Aalto University
School of Science

Practical Tools

CS-E4110 Concurrent Programming
Autumn 2020 (II), week 2

Vesa Hirvisalo

2020-11-02

V. Hirvisalo
CS/Aalto

Week 2 in a nutshell

- System Considerations
  - Shared Memory Symmetric Multiprocessors
  - Registers (compiler) and caches (hardware)
- Correctness
  - Atomicity and program transformations
  - Safeness, liveness and weak fairness
- Languages and Concurrency (warnings!)
  - Stacks, nesting, mixed use
  - Spinlocks
- Basic Scala Concurrency
  - Language features
  - Happens-before relationship

System Considerations

On Applicability of our Computational Model

- Any possible ordering of the actions is OK, provided that it preserves their atomicity, and their relative order in each process.
- Time is abstracted away
- We can often ignore process-local actions, which simplifies creating and studying the state diagrams
- The order of actions compared to shared variables and resources is important

Shared Memory Symmetric Multiprocessors

- Often various types Shared Memory Symmetric Multiprocessors (SMP)* are used
  - The typical multicores, Chip Multiprocessors (CMP)
- More than one processor can access the same memory word physically at the same time
  - If two processors try to write on the same common memory word at the same time, the HW arbiter of the memory system forces the writes to take place in some order so that above situation may not rise, but the system ends up with either of the original values.
  - The atomicity of LOAD and STORE of single words into the global memory is preserved
  - May result in a race condition

(*) Note that the acronym SMP has other relevant meanings, e.g., Symmetric Multiprocessing (=~ all the processors do their memory/resource sharing under the same management SW).

Memory

- There are typically several levels of memory
  - Processor registers
    - Typically per processing element (PE) / core
    - The compiler typical decides on the use of these
  - Caches (can be several levels)
    - Some may be shared, but L1 caches are typically per PE
    - These are typically hardware managed
  - The main memory
    - This is the default memory for computation
    - The other layers are optimizing or extending the main memory
  - The secondary memory
    - Typically managed by the operating system

SMP Cache Consistency

- Cache memory is used to store automatically the most recent data items accessed by the processor close to it in order to get speed-up
- The *Sequential Consistency** concept
  - Can we guarantee that updating a data item in memory does not leave its copy stored in the cache of another processor outdated
  - A *Cache Coherence* protocol provided by HW will keep caches consistent in spite of concurrent LOADs and STOREs
  - However, a compiler may decide to keep a value of a variable in CPU registers for local code optimizations, so cache consistency is not always enough
- Note that sequential consistency may not hold

(*) More details will be studied during week 6.

Correctness

Atomicity of Statements

- Thus far we have neglected to examine atomicity in detail
  - Our LCL source statements are compiled into machine instructions
  - The target may be a register machine, a stack machine, …
- Warning: what may seem atomic, is sometimes not!
  - Ideally our source pseudo-code should behave enough like the compiled machine code, so that our reasoning based on the pseudo-code will apply to the compiled/interpreted program
- A key issue is the atomicity of assignment

The Assignment Statement (1/2)

- With true atomicity there are no problems…

Trivial program assuming atomic assignment	
integer n ← 0	
P	Q
p1: n ← n + 1	q1: n ← n + 1

Process p	Process q	n
p1: n ← n+1	q1: n ← n+1	0
(end)	**q1: n ← n+1**	1
(end)	(end)	2

Process p	Process q	n
p1: n ← n+1	**q1: n ← n+1**	0
p1: n ← n+1	(end)	1
(end)	(end)	2

… but: remember the underlying machinery!

The Assignment Statement (2/2)

- E.g., the compiler is there

Assignment statement on a register machine	
integer n ← 0	
P	Q
p1: `LOAD R1 n`	q1: `LOAD R1 n`
p2: `ADD R1 1`	q2: `ADD R1 1`
p3: `STORE R1 n`	q3: `STORE R1 n`

- Consider the trace: p1, p2, q1, q2, p3, q3

Sequential Correctness

- For sequential programs
  - A sequential program is correct with regards to a pre-condition and a post-condition if it always satisfies the postcondition at its termination, provided that it was started in a state satisfying the pre-condition
  - The standard term for this is partial correctness
  - If the program also always terminates, it is called totally correct
- For concurrent programs we need more machinery
  - Remember the interleavings
  - The possible state transitions are often not obvious from the program code
  - Simple pre and post conditions do not work so easily

Correctness of a Concurrent Program

- Partial correctness is generalized in a concurrent system to *safeness*
  - A program is *safe* with respect to some *invariant*, which holds always (i.e., a criteria which holds initially and is preserved by all the atomic actions of the system)
- Termination is generalized to *liveness*
  - A program satisfies a liveness criteria *T* with respect to a state *S*, if started in *S* it will eventually reach a state satisfying *T*
- To guarantee liveness we must make an assumption about scheduling
  - *Weak fairness assumption*: Any statement that is continuously ready, will be executed eventually.

Implicit and Explicit Processing

- Implicit parallelism
  - A sequential program is compiled to a lower lever explicitly parallel program
  - Done by a compiler or run-time environment and often HW dependent
- Explicit parallelism
  - Developer models parallelism on a coarse grained level
  - More portable, but requires non-trivial extra effort
- Declarative programming (e.g., functional programming)
  - Elimination of state and the use of pure functions would be ideal

Languages and Concurrency

Support for concurrency

- There are several ways of supporting, for example
  - Concurrency mechanisms can be a part of a language (PL)
  - Libraries with concurrency support are typical
    - Generic libraries (with several language bindings)
    - Built-in libraries of languages
  - System-level support
    - Operating systems (OS), virtual machines, true hardware
    - Execution environments, runtime systems
- There are also several ways to implement
  - Hardware-based, software-based
  - System-wide, for an application, etc.

Nested and Mixed Use

- Properly designed mechanisms can be nested
  - E.g., take a lock while already holding one
    - Note that you have to be careful when doing so
    - Deadlocks, performance problems, *etc.* can happen
- Also, different sync mechanisms can be co-used
  - E.g., use a semaphore while holding a lock
  - Also at different levels
    - E.g., take an OS lock while holding a PL lock
- Such things are not trivial
  - But they are very typical
    - Actually, the main stream way of concurrent programming
  - *I.e.*, there is plenty of things to study after completing a basic course on concurrent programming

Using Scala

- Scala is a real programming language
  - It does not have a single concurrency model
    - It has many different basic models
    - And, there are plenty of models built on top of those
  - There are two underlying implementations
    - Scala/JVM (the course uses this)
    - Scala/JavaScript
- We begin with the `synchronized` abstraction
  - Viewed from the classical theory, it is a built-in monitor
    - We will handle this in more detail later on
  - Viewed from the language, it is a intrinsic lock
    - Available for everywhere in Scala
    - The language abstracts the details (lock, release, except, …)

Scala, libraries and low-level code

- In concurrent programming *low-level coding* is implementing the concurrency mechanisms yourself
  - I.e., you write the algorithms that take locks, etc.
  - This is **very hard**, but sometimes really necessary
  - But very often you use a different language (e.g., C++)
- Mixing low-level code with PL mechanisms is complex
  - But often, you get good performance in this way
  - Lock-free code and spin-locks are a good example
- A much more common thing is to use libraries
  - And (more modernly), **concurrency frameworks** *
  - Frameworks are holistic, with libraries you have to know more

(*) More on the frameworks on Week 5 Slides.

About spinlocks

- A spinlock is easy to implement, but inefficient
  - Hardware instruction support for low-level implementations
  - A spinlock burns processor time on a uniprocessor and cause serious memory contention in a SMP and they can not guarantee any kind of fairness
  - A variant of this method, where the skip instruction is replaced by a timed sleep is called *polling*
- Do NOT use busy-waiting or polling as solutions to course problems
  - Weekly exercises and the programming assignments require the use of better synchronization mechanisms, introduced later on during the course

Spinlocks in practice

- Spinlocks
  - Simple approach to locking
    - uses active waiting and atomic operations
  - Memory is locked during atomic comparison
    - Performance is degraded
  - Cache thrashing and expensive operations (barriers)
    - Barriers keep loads and stores in order
- RW spinlocks
  - Spinlocks for readers-writers
    - No lock contention for readers
  - Used when writers are rare
  - Higher cost than normal spinlocks when there is contention

Basic Scala Concurrency

Threads (1/7)

- What are threads?
  - Threads in Scala are concurrent streams of execution within a Java Virtual Machine (JVM).
  - Depending on implementation, they may run on separate processors or through time-slicing of one or more processors
  - JVM threads may or may not correspond to the underlying operating system's notion of threads
  - The VM may create additional daemon threads for its own housekeeping needs

Threads (2/7)

- How do I use threads?
  - The main method is invoked in the main thread, which can create more threads. Threads can be manipulated using Thread objects
  - The method `Thread.currentThread()` can be used to get information about the current thread
  - This is mostly useful for debugging or demonstrating thread behavior. Can be bad practice for a production environment
  - Threads contain a `run()` method that contains the code executed in the thread. This is typically where you put your own code

Threads (3/7)

- **Creating threads (one way)**
  - Extend the Thread class and override its `run()` method with your own, having loop that prints integers from 1 to 10000:

```
class ThreadLoop extends Thread {
    override def run = {
        /* This method is executed in the new thread. */
        for (value <- 1 to 10000) { /* Each thread has its own value variable */
            println(value)
        }
    }
}
```

Threads (4/7)

- **Starting threads**
  - When the `start()` method of a Thread is called, a new thread is started and the `run()` method is executed in the new thread.
  - Start two instances of the `ThreadLoop` thread:

```
object ThreadDemo {
    def main(args: Array[String]) = {
        val demo1 = new ThreadLoop()
        val demo2 = new ThreadLoop()
        demo1.start()
        demo2.start()
    }
}
```

Threads (5/7)

- Letting other threads execute
  - A running thread may want to give the other threads a chance
  - `Thread.‘yield‘()` lets the current thread pause and let other threads run. The backticks are required because yield is a reserved keyword in Scala
  - “The scheduler is free to ignore this hint.”
  - Using this often can lead to busy-waiting on some platforms
  - Busy-waiting threads may prevent threads that could actually do something useful from getting a chance to run
  - Even if every thread gets to execute, it is tremendously wasteful of CPU power. Other processes and threads now have less time for actual work
  - Power-saving (and cooling) measures such as underclocking idle CPUs do not work if the CPU is busy-waiting. This method is mostly useful for testing and debugging

Threads (6/7)

- Letting other threads execute
  - `Thread.sleep(long)` makes a thread sleep for a time given in milliseconds
    - This still does not guarantee that other threads get a chance to execute
  - If there are lots of threads waiting like this, the system may spend all its time switching back and forth between a few of them
  - Other processes can prevent your program from running at all for a while
  - Increasing the time to sleep introduces unnecessarily long delays. sleeping can be a good idea when doing animation or similar tasks
  - Clearly, we need a way to tell the system that a thread does not need to run until another thread has done its job

Threads (7/7)

- **Waiting for threads to end**
  - When the `join()` method of a Thread is called, the calling thread waits until the `run()` method is completed in the referenced thread.
  - Start two instances of `ThreadLoop` and wait for them to complete:

```
object ThreadDemo2 {
    def main(args: Array[String]) = {
        val demo1 = new ThreadLoop()
        val demo2 = new ThreadLoop()
        demo1.start()
        demo2.start()
        try {
            demo1.join()
            demo2.join()
        } catch {
            case e: Exception => println("Exception!") /* Unreachable in this example */
        }
    }
}
```

Happens-before relationship

- Happens-before relationship (*hb*)* is...
  - Transitive: hb(a, b) ∧ hb(b, c) ⇒ hb(a, c)
  - Antisymmetric: hb(a, b) ⇒ a = b ∨ ¬hb(b, a)
  - Naturally visualized as a directed, acyclic graph
  - Not the same as "A happens before B" in real time!
    - The logical relationship and the order in which instructions are executed on the CPU are two different things
- The *hb*-relationship is defined in the Java Language Specification
  - http://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html# jls-17.4.5

(*) This connects to weak behaviors, check Week 6 Slides for more details.

Using Shared Memory (1/2)

- When using shared memory (non-local variables, etc.) synchronization is needed
  - Synchronization actions can be used to ensure one thing happens-before another in another thread. If *a* happens-before *b*, the effects of *a* will be visible to *b*. If not, *b* may or may not see what *a* did.
  - Operations in a thread happen-before following operations in the same thread
  - The happen-before relation is a partial order
  - As noted previously, starting a thread happens-before the thread starts and a thread ending happens-before another thread notices that it has ended (through, *e.g.*, `join()`)
  - Default initialization of variables (the 0, false or null automatically written to new non-local variables) happens-before all threads start

Using Shared Memory (2/2)

- Bad things can happen
  - If proper synchronization is not done:
    - “Normal” non-deterministic behavior can occur (interleaving of operations). Other threads may not see changes to shared variables.
    - double and long variables may have incorrect values (these reads and writes are not guaranteed to be atomic)
  - If there are two operations on a variable such that one of them is a write and which are not ordered by happens-before, we have a data race
  - Correct synchronization (freedom from data races) for shared variables is always required on assignments of this course
    - . . . unless you can prove that the program behaves correctly with respect to the quite subtle JVM memory model…

Volatile variables (1/3)

- Ordering of reads and writes
  - All reads and writes to the same volatile variables are totally ordered
  - Every access to a volatile variable uses the shared main memory; each write
  - to the variable happens-before the following reads of the volatile variable
  - By transitivity on happens-before: This means that all writes of any other variables of the thread writing the volatile variable also happen-before any reads of a thread done after reading the volatile variable

Volatile variables (2/3)

- Non-atomic reads and writes
  - Declaring a long/double variable volatile makes its reads & writes atomic. long/double read/write operations may otherwise consist of two operations
  - 32-bit machines often read and write 64-bit values in two steps.
  - A read operation may see half of one value and half of the other. Irrespective of their size, references are always read and written atomically

Volatile variables (3/3)

- Declaring
  - Variables can be declared volatile with the volatile annotation.
    ```
    @volatile var a = 10
    ```
  - Declaring a reference variable volatile affects only the reference to the object, not the object's contents.
  - Volatile variables can be useful as flag variables, but are of limited use in practice

Mutual exclusion (1/7)

- Mutual exclusion is necessary to ensure multiple variables are read and written atomically
- Scala provides **locks** that can be used to solve this problem
- Each `AnyRef` has an associated **intrinsic lock**.
  - Locking is done by declaring parts (methods or blocks) of the code synchronized
  - The parts are mutually excluded from each other, *i.e.*, executed as object-specific critical sections, since entering a synchronized block or method acquires the lock and leaving releases it
  - The feature is inherited by all classes from the `AnyRef` root class

Mutual exclusion (2/7)

- synchronized **blocks**
  - To ensure ownership of myObject's lock:

```
def exampleMethod = {
    myObject.synchronized {
        /* code here will be executed only
            by one thread at a time */
    }
}
```

Mutual exclusion (3/7)

- synchronized **methods**
  - A whole method may be declared synchronized by using `synchronized`:

```
def myMethod = synchronized {
    /* code here will be executed only
        by one thread at a time */
}
```

Mutual exclusion (4/7)

- synchronized **methods versus blocks**
  - A synchronized method works exactly as if the body was in a synchronized block on this:

```
def myMethodAlternate = {
    this.synchronized {
        /* code here will be executed only
           by one thread at a time */
    }
}
```

Mutual exclusion (5/7)

- How synchronized works
  - Before a thread exits a synchronized block (normally, or by throwing a `Throwable`), it releases the lock
  - Before a thread starts executing a synchronized block, it acquires the object’s lock (waiting if not available)
  - Any variable values written before a lock is released are guaranteed to be visible after the lock is then acquired; the release happens-before the next time the lock is acquired
  - In other words, changes written in a synchronized block are guaranteed to be visible to other code synchronized on the same monitor; volatile is not necessary in this case

Mutual exclusion (6/7)

- Note on the use
  - Locks are re-entrant; a thread that already owns the lock can acquire it again without deadlocking
  - Example: A synchronized block is called from another synchronized block of the same object
  - Example: `a.m1()` calls `b.m1()`, which calls `a.m2()` when both methods m1 and m2 of object a are synchronized
  - In such a case the thread gives up the lock only after exiting the last synchronized block or method

Mutual exclusion (7/7)

- Problem
  - Two or more threads trying to acquire locks the other holds lead to *deadlock*
- Deadlock avoidance
  - If possible, use synchronized only on this
  - While holding a lock, avoid calling methods with locking on another object or otherwise acquiring another lock
  - If necessary, merge several locks into one
  - When nesting locks, always use the same locking order

Conclusions

- References
  - Scala documentation
    https://docs.scala-lang.org/
  - The Java Language Specification, Java SE 7 Edition
    - Especially Chapter 17
      http://java.sun.com/docs/books/jls/
  - Java 2 Platform, Standard Edition 7 API Specification
    - Especially classes java.lang.Object and java.lang.Thread.
      http://docs.oracle.com/javase/7/docs/api/

Related readings and exercises

=> Check A+