IndiaFOSS 2026 · AOSP Track · INTRO

Zygote to First Frame

Owm Dubey ·

Under the hood of Android: the **exact chain of syscalls, IPC, and memory tricks** that runs between your tap and the first pixel on screen.



Six Stops

01

What's already alive

02

Tap reaches the activity manager

03

The zygote fork

04

Binder: the wire under every hop

05

What wakes up inside the new process

06

First frame: measure, layout, draw, composite

MOTIVATION: THE INVISIBLE GAP

You tapped an icon. 500 ms later, an app appeared. What filled the gap?

> "Fully understanding the internals of Android's system services is like trying to swallow a whale." Karim Yaghmour, quoted in Levin, *Android Internals Vol. I*

We take exactly **one bite: tap → first rendered frame.** That gap is engineered to be invisible, and it is doing an extraordinary amount of work.

New Linux Process

A full OS process, not a thread.

`zygote → fork()`

Managed Runtime

ART fully initialized and specialized.

`ActivityThread.main()`

IPC Round-Trips

Several Binder transactions, kernel-mediated.

`/dev/binder`

Render Passes

Measure → layout → draw, then a compositor pass.

`RenderThread → SurfaceFlinger`

All of this, in under 500 ms.

One tap crosses five processes before a pixel lands.

1. **Launcher** — `touch`
   Builds the Intent, calls startActivity()
2. **system_server** — `Binder`
   Resolves the component, checks the caller UID
3. **zygote** — `socket`
   fork()s a child and specializes it
4. **Your app process** — `fork()`
   ActivityThread.main(), then onCreate()
5. **SurfaceFlinger** — `BufferQueue`
   Composites the first frame onto the display

STOP 1: WHAT'S ALREADY ALIVE

Before the launcher draws, three processes are already running, and one of them forked another.



init (PID 1)

First user-space process; the kernel hand-picks it.



servicemanager

The Binder **context manager**, reachable at **handle 0**.



zygote / app_process

The binary on disk is `app_process`; it renames itself after loading `ZygoteInit`.



system_server

`fork()`ed by the zygote as its **very first child** (`--start-system-server` flag).

⚠ One system_server crash restarts the whole framework: the price of near-free calls between ~100 tightly-coupled services.

STOP 2: TAP REACHES ACTIVITY MANAGER

The Launcher has zero privilege to start a process. It can only ask.

1. **Build the Intent**

   Tapping an icon builds an `Intent` (`ACTION_MAIN` / `CATEGORY_LAUNCHER`) and calls `startActivity()`.

2. **Ask across a process boundary**

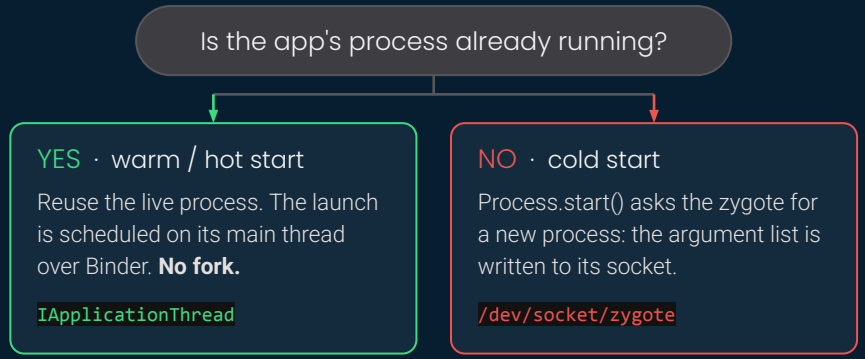
   The Launcher cannot create processes, so the call becomes a **Binder transaction** into `system_server`, handled by `ActivityTaskManagerService`.

3. **Resolve, then verify**

   The service resolves the component, then checks permissions using the **kernel-verified caller UID** carried by the Binder transaction.

The Decision That Matters

Is a process already hosting this component? If not, the activity manager asks the zygote to fork one.



*The rest of this talk follows the **NO** branch.*

WHY THE ZYGOTE EXISTS

A cold managed-runtime start is 600 ms – 1.2 s of work before a single line of app code runs.

The activity manager needs a fresh runtime in a few hundred milliseconds. Building one from scratch cannot hit that. So the zygote does not.

1 Disk I/O

Map and load thousands of classes from the boot classpath and the APK, resolving references across them.

2 Verification

Type-safety and control-flow checks on bytecode.

3 Class Initialization

Run `<clinit>` static initializers across hundreds of framework classes: the resource system, the networking stack, font and Typeface loading, the entire view toolkit.

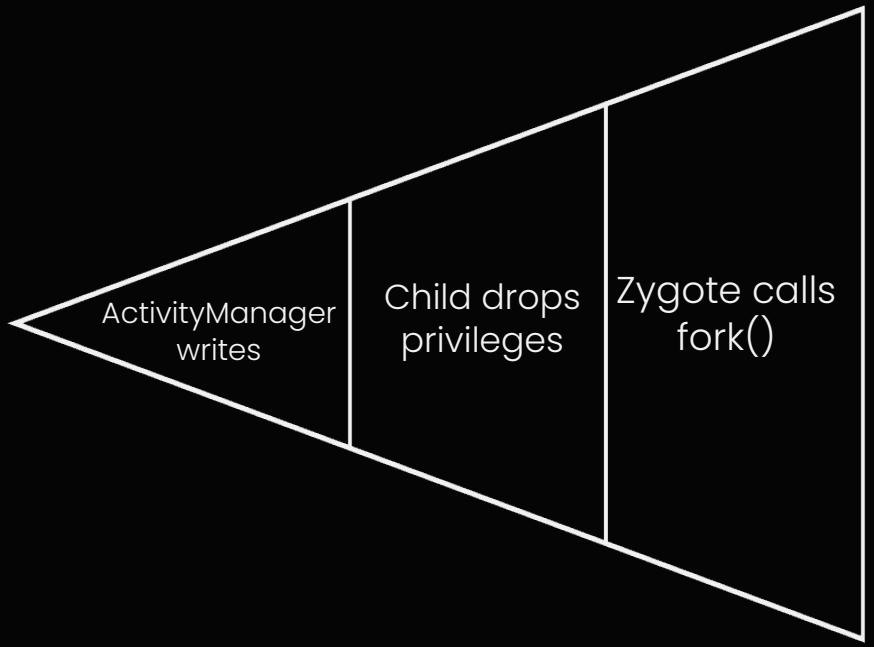
4 Heap Construction

ART allocates a fresh managed heap, card table, mark stack, and large-object space per process, then has to warm it.

The zygote pays all of this for the framework once, at boot, inside the frozen VM.

STOP 3: THE FORK

Nobody asks the kernel for a fresh app process. A sleeping process is messaged, and it clones itself.



1

activity manager → socket

`ZygoteProcess` writes an argument list (target UID/GID, supplementary groups, nice-name, seinfo, target SDK, ABI, runtime flags) into `/dev/socket/zygote`.

2

zygote → `fork()`

`ZygoteConnection` handles the command and calls `Zygote.forkAndSpecialize()`.

3

child → drop privilege

`setgid()` then `setuid()` (**that order is security-critical**), `setgroups()`, clamp the capability bounding set, install a `seccomp-bpf` filter, transition to the app's SELinux domain.

4

child → specialize

Close the inherited zygote socket, rename the process to the package name (`Process.setArgV0()` / `prctl(PR_SET_NAME)`, which is why `ps` shows it), start the Binder thread pool, then enter `ActivityThread.main()`.

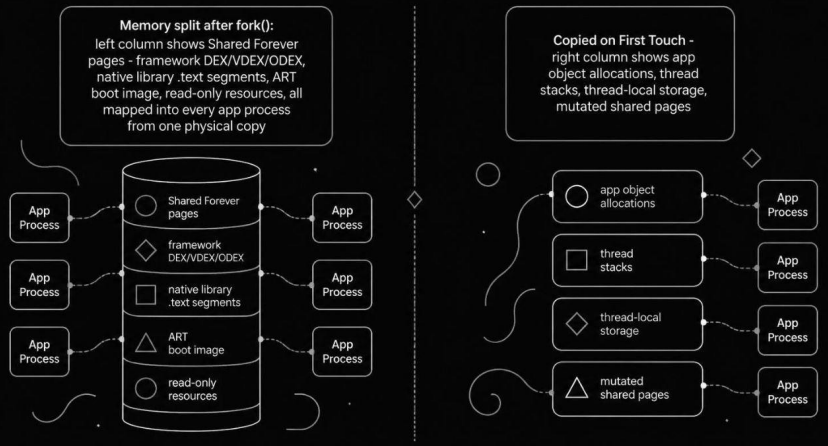
Security invariant: every privilege drop in step 3 completes **before one instruction of app bytecode executes**.

STOP 3: COPY-ON-WRITE MEMORY

Right after `fork()`, the child owns almost nothing. It borrows the parent's pages until it writes.

The Copy-on-Write Mechanism

The child gets a fresh page table whose entries point at the **parent's physical pages**, all marked read-only.



Shared for the Life of the Device

Framework DEX bytecode, native library `.text` segments, the ART boot image, read-only resources.

Copied on First Touch

App object allocations, thread stacks, thread-local storage, and any previously-shared page the app mutates.

The GC Caveat

A tracing collector that rewrites object headers across the heap would dirty shared pages and destroy the sharing.

USS vs. PSS

Net effect: a freshly specialized process has a tiny **USS** (unique set size) and a large shared component in its **PSS**. Measure with `dumpsys meminfo` or `procrank` (Levin Ch. 7).

STOP 3: TRADEOFFS

Fork buys sub-50 ms launches and 80–90% shared RAM. It is paid for in ASLR entropy.

The Payoffs

<50 ms Fork time

vs. 600 ms–1.2 s for a cold managed start. A **10–20×** cut in launch time the app author cannot optimize.

Zygote fork <50 ms

Cold managed start 600 ms–1.2 s

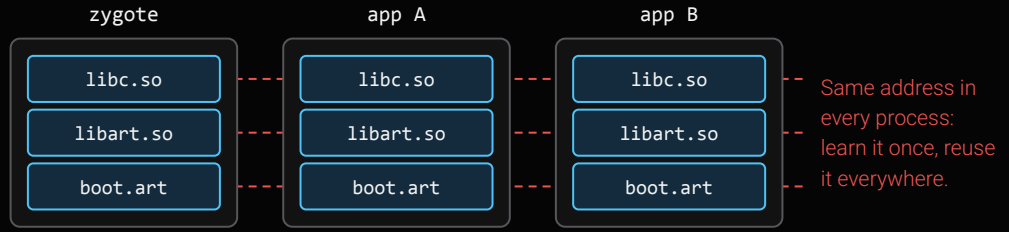
80–90% Shared RAM

of a running app's pages are implicitly shared with other apps and with `system_server` via CoW.

shared via copy-on-write — private

The Cost: Weakened ASLR

Every app inherits the zygote's address-space layout, so `libc`, the ART runtime, and framework code sit at **identical virtual addresses in every app process.**



Mitigations (Attacking from a Different Direction)

PAC
ARMv8.3 Pointer Auth
Signs return addresses; forgeries fail the check.

CFI
Control Flow Integrity
Indirect calls may only land on valid targets.

Shadow Stacks
Shadow call stacks
Return addresses kept on a separate, hidden stack.

These cut the *exploitability* of a known layout without restoring the lost entropy.

STOP 4: THE WIRE UNDER EVERY HOP

Launcher→activity manager, child process→activity manager, every getSystemService call. Every hop was Binder, a kernel driver, not a socket

Mechanism: One Copy

A character device `/dev/binder`, driven with `ioctl(BINDER_WRITE_READ)`.

Discovery: servicemanager (handle 0)

A service registers a name with servicemanager (handle 0); a client asks handle 0 for a handle to that name; the two then talk **directly**, over an **AIDL**-generated Stub/Proxy pair, with arguments marshalled into a `Parcel`.

Identity: Kernel-Stamped UID

The driver stamps every transaction with the caller's real UID/PID.

You Already Use This

Every `Context.getSystemService(...)` is a servicemanager lookup plus Binder RPC.

Limits Worth Knowing

~1 MB Transaction Buffer

15 + 1 Threads

Synchronous on Caller's Thread

Variants: Project Treble

`/dev/hwbinder` (HIDL) and `/dev/vndbinder` (vendor, stable AIDL), the same driver with isolated contexts, introduced with Project Treble to allow vendor code to evolve independently of the framework.

STOP 5: THE NEW PROCESS

The child from Stop 3 has finished specializing. The process is alive, but your code still has not run. The entry point is a static method you never call.

1 **Looper.prepareMainLooper()**

Creates the MessageQueue and binds it to this thread.

2 **thread.attach(false, …)**

A Binder call handing the activity manager an `IApplicationThread` back-channel.

3 **Looper.loop()**

Dispatches forever. The `throw` after it is unreachable by design: the loop is never supposed to return.

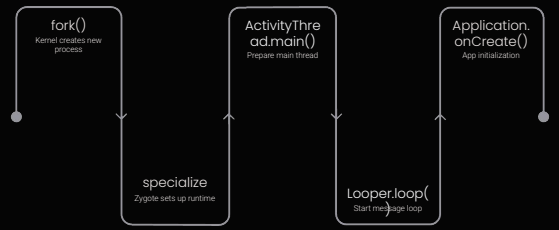
4 **System drives via ApplicationThread → ClientTransaction / TransactionExecutor**

`bindApplication()` → `ContentProvider.onCreate()` → `Application.onCreate()` → `Activity.onCreate()` → `onStart()` → `onResume()`

⚠ Provider ordering: content providers are installed **before** `Application.onCreate()`.

frameworks/base/core/java/android/app/ActivityThread.java (abridged)

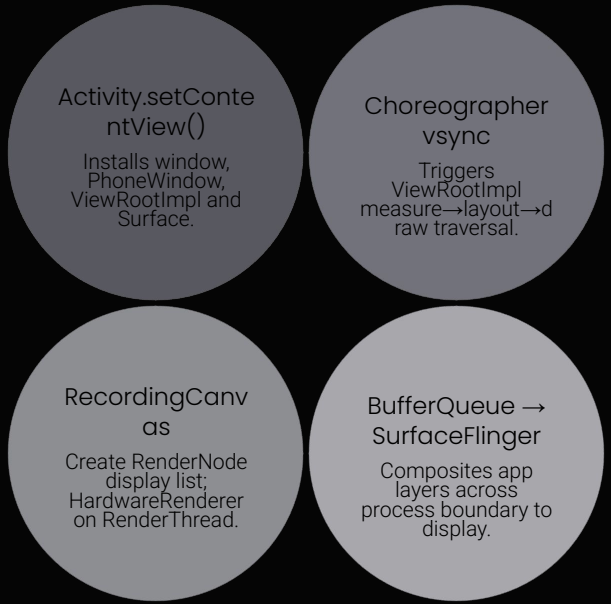
```
public static void main(String[] args) {
  Looper.prepareMainLooper();   // 1 · main-thread queue
  ActivityThread thread =
      new ActivityThread();
  thread.attach(false, startSeq); // 2 · register with AMS
  Looper.loop();                // 3 · run forever
  throw new RuntimeException(
    "Main thread loop unexpectedly exited");
}
```



The reversal: your `onCreate()` is roughly step 4 of a sequence that began in the kernel with `fork()`.

STOP 6: FIRST FRAME

Your code has run. Now, finally, a pixel. View and Compose take the identical platform path to the first frame, and differ only above it.



Step 1: Window & Surface

`Activity.setContentView()` → PhoneWindow installs the content view.

Step 2: Traversal

`Choreographer doFrame` (driven by the vsync signal) → `ViewRootImpl.performTraversals()`, which runs **measure → layout → draw** over the hierarchy.

Step 3: GPU work (your process)

draw records into a `RecordingCanvas`, producing a `RenderNode` display list, handed to `HardwareRenderer` on the `RenderThread`, which issues GPU work through Skia (Vulkan or GLES).

Step 4: SurfaceFlinger (separate process)

`RenderThread` queues the finished buffer into a `BufferQueue`.

Step 5: Display scanout

Display scans out the composited result. **First frame on screen.** The system logs:
`Displayed <pkg>/<activity>: +NNNms`

View vs. Compose divergence

The only divergence is **how the tree that step 2 traverses gets built.**

NOT A BLACK BOX · REPRODUCE ON ANY STOCK DEVICE

None of this is hidden. It is just under-documented. Reproduce it on any stock device.

Commands

```
# 1 · already alive + the fork: a new app's PPID is the zygote
adb shell ps -A -o PID,PPID,NAME | grep -E 'zygote|system_server'
# 2 · cold-start cost: read TotalTime in the output
adb shell am start -W -n <pkg>/<activity>
# 3 · copy-on-write: compare shared dirty vs private dirty
adb shell dumpsys meminfo <pkg>
# 4 · Binder: every service registered with servicemanager
adb shell service list
# 5 · the new process: the framework logs the start event
adb shell logcat -b events -d | grep am_proc_start
# 6 · first frame: "Displayed <pkg>/<activity>: +NNNms"
adb shell logcat -d | grep Displayed
```

CLOSE

So the next time you tap an icon:

> A socket write wakes a sleeping process. It clones itself, drops its privileges, loads your code, becomes your app, and draws its first frame. And borrowed memory from every other app on the phone keeps the whole thing light enough to fit in your pocket.

Zygote to first frame.



What's alive

Zygote sleeping, SystemServer running, init holding the socket



The tap

Socket write → Zygote wakes → `fork()` issued



The fork

Copy-on-write clone, privilege drop, specialize, 80–90% shared memory



Binder

`IApplicationThread` registered; AMS drives lifecycle over Binder



The new process

`ActivityThread.main()` · `Looper.loop()` · `onCreate()`



First frame

`ViewRootImpl` → `HardwareRenderer` → `SurfaceFlinger` → display



Thank you.

Owm Dubey

IndiaFOSS 2026 · AOSP Track