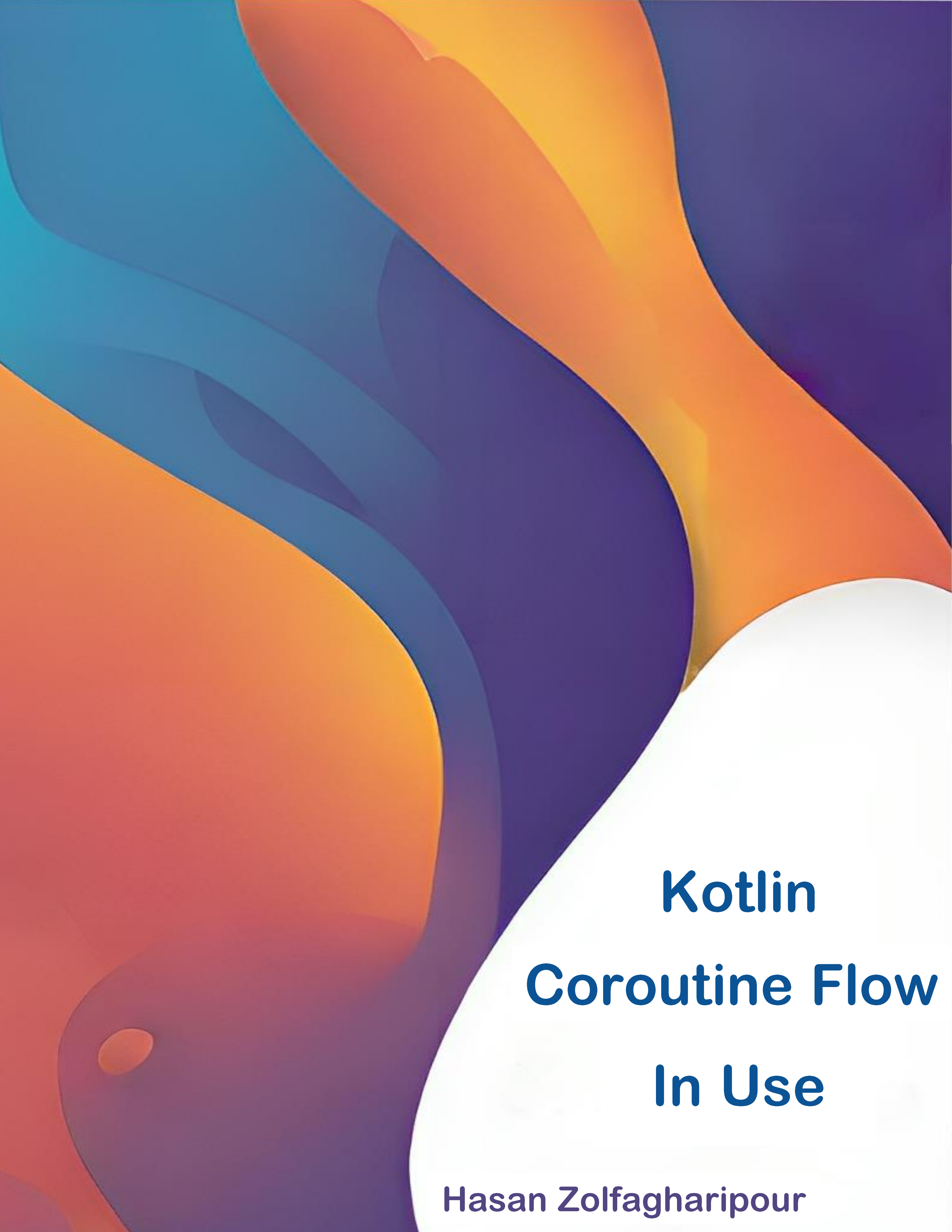


Kotlin Coroutine Flow In Use

Hasan Zolfagharipour

About this book

This is a comprehensive guide that takes you on an immersive journey into the world of Android development and Kotlin programming. With over 350 pages of carefully crafted content, this book is your ultimate companion for mastering the art of asynchronous programming using Coroutine and Flow. Building responsive and efficient applications is paramount in today's fast-paced mobile app development.

This book equips you with the knowledge and practical skills required to harness the power of Kotlin Coroutine and Flow to create smooth, responsive, and resource-efficient Android applications. Drawing from a wealth of authoritative sources, including articles, official documents, and expert-authored books, 'Coroutine-Flow In Use' distills the most essential concepts into a single, accessible resource. The book combines theoretical insights with real-world examples, providing a holistic understanding of using Coroutines and Flow effectively. Whether you're a new programmer or an experienced developer seeking to refine your skills, this book accommodates all proficiency levels. You'll explore foundational concepts, advanced techniques, and discover best practices. 'Coroutine-Flow In Use' is your gateway to building Android applications that are not just functional, but also highly responsive and user-friendly.

Join us on this journey and unlock the full potential of Kotlin Coroutines and Flow for your Android development projects. At the end of the book, we invite you to provide feedback, helping us improve future editions. Your insights are invaluable in our mission to deliver the most relevant and helpful content.

About Author

Hasan Zolfagharipour is an Android developer with 3+ years of experience developing cutting-edge Android applications. In his free time, Hasan likes to do volunteer work, read a book, be in nature, or go to the gym.

Contact me: Gmail, LinkedIn

Contents

1 Coroutine 1

1.1 Threads 1

1.1.1 Concurrency & Parallelism 3

1.1.2 Concurrency is Not Parallelism 3

1.1.2.1 The Gopher example 4

1.1.2.2 Coroutines in terms of concurrency and parallelism 7

1.2 Asynchronous work with threads 9

1.3 Coroutines 11

1.4 suspend 12

1.4.1 Suspend under the hood 12

1.5 Delay 17

1.5.1 awaitCancellation 18

1.5.2 Delay under the hood 18

1.6 Dispatchers 20

1.6.1 Blocking threads 21

1.6.2 Default dispatcher 22

1.6.2.1 Limiting the default dispatcher 23

1.6.3 IO dispatcher 23

1.6.3.1 Limiting the IO dispatcher 24

1.6.4 Main dispatcher 24

1.6.4.1 Immediate main dispatching 24

1.6.5 Unconfined dispatcher 25

1.6.6 Performance of dispatchers against different tasks 26

1.7 withContext 27

1.7.1 NonCancellable 27

1.7.2 Performance of *withContext* 28

1.7.3 Main-safety and withContext 29
1.8 Coroutine builders 31
1.8.1 runBlocking 31
1.8.2 launch 32
1.8.3 async 33
1.9 CoroutineStart 34
1.10 CoroutineContext 36
1.10.1 Contexts 38
1.10.1.1 EmptyCoroutineContext 38
1.10.1.2 Context in suspend function 38
1.10.2 Jobs 39
1.10.2.1 Job states 39
1.10.2.2 Job 40
1.10.2.3 SupervisorJob 41
1.10.2.4 Deferred 42
1.10.2.5 parent-child relation 43
1.10.3 Properties and methods 46
1.10.3.1 children 46
1.10.3.2 isActive, isCancelled, isCompleted 47
1.10.3.3 join 48
1.10.3.4 onJoin 48
1.10.3.5 cancel 48
1.10.3.6 cancelAndJoin 49
1.10.3.7 cancelChildren 49
1.10.3.8 invokeOnCompletion 49
1.10.3.9 start 50
1.10.3.10 ensureActive 50
1.11 Structured Concurrency 51

1.11.1 LaunchedEffect..53
1.12 Cancellation..55
1.12.1 Catching exceptions..56
1.12.1.1 Catching exceptions in async builder..58
1.12.2 Handling exceptions..59
1.12.2.1 SupervisorJob..59
1.12.2.2 supervisorScope..60
1.12.3 Cancellation under the hood..62
1.13 CoroutineScope..66
1.13.1 coroutineScope..66
1.13.1.1 supervisorScope..67
1.13.1.2 withTimeout..67
1.13.2 GlobalScope..68
1.13.3 CoroutineScope..69
1.13.4 MainScope..70
1.13.5 rememberCoroutineScope..70
1.13.6 LaunchedEffect..71
1.13.7 lifecycle-aware scopes..73
1.13.7.1 LiveDataScope..73
1.13.7.2 viewModelScope..76
1.14 lifecycleScope..79
1.14.1 Flow collection..80
1.14.2 Restartable Coroutines..82
1.14.2.1 repeatOnLifecycle..83
1.14.2.2 flowWithLifecycle..84
1.14.2.3 collectAsStateWithLifecycle..86
1.14.3 ProcessLifecycleOwner..90
1.14.4 lifecycleScope under the hood..90

1.15 Test 91

1.15.1 Overview 93

1.15.2 TestDispatchers 93

1.15.2.1 StandardTestDispatcher 94

1.15.2.2 UnconfinedTestDispatcher 96

1.15.3 Injecting test dispatchers 99

1.15.4 Setting the Main dispatcher 99

1.15.5 Creating dispatchers outside a test 101

1.15.6 Creating your own TestScope 102

1.16 Coroutine solutions 103

1.16.1 Convert callback-based APIs to Coroutine 103

1.16.2 Coroutines that outlive the consumer 105

1.16.3 Shared mutable state 108

1.16.3.1 Changing the problem 109

1.16.3.2 Changing is not possible 109

1.16.3.3 Understanding the problem 111

1.16.3.4 Atomic 114

1.16.3.5 Read/Write Locks 116

1.16.3.6 Thread confinement 116

1.16.3.7 Mutex 118

1.16.3.8 Actors 121

1.16.4 One shot request 121

1.16.4.1 Disable the button 122

1.16.4.2 Cancel the previous work 123

1.16.4.3 Queue the next work 124

1.16.4.4 Join previous work 124

1.17 Coroutines best practices 125

2 Flow 128

2.1 Introduction 128
2.1.1 Why Flow 128
2.1.2 Sequentiality...131
2.1.3 Cold & Hot stream 133
2.1.4 Flow constraints 135
2.1.4.1 Context preservation 135
2.1.4.2 Exception transparency 136
2.2 Flow builders...137
2.2.1 emptyFlow..137
2.2.2 flow 137
2.2.3 flowOf 138
2.2.4 asFlow 139
2.2.5 channelFlow...140
2.2.6 callbackFlow 141
2.2.7 MutableStateFlow & MutableSharedFlow...141
2.2.8 snapshotFlow 141
2.3 HotFlow 142
2.3.1 SharedFlow 142
2.3.1.1 Read-only SharedFlow 143
2.3.1.2 MutableSharedFlow parameters.. 144
2.3.1.3 SharedFlow vs BroadcastChannel... 153
2.3.2 StateFlow 154
2.3.2.1 Read-only StateFlow 155
2.3.2.2 Conflation on StateFlow 156
2.3.2.3 StateFlow or SharedFlow 158
2.3.2.4 StateFlow as a substituting of LiveData 158
2.3.2.5 StateFlow support in data binding.. 159
2.3.2.6 StateFlow vs ConflatedBroadcastChannel 159

2.3.3 stateIn & shareIn....160
2.3.3.1 stateIn & shareIn parameters.... 162
2.3.3.2 Custom SharingStarted.... 169
2.3.4 Hot *Flow* never completes....174
2.4 Terminal operators....174
2.4.1 Collect....175
2.4.1.1 collect { ... }.... 175
2.4.1.2 collect().... 176
2.4.1.3 collectIndex.... 177
2.4.1.4 collectLatest.... 178
2.4.1.5 collectAsState.... 180
2.4.1.6 collectAsStateWithLifecycle.... 180
2.4.1.7 collectWhile.... 181
2.4.2 launchIn....181
2.4.3 toCollection....182
2.4.4 toList....183
2.4.5 toSet....184
2.4.6 First....184
2.4.6.1 first.... 184
2.4.6.2 first(predicate).... 185
2.4.6.3 firstOrNull.... 186
2.4.6.4 firstOrNull(predicate).... 187
2.4.7 Last....188
2.4.7.1 last.... 188
2.4.7.2 lastOrNull.... 189
2.4.8 Single....190
2.4.8.1 single.... 190
2.4.8.2 singleOrNull.... 191

2.4.9 reduce 192
2.4.10 fold 193
2.4.11 Count 194
2.4.11.1 count 194
2.4.11.2 count(predicate) 195
2.4.12 produceIn 195
2.5 Emitting 196
2.5.1 emit 196
2.5.2 tryEmit 199
2.5.3 value 202
2.5.4 emitAll 202
2.5.5 compareAndSet 204
2.5.5.1 update 204
2.5.5.2 updateAndGet 205
2.5.5.3 getAndUpdate 205
2.6 Intermediate operators 206
2.6.1 catch 206
2.6.2 onCompletion 209
2.6.3 onStart 212
2.6.4 onSubscription 213
2.6.5 onEach 215
2.6.6 onEmpty 216
2.6.7 cancellable 216
2.6.8 flowOn 220
2.6.9 buffer 221
2.6.10 conflate 228
2.6.11 Retry 230

2.6.11.1 retryWhen 230
2.6.11.2 retry 232
2.6.12 Transform 234
2.6.12.1 transform 234
2.6.12.2 transformLatest 234
2.6.12.3 transformWhile 235
2.6.13 Filter 237
2.6.13.1 filter 237
2.6.13.2 filterNot 237
2.6.13.3 filterNotNull 238
2.6.13.4 filterInstance 238
2.6.14 Map 239
2.6.14.1 map 239
2.6.14.2 mapLatest 240
2.6.14.3 mapNotNull 241
2.6.15 Drop 242
2.6.15.1 drop 242
2.6.15.2 dropWhile 242
2.6.16 Take 244
2.6.16.1 take 244
2.6.16.2 takeWhile 245
2.6.17 Reduce, Fold & Scan 246
2.6.17.1 runningReduce 246
2.6.17.2 runningFold 247
2.6.17.3 scan 248
2.6.18 sample 248
2.6.19 timeout 249
2.6.20 withIndex 249

2.6.21 Debounce..250

2.6.22 Distinct...252

2.6.23 flattenConcat 257

2.6.24 flatMapConcat..258

2.6.25 flatMapLatest...260

2.6.26 flattenMerge 264

2.6.27 flatMapMerge...265

2.6.28 merge 268

2.6.29 zip 269

2.6.30 combine...270

2.6.31 combineTransform 275

2.6.32 flowWithLifecycle 281

2.7 Test...281

2.7.1.1 Test the consumer of a flow 281

2.7.1.2 Test the producer flow 282

2.7.1.3 Turbine 283

2.7.1.4 Testing StateFlows 285

2.8 Flow solutions..287

2.8.1 Convert callback-based APIs to flow...287

2.8.1.1 channelFlow 290

2.8.2 ViewModel One-off event...294

3 Channel 296

3.1 How do Channels work?...296

3.1.1 Fan-out...297

3.1.2 Fan-in...299

3.2 How to create channels? 299

3.2.1 Factory builder...300

3.2.1.1 Capacity.. 300

3.2.1.2 onBufferOverflow 301

3.2.1.3 onUndeliveredElement 302

3.2.2 produce builder..302

3.3 Channel types...304

3.3.1 SendChannel 304

3.3.2 ReceiveChannel 304

3.3.3 Broadcasting...305

3.3.3.1 BroadcastChannel 306

3.3.3.2 ConflatedBroadcastChannel 308

3.3.4 Ticker channel...310

3.4 Channel Functions 311

3.4.1 send 311

3.4.2 trySend 312

3.4.3 trySendBlocking 312

3.4.4 isClosedForSend..314

3.4.5 onSend 315

3.4.6 receive..315

3.4.7 tryReceive 315

3.4.8 receiveCatching..316

3.4.9 receiveAsFlow 317

3.4.10 isClosedForReceive 319

3.4.11 OnReceive...320

3.4.12 onReceiveCatching 320

3.4.13 close 320

3.4.14 invokeOnClose 323

3.4.15 cancel..323

3.4.16 isEmpty....324
3.4.17 consume....324
3.4.17.1 BroadcastChannel.consume.... 324
3.4.17.2 ReceiveChannel.consume.... 325
3.4.18 consumeEach....328
3.4.18.1 BroadcastChannel.consumeEach.... 328
3.4.18.2 ReceiveChannel.consumeEach.... 329
3.4.19 consumeAsFlow....330
3.4.20 Iterator....331
3.4.21 toList....332
3.4.22 Select....332
3.4.22.1 Support select methods.... 333
3.4.23 broadcast....339
3.4.23.1 CoroutineScope.broadcast.... 339
3.4.23.2 ReceiveChannel.broadcast.... 341
4 References....343

1 Coroutine

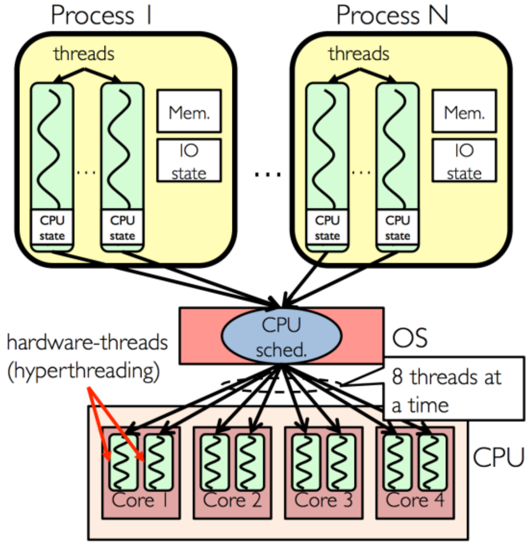
Coroutines are computations that run on top of threads and can be suspended. A coroutine is an instance of suspendable computation. Coroutines is a recommended solution for asynchronous programming on Android. Noteworthy features include the following:

- **Lightweight:** You can run many coroutines on a single thread due to support for suspension, which doesn't block the thread where the coroutine is running.
- **Fewer memory leaks:** Using structured concurrency to run operations within a scope.
- **Built-in cancellation support:** Cancellation is automatically propagated through the coroutine hierarchy.
- **Jetpack integration:** Many Jetpack libraries include extensions that support full coroutines.

Conceptually a Coroutine is like a Thread, it takes a block of code to run that works concurrently with the rest of the code.

1.1 Threads

A Thread is a very lightweight process, or the smallest part, that allows a program to operate more efficiently by running multiple tasks simultaneously.



Considering the above image, all process threads live in the same memory space, whereas processes have separate memory space. So, sharing objects between threads is more accessible, as they share the same memory space. To achieve the same between processes, we must use some IPC (inter-process communication) model, typically provided by the OS. Also, the context switch time between threads is lower than the process context switch. Process context switching requires more overhead from the CPU. So, we have:

- **Single-process:** On a single-core processor, we can only run one thread at a time, but the Operating system achieves multithreading using time-slicing. A core CPU appears to be doing two things simultaneously but is time-slicing between threads (tasks) so quickly; it means a core is still running only one hardware thread, quickly switching between many software threads.
- **Multi-process:** Now, most modern phones have multi-core CPUs. That means you have more than one process (dual-core, quad-core, etc.). So technically, multiprocessing is impossible on a single-core CPU; multitasking is possible.

CPU Core: A CPU core is a hardware component called a CPU's brain. It is like a small CPU within a bigger CPU. The core can process all the computational jobs independently. It receives commands and performs the related operations or calculations.

What is a Thread? It's critical to distinguish between hardware and software threads.

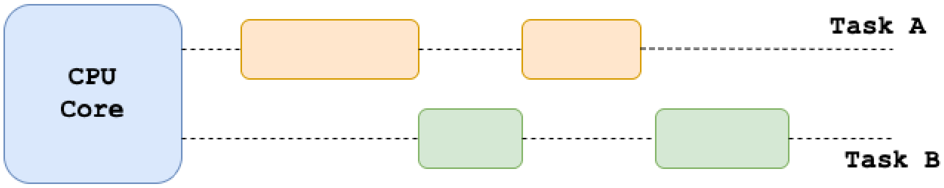
- **Hardware Thread:** A single-core CPU can have up to 2 hardware threads per core. For example, a dual-core CPU will have four threads. Multiple cores allow the CPU to execute code simultaneously.
- **Software Thread:** One hardware thread can run many software threads. We need to be able to divide problems into smaller tasks. So, we can start as many threads as we require, regardless of the number of physical CPU cores. All software threads assigned to the hardware threads and hardware threads assigned to the processing core. The OS tries to map threads to cores if sufficient cores exist. Otherwise, it uses time-slicing

to do its tasks.

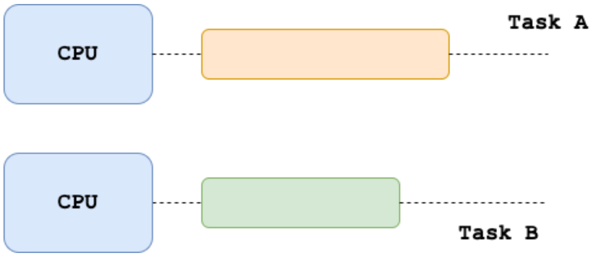
1.1.1 Concurrency & Parallelism

Let's briefly define concurrency and parallelism, and then we'll dive deeper into how to improve it.

- Concurrency: One of the main goals of concurrency is to prevent tasks from blocking each other by switching back and forth when one of the tasks is forced to wait. For example, Task A progresses till a certain point, then the CPU stops working on Task A, switches to Task B, and starts working on it for a while, and so on.



- Parallelism: In parallelism, multiple tasks can run simultaneously (e.g., on a multi-core processor or a machine with multiple CPUs). Therefore, it is not possible to have parallelism on machines with a single processor and a single-core. With parallelism, we can maximize the use of hardware resources.



- Combination of concurrency and parallelism: A system can be concurrent and parallel.

1.1.2 Concurrency is Not Parallelism

This is a nice summary by Rakhim Davletkaliyev of an excellent talk by Rob Pike, "Concurrency is Not Parallelism".

Concurrency is about dealing with a lot of things at once. Parallelism is about doing a lot of things at once. Concurrency is structuring things in a way that might allow parallelism to execute them simultaneously. However, parallelism is not the goal of concurrency. The