Aalto University
School of Science

CS-E4110 Concurrent Programming

Week 1 – Exercise session

2021-11-05

14:15 – 16:00

Jaakko Harjuhahto

Weekly Exercise Sessions

- Exercises on Fridays 14:15 – 16:00 on Zoom
  - Link to Zoom session available on MyCourses and Zulip
- Wednesday sessions are focused on theory
- Friday sessions are more practical in nature, including:
  - Additional material
  - Tips and tricks for concurrent programming
  - Demonstrations
  - Q&A for assignments
  - Reviews of correct answers

Exercise Session Materials

- Exercise session materials are supplementary
  - Use the weekly slides and the textbook as the primary reading material for the exam
- The exercise slides will be posted on Zulip
  - Including code samples, when appropriate

Today

- Thinking about concurrency
- Scala for concurrent programming
- Program semantics
- A+ in practice
- General Q&A

Thinking About Concurrency

A Thread-centric View

- One or more threads per process
- Threads for a process share memory and interact by reading and writing to this shared memory
- We assume that we are working with general purpose computing hardware running a typical desktop OS
  - The OS has a pre-emptive scheduler responsible for allocating CPU execution time
  - Any discussion on concurrency will require assumptions about the hardware and runtime environment
  - More specialized systems exist...

The Problem With Threads

A programmer can **not** control how the OS and hardware schedule threads

The programmer must ensure that the program is correct for **all** the possible schedules

The Problem With Threads

A programmer can **not** control how the OS and hardware schedule threads

The programmer must ensure that the program is correct for **all** the possible schedules

DIFFICULT

Interleaving

- Interleavings: $\frac{(n*m)!}{(m!)^n}$
  - *n* number of traces (~threads)
  - *m* number of actions per trace
  - Example: 10 threads with 6 action per thread
    - 2.22e53 possible schedules
    - Comparable complexity to our course assignment
    - Even with 1000 schedules tested per seconds, it would take 2.22e49 seconds to exhaustively test this small system
    - For comparison, the age of the universe is around 1.38e17 seconds
- State space explosion!

Correctness Properties

- Safety
  - Nothing bad ever happens
    - If preconditions hold, postconditions will also hold
- Liveness
  - Something good will eventually happen
    - E.g. The program does not deadlock
- Memory consistency
  - Reads from and writes to a specific memory address from different threads are well ordered
    - Necessary if we want to reason about the possible results of computations that interact with shared memory
    - E.g. Sequential consistency

Scala for Concurrent Programming

Why Scala?

- The bachelor’s curriculum teaches Scala
  - Many students, especially from other universities, know Java and the differences between Java and Scala for our uses case is quite small
- The Java Virtual Machine (JVM) offers a unified abstraction layer
  - Was one of the first languages to start formalizing **language level** abstractions for concurrency
  - Most OS and hardware details abstracted away
    - Many older environments relied on OS and hardware primitives
    - Portability and testing on different systems was a challenge

Other Alternatives?

- C/C++
  - Modern ISO C++11 and newer introduced standardized concurrent programming
    - E.g. `std::thread`, `std::scoped_lock`
  - Historically reliant on OS dependent libraries and standards
    - E.g POSIX on Linux and other *NIX derivatives
  - Not every student takes a course teaching C/C++
- Rust
  - Modern systems programming language, designed to support concurrency from the ground up
  - Is not currently taught at Aalto. Learning a new language and concurrent programmin in parallel is extra effort

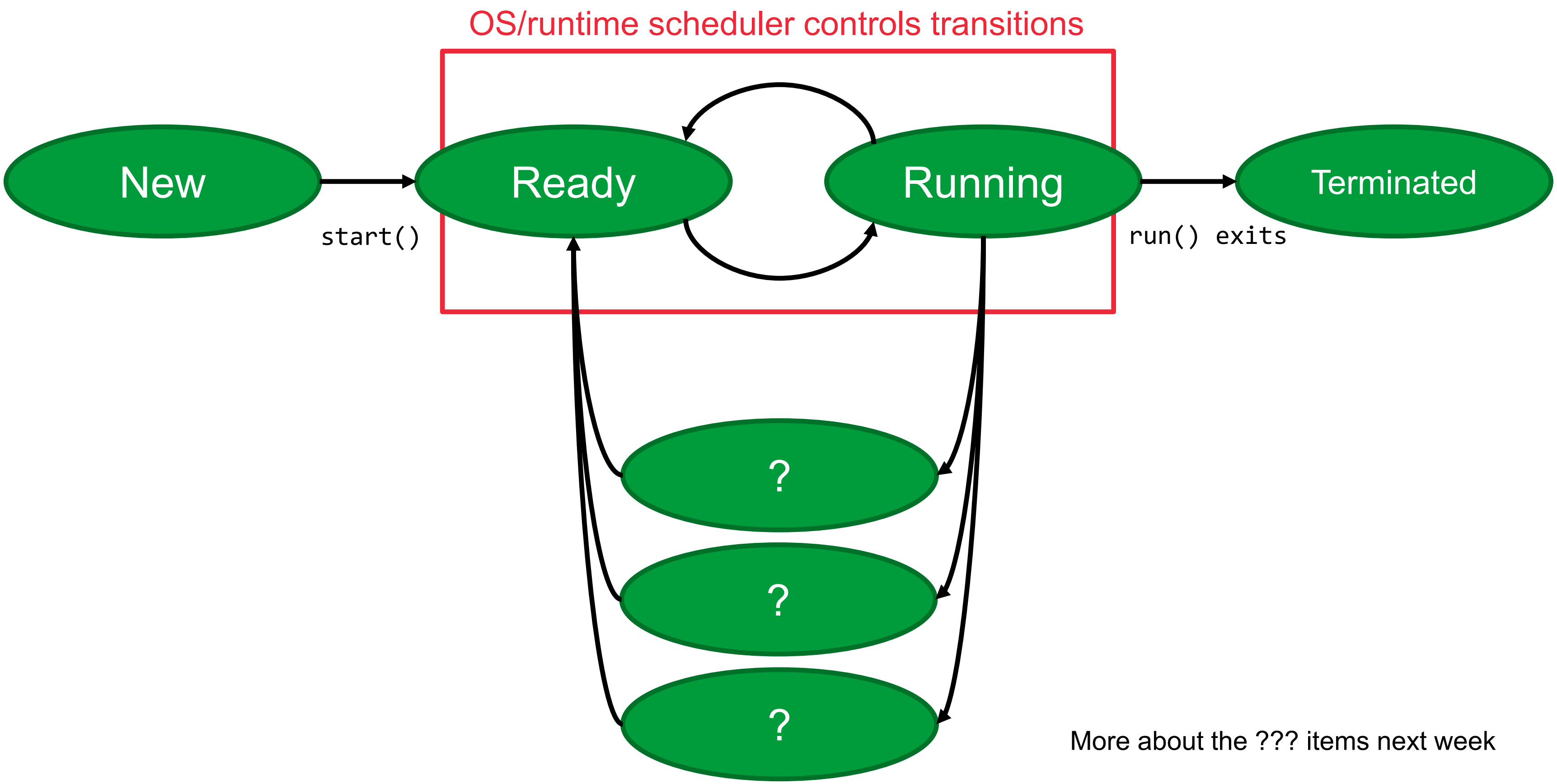
Non-Alternatives

- Python
  - It is possible to create threads, but they will not actually run concurrently
  - The global interpreter lock (GIL) will restrict threads to running one at a time
  - Not true shared memory concurrency
- JavaScript
  - The runtime uses threads internally to run asynchronous tasks, requests and callbacks in the background, and the results are processed in the event loop
    - The event loop runs sequentially in a single thread
  - Again, not true shared memory concurrency

Scala Threads

- Behaviour inherit from the JVM, the same as Java
- When the JVM is started, the first thread is launched to execute the main method
  - Often called colloquially called the ”main thread”
  - Identical and equal to every other thread from the JVM’s perspective
- The first thread can launch new threads
- The JVM terminates when every single thread has died
  - Excluding daemon threads, but these should be used only when you know exactly what you are doing

Thread Lifecycle



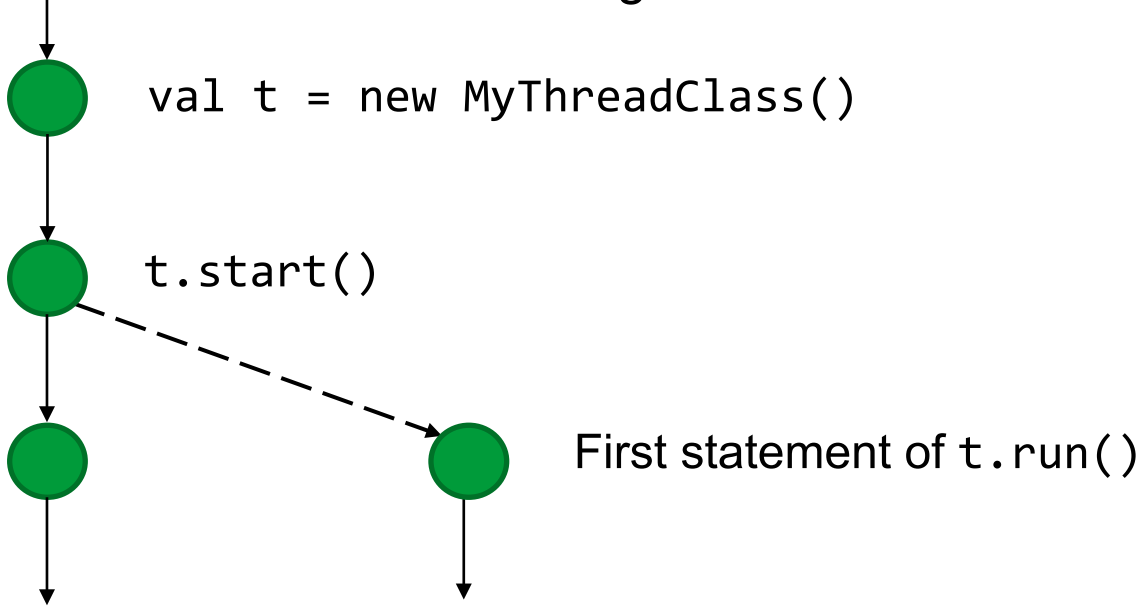
More about the ??? items next week

Creating Threads

- Write a class extending `java.lang.Thread`
  - Override the method `run()` and implement your code
  - The code inside `run()` is often called the ”thread’s body”
- Write a class implementing `java.lang.Runnable`
  - Write an implementation for run()
  - Create a new instance of Thread and give the runnable as a constructor argument
    - `val t: Thread = new Thread(new myRunnable())`
- References to threads can be manipulated just like any other objects
  - E.g. you can store a collection of threads as a `Seq[Thread]`

Launching Threads

- Call `start()` on the thread instance to start the thread
  - Beware, a common mistake is to call `run()` instead of start(). This will not actually launch a new thread, just execute the body in the thread making the call



Wait for Threads to Die

- Call `join()` on a thread
  - The calling thread will wait until the target thread has finished executing
  - E.g. Thread `A` calls `B.join()`, and will only continue after B has executed it's body and died
- Usefull for fork-join style parallelism
  - Split a problem into smaller parts
  - Start a thread for each part
  - Wait until these threads complete using `join()`
  - Combine results in the original thread and proceed

JVM Thread API

- `Thread.currentThread()` returns a reference to the currently executing thread
  - Mostly useful for debugging
- `sleep(ms: long)` makes the thread sleep/pause for the given duration in milliseconds
  - OS timers for sleeping can be very imprecise, don't expect predictable behaviour
  - On this course, do not use sleep to ”wait for things”!
- `´yield´()` lets other threads execute
  - The backtics are necessary, as yield is a reserved word in Scala
  - Only a hint for the OS scheduler, which can choose to ignore it

More JVM Thread API

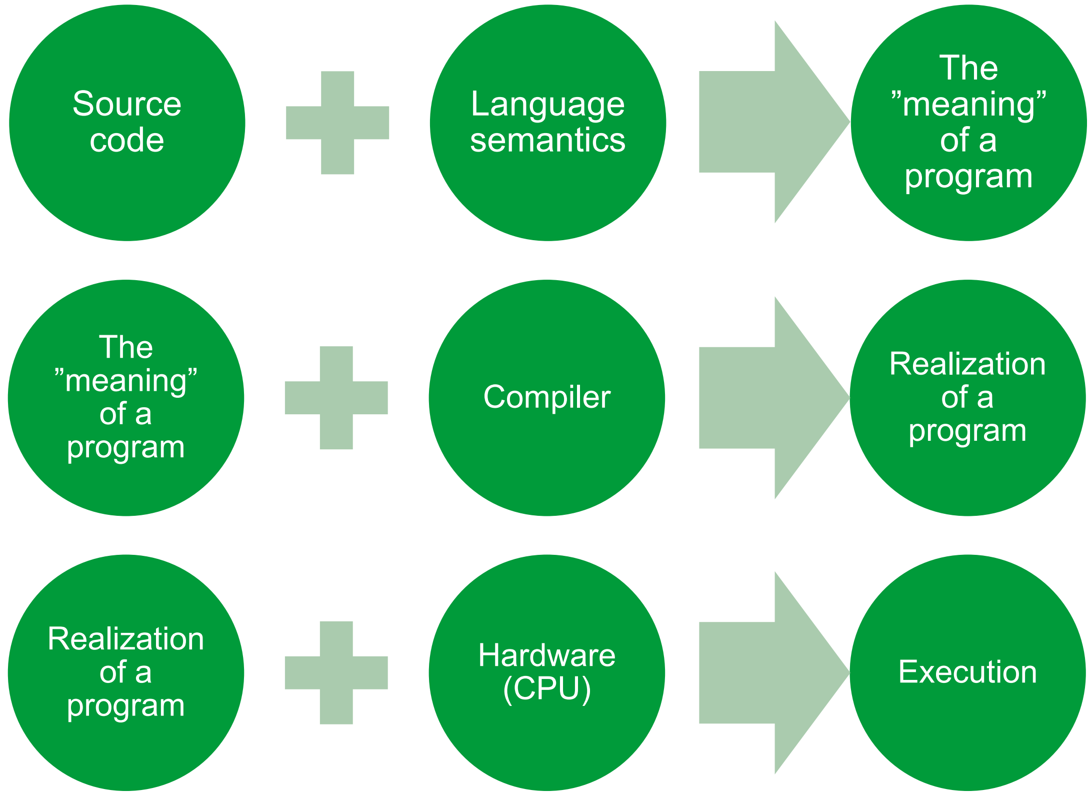
- Interrupt threads and check their interrupt status
  - More details next week when we discuss JVM monitors
- Define priority for threads
  - Hint for the JVM which threads should receive more or less CPU execution time
  - Note that this is not the same as OS process/thread priority. The priority hints inside the JVM won't affect how much CPU time the OS will dedicate to the JVM
- Give threads human readable names
  - Occasionally useful for debugging
- Deprecated methods from early Java editions
  - Obviously, don't use them

Other Considerations

- Each threads has it's own control structure
  - Requires a fixed amount of memory
  - Depending on the implementation, also OS resources
  - There is a limit to how many threads a program can create before running out of resources
- Creating threads is relatively expensive
  - Allocating resources and registering a thread with the JVM control structures requires time
  - Better to keep them alive and re-use when possible
    - Large applications often use thread pools, executor services etc...
- Lots of JVM parameters for tuning thread behaviour
  - Not relevant for our course...

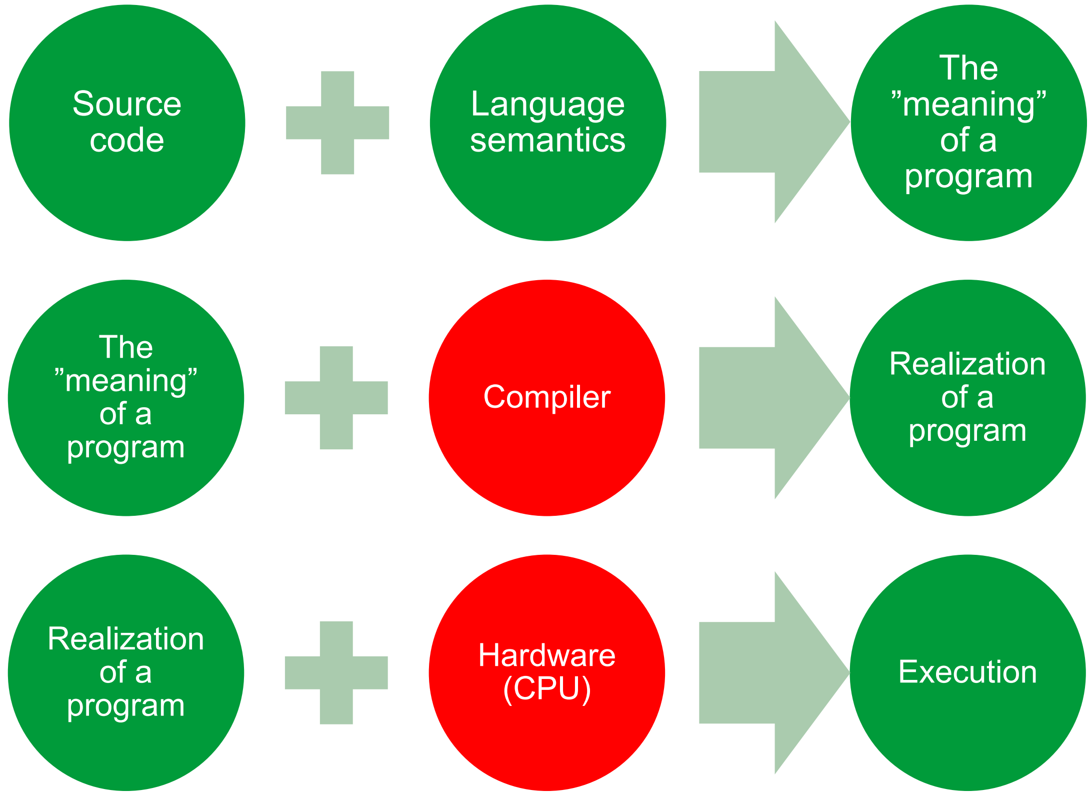
Program Semantics

From Source Code to Execution



Note that this process is one-directional

Transformations



The red steps can transform the program in any way that is not explicitly forbidden by the ”meaning” of the program

Transformations as Optimizations

- Transformations only retain single threaded behaviour
  - Inter-thread behaviour can and will change, unless program semantics explicitly and actively disallow this
    - It’s up to the programmer...
- The performance impact of compiler transformations is significant
  - Possibly an order of magnitude (10x or more) speedup
- Hardware transformations are always enabled
  - E.g. Out-of-order execution of instructions
  - Again, possible to gain an order of magnitude speedup

Running Example

- The Scala class `Box[A]`
  - A container class able to store a single item of type A
  - The box is either empty, or it contains an item
- The public API
  - `put(elem: A): Unit`
    - Write the argument `elem` into the Box
    - `elem` can not be null
  - `get(): A`
    - Returns an item of type A that was previously written into the box
    - If the Box is empty, will wait until an item becomes available
    - Contract: must never return a null value

Example

- What can you say about the program on the right?
- What does `get()` return?
- Is it possible that `get()` returns a `null` value?
- Is the ”meaning” of the program universal regardless of execution hardware?

```
class Box[A] {
  var item: A = null
  var isPresent: Boolean = false

  def put(elem: A): Unit = {
    require(elem != null)
    item = elem
    isPresent = true
  }

  def get(): A = {
    while(!isPresent) {
      //busy wait
    }
    item //should always != null
  }
}
```

Example – Reordering of Writes

- Reordering of writes is allowed on ARM CPUs
- For single threaded execution, the meaning of put() does not change if the two writes change places
- Consider the multithreaded case: what happens when thread P calls `get()` and thread Q calls `put()`, if the two writes in `put()` change places?

```
class Box[A] {
  var item: A = null
  var isPresent: Boolean = false

  def put(elem: A): Unit = {
    require(elem != null)
    item = elem
    isPresent = true
  }

  def get(): A = {
    while(!isPresent) {
      //busy wait
    }
    item //should always != null
  }
}
```

Example – The Problem

- The problem with the code on the right is that it does not capture the semantics of ”`item` and `isPresent` are related”
  - A human reader can understand this
  - The compiler won't
- The programmer must make this relationship explicit with:
  - Locks
  - Enforced happens-before
  - Memory barriers
  - Etc...

```
class Box[A] {
  var item: A = null
  var isPresent: Boolean = false

  def put(elem: A): Unit = {
    require(elem != null)
    item = elem
    isPresent = true
  }

  def get(): A = {
    while(!isPresent) {
      //busy wait
    }
    item //should always != null
  }
}
```

A+ in Practice

How A+ Grading Works

- The testing framework creates *scenarios* of calls to the public methods of the class it is testing
  - Thousands of scenarios, tries to find problematic scenarios
  - Overrides Scala primitives and monitors these for deadlocks
- Unit testing concurrent programs is not deterministic
  - If A+ finds error, your submission **is not** correct
  - If A+ does not find errors, your submission **might be** correct
- ”For concurrency related bugs, absence of evidence is not evidence of absence”

A+ Gotchas

- When writing classes that use or require locking, use the intrinsic lock for the class instance itself
  - i.e. `this.synchronized{ ... }` and not `anotherObject.synchronized{ ... }`
  - Otherwise the testing framework will not work properly
- If you see unexpected exceptions, contact course staff through Zulip
  - For example: `java.lang.OutOfMemoryError` exceptions
  - Might be a A+ problem, please notify course staff through Zulip, they can rerun your submissions

Q&A