Asynchronous Programming & Kotlin Coroutines

Why Asynchronous Programming?

The Operating System is synchronous.

Every system call parks a Thread and waits for some action to complete before it unparks the thread and continues computation.

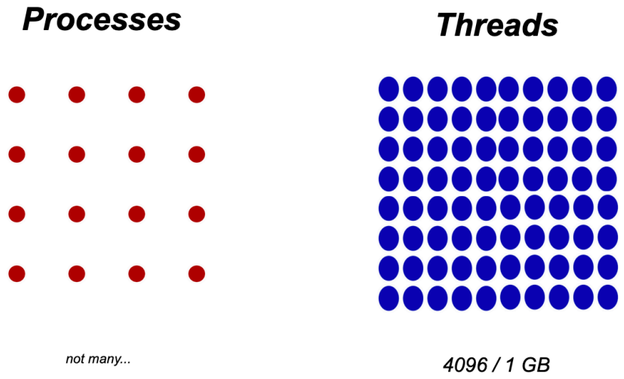
To perform multiple operations at the same time we can use different processes.

So one process can be waiting for a system call, while another can leverage the CPU to compute.

But a process is a **heavy** abstraction.

This is where threads come in, the operation system can start multiple computations at the same time within a single process through threads.

Threads are **cheaper** than processes, and they allow two separate threads to block at the same time.



Modern applications need to perform lots of tasks concurrently. So giving each task a thread is not scalable. What if we could perform system calls without taking up an entire thread, but instead pass the thread on to another task until our system call has finished.

This is what we call asynchronous programming.

Different styles of Asynchronous Programming

- Callbacks
- Futures / Promises
- Suspending functions

Callbacks

```
fun findProductByName(title: String, cb: (String) -> Unit) {
    client.get("/api/product?name={name}") { response -> cb(response.json().get("name")) }
}
```

🔥Callback Hell 🔥

```
fun findProductByName(title: String, cb: (String) -> Unit) {
    client.get("/api/product?name={name}") { response ->
        client.get("/api/product/{(response.json().get("id"))}") { product ->
            cb.apply(product)
        }
    }
}
```

Future / Promises

- Help with composition of multiple asynchronous operations
- Many different libraries on the JVM:
  - Standard library has CompletableFuture
  - Reactiv libraries: RxJava, Reactor, Akka Streams

Future / Promises

```
fun findProductByName(name: String): CompletableFuture<String> {
    client.get("/api/product?name={name}")
        .thenCompose { id -> client.get("/api/product/{id}") }
        .thenApply { product -> product.get("name") }
}
```

Combinator Jungle

The *business intent* of my code gets lost in all the 'combinator jungle'

Kotlin Suspend

Kotlin Suspend

```
suspend fun findProductByName(name: String) {
    val response = client.get("/api/product?name={name}")
    val product = client.get("/api/product/{response.get("id")}")
    return product
}
```

The heart of coroutines are *suspend functions* marked with the `suspend` keyword

They look like 'normal functions' but can be *paused* and *resumed at a later time* by a Coroutine *without blocking*.

```
suspend fun delay(timeMillis: Long): Unit = ...
```

Suspend functions can only be called by another *suspending function* or a *Coroutine*

```
fun main() {
  delay(1000)
}
```

Compilation Error

```
launch {
    delay(1000)
}
```

```
suspend fun easyOn() {
    delay(1000)
}
```

At it's heard **suspend** methods are about returning from a method and later re-entering, without losing it's scope.

This is not something 'normal' methods in Java allow us to do.

We can return once, and there is no way to re-enter the method at a later stage.

Suspend Functions Under the Hood

```
suspend fun delay(timeMillis: long): Unit

suspend fun delay(timeMillis: long, callback: Continuation<Unit>): Unit

interface Continuation<in T> {
  val context: CoroutineContext
  fun resume(value: T)
  fun resumeWithException(exception: Throwable)
}
```

Suspend Functions Under the Hood

Suspend functions use callbacks under the hood. This is why Kotlin can run on the JVM, it uses the same concurrency primitive available to Java.

From Suspend to Coroutine

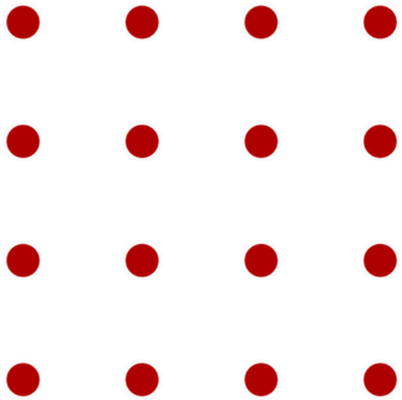
Threads

```
thread {
    sleep(500)
    println("Rabbit")
}
println("Turtle")
thread.join()
```

Coroutines

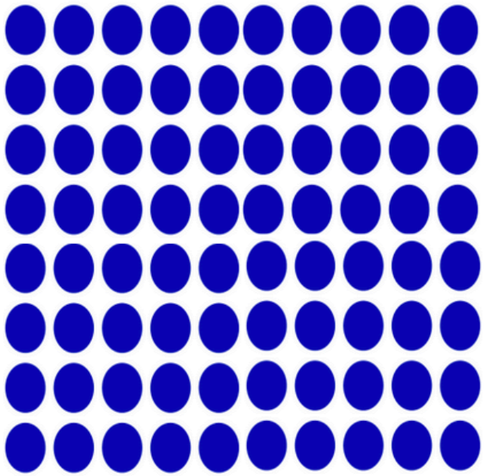
```
runBlocking {
    launch {
        delay(500)
        println("Rabbit")
    }
    println("Turtle")
}
```

Processes



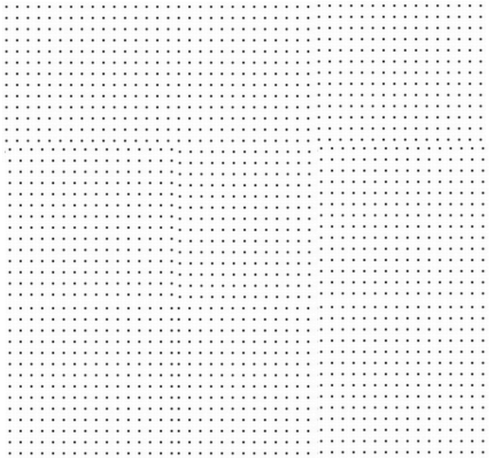
not many...

Threads



4096 / 1 GB

Coroutines



2,4 million / 1 GB