Having fun with Kotlin coroutines

A first tour of concurrency models in Kotlin

Alessandro Candolini

June 27, 2018

Agenda

1. We live in a concurrent world
2. Blocking vs non-blocking
3. Kotlin coroutines demystified
4. Coroutines-powered concurrency models

We live in a concurrent world

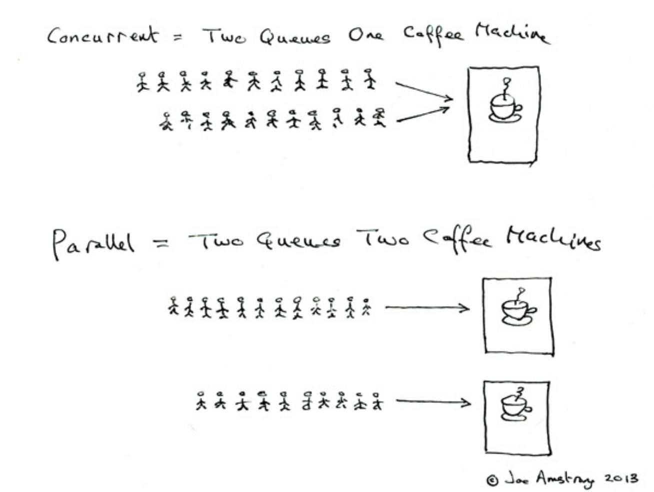


Figure 1: https://joearms.github.io/published/2013-04-05-concurrent-and-parallel-programming.html

Is concurrency relevant for mobile development?

- IO (e. g., network, etc)
- sensors (e. g., gps, etc)
- UI events
- platform lifecycle

Is concurrency relevant for mobile development?

- IO (e. g., network, etc)
- sensors (e. g., gps, etc)
- UI events
- platform lifecycle

Is concurrency relevant for mobile development?

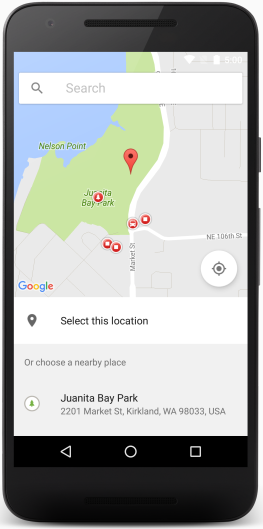
- IO (e. g., network, etc)
- sensors (e. g., gps, etc)
- UI events
- platform lifecycle

Is concurrency relevant for mobile development?

- IO (e. g., network, etc)
- sensors (e. g., gps, etc)
- UI events
- platform lifecycle

Is concurrency relevant for mobile development?

- IO (e. g., network, etc)
- sensors (e. g., gps, etc)
- UI events
- platform lifecycle



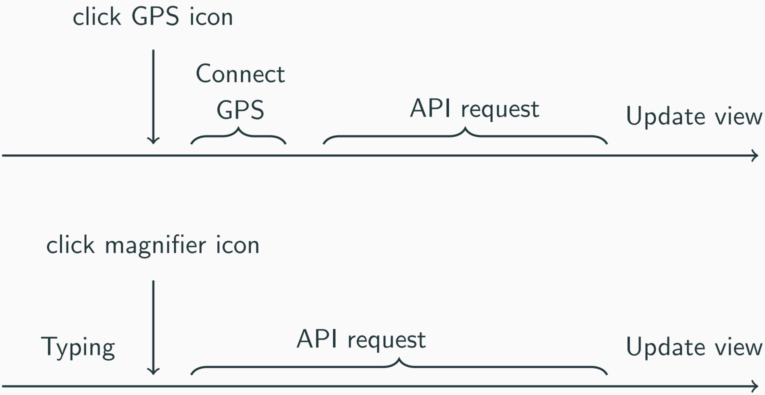
acceptance criteria:

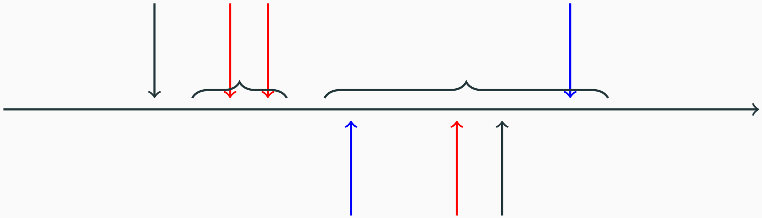
- search by current location
- search by location name

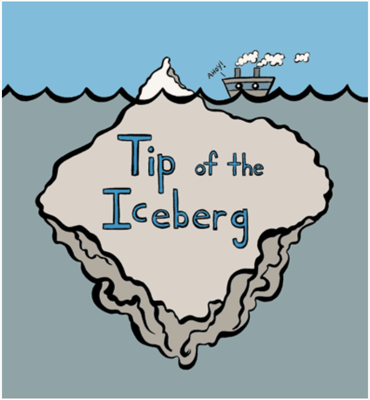
advanced

- search suggestions when tying

Translate ACs into code: simple *sequential* state machine (simplified)







- Delays
- User inputs
- Failures (connectivity, gps on mobile devices)
- Sort api responses by time
- *android/ios lifecycle*, etc

Naive approach: put constrains in place to restrict the combinatorial range of possible options

- Conditionally forbid user events (disable buttons, loading spinners, etc)
- Boolean flags
- Be defensive (if/else)
- Bind/unbind from lifecycle , etc

(Or more technical constrains like single thread executors, queues, synchronization, etc)

The approach

- doesn't scale.
    - pre-fatching
    - background upload
    - recovery/retry logic
    - debouncing, timeouts
    - no control on platform lifecycle
- can result in slower execution

Slows down the performance/experience, unlocking parallelism will make things worst

Key motivators:

- Scalability/performance (both server-side and client-side)
- Better user experience

“Concurrency is the composition of independently executing processes, typically functions, but they don’t have to be.”

“Parallelism is the simultaneous execution of multiple things, possibly related, possibly not.”

Rob Pike

Figure 2: `https://www.youtube.com/watch?v=cN_DpYBzKso&t=1061s`

Programming Techniques
S. L. Graham, R. L. Rivest Editors

Communicating Sequential Processes

C.A.R. Hoare
The Queen's University
Belfast, Northern Ireland

This paper suggests that input and output are basic primitives of programming and that parallel composition of communicating sequential processes is a fundamental program structuring method. When combined with a development of Dijkstra's guarded command, these concepts are surprisingly versatile. Their use is illustrated by sample solutions of a variety of familiar programming exercises.

Key Words and Phrases: programming, programming languages, programming primitives, program structures, parallel programming, concurrency, input, output, guarded commands, nondeterminacy, coroutines, procedures, multiple entries, multiple exits, classes, data representations, recursion, conditional critical regions, monitors, iterative arrays

CR Categories: 4.20, 4.22, 4.32

1. Introduction

Among the primitive concepts of computer programming, and of the high level languages in which programs are expressed, the action of assignment is familiar and well understood. In fact, any change of the internal state of a machine executing a program can be modeled as an assignment of a new value to some variable part of that machine. However, the operations of input and output, which affect the external environment of a machine, are not nearly so well understood. They are often added to a programming language only as an afterthought.

Among the structuring methods for computer programs, three basic constructs have received widespread recognition and use: A repetitive construct (e.g. the **while** loop), an alternative construct (e.g. the conditional **if..then..else**), and normal sequential program composition (often denoted by a semicolon). Less agreement has been reached about the design of other important program structures, and many suggestions have been made: Subroutines (Fortran), procedures (Algol 60 [15]), entries (PL/I), coroutines (UNIX [17]), classes (SIMULA 67 [5]), processes and monitors (Concurrent Pascal [2]), clusters (CLU [13]), forms (ALPHARD [19]), actors (Hewitt [1]).

The traditional stored program digital computer has been designed primarily for deterministic execution of a single sequential program. Where the desire for greater speed has led to the introduction of parallelism, every attempt has been made to disguise this fact from the programmer, either by hardware itself (as in the multiple function units of the CDC 6600) or by the software (as in an I/O control package, or a multiprogrammed operating system). However, developments of processor technology suggest that a multiprocessor machine, constructed from a number of similar self-contained processors (each with its own store), may become more powerful, capacious, reliable, and economical than a machine which is disguised as a monoprocessor.

In order to use such a machine effectively on a single task, the component processors must be able to communicate and to synchronize with each other. Many methods of achieving this have been proposed. A widely adopted method of communication is by inspection and updating of a common store (as in Algol 68 [18], PL/I, and many machine codes). However, this can create severe problems in the construction of correct programs and it may lead to expense (e.g. crossbar switches) and unreliability (e.g. glitches) in some technologies of hardware implementation. A greater variety of methods has been proposed for synchronization: semaphores [6], events (PL/I), conditional critical regions [10], monitors and queues (Concurrent Pascal [2]), and path expressions [3]. Most of these are demonstrably adequate for their purpose, but there is no widely recognized criterion for choosing between them.

This paper makes an ambitious attempt to find a single simple solution to all these problems. The essential proposals are:

(1) Dijkstra's guarded commands [8] are adopted (with a slight change of notation) as sequential control structures, and as the sole means of introducing and controlling nondeterminism.
(2) A parallel command, based on Dijkstra's *parbegin* [6], specifies concurrent execution of its constituent sequential commands (processes). All the processes start simultaneously, and the parallel command ends only when they are all finished. They may not communicate with each other by updating global variables.
(3) Simple forms of input and output command are introduced. They are used for communication between concurrent processes.

This research was supported by a Senior Fellowship of the Science Research Council.

Author's present address: Programming Research Group, 45, Banbury Road, Oxford, England.

Figure 3: Tony Hoare's seminal paper

“The most obvious application of the new ideas is to the specification, design, and implementation of computer systems which continuously act and interact with their environment. The basic idea is that these systems can be readily decomposed into subsystems which operate concurrently and interact with each other as well as with their common environment. The parallel composition of subsystems is as simple as the sequential composition of lines or statements in a conventional programming language.“

Tony Hoare (CSP book, 2015)

Concurrency

Two operations are concurrent if they are not ordered by *happens before* relation[1].

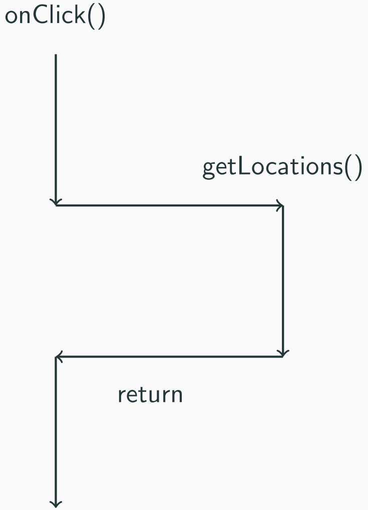
[1] Leslie Lamport's paper
`https://www.microsoft.com/en-us/research/uploads/prod/2016/12/Time-Clocks-and-the-Ordering-of-Events-in-a-Distributed-System.pdf`

Blocking vs non-blocking

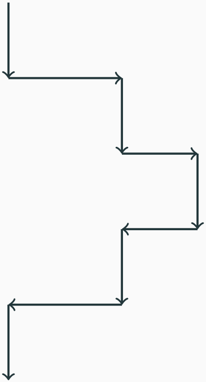
```kotlin
/** inject resource here */

fun onClick() {
  val position = gpsService.getPositionFromGps()
  val locations = apiService.getLocations(position)
  view.showLocations(locations)
}
```

```kotlin
typealias LatLng = Pair<Double, Double>

interface GpsService {
    fun getPositionFromGps() : LatLng
}

interface ApiService {
    fun getLocations(position : LatLng)
            : List<String>
}
```

```kotlin
fun getLocations(position : LatLng)
              : List<String> {

  /** Network request */
  Thread.sleep(3000)

  return listOf("etc")
}
```


onClick()
getLocations()
return

```kotlin
fun getLocations(position : LatLng)
             : List<String> {

  val data = networkCall()
  return listOf(data)
}
```



```kotlin
interface ApiService {

    fun getLocations(position : LatLng,
          callback : Callback): Unit

    interface Callback {
        fun onSuccess(locations : List<String>)
        fun onError(throwable : Throwable)
    }

}
```

Few preliminary troubles

- Unnatural contract (the output is represented via input)
- Don't chain nicely (callback hell, pyramid of doom, hadouken, etc)
- Error propagation, and ...

. . . so far they don't solve our problem

```
fun getLocations(position: LatLng,
         callback: ApiService.Callback) : Unit {
    Thread.sleep(3000)
    callback.onSuccess(listOf("etc"))
    return
}
```

```kotlin
@Inject
lateinit var executor : Executor

fun getLocations(position: LatLng,
            callback: ApiService.Callback) : Unit {
    executor.execute {
        Thread.sleep(3000)
        callback.onSuccess(listOf("etc"))
    }
    return
}
```

Question

How do callbacks work when the programming language is single-thread?

Now the *consumer* of the service is *not* blocked (it does not have to wait until completion).

However,

- we have to update the view on the UI thread,
- the thread running the runnable is blocked $\rightarrow$ thread pools, etc

Another option: Futures

```
interface ApiService {

    fun getLocations(position : LatLng):
      Future<List<String>>

}
```

Monadic chainability, error handling, etc (not in this talk :))

Kotlin coroutines demystified

How do we write code that waits
for something most of the time?

```kotlin
typealias LatLng = Pair<Double, Double>

interface GpsService {
    suspend fun getPositionFromGps() : LatLng
}

interface ApiService {
    suspend fun getLocations(position : LatLng)
            : List<String>
}
```

Questions:

- How to implement the body of the suspend function?
- How to invoke a suspend function?

This means respectively

- How does the corotuine know when to pause/suspend?
- How does the coroutine know what to execute when the suspend function resumes?

Coroutines-powered concurrency models

Disadvantages of Java locking:

- Hard to maintain and error prone
- Priority inversion
- Resumption can be expensive
- Wasting resources

- CSP (aka, channels)
- actors

Questions?