



SISTEMAS OPERATIVOS REALES: ANDROID

TRABAJO PRÁCTICO

INTEGRANTES DEL GRUPO

Alvarez Gorozito, Elias Santiago
Arias, Mauro Sebastián
Bialas, Marcos Ariel
Calvo, Juan Carlos
Franchesquez, Juan Pablo
Lardo, Miguel Angel
Lopez Olivera, Selva Macarena
Mastrocolo, Mariano Sebastian
Mitra, Juan Cruz
Ramirez, Agustin

Índice

INTRODUCCIÓN	2
HISTORIA	3
LICENCIAS	5
CPU.....	12
SISTEMA OPERATIVO.....	21
Carga SO pc	21
ARQUITECTURA DE ANDROID.....	37
KERNEL	44
Tipos de kernel	44
SHELL.....	48
Scheduler y Dispatcher	49
PROCESOS.....	53
Gestión de Procesos en Android.....	58
Tipos de procesos en Android:	58
Jerarquía de Procesos de Android	59
Planificación de procesos.....	60
.....	64
SWAP.....	65
MEMORIAS	72
MÁQUINA VIRTUAL JAVA	77
Gestión de Memoria en Android.....	77
REDES	82
Seguridad con protocolos de red.....	91
Configuración de seguridad de la red.....	92
Como administrar el uso de la red.....	93
FILE SYSTEM	110
Conclusiones:.....	124
Bibliografía:.....	125

INTRODUCCIÓN

I'm going to destroy Android, because it's a stolen product. I'm willing to go thermonuclear war on this.

—Steve Jobs, Apple Inc.

“Free software” is a matter of liberty, not price. To understand the concept, you should think of “free” as in “free speech,” not as in “free beer.”

—Richard M. Stallman

El sistema operativo Android se encuentra clasificado dentro de los sistemas operativos móviles, basados en una versión modificada del núcleo de Linux, los cuales forman parte de la familia de sistemas Unix-like, permitiendo ser eficiente y seguro con las características de este núcleo.

Se logra destacar de entre los sistemas operativos móviles, este se destaca por su primordial código abierto, lo que permite a los desarrolladores y fabricantes modificar y distribuir las versiones personalizadas del sistema, incentivando, así la creación de ecosistemas diversos con varios dispositivos y aplicaciones únicas.

Este sistema operativo se remarca una gran diferencia con el sistema de iOS de Apple, ya que este es cerrado y completamente controlado por Apple el cual maneja tanto el hardware como el software, garantizando así una experiencia quizás un poco más homogénea.

HISTORIA

Uno de los creadores de Android es Andy Rubin. El año donde “nació” este S.O. fue en el 2003, en Aplo Alto, California. Android no se dio a conocer hasta el 2005, cuando Google la compró, y recién en 2007 se expandió más la información al respecto, cuando se implementó la primera plataforma para dispositivos móviles. Más en detalle, la primera distribución fue el 5 de noviembre del 2007, siendo en un comunicado de prensa de la Open Handset Alliance (Alianza del Dispositivo Móvil Abierto) su primer anuncio al mundo, en su versión de Android 1.0.

En otras palabras, al mismo tiempo que se anunciaba la formación de la Open Handset Alliance el 5 de noviembre de 2007, la OHA presentó Android, una plataforma de código libre para teléfonos móviles basada en el núcleo operativo de linux.

Con la implementación de Android se eliminó la necesidad de implementar las aplicaciones del teléfono y el middleware. Esto posibilitó que las compañías creen nuevos productos pero enfocándose en mejorar el hardware.

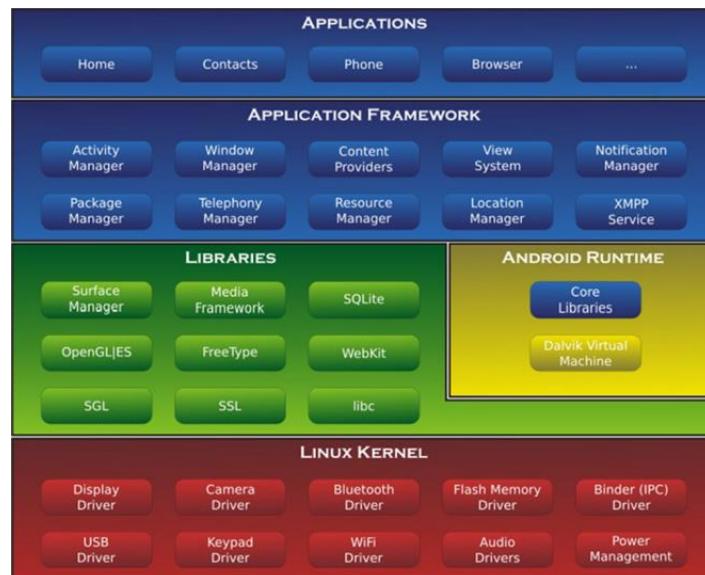
No solo las compañías se beneficiaron, también los desarrolladores de aplicaciones se vieron con ventajas ya que con pocos cambios en su código las aplicaciones funcionan en una gran multiplicidad de dispositivos.

Android es de código abierto y utiliza las licencias de Apache. Existen múltiples licencias de código abierto, Apache es una de ellas. Esto significa que se otorga la posibilidad de usar el software para cualquier propósito, distribuirlo y modificarlo libremente.

Android puede dividirse en 5 estructuras primarias:

1. Aplicaciones
 - a. Existen en el más alto nivel.
 - b. Tiene herramientas nativas, como mensajería, navegadores, etc.
 - c. El principal lenguaje de programación es JAVA.
 - d. Se podían encontrar aplicaciones de terceros en Marketplace, y actualmente en Google Play Store.
2. Framework
 - a. Los programadores pueden acceder a las API que se utilizan para las aplicaciones principales.
 - b. También puede acceder a la mayoría de las bibliotecas de JAVA que existen.
3. Librerías nativas
 - a. Son binarios de C y C++ de los que android depende.

- b. Por ejemplo SQLite es una base de datos que almacena información de las sesiones.
 - c. Otro ejemplo es OpenGL/ES el cual es un motor gráfico en Android.
4. Runtime de android (se divide en dos)
- a. 4.1 Core Java Libraries.
 - b. 4.2 Dalvik Virtual Machine.
5. Kernel
- a. En los inicios el Kernel era el 2.6 de Linux. Optimizado para el uso móvil.
 - b. Esto fue evolucionando hasta el Kernel 3.1 de linux.
 - c. El Kernel posibilita el acceso más cercano posible el Hardware.
 - d. Radio interna, Cámara, manejo de la batería, el uso del teclado físico son ejemplos de los controladores internos a los que se accede mediante el Kernel.



Para finalizar la historia, vamos a comentar resumidamente las diferentes versiones de Android que salieron al mercado.

- **Astro(1.0):** Se lanzó en noviembre del 2007 en su versión Beta y en el 2008 la final.
- **Cupcake(1.5):** Se lanzó en abril del 2009. Usaba el Kernel 2.6.27 de Linux.
- **Donut(1.6):** Se lanzó en Septiembre del 2009. El Kernel era el 2.6.29 de Linux.
- **Éclair(2.0/2.1):** La 2.0 Se lanzó en Octubre del 2009. Usaba el Kernel 2.6.29 de Linux. A consecuencia de unos Bugs se actualizó a la versión 2.1 en Enero del 2010.

- **Froyo(2.2.X):** Se lanza en Mayo del 2010 la V2.2. Usaba el Kernel 2.6.32 de Linux. Para resolver Bugs y actualizaciones de seguridad se lanzaron la versión 2.2.1, 2.2.2, y la última 2.2.3 en Noviembre del 2011.
- **Gingerbread(2.3.X):** Se lanza en Diciembre del 2010. Usando el Kernel 2.6.35 de Linux. Se Actualizó hasta la versión 2.3.7 en Septiembre del 2011.
- **Honeycomb(3.X):** Este apareció en la primera tablet con Android en Febrero del 2011 en la Motorola Xoom. Honeycomb obtuvo 6 actualizaciones, 4 mejoras pequeñas y dos importantes. La versión 3.1 fue una mejora importante en Mayo del 2011, pero la más robusta fue la segunda mejora para la versión 3.2 en Julio del 2011.
- **Ice Cream Sandwich(4.0.X):** Se lanzó en octubre del 2011. Utiliza el Kernel 3.0.1 de Linux. La última actualización fue en marzo de 2012.
- **Jelly Bean (4.1.X):** Se lanzó en Julio de 2012, utilizando el Kernel 3.1.10 de Linux. Se actualizó hasta la versión 4.3 lanzada en Julio del 2013.
- **KitKat(4.4.X):** La versión 4.4 fue lanzada en Septiembre del 2013.
- **Lollipop (5.0):** La versión 5.0 fue lanzada en Noviembre del 2014.
- **Marshmallow(6.0):** La versión 6.0 fue lanzada en el año 2015.
- **Nougat(7.0):** La versión 7.0 fue lanzada en el año 2016.
- **Oreo(8.0):** La versión 8.0 fue lanzada en el año 2017.
- **Pie(9):** La versión 9 fue lanzada en Agosto del año 2018.
- **Android Q(10):** La versión 10 de Android fue lanzada en Septiembre del Año 2019.
- **Android 11:** Fue lanzado en Septiembre del año 2020.
- **Android 12:** La versión fue lanzada en Octubre del año 2021.
- **Android 13:** Fue lanzada en Agosto del año 2022.
- **Android 14:** Fue lanzada en Octubre del año 2023.
- **Android 15:** fue lanzada el 3 de Septiembre del año 2024

LICENCIAS

El Proyecto de Código Abierto de Android (AOSP) utiliza algunas licencias aprobadas por [iniciativas de código abierto](#) para nuestro software.

Licencia del AOSP

El AOSP (Android Open Source Project) utiliza principalmente la Licencia Apache Versión 2.0 (Apache 2.0) como su licencia preferida, y la mayoría del software de Android está bajo esta licencia. Sin embargo, hay algunas excepciones que se manejan caso por caso. Por ejemplo, los parches del kernel de Linux se distribuyen bajo la licencia GPLv2 con excepciones de sistema, información que se puede encontrar en [kernel.org](#).

Contratos de Licencia de Colaboradores

Para contribuir al Android Open Source Project (AOSP), los colaboradores individuales deben firmar un **Contrato de Licencia para Colaboradores Individuales**. Este contrato, que puede completarse en línea, establece claramente las condiciones para contribuir con propiedad intelectual al AOSP, protegiendo tanto al colaborador como al proyecto. Este acuerdo no limita el derecho del colaborador a usar sus contribuciones para otros fines.

Para empresas que tienen empleados trabajando en el AOSP, existe un **Contrato de Licencia para Colaboradores Corporativos**. Este contrato permite que la empresa autorice las contribuciones de sus empleados y otorgue licencias de derechos de autor y patentes. Sin embargo, incluso en este caso, los empleados deben firmar su propio Contrato de Licencia para Colaboradores Individuales para cubrir cualquier contribución que no pertenezca a la empresa. Estos acuerdos se basan en los utilizados por la Apache Software Foundation.

Descripción general del AOSP

El AOSP (Android Open Source Project) es la versión de código abierto de Android, diseñada para que cualquiera pueda crear variantes personalizadas del sistema operativo para sus propios dispositivos. Está pensado para evitar un control centralizado, permitiendo la innovación libre de cualquier restricción. El AOSP ofrece un sistema operativo completo y de calidad, listo para ser personalizado y adaptado. La documentación del AOSP proporciona guía a los desarrolladores que desean comenzar a trabajar con la plataforma.

Restricciones del AOSP y GMS: Aunque el código fuente de Android es abierto (AOSP), el uso de Google Mobile Services (GMS) requiere cumplir con el programa de compatibilidad de Google, restringiendo el uso de servicios de Google solo a dispositivos certificados.

EL ROL DE GOOGLE EN ANDROID

Google supervisa el desarrollo de Android, garantizando que sea una plataforma de software competitiva y atractiva para los usuarios. La apertura del código es fundamental para el éxito a largo plazo de Android, ya que fomenta la inversión de desarrolladores y asegura condiciones equitativas en el mercado. Para ello, Google destina recursos de ingeniería profesionales, lo que permite el lanzamiento de dispositivos de calidad que ejecuten el sistema operativo.

El objetivo principal de Google es crear un ecosistema eficaz alrededor de Android, abriendo su código fuente para que cualquier persona pueda modificar y distribuir el software según sus necesidades. Esto asegura que el código fuente de Android permanezca relevante y atractivo, evitando que se convierta en un producto poco exitoso.

DESARROLLO PRIVADO DE ANDROID

Algunas partes de Android se desarrollan en privado para abordar la complejidad del lanzamiento de nuevos dispositivos. Dado que el desarrollo de un dispositivo puede tardar más de un año, es crucial que los fabricantes ofrezcan un software actualizado y que los

desarrolladores no tengan que seguir constantemente nuevas versiones. Para solucionar este problema, se crea una rama privada donde se desarrollan las APIs de la próxima versión de Android, permitiendo que los desarrolladores y OEM utilicen una única versión estable.

Aunque algunas partes del sistema se desarrollan de forma abierta, la intención es trasladar más componentes al desarrollo abierto con el tiempo.

RELACION ENTRE AOSP Y EL PROGRAMA DE COMPATIBILIDAD DE ANDROID

AOSP (Android Open Source Project) mantiene el software de Android y desarrolla nuevas versiones. Como es de código abierto, el software se puede usar para cualquier fin, incluyendo el desarrollo de dispositivos no compatibles con otros dispositivos basados en la misma fuente.

El Programa de compatibilidad de Android establece un modelo de referencia que asegura que los dispositivos sean compatibles con aplicaciones de terceros. Solo los dispositivos que cumplen con estos requisitos pueden participar en el ecosistema de Android, lo que incluye el acceso a Google Play. Esto permite distinguir entre dispositivos compatibles y aquellos que simplemente ejecutan versiones derivadas del código fuente.

COLABORACION CON ANDROID

Los interesados pueden colaborar con Android informando errores, desarrollando aplicaciones o contribuyendo con código fuente al AOSP. Sin embargo, existen límites en los tipos de contribuciones aceptadas. Por ejemplo, contribuciones que impliquen bibliotecas GPL o LGPL que sean incompatibles con los objetivos de licencia de Android no serán aceptadas.

CONFIRMACIONES EN ANDROID

AOSP no tiene un sistema de confirmación tradicional. Todas las contribuciones, incluidas las de empleados de Google, son gestionadas a través de un sistema web llamado Gerrit, que trabaja en conjunto con Git. Un responsable de aprobación, generalmente un empleado de Google, debe aceptar todos los cambios, asegurando una gestión organizada de las contribuciones de código.

El sistema operativo Android ha generado una auténtica revolución en el mundo de la tecnología móvil, un fenómeno que no fue anticipado por ninguna consultora del sector. Android se basa en el kernel de Linux y es desarrollado por la Open Handset Alliance (OHA), un consorcio de fabricantes de hardware y software que se dedica a establecer estándares abiertos para dispositivos móviles. La OHA es la responsable de orientar la evolución de la plataforma Android, que se caracteriza por su modelo de código abierto.

Una de las características más significativas de Android es que la mayor parte de su código está registrado bajo la licencia Apache, una licencia de software libre que permite a cualquier

persona estudiar, modificar y distribuir el sistema operativo. Esta apertura contrasta con la naturaleza propietaria de otros sistemas operativos móviles como Windows Mobile, Blackberry OS e iOS, que no permiten el acceso al código fuente ni modificaciones por parte de los usuarios. Como resultado, Android se beneficia de una activa comunidad de desarrolladores independientes que crean aplicaciones para ampliar la funcionalidad de los dispositivos, lo que enriquece la plataforma y proporciona a los usuarios una variedad casi infinita de opciones.

El manejo del mercado de aplicaciones es otro aspecto en el que Android se distingue de sus competidores. A través de la Android Market, ahora conocido como Google Play, cualquier desarrollador independiente puede publicar nuevas aplicaciones, simplificando así el proceso para acercar el software a los usuarios. Esta accesibilidad ha sido clave para el crecimiento de la plataforma, fomentando un ecosistema en el que tanto desarrolladores como usuarios pueden beneficiarse de una amplia gama de aplicaciones y servicios.

La Open Handset Alliance no fabrica su propio hardware, a diferencia de compañías como Apple y Nokia, sino que ofrece su sistema operativo de forma gratuita para que otros fabricantes de dispositivos lo adapten a sus necesidades específicas. Esta política ha llevado a que Android se convierta en un estándar de facto para muchos fabricantes de teléfonos. Para enero de 2011, ya existían en el mercado 74 modelos de teléfonos que utilizaban Android, con muchos más en desarrollo, lo que ha contribuido a su popularidad y adopción masiva.

Además, el sistema de mejoras continuas de Android ha sido fundamental en su evolución. Desde su versión 1.0 hasta la 3.0, el sistema ha evolucionado de manera rápida y constante, incorporando actualizaciones y mejoras que han mejorado la experiencia del usuario. Al integrar un kernel de Linux, Android se beneficia de años de desarrollo y depuración continua por parte de la comunidad de software libre, lo que asegura su robustez y fiabilidad.

Licencia Apache 2.0

La licencia Apache 2.0 es fácil de implementar en proyectos que no son de Apache y permite incluir el código mediante referencia, en lugar de requerir que se incluya en cada archivo. Esta licencia también exige el reconocimiento de las posibles patentes que el contribuyente haya utilizado. Actualmente, en plataformas como SourceForge, existen más de 5000 proyectos que utilizan esta licencia y que no son de Apache. La compatibilidad del código de Android con la licencia Apache es notable, siendo compatible con la GPLv3, pero no con la GPLv2.

Las licencias de software libre se definen por las cuatro libertades fundamentales que otorgan a los usuarios. Estas son:

1. **Libertad de uso** del programa para cualquier propósito.
2. **Libertad para estudiar y modificar** el programa según las necesidades del usuario.
3. **Libertad para hacer copias** del programa y distribuirlas.

4. Libertad para redistribuir versiones modificadas del programa.

Si se permite realizar estas acciones, se considera que se está ante una licencia de software libre. Para garantizar estas libertades, es fundamental que el código fuente sea accesible.



En 2007, Google presentó la Open Handset Alliance (OHA), que incluye una variedad de operadores de redes, fabricantes de semiconductores, desarrolladores de software y proveedores de teléfonos. El resultado del trabajo de la OHA es un software libre y abierto, sin costo para los usuarios, que está sujeto a la licencia open source de la Apache Software Foundation. Esto significa que los desarrolladores pueden modificar el código base del sistema y poner sus mejoras a disposición del público para su evolución posterior. La apertura del código fuente y su naturaleza de software libre permiten que tanto desarrolladores aficionados como profesionales se beneficien de esta base, enriqueciendo la plataforma con sus contribuciones.

Importantes figuras en el movimiento del software libre, como Richard Stallman, han afirmado que el kernel de Android, que es genuinamente Linux y está bajo la GPL versión 2, está obligado a publicar su código fuente. Sin embargo, otros componentes del sistema se publican bajo la licencia Apache 2.0, lo que permite a Google, por ejemplo, mantener cierta discreción sobre el código que no se desea hacer público.

El modelo de licenciamiento de Android, con su combinación de código abierto y una comunidad activa de desarrolladores, ha sido fundamental para su éxito y expansión en el mercado de dispositivos móviles, promoviendo un ecosistema donde la innovación y la colaboración son la norma.

AOSP Y GMS

El proyecto de código abierto android (aosp) fue presentado en 2007 coincidiendo con el lanzamiento comercial inicial de Android por parte de Google para proporcionar un sistema

operativo gratuito y de código abierto para dispositivos móviles, permitiendo a fabricantes y desarrolladores personalizar e innovar en la plataforma Android.

AOSP proporciona el código base del sistema operativo Android, que cualquiera puede utilizar, modificar y distribuir libremente. Esta naturaleza de código abierto ha sido crucial para fomentar un vasto ecosistema de dispositivos Android, permitiendo a una amplia gama de fabricantes crear sus versiones de Android sin depender de los servicios propietarios de Google.

Esta apertura de AOSP ha permitido directamente el desarrollo de versiones comerciales de Android sin GMS. Aprovechando el código base de AOSP, las empresas pueden crear sistemas operativos Android personalizados que omitan Google Mobile Services (GMS) y utilicen en su lugar tiendas y servicios de aplicaciones alternativos.

Esto ha sido especialmente significativo en regiones donde los servicios de Google están restringidos o donde las empresas buscan diferenciarse y/o reducir la dependencia de Google, como con HarmonyOS de Huawei o varios smartphones chinos que utilizan ecosistemas alternativos. Esta flexibilidad ha permitido a Android convertirse en una plataforma global sostenible que se mantiene adaptable a las diversas necesidades del mercado y a los entornos normativos.

Android sin GMS se refiere a los dispositivos Android que no incluyen las aplicaciones y servicios propiedad de Google. Estos dispositivos suelen utilizar sistemas operativos, tiendas de aplicaciones y servicios alternativos para ofrecer funcionalidades similares. Ejemplos de ello son algunos smartphones chinos como los de Huawei, que utilizan la Huawei AppGallery en lugar de Google Play Store, y ROM personalizadas como LineageOS, favorecidas por los usuarios que buscan una mayor personalización y privacidad.

Los dispositivos con GMS ofrecen una experiencia de usuario fluida gracias a su profunda integración con los servicios de Google. Por ejemplo, Google Assistant puede interactuar con varias apps de Google para ofrecer una experiencia cohesionada, como el uso de Google Maps para navegar hasta ubicaciones mencionadas en Gmail. Esta integración garantiza que los servicios funcionen juntos sin problemas en todo el dispositivo.

Los dispositivos sin GMS, sin embargo, pueden ofrecer una experiencia más fragmentada, al depender de múltiples aplicaciones de terceros que pueden no funcionar tan bien juntas, lo que lleva a una experiencia de usuario menos cohesiva. La existencia de dispositivos Android sin GMS se debe principalmente a restricciones regionales, preferencias de los fabricantes y el deseo de independencia del ecosistema de Google.

En regiones como China, la normativa gubernamental restringe el uso de los servicios de Google, lo que empuja a los fabricantes a desarrollar sus ecosistemas de acuerdo con las leyes locales. Además, algunos fabricantes eligen Android sin GMS para diferenciar sus productos y reducir la dependencia de Google, lo que les permite una mayor innovación y control sobre sus ofertas de software.

Aplicaciones y servicios comunes en Android sin GMS

Los dispositivos Android sin GMS dependen de aplicaciones y servicios alternativos para sustituir a las ofertas de Google:

Tiendas de aplicaciones: Huawei AppGallery, Amazon Appstore.

Servicios de correo electrónico: Microsoft Outlook, Yahoo Mail.

Aplicaciones de navegación: HERE WeGo, Sygic.

Estas alternativas a menudo se centran en la privacidad y la personalización, ofreciendo a los usuarios un mayor control sobre sus datos y la experiencia del dispositivo, aunque pueden carecer de la integración perfecta de GMS. Además, en algunos casos son estadísticamente más a menudo víctimas de filtraciones y robos de datos que Google; incluso Outlook fue penetrado recientemente, con robo de datos de clientes.

CPU

EL CPU es un mediador entre el Hardware y el Software, siendo necesario que la arquitectura de los CPU que usan los móviles pueda tener una buena eficiencia para mejorar la experiencia de usuario. Existen diferentes tipos de CPU, en el capítulo de Carga de SO hablamos de las diferencias de los CPU en término generales, desde los usados por los ordenadores hasta los usados por los móviles. Aquí veremos más específicamente los utilizados por los móviles.

El CPU Central Processing Unit debe ser considerado para saber cómo será el desarrollo de la aplicación.

Entendiendo las terminologías

Como es de costumbre, hablaremos de los fundamentos de los CPU para que entendamos las terminologías comúnmente utilizadas en las bibliografías.

El procesador de Intel 4004 operaba con 4-bit word, lo que significa que usaba 4 bits “agrupados” por vez para procesar datos.

El procesador de Intel 8008 tenía 8 Bist. Ofrece 14 BITS de “Address Capacity”. Una evolución importante en este procesador fue el concepto y uso de “INDIRECT ADDRESSING” o indirección. Donde la referencia es un puntero a otra dirección donde el dato realmente está.

Otro proceso de la evolución dando un hito importante fue en el 1974, dando el lanzamiento del progenitor de lo que serían los modernos CPU de escritorio o “Desktop CPU”. Este suceso se dio con el Intel 8080. Posee un bus de datos de 8 bits y 16 bits de bus de dirección.

Así vemos como los primeros procesadores eran de 8 bits, para luego continuar con los de 16 bits.

Otro Hito importante se da el 8 de junio del 1978 Intel lanza el CPU 8086 y en el año 1979 el modelo 8088. Ambos tienen la misma arquitectura interna siendo ambos de 16 bits.

El famoso 8086 fue reconocido por su arquitectura x86, siendo así su apodo más famoso en la industria de la informática. Está diseñado como un CPU de 16 BITS, 8 de esos son para uso general, 4 son dedicados y tienen la habilidad de ser usados como registros de 8 bits por su capacidad. Este CPU implementa el concepto de STACK siendo Push y Pop los modo de usos, siendo conocido como LIFO (Last In, First Out).

El 8086 usaba CISC (Complex Instruction Set Computer) como una característica clave de su arquitectura. Utilizando el concepto de “Orthogonal Instruction Set” pudiendo los datos operar en todas las formas de dirección que admite el CPU. Por ejemplo si una instrucción puede operar en cualquier registro y en cualquier dirección e indirección es considerado como “Orthogonal”.

Por ejemplo siendo igual de validos estos comandos: ADD AX, 5, ADD AX, BX, ADD AX, memory_location, ADD AX, [BX]. Esto reduce la complejidad y aumenta la performance, cuyas características en el RISC no estaban disponibles.

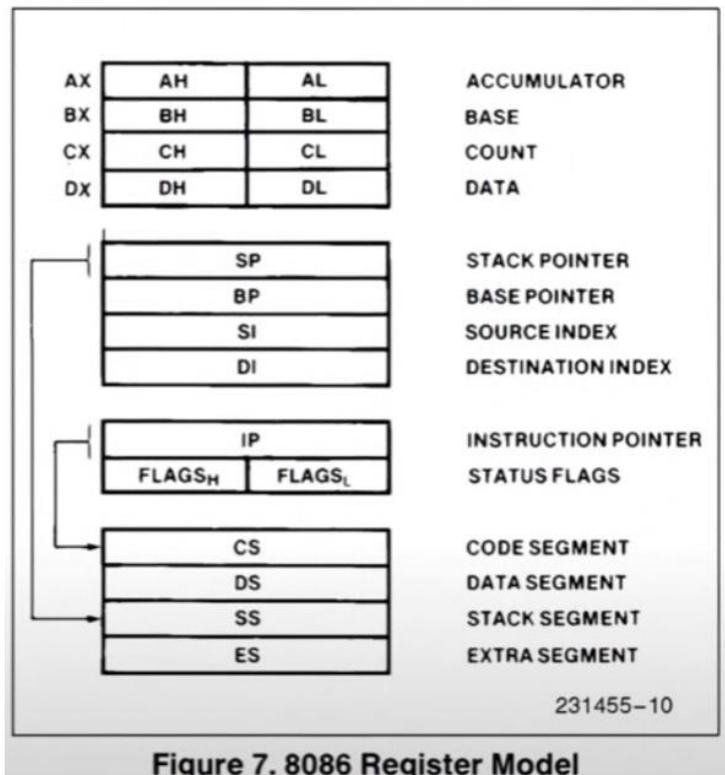


Figure 7. 8086 Register Model

El 8088 posee un bus de datos de 8 bits y el 8086 es de 16 bits. Siendo el 8088 más económico, usado por algunas empresas como IBM para que sus costos de PC para el hogar (las primeras que salieron) sean más económicas.

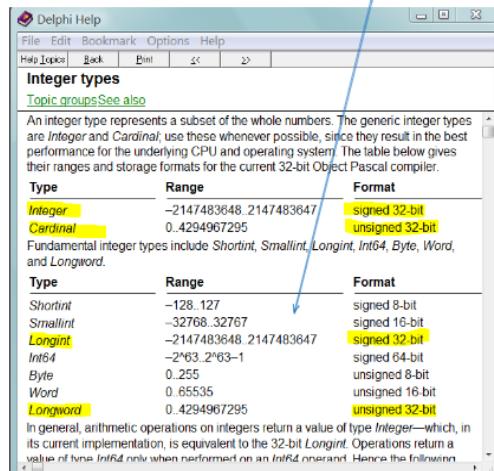
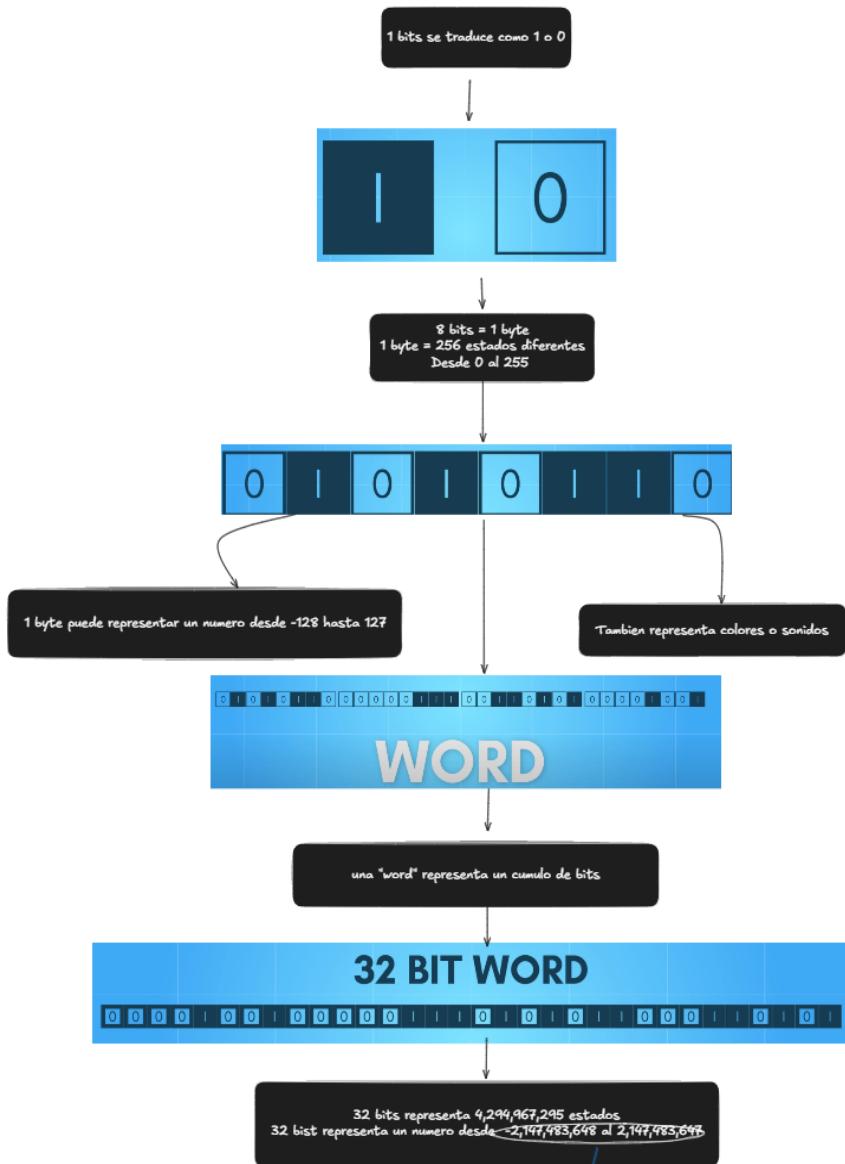
No debemos confundir los 16 bits internos con los 8 o 16 bits usados en el bus. Los bits que pertenecen al bus son para intercambiar datos con otros componentes como por ejemplo la memoria RAM. Por ejemplo, el **Intel 8088** es un procesador de **16 bits internamente**, lo que significa que puede realizar operaciones y manejar datos en bloques de 16 bits dentro de su unidad de procesamiento. Sin embargo, la diferencia es que **su bus de datos externo es de 8 bits**, lo que lo hace más lento a la hora de transferir datos entre el procesador y la memoria u otros dispositivos.

El CPU se maneja con **INSTRUCCIONES**, las cuales se denominan “Instruction Set” y estas tienen la funcionalidad realizar cálculos o mover datos a celdas de memoria. Para que esta pueda ser legible por humanos se utiliza el “Assembly Language”. Por lo que un CPU se podría definir como “Una máquina de procesamiento de instrucciones”.

Hay 3 fases básicas en el procesamiento de datos: Fetch, Decode, Execute.

32 BIT VERSUS 64 BIT	
32 BIT	64 BIT
32 bit is a type of CPU architecture that is capable of transferring 32 bits of data per clock cycle	64 bit is a type of CPU architecture that is capable of transferring 64 bits of data per clock cycle
Requires more time to process and response	Requires a minimum time to process and response
Can address memory up to 4 GB of RAM	Can address memory up to 16 Exabytes of RAM
Cheaper	Expensive
Can be used as a personal computer and to run office routine tasks	Can be used as personal computers and for video edition, audio editing, server applications etc.

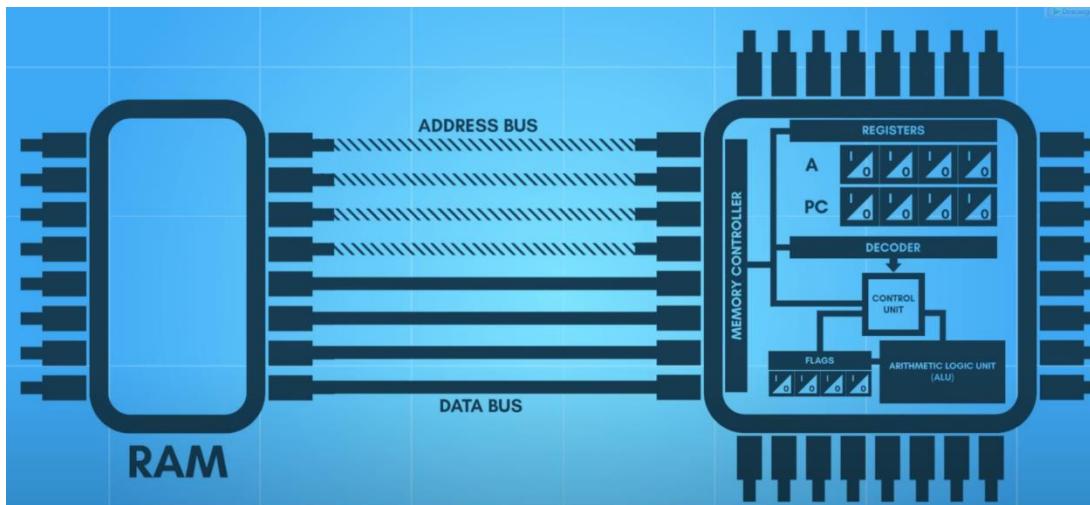
Visit www.PEDIAA.com



Los primeros CPU fueron creados bajo el paradigma del Classic RISC (Classic Reduced Instruction Set Computer) del cual hablamos al detalle en el capítulo de Carga del S.O. Del cual el Intel 4004 utilizaba en su forma de procesamiento.

En el Fetch el CPU carga las instrucciones, las procesa y las devuelve, almacenándose en la RAM. Los datos “corren” a través del BUS, el que es considerado como el “carril” de la “autopista”, donde cada BIT contiene su propio “carril” o línea de datos.

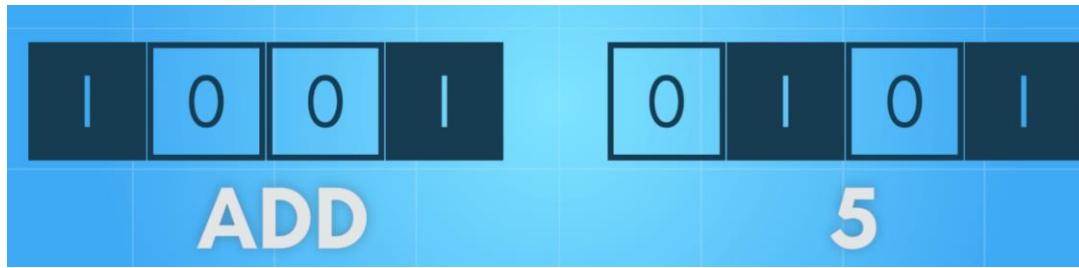
¿Y qué sucede con la ubicación de los datos? Bueno, la ubicación que se solicita también se transmite por BUS. El tamaño del Bit-Word y la dirección del Bit-Word son transmitidos por un Bus. Esto son el Data-bus y Address-bus. Estos Buses pueden verse en la placa mother como líneas impresas. Cuando un CPU realiza una solicitud de acceso a la RAM, se activa el “Memory Controller” en el CPU, donde se carga el “Address-bus” donde está la dirección de memoria a la cual se desea acceder. Luego la Ram utiliza el “Data bus” para que el CPU pueda acceder a estos datos. Para escribir datos el CPU utiliza el “Data-Bus”, la señal de



escritura es recibida por la memoria RAM, localizando en una celda a través del “Address-bus”.

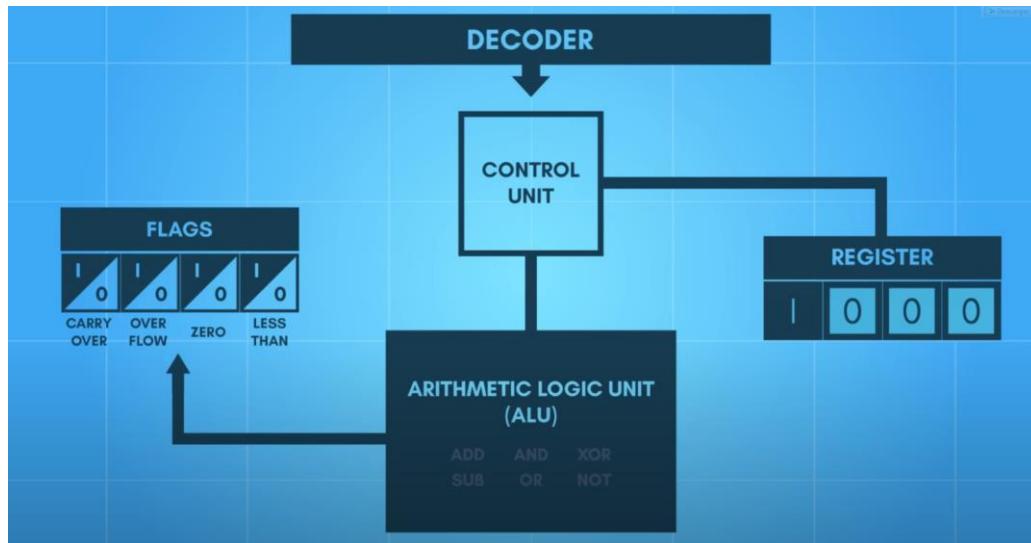
Una vez terminado el FETCH, comienza la etapa de DECODE. En el clásico RISC una WORD es considerada una instrucción completa. Esto cambia en el paradigma que utiliza CISC (hablamos de esa arquitectura en el capítulo de Carga del SO).

Aquí el Word se separa en dos partes conocidas como “BIT FIELDS” llamándose OPCODE y OPERAND CODE. El OPCODE representa una función específica en el CPU. En el OPERAND se consideran bits que son usados como datos.



La función de Suma es dada por el OPCODE pero el valor a sumar es dado por el OPERAND.

En la fase de EXECUTE, una de las secciones más comunes es el ALU (Arithmetic Logic Unit), este recibe dos operandos y realiza las operaciones aritméticas básicas. El resultado se genera gracias a las “FLAGS” utilizando el Carry Over, Over Flow, Zero, Less Than.

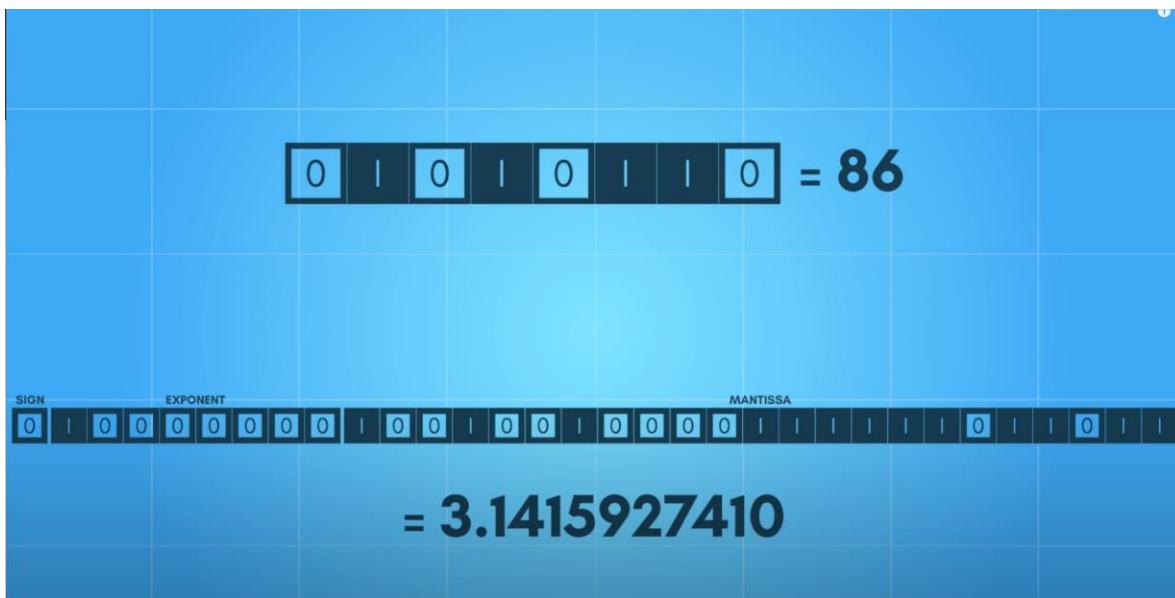


El CLOCK está compuesto por estos tres procesos. La velocidad frecuencia del CLOCK del CPU está medida por números de pulsos por segundo (Hertz). Por ejemplo, el Intel 4004 utilizaba 740 KHz o 740.000 pulsos por segundo. Los procesadores modernos tienen una frecuencia de 5 GHz o 5 billones de pulsos por segundo. Mejorar la frecuencia del CLOCK es un proceso crucial para aumentar el poder de procesamiento del CPU.

Volviendo al concepto de WORD, cuando el CPU Intel 8088 evolucionó al 8087, implementó un “coprocessor” dedicado al punto flotante.



Como vimos anteriormente, los datos binarios representan valores INTEGER, siendo una limitante el uso de valores decimales. Es así como el Floating Point Numbers resolvió ese problema. Lo que hacen es codificar el número decimal entre los bits del word correspondiente.



PROCESADORES UTILIZADOS EN LOS DISPOSITIVOS MÓVILES.

Podemos hablar de 3 principales arquitecturas:

1. **ARM**: ARMv7 o armabi. Es el más común de los tres, está optimizado para el uso de la batería.
2. **ARM64**: AArch64 o arm64. Es una evolución del ARM original, soporta 64-bits de proceso, por lo que es más poderoso y rápido.
3. **x86**: x86 o x86abi. Éste es más potente que el ARM pero es el menos usado de los 3.



ARM es la nomenclatura de los procesadores de los dispositivos móviles, estos tienen un consumo de energía bajo y son eficientes.

También existe el ARMv8 siendo este una evolución del ARMv7.

Una de las principales diferencias entre ARMv8 y ARMv7 es la capacidad de manejar instrucciones de 64 bits. Mientras que ARMv7 solo puede manejar instrucciones de 32 bits, ARMv8 puede procesar instrucciones de 64 bits, lo que significa que puede manejar una mayor cantidad de datos y realizar cálculos más complejos.

El AArch64 o también conocido como ARM64 es la extensión de 64 bits de la familia de la arquitectura ARM.

Arquitecturas de CPU en Android: tipos, diferencias y compatibilidades

Las CPU en dispositivos Android son como el cerebro del dispositivo, encargándose de coordinar aplicaciones, tareas, memoria y gráficos. Con más de 2.500 millones de usuarios en todo el mundo, Android se utiliza en una amplia variedad de dispositivos como smartphones, tabletas, televisores, y más. Dado que diferentes fabricantes como Samsung, Xiaomi, y OPPO usan Android, hay muchas variaciones en las CPUs y sus arquitecturas. Esto implica que Android debe adaptarse para trabajar con distintos procesadores, cámaras, micrófonos y otros componentes, asegurando que todos funcionen bien en conjunto.

Qué es la arquitectura de un procesador o CPU móvil

La arquitectura de un procesador móvil es básicamente su "esqueleto" interno, es decir, cómo están organizados y conectados los componentes que lo forman. La mayoría de los procesadores actuales son del tipo SoC (Sistema en un Chip), donde en un pequeño cuadrado se encuentran integrados la CPU, GPU, memoria RAM, ROM y otros elementos. Esto permite que tu teléfono haga cosas como mostrar videos en 4K, reproducir música de alta calidad o correr juegos intensivos como Fortnite sin problemas.

Los procesadores se componen de partes como cachés, núcleos (cores), controles de memoria y decodificadores de instrucciones, entre otros. En general, se evalúa su rendimiento por la cantidad de núcleos y la potencia, medida en gigahercios. Al final, la arquitectura define cómo funciona el procesador, influyendo en la compatibilidad del sistema operativo y las aplicaciones que puedes usar en tu dispositivo.

Cuáles son las arquitecturas de procesador más comunes en Android

En los dispositivos Android, las arquitecturas de procesador más comunes son ARM y ARM64. La arquitectura ARM es popular por su bajo consumo de batería, pero ha sido reemplazada en gran parte por ARM64, que ofrece más potencia con un consumo moderado. De hecho, a finales de 2022, más del 60% de los dispositivos Android usaban ARM64, mientras que un 19% aún usaban ARM (versión 7). La arquitectura x86, común en PCs, es poco frecuente en Android, representando menos del 0,15%.

Diferencias entre armeabi-v7, arm64 y x86

Las diferencias entre armeabi-v7, arm64 y x86 se relacionan con la evolución y versiones de las arquitecturas de procesadores en Android.

- **x86:** Esta arquitectura es similar a la que se usa en PCs. Puede ser de 32 bits (x86) o 64 bits (x86-64), pero es poco común en dispositivos Android.
- **armeabi-v7:** Esta es una versión de la arquitectura ARM de 32 bits, específicamente compatible con ARMv7-A. Es más eficiente en el uso de energía que x86 y ha sido muy utilizada en dispositivos móviles.
- **arm64 (ARM64-v8a):** Es la evolución de ARM a 64 bits, compatible con la versión v8a. Ofrece más potencia y mejor rendimiento que las versiones anteriores, siendo la arquitectura más común hoy en día en dispositivos Android modernos.

En resumen, armeabi-v7 se refiere a una arquitectura ARM de 32 bits, mientras que arm64 es la versión de 64 bits más avanzada y x86 es la menos utilizada en Android.

SISTEMA OPERATIVO

Carga SO pc

A saber, Android se basa en el Kernel de Linux por lo que entender el arranque nos dará un entendimiento de cómo es el arranque de Linux.

Cuando consultamos la bibliografía al respecto, nos indican que Android generalmente se ejecuta en procesadores ARM. Pero, ¿qué significa “procesador ARM”?

Si bien en este capítulo no profundizaremos en el concepto del PROCESADOR, si hablaremos al respecto para entender los conceptos desde los fundamentos, las bases sólidas que aplican a la construcción del conocimiento acumulativo, de esta manera el lector podrá comprender como funciona Android a más bajo nivel.

Leyendo la bibliografía sobre como Android funciona, específicamente hablando del S.O, nos encontramos con terminología y conceptos específicos que pueden traer a confusión. Es por eso que vamos a explicar los fundamentos necesarios para así, poder comprender más adelante, cómo es que funciona el sistema Operativo.

Hay 3 conceptos que necesitamos comprender: ARM, ISA, RISC.

- **ARM** significa **Advanced RISC Machine**. Es un procesador que usa la arquitectura **RISC**.
- **RISC** significa **Reduced Instruction Set Computer** (Computadora con conjunto reducido de instrucciones).
- Se llama **ISA** (Instruction set Architecture) a una **arquitectura de conjuntos de instrucciones**. Una **ISA** es la parte del procesador que contiene todas las instrucciones básicas que este puede ejecutar.

En este punto, es importante analizar y conceptualizar las diferencias entre “**Arquitectura del Procesador**” y “**Arquitectura de Instrucciones (procesamiento)**”, no significan lo mismo, aunque a veces se lo puede utilizar indistintamente.

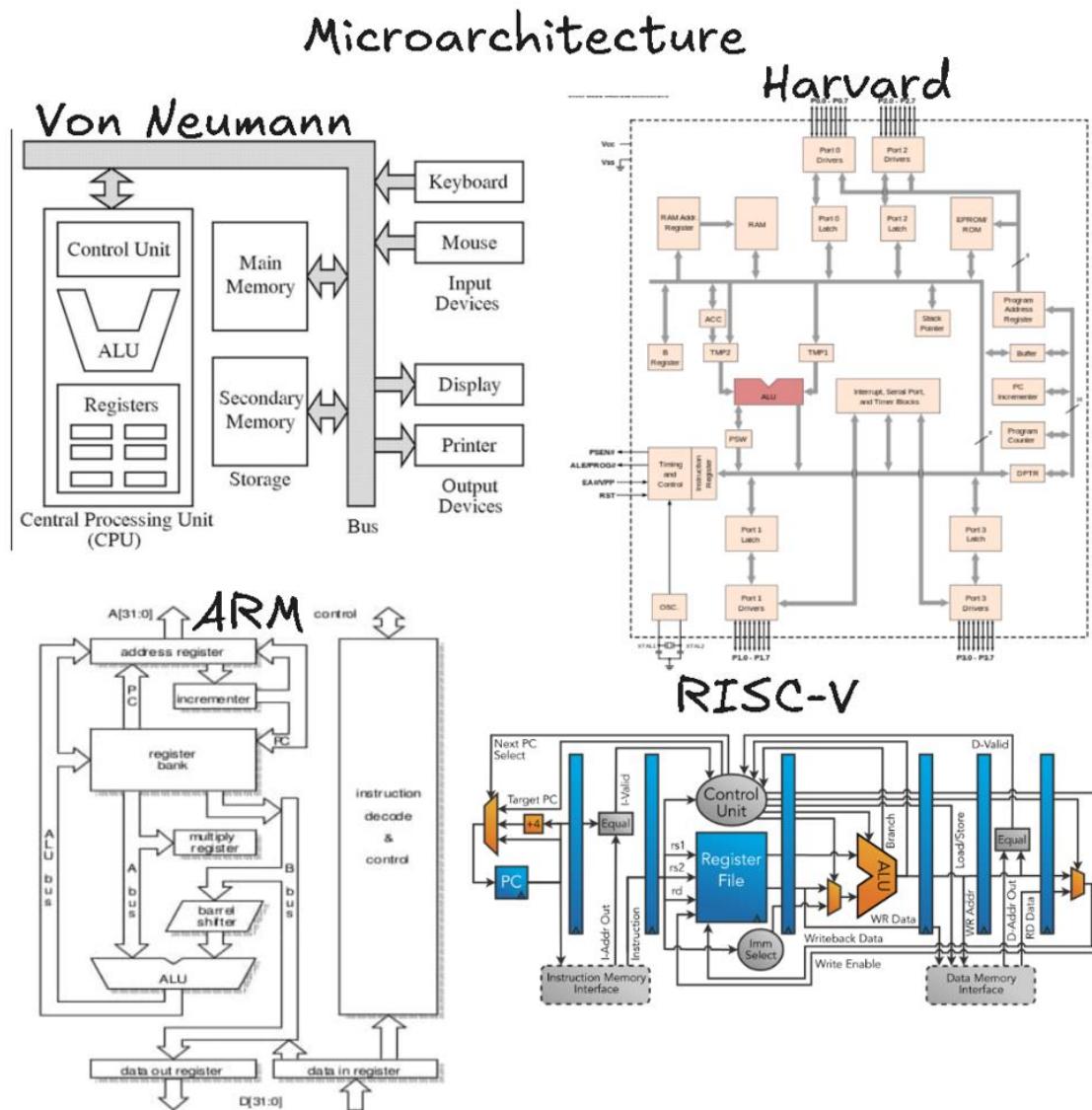
Cuando hablamos de “**Microarchitecture**” hacemos referencia a la arquitectura del procesador, esto es a nivel del Hardware del mismo.

Dentro de la Microestructura hay otro desarrollo vinculado que es el de “**Instruction Format**”, este último depende de la ISA. La ISA no especifica cómo será implementada, por el contrario, es el microprocesador el que decide cómo implementar la “**Instruction Set Achitecture**”.

Por ejemplo, dos marcas de procesadores diferentes pueden estar basados en la misma ISA, pero pueden poseer implementaciones diferentes dentro de la microarquitectura del procesador.

Básicamente, la ISA trabaja como “**Interface**” entre el Hardware y el Software

Hay 4 arquitecturas principales en el Hardware. Dos de están enfocadas en PC de escritorio



y las otras dos están enfocadas en móviles o dispositivos Smart.

1. Arquitectura Von Neumann
2. Arquitectura Harvard
3. Arquitectura ARM
4. Arquitectura RISC-V

VON NEUMANN ARCHITECTURE VERSUS HARVARD ARCHITECTURE

It is a theoretical design based on the stored-program computer concept.	It is a modern computer architecture based on the Harvard Mark I relay-based computer model.
It uses same physical memory address for instructions and data.	It uses separate memory addresses for instructions and data.
Processor needs two clock cycles to execute an instruction.	Processor needs one cycle to complete an instruction.
Simpler control unit design and development of one is cheaper and faster.	Control unit for two buses is more complicated which adds to the development cost.
Data transfers and instruction fetches cannot be performed simultaneously.	Data transfers and instruction fetches can be performed at the same time.
Used in personal computers, laptops, and workstations.	Used in microcontrollers and signal processing.

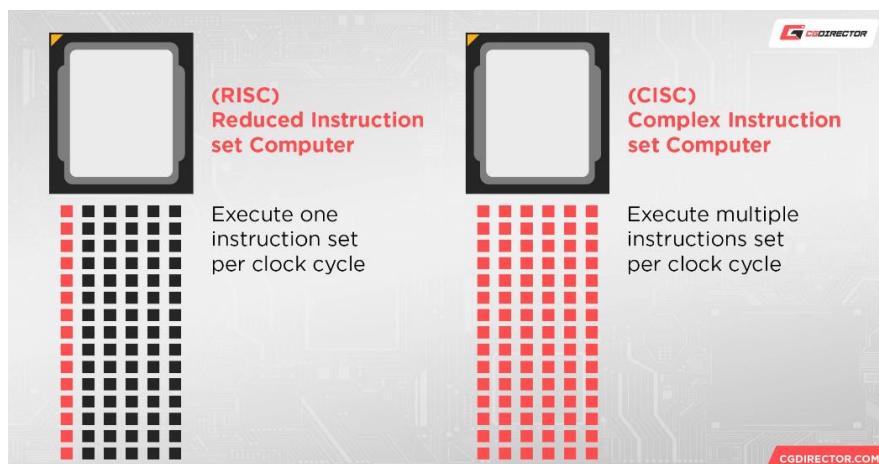
[Difference Between](http://DifferenceBetween.net).net

ARM vs RISC-V

Similitudes	Diferencias
Ambas son arquitecturas RISC	ARM es propietario, mientras que RISC-V es de código abierto
Ambos se utilizan en una amplia variedad de dispositivos, incluidos teléfonos inteligentes, tabletas, computadoras portátiles y servidores.	ARM se utiliza más ampliamente que RISC-V
Ambos se encuentran en constante desarrollo y mejora.	RISC-V es más nuevo que ARM, por lo que tiene el potencial de ser más innovador

Hay 2 tipos de conjuntos de instrucciones(procesamiento) o dos tipos de ISA:

1. **CISC** (Complex Instruction Set Computer): Computadoras con conjunto complejo de instrucciones.
 - a. Instrucciones complejas multi-ciclo.
 - b. Instrucciones complejas, siendo estas más de 1000.
 - c. El código final es pequeño
2. **RISC** (Reduced Instruction Set Computer): Computadoras con un conjunto reducido de instrucciones
 - a. Las instrucciones se pueden ejecutar en un solo ciclo de reloj.
 - b. Instrucciones sencillas, siendo estas menos de 100.
 - c. El tamaño final del código es mayor.



Generalmente la arquitectura de Von Neumann tiene un conjunto complejo de instrucciones (CISC). En cambio, la arquitectura de Harvard generalmente, tiene un conjunto complejo de instrucciones (RISC).

Ambas arquitecturas de instrucciones son usadas hoy en día. Por ejemplo, el X86 (procesadores de Intel y AMD) es el principal procesador que usa CISC, mientras que ARM (procesadores Qualcomm y Media Tek) usa la arquitectura de RISC.

ARM es un ISA de código cerrado basado en RISC.

RISC-V es una ISA de código abierto basado en RISC.



Esto abre una pregunta ¿Hay S.O Android usando en RISC-V? Sí, los hay, pero la gran mayoría usa ARM en los dispositivos SMART en la actualidad. RISC en sí no es una tecnología, sino que es una ideología de diseño. Los procesadores ARM están diseñados para ser muy eficientes, aceptando instrucciones que se pueden ejecutar en un único ciclo de memoria.



RISC es una arquitectura diseñada y optimizada para usar menos instrucciones en comparación con CISC. Esto permite que se use menos energía en los teléfonos inteligentes, cámaras, relojes inteligentes y en todo tipo de dispositivos IoT (dispositivos que usan internet como Smart Belt, Smart Gloves, Smart Shoes, fitness Tracker etc).

Por ejemplo, los procesadores de una PC de escritorio son de arquitectura de 64 BITS, pero las unidades RISC son de 32 BITS. Aunque también los hay de instrucciones de 64 BITS en algunos diseños.



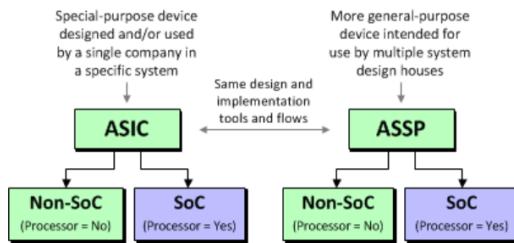
Cuando nos adentramos en la forma en la cual el S.O inicia, lo primero que podemos leer es como funciona su estructura física que es la que da el comienzo al sistema operativo. Este comienza con el encendido y es esa la razón por la cual podemos leer o escuchar sobre términos como “ASICs”, “ASSPs”, “SoCs” y “FPGAs”, vamos a explicar las diferencias para entender estructuralmente donde es que empieza el BootRom (el inicio del S.O).

ASICs: Application-Specific Integrated Circuit. Cualquier chip fabricado a medida es un ASIC

ASSPs: Application-Specific Standard Parts. Su propósito es más general y está destinado a ser usado por múltiples empresas. Por ejemplo, un standalone USB interface chip es considerado como ASSP.

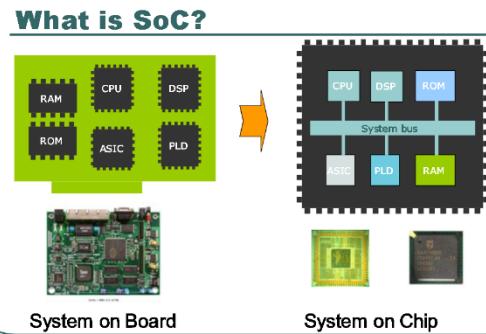
SoC: System-on-Chip. Es un chip de silicio que contiene uno o más núcleos de procesador. Si un ASIC contiene uno o más núcleo de procesador es un SoC. De la misma manera un ASSP que contiene uno o más núcleos de procesador es un SoC.

FPGAs: Field-programmable Gate Arrays. Aquí la característica diferenciadora con los anteriores nombrados es que se puede configurar programando la estructura para ejecutar cualquier combinación de función digital que deseemos. Es decir, ASIC, ASSP y SOC tienen algoritmos, pero “congelados en silicio”.



En los viejos tiempos un mother tenía muchos CHIPS entre ellos uno para el ROM. Hoy los chips incorporan muchas de las funciones.

Como dijimos anteriormente SoC “System-on-a-Chip” está muy desarrollado por ARM. Soc significa “Sistema en un chip”. Al evolucionar los móviles, los fabricantes tuvieron la necesidad de optimizar el espacio. El SoC contiene dentro el CPU, GPU, RAM, Puente Sur, Puente Norte. A diferencia de un procesador único conocido como CPU (Central Processing Unit) usados en las computadoras, un SoC es un sistema completo el cual incluye Hardware y Software en una sola unidad. Es decir, un SoC de los teléfonos inteligentes contiene un CPU, pero también muchos otros componentes. Al combinar en único chip integrado varios componentes es más eficiente energéticamente y ocupa menos espacio.

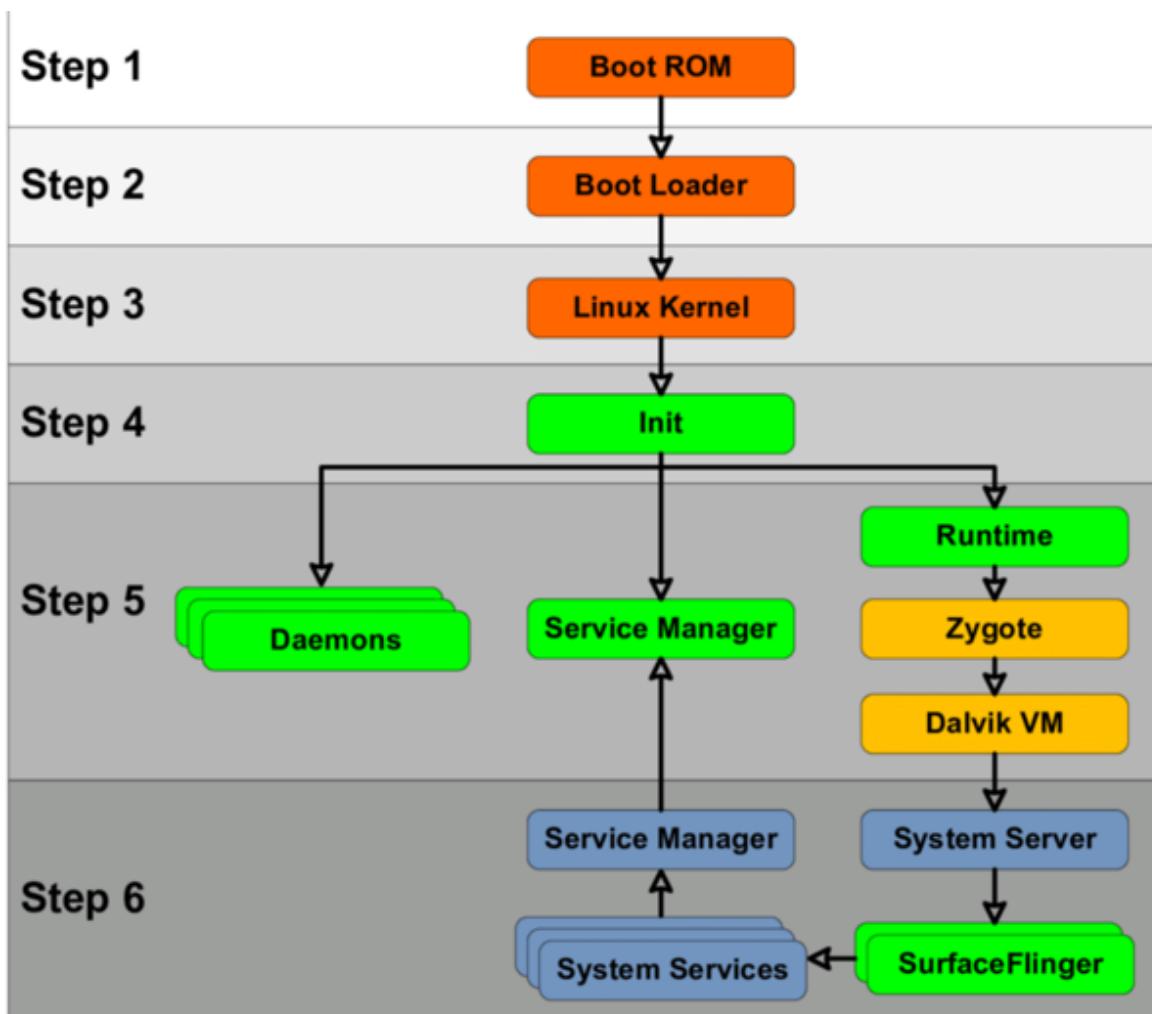


Dicho esto, hablaremos ahora del proceso de arranque en específico. Los procesos de arranque (Booting Process) es la principal preocupación de los sistemas integrados (Embedded operating System) como Linux o Android, por ejemplo, siendo los tiempos de arranque rápido fundamentales para el usuario final.

Los sistemas como Android, son grandes y compactos con muchas funcionalidades. Se considera que inicia el S.O Android al obtener corriente con el “Power-On” y finaliza cuando se muestra “Android” en el display.

Hay 4 pasos en el proceso de arranque (Boot sequence):

1. **Encender y arrancar la ROM.** Todo sistema comienza con el código de la ROM.
 - a. La ROM carga el Boot Loader en la RAM.
2. **Boot Loader:** Es el primer programa que se ejecuta.
3. **Linux Kernel:** El Kernel de Android comienza de manera similar al Kernel de Linux.
4. **AUI (Android userspace initialization):** Este paso inicia el Init, Zygote, Dalvik VM, System Server, Home App (launcher Application).
5. **Zygote - DalvikVM o RunTime - System Server.**

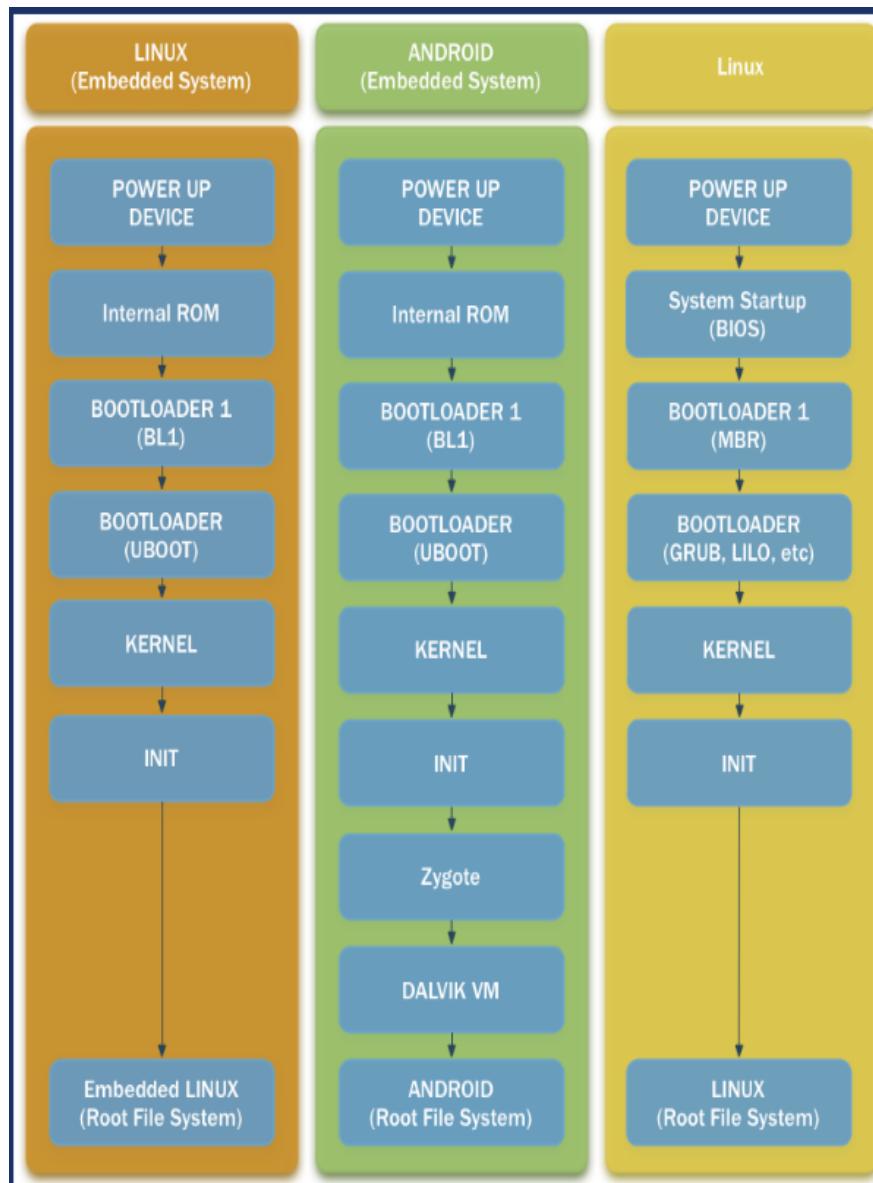


0-Power ON: es el encendido.

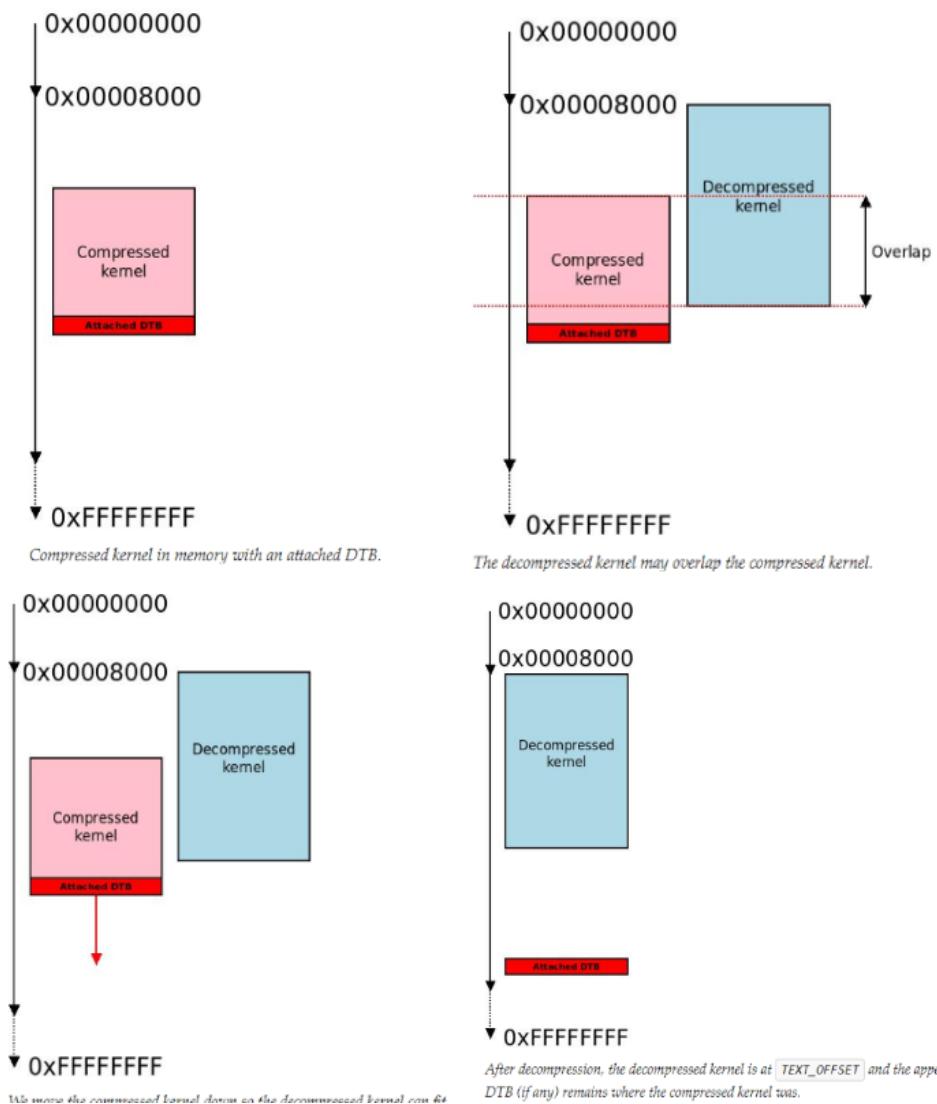
1-BootRom: Es la primera etapa luego del encendido. Aquí se realiza un chequeo del Hardware y se busca al **Bootloader** (paso 2).

Este código reside en un chip de memoria no volátil en la placa madre.

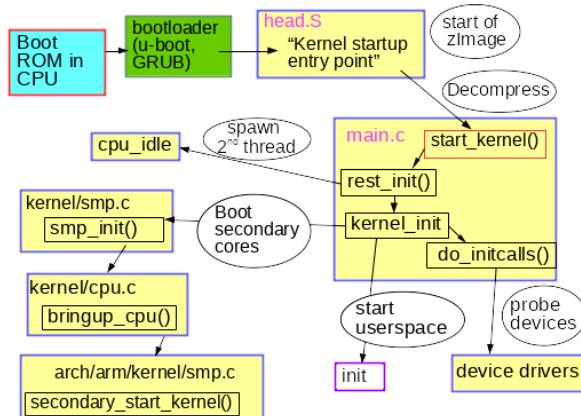
El código ROM tiene un tamaño de 128 KB y se encuentra en la ROM integrada en el chip. La función del ROM es determinar el boot source, inicializar el **HPS** y “saltar” al segundo paso, es decir, el **boot loader**. Este es un pequeño código integrado en CPU ASIC (más atrás ya detallamos este concepto). Una vez que se establece la secuencia del medio de arranque, boot ROM intentará cargar el Boot Loader en la memoria RAM. Si el usuario desea profundizar más específicamente en cómo funciona el ROM podría investigar sobre las direcciones 0xFFFFC0000 o 0xFFFFDFFFF.



2. BootLoader: es el “hardcoded” dentro del del CPU. BootLoader es asociado a cada arquitectura del chip específicamente. Por ejemplo, Qualcomm usa a LittleKernel bootloader, Intel usa UEFI, U-Boot es un Bootloader usado por OMAP chip de Texas Instruments. Según la especificidad de la arquitectura hay numerosas diferencias en su proceso.

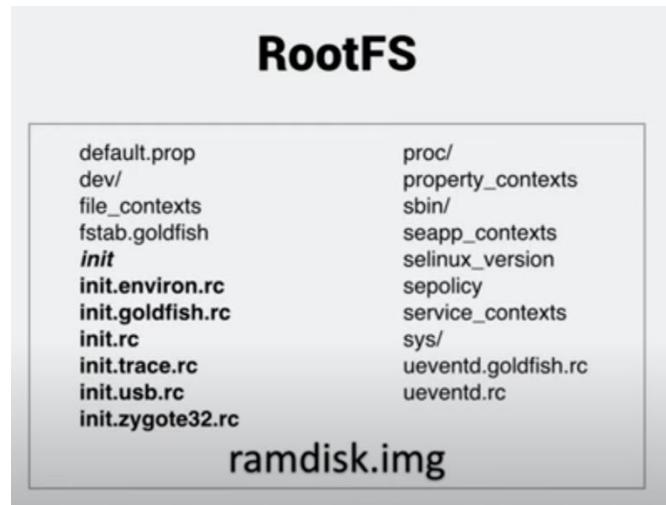


Es decir, el **BootLoader** tiene como tarea la de localizar y cargar el “Kernel Image” descomprimirla (ya que se encuentra almacenada y comprimida en la memoria Flash) y llevarla a la memoria.

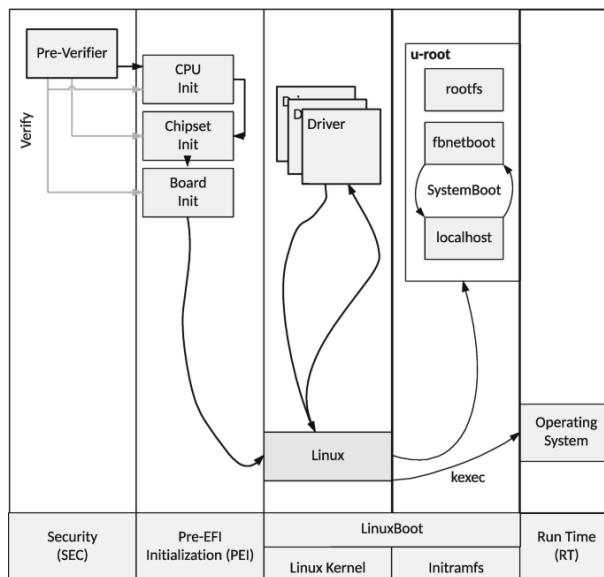
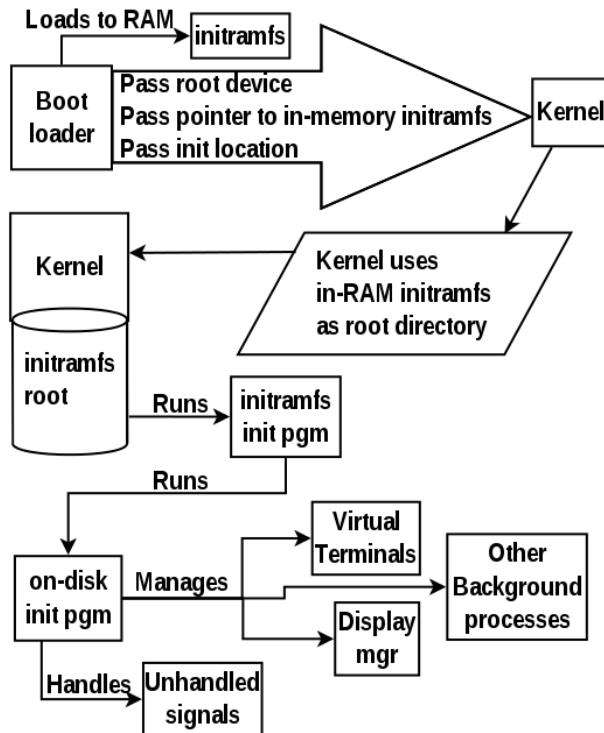


opensource.com

Esta implementación no es como la de un ordenador tradicional que use nativamente Linux, ya que al ser Android hay procesos diferentes al tratarse de un dispositivo móvil. Además, de que también varían de cada dispositivo a otro ya que posee diferentes driver y controladores.



InitRamFs (Initial RAM File System): Otra pieza interesante que tenemos que conocer en este proceso es el llamado initramfs. El initramfs trabaja en conjunto con ramdisk.img, llevándolo a la memoria en un archivo temporal. Una vez hecho esto, el initramfs buscará un binario llamado **init** para ejecutarlo (ver paso numero 4).



El BootLoader puede encontrarse en la siguiente ruta:

<Android Source>\bootable\bootloader\legacy\usbloader

EL LegacyLoader contiene dos archivos importantes, el init.S y el main.c

Init.s: Inicializa las pilas, pone en cero el segmento VSS, llama al _main () en el main.

main.c: inicializa el hardware (clocks, teclado, consola).

Podríamos resumir en 5 pasos la funcionalidad del boot loader:

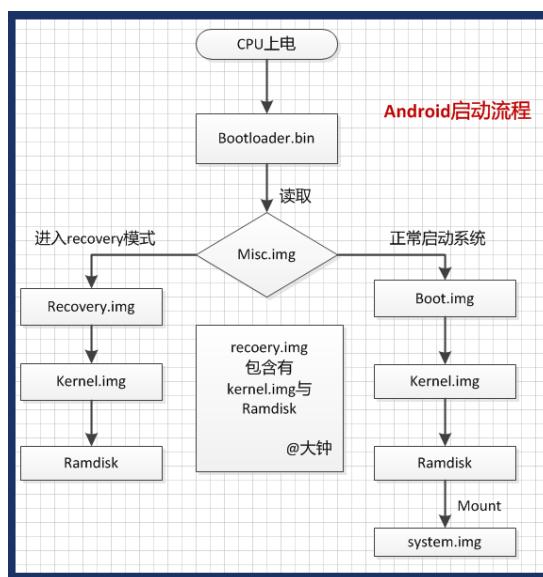
1. EL boot loader detecta y configura la RAM externa.
2. Se carga el boot loader en la RAM Externa.
3. Ejecuta el código para configurar sistemas de archivos, memoria adicional, soporte de red, etc.
4. Cuando el boot Loader haya terminado. buscará al Linux Kernel para cargarlo en la RAM.
5. Se realizará una rutina de descompresión para que una vez terminado el boot loader pueda “saltar” al Kernel de Linux.

3. Kernel:

El kernel “vive” en el dispositivo como un “Binary Blobs (Binary Large Object)”. Este Kernel se encuentra en un lugar específico donde el **BootLoader** sabe cómo encontrarlo. El Kernel provee funciones básicas del sistema, como la administración de procesos, administración de memoria, administración de dispositivos como la cámara, la pantalla, teclado.

El núcleo de Linux se inicia de forma similar en Android y en otros sistemas. Configura todo lo necesario para que el sistema funcione: inicializa los controladores de interrupciones, configura las protecciones de memoria, los cachés y la programación.

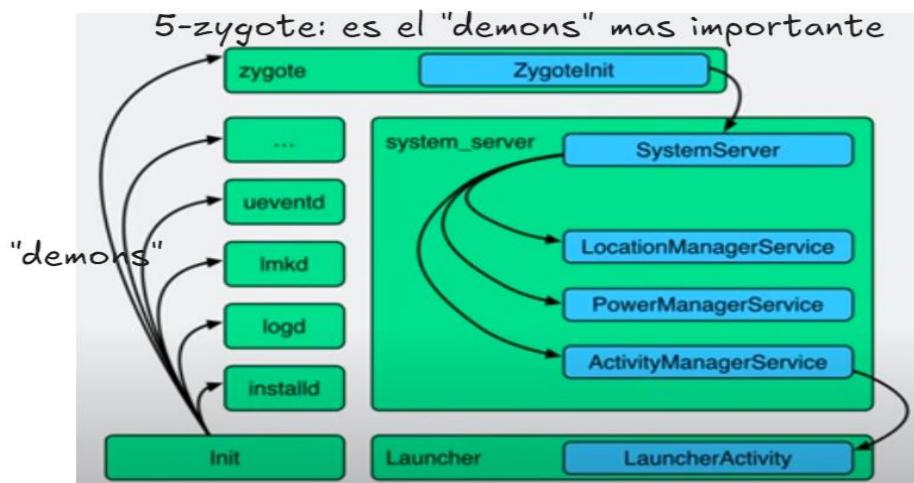
El kernel buscará en el “Root File System” el proceso “init” (que se encuentra en **system/core/init**) y lo lanzará como el proceso inicial del espacio de usuario.



4. init:

Hasta ahora los procesos no eran propios de Android, ya vimos estructuras más duras y estructurales como CPU, chips, FLASH, ROM, inclusive como el Kernel deriva de LINUX, pero en este punto el proceso init es muy particular de android , se podría decir que es el componente específico de android (OASP).

Init trabaja con archivos de sistemas, creando directorios, configura driver para que funcionen los periféricos.



Init ejecuta aplicaciones nativas llamadas “Demons”, estas son: installd, logd, Imkd, ueventd, entre otras. Lo que sucede es que se ejecuta un `init.*.rc` donde hay comandos como `on init`, `on fs`, `on property`, ejecutándose en orden específico respetando la ejecución secuencial. Estos comandos son similares a los Shell de Linux. De esta manera, por ejemplo se crean directorios que serán puntos de montaje para diferentes particiones que luego se cargarán en disco.

Específicamente, hay las llamadas “property” estas son configuradas como Key-Value por ejemplo: `on property:selinux.reload_policy=1`, es así como una vez leído el valor, se ejecutarán comandos específicos de esa property.

```

on property:selinux.reload_policy=1
  restart ueventd
  restart installd
  
```

Veamos un ejemplo práctico, si entro al modo “desarrollador” en un dispositivo y habilito o deshabilito el “*USB debuggin*” lo que hará es cambiar la property de esta manera:

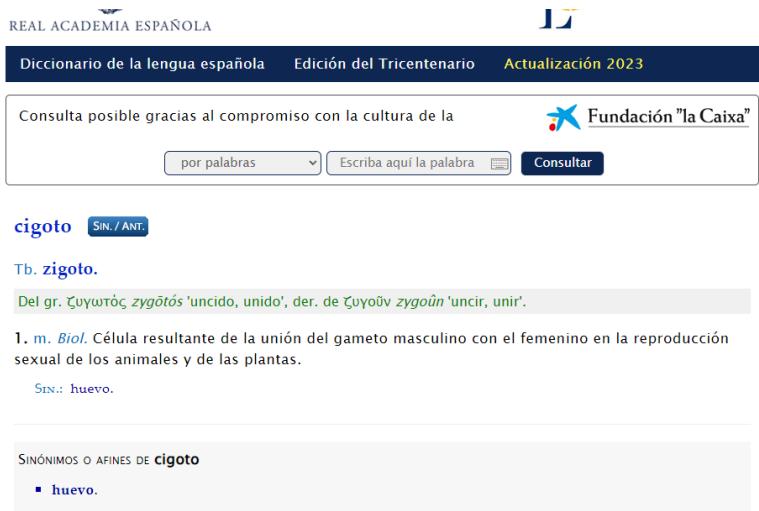
on property:sys.usb.config=adb. De esta manera se “prende” o “apaga” el ADB del sistema.

```
on property:sys.usb.config=adb
    write /sys/class/android_usb/android0/enable 0
    write /sys/class/android_usb/android0/idVendor 18d1
    write /sys/class/android_usb/android0/idProduct 4EE7
    write /sys/class/android_usb/android0/functions ${sys.usb.config}
    write /sys/class/android_usb/android0/enable 1
    start adbd
    setprop sys.usb.state ${sys.usb.config}
```

5. zygote(“demons”):

Hasta ahora vimos cómo el sistema se inicia, se carga el Boot Loader, luego esta carga el Kernel de Linux, y así comienza el proceso init. El init genera otros procesos llamados “deamons”, estos gestionan interfaces de hardware de bajo nivel. Luego el proceso Init comienza otro proceso llamado Zygote.

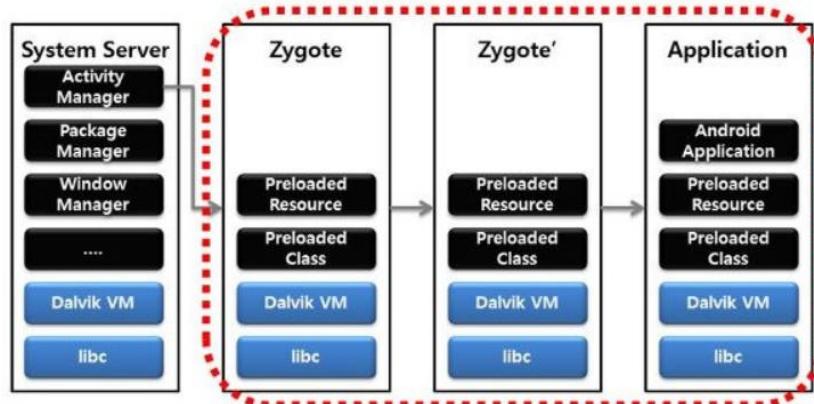
Cigoto significa “huevo” y es utilizado en biología para referirse al primer contacto celular en el ciclo reproductivo.



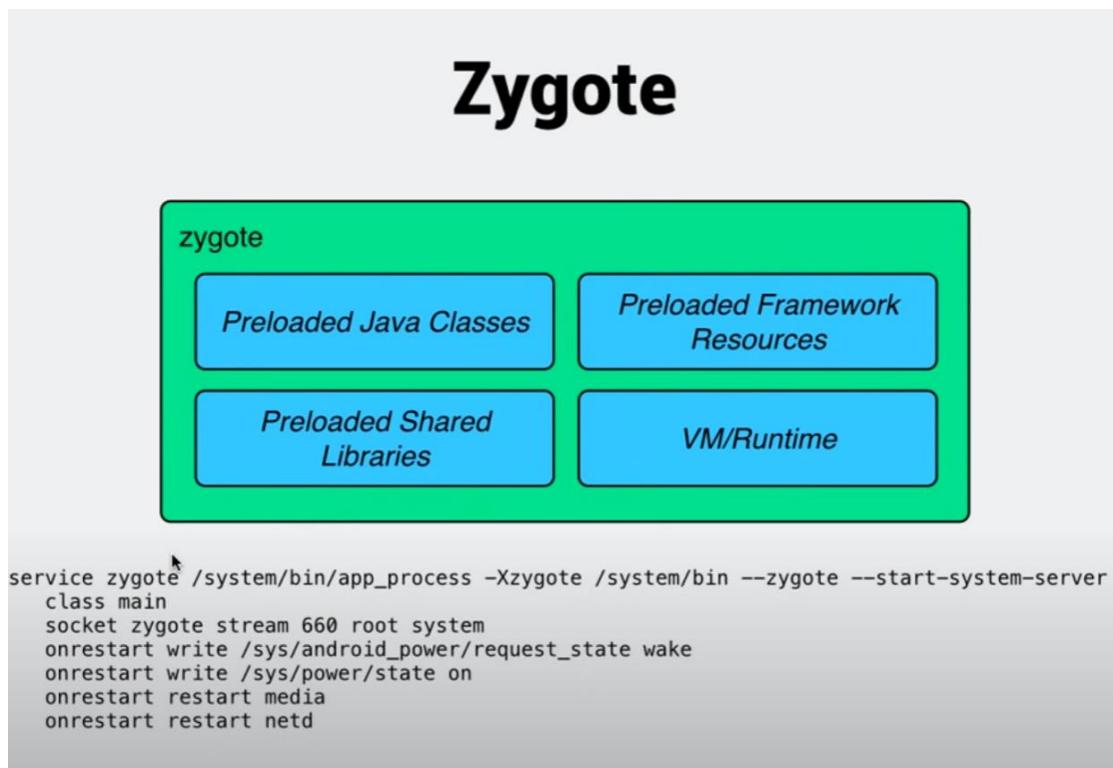
The screenshot shows the homepage of the Real Academia Española's dictionary. At the top, it says "REAL ACADEMIA ESPAÑOLA" and "Diccionario de la lengua española Edición del Tricentenario Actualización 2023". Below that is a search bar with the placeholder "Consulta posible gracias al compromiso con la cultura de la Fundación 'la Caixa'". The search bar includes dropdown menus for "por palabras" and "Escriba aquí la palabra", and a "Consultar" button. The main content area shows the word "cigoto" with its definition: "Tb. **zigoto**. Del gr. ζυγωτός zygōtós 'uncido, unido', der. de ζυγοῦν zygoûn 'uncir, unir'. 1. m. *Biol.* Célula resultante de la unión del gameto masculino con el femenino en la reproducción sexual de los animales y de las plantas. Sin.: [huevo](#)." Below this, there is a section titled "SINÓNIMOS O AFINES DE cigoto" with the entry "■ [huevo](#)".

Lo que hace el demonio zygote es iniciar muchas aplicaciones para que todas puedan “nacer” del mismo “cigoto”. Lo primero que hace el zygote es ejecutar la aplicación **runtime o Dalvik VM (depende la version de android)**. También pre-carga todas las clases comunes usadas en el “Android application framework” y varias apps instaladas en el sistema.

Una función principal es hacer los procesos rápidos, ya que genera un “pre-loading” en las clases de Java que están en el “Preloaded Class”. Cada vez que una aplicación tiene que correr genera un “Fork” aquí mismo. Esto hace que la memoria sea muy eficiente ya que en vez de hacer copias individuales se genera un único proceso.



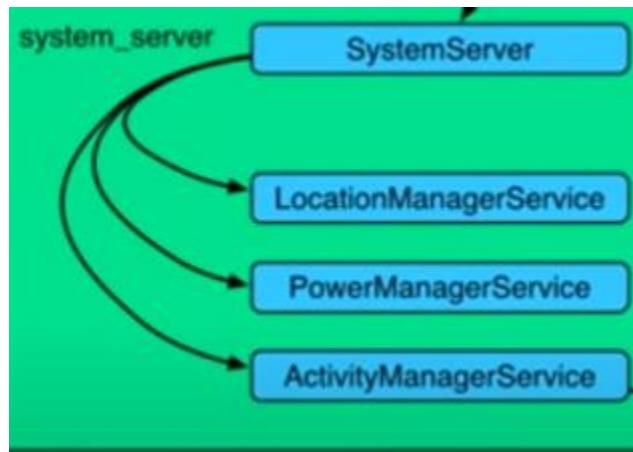
Una vez el “pre-loading” es finalizado, se crea un nuevo proceso llamado **System Server**. Lo que sucede es que se conecta con un socket para poner el Zygote en “Listening mode”, de esta manera le dice cuándo ejecutar una nueva aplicación y hacer una nueva copia.



¿Qué lenguaje es el que se usa para el Zygote y el System Server? Es una buena pregunta, y la respuesta es que se usa JAVA.

6. System Server:

Es la función real que tiene el Java RunTime dentro, Syster Server es un único proceso el cual tiene como función la de controlar la mayoría de servicios como por ejemplo, Location Manager, PowerManager, WiFiManager, SensorManager.



¿Pero qué sucede con otros servicios? Por ejemplo, Bluetooth, NFC (Near Field Communication, es un campo electromagnético con la funcionalidad de intercambios de datos), e incluso la funcionalidad propia del Teléfono no es responsabilidad del System Server. Estos servicios mencionados son aplicaciones individuales, pero se podría decir que la mayoría es responsabilidad del System Server.

Esto es así porque no todos los dispositivos tendrán la posibilidad de tener algunas funcionalidades, como por ejemplo la NFC o el Bluetooth inclusive, aunque aparezcan comunes, no son responsabilidad del manejo de Android ya que no todos los dispositivos móviles lo traen.

Es decir que el código del Server System es una gran cantidad de Clases que luego son agregadas a una colección.

Un Manager importante es el llamado “**ActivityManagerService**” el cual es el responsable del ciclo de vida de todas las aplicaciones dentro del sistemas, por lo que cuando una aplicación pasa a del “foreground” al “background” o su actividad es pausada, esta es controlada por el “**ActivityManagerService**”.

EL ActivityManagerService posee 3 trabajos principales:

1. **Ejecutar el Broadcast (action.PRE:BOOT COMPLETED):** Es usado internamente sin que las aplicaciones puedan acceder. Éste broadcast es utilizado únicamente por el sistema operativo Android para realizar tareas internas antes de que el sistema esté completamente operativo. Es importante saber que este broadcast es exclusivo del sistema y no puede ser interceptado ni utilizado por aplicaciones de terceros. Las aplicaciones comunes no tienen acceso a este proceso.
2. **Ejecutar el Launch Home Activity:** Este posee un “Intent-filter” el cual define si una aplicación es de “Home activity” o “Launcher”. Por ejemplo, cuando compramos

un celular lo primero que ves es el “Setup Wizard” donde te permite configurar la conexión de Wi-Fi, Entrar en tu cuenta google. El “Setup Wizard” es en realidad el “Home App”. Una vez seteado el Launcher este se desactiva, para que la próxima vez que se prenda el dispositivo se ejecute el “Home”. Una vez que la actividad de “Home” es ejecutada la tercera parte.

3. **Broadcast (action.BOOT_COMPLETED)**: Recibe registros de aplicación asumiendo que ya tiene los servicios apropiados. Este es uno de los momentos más importantes del proceso de arranque. Una vez que todos los servicios esenciales del sistema están activos, Android envía este broadcast. El propósito del broadcast **action.BOOT_COMPLETED** le dice a las aplicaciones que el sistema ha completado su arranque y está listo para que las aplicaciones inicien sus propios procesos y servicios.

Default Launcher Application

```
<intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.HOME" />
</intent-filter>
```

ARQUITECTURA DE ANDROID

El sistema operativo de Android se organiza en una arquitectura de cinco capas que incluyen: **Kernel de Linux**, **Capa de Abstracción de Hardware (HAL)**, **Bibliotecas Nativas**, **Framework de Aplicaciones y Aplicaciones**. A continuación, se presenta una imagen que ilustra la arquitectura. Esta representación visual muestra cómo cada capa se integra y contribuye al funcionamiento general del sistema, desde el Kernel de Linux en la base hasta las Aplicaciones en la capa superior.



Kernel de Linux

El Kernel de Linux es la base fundamental sobre la cual está construido Android. Aunque Android utiliza una versión modificada del núcleo de Linux, esta elección se hizo debido a la portabilidad, flexibilidad y seguridad que ofrece. Es el sector del sistema operativo que gobierna al hardware, manejando funciones esenciales como la gestión de memoria, la seguridad y la comunicación entre el hardware y el software, actuando como una capa de abstracción que permite a las aplicaciones interactuar con el hardware del dispositivo.

Android incluye ciertas modificaciones adicionales al kernel de Linux para adaptarlo mejor a los dispositivos móviles. Algunas de estas mejoras incluyen:

- **Low Memory Killer:** Un sistema de gestión de memoria que es más agresivo, diseñado para preservar la memoria en dispositivos móviles.
- **Wake locks:** Una función del servicio PowerManager para gestionar el estado de energía.
- **Binder IPC:** Un controlador que permite la comunicación entre procesos de Android.

A pesar de estas modificaciones, los desarrolladores pueden usar cualquier versión del kernel que soporte las funciones necesarias, como los controladores de dispositivo, sin que estas adiciones especiales afecten el desarrollo del controlador en sí. Además, dado que el kernel de Linux está bajo la licencia GPL, Android también debe seguir esta licencia.

La Capa de Abstracción de Hardware (HAL)

La HAL actúa como intermediario entre el sistema Android y los controladores que gestionan directamente el hardware, sirviendo como un "puente" entre el hardware y el software. Esto permite que Android interactúe con la capa de **kernel** para utilizar los drivers del hardware a través de módulos de bibliotecas, cada uno implementando una interfaz específica para un tipo particular de componente. Por ejemplo, en el caso de una cámara (hardware), el driver se encuentra en el kernel, y para comunicarse con este, se usan módulos de bibliotecas que permiten acciones como encender, apagar o tomar fotos.

Esta capa permite que Android sea independiente de la implementación específica del hardware, definiendo una interfaz estándar que los proveedores de hardware deben implementar. Esto posibilita la compatibilidad con una amplia gama de dispositivos sin necesidad de modificar el sistema operativo. Las implementaciones de HAL se empaquetan en módulos que son cargados por Android en el momento adecuado, asegurando una integración fluida entre el hardware y el software.

El **Lenguaje de Definición de Interfaz HAL** ha evolucionado con el tiempo. En Android 8.0, como parte del proyecto **Treble**, se introdujo **HIDL** (HAL Interface Definition Language), que especificaba la interfaz entre la HAL y sus usuarios, permitiendo a los fabricantes actualizar Android sin necesidad de reconstruir las HAL. A partir de Android 10, **HIDL** fue gradualmente reemplazado por **AIDL** (Android Interface Definition Language), facilitando aún más las actualizaciones del sistema, aunque algunos subsistemas todavía utilizan HIDL.

Para garantizar que las HAL sean consistentes y predecibles, cada interfaz tiene propiedades definidas. Estas interfaces permiten que el sistema Android cargue las versiones adecuadas de los módulos HAL, que constan de dos componentes: **módulos** y **dispositivos**. Las implementaciones de HAL suelen integrarse en bibliotecas compartidas (.so), lo que da flexibilidad a los fabricantes para ajustar sus controladores sin sacrificar la funcionalidad del sistema.

En conjunto, la HAL, junto con las **bibliotecas nativas de Android**, permite que el sistema operativo funcione en una amplia gama de dispositivos, independientemente de su arquitectura física. Las bibliotecas, escritas en **C/C++**, gestionan funciones esenciales como la reproducción multimedia, el acceso a redes y el manejo de bases de datos. Estas funciones son fundamentales para las aplicaciones y servicios de Android, y se exponen a los desarrolladores a través del marco de las aplicaciones.

ART (Android Runtime)

Es la capa del sistema operativo Android que permite la ejecución de aplicaciones. Una aplicación escrita en **Java** se traduce a **bytecode**, que luego es interpretado por una máquina virtual. Inicialmente, antes de **Android 5.0 (API 21)**, se utilizaba la máquina virtual **Dalvik**, que compilaba el bytecode en tiempo de ejecución mediante **compilación Just-In-Time (JIT)**.

A partir de Android 5.0, **Dalvik** fue reemplazada por **ART**, que utiliza **compilación Ahead-Of-Time (AOT)**. Esto significa que las aplicaciones se **compilan completamente al momento de la instalación**, optimizando su rendimiento y reduciendo los tiempos de inicio. **ART** permite ejecutar múltiples máquinas virtuales en dispositivos de baja memoria, mejorando también la compatibilidad con la depuración y optimizando el uso de recursos.

Además, ART está vinculado con mejoras en la compatibilidad con nuevas versiones de **Java** y en el acceso a sus bibliotecas.

Bibliotecas Nativas (C/C++)

Varios de los componentes centrales de Andorid se basan en código nativo que requiere de librerías escritas en C y C++, esta capa nos permite precisamente acceder a componentes nativos dependiendo de nuestras necesidades. Estas librerías proveen funcionalidades esenciales para el sistema operativo y para las aplicaciones que dependen de ellas. Entre las más destacadas se encuentran librerías para la reproducción de medios, gráficos 2D y 3D, bases de datos (SQLite), conectividad web (WebKit) y seguridad (SSL), entre otras.

Estas bibliotecas nativas son accesibles a los desarrolladores a través del marco de las aplicaciones de Android, permitiendo la interacción con componentes de bajo nivel cuando sea necesario. Para el desarrollo nativo en C y C++, Android proporciona el **NDK (Native Development Kit)**, que facilita la integración de código nativo en aplicaciones Android.

Muchos de los componentes centrales de Android, como la gestión de gráficos y medios, están basados en código nativo, lo que permite un acceso directo a funciones optimizadas para el rendimiento del sistema.

Framework de Aplicaciones

Es la capa que proporciona a los desarrolladores un conjunto de **API de alto nivel** para interactuar con el sistema operativo Android. Estas APIs, permiten la creación y gestión de componentes esenciales como **actividades, servicios, proveedores de contenido, gestores de recursos** y el manejo del **ciclo de vida** de las aplicaciones, entre otras funcionalidades.

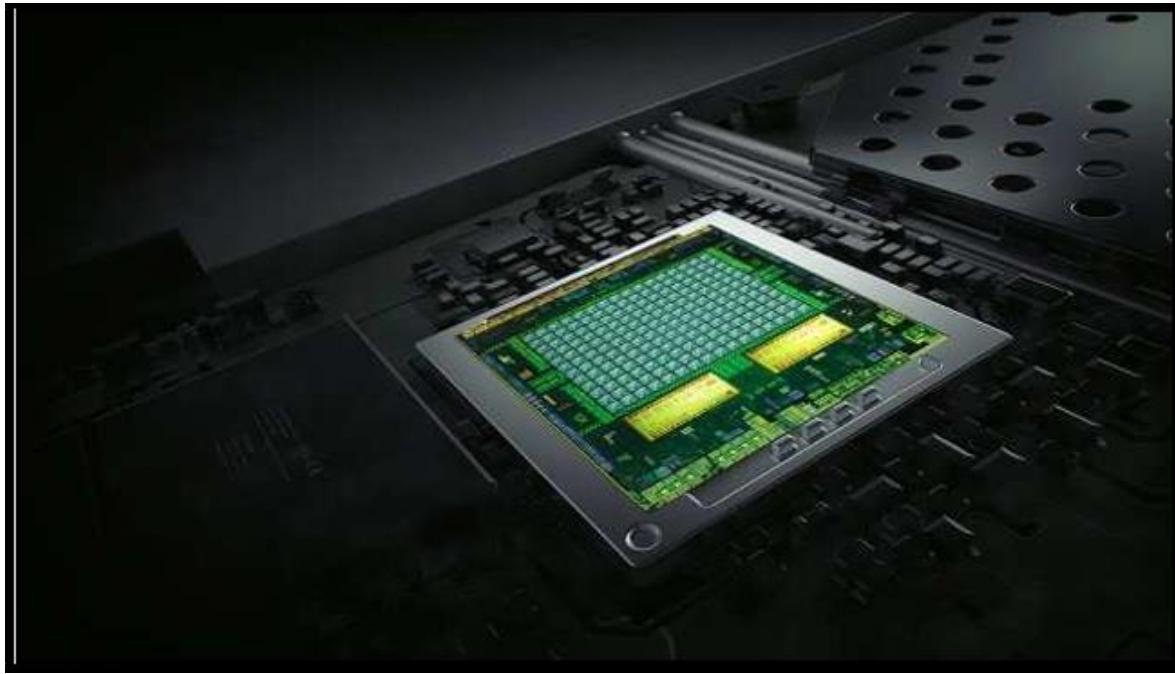
Este framework está disponible principalmente a través de **librerías Java**, agrupadas en paquetes como **android**. En ellos se alojan todas las características necesarias para construir aplicaciones Android. Aunque el desarrollo de aplicaciones puede hacerse en **Java o Kotlin**, ambos lenguajes dependen del **SDK (Software Development Kit)** de Android para acceder a estas APIs y construir las aplicaciones.

El **Framework de Aplicaciones** es independiente del lenguaje de programación, pero brinda soporte integral para manejar componentes gráficos, elementos de localización, y otras funcionalidades cruciales para el desarrollo de aplicaciones móviles en la plataforma Android.

Aplicaciones

Es la capa superior la cual se centra en la ejecución, comunicación y estabilidad de las aplicaciones tanto las preinstaladas por el fabricante como las que el usuario puede **descargar** o desarrollar. Esta capa incluye un conjunto de aplicaciones como correo electrónico, mensajería, calendario, contactos, calculadora. Estas aplicaciones utilizan las APIs del framework para funcionar y ofrecer diversas funcionalidades a los usuarios.

El procesador que mejor se adapta a los requerimientos de dispositivos Android



El diseño de procesadores para smartphones ha avanzado de manera vertiginosa, especialmente en torno a los procesadores ARM y la tendencia al uso de múltiples núcleos y reducción del tamaño de fabricación en nanómetros. Este desarrollo permite una menor generación de calor y un consumo eléctrico optimizado, lo cual facilita la integración de más núcleos en un solo chip, mejorando así el rendimiento general del dispositivo. En términos simples, el microprocesador es el núcleo de cualquier dispositivo electrónico, encargado de ejecutar todas las operaciones a través de un lenguaje de bajo nivel (unos y ceros) y realizando operaciones aritméticas y de lógica binaria. La eficiencia del procesador se mide generalmente en Gigahercios (GHz), que representa la frecuencia de reloj o el número de operaciones que puede realizar en una unidad de tiempo.

Arquitectura ARM



La arquitectura ARM, que es la principal en dispositivos Android, se caracteriza por ser un tipo de arquitectura RISC (Reduced Instruction Set Computer) de 32 bits, desarrollada por la multinacional ARM Holdings plc. Este diseño RISC permite simplificar el conjunto de instrucciones, lo que resulta en un procesamiento más rápido y eficiente en comparación con otras arquitecturas más complejas, como la CISC (Complex Instruction Set Computer) de la familia x86. En la actualidad, aproximadamente el 98% de los procesadores utilizados en smartphones están basados en arquitectura ARM, lo que la convierte en el estándar de la industria móvil. Empresas como Qualcomm, Mediatek, Samsung, y Huawei implementan esta arquitectura, adaptándola en algunos aspectos para mejorar el rendimiento, reducir el consumo de energía o satisfacer objetivos específicos.

En resumen, los procesadores ARM dominan el mercado debido a su diseño eficiente en consumo energético y en tamaño de código. En dispositivos embebidos como los smartphones, la memoria es un recurso limitado debido al tamaño y a las restricciones energéticas. ARM compensa esta limitación mediante un conjunto adicional de instrucciones de 16 bits conocido como “pulgar”, que puede intercalarse con instrucciones ARM de 32 bits. Este conjunto reducido de instrucciones permite disminuir el tamaño del código hasta en un 30%, aunque con una pequeña pérdida de rendimiento. Además, el bajo consumo energético es uno de los pilares del diseño de ARM, que, por ejemplo, en el modelo ARM7100, presenta un consumo de solo 72 mW al operar a 14 MIPS, 33 mW en modo inactivo y 33 µW en modo de espera. En contraste, un procesador x86 como el Intel Atom consume aproximadamente 1W, lo cual lo hace mucho menos eficiente en comparación con

ARM.

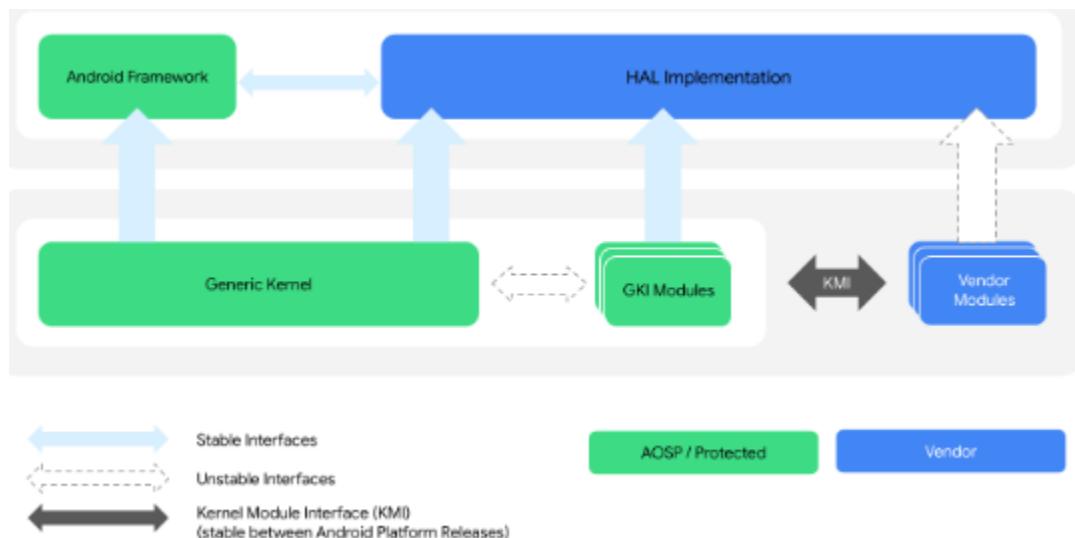
Dentro de los fabricantes de procesadores para dispositivos Android, Qualcomm y Mediatek son los más destacados. Qualcomm, con su serie de procesadores Snapdragon, se posiciona como líder en el mercado de gama alta debido a su alta capacidad de procesamiento y eficiencia energética. Mediatek, por su parte, se especializa en procesadores más económicos, orientados a dispositivos de gama media y baja. Aunque sus chips son aproximadamente un 25% más económicos que los de Qualcomm, suelen compensar su menor rendimiento mediante “overclocking”, es decir, aumentando la frecuencia de reloj para lograr una mayor capacidad de trabajo, aunque esto impacta en la eficiencia energética y genera más calor. Algunos fabricantes de smartphones han optado por desarrollar sus propios chips con arquitectura ARM, como Samsung con su serie Exynos y Huawei con su serie Kirin, diseñados para optimizar el rendimiento de sus dispositivos específicos.

El predominio de la arquitectura ARM en el mercado de dispositivos móviles responde a su capacidad para ofrecer un alto rendimiento con bajo consumo de energía, características que resultan esenciales para los smartphones y su limitada capacidad de batería. En comparación, los procesadores basados en la arquitectura x86, utilizados principalmente en computadoras, consumen más energía y son menos eficientes para la tecnología móvil. Además, los dispositivos móviles, a diferencia de las computadoras de escritorio y portátiles que utilizan componentes separados, utilizan una arquitectura System on a Chip (SoC), que integra en un solo chip el CPU, la GPU y otros componentes, maximizando el rendimiento en un espacio reducido. La arquitectura ARM permite aprovechar al máximo los recursos del sistema operativo Android y es la razón por la cual el 98% de los procesadores en dispositivos Android se basan en esta arquitectura.

KERNEL

El kernel de Android está basado en el kernel LTS (Long-Term Support) de Linux, al que Google añade parches específicos para crear lo que se conoce como Kernels Comunes de Android (ACK). Las versiones más recientes de ACK (a partir de la 5.4) se denominan kernels de GKI (Generic Kernel Image).

Los kernels de GKI permiten separar el núcleo del sistema del hardware, haciendo que el kernel sea genérico e independiente. Esto significa que pueden interactuar con módulos específicos de hardware proporcionados por los fabricantes (como el SoC y otros componentes). Esta interacción es posible gracias a la Interfaz de Módulo de Kernel (KMI), que establece una lista de símbolos y datos globales que los módulos de hardware necesitan para comunicarse con el kernel de GKI. Esto facilita la adaptación y el funcionamiento en diferentes dispositivos Android.



Tipos de kernel

Kernel común de Android (ACK)

Un kernel que está en un nivel inferior a un kernel de LTS y que incluye parches de interés para Android que no se hayan combinado en kernels de Linux con compatibilidad a largo plazo o LTS. Los ACK más nuevos (versión 5.4 y superiores) también se conocen como kernels de GKI, ya que admiten del código de kernel genérico y los módulos de GKI independientes del hardware.

Kernel del Proyecto de código abierto de Android (AOSP)

Kernel de funciones

Es un kernel para el que se garantiza la implementación de las funciones de la versión de la plataforma. Por ejemplo, en Android 12, los dos kernels de funciones eran android12-5.4 y android12-5.10. No se puede aplicar portabilidad a versiones anteriores de las funciones de Android 12 a kernels de 4.19. el conjunto de atributos sería similar a un dispositivo que se lanzó con 4.19 en Android 11 y se actualizó a Android 12.

Kernel genérico principal

Es la parte del kernel de GKI que es común en todos los dispositivos.

Kernel de imagen genérica de kernel (GKI)

Cualquier kernel ACK más reciente (5.4 y posterior) (actualmente solo aarch64) Esta tiene dos partes: el kernel principal de GKI con código común en todos los dispositivos y módulos de GKI desarrolladas por Google que se pueden cargar de forma dinámica en dispositivos cuando corresponda.

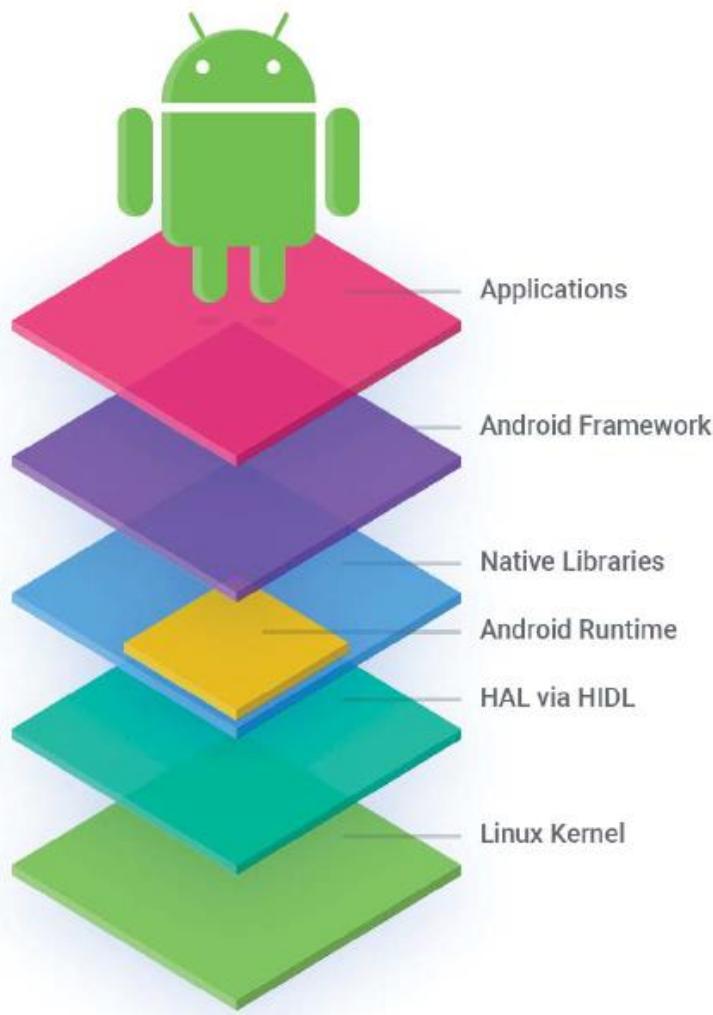
Iniciar kernel

Es un kernel válido para iniciar un dispositivo para una versión determinada de la plataforma de Android. Por ejemplo, en Android 12, los kernels de lanzamiento válidos fueron 4.19, 5.4 y 5.10.

Kernel compatible a largo plazo (LTS)

Un kernel de Linux que se admite durante 2 a 6 años. LTS kernels se lanzan una vez al año y son la base de cada uno de Las confirmaciones de Google.

Capas de Android



Android está construido un núcleo de Linux y una avanzada y optimizada Máquina Virtual para sus aplicaciones Java.

Ambas tecnologías son cruciales para Android. El núcleo de Linux proporciona agilidad y portabilidad para las numerosas opciones de hardware. El entorno Java de Android es clave porque es accesible al gran número de desarrolladores en Java.

Como se puede apreciar en la foto el fondo es el kernel de Linux, sin embargo, no utiliza un kernel estándar de Linux. El Kernel mejorado de Android incluye un controlador de alarma, controlador de memoria compartida (ashmem), conductor aglomerante (entre procesos de la interfaz de comunicación), la administración de energía, eliminador de memoria baja, depurador del núcleo y el registrador. Ademas contiene mucho código desarrollado por Google y por las empresas ya que cada empresa cuando decide desarrollar sus propios dispositivos tiene que adaptar el kernel al hardware que lo necesita. Estas

mejoras al kernel han contribuido a la comunidad de código abierto bajo licencia Pública GNU (GPL).

Otro de los elementos importantes del Kernel son los drivers, estos de ejecutan en modo núcleo y su fiabilidad es significativa para los sistemas operativos. Los drivers utilizan funciones de extensión del kernel para operar y administrar dispositivos. Sin embargo, en algunos casos, estas funciones pueden devolver errores que son mortales para los drivers, provocando fallos en la asignación de memoria y fallos en la configuración.

Para hacernos una mejor idea en esta imagen podemos ver como el kernel contiene todos los drivers del hardware, vemos que HAL hace referencia a partes del hardware, ya que el HAL permite a las aplicaciones de Android comunicarse con los drivers del dispositivo.

Estudios demuestran que el 99 % de las funcionalidades del kernel de Linux fueron reutilizados en Android y sólo el 0,7 % de archivos del Linux del kernel fueron modificados durante esta adaptación. El 95 % de los errores reportados en el kernel de Android están fijados por los desarrolladores de Linux. Los equipos de desarrollo deben considerar posible la adaptación de Linux a Android, ya que hay una buena probabilidad de que la mayor parte del esfuerzo de mantenimiento del sistema adaptado sea levantado por equipos de desarrollo de Linux.

SHELL

¿QUE LE PERMITE HACER UN SHELL A UN USUARIO?

Un shell es una interfaz de línea de comandos que permite a los usuarios interactuar con un sistema operativo. Permite a un usuario ejecutar comandos y ejecutar programas interpretando la entrada del usuario. El shell proporciona una forma para que los usuarios manipulen archivos, realicen tareas como la administración del sistema y automatizan tareas repetitivas mediante la creación de scripts. Además, los shells a menudo brindan opciones de personalización, lo que permite a los usuarios configurar su entorno y adaptarlo a sus necesidades específicas. En general, el shell proporciona un poderoso conjunto de herramientas para administrar e interactuar con un sistema operativo.

VENTAJAS DE UTILIZAR SHELL EN ANDROID

- Sustitución de comodines en nombres de archivos (coincidencia de patrones): Esto es posible en Android como en cualquier sistema Unix. Puedes usar comodines como * o ? en el Shell de Android para trabajar con múltiples archivos que cumplan ciertos criterios. Esto es útil cuando necesitas manejar varios archivos a la vez, como cuando haces operaciones de copia o eliminación.
- Proceso en segundo plano: En el Shell de Android, puedes ejecutar comandos en segundo plano usando el símbolo & al final de un comando. Esto es útil para ejecutar tareas largas (como descargas o actualizaciones) sin bloquear la terminal, permitiendo continuar con otras operaciones.
- Creación de alias de mandatos: En Android, especialmente si usas un emulador de terminal como Termux, puedes crear alias para comandos frecuentes. Esto facilita el acceso rápido a operaciones comunes sin tener que escribir comandos largos o complejos.
- Histórico de mandatos: Como en Linux, el Shell de Android mantiene un historial de los comandos que has ejecutado, lo que permite acceder fácilmente a comandos anteriores usando las flechas del teclado o el comando history. Esto es especialmente útil para repetir o modificar comandos anteriores.
- Sustitución de nombres de archivos: Igual que la coincidencia de patrones, puedes generar listas de archivos que coincidan con ciertos patrones. Esto es útil para trabajar con lotes de archivos sin tener que especificar cada nombre manualmente.
- Redirección de la entrada y la salida: El Shell de Android también permite redirigir la salida de un comando a un archivo o dispositivo usando los operadores > y >> para redirección de salida y < para redirección de entrada. Esto permite guardar resultados o procesar datos sin usar la pantalla del terminal.
- Conductos (Pipes): En Android, también puedes usar tuberías o "pipes" () para conectar la salida de un comando a la entrada de otro. Esto es útil para construir comandos más complejos combinando varios programas pequeños.
- Sustitución de variables de shell: El Shell de Android permite almacenar datos en variables, tanto definidas por el usuario, como predefinidas por el sistema. Estas

variables pueden usarse para simplificar comandos o almacenar resultados de operaciones anteriores.

- En dispositivos Android con acceso al Shell los usuarios pueden crear scripts en Bash para automatizar operaciones repetitivas o complejas. Esto es útil para desarrolladores y usuarios avanzados que desean ejecutar múltiples comandos en secuencia sin intervención manual. Los scripts pueden usarse, por ejemplo, para programar copias de seguridad de archivos o cerrar aplicaciones que consumen muchos recursos después de un período de inactividad.

Scheduler y Dispatcher

A-Scheduler

Estos términos son los más importantes en un sistema operativo, ya que son indispensables en la gestión de procesos y en la ejecución de estos.

La función de **Scheduler** es seleccionar los procesos para ejecutarse, mientras que el **Dispatcher** los carga en el CPU.

Estos procesos trabajan en conjunto para optimizar el uso del CPU y posibilitar realizar múltiples tareas y ejecutar esos procesos.

Los Scheduler son un tipo especial de software dentro del sistema operativo que administra la programación de los procesos de diferentes maneras. La principal función es seleccionar las tareas que son enviadas al sistema y decidir qué proceso será ejecutado.

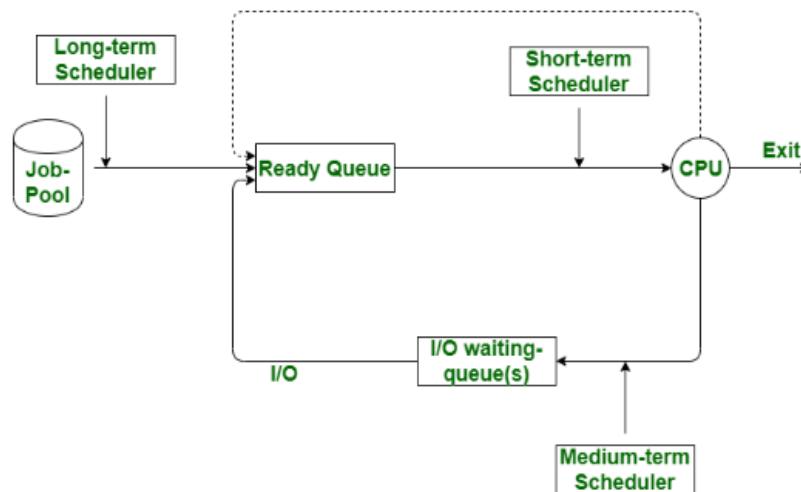
Hay 3 tipos de Scheduler:

1. **Long-Term(job) Scheduler:** La memoria principal tiene un pequeño tamaño, no así la memoria secundaria, es por eso que cuando los programas son cargados en la memoria principal se los conoce como “procesos”. El Long-Term decide cuántos procesos serán los que estén listos en la cola.
2. **Medium-Term Scheduler:** Frecuentemente un proceso en ejecución necesita un **I/O** la cual no requiere tiempo en el CPU.

I/O: es Input u Output, El **Medium-Term Scheduler** gestiona los procesos que están esperando operaciones de **I/O**. Cuando un proceso necesita realizar una operación de entrada/salida (por ejemplo, leer datos de un archivo o esperar la respuesta de la red), este no necesita la **CPU** durante ese tiempo. Entonces, el sistema operativo lo pone en una cola de espera llamada **cola de bloqueados**, para que otros procesos que sí necesitan la CPU puedan ejecutarse.

Una vez que la operación de **I/O** termina, el proceso vuelve a la **cola de listos** (ready queue) para que la **CPU** lo retome. Este intercambio entre procesos que esperan I/O y procesos que necesitan la CPU es gestionado por el **Medium-Term Scheduler** mediante un mecanismo llamado **swapping**.

3. **Short-Term (CPU)Scheduler:** Cuando hay muchos procesos inicializados en la memoria principal, todos están presentes en la cola de bloqueados. De todos los procesos, uno solo es seleccionado para la ejecución, de eso se encarga el Short-Term (CPU Scheduler).



JobScheduler es un servicio del sistema que permite programar tareas o trabajos para que se ejecuten en segundo plano (Background). Está diseñado para ayudar a administrar los recursos del sistema y la duración de la batería agrupando tareas similares y ejecutándolas juntas.

JobScheduler en Android es un servicio del sistema que proporciona una forma de programar varias tareas o trabajos para que se ejecuten en segundo plano. Es particularmente útil para realizar tareas que no requieren la interacción inmediata del usuario y que se pueden posponer a un momento más adecuado.

Estos son los componentes clave de JobScheduler:

JobInfo: esta clase representa un conjunto de criterios que JobScheduler utilizará para decidir cuándo ejecutar un trabajo. Incluye elementos como las condiciones de la red, el estado de carga, entre otros.

JobScheduler: este es el servicio del sistema con el que se puede interactuar con Schedule y administrar los trabajos.

JobService: esta es una clase que necesitamos extender en nuestra aplicación de Android. Se encarga de definir el trabajo que se debe realizar cuando se programa un trabajo.

Kotlin es un lenguaje de programación, siendo el lenguaje oficial de Android para el desarrollo de aplicaciones.

Es por eso veremos un ejemplo en Kotlin aplicado el JobScheduler para que el elector pueda interpretar lo antes dicho.

Paso 1: Crear un JobService

```

1 import android.app.job.JobParameters
2 import android.app.job.JobService
3 import android.os.Build
4 import android.util.Log
5
6 class MyJobService : JobService() {
7
8     override fun onStartJob(params: JobParameters): Boolean {
9         // This is where you put the code to perform the background task
10        // For this example, we'll just log a message
11        Log.d("MyJobService", "Job started")
12
13        // Return true if the job needs to continue in a separate thread, false otherwise
14        return false
15    }
16
17
18    override fun onStopJob(params: JobParameters): Boolean {
19        // This is called if the job is interrupted, you can return true to reschedule the job
20        return false
21    }
22
  
```

'MyJobService' hereda de 'JobService'
 De esa manera se diseña para ejecutar trabajos programados
 Aquí va el código para el método 'onStartJob'
 Muestra en pantalla "Job started"
 como no necesita continuar con otro hilo de trabajo
 retornamos 'false'
 Este método es llamado si el trabajo es interrumpido.
 Retorna 'false' para que el sistema no reprograme el trabajo

Paso 2: configurar el JobScheduler en la actividad o fragmento

```

1 import android.app.job.JobInfo
2 import android.app.job.JobScheduler
3 import android.content.ComponentName
4 import android.content.Context
5 import android.os.Bundle
6 import androidx.appcompat.app.AppCompatActivity
7
8 class MainActivity : AppCompatActivity() {
9
10    override fun onCreate(savedInstanceState: Bundle?) {
11        super.onCreate(savedInstanceState)
12        setContentView(R.layout.activity_main)
13
14        val componentName = ComponentName(this, MyJobService::class.java)
15        val jobInfo = JobInfo.Builder(1, componentName)
16            .setRequiresCharging(true) // You can set any criteria here
17            .setRequiredNetworkType(JobInfo.NETWORK_TYPE_ANY)
18            .build()
19
20        val jobScheduler = getSystemService(Context.JOB_SCHEDULER_SERVICE) as JobScheduler
21        jobScheduler.schedule(jobInfo)
22    }
23
  
```

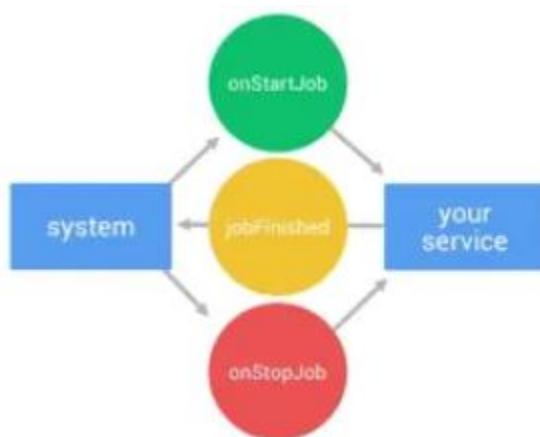
JobInfo contiene información sobre el trabajo que JobScheduler debe ejecutar (tipo de red, carga del dispositivo, etc).
 Es el servicio que programa y ejecuta trabajos en segundo plano.
 identifica un componente específico dentro de la aplicación.
 Proporciona acceso a los recursos y servicios del sistema.
 Esta clase se usa para pasar datos entre actividades o guardar/restaurar el estado de una actividad.
 proporciona compatibilidad para versiones antiguas de android
 Cuando la actividad es creada se llama al método 'onCreate'
 se define el layout que usará la actividad
 se crea un objeto 'componentName' que indica el servicio 'MyJobService' que se usará para ejecutar el trabajo en segundo plano
 El trabajo definido por 'jobInfo' se programa para que el 'JobScheduler' lo ejecute cuando se cumplan las condiciones

Tenemos que tener en cuenta que a partir de Android 8.0 (nivel de API 26), los servicios en segundo plano (incluido 'JobService') tienen algunas limitaciones.

Están sujetos a límites de ejecución en segundo plano más estrictos impuestos por el sistema para mejorar la duración de la batería. Por lo tanto, para tareas en segundo plano más intensivas, es posible que desee considerar el uso de WorkManager u otros enfoques de procesamiento en segundo plano según su caso de uso.

JobScheduler en Android nos permite:

- **Optimización de la batería:** JobScheduler agrupa tareas similares.
- **Restricciones de Red:** Por ejemplo, programar un trabajo que descargue datos cuando el dispositivo esté conectado a Wi-fi.
- **Estado de Carga:** Se puede especificar que un trabajo se ejecute cuando el dispositivo esté enchufado.
- **Estado inactivo:** Programar trabajos para que sean ejecutados cuando el dispositivo este en estado inactivo.
- **Agrupar tareas similares:** Esto hace que el uso de los recursos sea más eficiente.
- **Compatibilidad con el modo Doze:** Se programan tareas cuando el estado de consumo de energía es bajo.
- **Respeto de las preferencias de usuario:** Respeta la preferencia del usuario con las políticas del sistema considerando las tareas en segundo plano, dando una mejor experiencia de usuario.
- **Cómo evitar bloqueas de activación:** Gracias a JobScheduler se puede activar el estado de bajo consumo cuando el dispositivo no se está en uso.
- **Políticas automáticas de reintentos y devoluciones:** JobScheduler permite gestionar políticas automáticas de reintentos, sin que sea un manejo manual de los mismos.
- **Compatibilidad con intervalos flexibles:** Para tareas que no son urgentes, se puede determinar un plazo más flexible.



B-Dispatcher

Una vez que finaliza su trabajo el Scheduler, comienza a trabajar el Dispatcher. Y una vez que Scheduler selecciona el proceso, el Dispatcher lo mueve a la cola a la cual debe ir. El dispatcher es el módulo que entrega el control al CPU para que gestione el proceso que fue seleccionado por **3-Short-Term (CPU) Scheduler**.

Hay procesos que pasan en esta función

1-Switching Context: Se crea un núcleo del proceso actual y se restaura el estado del proceso que se ejecutará.

2-Switching to User Mode: Se asegura que se ejecute en el “modo usuario” (user mode) y no en el “Modo Kernel” (Kernel Mode), esto es por razones de seguridad y no romper privilegios.

3-Jumps to the correct Location: Aquí el programa puede ser reiniciado.

PROCESOS

Process State, PCB (Process Control Block) y RAM

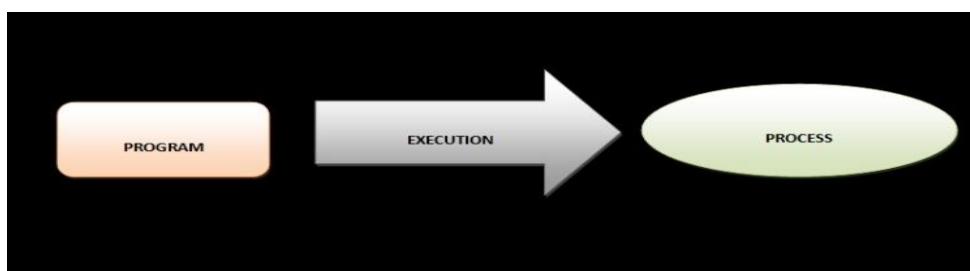
Antes de profundizar en la temática que nos corresponde en este capítulo, vamos a repasar los conceptos desde las bases fundamentales.

Un proceso es un programa en ejecución, usualmente para completar tareas el proceso necesita recursos, como pueden ser CPU, RAM. Un proceso es más que un código, ya que incluye la actividad actual, el contenido de los registros del procesador, entre otros. Si el programa es la “entidad pasiva” el proceso es “la entidad activa”.

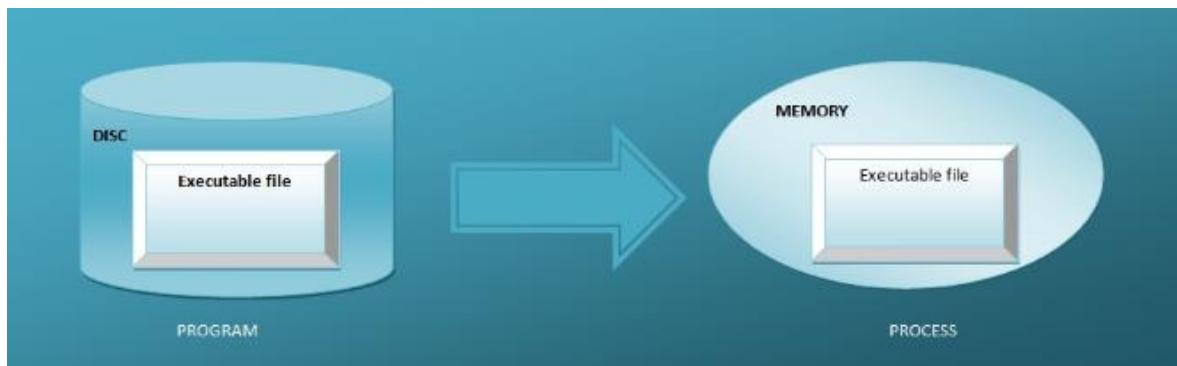
Un proceso es entonces un PROGRAMA EN EJECUCIÓN.

Un proceso incluye:

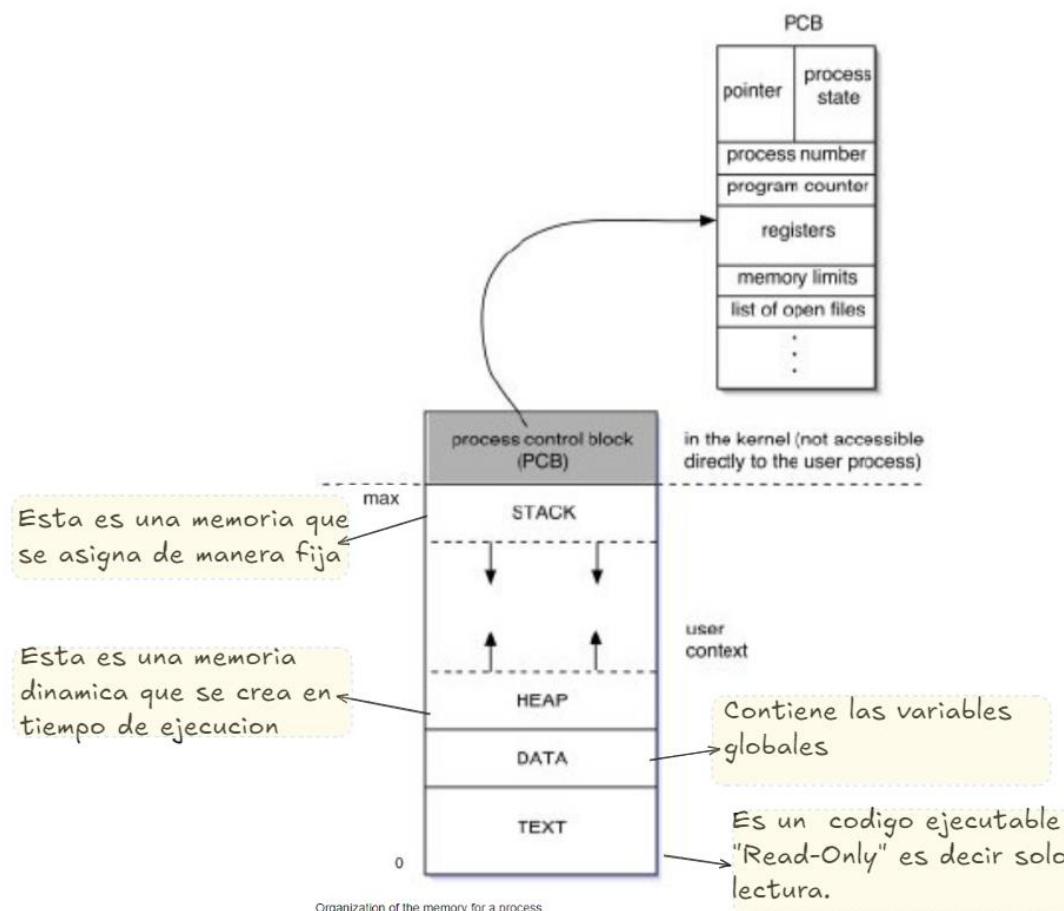
- **Text Section:** La sección de texto consta del conjunto de instrucciones que se ejecutarán para el proceso.
- **Stack data Section:** La sección de datos contiene los valores de las variables globales inicializadas y no inicializadas en el programa.
- **Program Counter:** La sección “Text, “data” y “Stack” comprenden el espacio de direcciones del proceso. El “Program Counter” tiene la dirección de la siguiente instrucción que se ejecutará en el proceso.



Cuando el programa está en memoria ya está activo, de esta manera es como se considera la funcionalidad del concepto “Process” o “Proceso”.

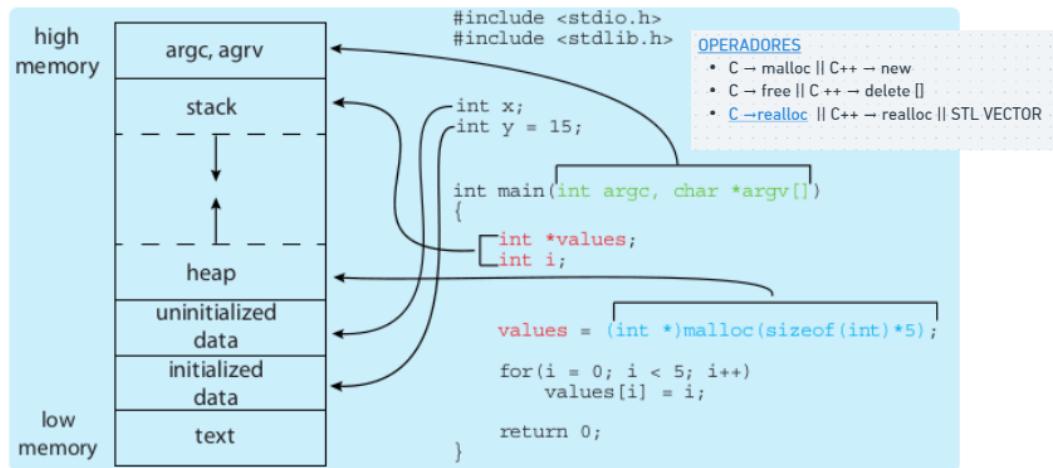


Ahora bien, cuando hablamos de procesos y del Process Control Block (PCB), la memoria RAM se menciona en relación con las secciones de **heap** y **stack** porque el sistema operativo necesita gestionar el uso de memoria de cada proceso en ejecución.



Los tamaños de **Text** y **Data** son fijos, ya que no cambian durante la ejecución del programa, pero la Stack y la Heap pueden reducirse y crecer dinámicamente durante la ejecución del programa.

Memory layout of a C program:



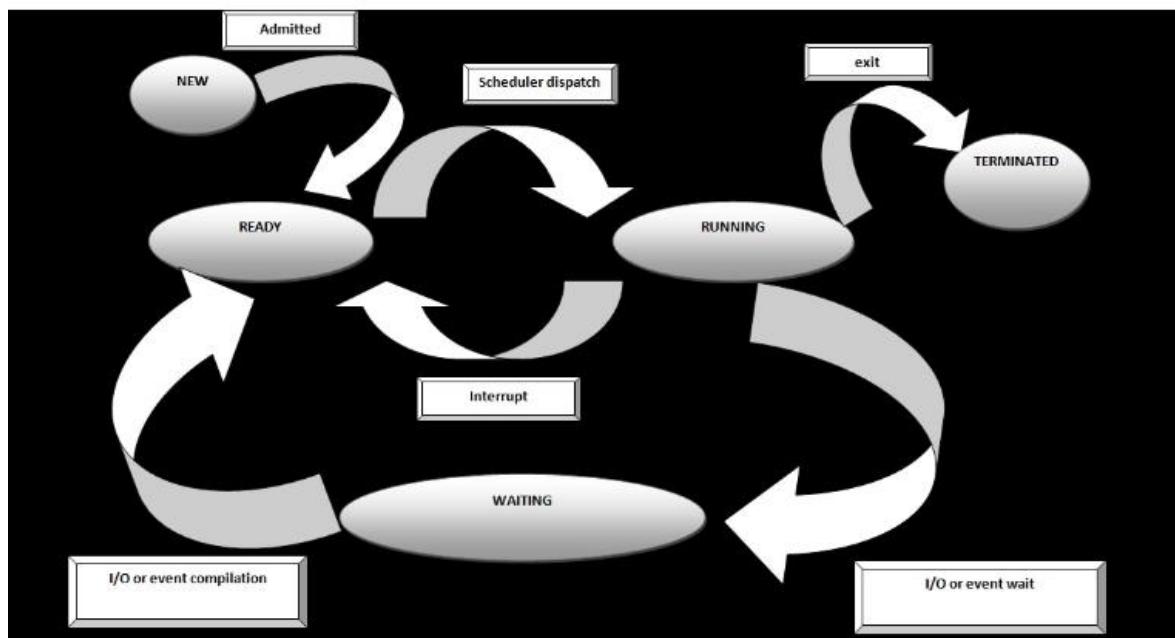
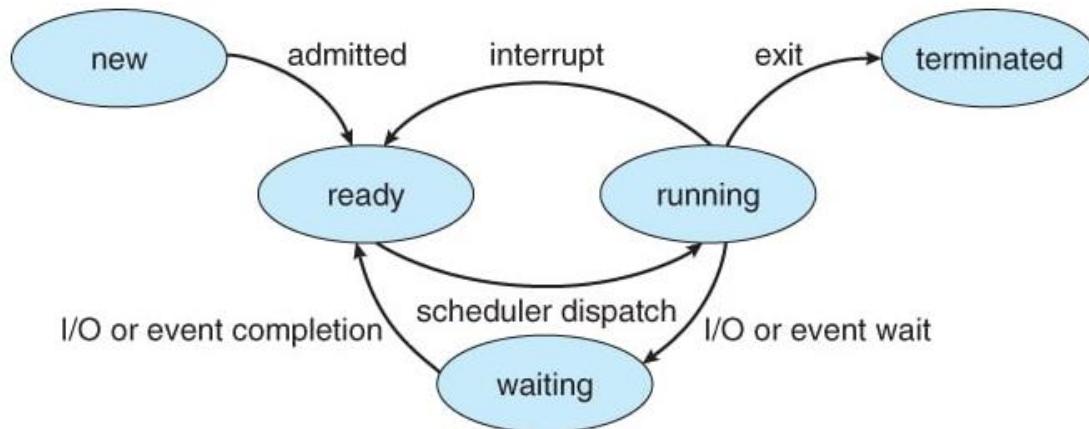
Estados de procesos y PCB (Bloque de control de proceso)

EL Process State o estado de proceso:

El estado de un proceso se define en parte por la actividad actual de ese proceso. Un proceso puede encontrarse en uno de los siguientes estados:

1. New: Se crea el proceso.
2. En ejecución (Running): Se ejecutan las instrucciones.
3. Listo (Ready): El proceso está listo para que se le asigne al procesador.
4. Espera (Waiting): El proceso espera a que ocurra el evento.
5. Terminated: Se completa la ejecución del proceso.

Estos términos son los más importantes en un sistema operativo, ya que son indispensables en la gestión de procesos y en la ejecución de estos



PCB o Process Control block

Para cada Process hay un PCB (Process Control block o Task Control Block). El PCB representa de forma única a cada proceso.

La información del PCB es la siguiente:

1. **Process State:** El proceso puede estar en “New”, “Ready”, “Runnin”, etc.
2. **Program Counter:** Contiene la dirección de la próxima instrucción que será ejecutada.
3. **Registers:** Los registros del CPU incluyen “Acumuladores”, “Registros de Indice”, “Stack Pointers” y “General-purpose registers”. La cantidad de registros y el tipo pueden ser diferentes según las arquitecturas.

4. **Memory Management**: Dependiendo del sistema de memoria que use el S.O, puede incluir información como el valor de los registros límites, “Page Tables, o “Segment Tables”.
5. **CPU-Scheduling System**: Incluye información como la prioridad de procesos, parámetros Scheduling, punteros a las colas de Scheduler.
6. **Accounting Information**: Aquí hay información como “Time Limits”, tiempo real de uso, cantidad de CPU, números de procesos o trabajo.
7. **I/O status information**: Incluye información como la lista de archivos abiertos, “IO devices” entre otros.

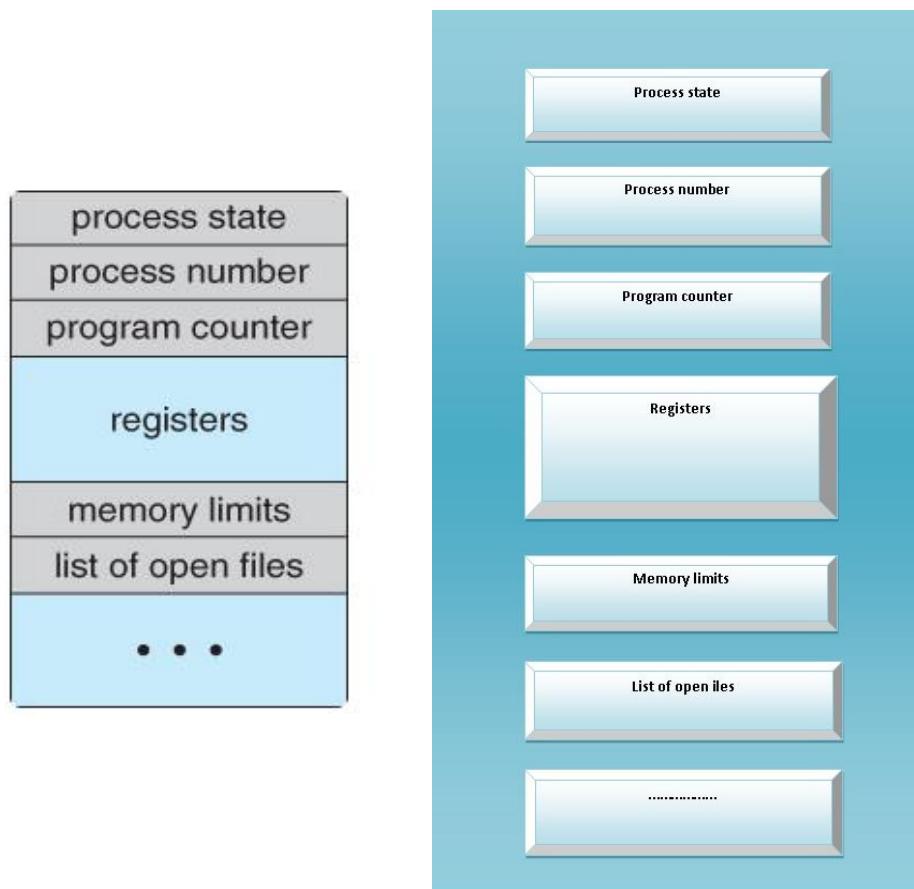


Figure Process Control Block (PCB)

Android puede albergar una variedad de aplicaciones y está diseñado de manera que elimina la redundancia o duplicación de funcionalidades en diferentes aplicaciones.

Existen dos técnicas principales relacionadas con la comunicación entre procesos, a saber:

- **Intents:** Permiten que una aplicación seleccione una actividad en función de la acción que desea invocar y los datos en los que opera. Se necesita un “Hardcoded Path” para usar sus funciones e intercambiar datos.
- **Remote Methods:** Las “Remote Procedure Calls (RPCs) con estas APIs pueden ser accedidas de forma remota. Cuando se utiliza este funcionamiento de forma remota aparece el AIDL o “Android Interface Definition Language” para esta funcionalidad.

Las aplicaciones de Android evitan la comunicación entre procesos. Proporciona funciones en paquetes cargados por las aplicaciones que las requieren. Para que las aplicaciones intercambien datos, deben utilizar el sistema de archivos u otros mecanismos tradicionales de IPC de Unix/Linux.

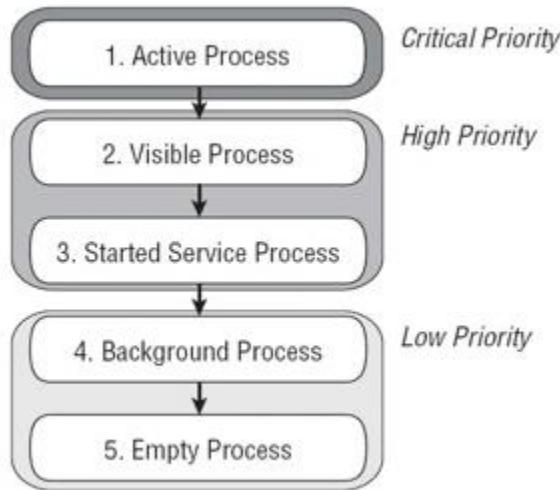
Desde sus orígenes, Android ha admitido la multitarea y no impone restricciones sobre los tipos de aplicaciones que pueden ejecutarse en segundo plano. Si una aplicación requiere procesamiento mientras está en segundo plano, la aplicación debe utilizar un servicio, un componente de aplicación independiente que se ejecutará.

Por ejemplo, una aplicación de audio: si la aplicación pasa a segundo plano, el servicio continúa enviando datos de audio al controlador del dispositivo de audio. De hecho, el servicio seguirá ejecutándose incluso si se suspende la aplicación en segundo plano. Los servicios no tienen una interfaz de usuario y ocupan poco espacio en la memoria, lo que proporciona una técnica eficiente para la multitarea en un entorno móvil.

Gestión de Procesos en Android.

Tipos de procesos en Android:

1. **Activos:** Son los procesos que se están ejecutando y son necesarios para la interacción con el usuario, como los procesos en primer plano y los servicios que los apoyan.
2. **Visibles:** Están un nivel por debajo de los activos y suelen estar en segundo plano.
3. **Servicio:** Procesos que brindan servicios importantes, como mantener la conexión a internet, aunque no sean visibles.
4. **Segundo Plano:** No son visibles, pero siguen en ejecución, y reciben recursos si hay disponibilidad.



5. **Vacíos:** Procesos que se lanzan rápidamente, pero están en espera de que el sistema libere memoria.

Jerarquía de Procesos de Android

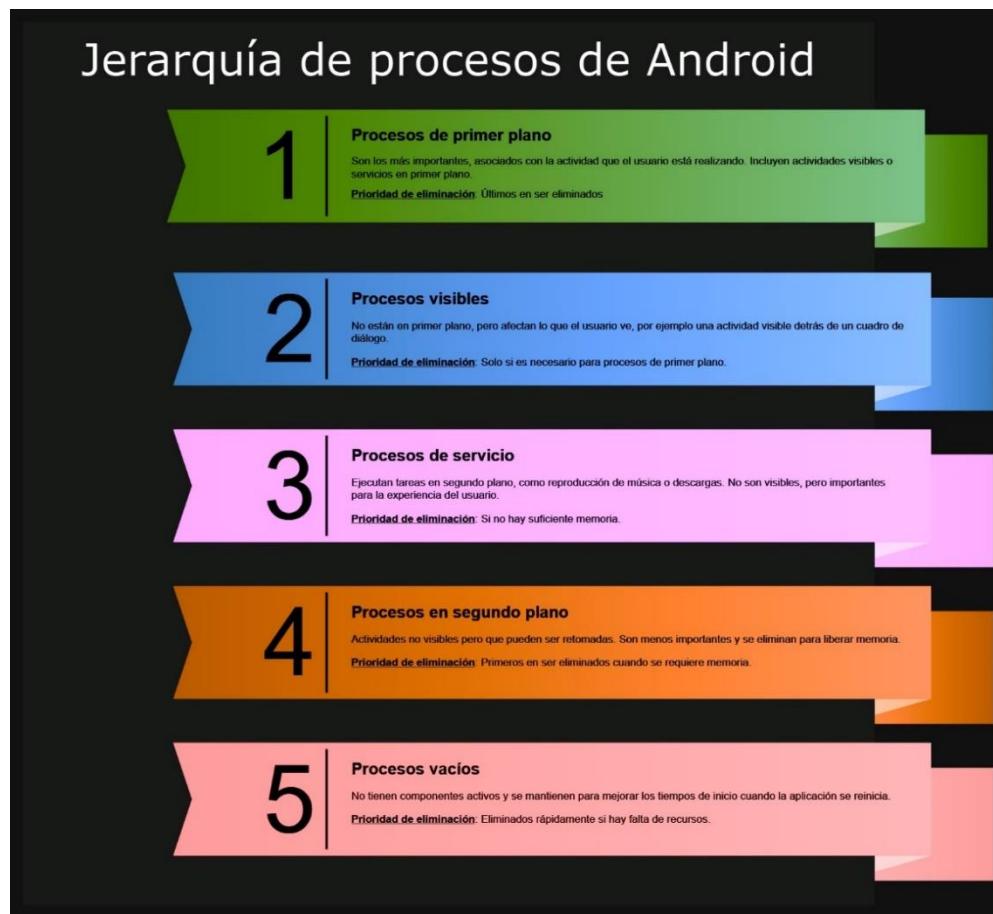
En el sistema Android cada aplicación se ejecuta en su propio proceso, cada proceso se crea cuando una parte del código necesita ejecutarse. Los procesos son fundamentales para la ejecución de las aplicaciones, su prioridad de eliminación es gestionada por el SO, este determina cuando crear o eliminar procesos en base a la carga del sistema y necesidades de usuario.

Los procesos son creados a través de system calls o forks y tambien mediante la creacion de threads que comparten el contexto de ejecución del proceso principal.

El sistema Android organiza los procesos en una jerarquía de importancia para decidir cuáles eliminar primero cuando los recursos son limitados. Esta jerarquía tiene cinco niveles:

1. Procesos de primer plano o foreground: Estos son los más importantes y están asociados directamente con la actividad que el usuario este realizando. Estos procesos incluyen actividades visibles en pantalla o servicios en ejecución en primer plano. Solo se eliminan cuando la memoria es extremadamente baja
2. Procesos visibles: Estos procesos no están en primer plano, pero siguen afectando lo que el usuario ve. Por ejemplo, una actividad visible en segundo plano, atrás de un cuadro de diálogo
3. Procesos de servicio: Corren en segundo plano, ejecutando tareas como descargas o reproducción de música. No son visibles, pero son importantes para la experiencia del usuario casi siempre se eliminan cuando no hay suficiente memoria
4. Procesos en segundo plano o background: Son actividades que no son visibles pero que pueden ser retomadas. Estos procesos son más prescindibles y son los primeros en ser eliminados si es necesario liberar memoria

5. Procesos vacíos: No tienen componentes activos y son mantenidos en memoria solo para mejorar el rendimiento al iniciar de nuevo la aplicación. Son los primeros en ser eliminados cuando los recursos son bajos.



Planificación de procesos

Android utiliza una planificación expropiativa basada en el algoritmo **Round-Robin** y una jerarquía de prioridad. Los procesos se clasifican según su importancia y se ordenan en una lista pseudo-LRU (Last Recently Used) para gestionar los recursos. Los procesos de mayor prioridad influyen en la prioridad de los subprocesos asociados.

Cuando un proceso está en espera o finaliza, se guarda en **zRAM** para acelerar su recarga si se necesita de nuevo. Android garantiza la fluidez deteniendo o finalizando procesos menos prioritarios para asegurar que las aplicaciones más importantes tengan los recursos necesarios.

En Android, la comunicación entre procesos se realiza mediante el paso de mensajes tipo objeto utilizando el **IPC Binder**. Este es el mecanismo de InterProcess Communication (IPC) de Android, que permite que los procesos en ejecución se comuniquen entre sí de forma eficiente. El IPC Binder facilita la sincronización y el intercambio de información entre

procesos independientes del sistema, garantizando la correcta interacción y funcionamiento en el entorno de Android.

Finalización de Procesos

Android puede decidir finalizar un proceso en cualquier momento, especialmente cuando la memoria es baja y otros procesos más críticos requieren recursos. Como resultado, los componentes de la aplicación que se ejecutan en el proceso que se cancela son destruidos. Si es necesario, el sistema puede reiniciar el proceso para esos componentes cuando vuelve a haber trabajo para ellos. Al decidir qué procesos finalizar, Android pondera la importancia relativa de cada uno para el usuario. Por ejemplo, es más probable que se cierre un proceso que aloja actividades que ya no son visibles en pantalla, en comparación con uno que mantiene actividades visibles. Por lo tanto, la decisión de finalizar un proceso depende del estado de los componentes que se ejecutan en ese proceso.

Ciclo de Vida de los Procesos

El sistema Android intenta mantener el proceso de una aplicación el mayor tiempo posible; sin embargo, eventualmente necesita eliminar procesos antiguos para liberar memoria para nuevos procesos o procesos más importantes. Para determinar qué procesos mantener y cuáles finalizar, el sistema clasifica cada proceso en una "jerarquía de importancia" según los componentes que se ejecutan en él y su estado. Los procesos con la importancia más baja se eliminan primero, seguido de aquellos con una importancia ligeramente mayor, y así sucesivamente, según sea necesario para recuperar recursos del sistema.

Android mantiene tantos procesos como permitan los recursos del dispositivo. Cada proceso, correspondiente a una aplicación, estará formado por una o varias actividades independientes llamadas componentes Activity de esa aplicación. Cuando el usuario navega de una actividad a otra, o abre una nueva aplicación, el sistema "pospone" dicho proceso y realiza una copia de su estado para poder recuperarlo más tarde. El proceso y la actividad siguen existiendo en el sistema, pero están en "standby" y su estado ha sido guardado.

Cada uno de los componentes básicos de Android tiene un ciclo de vida bien definido, a continuación, se describen cada uno de ellos:

-**OnCreate(), onDestroy()**

Abarcan todo el ciclo de vida. Cada uno de estos métodos, representan el principio y el fin de la actividad.

-**OnStart(), onStop()**

Representan la parte visible del ciclo de vida. Desde onStart() hasta onStop(), la actividad será visible para el usuario, aunque es posible que no tenga el foco de acción por existir otras actividades superpuestas con las que el usuario está interactuando. Pueden ser llamados múltiples veces.

-**OnResume(), onPause()**

Delimitan la parte útil del ciclo de vida. Desde onResume() hasta onPause(), la actividad no sólo es visible, sino que además tiene el foco de la acción y el usuario puede interactuar con ella.

El proceso que mantiene a esta Activity puede ser eliminado cuando se encuentra en onPause() o en onStop(), es decir, cuando no tiene el foco de la aplicación. (Android nunca elimina procesos con los que el usuario está interactuando en ese momento). Una vez se elimina el proceso, el usuario desconoce dicha situación y puede incluso volver atrás y querer usarlo de nuevo. Entonces el proceso se restaura gracias a una copia y vuelve a estar activo como si no hubiera sido eliminado.

Aunque los recursos son siempre limitados, en el momento en el que Android detecta que no hay los recursos necesarios para poder abrir una nueva aplicación, analiza los procesos existentes en ese momento y elimina los procesos que sean menos prioritarios para poder liberar sus recursos.

Cuando el usuario regresa a una actividad que está en estado standby, el sistema simplemente la activa. En este caso, no es necesario recuperar el estado guardado porque el proceso todavía existe y mantiene el mismo estado. Sin embargo, cuando el usuario quiere regresar a una aplicación cuyo proceso ya no existe porque se necesitaba liberar sus recursos, Android lo crea de nuevo y utiliza el estado previamente guardado para poder restaurar una copia fresca del mismo.

Comunicación entre procesos

Android ofrece un mecanismo para comunicación entre procesos (IPC) Binder, se trata de una parte esencial para Android pues es la que permite a los procesos, las aplicaciones en ejecución, relacionarse con otros procesos de otras partes del sistema, en el caso de cuando se abre una aplicación, establece una comunicación entre el proceso del sistema y el de la aplicación para que la aplicación sepa que se ha abierto.

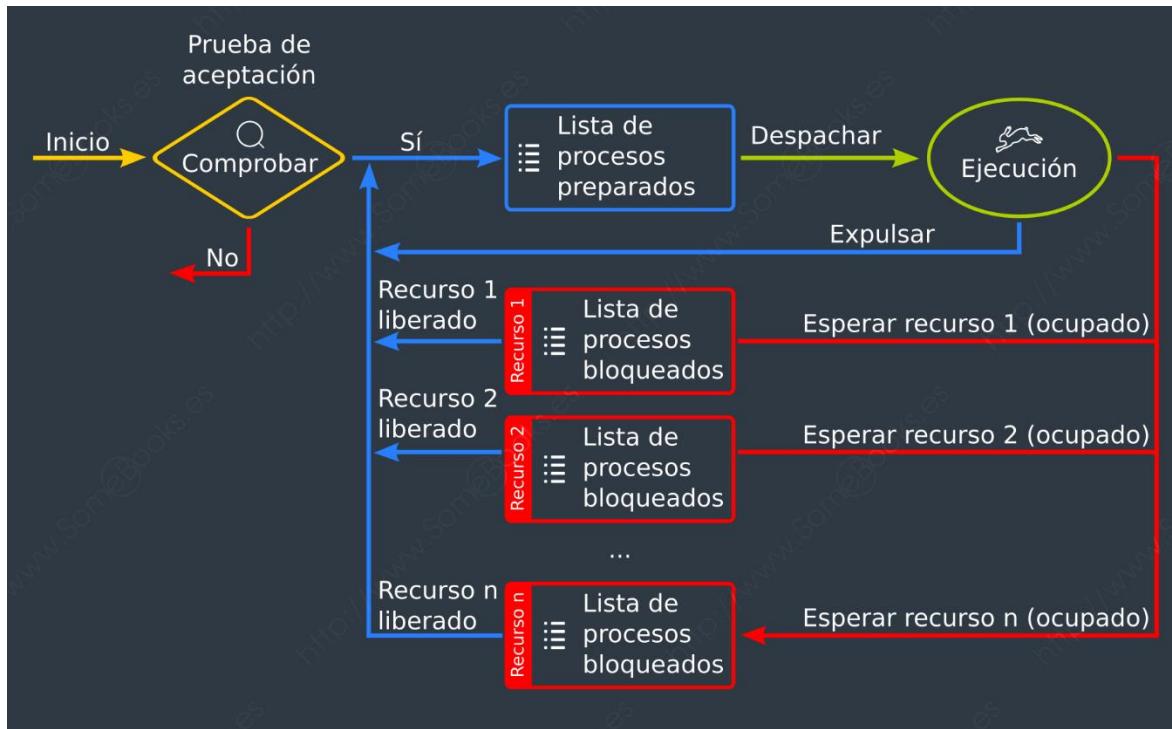
A continuación, se informarán los criterios por los cuales el planificador del procesador selecciona los procesos que serán ejecutados.

En Android básicamente se utilizan dos algoritmos de planificación que son el denominado JSF y el Round Robin.

- Más corto primero con desalojo (JSF): en esta política de asignación es utilizada debido a que da prioridad a los procesos más cortos a la hora de ejecución y los va colocando en una cola, selecciona al proceso con el próximo tiempo de ejecución más corto y lo ejecuta hasta que este proceso termine.
- Round Robin: Este algoritmo consiste en asignar un intervalo de tiempo de ejecución, a este intervalo se lo denomina quantum, por lo cual, si el proceso agota su quantum, este va a elegir otro proceso para que ocupe la CPU.

Para aplicar de manera más eficiente estos criterios, Android los utiliza en forma conjunta con la planificación de colas múltiples. Esta consiste en fragmentar la cola de procesos en

estado Preparado en varias colas más pequeñas, de modo que cada una puede estar administrada por un algoritmo de planificación diferente. De este modo, cada proceso será asignado a una determinada cola en función de sus características pudiendo tratar de manera diferente.



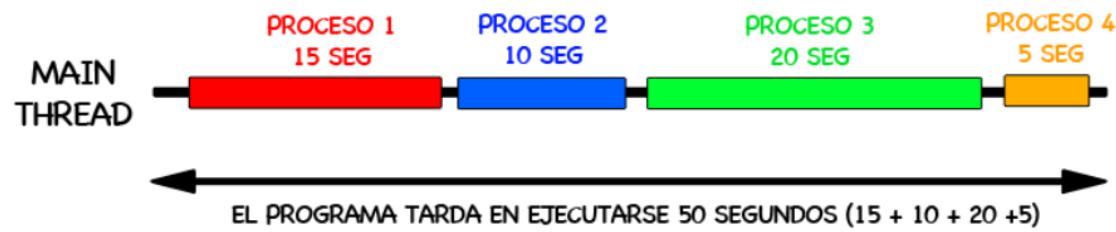
Hilos

Función y ejecución de hilos.

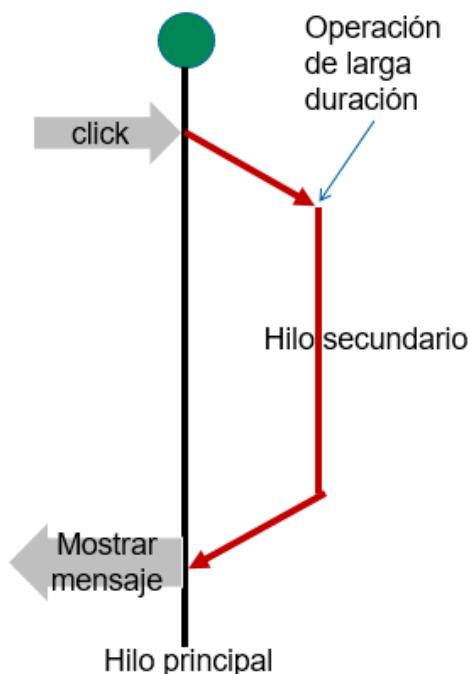
Un hilo, proceso ligero o subproceso es una secuencia de tareas encadenadas muy pequeñas que pueden ser ejecutada al mismo tiempo que otra tarea por el SO.

Como mis compañeros explicaron en las secciones pertinentes al SO, Android es un sistema operativo multitarea. Para proporcionar esta funcionalidad, los procesos que se ejecutan, por ejemplo, el de una aplicación, deben poder descomponerse en hilos. Manteniendo un hilo principal de ejecución.

Dicho hilo es muy importante dado que se encarga de atender los eventos de los distintos componentes. Todas las actividades y servicios de una aplicación son ejecutados por el hilo principal. Para mantener la respuesta de la aplicación, es esencial evitar el uso del hilo principal para realizar cualquier operación que pueda terminar bloqueándola.



Como solución, Android permite crear hilos o subprocessos adicionales, para ejecutar tareas de larga duración. Una vez terminada la tarea podemos regresar al hilo principal para actualizar la interfaz de usuario.



SWAP

1-Intercambio(Swap) en Android:

Todas las aplicaciones que usamos se ejecutan como procesos nativos de Linux. El código ejecutable se guarda como un archivo (un binario compilado, una librería o un código de máquina virtual), en un almacenamiento SD card, SSD, hard disk, etc.

Cuando se inicia un proceso, su código se carga en la RAM para que la CPU pueda realizar operaciones en él. Luego, el proceso sigue asignando más RAM según sea necesario y liberando cuando no es necesario. Pero, ¿qué sucede si hay demasiados procesos en ejecución y la RAM se llena por completo?

La RAM se divide en “pages”, normalmente de 4 KB cada una. Por lo tanto, el kernel de Linux comienza a liberar RAM eliminando páginas que no son compartidas por otros procesos.

Pero, ¿qué sucede si el proceso ha realizado algunos cambios en las páginas, por ejemplo, cuando editas o creas un archivo en una aplicación de edición de texto?

En este caso, el kernel de Linux busca las páginas que no se han tocado (cambiado o utilizado) en los últimos segundos o minutos. Por lo tanto, estas páginas se mueven temporalmente a una parte del almacenamiento llamada Swap. Esta parte puede ser una partición o un archivo. Y tan pronto como el proceso necesita una de estas páginas, se intercambian nuevamente a la RAM.

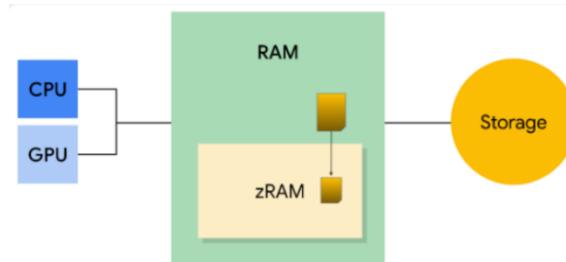
Pero este enfoque presenta dos problemas:

1-El almacenamiento es mucho más lento que la RAM, por lo que afecta al rendimiento.

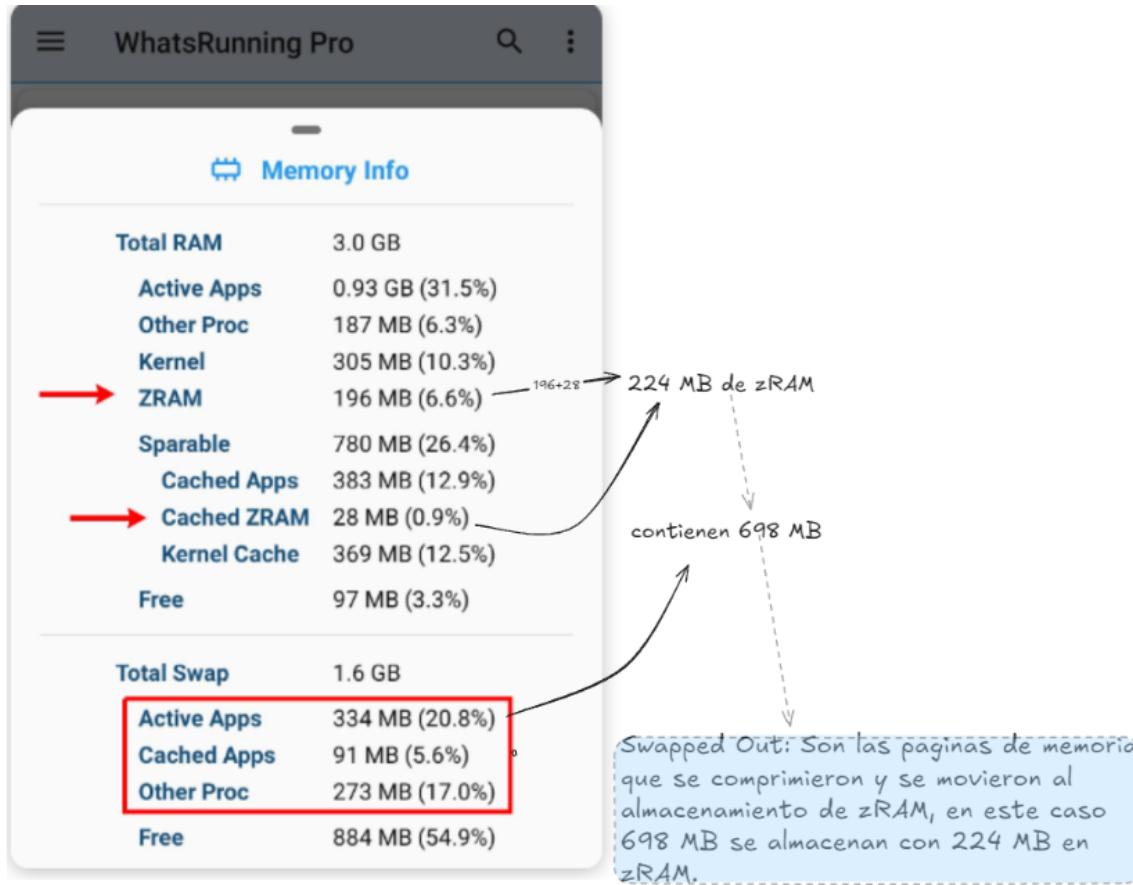
2-Los dispositivos de almacenamiento flash como eMMC, UFS o tarjetas SD tienen una vida útil limitada debido a la cantidad fija de operaciones de lectura y escritura que permiten, por lo que el “Swapping” los elimina rápidamente.

Por lo tanto, Android utiliza una parte comprimida de RAM llamada zRAM para el intercambio, lo que resuelve ambos problemas anteriores.

zRAM es un mecanismo que permite comprimir páginas de memoria y almacenarlas en un área reservada dentro de la propia RAM, lo que es más eficiente en términos de velocidad que intercambiar datos a un disco de almacenamiento (como se hace en sistemas con swap tradicional).



Cuando hablamos de "swapped out pages", nos referimos a las páginas de memoria que ya no están activamente en uso por el sistema o los procesos y que han sido movidas fuera del espacio de RAM principal.



En un sistema tradicional, estas páginas se moverían al **disco (swap)**. Pero en Android, cuando se utiliza zRAM, en lugar de moverse a un disco físico, las páginas se comprimen y se almacenan dentro de un área de la RAM. Así, "**swapped out**" en este caso significa que esas páginas han sido comprimidas en zRAM.

OOM (Out Of Memory) Killer en Android

Android tiene la capacidad de ahorrar memoria, pero aun así puede quedarse sin memoria. Incluso si eso sucede, el funcionamiento del sistema no se ve afectado, vamos a detallar el proceso que utiliza Android.

El kernel de Linux tiene su propio mecanismo de monitoreo, que es **OOM Killer**. Cuando el bloque del sistema alcanza el punto crítico de memoria, esta administración de OOM saltará automáticamente para limpiar el programa en segundo plano.

¿Qué sucede con Android cuando la memoria es baja?

Cuando la memoria es insuficiente, el **kernel** y **ActivityManagerService** (ver System Server) asignan memoria. ActivityManagerService calcula la actividad de baja importancia. Según el resultado, el lado del kernel elimina los procesos menos importantes.

ActivityManagerService calcula la importancia de la actividad en función de las 3 características siguientes:

1-Servicios necesarios para mantener el sistema: Cuando iniciamos Android, sabemos que servicios necesitamos.

2-Proceso de aplicación que proporciona el SDK Service: ActivityManagerService verifica cada proceso para ver si existe un Servicio o Actividad.

3-La actividad que se muestra actualmente en pantalla: ActivityManagerService realiza el cálculo al cargar una nueva Actividad en la stack. El resultado se envía a `oom_score_adj`.

Eliminación de procesos por el Kernel

Según `oom_score_adj`, **lowmemorykiller** y **OOM Killer** eliminan los procesos con puntuaciones altas cuando la memoria es baja. Se realiza en el siguiente orden.

1-El Kernel solicita a “File System” que libere la memoria caché.

2-**Lowmemorykiller** elimina los procesos de aplicación innecesarios.

Este mecanismo libera memoria antes de OOM Killer.

3-Si la memoria no es suficiente luego del paso 2, se setea el Score para notificar el proceso para que se pueda eliminar.

4-OOM Killer elimina el proceso.

Es una función para matar a la fuerza el proceso de `oom_score` que existe en Linux.

Cuando el OMM Killer se activa las aplicaciones se cierran solas o errores como si una base de datos entra en modo “recovery mode”.

OOM Killer, se trata de un mecanismo que posee el kernel para liberar memoria RAM de forma abrupta para evitar el colapso del sistema.

En términos generales, el proceso se finalizará en orden de prioridad, comenzando por el proceso menos importante teniendo en cuenta una puntuación específica.

También se valora la Memoria consumida por el proceso, el tiempo de CPU ocupado por el proceso y el peso OOM dado por omm_adj.

Las prioridades de OOM por defecto serían:

- Mantener un mínimo de memoria RAM libre para que el Kernel pueda funcionar correctamente
- Elimina el mínimo número de procesos posibles
- Priorizar procesos que consuman mucha memoria
- No eliminar procesos que consuman poca memoria

Hay un sistema de prioridades para que el OOM Killer sepa si hay que eliminarlo o no y lo hace revisando el **oom_score**.

Por defecto la mayoría de los procesos tienen una puntuación de 0, es decir que son servicios que ni tienen la prioridad de ser eliminados, ni tampoco tienen ninguna “protección” que les impida ser eliminados.

La puntuación puede ir de -1000 a 1000. -1000 es el valor más bajo y significa que el proceso no se podrá eliminar, por el contrario, 1000 es el valor que hará fuera eliminado primero.

Android y LMK (Low Memory Killer)

Como vemos, Android se basa en la idea central de OOMKiller de linux implementando diferentes clases de Killers.

En el código fuente / drivers / staging / android / lowmemorykiller.c, se registrará una devolución de llamada de reducción durante la inicialización y se llamará a *lowmem_shrinker* cuando la memoria alcance el punto crítico.

```
1 // lowmemorykiller.c
2
3 // 當記憶體到達臨界點時就會呼叫 lowmem_shrinker
4 static struct shrinker lowmem_shrinker = {
5     .shrink = lowmem_shrink,
6     .seeks = DEFAULT_SEEKS * 16
7 };
8
9 static int __init lowmem_init(void)
10 {
11     register_shrinker(&lowmem_shrinker);           // 註冊監聽
12     return 0;
13 }
```

Android hace uso de las funciones lowmem_adj y lowmem_minfree para validar el uso de memoria.

```

1 static int lowmem_adj[6] = {    // 默認 4 個
2     0,
3     1,
4     6,
5     12,
6 };
7 static int lowmem_adj_size = 4;      // 單位大小
8 static int lowmem_minfree[6] = {
9     3 * 512,           /* 6MB */
10    2 * 1024,          /* 8MB */
11    4 * 1024,          /* 16MB */
12    16 * 1024,         /* 64MB */
13 };

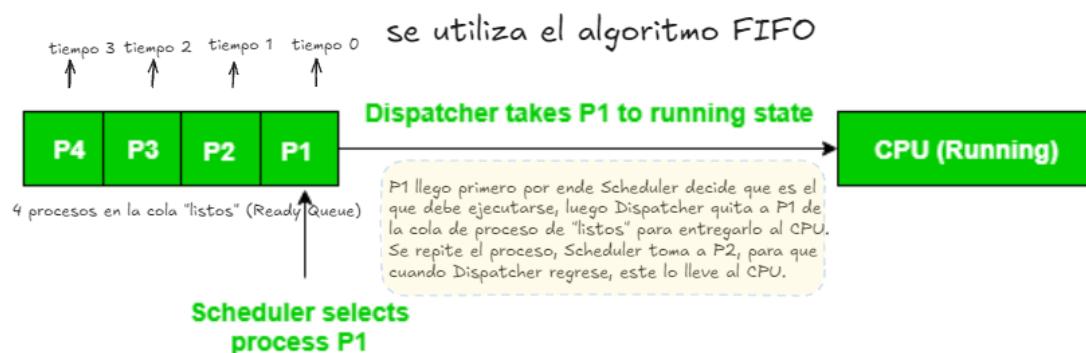
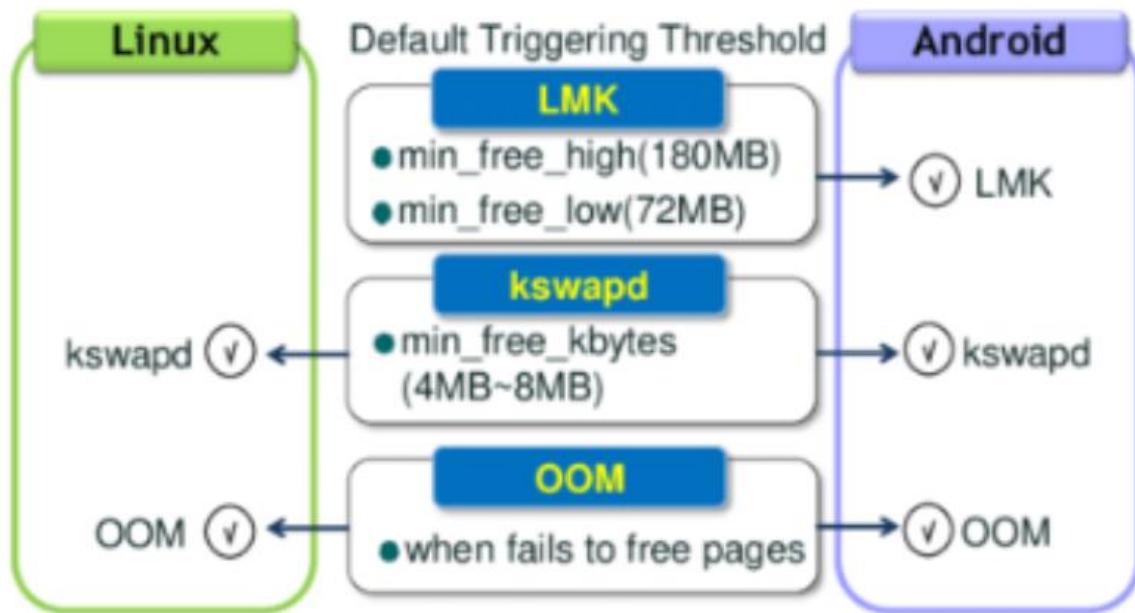
```

OMM Killer y LMK intentan liberar memoria eliminando aplicaciones.

LMK intenta eliminar las aplicaciones en segundo plano, las aplicaciones ocultas y las que están en pausa usando al **ActivityManager de Android** para saber quién se está ejecutando y quién no. Esto permite que el usuario pueda continuar usando su aplicación actual mientras se eliminan otras aplicaciones.

La mayor prioridad de LMK es permitir que el usuario use su aplicación sin problemas. En segundo lugar, LMK en general evitará eliminar aplicaciones del sistema, dando preferencia a las aplicaciones del usuario y permitiendo que el sistema se ejecute.

Por otra parte, OOM intentará matar con prioridad las aplicaciones que utilizan la mayor parte de la memoria, sin preocuparse por el hecho de que esta aplicación estaba siendo utilizada por el usuario en ese momento, la función de OOM es mantener todo el sistema "seguro" y funcionando bien.



EL CPU no puede ejecutar todos los procesos simultáneamente, es por eso que el Scheduler decide cual elegir en base a su algoritmo.

Cuando el Scheduler selecciono un proceso de la cola, el Dispatcher toma al proceso y lo mueve al estado de ejecucion.

MEMORIAS

Generalidad de la administración de la memoria en Android

En el capítulo de carga del S.O vimos la secuencia de arranque la cual se compone de 4 pasos.

1-Encender y arrancar la ROM. Todo sistema comienza con el código de la ROM.

a. La ROM carga el Boot Loader en la RAM

2-Boot Loader: Es el primer programa que se ejecuta.

3-Linux Kernel: El Kernel de Android comienza de manera similar al Kernel de Linux

4-Init

Luego del Init se ejecuta el Zygote, el cual ejecuta al Android Runtime (ART) y la máquina virtual Dalvik, estas usan las funciones de paginación y mapeo de memoria (o llamada mapping) para poder así, administrar la memoria. La única forma de liberar memoria de una app es liberar referencias a objetos contenidos para que la memoria esté disponible para el “recolector de elementos no utilizados”.

Hay una excepción en este proceso, y es que si al código o algún archivo hay que realizarlo un proceso de mapeo sin modificación, este puede quitarse de la RAM si el sistema desea usar esa memoria en otro lugar.

Garbage collection

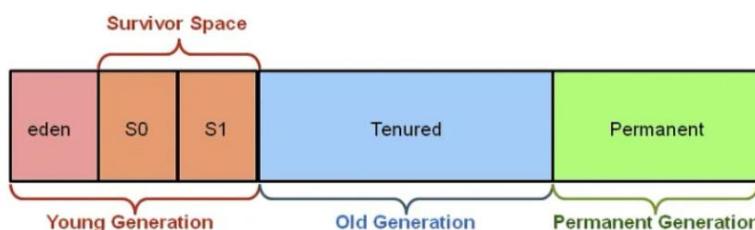
La ART (Android Runtime) o la máquina virtual Dalvik, realiza un seguimiento de cada asignación de memoria. Una vez que determina que el programa ya no usa una porción de memoria, esta es liberada y la devuelve a la **heap**.

El mecanismo para reclamar memoria no utilizada dentro de un entorno de memoria administrada se conoce como **garbage collection**.

Esta tiene dos objetivos:

1. Encontrar objetos de datos en un programa al que no se podrá acceder en el futuro.
2. Reclamar los recursos que esos objetos usan.

La memoria heap tiene “generaciones”, estas son denominadas “young generation”, “older generation” y “permanent generation”.



El concepto de generación Young (o "joven") es el área de memoria donde se almacenan los objetos que han sido creados recientemente y que suelen tener una vida útil corta. Si el objeto pasa un determinado tiempo este es promovido a la “older generation” para luego pasar a “permanent generation”.

Cada “heap generation” tiene su propio límite máximo para la cantidad de memoria que los objetos pueden ocupar. Cada vez que una generación comienza a llenarse, el “garbage collection” ejecuta un evento de recolección de elementos no utilizados a fin de liberar memoria. La duración del “garbage collection” depende de qué generación de objetos se está recolectando y cuántos objetos activos hay en cada generación.

El “**garbage collection**” puede ser rápido, pero también puede afectar el rendimiento de la aplicación, por lo que, es el sistema quien determina cuándo ejecutarlo y no el código en sí. Cuando se cumplen los criterios, el sistema deja de ejecutar el proceso y comienza la “recolección de basura”. Por ejemplo, si el “**garbage collection**” se ejecuta en medio de un bucle de procesamiento intensivo (una animación o durante la reproducción de música), puede aumentar el tiempo de procesamiento.

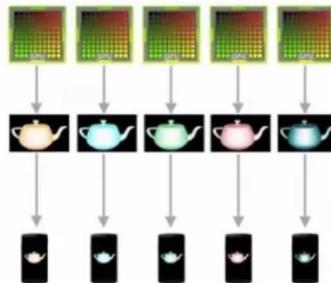
Este aumento puede impulsar la ejecución de código en tu app más allá del umbral recomendado de 16 ms para una representación de fotogramas eficiente y uniforme.

El sistema Android envía una señal VSYNC (Vertical Sync, que en castellano sería “sincronización vertical”) cada 16 ms, que activa el renderizado de la interfaz de usuario. Si cada renderizado es exitoso, puede alcanzar los 60 fps necesarios para una visualización fluida. Para lograr 60 fps, significa que la mayoría de las operaciones de la aplicación deben completarse en 16 ms.

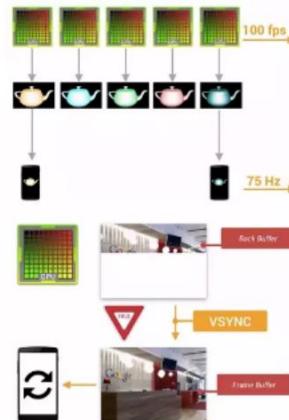


Si una operación específica demora 24 ms, el sistema no podrá realizar la renderización normal cuando reciba la señal VSYNC, lo que provocará la pérdida de fotogramas. En este caso, el usuario verá el mismo fotograma durante 32 ms.

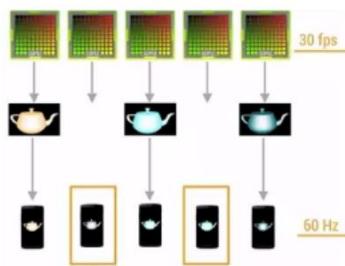




Refresh Rate: representa la cantidad de veces que se actualiza la pantalla (por segundo).
 Depende de los parámetros fijos del hardware. Ej: como 60 Hz.
 Frame Rate: Representa la cantidad de cuadros que dibuja la GPU en un segundo, como 30 fps o 60 fps.
 Trabajan juntos continuamente.
 La GPU renderiza los datos gráficos y el hardware es responsable de presentar el contenido renderizado en la pantalla.



Si Refresh Rate y Frame Rate son inconsistentes, es fácil que se produzcan cortes en la imagen



La peor parte es cuando la velocidad de fotogramas cae repentinamente de más de 60 fps a menos de 60 fps, lo que provoca retrasos

Cómo compartir memoria

Para que todos los procesos necesarios “entren” en la RAM, Android intenta compartir “RAM pages” entre los procesos. Para eso tiene maneras que veremos a continuación:

1-El proceso Zygote comienza cuando el sistema arranca y carga los procesos necesarios (hablados anteriormente). Para iniciar un nuevo proceso de aplicación, el sistema duplica el proceso Zygote, luego carga y ejecuta el código de la aplicación en el nuevo proceso.

El **Zygote** es un proceso pre-inicializado que contiene el código y los recursos del framework de Android, incluyendo bibliotecas y configuraciones comunes que casi todas las aplicaciones necesitan.

Al crear un nuevo proceso para una aplicación, en lugar de cargar desde cero todos estos elementos, el sistema **duplica el proceso Zygote**. Esto hace que el nuevo proceso de aplicación ya contenga muchos elementos básicos del sistema listos para usarse, lo que ahorra tiempo y memoria.

2- Utilización del mmapped (**Memory-mapped**) en el contexto de “datos estáticos”. Esta técnica permite compartir datos entre procesos y también permitir “paginarlos” cuando sea necesario.

Un **dato estático** se refiere a la información que no cambia mientras la aplicación está en ejecución. Es información predefinida y fija que puede ser cargada en memoria y compartida sin modificaciones.

Ejemplos de datos estáticos incluyen:

Código Dalvik (en un archivo .odex precompilado).

Recursos de la aplicación (como imágenes, texto o archivos multimedia que no se modifican en tiempo de ejecución).

Código nativo en archivos .so (bibliotecas de código compilado).

3-En muchos lugares, Android comparte la misma RAM dinámica entre procesos que utilizan regiones de memoria compartida asignadas explícitamente (ya sea con ashmem o gralloc). Por ejemplo, “**Window Surfaces**” utiliza la memoria compartida entre la aplicación y el “screen compositor”, y “cursor buffers” utilizan memoria compartida entre el proveedor de contenido y el cliente.

Ashmem (Android SHared MEMory) y **Gralloc** (Graphics Allocation) son mecanismos para manejar memoria compartida de manera eficiente entre diferentes procesos, especialmente para componentes gráficos y de pantalla.

Ashmem (Android SHared MEMory) es un sistema de gestión de memoria compartida diseñado para que varios procesos puedan acceder a una misma región de memoria sin

duplicarla. Esto permite que aplicaciones y el sistema compartan datos temporales de forma rápida y eficiente. Ashmem es usado para regiones de memoria que no necesitan persistir durante mucho tiempo; por eso, es ideal para compartir datos temporales o buffers que pueden ser eliminados cuando no se necesiten.

Gralloc es un sistema de asignación de memoria específico para gráficos. Es utilizado principalmente para gestionar la memoria de superficies de ventanas y buffers de imagen (texturas, cuadros de video, etc.) que requieren alta eficiencia y baja latencia. Cuando una aplicación necesita dibujar algo en la pantalla (por ejemplo, un video o una imagen), Gralloc se encarga de asignar y gestionar la memoria necesaria para estos gráficos. La memoria que Gralloc asigna también puede ser compartida entre el proceso de la aplicación y el proceso que maneja la pantalla.

Recolección de datos en memoria

Una opción para la fácil recolección de datos consiste en la utilización del Android Debug Bridge (adb) para listar información referida a los procesos, conexiones de red, archivos de registro, etcétera. A través del comando adb shell , el analista puede acceder la shell del equipo para volcar datos desde los archivos del sistema.

También es posible ver las diferentes asignaciones de memoria que afectan a una aplicación determinada, retornando información tanto de memoria Dalvik como de ART.

Una alternativa para la recolección de este tipo de datos es la utilización del Dalvik Debugging Monitor Server (DDMS) mediante el Android Device Monitor. Esta herramienta puede ser inicializada ejecutando el archivo /tools/monitor.

Con ella podemos ver actualizaciones de la pila de memoria, causar la ejecución del GC, o realizar el seguimiento de las asignaciones de memoria entre los diferentes objetos activos. Igualmente podremos acceder a los mensajes generados por la aplicación y el sistema a través de LogCat, a las estadísticas de uso de red y al sistema de archivos presente en el dispositivo.

Un método simple para volcar el contenido de la memoria de un proceso en ejecución utilizando DDMS es crear un archivo HPROF. Para eso, solo necesitamos seleccionar el proceso y hacer clic en la opción “DUMP HPROF file”. De ser necesario, la creación de este archivo puede ser programada desde la aplicación mediante el método dumpHprofData().

Para analizar el archivo resultante es necesario convertirlo previamente a formato J2SE HPROF usando la herramienta /platform-tools/hprof-conv. Lo anterior, solo si no hemos utilizado el Android Device Monitor embebido en Eclipse.

Finalmente, podremos analizar el volcado de memoria con la aplicación Eclipse Memory Analyzer Tool (MAT) o cualquier otro analizador de memoria de nuestra preferencia. Entre otras cosas, podremos analizar las cadenas de texto encerradas en objetos del tipo String que estén activos en memoria; para ello, podemos ingresar la consulta “select * from java.lang.String” en el Object Query Language studio de Eclipse MAT.

La gran desventaja del análisis de memoria con DDMS es que para acceder a ciertas estadísticas la aplicación debe poder ser depurada; es decir, su manifiesto necesita presentar la etiqueta `android:debuggable` con un valor booleano verdadero para indicar al sistema que abra un puerto de depuración. Para superar esta limitación, siempre es posible alterar el APK.

Obtener información en tiempo real de la memoria del equipo puede resultar un proceso extremadamente sencillo mediante las herramientas de análisis incluidas en el kit de desarrollo de la plataforma. La utilización de estas utilidades originariamente pensadas para la evaluación del rendimiento de aplicaciones se vuelve imprescindible en el análisis de códigos maliciosos cada vez más cambiantes.

La capacidad de recuperar archivos desde la memoria, datos en texto plano, historial de la navegación web, mensajes SMS, correos electrónicos, contactos, archivos APK y DEX instalados en el sistema, es esencial no solo para generar un perfil forense completo del equipo, sino también para analizar cómo cambian muestras de malware capaz de mutar dinámicamente.

MÁQUINA VIRTUAL JAVA

La máquina virtual Java es un componente clave en Android, pero en lugar de usar una VM Java tradicional, Android utiliza su propia versión para optimizar la multitarea en dispositivos. Inicialmente, se empleó la máquina virtual **Dalvik** con compilación JIT (Just In Time), que traducía a bytecode Java durante la ejecución. A partir de Android 4.4, se introdujo **ART (Android RunTime)**, que compila las aplicaciones en el momento de la instalación y desde Android 5.0 reemplazó completamente a Dalvik.

Las aplicaciones Android se distribuyen como archivos **.APK** y son compiladas a bytecode por ART durante la instalación. La gestión de memoria y procesos no la maneja directamente la VM, sino el núcleo de Linux, lo que permite un entorno multitarea fiable, similar a un sandbox, con una instancia de la máquina virtual para cada proceso.

Gestión de Memoria en Android.

La gestión de memoria en Android se basa en aprovechar al máximo la memoria disponible, ya que considera que la memoria libre es memoria desperdiciada. Para esto, **Android Runtime (ART)** y la antigua máquina virtual **Dalvik** utilizan funciones de **paginación** y **mapeo de memoria (mmapping)** para gestionar eficientemente el uso de la memoria en los dispositivos.

La jerarquía de memoria en Android se organiza en tres niveles:

1. **RAM**: Es la memoria principal del dispositivo y se gestiona mediante paginación.
2. **zRAM**: Es un área de intercambio (SWAP) donde se almacenan datos de forma comprimida para maximizar el uso de la RAM.
3. **ROM**: Está compuesta por la memoria EEPROM Flash integrada (eMMC) y, opcionalmente, por una tarjeta SD.

Además, existe un nivel cero que incluye la **memoria caché y registros del microprocesador**, pero esta es gestionada directamente por el microprocesador y no por Android.

Paginación de la memoria.

La memoria RAM en Android se divide en páginas de 4 KB, que pueden estar libres o en uso. Las páginas usadas se clasifican en:

1. **Almacenamiento en caché:**
 - **Privada:** Utilizada por un solo proceso.
 - **Compartida:** Utilizada por varios procesos.
2. **Anónima:** No está respaldada por un archivo, como en el caso de bloques de datos temporales.

Las páginas pueden ser **limpias** o **sucias**. Las páginas limpias contienen una copia exacta respaldada en ROM y pueden descartarse para liberar memoria. Las páginas sucias, que han sido modificadas, no se pueden eliminar, ya que contienen información no respaldada y se perdería al hacerlo.

Algoritmo de Reemplazo de página.

El algoritmo de reemplazo de página en Android es el **PFRA (Page Frame Reclamation Algorithm)**, basado en el algoritmo **LRU (Least Recently Used)** pero adaptado para priorizar procesos activos. Las páginas de memoria se dividen en dos listas: una para aplicaciones activas y otra para inactivas. Las páginas inactivas pueden ser marcadas como **swappables** o **descartables**.

El PFRA solo se ejecuta cuando la memoria empieza a escasear, seleccionando primero las páginas descartables/limpias y luego las menos usadas recientemente de la lista de inactivas. Este proceso es incremental, utilizando **zRAM** para almacenar páginas sucias y liberando más memoria si es necesario.

La implementación de este algoritmo es gestionada por el daemon **kswapd**, que se activa cuando la memoria baja a un umbral mínimo. Si aún no hay suficiente memoria, el **Low Memory Killer (LMK)** interviene, eliminando procesos activos según su prioridad, comenzando con los procesos en segundo plano menos utilizados y, si es necesario, avanzando hasta los procesos más importantes del sistema.

Administración de memoria baja

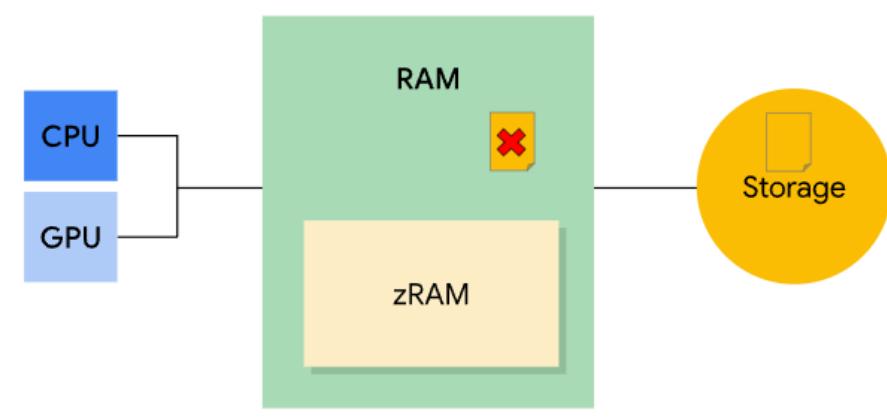
Android tiene dos mecanismos principales para controlar situaciones de poca memoria: el daemon de intercambio de kernel y el optimizador de poca memoria.

Daemon de intercambio de kernel (kswapd)

El daemon de intercambio de kernel (kswapd) es parte del kernel de Linux y convierte la memoria usada en memoria libre. Se activa el daemon cuando queda poca memoria libre en

el dispositivo. El kernel de Linux mantiene umbrales de memoria libre mínimos y máximos. Cuando la memoria libre cae por debajo del umbral mínimo, kswapd comienza a reclamar memoria. Una vez que la memoria libre alcanza el umbral máximo, kswapd deja de reclamar.

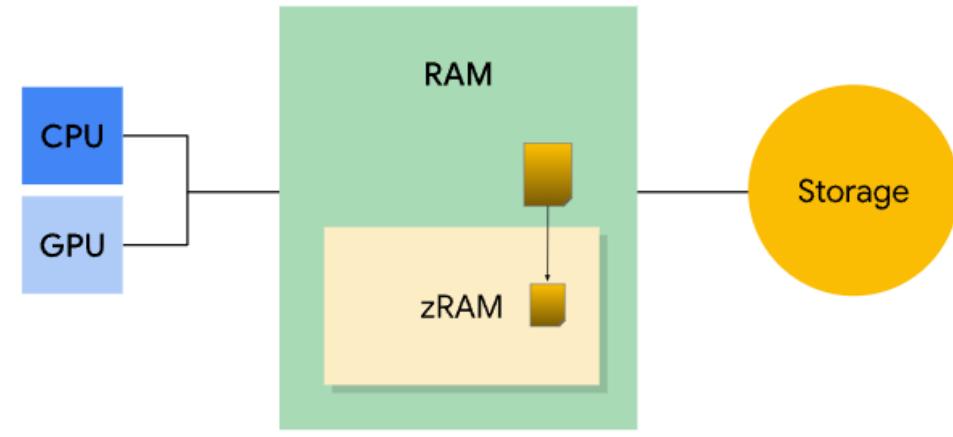
kswapd puede reclamar páginas limpias borrándolas porque están respaldadas por el almacenamiento y no se modificaron. Si un proceso intenta abordar una página limpia que se borró, el sistema copia la página del almacenamiento y la coloca en la memoria RAM. Esta operación se conoce como paginación por demanda.



Página limpia, respaldada por almacenamiento, borrada

kswapd puede mover páginas no sincronizadas privadas en caché y páginas no sincronizadas anónimas a zRAM, donde se comprimen. Esto hace que se libere memoria disponible en RAM (páginas libres). Si un proceso intenta tocar una página no sincronizada en zRAM, se descomprime la página y vuelve a la RAM. Si se cierra el proceso asociado con una página comprimida, esta se borra de zRAM.

Si la cantidad de memoria libre es inferior a un umbral determinado, el sistema comienza a cerrar procesos.



Página sucia trasladada a zRAM y comprimida

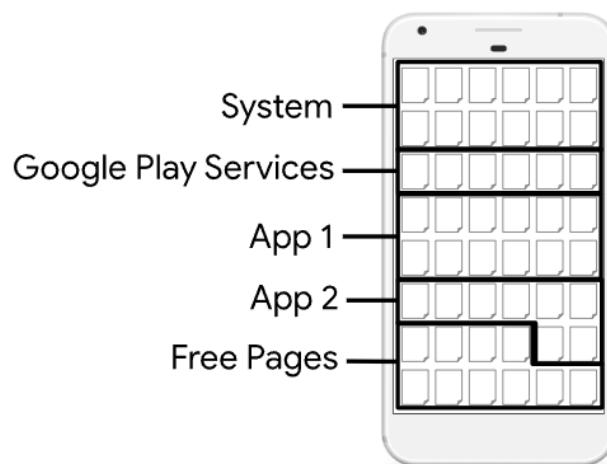
Optimizador de poca memoria

Muchas veces, kswapd no puede liberar suficiente memoria para el sistema. En este caso, el sistema usa onTrimMemory() a fin de notificar a una app que se está agotando la memoria y que debería reducir sus asignaciones. Si esto no es suficiente, el kernel comienza a cerrar procesos para liberar memoria. Para lograrlo, usa el optimizador de poca memoria (LMK).

A los efectos de decidir qué proceso finalizar, LMK usa una puntuación de "memoria insuficiente" llamada oom_adj_score para priorizar los procesos en ejecución. Los procesos con una puntuación alta son los primeros en cerrarse. Primero se deben cerrar las apps en segundo plano, mientras que los procesos del sistema son los últimos que se cierran.

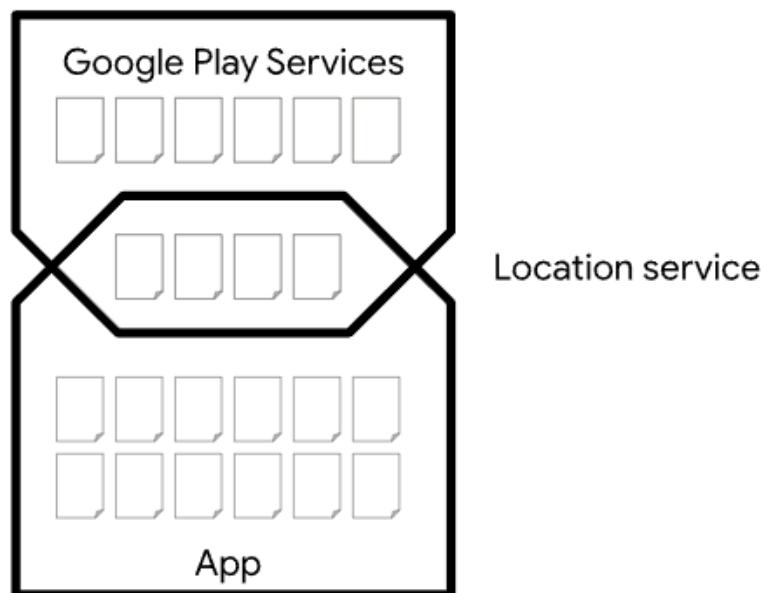
Como calcular el uso de memoria

El kernel realiza un seguimiento de todas las páginas de memoria del sistema.



Páginas usadas por diferentes procesos

Cuando el sistema determina cuánta memoria utiliza una app, debe tener en cuenta las páginas compartidas. Las apps que accedan al mismo servicio o a la misma biblioteca compartirán páginas de memoria. Por ejemplo, es posible que los Servicios de Google Play y una app de juegos compartan un servicio de ubicación. Esto dificulta la tarea de determinar cuánta memoria pertenece al servicio en general y cuánta a cada app.



Páginas compartidas por dos aplicaciones (centro).

Para determinar el uso de memoria de una app, se puede usar cualquiera de las siguientes métricas:

- Tamaño del conjunto residente (RSS): Es la cantidad de páginas compartidas y no compartidas que utiliza la app.
- Tamaño del conjunto proporcional (PSS): Es la cantidad de páginas no compartidas que utiliza la app y una distribución uniforme de las páginas compartidas (por ejemplo, si tres procesos comparten 3 MB, cada uno obtiene 1 MB en PSS).
- Tamaño del conjunto único (USS): Es la cantidad de páginas no compartidas que utiliza la app (las páginas compartidas no están incluidas).

El PSS es útil para el sistema operativo cuando necesita determinar cuánta memoria usan todos los procesos, ya que las páginas no se cuentan varias veces. El cálculo del PSS es lento porque el sistema debe determinar qué páginas se comparten y cuántos procesos las comparten. El RSS no distingue entre páginas compartidas y no compartidas (lo que hace que el cálculo sea más rápido) y es mejor para hacer un seguimiento de los cambios en la

asignación de memoria.

REDES

La capacidad de conectarse a redes es uno de los pilares clave del sistema operativo Android, ya que permite a las aplicaciones y al sistema interactuar con servidores remotos, compartir datos y acceder a servicios en la nube. Android utiliza una pila de protocolos de red similar a la de Linux, que incluye *Capa de Enlace de Datos* que maneja la comunicación a nivel de hardware, como Wi-Fi o redes móviles (3G, 4G, 5G). *Capa de Red*, utiliza protocolos como IP para enrutar los datos, *Capa de Transporte* usa TCP/UDP para la transmisión de datos y *Capa de Aplicación* son protocolos como HTTP/HTTPS para la comunicación de aplicaciones. Las pilas de protocolos son importantes ya que permite una comunicación fiable y eficiente entre dispositivos en una red. Consiste en múltiples capas y cada una es responsable de una tarea específica en la transmisión y recepción de datos. Android soporta múltiples interfaces de red como *wi-fi*, *datos móviles*, *bluetooth* y *ethernet*, las cuales se detallan a continuación.

Tecnologías de Red Soportadas

Wi-Fi: Android tiene un soporte extenso para redes Wi-Fi, lo que permite a los dispositivos conectarse a redes locales para acceso a Internet o compartir archivos de manera local. Las funciones avanzadas de Wi-Fi incluyen Wi-Fi Direct, que permite la comunicación entre dispositivos sin necesidad de un punto de acceso intermedio.

Bluetooth: Android admite versiones avanzadas de Bluetooth, como Bluetooth 5.0, que permiten conexiones rápidas y estables para transferencias de datos entre dispositivos y la comunicación con dispositivos de bajo consumo (BLE, Bluetooth Low Energy). Esto es esencial para periféricos como auriculares, relojes inteligentes y sensores.

NFC (Near Field Communication): Esta tecnología permite la comunicación entre dispositivos a corta distancia. Es utilizada principalmente para pagos móviles (Google Pay) y para compartir datos entre dispositivos mediante Android Beam (descontinuado en versiones más recientes).

Redes Móviles (3G, 4G, 5G): Android es compatible con todas las tecnologías de redes móviles más importantes, desde las redes 3G hasta las más avanzadas redes 5G. La conectividad a redes móviles es fundamental para el acceso a Internet en movilidad y la sincronización en tiempo real de servicios como correo electrónico y redes sociales.

VPN (Red Privada Virtual): Android ofrece soporte nativo para redes privadas virtuales, lo que permite a los usuarios y las empresas crear túneles seguros para el acceso a redes corporativas o para la navegación segura.

Hotspot y Tethering: Los dispositivos Android permiten a los usuarios compartir su conexión a Internet con otros dispositivos mediante la creación de un punto de acceso inalámbrico (Wi-Fi hotspot) o mediante la conexión directa a través de USB o Bluetooth (tethering).

API de Conectividad

Android proporciona un conjunto de APIs para que los desarrolladores gestionen las conexiones de red dentro de sus aplicaciones. A través de la API de Conectividad, las aplicaciones pueden verificar el estado de la red, conectarse a redes específicas, manejar transferencias de datos en segundo plano y realizar tareas de red sin comprometer el rendimiento del dispositivo.

Android gestiona de manera inteligente el uso de redes para ahorrar batería. Por ejemplo, muchas aplicaciones sincronizan datos en segundo plano solo cuando el dispositivo está conectado a Wi-Fi y a una fuente de energía.

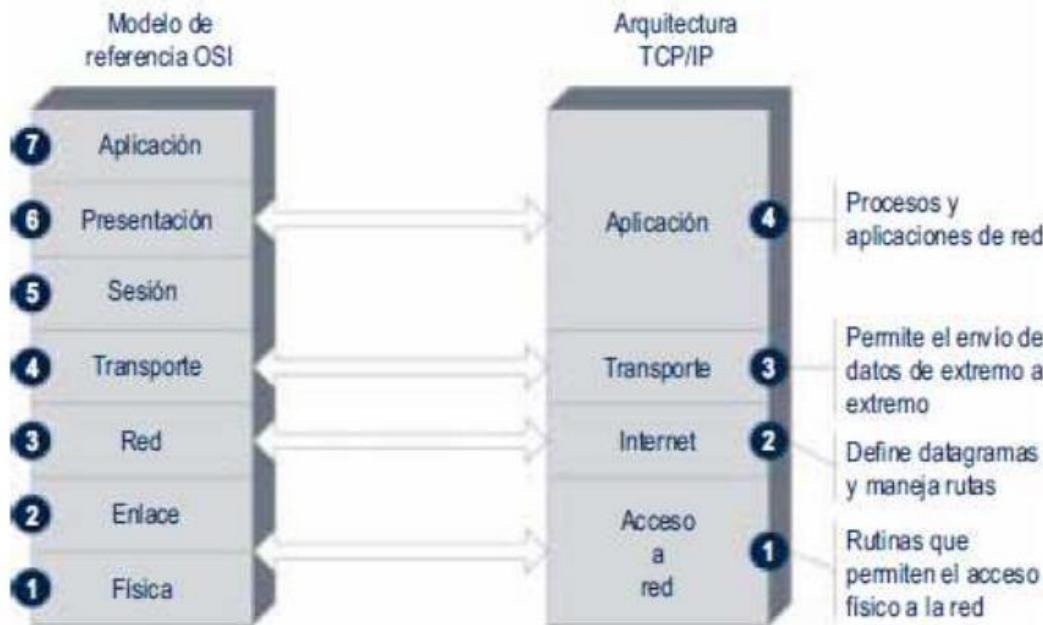
Gestión de Red Móvil

En cuanto a las redes móviles, Android ha introducido mejoras significativas a lo largo de sus versiones para optimizar la transferencia de datos, mejorar la conectividad y manejar de manera eficiente las redes 5G, que permiten velocidades de descarga y carga mucho más rápidas, así como una menor latencia. Además, Android ofrece soporte para VoLTE (Voice over LTE), que permite realizar llamadas de alta calidad a través de redes LTE.

Protocolos de comunicación en red

Los protocolos del modelo TCP/IP cuentan con una gran ventaja: funcionan **con independencia del hardware y el software subyacente**. No importa qué sistema operativo o dispositivo se use para la comunicación a través de la red, porque los protocolos están estandarizados de tal forma que funcionan en cualquier contexto. Sin embargo, se puede afirmar que el modelo OSI es el empleado en el estudio de las redes de datos mientras que el modelo o arquitectura TCP/IP es un modelo real empleado en las redes actuales.

En la siguiente figura se aprecian los niveles o capas de los modelos OSI y TCP/IP.



A continuación, se describe cada una de las capas y los protocolos incluidos en ellas dentro del modelo TCP/IP.

1) Capa de acceso a red

Ofrece la capacidad de acceder a cualquier red física, es decir, brinda los recursos que se deben implementar para transmitir datos a través de la red local. Por tanto, la capa de acceso a la red contiene especificaciones relacionadas con la transmisión de datos por una red física (red local) Ethernet, en anillo, FDDI, etc. En este nivel, dependiendo del hardware de acceso, se define la estructura de datos conocida como **trama** que tendrá una vida útil únicamente en la red local.

En este nivel se definen direcciones físicas o direcciones MAC de los dispositivos. Las tramas emplearán estas direcciones para especificar origen y destino de los datos que transportan. Las direcciones MAC están compuestas por 6 bytes (48 bits) donde los tres primeros bytes identifican al fabricante de la tarjeta de red. Los otros tres bytes identifican a la tarjeta.

2) Capa de Internet

La capa de Internet, también conocida como capa de red o capa IP, acepta y transfiere paquetes para la red. Esta capa incluye el famoso protocolo de Internet (IP), el protocolo de resolución de direcciones (ARP) y el protocolo de mensajes de control de Internet (ICMP).

Protocolo IP

El protocolo IP y sus protocolos de enrutamiento asociados son posiblemente la parte más significativa del conjunto TCP/IP. El protocolo IP se encarga de:

- Direcciones IP: Las convenciones de direcciones IP forman parte del protocolo IP. Se corresponde con la identificación de equipos en la red. Las direcciones IP cambian en función de la red en la que está presente el dispositivo. Las direcciones IP identifican equipos en la conexión extremo a extremo. Actualmente se emplean direcciones de versión 4 (32 bits) y versión 6 (128 bits).
- Encaminamiento: El protocolo IP determina la ruta que debe utilizar un paquete, basándose en la dirección IP del destinatario.
- Formato de paquetes: el protocolo IP agrupa paquetes en unidades conocidas como datagramas. Los datagramas viajan entre el origen y destino IP dentro de las tramas de datos.
- Fragmentación: Si un paquete es demasiado grande para su transmisión a través del medio de red, el protocolo IP del sistema de envío divide el paquete en fragmentos de menor tamaño. Cuando los fragmentos llegan al receptor, el protocolo IP del sistema receptor reconstruye los fragmentos y crea el paquete original.

Protocolo ARP

El protocolo de resolución de direcciones (ARP) se encuentra conceptualmente entre el vínculo de datos (acceso a red) y las capas de Internet. ARP ayuda al protocolo IP a dirigir los datagramas al sistema receptor adecuado realizando la correspondencia entre direcciones MAC (48 bits de longitud) y las direcciones IP conocidas (32 bits de longitud).

Protocolo ICMP

El protocolo de mensajes de control de Internet (ICMP) detecta y registra las condiciones de información y error de la red. Situaciones como la falta de conectividad, problemas en la fragmentación o la redirección de datagramas son detectadas e informadas al origen con mensajes ICMP.

3) Capa de transporte

La capa de transporte TCP/IP se encarga de identificar las aplicaciones que desean conectarse en red. Recibe y envía datos de las aplicaciones y, en función de la seguridad y/o fiabilidad necesaria, así como la rapidez, puede prestar dos tipos de servicio a las aplicaciones: seguro y fiable (TCP), frente a rápido y no fiable (UDP).

Protocolo TCP

TCP permite a las aplicaciones comunicarse entre sí como si estuvieran conectadas físicamente. TCP envía los datos en un formato que se transmite carácter por carácter, en lugar de transmitirse por paquetes discretos. Esta transmisión consiste en establecer una conexión y un control de la llegada de los datos. Cuando todo ha

sido enviado exitosamente se procede al cierre de la conexión. El protocolo de transporte denomina a su estructura de datos como segmento.

TCP, por tanto, confirma que un paquete ha alcanzado su destino dentro de una conexión establecida entre los hosts de envío y recepción. El protocolo TCP se considera un protocolo fiable orientado a la conexión.

Protocolo UDP

UDP proporciona un servicio de entrega de datagramas. UDP no verifica las conexiones entre los hosts transmisores y receptores. Dado que el protocolo UDP elimina los procesos de establecimiento y verificación de las conexiones, resulta ideal para las aplicaciones que envían pequeñas cantidades de datos o cuando el volumen de datos es elevado y éste tiene que llegar en tiempo real (aquí se prefiere la rapidez a la fiabilidad).

4) Capa de aplicación

La capa de aplicación define las aplicaciones de red y los servicios de Internet estándar que puede utilizar un usuario. Estos servicios utilizan la capa de transporte para enviar y recibir datos. Existen varios protocolos estandarizados de aplicación:

- HTTP (páginas web)
- FTP (transferencia de archivos)
- telnet (conexión a servidor)
- SSH (conexión segura a servidor)
- SMTP (correo electrónico)
- POP (correo electrónico)
- DNS (resolución de nombres de dominio)

Sockets

Los sockets son un sistema de comunicación entre procesos de diferentes máquinas en red (ordenadores, smartphones, etc.). Más exactamente, un socket es un punto de comunicación por el cual un proceso puede emitir o recibir información.

Cuando utilizamos Sockets para comunicar procesos nos basamos en la arquitectura cliente y servidor. Así pues, estableceremos dos sockets: uno será la parte servidor y recibirá la transmisión del cliente y otro será la parte cliente que recibirá la respuesta del servidor.

Dependiendo el protocolo con el que vamos a realizar la conexión, tendremos dos tipos de socket, los que utilizan el protocolo TCP, y los que utilizan el protocolo UDP. Ambos sockets utilizan la dirección IP (32 bits) así como el número de puerto (16 bits) de las máquinas

cliente y servidor para poder establecer los puntos de comunicación de los procesos (una conexión estará formada por un par de sockets, que son los extremos de la conexión).

Sockets stream (TCP)

Los sockets stream ofrecen un servicio orientado a conexión, donde los datos se transfieren como un flujo continuo, sin encuadrarlos en registros o bloques. Este tipo de socket se basa en el protocolo TCP que, tal y como se ha comentado antes, es un protocolo orientado a conexión. Esto implica que antes de transmitir información hay que establecer una conexión entre los dos sockets. Mientras uno de los sockets atiende peticiones de conexión (servidor), el otro solicita la conexión (cliente). Una vez que los dos sockets están conectados, ya se puede transmitir datos en ambas direcciones. El protocolo incorpora de forma transparente al programador la corrección de errores. Es decir, si detecta que parte de la información no llegó a su destino correctamente, esta volverá a ser trasmitida. Además, no limita el tamaño máximo de información a transmitir.

Sockets datagram (UDP)

Los sockets datagram se basan en el protocolo UDP y ofrecen un servicio de transporte sin conexión. Es decir, podemos mandar información a un destino sin necesidad de realizar una conexión previa. El protocolo UDP es más eficiente que TCP, pero tiene el inconveniente que no se garantiza la fiabilidad. Además, los datos se envían y reciben en datagramas (paquetes de información) de tamaño no limitado a un valor concreto. La entrega de un datagrama no está garantizada: estos pueden ser duplicados, perdidos o llegar en un orden diferente al que se envió.

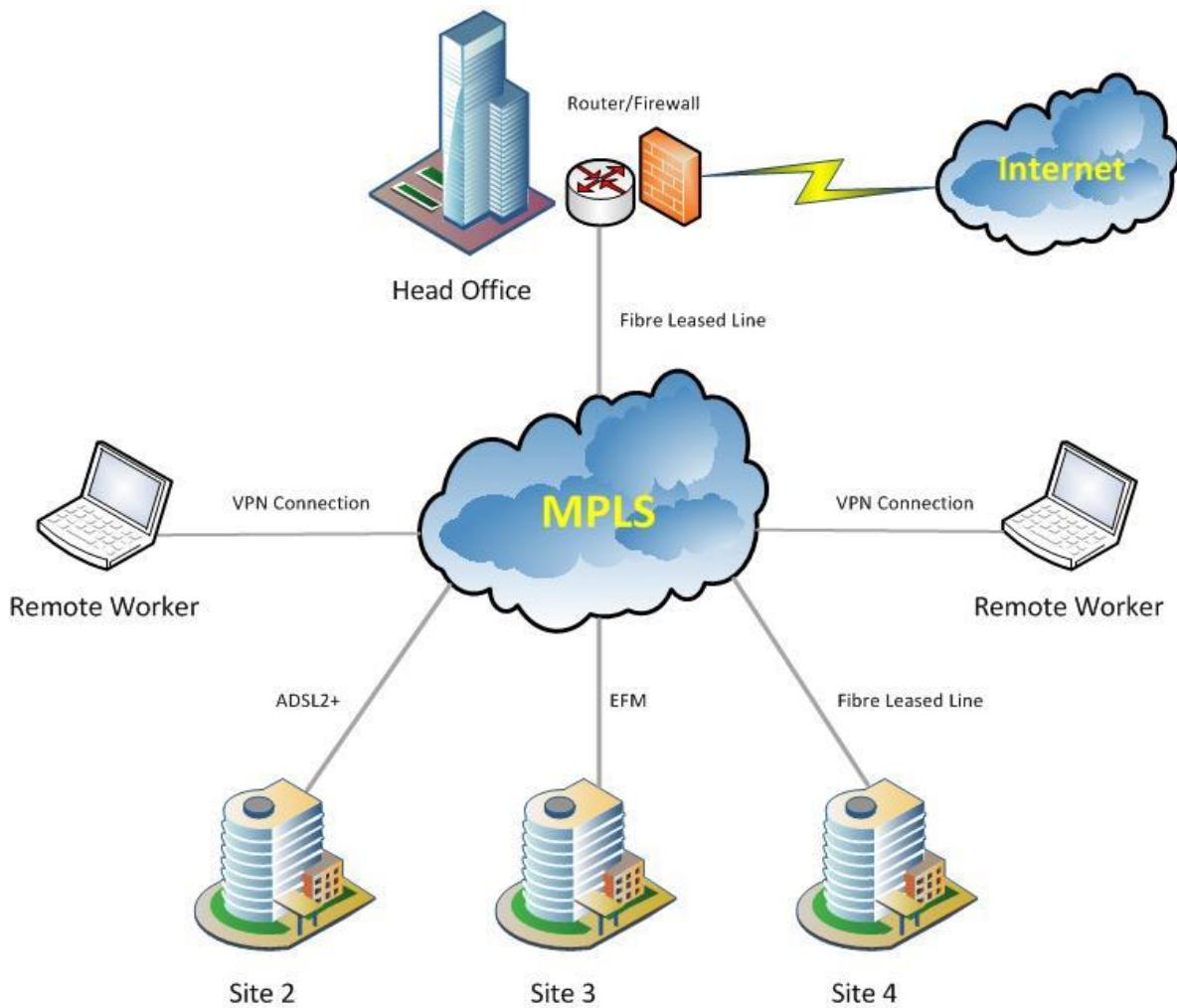
La gran ventaja de este tipo de sockets es que apenas introduce sobrecarga sobre la información transmitida. Además, los retrasos introducidos son mínimos, lo que los hace especialmente interesantes para aplicaciones en tiempo real, como la transmisión de audio y vídeo sobre Internet.

Redes en la actualidad

En el plano de transporte, las redes actuales están basadas en tecnologías de Internet incluyendo el protocolo IP y el MPLS.

MPLS reemplazó a Frame Relay y ATM como la tecnología preferida para llevar datos de alta velocidad y voz digital en una sola conexión. MPLS no solo proporciona una mayor fiabilidad y un mayor rendimiento, sino que a menudo puede reducir los costes de transporte mediante una mayor eficiencia de la red. Su capacidad para dar prioridad a los paquetes que transportan tráfico de voz hace que sea la solución perfecta para llevar las llamadas de voz sobre IP o VoIP.

Representación de las bondades de MPLS para la transmisión de datos.

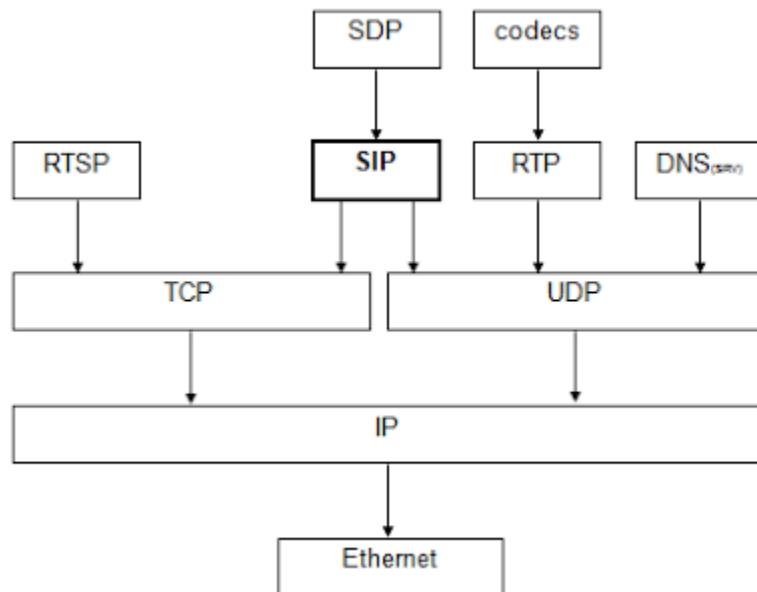


En el plano de control tenemos el subsistema multimedia IP (IMS). IMS es una manera completamente nueva de distribuir multimedia (voz, video, datos, etc.) independiente del dispositivo (teléfono, móvil, o fijo, IPTV, notebook, etc.) o de medio de acceso (3G / EDGE / GPRS, Wi-Fi, banda ancha, línea telefónica, etc.). En este plano también destaca el Softswitch que provee control de llamadas y servicios inteligentes para todo tipo de tráfico multimedia generado.

En el nivel de aplicación, es el protocolo SIP (Session Initiation Protocol) el que destaca como la nueva estandarización para servicios audiovisuales, sustituyendo a H.323. El Protocolo H.323 de la ITU-T (International Telecommunication Union), ha sido hasta día de hoy el protocolo común en sesiones de comunicación audiovisual sobre paquetes de red, especialmente Voz IP y videoconferencia. Es importante indicar que H.323 no garantiza una calidad de servicio, y en el transporte de datos puede, o no, ser fiable; en el caso de voz o video, nunca es fiable. A esto se añade su pésima gestión de NAT y firewalls.

SIP, o Session Initiation Protocol es usado en los sistemas de Telefonía IP. Está basado en HTTP (HyperText Transfer Protocol) adoptando las características más importantes de este

estándar como son la sencillez de su sintaxis y una estructura cliente/servidor basado en un modelo petición/respuesta. Otra de las ventajas de SIP es su sistema de direccionamiento. Las direcciones SIP tienen una estructura parecida a un correo electrónico dotando a sus clientes de una alta movilidad facilitando una posible integración en comunicaciones móviles.



Android proporciona una API que soporta el protocolo de inicio de sesión (SIP). Esto permite añadir funciones de telefonía de Internet basadas en SIP a las aplicaciones. Android incluye una pila de protocolos SIP completa y servicios integrados de gestión de llamadas que permiten a las aplicaciones configurar fácilmente las llamadas de voz entrantes y salientes, sin tener que administrar sesiones, la comunicación a nivel de transporte, grabar audio o reproducir directamente. Las aplicaciones que pueden utilizar la API de SIP proporcionan videoconferencias y/o mensajería instantánea.

SIP se ejecuta a través de una conexión de datos inalámbrica, por lo que el dispositivo debe tener una conexión de datos (con un servicio móvil de datos o Wi-Fi).

Configuración de redes de Android Emulator

El Android Emulator ofrece diversas funciones de red que permiten simular entornos complejos para probar aplicaciones. Cada instancia del emulador opera detrás de un router virtual o firewall, lo que la aísla tanto de Internet como de la red de la máquina de desarrollo. Un dispositivo emulado no puede detectar tu máquina de desarrollo ni otras instancias del emulador en la misma red. Solo reconoce que está conectado a través de Ethernet a un router o firewall.

Para cada instancia, el router virtual administra el espacio de direcciones de red 10.0.2/24. Todas las direcciones administradas por el router tienen el formato 10.0.2.xx, donde xx es un número. El emulador o el router asignan previamente las direcciones dentro de este espacio de la siguiente manera:

Dirección de red	Descripción
10.0.2.1	Dirección del router o la puerta de enlace
10.0.2.2	Alias especial de la interfaz de bucle invertido del host (es decir, 127.0.0.1 en tu máquina de desarrollo)
10.0.2.3	Primer servidor DNS
10.0.2.4/10.0.2.5/10.0.2.6	Segundo, tercero y cuarto servidor DNS opcional
10.0.2.15	La red del dispositivo emulado cuando se conecta mediante Ethernet
10.0.2.16	La red del dispositivo emulado cuando se conecta mediante Wi-Fi
127.0.0.1	La interfaz de bucle invertido del dispositivo emulado

Todas las instancias del emulador en ejecución utilizan las mismas asignaciones de direcciones. Eso significa que, si tienes dos instancias ejecutándose simultáneamente en una máquina, cada una tendrá su propio router y una dirección IP de 10.0.2.15. Las instancias están aisladas por un router y no pueden detectarse entre ellas en la misma red.

La dirección 127.0.0.1 en una máquina de desarrollo corresponde a la interfaz de bucle invertido del emulador. Para acceder a los servicios que se ejecutan en la interfaz de bucle invertido de la máquina de desarrollo, se debe usar la dirección especial 10.0.2.2.

Las direcciones preasignadas de un dispositivo emulado son específicas de Android Emulator y es posible que sean muy diferentes en dispositivos reales (que es probable que también sean traducciones de direcciones de red, específicamente detrás de un router o firewall).

Limitaciones de redes locales

Las aplicaciones Android que se ejecutan en un emulador pueden conectarse a la red de tu estación de trabajo a través del emulador, que actúa como una aplicación más en tu máquina. Esto puede generar algunas limitaciones, como:

Bloqueos en la comunicación con el dispositivo emulado debido a firewalls instalados en tu máquina.

Interrupciones en la conexión causadas por routers o firewalls externos a tu red.

El router virtual del emulador puede gestionar conexiones TCP y UDP salientes del dispositivo emulado, siempre que la configuración de la red de la máquina de desarrollo lo permita. No hay restricciones impuestas por el emulador en los números o rangos de puertos, salvo aquellas establecidas por el sistema operativo o la red host.

El emulador también permite configurar servidores DNS de manera automática al iniciarse, utilizando la lista de servidores DNS del sistema host. Si se experimentan problemas con la resolución de nombres, es posible especificar manualmente los servidores DNS a través de la línea de comandos.

Seguridad con protocolos de red

Las interacciones encriptadas entre cliente y servidor usan la seguridad de la capa de transporte (TLS) para proteger los datos. Un servidor con un certificado TLS tiene una clave pública y una privada coincidente. El servidor usa criptografía de clave pública para firmar su certificado durante el protocolo de enlace TLS.

Un protocolo de enlace simple solo demuestra que el servidor conoce la clave privada del certificado. Para abordar esta situación, permite que el cliente confie en varios certificados. Un servidor determinado no es confiable si su certificado no aparece en el conjunto de certificados de confianza del cliente.

Sin embargo, los servidores pueden usar la rotación de claves para cambiar la clave pública de su certificado por una nueva. El cambio en la configuración del servidor requiere que se actualice la app cliente. Si el servidor es un servicio web de terceros, como un navegador web o una app de correo electrónico, será más difícil saber cuándo actualizar la app cliente.

Por lo general, los servidores dependen de certificados de autoridades certificadoras (AC) para emitir certificados, lo que mantiene la configuración del cliente más estable con el tiempo. Una AC firma un certificado de servidor con su clave privada. Luego, el cliente puede verificar que el servidor tenga un certificado de la AC conocido por la plataforma.

Las AC de confianza suelen enumerarse en la plataforma host. Android 8.0 (nivel de API 26) incluye más de 100 AC que se actualizan en cada versión y no cambian entre dispositivos.

Las apps cliente necesitan un mecanismo para verificar el servidor, ya que la AC ofrece certificados para varios servidores. El certificado de la AC identifica el servidor con un nombre específico, como gmail.com, o con un comodín, como *.google.com.

Para ver la información del certificado del servidor de un sitio web, usa el comando `s_client` de la herramienta de openssl y pasa el número de puerto. De forma predeterminada, HTTPS usa el puerto 443.

El comando transmite el resultado de openssl s_client a openssl x509, que formatea la información del certificado según el estándar X.509. El comando solicita el tema (nombre del servidor) y la entidad emisora (AC).

```
$ openssl s_client -connect WEBSITE-URL:443 | \
openssl x509 -noout -subject -issuer
```

Si suponemos que tienes un servidor web con un certificado emitido por una AC conocida, puedes realizar una solicitud segura, como se muestra en el siguiente código:

```
Kotlin Java

val url = URL("https://wikipedia.org")
val urlConnection: URLConnection = url.openConnection()
val inputStream: InputStream = urlConnection.getInputStream()
copyInputStreamToOutputStream(inputStream, System.out)
```

Para personalizar las solicitudes HTTP, transmite a HttpURLConnection. En la documentación de HttpURLConnection de Android, se incluyen ejemplos para controlar los encabezados de solicitud y respuesta, publicar contenido, administrar cookies, usar proxies, almacenar respuestas en caché y mucho más. El framework de Android verifica los certificados y los nombres de host con estas APIs.

Usa estas APIs siempre que sea posible. En la siguiente sección, se abordan problemas habituales que requieren soluciones diferentes.

Configuración de seguridad de la red

La función Configuración de seguridad de red te permite personalizar los parámetros de configuración de seguridad de red de tu app con un archivo de configuración declarativo y seguro, sin modificar el código de la app. Estas opciones se pueden configurar para dominios específicos y para una app específica. Las funciones clave son las siguientes:

- Anclas de confianza personalizadas: Personaliza qué autoridades certificadoras (AC) son de confianza para las conexiones de seguridad de una app. Por ejemplo, confiar en certificados autofirmados particulares o restringir el conjunto de AC públicas en las que confía la app.

- Anulaciones exclusivas de depuración: Depura de forma segura conexiones de una app sin riesgos adicionales para la base instalada.
- Inhabilitación de tráfico de texto simple: Protege las apps contra el uso accidental del tráfico de texto simple (sin encriptar).
- Fijación de certificados: Restringe la conexión segura de una app con certificados específicos.

Cómo agregar un archivo de configuración de seguridad de red

La función Configuración de seguridad de la red usa un archivo XML en el que se especifican los parámetros para tu app. Debes incluir una entrada en el manifiesto de tu app que haga referencia a este archivo. En el siguiente fragmento de código de un manifiesto, se muestra cómo crear esta entrada:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ... >
    <application android:networkSecurityConfig="@xml/network_security_config"
        ...
    </application>
</manifest>
```

Como administrar el uso de la red

Un dispositivo android puede tener varios tipos de conexiones de red, pero nos centraremos en el uso de una conexión de red Wi-Fi o móvil. Para obtener una lista completa de los tipos de red posibles, consulta **ConnectivityManager**.

Una estrategia común para las apps es recuperar una gran cantidad de datos solo si hay una red Wi-Fi disponible ya que los datos móviles son costosos.

Primero hay que verificar el estado de conectividad de la red. De esta manera, puedes evitar, entre otras cosas, que tu app use el radio incorrecto.

Para verificar la conexión de red, por lo general, se usan las siguientes clases:

- **ConnectivityManager:** responde consultas sobre el estado de la conectividad de red. También notifica a las aplicaciones cuando cambia la conectividad de red.
- **NetworkInfo:** describe el estado de una interfaz de red de un tipo determinado (actualmente, móvil o Wi-Fi).

En este fragmento de código, se prueba la conectividad de red para Wi-Fi y redes móviles.

Ejemplo usando Java

```

private static final String DEBUG_TAG = "NetworkStatusExample";
...
ConnectivityManager connMgr =
        (ConnectivityManager) getSystemService(Context.CONNECTIVITY_SERVICE);
boolean isWifiConn = false;
boolean isMobileConn = false;
for (Network network : connMgr.getAllNetworks()) {
    NetworkInfo networkInfo = connMgr.getNetworkInfo(network);
    if (networkInfo.getType() == ConnectivityManager.TYPE_WIFI) {
        isWifiConn |= networkInfo.isConnected();
    }
    if (networkInfo.getType() == ConnectivityManager.TYPE_MOBILE) {
        isMobileConn |= networkInfo.isConnected();
    }
}
Log.d(DEBUG_TAG, "Wifi connected: " + isWifiConn);
Log.d(DEBUG_TAG, "Mobile connected: " + isMobileConn);
  
```

Ejemplo usando Kotlin

```

private const val DEBUG_TAG = "NetworkStatusExample"
...
val connMgr = getSystemService(Context.CONNECTIVITY_SERVICE) as ConnectivityManager
var isWifiConn: Boolean = false
var isMobileConn: Boolean = false
connMgr.allNetworks.forEach { network ->
    connMgr.getNetworkInfo(network).apply {
        if (type == ConnectivityManager.TYPE_WIFI) {
            isWifiConn = isWifiConn or isConnected
        }
        if (type == ConnectivityManager.TYPE_MOBILE) {
            isMobileConn = isMobileConn or isConnected
        }
    }
}
Log.d(DEBUG_TAG, "Wifi connected: $isWifiConn")
Log.d(DEBUG_TAG, "Mobile connected: $isMobileConn")
  
```

Siempre debes verificar **isConnected()** (y no en función del estado “disponible” de una red), antes de realizar operaciones de red, ya que **isConnected()** se encarga de aspectos como redes móviles inestables, el modo de avión y datos restringidos en segundo plano.

Ejemplo en Java

```
public boolean isOnline() {
    ConnectivityManager connMgr = (ConnectivityManager)
        getSystemService(Context.CONNECTIVITY_SERVICE);
    NetworkInfo networkInfo = connMgr.getActiveNetworkInfo();
    return (networkInfo != null && networkInfo.isConnected());
}
```

Ejemplo en Kotlin

```
fun isOnline(): Boolean {
    val connMgr = getSystemService(Context.CONNECTIVITY_SERVICE) as ConnectivityManager
    val networkInfo: NetworkInfo? = connMgr.activeNetworkInfo
    return networkInfo?.isConnected == true
}
```

Para finalizar **NetworkInfo.DetailedState** se usa poco y es para consultar un estado con mayor profundidad.

Cómo administrar el uso de la red

Puedes implementar una actividad de preferencias que brinde a los usuarios un control explícito sobre el uso de los recursos de red de tu app. Por ejemplo:

- Puedes permitir que los usuarios carguen videos solo cuando el dispositivo esté conectado a una red Wi-Fi.
- Puedes sincronizar la aplicación (o no) según criterios específicos, como la disponibilidad de la red, el intervalo de tiempo, etc.

Para desarrollar una aplicación que permita el acceso a la red y gestione su uso, es esencial incluir los permisos y filtros de intentos adecuados en el manifiesto de la aplicación. A continuación, se presenta un extracto del manifiesto que contiene los siguientes permisos:

android.permission.INTERNET: Este permiso habilita a las aplicaciones para abrir conexiones de red a través de sockets.

android.permission.ACCESS_NETWORK_STATE: Este permiso permite a las aplicaciones acceder a información relacionada con el estado de las redes disponibles.

Además, puedes declarar el filtro de intentos para la acción

ACTION MANAGE NETWORK USAGE. Esto indica que tu aplicación tiene una actividad que proporciona opciones para controlar el uso de datos.

Al utilizar **ACTION_MANAGE_NETWORK_USAGE**, se mostrará la configuración que permite gestionar el uso de datos de red de una aplicación específica. Si tu aplicación incluye una actividad de configuración para que los usuarios puedan regular el uso de la red, es necesario declarar este filtro de intentos para esa actividad.

Las apps que trabajan con datos sensibles del usuario y que se orientan a Android 11 y versiones posteriores pueden otorgar acceso a la red por proceso. Si específicas de manera explícita a qué procesos se les permite el acceso a la red, aislarás todo el código que no necesite subir datos.

Si bien no se garantiza que tu app no suba datos de forma accidental, es una manera de reducir las posibilidades de que se produzcan errores en tu app debido a filtraciones de datos.

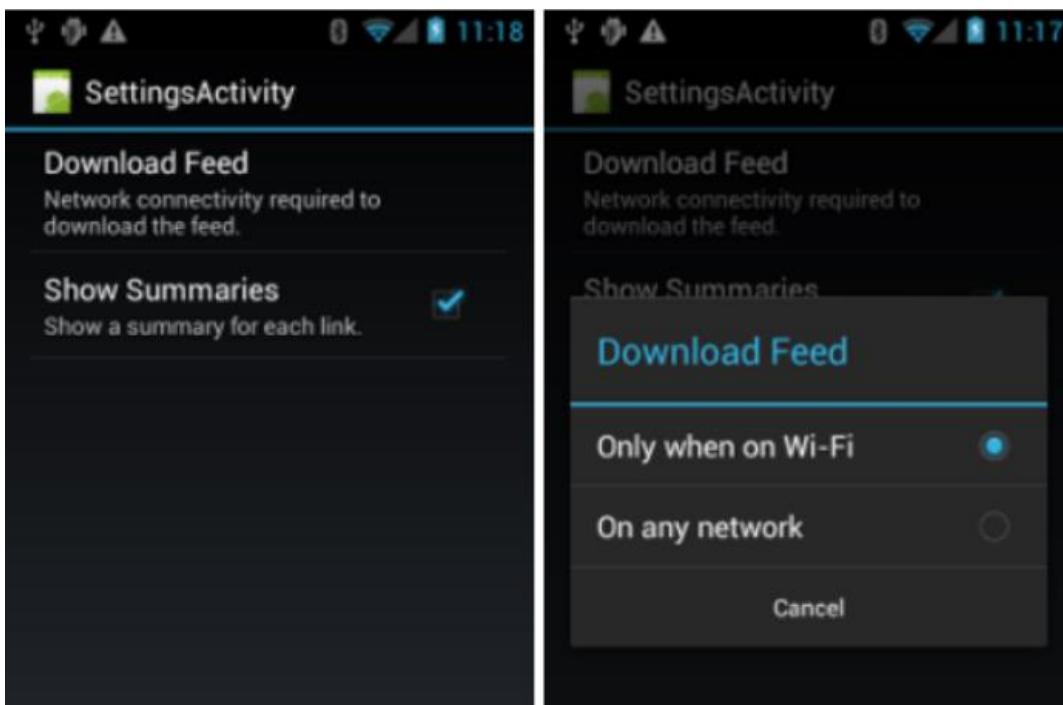
A continuación, se muestra un ejemplo de un archivo de manifiesto en el que se usa la función por proceso:

```
<processes>
  <process />
  <deny-permission android:name="android.permission.INTERNET" />
  <process android:process=":withoutnet1" />
  <process android:process="com.android.cts.usепrocess.withnet1">
    <allow-permission android:name="android.permission.INTERNET" />
  </process>
  <allow-permission android:name="android.permission.INTERNET" />
  <process android:process=":withoutnet2">
    <deny-permission android:name="android.permission.INTERNET" />
  </process>
  <process android:process="com.android.cts.usепrocess.withnet2" />
</processes>
```

Cómo implementar una actividad de preferencia

La actividad de la app muestra **SettingsActivity** tiene un filtro de **intents** para la acción **ACTION_MANAGE_NETWORK_USAGE**. **SettingsActivity** es una subclase de **PreferenceActivity**. Muestra una pantalla de preferencias (aparece en la figura 1) que permite a los usuarios especificar lo siguiente:

- Si desean mostrar resúmenes para cada entrada de feed XML o solo un vínculo para cada entrada.
- Si quieren descargar el feed XML en caso de que haya alguna conexión de red disponible o solo si la conexión Wi-Fi está disponible.



En este caso, es `SettingsActivity`. Ten en cuenta que implementa `OnSharedPreferenceChangeListener`.

Cuando un usuario cambia una preferencia, se activa `onSharedPreferenceChanged()`, que establece el elemento `refreshDisplay` como verdadero. Esto hace que se actualice la pantalla cuando el usuario vuelve a la actividad principal.

Ejemplo Java

```
public class SettingsActivity extends PreferenceActivity implements OnSharedPreferenceChangeListener

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Loads the XML preferences file
        addPreferencesFromResource(R.xml.preferences);
    }

    @Override
    protected void onResume() {
        super.onResume();

        // Registers a listener whenever a key changes
        getPreferenceScreen().getSharedPreferences().registerOnSharedPreferenceChangeListener(this);
    }

    @Override
    protected void onPause() {
        super.onPause();

        // Unregisters the listener set in onResume().
        // It's best practice to unregister listeners when your app isn't using them to cut down on
        // unnecessary system overhead. You do this in onPause().
        getPreferenceScreen().getSharedPreferences().unregisterOnSharedPreferenceChangeListener(this);
    }

    // When the user changes the preferences selection,
    // onSharedPreferenceChanged() restarts the main activity as a new
    // task. Sets the refreshDisplay flag to "true" to indicate that
    // the main activity should update its display.
    // The main activity queries the PreferenceManager to get the latest settings.

    @Override
    public void onSharedPreferenceChanged(SharedPreferences sharedPreferences, String key) {
        // Sets refreshDisplay to true so that when the user returns to the main
        // activity, the display refreshes to reflect the new settings.
        NetworkActivity.refreshDisplay = true;
    }
}
```

Ejemplo con Kotlin

```

class SettingsActivity : PreferenceActivity(), OnSharedPreferenceChangeListener {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        // Loads the XML preferences file
        addPreferencesFromResource(R.xml.preferences)
    }

    override fun onResume() {
        super.onResume()

        // Registers a listener whenever a key changes
        preferenceScreen?.sharedPreferences?.registerOnSharedPreferenceChangeListener(this)
    }

    override fun onPause() {
        super.onPause()

        // Unregisters the listener set in onResume().
        // It's best practice to unregister listeners when your app isn't using them to cut down on
        // unnecessary system overhead. You do this in onPause().
        preferenceScreen?.sharedPreferences?.unregisterOnSharedPreferenceChangeListener(this)
    }

    // When the user changes the preferences selection,
    // onSharedPreferenceChanged() restarts the main activity as a new
    // task. Sets the refreshDisplay flag to "true" to indicate that
    // the main activity should update its display.
    // The main activity queries the PreferenceManager to get the latest settings.

    override fun onSharedPreferenceChanged(sharedPreferences: SharedPreferences, key: String) {
        // Sets refreshDisplay to true so that when the user returns to the main
        // activity, the display refreshes to reflect the new settings.
        NetworkActivity.refreshDisplay = true
    }
}

```

Cómo responder a los cambios de preferencias

Cuando el usuario cambia las preferencias en la pantalla de configuración, suele haber consecuencias en el comportamiento de la app. En este fragmento, la app verifica la configuración de preferencias en onStart(). Si hay una coincidencia entre la configuración y la conexión de red del dispositivo (por ejemplo, si la configuración es "Wi-Fi" y el dispositivo tiene una conexión Wi-Fi), la app descarga el feed y actualiza la pantalla.

Ejemplo en Kotlin

```

class NetworkActivity : Activity() {

    // The BroadcastReceiver that tracks network connectivity changes.
    private lateinit var receiver: NetworkReceiver

    public override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        // Registers BroadcastReceiver to track network connection changes.
        val filter = IntentFilter(ConnectivityManager.CONNECTIVITY_ACTION)
        receiver = NetworkReceiver()
        this.registerReceiver(receiver, filter)
    }

    public override fun onDestroy() {
        super.onDestroy()
        // Unregisters BroadcastReceiver when app is destroyed.
        this.unregisterReceiver(receiver)
    }

    // Refreshes the display if the network connection and the
    // pref settings allow it.

    public override fun onStart() {
        super.onStart()

        // Gets the user's network preference settings
        val sharedPrefs = PreferenceManager.getDefaultSharedPreferences(this)

        // Retrieves a string value for the preferences. The second parameter
        // is the default value to use if a preference value is not found.
        sPref = sharedPrefs.getString("listPref", "Wi-Fi")

        updateConnectedFlags()

        if (refreshDisplay) {
            loadPage()
        }
    }

    // Checks the network connection and sets the wifiConnected and mobileConnected
    // variables accordingly.
    fun updateConnectedFlags() {
        val connMgr = getSystemService(Context.CONNECTIVITY_SERVICE) as ConnectivityManager

        val activeInfo: NetworkInfo? = connMgr.activeNetworkInfo
        if (activeInfo?.isConnected == true) {
            wifiConnected = activeInfo.type == ConnectivityManager.TYPE_WIFI
            mobileConnected = activeInfo.type == ConnectivityManager.TYPE_MOBILE
        } else {
            wifiConnected = false
            mobileConnected = false
        }
    }

    // Uses AsyncTask subclass to download the XML feed from stackoverflow.com.
    fun loadPage() {
        if (sPref == ANY && (wifiConnected || mobileConnected) || sPref == WIFI && wifiConnected) {
            // AsyncTask subclass
            DownloadXmlTask().execute(URL)
        } else {
            showErrorPage()
        }
    }

    companion object {

        const val WIFI = "Wi-Fi"
        const val ANY = "Any"
        const val SO_URL = "http://stackoverflow.com/feeds/tag?tagname=android&sort=:newest"

        // Whether there is a Wi-Fi connection.
        private var wifiConnected = false
        // Whether there is a mobile connection.
        private var mobileConnected = false
        // Whether the display should be refreshed.
        var refreshDisplay = true

        // The user's current network preference setting.
        var sPref: String? = null
    }
}

```

Ejemplo en Java

```

public class NetworkActivity extends Activity {
    public static final String WIFI = "Wi-Fi";
    public static final String ANY = "Any";
    private static final String URL = "http://stackoverflow.com/feeds/tag?tagname=android&sort=newest";

    // Whether there is a Wi-Fi connection.
    private static boolean wifiConnected = false;
    // Whether there is a mobile connection.
    private static boolean mobileConnected = false;
    // Whether the display should be refreshed.
    public static boolean refreshDisplay = true;

    // The user's current network preference setting.
    public static String sPref = null;

    // The BroadcastReceiver that tracks network connectivity changes.
    private NetworkReceiver receiver = new NetworkReceiver();

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Registers BroadcastReceiver to track network connection changes.
        IntentFilter filter = new IntentFilter(ConnectivityManager.CONNECTIVITY_ACTION);
        receiver = new NetworkReceiver();
        this.registerReceiver(receiver, filter);
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
        // Unregisters BroadcastReceiver when app is destroyed.
        if (receiver != null) {
            this.unregisterReceiver(receiver);
        }
    }

    // Refreshes the display if the network connection and the
    // pref settings allow it.

    @Override
    public void onStart () {
        super.onStart();

        // Gets the user's network preference settings
        SharedPreferences sharedPrefs = PreferenceManager.getDefaultSharedPreferences(this);

        // Retrieves a string value for the preferences. The second parameter
        // is the default value to use if a preference value is not found.
        sPref = sharedPrefs.getString("listPref", "Wi-Fi");

        updateConnectedFlags();

        if(refreshDisplay){
            loadPage();
        }
    }

    // Checks the network connection and sets the wifiConnected and mobileConnected
    // variables accordingly.
    public void updateConnectedFlags() {
        ConnectivityManager connMgr = (ConnectivityManager)
            getSystemService(Context.CONNECTIVITY_SERVICE);

        NetworkInfo activeInfo = connMgr.getActiveNetworkInfo();
        if (activeInfo != null && activeInfo.isConnected()) {
            wifiConnected = activeInfo.getType() == ConnectivityManager.TYPE_WIFI;
            mobileConnected = activeInfo.getType() == ConnectivityManager.TYPE_MOBILE;
        } else {
            wifiConnected = false;
            mobileConnected = false;
        }
    }

    // Uses AsyncTask subclass to download the XML feed from stackoverflow.com.
    public void loadPage() {
        if (((sPref.equals(ANY)) && (wifiConnected || mobileConnected))
            || ((sPref.equals(WIFI)) && (wifiConnected))) {
            // AsyncTask subclass
            new DownloadXmlTask().execute(URL);
        } else {
            showErrorPage();
        }
    }
}

```

Cómo detectar cambios de conexión

La pieza final del rompecabezas es la subclase de **BroadcastReceiver**, **NetworkReceiver**. Cuando cambia la conexión de red del dispositivo, **NetworkReceiver** intercepta la acción **CONNECTIVITY_ACTION**, determina el estado de la conexión de red y establece las marcas **wifiConnected** y **mobileConnected** en verdadero o falso, según corresponda. El resultado es que, la próxima vez que el usuario regrese a la app, esta solo descargará el feed más reciente y actualizará la pantalla si **NetworkActivity.refreshDisplay** se establece como true.

Configurar un elemento **BroadcastReceiver** al que se llama de forma innecesaria puede consumir los recursos del sistema. La aplicación de ejemplo registra el **BroadcastReceiver** de **NetworkReceiver** en **onCreate()** y cancela el registro en **onDestroy()**. Esto es más sencillo que declarar un elemento **<receiver>** en el manifiesto. Si declaras un **<receiver>** en el manifiesto, se podría activar tu app en cualquier momento, incluso si no se ejecutó en semanas. Si registras y anulas el registro de **NetworkReceiver** en la actividad principal, te aseguras de que la app no se activará una vez que el usuario salga de ella. Si declaras un elemento **<receiver>** en el manifiesto y sabes exactamente dónde lo necesitas, puedes usar **setComponentEnabledSetting()** para habilitarlo o inhabilitarlo según corresponda.

En este caso, es **NetworkReceiver**:

Ejemplo en Kotlin

```
class NetworkReceiver : BroadcastReceiver() {

    override fun onReceive(context: Context, intent: Intent) {
        val conn = context.getSystemService(Context.CONNECTIVITY_SERVICE) as ConnectivityManager
        val networkInfo: NetworkInfo? = conn.activeNetworkInfo

        // Checks the user prefs and the network connection. Based on the result, decides whether
        // to refresh the display or keep the current display.
        // If the userpref is Wi-Fi only, checks to see if the device has a Wi-Fi connection.
        if (WIFI == sPref && networkInfo?.type == ConnectivityManager.TYPE_WIFI) {
            // If device has its Wi-Fi connection, sets refreshDisplay
            // to true. This causes the display to be refreshed when the user
            // returns to the app.
            refreshDisplay = true
            Toast.makeText(context, R.string.wifi_connected, Toast.LENGTH_SHORT).show()

            // If the setting is ANY network and there is a network connection
            // (which by process of elimination would be mobile), sets refreshDisplay to true.
        } else if (ANY == sPref && networkInfo != null) {
            refreshDisplay = true

            // Otherwise, the app can't download content--either because there is no network
            // connection (mobile or Wi-Fi), or because the pref setting is WIFI, and there
            // is no Wi-Fi connection.
            // Sets refreshDisplay to false.
        } else {
            refreshDisplay = false
            Toast.makeText(context, R.string.lost_connection, Toast.LENGTH_SHORT).show()
        }
    }
}
```

Ejemplo en Java

```

public class NetworkReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
        ConnectivityManager conn = (ConnectivityManager)
            context.getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = conn.getActiveNetworkInfo();

        // Checks the user prefs and the network connection. Based on the result, decides whether
        // to refresh the display or keep the current display.
        // If the userpref is Wi-Fi only, checks to see if the device has a Wi-Fi connection.
        if (WIFI.equals(sPref) && networkInfo != null
            && networkInfo.getType() == ConnectivityManager.TYPE_WIFI) {
            // If device has its Wi-Fi connection, sets refreshDisplay
            // to true. This causes the display to be refreshed when the user
            // returns to the app.
            refreshDisplay = true;
            Toast.makeText(context, R.string.wifi_connected, Toast.LENGTH_SHORT).show();

            // If the setting is ANY network and there is a network connection
            // (which by process of elimination would be mobile), sets refreshDisplay to true.
        } else if (ANY.equals(sPref) && networkInfo != null) {
            refreshDisplay = true;

            // Otherwise, the app can't download content--either because there is no network
            // connection (mobile or Wi-Fi), or because the pref setting is WIFI, and there
            // is no Wi-Fi connection.
            // Sets refreshDisplay to false.
        } else {
            refreshDisplay = false;
            Toast.makeText(context, R.string.lost_connection, Toast.LENGTH_SHORT).show();
        }
    }
}

```

Arquitectura Wlan en Android.



Android utiliza bibliotecas centrales que para ejecutar los servicios; las bibliotecas centrales también llamadas interfaces de programación de aplicaciones (API) se desarrollan utilizando el lenguaje de programación Java (SDK de Android). La API responsable de la gestión inalámbrica es la WiFiManager y se ejecuta en su propio proceso llamado instancia de la DVM. El SDK de Android ofrece un conjunto de aplicaciones para el acceso de datos sobre las redes Wi-Fi, tales como intensidad de la señal, descubrimiento de los puntos de acceso y la ejecución de procesos vinculados a puntos de acceso. Para permitir la conexión Wi-Fi gratuita, hay cierta información requerida que se logra mediante la creación de un permiso explícito mediante el archivo AndroidManifest.xml

Transfer Control Protocol /Internet Protocol (TCP / IP).

Consiste en:

1. Protocolo de Control de Transmisión (TCP), el intercambio de mensajes entre aplicaciones; 34
2. Protocolo de datagramas de usuario (UDP) de comunicación simple entre aplicaciones;
3. Protocolo de Internet (IP) el intercambio de mensajes entre computadoras;
4. Protocolo de mensajes de control de Internet (ICMP) para el envío de mensajes de error y las estadísticas de uso de los recursos;
5. Dynamic Host Configuration Protocol (DHCP) para direccionamiento dinámico de las tabletas o smartphones

Android utiliza bibliotecas centrales que para ejecutar los servicios; las bibliotecas centrales también llamadas interfaces de programación de aplicaciones (API) se desarrollan utilizando el lenguaje de programación Java (SDK de Android). La API responsable de la gestión inalámbrica es la WiFiManager y se ejecuta en su propio proceso llamado instancia de la DVM. El SDK de Android ofrece un conjunto de aplicaciones para el acceso de datos sobre las redes Wi-Fi, tales como intensidad de la señal, descubrimiento de los puntos de acceso y la ejecución de procesos vinculados a puntos de acceso. Para permitir la conexión Wi-Fi gratuita, hay cierta información requerida que se logra mediante la creación de un permiso explícito mediante el archivo AndroidManifest.xml

Pki y SSL

Android garantiza la seguridad en la comunicación de datos en redes, basándose en herramientas criptográficas modernas como el cifrado y la autenticación. Android implementa estas funciones a través de proveedores criptográficos que permiten manejar operaciones seguras como hashing y cifrado, tanto simétrico como asimétrico. Para proteger la privacidad y la integridad de los datos que viajan en la red, Android usa protocolos de seguridad estándar como SSL y TLS.

Estos protocolos, aunque diferentes, cumplen un propósito similar: crear un canal de comunicación segura entre dos puntos (como una aplicación y un servidor). En este proceso, ambas partes negocian un conjunto de algoritmos de cifrado, conocido como "suite de cifrado", y comparten una clave secreta que se usará para cifrar los datos en tránsito.

Para que una comunicación sea segura, el cliente (como un navegador o una app) se comunica primero con el servidor para verificar que ambos soporten la misma versión de SSL/TLS. Luego, se negocia un "conjunto de cifrado" o grupo de algoritmos que aseguran la autenticación y el cifrado de los mensajes. Posteriormente, ambos lados establecen una clave compartida para proteger los datos enviados y recibidos en adelante. En la mayoría de los casos, solo se verifica la identidad del servidor, mientras que la del cliente se valida en casos especiales.

Android utiliza una implementación específica llamada JSSE (Java Secure Socket Extension) para manejar SSL y TLS. A diferencia de otras implementaciones, en Android, JSSE está muy integrado con el sistema para gestionar certificados y verificar anclas de confianza, lo que garantiza que los dispositivos confíen solo en servidores y entidades auténticas y autorizadas.

Además, aunque existen versiones de SSL que permiten conexiones sin autenticar (anónimas), Android evita su uso debido a los riesgos de seguridad, como los ataques de intermediario (MITM), donde alguien podría interceptar la comunicación. En su lugar, se prefieren métodos que incluyan autenticación para reducir vulnerabilidades.

Certificados de clave pública

El protocolo SSL usa **certificados de clave pública** para la autenticación, garantizando así que una comunicación se realice entre entidades de confianza. Un certificado de clave pública es un documento que vincula una identidad específica (como una organización o persona) a una clave pública. En los certificados X.509, que son los usados en SSL, esta identidad incluye varios detalles como el nombre común (CN), la organización, y la ubicación, lo cual forma el "nombre distinguido" (DN) de la entidad. Otros datos importantes son el DN del emisor (quien emitió el certificado), el periodo de validez y extensiones que indican usos específicos para la clave.

La autenticación se logra porque el certificado incluye una **firma digital** creada sobre la clave pública de la entidad y sus datos. Esta firma puede generarse usando la clave privada de la propia entidad (creando un certificado autofirmado) o puede estar firmado por una **autoridad de certificación (CA)** confiable, un tercero que confirma la validez del certificado. Este proceso ayuda a confirmar que la entidad es quien dice ser y que la clave pública es legítima.

Un ejemplo típico de certificado de servidor X.509 puede ser visto a través de herramientas como el comando x509 de OpenSSL. Un certificado específico de Google, por ejemplo, incluiría detalles como el DN (en este caso, "US, California, Mountain View, Google Inc"), una clave pública RSA de 2048 bits y, en ocasiones, nombres DNS alternativos para indicar otros dominios vinculados a la

misma clave pública. Este certificado podría estar firmado por la CA de Google, llamada "Google Internet Authority," lo cual le da confianza en la red.

Confianza directa y CA (Certificate Authority) privadas.

Cuando un cliente SSL se conecta a un grupo pequeño de servidores específicos, se puede configurar una lista de certificados de confianza llamados "anclas de confianza". Con este enfoque, el cliente verifica si el certificado del servidor está en esta lista y, si lo está, confía en la conexión. Este método permite un control muy preciso, pero puede ser incómodo porque requiere emitir nuevos certificados cada vez que se hace un cambio en las claves de los servidores.

Una solución alternativa para evitar emitir certificados constantemente es usar una **CA privada**. Aquí, tanto los clientes como los servidores están configurados para confiar en cualquier certificado emitido por esta CA privada. Esto permite que, mientras la CA siga siendo válida, los certificados se actualicen sin que los clientes tengan que cambiar nada. Sin embargo, esta configuración crea un "punto único de fallo": si la clave de la CA se compromete, un atacante podría emitir certificados fraudulentos y todos los clientes confiarían en ellos. Para resolver esto, sería necesario actualizar el certificado de la CA en todos los clientes, lo cual puede ser complejo.

Limitaciones para clientes genéricos de internet: Este modelo de CA privada no es adecuado para clientes genéricos, como navegadores o aplicaciones de correo, que no saben a qué servidores se conectarán en el futuro. Estos clientes suelen venir preconfigurados con "CA públicas" conocidas. Estas CA son seleccionadas por cada navegador o sistema operativo siguiendo requisitos que varían mucho entre proveedores. Por ejemplo, Mozilla exige que las CA públicas cumplan con prácticas estrictas, como tener documentos de política de certificación pública y usar autenticación multifactor para acceder a cuentas importantes. La mayoría de los navegadores y sistemas operativos incluyen más de 100 CA públicas en las que confían por defecto.

Infraestructura de clave pública

Cuando una autoridad de certificación (CA) pública emite un certificado, realiza una verificación de identidad que varía según el nivel de seguridad del certificado. Los certificados básicos pueden requerir solo confirmación por correo electrónico, mientras que los certificados de Validación Extendida (EV) exigen verificaciones más exhaustivas, como documentos de identidad y registros comerciales.

La infraestructura de clave pública (PKI) incluye personas, sistemas y políticas para emitir y gestionar estos certificados, asegurando que las identidades en las comunicaciones sean confiables. En este proceso, la CA raíz y la CA emisora son piezas clave. La CA raíz firma los certificados de las CAs emisoras, mientras que estas últimas verifican y emiten los certificados finales para usuarios o servidores. La cadena de certificados resultante, que va desde el certificado de la entidad final hasta la CA raíz, actúa como una ruta de confianza. Esta estructura permite validar y asegurar las identidades en la red, proporcionando la base de autenticación y confianza en las comunicaciones seguras.

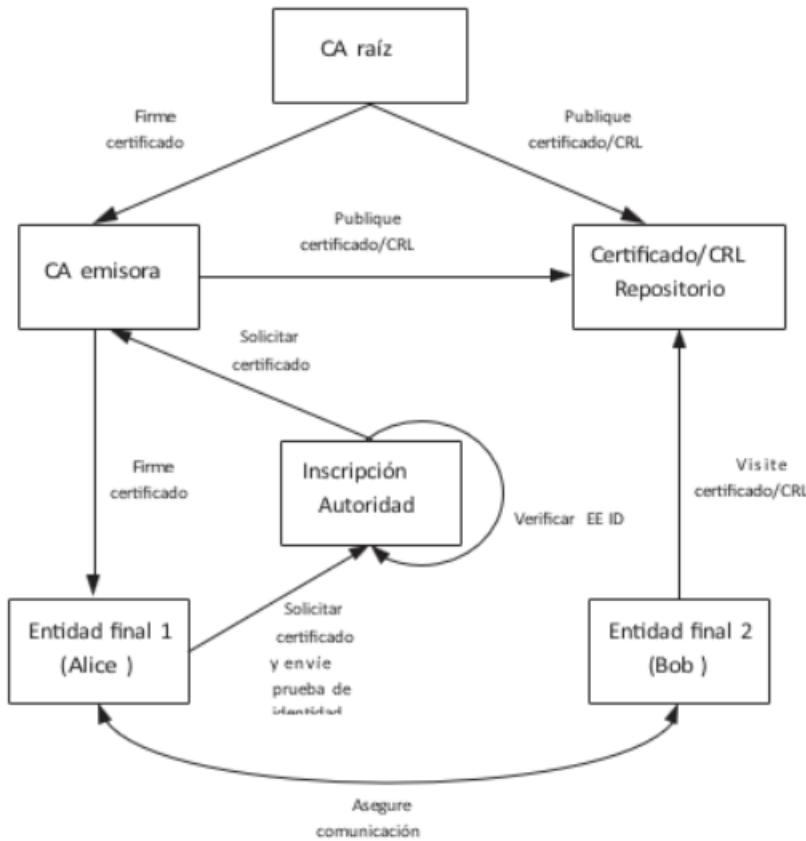


Figura 6-1: Entidades PKI

Zócalos seguros

JSSE en Android permite conexiones seguras SSL/TLS mediante sockets tanto de entrada/salida de bloqueo como de no bloqueo. La clase SSLSocket es esencial para la comunicación de flujo y se configura a través de SSLSocketFactory mediante SSLContext, que gestiona detalles de autenticación y configuración de los sockets SSL. Cuando no se especifica una configuración, se aplican los valores predeterminados del sistema.

Para la autenticación, KeyManager y TrustManager desempeñan roles clave, utilizando KeyStore como almacén de claves y certificados. TrustManager valida las credenciales de la otra parte, mientras que KeyManager selecciona qué credenciales enviar en una conexión, asegurando la autenticación del cliente o del servidor según sea necesario.

Por otro lado, la clase SSLEngine permite un control más directo de la entrada/salida en SSL no bloqueante, ideal para quienes necesitan manejar la comunicación de forma avanzada, asumiendo el control de los buffers de datos y la gestión de hilos.

JSSE también soporta HTTPS mediante HttpsURLConnection, que emplea SSLSocketFactory para conexiones web seguras. Si se necesita personalizar esta configuración, es posible reemplazar SSLSocketFactory globalmente con setDefaultSSLSocketFactory o para una instancia específica

con `setSSLSocketFactory`. Esto permite flexibilidad en la gestión de seguridad SSL, adaptando cada conexión a los requerimientos específicos de la aplicación.

Verificación del nombre de host

En SSL, aunque el certificado del servidor se verifica para confirmar su identidad, el protocolo en sí no obliga a realizar una verificación del nombre de host. Esto significa que, cuando se usan sockets SSL sin procesar, el nombre de host del servidor no se compara con el asunto del certificado, lo que puede crear riesgos de seguridad. Sin embargo, el estándar HTTPS sí requiere esta comprobación, y `HttpsURLConnection` la implementa automáticamente.

Para ajustar la verificación del nombre de host, es posible personalizar el algoritmo de verificación asignando una instancia de HostnameVerifier a `HttpsURLConnection`, ya sea a nivel de clase o por instancia. El método `verify`, que debe implementarse en `HostnameVerifier`, usa la clase `SSLSession` para acceder a los detalles de la conexión SSL, incluyendo el protocolo, el conjunto de cifrado y las cadenas de certificados, así como el nombre de host y el puerto del servidor. Este método evalúa si el nombre de host es confiable en función de la sesión SSL actual.

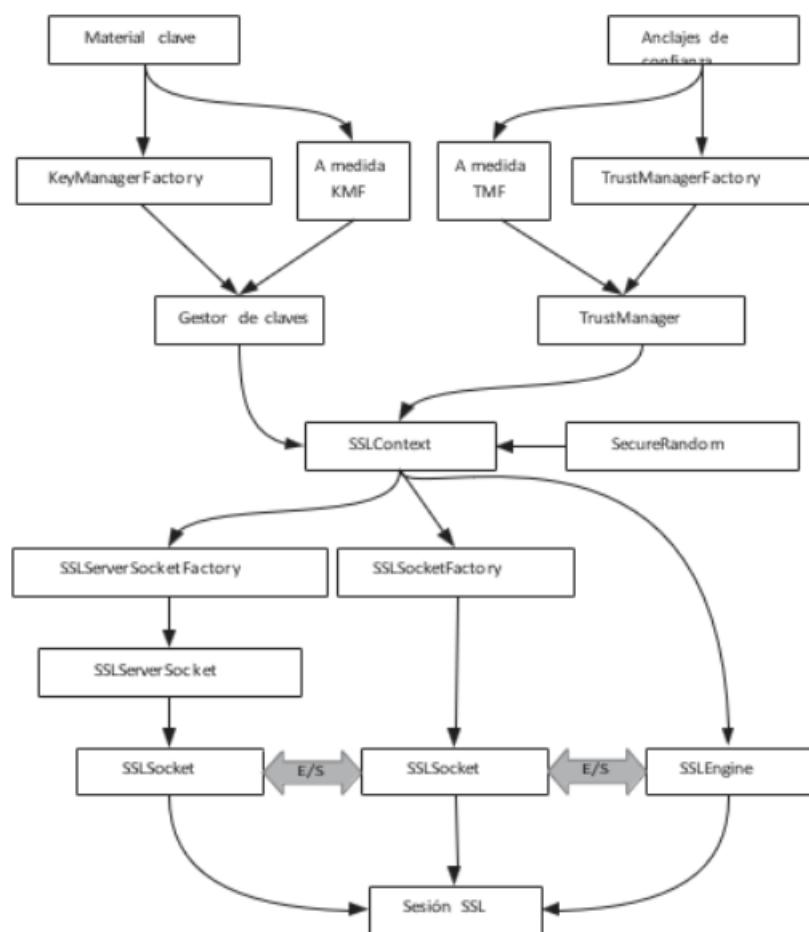


Figura 6-2: Clases JSSE y sus relaciones

Android Implantación de JssE.

Android incluye dos proveedores de JSSE: HarmonyJSSE, basado en Java, y AndroidOpenSSL, implementado en código nativo con acceso a la API de Java mediante JNI. HarmonyJSSE utiliza clases Java para implementar SSL y se limita a soportar SSLv3.0 y TLSv1.0, mientras que AndroidOpenSSL, basado en OpenSSL, es el proveedor preferido en Android y soporta también TLSv1.1 y TLSv1.2. AndroidOpenSSL, además, proporciona las implementaciones predeterminadas de SSLSocketFactory y SSLServerSocketFactory.

Ambos proveedores están integrados en el núcleo de la biblioteca Java, pero AndroidOpenSSL es más completo, permitiendo el uso de SNI (Server Name Indication) desde Android 3.0, lo cual facilita conexiones a servidores con múltiples hosts virtuales. No obstante, esta funcionalidad de SNI no es compatible con la biblioteca cliente HTTP de Apache que también se encuentra en Android.

En cuanto a la evolución de la pila de HTTP, Android dependía del código de Apache Harmony hasta la versión 4.2. A partir de esa versión, Android adoptó OkHttp, una biblioteca de Square, para gestionar conexiones HTTP y SPDY, ofreciendo una implementación más moderna y eficiente.

FILE SYSTEM

Introducción

Un sistema de archivos es un componente de un sistema operativo (en este caso Android) que se encarga de administrar el uso de la memoria física de nuestro dispositivo. Su principal función es la de asignar el espacio a los archivos y administrar el espacio libre que queda, así como gestionar el acceso a los datos protegidos.

Del sistema de archivos depende la velocidad de acceso, lectura y escritura de nuestros datos en la memoria de nuestro dispositivo, por lo que un sistema archivos eficiente es un componente clave si queremos obtener un mayor rendimiento. Otra característica deseable de un sistema de archivos es su robustez, ya que de la fiabilidad del sistema de archivos depende la integridad de nuestros datos.

Así mismo, este tipo de sistemas deben ofrecer un control de acceso a determinados datos, ya que por ejemplo no será recomendable dar acceso a un usuario normal a archivos críticos del sistema, ya que cualquier modificación de estos archivos podría dejar el dispositivo inoperativo.

¿Qué es un sistema de archivos?

En pocas palabras, un sistema de archivos determina cómo se almacenan y recuperan los datos en el almacenamiento local. El sistema operativo Android tiene dos sistemas de archivos, **EXT4** y **F2FS**. Históricamente, se ha utilizado el sistema de archivos EXT4, aunque algunos teléfonos a lo largo de los años han experimentado con el nuevo sistema de archivos F2FS.

Curiosamente, F2FS se presentó en 2012 y fue desarrollado inicialmente por Samsung Electronics como un sistema de archivos con estructura de registro capaz de tener en cuenta las características de los dispositivos de almacenamiento basados en memoria flash NAND. En cambio, la primera versión estable del sistema de archivos EXT4 se lanzó en 2008 como sucesor directo de EXT3.

Si bien hay una gran cantidad de “File System” con implementaciones en el kernel de Linux, muchos no han sido aprobados para su uso en producción en Android y no son compatibles con Android.

Android requiere compatibilidad con cifrado basado en fscrypt (Filesystem-level encryption) y autenticación basada en fsverity (File System Verity o verificación de integridad del sistema de archivos), es decir que si el “File “System” no admite fscrypt o fsverity no pueden ser utilizados en producción.

La siguiente es una tabla que resume las diferencias clave entre los sistemas de archivos ext2, ext3, ext4 y XFS:

Ext2:

El más antiguo de los ext file systems.

Sin registro en diario, lo que lo hace menos confiable.

Adecuado para sistemas más antiguos y escenarios de bajo rendimiento.

Ext3:

Versión de registro en diario de ext2.

Confiabilidad mejorada debido al registro en diario.

Adecuado para la mayoría de los casos de uso de propósito general.

Ext4:

Última versión del ext file systems.

Funciones mejoradas como cambio de tamaño en línea y compatibilidad con archivos grandes.

Mejor rendimiento y confiabilidad que ext2/3.

Ideal para servidores de alto rendimiento y grandes conjuntos de datos.

XFS:

Sistema de archivos de registro en diario de alto rendimiento.

Excelente escalabilidad y confiabilidad.

Admite archivos muy grandes.

Se usa comúnmente en entornos empresariales para cargas de trabajo exigentes.

Diferencias clave:

Registro: Ext3, Ext4 y XFS tienen registro para mejorar la integridad de los datos.

Actualizaciones de metadatos: todos los sistemas de archivos utilizan actualizaciones atómicas de metadatos para lograr coherencia.

Asignación basada en extensión: Ext3, Ext4 y XFS utilizan “extent-based allocation” para un mejor rendimiento.

Redimensionamiento en línea: Ext4 y XFS admiten el redimensionamiento en línea sin necesidad de desmontar el sistema de archivos.

Compatibilidad con archivos grandes: XFS admite los tamaños de archivo más grandes.

Rendimiento: XFS generalmente ofrece el mejor rendimiento.

Fiabilidad: XFS y Ext4 se consideran los más confiables.

Características: XFS ofrece funciones más avanzadas, como capacidades similares a Btrfs.

Casos de uso: Ext2 es adecuado para sistemas más antiguos, Ext3 para uso de propósito general, Ext4 para servidores de alto rendimiento y XFS para cargas de trabajo exigentes.

El mejor sistema de archivos dependerá de las necesidades específicas. Si se necesita un sistema de archivos confiable que sea adecuado para la mayoría de los casos de uso de propósito general, ext3 o ext4 es una buena opción. Si necesita un sistema de archivos de alto rendimiento para grandes conjuntos de datos, XFS es una buena opción.

Ahora que ya sabemos en qué consisten los sistemas de ficheros, cabe preguntarse cuáles tenemos disponibles en Android. En este caso, no tenemos muchas opciones disponibles, solo dos, EXT4 y F2FS.

Jerarquía del sistema de archivos de Android



Los sistemas de archivos de Android se basan en Linux y Unix. El sistema de archivos de Android está organizado en directorios, que pueden contener archivos y subdirectorios. El sistema de archivos de Android está compuesto por seis particiones principales que se encuentran en todos los dispositivos: **/boot**, **/system**, **/recovery**, **/data**, **/cache**, y **/misc**. Algunos dispositivos pueden tener particiones adicionales, como **/sdcard** y **/sd-ext**.

- **La partición de arranque del sistema operativo Android** contiene el código de arranque necesario para iniciar el dispositivo cuando lo enciendes o lo reinicias después de una falla. Contiene diferentes archivos, incluida la imagen del núcleo, la imagen del disco RAM y el proceso zygote (que permite ejecutar las aplicaciones).

- **La partición de datos** contiene todos tus datos, como fotos, videos, archivos de música y otro contenido multimedia almacenado en la memoria interna (o RAM) de tu dispositivo. El software principal de Android no utiliza esta partición; en cambio, está reservada únicamente para el almacenamiento de datos de las aplicaciones del usuario.

Android utiliza su propio formato “ext4” para almacenar estas particiones (y otras similares hechas por otros fabricantes) en lugar de FAT32 debido a mejores características de rendimiento, como velocidades de indexación de archivos más rápidas junto con mejores características de seguridad integradas, como capacidades de cifrado donde solo los usuarios autorizados pueden acceder a esos archivos almacenados dentro de esta parte de los discos duros de sus dispositivos si es necesario.

- **La partición de recuperación** es una pequeña partición en tu dispositivo que se utiliza para iniciar en un modo especial de recuperación del sistema Android. Esto te permite realizar tareas avanzadas de resolución de problemas, como instalar ROM personalizadas y realizar copias de seguridad de tu sistema operativo actual, que suelen ser necesarias cuando algo sale mal con su teléfono o tableta.

Estructura del Directorio de Android

1. **Directorio Raíz (/):** Es el directorio más importante de un dispositivo Android. Los usuarios no pueden modificarlo ni eliminarlo, ya que solo el sistema tiene acceso a él.
2. **Directorio de inicio(/boot):** Contiene los archivos necesarios para iniciar el dispositivo, como el núcleo de Android. No debe borrarse a menos que sea necesario y siempre debe reemplazarse con una nueva partición si se elimina.
3. **Directorio del Sistema (/system):** Contiene los archivos esenciales del sistema operativo Android, como:
 - Archivos APK de las aplicaciones del sistema.
 - Archivos binarios ejecutables.
 - Bibliotecas (**libs**).

Este directorio es de solo lectura, y solo puede modificarse mediante herramientas especializadas que permitan el acceso a archivos protegidos.

4. **Directorio de recuperación(/recovery):** Permite arrancar en un modo de recuperación que facilita realizar copias de seguridad, restaurar el teléfono a la configuración de fábrica y otras tareas de mantenimiento.
5. **Directorio de Datos (/data):** En este directorio se almacenan los datos del usuario, como:
 - Fotos.
 - Música.
 - Videos.
 - Configuraciones y datos de las aplicaciones instaladas.

Al eliminar elementos de este directorio, es importante asegurarse de no eliminar archivos esenciales para el funcionamiento de las aplicaciones.

6. **Directorio de Caché (/cache):** Almacena copias temporales de archivos o datos descargados de Internet. Se puede borrar esta información sin afectar otras partes del sistema, pero es importante revisar lo que se elimina, ya que algunos datos de las aplicaciones podrían necesitarse para un correcto funcionamiento tras reiniciar el dispositivo
7. **Directorio de ajustes(/misc):** Guarda ajustes del sistema, como configuraciones USB y de hardware. Si se daña, puede provocar fallos en el dispositivo.

Por otro lado, la partición **/sdcard** almacena archivos del usuario y, dependiendo del dispositivo, puede referirse tanto al almacenamiento interno como a una tarjeta SD externa. Borrarla no afecta el sistema, pero elimina datos importantes si no se realiza una copia de seguridad previa.

Finalmente, **/sd-ext** es una partición adicional presente en algunos dispositivos, especialmente aquellos con ROM personalizadas, que permite almacenar datos de usuario en la tarjeta SD externa, útil para dispositivos con poca memoria interna.

En resumen, cada partición tiene una función específica en el sistema de archivos de Android, desde gestionar el arranque hasta almacenar datos de usuario, y su manipulación debe realizarse con cuidado para evitar problemas en el funcionamiento del dispositivo.

Sistema de Ficheros de Android

Cada archivo y directorio en el sistema de ficheros de Android tiene permisos de acceso que determinan quien puede leer, escribir o ejecutar el archivo o directorio.

Existen diferentes tipos de archivos que podemos encontrar dentro del sistema de ficheros de Android. A continuación, veremos algunos de ellos:

EXT4 (Extended File System)

En el caso de EXT4, hablamos del sistema de archivos más utilizado en Linux, un sistema que es la cuarta evolución de la familia extended filesystems("fourth extended filesystem") y que es ampliamente utilizado en dispositivos con kernel Linux, no solo en dispositivos Android. Fue creado por Andrew Morton en 2006, como mejora a EXT3. El 25 de diciembre de 2008 se publica el kernel Linux 2.6.28

Es un sistema de archivos en Android diseñado para gestionar grandes volúmenes de datos y funcionar de manera eficiente con discos duros de gran capacidad y alta velocidad. Puede mover archivos de hasta 16 TB y también existe la posibilidad de crear hasta 64.000 subdirectorios, el doble que con EXT3.

Además, cuenta con importantes características de seguridad, como la capacidad de realizar verificaciones de integridad para asegurarse de que los datos no estén dañados, y también permite la recuperación automática en caso de errores. Esto lo convierte en un sistema robusto y confiable para el manejo de archivos en dispositivos Android. Este sistema de archivos es el que la gran mayoría de terminales Android utiliza por defecto.

VFAT

Es un sistema de archivos utilizado en Android principalmente para formatear particiones de almacenamiento externo, como las tarjetas SD. Es una versión mejorada de FAT32, que permite manejar archivos de más de 4GB, lo que lo hace más flexible para gestionar grandes cantidades de datos. En Android, las particiones formateadas en VFAT suelen

montarse en directorios como **/mnt/sdcard** o **/sdcard**, aunque esto puede variar según la versión del sistema o el fabricante del dispositivo. Es común encontrarlo en las tarjetas SD de muchos dispositivos Android.

YAFFS2 (Yet Another File System 2)

YAFFS2 es un sistema de archivos en Android diseñado para funcionar de manera eficiente con memorias flash NAND. Su principal ventaja es garantizar la integridad de los datos, incluso en casos de cortes de energía. Aunque dejó de ser compatible desde la versión Gingerbread de Android, siendo reemplazado por EXT4, YAFFS2 fue diseñado con un algoritmo especial que gestionaba el desgaste de las celdas de memoria flash. Utiliza una estructura de árbol para organizar los datos en bloques y nodos, lo que facilita una lectura y escritura más rápida, además de una mejor administración de la memoria.

F2FS (Flash Friendly File System)

Las siglas significan en inglés "Flash-Friendly File System". Se trata del sistema de archivos creado por Kim Jaegeuk en Samsung para el núcleo Linux (en lo que Android se basa). Fue creado de forma específica por y para que tuviera muy en cuenta las características de los dispositivos de almacenamiento flash, es decir, la forma de memoria que tienen los smartphones en su grandísima mayoría.

F2FS, Flash-Friendly File System es un sistema de archivos creado por Kim Jaegeuk en Samsung y que se integró en el kernel de Linux en la versión 3.6, en octubre de 2012. Como su propio nombre indica, este sistema está diseñado y optimizado para funcionar en memorias flash como las que equipan nuestros smartphones o tablets, los cuales traen una memoria de estado sólido, o SSD, junto a tarjetas SD, en ambos casos con tecnología flash.

Utiliza memoria flash NAND. Diseñado específicamente para este tipo de almacenamiento, F2FS ofrece un rendimiento superior y mayor eficiencia energética en comparación con sistemas como EXT4 o FAT. Su estructura en forma de árbol organiza los datos en bloques y nodos, permitiendo una lectura y escritura más rápida. Además, emplea una técnica de mapeo de direcciones físicas que optimiza el acceso a los datos y reduce la fragmentación, mejorando tanto el rendimiento como la durabilidad de la memoria.

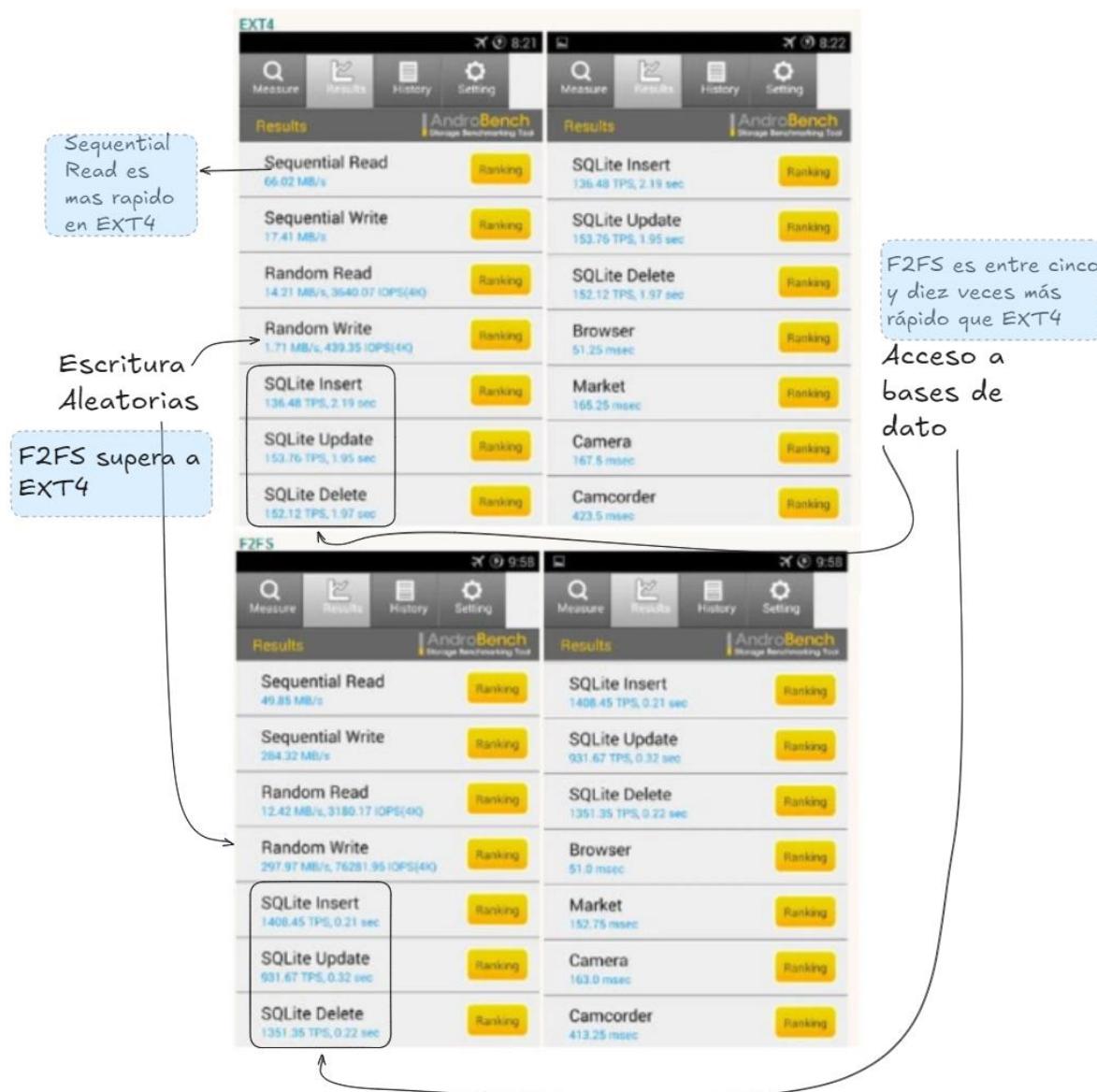
Una de las grandes ventajas de F2FS es su bajo consumo energético durante la lectura y escritura de datos, lo que contribuye a prolongar la duración de la batería en dispositivos Android. También ofrece mayor resistencia al desgaste de las celdas de memoria, garantizando una mayor vida útil para el almacenamiento flash.

RFS (Robust File System)

RFS fue un sistema de archivos utilizado en versiones antiguas de Android, principalmente en dispositivos Samsung, debutando con el Samsung Galaxy S. Diseñado para trabajar con memorias flash NAND, RFS ofrecía una velocidad de lectura y escritura superior a otros sistemas de archivos como EXT4 y FAT. Sin embargo, una de sus principales desventajas era

su baja resistencia al desgaste de las celdas de memoria, lo que podía provocar fallos del sistema y pérdida de datos con el tiempo. Por estas limitaciones, eventualmente fue reemplazado por sistemas más avanzados como EXT4.

En el siguiente cuadro observamos como F2FS es superior a EXT4 tanto en Escrituras Aleatorias como en los accesos a base datos (siendo estos muy comunes en Android). EXT4 solo es mayor a F2FS en algunas pruebas de lectura secuencial.



F2FS vs. EXT4 - Velocidad vs. estabilidad

EXT4 se considera más estable que **F2FS**, ya que ha existido durante más tiempo y no recibe demasiadas actualizaciones en la estructura subyacente. F2FS, por otro lado, puede recibir grandes actualizaciones que traen consigo importantes cambios internos. Esto lo convierte en una opción más difícil para una plataforma como Android OS, donde las configuraciones de hardware pueden variar enormemente y la fragmentación es un problema importante.

Sin embargo, F2FS también puede ser considerablemente más rápido en parte porque es un sistema de archivos estructurado en registros creado principalmente para la memoria flash NAND. Es posible que hayan pasado suficientes años para que F2FS se vuelva más estable, particularmente cuando se combina con UFS 3.0, y ahora parece que Samsung está considerando la idea de realizar la transición a F2FS.

El uso de este sistema de archivos podría ser el principal factor diferenciador entre los dispositivos que usan almacenamiento UFS 3.0. particularmente en términos de velocidades de escritura aleatorias y SQLite, por lo que parecería que el sistema de archivos más nuevo podría llegar muy lejos.

Modelo de seguridad

¿Qué es un daemon?

Un daemon un programa que se ejecuta sin la interacción directa del usuario como un proceso en segundo plano de un sistema operativo (SO). Los daemons son comúnmente utilizados por sistemas tipo Unix como los dispositivos del sistema operativo (SO) Linux. En los sistemas Linux, systemd suele ser el primer daemon que comienza a ejecutarse al arrancar el sistema y es el último en terminar cuando el dispositivo se apaga. Obviamente los deamons también son aplicables en Android ya que este se basa en Linux.

El modelo de seguridad de Android también aprovecha las características de seguridad que ofrece el Shell Linux. Linux es un sistema operativo multiusuario y el núcleo puede aislar los recursos de los usuarios entre sí, al igual que aísla los procesos. En un sistema Linux, un usuario no puede acceder a los archivos de otro usuario (a menos que se le conceda permiso explícitamente) y cada proceso se ejecuta con la identidad (identificación de usuario y grupo, generalmente denominada UID y GID) del usuario que lo inició, a menos que los bits set-user-ID o set-group-ID (SUID y SGID) estén configurados en el archivo ejecutable correspondiente. Android aprovecha este aislamiento de usuarios, pero trata a los usuarios de forma diferente a como lo hace un sistema Linux tradicional (de escritorio o servidor). En un sistema tradicional, se asigna un UID a un usuario físico que puede iniciar sesión en el sistema y ejecutar comandos a través del shell, o a un servicio del sistema (daemon) que se ejecuta en segundo plano (dado que los daemons del sistema suelen ser accesibles a través de la red, ejecutar cada daemon con un UID dedicado puede limitar el daño si uno se

ve comprometido). Android fue diseñado originalmente para teléfonos inteligentes y, dado que los teléfonos móviles son dispositivos personales, no era necesario registrar diferentes usuarios físicos en el sistema. El usuario físico es implícito y los UID se utilizan para distinguir aplicaciones.

Sandbox de aplicaciones

Android asigna automáticamente un UID único, a menudo llamado ID de aplicación, a cada aplicación en la instalación y ejecuta esa aplicación en un proceso dedicado que se ejecuta como ese UID. Además, a cada aplicación se le asigna un directorio de datos dedicado en el que solo ella tiene permiso para leer y escribir. De este modo, las aplicaciones quedan aisladas, o en un entorno aislado, tanto a nivel de proceso (al ejecutarse cada una en un proceso dedicado) como a nivel de archivo (al tener un directorio de datos privado). Esto crea un entorno aislado de aplicación a nivel de kernel, que se aplica a todas las aplicaciones, independientemente de si se ejecutan en un proceso nativo o de máquina virtual. Los daemons y las aplicaciones del sistema se ejecutan con UID constantes y bien definidos, y muy pocos daemons se ejecutan como el usuario raíz (UID 0). Android no tiene el archivo /etc/password tradicional y sus UID de sistema se definen estáticamente en el archivo de encabezado android_filesystem_config.h. Los UID para los servicios del sistema comienzan en 1000, siendo 1000 el usuario del sistema (AID_SYSTEM), que tiene privilegios especiales (pero aún limitados). Los UID generados automáticamente para las aplicaciones comienzan en 10000 (AID_APP), y los nombres de usuario correspondientes tienen el formato app_XXX o uY_aXXX (en las versiones de Android que admiten varios usuarios físicos), donde XXX es el desplazamiento desde AID_APP e Y es el ID de usuario de Android (no es lo mismo que UID).

Los UID de la aplicación se administran junto con otros metadatos del paquete en el archivo /data/system/packages.xml (la fuente canónica) y también se escriben en el archivo /data/system/packages.list.

Permisos

Debido a que las aplicaciones de Android están aisladas, solo pueden acceder a sus propios archivos y a cualquier recurso accesible desde el resto del mundo en el dispositivo. Sin embargo, una aplicación tan limitada no sería muy interesante, y Android puede otorgar derechos de acceso adicionales y detallados a las aplicaciones para permitir una funcionalidad más rica. Esos derechos de acceso se denominan permisos y pueden controlar el acceso a dispositivos de hardware, conectividad a Internet, datos o servicios del sistema operativo. Las aplicaciones pueden solicitar permisos definiéndolos en el archivo AndroidManifest.xml. En el momento de la instalación de la aplicación, Android inspecciona la lista de permisos solicitados y decide si los concede o no. Una vez concedidos, los permisos no se pueden revocar y están disponibles para la aplicación sin ninguna confirmación adicional. Además, para funciones como la clave privada o el acceso a la cuenta de usuario, se requiere la confirmación explícita del usuario para cada objeto al que se acceda, incluso si la aplicación solicitante ha obtenido el permiso correspondiente. Algunos permisos solo se pueden conceder a aplicaciones que forman parte del sistema

operativo Android, ya sea porque están preinstaladas o firmadas con la misma clave que el sistema operativo. Las aplicaciones de terceros pueden definir permisos personalizados y definir restricciones similares conocidas como niveles de protección de permisos, restringiendo así el acceso a los servicios y recursos de una aplicación a las aplicaciones creadas por el mismo autor. El permiso se puede aplicar en diferentes niveles. Las solicitudes a recursos del sistema de nivel inferior, como archivos de dispositivos, se aplican mediante el kernel de Linux al verificar el UID o GID del proceso de llamada con el propietario del recurso y los bits de acceso. Al acceder a componentes de Android de nivel superior, la aplicación la realiza el sistema operativo Android o cada componente (o ambos).

IPC

Android utiliza una combinación de un controlador de kernel y bibliotecas de espacio de usuario para implementar IPC. El controlador de kernel de Binder garantiza que el UID y el PID de los llamantes no se puedan falsificar, y muchos servicios del sistema dependen del UID y el PID proporcionados por Binder para controlar dinámicamente el acceso a las API sensibles expuestas a través de IPC. Por ejemplo, el servicio de administrador de Bluetooth del sistema solo permite que las aplicaciones del sistema habiliten Bluetooth de manera silenciosa si el llamante se ejecuta con el UID del sistema (1000) mediante el código que se muestra en el Listado 1-7. Se puede encontrar un código similar en otros servicios del sistema. El sistema puede aplicar automáticamente permisos más generales que afecten a todos los métodos de un servicio expuesto a través de IPC especificando un permiso en la declaración del servicio. Al igual que con los permisos solicitados, los permisos requeridos se declaran en el archivo AndroidManifest.xml. Al igual que la verificación de permisos dinámica en el ejemplo anterior, los permisos por componente también se implementan consultando el UID del llamante obtenido de Binder en segundo plano. El sistema utiliza la base de datos de paquetes para determinar el permiso requerido por el componente que recibe la llamada y luego asigna el UID del que la llama a un nombre de paquete y recupera el conjunto de permisos otorgados al que la llama. Si el permiso requerido está en ese conjunto, la llamada se realiza correctamente. Si no, falla y el sistema lanza una SecurityException.

Soporte Multiusuario

Debido a que Android fue diseñado originalmente para dispositivos móviles (teléfonos inteligentes) que tienen un solo usuario físico, asigna un UID de Linux distinto a cada aplicación instalada y tradicionalmente no tiene una noción de usuario físico. Android obtuvo soporte para múltiples usuarios físicos en la versión 4.2, pero el soporte multiusuario solo está habilitado en tabletas, que tienen más probabilidades de ser compartidas. El soporte multiusuario en dispositivos móviles se deshabilita al establecer el número máximo de usuarios en 1. A cada usuario se le asigna un ID de usuario único, comenzando con 0, y a los usuarios se les da su propio directorio de datos dedicado en /data/system/users/<user ID>/, que se llama el directorio del sistema del usuario. Este directorio aloja configuraciones específicas del usuario, como parámetros de la pantalla de

inicio, datos de la cuenta y una lista de aplicaciones instaladas actualmente. Si bien los binarios de la aplicación se comparten entre los usuarios, cada usuario obtiene una copia del directorio de datos de una aplicación. Para distinguir las aplicaciones instaladas para cada usuario, Android asigna un nuevo UID efectivo a cada aplicación cuando se instala para un usuario en particular. Este UID efectivo se basa en el ID de usuario del usuario físico de destino y el UID de la aplicación en un sistema de usuario único (el ID de la aplicación). Esta estructura compuesta del UID otorgado garantiza que, incluso si dos usuarios diferentes instalan la misma aplicación, ambas instancias de la aplicación obtienen su propio espacio aislado. Además, Android garantiza un almacenamiento compartido dedicado (alojado en una tarjeta SD para dispositivos más antiguos), que es legible para todo el mundo, para cada usuario físico. El usuario que inicializa primero el dispositivo se denomina propietario del dispositivo, y solo él puede administrar otros usuarios o realizar tareas administrativas que influyan en todo el dispositivo.

SELinux

El modelo de seguridad tradicional de Android se basa en gran medida en los UID y GID otorgados a las aplicaciones. Si bien estos están garantizados por el núcleo y, por defecto, los archivos de cada aplicación son privados, nada impide que una aplicación otorgue acceso mundial a sus archivos (ya sea intencionalmente o debido a un error de programación). De manera similar, nada impide que las aplicaciones maliciosas aprovechen los bits de acceso excesivamente permisivos de los archivos del sistema o los sockets locales. De hecho, los permisos inadecuados asignados a las aplicaciones o los archivos del sistema han sido la fuente de varias vulnerabilidades de Android. Esas vulnerabilidades son inevitables en el modelo de control de acceso predeterminado empleado por Linux, conocido como control de acceso discrecional (DAC). Discrecional aquí significa que una vez que un usuario obtiene acceso a un recurso en particular, puede pasarlo a otro usuario a su discreción, por ejemplo, configurando el modo de acceso de uno de sus archivos como legible para todo el mundo. Por el contrario, el control de acceso obligatorio (MAC) garantiza que el acceso a los recursos se ajuste a un conjunto de reglas de autorización de todo el sistema llamado política. La política solo puede ser modificada por un administrador, y los usuarios no pueden anularla ni omitirla para, por ejemplo, otorgar a todos acceso a sus propios archivos. Security Enhanced Linux (SELinux) es una implementación MAC para el núcleo Linux y ha estado integrada en el núcleo principal durante más de 10 años. A partir de la versión 4.3, Android integra una versión modificada de SELinux del proyecto Security Enhancements for Android (SEAndroid)9 que se ha ampliado para admitir funciones específicas de Android como Binder. En Android, SELinux se utiliza para aislar los daemons del sistema central y las aplicaciones de usuario en diferentes dominios de seguridad y para definir diferentes políticas de acceso para cada dominio. A partir de la versión 4.4, SELinux se implementa en modo de aplicación (las violaciones a la política del sistema generan errores de tiempo de ejecución), pero la aplicación de la política solo se aplica a los daemons del sistema central. Las aplicaciones aún se ejecutan en modo permisivo y las violaciones se registran, pero no causan errores de tiempo de ejecución.

Sistema operativo Android y sistema de archivos Ext4

El sistema operativo Android tiene varias particiones internas y, desde la versión 4.0 de Android (Ice Cream Sandwich), que es actualmente la más utilizada, las particiones /system, /efs, /tombstones, /cache, que contienen datos relacionados con el sistema excluyendo / La partición sdcard, son particiones de usuario. La partición /data, que contiene datos relacionados con datos, utiliza el sistema de archivos Ext4. Android funciona principalmente en aplicaciones como una PC normal, pero los datos no relacionados con las aplicaciones (documentos descargados, archivos de fotos, etc.) se almacenan en la memoria externa (partición/sdcard) con el sistema de archivos vFAT o exFAT, y los datos creados mientras se usan las aplicaciones (SQLite base de datos, archivos XML, etc.) se almacenan en la partición /data. Por lo tanto, la mayoría de los archivos relacionados con las aplicaciones se almacenan en el área del sistema de archivos Ext4.

Estructura del sistema de archivos Ext4

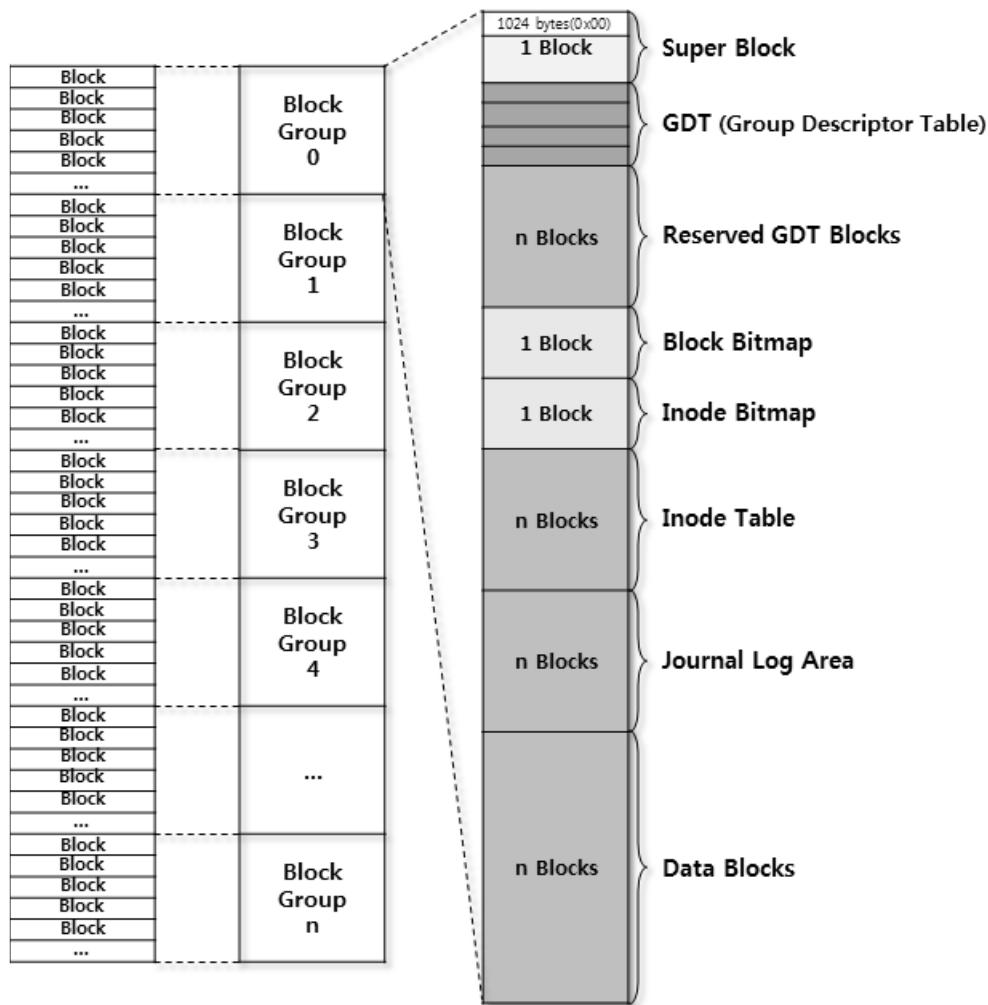
El sistema de archivos Ext4 consta de bloques, que son la unidad mínima de almacenamiento de datos. El tamaño del bloque se puede seleccionar entre 1 KB, 2 KB o 4 KB, pero se utiliza 4 KB de forma predeterminada. Para administrar los datos de manera efectiva, el sistema de archivos Ext4 administra todos los bloques dentro del sistema de archivos Ext4 agrupándolos en varios grupos de bloques.

Un grupo de bloques administra hasta 32,768 bloques, por lo que, si el tamaño del bloque es de 4 KB, el tamaño de un grupo de bloques es de hasta 128 MB, y si el tamaño de la partición del sistema de archivos Ext4 es de 1 GB (1,048,576 MB), hay 8 bloques. grupo ($128\text{ MB} \times 8 = 1.048.576\text{ MB}$).

Dentro del grupo de bloques, hay áreas de metadatos como superbloque, GDT (tabla de descripción de grupo), mapa de bits de bloque, mapa de bits de inodo, tabla de inodo y área de registro de diario. El superbloque contiene información general del sistema de archivos, como el tamaño del bloque, el número total de bloques, el tamaño de la tabla de inodos, el número de tablas de inodos para cada grupo de bloques, etc. Esta área es para los grupos de bloques $0, 2n-1$ ($n \geq 1$). GDT contiene información de ubicación de las áreas de mapa de bits de bloque, mapa de bits de inodo¹ y tabla de inodo que existen en cada grupo de bloques. En las áreas de mapa de bits de bloques y de mapa de bits de inodos, la información de asignación de bloques e inodos dentro del grupo de bloques correspondiente se mapea y se almacena en unidades de bits. El área de la tabla de inodos contiene tablas de inodos para todos los datos del grupo de bloques en orden según el número de inodos, y cada tabla de inodos contiene información de tiempo, como los tiempos de creación, acceso, cambio y eliminación de datos, el tamaño y los datos reales del bloque, punteros, etc

¹ Registro en el disco, este registro contiene la información sobre el archivo o carpeta como su peso, propietario, etc., excepto el contenido y el nombre del archivo.

Por último, el área de registro de diario contiene registros de las transacciones que ocurrieron en el sistema de archivos. A diferencia de otras áreas, existe solo en un grupo de bloques específico y la ubicación y el tamaño de esta área se almacenan en el superbloque. Después de estas áreas de metadatos, hay un área de bloques de datos. En esta área, hay bloques correspondientes a entradas de directorio donde se almacenan metadatos, como números y nombres de inodos de datos, y bloques correspondientes a datos reales



Asignación de archivos

El sistema de archivos Ext4 almacena archivos en bloques tan contiguos como sea posible para minimizar la fragmentación de archivos, por lo que, en teoría, un archivo no se fragmenta hasta 127 MB.

El sistema de archivos Ext4 almacena archivos secuencialmente comenzando desde el bloque frontal dentro del grupo de bloques. Para lograr eficiencia de E/S, los archivos dentro del mismo directorio se almacenan en el mismo grupo de bloques y todas las áreas no asignadas dentro del grupo de bloques no se utilizan para archivos adicionales. para ser almacenado. Por lo tanto, para almacenar datos, primero se usa el espacio frontal del grupo

de bloques y, cuando el área no asignada del grupo de bloques se reduce hasta cierto punto, el siguiente archivo guardado se almacena en otro grupo de bloques.

Fragmentación de archivos.

El sistema de archivos Ext4 evita la fragmentación de archivos tanto como sea posible para lograr eficiencia de E/S, pero hay dos razones principales por las que se produce la fragmentación. En primer lugar, si el tamaño del archivo creado es superior a 127 MB o no hay más espacio para almacenar en el bloque de datos administrado por el grupo de bloques, el archivo se almacena en fragmentos y, en este caso, se utilizan dos o más extensiones. En segundo lugar, cuando se modifica el contenido de un archivo existente y aumentan los datos internos, es posible que los datos agregados no se almacenen continuamente en los bloques de datos del archivo existente y se almacenen en fragmentos en otras áreas. El primer caso ocurre de manera similar en sistemas operativos Linux y Android en general que usan el sistema de archivos Ext4, pero el segundo caso ocurre principalmente en el sistema operativo Android y es muy frecuente.

Conclusiones:

La investigación en este trabajo ha demostrado que cada parte del sistema cumple una función específica y esencial en el ciclo de vida de Android, contribuyendo a la fluidez y estabilidad que caracterizan a este sistema operativo.

Queremos enfatizar el valor de aprender elementos importantes de esta tecnología móvil. Hemos investigado y aprendido, no solo sobre el funcionamiento de Android, sino que también comprendimos lo flexible y rápido que puede llegar a ser.

Este trabajo, nos sirve para en un futuro, poder hacer aplicaciones móviles comprendiendo más como se administran los recursos y los pasos que realizan las mismas, como también al investigar sobre software ya creado, podemos comprender por ejemplo, como las tareas en segundo plano, hacen que las aplicaciones consuman menos batería.

También vimos como maneja tecnologías de código abierto y licencias fáciles de integrar, como por ejemplo el kernel de Linux, Apache 2.0, etc.

Además, este informe proporciona una visión integral del sistema operativo Android, abarcando desde su origen y evolución hasta sus componentes técnicos fundamentales. La revisión de su historia y licencias revela la importancia de Android en la democratización de la tecnología móvil, mientras que el análisis del CPU, kernel, shell y procesos dan forma a su arquitectura perfecta para dispositivos móviles.

Destacamos que este sistema operativo evolucionó considerablemente desde sus primeras versiones, mejorando la seguridad, rendimiento y funcionalidad.

Es uno de los más utilizados en dispositivos móviles debido a su versatilidad, interfaz intuitiva. Su estructura de código abierto es fundamental para su comunidad de desarrolladores. Conocer sobre este sistema operativo puede ofrecer diferentes oportunidades en el desarrollo de aplicaciones, de software, entre otros.

Considero que uno de los puntos clave es la gestión de memoria, que se encarga de eliminar procesos que no se usan activamente y así liberar recursos.

En resumen, Android no solo se destaca por su flexibilidad y su comunidad de código abierto, sino también por la eficiencia de sus procesos internos, los cuales aseguran un desempeño confiable. La comprensión de estos conceptos puede ser invaluable para quienes deseen desarrollar aplicaciones móviles, ya que permite un aprovechamiento óptimo de los recursos del sistema, lo que a su vez mejora la experiencia del usuario final.

Bibliografía:

- SoftwareLab. (s.f.). ¿Qué es Android? Todo lo que necesita saber. Recuperado de <https://softwarelab.org/es/blog/que-es-android/>
- Wikipedia. (2023). Android (operating system). En *Wikipedia, la enciclopedia libre*. Recuperado de [https://en.wikipedia.org/wiki/Android_\(operating_system\)](https://en.wikipedia.org/wiki/Android_(operating_system))
- Abacus NT. (s.f.). *Gestión de memoria en Android*. Recuperado de <https://abacusnt.es/informatica/tag/gestion-de-memoria-en-android/>
- Wikipedia. (s.f.). *Android*. Recuperado de <https://es.wikipedia.org/wiki/Android>
- Android Open Source Project. (s.f.). *Android OS Documentation*. Recuperado de <https://androidos.readthedocs.io/>
- Google Developers. (s.f.). *Gestión de memoria en Android*. Recuperado de <https://developer.android.com/topic/performance/memory-management?hl=es>
- Android Open Source Project. (s.f.). *Licencias de Android*. Recuperado de <https://source.android.com/docs/setup/about/licenses?hl=es-419>
- Android Open Source Project. (s.f.). *Introducción a Android*. Recuperado de <https://source.android.com/docs/setup/about?hl=es-419>
- Malavida. (s.f.). *Arquitecturas de CPU en Android*. Recuperado de <https://www.malavida.com/es/articulos/arquitecturas-de-cpu-en-android>
- Android Open Source Project. (s.f.). *Kernel de Android*. Recuperado de <https://source.android.com/docs/core/architecture/kernel?hl=es-419>
- Android Open Source Project. (s.f.). *Kernel Android común*. Recuperado de <https://source.android.com/docs/core/architecture/kernel/android-common?hl=es-419>
- Sozpic. (s.f.). *Gestión de memoria en Android*. Recuperado de <https://www.sozpic.com/gestion-de-memoria-en-android/>
- Android Open Source Project. (2022, junio 6). *Arquitectura Android*. Recuperado de <https://source.android.com/devices/architecture?hl=es-419>
- TestingBaires. (2024, junio 8). *Arquitectura de las aplicaciones móviles*. Recuperado de <https://testingbaires.com/arquitectura-de-las-aplicaciones-moviles/>
- ADSLZone. (2024, julio 18). *Qué es Android: todo sobre el sistema operativo de Google*. Recuperado de <https://www.adslzone.net/reportajes/software/que-es-android/#515398-principales-componentes-del-sistema-operativo-android>
- Develou. (s.f.). *Aprendiendo sobre la arquitectura de Android*. Recuperado de <https://www.develou.com/aprendiendo-la-arquitectura-de-android/>

- Henao, C. (2022, enero 15). *¿Por qué Android? Características, versiones y arquitectura* [Video]. YouTube. <https://www.youtube.com/watch?v=AbaekXNKpJ4>
- Android Open Source Project. (2024, septiembre 23). *Preguntas sobre el código abierto*. Recuperado de <https://source.android.com/docs/setup/about/faqs?hl=es-419>

Gestión de procesos y memoria en Android.

- <https://tab1-so-g13.blogspot.com/2017/10/gestion-de-procesos-y-memoria-en-android.html>
- Androlib.(2023, 5 de abril). Explicación del sistema de archivos de Android. <https://www.androlib.com/android-file-system-explained/>
- IBM. (2021, 23 DE MARZO). FUNCIONES DE SHELL
- <https://www.ibm.com/docs/es/aix/7.1?topic=concepts-shell-features>
Mundotecnologico.click.(S.f). ¿Qué es la aplicación Shell en Android?.
- https://mundotecnologico.click/que-es-la-aplicacion-shell-en-android/#%C2%BFQu%C3%A9_le_permite_hacer_un_shell_a_un_usuario?
- Process in Android. <https://www.wideskills.com/android/intrprocess-communication/processes-in-android>
- (2020, Septiembre 18). Operating System - Process Concepts: 03 <https://medium.com/@effectivereader/operating-system-process-concepts-03-45572edd1aec>
- Fabio Massimo Zanzotto. https://www.researchgate.net/figure/Organization-of-the-memory-for-a-process_fig2_226822965
- Abanoub Asaad. (2022, Mayo 29). Operating Systems Concepts: Processes. <https://dev.to/abanoub7asaad/operating-systems-concepts-ch-3-part-1-processes-1lcj>
- What's the purpose of swap and zRAM on Android? https://mirfatif.github.io/IAnswers/swap_zram_android
- Mika Hashimoto. (2019, Abril 2). Understand behavior of Android at the time of lack of memory. <https://medium.com/@simon.russiazushi/understand-behavior-of-android-at-the-time-of-lack-of-memory-a3a0f00dc4f7>
- (2021, Mayo 11). Linux OOM (Out Of Memory) Killer: Qué es y qué importancia tiene. <https://bytelearning.blogspot.com/2021/05/linux-omm-killer.html>
- Kyle Shanks. (2021, Junio 5). Android - Low Memory Killer- https://hackmd.io/@AlienHackMd/Hy103ZPc_
- Jisun Kim. (2020, Mayo). Triggering conditions of LMK, kswapd, and OOM in Android and Linux. https://www.researchgate.net/figure/Triggering-conditions-of-LMK-kswapd-and-OOM-in-Android-and-Linux_fig1_341159546
- F2FS vs EXT4: la guerra de los sistemas de archivos en Android ha comenzado. <https://developer.android.com/topic/performance/memory-overview?hl=es-419>

- Yúbal Fernández. (2019, agosto 9). VSync: qué es y cuáles son sus ventajas y desventajas.<https://www.xataka.com/basics/vsync-que-cuales-sus-ventajas-desventajas>
- (2023, noviembre 28). Android Performance Optimization: Best Practices and Tools. <https://www.wetest.net/blog/android-performance-optimization-best-practices-and-tools-901.html>
- Khurshidbek Bakhromjonov. (2023, Julio 4). Java Memory Management: Understanding the JVM, Heap, Method Area, Stack. <https://medium.com/@khurshidbek-bakhromjonov/java-memory-management-understanding-the-jvm-heap-method-area-stack-24a4d4fa2363>
- <https://www.welivesecurity.com/la-es/2015/08/07/tecnicas-sencillas-para-el-analisis-de-memoria-en-android/>

Historia

- Laureano Garcia Crespo. <https://lau-re.wixsite.com/laure/post/android-historia-del-sistema-operativo-de-google>
- Wikipedia (Última Actualización en 2023)https://es.wikipedia.org/wiki/Open_Handset_Alliance
- Michael Jennings (US), Conferencia (15 mayo 2008)<https://nextconf.eu/2010/12/driving-innovation-on-new-platforms-android-and-the-open-handset-alliance/>
- <https://www.temok.com/blog/android-version-list/>
- https://betawiki.net/wiki/Android_11
- https://betawiki.net/wiki/Android_12
- https://betawiki.net/wiki/Android_13
- https://betawiki.net/wiki/Android_14
- Android on X86 An Introduction to Optimizing for Intel Architecture(Hoja 1 a 8)

Licencias

- Castellet, A., Aguado, J. M., & Martínez, I. (2013). Los nuevos actores que cambian las reglas y el juego: cómo Apple y Google han transformado la distribución de contenidos. *Breaking the Media Value Chain*, 333.
- de-Matteis, L. J., Stankevicius, A. G., & Capobianco, M. Sobre el impacto y la preponderancia de las licencias de software en el mercado de los dispositivos móviles.
- Romera, T. (2005). Licencias Libres Software y otros contenidos.

CPU

- Abdalqader Najjar. (2022, Noviembre 26). **¿Qué es la arquitectura de CPU móvil?** <https://medium.com/@abdalqader27.najjar/what-is-mobile-cpu-architecture-7a2d0b719c0a#:~:text=As%20of%20now%2C%20there%20are,ARM64%3A%20AArch64%20or%20arm64>
- JuaniX. (2022, 14 julio). Como hacer funcionar el primer microprocesador x 86 de la historia. <https://www.youtube.com/watch?app=desktop&v=OhykVf0CvRE>
- New Mind. (2018, Noviembre 29). The Evolution of CPU Processing Power PArt1: The mechanics Of a CPU. <https://www.youtube.com/watch?v=sK-49uz3lGg>
- New Mind. (2018, Diciembre 18). The Evolution of CPU Processing Power PArt1: The mechanics Of a CPU https://www.youtube.com/watch?v=kvDBJC_akyg&t=8s
- How are the basic Delphi types related to each other?
<https://stackoverflow.com/questions/997795/how-are-the-basic-delphi-types-related-to-each-other>
- Lithmee. (2018, Mayo 3). Difference Between 32 bits And 64 Bits.
<https://pediaa.com/difference-between-32-bit-and-64-bit/>
- Aitor Eizagirre. (2023, Abril 20). Cómo saber qué tipo de procesador tiene tu Android (ARM, ARM64 o X86). <https://elandroidefeliz.com/identificar-tipo-de-procesador-en-android/>
- Ramírez Martínez, F. J. (2015). Estudio comparativo tecnológico de terminales móviles Android.
- Molera Codina, J. M. (2013). Arquitectura de diseño de accesorios electrónicos inalámbricos para dispositivos móviles inteligentes.
- <https://www.ninjaone.com/es/blog/con-o-sin-gms/>

Kernel

- Zambrano, G. R. (2016). Análisis de Comparación de Android y GNU/Linux/Comparison Analysis Android and GNU/Linux. International Journal of Innovation and Applied Studies, 18(4), 1039.
- Domínguez López, L. (2019). Creación y personalización de ROM de Android orientado a uso en comercios y empresas.

Sistema Operativo

- Salazar López Jesús Manuel. (2020, Octubre 21). Tesina Procesadores ARM el futuro del desarrollo social y laboral. <http://repositorio.upsin.edu.mx/formatos/TesinaSalazarLopezJesus711410207.pdf>. (P. 26-30).

- R. Prasanna Kumar. (2015, Noviembre)<https://ijsetr.com/uploads/613452IJSETR7743-1768.pdf> (P.1-3).
- Jayric Manning. (2022, Noviembre 14). RISC vs RISC-V vs ARM: What is the Difference?. <https://www.makeuseof.com/risc-vs-arm-what-is-the-difference/>
- Rodrigo Alonso. (2024, Mayo 16). Todo lo que necesitas saber sobre los procesadores ARM. <https://hardzone.es/tutoriales/componentes/procesador-arm/>
- Juan Francisco Villa Medina(2020, Septiembre 14). Arquitecturas de Computadoras: Von Neumann y Harvard, RISC, CISC. <https://www.youtube.com/watch?v=nC4MHIIMVxU>
- Christopher Harper. (2023 Septiembre 12). RISC-V vs ARM: AnIntroduction and Which Is “Better”??. <https://www.cgdirector.com/risc-v-vs-arm-introduction/>
- ARM Microcontroller. <https://piembystech.com/arm-microcontroller/>
- RISC-V Simulators. <https://teamup.org/explore/>
- Shawn. (2022, Diciembre 31). <https://shawngraham.io/?p=1147>
- Zhauniarovich, Yury. (2014, Junio 01). Android Boot Sequence. https://www.researchgate.net/figure/Android-boot-sequence_fig4_292906634
- Harvard Architecture. <https://www.anyrgb.com/en-clipart-hw4jt>
- Shankara Narayana. Most microprocessor/microcontroller uses either the Harvard architecture or von Neumann. <https://www.quora.com/Most-microprocessor-microcontroller-uses-either-the-Harvard-architecture-or-Von-Neumann-computer-architecture-for-memory-What-is-the-main-difference-between-these-architectures-Which-memory-architecture-does-the>
- (2021, Octubre 18). Una introducción a la arquitectura ARM con el principio de funcionamiento de cada módulo.<https://es.fmuser.net/content/?10492.html>
- Learn Computer Science. (2022, Abril 13). What is Instruction Set Architecture? <https://www.youtube.com/watch?v=6fgbLOL7bis>
- Ohmazing Physics.(2023, Junio 5). Understanding Von Neumann and Harvard Architecture | A Side-by-Side Comparison <https://www.youtube.com/watch?v=ua7xQwwGPO4>
- Vikash Choudhary. (2023, Julio 4). Android Boot Sequence. <https://medium.com/@vikseln/android-boot-sequence-fccb96bd17c4>
- (2024, Abril 19). Proceso de arranque / Carga de Android.<https://www.palentino.es/blog/proceso-de-arranque-carga-de-android/>
- Soviet Sistemas. (2017, Junio 16). Qué es SoC (System On a Chip) y sus características. <https://www.solvetic.com/page/recopilaciones/s/profesionales/que-es-soc-system-on-a-chip-caracteristicas>
- Deeptabho Misra. (2019, Enero 26). Understanding Smartphone: The SoC (System on a Chip). <https://www.droidviews.com/understanding-smartphone-specs-soc-system-chip/>
- Eze Wholesale. (2021, Julio 1). SoC 101: Smartphone Chipsets Explained <https://blog.ezewholesale.com/soc-101-smartphone-chipsets-explained-76150d4ed748>

- Umer Farooq. (2018, Julio). DOI: 10.13140/RG.2.2.20829.72169. Android Operating System Architecture. https://www.researchgate.net/profile/Umer-Farooq-22/publication/326507076_Android_Operating_System_Architecture/links/5b5186750f7e9b240ff0b053/Android-Operating-System-Architecture.pdf
- Alison Chaiken. (2018. Enero 16). Analyzing the Linux boot process. <https://opensource.com/article/18/1/analyzing-linux-boot-process>
- (2017, febrero 25). Zygote란? #1 Zygote Class 실행까지 출처 <https://alnova2.tistory.com/1104>
- P Srinivas Rao. (2022, Abril 5). Android Booting process from Power On. <https://www.linkedin.com/pulse/android-booting-process-from-power-p-srinivas-rao/>
- (2019, Marzo 1). The System on a Chip (SOC) Inside your Phone. <https://www.mytcr.com/blog/inside-phone-system-chip-soc/>
- Vikash Choudhary. (2023, Julio 4). Android Boot Sequence. <https://medium.com/@vikseln/android-boot-sequence-fccb96bd17c4>
- Max Maxfield. (2014, Junio 23). ASIC, ASSP, SoC, FPGA – What's the Difference?. <https://www.eetimes.com/asic-assp-soc-fpga-whats-the-difference/>
- Android Security and Forensics. (2022, Marzo 20). Android Boot Process Practical. <https://www.youtube.com/watch?v=kQc-CT8OOqY>
- Radhika Karandikar. (2018, Abril 30). Android Application Launch explained: from Zygote to your Activity.onCreate() <https://medium.com/android-news/android-application-launch-explained-from-zygote-to-your-activity-oncreate-8a8f036864b>
- Dave Smith. (2015. Mayo 21). Digging Into Android Startup. <https://www.youtube.com/watch?v=5SQP0qfUDjI>
- Robert Feranec. (2013, Octubre 24). How does ARM boot? <https://fedevel.com/blog/how-does-arm-boot>
- Michael Opdenacker. (2023, Febrero 24). Boot time: choose your kernel loading address carefully. <https://bootlin.com/blog/tag/boot-time/>
- (2020, Agosto 12). How the ARM32 Linux kernel decompresses. <https://people.kernel.org/linusw/how-the-arm32-linux-kernel-decompresses>
- Begum Avci Kocaman. (2023, Octubre 31). JobScheduler en Android. <https://medium.com/huawei-developers/jobscheduler-in-android-3b269e3a4e49>
- (2023, Enero 18). Long Term Scheduler in Operating System. <https://www.geeksforgeeks.org/long-term-scheduler-in-operating-system/>
- (2024, Septiembre 02). Difference between dispatcher and scheduler. <https://www.geeksforgeeks.org/difference-between-dispatcher-and-scheduler/>

Jerarquía de Procesos en Android

- <https://developer.android.com/guide/components/processes-and-threads?hl=es-419>
- <https://stuff.mit.edu/afs/sipb/project/android/docs/guide/components/processes-and-threads.html>

- <https://developer.android.com/guide/components/activities/process-lifecycle?hl=es-419>

File System

- Kale, A. (2021, December 31). *File system of Android*. Medium. <https://medium.com/@aditi.kale20/file-system-of-android-a89dcbb693f1>
- KeepCoding. (n.d.). *Sistema de ficheros Android*. KeepCoding Blog. Retrieved October 3, 2024, from <https://keepcoding.io/blog/sistema-de-ficheros-android/>
- Sumbul Zahra. (2024, Septiembre). https://www.linkedin.com/posts/sumbul-zahra-46a7582b2_linux-cybersecurity-ext2-activity-7236658098969481216-CZD2/
- Agustín García. (2020, Junio 11). F2FS vs EXT4: la guerra de los sistemas de archivos en Android ha comenzado. <https://www.lavanguardia.com/andro4all/sony/f2fs-vs-ext4>
- Mihai Matei. (2019, Agosto 12). Galaxy Note 10 uses F2FS, not EXT4 file system: What's the difference? <https://www.sammobile.com/news/galaxy-note-10-uses-f2fs-not-ext4-file-system-whats-the-difference/>
- Mulliner, C., & Miller, C. (2013). *Android security internals: An in-depth guide to Android's security architecture*. No Starch Press. https://nostarch.com/download/Android_SecurityInternals_ch1.pdf
- Chiara Quiocho.(9/7/2024). ¿Qué es un Daemon? [https://www.ninjaone.com/es/it-hub/it-service-management/que-es-un-daemon/#:~:text=Un%20daemon%20un%20programa%20que,un%20sistema%20operativo%20\(SO\).](https://www.ninjaone.com/es/it-hub/it-service-management/que-es-un-daemon/#:~:text=Un%20daemon%20un%20programa%20que,un%20sistema%20operativo%20(SO).)
- Wang, Z., Murmuria, R., & Stavrou, A. (2012, July). Implementing and optimizing an encryption filesystem on android. In *2012 IEEE 13th International Conference on Mobile Data Management* (pp. 52-62). IEEE.
- Viejo, Daniel (2014) **Qué son los sistemas de archivos en Android - F2FS vs EXT2.** <https://www.nextpit.es/que-son-sistemas-de-archivos-android>

Redes

- (2016) Universidad de Alicante. <https://mastermóviles.gitbook.io/tecnologías2>
- <https://www.ionos.com/es-us/digitalguide/servidores/know-how/tcpip/>
- Franklin Matango (18 agosto,2016). Conceptos básicos protocolo TCP/IP. <http://www.servervoip.com/blog/conceptos-basicos-protocolo-tcpip/>
- <https://developer.android.com/develop/connectivity?hl=es-419>
- <https://developer.android.com/studio/run/emulator-networking?hl=es-419>
- <https://developer.android.com/privacy-and-security/security-ssl?hl=es-419>

- Cómo administrar el uso de la red .
<https://developer.android.com/develop/connectivity/network-ops/managing?hl=es-419#java>
- VPN Unlimited (2024) Pila de protocolos. Pila de Protocolos: Mejorando la Comunicación en Redes
<https://www.vpnunlimited.com/es/help/cybersecurity/protocol-stack>
- Developers (2023). Cómo administrar el uso de la red.
<https://developer.android.com/develop/connectivity/network-ops/managing?hl=es-419>
- Jaramillo, O. B., & Hanrryr, S. E. (2015). Seguridad en Dispositivos Móviles Android. *Universidad Nacional Abierta ya Distancia-UNAD*.
- Elenkov, N. (2014). *Android Security Internals: An in-depth guide to Android's security architecture* (Sección 6: Network Security and PKI). No Starch Press.