Practical Concurrent and Parallel Programming X

Coroutines II

IT University of Copenhagen

Friday 2020-11-06

Kasper Østerbye

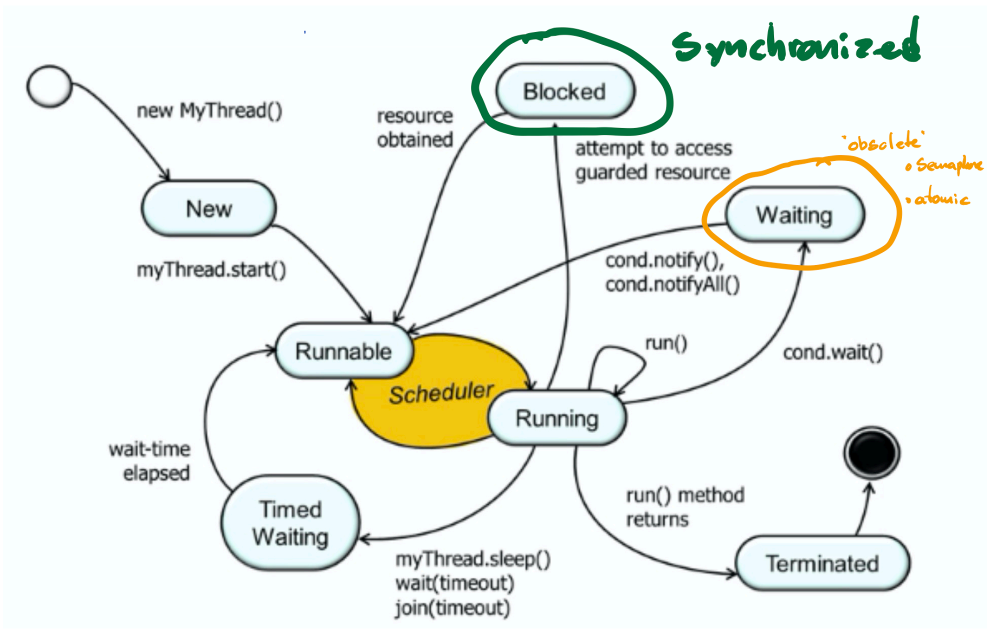
Starts at 8:00

Plan for today

Follow up on last week

Follow up and loose ends

Scheduler of Java threads



Recab

The java virtual machine

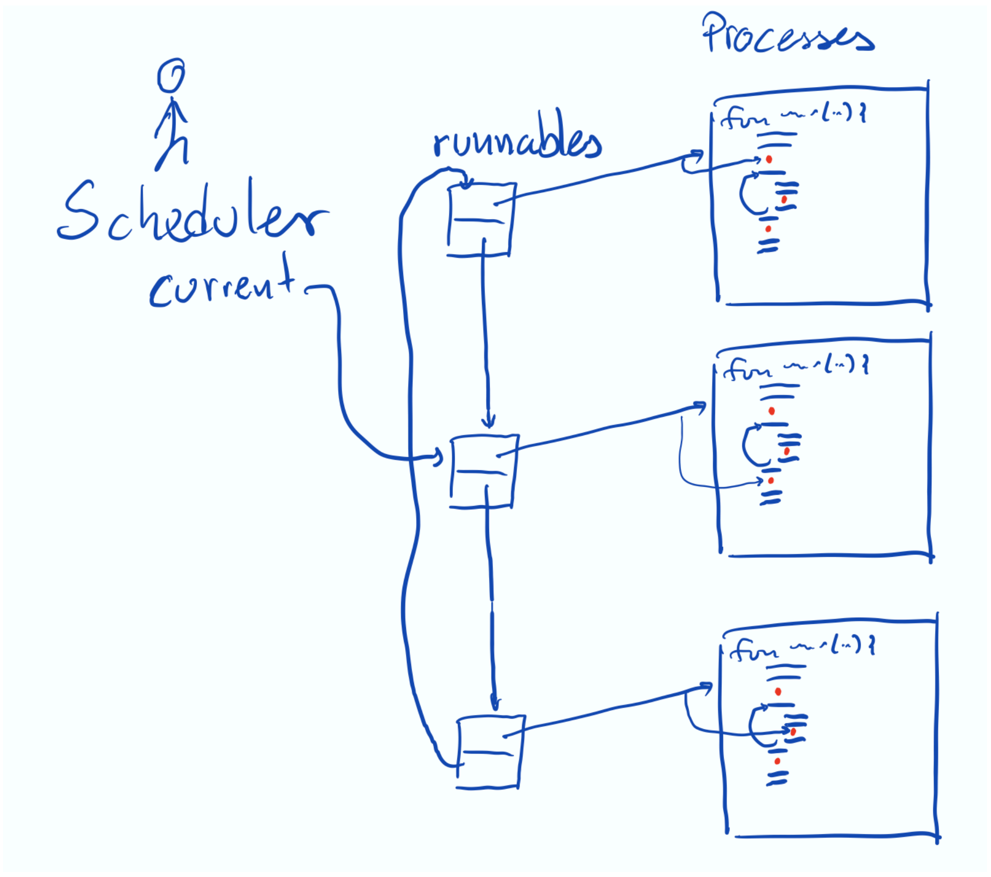
1. starts the sheduler
2. creates a (main) thread
3. which becomes runnable
4. and the scheduler runs it

The JVM scheduler maintains:

1. a list of all threads
2. a low-level mechanism which can put run a thread
3. policies for when to "pre-empt" a running thread and move it to "runnable"

The JVM scheduler runs until the jvm is terminated

Data structures in a Scheduler



Scheduler algorithm

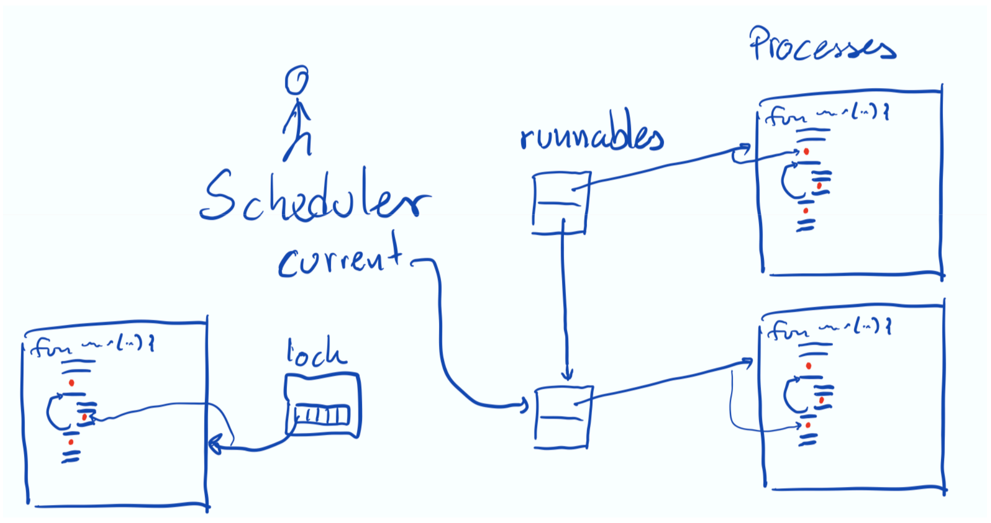
```
void Scheduler.run(){

    current := runnables.first()

    while (true) {

        execute(current, 50ms)

        current := runnables.next()

    }
}
```

This is *pseudo code*.

The `execute( current, 50ms )` will

- terminate after 50ms,
- or if the process itself pauses (locks, delays,...)

With locking



Notice, it is the Processes themself which move in and out of the locks.

*The fundamental objects like the runnable list, the locks, the processes, and scheduler is **designed together***

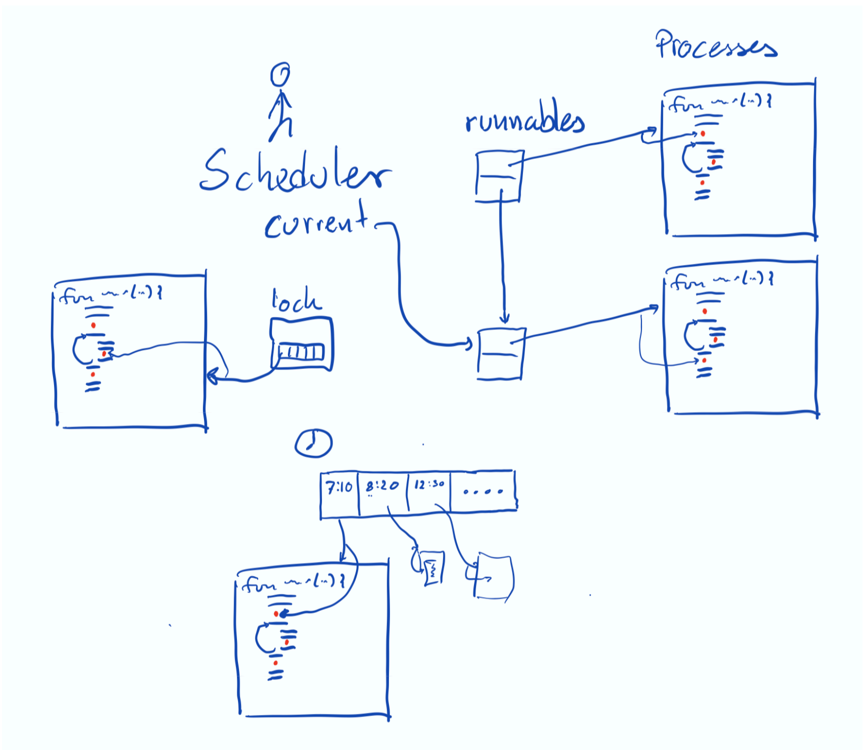
Locking code

```
// In some Process
myLock.lock()
// do safe stuff
myLock.release()
...
```

```
atomic void Lock.lock(){
    if ( this.closed() ) {
        this.queue.insert( Scheduler.current )
        Scheduler.runnables.remove( Scheduler.current )
        Scheduler.Pause()
    }
    this.close()
}
```

```
atomic void Lock.release(){
    if ( !this.lock.queue.empty() ){
        Scheduler.runnables.add( this.lock.queue.removeFirst() )
    }
}
```

With locking and timing



It is the current process itself which calls `sleep`.

Timing code

```
atomic void Process.sleep(int millies){
    timerQueue.insert(this, currentTime+millies)
    Scheduler.runnables.remove( Scheduler.current )
    Scheduler.pause()
}
```

The scheduler loop now looks like:

```
atomic  void Scheduler.run(){
    current := runnables.first()
    while (true) {
        execute(current, 50ms)
        if (currentTime >= timerQueue.first.time){
            current = timerQueue.removeFirst()
            Scheduler.runnables.add( current )
        } else {
            current := runnables.next()
        }

}   }
```

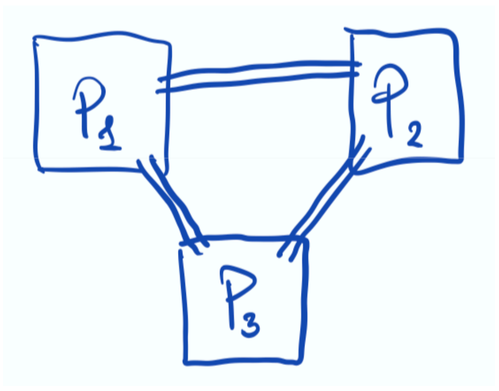
Why locking?

To ensure safe access to shared data (scheduling & visibility)

Any alternatives?

Indeed

Message based concurrency

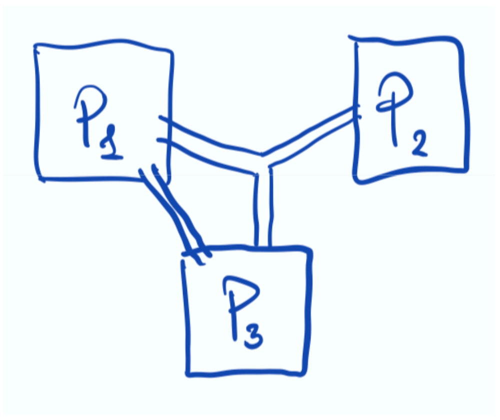


- Each process has its own state
- Can send/receive immuatable messages on channels
- Can wait for messages on one or more channels
- (In general it will be rare that all processes are connected to all others)

The **go programming language** use this as their model for concurrency/parallelism

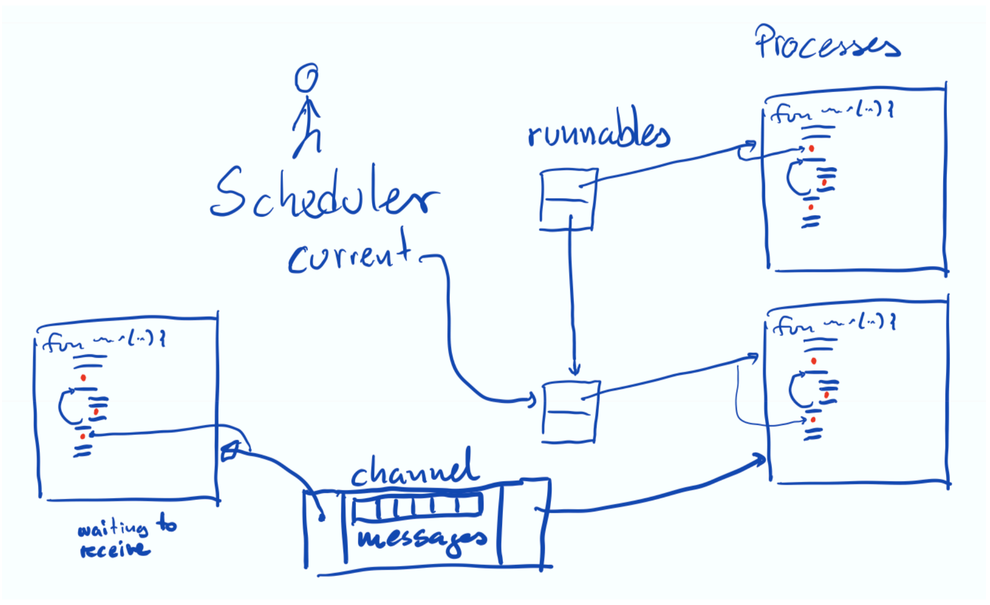
The idea trace back to Communicating sequential processes by Tony Hoare in 1978.

Message based variations



- a channel *might be* multiway (connect more than two processes)
    - a sent message *might be* received one or by all
- a channel *might have* direction
- a channel *might have* capacity
- waiting *might have* timeout

Message based scheduler

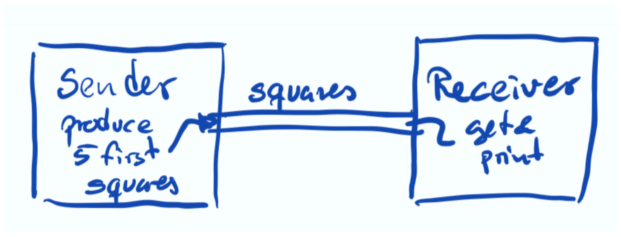


The left process is not runnable as it is waiting for some other process to send a message on the channel.

(There is only drawn one channel, many could exist. The channel is depicted as being able to store multiple messages, there is no explicit direction on this channel, no clock is associated with this wait)

Channels in kotlin

```
fun main() = runBlocking {
    val channel = Channel<Int>()
    launch { senderRoutine( channel ) }
    launch { receiverRoutine( channel ) }
    println("Coroutines started")
}
```

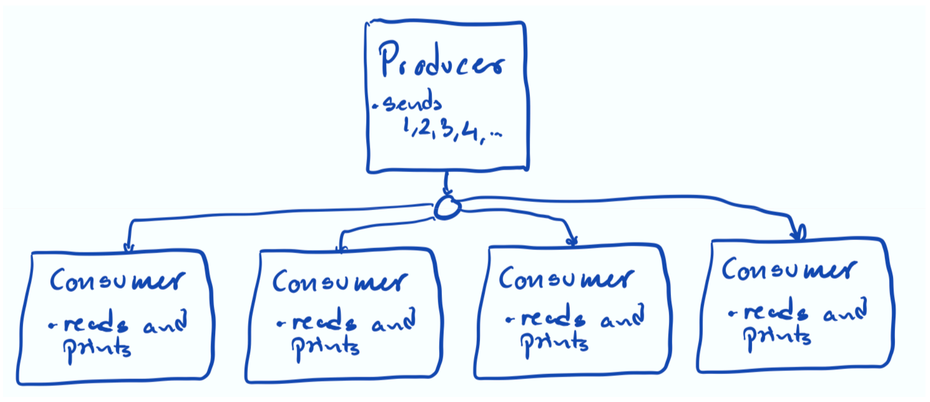


The code for the two coroutines

```
suspend fun senderRoutine(channel: Channel<Int>) {
    for (x in 1..5) {
            channel.send( x * x )
            println("Sent ${x*x}")
        }
        channel.send(-1) // we're done sending
        println("Done sending")
}
```

```
suspend fun receiverRoutine(channel: Channel<Int>) {
    while(true) {
        val square = channel.receive()
        if (square == -1) break
        println( "Received: $square" )
    }
    println("Recieved all!")
}
```

Fan-out



In the kotlin implementation, only one consumer will receive a sent message.

```
fun main() = runBlocking<Unit> {
    val channel = Channel<Int>()
    launch { produceNumbers( channel ) }
    repeat(5) { launch { consumer(it, channel) } }
    delay(1000) // main coroutine is waiting a second
    channel.cancel()
}
```

Code for producer and consumer

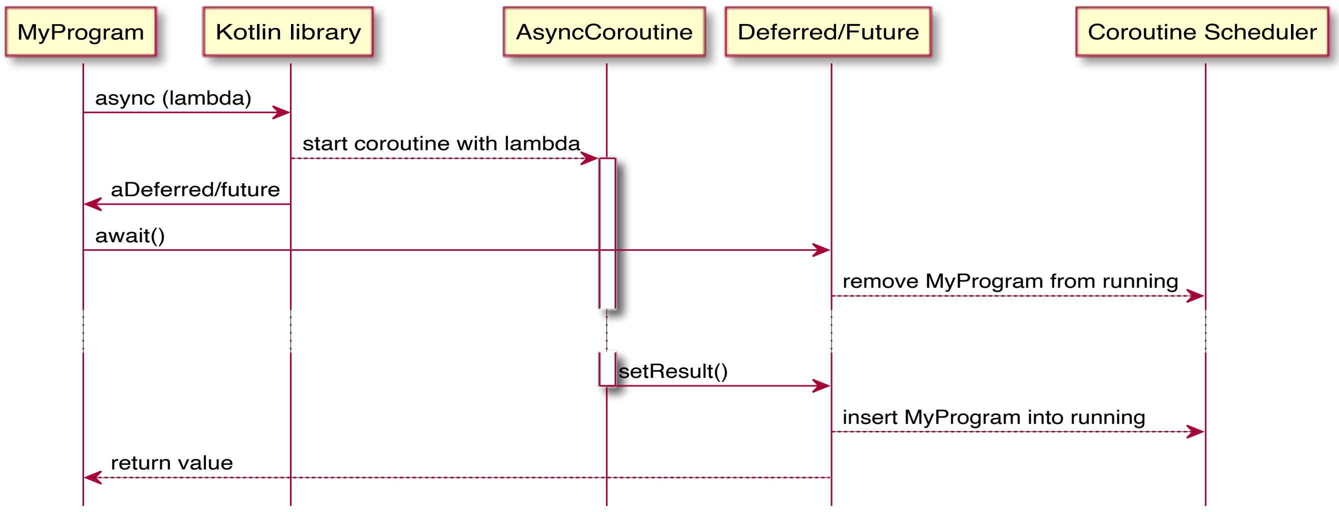
```
suspend fun produceNumbers(ch : Channel<Int>) {
    var x = 1 // start from 1
    while (true) {
        ch.send(x++) // produce next
        delay(100) // wait 0.1s
    }
}
```

```
suspend fun consumer(id: Int, channel: ReceiveChannel<Int>) {
    for (msg in channel) {  // reading from a channel
        println("#$id --> $msg")
    }
}
```

Async & await

```
val one = async { computation1() }
val two = async { computation2() }
res = one.await() + two.await()
```

- The function `async` launces a coroutine,
- returns a `Deferred` (Kotlin name for future/promise)
- The `await()` method on a deferred waits for the result to be ready



Library defined scheduler

In some languages it is possible to write a scheduler in the language itself.

```
Scheduler sch = new Scheduler()
sch.add( new Process() )
sch.add( new Process() )
sch.run()
```

Where `Scheduler` and `Process` are library defined classes (opposed to compiler defined).

Design options

1. Should `run` return a value
2. How should `Sheduler` work together with native `Threads`
3. Can you add `Process` after the scheduler has started?
4. Which *states* do a process have (running, runable, waiting, terminated,…)
5. How do you specify a Process (using Lambda or subclassing for example)
6. Can `Scheduler` be preemptive, or do `Process` have to be cooperative

Mapping coroutines to JVM threads

```
fun main() = runBlocking<Unit> {
    launch { printInfo("Parent     ") }
    launch(Dispatchers.Unconfined) { printInfo("Unconfined") }
    launch(Dispatchers.Default) { printInfo("Default    ") }
    launch(newSingleThreadContext("MyOwnThread")) {  printInfo("New        ") }
}
```

`Dispatchers` are schedulers that run coroutines.

- launch() - context of the parent, main runBlocking coroutine
- launch(Dispatchers.**Unconfined**) - main thread
- launch(Dispatchers.**Default**) - a CachedThreadPool
- launch(newSingleThreadContext("MyOwnThread")) - will get its own new thread

Link to code

Move coroutines between schedulers

```kotlin
fun log(msg: String) = println("[${Thread.currentThread().name}] $msg")

fun main() {
    val ctx1 = newSingleThreadContext("Ctx1")
    val ctx2 = newSingleThreadContext("Ctx2")
    runBlocking(ctx1) {
      log("Started in ctx1")
      withContext(ctx2) {
              log("Working in ctx2")
        }
      log("Back to ctx1")
    }
}
```

Code

OK, it is possible, but can that have any possible use????

Thread confinement of access to shared data

No need to sync if only one thread modifies data

If coroutines are run in thread-pool with true parallelism, bad things™ might happen.

The problem - in kotlin, but we saw this in first lesson

Sample from documentation

The fastest solution

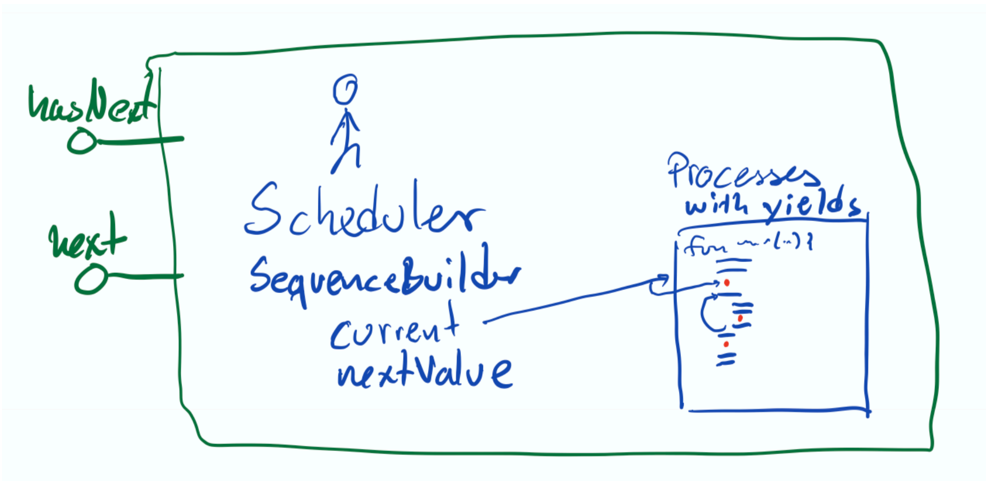
(because this has an easy solution)

The sheduling solution

A final word on yield

When the scheduler is an object (as opposed to a JVM concept), one can call methods on a scheduler.

The `sequence` function (where we use `yield`) is actually a scheduler with just one Process



hasNext
next
Scheduler
SequenceBuilder
current
nextValue
Processes with yields
fun ~~(..){