Aalto University
School of Science

Concurrency Models

CS-E4110 Concurrent Programming
Autumn 2021 (II), week 6

Vesa Hirvisalo

Week 6 in a nutshell

- Models for machines and concurrency
  - There is a diversity of models
- Scala (JVM) Memory Model
  - Data race free programs
  - Weak behavior (happens-before)
  - Cyclic things (out-of-thin-air)
- Hardware models
  - x86-TSO, POWER and ARM
- Weak Memory Model in C/C++
  - Low-level atomics for high performance
- Conclusion and things about the exam

Models for machines and concurrency

Machine models

- The classical machine models are developed mostly for understanding *performance*
  - The RAM model for complexity analysis (the Big O notation)
    - This is very simple, basically assume a *single processor* that is able to *read* and *write* to an *uniform memory*
  - The PRAM model (Parallel Random Access Machine)
    - Similarly, to reason about and classify parallel algorithms
- For understanding the *correctness* of shared-memory synchronization, the key issue is the memory
  - Assumptions on the memory architecture
    - Real machines have often caches affecting the R/W semantics
  - But also, assumptions on the *consecutive* reads and writes
    - In practice, speculative hardware or compilers may affect this

Parallel Random Access Machine

- Memory is infinite, number of processors in unbounded
  - But, you can consider subsets (e.g., 4 processors running)
- No direct communication between processors
  - they communicate via the memory
  - they can operate asynchronously
- Processors access any memory location in 1 cycle while executing algorithms in phases
  - READ phase
  - COMPUTE phase
  - WRITE phase
- Some subset of the processors can stay idle
  - But, usually it assumed that there is **no resource contention**

PRAM memory accesses

- Assumptions on the reads (R) and the writes (W)
  - E and C stand for 'exclusive' and 'concurrent' respectively
- Read/write conflicts (i.e., interlocking) in accessing the same memory location simultaneously are resolved by one of the following strategies
  - EREW: location can be read or written to by only one processor
    - The realistic one (but limits the *performance*)
  - CREW: multiple can read a location but only one can write
    - The commonly used with PRAM (i.e., theoretical analysis)
  - ERCW: never considered (does not make sense)
  - CRCW: multiple processors can read and write
    - A *concurrent random access machine* (hard)

Some comments

- The theoretical machines machines are mostly designed for theoretical studies
  - E.g., the famous Turing machine
  - But with rather limited practical use
- For parallel computing, some nice basic results exist
  - E.g., the very elegant Amdahl's law
  - But for real hardware many of the results are unrealistic: true HW has caches, etc., true SW uses synchronization, etc.
- However, understanding the problems in *parallel computation* is useful *concurrent programming*
  - For correctness, we have to consider *concurrent accesses*
  - But, have also to consider *consecutive accesses*

Programs and execution

- To reason about consecutive access, note the distinction between *program order* and *execution order*
  - A program defines an order for actions, but in execution that order holds only for a thread
    - Note that even with a single processor, a finite program can have infinitely many different executions (traces)
  - With multiple processors, the *global* execution order is undefined by default (the interleaving of the threads)
    - Even if the global order is undefined, locally a processing element can follow the program order
- For concurrency, there exists a number of different models at different levels
  - The **models do not agree**, they do not form a clear hierarchy
  - The apparent program order may not hold in practice (compilers)

Concurrency models

- There are a number of different concurrency models, some examples are listed below
  - *Strict consistency*: a write to a variable by any processor is seen instantaneously by all processors (note: no parallelism!)
  - *Sequential consistency* (SC): writes to variables by different processors are seen in the same order by all processors
  - *Processor consistency*: like SC but not when different processors or different memory locations are considered (or, adding some rule of sequential execution to PRAM)
  - *Weak consistency*: less strict than the above (there exists several definitions, remember the variance of caches)
- It is important to check the model you use
  - In practice, co-using several models is rather common!

Scala (JVM) Memory Model

Introduction

- Scala Memory model
  - Uses the Java Virtual Machine (JVM) Memory Model
  - Gives programmer guarantees on the behavior of a program
  - Guides compiler writing (sound program transformations)
- The JVM Memory Model (JMM) guarantees that
  - Data race free programs have exactly the same behavior as if the JVM program would have been run under the sequentially consistent memory model
    - How are data races defined?
      - Note the distinction between a race condition and a data race
    - How difficult is it to decide whether a program has a data race?
      - Note the distinction between a program and its execution

The JMM rules for happens-before

1. “Program order rule. Each action in a thread happens before every action in that thread that comes later in the program order.
2. Monitor lock rule. An unlock on a monitor lock happens before every subsequent lock on that same monitor lock.
3. Volatile variable rule. A write to a volatile field happens before every subsequent read of that same field.
4. Thread start rule. A call to Thread.`start` on a thread happens before every action in the started thread.
5. Thread termination rule. Any action in a thread happens before any other thread detects that thread has terminated, either by successfully return from Thread.`join` or by Thread.`isAlive` returning `false`.
6. Interruption rule. A thread calling interrupt on another thread happens before the interrupted thread detects the interrupt (either by having `InterruptedException` thrown, or invoking `isInterrupted` or `interrupted`).
7. Finalizer rule. The end of a constructor for an object happens before the start of the finalizer for that object.
8. Transitivity. If A happens before B, and B happens before C, then A happens before C.“

Goetz&al.: Java Concurrency in Practice. (You can check it for a detailed view)

Programs without data races

- Data race freedom
  - This has a big practical value
  - In general, a program is *correctly synchronized* if and only if all sequentially consistent executions are free of data races
    - Also called a *data race free* (DRF) program
    - Correct compilation of such programs into *efficient* machine code (for *efficient* machines) is much easier than for non-DRF programs
- Data race definition for JMM
  - “A data race occurs when a variable is read by more than one thread, and written by at least one thread, but the reads and writes are not ordered by happens before” (Goetz&al)

Weak behaviors

- Correctly synchronized programs exhibit only sequentially consistent executions
  - Non-DRF programs exhibit *weak memory model* semantics defined by the happens-before relation
- JVM specification allows compiler/JVM reordering
  - `http://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html`
  - “The presence of a happens-before relationship between two actions does not necessarily imply that they have to take place in that order in an implementation. If the reordering produces results consistent with a legal execution, it is not illegal.”

Programs with data races

- Semantics of programs with data races
  - A read *r* of a variable *v* is allowed to observe a write *w* to *v* if in the happens-before (hb) partial order of the execution trace
    - *r* is not ordered before *w* (*i.e.*, it is **not** the case that *hb(r,w)*), and
    - there is no intervening write *w'* to *v* (*i.e.*, **no** write *w'* to *v* exists such that *hb(w,w')* and *hb(w',r)*)
  - Note that weak behaviors are a practical approach for getting performance, energy-efficiency, etc.
    - This is *a very real thing for programming*
    - Not only for the folks interested in theoretical things
    - But actually, **not** for ordinary programming **with Scala** or Java (use some other language, e.g., C++)
    - By default, write only DRF programs!

Out-of-thin-air behaviors

- The difference of static (program) and dynamic (execution) cyclic dependencies
- The happens-before relation will not order any of the reads or writes below

Initially x = y = 0

Thread 0	Thread 1

r1 = x;
if (r1 == 42) {
y = 42 };
	r2 = y;
	if (r2 == 42) {
	x = 42 };

Out-of-thin-air in JMM

- No out of thin air executions are allowed (in addition to the restrictions by the happens-before rule)
- The out of thin air requirement is needed for JVM sandbox security to work!
- Note
  - Implementations may not follow the specifications
  - C/C++11 memory model allows these out of thin air executions
    - This is (of course) rather easy
  - C/C++14 does not because of the added sentence: "Implementations should ensure that no out-of-thin-air values are computed that circularly depend on their own computation."
    - This is hard (the implementations have difficulties in following this)

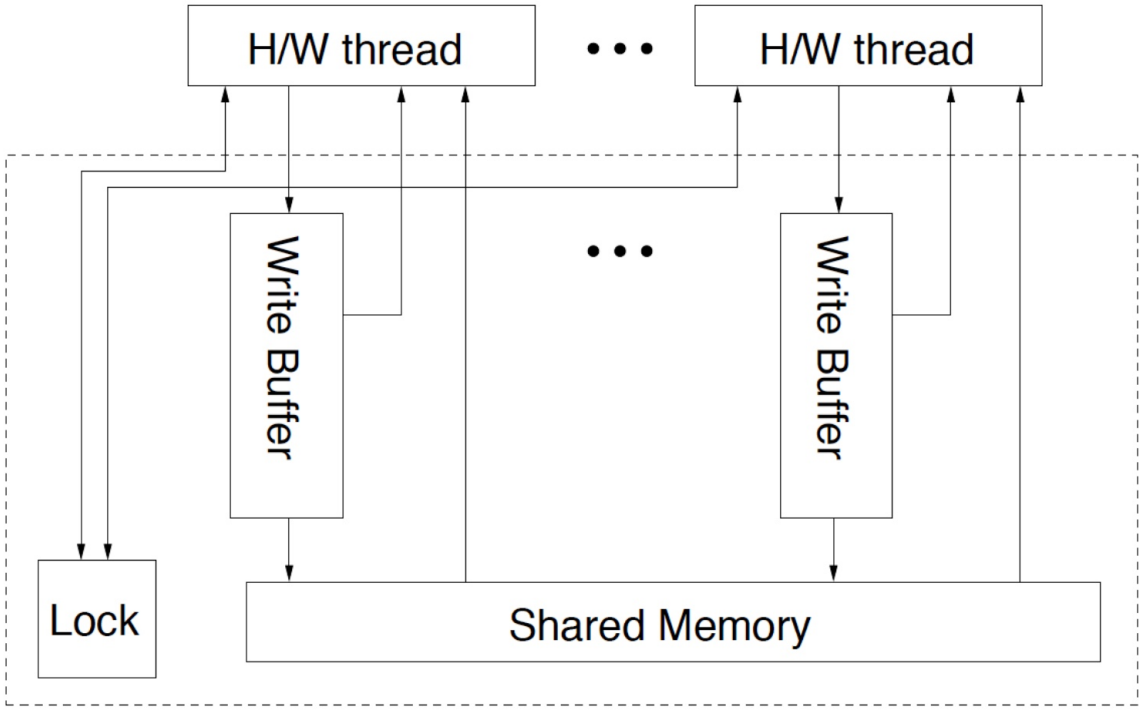
Hardware models

x86-TSO Memory model

- Goals of the x86-TSO definition
  - Always allow more behaviors than any hardware implementation (e.g., FIFOs of unbounded size, while any real HW will have bounded FIFOs)
  - Formalize the Intel and AMD technical specifications written in natural language (finding specification bugs in the process)
  - Act as a model to prove how to compile e.g., Scala into x86 assembly in a provably correct fashion
  - Act as a model to prove which compiler optimizations are correct for concurrent programs that contain data races
  - Enable to show that data race free programs under x86-TSO are sequentially consistent

x86-TSO memory subsystem model

- Has write buffers for HW threads
  - Note both the buffers and threads are abstract



x86-TSO Abstract Machine Semantics

- Write buffers
  - Each hardware thread has its own write buffer, which is a FIFO of unbounded size storing writes made by the hardware thread together with their store addresses
    - Read operations must read their data from the latest write to a memory location in the write buffer, if there is one matching the address of the read; otherwise reads are satisfied from the shared memory
    - When a write operation is performed, the write is appended to the store buffer
  - Each one of the writes stored in the store buffer will be eventually flushed to the shared memory in the background
    - There are fence instructions (MFENCE, …) to flush the buffer of a hardware thread before continuing with the next instruction

x86-TSO Abstract Machine LOCK

- The LOCK prefix
  - The x86 supports a range of atomic instructions: one can add a LOCK prefix to many read-modify-write instructions (ADD, INC, etc.), and the XCHG instruction is implicitly LOCK'd.
  - For example
    - The INC increment command consist of a read operation followed by a write operation. Thus it is not atomic.
    - The LOCK prefix can be added to an instruction to make it atomic.
- Implementing LOCK prefix
  - To implement the LOCK prefix, a global lock needs to be grabbed, and this is to be reflected in the semantics, see next slide

LOCK implementation

- Modifications to implement the LOCK prefix
  - Read operations are only allowed when the global lock is not held by another hardware thread
  - Write operations appending to the store buffer FIFO are always allowed
  - Flushing writes from the store buffer to the shared memory is only allowed if the global lock is not held by another hardware thread
  - If the global lock is free, a hardware thread can acquire the global lock and start an instruction with the LOCK prefix
  - When the instruction with a LOCK prefix finishes execution, it frees the global lock

Power and ARM Memory Model

- Allow among other things
  - Reads to different locations can be scheduled out-of-order
  - Writes to different locations can be scheduled out-of-order
  - Read operations can be performed speculatively, i.e., even before branches before them are resolved to be taken or not
  - Writes can come visible to different hardware threads at different times
- The formal model is quite complex
- In addition, real hardware can have bugs
  - Especially for ARM, low power consumption has been one of the (original) main design goals (i.e., not clear concurrency models)

Weak Memory Model in C/C++

C11/C++11 Memory model

- The C and C++ programming languages were standardized in 2011
  - ISO/IEC 14882:2011 - Information technology - Programming languages - C++ The C++ language standard is over 1300 pages long
  - The idea is the same as for Scala: Programs without race conditions have sequentially consistent behaviors
  - Also provides a semantics for programs with data races, and expert concurrency features including explicit weak memory model features called “low level atomics”

C/C++ Low level atomics

- The standard includes various atomic datatypes with sequential consistency helping write race free programs (please use these with default memory order) http://en.cppreference.com/w/cpp/atomic
- For memory order, see: http://en.cppreference.com/w/cpp/atomic/memory_order
- Problem: Sequentially consistent atomics are sometimes not performant enough for performance critical code
- C/C++ Low level atomics provide a way to write programs with weaker memory semantics than sequential consistency in a well defined way (only for programmers who really know what they are doing!)

Semantics

- For performance critical code (with races) the load and store operations can be given a memory order:
  - memory_order_relaxed, memory_order_consume, memory_order_acquire, memory_order_release, memory_order_acq_rel, memory_order_seq_cst
- Correct formalization of the C++ low-level atomics has turned out to be very hard
  - The C++ compilers use often some ad-hoc semantics when they do optimizations for different pieces of HW
  - However, many C/C++ compiler optimizations are safe once data race freedom is assumed

Conclusions

(This the last weekly slideset)

The exam

- The exam will concentrate on the theory
  - The A+ exercise try to underline the practical issues, they have been mostly about programming
  - The exam is not a programming test, it is mostly about explaining the concepts, etc. that have been described and discussed during the course
- Use the slides when preparing to the exam
  - Especially, if you have not attended the lectures, use the material that is pointed on the lecture slides
  - Note that there is short guide in Week3-video on how to read the textbook (but much depends on the grade you are targeting)
- In addition to generic concurrency issues, there are specific issues
  - Especially Scala/JVM, but also the other things (there are plenty of details described on the lecture slides)

Acknowledgements

- The set of course lecture slides is based on contributions by numerous parties
  - Including the previous teachers (which are many) and the source material they have used
  - The reader may refer to the previous implementations of this course and its predecessors (which are many)
- Thanks to all of you!