Aalto University
School of Science

CS-E4110
Concurrent Programming

Week 2 – Exercise session

2021-11-12

14:15 – 16:00

Jaakko Harjuhahto

Today

- Building blocks: mutual exclusion and waiting
- Java Virtual Machine monitors
- General monitors
- Happens-before
- General Q&A

Building Blocks:
Mutual Exclusion and Waiting

Mutual Exclusion

- We want to limit access to data or resources to a single thread at a time, to avoid data races and schedules where interleaving multiple actions leaves the system in an invalid state
- Mutual exclusion is necessary to make a group of actions appear atomic
- Remember limited critical reference (LCR) rules
  - Even `a = a + 1` is a group of actions

Waiting: Why Is It Necessary?

- In an application with multiple threads running concurrently, it is often necessary to **wait for** things
  - Data becomes available from other threads, network messages, I/O, user input, hardware responds, etc
  - Classic: producers and consumers
    - Consumers can only proceed when the producer is ready to hand over data or resources. The consumer must wait for the producer.
- Process synchronization
- The thing we are waiting for is the **wait condition**

The BAD Solution – Busy Wait

```
while(waitCondition) {
  //do nothing
}
//proceed
```

- Use a while loop to repeatedly check the wait condition and leave the loop only after the condition is met
- The while-loop will run as fast as possible, consuming all the CPU execution time it can get
  - Other threads and processes can't use this CPU time
  - Excessive energy consumption
- Also known as a spinlock
  - Used for low level programming (e.g. OS kernel) when a very short upper limit to the spin time can be guaranteed

NOT A VALID SOLUTION TO COURSE PROBLEMS

The BAD Solution – Polling

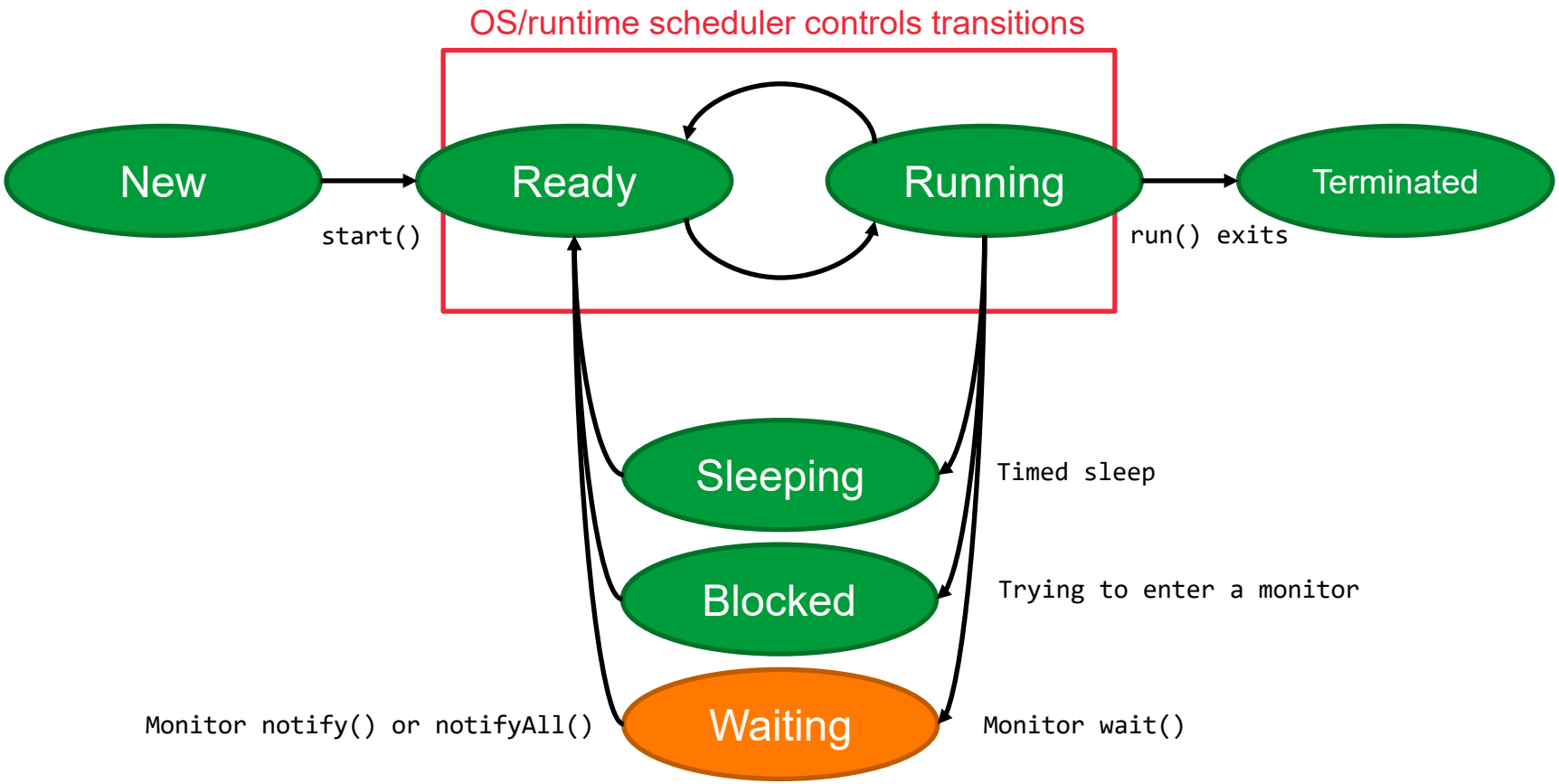
```
while(waitCondition) {
  sleep(x) //milliseconds
}
//proceed
```

- Add a timed sleep to a busy-wait loop
  - Less CPU cycles wasted
  - Still, not a good solution...
- Problem: how long should the sleep be?
  - A short sleep is almost as bad as a busy-wait loop
  - A long sleep makes the thread unresponsive, laggy and slow
  - Timing can be inaccurate

The Good Solution – Waiting State

- Place a thread into a **waiting state**
  - The OS/JVM scheduler will not select the thread for running
  - Will not consume CPU resources while in the waiting state
- Once the wait condition is met, re-activate the thread
  - Called ”signaling the thread” or ”notifying the thread”
- The JVM provides waiting as a feature of the JVM monitors

JVM Thread Lifecycle



Java Virtual Machine Monitors

JVM Monitors

- The JVM monitors offer us three critical features:
  - Mutual exclusion
  - Proper waiting
  - Happens-before
- The JVM provides us with monitors
  - We can build other primitives and data structures from these
- The JVM monitors are **an implementation** of the more general concept
  - Other language and library implementations very significantly

JVM Monitors

- Every JVM object has an associated monitor
- Objects inherit the monitor and three related methods from the class hierarchy superclass
  - AnyRef in Scala and java.lang.Object in Java
  - These three methods may be called only when holding the objects intrinsic lock, i.e. from inside a `synchronized` block
- Often called the object's **intrinsic lock**

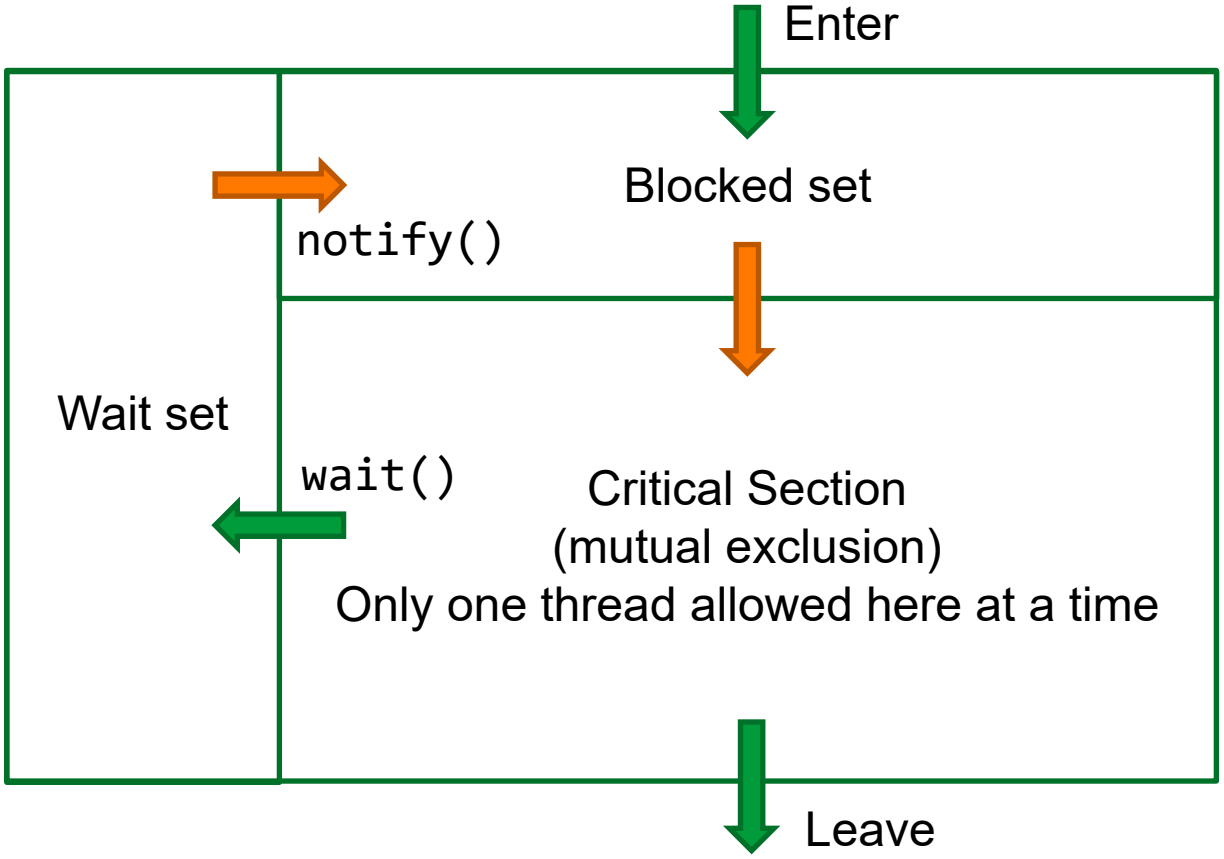
JVM Monitors – Intrinsic Locks

- Every class descending from AnyRef has an intrinsic lock, accessed with the by the `synchronized` keyword
  - Declare methods or blocks of code as synchronized
    - `def put(elem: A): Unit = synchronized { … }`
    - `synchronized { ... }`
  - Entering a synchronized block requires acquiring the intrinsic lock and leaving the block releases the lock
  - As only one thread can hold the intrinsic lock at a time, the synchronized blocks are mutually exclusive
- Enforce atomicity for a group of operation

JVM Monitors – API

- `wait()`
  - Places the current thread in a waiting state and adds it to the objects wait set
- `notify()`
  - Signals one (effectively random) thread in the wait set
  - The thread calling signal will continue executing
- `notifyAll()`
  - Signals all the threads in the wait set
- These can be called only from inside a `synchronized` block for the same monitor
  - Otherwise, an `IllegalMonitorStateException` is thrown

JVM Monitors Visualized



The sets are actual sets, i.e. they do not have any ordering and the orange transitions are effectively ”one random member of the set”

notify() vs notifyAll() – Correctness

- Problem: multiple wait conditions and a single monitor
- Wait conditions W1 and W2
  - There are threads waiting for both W1 and W2
  - W1 is met and a single `notify()` is called
  - The signal will randomly select a thread to wake up, it does not know it the signaled thread was waiting on W1 or W2
    - If a thread woken up was waiting for W2, it can not proceed and will return to waiting
    - The *signal was lost* and a deadlock can ensue
- Beware reasoning via natural language about wait conditions
  - Is it possible that threads are waiting for both isEmpty and isFull at the same time?

notify() vs notifyAll() – Performance

- Problem: notifyAll will wake up all the waiting threads, even if only one of the threads can actually proceed
- All the signaled threads will compete for the lock
  - The first thread to gain the lock can proceed
  - All the remaining threads will still try and this can cause heavy lock congestion as well as resource consumption
- Weak fairness assumption
- Remember that correctness >> performance
  - Producing the wrong answer faster is not a good optimization

Spurious Wakeups

- Threads may leave the waiting state before the wait condition is met and the threads receives a signal
  - This is known as a spurious wakeup
  - Legacy limitation of some operating systems
  - Very rare. From a performance point of view: ignore
- Simple and universal fix
  - Always place calls to wait() inside a while loop

```
while(waitCondition) {
  wait()
}
//proceed
```

The Box Example

- Implement the example from last week using JVM monitors
- Make `put()` and `get()` mutually exclusive by synchronizing them
- Use the monitor to wait properly

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

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

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

Alternative Syntax

- Make explicit which monitor is used by including the object name in the method invocation
- Here: the object itself, referred to with the keyword `this`
- This version is functionally identical to the previous one
  - Only syntax changes
- Suggestion: prefer this style for clarity for more complex code

```
def put(elem: A): Unit = {
  this.synchronized {
    require(elem != null)
    item = elem
    isPresent = true
    this.notify()
  }
}

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

Yet Another Alternative

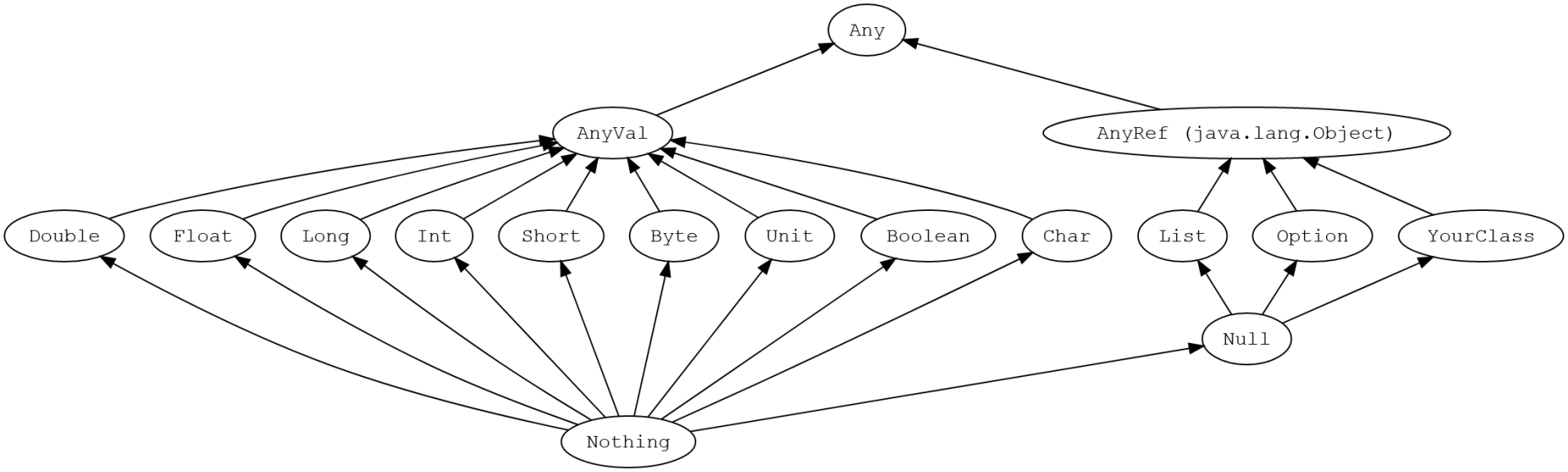
- The explicit object used can by anything descending from `AnyRef`
- Note: A+ will assume you always use the intrinsic lock for the class instance itself
  - I.e. `this`
- Beware object instance equality
  - E.g. Newer use Strings as monitor objects

```
val foo = new java.lang.Object()

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

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

Scala Type Hierarchy



Note the difference between value types and their boxed variants:

- Int is a value type, descending from AnyVal
- Integer is a class, which contains a single Int. Since it is a class, it descends from AnyRef

Interrupts

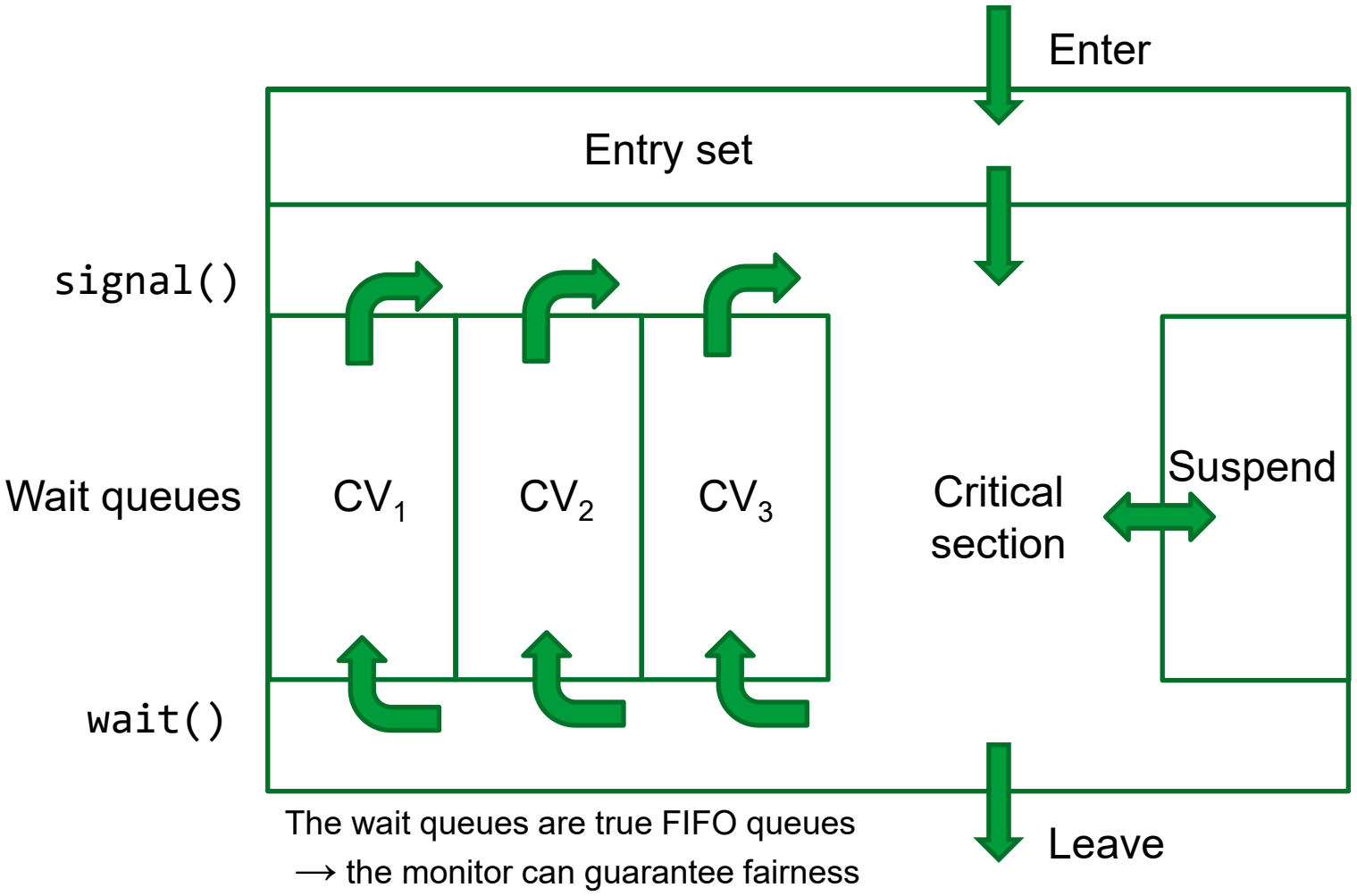
- Conceptually: tell a thread that it should stop what it is doing and handle the interruption request
- Each thread has an interrupt flag
  - Raise with `Thread.interrupt()`
  - When the flag is raised, encountering a `wait()` will throw an `InterruptedException`
  - If the thread is currently waiting, raising the flag will wake up the thread and throw an `InterruptedException`
- Often used to handle non-routine cases
  - Example: the application is shutting down, inform every thread that they should stop what they are doing and terminate, even if they are waiting for something else to happen

Monitors in General

General Monitors

- Monitors can have explicit **conditional variables** (CV)
  - The CV is a queue of threads waiting for that condition to be met
  - A monitor can have multiple conditional variables
  - In the JVM, there is only one invisible and implicit CV
- JVM monitors use signal-and-continue semantics
  - The thread calling notify() will continue executing until it leaves the monitor critical section
  - Alternatives include signal-and-urgent-wait
    - The thread issuing the signal will give the thread that receives the signal priority access to the critical section
- The course textbook (Ben-Ari) uses stronger general monitors

Example – Stronger Monitor Semantics



The wait queues are true FIFO queues
⟶ the monitor can guarantee fairness

Happens-before

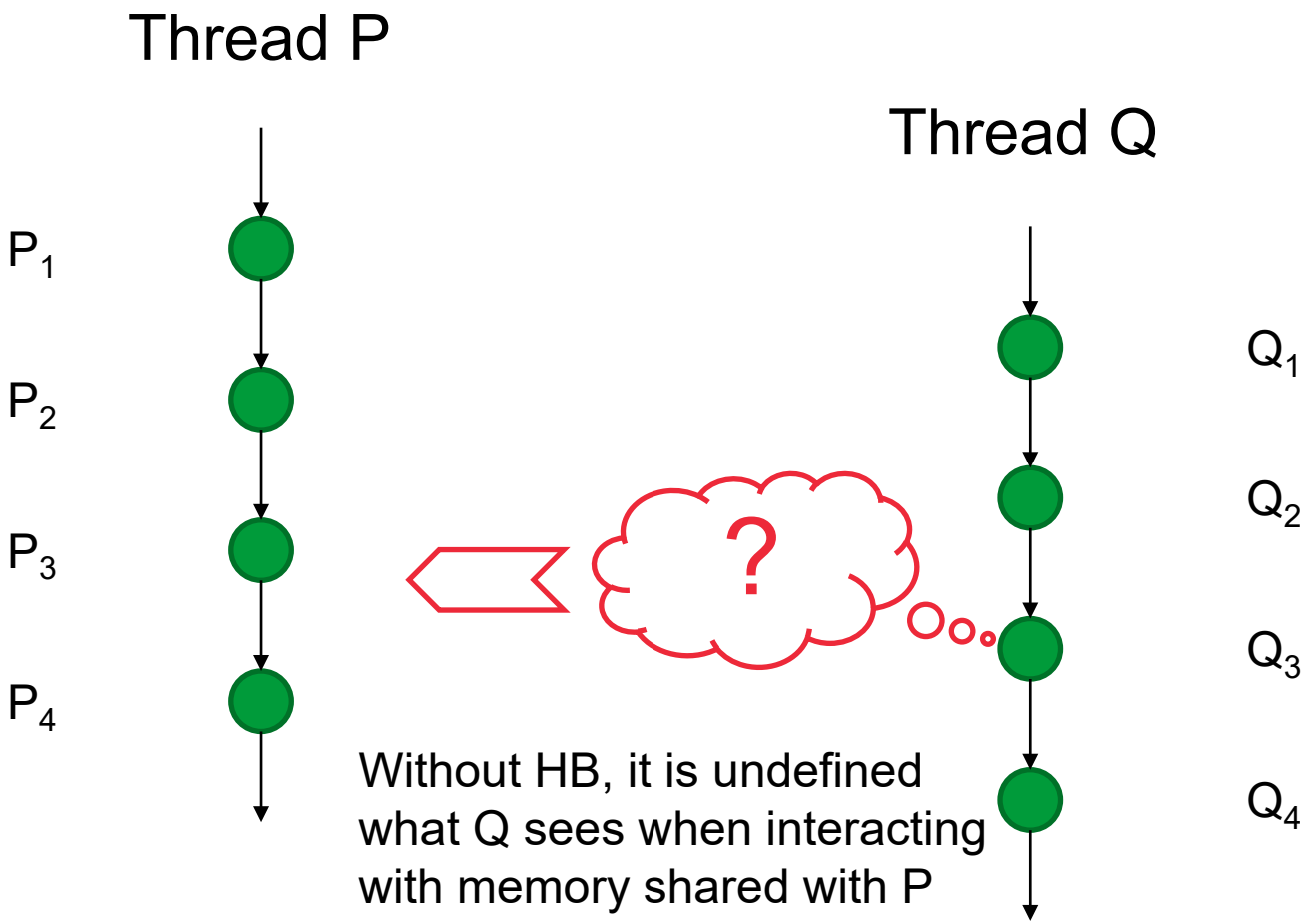
Happens-before

- Last week we saw how write reordering can break code if the implementation does not capture the necessary semantics
- Informally: happens-before defines a logical order for actions, which must hold even for inter-thread actions
- Formally: Two actions can be ordered by a *happens-before* relationship. If one action *happens-before* another, then the first is visible to and ordered before the second.

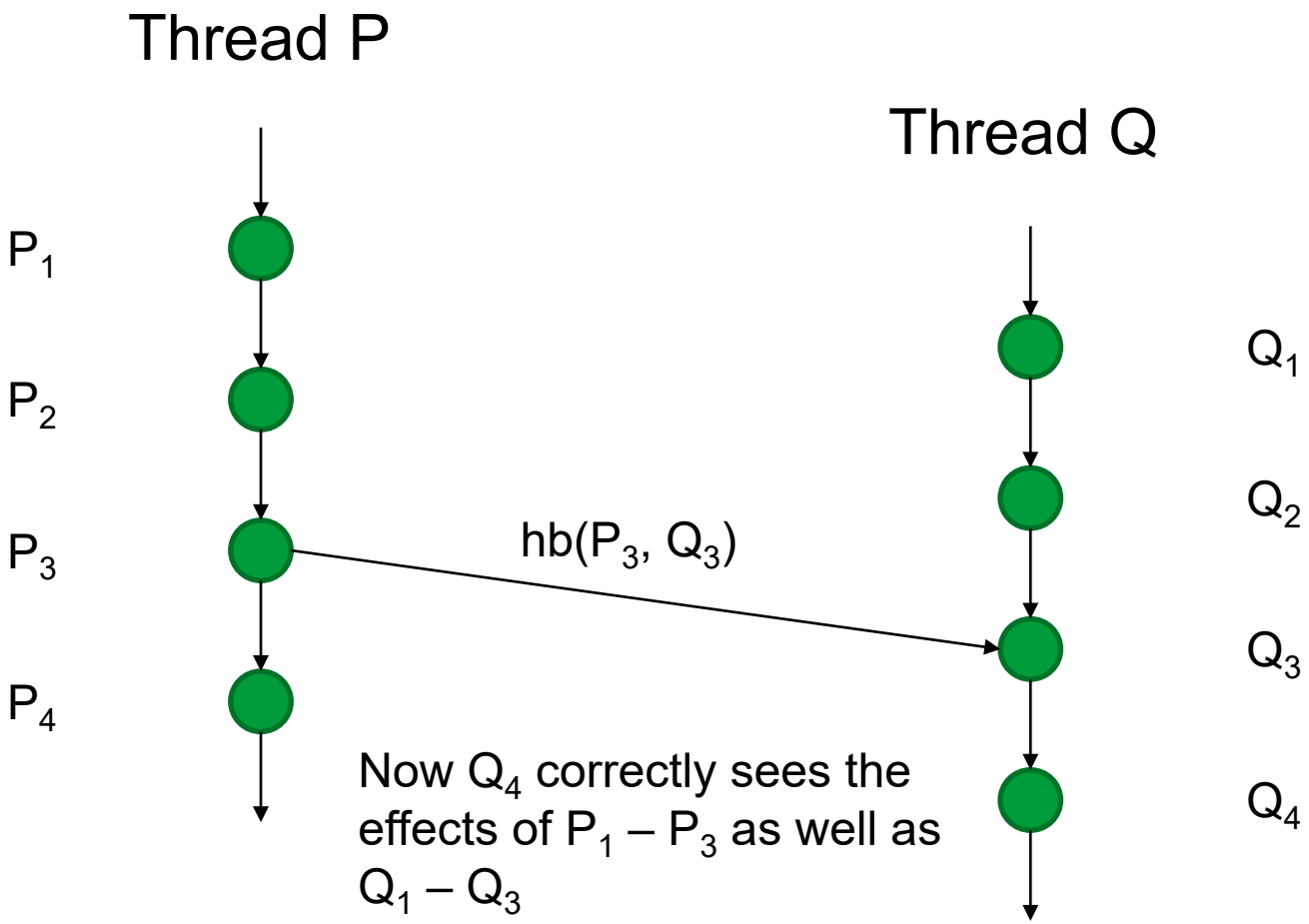
The HB-relationship

- Logical relationship between parts of a program
  - Note that hb is a logical relationship, that defines what the program should mean, not an instruction on how a physical CPU should execute the program
  - Transitive: hb(x,y) and hb(y,z) then hb(x,z)
  - Antisymmetric: $hb(a, b) \Rightarrow a = b \vee \neg hb(b, a)$
- Visualize as a directed acyclic graph of actions
- Fully defined for the JVM in JLS chapter 17.4.5
  - https://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.4.5

Happens-before Visualized



Happens-before Visualized



Shortlist of HB Rules* (1/2)

- For a single thread, if x come before y in program order, then hb(x, y)
  - Self evident, but necessary to use in combination with other rules to prove that a multi-threaded program is properly synchronized
- Unlocking a monitor happens-before every subsequent locking on that monitor
  - Code inside `myObject.synchronized{...}` blocks is properly ordered according to happens-before
  - Note that this only applies to the lock of one specific object
  - `a.synchronized{...}` and `b.synchronized{...}` are NOT ordered with a happens-before relationship

* Read the Java Language Specification chapter 17.4.5 for the full list

Shortlist of HB Rules* (2/2)

- A write to a `volatile` field happens-before every subsequent read of that field
  - In Scala, declared as `@volatile var X = ...`
  - Making an object reference volatile does not affect the object the reference points to
  - Very limited practical use
- A call to `start()` happens-before any subsequent action in a the started thread
- All actions in a thread happens-before any other thread succesfuly returns from a `join()` on that thread

* Read the Java Language Specification chapter 17.4.5 for the full list

The Box Example

- Is the implementation using monitors correctly synchronized?

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

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

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

Box With Volatiles

- We can also fix the Box example without JV monitors by manually introducing a hb-relationship between writes to and reads from the variable `isPresent`
  - Prevents the problematic write reordering scenario
  - Note that `synchronized` is not used
- This is a rare example of when `@volatile` is usefull
- Monitors should be your go-to choice for enforcing HB, especially on this course!

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

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

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

Q&A