



30-Day Android Challenge

From Basics to Advanced
(Expert-Level Deconstruction 📱)

🚀 **Day 24**

Conquering Concurrency

Conquering Concurrency – Multi-threading &
Concurrency for Unbeatable Android Apps ⚡



Esmaeeil Moradi
@esmaeeil-moradi700

Day 24



Conquering Concurrency – Multi-threading & Concurrency for Unbeatable Android Apps ⚡

Hey Android Architects, Ambitious Developers, and anyone tired of "frozen" UIs!

Does your Android app occasionally "freeze," become sluggish, or display the "Application Not Responding" (ANR) message? Do you want your users to experience a super-fast and smooth app, even when dealing with heavy processing? It's time to get rid of those nightmares! On Day 24 of the #30DayAndroidChallenge, we're diving deep into the heart of multi-threading and concurrency management in Android. Let's learn how to build incredibly fast, responsive, and stable apps that will never disappoint your users!

Multi-threading and Concurrency are not just fancy words; they are the backbone of any modern, High-Performance Android application. Without a deep understanding of these concepts, your app is doomed to be slow, crash-prone, and deliver a poor user experience. Today, we'll go beyond obsolete AsyncTasks and introduce you to the most modern and powerful tools in the Android ecosystem to tame the beast of Concurrency.



Esmaeel Moradi

@esmaeel-moradi700



Before we dive deep into multi-threading and thread management, let's recall that on Day 20 and Day 21 of this challenge, we provided two comprehensive articles on background task management with Coroutines and Flow, as well as advanced asynchronous programming with Kotlin Coroutines & Flow. These articles lay a strong foundation for a deeper understanding of today's concepts:

Day 20: Managing Background Tasks with Coroutines and Flow (Modern Approaches)

<https://www.linkedin.com/feed/update/urn:li:activity:7331315110541488128/>

Day 21: Advanced Asynchronous Programming: Kotlin Coroutines & Flow (Advanced)

<https://www.linkedin.com/feed/update/urn:li:activity:7331661929251241985/>



Esmaeel Moradi

@esmaeel-moradi700

Introduction:

Why is Concurrency the Lifeline of Your Android App? —→

An Android app that is slow or unresponsive is doomed to fail! In today's fast-paced world, users expect apps to respond instantly to their input and never "freeze." This is where Concurrency, or concurrent programming, comes in.

Common Problems:

- **ANR (Application Not Responding):** If the Main Thread (UI Thread) is blocked for too long (around 5 seconds), the Android system displays an ANR message to the user. This is the worst nightmare a developer can experience and directly leads to app uninstallation.
- **Jank (UI Frame Drops):** Even if the app doesn't crash, minor slowdowns in the UI (less than 60 frames per second) create an unpleasant "laggy" user experience.

Root Cause:

The Main Thread or UI Thread in Android has very heavy responsibilities: receiving user input (touch, keyboard), drawing the UI, managing animations, and so on. Executing any Blocking operation (an operation that takes time and stops the code until it finishes), such as network calls, heavy database operations, or processing large files on the Main Thread, will cause the UI to completely freeze and become unable to respond to user input.



Esmaeel Moradi

@esmaeel-moradi700



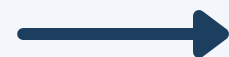
The Solution:

Multi-threading and **Concurrency** **Multi-threading** allows us to move heavy and time-consuming operations to Background Threads. This frees up the Main Thread to continue its primary job of maintaining UI responsiveness. Concurrency isn't just about parallel execution; it's about safely and correctly managing these concurrent executions to prevent hidden issues. In fact, Concurrency is no longer an "extra feature"; it's a "fundamental necessity" for any modern app to compete in today's market.



Esmaeel Moradi

@esmaeeil-moradi700



Anatomy of Threads and the Battleground of the **Main Thread**

To master Concurrency, we must first understand the anatomy of Threads and the pivotal role of the Main Thread.

Main Thread (UI Thread): The King of Responsiveness!

- **Crucial Role:** This is the only Thread allowed to update the UI, receive user events, and manage the lifecycle of Android components (like Activities and Fragments). All code related to displaying and interacting with the user interface must be executed on this Thread.
- **Limitations:** This Thread must never be blocked for an extended period. Executing any Blocking operation (like network API calls, reading/writing large files, complex numerical computations, or database access) on the Main Thread will be catastrophic.
- **ANR Scenario:** When the Main Thread is blocked for more than 5 seconds, the Android system detects that the app is "Application Not Responding" (ANR) and displays a message to the user, potentially allowing them to close the app. This destroys the user experience.



Esmaeel Moradi

@esmaeel-moradi700



- **Background Threads: The Unsung Heroes!**

1. **Their Role:** Background Threads provide a safe environment for executing heavy and time-consuming operations. By offloading these operations to the background, the Main Thread remains free to continue its primary UI responsibilities.

2. **Process vs. Thread (Briefly):** Simply put, a Process is an independent memory space for running a program, while Threads are smaller units of execution within a Process that share common memory resources. Multiple Threads can run concurrently within a single Process.

- **The Communication Challenge: Returning to the UI**

Since Background Threads are not allowed to directly access the UI, the main challenge is how to safely return the results of background operations to the Main Thread so the UI can be updated. This communication must be done securely and without causing Race Conditions or Memory Leaks.



Esmaeel Moradi

@esmaeeil-moradi700

Tools for Mastering Concurrency in Android (From Classic to Modern)

Android offers various tools for managing Concurrency, each with its own advantages and disadvantages. The choice of the right tool depends on the project's complexity and team preferences.

A. Kotlin Coroutines:

The Modern, Lightweight, and Powerful Approach

Kotlin Coroutines is the modern and recommended approach for asynchronous programming in Android. It provides a lightweight and efficient alternative to traditional Threads, making asynchronous code look like sequential and readable code.

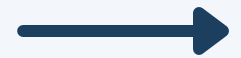
- **Why Coroutines?**
- **Lightweight:** Coroutines are much lighter than Threads, allowing thousands of Coroutines to run with minimal overhead.
- **High Readability:** You write asynchronous code in a Sequential and readable manner (suspend functions).
- **Easier Error Handling:** They have built-in mechanisms for error management.
- **Reduced Boilerplate:** Eliminates the need for nested callbacks (Callback Hell).



Esmaeel Moradi

@esmaeeil-moradi700

Key Concepts:



- **Suspend Functions:** Functions that can pause their execution and resume later (possibly on a different Thread), without blocking the main Thread.
- **Coroutines Scope:** A Scope for Coroutines that manages their Lifecycle. When the Scope is cancelled (e.g., when a ViewModel or Activity is destroyed), all Coroutines launched within that Scope are also cancelled, preventing Memory Leaks. (ViewModelScope, LifecycleScope are examples of these Scopes.)
- **Dispatchers:** Determine which Thread a Coroutine will run on.
 - **Dispatchers.Main:** For UI operations and interaction with the user interface.
 - **Dispatchers.IO:** For Blocking operations like network, database, and file I/O.
 - **Dispatchers.Default:** For CPU-bound heavy operations (e.g., complex computations).
- **Coroutines Builders:launch:** To start a Coroutine that doesn't return a result (fire-and-forget).
 - **async/await:** To start a Coroutine that returns a result asynchronously. await waits for the result to be ready.
 - **withContext:** A powerful function for safely and easily switching between Dispatchers within a Coroutine.



Esmaeel Moradi

@esmaeel-moradi700

Practical Code Example:

Network API Call and UI Update with Coroutines This code demonstrates how a long-running network call is executed on a Background Thread (using Dispatchers.IO) and then the result is returned to the Main Thread to update the UI, all without blocking the Main Thread. →

```
// Example: Fetching data from network and updating UI
suspend fun fetchDataAndDisplay() {
    try {
        // Switch to IO Dispatcher for network operation
        val data = withContext(Dispatchers.IO) {
            delay(2000) // Simulate network call delay
            "Data from network!"
        }
        // Switch back to Main Dispatcher to update UI
        withContext(Dispatchers.Main) {
            // Update UI (e.g., someComposeState.value = data)
            Log.d("App", "UI updated with: $data")
        }
    } catch (e: Exception) {
        // Handle error on Main Thread if something goes wrong
        withContext(Dispatchers.Main) {
            Log.e("App", "Error fetching data: ${e.message}")
            // showToast("Error: ${e.message}")
        }
    }
}

// How to call it from a Composable/ViewModel
@Composable
fun MyScreen() {
    // ... some UI state
    LaunchedEffect(Unit) {
        fetchDataAndDisplay()
    }
}
```



Esmaeel Moradi

@esmaeeil-moradi700

B. Executors & Thread Pools:

Low-Level Control and Resource Management

Executors provide a lower-level API for managing Threads, allowing you to create Thread Pools and submit tasks to them. This is useful for scenarios where you need precise control over the number of Threads or for Interoperability with older codebases.

- **Why Executors?**

- Precise control over the number and Lifecycle of Threads.
- Optimizes resource consumption by reusing Threads.
- Suitable for executing multiple CPU-bound or IO-bound tasks in parallel.

- **Common Executors Types:**

- `Executors.newSingleThreadExecutor()`: Creates a Thread Pool with a single Thread that executes tasks sequentially.
- `Executors.newFixedThreadPool(n)`: Creates a Thread Pool with a fixed number of n Threads.
- `Executors.newCachedThreadPool()`: Creates a Thread Pool with dynamic Threads that add/remove Threads as needed (use with caution, can lead to too many Threads).



Esmaeel Moradi

@esmaeel-moradi700



- **Lifecycle Management:** It is crucial to shut down Thread Pools when you are finished with them using `shutdown()` or `shutdownNow()` to prevent Memory Leaks and unnecessary resource consumption.
- **Code Example:** Executing Multiple Heavy Processing Operations with `FixedThreadPool` This example shows how to run multiple heavy tasks (e.g., data processing) in parallel in the Background.

```
// Example: Using FixedThreadPool for background tasks
// Best practice: declare this in a singleton or class lifecycle
val executorService = Executors.newFixedThreadPool(4)

fun processHeavyData(dataItem: String, onResult: (String) -> Unit) {
    executorService.execute {
        // Simulate heavy computation (e.g., image processing)
        Thread.sleep(3000)
        val processedResult = "Processed: $dataItem"

        // Post result back to Main Thread using Handler
        Handler(Looper.getMainLooper()).post {
            onResult(processedResult)
        }
    }
}

// How to use:
fun exampleUsage() {
    processHeavyData("Item A") { result ->
        Log.d("App", result) // Update UI with result A
    }
    processHeavyData("Item B") { result ->
        Log.d("App", result) // Update UI with result B
    }
}
```



Esmaeel Moradi
@esmaeeil-moradi700

C. Handlers & Looper: Classic Tools for Thread Communication

Handlers and Loopers are fundamental tools in the Android threading system, used to send Messages or Runnables from one Thread to the MessageQueue of another Looper (typically the Main Thread).

- **Why Handlers?**
 - The primary way to communicate between Background Threads and the Main Thread in the traditional View system.
 - Still useful in some scenarios or for Interoperability with older codebases.
- **Looper:** Not every Thread has a Looper by default. The Main Thread has a Looper (which manages a MessageQueue). Handlers are associated with a specific Looper and send messages to that Looper's MessageQueue.
- **HandlerThread:** A helper class that creates a Thread with its own Looper, which is very useful for executing sequential operations on a dedicated Background Thread.



Esmaeel Moradi

@esmaeeil-moradi700



- **Code Example:** Updating a ProgressBar from a Background Thread using a Handler This example shows how a long-running operation in the Background can periodically update a ProgressBar on the Main Thread.

```
// Example: Updating progress bar from background thread
val mainThreadHandler = Handler(Looper.getMainLooper())

fun downloadFileInBackground(onProgress: (Int) -> Unit) {
    Thread {
        for (progress in 0..100 step 10) {
            Thread.sleep(500) // Simulate download progress
            mainThreadHandler.post {
                onProgress(progress) // Update UI on Main Thread
            }
        }
        mainThreadHandler.post {
            onProgress(100) // Final update
            Log.d("App", "Download Complete!")
        }
    }.start()
}

// How to use:
// Call downloadFileInBackground { progress ->
//     yourProgressBar.progress = progress
// }
```



Esmaeel Moradi
@esmaeeil-moradi700



D. RxJava/Flow (Reactive Programming - Brief Mention)

- **RxJava:** A powerful library for Reactive programming that allows you to manage data streams asynchronously and event-driven. Schedulers in RxJava act similarly to Dispatchers in Coroutines.
- **Kotlin Flow:** (which we covered on Day 21) is Kotlin's native tool for implementing Reactive Streams and is a modern alternative to RxJava within the Coroutines ecosystem. Both tools are highly beneficial for complex Concurrency and Data Transformation scenarios.



Esmaeel Moradi
@esmaeel-moradi700

Challenges of Concurrency and Smart Solutions



Concurrency, despite its immense power, can be a source of difficult and unexpected bugs. Understanding common challenges and their resolution patterns is crucial.

- **Race Conditions and Synchronization:**

- **Explanation:** Occurs when multiple Threads simultaneously access a shared resource (like a variable or data structure) and try to modify it, and the final outcome depends on the unpredictable order in which Threads access the resource.
- **Solutions:** synchronized blocks/methods: In Java/ Kotlin, synchronized ensures that only one Thread can access a specific code block or method at a time.
- **Mutex (in Coroutines):** In Coroutines, a Mutex (Mutual Exclusion) is a lock that ensures only one Coroutine can access a protected section of code at a time.
- **Atomic Classes:** Classes like AtomicInteger, AtomicLong whose operations are performed Atomically (indivisibly) and are Thread-safe.



Esmaeel Moradi

@esmaeeil-moradi700



- **Deadlock:**
 - **Explanation:** A dangerous scenario where two or more Threads are permanently waiting for locks that are held by each other. Each Thread holds a resource and waits for a resource held by another Thread.
- **Identification and Prevention:** Deadlock is difficult to detect at runtime. To prevent:
 - Acquire locks in a consistent order (Lock Ordering).
 - Use Timeout when attempting to acquire a lock.
 - Minimize the number of locks.
 - Where possible, use Lock-Free or Non-Blocking algorithms.



Esmaeel Moradi
@esmaeeil-moradi700



- **Memory Leaks:**
 - **Connection to Context and Lifecycle:** One of the most common Memory Leaks in Android occurs when a Background Thread or Runnable (e.g., within a Handler), holds a strong reference to an Activity or Fragment (which holds the Context). If the Activity is destroyed but the Thread is still running, the Activity cannot be garbage collected.
- **Solutions:** Use Weak References for Context references (if absolutely necessary, though less recommended).
- **Lifecycle Management of Coroutines:** Using ViewModelScope and LifecycleScope ensures that Coroutines are automatically cancelled with the Lifecycle of Android components, preventing Memory Leaks.
- **Ensure proper shutdown of Executors and cancelling of Coroutines.**



Esmaeel Moradi
@esmaeeil-moradi700



- **UI Blocking (Absolute Prevention):**
 - **Golden Rule:** Never execute Blocking operations (network, database, disk I/O) on the Main Thread. Consider this an fundamental principle.
- **Techniques:** Always use Dispatchers.IO (in Coroutines) or dedicated Threads (with Executors or HandlerThread) for these tasks.
- **Error Handling:**
 - In concurrent programming, errors can occur on Background Threads and might not propagate back to the Main Thread, or cause unexpected crashes.
 - In Coroutines: Use try-catch blocks and CoroutineExceptionHandler for error management.
 - In Executors: You can use try-catch within Runnables and pass error results back to the Main Thread.



Esmaeel Moradi

@esmaeeil-moradi700

Best Practices and Design Patterns for Stable Concurrency

To build apps that remain stable and responsive as they grow in complexity, following best practices is essential.

- **Single Source of Truth:**

Even in Multi-threaded environments, application data should be managed from a single, reliable source. This prevents data inconsistencies and Race Conditions. The Repository layer is typically the ideal place to manage the Single Source of Truth and abstract threading logic.

- **Separation of Concerns:**

Separate the logic related to Threading and Concurrency (like choosing a Dispatcher or Executor) from Business Logic and UI Logic. This makes your code more Testable, maintainable, and scalable. Architectural layers (as discussed in previous days) aid in this separation.

- **Using Flow or LiveData for Background Task Results:**

Instead of direct callbacks, use Reactive Streams like Kotlin Flow (alongside Coroutines) or LiveData to deliver the results of Background operations to the UI Layer. These tools are Lifecycle-Aware and automatically manage Subscriptions with the Lifecycle of UI components, preventing Memory Leaks.



Esmaeel Moradi

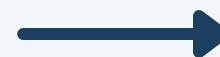
@esmaeel-moradi700



- **Profiling and Debugging Concurrent Code:**
 - **Concurrency issues** often manifest in production and are difficult to debug during development.
 - **Android Studio Tools:** The CPU Profiler (for identifying busy Threads and Blocking operations) and Thread Monitor within the Profiler are vital tools for identifying Bottlenecks and threading issues.
 - **Reporting:** Use crash reporting tools like Firebase Crashlytics to identify ANRs and Deadlocks on real user devices.
- **Testability:**
 - Writing unit tests for concurrency logic and integration tests to ensure the correctness of Thread operations and data management is crucial.
 - In Coroutines, using `Dispatchers.setMain()` makes it easier to test Coroutine code on the Main Thread.



Esmaeel Moradi
@esmaeeil-moradi700



Conclusion:

The Power of Concurrency in Your Hands

Concurrency is perhaps one of the most complex yet powerful aspects of Android app development. Mastering it marks the boundary between an ordinary app and a Performant, user-friendly masterpiece. By deeply understanding the anatomy of Threads, making informed choices about modern tools like Kotlin Coroutines, and being aware of the hidden challenges of Concurrency (like Race Conditions and Deadlock), you will be able to create apps that are not only beautiful and functional but also unrivaled in terms of performance.

No more ANRs or frozen UIs! The power of Concurrency is in your hands. Start now and elevate your Android apps to the pinnacle of responsiveness and stability!

Which tool do you prefer for managing Concurrency and why?

Have you ever battled a nightmare Deadlock?

Share your experiences with us! 🙋



Esmaeeil Moradi

@esmaeeil-moradi700

Looking forward to your
insights and experiences!
Let's elevate Android
development together.



Esmaeeil Moradi

@esmaeeil-moradi700