

# Android programming

## TX00CK66 Sensor Based Mobile

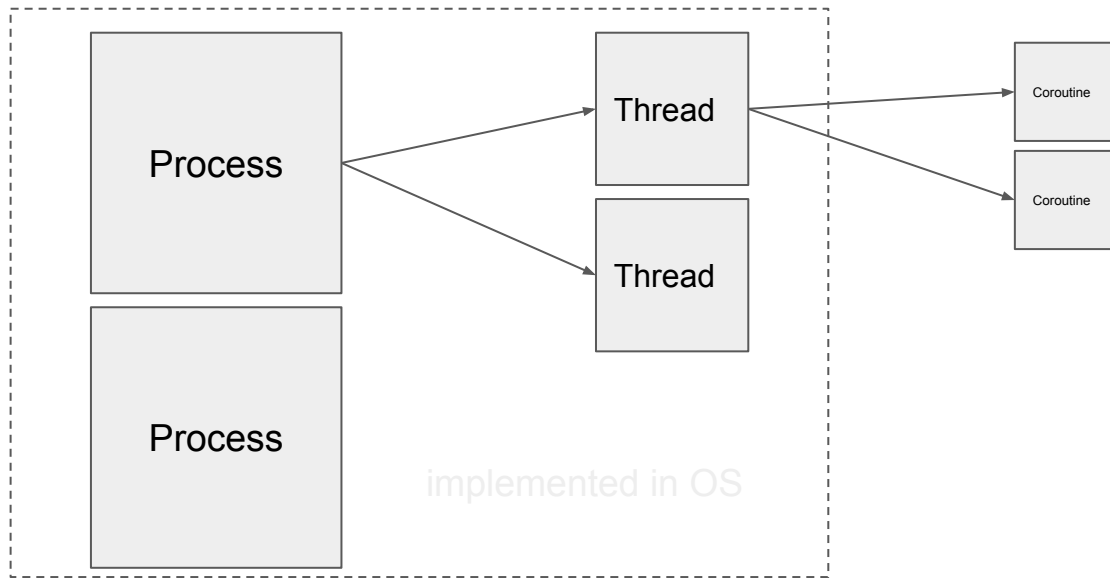
### Applications

Lecture Coroutines - 17.01.2022

Jarkko.Vuori@metropolia.fi

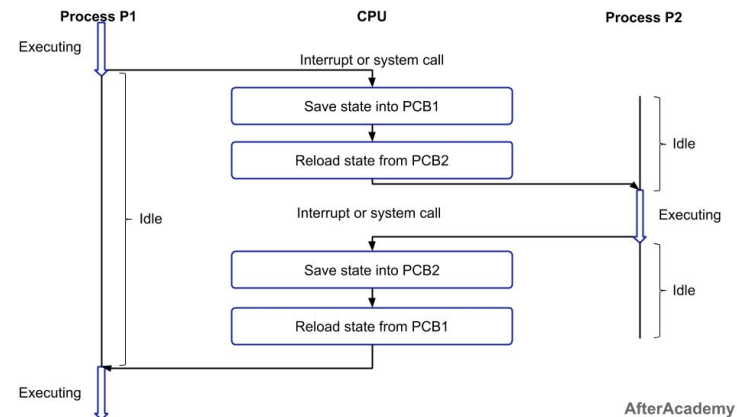
# Introduction

- Multitasking (doing things concurrent) can be modelled using
  - processes
  - threads
  - coroutines



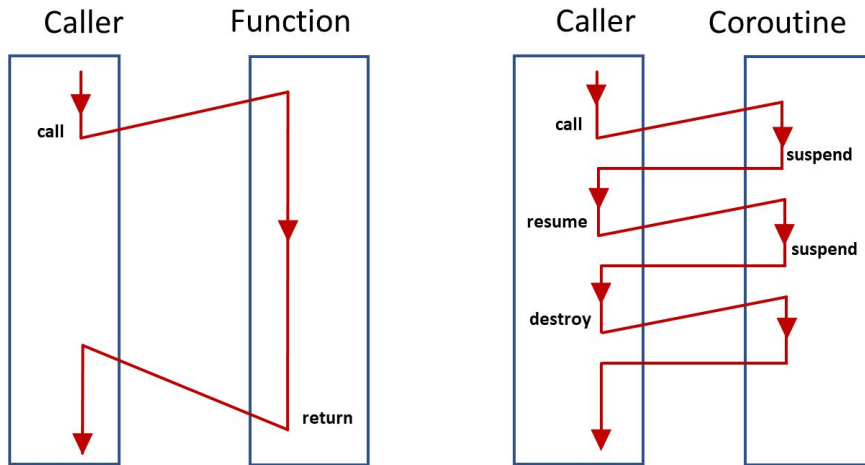
# Multitasking

- With only one execution unit (e.g. microprocessor), multitasking must be done by task swapping
  - executing P1 for a while, switching to P2 and executing it for a while, switching back to P1 and executing it for a while etc.
- Task swap can be time consuming operation if the task context (memory mapping, processor registers etc) is large
  - $\text{task\_swap\_time} / \text{task\_running\_time}$  is called overhead of the multitasking
- Processes are totally independent execution units (totally protected)
  - therefore task switching between processes is a long operation → large overhead when tasks are relatively simple
- Threads have same memory areas, only stacks (e.g. local variables of functions) are different (but in the same memory area, i.e. not protected from each other)
  - therefore task switching between threads is a shorter operation
  - generally in a threaded environment you have at most 30-50 threads before the amount of overhead wasted actually scheduling these threads (by the system scheduler) significantly cuts into the amount of time the threads actually do useful work



# Coroutine

- Coroutines or co-routines (sometimes called as Fibers, lightweight threads, and green threads)
  - They provide very high level of concurrency with very little overhead
    - because no separate stack is needed (as opposed to threads)
  - Coroutine can still do concurrency without overhead - it simply manages the context-switching itself
    - For example, if you have a routine doing some work and it performs an operation you know will block for some time (i.e. a network request), with a co-routine (coroutine) you can immediately switch to another routine without the overhead of including the system scheduler in this decision (you, the programmer must specify when coroutines can switch)
  - Coroutines implements concurrency, not parallelism
- A coroutine is a function which:
  - can suspend its execution (the expression where it suspends is called suspension point)
  - can be resumed from suspension point (keeping its original arguments and local variables)
- Switching between coroutines can be done without the help of operating system
  - therefore they are so efficient
- First high-level programming language implementations of Coroutines were in Simula 67 and Scheme in 1972



While you can only call a **function** and return from it, you can call a **coroutine**, suspend and resume it, and destroy a suspended coroutine

# Kotlin coroutine

- Kotlin implementation of coroutine is build around the suspend function
  - This kind of function will execute, pause execution and resume at some point in time
  - This kind of function can be suspended without blocking the current thread. Instead of returning a simple value, it also knows in which context the caller suspended it. Using this, it can resume appropriately, when ready
  - The suspend name is a little misleading; this function is able to suspending the coroutine who is calling it
- The function signature remains exactly the same. The only difference is suspend being added to it. The return type however is the type we want to be returned
- The code is still written as if we were writing synchronous code, top-down, without the need of any special syntax, beyond the use of a function called launch which essentially kicks-off the coroutine
- The programming model and APIs remain the same. We can continue to use loops, exception handling, etc. and there's no need to learn a complete set of new APIs
- It is platform independent. Whether we targeting JVM, JavaScript or any other platform, the code we write is the same. Under the covers the compiler takes care of adapting it to each platform
  - Because coroutines need no OS support

```
companion object {
    private val parentJob = Job()
    private val coroutineScope = CoroutineScope(Dispatchers.Main + parentJob)
    private val secondScope = CoroutineScope(Dispatchers.IO + parentJob)
}

coroutineScope.launch {
    val result1 = secondScope.async { doCalculation1() }
    val result2 = secondScope.async { doCalculation2() }
    Log.i("DBG", "Counting in progress")
    showCalculation( result1.await(), result2.await())
}

private suspend fun doCalculation1(): Int {
    Log.i("DBG", "doCalculation1 started")
    delay( 5000);
    Log.i("DBG", "doCalculation1 ended")

    return 42
}

private suspend fun doCalculation2(): Int {
    Log.i("DBG", "doCalculation2 started")
    delay( 3000);
    Log.i("DBG", "doCalculation2 ended")

    return 43
}
```

```
19:25:20.698 I/DBG: doCalculation1 started
19:25:20.699 I/DBG: Counting in progress
19:25:20.701 I/DBG: doCalculation2 started
19:25:23.725 I/DBG: doCalculation2 ended
19:25:25.731 I/DBG: doCalculation1 ended
19:25:25.733 D/DBG: Result is 42 and 43
```

# Kotlin coroutine

- Because control of the coroutine is able to switch to an another coroutine (under the same thread) only at special points (like in `delay()` call), coroutine may block the thread for a long time if it is doing time consuming tasks
  - Because threads are not scheduled by the operating system (they are not pre-emptive)
- Calculation of the mathematical problem on the right blocks the execution of the thread until the calculation is ready
  - After the computation task we will call the `delay()` function, and control is able to continue after the first `launch()` operation
  - `yield()` function may be used to give other coroutines (under the same thread) execution time

```
btLaunch.setOnClickListener{
    GlobalScope.launch(context = Dispatchers.Main) {
        Log.i("DBG", "launched coroutine 1")
        // Here we spend 1.2s in
        // heavy mathematical calculation
        var y = 0.0
        var x = -1.0
        repeat(10000000) {
            y += Math.exp(x)
            x /= 2
            // yield()
        }
        Log.i("DBG", "Answer is $y")
        delay(5000)
        Log.i("DBG", "Here after a delay of 5 seconds")
    }

    GlobalScope.launch(context = Dispatchers.Main) {
        Log.i("DBG", "launched coroutine 2")
    }
}
```

```
15:18:39.107 8035-8035/net.homeip.kotilabra.koroutine I/DBG: launched coroutine 1
15:18:40.325 8035-8035/net.homeip.kotilabra.koroutine I/DBG: Answer is 9999998.51326612
15:18:40.326 8035-8035/net.homeip.kotilabra.koroutine I/DBG: launched coroutine 2
15:18:45.332 8035-8035/net.homeip.kotilabra.koroutine I/DBG: Here after a delay of 5 seconds
```

# Coroutine concepts

- CoroutineBuilders: These take a suspending lambda as an argument to create a coroutine
  - There are two main of coroutine builders provided by Kotlin Coroutines: `async()` and `launch()`
- CoroutineScope: Helps to define the lifecycle of Kotlin Coroutines
  - It can be application-wide or bound to a component like the Android Activity
  - You have to use a scope to start a coroutine
- Coroutine Job: As the name suggests, a Job represents a piece of work that needs to be done
  - Additionally, every Job can be cancelled, finishing its execution
    - Because of that, it has a lifecycle and can also have nested children
  - You can use Job to cancel and clear up all coroutines which you launched inside that Job
- CoroutineDispatcher: Defines thread pools to launch your Kotlin Coroutines in
  - This could be the background thread pool, main thread or even your custom thread pool
  - You'll use this to switch between, and return results from, threads

# Coroutine Dispatcher

- Dispatchers.Default: CPU-intensive work, such as sorting large lists, doing complex calculations and similar
  - A shared pool of threads on the JVM backs it
- Dispatchers.IO: networking or reading and writing from files
  - In short – any input and output, as the name states
- Dispatchers.Main: recommended dispatcher for performing UI-related events
  - For example, showing lists in a RecyclerView, updating Views and so on



# Coroutine Builders: Launch() and Async()

- Fire and forget – no response expected:
  - launch()
- Response expected from a single task
  - launch() + withContext()
- Response expected from several parallel tasks
  - async() + await()

```
btLaunch.setOnClickListener {  
    GlobalScope.launch(context = Dispatchers.Main) {  
        Log.i("DBG", "launched coroutine 1")  
        //Thread.sleep(5000)  
        delay(5000)  
        Log.i("DBG", "Here after a delay of 5 seconds")  
    }  
  
    GlobalScope.launch(context = Dispatchers.Main) {  
        Log.i("DBG", "launched coroutine 2")  
    }  
}
```

```
20:01:10.012 I/DBG: launched coroutine 1  
20:01:10.013 I/DBG: launched coroutine 2  
20:01:15.018 I/DBG: Here after a delay of 5 seconds
```

# Example – launch()

```
override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
    setContentView(R.layout.activity_main)  
    GlobalScope.launch(Dispatchers.IO) {  
        notifyServer("message")  
    }  
}  
  
private suspend fun notifyServer(msg: String){  
    ...  
}
```

Application level scope - coroutine launched in this scope is canceled when the application is destroyed

notifyServer executed on a thread from I/O thread pool

Suspending function

Coroutine builder - takes a suspending lambda as an argument to create a coroutine

# Example – launch() + Job

```
lateinit var job: Job
```

```
override fun onCreate(savedInstanceState: Bundle?)  
{
```

```
    super.onCreate(savedInstanceState)  
    setContentView(R.layout.activity_main)  
    job = GlobalScope.launch(Dispatchers.IO) {  
        notifyServer("message")  
    }  
}
```

Coroutine builders return a Job  
(=handler to coroutine)

Job class has cancel method that  
can be invoked to cancel running  
coroutine


```
override fun onDestroy() {  
    job.cancel()  
    super.onDestroy()  
}
```

```
private suspend fun notifyServer(msg: String){  
    ...  
}
```

# Example – launch() + withContext()

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    GlobalScope.launch(Dispatchers.IO) {
        val resultOne = withContext(Dispatchers.Default) {
            requestServer("message")
        }
    }
}

private suspend fun requestServer(msg: String): Int {
    ...
    return x
}
```



# Example – async() + await()

```
runBlocking {  
    val deferredResults = arrayListOf<Deferred<Int>>()  
  
    deferredResults += async {  
        doCalculation1(2)  
    }  
  
    deferredResults += async {  
        doCalculation2(2)  
    }  
  
    deferredResults += async {  
        doCalculation3(2)  
    }  
  
    // wait for all results (at this point tasks are running)  
    val results = deferredResults.map { it.await() }  
    Log.i("DBG", "results: $results")  
}
```

# Reading list

- <https://kotlinlang.org/docs/coroutines-guide.html>
- <https://www.baeldung.com/kotlin/kotlin-threads-coroutines>
- <https://dmitrykandalov.com/coroutines-as-threads>
- <https://stackoverflow.com/questions/47871868/what-does-the-suspend-function-mean-in-a-kotlin-coroutine>
- <https://medium.com/l-r-engineering/launching-kotlin-coroutines-in-android-coroutine-scope-context-800d280ebd80>