - dead code elimination
    - loop unrolling
    - loop invariant hoisting
    - common sub-expression elimination
    - constant propagation
    - full inlining
    - full deoptimization
    - escape analysis

    - null check elimination
    - pattern-based loop vectorization and super word packing (SIMD)

- **Tiered Compilation**
  With Tiered compilation, the Client compiler is used at the beginning to make startup fast, then, when the code becomes hot and profile data is collected, it is recompiled by the Server compiler. Even though there are only two basic compilers (+ interpreter) in Java, there are five levels of execution, because the Client compiler (C1) has three different levels of compilation and the Server compiler (C2) has only one.

  1. Level 0 – interpreted code
  2. Level 1 – simple C1 compiled code (with no profiling)
  3. Level 2 – limited C1 compiled code (with light profiling)
  4. Level 3 – full C1 compiled code (with full profiling)
  5. Level 4 – C2 compiled code (uses profile data from the previous steps)

  The usual path is 0 -> 3 -> 4, so the code is interpreted first, then, when it gets hot enough, it's compiled by C1 with full profiling enabled and, finally, C2 compiles the code using profile data collected by C1.

- **On-stack replacement**
  On-stack replacement (OSR) is a programming language implementation technique that allows a running program to switch to a different version of code. For example, a program could start executing optimized code, and then transfer to and start executing unoptimized code. This was the original use case for OSR, to facilitate debugging of optimized code. After its original use was established, OSR shifted to a different use case: optimizing programs. OSR allows the run-time system to detect if a program is executing an inefficient loop, recompile and optimize the method that contains the loop, and then transfer control to the newly compiled method. Another strategy is to optimize code based on some assumptions, then, if the assumptions are invalidated at run-time, transfer control back to the original, unoptimized code.

- **SISD vs SIMD**
  Single instruction, multiple data (SIMD) is a class of parallel computers in Flynn's taxonomy. It describes computers with multiple processing elements that perform the same operation on multiple data points simultaneously. Such machines exploit data level parallelism, but not concurrency: there are simultaneous (parallel) computations, but only a single process (instruction) at a given moment.

- **TLAB**
  TODO

- **Safepoint:**
  A safepoint is a range of execution where the state of the executing thread is well described. Safepoints are a common JVM implementation detail At a safepoint the mutator thread is at a known and well defined point in it's interaction with the heap. This means that all the references on the stack are mapped (at known locations) and the JVM can account for all of them. As long as the thread remains at a safepoint we can safely manipulate the heap + stack such that the thread's view of the world remains consistent when it leaves the safepoint.

  - Global safepoint
    All threads are stoped (Stop the world). It is used for garbage collection,...
  - Local safepoint
    Just the executing thread is in the safepoint.
  - TTSP - time to safe point
    Suspension needs to be as quick as possible, because the time taken contributes to the overall GC pause. Therefore this Time To Safe Point (TTSP) as it's known, needs to be minimised.

- **Escape analysis**
  Escape analysis is the process that the compiler uses to determine the placement of values that are created by your program. Specifically, the compiler performs static code analysis to determine if a value can be placed on the stack frame for the function constructing it, or if the value must "escape" to the heap.

- **Strength reduction**
  In compiler construction, strength reduction is a compiler optimization where expensive operations are replaced with equivalent but less expensive operations.

- CPU and Memory profiling: (slide 43)
  - Profiling
    We can profile the CPU usage, which means the time spent in the methods or memory usage and overall allocations.
  - Modes
    We can perform sampling which means to periodicaly check the stact of running threads. No bytecode modifications are needed. It is done during the TTSP therefore it has 1-2 percent impact on the performance. Or we can perform tracing (instrumentation) which modifies the bytecode and also has an effect on the performance of the app.
  - jvisualvm
    It is a monitoring, troubleshooting and profiling tool in java.
  - jmc
    The same as above.
- **Warm-up time**
  During the first few launches of the app, the JVM is performing the tiered compilation and it takes few number of executions before the code is optimized. That influences the performance. We should be aware of that if we want to benchmark our code and not the compilers and optimizations done by them.
- **Microbenchmarking**
  Measure of the performance of the small peace of the code.
- **Macrobenchmarking**
  Measure of the performance of the whole app.

List of extra terms from the lecture:

- Descriptor
- Signature
- Full inline
- Dead code elimination
- Loop invarian hoisting
- 32 vs 64 bit JVM

Sources:

- https://en.wikipedia.org/wiki/Java_bytecode
- https://www.journaldev.com/4098/java-heap-space-vs-stack-memory
- https://en.wikipedia.org/wiki/Java_class_file
- https://en.wikipedia.org/wiki/Disassembler
- https://en.wikipedia.org/wiki/Decompiler
- https://sweetcode.io/importance-of-code-obfuscation/
- https://stackoverflow.com/questions/37309074/what-is-stackmap-table-in-jvm-bytecode/37310409
- https://dzone.com/articles/client-server-and-tiered-compilation
- http://prl.ccs.neu.edu/blog/2019/01/28/on-stack-replacement/
- https://en.wikipedia.org/wiki/SIMD
- https://medium.com/software-under-the-hood/under-the-hood-java-peak-safepoints-dd45af07d766
- https://mattwarren.org/2016/08/08/GC-Pauses-and-Safe-Points/
- https://www.ardanlabs.com/blog/2017/05/language-mechanics-on-escape-analysis.html
- https://en.wikipedia.org/wiki/Strength_reduction

3 Lecture 3

- **Benchmarking**
  The act of running a computer program, a set of programs, or other operations, in order to assess the relative performance of an object, normally by running a number of standard tests and trials against it. Also it can be a benchmarking program itself.
  - Micro-Benchmark
    Evaluates very little part of an application. It is easy to determine source of speed-up/slow down. It does not typically imply the improvements of application performance if you speed up the benchmarked part.
  - Application Benchmark
    Evaluate performance of the whole applications. Performance of the application is influenced by many real-world factors. For complex applications it might be difficult to determine the source of speed-up or slow-down.
- **Measuring energy**
  One of the variables which we are interested in is the energy consumption by our application. Typically, the efficiently written app is also energy consumption friendly. We can use power meter to directly measure consumed energy or more common these days is to use hardware-provided interfaces like the **RAPL**.
- **RAPL**
  Running Average Power Limit (RAPL) is Intel feature of some interl CPUs. Linux Perf is able to use it for the performance measurements as well. RAPL provides a set of counters providing energy and power consumption information. RAPL is not an analog power meter, but rather uses a software power model. This software power model estimates energy usage by using hardware performance counters and I/O models. Based on these measurements, they match actual power measurements. RAPL provides a way to set power limits on processor packages and DRAM. This will allow a monitoring and control program to dynamically limit max average power, to match its expected power and cooling budget.
- **Measuring memory consumption** Under modern OSes, measuring the memory consumption is surprisingly complex, because memory is used by program itself, the other programs and processes, operating system kernel and others.
  - Program memory
    Consists of code itself, static data, heap, stack,... Stack is allocated for each thread.
  - Operating system kernel memory Allocated by OS Kernel on behalf the application. Sometimes it is not possible to account this memory to individual process, for example the network buffers.
  - Shared libraries
    We can't easily to determine the memory consumption of shared libraries used by several programs.
  - TOP & HTOP
    These are linux tools for reporting several memory statistics like total amount of virtual memory reserved by process, currently resident physical memory, memory shared with other processes and other.
- **Measuring execution time**
  Is the time for how long the program is running (equals to run time). Execution time exhibits variations. It is influenced by many factors like Hardware, input data, compiler, memory layout, measuring overhead, rest of the system, network load,... Some factors can be controlled and some cannot.
  - Timestamping
    A timestamp is a sequence of characters or encoded information identifying when a certain event occurred, usually giving date and time of day, sometimes accurate to a small fraction of a second. In computing timestamping refers to the use of an electronic timestamp to provide a temporal order among a set of events. It is usually achieved by:
    * Use of system calls
      Linux example: gettimeofday, clock_gettime,... Resolution depends on available hardware but can be down to 1ns. Unfortunatelly the overhead is hundreds of CPU cycles.

  - Use of hardware directly
    Sometimes called timestamp counter. For example TSC register on x86 architecture. The resolution is 1 CPU cycle with overhead ~8 cycles. The con is that it can be subject of CPU frequency scaling and TSC counter on different cores or sockets may not be synchronized.
  - Combination of both - Virtual syscall
    Reading TSC is fast, but HW/frequency/socket dependent. The OS knows all the information about HW/frequency/socket but calling Kernel directly has big overhead. Idea of virtual syscall is that the Kernel publishes enough data to user space to reliably convert TSC without calling of the Kernel. For that the **VDSO** are used - Virtual Dynamic Shared Object. It is Kernel memory mapped to process address space and looks like a shared library. gettimeoftheday and clock_gettime are implemented in VDSO.

- Benchmark design
  The goal of our benchmark is to estimeate a confidence interval for mean of execution time of a given benchmark on one or more platforms. The mean is the property of the probability of distribution of the random execution times. We can only estimate the mean value from the measurements. Results variance occurs typically at multiple levels, e.g.:
  - (re-)compilation
  - execution
  - iteration inside a program

  Sound benchmark methodology should evaluate all the levels with random variables. We encounter three kinds of variables influencing the results:
  - Controlled variables
    e.g. compiler flags, hardware, algorithm changes. We are interested how these influences the results.
  - Random variables
    e.g. hardware interrupts, OS scheduler. We are interested in statistical properties of our results in face of these variables.
  - Uncontrolled variables
    Mostly fixed but can cause the bias to our results.
- Summarizing benchmark results
  We want to answer the question: Is it likely that two systems have different performance? Or better question is to ask for a speedup of two systems. Statistics can answer this question with the significance testing. For that we can use a ministat tool at Linux. The test should be visualized. If the two confidence intervals overlap then we need more statistics to determine the differences of the performance. If the two confidence intervals don't overlap, it is likely the two systems have different performance. It is hard to estimate the speedup and its confidence intervals. Analysis of the results should be statistically rigorous and in particular should quantify any variation.
- Repearing iterations
  Iteration = one execution of a loop body. We are interested in steady state performance. Initialization phase - First few iterations typically include the initialization overheads composed of warming up caches, teaching the branch predictor, memory allocations, etc. Independent state - it is already initialized. Should be independent on previous measurements and ideally the measurements should be identically independently distributed (iid). To determine if the benchmark is in independent state we have to visualize the measurements and manually inspect the graph. If no patterns occur, the measurements are iid. We should use the manual inspection only once to determine the minimum number of iterations needed to reach iid state. If the app does not reach this state in reasonable amount of iterations, we should use same iteration from each run for our measurements.
- Repeating executions
  Runnuing a benchmark program multiple times (effect of JIT compiler etc.). We should determine independent state for executions the same way as for iterations.
- Repeating compilation
  Sometimes even a compiler can influence the benchmark results. For example by the code layout produced by the compiler. If we cannot control some factor of benchmark we need to randomize it!
- Multi-level repetition
  All experiments and measurements have to be repeated to narrow the confidence intervals. If the variance occurs at higher levels (execution, compilation) we need to repeat at least at that level. We

always want to find the minimum number of repetitions. This can be formulated mathematically, the equations are in the lecture slides.

Sources:

- https://01.org/blogs/2014/running-average-power-limit-—rapl
- https://stackoverflow.com/questions/63166/how-to-determine-cpu-and-memory-consumption-from-inside-a-process
- https://en.wikipedia.org/wiki/Run_time_(program_lifecycle_phase)
- https://en.wikipedia.org/wiki/Timestamping_(computing)

4 Lecture 4

- **C/C++ compiler**
  You can read more at 1. Mostly used open source compilers are gcc and clang.
  - Frontend
    All important is in 1. Moreover to say, it consist of parser and preprocessor.
  - Optimization passes
    These are done by the compiler. The optimization is done on intermediate representation and optimized code can be observed in assembly language. More to be read is in 1.
  - Backend
    All important is in 1.
  - Linker
    All important is in 1.
- **Pointer Aliasing**
  Multiple pointers of the same type can point to same memory. This prevents certain optimizations.
- **Restrict**
  C qualifier which tells that the pointers want alias.
- **AST - Abstract syntax tree**
  The frontend of the compiler first creates AST out of the code. The AST is used intensively during semantic analysis, where the compiler checks for correct usage of the elements of the program and the language. The compiler also generates symbol tables based on the AST during semantic analysis. A complete traversal of the tree allows verification of the correctness of the program. After verifying correctness, the AST serves as the base for code generation. The AST is often used to generate an intermediate representation (IR), sometimes called an intermediate language, for the code generation.
- **IR - Intermediate representation**
  The intermediate representation is produced by conversion of AST, usually by dumb expansion of the templates. An Intermediate representation is the data structure or code used internally by a compiler or virtual machine to represent source code. An IR is designed to be conducive for further processing, such as optimization and translation.
- **Common optimization options**
  Compilers can do many and many optimizations in high level as well as low level. High-level optimization is a language dependent type of optimization that operates at a level in the close vicinity of the source code. High-level optimizations include inlining where a function call is replaced by the function body and partial evaluation which employs reordering of a loop, alignment of arrays, padding, layout, and elimination of tail recursion. Low-level optimization is highly specific to the type of architecture. This includes for example: register allocation, instruction scheduling, floating-point units utilization, branch prediction, peephole and profile-based optimization,... Common optimization flags are:
  1. O1 Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function.

2. O2 Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. The compiler does not perform loop unrolling or function inlining when you specify -O2. As compared to -O, this option increases both compilation time and the performance of the generated code.
3. O3 Optimize yet more. -O3 turns on all optimizations specified by -O2 and also turns on the -finline-functions, -funswitch-loops, -fpredictive-commoning, -fgcse-after-reload and -ftree-vectorize options.
4. O0 Reduce compilation time and make debugging produce the expected results. This is the default.
5. Os Optimize for size. -Os enables all -O2 optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size.

- **Dead store elimination pass**
  It is best to explain it on the example. Let's have a variable a in which we store a value 1 and right begind it, value 2. Without the dead store elimination pass, the assembly code would contain both stores into this variable despite the fact, the first value is never used. With dead store elimination pass, the first store never appears in assembly code, therefore the final code is faster and shorter.

- **CFG**
  A control flow graph (CFG) in computer science is a representation, using graph notation, of all paths that might be traversed through a program during its execution. The reason for drawing one would be to determine all possible paths taken by the program, which may help us determine things like test coverage without actually running the program (static analysis).

- **SROA**
  An optimization pass providing Scalar Replacement of Aggregates. This pass takes allocations which can be completely analyzed (that is, they don't escape) and tries to turn them into scalar SSA values.

- **Global variable optimizer**
  Global variables can be modified indirectly via pointers or function calls. This typically means that the compiler cannot keep the value of a global variable in a register; it must write the value back to memory and re-load it each time it is used. For this reason, the use of global variables should be avoided in worst-case hotspot code. If the use of global variables is unavoidable, then it can be more efficient to cache the value into a local variable, which is then used extensively before its value is written back into the global variable.

- **Profile-guided optimization**
  Optimization techniques based on analysis of the source code alone are based on general ideas as to possible improvements, often applied without much worry over whether or not the code section was going to be executed frequently though also recognising that code within looping statements is worth extra attention. Rather than programmer-supplied frequency information, profile-guided optimisation uses the results of profiling test runs of the instrumented program to optimize the final generated code. The compiler is used to access data from a sample run of the program across a representative input set. The results indicate which areas of the program are executed more frequently, and which areas are executed less frequently. All optimizations benefit from profile-guided feedback because they are less reliant on heuristics when making compilation decisions. The caveat, however, is that the sample of data fed to the program during the profiling stage must be statistically representative of the typical usage scenarios; otherwise, profile-guided feedback has the potential to harm the overall performance of the final build instead of improving it. Just-in-time compilation can make use of runtime information to dynamically recompile parts of the executed code to generate a more efficient native code. If the dynamic profile changes during execution, it can deoptimize the previous native code, and generate a new code optimized with the information from the new profile.

- **Volatile**
  C's volatile keyword is a qualifier that is applied to a variable when it is declared. It tells the compiler that the value of the variable may change at any time–without any action being taken by the code the compiler finds nearby. A variable should be declared volatile whenever its value could change unexpectedly. In practice, only three types of variables could change:

  1. Memory-mapped peripheral registers
  2. Global variables modified by an interrupt service routine
  3. Global variables accessed by multiple tasks within a multi-threaded application

In short, this qualifier tells to compiler not to optimize the access to the variable.

- **Execution**
  A system that executes a program is called an interpreter of the program. Loosely speaking, an interpreter actually does what the program says to do. This contrasts with a language translator that converts a program from one language to another. The most common language translators are compilers. Translators typically convert their source from a high-level, human readable language into a lower-level language (sometimes as low as native machine code) that is simpler and faster for the processor to directly execute. The idea is that the ratio of executions to translations of a program will be large; that is, a program need only be compiled once and can be run any number of times. This can provide a large benefit for translation versus direct interpretation of the source language. An executable is invoked, most often by an operating system, which loads the program into memory (load time), possibly performs dynamic linking, and then begins execution by moving control to the entry point of the program; all these steps depend on the Application Binary Interface of the operating system. At this point execution begins and the program enters run time. The program then runs until it ends, either normal termination or a crash.

- **Statically linked binaries**
  statically-linked library is a set of routines, external functions and variables which are resolved in a caller at compile-time and copied into a target application by a compiler, linker, or binder, producing an object file and a stand-alone executable.

- **Dynamically linked binaries**
  Dynamic linking or late binding is linking performed while a program is being loaded (load time) or executed (run time), rather than when the executable file is created. A dynamically linked library (dynamic-link library, or DLL, under Windows and OS/2; dynamic shared object, or DSO, under Unix-like systems) is a library intended for dynamic linking. Only a minimal amount of work is done by the linker when the executable file is created; it only records what library routines the program needs and the index names or numbers of the routines in the library. The majority of the work of linking is done at the time the application is loaded (load time) or during execution (run time). Usually, the necessary linking program, called a "dynamic linker" or "linking loader", is actually part of the underlying operating system.

- **Shared library**
  A shared library or shared object is a file that is intended to be shared by executable files and further shared object files. Modules used by a program are loaded from individual shared objects into memory at load time or run time, rather than being copied by a linker when it creates a single monolithic executable file for the program. Shared libraries can be statically linked, meaning that references to the library modules are resolved and the modules are allocated memory when the executable file is created. But often linking of shared libraries is postponed until they are loaded. Most modern operating systems can have shared library files of the same format as the executable files. This offers two main advantages: first, it requires making only one loader for both of them, rather than two (having the single loader is considered well worth its added complexity). Secondly, it allows the executables also to be used as shared libraries, if they have a symbol table.

- **Lazy linking**
  Lazy loading is a design pattern commonly used in computer programming to defer initialization of an object until the point at which it is needed. It can contribute to efficiency in the program's operation if properly and appropriately used. The opposite of lazy loading is eager loading.

- **Bentley's rules**
  The aim of Bentley's rules is to help you to produce an effective code and to avoid the mistakes many make.

  - Modifying data
    Aim is to modify data structures used in order to make more effective application.

    * Space for time
      We increase a memory demand of data structures in order to save time.

      · Data structure augmentation
        Add some more info to the data structures to make common operations quicker.

      · Storing precomputed results
        Store the results of a previous calculation. Reuse precomputed results than redoing the calculation.

      - Caching
        Store some of the heavily used/recently used results so they don't need to be computed.
      - Lazy evaluation
        Differ the computation until the results are really needed.
    * Time for space Aim is to sacrifice time efficiency for memory efficiency.
      - Packing/Compression
        Reduce the space of the data by storing them as "processed" which will require additional computation to get the data.
      - Interpreters
        Instead of writing a program to do a computation, use a language to describe the computation at a high level and write an interpreter for that language.
    * Space and time Aim is to save memory as well as be time efficient.
      - SIMD
        Store short width data packed into the machine word. It leads to single operation on all the data items. Win-win both faster and less storage.

- Modyfying code
  Aim is to modify code to be more efficient.
    * Loop rules
      - Loop invariant code motion
        Move as much code as possible out of the loops.
      - Sentinel loop exit test
        When we iterate over a data to find a value, we have to check the end of the data as well as for the value. Add an extra data item at the end that matches the test.
      - Loop elimination by unrolling
        If we fully know the loop bounds, we can fully unroll the loop.
      - Partial loop unrolling
        Make a few copies of the loop body.
      - Loop fusion
        When multiple loops iterate over the same set of the data, put the computation in one loop body.
      - Eliminate wasted iterations
        Change the loop bounds so that it will not iterate over an empty loop body.
    * Logic rules
      - Exploit algebraic identities
        If the evaluation of a logical expression is costly, replace it by algebraically equivalent expression.
      - Short circuit monotone functions
        In checking if a monotonically increasing function is over a threshold, don't evaluate beyond the threshold.
      - Reordering tests
        Logical test should be arranged such that inexpensive and often succesful tests precede expensive and rarely succesful ones. Add inexpensive and often succesful test before expensive ones.
      - Precompute logic functions
        A logical function over a small finite domain can be replaced by a lookup in a table that represents the domain.
      - Boolean variable elimination
        Replace the assingment to a boolean variable by re-placing it by an IF-THEN-ELSE.
    * Procedure rules
      - Collapse procedure hierarchies
        Inline small functions into the main body.
      - Coroutines
        Multiple passes over data should be combined.
      - Tail recursion elimination
        In a self recursive function, if the last action is calling itself, eliminate recursion.

    - Expression rules
        - Compile-time initialization
          If a value is a constant, make it a compile-time constant.
        - Common subexpression elimination
          If the same expression is evaluated twice, do it only once.
        - Pairing computation
          If two similar functions are called with the same arguments close to each other in many occasions combine them.
    - Parallelism rules
        - Exploit implicit parallelism
          Reduce the loop carried dependences so that "software pipelining" can execute a compact schedule without stalls.
        - Exploit inner loop parallelism
          Facilitate inner loop vectorization.
        - Exploit coarse grain parallelism
          Outer loop parallelism, task parallelism.
        - Extra computation to create parallelism
          In many cases doing a little more work can make a sequential program a parallel one.

Sources:

- https://stackoverflow.com/questions/2391442/how-to-see-the-optimized-code-in-c
- https://en.wikipedia.org/wiki/Abstract_syntax_tree
- https://en.wikipedia.org/wiki/Intermediate_representation
- https://apachebooster.com/kb/what-is-code-optimization-and-its-types/
- https://stackoverflow.com/questions/50913708/how-to-draw-a-control-flow-graph-from-this-code
- https://en.wikipedia.org/wiki/Control-flow_graph
- https://llvm.org/doxygen/classllvm_1_1SROA.html#details
- https://www.rapitasystems.com/software_optimization_techniques_14
- https://en.wikipedia.org/wiki/Profile-guided_optimization
- https://barrgroup.com/Embedded-Systems/How-To/C-Volatile-Keyword
- https://en.wikipedia.org/wiki/Execution_(computing)
- https://www.geeksforgeeks.org/program-execution-in-the-cpu/
- https://www.geeksforgeeks.org/how-does-a-c-program-executes/
- https://en.wikipedia.org/wiki/Static_library

5 Lecture 5

- **Synchronization**
  Problem of todays effective programming is that we have multi-core or multi-CPU systems and to take an advantage of such a computational resources we need to program parallel (multi-threaded) algorithms. The threads are usually communicating, which means they operate on same data and therefore they need to be synchronized. Classical approaches are:
    - Mutual exclusion
    - Producer-consumer
    - Kernel Semaphore
    - Futex

- **Critical section**
  Code in the "locked" section. Data should be modified only by one thread at a time.
- **Deadlock**
  State of the program when all threads are waiting on each other and none can make any action, therefore the program stays in this state infinately.
- **Mutual exclusion**
  The problem which mutual exclusion addresses is a problem of resource sharing: how can a software system control multiple processes' access to a shared resource, when each process needs exclusive control of that resource while doing its work? The mutual-exclusion solution to this makes the shared resource available only while the process is in a specific code segment called the critical section. It controls access to the shared resource by controlling each mutual execution of that part of its program where the resource would be used. A successful solution to this problem must have at least these two properties: It must implement mutual exclusion: only one process can be in the critical section at a time. It must be free of deadlocks: if processes are trying to enter the critical section, one of them must eventually be able to do so successfully, provided no process stays in the critical section permanently.
- **Producer-consumer**
  The producer–consumer problem is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate data, put it into the buffer, and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

  The solution for the producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer. The solution can be reached by means of inter-process communication, typically using semaphores. An inadequate solution could result in a deadlock where both processes are waiting to be awakened. The problem can also be generalized to have multiple producers and consumers.
- **Atomic operations**
  These operations are uninterruptable even on the hardware level. You can avoid mutual exclusion using atomic operations. When a thread performs an atomic operation, the other threads see it as happening instantaneously. The advantage of atomic operations is that they are relatively quick compared to locks, and do not suffer from deadlock and convoying. The disadvantage is that they only do a limited set of operations, and often these are not enough to synthesize more complicated operations efficiently. But nonetheless you should not pass up an opportunity to use an atomic operation in place of mutual exclusion. Atomic operations are costly, 19-266 times than non-atomic operations.
- Kernel semaphores
  Semaphore as used in an operating system to restrict the access to resources by multiple processes at the same time. When a semaphore is used to restrict the access to only one process at a time, it is termed as mutex, as it serves the purpose of mutual exclusion. The part of the program which accesses the shared resource is called as the critical section. Hence a semaphore restricts the execution of the critical section by multiple processes at the same time. Traditionally semaphores have two functions associated with it, called "P" and "V" . Every process before entering a critical section calls the function "P", if no other process is executing the critical section the semaphore is held by the process and the process is allowed to enter the critical section. In case the semaphore is already held by some other process, that is some other process is executing the critical section, then "P" puts the current process to sleep. The function "V" on the other hand releases the semaphore that is being held by the process. In linux the equivalent calls for P and V are down() and up() respectively. When a process calls down() the value of the semaphore is decremented and if value after decrementing is zero or greater than zero the process is allowed to enter the critical section. But on the other hand if the new value is negative then the process is put to sleep on a wait queue. On the other hand when a process calls up() the value of the semaphore is incremented by one. When we want to use the semaphore as a mutex, the value of semaphore is initialized to 1. So at any give time only one process can execute the critical section. Unfortunatelly, the overhead for system call is about 100 cycles. It is preferable to use fine-graining to protect as little as possible and to prevent contention. If fine-graining is effective, lock is not contended and threads don't have to sleep, but we always have to pay system overhead which is not effective.

- Futex

- **Barrier**
  Macro barrier is only the compiler barrier therfore compiler won't optimize it out, but hardware still can, for example in case that the process is waiting for cache-line ownership and so it first makes a read and write after. To solve this problem, there are memory barrier instructions like mfence. Barriers cost is lower than the cost of atomic operations, the weaker the barrier, the lower the cost.

- Order
  - Relaxed
  - Consume
  - Acquire
  - Release
  - Acq_rel
  - Seq_cst

- **Locking overhead** Classical locks are typically implemented with atomic instructions and ensure that lock manipulation is no reordered with critical section content. Locking overhead is caused by extra resources for using locks, like the memory space allocated for locks, the CPU time to initialize and destroy locks, and the time for acquiring or releasing locks. The more locks a program uses, the more overhead associated with the usage. Contention occurs whenever one process or thread attempts to acquire a lock held by another process or thread. The more fine-grained the available locks, the less likely one process/thread will request a lock held by the other. (For example, locking a row rather than the entire table, or locking a cell rather than the entire row.)
  - Unconteded case
    During lock(), mutex is not in the cache, during unlock() it is.
  - Conteded case
    During lock(), mutex is not in the cache and even during unlock() mutex is not in the cache because some other CPU tried to lock the data, thus it stole mutex from the cache of mutex-owner.

- **Futex**
  It is fast userspace mutex. It's advantage is that unconteded mutex never goes to kernel (saves sys call overhead). futex_wait and futex_wake are system calls which are used only in conteded case.

- **Problem with mutex**
  In nowadays applications, the workload of processes/threads is mostly focused on reading the shared data. Therefore the mutex, which prohibits others to use same data leads to dead time when no one else can access data even if all processes wants just to read it. Next methods are to solve this problem and are also scalable.

- **Read-write lock**
  Classical solution. Multiple readers can access memory simultaneously and only the update blocks all readers. Can be implemented on top of the mutexes, but badly scales with number of updaters and readers and still results into dead time which we want to avoid.

- **Read-copy-update**
  Scalable solution to above mentioned problems. Updaters don't block readers. Locking does not scale. Basically, it works in following steps:
  1. Original list
  2. Copy updated item
  3. Change the copy
  4. Change the pointer
  5. Free the original (Tricky part and the reason why we need RCU)

Implementation of the state, when the RCU determines when it can free old data is indirect because it is cheaper then usual methods like reference counting. To explain how it is determined, that all readers have finished their reader side critical section thus we can delete an old version is very simple. Once the updater finishes his task it calls synchronize_rcu(). It starts so called *grace period*, which can extend until all readers critical sections which started before rcu_synchronize() was called finishes. When all readers are finished, rcu_synchronize() returns and it is guaranteed that the old version can be deleted. One of the simplest implementations of rcu synchronize can look as follows:

```
void synchronize_rcu(void)
{
int cpu;
for_each_cpu(cpu)
run_on(cpu);
}
```

This enforces the context switch on every cpu, therefore once synchronize_rcu() is finished, all readers must completed and it is safe to remove the old version.

- QSBR - Quiescent-state based reclamation
  Read https://www.efficios.com/pub/rcu/urcu-supp.pdf
- General-purpose
  Read https://www.efficios.com/pub/rcu/urcu-supp.pdf

Sources:

- https://en.wikipedia.org/wiki/Mutual_exclusion
- https://www.threadingbuildingblocks.org/docs/help/tbb_userguide/Atomic_Operations.html
- https://en.wikipedia.org/wiki/Producer–consumer_problem
- https://en.wikipedia.org/wiki/Lock_(computer_science)#Granularity
- https://en.wikipedia.org/wiki/Resource_contention
- https://ionutbalosin.com/2018/06/contended-locks-explained-a-performance-approach/
- https://www.quora.com/How-does-Kernel-support-semaphores
- http://tuxthink.blogspot.com/2011/05/using-semaphores-in-linux.html
- https://en.wikipedia.org/wiki/Futex
- https://en.wikipedia.org/wiki/Read-copy-update
- https://cs.wikipedia.org/wiki/Přepnut%C3%AD_kontextu
- https://lwn.net/Articles/262464/
- http://www.rdrop.com/users/paulmck/RCU/whatisRCU.html

6 Lecture 6

- **Data races**
  The Thread Analyzer detects data-races that occur during the execution of a multi-threaded process. A data race occurs when:

  - two or more threads in a single process access the same memory location concurrently, and
  - at least one of the accesses is for writing, and
  - the threads are not using any exclusive locks to control their accesses to that memory.

  When these three conditions hold, the order of accesses is non-deterministic, and the computation may give different results from run to run depending on that order.

- **Out-of-order execution** In computer engineering, out-of-order execution, OoOE, is a technique used in most high-performance microprocessors to make use of cycles that would otherwise be wasted by a certain type of costly delay. Most modern CPU designs include support for out of order execution. The key concept of OoO processing is to allow the processor to avoid a class of delays (termed: "stalls ") that occur when the data needed to perform an operation are unavailable. OoO processors fill these "slots" in time with other instructions that are ready, then re-order the results at the end to make it appear that the instructions were processed as normal.

- **Volatile variable**
  The Java volatile keyword guarantees visibility of changes to variables across threads. In a multithreaded application where the threads operate on non-volatile variables, each thread may copy variables from main memory into a CPU cache while working on them, for performance reasons. If your computer contains more than one CPU, each thread may run on a different CPU. That means, that each thread may copy the variables into the CPU cache of different CPUs. The Java volatile keyword is intended to address variable visibility problems. By declaring the variable volatile all writes to the variable will be written back to main memory immediately. Also, all reads of the variable will be read directly from main memory. Volatile is so called memory barrier and guarantess atomic read and write. Can be used for primitives as well as objects. Is much slower than non-volatile access due to cache flush or cache invalidation. It is still faster than locks/synchronization. In assembly code it is represented by lock prefix instruction.

- **Instruction lock prefix**
  Forbids all reordering around and synchronize previous writes to be visible by all other CPUs. Instruction *lock addl $0x0,(%rsp)* is fastest memory barrier - no operation inside CPU.

- **Immutable object**
  An immutable object is an object that will not change its internal state after creation. Immutable objects are very useful in multithreaded applications because they can be shared between threads without synchronization.

- **Synchronized**
  A piece of logic marked with synchronized becomes a synchronized block, allowing only one thread to execute at any given time.

- **Reentrant lock**
  The traditional way to achieve thread synchronization in Java is by the use of synchronized keyword. While it provides a certain basic synchronization, the synchronized keyword is quite rigid in its use. For example, a thread can take a lock only once. Synchronized blocks don't offer any mechanism of a waiting queue and after the exit of one thread, any thread can take the lock. This could lead to starvation of resources for some other thread for a very long period of time.

  Reentrant Locks are provided in Java to provide synchronization with greater flexibility.

  The ReentrantLock class implements the Lock interface and provides synchronization to methods while accessing shared resources. The code which manipulates the shared resource is surrounded by calls to lock and unlock method. This gives a lock to the current working thread and blocks all other threads which are trying to take a lock on the shared resource.

  As the name says, ReentrantLock allow threads to enter into lock on a resource more than once. When the thread first enters into lock, a hold count is set to one. Before unlocking the thread can re-enter into lock again and every time hold count is incremented by one. For every unlock request, hold count is decremented by one and when hold count is 0, the resource is unlocked.

  Reentrant Locks also offer a fairness parameter, by which the lock would abide by the order of the lock request i.e. after a thread unlocks the resource, the lock would go to the thread which has been waiting for the longest time. This fairness mode is set up by passing true to the constructor of the lock.

- **Java locks**
  When threads in a process share and update the same data, their activities must be synchronized to avoid errors. In Java, this is done with the synchronized keyword, or with wait and notify. Synchronization is achieved by the use of locks, each of which is associated with an object by the JVM. For a thread to work on an object, it must have control over the lock associated with it, it must "hold" the lock. Only one thread can hold a lock at a time. If a thread tries to take a lock that is already held by another thread, then it must wait until the lock is released. When this happens, there is so called "contention" for the lock.

  There are four different kinds of locks:

  - Fat locks: A fat lock is a lock with a history of contention (several threads trying to take the lock simultaneously), or a lock that has been waited on (for notification).
  - Thin locks: A thin lock is a lock that does not have any contention.
  - Recursive locks: A recursive lock is a lock that has been taken by a thread several times without having been released.
  - Lazy locks: A lazy lock is a lock that is not released when a critical section is exited. Once a lazy lock is acquired by a thread, other threads that try to acquire the lock have to ensure that the lock is, or can be, released. Lazy locks are used by default in Oracle JRockit JVM 27.6. In older releases, lazy locks are only used if you have started the JVM with the -XXlazyUnlocking option.

  A thin lock can be inflated to a fat lock and a fat lock can be deflated to a thin lock. The JRockit JVM uses a complex set of heuristics to determine when to inflate a thin lock to a fat lock and when to deflate a fat lock to a thin lock.

- **Biased lock**
  Enables a technique for improving the performance of uncontended synchronization. An object is "biased" toward the thread which first acquires its monitor via a monitorenter bytecode or synchronized method invocation; subsequent monitor-related operations performed by that thread are relatively much faster on multiprocessor machines Some applications with significant amounts of uncontended synchronization may attain significant speedups with this flag enabled; some applications with certain patterns of locking may see slowdowns, though attempts have been made to minimize the negative impact.

- **CAS operations**
  Compare-and-swap (CAS) is an atomic instruction used in multithreading to achieve synchronization. It compares the contents of a memory location with a given value and, only if they are the same, modifies the contents of that memory location to a new given value. This is done as a single atomic operation. The atomicity guarantees that the new value is calculated based on up-to-date information; if the value had been updated by another thread in the meantime, the write would fail. The result of the operation must indicate whether it performed the substitution; this can be done either with a simple boolean response (this variant is often called compare-and-set), or by returning the value read from the memory location (not the value written to it).

- Atomic operations
- AtomicMarkableReference
- AtomicStampedReference
- Non-blocking algorithms
- CMPXCHG
- Thread-safe collections:
  - ConcurrentHashMap

Sources:

- https://docs.oracle.com/cd/E19205-01/820-0619/geojs/index.html
- https://stackoverflow.com/questions/11276259/are-data-races-and-race-condition-actually-the-same-thing-in-context-of-conc
- https://simple.wikipedia.org/wiki/Out-of-order_execution
- http://tutorials.jenkov.com/java-concurrency/volatile.html
- https://dzone.com/articles/immutable-objects-in-java
- https://www.baeldung.com/java-synchronized
- https://www.geeksforgeeks.org/reentrant-lock-java/
- https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/thread_basics.html
- https://stackoverflow.com/questions/9439602/biased-locking-in-java
- https://www.vogella.com/tutorials/JavaConcurrency/article.html
- https://en.wikipedia.org/wiki/Compare-and-swap

7 Lecture 7

- **OSI Model**
  The Open Systems Interconnection model (OSI model) is a conceptual model that characterizes and standardizes the communication functions of a telecommunication or computing system without regard to its underlying internal structure and technology. Its goal is the interoperability of diverse communication systems with standard communication protocols. The model partitions a communication system into abstraction layers. The original version of the model had seven layers.

  A layer serves the layer above it and is served by the layer below it. For example, a layer that provides error-free communications across a network provides the path needed by applications above it, while it calls the next lower layer to send and receive packets that constitute the contents of that path.

  OSI model consists of following layers:

  1. Physical Layer
  2. Data Link Layer
  3. Network Layer
  4. Transport Layer
  5. Session Layer
  6. Presentation Layer
  7. Application Layer

- **C10k, C1M, C10M problems**
  Problem of serving a large number of clients. Good approach is to use event-driven I/O servers like Nginx. Also to use non-blocking operations with event interceptors.

- Threading servers (Apache)

- Event-driven I/O servers (Nginx)

- Event interceptor

- **Process vs thread**
  A process is an executing instance of an application. What does that mean? Well, for example, when you double-click the Microsoft Word icon, you start a process that runs Word. A thread is a path of execution within a process. Also, a process can contain multiple threads. When you start Word, the operating system creates a process and begins executing the primary thread of that process.

  It's important to note that a thread can do anything a process can do. But since a process can consist of multiple threads, a thread could be considered a 'lightweight' process. Thus, the essential difference between a thread and a process is the work that each one is used to accomplish. Threads are used for small tasks, whereas processes are used for more 'heavyweight' tasks – basically the execution of applications.

  Another difference between a thread and a process is that threads within the same process share the same address space, whereas different processes do not. This allows threads to read from and write to the same data structures and variables, and also facilitates communication between threads. Communication between processes – also known as IPC, or inter-process communication – is quite difficult and resource-intensive.

- **JAVA thread pool**
  In Java, threads are mapped to system-level threads which are operating system's resources. If you create threads uncontrollably, you may run out of these resources quickly.

  The context switching between threads is done by the operating system as well – in order to emulate parallelism. A simplistic view is that – the more threads you spawn, the less time each thread spends doing actual work.

  The Thread Pool pattern helps to save resources in a multithreaded application, and also to contain the parallelism in certain predefined limits.

  When you use a thread pool, you write your concurrent code in the form of parallel tasks and submit them for execution to an instance of a thread pool. This instance controls several re-used threads for executing these tasks.

- **Non-blocking I/O approach**
  - polling
    In polling is not a hardware mechanism, its a protocol in which CPU steadily checks whether the device needs attention. Wherever device tells process unit that it desires hardware processing, in polling process unit keeps asking the I/O device whether or not it desires CPU processing. The CPU ceaselessly check every and each device hooked up thereto for sleuthing whether or not any device desires hardware attention.
  - signals
    A signal is an asynchronous notification of an event:
    * Asynchronous: could occur at any time
    * Interrupts receiving process; jumps to signal handler in that process
    * A (limited) menu of event types to pick from.
  - callbacks
    There are two types of callbacks, differing in how they control data flow at runtime: blocking callbacks (also known as synchronous callbacks or just callbacks) and deferred callbacks (also known as asynchronous callbacks). While blocking callbacks are invoked before a function returns (in the C example below, which illustrates a blocking callback, it is function main), deferred callbacks may be invoked after a function returns. Deferred callbacks are often used in the context of I/O operations or event handling, and are called by interrupts or by a different thread in case of multiple threads. Due to their nature, blocking callbacks can work without interrupts or multiple threads, meaning that blocking callbacks are not commonly used for synchronization or delegating work to another thread.
  - interrupts
    Interrupt is a hardware mechanism in which, the device notices the CPU that it requires its attention. Interrupt can take place at any time. So when CPU gets an interrupt signal trough the indication interrupt-request line, CPU stops the current process and respond to the interrupt by passing the control to interrupt handler which services device.
  - event-based
    * select
      select() and pselect() allow a program to monitor multiple file descriptors, waiting until one or more of the file descriptors become "ready" for some class of I/O operation (e.g., input possible). A file descriptor is considered ready if it is possible to perform a corresponding I/O operation.
    * poll
      Poll() performs a similar task to select(2): it waits for one of a set of file descriptors to become ready to perform I/O.
    * epoll
      Epoll stands for event poll and is a Linux specific construct. It allows for a process to monitor multiple file descriptors and get notifications when I/O is possible on them. By default, epoll provides level-triggered notifications. Every call to epoll_wait only returns the subset of file descriptors belonging to the interest list that are ready. So if we have four file descriptors (fd1, fd2, fd3 and fd4) registered, and only two (fd2 and fd3) are ready at the time of calling epoll_wait, then only information about these two descriptors are returned.

      Why epoll is more performant that select and poll
      As stated in the previous post, the cost of select/poll is O(N), which means when N is very large (think of a web server handling tens of thousands of mostly sleepy clients), every time select/poll is called, even if there might only be a small number of events that actually occurred, the kernel still needs to scan every descriptor in the list. Since epoll monitors the underlying file description, every time the open file description becomes ready for I/O, the kernel adds it to the ready list without waiting for a process to call epoll_wait to do this. When a process does call epoll_wait, then at that time the kernel doesn't have to do any additional work to respond to the call, but instead returns all the information about the ready list it's been maintaining all along. Furthermore, with every call to select/poll requires passing the kernel the information about the descriptors we want to monitored. This is obvious from the signature to both calls. The kernel returns the information about all the file descriptors passed in which the process again needs to examine (by scanning all the descriptors) to find out which ones are ready for I/O. With epoll, once we add the file descriptors to the epoll instance's interest list using the epoll_ctl call, then

when we call epoll_wait in the future, we don't need to subsequently pass the file descriptors whose readiness information we wish to find out. The kernel again only returns back information about those descriptors which are ready for I/O, as opposed to the select/poll model where the kernel returns information about every descriptor passed in. As a result, the cost of epoll is O(number of events that have occurred) and not O(number of descriptors being monitored) as was the case with select/poll.

- **TCP**
  - Socket
    Client end-point of network TCP/IP connection.
  - ServerSocket
    Special socket representing listening TCP/IP end-point.
- **UDP**
  - DatagramPacket
  - DatagramSocket
  - MulticastSocket
- **NIO**
  See following links about NIO https://medium.com/coderscorner/tale-of-client-server-and-socket-a6ef54a74763 and http://kasunpanorama.blogspot.com/2015/04/understanding-reactor-pattern-with-java.html
  - java.nio.buffer
  - Direct buffers
  - Selector
  - SelectorKey
  - Channel
  - FileChannel
  - SocketChannel
  - ServerSocketChannel
  - NIOServer
  - NIOHandler
  - NIOReactor
  - NIOAcceptorHandler
  - NIOClientHandler

Sources:

- https://en.wikipedia.org/wiki/OSI_model
- https://www.programmerinterview.com/operating-systems/thread-vs-process/
- https://www.baeldung.com/thread-pool-java-and-guava
- https://medium.com/@copyconstruct/the-method-to-epolls-madness-d9d2d6378642
- http://man7.org/linux/man-pages/man2/poll.2.html
- http://man7.org/linux/man-pages/man2/select.2.html
- https://medium.com/coderscorner/tale-of-client-server-and-socket-a6ef54a74763
- http://kasunpanorama.blogspot.com/2015/04/understanding-reactor-pattern-with-java.html
- https://www.geeksforgeeks.org/difference-between-interrupt-and-polling/
- https://courses.engr.illinois.edu/cs241/sp2012/lectures/31-select.pdf
- https://en.wikipedia.org/wiki/Callback_(computer_programming)

8 Lecture 8

- **Serialization**
  In computer science, in the context of data storage, serialization is the process of translating data structures or object state into a format that can be stored or transmitted and reconstructed later. When the resulting series of bits is reread according to the serialization format, it can be used to create a semantically identical clone of the original object.
- **RPC - Remote Procedure Calls**
  Remote procedure call (RPC) is when a computer program causes a procedure (subroutine) to execute in a different address space (commonly on another computer on a shared network), which is coded as if it were a normal (local) procedure call, without the programmer explicitly coding the details for the remote interaction. That is, the programmer writes essentially the same code whether the subroutine is local to the executing program, or remote.
- **RMI - Remote Method Invocation**
  Invoking a method on a remote object is known as remote method invocation (RMI) or remote invocation, and is the object-oriented programming analog of a remote procedure call (RPC).
- Less eficient data serialization
  - XML
  - JSON
- Faster alternative (C/C++)
  - Raw memory
- Frameworks
  - CORBA
  - Protobufs
    - Wire encoding
  - Cap'n'proto
  - Apache Avro
- IDL - Interface Description Language
- CDR - Common Data Representation
- Schema

Sources:

- https://en.wikipedia.org/wiki/Serialization
- https://en.wikipedia.org/wiki/Remote_procedure_call
- https://en.wikipedia.org/wiki/Distributed_object_communication
- https://en.wikipedia.org/wiki/Serialization

9 Lecture 9

- **Types of RAM**
  - SRAM
    Fast but expensive, consists of 6 transistors per bit. Usually used in caches.
  - DRAM
    One transistor and one capacitor. Cheaper but 10 times slower than SRAM. To read/write data, we need to count with *tclk* time/frequency and also *trcd* time and CL delays. Data are sent in bursts, which are the same size as a cache line to increase the speed and efficiency.

  - SDRAM
    Term for synchronous DRAM.
  - Timing parameters

    * CAS latency - CL delay
      It tells us how many clock cycles the memory will delay to return requested data. A memory with CL = 7 will delay seven clock cycles to deliver data, while a memory with CL = 9 will delay nine clock cycles to perform the same operation.
    * RAS to CAS Delay (tRCD)
      Each memory chip is organized internally as a matrix. At the intersection of each row and column we have a small capacitor that is in charge of storing a "0" or a "1" – the data. Inside the memory, the process of accessing the stored data is accomplished by first activating the row then the column where it is located. This activation is done by two control signals called RAS (Row Address Strobe) and CAS (Column Address Strobe). The less time there is between these two signals the better, as the data will be read sooner. RAS to CAS Delay or tRCD measures this time.

- **Caches**
  - Spacial locality
    Accessed memory objects are close to each other.
  - Temporal locality
    The same data will be used multiple times in a short period of time.
  - Cache hit
    Mempry request is serviced from the cache, without going to higher level memory.
  - Cache miss
    Opposite of cache hit. Data needs to be fetched from higher level memory.

    * Cold miss
      On the first access to a block; the block must be brought into the cache; also called cold start misses, or first reference misses.
    * Capacity miss
      Occur because blocks are being discarded from cache because cache cannot contain all blocks needed for program execution (program working set is much larger than cache capacity).
    * Conflict miss
      In the case of set associative or direct mapped block placement strategies, conflict misses occur when several blocks are mapped to the same set or block frame; also called collision misses or interference misses.
    * True sharing miss
      Bellow.
    * False sharing miss
      Bellow.
  - Cache line eviction
    Cache line is removed from the cache to make space for new data.
  - Cache replacement policy

    * LRU
    * Pseudo LRU
    * Random
  - Cache asociativity

    * Direct-mapped cache
      Each memory location has just one cache line associated with it. Memory locations at multiples of cache size always collide.
    * Fully associative cache
      In a Fully associative cache, the cache is organized into a single cache set with multiple cache lines. A memory block can occupy any of the cache lines. The cache organization can be framed as (1*m) row matrix.

    * Set associative cache
      Set associative cache is a trade-off between Direct mapped cache and Fully associative cache. The Set associative cache can be imagined as a (n*m) matrix. The cache is divided into 'n' sets and each set contains 'm' cache lines. A memory block is first mapped onto a set and then placed into any cache line of the set.

- Code self-eviction
  Poorly written program may cause self eviction of its data/instructions from the cache.
- Cache writing policies
  When a system writes data to cache, it must at some point write that data to the backing store as well. The timing of this write is controlled by what is known as the write policy. There are two basic writing approaches:

  Write-through: write is done synchronously both to the cache and to the backing store.
  Write-back (also called write-behind): initially, writing is done only to the cache. The write to the backing store is postponed until the modified content is about to be replaced by another cache block.

  A *write-back* cache is more complex to implement, since it needs to track which of its locations have been written over, and mark them as dirty for later writing to the backing store. The data in these locations are written back to the backing store only when they are evicted from the cache, an effect referred to as a lazy write. For this reason, a read miss in a write-back cache (which requires a block to be replaced by another) will often require two memory accesses to service: one to write the replaced data from the cache back to the store, and then one to retrieve the needed data.
- TLB - Translation Lookaside Buffer
  A translation lookaside buffer (TLB) is a memory cache that is used to reduce the time taken to access a user memory location. It is a part of the chip's memory-management unit (MMU). The TLB stores the recent translations of virtual memory to physical memory and can be called an address-translation cache. A TLB may reside between the CPU and the CPU cache, between CPU cache and the main memory or between the different levels of the multi-level cache. The majority of desktop, laptop, and server processors include one or more TLBs in the memory-management hardware, and it is nearly always present in any processor that utilizes paged or segmented virtual memory.
- Thread preemption
  When a thread is preempted by another thread, the preempting thread likely evicts some data from the cache. After preemption ends, the preempted thread continues executing and experiences a lot of cache misses.
- Cache coherency
  Maintaining of uniform view of memory for all processors. If some processor writes to a cache line, other processors have to clean the corresponding cache line from their caches.
- Dirty cache line
  The data in the cache is called dirty data , if it is modified within cache but not modified in main memory.
- Clean cache copy
  Unmodified/clean copy means that any line change must be forwarded to memory immediately.
- True sharing
  Data are shared.
- False sharing
  Data are not shared but are stored in one cache line.
- NUMA - Non-Uniform Memory Access
  See 1.

Sources:

- https://www.microcontrollertips.com/dram-vs-sram/
- https://www.hardwaresecrets.com/understanding-ram-timings/
- http://meseec.ce.rit.edu/eecc551-winter2001/551-1-30-2002.pdf
- https://en.wikipedia.org/wiki/Cache_(computing)#Writing_policies

- https://en.wikipedia.org/wiki/Cache_placement_policies
- https://en.wikipedia.org/wiki/Translation_lookaside_buffer
- https://www.quora.com/What-does-dirty-mean-in-the-context-of-caching

10 Lecture 10

- **Performance factors**
  - Total runtime
    - Algorithms - their complexity, used instructions, synchronization overhead
    - Memory management overhead - in JVM garbage collection
    - data structures - speed of data access, cache efficiency, GC pressure
  - Memory consumption
    - Data structures - memory usage efficiency
- **Memory analysis**
  - Static memory analysis
    Analyze memory usage at particular time. Suitable for data structure efficiency analysis, inspection of content. Advanced static analysis inspections:
    1. Wasting memory
       Memory doesn't keep any useful content.
       (a) Duplicate strings
       (b) Duplicate objects
       (c) Zero length arrays
       (d) Null fields
       (e) Sparse arrays
       (f) Inefficient data structure
    2. Memory leak
       Objects are no longer used but there are still references to them.
       (a) Object retained from inner non-static class back reference
    3. Performance
       Speed of data read/write.
       (a) Hash tables with non-uniformly distributed hash codes
  - Dynamic memory analysis
    Analyze dynamic changes over time. Suitable for object allocation analysis and memory leak identification.
    1. GC telemetry:
       (a) usage of eden space in time
       (b) GC collections and their duration
       (c) Does not affect performance of monitored application
    2. Heap dumps comparison:
       (a) Difference in object count and size in various application state
       (b) Dumps with all objects (not just those alive) can help analyze object allocations if there is no GC run in between.
       (c) Each heap dump requires global safepoint and time depends on the heap size.
    3. Memory profiler - allocation tracking
       (a) Track every n-th object allocation (trade-off between precision and speed)
       (b) Affect performance of profiled application, because of injection of bytecode traceObjAlloc. This decreases possibility of JIT optimizations and introduces a lot of byte code and consumes memory.
       (c) **IF flight recording** used, no byte code instrumentation happens and the allocation is tracked outside TLAB (Thread local allocation buffer).

- **Heap dump**
  A heap dump is a snapshot of the memory of a Java process. The snapshot contains information about the Java objects and classes in the heap at the moment the snapshot is triggered. Because there are different formats for persisting this data, there might be some differences in the information provided. Typically, a full garbage collection is triggered before the heap dump is written, so the dump contains information about the remaining objects in the heap.

- **Shallow size**
  Shallow size of an object is the amount of memory allocated to store the object itself, not taking into account the referenced objects

- **Retained size**
  Retained size of an object is its shallow size plus the shallow sizes of the objects that are accessible, directly or indirectly, only from this object. In other words, the retained size represents the amount of memory that will be freed by the garbage collector when this object is collected.

- **Primitives**
  The eight primitives defined in Java are int, byte, short, long, float, double, boolean, and char – those aren't considered objects and represent raw values. They're stored directly on the stack.

- **Objects**
  Java objects reside in an area called the heap. The heap is created when the JVM starts up and may increase or decrease in size while the application runs. When the heap becomes full, garbage is collected. During the garbage collection objects that are no longer used are cleared, thus making space for new objects. Every object is descendant of Object by default. There are also objects for primitives and these can be null. Objects with multiple fields use type group alingment and padding in memory.

  During object allocation, the JRockit JVM distinguishes between small and large objects. The limit for when an object is considered large depends on the JVM version, the heap size, the garbage collection strategy and the platform used, but is usually somewhere between 2 and 128 kB.

  Small objects are allocated in thread local areas (TLAs). The thread local areas are free chunks reserved from the heap and given to a Java thread for exclusive use. The thread can then allocate objects in its TLA without synchronizing with other threads. When the TLA becomes full, the thread simply requests a new TLA. The TLAs are reserved from the nursery if such exists, otherwise they are reserved anywhere in the heap.

- **Arrays**

  - Single-dimension
  - Multi-dimensional

- **Memory efficiency**
  Simply put, it is useful content size in the data structure like an array divided by retained size by the same structure. It correlates with cache efficiency as all data in cache line are read as well.

- **Auto boxing & unboxing**
  Autoboxing is a feature, which was added in Java 5. Autoboxing is the automatic conversion of primitive data types like int, double, long, boolean to its wrapper Object Integer, Double... and vice versa. Advantages of autoboxing are:

  1. Less code to write. The code looks cleaner.
  2. The best method for conversion is automatically chosen, e.g. Integer.valueOf(int) is used instead of new Integer(int)

  Disadvantages are:

  1. Can lead to unexpected behaviour
     The usage of autoboxing can lead to difficult to recognize errors. Especially if you mix wrapper with primitives in 'equals'/'=='.
  2. Hiding
     It hides the object creation, which can lead to a big performance loss.

3. Overloading

4. NullPointerException
   You can get a NullPointerException, if the wrapper object is null and is unboxed. Pointing out the obvious there can't be a NullPointer with primitive variables, but they can have the value zero.
5. **Immutable**
   All primitive wrapper objects in Java are final, which means they are immutable. When a wrapper object get its value modified, the compiler must create a new object and then reassign that object to the original. This creation and eventual garbage collection of objects will add a lot of overhead, especially when doing large computations in loops.

- **Java collections**
  - LinkedList
  - ArrayList
  - HashMap
- **Collections for performance**
  These use open addresing hashing in Maps instead of chaining approach.
  - Trove
  - FastUtil

Sources:

- https://www.ibm.com/support/knowledgecenter/en/SS3KLZ/com.ibm.java.diagnostics.memory.analyzer.doc/heapdur
- https://www.yourkit.com/docs/java/help/sizes.jsp
- https://www.baeldung.com/java-primitives
- https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html
- https://javaeffective.wordpress.com/2010/05/21/the-advantages-and-traps-of-autoboxing/
- https://medium.com/@bpnorlander/java-understanding-primitive-types-and-wrapper-objects-a6798fb2afe9

11 Lecture 11

- **Fast object allocation**

1. Bumb the pointer
   Allocations in free memory are efficient, using a simple bump-the-pointer technique. That is, the end of the previously allocated object is always kept track of. When a new allocation request needs to be satisfied, all that needs to be done is to check whether the object will fit in the remaining part of the generation and, if so, to update the pointer and initialize the object.
2. TLABs - Thread-local allocation buffers
   For multithreaded applications, allocation operations need to be multithread-safe. If global locks were used to ensure this, then allocation into a generation would become a bottleneck and degrade performance. Instead, the HotSpot JVM has adopted a technique called Thread-Local Allocation Buffers (TLABs). This improves multithreaded allocation throughput by giving each thread its own buffer (i.e., a small portion of the generation) from which to allocate. Since only one thread can be allocating into each TLAB, allocation can take place quickly by utilizing the bump-the-pointer technique, without requiring any locking. Only infrequently, when a thread fills up its TLAB and needs to get a new one, must synchronization be utilized. Several techniques to minimize space wastage due to the use of TLABs are employed. For example, TLABs are sized by the allocator to waste less than 1% of Eden, on average. The combination of the use of TLABs and linear allocations using the bump-the-pointer technique enables each allocation to be efficient, only requiring around 10 native instructions.

- **Escape analysis**
  Anytime a value is shared outside the scope of a function's stack frame, it will be placed (or allocated) on the heap. It's the job of the escape analysis algorithms to find these situations and maintain a level of integrity in the program. The integrity is in making sure that access to any value is always accurate, consistent and efficient. C2 compiler performs escape analysis of new object after inline of hot methods. Each new object does is classified into one of the following types:
  - NoEscape
    Object does not escape the method in which it is created. All it's usages are inlined. Never assigned to static or object field, just to local variables. At any point must be JIT-time determinable and not depending on any unpredictable control flow. NoEscape objects are not allocated at all but JIT does scalar replacement. Object is deconstructed into it's constituent fields and stack allocated. It dissappears automatically after stack frame pop. No GC impact at all and it does not need track references.
  - ArgEscape
    Object is passed as. or referenced from, an argument to a method but does not escape the current thread. ArgEscape objects are allocated on the heap but all monitors are eliminated.
  - GlobalEscape
    Object is accessed by different method and thread.
- **Bloom filter**
  A Bloom filter is a space-efficient probabilistic data structure, that is used to test whether an element is a member of a set. False positive matches are possible, but false negatives are not – in other words, a query returns either "possibly in set" or "definitely not in set". Elements can be added to the set, but not removed (though this can be addressed with a "counting" filter); the more elements that are added to the set, the larger the probability of false positives. It's advantage is strong memory reduction and always constant add and test query. It is used to test that the object is certainly not member of the set.
- **Refrence types**
  - Strong reference
    This is the default type/class of Reference Object. Any object which has an active strong reference are not eligible for garbage collection. The object is garbage collected only when the variable which was strongly referenced points to null.
  - Soft reference
    In Soft reference, even if the object is free for garbage collection then also its not garbage collected, until JVM is in need of memory badly. The objects gets cleared from the memory when JVM runs out of memory.
  - Weak reference
    A weak reference, simply put, is a reference that isn't strong enough to force an object to remain in memory. Weak references allow you to leverage the garbage collector's ability to determine reachability for you, so you don't have to do it yourself. If JVM detects an object with only weak references (i.e. no strong or soft references linked to any object object), this object will be marked for garbage collection.
  - Final reference
    TODO
  - Phantom reference
    The objects which are being referenced by phantom references are eligible for garbage collection. But, before removing them from the memory, JVM puts them in a queue called 'reference queue' . They are put in a reference queue after calling finalize() method on them.
- **Object reachibility**
  - Strongly
  - Softly
  - Weakly
  - Eligible for finalization

  - Phantom reachable
  - Unreachable

- **Performance cost**

  - Creation cost
  - Garbage-collection cost
  - Enqueue cost
  - Reference queue processing cost

Sources:

- http://www.voidcn.com/article/p-mhqnhzmj-ka.html
- https://www.ardanlabs.com/blog/2017/05/language-mechanics-on-escape-analysis.html
- https://en.wikipedia.org/wiki/Bloom_filter
- http://www.kdgregory.com/index.php?page=java.refobj
- https://www.geeksforgeeks.org/types-references-java/
- https://stackoverflow.com/questions/299659/whats-the-difference-between-softreference-and-weakreference-in-java
- https://docs.oracle.com/javase/7/docs/api/java/lang/ref/package-summary.html

12 Lecture 12

- **Automatic memory management** Advantages over explicit memory management are that there are no memory crashes due to errors and no memory leaks as well. Desired characteristics of such a system are:

  - Safety
  - Throughput
  - Completeness and promtness
  - Pause time
  - Space overhead
  - Scalability and portability

- **Identification of reachable objects**

  - Reference counting
    Requires additional counter for each object for the number of the references pointing to the object. That requires a lot of atomic operations to make it thread-safe and it slows down the application code. This technique does not support cyclic references. Cache is polluted by a lot of additional memory operations. Objects are removed once the counter reaches zero.
  - Reference tracking
    TODO

- **GC collectors**
  From the name, it looks like Garbage Collection deals with finding and deleting the garbage from memory. However, in reality, Garbage Collection tracks each and every object available in the JVM heap space and removes unused ones. In simple words, GC works in two simple steps known as Mark and Sweep:

  1. Mark – it is where the garbage collector identifies which pieces of memory are in use and which are not
  2. Sweep – this step removes objects identified during the “mark” phase

  JVM has four types of GC implementations:

1. Serial Garbage Collector
   This is the simplest GC implementation, as it basically works with a single thread. As a result, this GC implementation freezes all application threads when it runs. Hence, it is not a good idea to use it in multi-threaded applications like server environments.
2. Paralel Collector

   It's the default GC of the JVM and sometimes called Throughput Collectors. Unlike Serial Garbage Collector, this uses multiple threads for managing heap space. But it also freezes other application threads while performing GC.
3. CMS Garbage Collector
   The Concurrent Mark Sweep (CMS) implementation uses multiple garbage collector threads for garbage collection. It's designed for applications that prefer shorter garbage collection pauses, and that can afford to share processor resources with the garbage collector while the application is running. Simply put, applications using this type of GC respond slower on average but do not stop responding to perform garbage collection.
4. G1 Collector

   G1 (Garbage First) Garbage Collector is designed for applications running on multi-processor machines with large memory space. G1 collector will replace the CMS collector since it's more performance efficient.

   Unlike other collectors, G1 collector partitions the heap into a set of equal-sized heap regions, each a contiguous range of virtual memory. When performing garbage collections, G1 shows a concurrent global marking phase (i.e. phase 1 known as Marking) to determine the liveness of objects throughout the heap.

   After the mark phase is completed, G1 knows which regions are mostly empty. It collects in these areas first, which usually yields a significant amount of free space (i.e. phase 2 known as Sweeping). It is why this method of garbage collection is called Garbage-First.

Parallel vs G1 collector:

- Parallel still keeps memory layout separated in generations and makes all threads to stop during minor and major GC.
- G1GC has different memory layout and is concurrent. The memory is seprated into the regions. Set of regions composes the eden, survivor and old generation space.

Sources:

- https://medium.com/platform-engineer/understanding-java-garbage-collection-54fc9230659a
- https://www.baeldung.com/jvm-garbage-collectors
- https://www.youtube.com/watch?v=X8w3uqN-X98
- https://www.youtube.com/watch?v=YhKZe22tZlc
- https://blog.idrsolutions.com/2017/05/g1gc-java-9-garbage-collector-explained-5-minutes/
- https://www.oracle.com/technetwork/articles/java/g1gc-1984535.html
- https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/parallel.html

13 Lecture 13

- TODO

Sources:

- TODO

14 PERF optimizations

- **___attribute___((hot))**
  From slide 15, lecture 9, about self-eviction of code and cache misses.

  The hot attribute on a function is used to inform the compiler that the function is a hot spot of the compiled program. The function is optimized more aggressively and on many target it is placed into special subsection of the text section so all hot functions appears close together improving locality. When profile feedback is available, via -fprofile-use, hot functions are automatically detected and this attribute is ignored.

- ___attribute___((align(64)))
  From slide 39, lecture 9, about false-sharing.

  Data accessed from different CPUs is not shared but happen to be stored in a single cache line. Attribute allign(64) alligns the variable to 64 bits, the same size as the cache line (we have to determine cache line size first) therefore false sharing won't occur.

- Set CPU for thread to prevent migration and invalidating NUMA padding. From slide 42, lecture 9 about thread migration from one CPU to another.

  cpu_set_t cpuset; pthread_t thread; thread = pthread_self(); CPU_SET(1 « 3, &cpuset); s = pthread_setaffinity_np(
  sizeof(cpu_set_t), &cpuset);

15 Exam 2019

Not complete list of some questions from the test. There was also one question related to the memory types and true/false sharing, but I don't remember in which part.

15.1 C part

1. Analyze C code sample: Answer was ___align 64 to prevent false sharing in the caches.
2. Virtualization, methods, suitable hardware for virtualization
3. What are performance counters, description, usage
4. RCU, it's exact description, how it works, how we know we can free the memory, describe several methods.

15.2 Java part

1. Compare parallel and G1GC garbage collectors.
2. What is boxing/unboxing
3. Types of the referrences and their description
4. Data races

15.3 Oral part

1. **JAVA** Boxing and unboxing
2. **JAVA** Retained and shallow size
3. **JAVA** Refferences types
4. **JAVA** Generational hypothesis
5. **JAVA** Allocation of new memory
6. **JAVA** Static and dynamic analysis
7. **C** How is the C code compiled and how it looks like after each step
8. **C** Bentleys rules
9. **C** Epoll
10. **C** Mutex, RW lock, RCU

16 Self-test questions

16.1 Lecture 1

Lecture from C.

1. How C compiler works
2. How Java interpretation works
3. Instruction parallelism:
   (a) Pipelining
   (b) Branch prediction
   (c) Superscalar CPU
4. Task parallelism:
   (a) Multi-core CPUs, Multi-socket CPUs
   (b) NUMA
   (c) Out-of-order execution
   (d) Heterogenneous CPUs
   (e) Hyper-threading
5. Computer memories
6. Performance counters
7. Event sampling
8. Static instrumentation
9. Dynamic instrumentation

16.2 Lecture 2

Lecture from JAVA.

1. Java Virtual Machine
2. Bytecode
3. JIT compilers
   (a) C1
   (b) C2
   (c) Tiered compilation
4. JAVA memory layout
5. Dissassembler
6. Decompiler
7. Obfuscation
8. Hot spots
9. Ordinary Object Pointers
10. Safepoint
11. Warm-up time
12. Profiling
13. On-Stack replacement

   (a) CPU
   (b) Memory
14. Sampling
15. Tracing
16. Microbenchmark
17. Macrobenchmark

16.3 Lecture 3

1. Bechmarking
2. Timestamping
   (a) Use of syscalls
   (b) Use of HW directly
   (c) Virtual syscalls

16.4 Lecture 4

Lecture from C.

1. restrict qualifier
2. volatile qualifier
3. C/C++ compiler
   (a) Frontend
   (b) Optimization passes
      - High pass
      - Low pass
      - Profile-guided
   (c) Backend
   (d) Linker
4. Abstrax syntax tree
5. Intermediate representation
6. Assembly code
7. Bentley's rules
   (a) Data
   (b) Code

16.5 Lecture 5

Lecture from C.

1. Synchronization
2. Critical section
3. Atomic variable + ordering
4. Barrier macro
5. Deadlock
6. Spinlock

7. Locking overhead
    (a) Uncontended case
    (b) Contended case
8. Mutex
9. RW lock
10. RCU https://www.efficios.com/pub/rcu/urcu-supp.pdf

16.6 Lecture 6

Lecture from Java.

1. Data races
2. Volatile qualifier
3. lock addl $0x0,(%rsp) instruction
4. Synchronize qualifier
5. Thin lock
6. Fat lock
7. Biased lock
8. Reentrant lock
9. Non-blocking approach in Java
10. Non-blocking collections

16.7 Lecture 7

Lecture from Java.

1. OSI model
2. Process vs Thread
3. Concept of thread pooling
4. Non-blocking I/O approach
    (a) polling
    (b) signals
    (c) callbacks
    (d) interrupts
    (e) event-based
        i. select
        ii. poll
        iii. epoll
5. Socket
6. ServerSocket
7. NIO

16.8 Lecture 8

Lecture from C and Java.

1. Serialization
2. XML
3. JSON
4. Remote Method Invokation
5. Remote Procedure Call
6. Raw Memory
7. Schema/Interface Description language
8. Common Data Representation
9. Protobuf
10. Avro
11. CapnProto

16.9 Lecture 9

Lecture from C.

1. SRAM
2. DRAM
3. Cache locality
4. Temporal locality
5. Cache hit
6. Cache miss
   (a) Cold miss
   (b) Capacity miss
   (c) Conflict miss
   (d) True sharing miss
   (e) False sharing miss
7. Cache line eviction
8. Cache replacement policy
   (a) LRU
   (b) PSEUDO-LRU
   (c) Random
9. Cache associativity
   (a) Direct mapped
   (b) Fully associative
   (c) Set associative
10. Cache write policies
    (a) Write-through
    (b) Write-back

   (c) Write-combining
11. TLB - Translation Lookaside Buffer
12. Cache friendly data structures
13. Malloc, new
14. Cache Coherency
15. Hot cache line
16. Dirty cache line
17. True sharing
18. False sharing
19. Contention
20. NUMA
21. Threads/Load balancing

16.10 Lecture 10

Lecture from Java.

1. Performance factors
2. Memory analysis
   (a) Static memory analysis
      i. wasting memory
      ii. memory leak
      iii. performance
   (b) Dynamic memory analysis
      i. GC telemetry
      ii. Comparison of heap dumps
      iii. Allocation tracking
3. Heap dump
4. Shallow vs Retained size
5. Primitives
6. Objects
7. Primitive objects
8. Arrays and collections
9. Memory efficiency
10. Auto boxing/unboxing
11. Collections for performance
    (a) Trove
    (b) FastUtil

16.11 Lecture 11

Lecture on Java.

1. Fast object allocation
   (a) Bump the pointer
   (b) TLAB
2. Escape analysis
3. Bloom filter
4. Counting bloom filter
5. References
   (a) Strong
   (b) Soft
   (c) Weak
   (d) Final
   (e) Phantom

16.12 Lecture 12

Lecture on Java.

1. Automatic memory management
   (a) Advantages
   (b) Components
   (c) Desired characteristics
   (d) Generational hypothesis
   (e) Segregation of the objects
   (f) Promotion by minor collection
2. Identification of reachable objects
   (a) Reference counting
   (b) Reference tracing approach
      - Marking phase
3. Garbage collectors
   (a) Parallel collectors
      i. Minor collector
      ii. Full collector
   (b) G1
      i. Minor
      ii. Mixed
      iii. Full

16.13 Lecture 13

Lecture from Virtualization.

1. What is virtualization
2. Main components
   (a) Host
   (b) Hypervisor or VMM
      i. Difference between Hypervisor and VMM
   (c) Guest
3. Types of hypervisors
   (a) Native
   (b) Hosted
4. Advantages of virtualization
5. Cloud Computing
6. Types of virtualization
   (a) Virtualization of whole system
      i. Virtualization of CPU
         A. trap-and-emulate
         B. Virtualization of MMU
         C. Virtualization of page translation table
      ii. Hardware assisted virtualization
         A. non-root execution
         B. two layer paging
      iii. Virtualization of I/O
         A. trap-and-emulate
   (b) Paravirtualization
   (c) Emulation of the whole system
   (d) Virtualization of the runtime system
   (e) Containarization
      i. Namespaces
      ii. Cggroups
7. Trap and emulate
   (a) Sensitive instructions
   (b) Priviledged instructions