Click to verify

Android Boot Process is a process of starting a computer for using it. A computer system can be started by switching on the power supply. When the computer is started it is able to read only the part of storage called Read only memory. A small program is started on launching the computer which is stored and called as firmware. It allows accessing other types of memory such as main memory and hard disk. The firmware is use to load big programs and run it into the computer's main memory. Boot Manager is run in all devices whether it may be a mobile device, computer system or others. Boot Process in Android is of 6 steps which are as follows ?Boot ROM.BootLoader.Kernel.Init.Zygote and Dalvik VM. System Servers. Boot ROM is a step which is called when the computer is powered ON on system startup. When the system is powered on the Boot ROM code starts executing a set of instructions from a predefined location which is hardwired in the ROM. Boot ROM loads BootLoader into RAM and starts executing.BootLoaderBootLoader is a small program which runs before Android OS starts up. Mobile manufacturers put their locks and restrictions in boot loader. Boot Loader contains a code which is executed before starting any operating system. Boot Loader executes two steps which are as follows ?In the first stage the boot loader detects the external RAM of the device and loads a program in it for the second stage. In the second stage, the boot loader sets up the memory and network which is required to run the Kernel. KernelThe Android OS consists of an operating system consisting of Linux Kernel and a collection of user-space applications and libraries. The kernel is the lowest level of an operating system which interacts directly with the hardware and provides basic services to user space components. When the kernel finishes system setup it looks for init in system files and launches the root process or very first process of the system.InitInit is the first user space process which runs on Android devices. It is responsible for starting other user-space processes and services, and setting up the device environment. This process reads the configuration files in /init.rc directory and starts necessary services and daemons. Init is the first process that runs with root privileges, and it is responsible for setting up the device's security policies, mounting the file systems, and setting up the network. It also starts the Android runtime, which is responsible for running the Android framework and applications.Zygote and Dalvik VMZygote is a virtual machine process which starts as the system boots. When the app process launches the Zygote it then creates Dalvik VM and then calls Zygotes main method to receive a request to launch an app through dev/zygote. Once the app is launched the fork() method is triggered which will create a clone of itself to replicate it in another memory space.System ServersWhen the zygote has loaded all the Java classes and resources, it will start the System Server. This server is a core of android system. The android_servers native library is loaded by this system servers to provide interface to native functionalities. The native init method will then setup native services called. After the services have been created the server thread will be created. The remaining services will be started by the thread according to the specific order. Once the system services has started running in memory the boot process of android is completed and at this time "ACTION_BOTT_COMPLETED" action will be fired.ConclusionIn this article we have taken a look on What is Boot Process in Android and what are the different process which are included in the boot process.The Android boot process is a complex, multi-stage journey that begins with the hardware-level operations and ends with the familiar Android home screen. Each step in this process involves several critical components, from the BootROM and bootloader to the Android runtime and SystemServer. While much of the early boot process is specific to the hardware, the open-source nature of Android allows us to explore the latter stages through its codebase.In this article, we will look at the Android boot process through a code walkthrough. We will begin by understanding the conceptual roles of the BootROM and bootloader, then dive into the core Android components like the Linux kernel, the init process, and Zygote. By following the code flow from initialization to the launching of the home screen, we aim to provide a clear picture of how Android transforms from a powered-off state to a fully operational system.Whether you are a developer working on Android internals or simply curious about how Android boots up, this guide will walk you through the process step by step, with references to key source files and functions in the Android Open Source Project (AOSP).BootROM:This is the very first step in the boot process, and its embedded into the hardware. The BootROM code is stored in the chip (SoC) by the manufacturer and is non-modifiable. Its primary job is to initialize basic hardware like RAM and storage and to find and load the next stage bootloader from a predefined location (like an SD card or internal flash).Since BootROM is hardware-specific and deeply integrated with the SoC design, no source code is publicly available.BootROM usually performs the hardware checks, and its behavior differs based on the SoC. If there is a valid bootloader image, it passes control to it.Bootloader:The Bootloader is the first piece of software that runs after BootROM. Its responsible for setting up the environment for the Android kernel to run, including initializing hardware components and loading the kernel image from storage into memory.Depending on the manufacturer, bootloaders like U-Boot or proprietary ones (like Qualcomms aboot) are used.The bootloader typically handles features like boot partition verification and may include the ability to boot into recovery mode or fastboot mode for development and debugging.Code Example: (for illustration)Kernel:Once the bootloader finishes its job, it hands over control to the Linux kernel (which is highly customized for Android). The kernel initializes the systems hardware, mounts the root file system, and starts the init process.The kernel can be customized by OEMs for their specific hardware, but generally, it contains Android-specific drivers (such as those found in /drivers/* or /arch/arm64/).Kernel Command Line: When the kernel starts, it receives parameters from the bootloader (like root= which specifies the root partition).Code Example: (hypothetical)Init Process and main.cppFile Location: main.cppThe init process is the cornerstone of Androids user-space initialization. This process begins in main.cpp, which is located in the system/core/init directory.Key functions in main.cpp:main(): The entry point for the init processFirst, it sets a high process priority (setpriority()).It checks if the program name is ueventd. If true, it starts the ueventd process by calling ueventd_main().Depending on the command-line argument (argv), it determines which function to call:If subcontext is passed, it calls SubcontextMain().If selinux_setup is passed, it calls SetupSelinux().If second_stage is passed, it calls SecondStageMain().Otherwise, by default, the first stage of the initialization is handled by calling FirstStageMain().FirstStageMain() (from first_stage_init.cpp)File Location: first_stage_init.cppThis function handles the first stage of the Android boot process, where the basic system components are initialized.Key steps in FirstStageMain():Initializes the basic system environment, such as setting up the kernel arguments and mounting essential filesystems.It calls PropertyInit(), which is responsible for initializing the Android property service (discussed later).Handles the early setup of SELinux by calling SetupSelinux().Once completed, it will transition to the second stage of the boot process.SetupSelinux() (from selinux.cpp)File Location: selinux.cppThis function is responsible for setting up SELinux in Android. SELinux is a mandatory access control system that enforces security policies.Key steps in SetupSelinux():It loads the SELinux policy and configures the system based on security contexts.Verifies and enforces SELinux modes like permissive or enforcing mode.This setup ensures that the necessary security contexts are applied before the Android system moves further in the boot process.PropertyInit() (from property_service.cpp)File Location: property_service.cppThe property service in Android is responsible for system-wide properties, similar to environment variables, used to configure various aspects of the system.Key steps in PropertyInit():It sets up the property service and loads initial system properties.Properties are essential for various components in Android, including system services and apps.This function initializes the service by reading property values from /system, /vendor, and other system partitions.Second Stage Initialization (SecondStageMain())After the first stage completes, the second stage is triggered by SecondStageMain(). This process includes:Starting additional system services.Running Androids init scripts (from .rc files).Starting the zygote process, which spawns system processes.system/core/rootdir/init.rc -File Location: init.rcThis file is one of the main Android init scripts, and it defines the initial set of commands that the init process runs during system boot. It sets up the core environment, mounts filesystems, and starts essential services.Example entries in init.rc:on boot: Commands executed during boot, such as setting up SELinux and initializing properties.on zygote-start: This defines how to start the Zygote process, which is responsible for launching apps and system services.After init.rc completes its setup, it proceeds to start the Zygote process by executing the zygote service defined in init.zygote64.rc.init.zygote64.rcFile Location: init_zygote.rcThis script focuses on starting the 64-bit Zygote process, which is responsible for forking new processes, including system services and apps.This file is an init script written in the Android init language. It defines various commands that need to be executed by the init process during system startup.Key steps:The .rc file is parsed and executed by the init system during the second stage of the boot.init.zygote64.rc specifically defines how the Zygote process (the process that spawns Android apps and services) should be started. It also handles starting the system server and other essential processes.For example:The service zygote64 entry in init.zygote64.rc starts the Zygote process.Other entries in the file can define services, mount points, and configuration directives that the init process follows during the second stage.Explanation:The zygote service starts the Zygote process by calling /system/bin/app_process64 with the necessary arguments.The start-system-server argument instructs Zygote to start the SystemServer process after initialization.At this stage, the Zygote process is initialized and starts listening for fork requests from the system to launch other processes. This leads to the app_process and AndroidRuntime layers.Code Flow Summary:main() in main.cpp: This is the entry point of the Android init process. It calls FirstStageMain() if no special command-line arguments are provided.FirstStageMain() initializes the base system environment, mounts necessary filesystems, and calls PropertyInit() to initialize the property service and SetupSelinux() to handle SELinux.SetupSelinux() loads and configures SELinux policies.PropertyInit() sets up Androids property service, which allows different system components to interact through shared properties.After the first stage, SecondStageMain() is called to start system services and parse the init scripts, including init.zygote64.rc, which starts the Zygote process.File Location: app_main.cppThis file is responsible for setting up and starting the Zygote process. It acts as the entry point for both the Zygote and SystemServer.Key steps:The main() function in this file checks the arguments to determine whether it should start Zygote or SystemServer.Zygote initialization:If the zygote flag is set (based on arguments like zygote), it invokes the ZygoteInit class, which handles the next phase of Zygote initialization.This leads to the AndroidRuntime and ZygoteInit classes being initialized.frameworks/base/core/jni/AndroidRuntime.cppFile Location: AndroidRuntime.cppThis file is responsible for initializing the Android Runtime (ART) and starting the ZygoteInit process.Key function:Explanation:The start() function initializes the JVM (ART) and invokes the main() method of the ZygoteInit class in Java.This sets up the environment to start running Androids Java-based system services.This leads us to the Java-based ZygoteInit class.ZygoteInit.javaFile Location: ZygoteInit.javaThis class is responsible for further initializing the Zygote process and starting the SystemServer.Key steps:Explanation:It sets up the Zygote process, enables DDMS (Dalvik Debug Monitor), and if start-system-server was passed as an argument, it calls startSystemServer() to start the SystemServer process.It then enters the runSelectLoop(), where it listens for process forking requests.This leads to the ZygoteServer and SystemServer classes.ZygoteServer.javaFile Location: ZygoteServer.javaThis class handles the server-side of Zygote, which waits for requests to fork new processes.Key method:Explanation:The runSelectLoop() function runs a continuous loop to accept new connections (process requests) and forks them.This keeps the Zygote process running and responsive to requests to start new processes.Zygote.javaFile Location: Zygote.javaThe Zygote class provides static methods for forking new processes. Its responsible for handling the actual process creation when requests are received.Key function:Explanation:This method invokes a native function nativeForkSystemServer() (implemented in C++), which forks the SystemServer process.The SystemServer process is then initialized.com_android_internal_os_Zygote.cppFile Location: com_android_internal_os_Zygote.cppThis file contains the native code that implements the forkSystemServer() method.Key function:com_android_internal_os_Zygote_nativeForkSystemServer():Explanation:It uses the fork() system call to create a new process for SystemServer.The child process becomes the SystemServer that will later start various Android services.SystemServer.javaFile Location: SystemServer.javaThe SystemServer is a crucial process that starts up core system services in Android. It handles the initialization of system services like the activity manager, window manager, and others.Key functions:main():Explanation:startBootstrapServices(): Starts essential services like the activity manager, power manager, and package manager.startCoreServices(): Starts core system services, such as the battery service and alarm manager.startOtherServices(): Starts all other system services that arent part of the core.startSystemUi(): Finally, this method starts the SystemUI process, which manages the Android status bar, navigation, and system interface.After SystemServer.java starts SystemUI via the startSystemUi() method, the launcher app (which is typically Launcher3 in AOSP) is started as part of the Android Activity Manager and Window Manager system. Heres a breakdown of the sequence of function calls and events that lead to the launcher starting.startSystemUi()The method startSystemUi() is invoked within SystemServer.java:Explanation: This method sends an intent to start SystemUI, which is responsible for things like the status bar, navigation bar, and other system-level UI components. SystemUI runs in its own process.The launcher app is not started directly by SystemUI. It is started by the Android ActivityManagerService after the boot completes, which we will discuss later in this article.SystemUI1. startSystemUi()The startSystemUi() method begins by creating an Intent object with the component set to the SystemUI service component, which is retrieved from PackageManagerInternal. It then calls context.startServiceAsUser(intent, UserHandle.SYSTEM), which launches the SystemUI service.The key component is retrieved by calling pm.getSystemUiServiceComponent(), which provides the component information for SystemUI.2. Intent starts SystemUIThe startServiceAsUser() method starts the SystemUI process by sending the intent. This intent is directed to the SystemUI component, starting the SystemUIService.java in the SystemUI process.Context.startServiceAsUser() sends the intent to launch the SystemUIService.java class, which is a bound service that manages the initialization of SystemUI components.3. SystemUIService.javaSystemUIService.java is one of the first classes that are executed in the SystemUI process. This service is responsible for coordinating and managing system-level UI components. The service also interacts with SystemUIApplication.java to start the required services for SystemUI.SystemUIService serves as a central service for starting and managing UI components like the status bar, lock screen, and other visual elements of the Android system.4. SystemUIApplication.javaOnce the SystemUIService is initiated, SystemUIApplication.java is executed. This class is responsible for the overall setup of the SystemUI application. In particular, it calls the startServicesIfNeeded() method, which bootstraps all the core SystemUI components.SystemUIApplication is the entry point for the SystemUI application and is responsible for starting the system services related to the UI.5. startServicesIfNeeded() in SystemUIApplication.javaThe startServicesIfNeeded() method within SystemUIApplication.java checks whether the essential services required for SystemUI are running. If not, it starts those services. These services include:StatusBar: Manages the status bar (battery, network indicators, notifications).NavigationBar: Manages the system navigation (back, home, multitask buttons).LockScreen: Manages the lock screen functionality.This method ensures that all the system-level UI components are set up and running.6. SystemUiFactory.javaIn the process of starting various services in SystemUI, SystemUiFactory.java is often called to instantiate and configure specific services. For instance, if the SystemUIApplication needs to start a specific service or component dynamically, it can use SystemUiFactory to create it.SystemUiFactory is responsible for creating key services that are part of the system UI. It ensures that all necessary services are instantiated and running correctly.7. WindowManagerService.onSystemUiStarted()Back in the SystemServer.java, after starting the SystemUI service, the method windowManager.onSystemUiStarted() is called. This lets the WindowManagerService know that SystemUI has started, so it can continue with any tasks that depend on SystemUI being available.WindowManagerService manages the window system in Android, which includes tasks like rendering the system bars, displaying application windows, and handling transitions between activities.This call ensures that SystemUI is fully functional before continuing with other window-related operations.ActivityManagerService systemReady()File Location: ActivityManagerService.javaThe systemReady() method in ActivityManagerService is responsible for finalizing the systems readiness, including launching the home (launcher) app.Key function:systemReady() in ActivityManagerService.java:Starting Persistent AppsstartPersistentApps(PackageManager.MATCH_DIRECT_BOOT_AWARE);This part starts only encryption-aware persistent apps, not directly related to the launcher, but ensures essential system services are up.Enabling the Home Activity for System UserThe home activity (i.e., the launcher) is enabled for the system user if the property SYSTEM_USER_HOME_NEEDED is set to true. This ensures that the system can always boot with a home activity:Starting the Home Activity (Launcher) for System UserIf the current user is the system user (i.e., UserHandle.USER_SYSTEM), AMS calls startHomeOnAllDisplays(), which triggers the launcher (home app) to start. This method is key to starting the home screen after the system has booted:Resuming Top Activities:mAtmInternal.resumeTopActivities(false /* scheduleIdle */);This ensures the top-most activities, including the home/launcher, are resumed and brought to the foreground.The Android boot process is a complex sequence involving low-level hardware initialization and high-level system setup. The journey begins with proprietary components like BootROM and Bootloader, transitions to the Linux kernel, and culminates in Androids user-space initialization with init, Zygote, SystemServer, and the launcher.By tracing through the Android source code, you can see how each component plays a role in bringing the system to life. This article covers the major steps from the kernel to launching the Android UI, providing code insights where applicable.Some questions arise when we think about the Android boot sequence. What is the Zygote, init.rc, what is the difference between the linux kernel and the android linux kernel?.This document is intended to explain how the booting process runs.Consider the following graph:Step 1: Power On and System StartupWhen we press the power button, the Boot ROM code starts executing from a pre-defined location which is hardwired in ROM. It loads the Bootloader into RAM and starts executing.Step 2: BootloaderThe bootloader is a small program which runs before Android does. This is NOT part of the Android operating system. The bootloader is the place where manufacturer puts their locks and restrictions.The bootloader executes in two stages. In the first stage it detects external RAM and loads a program which helps in the second stage.In the second stage, the bootloader setups the network, memory, etc, which requires to run kernel. The bootloader is able to provide configuration parameters or inputs to the kernel for specific purposes.The bootloader can be found at:/bootable/bootloader/legacy/usbloaderThis legacy loader contains 2 important files:1- Init.s :: Initializes stacks, zeros the BSS segments and call_main() in main.c2- Main.c :: Initializes hardware (clocks, board, keyboard, console) and creates linux tags.Step 3: KernelThe Android kernel starts in a similar way as the linux kernel. As the kernel launches, is starts to setup cache, protected memory, scheduling and loads drivers. When the kernel finishes the system setup, it looks for init in the system files.What is the difference between the linux and android kernels?, here's a list of changes/addons that the Android Project made to the Linux kernel:Binder: It is an Android specific interprocess communication mechanism and remote method invocation system.ashmem: "Android Shared Memory". It is a new shared memory allocator, similar to POSIX SHM but with a different behavior and sporting a simpler file-based API.pmem: "Process memory allocator": It is used to manage large (1-16+ MB) physically contigous regions of memory shared between userspace and kernel drivers.logger: This is the kernel support for the logcat command.wakelocks: It is used for power management files. It holds the machine awake on a per-event basis until wakelock is released.oom handling: It kills processes as available memory becomes low.alarm manager: It lets user space tell the kernel when it would like to wake up.RAM_CONSOLE: Allows to save kernel printk messages to a buffer in RAM, so that after a kernel panic they can be viewed in the next kernel invocation.USB gadget driver for ADByaffs2 flash filesystemStep 4: init processInit is the very first process, we can say it is a root process, or the grandfather of all processes. The init process has two responsibilities. 1- Mounts directories like /sys , /dev or /proc 2- Runs init.rc script- The init process can be found at /init :: /system/core/init- Init.rc file can be found at :: /system/core/rootdir/Android has specific format and rules for init.rc files. More information about this rules can be found in: What is inside the init.rc and what is it used for.At this stage, you can finally see the Android logo in your screen.Step 5: Zygote and DalvikIn Java, we know that a separate Virtual Machine instance will popup in memory for separate per app, but in the case of Android, the VM should run as quick as possible for an app. But what happens if you have several apps thus launching several instances of the Dalvik (VM)?, it would consume an immense amount of memory.To overcome this problem, the Android OS has a system called Zygote. The Zygote enables code sharing across the Dalvik VM, achieving a lower memory footprint and minimal startup time. Zygote is a virtual machine process that starts at system boot. The Zygote preloads and initializes core library classes.The Zygote loading process:Load Zygote Init class: /frameworks/base/core/java/com/android/internal/os/ZygoteInit.javaregisterZygoteSocket() :: It registers a server socket for zygote command connections.preloadClasses() :: Is a simple text file that contains a list of classes that need to be preloaded, you can find the file at /framework/basepreloadResources() :: Everything that is included in the android.R file will be loaded with this method (themes and layouts).At this time, you can see the boot animation.Step 6: System serviceAfter the above steps are completed, Zygote launches the system services. The Zygote forks a new process to launch the system services.Core services:Starting power managerCreating the Activity ManagerStarting telephony registryStarting package managerSet activity manager service as system processStarting context managerStarting system contact providersStarting battery serviceStarting alarm managerStarting sensor serviceStarting window managerStarting Bluetooth serviceStarting mount serviceOther services:Starting status bar serviceStarting hardware serviceStarting NetStat serviceStarting connectivity serviceStarting Notification ManagerStarting DeviceStorageMonitor serviceStarting Location ManagerStarting Search ServiceStarting Clipboard ServiceStarting checkin serviceStarting Wallpaper serviceStarting Audio ServiceStarting HeadsetObserverStarting AdbSettingsObserverNow we have finally completed the booting process (system service are up and running in memory).Need to analyze the Android Bootup?The logcat :: Use adb to get the booting process events from the logcat.adb logcat d b events | grep bootadb logcat d | grep preloadMore information about ADB can be found here: Using ADB with your Sabre BoardThis post is targeted for software engineers wanting to dive deep into the inner workings of Android's Internals. We'll take a look at what happens when your Android device boots up, but specifically Zygote in Android and what is its role in firing up an application. We'll see how an activity is fired up in Android usingActivityManagerand who's responsible for initializing that Activity Manager, as well as what happens in the init.rc file, which starts various daemons in Android. And more such interesting questions. Knowledge of Android Internals is necessary if you want to build a career as an Android Engineer. Internals will make you aware of all the optimizations that can be used in your regular development tasks. You'll be more cautious while allocating objects and creating new activities. Apart from that, it'll also help you in interviews! Let's get started. Prerequisites There are some prerequisites for this article. You must be familiar with concepts such as: Processes in Linux What is a Virtual Machine? Extras: What are daemons in Linux? Process vs Thread How Does Android Boot Up? Here's the cycle of the bootup process in Android: This is how it goes: Boot ROM -> Bootloader -> Kernel -> Init -> Dalvik VM -> Zygote -> System Server -> Managers. Boot ROM Boot ROM contains the initial code that's run as soon as the device wakes up. It is a mask ROM or write-protected flash drive. Embedded in the CPU chip, Boot ROM loads the Bootloader into RAM for execution. Here are some examples of Bootrom that I found on StackOverflow: iPhone boot ROM. Embedded in the mask ROM and can't be modified. Loads the next stage boot loader from flash or USB (in DFU mode) and verifies its signature using built-in RSA implementation. It would also provideaccelerated decryption functions for the next stage bootloader. TI's OMAP4 boot ROM. Can load user code from flash (NOR, NAND, OneNAND), external memory, SD/MMC, USB or UART. Boot order and options are set by strap (SYSBOOT) pins. Provides some functionality for later stages (cache/TLB management etc.) NXP's LPCxxxx series Boot ROM. Placed in a hidden portion of the internal flash which is mapped at 0 on power-on. Implements CRP (code read protection), ISP (In-System Programming) which allows to upload and flash new code over UART. If a valid user code is in flash (needs to have proper checksum), maps it to 0 and jumps to it. A part of bootrom remains mapped to provide IAP (In-Application Programming) and some other services. Bootloader Bootloader is a piece of code that runs before any operating system. It is responsible for loading an operating system from the device, setting up a minimal environment in which OS can run, and beginning the startup process. Hence, the bootloader is not specific to Android. In the context of Android (pun intended!), you might've heard of OEM manufacturers placing certain limitations on the operating system (e.g. limited background processes). This is where those rules are stored. One of the major tasks of bootloader involves setting up memory management, security options. This is essential for the Kernel. Bootloader contains two important files: init.s and main.c Init.s is responsible for initializing stacks and BSS segments. It eventually calls main.c, which is responsible for initializing hardware such as a keyboard, system clock, console, etc. Most importantly, the bootloader verifies the integrity of the boot and recovery partitions before moving execution to the Kernel and displays the warnings specified in the section Boot state. Kernel Once Kernel boots up, it starts to set up the cache, load drivers, mount the file system, start Kernel daemons, etc. Once it finishes the system setup, it looks for the init process in the system files and launches the root/first process. Without Kernel, your software won't have a chance to communicate with the hardware. Init This is a very crucial process. It is where the directories such as /dev, /sys, /proc are mounted. This is also the place where init.rc script is executed. Init process also starts the daemons, such as bluetooth daemon, adb daemon, etc. These handle low-level hardware interfaces including radio interface. If you take a look at init.rc script, you'll find that it includes commands such as "start vold" for the file system and"trigger zygote" for starting the Zygote process in Android. One of the most important things that happens during this init process is a call to start the Zygote. The app_processcommand starts the ART or Dalvik VM and also gives a call to Zygote's main()method. Zygote and VM When the command app_processlaunches the Zygote, first, a VM instance is created, and then, a call to Zygote's main()function happens. According to the dictionary definition: Zygote is the first cell that's formed during fertilization. Similarly, Zygote is the first Android-specific process when Android OS boots up! Zygote preloads all the system resources and classes used by the Android framework, thus achieving fast app launches. The Zygote forks itself to start a "system-server."The system-server starts services such asActivityManagerService, hardware services, etc. It starts listening on a socket interface for future requests to spawn off new virtual machines (VM) for managing new application processes. On receiving a new request, it forks itself to create a new process that gets a pre-initialized VM instance. This forking is available due to thecopy-on-write resource management technique. It doesn't copy anything actually, just points to the pages of the parent process. The actual copying happens when there is a new write to the process' pages. Forking happens very efficiently. A new Dalvik VM is created, and then, the process gets its own thread and resources to work with. This enables code sharing among VMs, resulting in minimal startup time. System Server System Server is the first service started by the Zygote. The first thing that the system server does is start the native library called android_servers that provides interfaces to native functionalities. It starts initializing each system service and registering them with the previously started Service Manager. Each service runs in a separate Dalvik thread in system server. ActivityManageris responsible for starting the Launcher App and registering click listeners to it. It also manages the activity lifecycle, maintains activity stack, etc. System Server also starts other services beginning with com.android, such as com.android.phone and com.android.email. It also starts other managers such as Location Manager, Bluetooth Manager, etc. Activity Manager It is responsible for activity thread process creation, activity stack management, and activity lifecycle management. In the end, it launches an intent to start Home Launcher and registers on click listeners to it. Whenever a click is detected, it launches new apps from icons on the home screen. The click events are transferred toActivityManagerServicevia Binder IPC. AMS performs multiple steps: Collectsinfo about the target intent of the intent object. It does so usingresolveIntent()method. Information about the target is saved back into the intent object. Now, it's checked if the user can actually access the target using grantUriPermissionLocked()method. Then, AMS checks if the intent needs to be opened as a new task. It checks for flags such as FLAG_ACTIVITY_NEW_TASK, etc. If the process doesn't already exist for the target, the AMS has to create a new process for this. Conclusion This article will help you host CloudAnchors using Google Cloud Anchors API. Let me know if you'll face any problems and I'll be happy to help :) *Important*: Join the AndroidVilleSLACK workspace for mobile developers where people share their learnings about everything latest in Tech, especially in Android Development, RxJava, Kotlin, Flutter, and mobile development in general. Click on this link to join the workspace. Its absolutely free! You can follow me on LinkedIn, Quora, Twitter, and Instagram where I answer questions related to mobile development, particularly Android and Flutter. We have reached the fourth stage of our journey into the deep, hidden and secret world of Android and the subject this time is far from simple. The topic of this stage is the Android boot process. The Android boot process is an intricate sequence that takes place from the moment you power on an Android device to the point where the system is ready for user interaction. This process involves several key stages, each of which is crucial for initializing the hardware and software components that make up the Android operating system. In this article, we will explore each phase of the boot process in meticulous detail, highlighting the role and functioning of the components involved. Here is a timeline diagram of Android boot process. Figure 1: timeline diagram of Android boot process. And here is a more complete activity diagram of the Android boot process. Figure 2: activity diagram of the Android boot process. So let's begin the "Boot Android Process" exploration ! When the device is powered on, the first thing that happens is that the power supply changes to a powered-up state. When the power supply senses that board power levels have stabilized in the new state, it sends a RESET signal to the CPU. This signal causes the CPU to load data from boot ROM and to start executing it. This is a small piece of code that is hardwired in the CPU ASIC (Application Specific Integrated Circuit). Boot ROM (Read-Only Memory). The boot process begins with the Boot ROM, a small piece of immutable code embedded into the device's hardware, typically within the System-on-Chip (SoC). This code is the first thing executed when the device is powered on. The Boot ROM's role is generally quite limited: it will often focus on initializing some basic hardware components like clocks, power controllers, and some on-chip memory (usually internal SRAM), but not the full system memory (RAM). This is because, at this stage, it doesnt yet have access to the external DRAM (which is the main system RAM), and it doesnt yet know how much DRAM is installed or how it's configured. The Boot ROM typically uses a small amount of on-chip memory (internal SRAM) for its execution. The Boot ROMs primary job is to find and load the Primary Bootloader (PBL) (the bootloader stored in non-volatile storage) into on-chip SRAM. The Boot ROM searches for the bootloader in predefined storage locations, such as eMMC, NAND flash, or an SD card. The exact location is typically hardcoded based on the device's architecture. Once the boot media sequence is established the Boot ROM will try to load the first stage bootloader (PBL) to internal SRAM. Then once the bootloader is in place the Boot ROM code will perform a jump and execution continues in the bootloader (PBL). Here is a pseudo code example. void BootROM_Start() { Initialize_CPU(); // Set up CPU and system clocks Discovery_Bootloader(); // eMMC, NAND flash, SD card Load_Bootloader(); // Locate the bootloader on SRAM Jump_To_Bootloader(); // Transfer execution to the bootloader} The bootloader is the next critical phase in the boot process. It is a piece of software that acts as an intermediary between the Boot ROM and the operating system, preparing the environment for the Android kernel to run. Bootloader. The bootloader is a special program separate from the Linux kernel that is used to set up initial memories and load the kernel to RAM. On desktop systems the bootloaders are programs like GRUB. In embedded Linux uBoot is often the bootloader of choice. Device manufacturers often use their own proprietary bootloaders. The bootloader is often divided into two parts: the Primary Bootloader (PBL) and the Secondary Bootloader (SBL). Primary Bootloader: After the Boot ROM initializes the hardware, it loads the Primary Bootloader (PBL). The PBL is responsible for loading and executing the Secondary Bootloader (SBL). The PBL main tasks are to initializing critical hardware components, to set up the memory controller and to loading the SBL into RAM. Here is a pseudo code example. void primary_bootloader_main() { init_hardware(); // Initialize basic hardware setup_memory(); // Set up the memory controller load_secondary_bootloader(); // Load the next stage jump_to_secondary_bootloader();} Secondary Bootloader: The Secondary Bootloader (SBL) is a more complex software that performs additional hardware initialization and loads the kernel. The SBL also includes a splash screen, boot animation, or diagnostic features. It prepares the system for the Android kernel and performs tasks like verifying the integrity of the system images. The SBL includes the following tasks: Hardware Initialization and Setup: The bootloader continues the hardware initialization process, configuring additional peripherals like display controllers, input devices, and network interfaces. Partition Verification: The bootloader verifies the integrity of key system partitions such as the boot, recovery, and system partitions. This verification step may involve checking cryptographic signatures or hash values to ensure that the partitions have not been compromised. Kernel Loading and Initialization: The bootloader loads the Linux kernel into memory from the boot partition. It also passes essential parameters to the kernel, such as the location of the root filesystem and the device tree blob (DTB), which contains hardware configuration data. Fastboot and Recovery Modes: The bootloader also provides interfaces for accessing Fastboot and Recovery modes. Fastboot is a protocol that allows flashing or erasing partitions, while Recovery mode is used for system recovery tasks, such as factory resetting the device or applying updates. Let's spend a few words on these two modality. Fastboot: A system is usually forced into fastboot mode using some kind of hardware flag: flipping a DIP-switch or holding some combination of buttons during system startup. The bootloader inspects the switches as part of initialization and determines whether it should run the fastboot program or not. If the Android device is connected to a host PC through the ADB (Android Debug Bridge) it is possible to restart the system in fastboot mode via adb with the "adb-reboot-bootloader" command. Recovery Mode: As shown in Figure 2, Android devices support an alternate boot path: Recovery Mode. Unlike fastboot, recovery mode boots a complete, if limited, operating system. It has a full network stack, can mount and unmount volumes, runs adb and a shell, and so on. Although it is a complete, if minimal, Linux, recovery mode does not start the Android container. AOSP contains code for a flashable recovery image in the bootable/recovery directory. To generate the flashable image file, recovery.img, use the make target the recoveryimage: make j8 recoveryimage Here is a pseudo code example. void secondary_bootloader_main() { verify_boot_partition(); //Check boot partition integrity display_splash_screen(); // display a splash screen load_kernel_image(); // Load the Android kernel setup_Device_Tree(); // Load device tree for kernel if (Fastboot_Mode_Requested) { enter_fastboot_mode(); // fastboot mode for flashing } else if (Recovery_Mode_Requested) { enter_recovery_mode(); // recovery mode } else { jump_to_kernel(); // Transfer control to Android kernel }} The Linux kernel starts up in a similar way on Android as on other systems. It will set up everything that is needed for the system to run. Initialize interrupt controllers, set up memory protections, caches and scheduling. Kernel. The kernel is the core component of the Android operating system. It is responsible for managing the system's resources, including memory, CPU, and hardware peripherals. The Android OS uses a modified version of the Linux kernel, which is specifically optimized for mobile devices. The key functions are: System Memory Management: The kernel sets up the memory management unit (MMU) and initializes the system's memory map, which includes configuring virtual memory, paging, and allocating memory regions for different system processes. Device Driver Initialization: The kernel initializes and loads the necessary device drivers for hardware components such as the display, audio, network interfaces, and storage devices. Device drivers act as intermediaries between the hardware and the operating system, enabling communication and control. Mounting the Root Filesystem: The kernel mounts the root filesystem, which is typically an initramfs (initial RAM filesystem) loaded into memory by the bootloader. The root filesystem contains the basic tools and scripts required to initialize the user space. Launching the Init Process: The kernel starts the first user-space process, known as the init process, which has a process ID (PID) of 1. The init process is responsible for starting all other system processes and services. Here is a pseudo code example (Kernel Initialization Flow). void start_kernel(void) { setup_arch(); // Set up architecture-specific config. init_IRQ(); // Initialize the interrupts system initialize_memory_management(); //Set up virtual memory initialize_device_drivers(); // Load and init drivers mount_root_filesystem(); // Mount initramfs start_init_process(); // Launch init process (PID 1)} The init process is the grandmother of all system processes. Every other process in the system will be launched from this process or one of its descendants. Init Process. The init process is the first user-space process started by the kernel. It plays a central role in the boot process by initializing the Android user space, starting essential system services, and setting up the overall system environment. The init process mainly does four things: 1. Create some folders and mount the device: the init process mounts various system partitions, such as /system, /data, and /cache, ensuring that they are available for use by the system and applications. 2. Set up SELinux Policies: the init process also initializes SELinux (Security-Enhanced Linux) policies if the device is configured to use SELinux. This adds an extra layer of security by enforcing mandatory access controls. I will dedicate an entire article in this series to SELinux in Android. 3. Initialize and start the property service: the init process is responsible for loading Android properties, which play a crucial role in configuring the system. These properties are key-value pairs that are used throughout the Android OS to control system behaviors, manage configuration settings, and define environment variables. Android properties are essentially a lightweight configuration system that allows the OS, services, and applications to access or modify certain settings at runtime. Some examples of Android properties include: ro.build.version.sdk (defines the version of the Android SDK the device is running), ro.product.model (identifies the model of the device) and ro.secure (Indicates whether the device is running in a secure mode). Android properties share some conceptual similarities with the Windows Registry, but they are much simpler. Like the Windows Registry, Android properties can be read and modified at runtime. However, the Windows Registry is a more complex hierarchical database system that stores various types of data, while Android properties are simple key-value pairs stored in memory. Android properties are used extensively throughout the system, as an example they can define how the system behaves at runtime, like enabling or disabling developer options, secure boot settings, and more. You can query properties within the Android framework like this in Java: String sdkVersion = System.getProperty("ro.build.version.sdk"); Overall, Android properties are a key aspect of the Android system, providing a flexible and dynamic way to manage configurations, in contrast to the more complex and structured Windows Registry system. Android provides commands in these files to define or modify properties: setprop (used to set a property) and getprop (used to retrieve the value of a property). 4. Parse the init.rc configuration file and start the zigote process: the init process parses a series of configuration files (init.rc and other .rc files) that contain instructions for starting system services, mounting filesystems, setting up networking and configuring system properties. Based on the init.rc configurations, the init process starts critical system services such as vold (volume daemon for managing storage devices), netd (network daemon for handling network interfaces), and servicemanager (a core Android service manager). But first of all it runs the zygote process. Since the whole init process in Android is quite complicated, to understand it better, let's look at the files and modules responsible for the init process. As I said in previous articles, all the source code I will show comes from the Android Open Source Project version 12.1.0_r11. Most of the modules involved are located in the "/system/core/init" folder. Here we have the main.cpp file where the main method is located, within this method we could find three method calls which are pretty important: // $AOSP/system/core/init/main.cpp (simplified)...int main(int argc, char** argv) { ... if (argc > 1) { ... if (!strcmp(argv[1], "selinux_setup")) { return SetupSelinux(argv); } if (!strcmp(argv[1], "second_stage")) { return SecondStageMain(argc, argv); } } return FirstStageMain(argc, argv);} The FirstStageMain() method is mainly responsible for creating the directories and mounting the devices. We can see a simplified version of this method below. // $AOSP/system/core/init/first_stage_init.cpp (simplified)...int FirstStageMain(int argc, char** argv) { ... CHECKCALL(mount("tmpfs", "/dev", "tmpfs", MS_NOSUID, "mode=0755")); CHECKCALL(mkdir("/dev/pts", 0755)); CHECKCALL(mkdir("/dev/socket", 0755)); CHECKCALL(mkdir("/dev/dm-user", 0755)); CHECKCALL(mount("devpts", "/dev/pts", "devpts", 0, NULL)); ... CHECKCALL(mount("sysfs", "/sys", "sysfs", 0, NULL)); CHECKCALL(mount("selinuxfs", "/sys/fs/selinux", "selinuxfs", 0, NULL)); ... CHECKCALL(mount("tmpfs", "/mnt", "tmpfs", MS_NOEXEC|MS_NOSUID|MS_NODEV, "mode=0755,uid=0,gid=1000")); // /mnt/vendor is used to mount vendor-specific partitions that can not be // part of the vendor partition, e.g. because they are mounted read-write. CHECKCALL(mkdir("/mnt/vendor", 0755)); // /mnt/product is used to mount product-specific partitions that can not be // part of the product partition, e.g. because they are mounted read-write. CHECKCALL(mkdir("/mnt/product", 0755)); ... // call init process with selinux_setup as cmd line const char* path = "/system/bin/init"; const char* args[] = {path, "selinux_setup", nullptr}; execv(path, const_cast(args)); return 1;} In the FirstStageMain() method, essential low-level partitions are mounted early in the boot process to prepare the system for higher-level operations that occur in the second stage of init. These partitions are usually related to the boot environment and are necessary for loading the Android system. The primary partitions mounted during FirstStageMain() include: /system: The /system partition contains the core Android operating system files, such as the Android framework, system libraries, and system apps. It's crucial to mount this early so that the rest of the system can access necessary binaries and libraries required during the boot process. /vendor: The /vendor partition holds device-specific binaries, drivers, and configurations. These are needed early in the boot process to ensure proper hardware functionality, especially in devices with a customized hardware environment. /proc, /sys, and /dev: These are virtual filesystems that are mounted early to provide access to kernel-level information (/proc), device drivers (/dev), and system information (/sys). They are mounted very early to ensure that other parts of the system can interact with hardware and system resources. So FirstStageMain() is responsible for preparing a minimal environment where the Android OS can initialize its core components. Finally the FirstStageMain() method passes control back to the init process with "selinux_setup" as the command line, then the init process will call the SetupSelinux() method. We can see a simplified version of this method below. // $AOSP/system/core/init/selinux.cpp (simplified)int SetupSelinux(char** argv) { ... LoadSelinuxPolicyAndroid(); ... SelinuxSetEnforcement(); .... // call init process with second_stage as cmd line const char* path = "/system/bin/init"; const char* args[] = {path, "second_stage", nullptr}; execv(path, const_cast(args)); return 1;} SetupSELinux() loads the SELinux policy from the systems predefined policy files. This policy defines which operations each process and user can perform on system resources like files, devices, and sockets. Depending on the device's configuration, SetupSELinux() sets SELinux to either enforcing mode (where policy violations are blocked) or permissive mode (where violations are logged but not blocked). Enforcing mode is typically used in production devices for better security. SELinux uses labeling to associate each file and process with a security context. SetupSELinux() method ensures that all relevant files and processes on the system are labeled correctly according to the SELinux policy. These labels dictate how different entities on the system can interact. We will look at this mechanism in depth in one of the next articles in this series. Finally the SetupSELinux() method passes control back to the init process with "second_stage" as the command line, then the init process will call the SecondStageMain() method. We can see a simplified version of this method below. // $AOSP/system/core/init/init.cpp (simplified)int SecondStageMain(int argc, char** argv) { ... // Step 1: Load system properties PropertyInit() .... // Step 1: Parse init.rc and other .rc files ActionManager& am = ActionManager::GetInstance(); ServiceList& sm = ServiceList::GetInstance(); LoadBootScripts(am, sm); .... return 0;} static void LoadBootScripts(ActionManager& action_manager, ServiceList& service_list) { Parser parser = CreateParser(action_manager, service_list); std::string bootscript = GetProperty("ro.boot.init_rc", ""); parser.ParseConfig(bootscript);} // $AOSP/system/core/init/property_service.cpp (simplified)void PropertyInit() { ... mkdir("/dev/__properties__", S_IRWXU | S_IXGRP | S_IXOTH); CreateSerializedPropertyInfo(); __system_property_area_init()) property_info_area.LoadDefaultPath()) ... PropertyLoadBootDefaults(); PropertyLoadDerivedDefaults();} As we can see the key responsabilities of SecondStageMain() are: Setting Up System Properties (PropertyInit method call): System properties that configure the behavior of various components are set during this stage. These properties can control everything from security settings to hardware configurations. Parsing and Executing .rc Scripts (LoadBootScripts method call): The primary task of SecondStageMain() is to parse configuration files like init.rc, init.zygote64.rc, and other device-specific *.rc scripts. These files define the system services, properties, and actions that need to be executed to bring the system to a fully operational state. Service Initialization: After parsing the scripts, SecondStageMain() starts various services that are defined in these *.rc files. These include core Android services like vold (for managing volumes), netd (for networking), and the Zygote process. The configuration files in Android, such as init.rc, are written in a custom language known as the "Android Init Language". Android Init Language is a domain-specific language (DSL) designed specifically for Android's init process. The language is declarative and used to define system properties, services, actions, and commands executed during the boot process. The syntax is simple, with commands like service, on, start, mount, and more. This language ensures that Android's init process can configure and launch system services and manage device initialization efficiently. For example, a basic of init.rc file (located in system/core/rootdir/) looks like this: # Basic init.rc file for Android on boot # Set initial system properties setprop ro.bootmode normal # Mount essential filesystems mount_all /fstab.$(ro.hardware) # Start key services start ueventd start servicemanager start vold start zygote # Additional services and configurations The init.rc is the primary configuration file that defines the overall structure and behavior of the Android system during boot. The Key tasks in init.rc include: Mounting Filesystems: It mounts necessary filesystems like /system, /data, and /vendor. Setting System Properties: It sets crucial properties like device identifiers, debug settings, and security configurations. Starting Core Services: It defines services like vold, netd, servicemanager, Zygote and others that are required for the system to function. These services are essential for managing hardware, storage, and network functionality. I hope that the sequence diagram below can clarify the complex interactions between different modules that we have described as occurring in the init process. The process of starting Zygote in Android is crucial because it serves as the parent process for all Android applications. The init process triggers the Zygote, which is responsible for creating and launching app processes via forking. While weve said that the init process is the grandmother of all system processes, we can say that the Zygote process is the mother of all application processes and Java-based components in the Android system. Do you know why the Zygote process was given such a name? The Zygote process in Android is named after the biological term "zygote," which is the cell formed when two gametes fuse, marking the beginning of a new organism. Similarly, in Android, the Zygote process is the starting point from which new app processes are generated or "born," making the name fitting for its role in creating new processes. To tell the truth, there is an even deeper similarity: In order to reduce the application memory footprint and improve startup time, Android does not start a new ART VM process for each application. Instead, it uses a partially initialized process called zygote and forks it when it needs to start a new application. However, instead of calling one of the exec() functions like it does when starting a native process, it merely executes the main() function of the specified Java class. This process is called specialization, because the generic zygote process is turned into a specific application process, much like cells originating from the zygote cell specialize into cells that perform different functions. Heres a detailed explanation of the flow for starting Zygote, including relevant code snippets from init.rc, init.zygote64.rc (or init.zygote32.rc), app_main.cpp and also some Android Runtime modules (ART) (1.). As I mentioned above, Zygote is started by init process. Recent Androids typically start multiple copies of Zygote. Depending on the chipset architecture or OEM preference, platform developers use the system variable, ro.zygote, set at platform at build time, to control which types of Zygotes are started and which one is primary. Most modern Android devices start two zygotes, one for 32-bit applications and one for 64-bit apps, and default to 64-bit version. This is how you can read the ro.zygote property of an Android device connected via adb to the host computer. c:\bin>adb shellmydevice:/ # getprop | grep ro.zygote[ro.zygote]: [zygote64_32]mydevice:/ # The script init.rc, of which we have already seen a basic snippet, includes the script that starts Zygote, and use the ro.zygote property in order to identify which scripts to include: # init.rcimport /init.${ro.zygote}.rc...... The Zygote process is configured via the init.zygote64.rc (for 64-bit systems) or init.zygote32.rc (for 32-bit systems). These files contain additional instructions for launching Zygote. For example, in init.zygote64.rc: # init.zygote64.rcservice zygote /system/bin/app_process64 -Xzygote /system/bin --zygote --start-system-server class mainpriority -20user rootgroup root readprocsocket zygote stream 660 root systemonrestart write /sys/android_power/request_state wakeonrestart write /sys/power/state ononrestart restart audioserveronrestart restart cameraserveronrestart restart mediaonrestart restart netdonrestart restart wificond we have the following highlights: service zygote /system/bin/app_process64: This line defines the Zygote service, and the app_process64 binary is invoked. This binary launches the Zygote process. -Xzygote: Indicates that the process being launched is Zygote. --start-system-server: Tells Zygote to start the Android System Server, which is responsible for managing key system services. socket zygote: Creates a socket named zygote for communication between Zygote and the system. This is the socket that the system will use to start new Android applications. The actual application that is started as user root at the very highest priority by init is /system/bin/app_process64. The script requests that init create a stream socket for the process and catalog it The script also contains an onrestart directive that will restart the Zygote if it fails. Therefore the command defined in the init.zygote64.rc file invokes /system/bin/app_process, which is responsible for launching Zygote. The first thing to do when starting Zygote is to initialize the ART runtime environment. It's important to note that Zygote is started only once, during the system boot. Any subsequent app launch simply clones the running Zygote instance. The applications app_process64 and app_process32 are small programs that parse their command line arguments and then either execute Zygote or the class whose name was passed as an argument. These programs are used to launch all processes (not apps!) compiled into DEX: anything that requires Dalvik or ART. They serve a purpose very similar to the java command, which starts the Java runtime. Here is the relevant part of app_main.cpp: // $AOSP/frameworks/base/cmds/app_process/app_main.cppint main(int argc, char* const argv[]) { // Snip: Command-line arguments parsing logic if (zygote) { // Initialize Zygote runtime.start("com.android.internal.os.ZygoteInit", args); } else if (startSystemServer) { // Start System Server along with Zygote runtime.start("com.android.internal.os.ZygoteInit", args); } // Snip: Handling other conditions} The app_process creates an instance of AppRuntime, a subclass of AndroidRuntime. It does a lot of thinks, setting up any runtime parameters (runtime.addOption()), the name for the process (runtime.setArgv0()), and the name of the class to run (used only when not running Zygote: runtime.setClassNameAndArgs()), and then calls AndroidRuntime.start() to invoke the runtime. The start method invokes startVM, look at this code snippet: // $AOSP/frameworks/base/core/jni/AndroidRuntime.cpp void AndroidRuntime::start(const char* className, const Vector& options, bool zygote){ .../* start the virtual machine */ JniInvocation jni_invocation; jni_invocation.Init(NULL); JNIEnv* env; if (startVm(&mJavaVM, &env, zygote, primary_zygote) != 0) { return; } onVmCreated(env); /* * Register android functions. */ if (startReg(env) < 0) { ALOGE("Unable to register all android natives"); return; } ... /* * We want to call main() with a String array with arguments in it. * At present we have two arguments, the class name and an option string. * Create an array to hold them. */ jclass startClass = env->FindClass(className); jobjectArray strArray = env->NewObjectArray(options.size() + 1, stringClass, NULL); jmethodID startMeth = env->GetStaticMethodID(startClass, "main", "([Ljava/lang/String;)V"); if (startClass == NULL){ env->CallStaticVoidMethod(startClass, startMeth, strArray); }} After processing the options, startVM invokes JNI_CreateJavaVM. The JNI_CreateJavaVM method is defined in art/runtime/jni/java_vm_ext.cc. It, in turn, calls Runtime::Create, which is defined in art/runtime/runtime.cc. Finally, Runtime::Create initializes the ART runtime, loading the system OAT files and the libraries they contain (In previous articles we talked about the OAT format: OAT files in Android are optimized executable files generated by the Android Runtime (ART) or the Dalvik Virtual Machine (DVM) during the process of compiling application code) (2.). // $AOSP/art/runtime/jni/java_vm_ext.ccextern "C" EXPORT jint JNI_CreateJavaVM(JavaVM** p_vm, JNIEnv** p_env, void* vm_args) { .... Runtime::Create(options, ignore_unrecognized) ....} // $AOSP/art/runtime/runtime.ccbool Runtime::Create(const RuntimeOptions& raw_options, bool ignore_unrecognized) { ... Runtime::Init(runtime_options) ...} At this point, the Android VM is ready to be started. The argument passed by app_process during its startup calls was com.android.internal.os.ZygoteInit. Its source can be found in /frameworks/base/core/java/com/android/internal/os/ZygoteInit.java. AndroidRuntime::Start then call the CallStaticVoidMethod method of virtual machine (env object) and passing it the className parameter which in our case is ZygoteInit. The key method in ZygoteInit.java is main(), the entry point for the Zygote process. It handles the creation of the Zygote server socket and listens for requests to fork new processes and start SystemServer which is essential for managing the Android system. Heres a snippet from ZygoteInit.java: // $AOSP/frameworks/base/core/java/com/android/internal/os/ZygoteInit.javapublic static void main(String argv[]) { // Initialize the Zygote server try { ... zygoteServer = new ZygoteServer(isPrimaryZygote); .... // preload resources and libraries ... if (startSystemServer) { Runnable r = forkSystemServer(abiList, zygoteSocketName, zygoteServer); // {@code r == null} in the parent (zygote) process, and {@code r != null} in the // child (system_server) process. if (r != null) { r.run(); return; } } Log.i(TAG, "Accepting command socket connections"); // The select loop returns early in the child process after a fork and // loops forever in the zygote. caller = zygoteServer.runSelectLoop(abiList); ... } catch(Exception e) { ... }} Most of the code in ZygoteInits main method (the method the runtime calls, once initialized) is wrapped in a Java try/catch block. The obvious reason is to allow Zygote to attempt to clean up after a failure. Zygote has three major tasks, on startup: Register the socket to which the system will connect to start new applications. Preload Android resources and libraries, if necessary. Start the Android System Server. After it has completed these three tasks, it enters a loop, waiting for connections to the socket. The first of the tasks, opening the socket, is handled in the registerServerSocket method. The method creates the socket using the name passed in the parameters from the init.zygote64.rc script originally. The second task is accomplished with a call to the preload method. The preload method loads everything it can think of: classes, libraries, resources. Note that, when using the ART runtime, many of the preloads are already in memory because they are in the system OAT file. At the completion of the preload method, Zygote is fully initialized and ready to clone new applications very quickly, sharing most of its virtual memory with them. The last of the tasks, starting the System Server, is performed if the startSystemServer parameter passed to the main method of ZygoteInit is TRUE. We have seen that init.zygote64.rc sets this parameter to true, so SystemServer is the first application cloned by Zygote. I hope the sequence diagram below helps clarify what has been explained so far, "they say a picture is worth a thousand words". Now that weve uncovered the intricate sequence of interactions between modules that initiate the Zygote process, its worth exploring the mechanism known as specialization, where the generic Zygote process is transformed into a specific application process, just as in biology, cells originating from the zygote specialize into cells with different functions. How does Zygote Works? Zygote plays a crucial role in Android's architecture, serving as the primary process that starts and manages all application processes. It is a special process that, as we just saw, is initialized during the boot process and acts as a template or parent for every new app process, ensuring efficient memory use and faster app startup by leveraging process forking. The key roles of Zygote in Android are: Preloading Common Resources: Zygote loads system libraries, classes, and resources that are shared across all applications. This reduces the overhead of loading these resources each time a new app starts, allowing

for efficient memory sharing. Forking New Processes: Zygote does not directly launch apps but instead forks new processes when needed. This means that new app processes are clones of the Zygote process. The forked process then initializes its specific application code. Starting System Server: Zygote also launches the System Server, which is a key process responsible for starting core system services like activity management, power management, and input handling. But how Zygote Works? Lets look at some key components from the ZygoteInit class to understand its role more deeply. The entry point for the Zygote process is the main() method of the ZygoteInit class. This method starts the Zygote server and sets up the environment for process forking. Here is a key part of the ZygoteInit class: // $AOSP/frameworks/base/core/java/com/android/internal/os/ZygoteInit.javaprivate static void preload() { ... // Preloads commonly used Java classes preloadClasses(); // Loads system resources such as images, fonts, and other assets preloadResources(); preloadOpenGL(); // Loads native shared libraries (e.g., libc, libm, etc.). preloadSharedLibraries(); ...} By preloading these components, Zygote ensures that all future processes forked from it have immediate access to these shared resources, which saves both memory and time. When Zygote receives a request to launch a new app, it forks a new process. The zygote.fork() method (a native method) is used to fork the new process, inheriting the Zygote state. Heres the critical part of the ZygoteConnection.java file that handles the fork request: pid = Zygote.forkAndSpecialize(parsedUid, gid, gids, runtimeFlags, rlimits, mountExternal, seInfo, niceName, fdsToClose, fdsToIgnore, instructionSet, appDataDir); Zygote.forkAndSpecialize() method forks the Zygote process to create a new child process, which will become the app process. niceName: The name of the new process (usually the app package name). parsedUid: The user ID for the new process (each app runs with a unique UID for security). gids: Group IDs to ensure proper permissions. After forking, the child process runs the RuntimeInit.main() method to load and run the specific application code. Once the Zygote has forked a new process, the child process enters the RuntimeInit class. Here, the application environment is initialized, and the actual app code is started. public static void main(String[] args) { // Step 1: Initialize system state (e.g., logging, exceptions) commonInit(); // Step 2: Call the main method of the target app invokeStaticMain(args);} commonInit(): Initializes system-level settings such as logging and exception handling. invokeStaticMain(): This method calls the main() function of the target app, passing any required arguments. At this point, the child process starts running the actual application code, while Zygote continues waiting for more fork requests. So to recap heres a simplified flow of how Zygote handles a request to launch a new app: Zygote receives a request to launch a new app (via socket communication). Zygote forks a new process using forkAndSpecialize(). The new process inherits the preloaded resources (libraries, classes, etc.) from Zygote. The new process enters RuntimeInit and starts executing the app-specific code. Zygote continues running and waits for more fork requests. I would like to conclude this first part on the Android startup process here. We have covered many topics, perhaps too many for just one phase of our journey into the hidden world of Android. In the next episode, we will continue our exploration of the Android startup process, starting from what happens after the Zygote process is initiated and following it through to the point where the home screen appears and the user can start launching applications and using the device I remind you my newsletter "Sw Design & Clean Architecture" : where you can find my previous articles and where you can register, if you have not already done, so you will be notified when I publish new articles. Thanks for reading my article, and I hope you have found the topic useful, Feel free to leave any feedback. Your feedback is very appreciated. 1. G.Meike, L.Schiefer, Inside the Android OS Addison Wesley (August 2021). Have you ever wondered what happens when you turn on your Android device? Have you ever asked:What actually goes on behind the screen when that boot animation or splash screen appears?How exactly does a device boot an Android Operating System (OS)?The answer is the Android Boot Process.What is the Android Boot SequenceThe Android Boot Sequence happens each time an Android device is powered on. For example, if you press and hold the Power key on a Pixel device briefly, you will be given the Power menu in which you can tap Power Off, or Restart. The boot sequence will either begin when you power the device on again, or almost immediately if you chose Restart. 1. Boot ROM code execution 2. The bootloader 3. The Linux kernel 4. The Init process 5. Zygote and Dalvik VM 6. The System Server and ManagersBoot ROM code executionExecution begins when the Android device is powered on. This boot ROM code is specific to the SOC the device is using. During this stage 2 things happen:When the boot ROM code is executed, it initializes the device hardware and tries to scan and detect the boot media. This is a pre-defined location which is hardwired in the boot ROM. Its almost similar to the BIOS function in the boot process of a computer.Once the boot sequence is set, the initial boot loader is copied to the internal RAM. After this, it starts executing code loaded in RAM.The bootloader is a small program which runs before Android OS starts to function. Surprisingly, it is not part of the Android Operating System. This program is where the OEM would normally put their locks and restrictions.Bootloaders are present in desktop computers, laptops, and mobile devices and their functions are similar. On Android Devices' the bootloader is divided into 2 stages:Initial Program Load (IPL)Second Program Load (SPL)Initial Program Load (IPL)IPL deals with detecting and setting up the external RAM which helps in SPL. Once the external RAM is available, SPL is copied into the RAM and execution is transferred to it.Second Program Load (SPL)The SPL is responsible for loading the main Android operating system and provides access to other boot modes, such as fastboot and recovery. The SPL also initializes several hardware components such as the display, keyboard, file systems, virtual memory, consoles and other features. After this, SPL then attempts to find the Linux kernel. Since the kernel is found in boot media it will copy from there to RAM. Following this, SPL will transfer execution to the kernel.Linux KernelThe Android kernel starts in a similar way as the Linux kernel. As the kernel fires up, it starts to setup cache, protected memory, scheduling and loads drivers. When the kernel finishes the system setup, it looks for init" in the system files.If Android kernel is similar to Linux kernels then what is difference between these two ? Lets find out below!Binder: An Android specific inter-process communication mechanism and remote method invocation system.Android Shared Memory (ASHMEM): New shared memory allocator, similar to POSIX SHM but with a different behavior and sporting a simpler file-based API.Process Memory Allocator (PMEM): Used to manage large (1-16+ MB) physically contiguous regions of memory shared between user space and kernel drivers.Logger: Kernel support for the logcat command.Paranoid Networking: Restricts access to some networking features depending on the group of the calling process.Wake locks: A way for an app to keep the CPU/screen/other things awake when the phone is idle in order to perform a specific background task. It is used for power management files. It holds the machine awake on a per-event basis until wake lock is released.OOM Handling: Has the unique task of checking if the system has enough memory available to execute tasks, verify when the system is running low on memory, and kill processes to free up memory.Alarm Manager: Allows user space to communicate with the kernel when it would like to wake up.Timed output / Timed GPIO: GPIO is a mechanism to allow programs to access and manipulate GPIO registers from user space.RAM Console: An area in RAM which is reserved at boot. This area is a persistent property, and is used to store the last kernel log messages in the event the kernel reboots or crashes. Logs stored here can be useful for kernel debugging, granting insight into the kernel process immediately prior to a crash or unexpected reboot. Viewed from /proc/last_kmsgUSB Gadget Driver for ADByaffs2 flash filesystemMany more!Init ProcessInit is the very first process, we can say it is a root process, or the grandfather of all processes. The Init process has two responsibilities.Mounts directories like /sys, /dev or /procRuns init.rc scriptThe init process can be found at /init :: /system/core/initThe init.rc file can be found at :: /system/core/rootdir/Android has a specific format and rules for init.rc files.At this stage, you can finally see the Android logo on your screen.Zygote and DalvikIn Java a separate Virtual Machine instance will popup in memory to separate per app. Androids Dalvik (the virtual machine) is different because it needs to be as quick as possible. That poses a problem.What happens if you have several apps launching several instances of the Dalvik, exhausting the available memory stores?Android solves this with a system called Zygote.Zygote is one of the first init processes created after the device boots. Zygote enables code sharing across the Dalvik VM achieving a lower memory footprint and minimal startup time. It is a virtual machine process that starts at system boot and tries to create multiple instances to support each Android process. Zygote preloads and initializes core library classes.The Zygote loading processLoad ZygoteInitclass: This loads the ZygoteInit class. Source Code: /frameworks/base/core/java/com/android/internal/os/ZygoteInit.javaregisterZygoteSocket(): This registers a server socket for zygote command connections.preloadClasses(): This is a simple text file containing a list of classes that need to be preloaded will be executed here. This file can be seen at this location: /frameworks/base.preloadResources(): This deals with native themes and layouts and everything that includes the android.R file will be loaded using this method:You will now see the boot animation!System serviceZygote launches the system services and forks a new process to launch. Core servicesStarting power managerCreating the Activity ManagerStarting telephony registryStarting package managerSet activity manager service as system processStarting context managerStarting system contact providersStarting battery serviceStarting alarm managerStarting sensor serviceStarting window managerStarting Bluetooth serviceStarting mount serviceOther servicesStarting status bar serviceStarting hardware serviceStarting NetStat serviceStarting connectivity serviceStarting Notification ManagerStarting DeviceStorageMonitor serviceStarting Location ManagerStarting Search ServiceStarting Clipboard ServiceStarting checkin serviceStarting Wallpaper serviceStarting Audio ServiceStarting HeadsetObserverStarting AdbSettingsObserverWe have finally completed the booting process: the system services are up and running in memory. At this point the system will send a broadcast action called ACTION_BOOT_COMPLETED which informs all dependent processes that the boot process is completed.After this device displays the Home screen and is ready to interact with the user.

Android boot process explained. What are the steps of booting process. Android booting sequence. Android boot process.