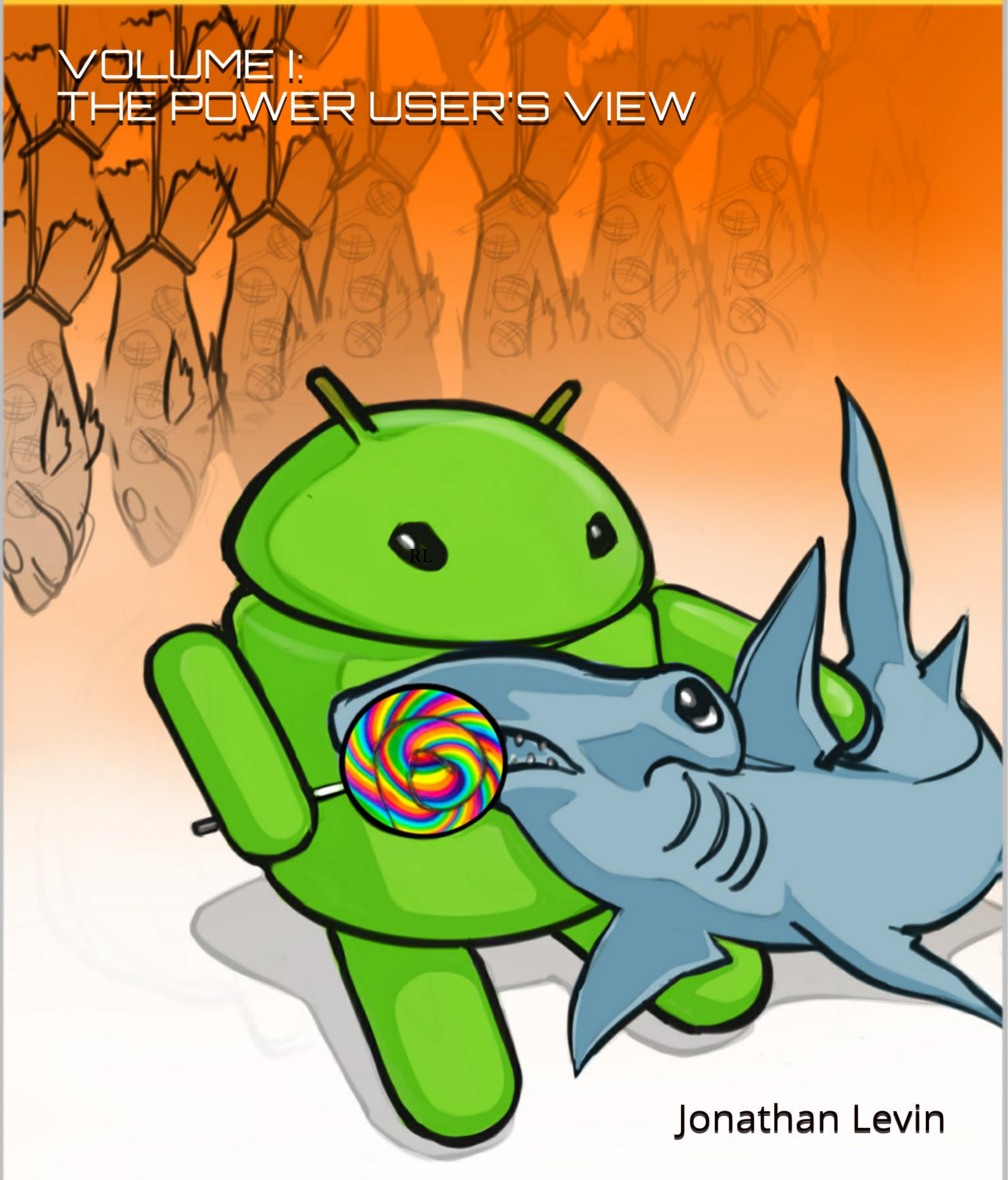
Android Internals:: A Confectioner's Cookbook

VOLUME I: THE POWER USER'S VIEW



Jonathan Levin

Android Internals

A Confectioner's Cookbook

Volume I: The Power User's View

With updates for Android M, Preview Release 1 (6/2015)

By Jonathan Levin

Cambridge, MA

***In memoriam**: **Frank R. Dye**. I missed you by a day, and will miss you for a lifetime.*

Android Internals - A Confectioner's Cookbook - Volume I - The Power User's View

Printed in Cambridge, MA, USA

First Printing (v1.15-Mp1-as)
ISBN-10: 9910555-2-4
ISBN-13: 978-0-9910555-2-4

Publisher: Technologeeks.com
Production Editor: Jonathan Levin
Interior Design: Jonathan Levin
Technical Reviewers: Moshe Kravchik, Arie Haenel
Cover Illustration: Dino Tsiopanos

For information on distribution, translations, or bulk sales, please contact Jonathan Levin, at:

Jonathan Levin
Phone: (617)-3000-667
Email: j@NewAndroidBook.com
Web: www.NewAndroidBook.com

Company and product names mentioned in this work may be trademarked by their respective owners. Any and every such name is used solely in an editorial fashion, to the benefit of the trademark owner, with no intention whatsoever of infringement.
The Android robot is reproduced and/or modified from work created and shared by Google, according to the terms of the Creative Commons 3.0 Attribution License.

Limit of Liability/Disclaimer of Warranty: The author neither makes or implies any representations or warranties with respect to the accuracy or completeness of the contents of this work. Further, this paragraph explictly and specifically serves to disclaim all warranties, including without limitation warranties of fitness for a particular purpose. No warranty may be created or extended by sales or promotional materials. The advice and strategies contained herein may not be suitable for every situation. This work is sold with the understanding that the author is not engaged in rendering legal, accounting, or other professional services. If professional assistance is required, the services of a competent professional person should be sought. The author shall not be liable for damages arising herefrom. The fact that an organization or Web site is referred to in this work as a citation and/or a potential source of further information does not mean that the author endorses the information the organization or Web site may provide or recommendations it may make. Further, readers should be aware that Internet Web sites listed in this work may have changed or disappeared between when this work was written and when it is read.

Information in this book is distributed on an "As Is" basis, with no warranty. Android is a shifting landscape, with versions being released faster than any work can keep up with. Every effort and precaution has been taken to update this work to reflect up to and Including Android Lollipop (5.0.2). Bear in mind, however, that different versions of Android may add, modify, or remove APIs and features with no notice. The author hereby disclaims any liability to any and every person or entity with respect to any loss or damage, whether caused or alleged to be caused directly or indirectly, by this work.

To report errors or omissions, please contact the author.

About the [Author/Editor/Formatter/Publisher]

I started in my teens by hacking (mostly in the good sense of the word) and trying to figure out how things worked back in 1993, from an XT with a 2400 baud modem onto a shell I'm not even sure was mine. With no help per se, I had to `man man` and then `man *`..

A lot has happened in the two decades since, and from UNIX to Linux, I got to Windows and OS X. Over the years, I took up consulting and training - initially, in security. I then realized security is largely a projection of internals. Finally, I gathered a few good men and started Technologeeks.com - which is where I presently pass my time and function as CTO.

Authoring is somewhat new: I first took on Apple's OSes with "Mac OS X and iOS Internals" *(Wiley, 2012)*, which was well received. It was a painful process, but the dopamine rush was awesome. Now an addict, I took on Android, and the first part of the result is what you see before you. This was also my first foray into the world of self publishing. With this done, I'm turning to the 2nd Edition of MOXiI, but you can expect Volume II of this series ("The Developer's View") soon.

About the Technical Reviewers

Moshe Kravchik of Cisco helped review my previous book, and immediately volunteered to help with this one. With his eye for detail and accuracy, he helped me lay out the contents of this book, expanded them with many a question a reader would pose, and caught my (oh-so-many) typos, or areas wherein I wasn't clear enough. For that, my thanks!

Arie Haenel, also of Cisco, is hands-down the most gifted reverse engineer I've ever known, and certainly one of the outright smartest people, period. A reviewer of my previous book, it was only natural I would seek his help for this one. He's already got the dubious pleasure of reviewing the 2nd Edition of MOXiI, as well.

Table Of Contents

About This Book

Overview

If you got this book, no doubt you recognize the importance of Android. From a start-up started back in 2003, it has been assimilated by Google, and morphed into one of its largest arms. Taking on Apple's iOS head on (some would say, too closely), it has not only achieved hegemony over mobile operating systems worldwide (with a staggering 82% of the market as this book goes to print) but has also permeated other platforms, becoming an operating system for wearable devices, TVs, and embedded devices.

Android is open source and freely available, meaning anyone can get it, and adopt it to any platform - indeed, it owes its overwhelming popularity to this. It's surprising, however, that some seven years since its public inception, no book to date has taken on the task of documenting and explicating its internals. A previous work on the subject - Embedded Android: Porting, Extending, and Customizing, by Karim Yaghmour - provides a good deal of detail about the general structure of the OS, but focuses on building and adapting the sources to new platforms, and stops shy of describing the structure of the operating system itself. In fact, in his "Internals Primer", Yaghmour states that "Fully understanding the internals of Android's system services is like trying to swallow a whale".

The analogy is very much an understatement, Which is why this work requires not one, but multiple volumes. The first (the one you are reading), focuses on Android from the perspective of the power user or administrator. In it, I try to tackle various aspects of the operating system - its design, filesystem structure, boot sequence, and native services, along with the Linux foundations and legacies which affect the operation. All this, without going into code, and trying to provide an illustrated, conceptual view as possible. This book can be considered, in a sense, a successor to Yaghmour's work, which remains a great resource and a recommended read.

The second volume of this work (to be published soon) dives far deeper, and looks at the structure of Android's frameworks - which is where its appeal to developers lies: Through a rich set of Java-level frameworks, developers obtain powerful abstractions of input devices, sensors, graphics and what not. All these abstractions, come at the price - the complexity that lies "under the hood" - which most developers are quite blissfully ignorant of (and would likely prefer to stay this way). There is no knowledge that is not power, however, and so deep familiarity with the frameworks is instrumental for anyone dealing with the low level implementations, and customizations for performance, hardware or security.

Android is a constantly shifting landscape. This work was started halfway through KitKat, and was postponed several times as Android mutated further to become Lollipop (L). This constant evolution is still ongoing, with Android M announced even as L is battling quite a few bugs. Nonetheless, it was about time to publish the book once L showed signs of stabilization - and so I can proudly say this book is updated till the latest and greatest.. at least at the time of publication. Fortunately, thanks to the wonders of self-publishing, I can keep the book along with the winding curve, and the version you're reading has been updated for M Preview Release 1 (June 2015).

I tried to learn from "mistakes" of my previous book, "Mac OS X and iOS Internals" (MOXiI). One of the main criticisms I had was that said work was far too technical, and riddled with source code, which made it hard for the non-developer type to get by. My own personal belief is of "Read the Source, Luke", in that source code - unlike natural language - contains (almost) no ambiguities, and is thus the right way of depicting facts. Nonetheless, in this book I've tried to provide as many illustrations as possible, all without sacrificing detail. (I'm also adopting this book to the 2nd edition of MOXiI, out later this year - not so much out of choice, but because the book dives into far deeper and shadier aspects of both systems, for which there is no open source).

The book is also a lot more "hands-on", taking some of the hands-on exercises from our Android training and recasting them in the form of Experiments. These are invaluable if you want to get a good sense of the topics in the relevant section. Android is a UN*X derivative (by virtue of Linux), and the only way one learns UN*X is through the fingers, and neither eyes nor ears. The experiments demonstrate many useful commands from the Android command-line-interface (CLI), and also techniques for looking deeper into the operating system. Furthermore, the experiments will likely produce different outputs on different strains of Android - which makes them worthwhile to try on your own device(s), so as to get different perspectives or implementations which may vary by vendor or OS version.

Contents, at a glance

The book is designed to be read either cover-to-cover or as random, quick access. Each chapter is largely self contained, and hyperlinks on topics allow quick associative navigation when reading the book in e-Form. For print edition, relevant chapter numbers (for internal links) or URLs (for external links) are provided. I have also added the paths to the AOSP files referenced, where relevant - albeit in abbreviated form, so as to conserve space in the tables as the paths can be a mile long.

Chapter One provides an introduction to the operating system: Examining the evolution of the OS over its versions (since Froyo, which is the only obsolete version which you might still encounter in the wild, and up to L. It also explains the architecture (at a high level view), and the Linux underpinnings, by traversing each layer of the Android stack. It then looks at Android derivatives, both Google's and other vendors (e.g. Amazon's FireOS), before concluding with some thoughts and ideas for future directions.

Chapter Two dives right into the technical bits - specifically, Android Partitions and filesystems. Starting with an examination of the partitioning scheme used by Android (which, unfortunately, is far from standardized across vendors), and filesystems - Ext4 and F2FS. What follows is a tour of filesystem contents, which should prove useful if you ever need to figure out what a specific system directory or file contains. A few of the built-in apps data directories are also covered, which is handy if you're doing forensics. The chapter also discusses the Android "Protected Filesystems" (OBB and ASEC), though those fail miserably on rooted devices. Finally, the role of the Linux pseudo-filesystems - cgroupfs, debugfs, procfs, sysfs and others - is described.

Chapter Three builds on its predecessor - which covered partitions - to explain the role of partitions in the Android boot process. Starting with a discussion of the Android boot images (what some refer to, albeit incorrectly, as ROMs), and how to flash them onto the device's boot partitions. The default Android boot loader is explained (with the more heavily technical aspects left as a bonus article on the book's companion website), and the other components of the boot image - the kernel, device tree, and initramfs - are described in detail. Experiments demonstrate how to unpack, customize and repack these components (assuming an unlocked bootloader). The chapter also discusses the images sent as over the air (OTA) updates, as well as the processes of backup/restore, and shutdown.

Chapter Four is dedicated in its entirety to one process - /init. This, like its UN*X namesake, is responsible for starting up the system in user mode. The process of startup is explained in detail, through examination of the /init.rc file syntax. Other roles of /init, such as maintaining system properties and watching for hardware changes (as ueventd) are detailed as well.

Chapter Five discusses the native services - i.e. those listed in the /init.rc and loaded as Linux binaries (in contrast to the Dalvik-level framework services, which are loaded as threads in system_server and covered in Volume II). This chapter provides a detailed reference of each and every daemon you're likely to find on your device - and there are quite a few of them.

Chapter Six provides a gentle introduction to Android's framework service architecture, by explaining the roles of the servicemanager and system_server processes, which together form the fulcrum on top of which all of Android's frameworks rest. Binder, the elephant in the chapter, is described but briefly, leaving most of the meticulous detail for Volume II, but hopefully explaining just enough to provide more insight as to how Android Inter Process Communication and Remote Procedure Calls work.

Chapter Seven is a view of Android through a Linux lens - that is, looking at Android system processes and apps through the /proc filesystem and Linux-level tools. This chapter is a "two-fer" in the sense that you can apply most (if not all) of the techniques shown there on your Linux system for native-level debugging.

Chapter Eight - the last chapter of this volume - concerns itself with Security. This chapter was made available as a preview (originally, as Chapter 21, back when I naively thought I could fit everything into one book!). It provides a detailed walk through of security features, both at the Linux level and that of the frameworks, as well as a special section on rooting Android devices - both in the "approved" ways, as well as some unexpected ones.

Conventions used in this book

Keeping this simple:

- filenames are specified like this
- commands(1), systemCalls(2) or framework classes are specified thus. The numbers in parentheses refer to the manual section describing them, using the Linux man

Additionally, this book is full of Figures, Listings and Outputs. Figures are illustrations of components or message flows. Listings are generally static files, as opposed to Outputs which are sequences of commands (often included as part of an experiment). In outputs, the idea was to show the flow as well as usage of the commands, so the outputs are fully annotated, e.g.

Output 0-1: A sample output

```
# Comment, explaining what's being done
user@hostname   (directory)  User input
Output...
Output..  # Annotation, explaining output
Output..
```

Attention has been paid to detail - the username (as well as the prompt sign, $ or #) will tell you if the command requires shell or root privileges. The hostname shows you the device the command was tried on, with "generic" being the emulator, "flounder" being the nexus 9 (L), and otherwise the device name (s3, s4, kindle, nexus5, etç) and "Forge" being the Linux host. I tried to avoid snippets of code (at least in Volume I), and in those places where it proved vital, I have also provided annotations. The color scheme was (finally) adjusted to be easy on the eyes in both color (if you're reading the PDF) or black & white (for the print edition).

Finally....

The book proved to be a massive undertaking. Sifting through Android's sources is akin to inspecting an organism down to the cellular level. For those who still wish to examine the sources themselves, I have pinpointed the relevant files pertaining to each discussion, and hyperlinked them (or put them as a table, for the print edition of the book). The interested reader should most definitely obtain the sources of the latest version, either by using `git` and `repo`, as explained in http://source.android.com/, or by looking through Google's Android Source Website.

The book has also been a one-man-project: With the exception of the cover art (provided for me by the gifted Dino Tsiopanos, a great engineer who's an even greater illustrator!), everything in this work - text, images, formatting, editing - has been done by myself. Thankfully, I enlisted the help of my regular reviewers - Moshe Kravchik and Arie Haenel, to whom I am both indebted. Nikolay Elenkov, who wrote the excellent "Android Security Internals: An In-Depth Guide to Android's Security Architecture" contributed very helpful insights and feedback as well. Aviv Greenberg - who at the last moment performed a binge reading and review - helped me with critical comments. Almost last, but hardly least - Eddie Cornejo - who not only caught even more typos which somehow eluded other eyes - but also made sure I was as unbiased as I should, given the "Other" OS. And - finally, I thank Nikola Veljkovic - who meticulously caught further typos everyone else missed.

Once more, I owe thanks to Yoav Chernitz. In a way, all my books will - because it was his inception which started my career as an author. Maybe because to me, this goes without saying, it actually did - The previous edition (before the Android M update) did not include this much needed thanks. But in this case, Yobo deserves extra special mention for having pushed me into Android. It was he who provided Technologeeks with requirements for "*Linux to Android*" and later "*Android Internals*" - Two courses which became best sellers, and upon which this work's two volumes are based.

A special, personal and most intimate thanks and gratitude goes to Amy, the Yin to my Yang, who provided infinite support and encouragement - as with my previous book (and with everything else). This is one thanks I will forever reiterate and never forget!

The book was painstakingly authored with `vim`, hand-typing standards-compliant HTML5 (yes, I'm serious, and no, I probably wouldn't try this again). Illustrations are either SVG (another traumatic ordeal), or drawn with PowerPoint. This should hopefully help explain why this book so long to take out, though the good news is that Volume II (which is double the size of this work!) should be available very soon. Pagination is a non-standard A4, meaning less pages than the usual tech-book, but far more detail per page. Crafting an index for the book would have proved so Sisyphean a task, I decided to not even go there (You can just search the PDF). Please keep all this in mind if you spot any styling errors or (gasp) technical ones - Errare est humanum. For technical errors only I offer rewards - a la Knuth - 0x100 cents will come your way if you report any (I'm hoping for rampant inflation when QE subsides 😉).

I maintain a companion web site with bonus material and quite a few custom tools - at http://NewAndroidBook.com/. Updates to this book, as well as typos and/or errata (which I probably have), will be published through that site.

For those of you more into Tweeting, my company - @Technologeeks often tweets about updates and bonus material to both my works. Technologeeks also provides expert consulting and training services - on Android, OS X, iOS, Linux and more - so I encourage you to check out http://Technologeeks.com/! The training on both Android and OSX/iOS, specifically, is based on my books. The company also heads the "Android Kernel Developers" group on LinkedIn - if you feel like dropping by and saying hi (or asking questions).

I do hope you find the book both interesting and entertaining (well, as entertaining as a technical book can get, I guess). I'm always available for comments/feedback through the companion web site, through a dedicated forum I have set up there.

.....Now let's get to it!

I: The Evolution of Android's Architecture

Though Android is built on Linux and relies heavily on much of its infrastructure - most notably the kernel - Android has become an operating system in a class by itself. Unlike OS X and iOS, which share the majority of their code base (with the exception of the UI and several frameworks), Android introduces a vast collection of frameworks, as well as a runtime to support them (Dalvik). Indeed, most of the user-facing features and enhancements in between versions have to do with additional frameworks and APIs being added, with only a relatively small portion of them at the system level.

This Chapter explores the evolution of Android, and examines its architecture. Beginning with the Android version history, from Cupcake (1.5) to Lollipop (5.1.1), and beyond, we cover system-related features and enhancements in each. We then turn to examine the Android architecture, comparing and contrasting with that of Linux. Each layer is described in detail, laying the foundations for the even deeper exploration carried out in the next chapters (and next volume) of this work. Finally, we consider the multitude of Android derivatives, as well as future enhancements which may be expected in the next versions of this rapidly evolving OS.

As this book went to print, Android 5.0 (Lollipop) was made available for select Google Nexi, and is scheduled for rollout by vendor. Android is moving so fast, in fact, chances are that no matter when you read this, a new version of Android will only be months away, as the mobile OS arms race ensues. This book has now been updated to reflect changes ithrough M (preview release 1), but Android's rampant advance continues onwards. It's therefore a good idea to check the companion web site (NewAndroidBook.com) for updates.

Android version history

Over its seven short years, Android has already undergone no less than a dozen versions. When one considers the API versions (which map the internal set of APIs to the catchier condiment type), this number increases to 22. Enumerating the many framework features introduced in each version would be tedious and likely miss out on a few, so this section instead aims to provide a more technical look, focusing on those API differences at the system (rather than framework) level, as well as other noteworthy observations. Those seeking more information about changes are suggested to read the comprehensive Wikipedia Entry[1], or check the Android documentation for the respective versions.

Table 1-1 shows the Android version history, and maps the official release version to that of the API and the kernel. Note that the kernel versions don't necessarily match in all devices, as some vendors compile their own kernel, or backport newer kernels.

Table 1-1: Android Versions, to date

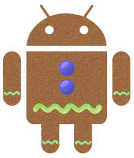
Date	Code Name	Release	API	Kernel
Late 2015	M (final name unknown)	5.2 (Likely)	22MRC	3.4(armv7)/3.10(arm64)
3/2015	Lollipop	5.1-5.1.1	22	
11/2014		5.0-5.0.2	21	
10/2013	KitKat	4.4-4.4.4	19 (20)	3.4
07/2013	JellyBean (MR2)	4.3	18	
11/2012	JellyBean (MR1)	4.2-4.2.2	17	
07/2012	JellyBean	4.1-4.1.1	16	3.0.31
12/2011	Ice Cream Sandwich (MR1)	4.0.3-4.0.4	15	
10/2011	Ice Cream Sandwich	4.0-4.0.2	14	3.0.1
07/2011	Honeycomb (MR2)	3.2-3.2.6	13	2.6.36
05/2011	Honeycomb (MR1)	3.1	12	
02/2011	Honeycomb	3.0	11	
02/2011	Gingerbread (MR1)	2.3.3-2.3.7	10	2.6.35
12/2010	Gingerbread	2.3-2.3.2	9	
05/2010	Froyo	2.2-2.2.3	8	2.6.32
10/2009	Éclair	2.0-2.01, 2.1	5-7	2.6.29
09/2009	Donut	1.6	4	2.6.29

Actual usage (and probably some behavioral) data is compiled by Google, and is made available through the Dashboards on the Android Developer Website[2]. Since there are virtually no devices remaining with versions older than Froyo, this work does not make any attempt to discuss them.

Froyo

FroYo (Frozen Yogurt) was the first version of Android to support application installation on external media (i.e. SDCards). It additionally introduced the notion of Android Secure Containers (ASEC), in order to provide security for files on external media, which by its nature is usually FAT formatted volumes (The ASEC mechanism is discussed in Chapter 2). Another useful feature introduced in this version was USB tethering (connecting the device and using its Internet connection, as discussed in Volume II). Lastly, Froyo brought significant speed improvements to Dalvik, with the introduction of Just-In-Time (JIT) compilation by a dedicated thread.

Gingerbread



Gingerbread was the first version of Android to gain widespread adoption, and with good reason: It introduced significant enhancements to the system. At the Dalvik layer, concurrent garbage collection was introduced, which improved application response time by running GC in parallel, rather than pausing the application during the process. Likewise, the JIT mechanism improved on Froyo's. The sensor APIs underwent a complete revamp, extending the sensor HAL to support more sensor types, and making them more accessible to native code. Support for native code was bolstered in other areas as well, providing native access to audio, graphics, storage and even the activity manager. Gingerbread was also first to introduce support for Near-Field-Communications (NFC), though it was only till later (with ICS) that NFC was to be adopted into ubiquity by Android vendors.

Another noteworthy addition is support for OBB - opaque binary blobs (referred to as "APK expansion files") as a workaround to the size limitation of application package sizes, and to provide optional encryption. OBB files are discussed in Chapter 2. Last, but not least, Gingerbread adopted Ext4 in place of YAFFS as the default filesystem.

All these improvements aside, Gingerbread is actually most notorious for being the most insecure version of Android to date. Apart from glitches with the stock SMS app (which routed messages to the wrong recipients), it was riddled with quite a few vulnerabilities which led to an explosion in rootkit-grade malware.

Honeycomb

Honeycomb brought Android to tablets. In fact, it was a "tablet-only" release, in that the source tree was never fully released nor meant to be used for phones (though some vendors still tried to use it nonetheless). The main change was the introduction of fragments, which - like Windows' Multiple Document Interface (MDI) allow several client areas to coexist simultaneously, rather than the single layout architecture which was previously used.

Honeycomb also offered significant improvements in graphics - introducing hardware accelerated OpenGL rendering for 2D, and introduced Renderscript, which is Android's own GL-like language.

Another feature of importance was the advent of storage encryption. Honeycomb was the first version of Android to offer low level encryption of the user data partition, bringing it in line with iOS 4, which introduced it as well. The disk encryption in Android is carried out by the Linux device mapper, and can be thought of as the next step, following the Android Secure Storage which was introduced in Froyo.

More important than the user space features was the introduction of multi-core support into Android. Primarily, this involved a recompilation of the Linux kernel to support SMP (as can be seen with the BusyBox uname tool, or /proc/version). Tablets were the first devices to utilize multi-core architectures, which have since proliferated to all but the cheapest devices. The Android Documentation[6] details the changes required for code to be SMP safe - most of these are primarily in native code, though some aspects apply to Java as well.

Honeycomb was the only version of Android whose source code was not made open (aside from select portions). This made some vendors wary, and brought to mind the fact that even though Android is free, Google still controls the system, and its licensing may change at any point in the future, if Google so sees fit.

Ice Cream Sandwich

Ice Cream Sandwich (ICS) brought many changes to Android, as can be expected from a 4.0 release. Aside from the myriad UI enhancements, those changes which were noticeable to users included significant connectivity enhancements - The Android VPN Framework, WiFi Direct and Android Beam.

ICS adds another API often overlooked by developers - the `onTrimMemory()` callback, which is called at times of memory pressure. According to the integer code specified, an application should release as much memory as possible. Note, however, that this API is advisory - applications can just choose to ignore the callback (which all too many, in fact, do).

JellyBean

JellyBean's most prominent user-facing feature is in its support for multiple users on the same device. This feature, more useful on tablets than phones (and formally only enabled on the former), allows several users to operate the device. Though only one user can be actively logged on, each user has a different UI, with separate widgets and applications and - most importantly - isolation of application data. We discuss the implementation of this feature in detail in Chapter 8.

In addition to this, and alongside the slew of UI features, JellyBean also provided application encryption and forward locking, building on Froyo's ASEC containers. One of the main drawbacks of Android's open nature at the time was that it was trivial to pirate apps by moving them between devices via the SDCard. ASEC provides a secure container for data, which can be encrypted by the application, and made readable only by the application's uid (but still fails miserably on rooted devices). This will, as mentioned, be covered in Chapter 2.

JellyBean went through three API versions, which introduced many changes, both over and under the hood. API 17 also brought SELinux to Android for the first time (as detailed in Chapter 8), and sealed a gaping USB debugging hole by forcing authentication over ADB. Notable changes in API 18 were support for OpenGL ES 3.0, Bluetooth Audio-Video Remote Control Profile (AVRCP) 1.3 and Bluetooth Low Energy (LE) support, as well as the App Ops service (whose UI was later removed in 4.4.1), which allows tweaking application permissions.

KitKat

Version 4.4 of Android was codenamed "KitKat" (and was actually launched in partnership with Hershey's). It represents a genuine attempt by Google to combat the fragmentation of the Android universe: Though JellyBean is the single most popular version, a large percentage of devices still use old versions - notably GingerBread, which are not only obsolete, but hamper apps from running due to their old API versions. Additionally, middle and high-end market become saturated, and in the entry-level category Android faces competition from FireFox OS and others.

KitKat's "pet project" was "svelte", an initiative to enable a smooth experience on virtually any device, including low-end devices, with 512MB of RAM. Part of the rationale behind it is that a smoother OS with less resource requirements would enable all vendors - even those with entry level devices - to offer the latest OS version, thereby ending fragmentation. Doing so involved many under-the-hood changes, such as rewriting framework code to use less memory, and starting services in a serial manner (to reduce memory pressure). A new API was added to detect low RAM devices (ActivityManager.isLowRamDevice(), which returns the value of the `ro.config.low_ram` property). Using this API, developers can detect the amount of RAM available, and plan resources accordingly. KitKat also added the `procstats` service to give developers as much information as possible on their application's footprint.

For those devices which do have RAM, KitKat utilizes a new feature of the Linux Kernel, called zRAM. This feature is in fact newer than KitKat itself, having only been officially stabilized and merged in version 3.14 (KitKat uses 3.4), but Google was an early adopter in both ChromeOS and Android. The features enables swapping to RAM, and thus overcomes one of the major limitations of mobile devices - the lack of swap on flash devices. While swapping to RAM might sound somewhat counterproductive, it is in fact a dramatic improvement, as the swapped pages are compressed (thus saving overall RAM usage) and quite fast to retrieve. Devices which use compressed RAM will have a special block device (e.g. `/dev/block/zram0`), indicated by the `/proc/swaps` file.

Compressed memory also made its debut in iOS version 7.0, a few months before KitKat was announced. Several other interesting features in KitKat may have borrowed from iOS 7.0 include a step counter (software-defined sensor, as an answer to Apple's M7), as well as timer coalescing and sensor batching. The latter two are a significant improvement that helps maximize battery life. To do so, Android actually reduces the granularity of timers and updates from sensors, making them more coarse, but also more likely to coincide. This can greatly improve battery time - both directly (longer periods of CPU idle time), and indirectly (reducing the overall number of wakeups, which are costly both in power and performance).

Other notable features in Kitkat include Bluetooth MAP support, Infrared Blaster (ConsumerIr) APIs, A new printing framework, and NFC host card emulation. Probably the most far reaching change, however, was unannounced and kept under the scenes: introducing the Android RunTime (ART), as an optional replacement to Dalvik.

At the time of writing, KitKat has undergone four minor revisions, and its most recent version is 4.4.4. Those revisions are mostly bug fixes and camera enhancements, and do not change the API version, though internal APIs have been modified. KitKat remains the most common Android version, installed on 40% of devices (as of late April 2015).

Lollipop

The latest version of Android (at the time of writing) is Android Lollipop. The most obvious user-facing change in this version is the introduction of "Material Design", a flat interface which aims to provide realistic lighting and motion effects, and print-based design which is strangely reminiscent of iOS 7's overhaul. Another emphasis in this release is on notifications, support for which has been greatly expanded.

Under the hood are far more significant changes: First and foremost is the adoption of the Android Runtime (ART), which brings performance improvements by compiling Dalvik code to native code Ahead Of Time (AOT), rather than Just In Time (JIT). Aside from performance, ART also allows Android apps to exploit the benefits of 64-bit architecture, as discussed in depth in Volume II. The graphics stack has been updated with support for OpenGLES 3.1, and the audio frameworks have been upgraded, particularly for better audio input handling. Likewise, camera APIs have been revamped. Sensor support (via the Hardware Abstraction Layer) has been upgraded, with support for more complicated gestures, and even a heart rate monitor. The "pet-project" of this release is "Project Volta", which aims to both improve battery life (through the new job scheduling API) and provide better power monitoring tools (notably, the `batterystats` service). Lollipop also serves as the foundation for the new "Android TV".

Lollipop's release was quite lengthy and somewhat painful - It took Google about six months from announcement (6/2014) to official launch (11/2014), and even at the time of writing, it is supported mainly on the Google Nexi, with a penetration rate of about 10% (and that, too, for versions before 5.1). Major bugs (ironically, relating to power management and performance) have been discovered in the earlier releases, and - much to the chagrin of vendors who are still playing catch up - pushed Google to update Lollipop (as of 03/2015) to 5.1 , with API level 22, and numerous bugfixes. 5.1 adds myriad UI tweaks, and - more importantly - notable features such as HD-Voice calling, Dual-SIM support, and "Device Protection" (the much needed "kill switch" to lock stolen devices remotely).

M

Android M (final name as yet is unknown) is Google's latest version of Android - It was announced in Google I/O on 5/28/2015. Having learnt from the mistakes made with L, Google has committed to a strict timeline consisting of three developer releases, each a month apart, with a final release date by the end of Q3 2015. Google provided both Emulator and factory images (for the Google Nexi), including - for the first time - images for ARM64, which the QEMU emulator in the Android SDK now supports. Sources are also available through the Android GIT repository.

From a feature perspective, M seems more of an evolutionary update, than a revolutionary one. While it adds several noteworthy features, these are mostly in response to iOS, and include support for payments, built-in fingerprint authentication (which was introduced in Lollipop, but is now made available for use by apps), and floating toolbars for text selection.

An important improvement comes int the form of revamping the App permission model, which finally moves the permission enforcement to runtime, rather than install time. This brings Android in line with the iOS model, by prompting the user to allow sensitive operations when they happen, rather than approve a mile long list of permissions in bulk upon installation (discussed more in Chapter 8), and greatly mitigates the potential for trojan apps surreptitiously trying to access your personal information or camera, while entertaining you with a flapping bird.

M also aims to improve on two drawbacks of its predecessors: Data encryption (which was introduced with HoneyComb and enabled by default in Lollipop) is now extended to external storage, by means of *adopted* devices. Power management - always a challenging issue - is further improved with "Doze" mode, sleeping for long intervals between periodic wakeups for app syncing and pending work. M also introduces App idle detection (somewhat reminiscent of OS X's "App Nap" feature, which suspends apps which are not in use.

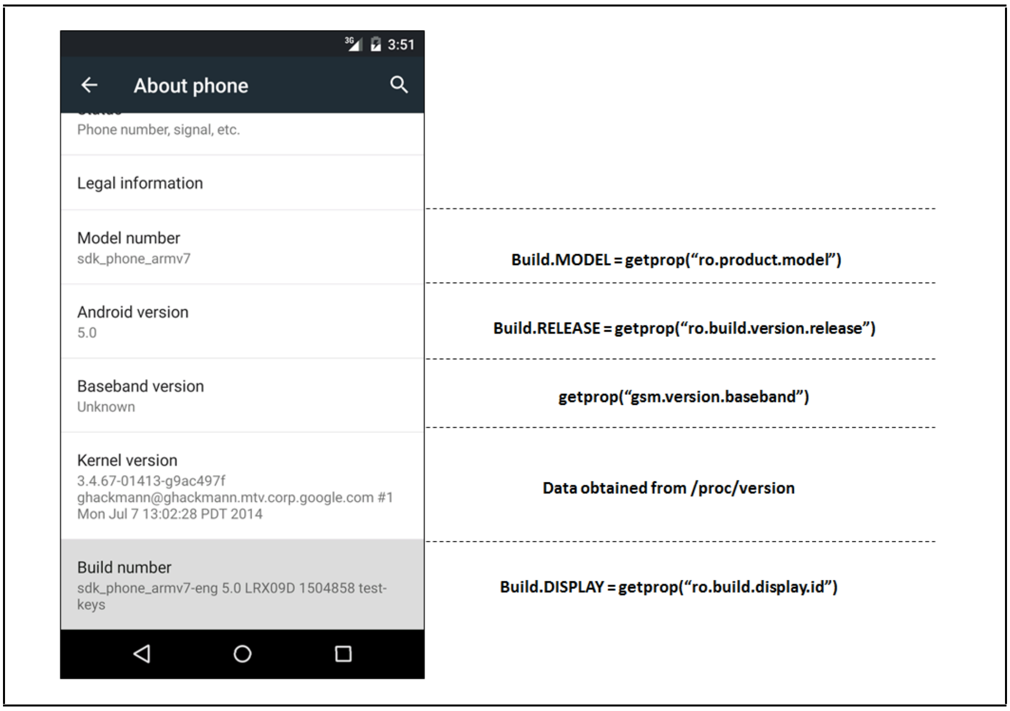
Other, more original features include Direct Share, App Linking, improvements to audio/video syncing (including fast or slow motion playback), MIDI support, direct flashlight (torch) support, camera API extensions, improved notifications, and significant enhancements for "Android for Work". A full list of changes can be found on the Android Developer Website (http://developer.android.com/preview/api-overview.html).

If Google is true to their own advertised schedule, M may overtake Lollipop, (whose adoption rate is still dwindling in the low teens, as best) . Vendors may choose to wait a bit, rather than have to go through the long process of upgrading to Lollipop - only to be forced to upgrade again when M comes out shortly after.

Experiment: Figuring out your device's Android version

Though vendors customize Android in a variety of ways, the basic underlying system is the same. Most Android users are familiar with the Settings >> System >> About Phone GUI, which provides details about the Android version used*. The relevant class is `com.android.settings.DeviceInfoSettings` (found under the AOSP's packages/apps/settings), which uses the android.os.Build class. The values, however, are obtained from system properties, so an often simpler way of getting to those values is directly, using the getprop tool. This is shown in Figure 1-1:

Figure 1-1: Mappings between the Settings app `DeviceInfoSettings` and system properties



Model number
sdk_phone_armv7

Android version
5.0

Baseband version
Unknown

Kernel version
3.4.67-01413-g9ac497f
ghackmann@ghackmann.mtv.corp.google.com #1
Mon Jul 7 13:02:28 PDT 2014

Build number
sdk_phone_armv7-eng 5.0 LRX09D 1504858 test-keys

Build.MODEL = getprop("ro.product.model")

Build.RELEASE = getprop("ro.build.version.release")

getprop("gsm.version.baseband")

Data obtained from /proc/version

Build.DISPLAY = getprop("ro.build.display.id")

The property settings, which are generated from the AOSP and placed into /system/build.prop, hold true on modified builds as well - even those as heavily customized as Amazon's "FireOS". The most useful properties are `ro.build.version.sdk` (API version), and `ro.build.fingerprint`, which is itself an amalgam of several other properties, for example:

```
generic/sdk_phone_armv7/generic:5.0/LRX09D/1504858:eng/test-keys
```

Property	Describes
ro.product.manufacturer	Vendor id
ro.product.name	Device code name. For Google - fish names
ro.build.product:version.release	Product name and Android base version
ro.build.id	first letter: version (rest described in the documentation[11])
ro.build.version.incremental	Internal build number, auto-incremented by AOSP build system
ro.build.type	user: user facing, eng: Engineers/internal
ro.build.tags	release-keys: production system, actual certificates. test-keys: development

* - As of JellyBean, the "Build Version" provides the backdoor functionality to the Developer Settings (which include ADB), by clicking seven times on the view.

Android vs. Linux

Not just another Linux Distribution

Linux, the core of Android, has been around for well over a decade before Android was even conceived. Linux is a fully open source operating system, whose kernel started as a Master's Thesis of one, Linus Torvalds, and has since gained worldwide fame and adoption. A kernel alone, however, does not a full operating system make. Torvalds decided to release his work as open source, and attracted developers who extended it further, by providing components for it - binaries both ported from other UN*X systems, as well as original ones. Linux exploded in popularity as a free alternative to the expensive UN*X systems of the time, effectively undercutting them and leading to the demise of most.

Along its rapid evolution, Linux attracted commercial interest. Companies, whose sole purpose was to package the kernel along with additional binaries, sprouted and provided "distributions" of Linux. These companies often provided Linux for free, basing their entire business model on support. At times, "professional" or "enterprise" grade distributions, containing custom tweaks or specialized tools, were provided, costing money to license.

Linux quickly became the de facto operating system of the embedded space. Contrary to other players in the field, such as Windows CE (which required too many resources), and real time operating systems such as PSOS or VxWorks (both of which involved heavy licensing fees), Linux offered a platform that was not only free, but fully customizable and light weight. One company, MontaVista, based its entire business model on porting Linux to the embedded space - notably, the ARM, MIPS and PowerPC architectures. The port provided for Embedded platforms the same functionality as that which was provided on the desktop - a fully featured shell environment. All for a generous licensing fee.

But developers needed more. Long gone are the days of shell interfaces, and all users (save for battle-hardened veterans) expect a graphical user interface from their operating system. Linux relied on X-Windows, the traditional UN*X Windows architecture, for its GUI. Setting up a GUI on an embedded platform was far from straightforward. Graphics programming using X-Windows API was also quite cumbersome. Additionally, vendors such as Montavista provided just the basic platform. Developers still had to port additional components and create their own, often having to start from scratch.

And then came Android.

Google spotted the promise in a mobile operating system back in 2005, when they acquired Android, then a small startup by Andy Rubin. Android disappeared off the map, till its resurgence some years later (shortly after Apple's "iPhoneOS"). Mobile vendors, trying to adapt to the revolutionary device, quickly wanted to provide a similar experience - and needed to catch up quickly.

Android's novelty arises from what it aims to provide - not just another Linux distribution - but a full **software stack**. The term "stack" implies several layers. Android provides not just the basic kernel and shell binaries, but also a self-contained GUI environment, and a rich set of frameworks. Coupled with a simple to use development language - Java - Android gives developers a true Rapid Application Development (RAD) environment, as they can draw on pre-written, well-tested code in the frameworks to access advanced functionality - such as Cameras, motion sensors, GUI Widgets and more - in a few lines of code. With features that at first borrowed heavily from iOS and later improved on them, Android became the de-facto OS for Mobile, much as Windows is for the Desktop, or Linux was elsewhere.

Android has since had its hegemony constantly reinforced by the feedback loop of its ecosystem - Android's "App MarketPlace", (which quickly followed Apple's "App Store"), adopted that model to allow developers to quickly distribute their apps in a manner far more (and some would say, too) relaxed with virtually no hurdles. The result is that Google Play (as the MarketPlace is now known) has surpassed the App Store and offers millions of apps. Adopting Android provides a mobile vendor with instant access and compatibility with those apps, but only if they comply with Google's Mobile Application Distribution Agreement (MADA), which mandates full integration of Google's apps and services.

In a sense, Android has done to MontaVista and other Embedded Linux firms what Linux has done to UN*X and other competitors - undercutting by providing a totally free alternative. Google pushes Android for free, with no licensing fees (at least, for now), and fairly relaxed terms of use (though those are getting tighter, slowly but surely). It's no wonder, then, that Android has risen in only a few years to achieve almost total hegemony of about 80% of the global mobile market, leaving only a persistent bastion of iOS (presently at about 20%), along with nigh-insignificant dregs of Windows Mobile and BlackBerry. A mobile vendor basically has only very limited options in a choice of operating system: develop a homegrown one, or go with a ready made one. Nearly all opt for the latter*, and then the choice boils down to Android, or Windows Mobile. Microsoft has tried to follow the Android model and offer its system for free - but the effort was too little, and far too late - as it lacks the ecosystem. BlackBerry, on its own part, has ported the Android runtime to its own OS, hoping to win back market share by providing runtime compatibility with the multitude of Android Apps.

Commonalities and Divergences from Linux

Android is built on top of Linux, but modifies it in substantial ways - including some which break compatibility with the mainstream. The Android kernel source tree diverged from the mainline kernel around version 2.6.27, but has been converging since version 3.3. In user-mode, Google maintains the frameworks and runtime of the AOSP (Android Open Source Project) in an entirely separate repository. From a high-level perspective, though it's hard to quantify exactly how much the two OSes differ, a safe estimate would be that Android and Linux are about 95% alike at the kernel level, and about 65% or so at the user-mode.

This guesstimate is drawn by taking into consideration that, at the kernel level, aside from a few differences (ARM platform and drivers not withstanding), the rest of the kernel source is unmodified. Those differences (which include IPC, memory and logging enhancements) are collectively referred to as **Androidisms**, and most have in fact by now been merged into the mainline - either replaced with similar kernel functionality, or included in the drivers/staging/android) directory.

At the user-mode level, there is more of a divergence, introducing two entirely new components - the Dalvik runtime and the Hardware Abstraction Layer - as well as replacing glibc with Bionic, and providing a custom version of init, the system startup daemon. The underlying OS, however, still remains for the most part unmodified, with native binaries, processes and threads behaving as they do on Linux. This enables the approach taken in this book, of discussing low-level Linux-based approaches for debugging and tracing, as is discussed in Chapter 7.

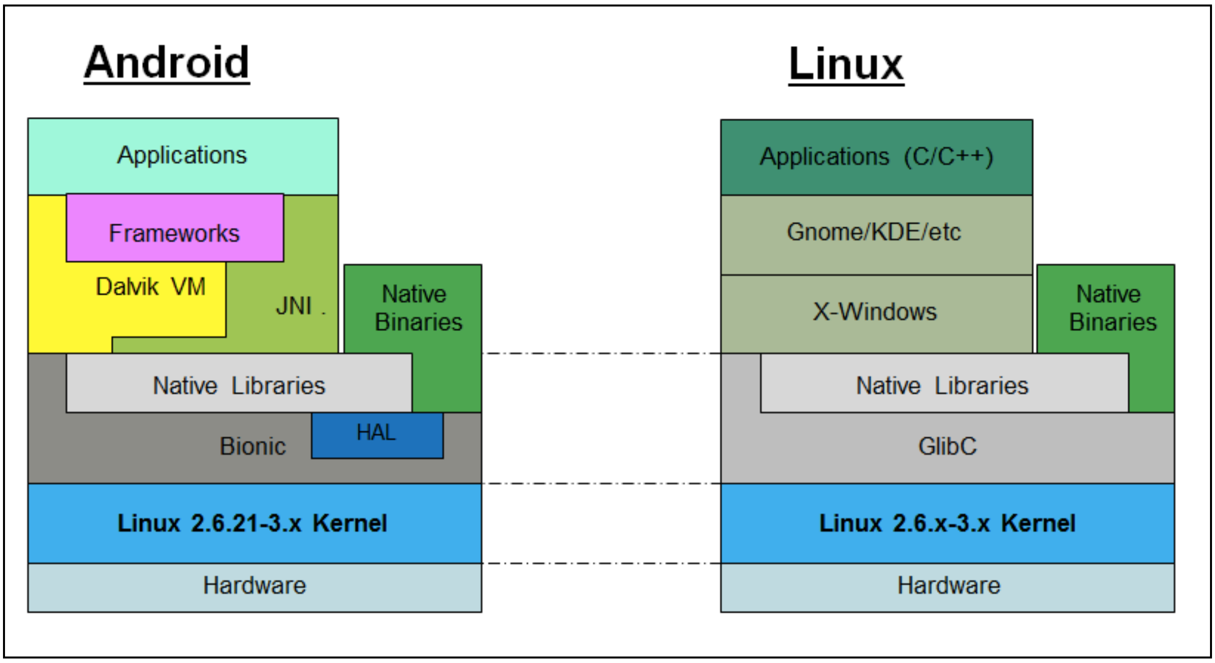
Android also makes more clever use of features present in Linux, though left unused in most desktop distributions. These include control groups, low-memory conditions (Linux OOM, which Android expands on with its Low Memory Killer), and security features - capabilities and SElinux (as discussed in Chapter 8).

Android also uses quite a few open source projects which were of limited popularity in Linux, but form the backbone of its feature set. These projects (in the external/ folder of the AOSP) are largely responsible for implementing Android's network capabilities, and include `racoon` (vpn), `mdns` (service discovery and Wi-Fi Direct), `dnsmasq` and `hostapd` (tethering and Wi-Fi Direct), and `wpa_supplicant` (Wi-Fi). Other open source projects provide library-level support (discussed and shown later in Table 1-3).

* - Mobile device vendors are becoming increasingly uneasy with several shortcomings of Android: The first, is the common feature base, which makes it hard to differentiate their product from others. The second, is increased dependency on Google, which actually strives to enforce the Android look and feel across devices. Lastly, Google's Mobile Application Distribution Agreement (MADA), which forces the inclusion of all Google Apps in order to gain access to the Play Market. This has led some vendors (notably, Samsung) to look at alternatives (e.g. Tizen). At present, Android seems to be fully entrenched and not likely to lose dominance any time soon.

Figure 1-2 compares and contrasts the software stacks provided by Linux and Android. We then move to explore the notable differences, in turn.

Figure 1-2: The Android Architecture, compared with that of mainstream Linux



Most developers are probably familiar with the Google-provided architectural diagram (returned by searching for "Android Architecture") ad nauseam (virtually all other books on Android include it in their introduction - often with the exact same colors). That diagram, in this author's opinion, is simplified and not at all accurate in its representation of layers (For example, JNI is entirely overlooked). As the layers unfold, this slightly different rendition of the architecture will hopefully "make sense" just as much as, if not more than, the traditional diagram. (Only as this book goes to print has Google finally provided a more accurate (and visually appealing) diagram at the Android Source website[12])

The Android Frameworks

Android owes a key part of its success to its rich set of frameworks. Without them, Android would have likely ended up as just another embedded Linux distribution (and would have in fact gone the way of MontaVista, which was highly popular before Android made its debut). By providing the frameworks, Android facilitates the application creation process, allowing developers to use the higher-level Java language, rather than low-level C/C++. The addition of the frameworks further expedites the process, as developers can draw on the plentiful APIs, which handle graphics, audio and hardware access. Unlike X-Windows and GNOME/KDE, these are far simpler, and operate in a much more straightforward manner.

Through the use of Java package naming, Android frameworks are divided into separate namespaces, according to their functionality. Packages in the android.* namespace are available for use by developers. Packages in com.android.* are internal. Android also supports most of the standard Java runtime packages in the java.* namespace. Table 1-2 shows the breakdown of the commonly used frameworks by package, sorted by the API version they were introduced in, so as to give an idea as to the evolution of the operating system features. Note that the table only shows when frameworks made their debut, and does not show their expansion, which does occur in between API versions, as more classes are added.

Table 1-2: The Android Frameworks

Package Name	API	Contents
android.app	1	Application Support
android.content		Content providers
android.database		Database support, mostly SQLite
android.graphics		Graphics support
android.opengl		OpenGL Graphics support
android.hardware		Camera, input and sensor support
android.location		Location support
android.media		Media support
android.net		Network support built over java.net APIs
android.os		Core OS Service and IPC support
android.provider		Built-in Android content-providers
android.sax		SAX XML Parsers
android.telephony		Core Telephony support
android.text		Text rendering
android.view		UI Components (similar to iOS's UIView)
android.webkit		Webkit browser controls
android.widget		Application widgets
android.speech	3	Speech recognition and Speech-to-Text
android.accounts	4	Support for account management and authentication.
android.gesture		Custom gesture support
android.accounts	5	User account support
android.bluetooth		Bluetooth support
android.media.audiofx	9	Audio Effects support
android.net.sip		Support for VoIP using the Session Initiation Protocol (RFC3261)
android.os.storage		Support for Opaque Binary Blobs (OBB)
android.nfc		Support for Near Field Communication
android.animation	11	Animation of views and objects
android.drm		Digital Rights Management and copy protection
android.renderscript		RenderScript (OpenCL like computation language)
android.hardware.usb	12	USB Peripheral support
android.mtp		MTP/PTP support for connected cameras, etc
android.net.rtp		Support for the Real-Time-Protocol (RFC3501)
android.media.effect	14	Image and Video Effects support
android.net.wifi.p2p		Support for Wi-Fi Direct (Peer-To-Peer)
android.security		Support for keychains and keystores
android.net.nsd	16	Neighbor-Service-Discovery through Multicast DNS (Bonjour)
android.hardware.input		Input device listeners

* - This table, while detailed, is not comprehensive, and only reflects the more important classes. A full list can be found at http://developer.android.com/sdk/api_diff/##/changes.html, replacing ## with the API level

Table 1-2 (cont.): The Android Frameworks

Package Name	API	Contents
android.hardware.display	17	External and virtual display support
android.service.dreams		"Dream" (screensaver) support
android.graphics.pdf	19	PDF Rendering
android.print[.pdf]		Support for external printing
android.app.job	21	Job scheduler
android.bluetooth.le		Bluetooth Low-Energy (LE) support
android.hardware.camera2		The new camera APIs
android.media.[browse/projection/session/tv]		Media browsing and TV support
android.service.voice		Activation by "hot words" (e.g. "OK Google")
android.system		`uname()`, `poll(2)` and `fstat[vfs](2)`
android.service.carrier	22	SMS/MMS support (CarrierMessagingService)

In practice, the entire set of frameworks is bundled into several Java ARchive (.jar) files on the device, in `/system/framework` and - in L - precompiled into the `boot.art` file. Although the AOSP is open source, it can come in quite handy to locate a package directly in the JAR itself, which you can do by invoking dexdump (or the dextra tool) on the classes.dex files inside the JARs.

Figure 1-3: Dalvik, Iceland (photo by Author)

The Dalvik Virtual Machine

Android's other notable addition is the introduction of the Dalvik Virtual Machine. This VM became key to making Android workable on mobile devices back when 256M of memory was considered "plenty". Dalvik was not the first type of Virtual Machine to be attempted on mobile devices - Sun Microsystems hoped to push Java 2 Mobile Edition (J2ME), but with very little success.

Dalvik is largely the brainchild of Dan Bornstein, whose Google I/O 2008 presentation serves as a great reference as to its design. The name - Dalvik - was chosen in honor of a fishing village in northern Iceland.

The Dalvik VM, though seemingly java-esque, is actually not a Java Virtual Machine. Though not too far-removed from one, it runs a different form of bytecode (called DEX, for Dalvik Executable), and is more optimized for efficiency and sharing memory than the JVM designed by Sun/Oracle. Those very optimizations enabled it to prevail despite the strict constraints of mobile platforms, which have felled Java (specifically, J2ME) from gaining ground outside limited implementations.

Android used a subset of the Apache Harmony files as basis for its core classes. Harmony was chosen as a free (Apache-license) open source clone of (then Sun's, now Oracle's) JVM. Oracle actually sued Google in 2010 for never properly acquiring a license for the Java class libraries, and the saga is far from conclusion even in early 2015.

As this book goes to print, Dalvik is being superseded by the Android RunTime (ART), as described later in this chapter. Contrary to popular belief, however, this does not mean Dalvik is going away: Only the Just-In-Time (JIT) compilation aspect of it has been replaced, but the file format (DEX) is still very much alive, as are the key architectural concepts. We therefore discuss both Dalvik and ART in great detail in Volume II.

JNI

Android Applications run in the virtual machine, but at times need to escape it - usually to access hardware or other device (or chipset) specific features. Dalvik therefore allows the inclusion of native libraries (ELF shared objects) in application code, through the **J**ava **N**ative **I**nterface (JNI).

Android has somewhat of a love/hate relationship with JNI. No doubt vendors would be happier with pure Dalvik applications, as those are confined in the VM, and thus remain agnostic to the underlying architecture. In this way, Android applications would run universally - on Intel, ARM, MIPS, and other architectures - with no modification. On the other hand, the VM environment is not without its limits (especially when it concerns graphics) and drawbacks (notably decompilation). It is therefore not at all uncommon to see JNI used in applications optimizing for performance, or seeking resistance to reverse engineering. Google therefore provides the Native Development Kit (NDK) (downloadable at Android Developer[14]), which developers can use to build native libraries (and binaries).

Not all applications use JNI, but in those that do, JNI libraries can be easily seen in the package (.apk) since they are in a separate folder: */lib/architecture*. A good example of this can be found in the DropBox App* (here in an output from a Galaxy Tab 3 10.1), providing JNI support for no less than four different architectures:

Output 1-1: Demonstrating JNI Libraries in an APK

```
root@Tab3:/ # % unzip -l /system/app/Dropbox.apk | grep lib/
    17532  05-14-13 23:11   lib/armeabi-v7a/libDbxFileObserver.so
    17528  05-14-13 23:11   lib/armeabi/libDbxFileObserver.so
     9352  05-14-13 23:11   lib/x86/libDbxFileObserver.so
    71076  05-14-13 23:11   lib/mips/libDbxFileObserver.so
```

JNI normally works seamlessly across ARM devices (which comprise the vast majority), though processor version differences (e.g. ARMv6, ARMv7) do require different libraries (hence "armeabi" and "armeabi-v7a" in the output). When it comes to x86 architectures, JNI is a major headache for Intel, who would like to see more vendors use its chipsets for Android. Rather than depend on the app developers to compile an x86 specific version (most don't), Intel provides a closed-source ARM emulation called Houdini (extending Dalvik/ART, as discussed later in Volume II) as part of their Android distribution. This emulator, (along with a few minor modifications in Dalvik), enables ARM native libraries to work on Intel architectures.

Native Binaries

From the Linux perspective, all executables are ELF binaries. Android's critical system component are implemented in C/C++, and are compiled into native binaries. User applications are compiled into Dalvik bytecode, but the bytecode runs (or, in ART, is compiled ahead-of-time) in the context of a Dalvik Virtual machine, which is, in and of itself, an ELF binary. Thus, while most developers remain oblivious to binaries, they nonetheless play an important role in Android.

Binaries are usually located in /system/bin, and /system/xbin (with a few critical binaries located in /sbin). Most binaries are usually the same across all devices, being part of the AOSP, but it is not uncommon to find additional binaries from the vendor or chipset manufacturer (e.g. `mpdecision`, on Qualcomm MSM multi-core devices). You can see a list of processes loaded from native binaries at any time by filtering the ps command. This is shown in Output 1-2 (from an HTC One M8), with the AOSP binaries highlighted:

* - Not to be confused with the commercial app of the same name, used to for cloud storage

Output 1-2: Native binaries executing on an an HTC One M8

```
shell@htc_m8wl:/ $ ps | grep " /" | cut -c1-22,55-
root      1     0      /init
root      218   1      /sbin/ueventd
root      365   1      /sbin/healthd
system    367   1      /system/bin/servicemanager
root      368   1      /system/bin/vold
radio     369   1      /system/bin/rild
system    370   1      /system/bin/surfaceflinger
root      371   1      /system/bin/pnpmgr
nobody    373   1      /system/bin/rmt_storage        # QCOM specific
radio     375   1      /system/bin/qmuxd              # QCOM specific
radio     376   1      /system/bin/netmgrd
root      379   1      /sbin/tpd
root      380   1      /system/bin/netd
root      384   1      /system/bin/debuggerd
drm       387   1      /system/bin/drmserver
media     388   1      /system/bin/mediaserver
install   389   1      /system/bin/installd
keystore  390   1      /system/bin/keystore
shell     391   1      /system/bin/dumpstate
root      393   1      /system/bin/thermal-engine     # QCOM specific
root      395   1      /system/bin/memlock            # HTC specific
root      397   1      /system/bin/clockd             # HTC Specific
system    400   1      /system/bin/qseecomd           # QCOM TrustZone
root      404   1      /system/bin/cand
system    505   400    /system/bin/qseecomd           # QCOM TrustZone
media_rw  844   1      /system/bin/sdcard
system    847   1      /system/bin/time_daemon        # QCOM specific
root      848   1      /system/bin/dmagent            # HTC specific: QCOM DIAG
nobody    850   1      /system/bin/hvdcp              # QCOM Quich charge support
system    851   1      /system/bin/wcnss_service      # QCOM WLan
root      852   1      /system/bin/htc_ebdlogd        # HTC Specific
root      868   852    /system/bin/logcat2
media     919   1      /system/bin/adsprpcd           # QCOM Application DSP
root      1170  1      /system/bin/logwrapper
wifi      1171  1170   /system/bin/wpa_supplicant
media_rw  1631  1      /system/bin/sdcard
root      1637  1      /system/bin/mpdecision         # QCOM SMP Policy
shell     12277 12149  /system/bin/sh
camera    23853 1      /system/bin/mm-qcamera-daemon  # QCOM camera support
```

Because ELF is a standard file format, you can use any of the Linux ELF parsing tools (such as `readelf`, `objdump`, or other tools in the set of `binutils`) to handle the Android binaries. The Android NDK provides the full toolset (cross compiled so it can run on the host) in the toolchains/directory, supporting x86, MIPS, ARM and - as of r10d - ARM64 - as shown in Output 1-3:

Output 1-3: Locating the Android NDK's binutils

```
# replace "arm-linux-androideabi-4.9" with "aarch64-linux-android-4.9" for 64-bit ARM
morpheus@Forge (~)$ ls $NDK_ROOT/toolchains/arm-linux-androideabi-4.9/prebuilt/linux-x86_64/bin
arm-linux-androideabi-addr2line  arm-linux-androideabi-gcc-4.6  arm-linux-androideabi-objcopy
arm-linux-androideabi-ar         arm-linux-androideabi-gcov     arm-linux-androideabi-objdump
arm-linux-androideabi-as         arm-linux-androideabi-gdb      arm-linux-androideabi-ranlib
arm-linux-androideabi-c++        arm-linux-androideabi-gprof    arm-linux-androideabi-readelf
arm-linux-androideabi-c++filt    arm-linux-androideabi-ld       arm-linux-androideabi-size
arm-linux-androideabi-cpp        arm-linux-androideabi-ld.bfd   arm-linux-androideabi-strings
arm-linux-androideabi-elfedit    arm-linux-androideabi-ld.gold  arm-linux-androideabi-strip
arm-linux-androideabi-g++        arm-linux-androideabi-ld.mcld
arm-linux-androideabi-gcc        arm-linux-androideabi-nm
```

Bionic

Contrary to Linux distributions, which use GNU's LibC (GLibC) as their core runtime (the familiar libc.so), Android elects to use its own C-runtime library, which is called **Bionic**. This is touted as being motivated chiefly by simplicity[15], though in practice there is a legal consideration as well - The GNU public license (GPL) places limitations on code which can use GLibC (similar in some respects to GPL portions in the kernel), and Google sought to avoid that*. Bionic is open source, but uses a hybrid of the BSD license (which is far more permissive for third party linkage) as well as Android's own license.

Omissions

Legal issues aside, Bionic is more lightweight than GLibC, and more efficient for Android's purposes, leaving out features deemed unnecessary or too complicated. Notable omissions are:

- **Streamlined system call support:** Since system calls are called frequently, Bionic aims to optimize them by providing the thinnest wrappers possible. The system call stubs are generated with the help of bionic/libc/SYSCALLS.TXT. Some system calls are not at all exported.
- **No support for System V IPC:** Among the system calls not exported by Bionic are those dealing with UN*X System V Inter-Process-Communication (`sem[ctl|get|op]` and Shared Memory (`shm[at|dt|get|ctl`). This was a design decision in Android, deprecating these forms of IPC in favor of Android's own (ASHMem and Binder, described in Volume II).
- **Limited Pthread functionality:** On the one hand, Pthread support is built-in to Bionic (i.e. not a separate libpthread.so). On the other, the pthread support is not full, with the most notable feature missing is support for thread cancellation, via pthread_cancel.

  Mutex support is also stripped down, made more efficient by relying on the kernel's fast mutex (the futex(2) system call), but higher level IPC objects (e.g. rwlocks) have been left out.
- **Limited C++ support:** Though C++ **is** supported (indeed, most of Android's code is written in C++), exceptions are not. Likewise, the Standard Template Library (STL) is not included, though there is no restriction for linking against it manually (a port can be found in external/stlport project).
- **No support for Locales and/or wide characters:** Bionic natively needs only ASCII, though Unicode is also supported via libutils.so

The omissions make sense, considering that most code is meant to be done in the virtual machine, and the VM itself is written to avoid needing these functions: For example, the VM has its own thread management and Unicode support (via ICU). These omissions do, however, pose challenges to native code developers, especially those who seek to port libraries and executables from Linux to Android, as we discuss later.

Additions

Bionic also adds quite a few features to the standard LibC, which are optimized for Android. These include:

- **System Properties:** Properties are a unique feature of Android, which allow both the system as well as applications to supply various configuration and operational parameters in a simple key/value store. This is similar to the notion of Java properties (and, in fact, is accessible through Java's System.properties). Android relies heavily on this mechanism, which is supplied through a shared memory region, accessible and read-only to all processes on the system, but settable only through /init. We discuss the implementation of properties in Chapter 4.

* - Google is avoiding GPL and licensing issues not just in Bionic, but in other components (e.g. udevd). GPL has strict (legal) restrictions requiring linkage with likewise GPL open source components. Avoiding GPL maintains an option for them to close the source at any time in the future (as they did once with parts of Honeycomb).

- **Hard-coded UID/GID implementation:** Rather than rely on the passwd and group files as traditional UN*X does, Android opts instead to hard-code the ids, and emulate getpwnam(3) and friends. The reasoning for this becomes clear when Android's security model is considered: Every application is assigned its own UID and GID (beginning with 10000) and those IDs are then mapped to the human readable app_u*XXX* (or, as of JellyBean, u*XX*_a*YYY*) format. Additionally, Android reserves the lower UID/GID range (1000-9999) for its own subsystems. The AIDs (defined along with directory permissions in android_filesystem_config.h) are described in more detail in Chapter 8, which deals with security.
- **Built-in DNS resolution:** Bionic integrates the DNS name-to-IP resolution code (traditionally in libresolv.so). The code used in Bionic is more secure (randomizes both source port and query ID, to mitigate birthday attacks), and introduces a novel feature - per-process DNS resolution. This allows capturing and redirecting DNS requests by specific applications, through the definition of net.dns.*pid* system properties. The DNS configuration itself is also stored in properties (net.dns#). The nsswitch.conf, which on Linux allows name resolution through alternate protocols (e.g. NIS, LDAP) is understandably not supported, though resolv.conf is still supported (in /system/etc).
- **Hard coded services and protocols:** Doing away with libresolv.so entirely, Android removes support for the protocols and services files (commonly found in /etc on UN*X), and emulates getservent(3) through its internal __res_get_static(). Other APIs, such as getprotoent(3), are not supported.

Porting Challenges

As with the omissions, the additions pose a challenge when trying to port code the other way around - that is, from Android to Linux. If these could be overcome, one could ostensibly port Dalvik to Linux or other OSes (as indeed some developers have, discussed in Volume II), and have Android apps working on desktops, as well.

Bionic presents the main hurdle for porting code to and from Android. While to some extent compatible with GNU LibC, the additions and omissions described above do mean that some more advanced features - notably multithreading - will not port. For some source packages, however, all it takes is recompilation with the NDK. In this way, many tar ball packages can be ported for Android as well, tweaking the configure script and Makefile.

Keeping in mind both Android and Linux export the same system calls, it should come as no surprise that statically linked binaries are often fully compatible (keeping in mind the same underlying CPU architecture). Static linking imports the specific dependencies from the various libraries into the core of the executable. A noteworthy example is Intel's Houdini (discussed in Volume II), which is provided on x86/64 versions of Android. A more common example still is BusyBox, which is an all-in-one binary supplying various shell command functionality: An ARM compiled static binary of Busybox taken from embedded Linux is mostly compatible, although minor aspects (such as displaying Android AIDs) don't always work well.

It's worth noting that there are open issues in Bionic, specified in bionic/ABI-bugs.txt, which affect some esoteric, but nonetheless potentially important datatypes, such as (at the time of writing) time_t (32-bit time, which will blow up in 2038) and off_t (32-bit file offsets). Also, Bionic itself is optimized for 32-bit, and Apple's move to 64-bit will force Bionic (and, indeed, all of Android) to be ported to 64-bit, as is already the case with L, and discussed later in this chapter.

Android Native Libraries

In addition to Bionic, Android contains quite a few other important libraries, which provide runtime support for Dalvik, the frameworks, and the system processes. Those are strewn around the source tree, so the following classifies them by the directories they are in .

Core Libraries

The libraries in `system/core` mostly provide wrappers over kernel Androidisms, or implement additional functionality in user-mode, and include:

- **libcutils**: Provides convenient support functions for kernel exported data (e.g. `/proc/cpuinfo`), socket support, and Androidisms such as ASHMem.
- **liblog**: Which wraps the Android `/dev/log` mechanism, to provide a fast and efficient, ring-buffer based mechanism for logging.
- **libion**: Wrapping the ION Memory Allocator, which was introduced in ICS.
- **libnl_2**: Which wraps the Linux NetLink socket mechanism.
- **libpixelflinger**: Used primarily by the SurfaceFlinger (the core of Android's Graphics stack, described in Volume II). "Flinging" refers to the act of composing two or more inputs so that in the case of graphics, for example, the resulting pixel is a (potentially alpha-blended) color combination of the ones merged.
- **libsuspend**: Which abstracts some aspects of power management, particulary those relating to sleep and suspension of the operating system.

Lesser libraries include:

- **libdiskconfig:** Abstracting disk (flash) configuration and partition management.
- **libcorkscrew:** Used by the debuggerd to unwind stacks and symbolicate application crashes ("tombstones").
- **libmemtrack:** providing process memory tracing services, with the help of hardware modules, if any.
- **libmincrypt:** providing basic implementations of RSA and SHA-[1|256], required for digital signature processing.
- **libnetutils:** Simplifying interface configuration and DHCP support.
- **libsync:** Which wraps the kernel's `sync` Androidism.
- **libsysutils:** Provides primitives used by system utilities. Includes Framework[Client|Listener|Command], Netlink[Event|Listener], Socket[Client|Listener] and ServiceManager
- **libzipfile:** Providing wrappers over `zlib` to handle zip files. Android uses zip extensively, with application packages (`.apk` files) being a special case of zip.

Framework support libraries

Libraries in frameworks/ provide native support services for the Android frameworks. Despite not being part of the "core", they are nonetheless important, and further classified by subdirectories.

- The frameworks/base/core/jni directory contains the very important libandroid_runtime.so, which provides the low level JNI support for the Dalvik VM. The directory contains the JNI components of over 85 framework (Dalvik-level) classes.
- The frameworks/base/services/jni directory contains the equally important libandroid_servers.so, which provides the low level JNI support the Android services.
- The frameworks/base/native/android directory contains libandroid.so, which provides a native interface to assets, storage manager, and more.
- Libraries in base/libs include libandroidfw.so and libhwui.so. The former provides miscellaneous support services such as zip file parsing and asset managements. The latter provides hardware accelerated UI rendering, via OpenGL and SKIA.
- Libraries in av handle media, audio and video. These include:
  - **Camera HAL libraries** - libcamera_client.so and libcamera_metadata.so (q.v. Volume II)
  - **DRM Framework support** (libdrmframework.so) supporting Android's Digital Rights Management mechanism.
  - **Media support libraries** - including libeffects.so, libmedia.so, libnbaio.so, libmediaplayerservice.so, and libstagefright.so.

  The subdirectory av/services contains further support libs for services -libcameraservice.so, libaudioflinger.so and libmedialog.so.

- Libraries in native/libs include:
  - **libbinder**: Binder support functions, discussed in depth in Volume II.
  - **libdiskusage** A tiny library providing directory sizing functions.
  - **libgui**: Provides GUI abstractions (such as the surface), built on top of libui.so
  - **libinput**: Provides basic primitives used by Android's input stack, as described in Volume II.
  - **libui**: Provides the native APIs for Windows and Buffers, used by surfaceflinger (*not* user events).

  The native/ subdirectory also contains the opengl/ directory, which hold EGL and OpenGLES, discussed in Volume II).

External Native Libraries

Android relies on quite a few "external" libraries. The name refers to their location in the Android source tree, and the fact that they are not formally a part of Android - rather, they are open source projects which lend powerful capabilities to the operating system.

There are well over 150 such external projects in the Android source tree, so this work does not make an attempt to cover them all. Table 1-3 nonetheless attempts to touch on the important ones, providing library support:

Table 1-3: External library projects in the Android source tree

Directory	Contents
bluetooth	Bluedroid library (libbluedroid.so), which supports user-mode bluetooth capabilities
icu4c	libicuuc and libicui18n, handling Unicode support and internationalization
mdnsresponder	Apple's Multicast DNS (Bonjour) - contains daemon (mdnsd) and library (libmdssd.so)
libselinux libsepol	SELinux support (JellyBean and later)
skia	The SKIA 2D graphics library (discussed in Volume II)
sqlite	The SQLite3 DB support, providing the core for many Android databases
svox	libttspico and libttscompat, for SVOX Pico Text-To-Speech Engine
tinyalsa	Minimal version of the Linux Advanced Sound Architecture (ALSA) library
webkit	The webkit browser core, used by `WebView` controls
zlib	Zlib - a library providing compression support for gzip and the like

Note that, once deployed on the device, external libraries are largely indistinguishable from those of the AOSP, since all libraries end up alongside one another in `/system/lib`. Similarly, it is possible your device has additional vendor-specific libraries in `/system/lib` (though by convention those should be placed in `/vendor/lib`).

Hardware Abstraction Layer

Android is meant to run on so many types of different devices - tablet, phones, STBs, treadmills, and what not - that the underlying hardware may greatly differ in its scope and support. In an effort to combat this, Android defines a Hardware Abstraction Layer (HAL) which aims to promote standardization by defining an adapter. Hardware vendors are free to implement their own drivers in kernel mode, but must supply a shim, to conform to the interface Android (and particularly, Dalvik) expects. The Hardware Abstraction Layer defines what an abstract camera, GPS, sensor, and other components look like to Android. This does not preclude vendors from extending or modifying functions - it only requires the vendor to drop the shim into /system/lib/hw, and the HAL - libhardware.so will automatically load them. Output 1-4 shows the HAL libraries used in the S5:

Output 1-4: Hardware Abstraction Layer libraries in the Galaxy S5

```
root@S5:/ # ls -l /system/lib/hw
..    9448 2014-03-09 18:21 audio.a2dp.default.so    # a2dp BT audio profile
..    5308 2014-03-09 18:21 audio.primary.default.so
.. 116348 2014-03-09 18:21 audio.primary.msm8974.so
..   17708 2014-03-09 18:21 audio.r_submix.default.so
..    9476 2014-03-09 18:21 audio.usb.default.so    # Audio over USB
..   13552 2014-03-09 18:21 audio_policy.msm8974.so # Audio Policy
. 1306732 2014-03-09 18:21 bluetooth.default.so    # BT, AOSP stock
.. 280728 2014-03-09 18:21 camera.msm8974.so       # Camera
...   5412 2014-03-09 18:21 consumerir.default.so   # Infra-Red
... 17640 2014-03-09 18:21 copybit.msm8974.so      # Hardware accelerated copy
... 26260 2014-03-09 18:21 flp.default.so          # Fused Location Provider
... 21756 2014-03-09 18:21 gps.default.so          # GPS
...   9736 2014-03-09 18:21 gralloc.default.so      # Graphics memory allocator, AOSP stock
... 14328 2014-03-09 18:21 gralloc.msm8974.so      # Graphics memory allocator, Qualcomm
.. 107820 2014-06-06 13:32 hwcomposer.msm8974.so   # Hardware accelerated surface composition
...   5308 2014-03-09 18:21 keystore.default.so     # Cryptographic storage
...   5308 2014-03-09 18:21 local_time.default.so
... 65412 2014-03-09 18:21 nfc_nci.MSM8974.so      # Near-Field-Connectivity
.... 5316 2014-03-09 18:21 power.default.so        # Power Mgmt, AOSP stock
... 21924 2014-03-09 18:21 sensorhubs.msm8974.so   # Sensors
... 54640 2014-06-06 13:32 sensors.msm8974.so
```

The Hardware Abstraction Layer is naturally a very important aspect of Android - both because it represents a divergence from Linux, and because it is instrumental in supporting the slew of Android devices. It is thus deserving of its own chapter, in Volume II.

The Linux Kernel

The Linux kernel, due to its open source and free license nature, provides an excellent substrate for Android*. Now parsecs away from Linus Torvalds' initial version, the kernel keeps evolving at remarkable speeds, with new features added every weeks or months. Android's own capabilities are significantly affected by the kernel's, with notable examples being compressed RAM and 64-bit support. The latter helps explain Table 1-1, which pits kernel version 3.10 as the minimum version for Lollipop: The kernel officially supports ARM64 (AArch64) as of 3.7.

Android kernels are compiled slightly differently than those of Linux, with the config files being generated by merging Android's base and recommended configuration templates with those of the default kernel distribution (as shown in the source.android.com website's kernel section)[16].

As previously mentioned, Android introduces its own idiosyncrasies, or Androidisms, into the kernel. A few of these are in the kernel core, guarded by `#ifdef` statements for conditional compilation, with the rest being in `drivers/staging/android` directory. These Androidisms, as of 3.10 and later, include:

- **Anonymous Shared Memory (ASHMem):** A mechanism to allow shared memory. Applications can open a character device (`/dev/ashmem`) and create a memory region which can then be mapped into memory. This is required to work around the restriction of no world-writable directories and System V IPC.
- **Binder:** The crux of all IPC in Android. A legacy of BeOS, Binder presents a character device (`/dev/binder`) which all applications can open. Android services register with Binder, and clients can connect to them, with the help of `servicemanager`. Binder provides efficient, advanced IPC, as discussed in Chapter 6, and explained in depth in Volume II.
- **Logger:** providing kernel-based ring buffers for fast, file-less logging. Android logs are maintained by character devices in `/dev/log`. Android L augments this with a user mode daemon, `logd`, discussed in Chapter 5.
- **The ION Memory Allocator:** Introduced in ICS, and offers efficient memory allocation to kernel drivers and user mode alike (through `/dev/ion`). ION replaces an older Androidism, PMEM, and aims to standardize memory management in the various SoC architectures.
- **Low Memory Killer:** A layer on top of Linux's own Out-Of-Memory (OOM) killer, which terminates processes in case of memory exhaustion. While the latter is heuristic driven, the former provides a more deterministic way of controlling process termination, and allows defining memory pressure levels. Android L augments this with a user mode daemon, `lmkd`, discussed in Chapter 5.
- **RAM Console:** A mechanism for preserving kernel panic output (thread dump and last `dmesg(1)` log). This has been deprecated in newer releases in favor of the Linux kernel's own `pstorefs` (described in Chapter 2).
- **Sync driver:** The latest Androidism, introduced to allow fast synchronization primitives, used primarily by Android's Graphics stack (in particular, `surfaceflinger`).
- **Timed Output and GPIO:** Allowing user mode programs to access GPIO registers from user space, and automatically reset their values after a timeout. The main client of this is the device vibrator functionality: The framework (via the HAL) can write a millisecond value into `/sys/classtimed_output/vibrator/enable`, to start the device vibration, which automatically quiets down after the timeout specified.
- **Wakelocks:** Originally a separate Androidism to control power management and prohibit the kernel's sleep functionality, wakelocks have gradually been merged with the kernel's own wakeup source mechanisms. (Power Management is detailed in Volume II, with a relevant excerpt on the book's companion website).

* The kernel, in fact, provides a substrate for myriad Linux offshoots, including Samsung's Tizen, Jolia's Sailfish, Firefox OS, and Ubuntu on Smartphones. All of these are seen as potential competitors to Android, though their market share, at least as of early 2015, is infinitessimal.

Android Derivatives

Google offshoots

Google has made it clear that it wants to make Android ubiquitous in all kinds of devices - not just phones and tablets. True to their vision, they announced three new offshoots of Android.

Android Wear

With rumors of Apple supposedly working on an "iWatch", it's no surprise Google quickly rushed to announce "Android Wear" back around KitKat. Android Wear is a version of Android optimized for wearable devices (which, at the time of writing, are a single-category domain, watches, though could ostensibly be extended to other wearable devices). At the core, Android Wear is the same Android used in phones and tablets, but the home activity (main screen) has been replaced by a simpler interface, owing to a watch's diminutive display. This includes an emphasis on voice commands (by tapping on the Google icon), notifications, and cue-cards. Some wear devices also support round screens, as well.

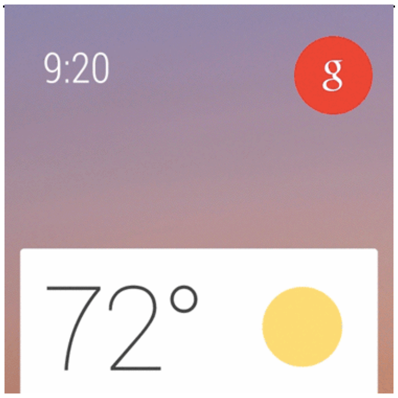


Figure 1-4: Android Wear launcher UI

Android Wear, identified by "clockwork" in the ro.build.fingerprint property, can be thought of as a "slimmed down" version of Android. Unnecessary frameworks and services have been removed, both to conserve memory, as well as CPU (battery life being a major limiting factor of wearable devices). A comparison between the phone and Wear flavor of KitKat reveals that all telephony services (phone, iphonesubinfo, simponebook, isms), as well as print, appwidget, backup, usb, wallpaper, device_policy, and the drmManager have been removed in Wear. Applications have likewise been slashed from over 180MB (in about 60 packages) to a mere 12MB in only 16 packages, leaving only watch specific applications (ClockworkSetup.apk, ClockworkSettings.apk and the PrebuiltClockworkHome.apk launcher), or those that can operate on a small screen (in other words, the default packages of `com.android.*` from KK are not present or loaded in Wear). The SDK for Wear has been released with documentation available in the Android Developer website[17].

Android Wear devices are, at present, designed to serve as satellites for more capable devices, such as a smartphone or tablet. Their only connectivity is via BlueTooth, and most of their frameworks are stubs, which connect to the full featured ones on a phone. Samsung, an early adopter for its "Galaxy Gear" watch, is migrating away from Wear in favor of its homegrown Tizen, citing issues with battery life and limited functionality as being the key drivers.

Android Auto

Shortly after Apple announced "CarPlay", integrating iOS 7 with cars, Google happened to announce "Android Auto", which aims to do surprisingly similar things: Provide a convenient interface to use mobile devices in cars, with access to useful apps such as navigation, the music player, and (of course) the phone. As with Android Wear, there's an emphasis on voice commands and notifications - this time not because of screen limitations, so much as the requirement for hands-free operation.

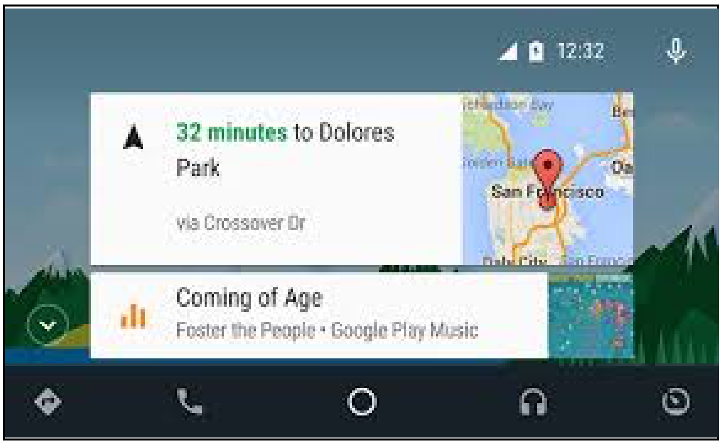

12:32
32 minutes to Dolores Park
via Crossover Dr
San Francisco
Coming of Age
Foster the People • Google Play Music

Figure 1-5: Android Auto UI (source: Google)

From the developer perspective, the important difference is that there is no need for a separate car-specific UI. In fact, there's no need for *any* UI in Android Auto, because the built-in system UI communicates with specific aspects of app functionality, and presents them as the "drawers", which are list driven menus. This enables the driver to select functions with the navigation buttons found on most steering wheels. Apps can still customize the built-in UI, by specifying icons and background images, but don't need to display any custom UI Views, as they would normally. Developers need to declare an additional XML file, with an `automotiveApp` element, and specify which features they use - with "media" and "notification" presently being the supported features.

The XML file is connected to the App via a `meta-data` element in the App's `AndroidManifest.xml`, specifying the `com.google.android.gms.car.application` reserved value for the name attribute.

Google has detailed the interface changes, such as the launcher, and the drawer-based UI at the Android Auto website[18].

Android TV

TV Makers have long been using proprietary OSes to run their device - Samsung's Tizen and LG's WebOS (formerly Palm/HP's) being the two most prominent examples. Google wishes to extend Android's hegemony into this space, as well (gaining the fringe benefits in the trove of user viewing habits). This is Google's second attempt at entering television, with their "Google TV" being less of a niche product than "Apple TV" is.

Android TV has been announced and released alongside Android L, with ample documentation on the Android TV website[19]. From the emulator images, one can discern the main difference is in the launcher (`com.android.mclauncher`, in /system/app/LeanbackLauncher.apk), the built in TV app (`com.android.tv`, in /system/priv-app/TV.apk), and the TV Content provider (`com.android.providers.tv` in /system/priv-app/TvProvider.apk). The content provider exports URIs in android.media.tv for `input` (the remote control), `channel`, `program` and `watched_program`. The latter three are stored in the provider database, /data/user/0/com.android.providers.tv/databases. Other features have also been adapted for TVs, notably remote-control based navigation, and huge screen sizes.

Android TV will likely evolve considerably in the future (perhaps evolving to compete with Apple's plans for extending Apple TV and iOS). Future enhancements would likely involve better streaming support, enhanced EPG (Electronic Programming Guide) functionality, integration with ChromeCast, and gaming platform support. But there is another foe to consider in the TV space -Amazon.

Non-Google ports

Because of its open nature, vendors are free to customize Android in oh-so-many ways. Most enhance (or detract) from the standard UI, in an effort to differentiate their device from "yet another Android". Notable examples include HTC and Samsung, with their "Sense" and TouchWiz UIs, repectively. Others pack Android into new types of devices - for example NVidia with their Shield console. In all the above cases, however, the base system is still very much the same Android. Further, Google provides the "Compatibility Test Suite" (cts/ subtree of AOSP), which vendors must pass in order to get the official blessing (and be assured that apps will function correctly). Additionally, Google makes the Play Market inseparable from its other services. As an price to enter the ecosystem, vendors must bundle the entire set of Google utilities - Maps, Mail, etc - making it more likely the device will be tied to a Google account (and thus, an identity).

Other vendors, however, only take Android as a substrate, and make vast modifications. They willingly give up the ecosystem, because they often create their own. One such example is Chinese smartphone maker XiaoMi, whose top-of-the-line devices at rock-bottom prices has propelled it to be one of China's (and possibly the world's) largest. XiaoMi built an entire business by investing in its own ecosystem, and has willingly abandoned Google's services, most of which are blocked in China anyway. One can imagine Google can't be too happy with it (missing out on order of 100 million or more users), but this is just a consequence of Android's open source nature. And it could be worse -Nokia, for example, experimented with versions of Android that have been "converted" to Microsoft's cloud services. And on this side of the ocean, there's Amazon.

FireOS

Amazon is one of the vendors that has no doubt benefitted the most from Android. The giant retailer made its foray into the tablet market with its Kindle line, which was based on a proprietary embedded Linux distribution, and an e-Ink display. With the Kindle Fire, Amazon modernized their tablet, using Android as the core operating system.

Much to Google's chagrin, however, Amazon fully customized their version of Android, and rebranded it as "FireOS". The interface was entirely revamped (sporting a "carousel" like selection of apps), the devices are locked and keyed to Amazon only - effectively useless without an Amazon ID, and any trace of Google - search, Play store, accounts, or otherwise - has been eradicated.

From a technical perspective, FireOS's core is still very much Android. Changes in it, however, are quite radical and include removal of all things Google, and replacement with Amazon. Specifically:

- Carousel as the home activity: The familiar Android launcher has been replaced by Amazon's custom launcher, com.amazon.kindle.otter.
- Default Browser is "Silk": or, by its other name, com.amazon.cloud9. This is a WebKit based browser, heavily modified and optimized to use Amazon's Elastic Compute Cloud (EC2) to optimize web browsing.
- Google Play replaced by Amazon App Store: Internally referred to as com.amazon.windowshop and com.amazon.venezia.
- Amazon Offers as screen saver: Utilizing Android's "Dreams" functionality to install a screen saver filled with ads. Internally, this is done by several components in the com.amazon.dcp package, and ads stored in /data/securedStorageLocation/dtcp/ (incidentally, revoking permissions on this folder effectively disables ads).
- Aggressive OTA updates: The com.amazon.dcp package contains a host of services meant to ensure the device is constantly up to date. Unlike other Android versions, FireOS doesn't ask to update - it just goes ahead and does so. (Automatic updates are explained in Chapter 3)

Amazon has taken several pages from Apple's playbook, most notably locking down the system to resist rooting (or at least, try) as well as prevent downgrading of the operating system once an update has been installed (which it often is, automatically). With FireOS as a whole, Amazon steps further away from the Google vision of Android, launching its own "Fire Phone", and its "Fire TV", each with proprietary interfaces and APIs.

Headless Android

Take Android, and remove Dalvik and its accompanying frameworks, and you are left with a operating system that has no GUI support, nor any use for an ecosystem. Such an OS, however, is still valuable in its own right, as a base embedded Linux distribution, which has already been adapted to work with ARM and MIPS processors. Before the advent of Android, embedded Linux was a complicated and highly difficult environment, owing in large part to the complexity of building the cross-compiler toolchain, and the user mode libraries. Companies which provided this toolchain and environment (along with support) were highly sought after.

Android, however, completely disrupted the realm, turning the tables on the major Embedded Linux players. Rather than acquire a license for tens of thousand of dollars, embedded Linux now became entirely free - by simply downloading the Android sources and NDK, anyone can build and customize the system to their own needs. Android in its headless deployment now forms the basis for many systems which don't need GUI - sensors, appliances and others, and is likely to be a major player in the "Internet-of-Things" revolution, which promises to embed ARM and MIPS (and maybe Intel) processors in everything but the kitchen sink.

It's possible in Android to enjoy the best of both worlds - that is, both the rich frameworks, and a system with no UI. The system can be made to operate with no UI by setting the `ro.headless` system property. This allows developers to use the frameworks for various non-UI related tasks (such as interfacing with sensors), as well as benefit from the object orientation and other advanced aspects of the Dalvik and ART environments.

Pondering the Way Ahead

Prophecy is the gift of fools, but it's interesting to contemplate the next direction to be taken by Android. The war between iOS and Android rages on, with Android quickly adopting (and, by some claims, blatantly copying) features from iOS - and in some cases vice versa. Still, it seems rather clear from the present landscape as to some features will very likely be included in the next Android - Macaroon, Meringue or whatever condiment name Google will choose for it.

64-Bit compatibility

With the introduction of the iPhone 5S, Apple caught the entire mobile industry by surprise, with the first 64-bit mobile architecture. This perplexed many, which were quick to dismiss it as useless marketing. 64-Bit support was initially discounted because its chief advantage, address spaces larger than 4GB, is in fact questionable in a mobile environment. Though some tablets already ship with RAM of 2GB, 4GB are still beyond the needs of most devices. 64-bit memory access is also slightly less efficient than 32-bit (involving more page table lookups), so many were quick to mock Apple for such a "feeble attempt at innovating" and a useless gimmick*.

In practice, however, there's more to 64-bit than meets the eye. Though ARM 64-bit processors still support 32-bit code, the native 64-bit (ARMv8) instruction set has been completely rewritten to be more efficient. Add to that, the width of 64-bit registers (and the larger register set), and the advantage quickly becomes apparent. The 64-bit architecture (along with some remarkable designs in Apple's custom A7 chip), blew past the performance of all other mobile processors, while maintaining an impresively low power footprint. In fact, this proved that the boasting quad and octo-cores was the useless gimmick, as Apple's flagship processor was still a dual-core. Further, adding more cores directly impacts power performance, so most cores are actually powered off the overwhelming majority of a device's life time.

The move - a vertical, rather than a horizontal expansion, thus proved to be a brilliant one, and an especially efficacious stratagem: Though requiring virtually no work in iOS other than a recompilation of the app, porting Android to 64-bit is a lengthy process. Android's core components - notably Dalvik and Bionic - are 32-bit optimized, and therefore need to be completely rewritten. Of all vendors, Intel has been quickest to jump on the 64-bit wagon, since its mobile processors are already fully 64-bit native. The various ARM vendors, however, need to adapt to the move (though Samsung was quick to announce their "next big thing" will naturally be 64-bit). HTC's Nexus 9 was among the first 64-Bit ARM processors (Nvidia's Tegra K1), and Qualcomm soon followed with the Snapdragon 810 (HTC One M9), and Samsung with their Exynos (in the S6). QEmu (which powers the Android emulator) has finally been updated to support ARM64 emulation with the M Preview Release 1 SDK.

Android RunTime (ART)

Android still proves inferior to iOS in several aspects, not the least of which is power management. This can be traced back to its Linux foundations (which are geared towards an immobile desktop or server, where power is rarely a concern), but also due to its many layers. While layers provide for elegant abstrations, portability and other aspects of fine design, they are often dismal in terms of performance and power management, as they require more processing. The main layer in Android - Dalvik - involves significant processing, and even its many enhancements (e.g JIT compilation) still require much more work than native code would. By comparison, iOS's runtime and frameworks are implemented in Objective-C, which is an extension of standard C, and still very much native**.

* - One of the strongest rebukes of this move was made by none other than Qualcomm's senior VP and CMO, who claimed "they are doing a marketing gimmick. There's zero benefit a consumer gets from that". A week later Qualcomm retracted his comment, and he was shortly after "reassigned"[20].

** - In iOS 8, Apple has made the first moves to break away from Objective-C with the introduction of Swift, a featureful yet lightweight programming language which boasts impressive runtime performance when compiled, but also when interpreted.

The **Android RunTime** (ART) provides an alternative. Silently introduced in KitKat and dubbed "experimental", ART aims to use Ahead-of-Time (AOT) compilation, to LLVM and even native code, thus bring it on par with iOS performance. ART presently offers only small advantages in power and performance over Dalvik (on order of 10-20%), and in some tests also falls behind it. Nonetheless, as of Lollipop, ART is the chosen runtime, and is vital in order for Android to provide 64-bit support*.

As alluded to earlier in this chapter, however, Dalvik is far from dead: Applications will still be packaged with Dalvik bytecode (`classes.dex`), with ART taking over and compiling to native code only when deployed on the device (replacing the on-device optimization stage usually carried out by dexopt). Both Dalvik and ART are discussed in depth in Volume II).

Split-Screen

Android already has the necessary foundations to allow different activities to run in parallel to one another on a split screen: Samsung has extended the GUI for this capability, which is also supported in Windows 8, and with rumors abuzz for this feature to be added to the iPad in iOS 8.1, it makes sense to see it mainstream on Android. This is a purely framework-level feature, since from the native perspective there's no real change - the activities as processes run concurrently anyway. This could be a major step on the road to making Android a full desktop OS replacement, as well.

Android as a desktop OS

With so many tablets vying to become a desktop replacement, why not make Android a desktop OS? Microsoft introduced Windows 8, which took desktop Windows, and improved(?) it to support mobile devices - tablets and phones. Android would need to make the reverse transition, bringing its mobile support to desktops, which could then run Android apps.

Doing so is not necessarily that hard - as we discuss in Volume II, Dalvik's open source nature makes it quite portable, and implementations for other OSes - naturally, Linux, but also Windows, OS X and even iOS(!) exist. None of those are sponsored nor supported by Google, but with iOS and OS X edging closer and closer still to one another, some have postulated that OS X will soon run iOS apps (some have even go so far as to suggest Apple will make the transition to ARM on its Macs). If that were the case, the binding between the ecosystems would become a strong differentiator in iOS's favor, which Google will surely not ignore for long.

There are a few obstacles, however. For one, it's not trivial to support full desktop applications. The Linux OpenOffice and most other apps are already built on top of X-Windows (and GNOME or KDE), and thus would have to be adapted to Android. In addition, Android would have to be extended to support mice (though arguably its InputManager already supports cursor devices), and multiple windows (again, technically supported to an extent by the WindowManager). Last but not least is ChromeOS, which Google is developing as its answer to Windows, in the hopes of ousting the latter the same way Chrome usurped the lead to become the world's most popular browser.

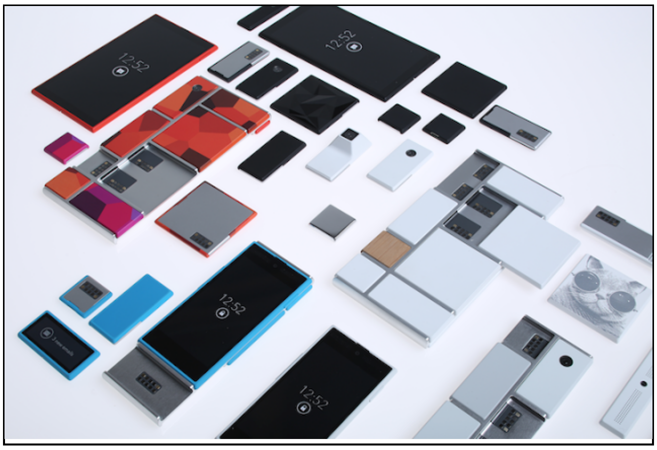
Android and Project ARA

ARA[21] is the code name for a project developed by Google with the goal of producing a fully modular smartphone. The idea is to make all system components swappable - the CPU, display, storage - are all replaceable, much in the same way in the PC world it's a farily simple matter to install a new hard drive or graphics adapter. ARA is a vestige of Google's Motorola Mobility acquisition (since sold off to Lenovo), developered by the former's Advanced Technology and Projects (ATAP) division, which was retained by Google.

* - It's important to note that Dalvik code is still 32-bit, rather than 64-bit optimized. While Dalvik does support "wide" data types, most operations are 32-bit. This means that, while compiling to native code does offer some benefits of the 64-bit architecture, the code is still not as efficient as "pure" 64 bit.

ARA makes the device, in effect, a chassis (more accurately, an endo skeleton), and components are separate modules - not unlike a PC. Electro permanent magnets (that can be turned on/off electronically, but do not require power for everyday use) hold modules in place. In theory, all components (save for the CPU, and possibly the display) are hot swappable (i.e. they can be replaced while the device is working). Coupled with 3D printing, this could lead to "printable" phone designs which could be downloaded, and an array of upgradeable modules which would render the annual full upgrades of mobile devices extinct.

Figure 1-6: Modular smartphones (from the Motorola blog)



As ARA is developed by Google, it's only natural that Android be the OS of choice for it. Supporting ARA, however, will require heavy modifications in Android - at the framework level, but even more so at the underlying Linux layers, all the way down to the kernel. Google has partnered with Linaro for these purposes, and is investing ridiculous sums of money at developing both the software and hardware necessary to standardize all the modules. ARA is still in its infancy as this book goes to print, with an estimated release (initially, in Puerto Rico) later in 2015. If successful, however, a truly modular mobile device would amount to nothing less than a second coming of the mobile revolution - and this time, Google wants to be there first.

Summary

This chapter explored the evolution of the Android architecture to the present day (KitKat), with an emphasis on its low-level features. It compared and constrasted the Android architecture with that of its parent - Linux, to show the two are in many cases not at all that far apart, though at present not interchangeable. Next, the many derivatives of Android were introduced, and though of different skins and appearance, they all, at their core, function as Android does, so you should find this work applicable to them just the same. The chapter concluded with pondering future directions for Android (L, and beyond), and the features it is likely (or not) to support.

The next chapters explore the various aspects of Android, each in as much detail as possible. We begin with the Android Filesystem - naturally based on that of Linux - but using defined partitions and filesystems (some more clearly defined than others).

References

1. Android version History, WikiPedia: http://en.wikipedia.org/wiki/Android_version_history
2. Android Dashboards (usage statistics): http://developer.android.com/about/dashboards/index.html
3. Froyo feature summary: developer.android.com/about/versions/android-2.2-highlights.html
4. Gingerbread feature summary: developer.android.com/about/versions/android-2.3-highlights.html
5. Honeycomb feature summary: developer.android.com/about/versions/android-3.0-highlights.html
6. SMP Primer for Android: http://developer.android.com/training/articles/smp.html
7. Ice Cream Sandwich feature summary: developer.android.com/about/versions/android-4.0-highlights.html
8. Jellybean feature summary: developer.android.com/about/versions/jelly-bean.html
9. Kitkat feature summary: developer.android.com/about/versions/kitkat.htmlKitkat feature summary
10. Lollipop feature summary: developer.android.com/about/versions/lollipop.html
11. Android version numbering convention: https://source.android.com/source/build-numbers.html#platform-code-names-versions-api-levels-and-ndk-releases
12. A much better Android architecural diagram: http://source.android.com/images/android_framework_details.png
13. Dan Bornstein presenting Dalvik, Google I/O 2008: https://www.youtube.com/watch?v=ptjedOZEXPM
14. Android NDK: http://developer.android.com/tools/sdk/ndk/index.html
15. Google Groups Bionic discussion: http://android-platform.googlegroups.com/attach/0f8eba5ecb95c6f4/OVERVIEW.TXT?view=1&part=4
16. Android Kernel Configuration: https://source.android.com/devices/tech/kernel.html
17. Android Wear Developer Website: http://developer.android.com/training/building-wearables.html
18. Android Auto Developer Website: http://developer.android.com/training/auto/index.html
19. Android TV Developer Website: http://developer.android.com/training/tv/index.html
20. Qualcomm reassigns exec after 64-bit criticism: http://www.cnet.com/news/after-apple-64-bit-a7-criticism-qualcomm-exec-reassigned/
21. Project ARA official Website: http://www.projectara.com

II - Android Partitions and Filesystems

This chapter begins with a discussion of the substrate for filesystems - the partitions themselves. The device's storage is broken up into disjoint chunks, each of them individually formatted and purposed. We discuss how you can analyze the partition layout, and investigate many of the partitions which are otherwise reserved and inaccessible.

We then turn our attention to the Android filesystems which the system does regularly use: `/system` (where the OS itself is installed), `/data` (where user data is stored), and others. The directory structure of these filesystems is discussed, with important files and folders pointed out.

Finally, we consider the Linux pseudo-filesystems, which - although not a part of Android per se (but of the Linux kernel), nonetheless serve important functions during system operation, primarily for diagnostics and hardware access.

Partitioning Scheme

Android users are often surprised to find that their device, stated as coming with "X GB of Flash", in practice has less than the advertised space. Power users, who go into ADB and type 'df' to add the numbers find that not only does the Android OS take up a sizeable chunk, the amounts reported under "used" and "free" simply don't add up to the stated device capacity.

Some of the difference can be explained by the fine print, which usually states that the definition of "GB" is a loose one. Rather than follow the 210 convention, wherein 1KB = 1024 bytes, 1MB = 1024KB and 1GB = 1024MB - # On device: use chmod as root to allow adb to read drive sectorswhich would define 1 GB to be 1,073,741,824 bytes, marketing pits 1GB at 1,000,000,000 bytes - already a noticeable 7% evaporated by false advertising. But the difference still leaves tens, and sometimes hundreds of MB or more unaccounted for. The missing megabytes stem from the device partitioning: most of the flash is used for Android - but some space is left for other purposes. Android flash storage is often partitioned into dozens of partitions, of which Android uses about 5. It's not uncommon to see 25 or more partitions on some devices, (such as the Kindle Fire or the Nexus 5), or even 70(!) in the case of the HTC One M9. Of those, the user can only write to one partition - /data and no other. In fact, most are not even mounted during regular use. This section discusses the partitioning scheme used, and how you can use tools to uncover those otherwise hidden partitions, which often may contain interesting content.

The exact partitioning scheme of Android may vary significantly in between vendors, and even individual devices. For the most part, however, the partitions use the same semantics, those size and number will surely vary. Most of the examples in this chapter are from Qualcomm chipset (msm) devices, which make up the vast majority of Android devices, anyway.

The need for separate partitions

Most desktop users, especially in the Windows world, are probably used to having one, or in some cases two partitions. The classic desktop view has always been that fewer partitions suffice - a view no doubt linked to the legacy MBR partitioning scheme, which by design allowed only four partitions. In UNIX, however, using multiple partitions has been much more of a norm, as it allows for greater flexibility during system upgrades, and other administrative operations. Multiple partitions do have one notable disadvantage - which is imposing artificial limitations on available space, as it is subdivided by partitioning. UN*X administrators have always found clever ways around that, however, by using symbolic links, or - when more space is needed - adding new disk space and redirecting to it, by means of mounting.

In the mobile domain, using multiple partitions makes sense as well, albeit for somewhat different reasons. One of the chief concerns of mobile devices is that they must always be repairable, and so some type of "recovery mode" must be enabled on them. To allow for system recovery or upgrades, there must be some way to boot the system from a known "safe" copy of an operating system. In fact, it is not uncommon on some devices to find multiple copies of the boot loader components - identical copies - to ensure bootability, just in case. Additionally, some components, such as the modem or other firmware components (and the bootloader itself), need their own storage space for storing configuration files or images.

Note, that not all partitions are actually mounted by Android: In fact, only a scant few often are, with the remainder either meant for use during recovery, or exclusively by system components. The latter are also unmountable by definition, because they often contain proprietary formatting, which the Linux kernel does not recognize.

The GUID Partition Table

Taking all the above considerations into account, the need for multiple partitions becomes clear, as does the realization that quite a few of them would be necessary. The MBR partition scheme is, therefore, ruled out, leaving the GUID Partition Table (GPT) as the viable option. MBR is still "technically" used, in the sense that the first sector of the device often contains a dummy MBR record, with one partition spanning the entire drive. The second sector contains the GPT header, which in turn maps out all the partitions. This is demonstrated in the following experiment:

Experiment: Obtaining the Partition Table from a Device

The GPT table is normally inaccessible from user mode, as it resides outside the partitions themselves, and therefore requires raw access to the disk device. If your device is rooted, however, you can examine it if you copy the first sectors and use `file(1)` (on your host) to analyze.

Output 2-1: Reading and identifying the GPT

```
# On device: use chmod as root to allow adb to read drive sectors
root@htc_m8wl:/ # chmod 604 /dev/block/mmcblk0
```

Because `adb` normally runs as uid:shell, there are several ways to obtain raw disk access:

1. **Re-run adb as root**: This requires setting the `ro.secure` and `ro.debuggable` properties during startup, or use a modified version of `adbd` which doesn't drop privileges.
2. **Use `dd` as root**: to copy data from the block device node to a file, which you can then place in /data/local/tmp, and `chmod` to be readable by uid shell (or anyone).
3. **Use `chmod` directly**: on the block device node, so it is readable by uid shell (and, in fact, everyone). Note this might fail in some cases on KitKat and later, depending on SELinux.

All methods carry with them a certain amount of risk: Running `adb` as root would compromise your phone if it falls into the wrong hands. Using `dd` incorrectly (e.g. confusing `if=` and `of=`) can wipe entire partitions in a heartbeat. Using `chmod` to make a device readable to anyone could ostensibly enable dormant malicious applications on your device to access data which would otherwise be well protected.

Though commonly the second method is used for handling raw devices, the last approach is the one employed in this book. Of the three, the author believes it carries with it the least risk: For one, it is perfectly reversible (and will not persist across a reboot). Additionally, it only provides read access (and therefore does not risk data corruption.

```
# On host: Grab the first sectors of the drive, and identify with file(1)
morpheus@Zephyr (~) % adb pull /dev/block/mmcblk0
^C # Hit CTRL-C before we copy all the drive image!
morpheus@Zephyr (~) % file mmcblk0
mmcblk0: x86 boot sector; partition 1: ID=0xee, starthead 0, startsector 1,
         4294967295 sectors, extended partition table (last), code offset 0x0
morpheus@Zephyr (~) % od -A x -t x1  mmcblk0 | more
# Sector 0 contains "protective MBR"
0000000    00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
*
00001c0    01  00  ee  ff  ff  ff  01  00  00  00  ff  ff  ff  ff  00  00
00001d0    00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
*
00001f0    00  00  00  00  00  00  00  00  00  00  00  00  00  00  55  aa
# Sector 1 (at 0x200 = 512 bytes) contains GPT
0000200    45  46  49  20  50  41  52  54  00  00  01  00  5c  00  00  00
#          E   F   I       P   A   R   T
0000210    71  00  18  78  00  00  00  00  01  00  00  00  00  00  00  00
0000220    ff  df  a3  03  00  00  00  00  22  00  00  00  00  00  00  00
0000230    ff  ff  9f  03  00  00  00  00  32  1b  10  98  e2  bb  f2  4b
0000240    a0  6e  2b  b3  3d  00  0c  20  02  00  00  00  00  00  00  00
0000250    30  00  00  00  80  00  00  00  b2  e5  23  34  00  00  00  00
#          48 partitions
```

A full discussion of GPT is beyond the scope of this book, and is also quite unncessary: The Linux kernel can figure out the partition map, and export it to user space via /proc - commonly, /proc/partitions. This is discussed next.

Flash Storage Systems

The storage used on Android devices is not entirely standardized - some devices use MTD (Memory Technology Devices), whereas others (e.g. HTC One) use eMMC (Embedded MultiMedia Card), and others still use MMC (MultiMedia Card). Depending on the system used, the partition map is made available to user mode via /proc/mtd, /proc/emmc (respectively), or the Linux standard /proc/partitions (often in addition to one of the former two).

For the most part, whatever system is actually used is largely transparent to most. At a high level, the chief difference between the systems is that MTD is an abstraction layer over raw flash, whereas the MMC and eMMC have their own Flash Translation Layer (FTL) and appear to the kernel as block devices. Most modern devices therefore use one of the latter two, as this makes them far more suitable for use with block based filesystems such as ext4, which has inherited YAFFS as the filesystem of choice in modern Android versions.

File Systems

Android enforces no constraints as to the filesystem types, but eMMC and MMC devices presently use the Linux Ext4 filesystem (as of Gingerbread, in place of the older YAFFS system), since the storage layer exports a block device. Ext4 has become the default filesystem in Linux as of 2.6.27, and is a well tested filesystem, albeit not a necessarily flash-optimized one.

Some devices (notably the Moto X) have begun adopting the Flash Friendly File System (F2FS) as the filesystem of choice for the data partition. As of L, this is also the default data filesystem of choice for Google's Nexi. The filesystem (designed by Samsung) is a log-structured one, optimized for NAND flash. It boasts performance improvements over Ext4, especially in random write requests. Indeed, extensive benchmarking tests posted to XDA-Developers[1] show F2FS has clear advantages over Ext4.

A good discussion of F2FS's features can be found in a Samsung presentation[2] and an article by Neil Brown on LWN.net[3]. It has been integrated into the mainline Linux kernel as of 3.8, and - as Android upgrades to a newer kernel - it is likely to be used on more devices.

Android also supports VFAT, an MS-DOS compatible filesystem which it uses for SD-Cards. Because it originated in the DOS and Windows 9x world, VFAT doesn't support the notion of permissions. Android therefore resorts to mounting the SDCard in a secondary mount using a specialized mechanism, discussed later in this chapter.

The kernel maintains a list of all supported filesystems in /proc/filesystems. This pseudo-file lists which filesystems are supported either natively (i.e. compiled into the kernel) or as a loaded module. In Android kernels, vendors often compile support for filesystems directly into the kernel, although it's quite possible to leave support for a filesystem in a module, and bundle the module as part of the root filesystem.

The good news about filesystems is that, so long as they work, the user can remain blissfully oblivious to which filesystem is in use. The bad news, however, hits when the filesystems don't work - specifically, when filesystem corruption occurs. Corruption is (thankfully) a rare event, and usually occurs when the device is improperly shut down (for example, due to power loss or an unexpected crash), or underlying hardware failure. Android provides default binaries to check and repair filesystems - `e2fsck`, `fsck_msdos` and `fsck.f2fs`, for the respective filesystems. The binaries are run automatically when mounting a file system (by `init` or `vold`).

Experiment: Examining partitions on an Android device

You can view the partition map of your Android device by examining the /proc/partitions file. This is a standard kernel /proc entry, which provides a listing of all block devices. The flash storage layer in MMC and eMMC devices is displayed in the mmcblk#[p#] form, where numbering for devices starts with zero, and for partitions with one. Block are 512-byte (½K) blocks. The "major" and "minor" refer to the device driver, with "major" being, in effect, the index used by the driver in the kernel's block device table, and "minor" being the index of the logical device (in this case, used to disambiguate the partitions from one another).

Output 2-3: /proc/partitions from a Nexus 5

```
shell@nexus5$ cat /proc/partitions
major     minor  #blocks  name
 179        0   15388672 mmcblk0      # Entire Flash disk size
 179        1      65536              # 1st Partition
 ...
 179       29          5              # 29th Partition
 179       32       4096 mmcblk0rpmb  # Resource Power Management backup
```

Telling apart the partitions just by their cryptic name is hard, but thankfully most devices have symbolic links, by-num and by-name, in the /dev/block/platform/*name.#* directory. The platform name refers to the controller, in Qualcomm's case msm_sdcc.1 for the main storage:

Output 2-4: //dev/block/platform/.../by-name from a Nexus 5

```
shell@nexus5$ ls -l /dev/block/platform/msm_sdcc.1/by-name| cut -c56-
DDR -> /dev/block/mmcblk0p24
aboot -> /dev/block/mmcblk0p6      # Application Boot loader (Android Boot)
abootb -> /dev/block/mmcblk0p11    # Backup of Application Boot Loader
boot -> /dev/block/mmcblk0p19      # Kernel + InitRAMFS used to boot system
cache -> /dev/block/mmcblk0p27     # Mounted as /cache (used for updates/recovery)
crypto -> /dev/block/mmcblk0p26
fsc -> /dev/block/mmcblk0p22
fsg -> /dev/block/mmcblk0p21
grow -> /dev/block/mmcblk0p29      # usually empty
imgdata -> /dev/block/mmcblk0p17   # Boot loader graphic images (imgdata format)
laf -> /dev/block/mmcblk0p18       # LG Advanced Flash Daemon
metadata -> /dev/block/mmcblk0p14  # usually empty
misc -> /dev/block/mmcblk0p15      # Reserved for system->Boot Loader communication
modem -> /dev/block/mmcblk0p1      #
modemst1 -> /dev/block/mmcblk0p12  # Modem (Radio) boot
modemst2 -> /dev/block/mmcblk0p13  #
pad -> /dev/block/mmcblk0p7        # usually empty
persist -> /dev/block/mmcblk0p16   # Persistent settings for system components
recovery -> /dev/block/mmcblk0p20  # Alternate kernel + InitRAMFS for recovery
rpm -> /dev/block/mmcblk0p3        # Resource/Power Management loader
rpmb -> /dev/block/mmcblk0p10      # Backup of Resource/Power Management loader
sbl1 -> /dev/block/mmcblk0p2       # Secondary Boot Loader
sbl1b -> /dev/block/mmcblk0p8      # Backup of Secondary Boot Loader
sdi -> /dev/block/mmcblk0p5
ssd -> /dev/block/mmcblk0p23
system -> /dev/block/mmcblk0p25    # mounted as /system
tz -> /dev/block/mmcblk0p4         # ARM TrustZone
tzb -> /dev/block/mmcblk0p9        # Backup of ARM TrustZone
userdata -> /dev/block/mmcblk0p28  # mounted as /data
```

Note, that your local Android device partition names can and likely will vary. While most MSM devices generally adhere to the above conventions, NVidia based devices deviate from it (with nigh-incomprehensible three letter abbreviations for partition names), as do OMAP-based ones.

Android Device Partitions

As the previous experiment has shown, partitions on Android devices have set names, but most of them are quick cryptic. To complicate matters, different device chipsets and vendors use different partitions, as well as different names for the same functional partitions. To explicate, we break the partitions into the following classes:

Standard Android Partitions

All devices find a common denominator in those partitions which are hard coded into Android itself, in various locations around the source tree. These partitions comprise the core of the OS.

The standard partitions are mostly mountable, with the exception of the boot and recovery partitions, which are (commonly) formatted with Android's proprietary bootimg format (explained in the next chapter). Table 2-1 shows these partitions:

Table 2-1: Android Standard Partitions

Name	format	Notes
boot	bootimg	Kernel + initramfs. Contains kernel and RAMdisk to boot by default.
cache	Ext4	Android's `/cache`: used for updates and recovery.
recovery	bootimg	Boot-to-recovery: Kernel + alternate initramfs to start system recovery.
system	Ext4	Android's `/system` partition - OS Binaries and frameworks.
userdata	Ext4/F2FS	Android's `/data` partition - User data and configuration.

Android devices contain a file system mounting table. This table, in `/system/etc/vold.fstab` or (in more recent versions of Android) /fstab.*hardware*, is loaded by the Volume Daemon (`vold`) during system startup, and provides the partitions which are to be mounted automatically (much like the classic UN*X `/etc/fstab`.

Chipset-specific Partitions

Chipset vendors often require dedicated partitions for their components. The most notable example is Qualcomm, whose MSM chipset (arguably the most popular) uses the partitions shown in table 2-2. The bootldr format is discussed in the next chapter.

Table 2-2: Partitions found on Qualcomm MSM devices

Name	Format	Notes
aboot	bootldr	Application Processor Boot:This contains the Android Boot Loader. Note some devices may use custom boot loaders instead (e.g. HTC's HBoot).
modem	MSDOS	Contains various ELF binaries and data files to support device modem
modemst[1|2]	proprietary	Non-Volatile data for modem
rpm	ELF 32-bit	Resource Power Management: This provides the first stage bootloader
sbl[123]	Proprietary	Secondary Boot Loader optionally split into up to three stages.
tz	ELF 32-bit	ARM TrustZone

Vendor-specific Partitions

The rest of the partitions found on an Android device are specific to vendors, who use custom partitions for their own purposes, mostly device configuration maintenance or upgrade operations. The formatting used is often proprietary. A partial list of such partitions is shown in table 2-3:

Table 2-3: Vendor custom partitions

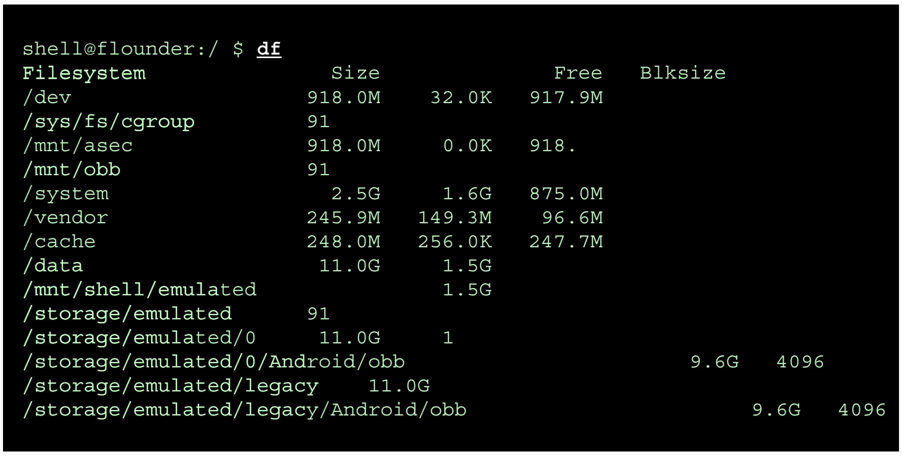
Name	Vendor	Notes
hboot	HTC	HTC's proprietary boot loader. Replaces aboot on HTC devices
efs	Samsung	Encrypted File System. Contains various configuration files
ssd	Samsung	Secure Software Download
ota,fota	Samsung	Firmware-Over-The-Air: Used in the process of phone updates
grow	Samsung, LG	Empty partition, to allow partition growth
laf	LG (G2, Nexus5)	Contains an alternate bootimg which loads lafd (LG Advanced Flash Daemon) used for device re-flashing. The laf partition is a recovery image format.
imgdata	LG (G-PAD, G2, Nexus5)	RLE images in IMGDATA format, similar to BOOTLDR

The XDA-Developers forum maintains an on-going list of partition maps from various devices (similar to the one in the previous experiment) in its El Grande Partition Table Reference[4].

Experiment: Viewing mounted partitions on a Device

You can examine the mount points by using `df` or `mount`. The former provides disk usage statistics for the mounted partitions. This is shown in listing 2-dfmount, which shows the output of the command on a Nexus 9, with L:

Output 2-5: Demonstrating `df` on a Nexus 9



```
shell@flounder:/ $ df
Filesystem               Size              Free   Blksize
/dev                   918.0M    32.0K   917.9M
/sys/fs/cgroup         91
/mnt/asec              918.0M     0.0K   918.
/mnt/obb               91
/system                  2.5G     1.6G   875.0M
/vendor                245.9M   149.3M    96.6M
/cache                 248.0M   256.0K   247.7M
/data                   11.0G     1.5G
/mnt/shell/emulated               1.5G
/storage/emulated      91
/storage/emulated/0     11.0G     1
/storage/emulated/0/Android/obb                       9.6G   4096
/storage/emulated/legacy    11.0G
/storage/emulated/legacy/Android/obb                       9.6G   4096
```

Note that the `df` output differs from the traditional Linux `df`, since the command is implemented as a `toolbox` tool. Only filesystems mounted over actual storage (that is, "real" devices) are shown.

By comparison, using `mount` provides far more verbose information, as it provides information on pseudo file systems - implemented in memory - as well as mount options, though at the cost of reporting space. The mount options displayed may either be generic, or filesystem specific. Table 2-4 provides an explanation on the options you are likely to encounter.

Experiment: Viewing mounted partitions on a Device (cont.)

Table 2-4: Commonly encountered mount options

Option	Kind	Specifies
ro rw	Generic	Read-Only: Allows only read operations, and cannot be modified. Read-Write: Allows both read and write operations
acl	Generic	Allow extended Access Control Lists, for finer control than user/group/other
seclabel	Generic	SELinux labels are enabled on the filesystem
nosuid	Generic	The filesystem does not allow SetUID binaries
noatime	Generic	File operations do not record access time - only modification or creation. This makes file access faster and reduces write activity
relatime	Generic	Update access time relative to creation/modification time
data=	Ext3/4	ordered: commit data to filesystem before writing journal. journal: commit data to journal before filesystems.
errors=	Ext3/4	continue: silently ignore errors remount-ro: remount filesystem as read only if errors are encountered panic: Crash the system
background_gc	f2fs	Reclaim space freed by deleted files in a kernel thread

The `mount` command is shown in the following output. The pseudo-filesystems are in *italics*, and can be recognized by the fact they are not mounted on a device (that is, the first column does not start with a `/dev` path.

Output 2-6: Demonstrating `mount` on a Nexus 9

```
shell@flounder:/ $ mount
rootfs / rootfs ro,seclabel,relatime 0 0
tmpfs /dev tmpfs rw,seclabel,nosuid,relatime,mode=755 0 0
devpts /dev/pts devpts rw,seclabel,relatime,mode=600 0 0
none /dev/cpuctl cgroup rw,relatime,cpu 0 0
adb /dev/usb-ffs/adb functionfs rw,relatime 0 0
proc /proc proc rw,relatime 0 0
sysfs /sys sysfs rw,seclabel,relatime 0 0
selinuxfs /sys/fs/selinux selinuxfs rw,relatime 0 0
none /sys/fs/cgroup tmpfs rw,seclabel,relatime,mode=750,gid=1000 0 0
pstore /sys/fs/pstore pstore rw,relatime 0 0
/sys/kernel/debug /sys/kernel/debug debugfs rw,relatime,mode=755 0 0
none /acct cgroup rw,relatime,cpuacct 0 0
tmpfs /mnt/asec tmpfs rw,seclabel,relatime,mode=755,gid=1000 0 0
tmpfs /mnt/obb tmpfs rw,seclabel,relatime,mode=755,gid=1000 0 0
/dev/block/platform/sdhci-tegra.3/by-name/APP /system ext4 ro,seclabel,relatime,data=ordered 0 0
/dev/block/platform/sdhci-tegra.3/by-name/VNR /vendor ext4 ro,seclabel,relatime,data=ordered 0 0
/dev/block/platform/sdhci-tegra.3/by-name/CAC /cache ext4 rw,seclabel,nosuid,nodev,noatime,
 errors=panic,data=ordered 0 0
/dev/block/dm-0 /data f2fs rw,seclabel,nosuid,nodev,noatime,background_gc=on,user_xattr,acl,
 errors=panic,active_logs=6 0 0
/dev/fuse /mnt/shell/emulated fuse rw,nosuid,nodev,relatime,user_id=1023,group_id=1023,
 default_permissions,allow_other 0 0
tmpfs /storage/emulated tmpfs rw,seclabel,nosuid,nodev,relatime,mode=050,gid=1028 0 0
/dev/fuse /storage/emulated/0 fuse rw,nosuid,nodev,relatime,user_id=1023,group_id=1023,
 default_permissions,allow_other 0 0
/dev/fuse /storage/emulated/0/Android/obb fuse rw,nosuid,nodev,relatime,user_id=1023,
 group_id=1023,default_permissions,allow_other 0 0
/dev/fuse /storage/emulated/legacy fuse rw,nosuid,nodev,relatime,user_id=1023,group_id=1023,
 default_permissions,allow_other 0 0
/dev/fuse /storage/emulated/legacy/Android/obb fuse rw,nosuid,nodev,relatime,user_id=1023,
 group_id=1023,default_permissions,allow_other 0 0
```

Android Filesystem Contents

Irrespective of vendor, The Android standard partitions have a well defined filesystem layout. Device vendors use the Android filesystem images provided by Google for the Android emulator as a point of departure, but the journey is often a very short one. We next describe the contents of the various filesystems starting at the root, and progressing by mount point.

Note, that while the filesystems are largely the same across devices, your device's contents can and will vary: many vendors, including Google, drop additional device specific binaries. These are referred to as **proprietary blobs**, and you can see those dropped into /system in the device/ subdirectory of the AOSP, in files called proprietary-blobs.txt per device.

The root file system

Android's root file system is mounted from a RAM Disk (the "initramfs"). Upon every boot, the boot loader (fastboot) loads the filesystem image from the boot partition onto RAM, and provides it for the kernel. The process is detailed in the next chapter, but for the purpose of the present discussion, the salient point is that the root filesystem image cannot be easily modified, unless the device is "flashed". This is important, because the root filesystem contains the most important component of the system - /init, which runs unfettered as root, and controls the system startup.

Vanilla Linux normally uses the initramfs to supply drivers (in the form of kernel modules) to the kernel during the initial boot, and eventually discards it in favor of the real filesystem. Android, however does not - Android's initramfs remains resident and provides the root filesystem functionality, which is in practice limited to housing /init and several other configuration files and binaries. These are shown in table 2-6:

Table 2-6: The contents of the Android root filesystem (excluding mount points)

Directory	Notes
default.prop	"Additional default property" file, sourced by init to load system-wide properties. Loads read-only properties which help enforce security
file_contexts	Kitkat: File contexts for SE-Linux. Restrict access to system files and directories. Described in Chapter 8
fstab.*hardware*	The filesystem mount table used by fs_mgr and vold (described in Chapters 4 and 5)
init	The binary launched by the kernel on startup as PID 1. Described in Chapter 4.
init[...].rc	The configuration file(s) for /init. The main configuration file is always /init.rc, with optional additional files which are device and vendor dependent. Likewise described in Chapter 4.
property_contexts	Kitkat: Property contexts for SE-Linux. Restrict access to system properties. Described in Chapter 8
seapp_contexts	Kitkat: Application contexts for SE-Linux. Restrict application operational scope. Described in Chapter 8
sepolicy	Kitkat: The compiled SELinux policy (q.v. Chapter 8)
sbin/	Contains critical binaries, such as **adbd**, **healthd** and (most importantly) **recovery**, which the system needs even if /system cannot be mounted. May also contain vendor binaries.
verity_key	L: Contains the DM-Verity RSA key required to authenticate the /system partition.

/system

The system partition is the home of all Android components, as provided by Google and/or the vendor. The directory and its contents are owned by root:root, and all have permissions of 0755 (rwxr-xr-x), but the filesystem is mounted read-only. A read-only mount makes sense for two reasons:

- **Stability:** Because the filesystem is mounted read-only, there is virtually no chance of it being corrupted, even if the device is powered-off abruptly. This reduces the chance of an error which might "brick" the device by preventing Android from booting.
- **Security:** A read-only mount is another layer of defence to protect the Android system components from being tampered with. In practice, though, it is trivial to remount the partition as read-write.

Some vendors, notably HTC, also ensure /system is read-only using flash partition protections (HTC calls this S-OFF). This means that even if /system is mounted read-write, any changes to it will not be made persistent. As of KitKat, Google offers integrity checking for /system, using the Linux kernel's **dm-verity** feature (discussed in Chapter 8).

The /system partition is (for the most part) the same on most devices. In a perfect world it would be exactly identical, though in practice vendors and carriers sometimes add their own apps and (rarely) directories, rather than in /vendor, which is the location designed for that. Table 2-7 shows the contents of the the /system partition you can expect on any Android device:

Table 2-7: The contents of the /system partition

Directory	Notes
app	System applications: These include the prebundled apps from Google, as well as any vendor or carrier-installed apps (though these should technically reside in /vendor/app, instead).
bin	Binaries: These include the various daemons, as well as shell commands (mostly links to toolbox, or - as of M - toybox).
build.prop	Properties generated as part of the build process. This file is sourced by init to load properties on boot
etc	Miscellaneous configuration files. Symlinked from /etc. q.v. Table fs-etc for contents.
fonts	True-Type Font (.ttf) files
framework	The Android frameworks. Frameworks are contained in .jar files, with their executable dex files optimized alongside them in .odex.
lib	Runtime libraries - native ELF shared object (.so) files. This directory serves the same role as /lib in vanilla Linux.
lost+found	Automatically generated directory for fsck operations on /system. Empty (unless the filesystem crashed, in which case it may contain unlinked inodes)
media	Alarm, notification, ringtone and UI-effect audio files in .ogg format, and the system boot animation (discussed in Chapter 5).
priv-app	Privileged Applications
usr	Support files, such as unicode mappings (icudt511.dat), key layout files for keyboards and devices, etc.
vendor	Vendor specific files, if any. Usually placed into subdirectories mirroring /system itself (e.g. bin/, lib/, and media/).
xbin	Special purpose binaries, not needed for normal operation (unlike those in bin. On the emulator, this is populated with various tools from the AOSPs /system/extras. On devices, this directory is normally empty, or contains only dexdump. Various rooting utilities drop "su" there as well.

/system/bin

The /system/bin directory contains the native executables used by the Android, as well as a host of debugging tools. Specifically, these binaries can be classified into the five categories:

- **Service binaries:** Invoked by /init throughout the lifetime of the system. These binaries are referenced in the rc files used by /init, and are required for system operation. Not all of these are directly from the AOSP: Those marked in yellow are external projects.

Table 2-8: Service binaries in /system/bin

Binary	Function
app_process[32/64]	Host process for Apps. Zygote (and all user apps) are instances of this binary, which initializes the DalvikVM/ART. On 64-bit devices both 32/64 are present.
applypatch[_static]	Used during OTA updates - applies patches according to scripts, as discussed in Chapter 3. The _static binary is a statically linked version, used for updates that would modify the dependencies of the normal (dynamically linked) binary.
bootanimation	Plays Android boot animation, while graphics subsystem (surfaceflinger) is loading. Often customized by vendor.
clatd	IPv4-to-IPv6 address translation
dalvikvm	Starts an instance of the Dalvik Virtual Machine
debuggerd	Generates tombstones from process crashes, optionally connects to a remote GDB
drmserver	Host process for 3rd party Digital Rights Management (DRM) modules
dnsmasq	DNS Masquerade: Provides DNS proxying services when device is providing tethering over Wi-Fi.
gatekeeper	M: New daemon to handle authentication
hostapd	Host Access Point Daemon: Provides access point emulation when device is providing tethering over Wi-Fi.
keystore	Android's key storage and management service
linker	Android's runtime linker. Not a service per se, but required for binary loading. Messing with this is a surefire way to brick your device.
mdnsd	Multicast DNS Daemon. Used for neighbor discovery over Wi-Fi Direct
mediaserver	Audio/Video Recording/Playback
mtpd	PPP/L2TP support
netd	Manages network interfaces, firewalling and more.
pppd	Point-to-Point Protocol Daemon. Required for VPNs
racoon	Provides VPN support
rild	Radio Interface Layer Daemon. In charge of all telephony services
sdcard	SDCard daemon. Manages SD-Cards so as to emulate multiple users via FUSE (discussed later in this chapter).
sensorservice	Sensor hub: coordiates reading from various sensors
servicemanager	Service locator and fulcrum for all binder related services.
surfaceflinger	Composes graphics surfaces and loads them onto the framebuffer
vold	Volume Daemon: Mounts/unmounts filesystems, and optionally decrypts.
uncrypt	Decrypts filesystem (for use before recovery)
wpa_supplicant	Wireless Protected Access Supplicant: Provides client support for Wi-Fi and Wi-Fi P2P.

Services are described in great detail in Chapter 5, in the context of their startup by /init.

- **Debugging tools:** These are native binaries left for debugging. The following list shows those found in the emulator, though vendors (at their discretion) may decide to omit them from production devices:

Table 2-9: Debugging tools in /system/bin

Binary	Function
adb	Android Debugger Bridge (client) - this is essentially the same binary as the host `adb` (the server portion is in /sbin/adbd).
asanwrapper	Address Sanitizer - Memory corruption detecting tool. 3rd party debugging tool
atrace	Android tracing tool - uses Linux ftrace to debug and trace execution.
bdt	BlueDroid (Bluetooth for Android) test app.
blkid	Displays GUIDs of partitions
cjpeg	JPG processing tool
dex2oat	DEX to ART conversion tool. Compiles the DEX file to device executable format. Supersedes dexopt.
dexopt	DEX optimization tool. Creates device optimized DEX files (deprecated when using ART)
dumpstate	Meta-tool combining several useful utilities (ps, dumpsys, etc) for capturing a debug snapshot of system state.
dumpsys	Service dump utility: Connects to Android services and requests their Dump() method, providing a plethora of debugging information.
e2fsck fsck_msdos fsck.f2fs	Ext2/3/4, VFAT and F2FS filesystem checkers. Run automatically by the system before mounting filesystems.
gdbserver	GDB server tool. Used to connect GDB over TCP/IP from host in order to debug processes. Omitted from most devices.
ip[6]tables	Manage the kernel IPTables (firewall and network quota) from the command line.
keystore_cli	Command line utility for interfacing with the keystore service
logcat	print the system logs (/dev/log/*) to stdout, with optional filters. This command is so useful that it can be used directly as `adb logcat`.
ndc	Command line utility for interfacing with the Network Management Daemon (netd)
perf	Extremely powerful profiling tool which uses the kernel's profiling support.
ping[6]	Packet Internet Grouper (ICMP echo request/reply)
radiooptions	Test utility for simulating Radio Interface Layer (RIL) events.
run-as	Run an application under specific AID.
screencap	Capture framebuffer to stdout or to a PNG file (used by ADB)
screenrecord	Record movie (as .mp4) of device display
screenshot	As screencap, with optional sound to play on screen shot.
service	Command line utility for interfacing with the servicemanager.
toolbox	Android's multi-call binary, as discussed above
vdc	Command line utility for interfacing with the Volume Daemon (vold)
wpa_cli	Command line utility for interfacing with wpa_supplicant

- **UN*X commands:** left as a convenience for the shell user. The UN*X commands are packaged into a single binary - /system/bin/toolbox (or, as of M, toybox). Either box is an Android specific version of the busybox binary, which is an all-in-one-tool common in embedded systems*. Rather than providing every single UN*X command, the *box commands contain basic implementations of those commands, and can emulate the commands based on their argument (e.g. "toolbox ls") or when invoked via a symbolic link (i.e. "ln -s /system/bin/toolbox /system/bin/ls"). The toolbox and toybox provide a reduced subset of the busybox commands*, including several Android specific commands (e.g. getprop/setprop/watchprop, start/stop and log) .

- **Dalvik upcall scripts:** allow the shell user to interact with the Dalvik runtime frameworks, mostly for debugging. All these scripts (with the exception of uiautomator) are cut-paste from the same template, which calls on /system/bin/app_process to load the Dalvik class* from its containing framework JAR, and directly passes any arguments to it. To see the template, it suffices to look at the "am" script, presented in Listing 2-1:

Listing 2-1: The script template for the Dalvik upcalls

```
#!/system/bin/sh
#
# Script to start "am" on the device, which has a very rudimentary
# shell.
#
base=/system
export CLASSPATH=$base/framework/am.jar
exec app_process $base/bin com.android.commands.am.Am "$@"
```

Table 2-10 shows the scripts and their purpose. Invocation with no arguments will yield a usage message.

Table 2-10: The app_process wrapper scripts in /system/bin

script	Usage
am	Interact with ActivityManager. Start activities, fire intents and much more.
appwidget	L: Grant/Revoke User Application Widgets
bmgr	Backup Manager Interface
bu	Start backup
content	Interface to Android's content providers
dpm	M : Device Admin/Profile Management
ime	Control Input-Method-Editors
input	Interact with InputManager, inject input events (discussed in Volume II).
media	Control the current media client (play/pause/etc)
monkey	Run an APK with randomly generated input events
pm	Interact with PackageManager, list/install/remove packages, list permissions, etc.
requestsync	Sync accounts
settings	Get/set system settings
sm	M: Storage Management
svc	Control the power , data, wifi and USB services
uiautomator	Performs UI Automation tests, dumps view hierarchy, etc.
wm	Interact with WindowManager, change display size/density, etc.

* - It's a good idea to install busybox on Android, as most custom ROMs do - busybox contains far more tools than toolbox does, making it indispensable for the power user. M's toybox is far better as well, providing (at last) more!

** - Starting the Dalvik VM by means of app_process from the shell, rather than forking off of Zygote (which is itself an instance of app_process) is considerably slower, as you can see for yourself by running any of the above scripts.

- **Vendor specific binaries:** These naturally vary with vendor, but are generally either services or debugging tools. Qualcomm provides a set of binaries which are common to msm based devices, including the following:

Table 2-11: Qualcomm specific binaries in `/system/bin`

binary	Function
mm-qcamera-daemon	Qualcomm built-in camera daemon.
mpdecision	Multi-Processor Decision: Proprietary tool to manage CPU frequency. Interacts with CPU governor to increase frequency and/or activate cores when system is busy, and decrease frequency and/or shutdown cores when system is idle.
qmuxd	Qualcomm baseband access multiplexer
qseecomd	Qualcomm Secure Execution Environment Communicator
thermal-engine-hh	Thermal Daemon, responsible for monitoring device temperature and preventing overheating

In addition to the chipset vendor (Qualcomm, NVidia, OMAP, etc), there may or may not be other binaries, provided by the handset vendor (HTC, Samsung, etc). As you'll see in a bit, these non-AOSP binaries can and should be placed elsewhere - specifically, `/vendor`. But whether or not to observe this convention is left up for the vendors to decide.

Vendor specific binaries are commonly closed source - and regrettably so: In many cases, these binaries can profoundly impact system performance or security, and may contain exploitable vulnerabilities. Qualcomm's `qseecomd` is a prime example of such a case, as is HTC's `dmagent` (which gave rise to the WeakSauce exploit, as detailed on the book's companion website)[5].

/system/xbin

The /system/xbin directory is akin to Unix's /sbin in the sense that it contains binaries which administrators find useful, but normal users are probably better off staying away from. The "x" was chosen rather than an "s" to avoid confusion with Android's own /sbin, which is part of the root filesystem, and contains binaries critical to system operation.

Binaries in this directory are compiled from the AOSP's system/extras directory. Because this directory is not strictly required for normal operation, it is left to the vendor's discretion as to what to populate it with. Some vendors, in fact, choose not to populate it at all, or leave only dexdump. Table 2-12 shows the contents of the extra tools found in the emulator.

Table 2-12: AOSP binaries found in the /system/xbin directory on the emulator

binary	Function
add-property-tag	Add properties to a system .prop file
check-lost+found	Check lost+found directory after a fsck operation
cpueater	Tight loop to consume 100% cpu
cpustats	Display CPU and governor (frequency controller) statistics
daemonize	Turn an executable into a daemon by running in background and closing stdin/stdout/stderr
dexdump	DEX file dumping tool. Provides header and bytecode dump
directiotest	Test I/O over block devices
iperf3	Handy profiling and benchmarking utility
kexecload	Overwrite kernel image with new kernel using kexec system call
ksminfo	Kernel Same-page Merger information. KSM saves RAM by detecting (via a hash) duplicate virtual memory pages, and keeping only one physical page.
latencytop	Displays data from /proc/sys/kernel/latencytop in a more readable form
librank	Display VSS/RSS/PSS/USS by shared memory region
memtrack	Tracks process memory utilization (via /proc/*pid*/smaps)
micro_bench	Memory benchmarking tool
nc	Netcat, the swiss army knife of TCP and UDP.
netperf netserver	Network performance tool (client and server)
perfprofd	M: perf profiling daemon. Collects data to /data/misc/perfprofd
procmem	Show process memory statistics (from /proc/*pid*/status)
procrank	Complement to librank, providing VSS/RSS/PSS/USS usage statistics, but by process.
rawbu	Low-level backup/restore of /data
sane_schedstat	A human-readable form of scheduler statistics
showmap	Display process memory map (/proc/*pid*/maps)
showslab	Display kernel slab allocator information (/proc/slabinfo)
sqlite3	SQLite3 command line tool. Becuase so many content providers in Android are backed by SQLite3, this is an indispensable tool for debugging and forensics.
strace	System call tracer, using the Linux ptrace(2) system call. Unbelieveably powerful tool for tracing and reverse engineering.
su	Switch user (to root or other)
taskstats	Provides detailed statistics using Linux's taskstats interface (if kernel supports it)
tcpdump	Packet capture tool. Capture files can then be opened with Wireshark.
timeinfo	Print realtime, uptime, awake percentage, and sleep percentage

The precompiled binaries are exceptionally useful as debugging tools on a real device. Moving them is a straightforward matter, as simple as using `adb pull /system/xbin` into a directory on the host, and then using `adb push .. /system/xbin` to the device (assuming a writable /system). Nearly all binaries, however - both those in /system/xbin and /system/bin - require shared libraries to work correctly. These, found in /system/lib, are discussed next.

/system/lib[64]

The /system/lib (and, on 64-bit devices, /system/lib64) directory contains the shared libraries used by the binaries in /system/bin and /system/xbin (The previous chapter provided a cursory glance at the various libraries). In most devices, /system/lib has several subdirectories. While some of these are device dependent, common ones include:

- drm/ (Providing DRM engines, such as libfwdlockengine.so, for forward-locking)
- egl/ (For Android's OpenGLES implementation, discussed in Volume II)
- hw/ (containing HAL modules, as discussed in the the previous chapter)
- ssl/engines (containing libkeystore.so, which allows OpenSSL integration with Android Keystore mechanism)

On Intel devices, /system/lib normally contains an additional subdirectory called arm/, which contains copies of the same libraries, compiled for the ARM architecture. These are used by Intel's binary translation layer - Houdini - to provide a full environment for any ARM binaries (commonly in APKs which contain native libraries).

Nexus devices contain even more subdirectories, containing JNI libraries for various Google services, such as Chrome/, Drive/, Wallet/, and others.

Nearly all of Android's binaries are dynamically linked. An exception to the rule are the binaries in/sbin, which (following the traditional UN*X model) are meant to be used in instances wherein /system (and therefore /system/lib) is not mounted. The following experiment demonstrates how you can find precisely which libraries a given binary requires.

Experiment: Displaying the dependent binaries for a given library, or vice versa

One tool sorely lacking from the Android NDK is `ldd(1)`, which is used in Linux to show loader dependencies. The Linux version of ldd actually simulates the loading of a binary, which is why it fails when processing a binary on a machine of a different architecture. The `deps` tool, found on the book's companion website, will enable you to display the dependencies of an executable, similar to ldd(1), but also scan all executables in a given path to see which depend on a given library.

As hinted previously, the tool can be quite useful when moving binaries between devices, or from the emulator to the device. Many of the binaries in the emulator's /system/xbin are invaluable for debugging and tracing on a real device. It's a fairly simple matter to move them from the emulator to device (when both use the same version of Android), provided all the dependencies are met as well. For example, `procrank` and `pagerank`, depend on `libpagemap.so`. Using the `deps` tool will show you this:

Output 2-7: Using the `deps` utility

```
root@Generic:/ # deps -tree /system/xbin/procrank
procrank
+--libc.so ---- libdl.so
+--libm.so ---- libc.so
+--libpagemap.so -+--libc.so
                  +--libm.so
+--libstdc++.so ----libc.so
```

/system/etc

Much like its UN*X namesake, Android's /system/etc contains miscellaneous configuration files - *et cetera*. The directory is also symbolically linked to from /etc, to maintain compatibility with external projects in AOSP, which expect to find their configuration there. Table 2-13 shows the contents commonly found in this directory:

Table 2-13: Files and directories commonly found in /system/etc

Name	Description
NOTICE.html.gz	Legal notices for the myriad open source components of Android, required for various obscure licenses and legal reasons. As these aren't read frequently (or ever..) they are put into one hyperlinked file and gzipped.
audio_effects.conf audio_policy.conf	Used by the Android audio HAL (described in Volume II)
apns-conf.xml	Telephony provider configuration file, listing carriers supported by device (used by `com.android.providers.telephony.TelephonyProvider`)
asound.conf	On some devices, the Advanced Linux Sound Architecture (ALSA) configuration file for the device
bluetooth/	The BlueDroid configuration files
clatd.conf	Configuration file for CLATd (handles IPv4 over IPv6)
event-log-tags	Log tags for various Android system components (used by `android.util.EventLog`)
fallback_fonts.xml	List of fallback fonts to load for families not specified in system_fonts.xml. Used by Android's layoutlib's FontLoader.
gps.conf	GPS configuration file
hosts	Hosts map, containing localhost (127.0.0.1) for compatibility
media_codecs.xml	StageFright's codec list (q.v. Volume II).
media_profiles.xml	LibMedia's profile list (q.v. Volume II).
ppp/	Contains binaries for starting/stopping VPN and PPP connectivity
permissions/	XML files containing permissions for built-in apps (AOSP's and Vendor's). Used by the PackageManager.
security/	Directory containing the device's hard coded certificate authorities (cacerts/), OTA update certificates (otacerts.zip) and SELinux labels for signed APKs. Detailed in Chapter 8.
system_fonts.xml	List of system fonts, organized by families and namesets, mapping font styles to TTF files in /system/fonts. Used by Android's layoutlib's FontLoader.
wifi/	Configuration directory for WPA supplicant, controlling Wi-Fi and Wi-Fi P2P Connectivity (see Volume II)

Depending on the device vendor (and, in particular, the chipset provider), /system/etc may hold any number of additional files. Table 2-14 shows some files commonly found on Qualcomm devices with the MSM chipset:

Table 2-14: Files in /system/etc on Qualcomm (MSM) devices

Name	Description
*.acdb	Miscellaneous Audio Calibration DataBase files, used by libacdbloader.so on Qualcomm devices
snd_msm/	ALSA files for Qualcomm MSM SoC sound device
thermal*.conf	Configuration file for the thermald daemon, which monitors device temperature

/data

The /data partition is where all the user's personal data resides. Providing a separate partition for this provides several important advantages:

- /data is decoupled from the underlying Android OS version: System upgrade and recovery can thus wipe and rewrite the entire /system partition, without affecting the user's data in any way. Conversely, the device can quickly be reset and all personal data wiped by formatting /data, which is exactly what happens during a "factory reset"*.
- /data may be encrypted, if the user requires it: Encryption, however efficient, adds a degree of latency, since reading and writing involves decryption and encryption, respectively. Because, by design, /system contains no sensitive information, there is no need to encrypt it, and therefore this latency is avoided.
- /data may also be made non-executable (i.e. mounted with the noexec option, or enforced with SELinux). As of KitKat, this isn't a default option. Doing so, however, would not only would make it more true to its name, but would greatly mitigating an attack vector for malware, since the latter would have no writable partition that it can drop executables to. This would not affect legitimate Dalvik/ART apps, because DEX and OAT run in a virtual machine, but would likely impact rooting (for example, by requiring a remount, the same as it does with /system).

The /data partition is mounted with nosuid, which makes rooting the device a bit more of a cumbersome operation - assuming that root access is somehow obtained, the su binary (which makes for an efficient, persistent backdoor) must be placed in /system, which is read-only. In practice, this is only a minor obstacle, since it's a simple enough operation to remount /system in read-write mode. Nonetheless, this is an example of defense-in-depth, and could actually prove effective when /system is cryptographically hashed, as with KitKat's dm-verity (q.v. Chapter 8).

Table 2-15 shows the contents of the /data partition. Note vendors and carriers may place additional files or directories.

Table 2-15: Directories under the /data partition

Directory	Notes
anr	Used by dumpstate to record stack traces of non-responsive Android Apps. Stack traces are recorded into traces.txt, as per the dalvik.vm.stack-trace-file property.
app	User-installed applications. Downloaded .apk files can be found here.
app-asec	Application asec containers (described later in this chapter).
app-lib	JNI libraries of applications (both system and user-installed) can be found here.
app-private	Provided for application private storage; In practice largely unused, since asec provides better security.
backup	Used by the backup service
bugreports	Used exclusively by bugreport for generated reports, which include a text file and screenshot (png), both named bugreport-*yyyy-mm-dd-hh-mm-ss*.
dalvik-cache	The optimized classes.dex of system and user applications. Each app's dex is preceded by the path to its apk, with "@" replacing the path separator (e.g. system@framework@bu.jar@classes.dex).
data	Data directories for installed applications, in reverse DNS format. Discussed next
dontpanic	Formerly used to store Android panic console and threads. Unused.
drm	Used by Android's Digital Rights Management
local	A readable/writable temporary directory for uid shell (usable in ADB sessions)

* - It should be noted that the quick formatting of /data is not secure, and data can still be recovered from the device. This has been the subject of several studies (q.v. http://www.cl.cam.ac.uk/~rja14/Papers/fr_most15.pdf), but the concern has been largely mitigated with the advent of /data partition encryption with dm-crypt (as discussed elsewhere in this book)

Table 2-15 (cont): Directories under the `/data` partition

Directory	Notes
lost+found	Automatically generated directory for fsck operations on `/data`. Empty (unless the filesystem crashed, in which case it may contain unlinked inodes)
media	Used by the `sdcard` service for mounted media
mediadrm	Used by the Media DRM service
misc	"Miscellaneous" data and configuration directories for components. q.v. Table 2-dm.
nfc	Stores NFC parameters
property	Contains persistent properties (i.e. saved across device reboots). Each property is saved in its own file, with the property name serving as the file name
resource-cache	Resources cached by the AssetManager (described in Volume II).
security	commonly empty
ssh	For devices which provide the Secure Shell service. (Usually empty)
system	A multitude of system configuration files, shown in table 2-18
tombstones	Application crash reports generated by debuggerd. Due to limited filesystem space, full core dumps are not feasible. The debuggerd provides basic autopsy services in absence of a core dump. Some vendors allocate a separate partition to this directory.
user	JB and later: provides "multi-user" capabilties, by symlinking user numbers (0,1..) to directories with installed applications and data for those users. In a single user system, 0 links to `/data/data`.

The `/data` directory permissions, as well as those of `/data/data` (discussed next) are both set to `chmod 771 system system`, and therein lies a tenet of Android's security model: The directory is executable (i.e. `cd`-able) to all applications, but unreadable (so applications or untrusted processes can't enumerate "neighbor" directories). This means that, as uid `shell` (in a non-rooted `adb` session) you will be able to change directory into `/data` and most of its subdirectories, but not necessarily be able to read their contents. The system subdirectories (e.g. `/data/system` and `/data/misc` will be readable, but `/data/data` and `/data` itself will refuse the `ls` command. This is also augmented as of KitKat by SELinux labels. You will therefore need root access to traverse subdirectories freely.

/data/data

The somewhat redundantly-named /data/data is the directory where all applications - both system and user-installed - store their information. Each application gets its own subdirectory, in reverse DNS format, which is `chmod 751` (`rwxr-x--x`), under the uid/gid of the owning application. The /data/data directory itself is `chmod 771 system system`, which makes it traversable by all applications, but readable to none but the system owned ones. The burden of securing specific application files, however, rests on each and every application, as the per-app directories are freely executable, though are unreadable by anyone other than the owner.

The /data/data per-app subdirectory is the only location in the entire filesystem which is writable by apps. Coupled with the fact that the stock applications for location, texting and calls can be found on every Android device, this makes several locations in it key for performing forensics. Subdirectories of particular interest are shown in table 2-16:

Table 2-16: But a few of the app directories of interest in /data/data

App subdirectory	Used by	Contains
com.android.providers.calendar	Calendar	Calendar: databases/calendar.db (in the events table).
com.android.providers.contacts	Phone Contacts	Virtually every tidbit of information which might be of remote interest on the device, in databases/contacts2.db: a SQLite3 master contact database, including tables like contacts (All contacts stored on the device) and calls (Log of last calls). files/thumbnail_photo_*xxxxx*.png are individual thumbnails of contacts.
com.android.providers.telephony	Messaging	Multimedia(MMS)/text(SMS) message database: database/mmssms.db
com.android.providers.settings	Settings	databases/settings.db: All Android framework runtime settings, and more in `global` and `secure` tables.
com.google.android.apps.maps	Google Maps	Destinations looked up: gmm_myplaces.db, gmm_storage.db and log_events.db. cache/http contains map tiles.
com.google.android.gm	GMail	databases/mailstore.*email*.db: a SQLite3 database containing all the user's mail which has been downloaded to the device, for each registered *email* address (in the messages table). Viewed attachments are stored in cache/*email*.
com.android.chrome	Chrome browser	State of Chrome browser (which replaces the old Android built-in com.android.browser). Files of interest include the cache/ directory (browser cache), and the app_chrome/Default/ directory, which contains many important SQLite3 databases, such as History and Archived History (browsing history in urls table), Login Data (saved credentials, in logins table) and Cookies.

Applications may also save data on the SD-Card (if they have permissions), but most of the data pertinent to the application state can often be found in its /data/data directory. This is useful if you want to manually save and rollback application state (for example, to cheat in most games). Applications can also register with the Android backup service for automated backups - locally or to Google's cloud services - as discussed in the next chapter.

Table 2-16 is naturally far from comprehensive. Nonetheless, if you're interested in finding specific application files, it's fairly straightforward to look for the app in /data/data by the reverse DNS notation (which matches the APK name). From there, it's a simple matter of grabbing the files (on a rooted device), then using `sqlite3` on the various databases and `file` to identify and view others. This is shown in the following experiment:

Experiment: Device forensics through /data/data

On a rooted device, you can easily examine application data directories with SQLite3. The Android emulator image contains a sqlite3 binary in /system/xbin, as do most rooting packages (for reasons which should now be fairly obvious).

Taking as an example Chrome, start the browser and navigate to any site of your choice. To look at the history database you will need to kill the process, since it holds a lock on the database. From there, a simple SQL query reveals all.

Output 2-8: Examining Chrome's history with sqlite3

```
root@htc_m8wl:/ # cd /data/data/com.android.chrome
# Using ".schema" shows the table definition:

root@htc_m8wl:/data/data/com.android.chrome #  sqlite3 app_chrome/Default/History
sqlite> .schema urls
CREATE TABLE urls(id INTEGER PRIMARY KEY,url LONGVARCHAR,title LONGVARCHAR,
visit_count INTEGER DEFAULT 0 NOT NULL,typed_count INTEGER DEFAULT 0 NOT NULL,
last_visit_time INTEGER NOT NULL,hidden INTEGER DEFAULT 0 NOT NULL,
favicon_id INTEGER DEFAULT 0 NOT NULL);
CREATE INDEX urls_url_index ON urls (url);
sqlite> select * from urls where url like "%android%";
id|url                          |title                  | | |last_visit_time  | |
52|http://newandroidbook.com/        |Android Internals      |2|2|13054934895637919|0|0
53|http://newandroidbook.com/TOC.html|Android Internals::TOC |1|0|13054934883061164|0|0
```

Demonstrating the same on the contacts2.db in /data/data/com.android.providers.contacts/databases:

Output 2-9: Examining the call log

```
sqlite> .schema calls
CREATE TABLE calls (_id INTEGER PRIMARY KEY AUTOINCREMENT,number TEXT,
presentation INTEGER NOT NULL DEFAULT 1,date INTEGER,duration INTEGER,
...
# E.g. find all toll free calls
sqlite> select _id, number, date, duration from calls where number like "%800%";
id|number     |date         |duration
2 |18001750930|1396019679278|0
16|18007562000|1402005179460|0
```

Another useful forensic trick - which merely requires the device to be unlocked, and not necessarily rooted - is to connect the device via adb to a host, and issue an adb backup request for the packages of interest. This calls on the the BackupManagerService, which - by virtue of running as system - can access /data/data with no restriction, and not only read all the files of any app, but also conveniently transport them to the host. (The backup process and the BackupManagerService are both described in detail in the next chapter and Volume II, respectively).

When initiating a backup, the BackupManagerService will prompt the user for confirmation (hence the need for an unlocked device). If the operation is approved, a backup archive is created on the host with an .ab (Android Backup) extension. Backups can be easily extracted on the host once you figure out the file format, as explained in the next chapter.

/data/misc

The /data/misc directory contains miscellaneous data and configuration directories for Android's subsystems. Contrary to its name, the contents include some of the most important files in the system. More detail can be found in Table 2-17:

Table 2-17: Directories in /data/misc

Directory	Contents
adb	Trusted ADB host public-keys (as of JB)
bluetooth	BlueZ (< 4.2 bluetooth subsystem) configuration files
bluedroid	Bluetooth subsystem (>4.2) configuration files
dhcp	Contains PID file of dhcp ctdent daemon, and any active lease
keychain	Android built-in certificate pins and blacklists
keystore	Per-user keystore data
sensors	Sensor debug data
sms	Contains the sms codes database
systemkeys	ASEC container keys (AppsOnSD.sks)
vold	M:Contains decryption keys for "adopted" external storage drive
vpn	VPN state configuration files
wifi	Wi-fi subsystem configuration files (e.g. wpa_supplicant.conf), and sockets

/data/system

Another important subdirectory of /data is /data/system, as it contains files critical to maintaining the state of device. As can be expected, access is restricted to system:system, so if your device is not rooted, you can't see any of the files shown in table 2-18:

Table 2-18:: The contents of /data/system

Directory	Notes
appops.xml	Used by the AppOps service, which controls application permissions.
batterystats.bin	Used by the BatteryStats service, which keeps power statistics by application.
called_pre_boots.dat	Used by the ActivityManager to hold pre boot broadcast receivers
device_policies.xml	Configuration file used by the DevicePolicyManagerService.
dropbox/	Directory used by the DropBox service.
entropy.dat	System entropy store, used by EntropyMixer for random number generation.
gesture.key	Lockscreen pattern, as discussed in Chapter 8.
framework_atlas.config	Used by the AssetAtlasService, which packs bitmaps into a single file.
ifw/	Intent FireWall rulebase (q.v. Chapter 8).
locksettings.db*	Lock screen settings: Contains device lock policy (q.v. Chapter 8).
netpolicy.xml	Configuration file used by the NetworkPolicyManagerService.
netstats/	Directory used to hold NetworkStatsService statistics - by device, uid, or xt. Previous versions of Android simply dropped the files in /data/system.
packages.list	PackageManager lists of all packages (APKs) installed in the system
packages.xml	Used by the PackageManager to hold metadata on all installed packages.
password.key	Lockscreen PIN/password hash, as discussed in Chapter 8.
procstats/	Directory used to store files for the ProcessStats service
registered_services/	Directory used by android.content.pm.RegisteredServicesCache
usagestats/	Used to store files for the UsageStats service. In particular, usage-history.xml
users/	Android's "Multi-User" support. Described in more detail in Chapter 8.

/cache

The /cache partitions is defined by Android for use during system upgrades. System updates are downloaded to this location, and the boot manager is aware of this partition, especially when booted in recovery/upgrade mode. Otherwise, the partition is normally empty.

If you've recently downloaded an OTA update, you will likely see it in the partition until it is installed. Additionally, the recovery binary and the system (specifically, the android.os.RecoverySystem class) make use of this partition to exchange information when booting into recovery (or update), as shown in table 2-19:

Table fs-cache: Paths in the /cache partition

recovery #define	Path	Usage
CACHE_LOG_DIR	/cache/recovery	Directory used exclusively by recovery binary
LAST_LOG_FILE	/cache/recovery/last_log	Log of previous recovery/update operation
LOG_FILE	/cache/recovery/log	Log of current recovery/update operation
COMMAND_FILE	/cache/recovery/command	Command line arguments to the recovery
INTENT_FILE	/cache/recovery/intent	Intent to fire after recovery is complete
LAST_INSTALL_FILE	/cache/recovery/last_install	Log of last installation
LAST_LOCALE_FILE	/cache/recovery/last_locale	Language settings, for next boot

The recovery and update processes are both detailed in Chapter 3.

/vendor

The /vendor directory is purposed to contain vendor-specific modifications to Android. Doing so allows for an efficient process of updating or upgrading of the OS when the need arises. Selected system components are hard-coded to check /vendor before or in addition to /system paths, as shown in table 2-20:

Table 2-20: /vendor paths searched by system components

Component	Path searched
Package Manager	/vendor/app
Fonts	/vendor/etc/fallback_fonts.xml
Shared Libraries	/vendor/lib
DRM libraries	/vendor/lib/drm /vendor/lib/mediadrm
eGL libraries	/vendor/lib/egl
Frameworks	/vendor/overlay/framework
Firmware	/vendor/firmware
Audio Effects	/vendor/etc/audio_effects.conf

The contents of /vendor greatly varies between devices, because vendors add their own apps and components as they see fit. Some vendors, e.g. Amazon, create their own subdirectory structure (/vendor/amazon) to include support for their custom frameworks and features (e.g. the Kindle's "smart volume" feature, which adjusts audio volume based on CSV files for each output device, placed in /vendor/amazon/smartvolume). Other vendors ignore this directory altogether and just add their modifications to /system. This is especially common with vendor apps, and in practice /vendor/app is often unused (even in the case of Amazon's FireOS), making it difficult to reduce the bloatware of vendor and carrier supplied apps. If Android L on the Nexus 9 is any indication, however, future versions of Android will have /vendor as a separate partition, which would allow it to be updated independently of the rest of the system.

The SD card

One of Android's strongest features is its built-in support for SD Cards, a feature which is sorely lacking for many users of iOS*. Most phones come built-in with an SD-card (albeit not an easily removable one) and tablets have a readily accessible expansion slot.

Most SD-Cards are formatted with the vfat or fat32 filesystems, but those filesysystems do not support permissions. In order to enforce permissions, as well as support multi-user configurations (starting with JellyBean), Android resorts to a somewhat contrived method of *emulating* the sdcards via FUSE (File systems in USEr mode). FUSE allows the implementation of filesystems in a user mode process (hence the semi-acronym), rather than in the kernel. A small kernel-level shim provides generic support, in the form of basic file system registration and interfaces to VFS, but the actual implementation is delegated to a user mode process, /system/bin/sdcard. The mount point for the SD card has changed several times over the course of Android's evolution, and is currently /storage/ext_sd. On devices with no SD-Card, the mountpoint is often an emulated one, pointing to a directory in the /data partition (usually /data/media/0). This is shown in output fs-sd, along with the default directory structure:

Output 2-10: The SD-Card directories.

```
shell@android:/ $ cd /mnt/sdcard
shell@android:/mnt/sdcard $ ls -F
Alarms/
Android/
DCIM/                # Shared with host when connected as camera
Documents/
Download/
Movies/
Music/
Notifications/
Pictures/
Playlists/
Podcasts/
Ringtones/
```

The standard directories are also defined as constants of the android.os.Environment class. Note that 3rd party applications can (and often do) create their own files and directories in the SD-Card.

Android provides an emulated SD Card file system on devices which do not have an SDCard, or in addition to the "real" SD Card file system. You can see the SD Card file systems using the mount command:

Output 2-11: Viewing SD Card file systems

```
shell@htc_m8wl:/ $ mount | grep fuse
/dev/fuse /mnt/shell/emulated fuse rw,nosuid,nodev,relatime,user_id=1023,group_id=1023,...
/dev/fuse /storage/ext_sd fuse rw,nosuid,nodev,relatime,user_id=1023,group_id=1023,...
```

A follow up to discussion, focusing on the technical aspects of the sdcard daemon can be found in Chapter 5.

* - Actually, iOS **does** support SD-Cards inherently, but the only way to add an SD-Card is to use the "Camera Connection Kit", which is, in fact, little more than a USB host adapter in disguise. Of course, that requires the eager Applite to fork over another $29.95 (or more), and use up the only slot on the device, which just so happens to be needed for power and USB connectivity..

Protected Filesystems

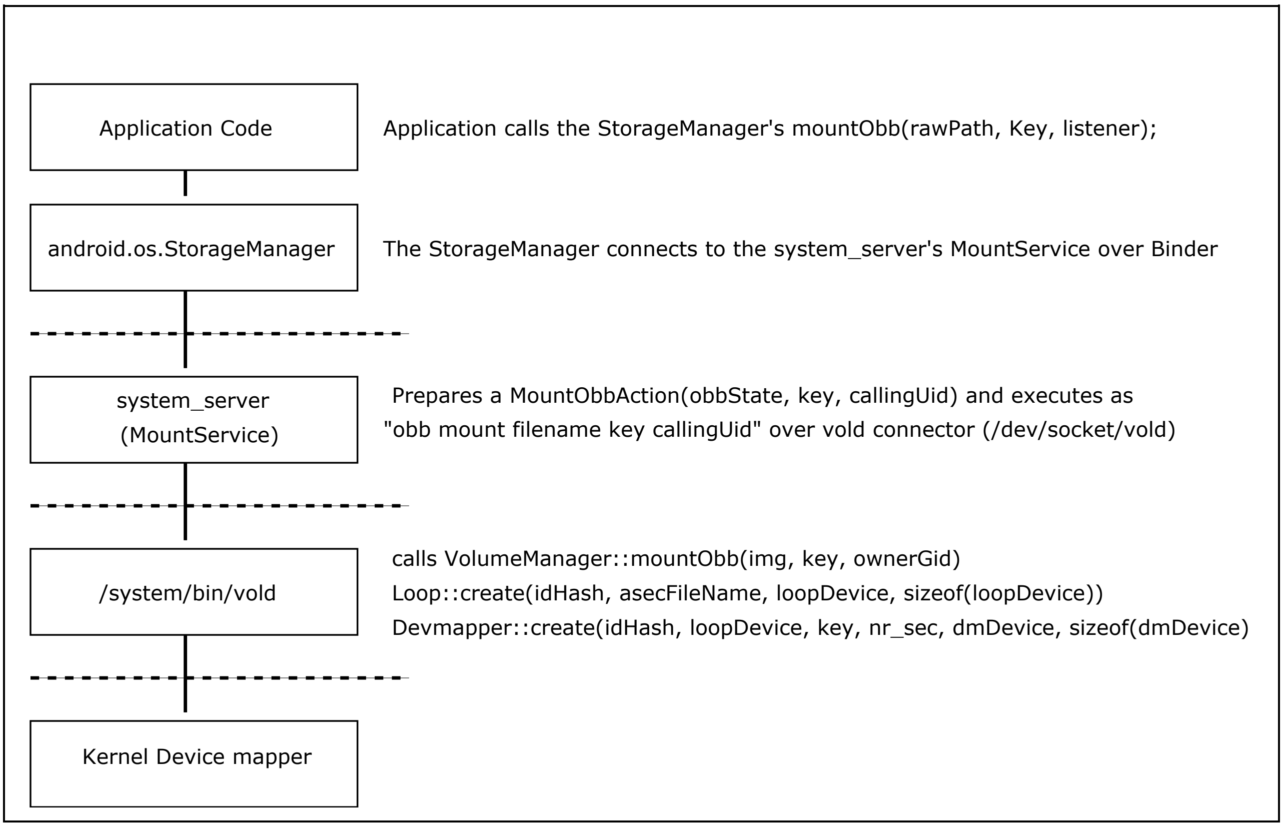
With Android being the open system that it is, come challenges to application deployment. A technically savvy-user could use `adb` to copy `apk` bundles between devices, and even an average user could do so, when the application is installed to an SD card, which is by its very nature removable. Unlike iOS, who was designed from the ground up with support for DRM (using its FairPlay mechanism, and encrypting application code by default), Android slowly adopted mechanisms to achieve the same ends over its evolution. Two such mechanisms - Opaque Binary Blobs (OBB) and Android Secure Storage (ASec) are covered in this section. The former grew out of a need to bypass restrictions in APK, and the latter - with security in mind.

OBB - Opaque Binary Blobs

Google Play restricts the size of an APK to about 50MB. For some applications, this could be a serious restriction, especially if they require or process multimedia files. With GingerBread, Android brought support for the Opaque Binary Blob (obb) format. This allows developers to provide additional data files (up to 2GB) to the applications, in the form of **opaque binary blobs** or **obb** files, which can archive multiple files into a single blob, and provide optional encryption.

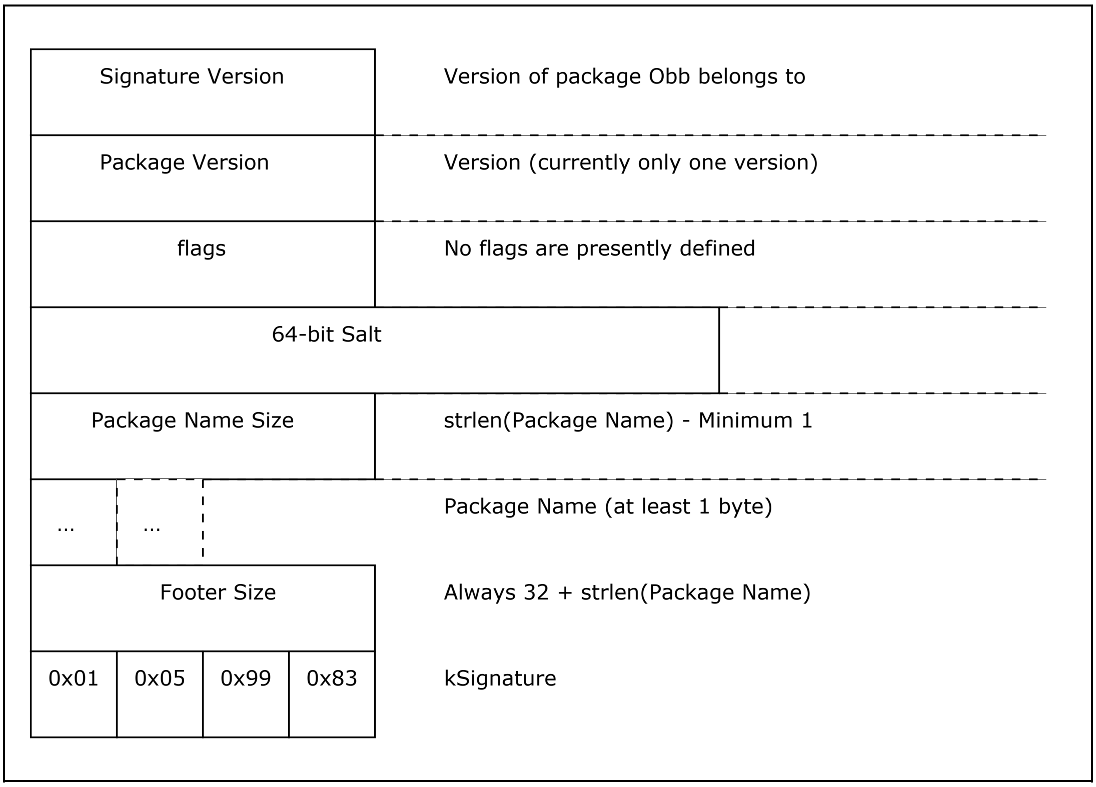
Implementation-wise, the obb is exactly that - opaque - meaning its contents and formatting are up to the application developer to decide. Oftentimes, though, it is a vfat filesystem image, which is mounted by a call to the volume daemon. The vold then calls on the Linux kernel's device mapper to perform a loop mount. The device mapper also supports twofish encryption, and the key is passed to it during the obb mount request. Applications can then call on the `android.os.StorageManager`'s `mountObb` method to mount the obb by specifying the key. This is shown in figure 2-1:

Figure 2-1: The OBB mounting process


Application Code
Application calls the StorageManager's mountObb(rawPath, Key, listener);
android.os.StorageManager
The StorageManager connects to the system_server's MountService over Binder
system_server
(MountService)
Prepares a MountObbAction(obbState, key, callingUid) and executes as
"obb mount filename key callingUid" over vold connector (/dev/socket/vold)
/system/bin/vold
calls VolumeManager::mountObb(img, key, ownerGid)
Loop::create(idHash, asecFileName, loopDevice, sizeof(loopDevice))
Devmapper::create(idHash, loopDevice, key, nr_sec, dmDevice, sizeof(dmDevice)
Kernel Device mapper

As opaque as they may be, Obb files still need to have some type of metadata to allow their parsing by the system. Support for Obbs is provided in the native /system/lib/libandroidfw.so, and inspecting its ObbFile.cpp implementation reveals the metadata is in a trailer (rather than a header, as one would normally expect). Obbs are thus parsed by seeking to the end of the file, and working backwards, uncovering the footer fields as shown in figure 2-2:

Figure 2-2: The Obb Trailer


Signature Version
Version of package Obb belongs to
Package Version
Version (currently only one version)
flags
No flags are presently defined
64-bit Salt
Package Name Size
strlen(Package Name) - Minimum 1
...
...
Package Name (at least 1 byte)
Footer Size
Always 32 + strlen(Package Name)
0x01
0x05
0x99
0x83
kSignature

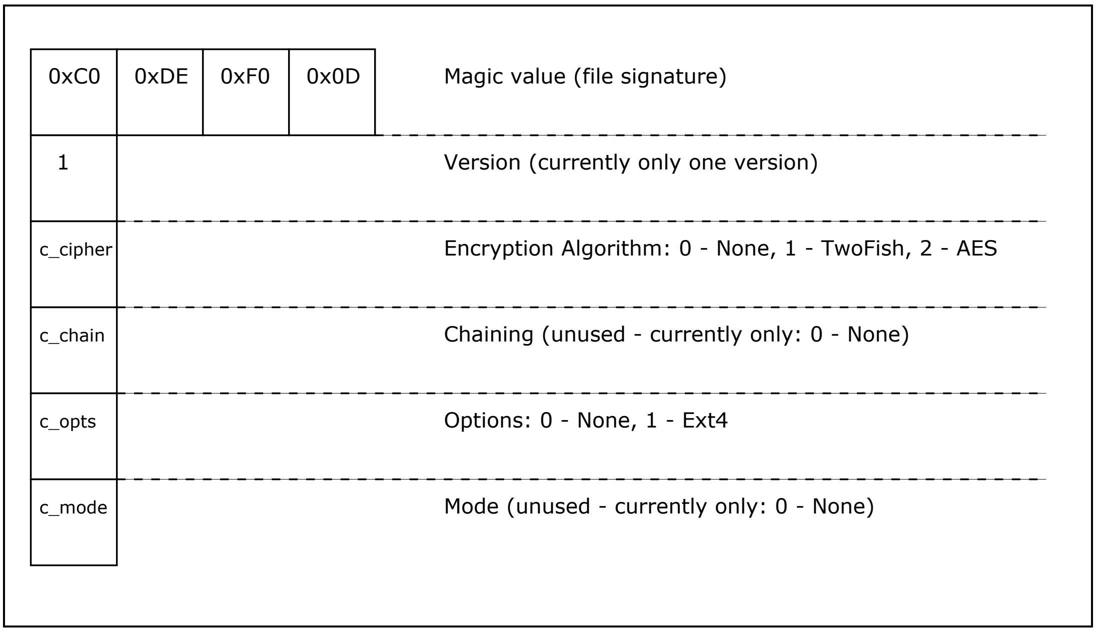
The Android source tree contains the obbtool, which is a shell script that can be used to create obb files on Linux - First creating an empty vfat image, then using the device mapper on the host to loop mount. Once mounted, files can be added to it, and the image is then committed when umounted. The SDK additionally provides the jobb[6] utility to create and manipulate obbs. The framework also provides the ObbScanner class, which can be used to obtain basic metadata about obb files (using JNI calls to the aforementioned libandroidfw.so). OBBs are discussed in the Android Documentation under APK Expansion Files[7]. You can interact with `vold` through the `vdc` command to list, mount and unmount obbs, as discussed in Chapter 5.

ASec - Android Secure Storage

Android's "Secure Storage" feature, commonly referred to as **asec**, provides a mechanism for applications to securely deploy onto the device, while maintaining a reasonable level of assurance that the user will not copy them to another device - a process often referred to as "forward-locking". By using asec **containers**, an application can be deployed anywhere. The feature was added in FroYo, which was the first version of Android to support external storage, such as SDCards. Indeed, the containers may reside on SDCard, but are unusable without the key. Naturally, the keys need to be stored somewhere, and Android maintains them in a system keystore (in /data/system/misc/systemkeys). Hence, the "reasonable" level of assurance - the root user can read the encryption keys.

Asec containers are, in essence, encrypted filesystem images which begin with a fixed header, the `asec_superblock`, defined in system/vold/Asec.h as shown in figure 2-3:

Figure 2-3: The ASec header (from system/vold/Asec.h)



0xC0	0xDE	0xF0	0x0D	Magic value (file signature)
1				Version (currently only one version)
c_cipher				Encryption Algorithm: 0 - None, 1 - TwoFish, 2 - AES
c_chain				Chaining (unused - currently only: 0 - None)
c_opts				Options: 0 - None, 1 - Ext4
c_mode				Mode (unused - currently only: 0 - None)

Asec creation and management is handled by the volume manager, vold, which performs operations as instructed by the MountService. Both asec creation and mounting require a key, and when the asec container is mounted, vold uses the kernel's device mapper and performs a loop mount, passing the key to the kernel's dm-crypt facility through a DM_TABLE_LOAD ioctl.

You can use df to see asec mounts, with vold's command line, vdc (described in Chapter 5), supplying detail as well:

Output 2-12: Viewing ASEC file systems

```
shell@s4$ df | grep asec
/mnt/asec                   930.8M      0.0K    930.8M    4096
/mnt/asec/com.tripadvisor.android.apps.cityguide.shanghai-1    23.0M
#
# Use the Volume Daemon Command utility (vdc) to examine mounted asec
#
shell@s4$ vdc asec list
111 0 com.tripadvisor.android.apps.cityguide.shanghai-1
200 0 asec operation succeeded
```

An age old challenge arising from encryption is key management - in other words - where does one store the encryption key to the ASec containers? If the key itself is encrypted, one runs into a chicken and egg problem. Android therefore chooses to place the (128-bit BlowFish) key in a single file called /data/misc/systemkeys/AppsOnSD.sks. The file contains the key in simple plaintext, but is set to be readable only by root. Naturally, this means on a rooted device relying on ASec as a form of intellectual property protection is quite useless.

For the reader interested in more hands-on experiments with asec containers, the Android Explorations blog post about JB's App Encryption[8] makes a great read.

If the asec feature seems similar to the OBBs that were just discussed, it is no conincidence: both features rely on the device mapper and its file encryption (dm-crypt) to both create and access the data. An asec can be seen as a logical progression of obb - from containing application extension files to encompassing the entire app. The same mechanism can be expanded to the full filesystem level, which is in fact what Android uses for its full disk encryption feature, described in Chapter 8. This has been expanded further in Android M's "adoptable" storage, enabling encryption of external USB storage through dm-crypt.

The Linux Pseudo-Filesystems

While not strictly Android filesystems, the Linux kernel provides three other filesystems of note, which are used by Android as well. These are especially important for our discussion in Chapter 7, which discusses (among other things) the Linux perspective of apps - that is, tracing and analyzing Android apps as the processes they are at the Linux level. This section is not meant to provide a comprehensive reference to these directories ; Rather, it illustrates the particular paths which are of interest to future discussion.

Note the term **pseudo-filesystems**: None of these filesystems are "real" in the sense of being backed by actual storage. Instead, the filesystems are maintained directly by in-kernel callbacks, so that upon access to a file or directory, a corresponding kernel-level handler function is invoked. This means that these filesystems take no actual space (in-kernel memory for inode and dentry representation not withstanding). Further, each access to a file or directory on a pseudo-filesystem triggers the kernel callback function, so the files and directories always reflect the most up-to-date data. As a corrolary, file sizes are meaningless, which is why an `ls -l` will show the files as seemingly empty (or with an arbitrary size of 4k, a pagesize, in older kernels). Note, that because the files are exported by kernel code (kernel proper or, in some cases, modules, the files greatly vary with kernel versions, and content (especially in *sysfs*) is highly hardware dependent.

Most pseudo-files created are read-only, and aim to provide real-time diagnostics, providing user-space with a mechanism to poll on variables and structures which would otherwise be inaccessible, in kernel mode. Some files, however, are actually writable, which provides an even more useful ability to directly affect kernel data, in real-time, from user space. Contrary to certain registry-based systems, wherein changes require excruciating manipulation of hidden and oft undocumented keys or values (not to mention a reboot), changes made to files in the pseudo filesystems - where allowed - are enforced immediately, but by default do not persist across a system reboot. That, however, is seldom a concern, because it's a trivial matter to re-enforce these changes during system startup, which is in fact what significant portions of the Android init.rc scripts (detailed in Chapter 4) are all about.

cgroupfs

The Linux kernel provides an important resource control mechanism called **cgroups**. A cgroup is a `container group` for one or more threads, allowing operations and policy settings to apply on the group as a whole. A fairly comprehensive documentation on cgroups can be found in the Linux kernel documentation[9]. To facilitate the placement of threads into groups, cgroups expose themselves via pseudo file systems. It then becomes a simple matter to add a thread to a group by "writing" into those files.

Though highly versatile and usable in oh so many ways, Android uses cgroups in a fairly limited manner, requiring it only for cpu accounting, and thread scheduling.

Output 2-13: cgroup-related mounts on a Nexus 5

```
shell@Nexus5 /: # mount | grep cgroup
/acct cgroup rw,relatime,cpuacct 0 0
none /sys/fs/cgroup tmpfs rw,seclabel,relatime,mode=750,gid=1000 0 0
none /dev/cpuctl cgroup rw,relatime,cpu 0 0
```

Bionic sets up accounting for every process it starts (therefore applying to every process on the system) through /acct. /sys/fs/cgroup/memory is accessed by the ActivityManager (via android.os.Process and its setSwappiness JNI method). Last, but not least, is the /dev/cpuctl directory: despite being in /dev, this is a cgroup directory set up by Android's scheduling policy. When /init starts, it sets up the directory and creates subdirectories (therefore, scheduling groups) for system tasks (/dev/cpuctl/tasks, foreground apps (/dev/cpuctl/apps/tasks, and background apps (//dev/cpuctl/apps/bg_non_interactive/tasks). Each group is assigned a number of cpu "shares", and given an upper bound on execution time. This prevents any wayward or misbehaving app from impacting execution as a whole. The /dev/cpuctl configuration, performed in /init.rc, is shown in the following listing:

Listing 2-2: Setting up the cpuctl cgroups

```
    mkdir /dev/cpuctl
    mount cgroup none /dev/cpuctl cpu
    chown system system /dev/cpuctl
    chown system system /dev/cpuctl/tasks
    chmod 0660 /dev/cpuctl/tasks
    write /dev/cpuctl/cpu.shares 1024
    write /dev/cpuctl/cpu.rt_runtime_us 950000
    write /dev/cpuctl/cpu.rt_period_us 1000000
    mkdir /dev/cpuctl/apps
    chown system system /dev/cpuctl/apps/tasks
    chmod 0666 /dev/cpuctl/apps/tasks
    write /dev/cpuctl/apps/cpu.shares 1024
    write /dev/cpuctl/apps/cpu.rt_runtime_us 800000
    write /dev/cpuctl/apps/cpu.rt_period_us 1000000
    mkdir /dev/cpuctl/apps/bg_non_interactive
    chown system system /dev/cpuctl/apps/bg_non_interactive/tasks
    chmod 0666 /dev/cpuctl/apps/bg_non_interactive/tasks
    write /dev/cpuctl/apps/bg_non_interactive/cpu.shares 52
    write /dev/cpuctl/apps/bg_non_interactive/cpu.rt_runtime_us 700000
    write /dev/cpuctl/apps/bg_non_interactive/cpu.rt_period_us 1000000
```

debugfs

The debug filesystem is strictly intended for kernel-level debugging information. Drivers and subsystems alike are free to dump droves of debugging information into the filesystem, which (as with the other pseudo-filesystems). If the filesystem is mounted, the myriad debugging information can be read like any other file.

Note, however, the "if" - The debug filesystem need not necessarily be mounted, and the kernel could possibly be compiled without debugfs support. If the kernel supports it, the filesystem can be mounted using a simple command line (usually in /init.*hardware*.rc), like so:

```
mount -t debugfs none /sys/kernel/debug
```

though any mountpoint can be chosen. Since it's so useful, it's not uncommon to find a symbolic link from the root, as is the default in the emulator image, from /d to the mount point.

The contents of the debugfs are highly dependent on the kernel version and whichever debug features have been implemented in it. The following table lists common entries found in Android kernels:

Table 2-21: Entries in the /sys/kernel/debug directory

Entry	Purpose
binder	plentiful data on the eponymous Android IPC mechanism
tracing	unbelievably useful, copious debugging and tracing information generated by the Linux kernel's ftrace mechanism
wakeup-sources	Kernel level wakelocks, used by drivers or the Android system to prevent device sleep

functionfs (/dev/usb-ffs/adb)

USB functionality in Android is controlled by a special "gadget" driver, which often requires dynamic reconfiguring according to user-selection (e.g. connect device for USB debugging, as Mass-Storage, etc) through init (as explained in Chapter 4).

The traditional driver (in kernels before L) needs to export its reconfiguration parameters through sysfs. Doing so is one of the reasons it is considered bloated, and in need of revamping.

Enter: *functionfs*. A relatively new addition to the Linux kernel (sometime in 2010), this is a generic file system provided by the Linux kernel to provide a way for drivers to pick up configuration changes from user space. The filesystem can be thought of a complement to sysfs, in that whereas the latter is designed for outputting kernel variables and driver information to user mode, the former is designed for input. The root user can use `mkdir(2)` to create directories, which in turn will create corresponding kernel objects, which can then be initialized from user-space by further write(2) operations to the pseudo-files in the directories.

procfs (/proc)

The *procfs* filesystem derives its name from its initial purpose - to provide a directory-based view of processes running in the system. The idea originated in Plan 9 operating system, and Linux quickly adopted it and modified it to provide a plethora of information - on processes, threads, and other system-wide diagnostics. In fact, some argue that /proc has become a virtual junkyard of diagnostic files, because Linux originally provided pseudo-file interfaces for this directory only.

Regardless of whether or not /proc provides too much of a good thing, it is undeniable that it makes for a highly important filesystem. Many Linux utilities (e.g. top, netstat, lsof and ifconfig), as well as Android tools (e.g. procrank, librank) depend on it as the source of diagnostic information.Linux keeps a fairly detailed and updated man page for `proc(5)`. We discuss the usage of procfs for debugging in Chapter 7.

pstore (/sys/fs/pstore)

The `pstore` mechanism is a Linux kernel feature (introduced in 3.5) which allows the kernel to set aside some RAM as a **persistent store**. This is used for one purpose - capture kernel panic data.

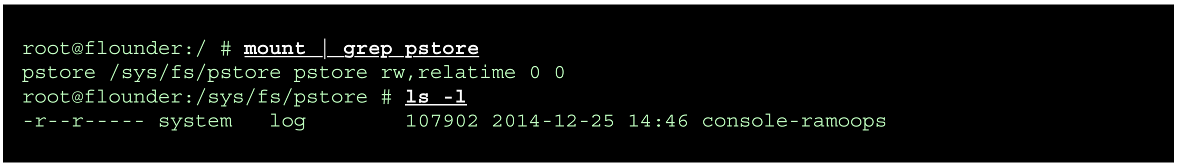
A panic indicates an internal kernel memory corruption, which may affect the filesystem logic. As such, any write to the filesystem could worsen things, and lead to filesystem corruption, as well. Normally, UN*X system dump panic data to the swap partition - which isn't meant to survive reboot anyway. But Android has no swap, and therefore the only reliable solution is to set aside some physical memory (i.e. a dedicated portion of the RAM), and have the kernel log its crash data (the bare minimum, at least) to there. The kernel then automatically performs a *warm reboot* - that is, a reboot without a power cycle, which means that the RAM does not undergo full re-initialization. During reboot, the kernel checks the persistent store for any relics of its past incarnation - and , if any are found, they are made available through /sys/fs/pstore.

In older versions of Android, this functionality was provided by an "Androidism" (i.e. specific Android kernel hack) called the **RAM console**. Traces of this can still be found in /init.rc files, capturing data from /proc/apanic_console and /proc/apanic_threads, and moving them to /data/dontpanic (with a wink to the "Hitchhiker's Guide to the Galaxy"). With the advent of the pstore functionality, this is deprecated in favor of /sys/fs/pstore.

Experiment: Testing the persistent store on Android L

On an Android L (or any system with a kernel version of 3.10 or later) it is very likely that the pstore is enabled by default. To see if it is, check for the existence of /sys/fs/pstore or any other mount point specifying the `pstore` file system:

Output 2-14: Locating the pstore



```
root@flounder:/ # mount | grep pstore
pstore /sys/fs/pstore pstore rw,relatime 0 0
root@flounder:/sys/fs/pstore # ls -l
-r--r----- system   log        107902 2014-12-25 14:46 console-ramoops
```

If your kernel rebooted and/or crashed recently, the mount point will be populated with a single file: console-ramoops, which holds the last `dmesg` output. The file's permissions - `system:log` are set in the /init.rc, and make it readable by the `adb shell` (which is a member of the `log` group). You can then `cat /sys/fs/pstore/console-ramoops` to get the last output of the kernel ring buffer, right up to the reboot.

If you cold booted your system, however, the directory may be empty. In this case, you can either reboot your system using `adb reboot`, or (if you dare), force a kernel crash using the command:

```
echo c > /proc/sysrq-trigger
```

which will make the file appear.

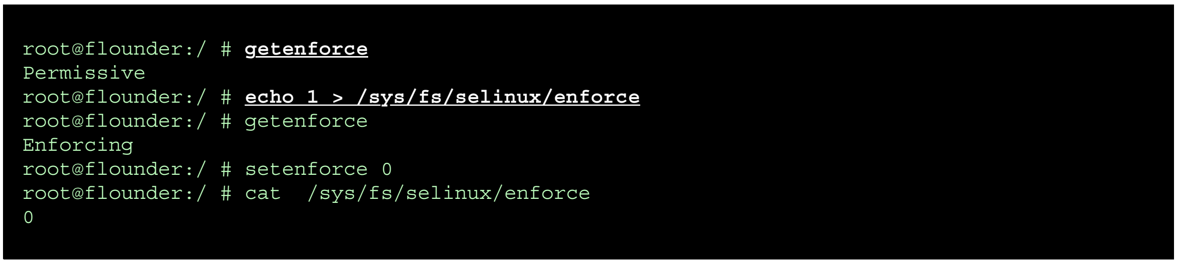
The /proc/sysrq-trigger pseudo-file is an incredibly useful (but dangerous) /proc entry. The file is writable only, and `echo`ing a single key into it simulates the functionality of pressing the little known `SysRQ` key along with ALT and the key specified - a magic key combination which works only from the console. The `SysRQ` functionality is meant as an emergency channel when the system is non-responsive, since `sysrq` requests are processed by the keyboard interrupt handler (which runs at the highest possible priority). Exercise extreme caution when handling this file, as most of the options there are for emergency use only and may be destructive.

selinuxfs (/sys/fs/selinux)

The SELinuxFS, like the debugfs, is traditionally mounted under /sys but is not part of the sysfs filesystem per se. The filesystem has been devised for exclusive use by SELinux, and it stores important files relating to the installed policy.

SELinux is discussed in more detail in Chapter 8, but at a bird's eye view, the most important files in this filesystem are policy - which provides the loaded (compiled, binary form) security policy, and the enable pseudo file, which toggles enforcement of the policy (and is in fact what the `getenforce`/`setenforce toolbox` tools use).

Output 2-15: Demonstrating enforcement of an SELinux policy



```
root@flounder:/ # getenforce
Permissive
root@flounder:/ # echo 1 > /sys/fs/selinux/enforce
root@flounder:/ # getenforce
Enforcing
root@flounder:/ # setenforce 0
root@flounder:/ # cat  /sys/fs/selinux/enforce
0
```

sysfs (/sys)

The **sysfs** may be alphabetically last, but in order of importance it is second only to procfs. The sysfs was introduced in kernel 2.6 as a complement to procfs - In an effort to declutter /proc, and move hardware and module related configuration files to a separate location, with more structure.

Under /sys you can find, therefore, a "cleaner" separation of pseudofiles, by category. The subdirectories you'll see are shown in Table 2-22:

Table 2-22: The subdirectories (classes) in /sys

Subdirectory	Contents
block	Block I/O Layer control files. One subdirectory per block device, containing parameters such as the I/O scheduler.
bus	Devices, by bus connection. One subdirectory per bus type (e.g. i2c/, mmc/, soc/)
class	Devices, by class. One directory per class type (e.g. input/, sound/).
dev	Devices, by device type: block/ or char/
devices	Devices, in device-tree compatible form
firmware	Used for firmware-update capable devices
fs	Used by filesystem drivers. Some subdirectories here are mountpoints (e.g. pstore/, selinux/ are mount points for other pseudo-filesystems, as previously discussed. Others provide exported parameters and statistics by filesystems (such as ext4/).
kernel	Various kernel parameters, by subsystem. debug/ serves as mount point for debugfs
module	One subdirectory per module, containing module statistics and (where applicable) module parameters (viewable and sometimes settable from user-space)
power	Power management statistics and settings. The Android WakeLocks are implemented here (via the wake_lock and wake_unlock pseudo-files)

Hardware configurations greatly differ in-between devices - and therefore so do the actual contents presented by the corresponding sysfs files. The Android frameworks are shielded from device-specific idiosyncrasies thanks to the Hardware Abstraction Layer (/system/lib/libhardware.so and its plugins), which wrap the calls to the specific files with more generic API calls (The HAL is discussed in more detail in Volume II).

Other device entries are somewhat more standardized. These include the CPU governor (frequency scaling) data, in /sys/devices/system/cpu/cpu#/cpufreq, and the vibrator (on devices which have one) in /sys/class/timed_output/vibrator. For a quick, fun experiment, you might want to try to echo a large value (say, 5000) to the enable sysfs entry in that directory.

Summary

This chapter provided a walkthrough of Android's partitions and filesystems. In particular, we focused on the partitions commonly found in Android devices - noting all but a few are actually unmountable. We then examined the two main filesystems - /system and /data, whose contents, subdirectory by subdirectory, were detailed. Lastly, the chapter touched on the Linux pseudo-filesystems, which contain a cornocupia of diagnostics and configuration files. Those files, with an emphasis on their use in debugging, will be revisited throughout this book, especially in Chapter 7.

The next chapter builds on this one, as it explores Android's boot and recovery processes. The non mountable partitions - and in particular aboot and boot will be examined in detail, as they play the part in starting up the device. The mountable but rarely used /cache will be revealed as central to OTA-updates.

References

1. XDA Developers on f2fs: http://forum.xda-developers.com/showthread.php?t=2697069
2. Samsung f2fs presentation: http://elinux.org/images/1/12/Elc2013_Hwang.pdf
3. Linux Weekly News on f2fs: http://lwn.net/Articles/518988/
4. XDA Developers "El Grande Partition Table Reference": http://forum.xda-developers.com/showthread.php?t=1959445
5. Companion Article: HTC WeakSauce Exploit: http://NewAndroidBook.com/Articles/HTC.html
6. Android Developer on the jobb utility: http://developer.android.com/tools/help/jobb.html
7. Android Developer on APK Expansion Files:http://developer.android.com/google/play/expansion-files.html
8. Android Explorations on JB App Encryption: http://nelenkov.blogspot.com/2012/07/using-app-encryption-in-jelly-bean.html
9. Linux kernel documentation on CGroups: https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt (or the kernel sources)

III: Android Boot, Backup & Recovery

Just like with their desktop counterparts, most users take the boot process of their mobile device for granted. They power on the device, wait for a few seconds, and the home (or lock) screen appears to greet them. The only time this sequence changes is in cases where the device is being updated ("flashed"), hasn't enough battery charge, or - in those rare cases where boot fails, and users find their device "bricked".

While the boot process of mobile devices follows the general lines of desktops, it is invariably more complicated, and involves more parts. Add to that the myriad device types and vendors in the Android landscape, and you get a process that is quite lengthy, and quite idiosyncratic for each device and vendor. This chapter aims to document this process, and focus on the common denominators between devices.

We begin with an examination of Android software images - which are flashed to the device partitions we discussed in the previous chapter. These can be manually downloaded, but are often fetched by the device in an Over-The-Air (OTA) update. While vendors are free in creating and formatting the images, most follow a general structure, consisting of the boot loader, the the boot image (containing the kernel and the RAM disk), and the sparse system images - which we discuss in turn. We follow these components as they each play their role in the boot and/or recovery process.

After discussing the boot process, it makes sense to discuss its inverse - shutdown. While at the Linux level shutdown and reboot are simple matters (handled by a kernel system call), Android's handling is more intricate, accommodating for Androidisms such as presenting the power menu, and booting into recovery mode.

The discussion of booting to recovery, begs more detail about just how recovery is performed, as well as the process of applying updates - so the next section is where OTA update packages and process are described.

Lastly, we consider custom firware images - "ROMs", as they are often (erroneously*) referred to, and discuss how the key components can be upgraded or entirely replaced. We leave out the natural sequitur to this discussion - device rooting techniques - for Chapter 8, which deals with security aspects of Android.

This chapter makes use of `imgtool`, a utility for viewing and unpacking Android system images. For your convenience, both source and a precompiled binary are freely available for download in one package[1].

* - Technically, ROM implies Read-Only-Memory, which cannot be updated or (in the case of EEPROM) can be erased and rewritten under special circumstances. Android devices do have a true boot ROM component, but the rest of the boot process is performed from flash partitions, which can be easily written to (assuming permissions hold).

Android Images

Various Android devices each have their own specific images. These are the system images, provided by the vendor, which are meant to be flashed as the "factory default" distribution of Android onto the device. The images are comprised of several files, which are flashed into their respective partitions:

- **The Boot Loader:** which provides the application processor bootstrap code. This code is normally responsible for finding and loading the boot image, but also handles firmware updates, and booting into recovery mode. Most bootloaders also implement a small USB stack, over which they can communicate with the host for purposes of controlling the boot or update process (normally over fastboot). The boot loader usually gets flashed into the `aboot` partition, though on some devices (e.g. HTC) this may be called "hboot".
- **The Boot Image:** which normally consists of the kernel and a RAM disk, and is used to load the system. Assuming normal boot, the RAM disk will serve as the root filesystem for Android, and its `/init.rc` and related files will provide directives as to how to load the rest of the system partitions. The boot image is flashed into the `boot` partition.
- **The Recovery image:** similarly consisting of the kernel and a (different) RAM disk, and is used to load the system into "Recovery mode", in cases where normal boot has failed, or in case of an OTA update. This gets flashed into the `recovery` partition.
- **The System Partition:** which is the full Android system, including the Google supplied binaries and frameworks, as well as any provided by the vendor, and/or the carrier.
- **The Data Partition:** containing the "factory default" data files, which support the binaries in the system partition. This image also provides the "factory default" state to which the device is restored when effaced.

Google makes the system images for the Nexus devices available at their factory image repository[2]. You are encouraged to follow along with the hands-on experiments in this chapter with those images, or your own device, if rooted. Refer to the method in Chapter 2 for safely extracting the images from the raw partitions of a live device. To unpack a Google stock image, follow these steps:

- Download the image from Google's factory images repository. This will be a gzipped tar file, with a name in the following convention:
  piscine_devicename-build-factory-*first_32_bits_of_SHA1_checksum*.tgz
- Unpack the file using tar: this will look something like the following Output:

Output 3-1: Unpacking a factory image for a Nexus 5

```
morpheus@Forge (~/Images)% tar zxvf hammerhead-ktu84p-factory-35ea0277.tgz
x hammerhead-ktu84p/
x hammerhead-ktu84p/image-hammerhead-ktu84p.zip
x hammerhead-ktu84p/radio-hammerhead-m8974a-2.0.50.1.16.img # Baseband update (best left alone)
x hammerhead-ktu84p/bootloader-hammerhead-hhz11k.img        # Bootloader components
x hammerhead-ktu84p/flash-all.bat                           # Batch file (Windows)
x hammerhead-ktu84p/flash-all.sh                            # Flash all images
x hammerhead-ktu84p/flash-base.sh                           # Bootloader and radio only
#
# Proceed to unpack the main image
#
morpheus@Forge (~/Images)% cd hammerhead-ktu84p
morpheus@Forge (~/.../-ktu84p)% unzip image-hammerhead-ktu84p.zip
Archive:  image-hammerhead-ktu84p.zip
  inflating: boot.img
  inflating: recovery.img
  inflating: system.img
  inflating: userdata.img
  inflating: cache.img
  inflating: android-info.txt
```

Next, we discuss each of these components (with the exception of the radio/baseband) in turn.

The Boot Loader

Android vendors are free to implement their own boot loaders, though most (Samsung being a notable exception) choose the "LK" (Little Kernel) Bootloader. The LK Bootloader is not part of the Android source tree, but is available (at least in part) from CodeAurora[3a] and from googlesource[3b].

LK as, as its name implies, a minimal implementation of boot functionality. The name is somewhat misleading, however, as it is **not** a Linux kernel, but a bootable ARM binary image. LK concerns itself with only the minimal functions one expects from a boot loader. These include:

- **Basic hardware support:** provided by LK's dev/ (basic common drivers, such as framebuffer, buttons, and USB target), platform/ (SoC/chipset drivers) and target/ (device specific) subtrees. Without this, none of the other requirements can be sated.
- **Finding and booting the kernel:** The raison d'etre of any boot loader, locating the bootimg (discussed next), and parsing its components - kernel image, ramdisk and device tree - then transferring control to the kernel with a given command line. This is carried out by app/aboot.
- **Basic UI:** for cases wherein the user interrupts the normal automatic boot sequence (commonly via `adb reboot bootloader` or pressing button combinations immediately after device power on). Aboot provides a simple text interface, which the user can navigate using the physical buttons on the device - using volume up/down to navigate, and the power button to select - but no touchscreen functionality.
- **Console support:** though most retail devices have no readily available console*, development boards provide console functionality through serial ports (RS232/UART). LK's lib/console (called from app/shell) provides a command interpreter (running in a separate thread) and support for extending the command list. lib/gfxconsole provides rudimentary graphics functions, such as font support.
- **USB Target Support:** which allows the bootloader to communicate with its host via a simple protocol, called **fastboot**, and discussed later in this chapter. A skeleton implementation can be found in app/aboot/fastboot.c, with vendors free to add their own extension ("oem") commands.
- **Flash partition support:** in order to enable the bootloader to erase or overwrite partitions, as required during upgrade or recovery. LK also contains basic filesystem support, through lib/fs.
- **Digital Signature Support:** to provide support for loading digitally signed images with SSL certificates, LK incorporates portions of the OpenSSL project in its lib/openssl subtree.

* - Surprisingly, it *is* possible to get a console connection to some devices, for example Google's Nexi, through one of the last places one would suspect - the headphone port! There is ample documentation on how to build your own headphone-jack-to-RS232, for the Nexus 4[4a] and Nexus9[4b].

The Boot Loader Image

Boot Loaders can be updated and flashed, just like other System Images. Though the format is not officially documented, the releasetools.py script in some of the device-specific directories of the Android source tree provides the header format. This enables imgtool to parse and extract boot images, as shown in this output, examining Google's Nexus 5 boot loader:

Output 3-2: The Nexus 5 Boot Loader Image

```
morpheus@Forge (~)% imgtool Images/hammerhead-kot49h/bootloader-hammerhead-hhz11k.img
Boot loader detected
6 images detected, starting at offset 0x200. Size: 2568028 bytes
Image: 0        Size:  310836 bytes     sbl1     # Secondary Boot Loader, stage 1
Image: 1        Size:  285848 bytes     tz       # TrustZone image
Image: 2        Size:  156040 bytes     rpm      # Resource Power Mgmt
Image: 3        Size:  261716 bytes     aboot    # Application Boot Loader
Image: 4        Size:   18100 bytes     sdi
Image: 5        Size: 1535488 bytes     imgdata  # RLE565 graphics used by boot loader
```

As you can see in the output, the Boot Loader image is comprised of several sub-images, each of which is meant to be flashed to a specific partition. The boot loader itself is in "aboot", which is the Application Processor Boot loader. The image also contains the Resource Power Management bootstrap (rpm), ARM TrustZone image (tz), and secondary boot loader (sbl1) (discussed later in this chapter).

None of the file formats of the boot loader components are documented. These are all highly architecture dependent, and the ones in the example above pertain to Qualcomm's SnapDragon processor (the msm chipset). The focus of this discussion - aboot - is incorrectly recognized by file(1) as an Hitachi SH big-endian COFF object, when in fact, it is formatted with a proprietary header, spanning 40 (or, in some cases more) bytes. The header format is shown in Table 3-1:

Table 3-1: The aboot proprietary header

Offset	Field	Contains
0x00	Magic	0x00000005 (constant)
0x04	Version	Version # (2 or 3)
0x08	?	NULL field
0x0c	Image Base	Virtual memory address to load rest of image into
0x10	Image Size	Size of aboot image
0x14	Code Size	Size of aboot code size
0x18	Last Code Addr	Image Base + Code Size
0x1C	Signature Size	Size of digital signature (usually 0x100 = 256 bytes)
0x20	Last Mapped Addr	Last Code Addr + Signature Size
0x24	Certificate Chain	Size of Certificate Chain, if any

Following the header is an ARM bootable image, which is mapped into memory at the address specified by the header. At its very base are the ARM exception vectors. These are a series of branch instructions, which define what addresses the processor will automatically jump to in certain cases (e.g. interrupts, exceptions, and aborts). The very first of those instructions - the reset handler - defines LK's entry point. The following experiment shows how you can remove the aboot header:

Experiment: Removing the header from the aboot image

If you have a Nexus 5 ROM update, using `imgtool` on its bootloader.img (as shown in Output 3-2) will extract aboot. Otherwise, on a rooted device you can obtain a partition dump by the method shown in Chapter 2, substituting /dev/mmbclk0 in that example with the partition of aboot (likely /dev/block/mmcblk0p6). One way or another, you will end up with aboot as a file.

Output 3-3: Making sense of aboot using od

```
morpheus@Forge (~/...-kot49h/)% od -A d -t x4 aboot | head -5
                 Magic             Version             NULL               ImgBase
0000000          00000005          00000003            00000000           0f900000
                 ImgSize           CodeSize      ImgBase+CodeSize         SigSize
0000016          0003fe2c          0003e52c            0f93e52c           00000100
         ImgBase+CodeSize+SigSize    Certs
0000032          0f93e62c          00001800            ea000006           ea00351c
```

The ARM instructions can be recognized by the "ea*XXXXXX*" form: "ea" is the opcode for the ARM B(ranch) instruction. Exception vectors in ARMv7 contain seven 32-bit slots, so the reset handler is usually instruction is usually ea000006 (as above): 6 * 4 bytes away from *next* instruction.

If you cut off the first 40 bytes of the file (using `dd bs=40 skip=1`), the resulting file can be loaded into a disassembler fairly easily. Cut again after *CodeSize* bytes, to remove the signature and the certificates, which should yield files that match the values of the header, namely:

Output 3-4: Getting the certificates from a bootloader image

```
morpheus@Forge (~/...-kot49h/)% dd if=aboot of=aboot.sans.header bs=40 skip=1
morpheus@Forge (~/...-kot49h/)% dd if=aboot.sans.header of=certs bs=0x3e62c skip=1
..
6144 bytes transferred in 0.000064 secs (96155984 bytes/sec)# 6144 = 0x1800 - all's well
```

You will need to rebase the image to 0x0f900000 (or whatever the field at offset 12 states). Reverse engineering of the boot loader is outside the scope of this chapter, but can be found in a companion article on the book's web site[5].

Boot loader locking

The boot loader on Android devices is usually **locked**, meaning it will refuse to flash or boot updates which are not digitially signed. The vendor provides its public key in ROM, and the key can be used to establish a chain of trust throughout the boot process. This way, all boot components - from the rpm through the sbl to the Android boot loader - can be verified. Reverse engineering of those components often reveals they contain an X.509v3 certificate, as well as the OpenSSL support needed to verify keys.

Boot loader locks are not to be confused with SIM Locks, which carriers often enforce to ensure that a phone purchased from them will only operate on their network. Rules in several countries already require carriers to unlock devices in certain cases, but no such rules force the sale of devices with unlocked boot loaders.

Thus, depending on the vendor, a user may or may not be able to unlock the phone. Some vendors refuse to do so, whereas in others (e.g. Google's Nexus 5 and the NVidia Shield) it's a simple a matter as issuing a "fastboot oem unlock" command. This is shown in Output 3-5, below:

Output 3-5: Unlocking the NVidia Shield Bootloader

```
morpheus@Forge (~)$ fastboot  devices
05141138021471071D9E    fastboot
morpheus@Forge (~)$ fastboot oem unlock
(bootloader) Showing Options on Display.
(bootloader) Use device keys for selection.
(bootloader) erasing userdata...
(bootloader) erasing userdata done
(bootloader) erasing cache...
(bootloader) erasing cache done
(bootloader) unlocking...
(bootloader) Bootloader is unlocked now.
OKAY [ 21.337s]
finished. total time: 21.337s
```

Other vendors (like HTC) take a middle ground, and have the device issue a challenge, in the form of a cryptographic token which must be responded to with a specific response. Some vendors sell both locked and unlocked phones (Samsung being the notable example). As of L, (at least in the Nexus 9), Android's default setting app allows the user to select whether or not the device is unlockable through **Settings >> Developer Options**. The user choice toggles a bit in a partition also readable by the bootloader.

Unlocking the bootloader, whenever possible, mandates that the boot loader entirely efface the `/data` partition. This is because unlocking the bootloader entirely compromises the device's security: An adversary gaining possession of the device can flash an update which will bypass any user PIN or pattern, or just copy the `/data` partition, and steal all the personal information found there.

If the boot loader cannot be unlocked, however, then the device - in theory - should be secure with no rooting method. In practice, however, Android is not without its share of exploits. As a matter of fact, at the time of writing a Linux kernel exploit which plagues versions below 3.13 has given rise to a root exploit, first publicized by GeoHot, known as "TowelRoot", which affects all Android devices on the market. This is but one of several exploits, commonly referred to as "one-click", which are akin to JailBreaking on iOS. These exploits, as well as rooting in general, are discussed in Chapter 21.

Boot Images

Android's boot images contain the core components of the operating system - the kernel and the RAM disk. The boot images (created with `mkbootimg`, in the Android source tree), bundle both with a minimal header, the kernel command line, a small hash, and an optional second stage boot loader (which in practice is unused). The images are recognizable by their magic (`ANDROID!`), similar to the bootloader magic (`BOOTLDR!`) discussed earlier.

Vendors are not strictly required to use this boot image format in their devices, and so results might vary with device. HTC, for example, prepends its own header, likely for use by their custom boot loader, HBOOT. You can usually spot the boot image header thanks to its magic value - `ANDROID!`, like so:

Output 3-6: The HTC boot image header

```
morpheus@Forge (~) $ od -A x -t c mmcblk0p43 # Dump hex offset + ASCII
0000000  250   2 032 244   ? 213  \0   u   O   W 220   e   I 300   J 235
..
0000100    A   N   D   R   O   I   D       Ð¥  **   `  \0  \0 200  \0  \0
```

And then use dd to skip the custom header (in the example above, dd bs=0x100 skip=1)

The format of the boot image is well documented in bootimg.h, as shown in listing 3-1:

Listing 3-1: The boot_img_hdr

```
struct boot_img_hdr {
    unsigned char magic[BOOT_MAGIC_SIZE];
    unsigned kernel_size;  /* size in bytes */
    unsigned kernel_addr;  /* physical load addr */
    unsigned ramdisk_size; /* size in bytes */
    unsigned ramdisk_addr; /* physical load addr */
    unsigned second_size;  /* size in bytes */
    unsigned second_addr;  /* physical load addr */
    unsigned tags_addr;    /* physical addr for kernel tags */
    unsigned page_size;    /* flash page size we assume */
    unsigned unused[2];    /* future expansion: should be 0 */
    unsigned char name[BOOT_NAME_SIZE]; /* asciiz product name */
    unsigned char cmdline[BOOT_ARGS_SIZE];
    unsigned id[8]; }     /* timestamp / checksum / sha1 / etc */
/*
** +-----------------+
** | boot header     | 1 page
** +-----------------+
** | kernel          | n pages
** +-----------------+
** | ramdisk         | m pages
** +-----------------+
** | second stage    | o pages
** +-----------------+
**
** n = (kernel_size + page_size - 1) / page_size
** m = (ramdisk_size + page_size - 1) / page_size
** o = (second_size + page_size - 1) / page_size
**
** 0. all entities are page_size aligned in flash
** 1. kernel and ramdisk are required (size != 0)
** 2. second is optional (second_size == 0 -> no second)
** 3. load each element (kernel, ramdisk, second) at
**    the specified physical address (kernel_addr, etc)
** 4. prepare tags at tag_addr.  kernel_args[] is
**    appended to the kernel commandline in the tags.
** 5. r0 = 0, r1 = MACHINE_TYPE, r2 = tags_addr
** 6. if second_size != 0: jump to second_addr
**    else: jump to kernel_addr
*/
```

The Kernel

The Linux kernel, unlike most OS kernels, is mostly compressed: The kernel file format, known as a `zImage`, consists of self-extracting code, which unpacks the rest of the kernel image in memory. As compression algorithms have greatly advanced, there are multiple options for compression, which can be decided during the build process (make config), as shown in Table 3-2:

Table 3-2: Kernel file formats

Magic	Compression	Notes
\x1f\x8b\x08\x00\x00\x00\x00\x00	GZip	Most common compression format
\x89LZOx00\x0d\x0a\x1a\x0a	LZO	Faster, but 10-15% less efficient than GZip. Used by Samsung

The kernel always starts with the self-extracting code portion, which means one has to scan well into the file in order to find the compression magic. Most ARM kernels traditionally use zImage, though there is no strict requirement to do so. The `imgtool` utility will automatically uncompress both GZip and LZO kernel images (if requested) and provide you with a binary you can disassemble or search strings in. When loading into a disassembler, you'll need to rebase the image at `0xC0000000` (assuming 32-bit).

The kernel is the most architecture specific component of Android: Whereas other components only care about the *processor type* (i.e. ARM, Intel, or MIPS), the kernel is also concerned with the *board type* and specific chipsets, because the processor is, in effect, a system-on-chip (SoC), which also contains additional components, for which specific drivers will be required. Those drivers are part of the source tree, and Google actually provides several kernel trees, for the chipsets shown in Table 3-3:

Table 3-3: Chipsets Devices, and board names for Google devices

Project Name	Chipset vendor	Devices (board names)
goldfish (M:Ranchu)	N/A	Android emulator
msm	Qualcomm MSM	Nexus One, Nexus 4, Nexus 5 (hammerhead)
omap	TI OMAP	Pandaboard, Galaxy Nexus, Glass (notle)
samsung	Samsung Hummingbird	Nexus S
tegra	NVidia Tegra	Motorola Xoom, Nexus 7 & 9, NVidia Shield
exynos	Samsung Exynos	Nexus 10 (manta)

Google's devices are commonly known by their piscine board project name, and their kernel binaries are available via git at `https://android.googlesource.com/device` subtrees. The kernel sources (which naturally must remain open) can similarly be obtained via `git` using

```
git clone https://android.googlesource.com/kernel/platform_project.git
```

as described further in Android Documentation[6]. Aside from Table 3-3, a good way of figuring out which branch a device's kernel is derived from is by looking at its strings and symbols.

The Device Tree (ARM)

Most ARM kernels also rely on the presence of a device tree file to provide the kernel with the hardware device definitions. This file provides a hierarchical view of devices by connection, and enables the kernel to boot the approriate drivers for them. The device tree is commonly appended to the end of the kernel image, but may at times reside in a separate partition.

The device tree format is a binary blob, identified by the magic value `0xd00dfeed`. A complete discussion of the device tree is beyond the scope of this book (it is an ARM feature, and not specific to Android). The format is well documented in the ePAPR specification[7], and a presentation by Thomas Pettazoni[8]. You can use the `imgtool` utility to extract the device tree from your kernel image. This is shown in the following experiment

Experiment: Retrieving the device tree from a boot.img

The `imgtool`, in addition to unpacking a boot.img and extracting its kernel and ramdisk, will also automatically extract the device tree component of the kernel image, if found. The extracted file, however, is in a binary format (.dtb, identifiable by its magic header of `0xd00dfeed`). To decompile the device tree, you will need to use the `dtc` utility, which is part of the `device-tree-compiler` package on Ubuntu, or `dtc` package on Fedora. Once installed, it's a simple matter to decompile the file and obtain the textual .dts file:

Output 3-7: Extracting and decompiling a device tree from the Nexus 5 boot.img

```
morpheus@Forge (~/Android/Book) % imgtool Images/hammerhead-kot49h/recovery.img extract
Part                Size                Pages     Addr
Kernel:             8331496
Ramdisk:            1095649
Secondary:          0
Tags:           2700000
Flash Page Size: 2048 bytes
Name:
CmdLine: console=ttyHSL0,115200,n8 androidboot.hardware=hammerhead user_debug=31 maxcpus=2
Extracting contents to directory: extracted/
Looking for device tree...  Found DT Magic @6bb8e8 - will extract tree(s)
Found GZ Magic at offset 18612 - will extract kernel
morpheus@Forge (~/Android/Book) % cd extracted
#
# Get the device tree compiler package. This example is on a Fedora system, so use yum
# On Ubuntu, you'd likely use "sudo apt-get install device-tree-compiler"
#
morpheus@Forge (~/../extracted) % yum whatprovides dtc
Loaded plugins: langpacks, refresh-packagekit
dtc-1.4.0-2.fc20.x86_64 : Device Tree Compiler
Repo        : fedora
morpheus@Forge (~/../extracted) % sudo yum install dtc
--> Running transaction check
---> Package dtc.x86_64 0:1.4.0-2.fc20 will be installed
...
Installed:
  dtc.x86_64 0:1.4.0-2.fc20
Complete!
morpheus@Forge (~/../extracted) % dtc -I dtb devicetree.dtb  -O dts -o devicetree.dts
morpheus@Forge (~/../extracted) % more devicetree.dts
/dts-v1/;
/ {
        #address-cells = <0x1>;
        #size-cells = <0x1>;
        model = "LGE MSM 8974 HAMMERHEAD";
        compatible = "qcom,msm8974";
        interrupt-parent = <0x1>;
        qcom,msm-id = <0x7e 0x96 0x20002 0xb>;
        chosen {
        };
        aliases {
                spi0 = "/soc/spi@f9923000";
                spi7 = "/soc/spi@f9966000"; # ...
```

The RAM disk

The second component of the boot or recovery image is the initial RAM disk, often referred to as the initrd. The RAM disk provides an initial filesystem, used as the rootfs when booting up the OS. It's pre-loaded by the bootloader into RAM alongside the kernel (hence the name), and enables quick access, without any special drivers. This is not a Linux specific feature - other UN*X have also been known to use it, most notably iOS (wherein it is contained in the .ipsw system image, alongside the kernelcache).

Traditionally, the initramfs is often used to provide device-specific drivers, which the kernel requires for operation. This enables the Linux distributer to provide a generic, relatively compact kernel, and package the necessary drivers (which vary between hardware configurations) into a separate file, created during the initial install process. To get around the chicken-and-egg case wherein drivers are required for storage access, critical ones are packaged into the initramfs, which the kernel can then access directly in RAM. It also contains the startup program (/init), which the kernel loads as PID 1, enabling early startup operations which require user mode (for example, loading modules).

Once the RAM Disk operation is done, Linux normally discards it, in favor of the on-disk filesystem (a process often refers to as "pivoting root"). In Android, however, the initramfs is kept in memory, and provides the root filesystem. This is useful since the files are consulted often, and the memory footprint is fairly small. It also makes tampering with the rootfs harder, since the boot image is signed.

Linux supports two file formats for the RAMdisk - initrd (ext4 filesystem image) and initramfs (CPIO archive). The latter is commonly used, though it is commonly referred to as an initrd. The CPIO archive makes for a simple format with very little RAM requirements. To further save space, the archive is gzipped (The kernel already has zlib support, which it needs to decompress itself).

Every vendor is free to build the RAM disk as it sees fit, though most take the Android emulator image as a baseline - which is why it's not surprising to see init.goldfish.rc in some. Most RAM disks are therefore very similar. Further, for a given device, the boot and recovery RAM disks will be largely the same, with the execption of subtle modifications in the /init.rc file, which controls system startup. In a recovery RAM disk, the /init.rc omits the standard set of services, leaving adbd, and /sbin/recovery.

As specified, the kernel is packaged along with the ramdisk into a separate partition. This has a very important design rationale behind it: By packing the two together, a single digital signature may be applied on both, securing two for the price of one against tampering. That the kernel is a critical component of the system should be obvious, but the RAM disk, as well, is quite important: It controls system start up by providing /init and its corresponding /init...rc files. /init starts up as root, and is responsible for starting up all the other system components. Getting root access to a device is as simple as modifying the /init.rc file - but cannot be done without violating the digital signature.

Experiment: Unpacking the RAM disk

Using the imgtool utility, you can obtain the RAM disk from either the boot or recovery images. Unpacking it is a simple matter with the standard utilities - gunzip and cpio, as shown here. If you don't possess a boot image, you can try this on the Android Emulator images.

Output 3-8: Unpacking a RAM disk from the recovery image using imgtool

```
morpheus@Forge (~/Android/Book) % imgtool Images/hammerhead-kot49h/recovery.img extract
Part                Size                Pages      Addr
Kernel:             8331496
Ramdisk:            1095649
Secondary:          0
Tags:           2700000
Flash Page Size: 2048 bytes
CmdLine: console=ttyHSL0,115200,n8 androidboot.hardware=hammerhead user_debug=31 maxcpus=2
Extracting contents to directory: extracted/
#
# File is easily recognizable as a gzip archive
morpheus@Forge (~/Android/Book) % file extracted/ramdisk
extracted/ramdisk: gzip compressed data, from Unix
#
# Create a temporary directory to extract archive to
morpheus@Forge (~/Android/Book) % mkdir tmp;cd tmp
# If your system doesn't have gzcat, you can use gunzip first
# cpio switches: -i(nput) -d(irectory) -v(erbose)
morpheus@Forge (~/Android/Book/tmp) % gzcat ../extracted/ramdisk | cpio -ivd
charger
..
init.rc
sbin/adbd
sbin/healthd   # Starting with KitKat, healthd is also in RAM disk
sbin/recovery  # Note recovery binary - instrumental for restore/update
```

For a description of initramfs contents (i.e. the Root filesystem), see Table 2-6 in the previous chapter. As a further experiment, you can compare the ramdisks of the boot and recovery images.

System and Data Partitions

The system and data partitions were discussed in the previous chapter. Vendors can provide these images in any format they see fit, since they can use proprietary binaries to flash, as well. Most vendors use fastboot, so their images are likely to follow the simg (sparse image) format used by Google's own images. Utilities for handling this file format can be found in the AOSP under system/core/libsparse.

Sparse images begin with a small (28-byte) header containing metadata about the image. The header format is shown in Table 3-4:

Table 3-4: The sparse image header

Offset	Length	Field
0	4	Magic value (0xed26ff3a)
4	4	Version (as Major + Minor). Currently 0x00000001.
8	2	Header Size. Always 0x001c (= 28)
10	2	Chunk Size
12	4	Block Size. Usually 0x1000 (4k) for Ext filesystems
16	4	Number of Blocks in filesystem
20	4	Number of Chunks in this file
24	4	Optional Checksum (usually zero)

Experiment: Mounting the Android system image on a host

Extracting a sparse image is a simple matter using the `imgtool` provided on the book's companion website. You can also compile the AOSP's `simg2img` from source. This is demonstrated on the system.img - the userdata and cache images are largely empty.

Output 3-9: Unpacking an Android system image

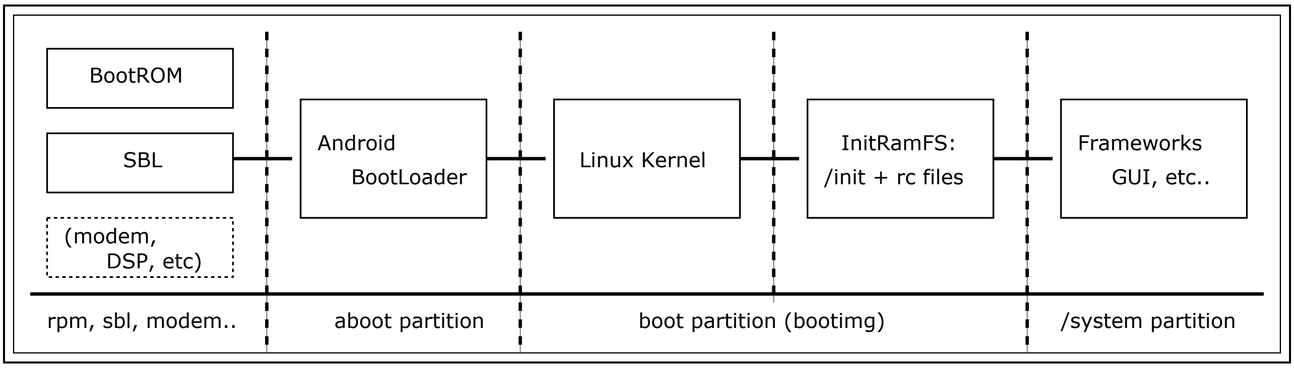
```
Forge (~/.../hammerhead-lpv79)$ file system.img
system.img: data    # unrecognized :-(
Forge (~/.../hammerhead-lpv79)$ od -A x -t x4 system.img | head -2
0000000 magic:  ed26ff3a version: 00000001 chnk/hdr: 000c001c blksize:  00001000
0000010 blocks: 00040000 chunks:  00000595 checksum: 00000000           0000cac1
# Using the simg_dump Python script you can get some details about the sparse image:
#
Forge (~/.../hammerhead-lpv79)$ $SDK_ROOT/system/core/libsparse/simg_dump.py system.img
system.img: Total of 262144 4096-byte output blocks in 1429 input chunks.
# Compile simg2img. Yes, you can do that through the NDK build system. But..
# this works without having to configure anything (or even download the NDK)
Forge (~/.../hammerhead-lpv79)$ cd $SDK_ROOT/system/core/libsparse
Forge (/.../core/libsparse)$ gcc backed_block.c output_file.c sparse*.c simg2img.c \
>                             -I./include -lz -o simg2img
sparse_read.c:122:10: warning: implicit declaration of .. #... whatever..
# We now have simg2img! Go back and unpack image:
#
Forge (/.../core/libsparse)$ cd -
Forge (~/.../hammerhead-lpv79)$ $SDK_ROOT/system/core/libsparse/simg2img system.img system.ex
# simg2img isn't the talkative type - but will generate the output file
#
Forge (~/.../hammerhead-lpv79)$ file system.ext4
system.ext4: Linux rev 1.0 ext4 filesystem data (extents) (large files)     # Success!
# Compare the sparse image to the full, raw image: Note decent savings
#
Forge (~/.../hammerhead-lpv79)$ ls -lh system.img system.ext4
-rw-r--r--  1 morpheus  staff   1.0G Jun 27 07:09 system.ext4
-rw-r--r--@ 1 morpheus  staff   668M Jan  1  2009 system.img
# Now we can mount filesystem as a loop device (need to be root for this step)
#
Forge (~/.../hammerhead-lpv79)$ sudo mount -o loop system.ext4 /mnt
Forge (~/.../hammerhead-lpv79)$ ls /mnt
app   build.prop   fonts      lib         media      recovery-from-boot.p     vendor
bin   etc          framework  lost+found  priv-app   usr                      xbin
```

Android emulator images (found in $SDK_ROOT/system-images) are simply raw filesystem images, and so you can loop mount them directly. Later in this chapter we show how you can use this experiment in reverse, to modify the system images in preparation for flashing to the device.

The Boot Process

With all the components of the system images dissected, we can now turn our attention to the actual boot process. Though device-dependent, the boot process can be generalized to the following stages:

Figure 3-1: The generalized Android Boot Process



Firmware Boot

The device's firmware is akin to BIOS (or, nowadays, EFI) on PCs. Its main component is a boot ROM, which is supplied by the hardware vendor. The boot ROM, being a component of read-only-memory, is quite often a very small component, and contains only the initial boot sequence, which initializes hardware components to the bare minimum required for usability. The boot ROM then proceeds to load a secondary boot loader (sbl), which - being software - can afford to be of a bigger size, and therefore perform more complicated initialization tasks (for example, displaying a startup graphic image).

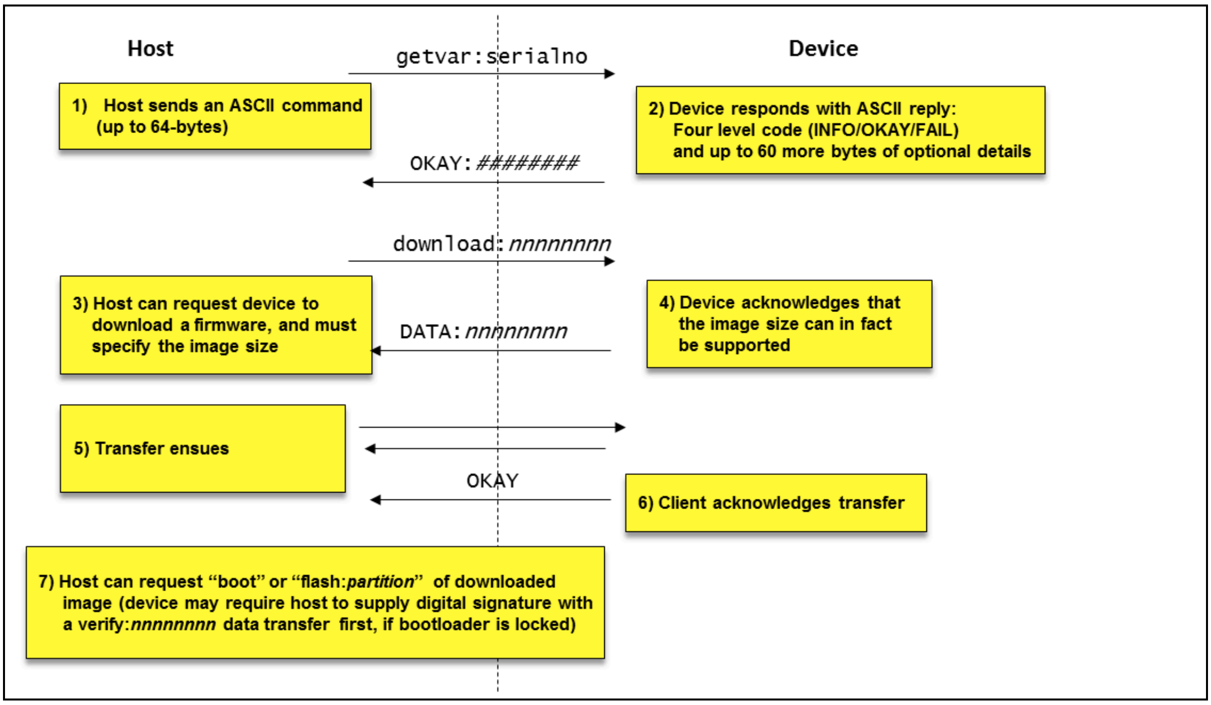
Unlike a PC, a mobile device's processor is not a single CPU, as would be the case with an Intel or AMD processor, but a complete system-on-chip (SoC). In practice, this means that there are several processors working in tandem, of which the application processor is only one. The Qualcomm SnapDragon processors, for example, contain no less than four sub-processor: RPM (Resource/Power Management), Krait (the application processor), Adreno (The graphics processor - CPU) and Hexagon (the Digital Signal Processor - DSP). MSM chipsets, therefore, involve a particularly lengthy boot process, wherein the boot ROM provides the primary boot loader (PBL), to initialize the RPM processor. This, in turn, loads the secondary boot loader (sbl), which is itself broken into three parts (sbl1→sbl2→sbl3). The parts load and authenticate one another in an intricate choreography*, which also involves code from the rpm and tz (ARM TrustZone) partition. The application processor then boots up the other components, and executes the application boot - which is where Android's boot loader comes into play.

* - Said choreography is actually quite complex, and entirely undocumented outside Qualcomm confidential documents, some of which have been leaked. For obvious reasons, this work cannot go into detail, but the information gleaned from said documents, as well as a very detailed discussion, can be found at the XDA-Developers forum[9a], with a plethora of information in thread 24100141[9b]

The FastBoot Protocol

Most Android bootloaders support the "FastBoot" protocol, which Google makes available as part of Android itself*. The FastBoot protocol is a simple, text-based protocol, which is meant to be used over a USB channel between the device and the host. It's not exceptionally fast in terms of performance (e.g. it is synchronous), so the name likely applies to it being very easy (and hence, fast) to implement. Figure 3-2 shows the message passing between host and device:

Figure 3-2: The fastboot choreography


Host
Device
getvar:serialno
1) Host sends an ASCII command (up to 64-bytes)
2) Device responds with ASCII reply: Four level code (INFO/OKAY/FAIL) and up to 60 more bytes of optional details
OKAY:########
download:nnnnnnnn
3) Host can request device to download a firmware, and must specify the image size
4) Device acknowledges that the image size can in fact be supported
DATA:nnnnnnnn
5) Transfer ensues
OKAY
6) Client acknowledges transfer
7) Host can request "boot" or "flash:partition" of downloaded image (device may require host to supply digital signature with a verify:nnnnnnnn data transfer first, if bootloader is locked)

The current protocol version at the time of writing (0.4) is fairly well detailed in system/core/fastboot/fastboot_protocol.txt. Table 3-5 lists the commands understood by the `fastboot` host-side binary, and their corresponding protocol messages:

Table 3-5: Default fastboot commands

Command Line	Protocol command	Description
flash <partition> [<filename>]	download:%08x, flash:*partition*	write a file to a flash partition
flash:raw boot <kernel> [<ramdisk>]		create bootimage and flash it
flashall		flash boot + recovery + system
update		reflash device from update.zip
erase <partition>	erase:*partition*	erase a flash partition
format <partition>		format a flash partition
getvar <variable>	getvar:*variable*	display a bootloader variable
boot <kernel> [<ramdisk>]	download:%08x,boot	download and boot kernel
devices	getvar:serialno	list all connected devices
continue	continue	continue with autoboot
reboot	reboot	reboot device normally
reboot-bootloader	reboot-bootloader	reboot device into bootloader
oem [command [args]]	command[:args]	send an OEM extension command

* - Vendors are not required to support FastBoot, and may support their own boot-loader protocols instead of, or in addition to FastBoot. An example of that can be found in Samsung's ODIN, and Amazon's bootloader.

Experiment: Using Fastboot

The Android SDK provides the `fastboot` command, which is a simple but complete implementation of the protocol. To see if your device's bootloader supports fastboot, you first need to force it to halt at the bootloader stage. Rather than start up with the magic button combination (which is tricky on some devices), you can use `adb reboot bootloader`. Your device will restart into the bootloader, and - if it supports fastboot, will be visible by "fastboot devices", by its serial number - similar to adb:

Output 3-10: Output from `fastboot devices`

```
morpheus@Forge (~)% adb reboot bootloader
morpheus@Forge (~)% $SDK_ROOT/platform-tools/fastboot devices
FA43BSF00073    fastboot
```

At this point, the device should present the bootloader UI, and you should be able to independently navigate the boot loader menus using the physical buttons (usually VOLUME UP/DOWN, and POWER to select). You can also use any one of the commands in Table 3-5, though because most are potentially dangerous (unless you know what you're doing), you can try "getvar all", to list all the bootloader variables. These will be different on every device, and the output from the HTC One M8 will show this:

Output 3-11: Output from an HTC-One M8 `fastboot getvar`

```
morpheus@Forge (~)% $SDK_ROOT/platform-tools/fastboot getvar all
(bootloader) version: 0.5
(bootloader) version-bootloader: 3.16.0.0000
(bootloader) version-baseband: 0.89.20.0222
(bootloader) version-cpld: None
(bootloader) version-microp: None
(bootloader) version-main: 1.12.605.11
(bootloader) version-misc: PVT SHIP S-ON # HTC: S(ecurity)-[ON/OFF]
(bootloader) serialno: Phone Serial #
(bootloader) imei: Device ID
(bootloader) imei2: Not Support
(bootloader) meid: Device ID (same as IMEI)
(bootloader) product: m8_wlv
(bootloader) platform: hTCBmsm8974       # Note msm chipset
(bootloader) modelid: 0P6B20000
(bootloader) cidnum: VZW__001
(bootloader) battery-status: good
(bootloader) battery-voltage: 0mV
(bootloader) partition-layout: Generic
(bootloader) security: on
(bootloader) build-mode: SHIP
(bootloader) boot-mode: FASTBOOT
(bootloader) commitno-bootloader: 3a4162f9
(bootloader) hbootpreupdate: 11
(bootloader) gencheckpt: 0
all: Done!
```

The really interesting part of fastboot, however, is in the oem extension: try `fastboot oem h` to obtain a list of all commands (which will surely vary between devices). The commands are extremely versatile and useful - HTC supports dmesg (to get bootloader log), get_temp (to read temperature sensors), read/writeusername (personalize the phone), read/writecid (carrier ID) and read/writeimei, which can be used by carriers to configure the phone for their networks. As noted previously, some devices - notably the Nexus 5 and NVidia's Shield - support "oem unlock" - which enables you to unlock the bootloader and free your phone, to load any custom firmware image.

Kernel Boot

Android's kernel boot process isn't much different than that of Linux. The bad news, however, is that the latter is somewhat of a moving target, with kernel version updates often adding or removing components, as well as variance between platforms. This section therefore aims to provide a high level overview of the kernel startup as it is implemented in the 3.x line of kernels. You might want to follow along with a device-specific kernel source tree.

Recall, that the boot loader is responsible for locating the kernel zImage and RAM disk. Once both are resident in memory, control is transferred to the zImage's entry point. The kernel is compressed at this time, so the entry point - implemented in arch/*architecture*/boot/compressed/head.S (head_32.S or head_64.S on x86) is responsible for calling `decompress_kernel`, which displays the familiar "Uncompressing Linux... done, booting the kernel" message, and transfer control to the "real" entry point. This is, again, an architecture specific function, implemented in assembly (`stext` in arch/arm/kernel/head.S or `startup_[32|64]` in arch/x86/kernel/x86/kernel/head_[32|64].S, respectively). What follows is a low level setup of the MMU and page tables (switching to virtual addressing), before control is transferred to the kernel's main function, `start_kernel`.

The `start_kernel` function is architecture independent, and is thus implemented in init/main.c. It is fairly well writ, in the sense that it has almost no variables, and most of the startup is performed by calling functions. To make a (very) long story short, `start_kernel` initializes all the critical framework using the specialized functions, then calls `rest_init`, which - as the name implies - initializes everything else. This function spawns the `kernel_init` thread, which is responsible for initializing the various subsystems.

With so many subsystems to initialize, the kernel code would be terribly long and messy. Instead, the **initcall** mechanism provides an elegant solution: it defines 8 initialization levels, which the `kernel_init` thread calls on (via `do_initcalls()`, in `do_basic_setup()`) in order, as shown in Table 3-6:

Table 3-6: The initcall levels

#	Level	Notes
0	early	Used to spawn initial helper threads, such as RCU, SoftIRQs, and workqueues
1	core	Used for "core" subsystems, such as binfmt and sockets
2	postcore	Used by bdi (block device flush threads) and kobjects
3	arch	Architecture dependent initialization
4	subsys	General subsystems, such as bio, crypto and sound
5	fs	Used by the VFS layer, for filesystem support
6	device	Used by drivers, and general modules. The `module_init` macro maps to this level.
7	late	Very last stage - Advanced memory management, oops handling and more

The idea is similar to the classic user-mode init's "runlevel" concept, which used run-levels to group subsystem startup scripts. The `initcalls` emulate this idea, by allowing subsystems to register their initialization functions with a `*level*_initcall` macro, which in turn will be invoked when `kernel_init` processes the level. Once all init levels have been iterated through, the kernel initialization is complete.

The messages output during the kernel boot can be seen using dmesg(1), but because the kernel uses a ring buffer, it will most likely be partially overwritten by the time you get the root shell necessary to run this command on a device. (The size of the kernel ring buffer can be configured when the kernel is built).

Rather than follow the sequence step by step, the following listing maps the `dmesg` output to the startup functions which emit them. The **bold** lines are architecture independent, so you should be able to see them (albeit with slightly different values) on x86 and ARM alike.

Listing 3-2: An annotated `dmesg` output from the Android Emulator

```
<6>Booting Linux on physical CPU 0  # smp_setup_processor_id
<6>Initializing cgroup subsys cpu # cgroup_init_early
<5>Linux version 3.4.0-gd853d22 (nnk@nnk.mtv.corp.google.com) ... # pr_notice("%s",
<4>CPU: ARMv7 Processor [410fc080] revision 0 (ARMv7), cr=10c53c7d
<4>CPU: PIPT / VIPT nonaliasing data cache, VIPT nonaliasing instruction cache
<4>Machine: Goldfish
<5>Truncating RAM at 00000000-7fffffff to -2f7fffff (vmalloc region overlap).
<4>Memory policy: ECC disabled, Data cache writeback
<7>On node 0 totalpages: 194560
<7>free_area_init_node: node 0, pgdat c04a7394, node_mem_map c084f000
<7>  Normal zone: 1520 pages used for memmap
<7>  Normal zone: 0 pages reserved
<7>  Normal zone: 193040 pages, LIFO batch:31
<7>pcpu-alloc: s0 r0 d32768 u32768 alloc=1*32768
<7>pcpu-alloc: [0] 0
<4>Built 1 zonelists in Zone order, mobility grouping on. ..  # build_all_zonelists
<5>Kernel command line: qemu.gles=0 qemu=1 console=ttyS0 ...# pr_notice("Kernel comm
<6>PID hash table entries: 4096 (order: 2, 16384 bytes) # pidhash_init()
<6>Dentry cache hash table entries: 131072 (order: 7, 524288 bytes) # vfs_caches_in
<6>Inode-cache hash table entries: 65536 (order: 6, 262144 bytes)
<6>Memory: 760MB = 760MB total         # mem_init();
<5>Memory: 764912k/764912k available, 13328k reserved, 0K highmem
<5>Virtual kernel memory layout:
<5>  ...
<6>NR_IRQS:256  #early_irq_init()
<6>sched_clock: 32 bits at 100 Hz, resolution 10000000ns, .. # sched_clock_init()
<6>Console: colour dummy device 80x30 # console_init()
<6>Calibrating delay loop... 412.87 BogoMIPS (lpj=2064384)  # calibrate_delay()
<6>pid_max: default: 32768 minimum: 301
<6>Security Framework initialized  #  security_init()
<6>SELinux:  Initializing.         # security_initcall(selinux_init);
<7>SELinux:  Starting in permissive mode
<6>Mount-cache hash table entries: 512 # vfs_caches_init()
<6>Initializing cgroup subsys debug    # cgroup_init();
<6>Initializing cgroup subsys cpuacct
<6>Initializing cgroup subsys freezer
<6>CPU: Testing write buffer coherency: ok   # check_bugs (macro)
#
# .. from this point on, we're at rest_init - and in kernel_init thread
# .. this calls do_basic_setup(), which in turn calls do_initcalls()
# .. Note ordering of output is consistent with initcall levels
#
<6>Setting up static identity map for  0x35e610.. # early_initcall(init_static_idma
<6>NET: Registered protocol family 16  # core_initcall(netlink_proto_init);
<6>bio: create slab  at 0              # subsys_initcall(init_bio);
<6>Switching to clocksource goldfish_timer
<6>NET: Registered protocol family 2      # fs_initcall(inet_init);
<6>IP route cache hash table entries: 32768 (order: 5, 131072 bytes)
<6>TCP established hash table entries: 131072 (order: 8, 1048576 bytes)
<6>TCP bind hash table entries: 65536 (order: 6, 262144 bytes)
<6>TCP: Hash tables configured (established 131072 bind 65536)
<6>TCP: reno registered
<6>UDP hash table entries: 512 (order: 1, 8192 bytes)
<6>UDP-Lite hash table entries: 512 (order: 1, 8192 bytes)
<6>NET: Registered protocol family 1
<6>RPC: Registered named UNIX socket transport module.  # fs_initcall(init_sunrpc);
<6>RPC: Registered udp transport module.
<6>RPC: Registered tcp transport module.
<6>RPC: Registered tcp NFSv4.1 backchannel transport module.
<6>Trying to unpack rootfs image as initramfs... # rootfs_initcall(populate_rootfs)
<6>Freeing initrd memory: 312K
<4>..   goldfish_new_pdev IRQ enumeration...
<4>..   goldfish_pdev_worker interrupt registration..
<6>audit: initializing netlink socket (disabled) #   initcall(audit_init);
<5>type=2000 audit(0.270:1): initialized
<6>Installing knfsd (copyright (C) 1996 okir@monad.swb.de).
<6>fuse init (API version 7.18)  # module_init(fuse_init);
<7>yaffs: yaffs built Jul  9 2013 17:46:43 Installing.
<6>msgmni has been set to 1494
<7>SELinux:  Registering netfilter hooks
```

Listing 3-2: An annotated `dmesg` output from the Android Emulator (cont.)

```
<6>io scheduler noop registered           #
<6>io scheduler deadline registered       # elv_register
<6>io scheduler cfq registered (default)  #
<4>allocating frame buffer 1080 * 1920, got   (null)
<4>goldfish_fb: probe of goldfish_fb.0 failed with error -12
<6>console [ttyS0] enabled
<6>brd: module loaded
<6>loop: module loaded
<6>nbd: registered device at major 43
<4> ... goldfish specific stuff..
<6>tun: Universal TUN/TAP device driver, 1.6
<6>tun: (C) 1999-2004 Max Krasnyansky
<4>smc91x.c: v1.1, sep 22 2004 by Nicolas Pitre
<4>eth0: SMC91C11xFD (rev 1) at fe013000 IRQ 13 [nowait]
<4>eth0: Ethernet addr: 52:54:00:12:34:56
<7>eth0: No PHY found
<6>mousedev: PS/2 mouse device common for all mice
<4>*** events probe ***
<4>events_probe() addr=0xfe016000 irq=17
<4>events_probe() keymap=qwerty2
<6>input: qwerty2 as /devices/virtual/input/input0
<6>goldfish_rtc goldfish_rtc: rtc core: registered goldfish_rtc as rtc0
<6>device-mapper: uevent: version 1.0.3
<6>device-mapper: ioctl: 4.22.0-ioctl (2011-10-19) initialised: dm-devel@redhat.com

<6>ashmem: initialized                     ashmem_init()
<6>logger: created 256K log 'log_main'     device_initcall(logger_init);
<6>logger: created 256K log 'log_events'
<6>logger: created 256K log 'log_radio'
<6>logger: created 256K log 'log_system'

<6>Netfilter messages via NETLINK v0.30.
<6>nf_conntrack version 0.5.0 (11956 buckets, 47824 max)
<6>ctnetlink v0.93: registering with nfnetlink.
<6>NF_TPROXY: Transparent proxy support initialized, version 4.1.0
<6>NF_TPROXY: Copyright (c) 2006-2007 BalaBit IT Ltd.
<6>xt_time: kernel timezone is -0000
<6>ip_tables: (C) 2000-2006 Netfilter Core Team
<6>arp_tables: (C) 2002 David S. Miller
<6>TCP: cubic registered
<6>NET: Registered protocol family 10
<6>ip6_tables: (C) 2000-2006 Netfilter Core Team
<6>IPv6 over IPv4 tunneling driver
<6>NET: Registered protocol family 17
<6>NET: Registered protocol family 15
<6>8021q: 802.1Q VLAN Support v1.8
<6>VFP support v0.3: implementor 41 architecture 3 part 30 variant c rev 0
<6>goldfish_rtc goldfish_rtc: setting system clock to 2014-04-16 01:15:32 UTC (139761
0932)
<6>Freeing init memory: 148K
<7>SELinux: 512 avtab hash slots, 1319 rules.
<7>SELinux: 512 avtab hash slots, 1319 rules.
<7>SELinux:  1 users, 2 roles, 288 types, 1 bools, 1 sens, 1024 cats
<7>SELinux:  84 classes, 1319 rules
<7>SELinux:  Completing initialization.
<7>SELinux:  Setting up existing superblocks.
<7>SELinux: initialized (dev sysfs, type sysfs), uses genfs_contexts
    ... iterates over all mounted filesystems ...
<7>SELinux: initialized (dev sysfs, type sysfs), uses genfs_contexts
<5>type=1403 audit(1397610932.610:2): policy loaded auid=4294967295 ses=4294967295
<4>SELinux: Loaded policy from /sepolicy
<5>type=1404 audit(1397610932.620:3): enforcing=1 old_enforcing=0 auid=4294967295 ses
=4294967295
<4>init (1): /proc/1/oom_adj is deprecated, please use /proc/1/oom_score_adj instead.
```

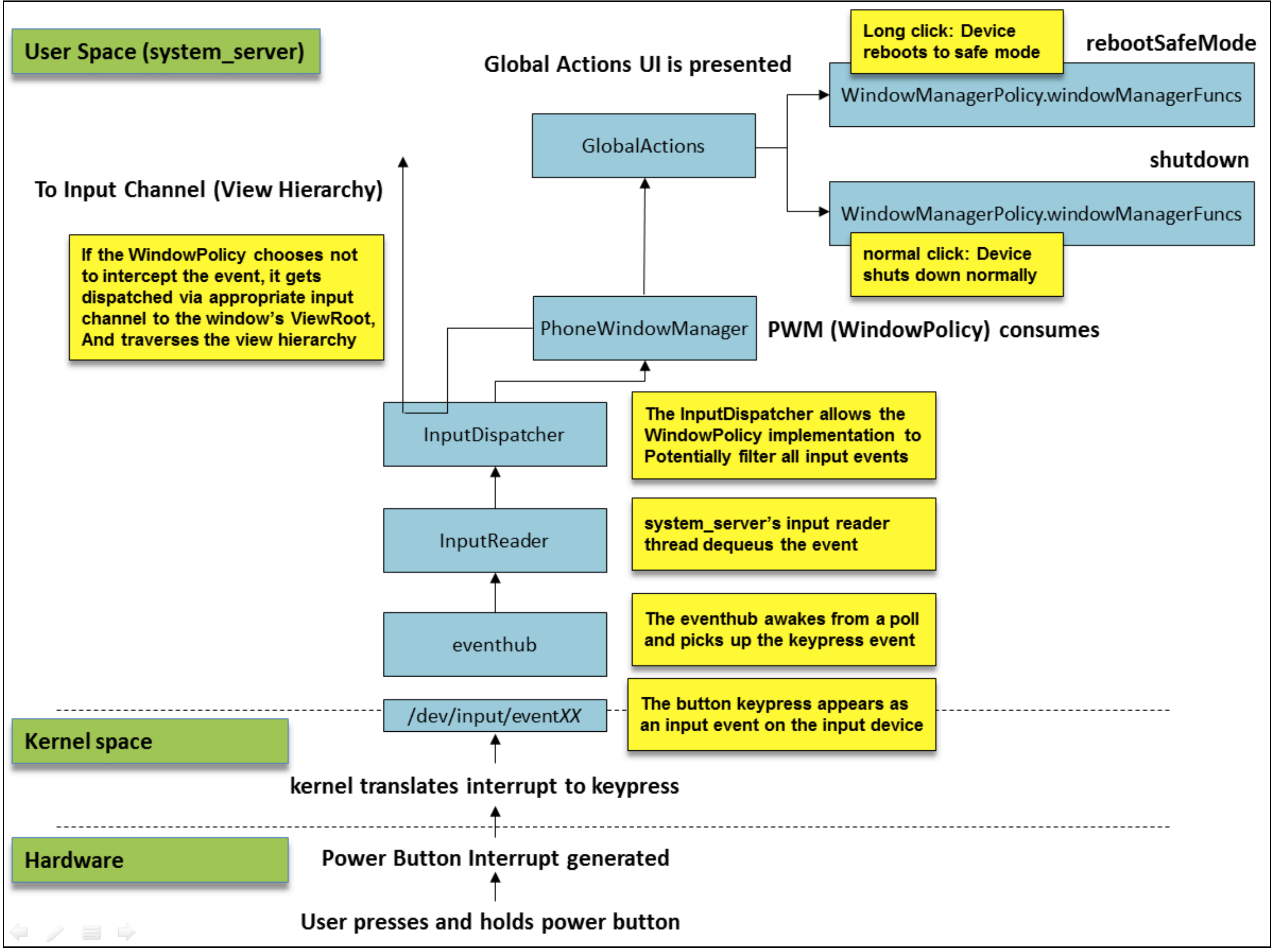
When the kernel startup thread is done, it emerges into user mode as PID 1 - `/init`. We discuss `/init` in detail in the next chapter, alongside the various Android-specific services it launches.

Shutdown & Reboot

Most users keep their devices on, but there are the occasional times when a user decides to shutdown or reboot the device. In those cases, the decision usually starts with holding the power button down for a few seconds, which pops up a confirmation dialog. If the user opts to actually shut down or reboot the phone (as opposed to, say, Airplane Mode), the phone reboots.

Behind the scenes, however, the process is quite lengthy, and involves a rather complicated choreography, as shown in Figure 3-3 (which should be read from bottom to top):

Figure 3-3: Pressing the shutdown button


User Space (system_server)
Global Actions UI is presented
Long click: Device reboots to safe mode
rebootSafeMode
WindowManagerPolicy.windowManagerFuncs
GlobalActions
shutdown
To Input Channel (View Hierarchy)
WindowManagerPolicy.windowManagerFuncs
normal click: Device shuts down normally
If the WindowPolicy chooses not to intercept the event, it gets dispatched via appropriate input channel to the window's ViewRoot, And traverses the view hierarchy
PhoneWindowManager
PWM (WindowPolicy) consumes
InputDispatcher
The InputDispatcher allows the WindowPolicy implementation to Potentially filter all input events
InputReader
system_server's input reader thread dequeus the event
eventhub
The eventhub awakes from a poll and picks up the keypress event
/dev/input/eventXX
The button keypress appears as an input event on the input device
Kernel space
kernel translates interrupt to keypress
Hardware
Power Button Interrupt generated
User presses and holds power button

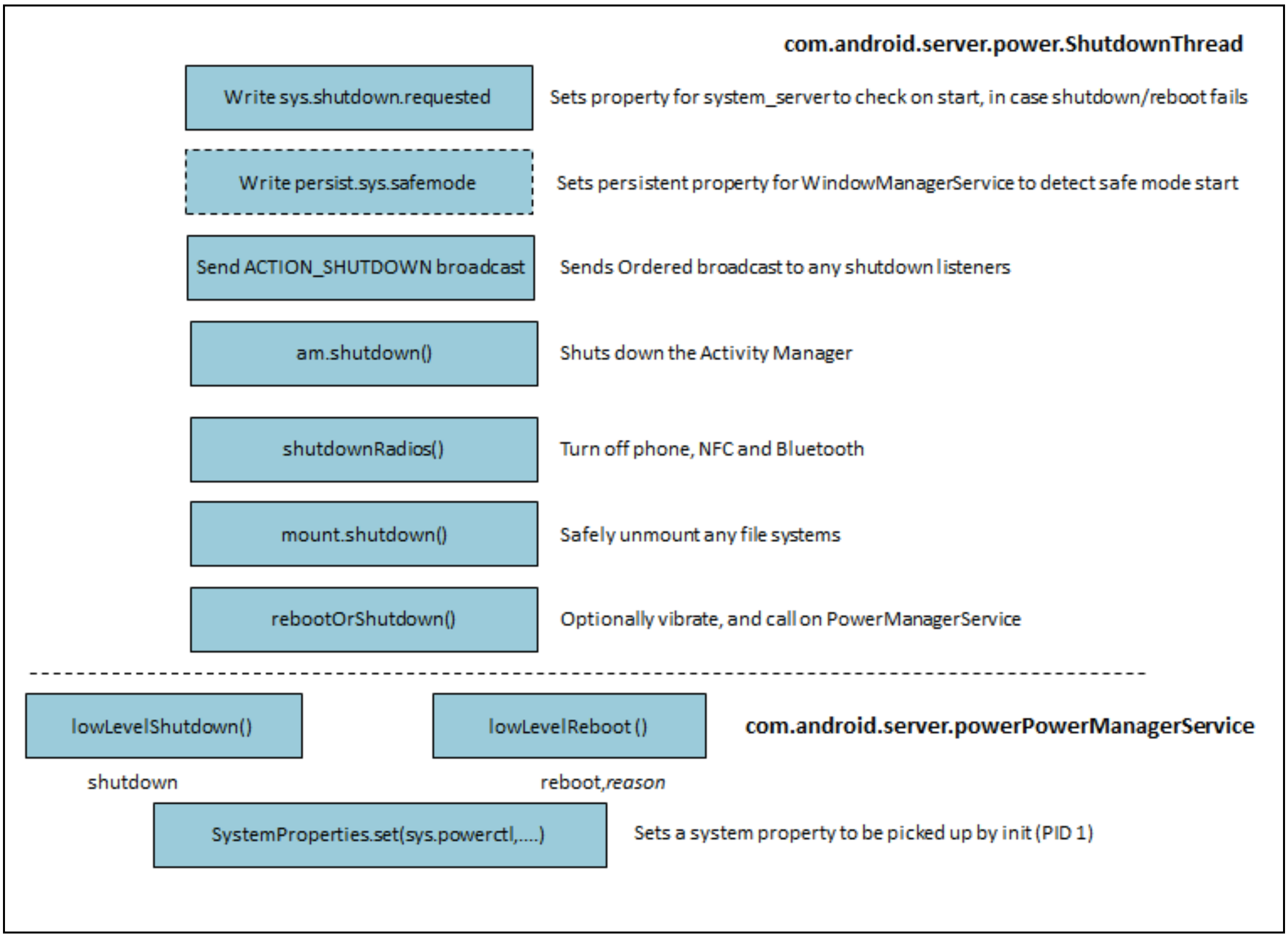
The button press generates an interrupt, which is picked up by the Linux kernel. The interrupt is translated by the kernel to an input event, which is then propagated to the Android runtime as an `EV_KEY/KEY_POWER/DOWN`. As with all other events, this is picked up by the Android's `InputReader` and `InputDispatcher` duo (both `system_server` threads), with the latter passes the event to the `WindowPolicy` object. The default Policy object (com.android.internal.policy.impl.PhoneWindowManager) intercepts the key if held for a sufficiently long period (which is defined in the `ViewConfiguration`'s `GLOBAL_ACTIONS_KEY_TIMEOUT` constant to be 500ms), and brings up the menu (through a call to `GlobalActions.showDialog()`).

* - The detailed discussion of Android's input architecture, from the low-level interrupt through the `InputManager`, `WindowManager`, Policy and Views can be found in Volume II.

If the user does opt to shut down, two roads diverge: A tap will proceed with a normal shutdown, but a long press will instead reboot to safe mode. Both tasks are handled by a dedicated `ShutdownThread`, whose `shutdownInner()` method optionally pops up a confirmation dialog before beginning the shutdown sequence.

If the user confirms the shutdown, `beginShutdownSequence()` sets two wakelocks, to keep the screen on during the process (for a better user experience). The shutdown thread can then be run. Its flow is shown in figure 3-4:

Figure 3-4: The flow of the Android shutdown sequence


com.android.server.power.ShutdownThread
Write sys.shutdown.requested
Sets property for system_server to check on start, in case shutdown/reboot fails
Write persist.sys.safemode
Sets persistent property for WindowManagerService to detect safe mode start
Send ACTION_SHUTDOWN broadcast
Sends Ordered broadcast to any shutdown listeners
am.shutdown()
Shuts down the Activity Manager
shutdownRadios()
Turn off phone, NFC and Bluetooth
mount.shutdown()
Safely unmount any file systems
rebootOrShutdown()
Optionally vibrate, and call on PowerManagerService
lowLevelShutdown()
lowLevelReboot ()
com.android.server.powerPowerManagerService
shutdown
reboot,*reason*
SystemProperties.set(sys.powerctl,....)
Sets a system property to be picked up by init (PID 1)

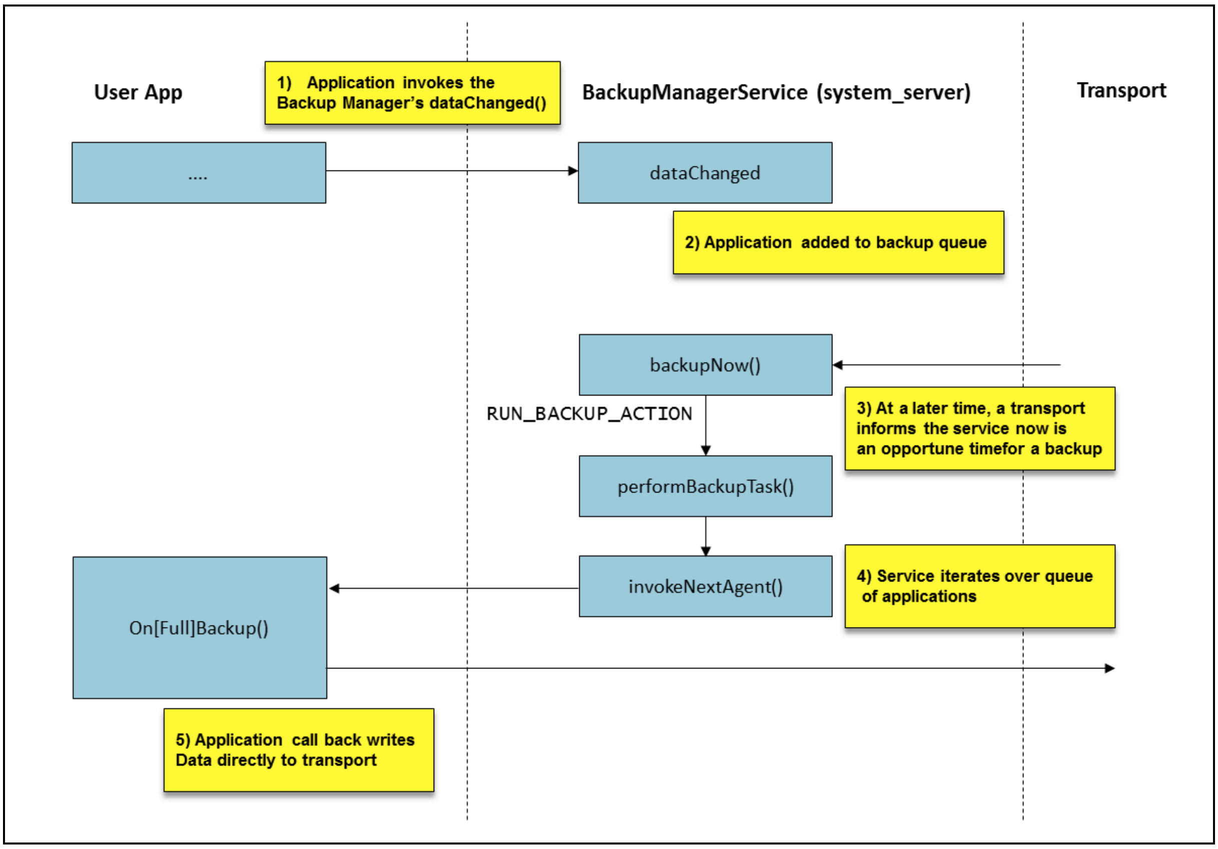
The last step of the shutdown - at the Linux native level - is performed by /init. As the process responsible for implementing system properties (q.v. Chapter 4), it picks up the setting of the `sys.powerctl` property to either **shutdown** or **reboot**,*reason*. The *reason* can be either `recovery` or `bootloader`. If the values seem familiar, it's because they are the very same ones used by `adb reboot`, which sets the very same property (as `ANDROID_RB_PROPERTY`) to the value chosen by the user. This way, all paths lead to /init, which in turn calls on `libcutils`' `android_reboot` function. This is nothing more than a wrapper to the kernel's `reboot()` system call, or `__reboot`, with the latter being a Linux specific invocation which allows the passing of the additional *reason*.

Application Backup & Restore

Just as humans grapple with sickness, Operating Systems face the risk of data corruption, or outright loss. Backup and Restore is therefore an important functionality which an operating system needs to provide. Applications need an ability to save and recover their configuration and data, and power users require a similar ability to backup the entire device to a well known, bootable configuration or a system checkpoint which can be rolled back to in case of calamity.

Indeed, as of API level 8, Android provides Applications with the `BackupManagerService`, a framework service which provides both per-application backups, as well as full backups of all apps. The internals of the framework service, including the Application Programming Interface it provides, is covered (along with the rest of the framework services) in Volume II. The backup architecture is quite elegant, delegating the responsibility of selecting which data is to be backed up to the application: The application notifies the backup manager when data has changed, and the backup manager adds the application to a queue.

Figure 3-5: A simplified view of the Android backup architecture


User App
1) Application invokes the Backup Manager's dataChanged()
BackupManagerService (system_server)
Transport
....
dataChanged
2) Application added to backup queue
backupNow()
RUN_BACKUP_ACTION
3) At a later time, a transport informs the service now is an opportune timefor a backup
performBackupTask()
invokeNextAgent()
4) Service iterates over queue of applications
On[Full]Backup()
5) Application call back writes Data directly to transport

At some later time, when the `BackupManagerService` gets a request to actually perform a backup, it creates a **backup set**, grouping together the one or more applications that were queued. For each application, it invokes the `onBackup()` callback. The `BackupService` passes the application a file descriptor in the callbacks, which the application is expected to use in order to write out (or read from) the backup data. The descriptor provided is connected to a **transport**, to which the application remains entirely oblivious. Data is written and read to the transport while leaving its implementation opaque - Data can be backed up either locally, or to "the Cloud" (i.e. Google's servers, or the device vendor's), but the choice of where to back up to remains at the system (or vendor) level. The common transports are shown in Table 3-7:

Table 3-7: Android Transports

Transport	Backs up to
`com.google.android.backup/.BackupTransportService`	Google's servers. Application needs a special API key to use this service
`com.android.server.enterprise/.EdmBackupTransport`	Enterprise backup, for managed devices
`android/com.android.internal.backup.LocalTransport`	Local backup, to device

Command line tools

From the perspective of the power user, there's a far simpler interface to backup and restore, in the form of two Dalvik upcall scripts, the `bmgr` and `bu` utilities. Both utilities require Java to facilitate communication with the `BackupManagerService`, which they perform over Binder (as discussed in Chapter 7). The `bmgr` utility is well documented[10], and explains its usage in detail when invoked with no arguments. A summary of its arguments is shown in Table 3-8:

Table 3-8: Commands and arguments understood by the `bmgr` upcall script

Command	Purpose
`backup` *`package`*	Mark package to be backed up on next run
`enable` *`0|1`*	Enable/disable the backup mechanism
`enabled`	Report if backup mechanism is enabled or disabled
`list` *`transports`*	List available transports, * specifying default (q.v. Table 3-7)
`list` *`sets`*	List restore sets
`transport` *`transportName`*	Set default transport
`restore` *`set [App]`*	Restore from a specific set - all apps, or only App specified.
`run`	Perform pending backups now
`wipe` *`transportName package`*	Erase all backups of *package* from *transportName*
`fullbackup` *`package`*	Perform a full backup of specified package

By contrast, the `bu` utility is entirely undocumented, and provides no user facing output, preferring instead to use the Android logging system. expects only one argument - `backup` or `restore`, but can handle quite a few switches when backing up. The switches expected by `bu` are shown in Table 3-9, with the defaults in **bold**:

Table 3-9: Switches understood by `bu backup`

Switch	Purpose
`-`**`[no]apk`**	Save or omit application .apk files
`-`**`[no]obb`**	Save or omit application opaque binary blobs (.obb) files
`-[no]shared`	Save or omit shared resources
`-[no]`**`system`**	Save or omit system applications in full backups
`-[no]widgets`	Save or omit widgets (default: -nowidgets)
`-[no]`**`compress`**	Compress backup
`-all`	Backup everything (requires user confirmation)

If the switches seem vaguely familiar, it's because they are the same as those passed to `adb backup` (though the latter does not advertise `-nocompress` as an option). Backups through `adb` are just direct invocations of the `bu` upcall script, which helps explain why it's not as user-friendly as `bmgr`.

Local backups

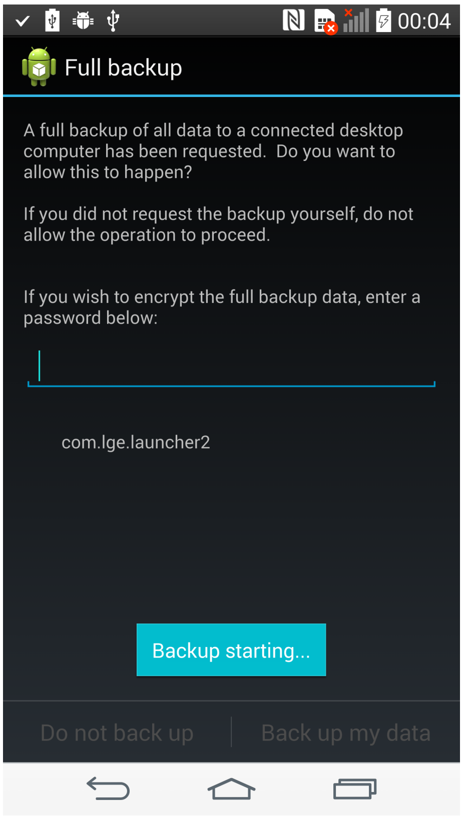
Using `adb backup -all`, triggers a full backup of all applications. Doing so causes the `bu` utility to call the `BackupManagerService`'s `fullBackup()` method, which pops up a customizable UI notification to the user.

The default notification UI activity is hardcoded to `com.android.backupconfirm`, and shown in Figure 3-6. Using a UI requires the device to be unlocked, adding a measure of security for users, by mitigating the chance a device could be taken for a minute or two, backed up and returned to the unwitting user. Another measure of security offers the user a chance to cancel the backup, as well as enter a password.

If the user approves the backup operation, a toast notification informs that the backup started, and the current package progress is displayed.

When backing up to a connected host, `adb` connects the other end of the transport file descriptor to a local file on the host, specified by the `-f` switch, or simply the `backup.ab` default. The backup file uses a proprietary format, which differs slightly if the backup is encrypted or not. The format's only documentation is embedded in the source of the [BackupManagerService]() class, but this provides comprehensive detail, as shown in Listing 3-3:

Figure 3-6: The default Backup UI (LG G3 running KitKat)



00:04

Full backup

A full backup of all data to a connected desktop computer has been requested. Do you want to allow this to happen?

If you did not request the backup yourself, do not allow the operation to proceed.

If you wish to encrypt the full backup data, enter a password below:

com.lge.launcher2

Listing 3-3: The format of an Android backup file

```
// Write the global file header.  All strings are UTF-8 encoded; lines end
// with a '\n' byte.  Actual backup data begins immediately following the
// final '\n'.
//
// line 1: "ANDROID BACKUP"
// line 2: backup file format version, currently "2"
// line 3: compressed?  "0" if not compressed, "1" if compressed.
// line 4: name of encryption algorithm [currently only "none" or "AES-256"]
//
// When line 4 is not "none", then additional header data follows:
//
// line 5: user password salt [hex]
// line 6: master key checksum salt [hex]
// line 7: number of PBKDF2 rounds to use (same for user & master) [decimal]
// line 8: IV of the user key [hex]
// line 9: master key blob [hex]
//     IV of the master key, master key itself, master key checksum hash
//
// The master key checksum is the master key plus its checksum salt, run through
// 10k rounds of PBKDF2.  This is used to verify that the user has supplied the
// correct password for decrypting the archive:  the master key decrypted from
// the archive using the user-supplied password is also run through PBKDF2 in
// this way, and if the result does not match the checksum as stored in the
// archive, then we know that the user-supplied password does not match the
// archive's.
```

Experiment: Examining Android Backups

The Android backup file header is easy to figure out using Listing 3-3, but its contents are compressed by default. Using the semi-documented `-nocompress`, which is supported by the `bu` upcall script but not readily advertised by `adb`, you can create an uncompressed backup:

Output 3-12: Creating and inspecting an uncompressed backup

```
morpheus@Forge (~) % adb backup -nocompress -all
# UI is displayed on device...
Now unlock your device and confirm the backup operation.
morpheus@Forge (~) % ls -l backup.ab
-rw-r-----  1 morpheus  staff  17158168 Jan  1 23:37 backup.ab
morpheus@Forge (~) % head -6 backup.ab
ANDROID BACKUP    # MAGIC
3                 # Version (3 = L)
0                 # Compression (0 = False)
none              # Encryption (none)
apps/android/_manifest000600 0175001750000000036240010767 0ustar001
android
..
```

The header is straightforward enough, but what of the actual backup contents? The first line looks suspiciously like meta data. We therefore strip the header, and try our luck with `file(1)`:

Output 3-13: Stripping the header from an Android archive

```
# The header was four lines long, so start as of line 5...
morpheus@Forge (~) % tail +5 backup.ab > a.ab
# Attempt to auto identify the file..
morpheus@Forge (~) % file a.ab
a.ab: POSIX tar archive
# Check file contents:
morpheus@Forge (~) % tar tvf a.ab | more
-rw-------  0 1000   1000     1940 Dec 31  1969 apps/android/_manifest
-rw-------  0 1000   1000       99 Jan  1 21:29 apps/android/r/wallpaper_info.xml
-rw-------  0 1000   1000     1961 Dec 31  1969 apps/com.android.browser.provider/_manifest
...
```

And thus we see that Android backups, internally, are nothing more than good ol' UN*X `tar` archives. Using compression applies the Deflate algorithm after the `tar`.

If you do use encryption, the header size and complexity both increase. The following shows the header of the same archive, when compressed and encrypted with "password":

Output 3-14: Examining an encrypted backup

```
# Note this time, no nocompress implies compress by default
morpheus@Forge (~) % adb backup -all
# UI is displayed on device... enter "password"
Now unlock your device and confirm the backup operation.
# File is significantly smaller this time
morpheus@Forge (~) % ls -l backup.ab
-rw-r-----  1 morpheus  staff  7518645 Jan  1 23:50 backup.ab
morpheus@Forge (~) % head -9 backup.ab
ANDROID BACKUP
3
1          # This time, compressed
AES-256    # Encryption algorithm
FBAEB6CF..# 128 hex digits = 512-bit salt
98A4BF42..# 128 hex digits = 512-bit master key checksum
10000      # Number of PBKDF2 key derivations
0B0D638F9856C5D4F040399AB28A0C5F # Random IV (32 hex digits = 128bit)
E8AD4E9948F356E15A1E41AA265660.. # 192 hex digits = 768 bit Master key blob
```

Monitoring backup operations

The BackupManagerService stores its configuration in two main locations:

- **The system secure settings:** common to all Android framework services, and accessible via the Settings class. The manager defines the following settings (with constants in the Settings class identical to the string values, uppercased:

Table 3-10: Settings controlling backup behavior

Setting	Purpose
backup_enabled	Is backup enabled? Equivalent to bmgr enable
backup_transport	Default transport. Settable by bmgr transport ..
backup_provisioned	Is backup provisioned? Useful for managed devices
backup_auto_restore	Can application data be automatically restored?

- **The /data/backup directory:** containing the list of transports (as directories), and backup queues.

Normally, you won't need to go into the directory or settings yourself, as you can use bmgr (or settings) to toggle the settings, and dumpsys backup to get verbose information on the queues. The annotated output is shown below:

Output 3-15: Using dumpsys to display the backup status

```
shell@flounder:/ $ dumpsys backup
Backup Manager is disabled / provisioned / not pending init
Auto-restore is enabled
Last backup pass started: 0 (now = 1420171109885)
  next scheduled: 0
# List of transports. Google cloud is default, but requires account
Available transports:
  * com.google.android.backup/.BackupTransportServ
       destination: Need to set the backup account
       intent: Intent { act=com.google.android.backup.SetBackupAccountActivity }
    android/com.android.internal.backup.LocalTransport
       destination: Backing up to debug-only private cache
       intent: null
Pending init: 0
# List of applications that can request backup, sorted by AID:
Participants:
  uid: 1000
    com.android.providers.settings
    android
  uid: 1027
    com.android.nfc
...
# Ancestral refers to full backups, which serve as a point of
# departure for incremental backup/restore operations
Ancestral packages: none
Ever backed up: 0
Pending key/value backup: 13
    BackupRequest{pkg=com.google.android.gm}
    BackupRequest{pkg=com.google.android.talk}
    ..
Full backup queue:47
# Last backup : package name
    0 : com.android.providers.downloads.ui
    0 : com.android.externalstorage
    0 : com.google.android.nfcprovision
    ..
```

Experiment: Delving deeper into backups

To get a better grip of backups on Android, have a look at the /data/backup directory, which is where the BackupManagerService maintains its metadata. As root, you should see something similar to the following:

Output 3-16: The /data/backup directories

```
root@flounder:/data/backup # ls -l
drwx------ system   system       ...  com.android.internal.backup.LocalTransport
drwx------ system   system       ...  com.google.android.bac
-rw------- system   system  1881 ...  fb-schedule
drwx------ system   system       ...  pending
```

Getting the default transport is a simple matter, either by calling on the bmgr upcall script, or querying the value directly from the system's secure settings:

Output 3-17: Finding the default transport

```
root@flounder:/data/backup # bmgr list transports
  * com.google.android.backup/.BackupTransportService
    android/com.android.internal.backup.LocalTransport
root@flounder:/data/backup # settings get secure backup_transport
com.google.android.backup/.BackupTransportService
```

The backup queue is maintained in memory, but also written to the pending directory, as a journal-*xxxx*.tmp temporary file, to provide recovery in case the backup service itself crashes. The file format is simply a concatenation of package names to be backed up. Since the package names are preceded by a length byte and NULL terminated, use cat -tv to display this file:

Output 3-18: Displaying the backup journal

```
root@flounder:/data/backup # cat  -tv pending/journal-168056423.tmp
^@$com.android.providers.userdictionary^@'com.google.android.googlequicksearchbox^@"
com.google.android.marvin.talkback^@$com.google.android.inputmethod.latin^@^Ucom.
google.android.gm^@^Ocom.android.nfc^@^Scom.android.vending^@^Gandroid^@^Wcom.google.
android.talk^@^_com.android.sharedstoragebackup^@)com.google.android.apps.genie.
geniewidget^@^[com.google.android.calendar^@^^com.android.providers.settingsroot@
```

Lastly, the fb-schedule file schedule is used to maintain a list of all installed packages which are backup eligible (i.e. declared a BackupAgent in their manifest, as we discuss in Volume II, and is well documented in the Android Developer Website[11]). The file format is very similar to that of the journal (albeit with a few more fields), but this is where dumpsys gets handy (which is even more useful since you don't need root privileges to use it)

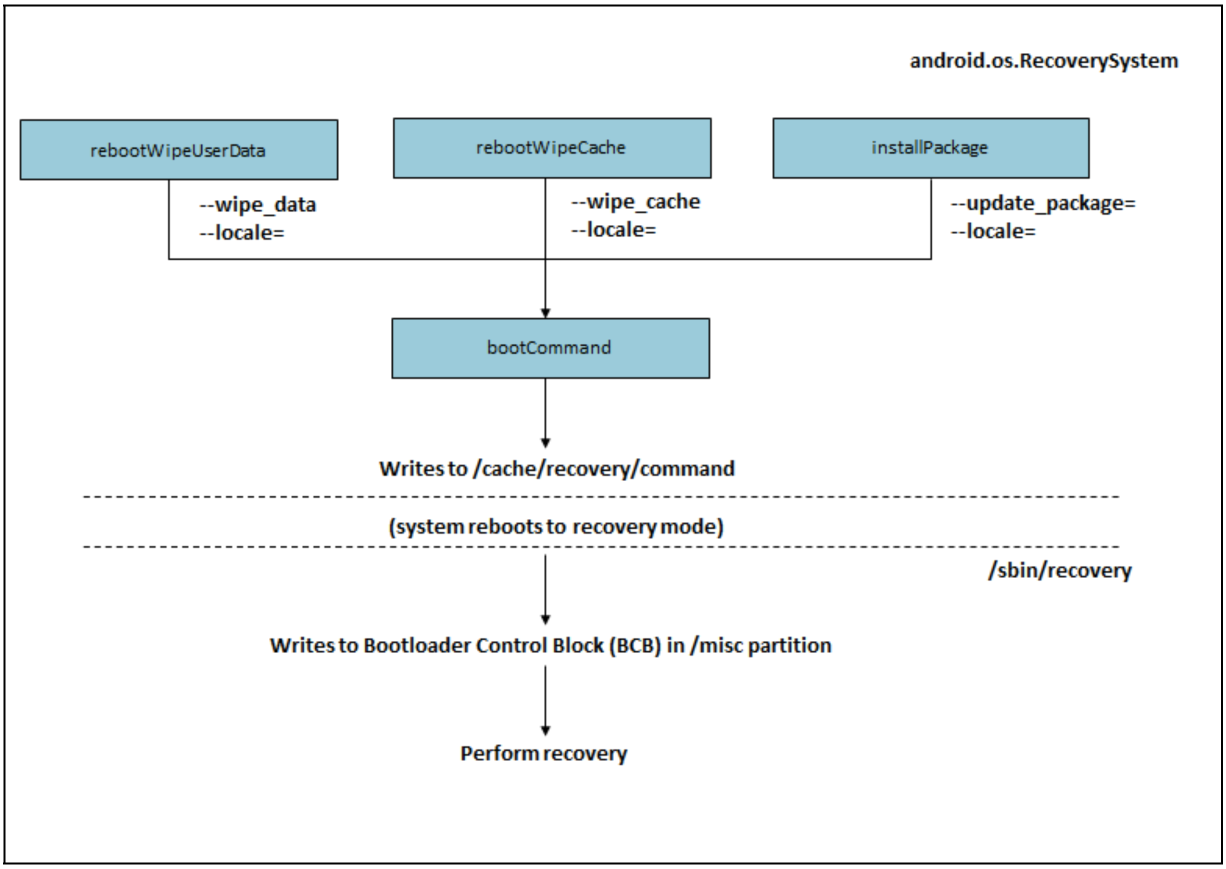
System Recovery & Updates

System Recovery and updates are similar processes: In both, the system needs to be diverted to an alternate boot sequence, which - rather than load the full OS UI - loads a minimal configuration, wherein a special binary - /sbin/recovery - can be used to handle the process in question.

Either process is normally started when the system is fully booted, and in UI mode, though the device can also be ordered into recovery through `adb reboot recovery` or via fastboot. When started from the UI, the `android.os.RecoverySystem` class provides the framework support needed for downloading and verifying an update, if one is required. The update must be digitally signed, and is validated against certificates taken from /system/etc/security/otacerts.zip keystore. If validation passes, the update is copied to the /cache partition. This is why on devices like the Amazon Kindle, with forced automatic updates which can break root, removing the otacerts.zip file will prevent updates. An equally effective measure can be to `chown root` and `chmod 755` the /cache partition.

The class also provides arguments to the recovery process, which it writes to the /cache/recovery/command file. The class then reboots the system, but passes an argument to the bootloader, to boot from the recovery partition, rather than the boot partition. Recall from the earlier discussion that the recovery and boot partitions are usually identical, save for the ramdisk image - which in the case of recovery, will load the /sbin/recovery, instead of the full Android framework. The flow of commands from `RecoverySystem` to /sbin/recovery is shown in the following figure:

Figure 3-7: Interaction of `android.os.RecoverySystem` with /sbin/recovery



The /sbin/recovery binary gets its arguments from its command line, if any. If those aren't supplied, the misc partition is searched for the "Bootloader Control Block" (BCB). If the partition cannot be found or its contents cannot be parsed, the binary turns to /cache/recovery/command. The Android runtime does not interface with the BCB directly, but /sbin/recovery saves any arguments supplied to it into the partition, to enable recovery to resume if somehow interrupted.

The /sbin/recovery binary is guaranteed to load - because it's part of the ramdisk, and not in any way dependent on /system. This is an important observation, because the system may be in an entirely unbootable state. At this point, then, the kernel has initialized, /init has loaded, but recovery (and possibly adbd) is the only process executing. The /sbin/recovery will then read the command left for it in the /cache/recovery/command file (as shown in the previous figure), and act according to its content, shown in Table 3-10:

Table 3-10: Arguments understood by /sbin/recovery

Argument	Purpose
--wipe_cache	Wipe the /cache partition and reboot.
--wipe_data	Wipe all the user data in the /data partition, i.e. a "Factory Reset". Throughout the device lifetime, /system is normally mounted read-only, and should therefore face little risk of corruption. Restoring the device to factory defaults therefore amounts to formatting /data, which both serves to efface personal user data, as well as clear any corrupted files which may be hindering the boot process. This option also implies --wipe_cache.
--update_package	Specifies the path to the an OTA update package, which needs to be applied as a patch over the system. OTA packages are discussed next.
--locale	Specify locale used. This goes into /cache/recovery/last_locale.
--send_intent	Name of intent to place in /cache/recovery/intent.
--show_text	Show textual messages
--just_exit	Exit without performing any actions. *unused*

During the process, it's important to keep the user informed and visually engaged. recovery therefore makes use of minui, a library which (as its name implies) provides basic GUI functionality. This library is discussed in more depth in Volume II.

Over-The-Air (OTA) Updates

Occasionally, the vendor or carrier (and sometimes Google itself) may provide an update to the Android OS in the form of an Over-The-Air (OTA) update. To be delivered over the air, updates must be kept as small as possible. It is for this reason that OTA updates are usually differential patches, based on a particular build of Android, which is assumed to be the one being updated.

Android's OTA updates are packaged as a single zip file (technically, more like a JAR, as it contains a META-INF/ subdirectory), digitally signed, which consists of:

- **Multiple patch files**: in the bsdiff(1) format, which is essentially a series of file offsets and lengths, along with the data to insert or delete from the offsets. The standard patch files have the names of the files they are patching, with a ".p" extension appended to them.
- **A patch binary**: (usually called update-binary) which can parse the patch files and apply them, according to directions given by..
- **A patch script**: (usually called updater-script) which executes the binary multiple times (one per patch), and specifies the expected hash of the file to be patched - pre/post patch operation.

- **Any additional files**: which are either newly added to the system, or are so heavily modified that a patch file would actually be larger than the full file.
- **A metadata file:**: consisting of post-build, post-timestamp, pre-build and pre-device entries, which provide the device properties before and after the update
- **An otacert file:**: A PEM-formatted certificate. Can be compared to and optionally imported into /system/etc/security/otacerts.zip.

Note, that vendors are free to add and/or modify any of the OTA components. A good example can be seen in the updates of Amazon's Kindle, which contains not only updates to the files in /system, but also additional firmware images, including the non mountable partitions.

Listing 3-4 shows the content of an OTA update, in this case the Google supplied KitKat update for the Nexus 5:

Listing 3-4: The contents of an OTA update for the Nexus 5

```
morpheus@Forge (/tmp)$ unzip -l 537.....740.signed-hammerhead-KOT49H-from-KOT49E.537367d5.zip
Archive:  537367d588afe31301268d0ace7e60725d5f6740.signed-hammerhead-KOT49H-from-KOT49E.537367d5.zip
signed by SignApk
  Length     Date   Time    Name
 --------    ----   ----    ----
      203  09-03-13 11:53   META-INF/com/android/metadata
   275280  09-03-13 11:53   META-INF/com/google/android/update-binary
   132207  09-03-13 11:53   META-INF/com/google/android/updater-script
     2045  09-03-13 11:53   patch/system/app/BasicDreams.apk.p
            ...
      575  09-03-13 11:53   recovery/etc/install-recovery.sh
   100345  09-03-13 11:53   recovery/recovery-from-boot.p
   487562  09-03-13 11:53   system/framework/telephony-common.jar
    23252  09-03-13 11:53   META-INF/MANIFEST.MF
    23305  09-03-13 11:53   META-INF/CERT.SF
     1346  09-03-13 11:53   META-INF/CERT.RSA
     1229  09-03-13 11:53   META-INF/com/android/otacert
```

The OTA update process

When started with --update_package, recovery binary calls install_package(), which loads the package specified as argument, and looks for the update-binary inside it. The binary name is hard-coded, #defined as the ASSUMED_UPDATE_BINARY_NAME of META-INF/com/google/android/update-binary. If found, it starts it, and the update-binary executes the updater-script, in a manner akin to shell scripts.

The standard update-binary's source can be found in the Android source tree (in the bootable/recovery directory). The update-binary used by most OTA packages is derived from this source, as vendors are encouraged to use the source as a point of departure. Rather than modifying it, vendors can easily add additional libraries as "device extensions". This is done by adding any such libraries to the TARGET_RECOVERY_UPDATER_LIBS variable in the Android.mk file, and providing a Register_*libname* function in each.

Looking at the source of the updater binary, you can find a list of all functions in the implementation of RegisterInstallFunctions():

Listing 3-5: The `updater-binary`'s `RegisterInstallFunctions()`

```
void RegisterInstallFunctions() {
    RegisterFunction("mount", MountFn);
    RegisterFunction("is_mounted", IsMountedFn);
    RegisterFunction("unmount", UnmountFn);
    RegisterFunction("format", FormatFn);
    RegisterFunction("show_progress", ShowProgressFn);
    RegisterFunction("set_progress", SetProgressFn);
    RegisterFunction("delete", DeleteFn);
    RegisterFunction("delete_recursive", DeleteFn);
    RegisterFunction("package_extract_dir", PackageExtractDirFn);
    RegisterFunction("package_extract_file", PackageExtractFileFn);
    RegisterFunction("symlink", SymlinkFn);
    // Maybe, at some future point, we can delete these functions? They have been
    // replaced by perm_set and perm_set_recursive.
    RegisterFunction("set_perm", SetPermFn);
    RegisterFunction("set_perm_recursive", SetPermFn);
    ...
    RegisterFunction("set_metadata", SetMetadataFn);
    RegisterFunction("set_metadata_recursive", SetMetadataFn);
    RegisterFunction("getprop", GetPropFn);
    RegisterFunction("file_getprop", FileGetPropFn);
    RegisterFunction("write_raw_image", WriteRawImageFn);
    RegisterFunction("apply_patch", ApplyPatchFn);
    RegisterFunction("apply_patch_check", ApplyPatchCheckFn);
    RegisterFunction("apply_patch_space", ApplyPatchSpaceFn);
    RegisterFunction("read_file", ReadFileFn);
    RegisterFunction("sha1_check", Sha1CheckFn);
    RegisterFunction("wipe_cache", WipeCacheFn);
    RegisterFunction("ui_print", UIPrintFn);
    RegisterFunction("run_program", RunProgramFn);
}
```

Listing 3-6 shows an annotated example of an updater-script, from the Samsung OTA update for 4.4.2. This demonstrates not just the commands from Listing 3-5, but also their usage with arguments:

Listing 3-6: An annotated example of updater-script

```
# Mount partitions, r/w, for purposes of overwriting and patching
mount("ext4", "EMMC", "/dev/block/platform/msm_sdcc.1/by-name/system", "/system");
mount("ext4", "EMMC", "/dev/block/platform/msm_sdcc.1/by-name/hidden", "/preload");
# Pre-verification: note the use of short circuit evaluation to either pass
# the check, or proceed to potentially abort update

file_getprop("/system/build.prop", "ro.build.fingerprint")
               == "samsung/jfltetmo/jfltetmo:4.3/JSS15J/M919UVUEMK2:user/release-key
                 ... ||
             abort("Package expects build fingerprint of  .......");
# getprop makes sure the device's built or product version match
assert(getprop("ro.product.device") == "jfltetmo" ||
       getprop("ro.build.product") == "jfltetmo" ||
       getprop("ro.product.device") == "jfltetmo" ||
       getprop("ro.build.product") == "jfltetmo");
# ui_print displays a text message for the user, and show_progress displays the bar
ui_print("Verifying current system...");
show_progress(0.100000, 0);
# apply patch check applies the corresponding .p file and
# validates the SHA-1 before and after
apply_patch_check("/system/framework/framework-res.apk",
                  "384b5aa8862c18a53fbfa8f8e3789d7edc4561ab",
                  "57d8eca2e983fc1031841d418e759246c9885245")
     || abort("\"/system/framework/framework-res.apk\" has unexpected contents.");
..
set_progress(0.078730);
# apply rename check is as apply_patch_check, for an mv.
# Assert verifies return value
assert(apply_rename_check("/system/app/SecGallery2013.apk",
                          "/system/priv-app/SecGallery2013.apk",
                          "8e0eb424e636a06733c3e08cdf29361d97d23929",
                          "66d8f058e1638124f618a90773aa4e8bad7765d8"));

# apply_patch_space returns false if space in bytes is less than quoted value
apply_patch_space(70620592) || abort("Not enough free space on /system to apply patc
assert(getprop("ro.secure")=="1");
..
# delete recursive removes files or directories
delete_recursive("/system/app/CABLService.apk",
                  ..
symlink("toolbox", "/system/bin/mkswap", "/system/bin/readlink",
        "/system/bin/swapoff",
        "/system/bin/swapon");
#...
# set_perm_recursive sets uid, gid, and permissions on files or directories
set_perm_recursive(0, 2000, 0755, 0755, "/system/xbin");
ui_print("Update tz.mbn ota");
assert(write_backup_fota("/dev/block/mmcblk0p8", "/dev/block/mmcblk0p22"));
assert(package_extract_file("tz.mbn", "/dev/block/mmcblk0p8"));
assert(erase_backup_fota("/dev/block/mmcblk0p22"));
..
show_progress(1.000000, 0);
unmount("/preload");
unmount("/system");
```

Customizing images

Armed with the information in this chapter, you should now have all the ingredients required to concoct your own firmware image. Note, that - as a prerequisite to this step - the bootloader on your device must be unlocked, because otherwise custom images will be rejected.

When customizing images, the usual caveats apply:

- Images downloaded from untrusted sources can contain malware - especially spyware.
- Using custom images may (or may not) void carrier's (or vendor's) warranty. That said, if the bootloader is not tampered with, the process is generally reversible. Some bootloaders don't keep track of tampering, so at times it's possible to just re-lock the bootloader, leaving no trace of any modification.
- Improperly installing a custom image may render the device unbootable ("brick"). The probability of this increases in direction relation to the parts of the image being modified:
  - If only /data is modified, the OS can still boot from /system. At worst, it is possible to undo changes by resetting to factory defaults (which effaces /data).
  - If both /data and /system are modified, the kernel and ramdisk can still boot normally (from the boot partition), and so the system can always enter rescue mode.
  - If the boot partition (kernel + initramfs) are modified, the kernel and ramdisk may fail to load, but the boot loader can still load, and hopefully offer fastboot (or another custom protocol) which still enable overwriting boot (or any other partition), thus recovering bootability.
  - If the bootloader itself is modified, you run the risk of corrupting it so that the device is unbootable, and possibly bricked for good

Given the above, it makes sense to proceed with appropriate caution. Most bootloaders allow you to boot alternate images (via fastboot) without overwriting any existing ones, which provides a safe environment for testing. We next describe methods for customizing an image

Adding files to an existing filesystem

If your device is already rooted, adding files to an existing filesystem is quite trivial - you need to first make sure the filesystem is writable, and if it isn't, remount it as such, and then simply proceed to copy the files onto it. Since `adb` normally doesn't run as root (though that, too can be modified), it's usually a two step operation of dropping the files (via adb push) at a writable directory (/data/local/tmp makes a good choice), and then `cp`ing them to their final destination (`mv` usually doesn't work with toolbox since it involves linking).

This method involves more work than others presented here, but is generally the safest - so long as you just add files and don't overwrite any existing ones, the chances of impacting the system are fairly small. A minor exception to the rule is when adding property files or other files whose presence changes the system behavior.

Modifying the initramfs

To modify files outside /data or /system filesystem, you will need to tweak the initramfs. Recall from our earlier discussion that the initramfs forms the root file system, and remains loaded in addition to /system and /data. The initramfs also contains the /init.rc file, which contains the startup commands for /init. This makes the initramfs a suitable vector for inserting commands for rooting the device, as well. This is demonstrated in the following experiment:

Experiment: Repacking an initramfs and writing it as a bootimg to the device

A previous experiment in this chapter demonstrated how to unpack the initramfs. Repacking the initramfs into a gzip image is as simple as executing the inverse operations, namely, run cpio with `-o` (output) rather than `-i` (input), and then run `gzip` to compress:

Output 3-19: Creating a compressed initramfs

```
morpheus@Forge (~/Android/Book/tmp) % find . | xargs cpio -ovd | gzip > output.gz
```

Note, however, the the initramfs is not standalone, but always bundled alongside the kernel. You therefore have to use the inverse of `imgtool` - the AOSP's `mkbootimg` to package the initramfs along with the kernel, so both can then be flashed to the boot partition. Assuming you have the kernel from the previous example handy (you wouldn't normally need to modify it), this would look something like:

Output 3-20: Creating a boot image from a kernel and a compressed initramfs

```
morpheus@Forge (~/Android/Book/tmp)% mkbootimg --kernel kernel \
                                     --ramdisk output.gz --output bootimg.img
```

And then write it to the device `fastboot flash` (if your bootloader is unlocked), or by using a simple (but careful!) `dd(1)` (if your device is otherwise rooted). The following uses the `by-name` symlinks to figure out where the boot partition is. Note results may vary (*and be **extremely careful** not to confuse boot with aboot or hboot!*).

Output 3-21: Booting from an image

```
morpheus@Forge (~/Android/Book/tmp)% adb push bootimg.img /data/local/tmp
morpheus@Forge (~/Android/Book/tmp)% adb shell
shell@Android /$ grep boot /proc/partitions
..
..
shell@Android /$ su
root@Android  /# dd if=/data/local/tmp/bootimg.img \
                 of=/dev/block/platform/msm-sdcc.1/by-name/boot
```

As a measure for added safety, you can repeat the previous experiment to unpack the boot image and verify it before writing it to the device. It's always a good idea to save the unmodified boot image you started with, since it can easily be written back to the device (via fastboot) should something go awry.

Overwriting an entire partition

Sometimes it's easier to just overwrite the entire partition, rather than just add files to it. This is usually the case when the customization involves many files (such as reducing vendor bloatware), or when it involves a partition that is not necessarily mountable. In either case, the first step involves copying the partition directly from the raw block device onto an image, using the `dd(1)` utility, or a direct `adb pull` of the raw block device to the host. This method is essentially the same as the process shown in Chapter 2. You can also use the Android stock images as a point of departure, as shown in the earlier experiment.

Experiment: Packing an Android System partition image

Recreating the partition image is generally an inverse of the process, and will require you to use `mkextfs` (on the host) to create a new filesystem, before adding the files you want to it. Unmounting the filesystem will commit changes to the image, and you can then copy it to the device, switching the `dd(1)` arguments (very carefully!). Alternatively, you can use fastboot to flash the image to its target partition. The `dd(1)` method may be preferable as a workaround for cases where the bootloader refuses to flash unsigned partition images.

Compiling `img2simg` instead of `simg2img` in the earlier experiment will provide you with a binary that can take a raw filesystem image and make it into a sparse one. This is useful if you want to customize your Android device's image in a way that will enable you to flash it. You can take the Android partition images as a base, add more files (a SetUID "su" comes to mind), and then create a raw filesystem image from the mounted filesystem. From there, using `img2simg` is trivial, and you can then flash the resulting file to your device using `fastboot` (discussed later in this chapter).

Resources for image modding

The discussion of customizing images in this book should provide a solid foundation for you if you choose to foray into this exciting, yet somewhat dangerous realm. Fortunately, there are quite a few resources which can be of great value, providing mods developed and tested by other users and sparing you the unfortunate experience of a bricked device.

XDA-Developers

Hands down the most important resource for Android power users, the XDA Developers[12a] website is a vast repository for all things Android. In particular, its extensive forums[12b] which contain threads and vivid discussions for just about any device on the market. Rooting and recovery tools often get advertised first via the forums, and with a community of nearly 6 million users (amongst them well known names in the modding world) it's always possible to get help in all but the most newbie of questions.

Cyanogen, AOKP, etc

With Android's open source nature, it was only a matter of time before groups outside of Google picked up the gauntlet of "modding" or customizing the operating system. Indeed, several (open source) projects were up to the task, such as the Android Open Kang Project and CyanogenMod. The latter has become a one-stop-shop for virtually every device on the market, with customized versions of firmware rebuilt for each device from the vendor-provided kernel and system image. Cyanogen developers also tweak the operating system in many ways, not the least of which is rooting (which is an easy by-product of a custom image). More often than not, images incorporating fixes and patches from newer versions of Android appear on Cyanogenmod well before the official vendors release an update.

What makes Cyanogen possible is the fact that vendors must supply the source for the kernel they use in Android. That, in addition to the AOSP remaining open, enables the collaborative effort by many developers to vastly improve vendor releases, and liberate others from being subject to the vendor's (often sluggish) update cycles. Thus, device specific tweaks to Android either get published as part of the vendor's kernel sources, or - in binary form - as modules. In the latter case, however, recompiling the kernel maintains compatibility with vendor modules. Vendor specific APKs can be moved (or removed) from the base image easily, and others added.

In most cases, rather than build-your-own image,Cyanogen can provide you with ready made images, which mitigate the potential for an error during image creation - which can brick your device. CyanogenMod have gone as far as to implement their own boot loader, and the entire process of customizing and installing an image has been reduced to only a few clicks. The avid reader is strongly encouraged to check out the CyanogenMod website[13a] for more. In particular, Cyanogen maintains an extensive Wiki, and their Development Learning Center[13b] is a particularly good reference.

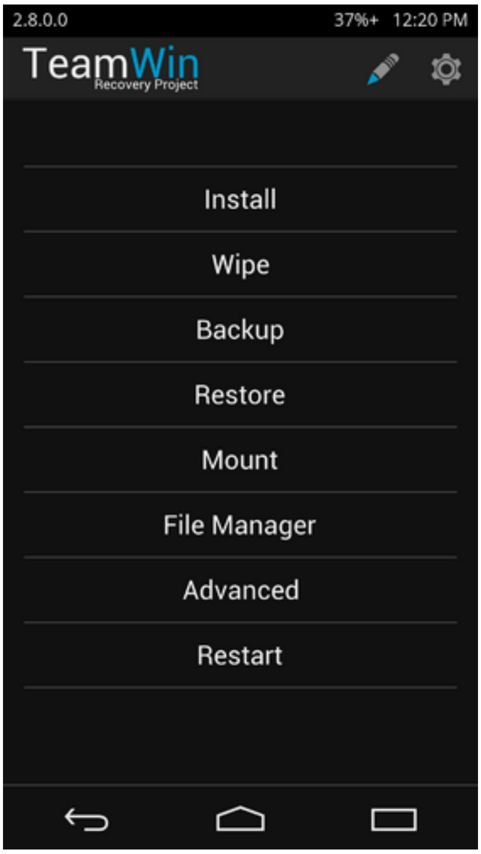
Cyanogen is becoming somewhat of an alternative to stock Android, as some vendors (notably One Plus) use it as the default version of Android, rather than Google's own. Now a full fledged startup, Cyanogen commanded over $100 in investments from Microsoft, FoxConn and others.

Team-Win Recovery Project

The Team-Win Recovery Project[14] is a custom recovery image created by a group of "Android Enthusiasts in it for the fun of the Win". Like all recovery images, it consists of a bootimg formatted partition, containing a kernel and RAM disk. The kernel is usually the same one found in the stock image, but the RAMdisk contains a full featured recovery binary, which includes a featureful GUI with backup/restore functionality, support for Ext file systems, ExFAT, and F2Fs, and ChainFire's SuperSU. The inclusion of the latter makes it a popular choice as a rooting tool for devices with unlocked bootloaders. For advanced users, TWRP's image provides a quick way to get the busybox binary, compiled against Bionic, and thus able to show Androidisms, such as AIDs in `ls -l` output.

TWRP's GUI layout is provided in the res/ui.xml (which can be customized with resources found at the Team's GitHub repository[15]). The res/ directory also contains fonts and PNG images used by the layout. The binary is a larger than usual /sbin/recovery, which uses a modified version of the AOSP's MinUI (`libminui.so`) library. The modifications enables the simple GUI and touch support (as opposed to standard recovery binaries, which require the physical keys of the device - volume and power buttons). None of the Android runtime features or services are needed (though a barebones /init does load, as is required by the boot process), which keeps the image compact. MinUI is covered in more detail in Volume II.

Figure 3-8: The TWRP 2.8 (latest at the time of writing) UI



Summary

This chapter explored aspects of the Android boot sequence and lifecycle: From the formats of the various images involved, through their step-by-step operation. Shutting down and restarting was explained, in particular "booting to recovery". Recovery mode concepts and the role of /sbin/recovery followed. Finally, the process of customizing or "modding" images was explained in detail.

One glaring, yet intentional omission was that of user mode boot - i.e. what happens following kernel initialization, when PID 1 (/init) starts up, and launches the various Android native services (and, eventually, the framework services). The roles of /init and the various native services are explored in great detail in Chapter 5, and the framework services - in Volume II.

References

1. Companion Website, ImgTool source + binary: http://NewAndroidBook.com/files/imgtool.tar
2. Google Nexus Factory Image Repository: https://developers.google.com/android/nexus/images
3. a. CodeAurora, LK: https://www.codeaurora.org/cgit/quic/la/kernel/lk/tree/app/aboot
   b. Google Source, LK: https://android.googlesource.com/kernel/lk/+/a9b07bbae16a0b1b6de07ec3a3e2005c99043757/
4. a. Accuvant, Building a Nexus 4 UART Debug cable: http://www.accuvant.com/blog/building-a-nexus-4-uart-debug-cable
   b. OSDevNotes Blog, 64-Bit ARM Kernel Development demo: http://osdevnotes.blogspot.com/2014/11/64-bit-arm-oskernelsystems-development.html
5. Companion article, Disassembling ABoot http://NewAndroidBook.com/Articles/aboot.html
6. Android Documentation, Building Kernels: http://source.android.com/source/building-kernels.html
7. ePAPR DTB specification: https://www.power.org/wp-content/uploads/2012/06/Power_ePAPR_APPROVED_v1.1.pdf
8. Thomas Pettazoni, "Device Tree for Dummies": http://elinux.org/images/a/a3/Elce2013-petazzoni-devicetree-for-dummies.pdf
9. a. XDA-Developers, Discussion of Qualcomm leaked documents: http://forum.xda-developers.com/showthread.php?t=1856327&page=1
   b. Thread 24100141
10. Android Developer, bmgr utility: http://developer.android.com/tools/help/bmgr.html
11. Android Developer, Declaring Backup Agent in Manifest: http://developer.android.com/guide/topics/data/backup.html#BackupManifest
12. a. XDA Developers: http://www.xda-developers.com
    b. XDA Developers - Forums: http://forum.xda-developers.com
13. a. CyanogenMod: http://www.cyanogenmod.org/
    b. CyanogenMod Wiki: http://wiki.cyanogenmod.org/w/Development
14. Team-Win Recovery Project: http://teamw.in/project/twrp2
15. Team-Win GitHub Repository: https://github.com/TeamWin/Team-Win-Recovery-Project/tree/jb-wip/gui/devices">

IV: init

All UN*X systems have a special process, a process which is the first to spring into existence in user-mode when the kernel has finished booting, and is charged with starting up the system and serving as the progenitor of all other processes. Traditionally, this process is called init, and Android follows that convention as well.

The Android init, however, is vastly different than that of UN*X or Linux, with the most important differences being in its support of System Properties and using a particular set of rc files. Following the explanation of those two features, we piece together the flow of init: its Initialization and Run-Loop.

As it so happens, init also fills additional roles - assuming the guise of ueventd and watchdogd, two important core services which are also implemented by init, loaded through a symbolic link.

The roles and responsbilities of init

Like most UN*X kernels, the Linux kernel looks for a hard-coded binary to launch as the first user mode process. On desktop Linux systems, this has traditionally been /sbin/init, which read the /etc/inittab file for a description of supported "run-levels", or runtime configurations (single user, multi-user, network file systems, etc), start-up processes, and ctrl-alt-del behavior. Android also uses an "init" binary, but most similarities end with the name. The following table shows the differences:

Table 4-1: Android's /init versus the traditional UN*X /sbin/init

	Linux /sbin/init	**Android /init**
Config file	/etc/inittab	/init.rc and any `import`ed file (commonly init.*hardware*.rc and init.usb.rc (sometimes init.*hardware*.usb.rc
Multiple configurations	Supported through the notion of "run-levels" (0: shutdown, 1, single user, 2-3 multi-user, etc). Each "run level" loads scripts from /etc/rc*runlevel*.d	No run-levels, but offers configuration options through triggers and system properties
Watchdog functionality	**Yes:** Daemons defined with the `respawn` keyword are restarted on exit, unless they repeatedly crash, in which case they are suspended for a few minutes.	**Yes:** Services are kept alive by default, unless defined as `oneshot`. Services may also further be defined as `critical`, which will force the system to reboot if they cannot be restarted.
Adopting orphan processes	**Yes:** `/sbin/init` will call `wait4()` to reap the return code, and avoid zombies.	**Yes:** `/init` registers a handler for `SIGCHLD` which the kernel will automatically send on child process exit. Most processes are silently `wait()`ed for and their exit code discarded.
System Properties	**No:** Linux `/sbin/init` does not support the notion of system properties	`/init` provides read access to properties (`getprop`) to all processes on the system via shared memory, and a `property_service` which allows write access (`setprop`).
Socket assignment	**No:** Linux's init cannot get sockets for its child processes. This functionality is available for inetd.	**Yes:** `/init` can bind a UNIX domain (or, as of L, seqpacket) socket for a child, which can then get it through `android_get_control_socket`
Triggered operation	**No:** Linux allows only very specific triggers, such as ctrl-alt-del and UPS power events, but does not allow arbitrary triggers	**Yes:** `/init` can execute commands on any system property change, allowing it to run pre-defined commands on triggers that can be set by any (or some) users
Handling uevents	**No:** Linux relies on the `hotplug` daemon (usually `udevd`)	**Sort of:** `/init` also spawns itself as `ueventd`, with separate config files

As a binary, /init is statically linked. This means that all of its dependencies are merged into the binary during compilation, so as to mitigate the risk that a corrupt or missing library abort system startup. When it is first launched, the only filesystem mounted is the root filesystem (i.e / and /sbin), which is packaged in the Android boot partition along with the kernel.

In a sense, the Android take on init is closer to another's - iOS's launchd. Triggers and sockets in particular are features offered by the latter, though Android shows novelty with the introduction of system properties.

System Properties

The Android System Properties provide a globally accessible repository of configuration settings. They are somehwat similar in form and function to `sysctl(2)` MIBs, but are implemented in user mode by `init`. The `property_service` code in `init` loads properties from several files, in the order shown in table 4-2:

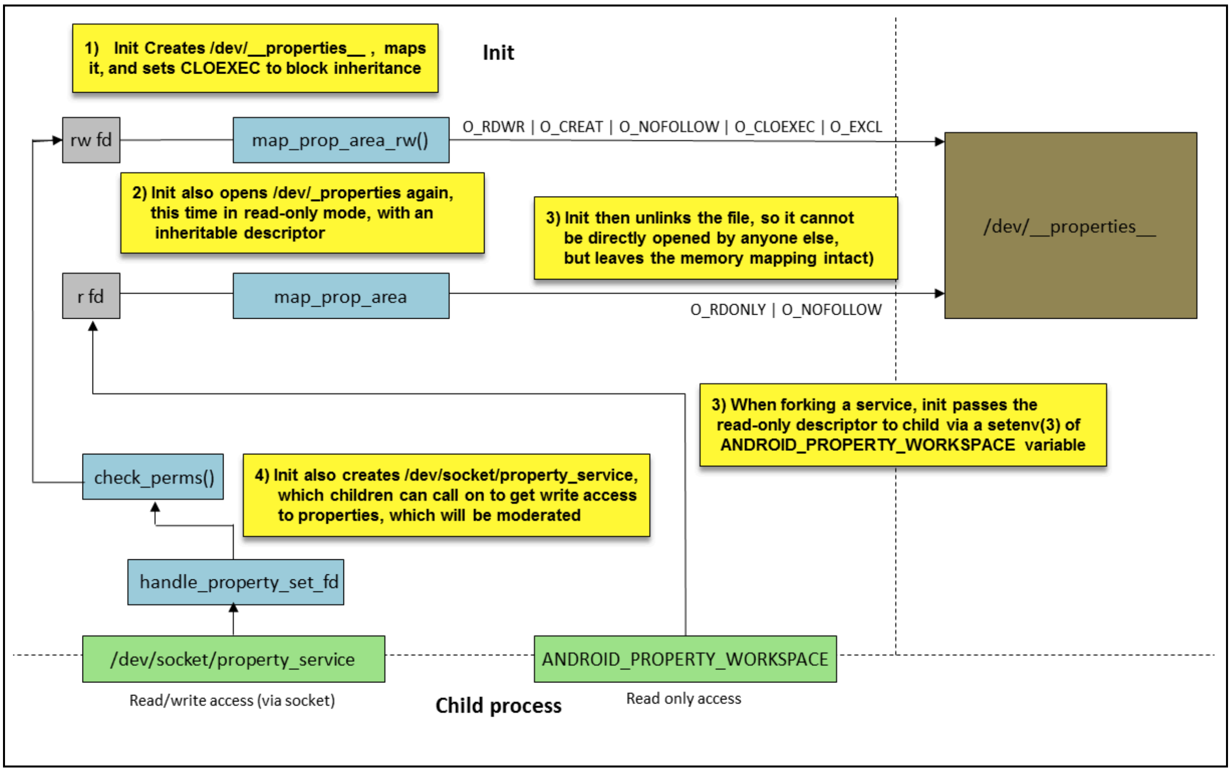
Table 4-2: Property Files in the Android file system

File	Contains
/default.prop (`PROP_PATH_RAMDISK_DEFAULT`)	Initial settings. Note this file is part of the initramfs, and not present on the device's flash partitions.
/system/build.prop (`PROP_PATH_SYSTEM_BUILD`)	Settings generated by the Android build process
/system/default.prop (`PROP_PATH_SYSTEM_DEFAULT`)	Settings usually added by vendor
/data/local.prop (`PROP_PATH_LOCAL_OVERRIDE`)	Loaded if `init` was compiled with `ALLOW_LOCAL_PROP_OVERRIDE`, and the `ro.debuggable` property is set to 1. This enables developers to override previous settings by dropping a file into /data.
/data/property/persist.* (`PERSISTENT_PROPERTY_DIR`)	Persistent properties. Prefixed by `persist`, these are saved across reboot individually in files in this directory. `Init` can also re-load them at any time using the `load_persist_props` directive in the /init.rc.

An additional property file, `PROP_PATH_FACTORY` (/factory/factory.prop) is #defined but no longer supported. Note the order of loading does matter, since setting the same property a second time will overwrite the previous value (unless the property is marked read-only).

Because `init` is the ancestor of all processes in the system, it is only natural that it implement the property store. Early in its initialization, the `init` code calls `property_init()` to set up system properties. This function (eventually) calls `map_prop_area()`, which opens the `PROP_FILENAME` (`#define`d as /dev/__properties__), and `mmap(2)`s into memory with read/write permissions, before closing it. Additionally, init *re-opens* the file, this time for `O_READONLY`, and then unlinks it.

Figure 4-1: Handling the property workspace mapping



1) Init Creates /dev/__properties__ , maps it, and sets CLOEXEC to block inheritance

2) Init also opens /dev/_properties again, this time in read-only mode, with an inheritable descriptor

3) Init then unlinks the file, so it cannot be directly opened by anyone else, but leaves the memory mapping intact)

3) When forking a service, init passes the read-only descriptor to child via a setenv(3) of ANDROID_PROPERTY_WORKSPACE variable

4) Init also creates /dev/socket/property_service, which children can call on to get write access to properties, which will be moderated

The read-only file descriptor of the property file is set to be inheritable by children. This allows any process in the system easy access to system properties, albeit read-only, by `mmap(2)`ing the descriptor early on. This clever approach effectively allows all users of the properties area to share the same physical memory backing the property area (`#define`d as `PA_SIZE`, 128k by default). The only write access to this area, however, remains in the hands (and memory) of `init`.

You can see the shared memory area in all user mode processes on the system easily by looking at the maps /proc entry:

Output 4-1: Viewing the mapping of the system property area, through the /proc filesystem

```
# In init: (note area is writable)
root@generic:/ # grep __properties /proc/1/maps
b6f2f000-b6f4f000 rw-s 00000000 00:0b 1369       /dev/__properties__
# In any user mode process (in this case, the shell)
root@generic:/ # grep __properties /proc/$$/maps
b6e5a000-b6e7a000 r--s 00000000 00:0b 1369       /dev/__properties__
```

Most developers remain agnostic to the internal structure of the shared property area. The area is prefixed by a short header, which contains a serial number (reflecting internal versioning), a magic value (`0x504f5250` or 'PROP'), and a version (0xfc6ed0ab for newer versions of Android, or `0x45434f76` for compatibility). Then, following another 112 bytes (padding the header to 128 bytes), are the properties themselves. Properties are stores in a data structure which hybridizes a trie (prefix tree) and a binary tree. This is rather nicely documented (if you appreciate ASCII art) in Bionic's system_properties.c:

Listing 4-1: Internal structure of system properties, from system_properties.c:

```
/*
 * Properties are stored in a hybrid trie/binary tree structure.
 * Each property's name is delimited at '.' characters, and the tokens are put
 * into a trie structure.  Siblings at each level of the trie are stored in a
 * binary tree.  For instance, "ro.secure"="1" could be stored as follows:
 *
 * +-----+   children    +----+   children    +--------+
 * |     |-------------->| ro |-------------->| secure |
 * +-----+               +----+               +--------+
 *                       /    \                /   |
 *                 left /      \ right   left /    |  prop   +===========+
 *                     v        v            v     +-------->| ro.secure |
 *                  +-----+   +-----+     +-----+            +-----------+
 *                  | net |   | sys |     | com |            |     1     |
 *                  +-----+   +-----+     +-----+            +===========+
 */
```

The `property_service`

In order to service write requests, `init` opens up a dedicated UNIX domain socket - /dev/socket/property_service, which is world-writable (0666), so that any client may connect. It is then up to `init` to enforce permissions on the properties, which are hard-coded in an ever increasing list called `property_perms`. The permissions are based on simple UID and GID checks, (which `init` obtains from the socket caller credentials), as shown in table 4-3. UID 0 is allowed full access to the properties. When SELinux is enabled (as of KitKat and L) property namespaces are further protected by security contexts, as defined in /property_contexts, and shown in the following listing. (SELinux on Android is explained in Chapter 8).

Table 4-3: Property namespaces and their permissions

Namespace	Owning UID	Contains

`net.rmnet0`
`net.gprs`
`net.ppp`
`net.qmi`
`net.ltr`
`net.cdma`
	`AID_RADIO`	Network properties, used by rild

`gsm`		GSM related settings
`persist.radio`		Persistent radio settings
`net.dns`		DNS resolver settings (in loco /etc/resolv.conf)
`sys.usb.config`		USB mode (adb, mtp, mass storage, rndis, etc)
`net`	AID_SYSTEM	All network settings (including those owned by AID_RADIO)
`dev`		All device settings
`runtime`		Unused
`hw`		hardware related settings
`[persist.]sys`		system related settings
`[persist.]service`		service start/stop keys
`persist.security`		security related settings
`wlan`		Wireless LAN (WiFi) settings
`selinux`		Security Enhanced Linux settings
`dhcp`	AID_SYSTEM AID_DHCP	DHCP settings
`debug`	AID_SYSTEM AID_SHELL	Debug settings
`log`	AID_SHELL	Logging settings
`service.adb.root`	AID_SHELL	Used by ADB if running as root
`service.adb.tcp.port`	AID_SHELL	Used by ADB if running over TCP/IP
`sys.powerctl`	AID_SHELL	Power Management Control
`bluetooth`	AID_BLUETOOTH	Bluetooth settings
`persist.service.bdroid`		Bluetooth settings for BlueDroid stack

Special namespace prefixes

`init` recognizes several special prefixes, which govern how it handles the properties:

- **The `persist` pseudo-prefix:** designates the property as meant to survive reboot. Persistent properties are backed up by files in /data/property/, which must be owned by root:root, with no links.
- **The `ro` pseudo-prefix:** is used for "read-only" properties. These, like C constants, may be set once and once-only, irrespective of owner UID. Normally these are set as early as possible, i.e. in the vendor supplied build files.
- **The `ctl` prefix:** is used to provide a convenient way to control init's services, by setting the `ctl.start` or `ctl.stop` properties (respectively) to the service name. (The start and stop tools of toolbox are nothing more than using ctl for zygote, surfaceflinger and netd). A separate ACL is maintained in the `control_perms` array, to restrict services by UID/GID. As of KitKat, this list defined `dumpstate` (`shell:log`) and `ril-daemon` (`radio:radio`). In L SELinux takes over ACL enforcement.

Accessing properties

The `toolbox` command provides command line property access through `getprop`/`setprop`, and a property listener in the `watchprops` command. The native API for properties are defined in system/core/include/cutils/properties.h:

- `int property_get(const char *`***`key`***`, char *`***`value`***`, const char *`***`default_value`***`)` - To retrieve a property, optionally specifying a default value if it does not exist. This simply accesses the shared memory area.
- `int property_set(const char *`***`key`***`, const char *`***`value`***`)` - To set the value of a property. This serializes the key and value, and sends them over the property service socket.
- `int property_list(void (*`***`propfn`***`)(const char *`***`key`***`, const char *`***`value`***`, void *`***`cookie`***`), void *`***`cookie`***`)` - To enumerate properties using a callback function which will be invoked per property, with a pre-specified *cookie*

The `<sys/_system_properties.h>` file includes a few other undocumented (though accessible) functions, the most useful of which is `__system_property_wait_any(unsigned int `***`serial`***`)`, which blocks until any property is set. This is used by the `watchprops` command.

Framework level access to system properties is carried out through `android.os.SystemProperties`, which accesses the properties via JNI calls to the API calls.

Experiment: Using watchprops

The `watchprops` tool can be used to monitor system property changes in real time. Starting this tool as close as possible to device boot (by using `adb wait-for-device; adb shell watchprops` on the host) will still miss the build properties (since those are sourced before `adb` is started), but nonethless reveal the setting of important properties during boot, as shown in the following annotated listing:

```
# Property (timestamp omitted)           Set by
persist.sys.dalvik.vm.lib.2 = 'libart.so' Zygote decides between Dalvik and ART
sys.sysctl.extra_free_kbytes = '24300'    ProcessList::updateOomLevels()
wlan.driver.status = 'unloaded'
net.hostname = 'android-35f4ac3424467a0f' ConnectivityService()
net.change = 'net.hostname'               internally, when a net.* prop has changed
persist.sys.profiler_ms = '0'             SamplingProfiler's settingObserver
debug.force_rtl = '0'
net.qtaguid_enabled = '1'
net.change = 'net.qtaguid_enabled'
sys.settings_global_version = '1'         android.provider.Settings()
service.bootanim.exit = '1'               SurfaceFlinger (causes bootanimation exit)
sys.boot_completed = '1'                  ActivityManagerService::finishBooting()
dev.bootcomplete = '1'                      ibid, when not decrypting filesystem
init.svc.bootanim = 'stopped'             internally, once bootanimation has exit
ro.runtime.firstboot = '1418626370520'    BootReceiver::logBootEvents()
# gsm.* properties set by rild/libril
gsm.version.ril-impl = '..'
gsm.network.type = 'Unknown'
...
# rild also sets parameters, based on connection detected
sys.sysctl.tcp_def_init_rwnd = '60'
net.dns1 = '10.0.2.3'
```

The .rc Files

init's main operation involves loading its configuration file, and acting upon its directives. Traditionally, two files were used: The main /init.rc, and a device-specific /init.*hardware*.rc, where *hardware* is obtained from the androidboot.hardware kernel argument, or /proc/cpuinfo. The default emulator hardware, for example, is "goldfish" (and in M, "ranchu"), and it is not uncommon to see /init.goldfish.rc on actual devices as well, probably due to most implementors copying the default filesystem without really paying attention to detail. The original idea might have been to have all Android devices use the same /init.rc, leaving the device-specific file for vendor customizations. In practice one finds quite often that implementors simply add more directives into /init.rc.

As of JB, the only hard-coded rc file is /init.rc, and the import directive is used to include additional rc files explicitly. JB's default /init.rc also includes /init.*hardware*.rc, (imported as /init.${ro.hardware}.rc, substituting the value of the property), and /init.usb.rc (or /init.${ro.hardware}.rc), which contains the USB related directives for init - as discussed later in this chapter. An additional /init.trace.rc is also present in the default build, to enable the ftrace kernel facility to be used for debugging (Discussed in Volume III).

Triggers, actions, and services

The rc files are composed of **trigger** and **service** blocks. Trigger blocks contain commands, to be executed when a trigger is satisfied. Service blocks define daemons, which init can start by command and be responsible for, with optional modifiers (options) per such service. Service blocks start with the **service** keyword, followed by a name and the command line. Triggers are defined by the **on** keyword, followed by an argument, which is either a well-known name of a boot stage, or the **property** keyword, followed by a *property=value* expression (in case the trigger is tied to a property value change). When executing a given action or command, init sets the init.action or init.command properties, respectively. Well known boot stages are shown in table 4-4, but note, that not all boot stages need be used, and vendors often deviate (e.g. mount filesystems in the init stage)

Table 4-4: The init boot stages

Init stage	Contents
early-init	Very first stage of initialization. Used for SELinux and OOM settings
init	Creates file systems, mount points, and writes kernel variables
early-fs	Run just before filesystems are ready to be mounted
fs	Specifies which partitions to load
post-fs	Commands to run after filesystems (other than /data) are mounted
post-fs-data	/data decrypted (if necessary) and mounted
early-boot	Run after property service has been initialized, but before booting rest
boot	Normal boot commands
charger	Commands used when device is in charger mode

init.rc syntax and command set

The init.rc and its imported files are very well annotated - but also quite long. Instead of cutting/pasting them and wasting bytes and pages, we next focus on their syntax and other features, which are relatively undocumented or little known. You may want to look at /init.rc alongside reading this section.

The init_parser recognizes two types of keywords when parsing the rc files: COMMANDs, naming actions to execute on a trigger/boot-stage (valid only in a trigger block) and OPTIONs, modifiers pertaining to a service declaration (valid only in a service block). Table 4-5 shows the commands supported by init,from keywords.h. Colors correspond to different versions:

Table 4-5: Init commands

Command syntax	Notes
bootchart_init	M: Start bootcharting (if configured). M makes bootcharting optional
chdir *directory*	as cd command (calls chdir(2))
chmod *octal_perms file*	Change *octal_perms* masks of *file*
chown *user group file*	Same as chown *user*:*group file*
chroot *directory*	as Linux chroot command (calls chroot(2))
class_reset *service_class*	JB: Stop/Start all services associated with *service_class*
class_[start|stop] *class*	Start or stop
copy *src_file dst_file*	Same as cp(1) command
domainname *domainname*	Writes *domainname* to /proc/sys/kernel/domainname
exec *command*	No longer supported
enable *service*	L: Enable an otherwise disabled service
export *variable value*	Export environment *variable*. Will be inherited by all children
hostname *hostname*	Writes *hostnname* to /proc/sys/kernel/hostname
ifup *interface*	Bring up an interface (same as ifconfig *interface* up)
insmod *module.ko*	Load a kernel module
import *filename.rc*	Include an additional rc file
load_all_props	L: (Re)-Load all properties from build, default and factory files
load_persist_props	JB: (Re)-Load all persistent properties from /data/property
loglevel *level*	Set kernel loglevel (printk)
mkdir *directory*	Create a directory (calls mkdir(2))
mount *fstype fs point*	Mount a file system of *fs_htype* from *fs* on mount *point*
mount_all	Mount file systems in vold's /fstab.*hardware*. This causes init to fork and perform mounts using fs_mgr. init detects any encrypted file systems.
powerctl *shutdown/reboot*	KK: shutdown/reboot wrapper
[re]start *service_name*	Start/restart service specified in **service** block matching *service_name*
restorecon[_recursive] *path*	Restore SELinux context for path (recursive added in L)
rm[dir] *filename*	JB: Remove a file or directory (calls unlink(2)/rmdir(2), respectively)
setcon *SEcontext*	JB: Set (change) SELinux context. Init uses u:r:init:s0
setenforce *[0|1]*	JB: Toggle SELinux enforcement on/off
setkey *table index value*	Set key table
setprop *key value*	Set a system property
setsebool *value*	Set an SELinux boolean property. *value* can be 0/false/off or 1/true/on
setrlimit *category min max*	use setrlimit(2) system call to enforce process (q.v. ulimit(1))
stop *service_name*	Stop service specified in **service** block matching *service_name*
swapon_all..	KK: Activate all swap partitions in fstab
symlink *target src*	Creates a symbolic link (as ln -s - calls symlink(2))
sysclktz *tzoffset*	Set system clock timezone (using settimeofday(2)
trigger *trigger_name*	Activate a trigger (causing init to re-run corresponding commands)
verity_load_state	M: Loads DM-verity state
verity_update_state *mount*	M: Updates DM-Verity (partition encryption state) for mount point
wait *file timeout*	Wait up to *timeout* seconds for *file* to be created.
write *file value*	Writes *value* to file. Same as echo *value* > file

If you look through your /init.rc files, you will likely see these commands used during the various boot stages to perform what one might expect during system startup: Setting up the directory structure, enforcing filesystem permissions, and setting up various kernel parameters via /proc or /sys. Once the boot stages are all defined, the rest of the file will deal with service definitions. As stated, service blocks are modified by **options**. These provide the parameters by means of which init determines how the services are to be run, and monitored. Table 4-6 lists the available options.

Table 4-6: Init options

Option Syntax	Notes
capability	Supports Linux capabilities(7) (or at least, will, at some point in the future)
class	Defines the service to be part of a service group. Classes can then be manipulated together by the class_[start|stop|reset] commands.
console	Defines the service as a console service. stdin/stdout/stderr linked to /dev/console.
critical	Defines the service as a critical one. Critical services are automatically restarted. If they crash more than CRITICAL_CRASH_THRESHOLD (4) times in CRITICAL_CRASH_WINDOW (240) seconds, the system will auto-reboot into recovery mode
disabled	Indicates service will not be started. Service can still be started manually
group	Specifies the gid to start the service as. init will call setgid(2) for this.
ioprio	Specifies the I/O priority for the service. init will call ioprio_set
keycodes	Specifies a key chord that can trigger this service. (discussed below)
oneshot	Tells init to start the service, but not worry about it (that is, ignore its SIGCHLD).
onrestart	Lists which commands to invoke if/when the service needs to be restarted. This is commonly used to restart dependent services
seclabel	Specifies the SELinux label to apply to this service
setenv	Set an environment variable prior to fork()ing and exec()ing the service. Unlike export, this environment variable will only be seen by the service
socket	Tells init to open this UNIX Domain socket and let the process inherit the open socket descriptor. This enables services to work with stdin/stdout, and not worry about which sockets to open or the permissions they may require
user	Specifies the uid to start the service as. init will call setuid(2) for this.

Starting services

Although the syntax is different, when starting services init assumes the traditional function of PID 1 (the traditional init, or launchd, to start up services: It fork()s, sets up the service's permissions (by calling setuid(2)/setgid(2)), sets up any input (UNIX domain) sockets and any environment variables, I/O priority (for services with ioprio), and SELinux context. For services defined with console, init connects /dev/console to stdin/stdout/stderr, and for all others it "zaps" stdio. Though presently unsupported, init will also set the capability set for services defined with capability (as discussed in Chapter 8). Only once all of these operations have been performed, will init call execve to launch the service binary.

After the service is started, init maintains a parental link to it - should the service terminate or crash, init will receive a SIGCHLD signal, notifying it of the event - and allowing the service to be restarted. The onrestart option allows init to form dependencies between services, and run additional commands or restart dependent services when a particular service needs restarting. The critical option defines the service as a "must-have", and if init encounters a restart loop for a service deemed critical (that is, it restarts the service, only to have it crash again), it will reboot the entire system into recovery mode. For every service, init also maintains a corresponding init.svc.*service* property to reflect the service status (running/stopped/restarting).

Keychords

An interesting, (though little known) function of `init` is starting services in response to *keychords*. The chords are defined as combinations of keys (on devices with a physical keyboard) or buttons pressed by the user at any time (akin to key combinations one would press on a piano). The keys are specified by their IDs, which are taken from Linux's `evdev` input mechanism.

Note the keychords follow codes specified in Android's key layout files (usually found in /system/usr/keylayout) and **not** the same codes as specified and used by the frameworks (i.e. the codes at frameworks/native/include/android/keycodes.h). The only default service tied to a keychord is `bugreport`, defined on some devices (like the Nexus 5) to be associated with the volume and power buttons. You can find its definition in the Nexus' /init.hammerhead.rc:

Listing 4-2: The BugReport service, demonstrating the use of keychords, from /init.hammerhead.rc

```
service bugreport /system/bin/dumpstate -d -p -B \
        -o /data/data/com.android.shell/files/bugreports/bugreport
    class main
    disabled
    oneshot
    keycodes 114 115 116
```

The dumpstate command is an AOSP provided binary which iterates over all subsystems and services and dumps all diagnostics and statistics available for them. Note the service is disabled, meaning it has to be started manually, and its startup is tied to keycodes 114, 115 and 116. These, as you can verify by /system/usr/keylayout/Generic.kl are mapped to `VOLUME_DOWN`, `VOLUME_UP` and `POWER`, respectively.

For keychords to be supported, /dev/keychord must exist. This is a device node exported by the keychord kernel driver, if the kernel was compiled with `INPUT_KEYCHORD`, or the driver was installed as a module. The driver can be considered an "Androidism" of sorts, and is discussed in more detail in Volume III.

> On a rooted device (i.e one with a modified root filesystem) you can add all sorts of functionality using keychords. In the default configuration you're somewhat limited (since only physical keys can be specified), but you can still override the combinations to implement any functionality of your choice. In devices with physical keyboard or additional buttons, using keychords opens up even more possibilities

Mounting File Systems

Though Android has a dedicated volume manager daemon (`vold`), init still has to perform some mount operations by itself: Recall, that when init is started only the root filesystem is mounted - no /system or /data and therefore it falls on it to at the very least mount /system, so that the various daemons - including `vold` can start. Naturally, this is a critical operation: If neither filesystem can be mounted, /init will drop the device into recovery mode.

init recognizes the `mount_all` directive in the /init.rc (usually placed in the `on fs` trigger) as a request to perform a mount of all the default file systems. These are specified in the /fstab.*hardware* file, which is one of the files built by the AOSP. The code to handle the mount is in `fs_mgr`, which is used by both /init and `vold`. When /init performs the mount, it first forks, so as to mitigate the chance of a critical error impacting its own startup.

The child process performs the mount operations, potentially running fsck on the filesystems, if required. fs_mgr hardcodes the paths to the various checkers (presently, /system/bin/e2fsck and, as of L, /system/bin/fsck.f2fs), and those, too, are fork()ed. The fs_mgr code bumps up its logging level, so you can see its messages in the kernel ring buffer (using dmesg). If you do so early on enough (provided the buffer hasn't cycled to overwrite older messages), you will find fs_mgr flagged messages interspersed with those of EXT4-fs, F2FS-fs (both kernel modules providing respective filesystem support), and SELinux (which is enforced on the filesystems if extended attributes are detected.

It is thus, that the child process handles the mounts, and returns a code to the parent (as all children do). It is according to this return value that /init sets the value of the vold.decrypt property, which will be later picked up by vold to handle decryption of the filesystem, if necessary. If no filesystems are encrypted, /init fires the nonencrypted trigger.

Putting it all together: The flow of init

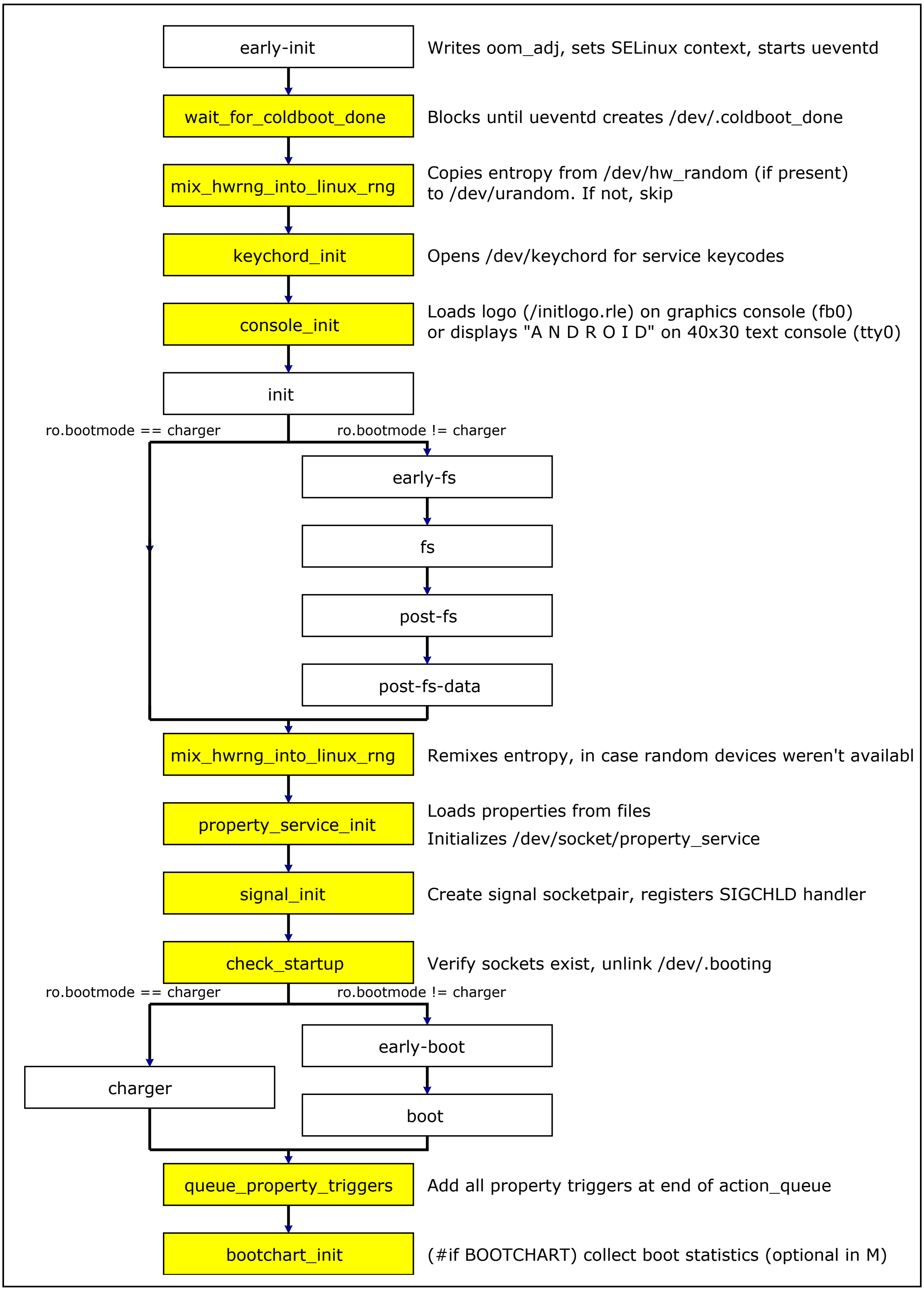
As is the pattern with most daemons, /init's code follows a classic server setup: initialization, followed by a run-loop, which (hopefully) never exits.

Initialization

/init's initialization consists of the following steps:

- Check if the binary was invoked as ueventd or (as of KitKat) watchdogd. If so, the rest of the flow is diverted to the corresponding main loop for either of those daemons, discussed later in this chapter.
- Create directory entries for /dev, proc, and sys, and mount them.
- Touch (open and then close) /dev/.booting. This file is cleared once startup is complete (by check_startup (q.v. Figure 4-2).
- open_devnull_stdio() to "daemonize" (link stdin/stdout/stderr to /dev/null).
- klog_init() creates /dev/__kmsg__ (Major 1, Minor 11), and immediately deletes it.
- property_init() creates the shared property area in memory, as discussed earlier in this chapter in "System Properties"
- get_hardware_name() gets the hardware name by reading /proc/cpuinfo and extracting the "Hardware:" line. Rather crude, but it works (at least, on ARM architectures)
- process_kernel_cmdline reads /proc/cmdline and imports as properties any arguments beginning with androidboot as ro.boot.
- SELinux is initialized, on JellyBean and later. In JB it is still conditionally #ifdef'ed HAVE_SELINUX. In KK SELinux is assumed to be available by default. The SELinux security contexts are restored for /dev and /sys.
- A special check is made to see if the device is in "charger mode" (as indicated by an androidboot kernel argument). This will divert the flow of init by skipping most of the initialization stages, and loading only the charger class of services (which presently contains only the charger daemon). If the device is not in charger mode, init proceeds to load /default.prop, and the boot up proceeds normally.
- init_parse_config_file() is called to parse /init.rc.
- init enqueues the actions supplied in the init.rc sections (using action_for_each_trigger()) and the built-in actions (queue_builtin_action) on an action_queue. The resulting queue is shown in Figure 4-2.

Figure 4-2: The init boot stages (in white) and built-in commands (in yellow)


early-init
Writes oom_adj, sets SELinux context, starts ueventd
wait_for_coldboot_done
Blocks until ueventd creates /dev/.coldboot_done
mix_hwrng_into_linux_rng
Copies entropy from /dev/hw_random (if present) to /dev/urandom. If not, skip
keychord_init
Opens /dev/keychord for service keycodes
console_init
Loads logo (/initlogo.rle) on graphics console (fb0) or displays "A N D R O I D" on 40x30 text console (tty0)
init
ro.bootmode == charger
ro.bootmode != charger
early-fs
fs
post-fs
post-fs-data
mix_hwrng_into_linux_rng
Remixes entropy, in case random devices weren't availabl
property_service_init
Loads properties from files
Initializes /dev/socket/property_service
signal_init
Create signal socketpair, registers SIGCHLD handler
check_startup
Verify sockets exist, unlink /dev/.booting
ro.bootmode == charger
ro.bootmode != charger
early-boot
charger
boot
queue_property_triggers
Add all property triggers at end of action_queue
bootchart_init
(#if BOOTCHART) collect boot statistics (optional in M)

Eventually, the main loop iterates through at all the init.rc commands, and `init` spends most of its days asleep, polling the file descriptors, optionally logging to bootchart, and waking up only when necessary. You can see init's file descriptors by looking at the /proc file system:

Output 4-2: Init's file descriptors, as seen through /proc/1/fd:

```
root@generic:/proc/1 # ls -l fd
lrwx------  .... 0 -> /dev/__null__ (deleted) #
lrwx------  .... 1 -> /dev/__null__ (deleted) #  stdin, stdout and stderr closed
lrwx------  .... 2 -> /dev/__null__ (deleted) #

l-wx------  .... 3 -> /dev/__kmsg__ (deleted) # Major: 1, Minor: 11

lr-x------  .... 4 -> /dev/__properties__     # read-only property store, for children

lrwx------  .... 5 -> socket:[1643]           # property_set_fd (/dev/socket/property_service)

lrwx------  .... 6 -> socket:[1645]           # signal_fd       (socketpair[0])
lrwx------  .... 7 -> socket:[1646]           # signal_recv_fd  (socketpair[1])

lrwx------  .... 9 -> socket:[1784]
```

The run loop

The main run-loop is also quite simple, consisting only of three steps:

- `execute_one_command()` - dequeues the action at the head of the queue, if any, and performs it.
- `restart_processes()` - which iterates over all registered services and restarts them, if necessary
- Set up and poll (monitor) three socket descriptors:
  - The `property_set_fd` (/dev/socket/property_service), through which client processes who wish to set a property pass the property key and value. The `property_service` code obtains the peer's credentials (using `getsockopt(..SO_PEERCRED..)`), and performs the permission checks on the property. If it can be set, any triggers or related services (for `ctl.start/stop` properties) are executed as well. If SELinux is enabled, /init calls on it to enforce the /property_contexts.
  - The `keychord_fd` (/dev/keychord, if it exists), which handles any service key-combinations, as discussed previously
  - The `signal_recv_fd` , one end of a `socketpair`, created to handle `SIGCHLD` from dead offspring. When the signal is received, the `sigchld_handler` writes data to the other end of the `socketpair` (`signal_fd`), making data available on the receiving end, and causing `init` to call `wait_for_one_process(0)`. This reaps the process' return value (so as to lay it to rest, and avoid a zombie), cleans up any sockets, and potentially restarts the process, if it is a tracked service.

It's important to emphasize that, aside from listening on the file descriptors, /init accepts no other input from any source. In other words, there is no way to affect /init's operation. This is by design, since /init remains an unrestricted, root-owned process. The only way to modify /init's operation requires editing of the /init.rc files, which - being part of the root file system - are on a separate partition (along with the kernel), and digitally signed, so as to reject modifications in all but bootloader-unlocked devices.

Init and USB

An Android device occasionally needs to change its behavior as a USB attachment based on user preference - act as a mass storage device, emulate a digital camera, start up or shutdown ADB, and more. Rather than use a dedicated daemon to toggle behavior, the responsibility falls on init, which communicates with the USB components in the kernel.

USB behavior is dictated the `sys.usb.config` system property. The frameworks (specifically, UsbDeviceManager and its related classes) set the value of this property according to the user choice, and init - being the keeper of all system properties - picks up any changes and applies them using a trigger. For convenience, the property triggers are maintained separately in init.*hardware*.usb.rc. This can be seen in the following listing, demonstrating the contents of this file on a Nexus 5:

Listing 4-3: USB settings from `init.hammerhead.usb.rc`, from a Nexus 5

```
on init
    write /sys/class/android_usb/android0/f_rndis/manufacturer LGE
    write /sys/class/android_usb/android0/f_rndis/vendorID 18D1
    write /sys/class/android_usb/android0/f_rndis/wceis 1

on boot
    write /sys/class/android_usb/android0/iSerial $ro.serialno
    write /sys/class/android_usb/android0/iManufacturer $ro.product.manufacturer
    write /sys/class/android_usb/android0/iProduct $ro.product.model

# MTP
on property:sys.usb.config=mtp
    stop adbd
    write /sys/class/android_usb/android0/enable 0
    write /sys/class/android_usb/android0/idVendor 18D1
    write /sys/class/android_usb/android0/idProduct 4EE1
    write /sys/class/android_usb/android0/bDeviceClass 0
    write /sys/class/android_usb/android0/bDeviceSubClass 0
    write /sys/class/android_usb/android0/bDeviceProtocol 0
    write /sys/class/android_usb/android0/functions mtp
    write /sys/class/android_usb/android0/enable 1
    setprop sys.usb.state ${sys.usb.config}
..
```

As shown in the figure, `init`'s response to property changes involves writing parameters to /sys/class/android_usb/android0. The receiving end of these pseudo files is the **USB gadget driver**. This, as the name implies, is a multipurpose driver which can emulate any aspect of USB functionality, as dictated from user mode. The functions this driver can handle correspond to the USB modes shown in Table 4-usb:

Table 4-usb: USB Modes recognized by the USB gadget driver

accessory	Connecting accessories to the device - implements AoA protocol
acm	Abstract Control Model (USB Modems)
adb	Android Debugger Bridge (adbd) functionality. Creates /dev/android_adb device node, over which a host can communicate with the device's adbd.
audio_source	USB Audio source (when connected to external speakers). Provides PCM playback
mass_storage	Mass storage device (portable disk)
mtp	Media Transfer Protocol. Identifies as camera, and creates kernel thread to handle MTP requests. Creates /dev/mtp_usb
rndis	USB Remote NDIS, used when USB tethering the device

To effectuate the changes, the driver needs to be disabled and reenabled. This is why toggling "USB Debugging" or tethering temporarily disconnects the device from its host (as you can see by observing the host's kernel messages or, if you have a virtual machine, getting a pop-up).

Experiment: Modifying device USB identification

The following experiment shows how you can control the device's USB personality. This is demonstrated on a Galaxy S3, but the steps work on all devices.

Output 4-3: The USB personality files on a Galaxy S3

```
shell@s3:/sys/class/android_usb/android0# ls
bDeviceClass
bDeviceProtocol
bDeviceSubClass
bcdDevice
enable            # Toggles enable/disable

f_accessory@      #
f_acm@            #
f_adb@            #
f_ccid@           #
f_diag@           #
f_mass_storage@   #  Exported gadget
f_mtp@            #   driver functions
f_ncm@            #
f_ptp@            #
f_rmnet@          #
f_rmnet_sdio@     #
f_rmnet_smd@      #
f_rmnet_smd_sdio@ #
f_rndis@          #

functions         # controls functionality
host_state
iManufacturer     # Holds vendor string (e.g SAMSUNG)
iProduct          # Holds product ID reported: e.g. SAMSUNG_Android_SGH-I747
iSerial           # Holds serial # reported by adb
idProduct         # vendor's product id
idVendor          # well known vendor id (e.g. Intel: 8086)
power
remote_wakeup
state
subsystem
terminal_version
uevent
shell@s3:/sys/class/android_usb/android0 $ cat functions
mtp,acm,adb
shell@s3:/sys/class/android_usb/android0 $ cat iProduct
SAMSUNG_Android_SGH-I747
```

Changing the iProduct will change the string with which the device identifies itself to the host (for example, Kindle HDX's string is "Lab126 Android", whereas some Chinese GooPhones identify as "Apple iPhone"..). Changing the iSerial will change the string reported by `"adb devices"` (useful for `"adb -s"`), and the iManufacturer or iProduct can similarly be changed. You can test this for yourself by writing a string of your choice, then disconnecting and reconnecting the USB cable to the host. Note, these changes do not persist across a reboot, but you can easily write them to the /init.*hardware*.usb.rc (or applicable file on your device) if you like your new device identity.

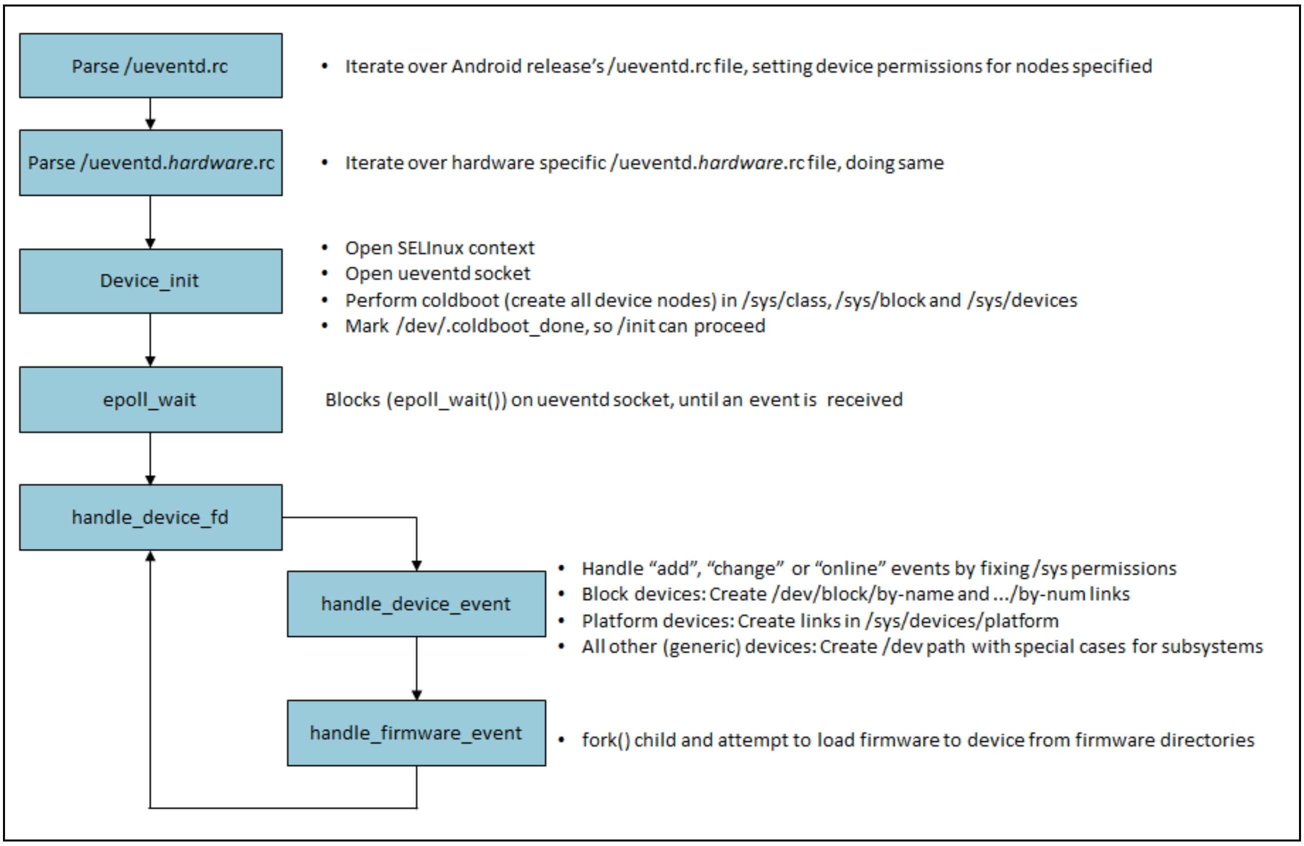
The Other Roles of init

As discussed in the last section, init can also be fill additional roles - that of ueventd and (as of KitKat) watchdogd. Even though these are filled by the same binary, the code path taken is an entirely different one, and is chosen before any other initalization step.

ueventd

As ueventd, init assumes the responsibility of managing hardware devices: Responding to kernel notifications and device representations in the /sys filesystem, and making them available to processes via symbolic links it creates in /dev. It uses different initialization files - that is, it consults /ueventd.rc and /ueventd.*hardware*.rc, where *hardware* is obtained from /proc/cpuinfo, or the androidboot.hardware kernel argument. Unlike init, however, the configuration file(s) only contain entries related to device nodes and their permissions. ueventd iterates over the lines of the file and calls set_device_permission() for every device entry.

Figure 4-3: The flow of ueventd


Parse /ueventd.rc
Iterate over Android release's /ueventd.rc file, setting device permissions for nodes specified
Parse /ueventd.hardware.rc
Iterate over hardware specific /ueventd.hardware.rc file, doing same
Device_init
Open SELInux context
Open ueventd socket
Perform coldboot (create all device nodes) in /sys/class, /sys/block and /sys/devices
Mark /dev/.coldboot_done, so /init can proceed
epoll_wait
Blocks (epoll_wait()) on ueventd socket, until an event is received
handle_device_fd
handle_device_event
Handle "add", "change" or "online" events by fixing /sys permissions
Block devices: Create /dev/block/by-name and .../by-num links
Platform devices: Create links in /sys/devices/platform
All other (generic) devices: Create /dev path with special cases for subsystems
handle_firmware_event
fork() child and attempt to load firmware to device from firmware directories

The next step is to call device_init(), which initializes a NETLINK_UEVENT socket. If a cold boot is detected (i.e. if the /dev/.coldboot_done cannot be found), ueventd iterates through the /sys/class, /sys/block and /sys/devices subtrees, writing "add" to uevent psuedo-files in each. This triggers uevent device addition notifications, which ueventd otherwise might have missed prior to its startup.

Once the socket (`device_fd`) is initialized, the role of `ueventd` becomes very simple: continuously poll it, read events as they become available, and handle them. Events may be of two general types:

- **Device events:** These events are generated by kernel subsystems when devices are added or removed. In this sense, `ueventd` functions like the traditional Linux `udevd`: It creates or removes `/dev` nodes corresponding to the devices.
- **Firmware events:** `ueventd` listens on firmware "add" events and attempts to load firmware updates from /etc/firmware, /vendor/firmware and /firmware/image.

If compiled with `-DLOG_UEVENTS`, `ueventd` will log events as `INFO` messages.

watchdogd

Just like `ueventd`, `watchdogd` is another facet of `init`. In this identity, it is responsible for interfacing with the hardware watchdog timer (/dev/watchdog, if present), by setting a timeout value, and sending a keepalive signal (a null byte) at regular intervals. If the timeout value passes and `watchdogd` fails to wake up in time, the hardware timer interrupt can be used by the kernel to reset. While a somewhat drastic measure, the only reason `watchdogd` wouldn't wake up in time would be a system hang. It's likely the system wouldn't recover from such a hang, and and therefore it is simpler to restart the device.

As `watchdogd`, the daemon accepts two command line arguments - the interval, and a margin - both in seconds, with initial values of 10. The overall device timeout value is the sum of both (i.e. 20, by default), allowing for some leeway before the drastic measure of a reboot is taken.

Summary

This chapter explored all aspects of /init, the most critical of system processes without which there would be no user-mode. We started by comparing it to its Linux and UN*X counterparts, then moved on to explore its single most important idiosyncratic feature - System Properties. We next discussed the syntax of its rc files, and constructed the full flow.

The next chapter explores the services themselves. Focusing on the default system daemons, and then going on to `system_server`, which provides support for all of Android's frameworks. The actual framework services - which number in the many dozens - require much detail from a programmatic perspective, which is why they have been left for Volume II.

Files discussed in this Chapter

Section	File/Directory	Contains
init	system/core/init	The code of /init
	/system/core/init/readme.txt	Documentation on commands and triggers
ueventd	system/core/init/ueventd.[ch]	The code of /init's `ueventd` persona
watchdogd	system/core/init/watchdogd.[ch]	The code of /init's `watchdogd` persona

V: Android Daemons

Android has quite a few daemons running in the background for providing its miscellaneous housekeeping and operational functions. The services are mostly strewn in /init.rc without much ordering, save the service class. The "core" services start first, followed by the "main" ones. The rc also defines a "late_start" class, for services which depend on the /data partition, though no default services belong to it. In this section, we adopt the service class division, but - since most services are in "main" - further subcategorize by function.

Following our discussion of init in the previous chapter, we continue to cover the Core Services - adbd, the servicemanager and KitKat's healthd, as well as new core services added in L: lmkd and logd.

All other services are generally classified into the "main" category, so a subcategorization by Network Services (netd, mdnsd, mtpd and rild), and Graphics and Media Services (surfaceflinger, bootanimation, mediaserver and drmserver) follows. The remaining services are hard to group, so they are placed into the "Other Services" category, which includes installd, keystore, debuggerd, sdcard and - last, but far from least - Zygote.

Core Services

The services in the "core" class are the first to be started during user-mode boot. These services do not access the /data partition, and therefore can run irrespective of whether or not it is mounted.

adbd

If you're reading this book, likely ADB needs no introduction: It is through this medium, known as the *Android Debugger Bridge**, that the host and the device communicate. The bridge can be used either directly (using the adb command) or indirectly (using ddms). The adb command itself is well documented, and running it without any arguments will display a (rather lengthy) usage message. Of more interest to our discussion is how ADB actually works.

In its basic configuration, the adbd, which is the device daemon providing the server functionality of ADB, is defined in the /init.rc, albeit disabled, and started on demand in /init.usb.rc, when the sys.usb.config property contains "adb" - which is what the well-known "USB Debugging" GUI option activates:

Listing 5-1: adb definitions in the rc files (KitKat)

```
# adbd is controlled via property triggers in init..usb.rc
service adbd /sbin/adbd
    class core
    socket adbd stream 660 system system
    disabled
    seclabel u:r:adbd:s0 # As of JB, ADB gets its own SELinux context
```

Note that the adbd is run by default as uid root. It does, however, drop privileges to run as uid shell:shell, along with several other groups, as shown in this snippet from adb.c:

Listing 5-2: The adb main startup function, showing privilege settings

```
int adb_main(int is_daemon, int server_port)
..

 property_get("ro.adb.secure", value, "0");
 auth_enabled = !strcmp(value, "1");

 if (auth_enabled)
        adb_auth_init();

 ...

     /* add extra groups:
     ** AID_ADB to access the USB driver
     ** AID_LOG to read system logs (adb logcat)
     ** AID_INPUT to diagnose input issues (getevent)
     ** AID_INET to diagnose network issues (netcfg, ping)
     ** AID_GRAPHICS to access the frame buffer
     ** AID_NET_BT and AID_NET_BT_ADMIN to diagnose bluetooth (hcidump)
     ** AID_SDCARD_R to allow reading from the SD card
     ** AID_SDCARD_RW to allow writing to the SD card
     ** AID_MOUNT to allow unmounting the SD card before rebooting
     ** AID_NET_BW_STATS to read out qtaguid statistics
     */


     gid_t groups[] = { AID_ADB, AID_LOG, AID_INPUT, AID_INET, AID_GRAPHICS,
                        AID_NET_BT, AID_NET_BT_ADMIN, AID_SDCARD_R, AID_SDCARD_RW,
                        AID_MOUNT, AID_NET_BW_STATS };
```

* - Interestingly, Samsung's Tizen uses the "smart debugger bridge" or sdb, which in almost all ways (including command line syntax!) is a complete clone of ADB.

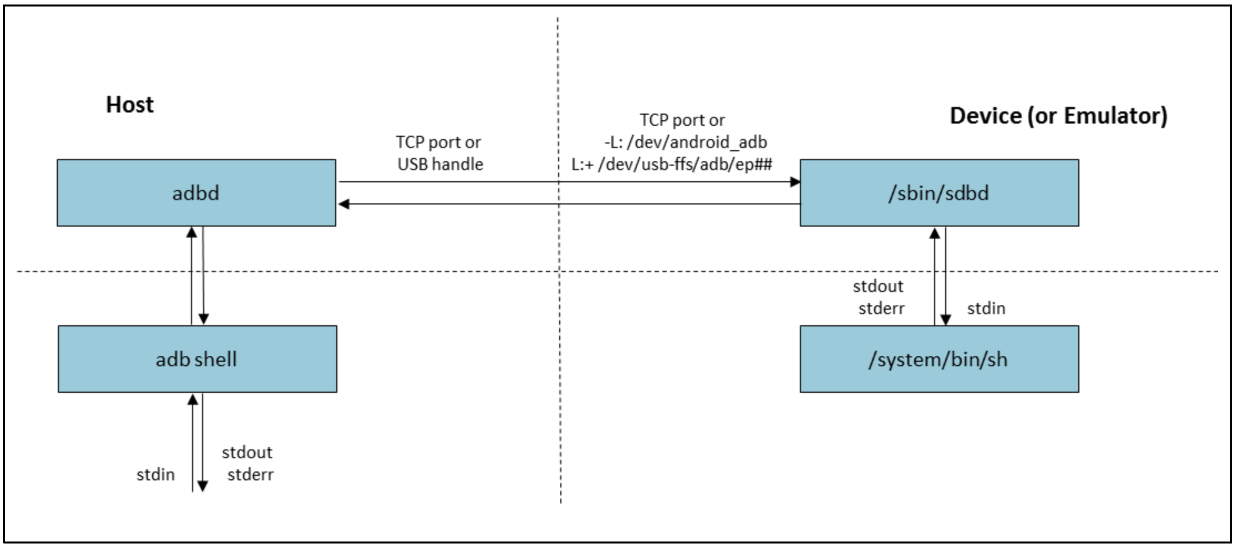
Listing 5-2 (cont.): The adb main startup function, showing privilege settings

```
     if (setgroups(sizeof(groups)/sizeof(groups[0]), groups) != 0) {
         exit(1);
     }

    /* don't listen on a port (default 5037) if running in secure mode */
    /* don't run as root if we are running in secure mode */
    if (should_drop_privileges()) {
        drop_capabilities_bounding_set_if_needed();

     /* then switch user and group to "shell" */
     if (setgid(AID_SHELL) != 0) {
         exit(1);
     }
     if (setuid(AID_SHELL) != 0) {
         exit(1);
     }
..
```

It's possible to make `adbd` to retain its privileges (from the host, running `adb root`, which sets the `service.adb.root` to 1). A limitation in the adb source permits this only if the `ro.debuggable` property is set to 1 (and otherwise prints the familiar error "adbd cannot run as root in production builds"). The `persist.adb.trace_mask` can contain a hexadecimal value specifying the logging mask (try setting it to 0xff for maximum verbosity). If the property exists and is valid, adb will log to /data/adb/adb-%Y-%m-%d-%H-%M-%S.

The `adbd` normally uses the /dev/socket/adb UNIX Domain socket, as set up by `init`, but also uses /dev/android_adb (or, as of L, the functionfs endpoints in /dev/usb-ffs/adb/ep##) when connecting to the host over USB (i.e. not in the emulator). The latter is a device node which is created by the USB Gadget Driver. The `adbd` can also be made to listen on the TCP port specified by the `service.adb.tcp.port` property, or in its absence the `persist.adb.tcp.port` property. In any of these cases, the architecture can be generialized as shown in Figure 5-1:

Figure 5-1: The adb Architecture



Tracing the ADB Protocol

The ADB protocol is described in the protocol.txt file in its implementation, and therefore does not merit much further discussion here. ADB has a simple, yet efficient tracing mechanism in the form of the ADB_TRACE environment variable. This variable, when exported to an adb session, causes the client side binary to verbosely print the protocol commands. The source for adb lists several options for this variable (all, adb, sockets, packets, rwx, usb, sync, sysdeps, transport, jdwp) though in practice "transport" is the most useful one for viewing ADB messages - though without indicating the message direction:

Output 5-1: Tracing adb protocol commands using ADB_TRACE=transport

```
morpheus@Forge (~)$ ADB_TRACE=transport adb shell
30303063 000c                                   # Message of length 12:
686f73743a76657273696f6e host:version
4f4b4159 OKAY                                   # Reply
30303034 0004
30303166 001f
30303132 0012                                  # Message of length 18:
686f73743a7472616e73706f72742d61 host:transport-a
4f4b4159 OKAY                                  # Reply
30303036 0006                                  # Message of length 6:
7368656c6c3a shell:
4f4b4159 OKAY                                  # Reply
```

ADB Security

Because ADB is a portal into such powerful debugging and tracing capabilities, it naturally poses a significant security risk. Running as uid shell is still rather far from root access, but nonetheless provides powerful abilities by virtue of the various group memberships (log and graphics, to name but a few). Using ADB it's trivial to access the user's personal data, including the lock screen sequence, as well as upload any application or binary to the device. For this reason, later versions of JellyBean take a step to secure ADB by introducing public key authentication, through the AUTH message, if the ro.adb.secure is enabled (as can be seen in listing 5-2).

The AUTH message is sent in response to an OPEN, demanding authentication before any more commands can be exchanged. The argument of AUTH is always a TOKEN, which is an array of 20 random bytes collected from the device's entropy source (/dev/urandom). The host is expected to answer by signing the token with its private key (which will be generated and stored in $HOME/.android/adbkey), using an AUTH reply specifying a SIGNATURE argument, and the random bytes encrypted (read: signed) by its private key. If the corresponding public key is known to the device, verification can ensue, and - if successful - the session may move to the online state.

As with all things related to public keys, there is the chicken and egg problem of making the public key known a priori, so it can be used for verification. The default is to allow the host to respond with a RSAPUBLICKEY argument. Since the key cannot be trusted, ADB shoves the key through its /dev/socket/adbd to system_server (specifically, UsbDebuggingManager, which is started by com.android.server.usb.UsbDeviceManager), which in turn pops up a dialog (com.android.systemui.usb.UsbDebuggingActivity), asking the user to confirm the thumbprint (MD5 hash) of the key. If the user agrees, the key is added to the adb key store, in /data/misc/adb/adb_keys. Note that vendors can easily recompile adbd to remove this functionality (in adb_auth_client.c:db_auth_confirm_key()) and allow only hard-coded, vendor supplied keys.

You can see the USB Debugging State, along with the adb_keys, in the output of dumpsys usb. Note the similarities to SSH known_hosts files, which likely served as inspiration:

Output 5-2: Dumping USB debugging state with dumpsys usb

```
shell@hammerhead:/ $ dumpsys usb
...
  USB Debugging State:
    Connected to adbd: true
    Last key received: null
    User keys:
QAAAAAGih7j/oQP+S8AmUvBrpjxGY/5yppWThz4mpP6U9wt/fzGyip4sNt/2cp+40rRb8whQLALvPS2fAwLm1LjSTmJ/
... Public Key (as Base64)
+a+2cNPxxtmOh6GzOcnmwPaVsQcMLkyx1yCCS2o4hnjKYmjqBQEAAQA= morpheus@Forge

    System keys:
IOException: java.io.FileNotFoundException: /adb_keys: open failed: ENOENT
 (No such file or directory)
 ...
```

servicemanager

The servicemanager is a key component of Android's IPC mechanism. Though a small binary, it is an important one, without which inter process communication would be severely impaired. This is reflected in its defition in the /init.rc, as shown in Listing 5-3:

Listing 5-3: The servicemanager definition in /init.rc

```
service servicemanager /system/bin/servicemanager
    class core
    user system
    group system
    critical
    onrestart restart healthd
    onrestart restart zygote
    onrestart restart media
    onrestart restart surfaceflinger
    onrestart restart inputflinger   // Android L
    onrestart restart drm
```

What makes servicemanager so critical, and makes so many other services dependent upon it, is its function as a service mapper. Virtually every IPC mechanism requires a mapper to enable clients to find and connect to the services - UN*X has its portmapper (for sunrpc), Windows has its DCE endpoint mapper - and the servicemanager fulfills this function in Android. Given this, the definition in /init.rc should make sense - it's not that the services actually require the servicemanager, so much as that in the case of its untimely demise, clients would be unable to find them. When the servicemanager is restarted, it does so with a tabula rasa - which requires services to re-register in order to be found. Since there is no method for services to detect the manager is dead, the only way is to get them to re-register is to force restart them as well.

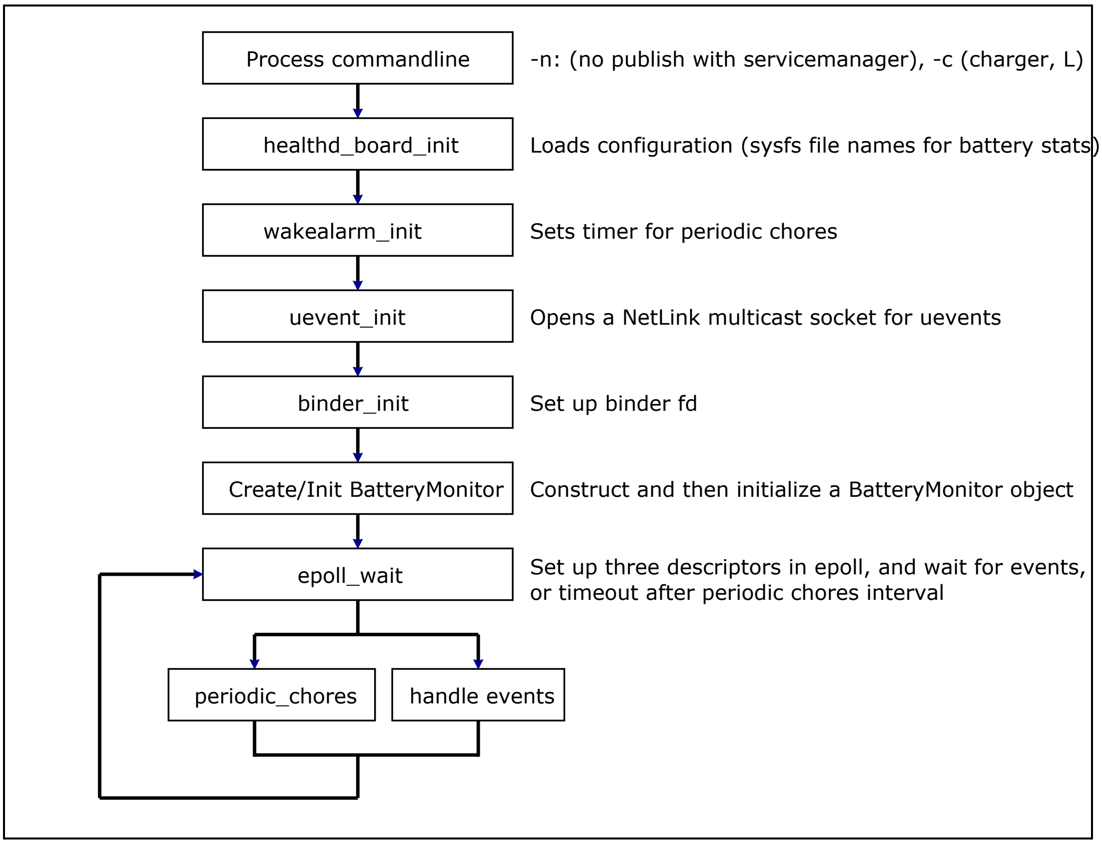
The servicemanager certainly merits more attention, as do all the framework services, which it supports. The next chapter discusses it in detail, alongside system_server (the process serves as the service host) and its individual services.

healthd

The "health daemon" is meant to service general "device health" tasks periodically, though at present the only tasks are battery related (this will likely change in future releases). The daemon registers itself as the BatteryPropertiesRegistrar service (batterypropreg or batteryproperties in L). As the Registrar, healthd provides the framework services (e.g. BatteryStatsService) with up-to-date battery statistics, which it obtains from sysfs.

Like most daemons, healthd sets up an initial configration, and then enters a run loop. The detailed flow is shown in Figure 5-2:

Figure 5-2: The flow of healthd


Process commandline
-n: (no publish with servicemanager), -c (charger, L)
healthd_board_init
Loads configuration (sysfs file names for battery stats)
wakealarm_init
Sets timer for periodic chores
uevent_init
Opens a NetLink multicast socket for uevents
binder_init
Set up binder fd
Create/Init BatteryMonitor
Construct and then initialize a BatteryMonitor object
epoll_wait
Set up three descriptors in epoll, and wait for events, or timeout after periodic chores interval
periodic_chores
handle events

Healthd main loop blocks on the Linux epoll(2) API to multiplex read operations on three descriptors, and registers actions for each, as shown in the following table:

Table 5-1: The file descriptors held by healthd and their purpose

Descriptor	Type	Purpose
wakealarm_fd	TimerFD	Timer set to fire every periodic_chores_interval seconds. Upon wakeup, healthd runs periodic_chores.
event_fd	NetLink	Reads kernel notification events. healthd only concerns itself with those of the power subsystem (SUBSYSTEM=POWER). These events include battery and charger notifications, and healthd runs battery_update().
binder_fd	/dev/binder	Listener updates by framework clients (when acting as batterypropreg)

The first descriptor polled is the `wakealarm_fd`, which `healthd` uses for its periodic chores. Two interval types are used: fast (1 minute, when on AC power), and slow (10 minutes, on battery)*. The only chore presently defined is `battery_update()`, which updates battery statistics in healthd's role as the `BatteryPropertiesRegistrar`. This is also called when events from the `POWER` subsystem are received over NetLink from `event_fd`: `healthd` makes no attempt to parse the events, and merely refereshes the battery statistics. The latter mode is required in order for `healthd` to respond to events such as charger [dis]connection, or other power management alerts. Finally, the `binder_fd` is used to interact with the framework listeners (primarily, the `BatteryStatsService`), as described in the next chapter.

Experiment: Observing `healthd`

Using the powerful `strace(1)` utility you can watch `healthd` behind the scenes: By attaching to its process ID (as root) and calling on the `ptrace(2)` API, `strace(1)` can get notifications of system calls. Because anything meaningful a process does goes through a system call, this will provide a detailed trace of the activity, and reveal the names of the sysfs files `healthd` uses to obtain its statistics, as shown in the following annotated output:

Output 5-3: Using `strace(1)` on `healthd`

```
root@htc_m8wl:/ # ls -l /proc/$healthd_pid/fd | cut -c'1-10,55-'
lrwx------ 0 -> /dev/null
lrwx------ 1 -> /dev/null
lrwx------ 2 -> /dev/null
l-wx------ 3 -> /dev/__kmsg__ (deleted)                   # Output: Log to kernel
lrwx------ 4 -> socket:[6951]                             # event_fd (NetLink socket)
lrwx------ 5 -> /dev/binder                               # binder_fd
lrwx------ 6 -> anon_inode:[eventpoll]                    # epollfd
l-wx------ 7 -> /dev/cpuctl/apps/tasks                    # fg_cgroup_fd (libcutils)
l-wx------ 8 -> /dev/cpuctl/apps/bg_non_interactive/tasks # bg_cgroup_fd (libcutils)
lr-x------ 9 -> /dev/__properties__                       # r/o property fd
root@htc_m8wl:/ # strace -p $healthd_pid
Process $healthd_pid attached - interrupt to quit
# healthd patiently polling (0xffffffff = indefinitely) until an fd signals an event
epoll_wait(0x6, 0xbebb5898, 0x2, 0xffffffff) = 1
# NetLink msg received on fd 4 (event_fd) - indicating core state change (going offline)
recvmsg(4, {msg_name(12)={sa_family=AF_NETLINK, pid=0, groups=00000001},
msg_iov(1)=[{"offline@/devices/system/cpu/cpu1"..., 1024}], msg_controllen=24,  ....
# healthd's not interested, so it goes back to polling
epoll_wait(0x6, 0xbebb5898, 0x2, 0xffffffff) = 1
# message indicating change in battery status:
recvmsg(4, {msg_name(12)={sa_family=AF_NETLINK, pid=0, groups=00000001},
msg_iov(1)=[{"change@/devices/platform/htc_bat"..., 1024}], msg_controllen=24,
{cmsg_len=24, cmsg_level=SOL_SOCKET, cmsg_type=SCM_CREDENTIALS{pid=0, uid=0, gid=0}},
msg_flags=0}, 0) = 488
#
# healthd goes into a flurry of statistics collection, opening and closing files:
#
open("/sys/class/power_supply/battery/present", O_RDONLY) = 10     # Is battery present?
read(10, "1\n", 16)                     = 2                        # Yes (1)
close(10)                               = 0
open("/sys/class/power_supply/battery/capacity", O_RDONLY) = 10    # What is its capacity?
read(10, "96\n", 128)                   = 3                        # 96%
close(10)                               = 0
open("/sys/class/power_supply/battery/batt_vol", O_RDONLY) = 10    # Voltage?
read(10, "4303\n", 128)                 = 5
close(10)                               = 0
...
open("/sys/class/power_supply/wireless/online", O_RDONLY) = 10   # Alas, no wireless charging
read(10, "0\n", 128)                    = 2                      # for the M8
close(10)                               = 0
write(3, "<6>healthd: battery l=96 v=4 t=2".., 51) = 51          # Report to kernel log
ioctl(5, BINDER_WRITE_READ, 0xbebb5070)        = 0               # Report to client listeners
epoll_wait(0x6, 0xbebb5898, 0x2, 0xffffffff) = ..                # Back to polling
```

* - Interestingly enough, in many Android releases the `timerfd_create` call returns -EINVAL (Invalid argument), not creating `wakealarm_fd` and thus defaulting to polling on the `event_fd` as an event source alone.

Experiment: Observing healthd (cont.)

Note the sysfs psuedo files (/sys/class/power_supply/*) are standard - in practice they are symbolic links to the specific platform device nodes, which change between devices.

As an improvement on the above, you might want to send the strace into the background (by using &) and then disconnect and reconnect the USB cable. You will then see the NetLink notification for battery change, followed by a change in /sys/class/power_supply/usb/online (from 1 to 0 on disconnect, or vice versa on connect).

As of Android L, healthd supports dumpsys. You can actually take the Android L binary (from a Nexus 5 or Emulator) and copy it to a device, as shown in this output:

Output 5-4: Running L's healthd on KK

```
# Before: Only KK healthd - note old service name (batterypropreg)
#
root@htc_m8wl:/ # service list | grep batteryprop
91      batterypropreg: [android.os.IBatteryPropertiesRegistrar]
root@htc_m8wl:/ # /data/local/tmp/healthd.L &                       # Run healthd from L
[1] 7287
# After: new service name (batteryproperties) added. Name is different, so no conflict
#
root@htc_m8wl:/ # service list | grep batteryprop
0       batteryproperties: [android.os.IBatteryPropertiesRegistrar] # L: diff. name, same iface
92      batterypropreg: [android.os.IBatteryPropertiesRegistrar]
root@htc_m8wl:/ # dumpsys batteryproperties                         # Calling dumpsys
ac: 0 usb: 1 wireless: 0
status: 5 health: 2 present: 1
level: 100 voltage: 4 temp: 273
```

If you use strace to watch behind the scenes of dumpsys, you'll see the following output (file descriptors are different here, so they've been symbolically replaced)

Output 5-5: Running strace concurrently on Output 5-4

```
epoll_pwait(epoll_fd, {{EPOLLIN, {u32=37597, u64=12884939485}}}, 2, -1, NULL) = 1
ioctl(binder_fd, BINDER_WRITE_READ, 0xbeab1748)         = 0     # Incoming binder req
write(...tasks, healthd_pid, 4)                         = 4     # Make healthd foreground
..
write(new_fd, "ac: 0 usb: 1 wireless: 0\n", 25) = 25            # Write output
write(new_fd, "status: 5 health: 2 present: 1\n", 31)  = 31     # to binder supplied
write(new_fd, "level: 100 voltage: 4 temp: 273\n", 32) = 32     # file descriptor.
fsync(new_fd)              = -1 EINVAL (Invalid argument)
ioctl(binder_fd, BINDER_WRITE_READ, 0xbeab1600)       = 0
close(new_fd)                               = 0
write(...tasks, healthd_pid, 4)                         = 4     # Make healthd background
..
ioctl(binder_fd, BINDER_WRITE_READ, 0xbeab1758)       = 0
```

This example, aside from showing the inner workings of healthd on L, also demonstrates an important part of Android: IPC over binder. In the above, you can see how a file descriptor has been passed from the calling process (dumpsys) to healthd. Binder internals are a complicated discussion in their own right, and are left for Volume II of this work.

healthd as charger

In Android versions leading up to L, Android had a special daemon - `charger` - which was started by `init` when the system was detected to boot in charger mode (via a `class_start charger` directive, containing only a single service). In L, `charger` has been merged into `healthd`, which makes sense, as `healthd`'s main task is observing the battery state anyway. When running as `charger`, healthd starts up in a manner similar to `init`'s additional personae (`watchdog` and `ueventd`) described previously. In other words, /bin/charger is now merely a symbolic link to /sbin/healthd, which also starts with a `-c` command line argument. The `charger` daemon is responsible for relaying the battery status graphically to the user while the device is charging. It does so using the `MinUI` library (which is covered in depth in Volume II).

Though merely a speculation, it is likely that `healthd` will be augmented and play an increasingly larger role in Android, possibly starting with L. A hint as to its importance can be found in the fact that, aside from it being critical, it is also one of the few daemons that have made it into the root file system (it's in /sbin, and not /system/bin like most others).

lmkd (Android L)

Android L uses another specialized core service class daemon called `lmkd`. It is defined in the /init.rc as follows:

Listing 5-4: The `lmkd` definition in /init.rc

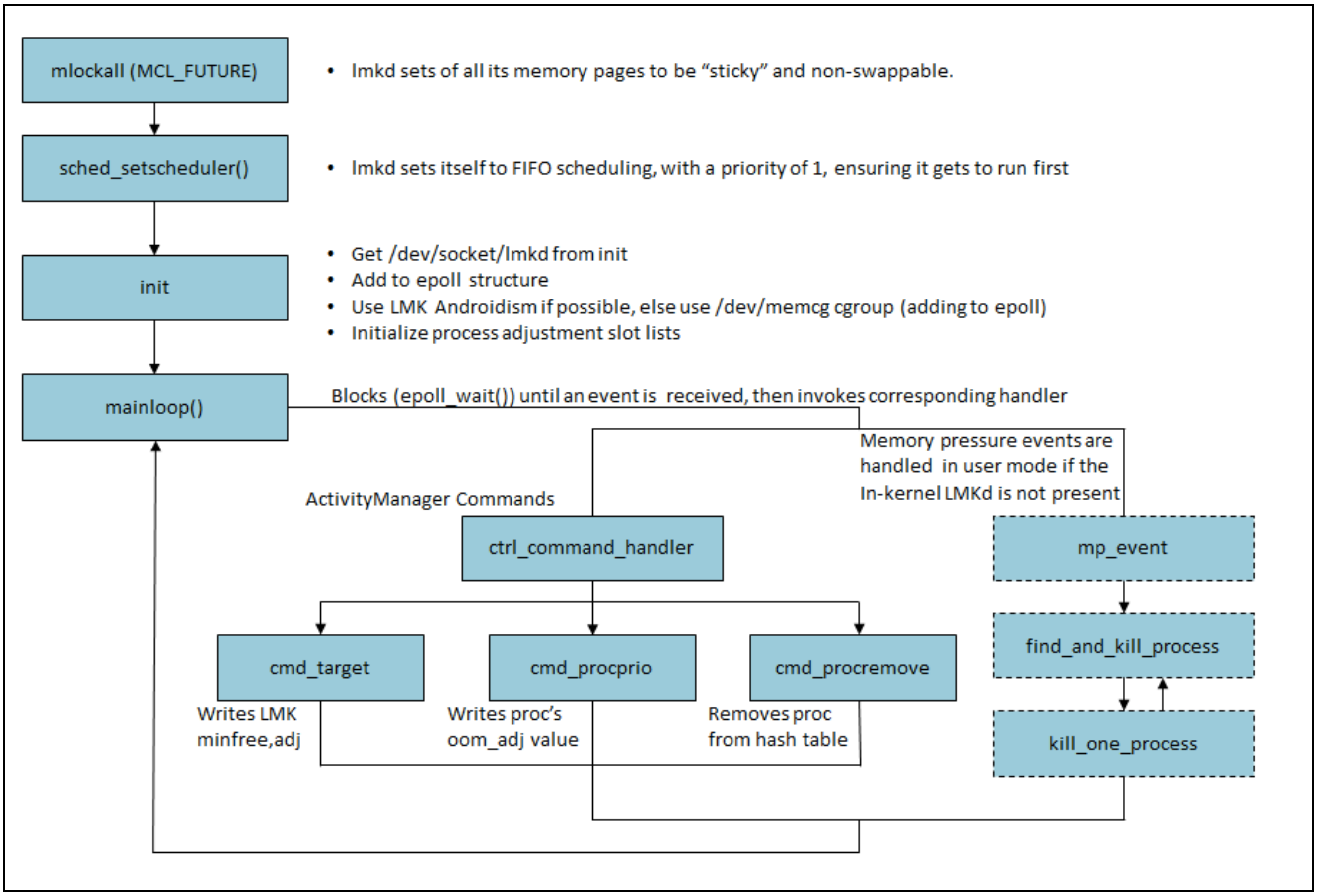
```
service lmkd /system/bin/lmkd
    class core
    critical
    socket lmkd seqpacket 0660 system system
```

The `lmkd` provides an interface to the kernel's **Low Memory Killer** (LMK) mechanism, which is an Androidism (i.e, a feature present in Android kernels, but not Linux ones). The LMK allows Android finer control over the Linux Out-Of-Memory (OOM) mechanism, which automatically kills tasks during memory pressure. Using the /proc/*pid*/oom_score_adj files, the `lmkd` can adjust the OOM score of processes, making them more or less "killable" - that is, prone to being killed when the system experiences memory pressure. The Linux OOM mechanism is discussed in detail in a later volume.

`lmkd` has two possible modes of operation - depending on whether or not the LMK Androidism is detected. If present, `lmkd` merely writes OOM score adjustment values to the target processes' /proc entries, leaving it to the LMK module to perform the actual killing on low memory pressure. In kernels without LMK, however, `lmkd` also takes it upon itself to respond to memory pressure events, and perform the actual killing (that is, sending `SIGKILL` to the process). `lmkd` maintains a process hash table to allow it to quickly look up processes and their memory scores.

As with all the other daemons discussed in this chapter, `lmkd` uses `epoll_wait` to simultaneously wait on input from multiple sockets. The main socket - /dev/socket/lmkd - is the one created for it by init, which is listening for connections. The only expected client is the `ActivityManagerService` (discussed in the next chapter), which uses this socket to notify the daemon which process needs to have its score adjusted (via the ProcessList class). When the in-kernel LMK is not available (i.e. its files in /sys/module/lowmemorykiller cannot be found), `lmkd` additionally listens on the memory cgroup files to pick up memory pressure events. This is shown in Figure 5-3:

Figure 5-3: The flow of `lmkd`


mlockall (MCL_FUTURE)
• lmkd sets of all its memory pages to be "sticky" and non-swappable.
sched_setscheduler()
• lmkd sets itself to FIFO scheduling, with a priority of 1, ensuring it gets to run first
init
• Get /dev/socket/lmkd from init
• Add to epoll structure
• Use LMK Androidism if possible, else use /dev/memcg cgroup (adding to epoll)
• Initialize process adjustment slot lists
mainloop()
Blocks (epoll_wait()) until an event is received, then invokes corresponding handler
Memory pressure events are handled in user mode if the In-kernel LMKd is not present
ActivityManager Commands
ctrl_command_handler
mp_event
cmd_target
cmd_procprio
cmd_procremove
find_and_kill_process
kill_one_process
Writes LMK minfree,adj
Writes proc's oom_adj value
Removes proc from hash table

When responding to memory pressure events, (that is, in cases where an in-kernel low memory killer cannot be used) `lmkd` parses the kernel's `/proc/zoneinfo` entry to extract the following values:

- **nr_free_pages:** Amount of free memory (in units of 4K)
- **nr_file_pages:** Amount of memory mapped by files (in units of 4K)
- **nr_shmem:** Amount of shared memory. These pages are used by multiple processes, and are therefore decremented from the file mapped page count.
- **nr_totalreserve_pages:** Amount of reserved system memory. These pages are free, but aren't really usable, so they are decremented from the free count.

The `lmkd` then proceeds to kill processes until meeting the adjusted free and file mapped targets.

Experiment: Observing `lmkd`

At the time of writing, the Android source code for L hasn't been made available (aside from a very limited preview, which hasn't proven helpful). The binaries, however, are available for both the Nexus 5 and the Android Emulator. It is therefore easy to reverse engineer them in a level of detail sufficient for this work. Both static analysis (i.e. disassembly) and dynamic analysis (runtime debugging) methods have been used. The method shown previously with `healthd` (using `strace`) proves its efficacy once again. Note that `lmkd` cannot be backported into KitKat, as it relies on the seqpacket sockets created for it by init to communicate with the frameworks.

Output 5-6: Using `strace` to figure out `lmkd`

```
root@LEmulator:/# ls -l /proc/$lmkd_pid/fd | cut -c1-10,55-
lrwx------ 0 -> /dev/null
lrwx------ 1 -> /dev/null
lrwx------ 10 -> socket:[7360]          # /dev/socket/lmkd (listening)
lrwx------ 2 -> /dev/null
lrwx------ 3 -> anon_inode:[eventpoll]
lrwx------ 4 -> socket:[7364]           # /dev/socket/logdw (to logd)
lrwx------ 5 -> socket:[7653]           # /dev/socket/lmkd (to ActivityManager)
lr-x------ 8 -> /dev/__properties__
root@LEmulator:/# strace -p $lmkd_pid
epoll_pwait(3, {{EPOLLIN, {u32=3069216345, u64=37428954713}}}, 2, -1, NULL, 8) = 1
read(5, "\0\0\0\1\0\0\4\5\0\0\0\v", 52) = 12
openat(AT_FDCWD, "/proc/1029/oom_score_adj", O_WRONLY) = 6
write(6, "647", 3)                      = 3
close(6)                                = 0
```

Looking at the above, you can see `lmkd`, like other daemon, blocks on the `epoll_wait` (FD 3), waiting for an event. The fd used for input - 5 - is the /dev/socket/lmkd, the other end of which is connected to the Android `ActivityManagerService`. Messages are variable length (up to 52 bytes), starting with a message type. Three message types have been observed:

Table 5-2: `lmkd` protocol messages

Constant	Type	Parameters
LMK_TARGET	0x00000000	Integer array of parameters which `lmkd` writes to /sys/module/lowmemorykiller/parameters/minfree
LMK_PRIO	0x00000001	PID to adjust (e.g. "\0\0\4\5" above for PID 1029), and oom_score_adj to set for it
LMK_PROCREMOVE	0x00000002	PID to remove from monitoring

logd (Android L)

Android L defines a new, much needed logging mechanism with its logd daemon. This daemon serves as a centralized user-mode logger, as opposed to the traditional Android's /dev/log/ files, implemented in kernel ring buffers. This not only addresses the main shortcomings of the ring buffers - their small size and resident memory requirements, but also allows logd to integrate with SELinux auditing, by registering itself as the auditd, which receives the SELinux messages from the kernel (via netlink), and records them in the system log.

Another important new feature provided by logd is **log pruning**, which allows the automatic clearing or retaining of log records from specific UID. This aims to solve the problem of logs being flooded with messages from overly-verbose processes, which make it harder to separate the wheat from the chaff. logd allows for white lists (UIDs or PIDs whose messages will be retained for longer) and ~blacklists (UIDs or PIDs whose messages will be quickly pruned), using the new -P switch of logcat.

The logd service is defined in /init.rc as follows:

Listing 5-5: The logd definition in /init.rc

```
service logd /system/bin/logd
    class core
    socket logd stream 0666 logd logd      # Used by CommandListener thread
    socket logdr seqpacket 0666 logd logd  # Used by LogReader thread
    socket logdw dgram 0222 logd logd      # Used by LogListener thread
    seclabel u:r:logd:s0
```

Note this service is designed with not one, but four sockets:

- **/dev/socket/logd:** The control interface socket.
- **/dev/socket/logdw:** A write-only socket (permissions 022 = -w--w--w-).
- **/dev/socket/logdr:** A read-write socket, designed for reading. Unlike the logd UN*X domain socket, this is a seqpacket (sequential packet) socket.
- **An unnamed NetLink socket:** Used when logd also provides auditd functionality for SELinux messages

The logd spawns listener threads over its sockets, as well as threads for clients (spawned on demand). The threads are individually named (using prctl(2)) so you can see them for yourself in logd's /proc/$pid/task/ when logd is running.

As with the traditional logs, logd recognizes the log buffers of main, radio, events, and system, along with a new log - crash - added in L. These logs are identified by their "log ids" (lids), numbered 0 through 5, respectively.

System properties used by logd

The logd recognizes several system properties, all in the logd namespace, which toggle its behavior. Those are well documented in the README.property file in logd's directory, shown here for convenience:

Listing 5-6: Properties used by `logd`

```
name                       type default  description
logd.auditd                 bool  true    Enable selinux audit daemon
logd.auditd.dmesg           bool  true    selinux audit messages duplicated and
                                         sent on to dmesg log
logd.statistics.dgram_qlen  bool  false   Record dgram_qlen statistics. This
                                         represents a performance impact and
                                         is used to determine the platform's
                                         minimum domain socket network FIFO
                                         size (see source for details) based
                                         on typical load (logcat -S to view)
persist.logd.size          number 256K   default size of the buffer for all
                                         log ids at initial startup, at runtime
                                         use: logcat -b all -G

# persist.logd.size.logname can be used to set buffer sizes for individual logs
```

Controlling `logd`

Clients can connect to /dev/socket/logd to control logd with an array of protocol commands. Commonly, the client doing so is the `logcat` command, which has been modified to use the socket, rather than the legacy `ioctl(2)` codes over /dev/log. The commands are shown in Table 5-3:

Table 5-3: `logd` protocol commands

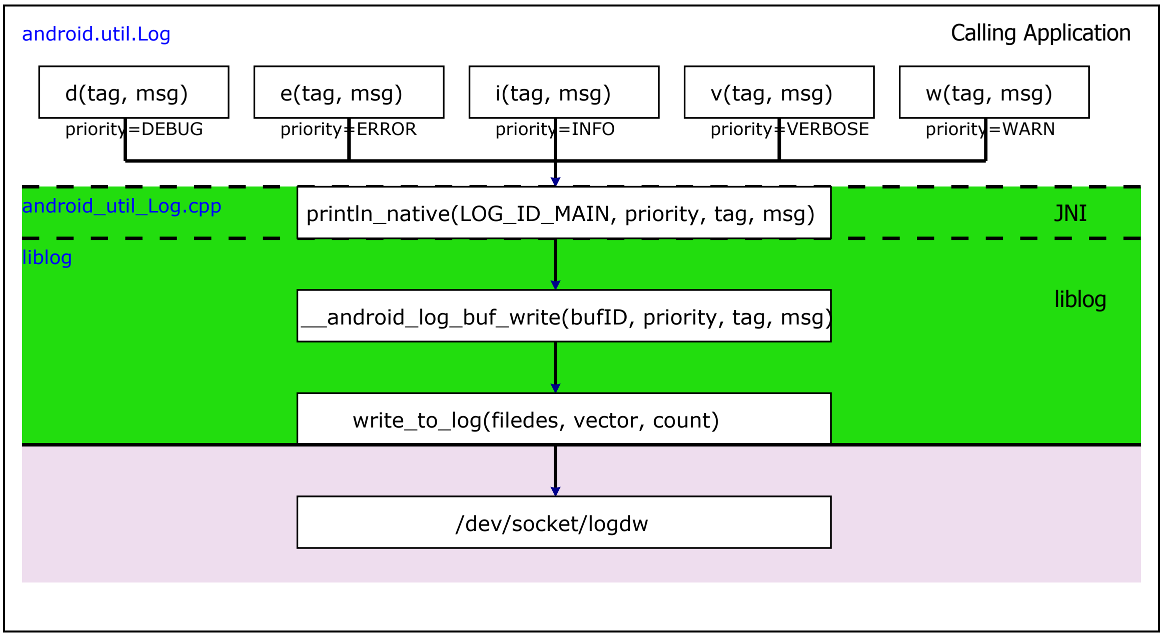
Command	logcat switch	Purpose
clear *lid*	`-c`	For callers with log credentials, this clears the specified log's buffers
getLogSize *lid*	`-g`	Get maximum size of log specified by *lid*
getLogSizeUsed *lid*		Get actual size of log specified by *lid*
setLogSize *lid*	`-G`	Set Maximum size of log specified by *lid*
getStatistics *lid*	`-S`	For callers with log credentials, this retrieves statistics - # of log messages by PID, etc.
getPruneList	`-p`	Get prune list (all logs)
setPruneList	`-P`	Set prune list (all logs)
shutdown		Force daemon exit. Surprisingly, this doesn't require any credentials.

The commands in gray require the caller to possess log credentials - be root, possess a primary GID of root, system, or log, or a secondary GID of log. To verify the last case the code of `logd` uses a crude method, of parsing the caller's /proc/*pid*/status and sifting through its "Groups:" line.

Writing to logd (logging)

Android's logging mechanism is supplied by `liblog`, and therefore applications remain oblivious to the underlying implementation of logging. As of L, both Bionic and `liblog` can be compiled to use `logd` (by `#define`ing `TARGET_USES_LOGD`), which then directs all the logging APIs to use `logd` rather than the traditional /dev/log device files, which have, in effect, become legacy (and apparently removed in M). Effectuating the change is a simple matter, since all system logging APIs eventually funnel to `liblog`'s `__android_log_buf_write` (or Bionic's `__libc_write_log`), which then open the `logdw` socket (instead of /dev/log), and write the log message to it. Figure 5-4 shows the flow of log messages from the application all the way to `logd`. A similar flow occurs for event log (android.util.EventLog) messages.

Figure 5-4: The Android logger architecture



Reading from logd (logcat)

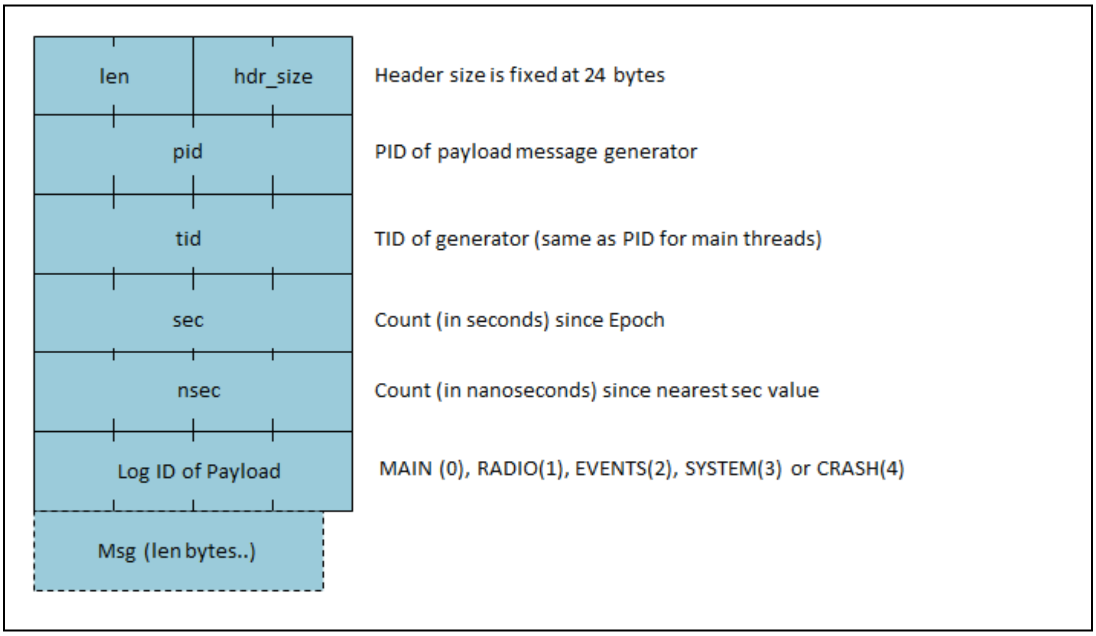
The familiar `logcat` command in L still sports the same command-line arguments it has in the past. Its underlying implementation, however, has rewritten to use `logd` through an updated `liblog` API. Clients such as logcat can connect to the logd reader socket (/dev/socket/logdr), and instruct the `LogReader` instance of `logd` to provide the log by writing parameters to it, as shown in the following table:

Table 5-4: Parameters recognized by `logd` over the reader socket

Parameter	Provides
lids=*value*	Log IDs
start=*value*	Start time from log to dump (default is EPOCH, start of log)
tail=*value*	Number of lines from log to dump (as per `tail(1)` command)
pid=*value*	Filter by PID originator of log messages
dumpAndClose	Tells reader thread to exit when log dumping is done

Log records are serialized into a `logger_entry_v3` structures before being passed to the reader over the socket. The structure format is shown in the following figure:

Figure 5-5: The format of a `logd` message


len
hdr_size
Header size is fixed at 24 bytes
pid
PID of payload message generator
tid
TID of generator (same as PID for main threads)
sec
Count (in seconds) since Epoch
nsec
Count (in nanoseconds) since nearest sec value
Log ID of Payload
MAIN (0), RADIO(1), EVENTS(2), SYSTEM(3) or CRASH(4)
Msg (len bytes..)

Putting all the above together, we can now observe `logd` in action, through the `logcat` command, as shown in the following experiment:

Experiment: Observing `logcat`

Using `strace` will allow you a behind-the-scenes look at the workings of `logcat` - including its connection to `logd`, the command it sends to dump the log, and the serialization of log messages:

Output 5-7: `logcat` under `strace`, annotated

```
# Tracing logcat during an adb logcat operation shows messages are received
# from file descriptor 3, and sent to file descriptor 1 (stdout)

root@generic:/# strace logcat
...
connect(3, {sa_family=AF_LOCAL, sun_path="/dev/socket/logdr"}, 20) = 0
write(3, "stream lids=0,3,4", 17)       = 17  # Dump main, system, crash
...
# \16 = 14 bytes (payload). \30 = 24 bytes (header). T\1 = 340 (PID) ... \3\0\0\0 = System
recvfrom(3, "<\16\30\0T\1\0\0m\1\0\0\275bbT$+\2374\3\0\0\0\6Act".., 5120, 0, NULL, 0) = 3668
write(1, "E/ActivityManager(  340): ANR in"..., 5472) = 5472
# In case you missed the connect(2) call above (e.g. if attaching to logcat), you can still
# look through its /proc/..fd entry, to see file descriptor 3 is a socket - which you can
# also deduce from the use of recvfrom(2):
root@generic:/# cd /proc/$LOGCAT_PID/fd
root@generic:/proc/337/fd # ls -l | grep "3 "
lrwx------ root     root              2014-11-11 14:24 3 -> socket:[2442]
# Looking through /proc/net/unix, which shows domain sockets, we can find the socket
and its remote endpoint (next inode number) - which happens to be logdr
root@generic:/proc/337/fd # grep 2442 /proc/net/unix
00000000: 00000003 00000000 00000000 0005 03  2442
root@generic:/proc/337/fd # grep 2443 /proc/net/unix
00000000: 00000003 00000000 00000000 0005 03  2443 /dev/socket/logdr
```

Sifting through `logd`'s thread to find and trace the `logd.reader.per` thread instance will show you the logging from the perspective of `logd`, and is left as an exercise for the reader.

vold

The Android vold is a volume-management daemon. This concept, which originally appeared in the (now deceased) Solaris operating system, employs a user-space daemon to automatically mount file systems ("volumes") as they are detected by the kernel. Beginning with HoneyComb, vold also enables file system encryption, in particular /data. Listing 5-7 shows its definition in /init.rc:

Listing 5-7: vold definitions in /init.rc (KitKat)

```
service vold /system/bin/vold
class core
socket vold stream 0660 root mount
ioprio be 2
```

vold is the only daemon to have an ioprio attribute, which specifies an I/O priority for the service.

vold and init share a common codebase in the form of fs_mgr, which is statically compiled into both binaries. The fs_mgr provides file system mounting and checking functionality, by wrapping together system calls (such as mount(2)) and hard-coded calls the external binaries (/system/bin/e2fsck) together.

Configuration

True to a mount daemon, vold requires a configuration file, to list known file systems and their mount points. This file is referred to as the **file system table**, or **fstab**, for short. Prior to 4.3, it was called /system/etc/vold.fstab, and mapped the file systems by their block device paths, in /sys. As of 4.3, however, the file has been moved to the rootfs, and has been made device specific by renaming to /fstab.${ro.hardware}, similar to the device specific .rc files. It has also been formatted along the lines of classic UN*X fstab files, like so:

Listing 5-8: Post 4.3 /fstab.${ro.hardware} syntax

```
# Android fstab file.
# The filesystem that contains the filesystem checker binary (typically /system) cannot
# specify MF_CHECK, and must come before any filesystems that do specify MF_CHECK

#<src>                      <mnt_point>   <type>   <mnt_flags and options>
/devices/msm_sdcc.2/mmc_host   auto         vfat      defaults     voldmanaged=ext_sd:auto,noemulatedsd
/devices/platform/xhci-hcd     auto         vfat      defaults     voldmanaged=usb:auto
```

You may remember we encountered the /fstab.${ro.hardware} file in the discussion of how /init mounts file systems. vold evaluates the file in a similar manner to /init (using the shared fs_mgr code), but whereas /init ignores lines with the voldmanaged, vold concerns itself with these lines only. The mnt_flags field, though incorrectly specified in the documentation as being ignored, is passed verbatim to the mount(2) system call. The options are parsed by fs_mgr:

Table 5-5: `fs_mgr` options

Option	Purpose
wait	Wait for file system to mount for up to 20 seconds before continuing
check	Perform a file system check on the file system prior to mounting
nonremovable	Volume is not a removable volume (i.e. not an SD-Card)
recoveryonly	File system only mounted during recovery
noemulatedsd	Tells vold this is not an emulated SD card. If vfat-formatted, ASEC can be used
verify	As of KitKat: Enable the Linux kernel's dm_verity to cryptographically verify the filesystem integrity (described in Chapter 8)
zramsize=	Compressed RAM (ZRAM) size
swapprio=	Specifies priority of partition as swap partition
length=	Denotes the size of the partition
voldmanaged=	Partition is managed by vold. Expects *tag*:*number* with partition number or "auto"
encryptable=	Specifies the location of the keys for an encrypted partition
forceencrypt	L: encrypt on first boot

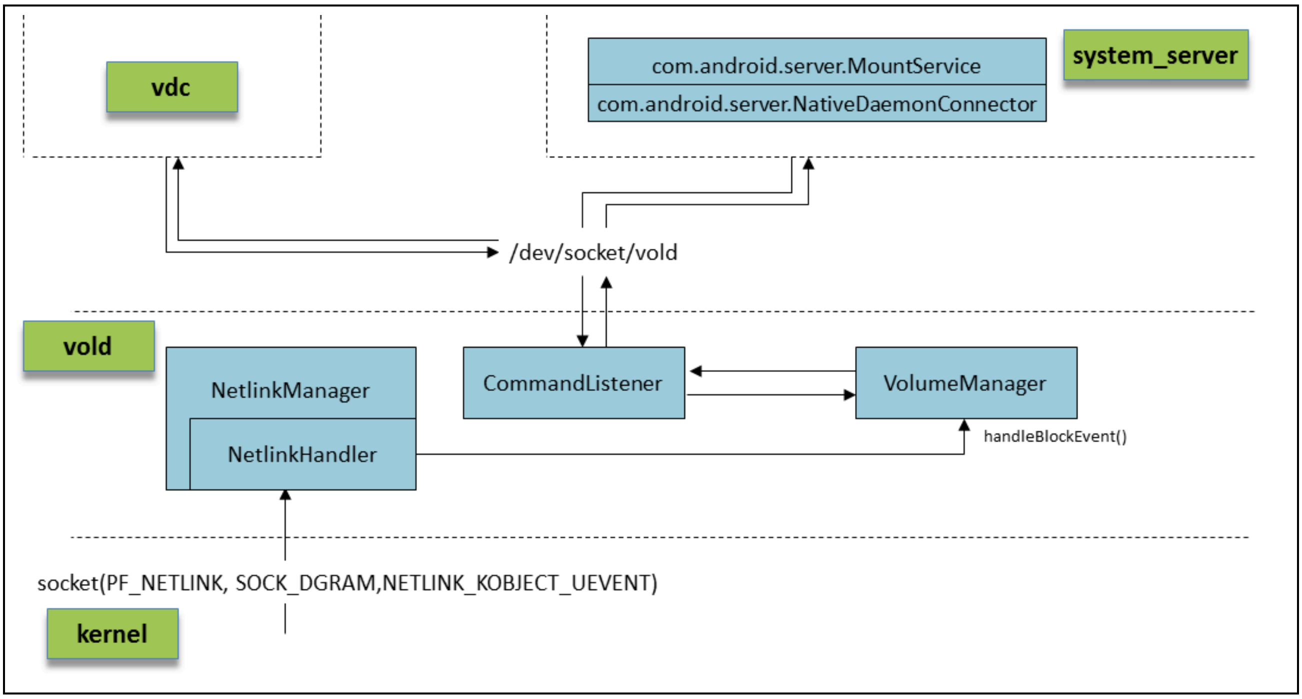
Architecture

The `vold` internally comprises three components:

- `VolumeManager`: Responsible for maintaining volume state, and handling various volume operations. This (singleton) class provides all the framework-facing functionality.
- `NetlinkManager`: Responsible for listening on kernel NetLink events of the `block`subsystems using the `NetlinkHandler`, which passes them to the volumeManager.
- `CommandListener`: Responsible for listening on the /dev/socket/vold socket, for commands issued by the framework, and relaying the output of those commands, or other events received from `VolumeManager`.

Figure 5-6 presents the structure of `vold`:

Figure 5-6: The internal architecture of `vold`


vdc
com.android.server.MountService
com.android.server.NativeDaemonConnector
system_server
/dev/socket/vold
vold
NetlinkManager
NetlinkHandler
CommandListener
VolumeManager
handleBlockEvent()
socket(PF_NETLINK, SOCK_DGRAM,NETLINK_KOBJECT_UEVENT)
kernel

The main client of `vold` is `com.android.server.MountService`, though applications cannot call this service directly, and must instead use `android.os.storage.StorageManager`. The `MountService` maintains a `NativeDaemonConnector` which uses the client side of the socket to send commands to `vold`'s `CommandListener`. Most Android devices have a vdc utility, which you can use to send these commands to vold yourself (as root), or listen to file system mounting events (using `vdc monitor`), as shown in Output 5-8:

Output 5-8: The `vdc monitor` output generated by SD-Card events

```
root@htc_m8wl:/ # vdc monitor
[Connected to Vold]
# SD-Card inserted
605 Volume ext_sd /storage/ext_sd state changed from 0 (No-Media) to 2 (Pending)
605 Volume ext_sd /storage/ext_sd state changed from 2 (Pending) to 1 (Idle-Unmounted)
630 Volume ext_sd /storage/ext_sd disk inserted (179:128)
630 Volume ext_sd /storage/ext_sd disk inserted (179:128)
605 Volume ext_sd /storage/ext_sd state changed from 1 (Idle-Unmounted) to 3 (Checking)
613 ext_sd /storage/ext_sd "8A07-A343"
614 ext_sd /storage/ext_sd
605 Volume ext_sd /storage/ext_sd state changed from 3 (Checking) to 4 (Mounted)
# SD-Card removed
632 Volume ext_sd /storage/ext_sd bad removal (179:129)
605 Volume ext_sd /storage/ext_sd state changed from 4 (Mounted) to 5 (Unmounting)
613 ext_sd /storage/ext_sd
614 ext_sd /storage/ext_sd
605 Volume ext_sd /storage/ext_sd state changed from 5 (Unmounting) to 1 (Idle-Unmounted)
631 Volume ext_sd /storage/ext_sd disk removed (179:128)
605 Volume ext_sd /storage/ext_sd state changed from 1 (Idle-Unmounted) to 0 (No-Media)
```

The `vdc` utility is nothing more than a tiny UNIX domain socket client, whose source can be found in system/vold/vdc.c. The commands, which it relays verbatim to `vold`, are shown in table 5-9:

Table 5-6: `vold` commands

Cmd	Subcmd	Arguments	Purpose
dump			Dumps loop, device mapper, and mounted filesystems
volume	list		List mounted volumes
	debug	on|off	Toggle debug messages for formatting/unmounting
	mount	*path*	Mount a file system
	unmount	*path*[force[_and_revert]]	Unmount a file system, possibly forcefully
	[un]share	ums	Share/unshare USB Mass Storage
	shared	ums	Return share state (enabled/disabled) of USB Mass Storage
	mkdirs	*path*	Make a directory/mount point
	format	[wipe] *path*	Format a FAT volume, optionally erasing its contents first
storage	users		List PIDs using a mounted volume (like `fuser`)
	mountall		Call on fs_mgr to mount all filesystems in fstab

Table 5-7: `vold` commands (cont.)

Cmd	Subcmd	Arguments	Purpose
asec	list		List all Android Secure Storage containers
	create	*cid mb fstype key uid*	Create new asec with *cid*, as a filesystem *fstype* of size *mb*
	destroy	*cid* [force]	Destroy the asec identified by *cid*, possibly forcefully
	finalize	*cid*	Finalize container *cid*.
	fixperms	*cid gid filename*	Fix permissions in container *cid* so as to be owned by *gid*.
	mount	*cid key uid*	Mount the container *cid* under app-id *uid*, with *key*.
	unmount	*cid* [force]	Unmount the container *cid*, possibly forcefully if in use.
	path	*cid*	Return the path to the container *cid*.
	rename	*old_cid new_cid*	Change the name of *old_cid* to *new_cid*
	fspath	*cid*	Return file system path corresponding to *cid*
obb	list		List all mounted opaque binary blobs
	mount	*filename key ownerGid*	mount the opaque binary blob specified by *filename* for app *ownerGid*, with optional *key*
	unmount	*source* [force]	unmount the opaque binary blob specified by *source* filename
	path	*source*	
cryptfs	restart		Signal `init` to restart frameworks
	cryptocomplete		Query if filesystem is fully encrypted
	enablecrypto	inplace|wipe *password*	Encrypt filesystem, possibly erasing first
	changepw	*default|password|pin|pattern new_passwd*	Change encryption password
	checkpw	*passwd*	Check if supplied password can mount encrypted fs
	verifypw	*passwd*	Used by `BackupManagerService`
	getfield	*name*	Get metadata field from cryptfs
	setfield	*name value*	Set metadata field in cryptfs
fstrim	do[d]trim		Issues an `FI[D]TRIM ioctl(2)`, allowing mmc driver to wipe unused blocks

Android L removes support for xwarp commands, a relic of older versions of Android, which used YAFFS. With the move to Ext4, this has been deprecated, and the commands, while they still exist through KK, fail on missing file requirements.

Of particular interest is `vold`'s filesystem encryption handling. The Android Documentation[1] provides a detailed explanation of the process as implemented in Honeycomb, as does this book, next.

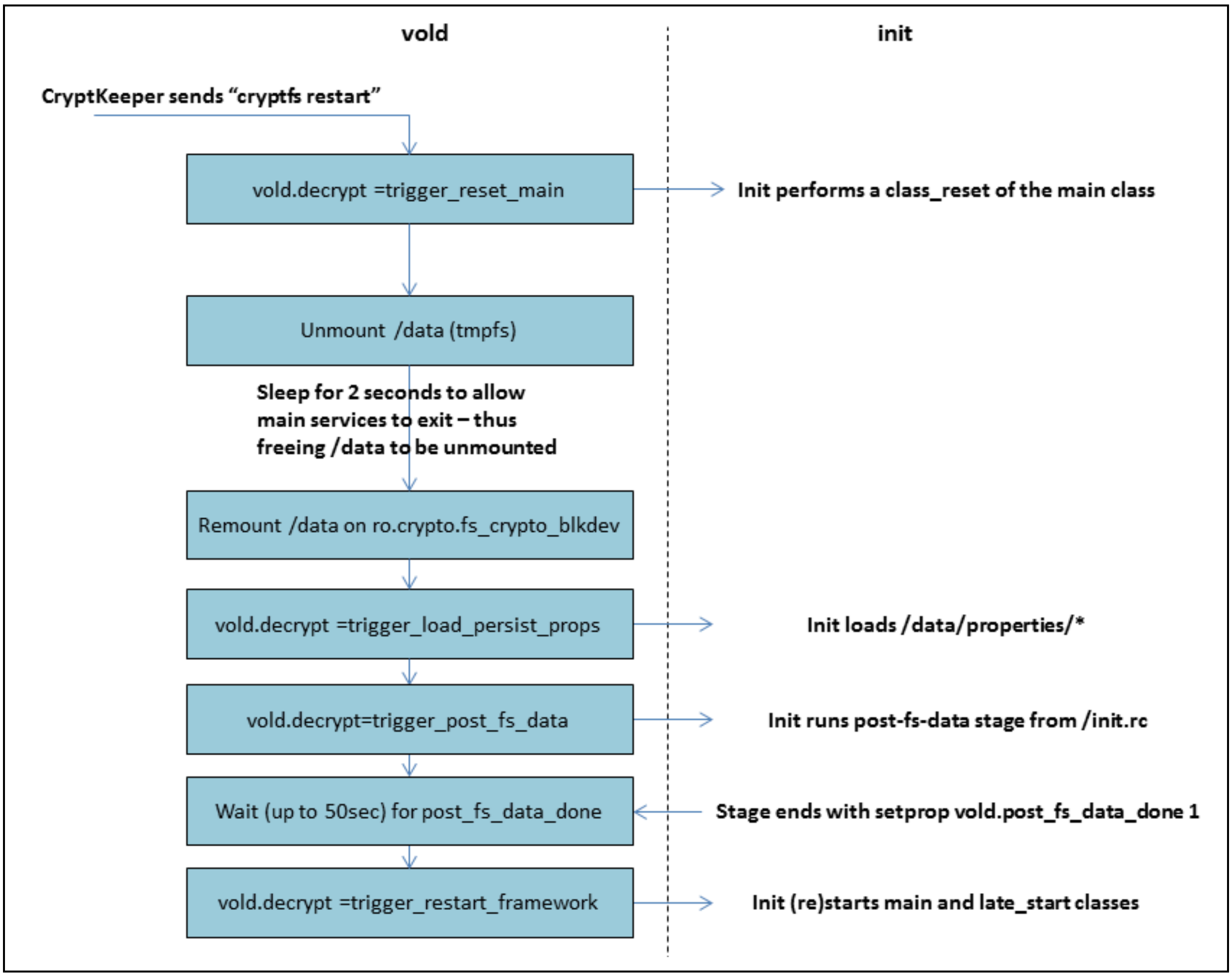
Decrypting filesystems

With Honeycomb, Android brings support for disk encryption. By extending Linux's dm-crypt mechanism, which already provides the foundation for the asec mechanism, Android enables the entire user data partition to be encrypted. The system partition still remains very much cleartext, because the system has to somehow boot, but this is quite fine - The system partition is, for all intents and purposes, identical on all devices, and never actually holds any user-specific data, so there would be little advantage in encrypting it.

The dm-crypt feature is described in more detail in Chapter 8. At a high level view, however, suffice it to say that dm-crypt transparently encrypts and decrypts block devices. The password required for doing so, however, needs to be supplied in user mode. Android derives the passcode or pattern the user is already using for the lock screen*, and uses the com.android.settings.CryptKeeper activity to prompt the user for the credentials required to unlock the device, without which /data cannot be mounted.

What follows, therefore, is a choreography between init (driving the system startup), vold (providing the actual mount services), and CryptKeeper (handling the UI displayed to the user). This is shown in Figure 5-7:

Figure 5-7: The interaction between vold and init


vold
init
CryptKeeper sends “cryptfs restart”
vold.decrypt =trigger_reset_main
Init performs a class_reset of the main class
Unmount /data (tmpfs)
Sleep for 2 seconds to allow main services to exit – thus freeing /data to be unmounted
Remount /data on ro.crypto.fs_crypto_blkdev
vold.decrypt =trigger_load_persist_props
Init loads /data/properties/*
vold.decrypt=trigger_post_fs_data
Init runs post-fs-data stage from /init.rc
Wait (up to 50sec) for post_fs_data_done
Stage ends with setprop vold.post_fs_data_done 1
vold.decrypt =trigger_restart_framework
Init (re)starts main and late_start classes

During boot, init calls on the fs_mgr to mount the file systems. If none are encrypted, it sets the ro.crypto.state to "unencrypted" and enqueues any actions associated with the "nonencrypted" trigger - usually those services in the late_start class.

** - The Android Explorations blog[2] shows how to decouple the encryption password from the pattern, by using vdc cryptfs changepw.

If a file system is encrypted, however, an encrypted mount will obviously fail, unless the password is supplied. The fs_mgr mounts the filesystem instead as a tmpfs, and returns 1 to init, which sets ro.crypto.state to "encrypted", and informs vold of the need to decrypt /data (by setting vold.decrypt to 1). Prior to kitkat, vold would use the value of the ro.crypto.tmpfs_options property as mount options, but these options are now hardcoded. The mounting of /data is a prerequisite for loading the UI frameworks, since those need to write various files. As a temporary filesystem, however, the /data mount holds no data. When the vold.decrypt is set, the SystemServer only runs the "core" apps and services.

The com.android.settings.CryptKeeper activity registers itself as the home screen (using an IntentFilter), with a higher priority, so as to ensure it starts first. When it loads, it checks the value of vold.decrypt in its onCreate(). If unset, it simply exits, making room for the "real" home screen. If the filesystem is encrypted, however, the CryptKeeper start an async ValidationTask in its onStart() to contact the com.android.server.MountService, calling its getEncryptionState() to see if the partition is indeed properly encrypted.

Recall, that the MountService is connected to the vold socket. It can thus send the daemon the cryptfs cryptocomplete command. This makes vold check if the encryption is indeed recoverable (as it may be that the encryption has been interrupted, rendering /data unmountable, and forcing the user to do a recovery/reset). If the cryptocomplete operation is successful, the CryptKeeper calls setupUi to input the user for the decryption password or sequence. It passes this again to MountService, which sends it to vold as a cryptfs checkpw command.

If the password is correct, the MountService sends the cryptfs restart command. This makes vold update the vold.decrypt property to trigger_reset_main, and sleep for 2 seconds in the hopes that all the services loaded under the main class will be stopped, and the tmpfs /data can be unmounted. If the unmount is successful, vold remounts the (now unecrypted) /data partition, and again updates the vold.decrypt property - first to load_persist_props (since those reside in /data), next to trigger_post_fs_data (to get init to set up any paths in /data defined in /init.rc) and then to trigger_restart_framework, which makes init restart the frameworks. The properties must be defined in /init.rc to arm the approriate triggers, as shown in listing 5-11:

Listing 5-9: Actions in init.rc relating to encryption events

```
on nonencrypted
    class_start late_start

on charger
    class_start charger

on property:vold.decrypt=trigger_reset_main
    class_reset main

on property:vold.decrypt=trigger_load_persist_props
    load_persist_props


on property:vold.decrypt=trigger_post_fs_data
# This will call on post-fs-data, which must end with a post-fs-data-done
    trigger post-fs-data

on property:vold.decrypt=trigger_restart_min_framework
    class_start main

on property:vold.decrypt=trigger_restart_framework
    class_start main
    class_start late_start

on property:vold.decrypt=trigger_shutdown_framework
    class_reset late_start
    class_reset main
```

Encrypting filesystems

Encrypting file systems is handled in a similar manner to decrypting them. Once again, the UI is supplied by CryptKeeper, with the MountService providing the Dalvik level bridge to vold. The UI prompts the user for the encryption password, and verifies the device is on AC power, to prevent any power outage which may disrupt the encryption. The MountManager's encryptStorage method is called, which sends vold the cryptfs enablecrypto, with either wipe (to format /data before encrypting it) or inplace argument, and the password.

Upon getting the command and verifying it can proceed, vold sets the vold.decrypt to trigger_shutdown_framework. This causes init to stop all services but the core ones. This is exactly the mirror image of the state the system is in while booting, before the user password is entered, and /data can be safely unmounted. vold then sets vold.encrypt_progress to start at 0, and vold.decrypt to trigger_restart_min_framework, to get init to restart the main services.

Once again, CryptKeeper loads as the home app. Upon seeing vold.encrypt_progress, it loads the status bar UI.. If all goes well, the progress bar reaches 100%. If not, vold.encrypt_progress is set to an error_ string. L offers resumable encryption, but if resumption fails the user may be left with no choice but to reset the device to defaults.

The technical aspects of encryption, as well as the kernel perspective, are discussed in Chapter 8.

Network Services

netd

Android uses a dedicated daemon to control network interfaces and configuration management. If you've ever used tethering, firewalling or WiFi Access-Point features, or even a basic DNS lookup - consider yourself a proud client of `netd`. The daemon is defined in /init.rc:

Listing 5-10: The `netd` service defition in init.rc

```
service netd /system/bin/netd
  class main
  socket netd stream 0660 root system      // Main interface for frameworks, and ndc
  socket dnsproxyd stream 0660 root inet   // Interface for Bionic, DNS resolution
  socket mdns stream 0660 root system      // Interface for NsdService, Neighbor Discovery
  socket fwmarkd stream 0660 root inet     // L: Firewall Marking interface
```

The `netd` shares many structural similarities with `vold`, and in fact shares some code with it, using the native `FrameworkListener` class (among others) in its socket handlers. Figure 5-8 shows the architecture (which you can compare and contrast with that of `vold`, q.v. Figure 5-6). Unlike `vold`, each of `netd`'s subcomponents uses a dedicated socket. `netd`'s structure comprises four components, discussed next.

CommandListener

The `CommandListener`: Responsible for listening on the /dev/socket/netd socket, for commands issued by the framework (specifically, `NetworkManagementService`, described in detail in the next chapter), and sending notifications (broadcasts) to connected clients. As with `vold`, the emulator includes a simple utility - `ndc` - which can be used as a client to issue commands to `netd`, and listen on events (using `ndc monitor`), as shown in output 5-9:

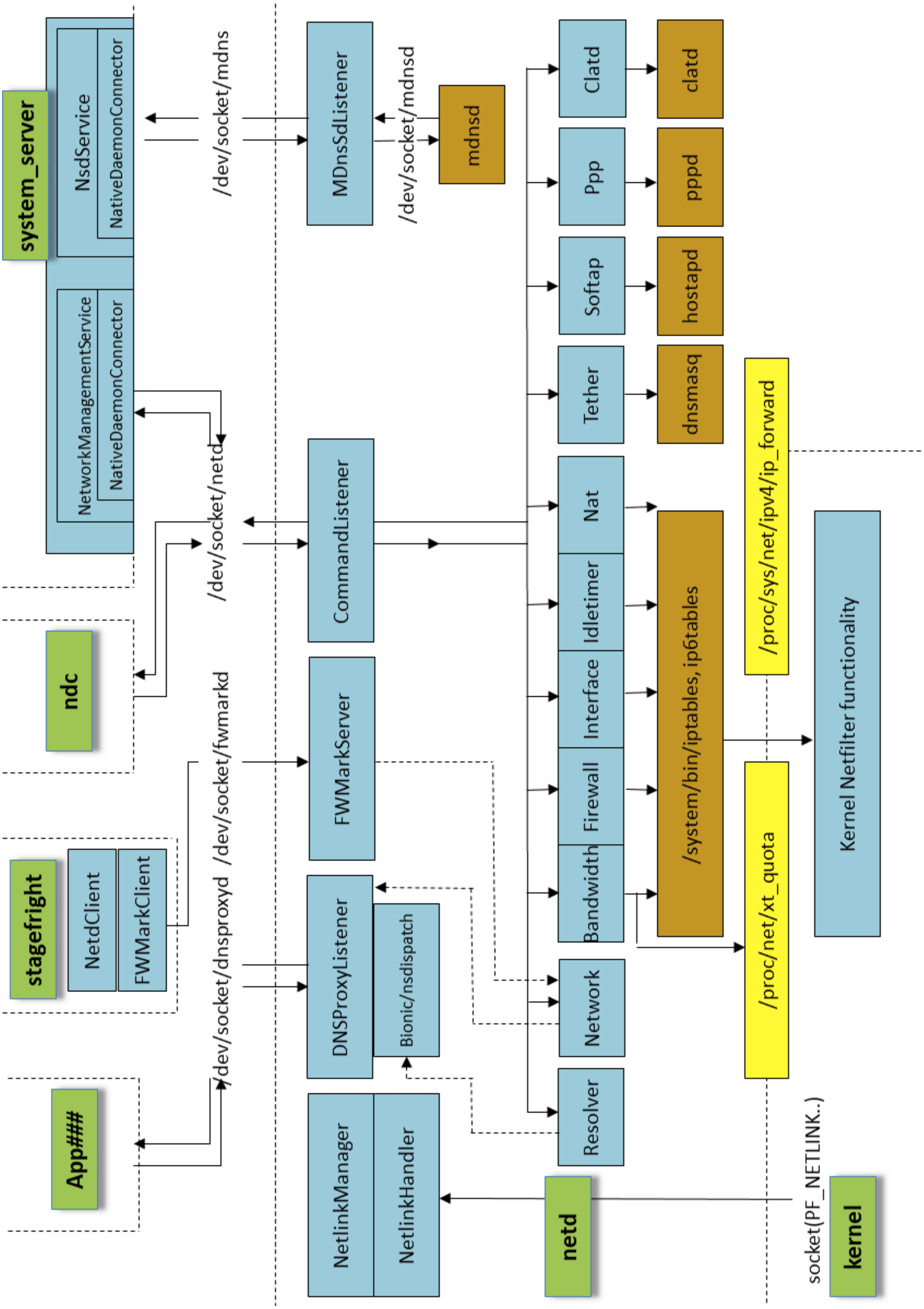
Output 5-9: The `ndc monitor` output generated by connecting to a Wi-Fi network

```
root@htc_m8wl:/ # ndc monitor
[Connected to Netd]
600 Iface linkstate wlan0 up
614 Address updated 10.100.1.192/21 wlan0 128 0
614 Address updated fe80::522e:5cff:fef3:9da6/64 wlan0 128 253
```

Internally, the `CommandListener` dispatches the commands to one of several internal **Controller** classes, each responsible for a specific aspect of functionality, shown in Table 5-8:

Table 5-8: `netd` controllers and their subcommands

Controller	Commands	Provided by	Used for
BandwidthController	`bandwidth`	/system/bin/ip[6]tables	Network quota control
ClatdController	`clatd`	/system/bin/clatd	464XLAT (IPv4 over IPv6) control
FirewallController	`firewall`	/system/bin/iptables	Firewalling
IdletimerController	`idletimer`	/system/bin/ip, ip[6]tables	Idle timer on interfaces
InterfaceController	`interface`	/proc/sys/net/*	Network interfaces
NatController	`nat`	/system/bin/ip, ip[6]tables	Network Address Translation
PppController	`ppp`	/system/bin/pppd	VPNs
SoftapController	`softap`	/system/bin/hostapd	Wi-Fi tethering/P2P
TetherController	`tether` `ipfwd`	/system/bin/dnsmasq /proc/sys/net/ipv4/ip_forward	USB and Wi-Fi tethering


system_server
NsdService
NativeDaemonConnector
NetworkManagementService
NativeDaemonConnector
ndc
stagefright
NetdClient
FWMarkClient
App###
/dev/socket/mdns
/dev/socket/netd
/dev/socket/dnsproxyd
/dev/socket/fwmarkd
MDnsSdListener
/dev/socket/mdnsd
mdnsd
CommandListener
FWMarkServer
DNSProxyListener
Bionic/nsdispatch
NetlinkManager
NetlinkHandler
Clatd
Ppp
Softap
Tether
Nat
Idletimer
Interface
Firewall
Bandwidth
Network
Resolver
clatd
pppd
hostapd
dnsmasq
/system/bin/iptables, ip6tables
/proc/sys/net/ipv4/ip_forward
/proc/net/xt_quota
Kernel Netfilter functionality
netd
socket(PF_NETLINK..)
kernel

From the figure, you can see that the controllers (for the most part) don't actually provide any functionality. Rather, controllers hide external commands, by calling `fork()/exec(2)` to spawn the respective daemons or `ip[6]tables`, in effect taking a shortcut. A full discussion of controller internals, along with the daemon they spawn and the framework interface is left for Volume II. Table 5-9 provides a quick reference to the command set expected by the various controllers:

Table 5-9: `ndc` commands understood by the various `netd` controllers

Controller	Subcommand	Purpose
Bandwidth	**enable**	Enable bandwidth quota control
	setiquota *iface qBytes*	Set quota on *iface* not to exceed *qBytes*
	removeiquota *iface*	Remove any quota previously set on *iface*
	setifacealert *iface qBytes*	Generate alert if bandwidth on *iface* exceeds *qBytes*
	removeifacealert *iface*	Remove an alert previously set by `setifacealert`
	setglobalalert *alertBytes*	Generate alert if any interface exceeds *alertBytes*
	gettetherstats	Get statistics for device tethering
	setsharedalert|ssa *bytes* **removesharedalert|rsa**	Set an alert on bandwidth *bytes* spanning all interfaces
	addniceapps|aha *uid* **removeniceapps|rha** *uid*	Add "Nice" (allowed) apps by UID
	addnnaughtyapps|ana *uid* **removenaughtyapps|rna** *uid*	Add "Naughty" (misbehaving) apps by UID
	happybox *enable/disable*	
Firewall	**enable/disable**	Globally toggle the firewall functionality
	set_interface_rule *iface rule*	Apply an iptables *rule* on an *interface*
	set_egress_source_rule *addr rule*	Set rule for outgoing traffic, by source
	set_egress_dest_rule *addr port rule*	Set rule for outgoing traffic, by destination
	set_uid_rule *uid rule*	Apply an iptables *rule* for a specific *uid*
IdleTimer	**list**	List all interfaces
	[**en/dis**]**able** *iface*	Enable Idletimer mechanism: starts and flushes iptables chains
	[**add/remove**] *iface timeout classLabel*	Add or remove a timer on *iface*
Interface	list	List all interfaces
	route *iface default/secondary dest prefix gateway*	Add a routing table entry
	setmtu *iface mtu*	Set Maximum Transferrable Unit size on *iface* to *mtu*
	ipv6 *iface enable|disable*	Toggle IPv6 support on *iface*
	clearaddrs *iface*	Remove IP addresses of *iface*
	getcfg *iface*	Display configuration of *iface*
	setcfg *iface (ifconfig args)*	Set configuration of *iface*
	fwmark..	Firewall marking (L: moved to fwmarkd)

Table 5-10: `ndc` commands understood by the various `netd` controllers

Controller	Subcommand	Purpose
nat	[**enable**|**disable**] *int_iface ext_iface*	Toggle Network Address Translation
PPP	**attach** *tty local remote [dns1] [dns2]*	Attach PPPd to *tty* to set up a point-to-point connection between *local* and *remote* IP addresses, specifying optional name server IPs.
	detach *tty*	Execs `/system/bin/pppd -detach`
	list_ttys	List ttys used by the PPP daemon as interfaces
tether	**start**	Starts the DNS Masquerading
	stop	Stop Tethering: Kills `dnsmasq` using a `SIGTERM` signal
	dns *[set|list]*	Set or list DNS settings
	interface *[add|remove] iface*	Add/remove tethering on *iface*
	ipfwd *enable|disable|status*	Toggle IP Forwarding (/proc/sys/net/ipv4/ip_forward) or query status
Resolver	**setdefaultif** *iface*	Assign *iface* to be default for DNS lookups
	flushdefaultif	Flush default interface's DNS cache
	flushif *iface*	Flush *iface* DNS cache
	setifaceforpid *iface pid*	Assign *iface* for process ID *pid* to use
	clearifaceforpid *pid*	Remove *iface* assignment for Process ID *pid*
	setifaceforuidrange *iface low high*	Assign *iface* for AIDs *low-high*
Softap	**startap**|**stopap**|**status**	Toggle Access point (by exec()ing or kill()ing `/system/bin/hostapd`), or query status
	fwreload *iface AP|P2P|STA*	Reload firmware
	set *iface SSID hidden/* channel security key*	Set access point parameters. If any other word but "hidden" is specified, AP will be broadcast

DnsProxyListener

`DnsProxyListener` is the `FrameworkListener` responsible for listening on the /dev/socket/dnsproxyd socket for name resolution commands. The commands are shown in the following table:

Table 5-11: `Netd`'s DNS Proxying command subset

Cmd	Arguments	Purpose
getaddrinfo	name service ai_flags ai_family ai_socktype ai_protocol iface	Call `getaddrinfo(3)` for the interface *iface*. GAI is a more advanced and forward compatible alternative to the other get*XXX*by*YYY* functions.
gethostbyname	iface name af	Perform a forward lookup (A/AAAA, according to *af*) of an IP address by its hostname
gethostbyaddr	addrStr addrLen addrFamily iface	Perform a reverse lookup (PTR) of a hostname by its IP address

Android's LibC implementation - Bionic - provides all processes with library calls matching those in 5-11, and implements them by opening the UNIX domain socket connection to /dev/socket/dnsproxy. In this way, all clients - both native and Dalvik - are redirected through the DNS proxy functionality. `netd` can then enforce restrictions on DNS functionality based the calling process uid. Since each uid represents a different application, this translates to fine grained control of DNS functionality on a per-app basis.

mdnsd

Multicast DNS (mDNS) is a popular discovery protocol first adopted by Apple (as its "Bonjour" service). Standardized in RFC6762, it is used extensively in iOS's family of "Air" protocols (e.g. "AirPlay"), and allows devices to find one another by sending a multicast request to group 224.0.0.253 (or IPv6's FF02::fc) and UDP port 5353. Unsurprisingly, Android chose to adopt this standard as well, and it serves as the basis for "WiFi Direct". Beginning with JellyBean, /init.rc defines the mDNS service as follows:

Listing 5-11: The mdns service defition in init.rc

```
service mdnsd /system/bin/mdnsd
    class main
    user mdnsr
    group inet net_raw
    socket mdnsd stream 0660 mdnsr inet
    disabled
    oneshot
```

Because of its multicast capabilities, the service is granted membership in both the inet group (allowing general TCP/IP capabilities) and the net_raw group (allowing "advanced" capabilities, such as raw sockets and crafting non-standard IP packets). The service listens on /dev/socket/mdnsd (MDNS_UDS_SERVERPATH) for requests, and has another socket (/dev/socket/mdns) connected to netd.

The frameworks wrap the mDNS functionality with the Network Service Discovery classes (android.net.nsd, as of API level 16). This is discussed in greater detail in Volume II, in a chapter dealing with connectivity. The mDNS implementation itself (found in the external/mdnsresponder directory) is largely the same as the open source mDNS project, with the Android specific modifications (such as the UNIX domain socket and Android logging) clearly marked by #ifdef __ANDROID__ blocks.

mtpd

Though the acronym MTP is normally associated in Android (and elsewhere) with the Media Transfer Protocol, the mtpd couldn't be further from it - It is the daemon responsible for PPP and L2TP (but not IPSec). It is defined in /init.rc as follows:

Listing 5-12: The mtpd service defition in init.rc

```
service mtpd /system/bin/mtpd
    class main
    socket mtpd stream 600 system system
    user vpn
    group vpn net_admin inet net_raw
    disabled
    oneshot
```

The group permissions granted to mtpd reflect its need for network access (inet) setting up a network interface (net_admin), and tunneling IP (net_raw). As a oneshot and disabled service, the mtpd must be started manually, by setting the ctl.start property. Indeed, the com.android.server.connectivity.Vpn does so (in its startLegacyVPN inner class). There is no programming interface for VPN functionality, which is meant to be started or stopped from the Android system GUI.

racoon

Racoon is a de facto standard VPN daemon. As an external project, it is not part of Android per se, but is used extensively (in both Android and iOS) to provide VPN services (VPN connectivity is discussed in Volume II).

Listing 5-13: The `mtpd` service defition in init.rc

```
service racoon /system/bin/racoon
    class main
    socket racoon stream 600 system system
    # IKE uses UDP port 500. Racoon will setuid to vpn after binding the port.
    group vpn net_admin inet
    disabled                 # Started manually by ConnectivityManager
    oneshot
```

Note that `racoon` starts as root (in order to bind the ISAKMP well known port, which is a privileged port), but then drops privileges. This is why it requires the extra group memberships, which (as we discuss in Chapter 8) allow network connectivity. From a strict security standpoint, it would have been better to relinquish root altogether, and use capabilities (in particular `CAP_NET_BIND_SERVICE`) instead. This is especially important considering racoon has had a history of exploits (and was in fact used to jailbreak iOS 5) before.

rild

If your Android device is a phone or 3G/LTE connected tablet, `rild` is undoubtedly one of the more important system processes. The **R**adio **I**nterface **L**ayer **D**aemon provides virtually all the telephony capabilities for these devices, by interfacing with the baseband. It is defined in /init.rc as follows:

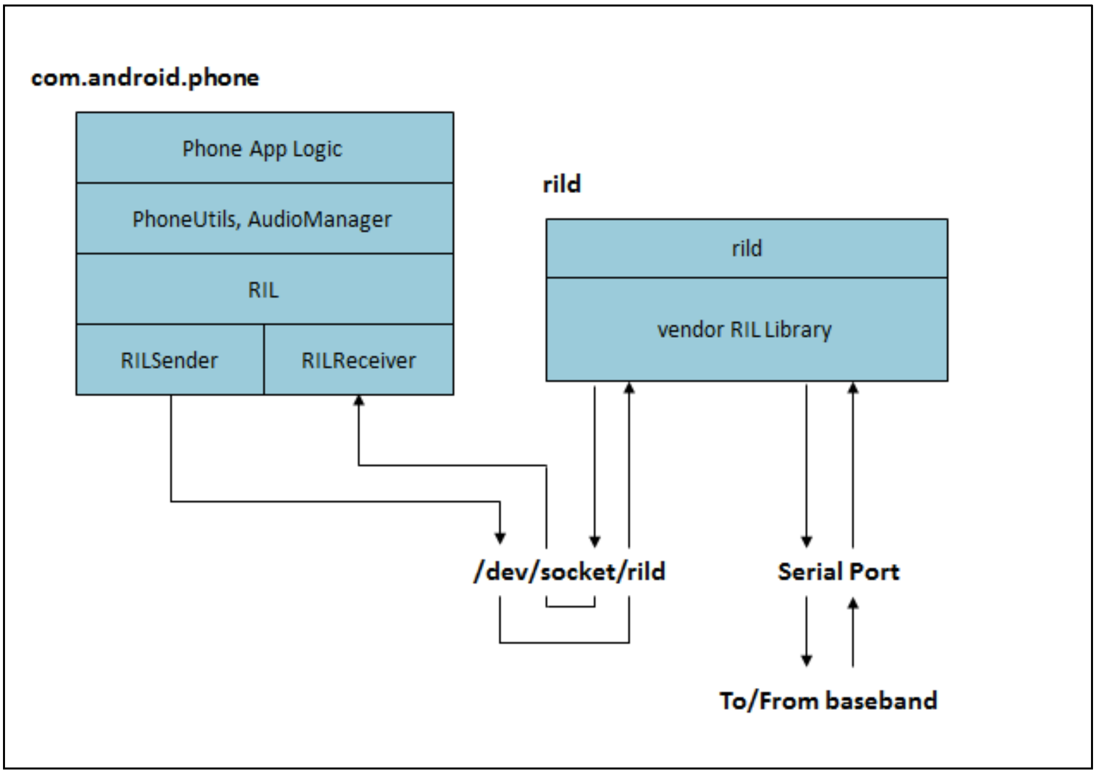
Listing 5-14: The `rild` service defition in init.rc

```
service ril-daemon /system/bin/rild
    class main
    socket rild stream 660 root radio
    socket rild-debug stream 660 radio system
    user root
    group radio cache inet misc audio log
```

The `rild` daemon supplied by the AOSP is an empty shell: After parsing its arguments, it seeks out the vendor supplied RIL library, which can be defined by the `-l` argument, specified in the `rild.libpath` property. The library is dynamically loaded, and its initializer - exported as `RIL_Init` is called. The initializer returns the library's exported RIL handlers, which are then registered, before the daemon goes into its event loop.

Though most users don't give this a passing thought, the telephony APIs used in today's mobile devices aren't that far from the modems employed in the previous millenium. In fact, the low-level call control is still carried out with the same set of commands used by modems - the "AT" commands - which may be familiar to anyone who's ever used minicom and kermit back in the day. The `rild` is responsible for opening the telephony device (which is basically a serial port) and generating these commands. The daemon also listens on the device for "unsolicited commands", which are events generated by the baseband - for example, an incoming call.

Figure 5-9: A bird's eye view of the Radio Interface Layer architecture



As shown in the figure and definition, the `rild` daemon uses the /dev/socket/rild, to provide the interface by means of which the phone application can connect to the daemon, and issue various phone-related **solicited** requests (e.g. dialing, answering, hanging up) to the baseband. The `rild` also uses the socket to propagate baseband generated events (e.g. incoming text, calls) to the application as **unsolicited** requests. The socket is not meant to be used directly, and is wrapped by the Java RIL implementation (`com.android.internal.telephony` package).

The daemon also listens on another UN*X domain socket - /dev/socket/rild-debug: As the name implies, this socket is intended for use in debugging, and is undocumented save for the source of the `debugCallback` of `libril`. It defines a set of codes, which will cause requests to be artificially injected into the RIL. (You can find a detailed discussion of the codes in Volume II).

Additionally, `rild` also has its own debug facility - radio - which results in a dedicated logging device - /dev/log/radio. Inspecting this log file using `logcat -b radio` will dump plentiful amounts of debug information, which may (in some versions of Android) show the "AT" commands used by `rild` to dial numbers and establish codes.

The Radio Interface Layer is described in greater detail in Volume II, along with the Java telephony frameworks, and the reference RIL code from the AOSP.

Graphics and Media Services

Android's graphics and media services are integral to provide the best user experience possible. This section provides a cursory glance, with a far more detailed discussion of their internals deferred to Volume II for both audio and Graphics.

surfaceflinger

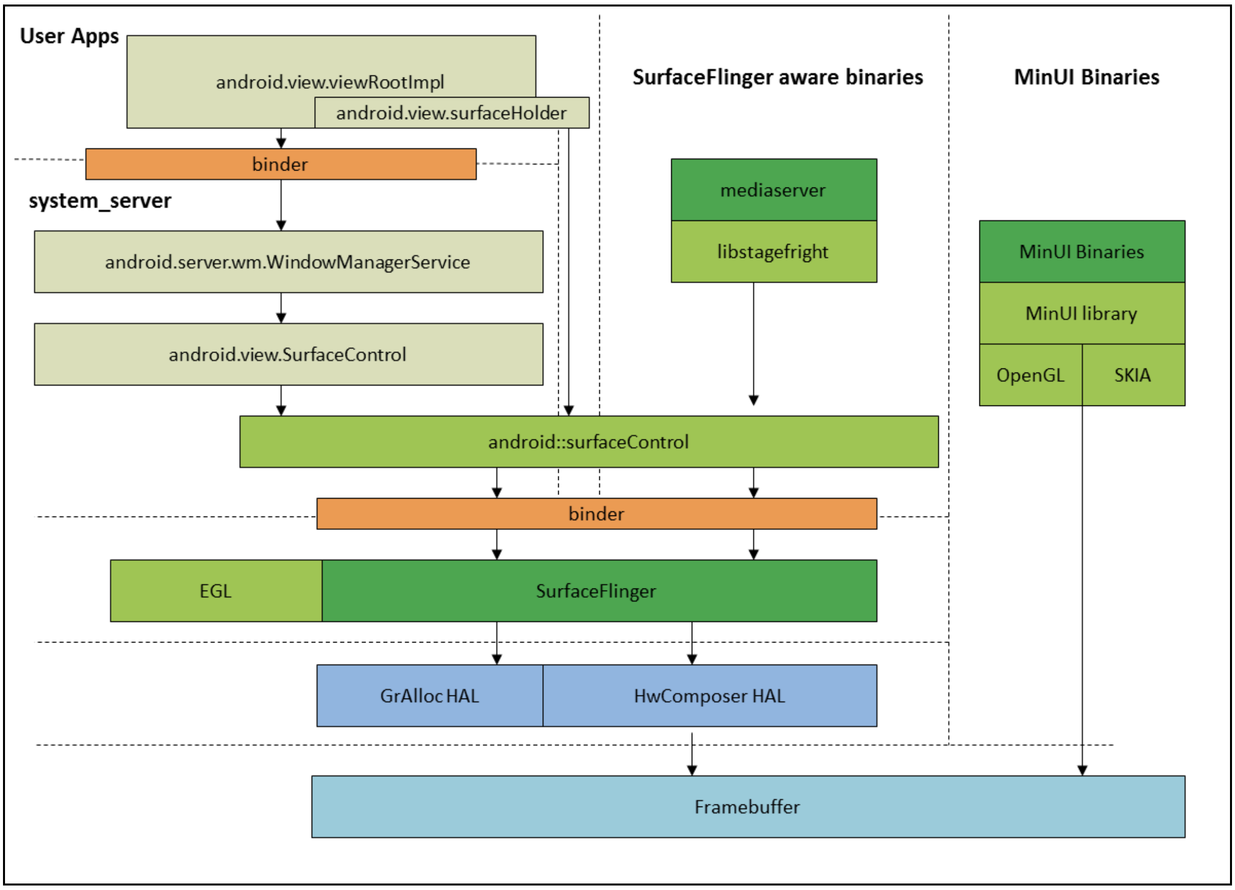
The `surfaceflinger` provides the heart of the Android Graphics Stack. The notion of a "flinger", or in other words, a "compositor", is a component which merges one or more layers of input into a single layer of output. In the case of `surfaceflinger`, the components are graphics "surfaces" (instances of `android.view.Surface`), which are either rendered by the framework as the user lays out various views, or by the developer, in the case of raw or GL Surfaces. To communicate with the `surfaceflinger`, the framework looks up the SurfaceFlinger service using `servicemanager`. The flinger therefore needs no sockets, and its definition in /init.rc is simple:

Listing 5-17: surfaceflinger definition in /init.rc

```
service surfaceflinger /system/bin/surfaceflinger
    class main
    user system
    group graphics
    onrestart restart zygote
```

Though certainly deserving of a deeper discussion (and the focus of the graphics chapter in Volume II), `surfaceflinger`'s place in the Android Graphics architecture can be conceptually grasped at a high level in the following (somewhat simplified) diagram:

Figure 5-10: A high level view of `surfaceflinger`'s functionality



bootanimation

The bootanimation service is a small binary in /system/bin, which is exclusive used by surfaceflinger as a placeholder while it (and the media frameworks) are loading. It is defined in /init.rc thus:

Listing 5-18: bootanimation definition in /init.rc

```
service bootanim /system/bin/bootanimation
    class main
    user graphics
    group graphics
    disabled        # started by SurfaceFlinger using ctl.start
    oneshot
```

The binary is essentially a very simple one - it starts up and looks for one of three zip files:

- /system/media/bootanimation-encrypted.zip: Used if the vold.decrypt property is set, indicating the file system is encrypted.
- /data/local/bootanimation.zip: Allowing the (advanced) user to drop their own animation file via adb push. If present, this will override the system boot animation.
- /system/media/bootanimation.zip: System default animation, usually supplied by vendor

The three files are tried in order, and if none of them can be found, bootanimation defaults to alternating between two images - android-logo-[mask|shine].png, both hidden in the /assets/images folder in the /system/framework/framework-res.apk (you can easily see the png files yourself if you pull the framework-res.apk to your host and unzip the file).

What makes bootanimation interesting is its raw (i.e., non-framework) graphics capability. Since it is one of the first services to load, the frameworks have yet to initialize, leaving bootanimation to fend for itself using low level OpenGL and SKIA calls. These result in direct access to the device's frame buffer (/dev/graphics/fb0), which is why is it run under uid graphics (the owner of the device node). We take a closer look at the low level graphics calls in Volume II.

Using low level calls and direct write operations to the framebuffer also enables bootanimation to override surfaceflinger even when it is active, as you can verify by running bootanimation via adb shell on your device. The device likely has it by default, though you can always upload it from the emulator image.
Running bootanimation when your device is active will hide your display behind the boot animation - either completely (in portrait mode) or partially (in landscape mode). Touch screen input will still work - but you'll likely not be able to see what you're doing until you exit (by using CTRL-C).

Most device vendors will provide a bootanimation.zip with their logo or, in some cases, the carrier's logo. Likewise cyanogen and other Android "mods" deploy a zip of their own. Such zip files must contain a desc.txt, and an assortment of images which bootanimation will cycle through. The first frame can be made to overlap with the ROM bootup image, if any, ensuring a smooth transition into the animation.

Note that some vendors may drop the default binary in favor of their own animation and accompanying sound (e.g. as Samsung has done, with /system/bin/samsungani and proprietary qmg files). Alternatively, they may change the implementation to look at other directories (e.g. HTC One M8, looking for hTC_bootup_one.zip and *vendor*_boot.zip in /system/customize/resource). The Kindle's "FireOS" is somewhere in between, retaining the bootanimation binary, but modifying it to display the "Kindle Fire" logo rather than that of Android.

The book's website[3] contains miscellaneous boot animations you can experiment with on your device. Try dropping them into /data/local (searched before /system/media and see how easy it is to replace the boot animation. Make sure the zip file is readable (if running bootanimation from adb as user shell), or else the animation will default to the "A N D R O I D" console text.

Experiment: Determining files used by bootanimation

In devices like the Nexus 5, boot animation is in /system/media/bootanimation.zip. You can pull it to a host using `adb`, and inspect its contents like so:

Output 5-10: bootanimation.zip example

```
morpheus@Forge (~)$ adb pull /system/media/bootanimation.zip
1275 KB/s (1068873 bytes in 0.818s)
morpheus@Forge (~)$ unzip -t bootanimation.zip
Archive:  bootanimation.zip
    testing: desc.txt                 OK
    testing: part0/                   OK
    testing: part0/000.png            OK
    testing: part0/001.png            OK
        ..
    testing: part1/059.png            OK
# After unzipping:
morpheus@Forge (~)$ cat desc.txt
1080 230 24
p 1 0 part0
p 0 0 part1
morpheus@Forge (~)$ file part1/000.png
000.png: PNG image data, 1080 x 230, 8-bit/color RGB, non-interlaced
```

The first line of the desc.txt specifies, in order, the width, height and frames-per-second (fps) of the boot animation. Note this is consistent with the dimensions of the individual .png files.

On other devices, however, figuring out the bootanimation files might require a little bit of reverse engineering on your part. Fortunately, using `strace` even the most complicated `bootanimation` binaries will yield their secrets. For example, consider the following output, from an HTC One M8:

Output 5-11: Figuring out files used by HTC's `bootanimation`

```
# Using strace: -f: to follow forks and threads (since binary may be multi threaded)
#               -o: to save output to a local file
#
shell@htc_m8wl:/$ strace  -f -o /data/local/tmp/out /system/bin/bootanimation
#
# Let the bootanimation run, watch logo, then hit CTRL-C.. and sift through output
#
shell@htc_m8wl:/$ grep open /data/local/tmp/out |
|                 grep -v /dev | grep -v /proc | grep -v /lib
...
21217 open("/system/etc/customer/bootanimation.zip", O_RDONLY) = -1 ENOENT
21217 writev(4, [{"\5", 1}, {"zipro\0", 6}, {"Unable to open '/system/etc/"..., 88}], 3) = 95
21217 open("/data/data/com.htc.CustomizationSetup/files/boot_anim_mns", O_RDONLY) = -1 ENOENT
21217 open("/system/customize/CID/default.xml", O_RDONLY) = 10
21217 open("/system/customize/resource/vzw_bootup.zip", O_RDONLY) = 10
...
```

As the above shows, the HTC One's animation is customized per carrier (in this example, a Verizon phone). Replacing the /system/customize/resource/vzw_bootup.zip, or adding /system/etc/customer/bootanimation.zip (searched first) will modify the boot animation.

mediaserver

The mediaserver is one of Android's most important components. It serves as a focal point for multimedia handling, controlling both playback and recording. It is defined in init.rc as follows:

Listing 5-19: mediaserver definitions in /init.rc

```
service media /system/bin/mediaserver
    class main
    user media
    group audio camera inet net_bt net_bt_admin net_bw_acct drmrpc mediadrm
    ioprio rt 4
```

As can be seen from the group membership, mediaserver requires permissions for audio, camera, network services, and the DRM framework (described next). The mediaserver, however, is really just a container for the actual services, somewhat like the concept of a service host (svchost.exe) in Windows. Table 5-12 shows the services hosted:

Table 5-12: The mediaserver services

Service	Published Name	Provides
AudioFlinger	media.audio_flinger	Audio playback. The service gets one or more PCM audio streams as input, and "flings" them into a merged stream.
AudioPolicyService	media.audio_policy	Audio policy. Informs AudioFlinger of the volume setting and target audio device
CameraService	media.camera	Camera services. Its main client is the camera app, whether the Android supplied one, or the vendor's
MediaPlayerService	media.player	Playing audio and video.

KitKat's mediaserver lays the groundwork for extensions, by providing a registerExtensions() function, though at present no extensions are defined. We discuss the services in more detail in Volume II.

Experiment: Debugging mediaserver through the media.log service

A useful debugging feature in mediaserver is that, upon startup, it checks the value of the property ro.test_harness. If set, it forks the mediaserver instance as a child of the MediaLogService (media.log) process, which (by virtue of parenthood) collects resource usage statistics on the mediaserver itself. This can be shown in the following annotated output:

Output 5-12: Starting the media.log service using ro.test_harness

```
# Make sure property check exists in the binary (note: Crude, may yield false positive)
root@htc_m8wl:/ # grep ro.test_harness /system/bin/mediaserver
Binary file /system/bin/mediaserver matches
# Set the property
root@htc_m8wl:/ # setprop ro.test_harness 1
# Kill the media server
root@htc_m8wl:/ # kill -9 $mediaserver_pid
# Et voila! media.log is now the parent of mediaserver
root@htc_m8wl:/ # ps | grep media
media     19122 1     20548  6444  ffffffff b6edaab0 S media.log
media     19123 19122 59876  9520  ffffffff b6edb26c S /system/bin/mediaserver
root@htc_m8wl:/ # service list | grep media.log
0       media.log: [android.media.IMediaLogService]
```

You can then view the mediaserver lifecycle and resource usage using logcat | grep media.log, and call dumpsys media.log.

drmserver

Android provides a Digital Rights Management (DRM) framework for copy-protected content, and the drmserver is the component responsible for being the focal point of all DRM requests. It is defined in /init.rc like so:

Listing 5-20: drmserver definitions in /init.rc

```
service drm /system/bin/drmserver
    class main
    user drm
    group drm system inet drmrpc
```

In truth, to say that Android provides a "framework" is somewhat of a misnomer, since it defines just the APIs, but not any actual content verification logic. The actual work is left for vendors to implement by a plug-in architecture. The plug-ins are shared object files, loaded by enumerating /vendor/lib/drm, and /system/lib/drm. The drmserver is thus quite small, consisting only of a main() with a few lines , which register a DrmManagerService (drm.drmManager) with the ServiceManager, and call on its internal DrmManager class to service incoming DRM calls from the framework, by finding the appropriate plug-in for the content.

Output 5-13: Viewing DRM plugins on a Galaxy S3

```
shell@android:/ $ ls -l /vendor/lib/drm
/vendor/lib/drm: No such file or directory # No vendor specific DRM modules
1|shell@android:/ $ ls -l /system/lib/drm
-rw-r--r-- root     root         48336 2012-05-2
-rw-r--r-- root     root        117224 2012-05-21
-rw-r--r-- root     root         68944 2012-05-
-rw-r--r-- root     root         48604 2012-05-21 1
-rw-r--r-- root     root         65312 2012-05-
-rw-r--r-- root     root         48212 2012-05-2
-rw-r--r-- root     root         65012 2012-05-21 17:
-rw-r--r-- root     root         64836 2012-05-2
```

To be considered valid and called by the DrmManager, a plug-in must conform to the IDrmEngine interface specification defined in IDrmEngine.h. A more detailed explanation of the specification, including an examination of DRM message flow using a PassThru module, can be found in Volume II.

Other Services

The remaining services in the main class are somewhat difficult to group, as they provide different facets of system support. Nonetheless, that does not make them any less important.

installd

The installd daemon is responsible for package installation and removal. Whichever way a package is installed - by downloading the .apk directly, via Google Play or via adb install - installd gets involved in the process. The daemon itself, however, is passive, listening on a socket set up by init, over which commands (generated by the Android framework) are delivered. The socket is defined in the daemon's /init.rc service definition:

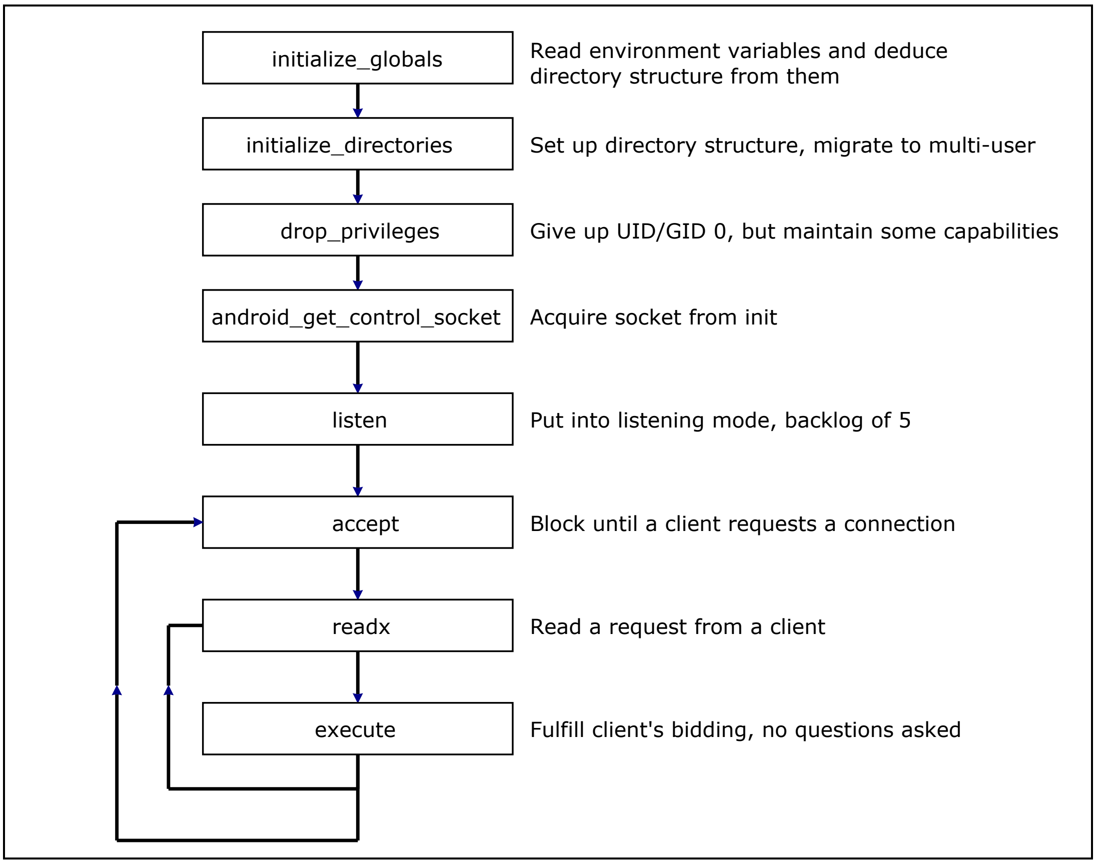
Listing 5-21: installd definition in /init.rc

```
service installd /system/bin/installd
    class main
    socket installd stream 600 system system
```

Startup

startup of installd proceeds as shown in Figure 5-11:

Figure 5-11: Startup of the installd daemon:


initialize_globals
Read environment variables and deduce directory structure from them
initialize_directories
Set up directory structure, migrate to multi-user
drop_privileges
Give up UID/GID 0, but maintain some capabilities
android_get_control_socket
Acquire socket from init
listen
Put into listening mode, backlog of 5
accept
Block until a client requests a connection
readx
Read a request from a client
execute
Fulfill client's bidding, no questions asked

Upon startup, the installd is charged with setting up and maintaining the directory structure where apps are to be installed. The base name is obtained from the ANDROID_DATA environment variable, set up by init to be /data. To this base, installd appends APP_SUBDIR (app/), PRIVATE_APP_SUBDIR (app-private/), APP_LIB_SUBDIR (app-lib) and MEDIA_SUBDIR (media/).

installd also obtains two other environment variables - ANDROID_ROOT (pointing to /system) and ASEC_MOUNTPOINT (pointing to /data/asec). Once it has deduced its directory structure, it proceeds to initialize the directories - making sure they exist (as /data initially starts up empty on factory default). As of Jellybean, installd is also charged with migrating the directory structure to allow multi-user. The steps taken are as follows:

- Create the /data/user directory, with ownership system:system and mode rwx--x--x
- Create a symbolic link from /data/user/0 to /data/data
- Upgrade /data/media to /data/media/0, moving any preexisting media there
- Create a /data/media/## directory for any other existing users
- Move OBBs from /data/media/0/Android/obb to /data/media/obb so they can be shared amongst users, and reduce overall filesystem usage
- Ensure user media folders (/data/media/##) exist and are media:media rwxrwx---.

Though it is started as root, as of Jellybean, the installd employs the principle of least privilege. One of the first calls it makes is to drop_privileges(), which sets the uid/gid to AID_INSTALL. It also makes use of Linux capabilities (q.v. Chapter 8) to maintain chown(2), setuid(2)/setgid(2) and DAC override, as it needs them to deploy and remove packages owned by different user and group ids.

Finally, installd acquires the control socket (/dev/socket/installd), and enters an accept loop, waiting for connections from client. Once a client forms a connection, an inner read/execute loop handles this connection until it closes (meaning that installd can only handle only one client at any given time). An interesting observation is the installd doesn't perform any verification of "caller id" on the socket, and relies on the socket being chmod()ed to system:system rw-------. Because only one client can be served at a time, there is the implicit assumption that it is held by the PackageManager. Note also, that installd performs no signature verification on APKs, assuming that its caller has done so already.

Commands

The framework uses the PackageManager, via the undocumented com.android.server.pm.Installer class, to provide a Dalvik-level API trusted applications can use in order to install or remove the various apps. The API methods are mapped to the commands sent over the socket (as strings, preceded by a two byte binary length),which are shown in Table 5-13:

Table 5-13: Installd commands (L commands in green)

Command	Arguments	Use
ping		Null command, used for connectivity
install	*pkgname uid gid seinfo*	Install package specified by *pkgname* under uid/gid with SELinux context specified by *seinfo*
dexopt	*apk_path uid is_public*	Optimize dex file of APK, creating an .odex file
movedex	*src dst*	Rename DEX file specified by *src* to *dst*.
rmdex	*pkg*	Remove DEX file of package specified by *pkg*
remove	*pkgname, userid*	Remove package specified by *pkg* installed under *uid*.
rename	*oldname newname*	Rename package from *oldname* to *newname*
fixuid	*pkgname uid gid*	Fix package *pkgname* so it is owned by *uid*:*gid*
freecache	*free_size*	Free cache so it has *free_size* bytes left.
rmcodecache	*pkgname uid*	Remove code cache of package *pkgname* owned by *uid* from cache.
rmcache	*pkgname uid*	Remove package *pkgname* owned by *uid* from cache.
getsize	*pkgdir uid apkpath*	Return the size of the directory specified by *apkpath*
rmuserdata	pkgname uid	Remove user data used by package *pkgname* owned by *uid*.
movefiles		Execute scripts in /system/etc/updatecmds
linklib	*pkgname asecLibDir uid*	Link native library to its real location
mkuserdata	*pkgname uid userid*	Creates data directory for package (owned by *id* for user *userid*), and installs symlinks
mkuserconfig	uid	Ensure that /data/misc/user/*uid* directory exists.
rmuser	*uid*	Remove user *uid*
idmap	target overlay	Runs /system/bin/idmap.
restorecondata	pkgname seinfo uid	Restore *seinfo* on *pkgname* owned by *uid*.
patchoat	*apk_path, uid, is_public, pkgname, instruction_set, vm_safe_mode, should_relocate*	Patch OAT file to relocate it in memory.

The phases of package installation, along with the `Installer` service, are both discussed in Volume II.

keystore

The keystore service is, as its name implies, provides a storage service for keys. By design, it can provide storage for any arbitrary name-value pairs, though in practice it is only used for key storage. It is defined in /init.rc as follows:

Listing 5-22: installd definition in /init.rc

```
service keystore /system/bin/keystore /data/misc/keystore
    class main
    user keystore
    group keystore drmrpc
```

The argument to the keystore daemon - /data/misc/keystore - is the directory used to hold the various keystore files. As of Jellybean, each user has its own keystore directory, with the primary user using /data/misc/keystore/0. The keystore password of the user is stored (encrypted with a derivative of the lock screen authenticator) in a .masterkey file, and per-app keystores are in files following the *AID_xxxxx* convention.

Unlike other daemons, and as of 4.4, keystore no longer uses a socket. It is accessible only via servicemanager, wherein it is published using the name android.security.keystore. The framework client of the keystore service is the java.security.KeyStore class. This is a developer-accessible class modeled after the Java standard, and is reasonably documented[4], but not fully so: There are quite a few other public methods available, which the Android documentation chooses to omit. The keystore_cli command allows partial command line native-level access to the keystore. Table 5-14 shows the commands exposed by the class and service, showing the commands not implemented by the cli as grayed.

Response codes are defined in system/security/keystore/include/keystore/keystore.h and mapped to error strings by keystore_cli as follows:

Table 5-14: Keystore error codes

Code	Constant	Code	Constant
1	[STATE_]NO_ERROR	6	PERMISSION_DENIED
2	[STATE_]LOCKED	7	KEY_NOT_FOUND
3	[STATE_]UNINITIALIZED	8	VALUE_CORRUPTED
4	SYSTEM_ERROR	10-13	WRONG_PASSWORD_[0123]
5	PROTOCOL_ERROR	14	SIGNATURE_INVALID

Access to keys is governed by uid (hence its use as an argument), so each application effectively has its own private store.In addition, similar to the ACLs hard coded in init, the keystore daemon maintains a user_perms array of permissions, with specific exclusions for AID_SYSTEM (all access), AID_VPN and AID_WIFI (get, sign and verify only). The AID_ROOT user is actually the most restricted, with 'get' being the only operation allowed (in practice, though, it's a simple matter to su to AID_SYSTEM).

Table 5-15: keystore commands, exported by `java.security.KeyStore`

1	`test()`	Test keystore daemon is active
2	`byte[] get(String name)`	Get value corresponding to *name*
3	`insert(String name, byte[] val, int uid, int flags)`	Insert a *name/value* combination into keystore belonging to *uid*, with *flags*
4	`int del(String name, int uid)`	Delete *name* (and value) from keystore belonging to *uid*
5	`exist(String name, int uid)`	Check if *name* exists in keystore belonging to *uid*
6	`saw(String prefix, int uid)`	List all keys beginning with *prefix* in *uid*'s keystore
7	`reset()`	Reset (wipe) keystore
8	`password(String password)`	Change keystore password to *password*
9	`lock()`	Lock keystore, requiring password to unlock
10	`unlock(String password)`	Unlock previously locked keystore by supplying *password*.
11	`zero()`	Check if keystore is empty.
12	`generate(String name, int uid, int keyType, int keySize, int flags, byte[][] args)`	Generate a private/public keypair in keystore owned by *uid* under *name*. The key can then be used to sign and verify, or retrieve the public key - but the private key will remain inaccessible.
13	`import_key(String name, byte[] data, int uid, int flags)`	Import key specified in *data* blob into keystore owned by *uid* into key *name*.
14	`byte[] sign(String name, byte[] data)`	sign *data* with key corresponding to *name* without actually retrieving key.
15	`verify(String name, byte[] data, byte[] signature)`	Verify *signature* on *data* using key specified by *name*
16	`byte[] get_pubkey(String name)`	Get a public key associated with *name*
17	`del_key(String name, int uid)`	Delete key identified by *name* in keystore belonging to *uid*
18	`grant(String name, int uid)`	Grant *uid* access to key *name*
19	`ungrant(String name, int uid)`	Revoke access to key *name* to *uid*
20	`long getmtime(String name)`	Get modification time of *name*
21	`duplicate(String srcKey, int srcUid, String destKey, int destUid)`	Copy the key specified by *srcKey* in keystore belonging to *srcUid* to keystore owned by *destUid* under name *destKey*.
22	`is_hardware_backed(String keyType)`	Return integer specifying whether or not *keyType* is backed by a hardware keystore implementation
23	`clear_uid(long uid)`	Clear keystore for user *uid*
24	`reset_uid(long uid)`	Reset keystore for *uid*
25	`sync_uid(long uid)`	Sync keystore for *uid*
26	`password_uid(long uid)`	Set password for *uid*

Experiment: Interfacing with `keystore`

The following experiment (if carried out incorrectly) can potentially lock you out of your keystore. It's therefore recommended to try it on an emulator image, rather than on a real device - at least until you feel confident

You can call on the the keystore service either through the `keystore_cli` utility, or directly through `service call`. Considering not all of its commands are (at the time of writing) implemented, it can make more sense to use `service call android.security.keystore` with the specific numeric codes and arguments instead. Output 5-14 demonstrates this in a "split-screen" like view:

Output 5-14: Using `keystore_cli` and `service` to interact with the keystore service

```
# Note root is not allowed access to the keystore
...# keystore_cli test                     service call android.security.keystore 1
test: Permission denied (6)                 Result: Parcel(00000000 00000006   '........')
#
# SU to system to gain full control
...# su system
...$ keystore_cli test                     service call android.security.keystore 1
test: No error (1)                          Result: Parcel(00000000 00000001   '........')
#
# Set keystore password to be "123", then lock keystore
...$ keystore_cli password 123             service call android.security.keystore 8 s16 123
password: No error (1)                      Result: Parcel(00000000 00000001   '........')
...$ keystore_cli lock                     service call android.security.keystore 9
lock: No error (1)                          Result: Parcel(00000000 00000001   '........')
#
# Attempt to unlock with bad password: System will count down attempts (errors 13,12,11,10)
# then reset (return UNINITIALIZED followed by SYSTEM_ERROR)
...$ keystore_cli unlock bad               service call android.security.keystore 10 s16 bad
unlock: Wrong password (4 tries left) (13)  Result: Parcel(00000000 0000000d   '........')
```

As table 5-16 shows, however, there is a large command subset you cannot call through `keystore_cli`. You can the grayed commands in the table by invoking them through their numeric code, passing arguments using the `service` utility's `s16` and `i32` specifiers.

Output 5-15: Generating a key directly through `service call`

```
system@generic$ service call android.security.keystore 12 s16 name1 \  # Name
                                       i32 13        \  # Len
                                       s16 hello     \  # Value
                                       i32 -1 i32       # UID (-1 = caller)
Result: Parcel(00000000 00000001   '........')
#
# Attempt to retrieve value from keystore
system@generic$ service call android.security.keystore 2 s16 name1
Result: Parcel(
  0x00000000: 00000000 0000000d 00000005 00650068 '...........h.e.'
  0x00000010: 006c006c 0000006f                   'l.l.o...       ')
```

The keystore implementation (over a Hardware Abstraction Module and/or possibly a hardware component) is discussed in Volume II.

debuggerd[64]

Try as hard as developers will, their applications will inevitably face bugs, which will result in crashes. In order to fix those bugs, there must be an efficient mechanism to collect the crash data. On a desktop system, the crash results in a core dump - but that is simply not an option in a mobile device. Core dumps are often very large - in the hundreds of MB and sometimes more - and space is limited. What more, even if the core dump were saved, it's not a trivial matter to move such large files out of the device.

Similar to iOS's CrashReporter, Android introduces `debuggerd`. This small daemon is normally dormant, sleeping on its socket, until an application crashes. All processes on Android, thanks to the Android linker, automatically install a signal handler for the lethal signals, shown in table 5-16:

Table 5-16: Signals caught by debuggerd

Signal	Full Name	Example
`ILL`	Illegal Instruction	Illegal machine opcode
`TRAP`	Debugger Trap	Breakpoint
`ABRT`	Voluntary Abort	Assertion failure
`BUS`	Bus Error	MMU fault
`FPE`	Floating Point Exception	Division by zero
`SEGV`	Segmentation Violation	NULL pointer dereference
`PIPE`	Broken Pipe	Termination of process on read end of pipe

All signals use the same action, `debuggerd_signal_handler`, which establishes a connection to `debuggerd` over its socket, and sends it a message. The message wakes up the daemon, and causes it to engrave a **tombstone**. A tombstone is, essentially, a crash report, which `debuggerd` generates by attaching to the failing process (using Linux's `ptrace(2)` APIs), catching its signal for it, and inspecting its memory. This way, rather than a full core dump, a tombstone can (hopefully) capture the essence of a crash and perform the basic crash processing. Tombstones are created in /data/tombstones.

If the `debug.db.uid` property is set to the uid of the crashing process, `debuggerd` freezes the process in its final death throes and waits for user to start `gdbserver`. It logs a message which can be easily seen in `logcat`:

Listing 5-21: Android log messages emitted by `debuggerd`

```
I/DEBUG   (   24): ********************************************************
I/DEBUG   (   24): * Process pid has been suspended while crashing.  To
I/DEBUG   (   24): * attach gdbserver for a gdb connection on port 5039
I/DEBUG   (   24): * and start gdbclient:
I/DEBUG   (   24): *
I/DEBUG   (   24): *     gdbclient app_process :5039 pid
I/DEBUG   (   24): *
I/DEBUG   (   24): * Wait for gdb to start, then press HOME or VOLUME DOWN key
I/DEBUG   (   24): * to let the process continue crashing.
I/DEBUG   (   24): ********************************************************
```

The `debuggerd` uses the low level Linux `EV_*` APIs (discussed in Volume II) to wait until the user presses one of the keys, and lights up the debug (red) led on the device to draw the user's attention.

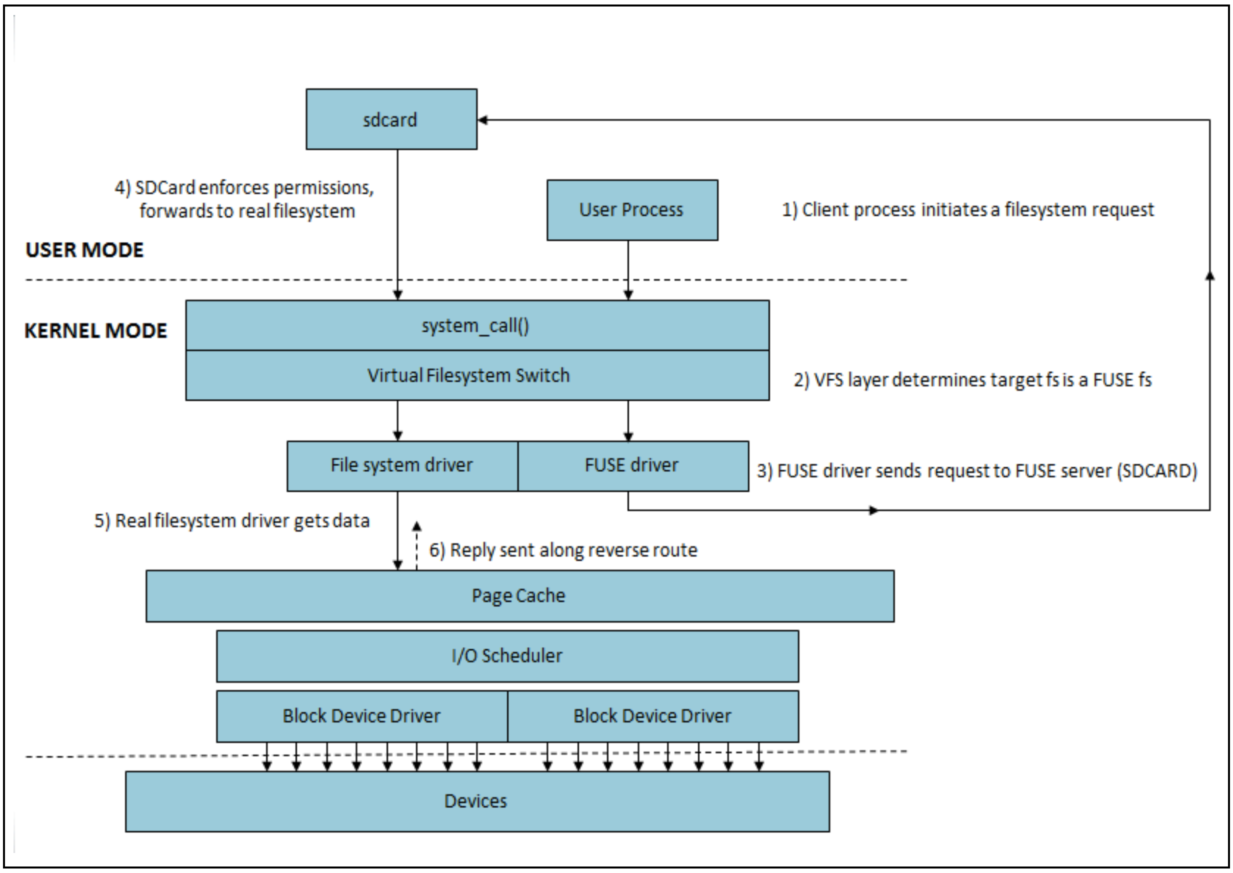
On 64-bit systems, an instance of `debuggerd64` is also spawned, in order to handle the different instruction set, memory layout and ABI. We discuss debugging in general and tombstones in particular, in Volume II.

sdcard

Not all Android devices necessarily support SDCards - but in those which do, the sdcard daemon provides the user-mode support, including the enforcement of permissions on the otherwise permission-less FAT filesystem. This is accomplished by using a mechanism known as FUSE (File systems in USEr mode). The mechanism registers a stub filesystem in the kernel, and passes all of its calls to the user space daemon - in this case, sdcard. Using FUSE allows for much more flexibility and stability than implementing a filesystem in kernel mode. The complexity involved with filesystem code, coupled with the untrustworthiness of possibly (potentially maliciously) corrupted structures make FUSE a good choice for relatively infrequently accessed filesystems. (There is a significant performance impediment involved in a kernel to user mode and back to kernel mode traversal, which makes FUSE somewhat of a poor performer in other cases).

Figure 5-12 shows the flow of a file system request from a user mode client to the kernel, its redirection via FUSE to the SD card daemon, and back to the originating client.

Figure 5-12: To there and back again: The SDCard daemon and its operation, via FUSE


sdcard
4) SDCard enforces permissions, forwards to real filesystem
User Process
1) Client process initiates a filesystem request
USER MODE
KERNEL MODE
system_call()
Virtual Filesystem Switch
2) VFS layer determines target fs is a FUSE fs
File system driver
FUSE driver
3) FUSE driver sends request to FUSE server (SDCARD)
5) Real filesystem driver gets data
6) Reply sent along reverse route
Page Cache
I/O Scheduler
Block Device Driver
Block Device Driver
Devices

The sdcard daemon accepts the command line parameters shown in the following table:

Switch	Purpose
-u *uid*	Specify user id to run as, for ownership of the filesystem. Usually 1023 (AID_MEDIA_RW)
-g *gid*	Specify group id to run as, for ownership of the filesystem. Usually 1023 (AID_MEDIA_RW)
-l *path*	Specify path to real mountpoint of the filesystem
-t *#*	Specify number of threads (default is 2)
-l	Specifies mount is a legacy (emulated) mount
-d	Derive permissions from path
-s	Split permissions for media, av, etc

After processing the command line parameters, `sdcard` opens the /dev/fuse node, (to communicate with the kernel driver), and then calls the `mount(2)` system call to perform a FUSE mount, specifying hard coded options to the system call: `fd=`*`/dev/fuse fd`*`,rootmode=40000,default_permissions,allow_other,user_id=,group_id=`***`gid`***. Once privileges are dropped (to the `-u`/`-g` specified GIDs), the daemon calls the aptly named `ignite_fuse()` and enters a message loop, to handle incoming requests from the /dev/fuse fd.

The SD card filesystem, as provided by the daemon, was discussed in Chapter 2. The following experiment further exemplifies the flow of the SD Card filesystem requests, via FUSE.

Experiment: Observing `sdcard`

Android devices use the `sdcard` daemon whether or not they have an actual SDCard. The /data/media directory is mounted via an `sdcard` daemon instance as /mnt/shell/emulated. If the device also has a physical SDCard, its filesystem is also mounted, though this requires an additional instance of the daemon, as shown in the following output:

Output 5-16: Viewing FUSE filesystems mounted with `sdcard`

```
#
# Use mount to view all mounted file systems, but isolate only FUSE ones
#
shell@htc_m8wl:/ $ mount | grep fuse
/dev/fuse /mnt/shell/emulated fuse rw,nosuid,nodev,relatime,user_id=1023,group_id=1023, ...
/dev/fuse /storage/ext_sd fuse rw,nosuid,nodev,relatime,user_id=1023,group_id=1023,     ...
#
# busybox's ps applet will show you the full command line (or you can cat -tv /proc/.../cmdline)
#
shell@htc_m8wl:/ $ busybox ps | grep sdcard
 844 1023 0:02 /system/bin/sdcard -u 1023 -g 1023 -l /data/media /mnt/shell/emulated
1599 1023 0:10 /system/bin/sdcard -u 1023 -g 1023 -w 1023 -d /mnt/media_rw/ext_sd /storage/ext_sd
```

Observing `sdcard` in action is a tad trickier, however. The method demonstrated time and again in this chapter - the all powerful `strace` - can be used in this case as well, but tracing the main thread will likely show nothing. Any one of `sdcard`'s threads may be serving the FUSE requests, which means you'll first need to see which threads were created, via /proc/*$SDCARD_PID*/task, and then use `strace` on them. (The handy `-f` won't be of use here since the threads were created prior to the `strace` attachment). A good experiment is to use two adb sessions - one to trace the `sdcard` threads in, and another to try operations on the FUSE mount (e.g. `ls -l`). Doing so will show you data passing to and from the `/dev/fuse` file descriptor, as the client's requests are, in effect, proxied by the daemon, which translates them into the underlying system calls - for example in the following output, demonstrating the trace of executing `ls -l /storage/ext_sd`:

Output 5-17: Tracing `ls -l` through `sdcard`

```
# Get stat() request from client
read(3, "8\0\0\0\3\0\0\0?P\1\0\0\0\0\0\1\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 262224) = 56
# perform stat() on underlying file system
lstat64("/mnt/media_rw/ext_sd", {st_mode=S_IFDIR|0770, st_size=32768, ...}) = 0
# relay to client
writev(3, [{"x\0\0\0\0\0\0\0?P\1\0\0\0\0\0", 16},  .....   = 120
# Get getdents64() request from client
read(3, "P\0\0\0\34\0\0\0?P\1\0\0\0\0\0\1\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 262224) = 80
# perform request
lseek(5, 0, SEEK_SET)                  = 0
getdents64(5, /* 60 entries */, 4200)   = 2216
# relay to client
writev(3, [{"0\0\0\0\0\0\0\0?P\1\0\0\0\0\0", 16},
           {"????\0\0\0\0\n\0\0\0\0\0\0\0\7\0\0\0\4\0\0\0Android\0", 32}], 2) = 48
```

zygote[64]

Though last, both alphabetically and in this chapter, the `zygote` is hardly the least of all services. It provides the core support for all of the Android Framework Runtime services, in the form of an initialized empty Dalvik Virtual Machine, stopped just shy of the main class loading. The /init.rc definition is as follows:

Listing 5-22: Zygote definitions in /init.rc

```
service zygote /system/bin/app_process -Xzygote /system/bin \
                                       --zygote --start-system-server
    class main
    socket zygote stream 660 root system
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
    onrestart restart media
    onrestart restart netd
```

As the listing shows, zygote's "true name" is `app_process`. The name "zygote" however, is far more apt, as this process mimics, in some senses, its namesake. Just like the biological zygote, this process is full of unlimited potential - it can load any Dalvik class specified, and can become any user. This, however, is a one-way process (again, just like the biological parallel). The rest of the command line provides the arguments, all of which but the double-dashed get passed directly to the Dalvik VM. The last two arguments get processed by app_process itself, and result in the VM loading the `Zygote` class, and `fork()`ing to start the `system_server` process.

The `system_server` process (discussed in detail in next chapter) goes on to load all of the Android runtime frameworks, whereas the `Zygote` binds its socket (/dev/socket/zygote) to listen for incoming requests. When such requests will arrive, they will contain a class name to load, and Zygote will similarly `fork()` and load the classes - which will result in the creation of a new app. A new "Life" will be born. But all these apps, and indeed zygote itself, are, from the Linux perspective, merely instances of `app_process`, which renames itself accordingly (and you can verify with an `ls -l /proc/`***`pid`***`/exe`).

Because `Zygote` can specialize into potentially any process, it must leave all its options open. Because of that, it maintains its root privileges, and an unlimited set of capabilities. Prior to forking, however, Zygote drops all privileges, and then calls `setuid(2)/setgid(2)` to assume the AID of the app in question. Because all this happens prior to loading any app code (both VM and native), this setup is considered secure. It has, however, suffered in the past from vulnerabilities (e.g. Froyo's Zysploit, due to not checking `setuid(2)`'s return value), and more recently (2013) from a fork-bomb denial of service attack.

Further, it follows that all apps must be spawns of Zygote - the only exception to that are direct invocations of `app_process` from the command line (for example, by the upcall scripts mentioned in Chapter 2). You can verify that for yourself by looking at the output of `ps` on your device: Processes will either be offspring of `init` (PID 1), as holds for all the daemons discussed in this chapter, or Zygote spawn, in which case some other PID will be the PPID - and you can bet the PID is that of Zygote.

The rationale behind Zygote

At this point, you might be asking yourself why go to all this trouble, just in order to load a new app. But it turns out that the effort invested yields plentiful dividends, in not one but two ways:

- **Application startup time is greatly decreased:** Regardless of actual app, all virtual machines need to be initialized in the same, deterministic way. The class loading of the app is the final stage of this process, but the real overhead is in the loading of the multitude of runtime classes which make up Android's rich frameworks. If you imagine the app loading to be a race, of sorts, using Zygote enables Android to "camp by the finish line", and run the last leg of the race - which is relatively short, reducing load time by orders of magnitude.
- **Memory sharing is optimized:** Because all virtual machines fork from Zygote, they can take advantage of implicit memory sharing performed by the kernel. Specifically, though each instance of `app_process` has its own virtual memory, the majority of that memory - being read only (class code) - can be backed by only one physical copy in RAM. The rest of the memory (class data, read write) - can be backed by additional pages only when absolutely necessary (a technique known as **copy-on-write**). thus, most of Android's apps implicitly share 80-90% of their memory with other apps (and with `system_server`, which is the first real instance of a full VM). This maximizes memory usage and allows quite a few apps to "fit" in RAM, even on relatively low memory devices. You can see an example of this if you skip ahead to Output 7-14, in a hands on experiment showing the use of the `procrank` and `librank` utilities, which provide memory usage diagnostics.

It is Zygote's unique design, which has enabled it to triumph where Java has failed. There are additional optimizations in the Virtual Machine architecture itself (for example, keeping reference counts separate from objects), but those merit a deeper discussion from the programmatic perspective, which is left for Volume II. Likewise, the step by step walk through of application startup can be found there.

The approach is not without some drawbacks: As we discuss in Chapter 8, forking all the applications from the same binary effectively undermines Address Space Layout Randomization (ASLR), which is an important layer of security against code injection attacks. That said, the needs of the many outweigh those of the few, and so performance trumps security. Recent academic research has proposed Morula[5] (another biological term, resulting from Zygote division) as an alternative architecture, which may address ASLR shortcomings, but that has yet to make its way to Android.

With the move to the Android RunTime (ART), Zygote's architecture becomes even more efficient, as all of the preloaded classes are also precompiled. That, however, complicates matters somewhat, because 32 and 64-bit layouts are not compatible.

Zygote 32 and Zygote 64

With the move to 64-bit computing on the one hand, and the need to retain 32-bit compatibility on the other, Android now has to maintain not one but two versions of Zygote. In a 64-bit architecture, the "secondary zygote" is a 32 bit process, which is started by an instance of `app_process32`. Because the primary (i.e. 64-bit) zygote instance holds the zygote socket, the secondary zygote requires an additional socket. This is shown in listing 5-23:

Listing 5-23: Zygote32 definitions in /init.zygote64_32.rc

```
service zygote /system/bin/app_process -Xzygote /system/bin \
service zygote_secondary /system/bin/app_process32 -Xzygote /system/bin \
                                     --zygote --socket-name=zygote_secondary
    class main
    socket zygote_secondary stream 660 root system
    onrestart restart zygote
```

User applications are entirely oblivious to which instance of Zygote they are using, though this makes a difference in terms of which libraries are loaded - and therefore JNI. 32-bit Zygote instances use /system/lib, whereas 64-bit ones uses /system/lib64. Inspecting the process address space maps (by using `cat /proc/`*`pid`*`/maps`) will reveal the different mappings, as further discussed in Chapter 7.

Summary

This chapter covered the native services of Android, which are the daemon processes spawned by init through the various `service` entries in the /init.rc files. The native processes are responsible for various housekeeping operations, as well as providing the basic level of support for the system frameworks.

The framework services, however, are another matter in entirety - Due to the large number of services and the detail required, we leave that discussion for Volume II. Nonetheless, the next chapter provides the preliminaries, by providing an overview of the service architecture, through an elaboration on `servicemanager` and `system_server`. It is the latter process which serves as the container for all services, and the one which takes over the UI from the `bootanimation`.

Files discussed in this Chapter

Section	File/Directory	Contains
adb	system/core/adb/	Implementation of adb, both client and server
	f/b/s/ja/com/and/ser/usb/UsbDebuggingManager.java	USB Debugging Manager server, used by `system_server`
vold	f/b/s/ja/com/and/ser/MountService.java	The Mount Service Manager, used by `system_server`
debuggerd	/system/core/debuggerd	Source of debuggerd
installd	f/native/cmds/installd	Source of installd
bootanimation	f/base/cmds/bootanimation/BootAnimation.cpp	Bootanimation source
sdcard	sys/core/sdcard/sdcard.c	Source of sdcard

References

1. Android Full disk encryption: http://source.android.com/devices/tech/encryption/android_crypto_implementation.html
2. Android Explorations: http://nelenkov.blogspot.com/2012/08/changing-androids-disk-encryption.html
3. BootAnimation collection: http://www.NewAndroidBook.com/bootanimations/
4. Android Developer, Keystore Documentation: http://developer.android.com/reference/java/security/KeyStore.html
5. Georgia Tech, "From Zygote to Morula": https://taesoo.gtisc.gatech.edu/pubs/2014/morula/morula.pdf

VI: The Framework Service Architecture

The previous chapter painted only a partial picture of the runtime services in Android. The services detailed therein were all native-level processes - implemented in C/C++, and with no direct programmatic interface from the Java layer. As such, they can be classified as services which support the operating system itself. Applications, however, make use of an entirely different set of services, provided by the Dalvik-level frameworks, with special interfaces. These services have a Java language interface, and most of which run in the context of one process: system_server, and are reachable with the help of `servicemanager`.

Both `servicemanager` and `system_server` were introduced throughout the previous chapter. `servicemanager` in the section dealing with core services, and `system_server` as a subset of Zygote - i.e. started by zygote when the `--start-system-server` argument is provided. Both, however, deserve a much more in-depth investigation, as together they provide the support and the context of the entire Android framework service architecture - which is what this chapter discusses.

We begin by revisiting the service manager, which provides the role of an endpoint mapper (that is, allows service location and invocation). The services make themselves visible to clients by registering with `servicemanager`, and from that point on clients may approach the `servicemanager` and request a connection (or a handle) to the service. All framework services are invoked in the same way, and this service calling pattern, is discussed next. In particular, two key components are introduced - The Android Interface Definition Language, or **AIDL**, providing the interface (or set of APIs) exported by the services, and the `service` utility, which allows the testing and debugging of those interfaces from the command line.

The underlying transport for service (and, indeed, all inter-app) communication is Android is the **Binder** mechanism, which is accessible to applications via `/dev/binder`. What looks like a simple device node is, in fact, an elaborately designed IPC framework, which is charged with not only dispatching messages, but also with passing around objects, descriptors, and more, as well as providing reliability and security. This is discussed as we take a a closer look at service internals.

Lastly, we take a look at system_server itself, which functions as the service host process, wherein most services* are implemented as threads. We detail the startup, operation, and internals of this important process. As for the services themselves - they're detailed in the next volume of this work.

* - A few notable exceptions are SurfaceFlinger and the media services. Note that application (3rd party) services run in their own process.

Revisiting servicemanager

If you recall from the previous chapter, one of the services classified by init in the "core" class is the servicemanager. The other key services are dependent on it, and must be restarted with it if it crashes. Further, servicemanager is designated as critical, which means that init will agressively attempt to restart it, or boot to recovery if it fails to do so.

The reason behind the utmost importance of the servicemanager is its function: It serves as the locator, or directory, for all other operating system services. If any application or system component needs to use another service, be it what may, it must first consult the servicemanager to obtain a handle. Similarly, services cannot expect clients until they register their presence with it. It is for this reason that, if the manager is restarted, so must all of its dependents - after all, restarting implies the service directory must be rebuilt from scratch, and services thus need to register. It likewise follows that, if servicemanager cannot operate, Inter-Process Communication (IPC) cannot subsist.

The IPC model of Android is discussed later in this chapter. For the moment, however, suffice it to say that it is provided by a dedicated kernel component - the Binder. User-mode services access the binder for IPC via a character device node - /dev/binder, which is readily accessible (readable/writable) to all processes. Only one user-mode process at a time, however, can request to register as a **context manager** with the Binder, however, and from that point on it becomes the focal point for all other processes - both clients, and servers. The servers must register their service name and interface with the context manager, and the clients must consult the context manager in order to lookup and find the service.

The servicemanager is therefore a pretty small binary, with a simple operation: a call to binder_open obtains the /dev/binder descriptor, and a call to binder_become_context_manager establishes its position. Thereafter, the servicemanager enters an endless binder_loop, which blocks on the descriptor, until a transaction (i.e. request from a client) occurs. This wakes process, and calls its svcmgr_handler callback, which processes the transaction.

The service lookup must somehow be bootstrapped - in other words, the servicemanager should be globally accessible, so that services can register with it, and clients can look them up. At the native level, services and clients alike can call on defaultServiceManager() to get a handle to the service manager (technically, to its interface, as a sp<IServiceManager>). The interface (defined in IServiceManager.h) exposes a simple set of transaction request codes. Table 6-1 shows the requests, as well as the native level calls which implement them. Note, that there is no API to remove the service. Services are automatically removed when their proceeses die, because Binder can detect that, and send a death notification.

Table 6-1: servicemanager requests and the programmatic methods to invoke them

Request Code	API	Notes
SVC_MGR_ADD_SERVICE	addService(name, service, allowIsolated)	Used by servers to register themselves with the service manager. Servers can decide whether or not they want to allow isolated (sandboxed) processes to connect.
SVC_MGR_GET_SERVICE SVC_MGR_CHECK_SERVICE	checkService(*name*)	Get a handle to the service specified by *name*.
SVC_MGR_LIST_SERVICES	listServices()	Return a vector (list) of all services. Not used by the framework, but used by service list.

The `addService` functionality is considered sensitive: UID 0 or 1000 (AID_SYSTEM) can freely register services, but other system services are restricted. Up to and including KitKat, this is done by a hard coded `allowed` list, restricting registration, as shown in Table 6-2:

Table 6-2: Hardcoded service registration restrictions

AID_MEDIA	media.audio_flinger, media.log, media.player, media.camera, media_audio_policy
AID_DRM	drm.drmManager
AID_NFC	nfc
AID_BLUETOOTH	bluetooth
AID_RADIO	radio.phone, radio.sms, radio.phonesubinfo, radio.simphonebook
AID_RADIO*	phone, sms, iphonesubinfo, simphonebook
AID_MEDIA	common_time.clock, common_time.config
AID_KEYSTORE	android.security.keystore

* - These are legacy service names, deprecated by their radio.* counterparts

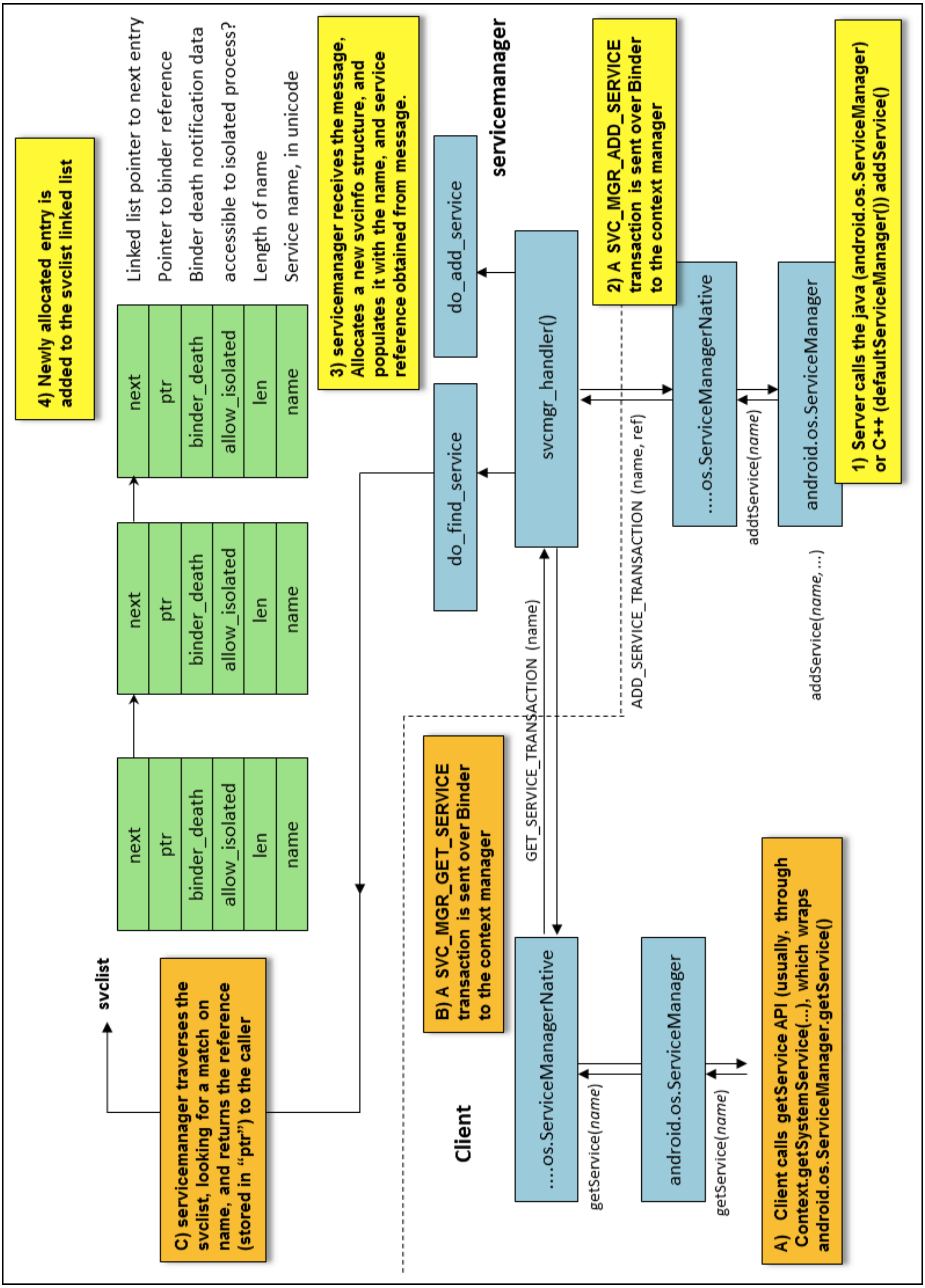
In Lollipop, the hard coded list is moved into the /service_contexts file of SELinux, which provides a far more scalable way to control services - system_server.c code is simplified by a call to `check_mac_perms()`, which then calls on `selinux_check_access()`. In this manner, service registration and lookup can be enforced for all services, further allowing the device vendor to add their own services, without the need to recompile any code.

Listing 6-1: The /service_contexts SELinux policy file

```
#line 1 "external/sepolicy/service_contexts"
accessibility                             u:object_r:system_server_service:s0
..
android.security.keystore                 u:object_r:keystore_service:s0
..
batteryproperties                         u:object_r:healthd_service:s0
batterypropreg                            u:object_r:healthd_service:s0
..
bluetooth                                 u:object_r:bluetooth_service:s0
..
common_time.clock                        u:object_r:mediaserver_service:s0
common_time.config                       u:object_r:mediaserver_service:s0
..
display.qservice                          u:object_r:surfaceflinger_service:s0
..
drm.drmManager                            u:object_r:drmserver_service:s0
..
inputflinger                              u:object_r:inputflinger_service:s0
..
media.audio_flinger                       u:object_r:mediaserver_service:s0
media.audio_policy                        u:object_r:mediaserver_service:s0
media.camera                              u:object_r:mediaserver_service:s0
media.log                                 u:object_r:mediaserver_service:s0
media.player                              u:object_r:mediaserver_service:s0
media.sound_trigger_hw                    u:object_r:mediaserver_service:s0
nfc                                       u:object_r:nfc_service:s0
..
*                                         u:object_r:default_android_service:s0
```

The programmatic APIs are wrapped by the framework class `android.os.ServiceManagerNative`, which is further encapsulated in `android.os.ServiceManager`. Apps aren't expected to use this directly, and instead call on `Context.getSystemService()` in order to look up system services, and use intents for third party services. Either way, communication with services - both system and third party - is performed over binder messages, with the `servicemanager` serving as the service directory, as shown in Figure 6-1:

Figure 6-1: Registering and accessing Android framework services


servicemanager
Client
svclist
1) Server calls the java (android.os.ServiceManager) or C++ (defaultServiceManager()) addService()
2) A SVC_MGR_ADD_SERVICE transaction is sent over Binder to the context manager
3) servicemanager receives the message, Allocates a new svcinfo structure, and populates it with the name, and service reference obtained from message.
4) Newly allocated entry is added to the svclist linked list
A) Client calls getService API (usually, through Context.getSystemService(...), which wraps android.os.ServiceManager.getService()
B) A SVC_MGR_GET_SERVICE transaction is sent over Binder to the context manager
C) servicemanager traverses the svclist, looking for a match on name, and returns the reference (stored in "ptr") to the caller
android.os.ServiceManager
....os.ServiceManagerNative
svcmgr_handler()
do_add_service
do_find_service
addService(name, ...)
addtService(name)
ADD_SERVICE_TRANSACTION (name, ref)
GET_SERVICE_TRANSACTION (name)
getService(name)
getService(name)
android.os.ServiceManager
....os.ServiceManagerNative
next
ptr
binder_death
allow_isolated
len
name
Linked list pointer to next entry
Pointer to binder reference
Binder death notification data
accessible to isolated process?
Length of name
Service name, in unicode

Experiment: Using the `service` command to interface with service manager

Android provides the `service` command line utility as a simple interface for the service manager. This simple utility also demonstrates how to use the programmatic APIs to query services. Using `service list` you can display all registered services, as well as their published interfaces (discussed later in this chapter), and using `service check`, see if a given service can be contacted.

Output 6-1 shows an output of `service list` on a Nexus 5 Android L. Because you can easily run this command on any device, the output is partial, highlighting only those services which are new in L, or are not present in the emulator.

Output 6-1: Using `service list` on an Android L Nexus 5

```
root@generic# service list
Found 93 services: # Emulator shows only 87 services, 75 in KK
1      sip: [android.net.sip.ISipService              # Not in emulato
2      phone: [com.android.internal.telephony.ITelephony
3      iphonesubinfo: [com.android.internal.telephony.IPhoneSubInfo
4      simphonebook: [com.android.internal.telephony.IIccPhoneBook
5      isms: [com.android.internal.telephony.ISms
6      nfc: [android.nfc.INfcAdapter]                # Not in emulato
7      telecomm: [com.android.internal.telecomm.ITelecommService]    # L
8      launcherapps: [android.content.pm.ILauncherApps
9      trust: [android.app.trust.ITrustManager]          #
10     media_router: [android.media.IMediaRouterServic
11     tv_input: [android.media.tv.ITvInputManager]         #
12     hdmi_control: [android.hardware.hdmi.IHdmiControlService]  #
13     media_session: [android.media.session.ISessionManager]     #
14     print: [android.print.IPrintManage
15     assetatlas: [android.view.IAssetAtla
16     dreams: [android.service.dreams.IDreamManage
..
20     voiceinteraction: [com.android.internal.app.IVoiceInteractionManagerService]
21     appwidget: [com.android.internal.appwidget.IAppWidgetService]
22     backup: [android.app.backup.IBackupManager]
23     jobscheduler: [android.app.job.IJobScheduler]  # L
...
..
38     ethernet: [android.net.IEthernetManager]    # Not in emulat
39     wifiscanner: [android.net.wifi.IWifiScanner]  # L
40     wifipasspoint: [android.net.wifi.passpoint.IWifiPasspointManager] # L
41     wifi: [android.net.wifi.IWifiManager]
42     wifip2p: [android.net.wifi.p2p.IWifiP2pManager]
43     netpolicy: [android.net.INetworkPolicyManager]
44     netstats: [android.net.INetworkStatsService]
45     network_score: [android.net.INetworkScoreService]  # L
...
55     bluetooth_manager: [android.bluetooth.IBluetoothManager]  # Not in emulator
..
87     display.qservice: [android.display.IQService]   # owned by SF,Not in Emulator
#
# Use "service check" with one of above names to see if service is alive
#
root@generic# service check media.camera
Service media.camera: found
```

The output from the command may vary considerably between devices. Some differences are obvious (for example, the Phone service will not be found on tablets), while others may be less so (vendor specific services, or Android version specific).

The `IBinder` interface defines a `dump()` method, which is used by the `dumpsys` command to provide diagnostics on services. When invoked without arguments, `dumpsys` iterates over all services in the same manner as `service list`, and dumps each in turn. In some cases, additional arguments may be supplied, which vary with each service. Some services also expose a "checkin" method, which can be used by `dumpsys -c` or `--checkin`.

The Service Calling Pattern

Android's framework services are implemented in `system_server` threads. Applications thus need to rely on Inter-Process Communication (IPC) in order to invoke them. This is where the **Binder**, Android's properietary IPC mechanism, comes into play. Applications need to call on the Binder in their own process to obtain an endpoint descriptor, which is then connected to the remote service. Methods can then be invoked through IPC messages, through a pattern known as **Remote Procedure Call** (RPC).

IPC? RPC?

The terms IPC and RPC are often used interchageably, though not often correctly. Because both terms are fundamental in the context of Android services, it's worth clarifying the difference:

- **Inter Process Communication (IPC)** is a blanket term for all forms of communication between processes. These include various forms of message passing, but also shared resources (most notably, shared memory), along with synchronization objects (mutexes and the like), meant to ensure safety in concurrent access to shared resources (i.e. prevent data corruption which occurs when two writers attempt to modify the same data item, or race conditions between readers and writers).
- **Remote Procedure Call (RPC)** is a specific term for a method of IPC, which hides the actual communication inside procedure (method) calls. The client calls a local method, which in turn is responsible for transparently handling the IPC with the remote server - which may at times be on a different machine. The method serializes its arguments into a message, which is then transported to the server's method, where the arguments are deserialized, acted upon, and the same occurs (in reverse) for passing the return values of the method, if any.

Thus, any RPC mechanism is also an IPC mechanism (the former being a special case of the latter), but not vice versa. Android's service calling pattern implements RPC, as we discuss and detail in this section. Table 6-3 compares the RPC mechanisms used in contemporary OSes:

6-3: Comparison of RPC mechanisms in common operating systems

OS	Mechanism	Scope	Directory	Preprocessor	Transport
UN*X	SunRPC	Local/Remote	portmapper	rpcgen	UDP/TCP
OS X/iOS	Mach	Local (Remote)	launchd (mach_init)	mig	Mach messages
Android	Binder	Local*	servicemanager	aidl	/dev/binder

As shown in the table, all RPC mechanisms have common denominators, specifically:

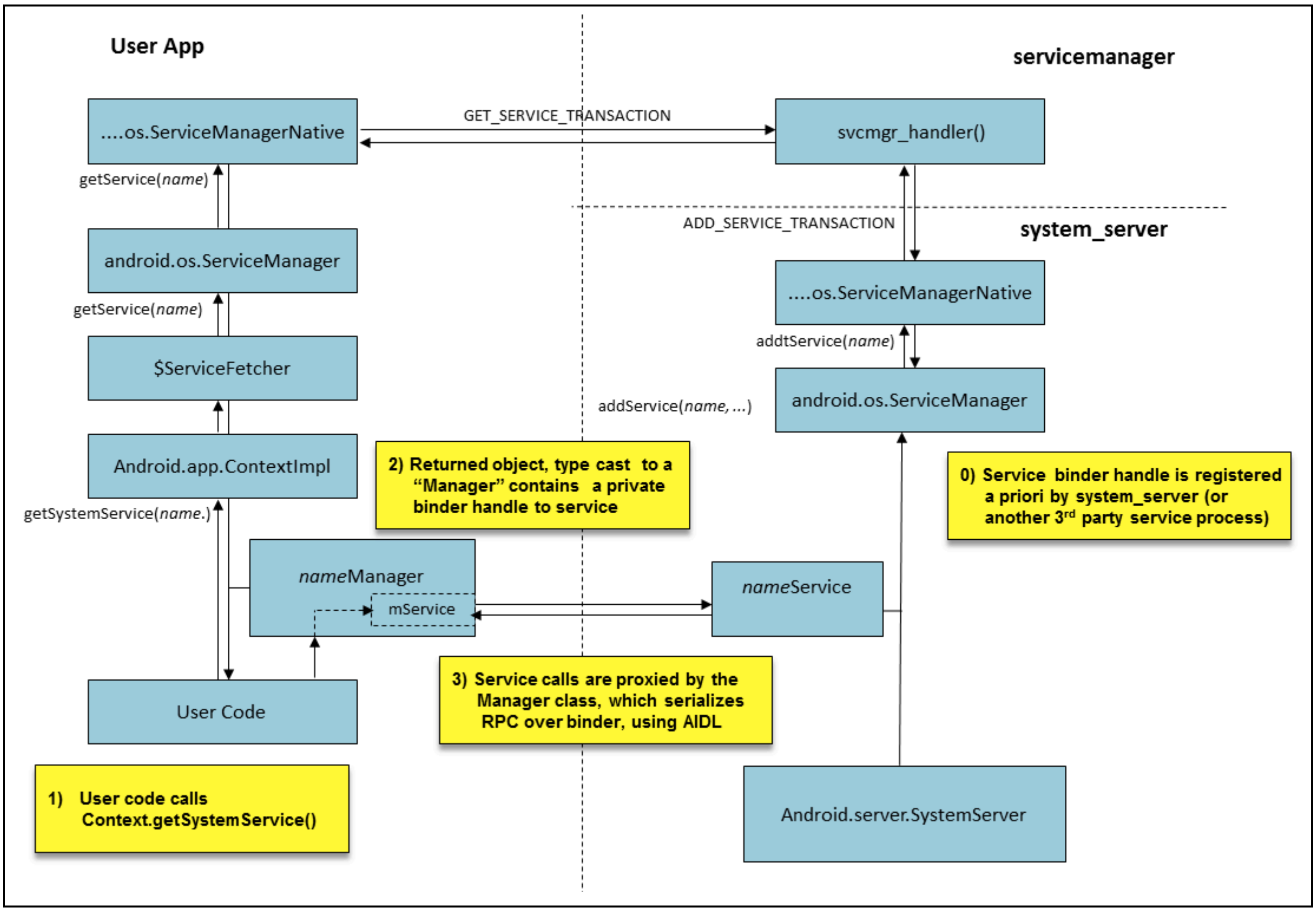
- scope: denoting whether the RPCs are used in between hosts (remote), or only on the local host
- Directory: The server providing the lookup functionality for locating services
- Preprocessor: The tool used to generate the serialization and deserialization code for messages
- Transport: The medium for message passing

We revisit RPC and discuss it in far more detail when dealing with Binder.

Android developers remain blissfully oblivious to the underlying implementation of service invocation. Instead, as most Android developers are familiar with, they are required to call on the `getSystemService()` method of the `Context` object, which accepts the name of an Android system service, and returns an opaque object. The object returned can then be type cast into the specific service object, and the service methods can be invoked through it.

Figure 6-2 shows the general pattern followed by most service method calls. The figure is somewhat simplified (for example, the system service handles are cached), but still presents the flow. Services are registered, a priori, by the server process (commonly, `system_server`, or a 3rd party process), through a call to `android.os.ServiceManager`. Recall this class provides a Java interface to the service manager.

Figure 6-2: Android system service call pattern


User App
servicemanager
....os.ServiceManagerNative
GET_SERVICE_TRANSACTION
svcmgr_handler()
getService(*name*)
ADD_SERVICE_TRANSACTION
system_server
android.os.ServiceManager
....os.ServiceManagerNative
getService(*name*)
addtService(*name*)
$ServiceFetcher
addService(*name*, ...)
android.os.ServiceManager
Android.app.ContextImpl
2) Returned object, type cast to a "Manager" contains a private binder handle to service
0) Service binder handle is registered a priori by system_server (or another 3rd party service process)
getSystemService(*name*.)
*name*Manager
mService
*name*Service
User Code
3) Service calls are proxied by the Manager class, which serializes RPC over binder, using AIDL
1) User code calls Context.getSystemService()
Android.server.SystemServer

Advantages and disadvantages

The system service architecture of Android follows a generic local client/server pattern, common to other OSes, such as iOS. Though iOS has no Binder, it uses its own implementation of a message passing architecture, called Mach messages. The role of `servicemanager` (i.e. the endpoint matter) is assumed by `iOS`'s `launchd` process, which (among other things) also handles the traditional PID 1 roles that Android's /init does.

A disadvantage which quickly stands out in this architecture is the overhead of IPC, particularly the need to serialize and deserialize messages, as well as the context switch required when alternating between the processes. This disadvantage does have a noticeable performance impact.

* - Android's Binder is, by design, limited to a local scope. It's a fairly simple hack to set up a local proxy process to further serialize and deserialize requests over a TCP or UDP socket, thus extending Binder's scope - a highly useful capability for a Remote Access Tool (RAT).

Given such a considerable disadvantage, it must be offset by advantages greater or equal in magnitude - and indeed, it is: Aside from the cleaner design and separation of privileges which follows, a client/server architecture gains security as a corollary. The client process - which is, by definition, an untrusted user app, is entirely devoid of any permissions, and therefore relies entirely on service calls to perform any operations. At the native level, this means that an app can be run sandboxed, without any access to devices and datastores, if any. Indeed, this is the case in iOS (wherein apps are "jailed"), though Android relies (for most processes) on filesystem permissions to deny access.

The server processes are trusted, and expected to perform all security checks, ensuring the client has the necessary permissions before agreeing to serve the request. Once again, the two arch rivals are similar here, with iOS relying on entitlements, (embedded in the binary's code signature), and Android on the application's Manifest file. In both cases, the permissions are declared outside of the application's runtime scope - i.e. they can be verified when installed (or, in iOS's case, when Apple vets the app), but cannot be modified by the App: Specifically, iOS's Entitlements are stored in kernel space (as part of the cached code signature blob), whereas Android's permissions are maintained by the `PackageManager`.

Serialization and the Android Interface Definition Language (AIDL)

In design pattern parlance, the object obtained from `getSystemService` serves as a **Proxy**: Internally, it holds a reference to the actual service, which it obtains over a Binder call. The methods exported by the object are, for the most part, merely stubs, which take their arguments, and serialize them into a Binder message, referred to as a `Parcel`. The methods and objects serializable in this way are specified using AIDL. AIDL isn't really a language, per se. It's essentially a derivative of Java which is understood by the `aidl` SDK utility, which is invoked in the build process when .aidl files are encountered. The `aidl` automatically generates the Java source code required to serialize any parameters into a Binder message, and extract the return value from it. The code is "boilerplate" - i.e. it can be automatically generated from the definition files and is guaranteed to compile cleanly. A sample .aidl file is shown in Listing 6-2:

Listing 6-2: A sample .aidl file

```
package com.NewAndroidBook.example;  // Creates java directory structure
import  com.NewAndroidBook.whatever; // Dependencies, if any

interface ISample {

 // Published interface - will be shown as com.NewAndroidBook.example.ISample
 // The numbers are the ones used when serializing (and using service call)

 /* 1 */ void    someFunc   (int    someArg); // no return value, integer argument
 /* 2 */ boolean anotherFunc(String someArg); // returns boolean, string argument

 // ... etc.. etc..

}
```

As you can see, an .aidl is somewhat similar to a header file, in that it defines methods (and possibly objects), but not their implementation. As we explore the individual framework services later in the book, you'll be able to see many more examples of actual .aidls from the AOSP.

The `aidl` tool does a marvelous job of hiding the implementation details of Android's IPC from the developers. So great a job, in fact, that most developers remain blissfully ignorant of the role of Binder, or its very existence. This work, however, recognizes the role of Binder, providing an introduction to it later in this chapter, and discussing internals in Volume II.

Power users can remain equally oblivious to Binder, especially with a powerful tool like the `service` utility, which enables the invocation of Android service methods right from the command line. This is shown in the following experiment.

Experiment: Using the `service` command to call services

A previous experiment demonstrated the basic usage of the `service` command line utility, as a method of interfacing with the `servicemanager` process. The true power of `service`, however, lies in its ability to call the services themselves.

Calling a service is a simple enough matter - using `service call`, and specifying the service name and method number: Internally, methods are assigned numbers in order of their appearance in the service's .aidl file. Depending on the method, optional arguments may be supplied. The `service` utility supports two types of arguments: `i32`, which are integer values, and `s16`, which are used for unicode strings. In practice, however, integers can be used for any 32-bit value (e.g. `float`), and strings - being unicode - can be used to serialize any object.

Any service retrieved by `service list` (Output 6-1) with an interface (specified in brackets) can be called on in this manner. Each interface has a corresponding .aidl file in the AOSP, wherein its methods and their arguments are clearly defined. Once you have the definitions, you can invoke any method of your choice, by figuring out its call number and passing the appropriate arguments. A few of the interesting ones are shown in Table 6-4:

Table 6-4: `service call` commands

service call...	Interface	Method	Action
phone 2 s16 "foo" s16 "555-1234"	`ITelephony`	call(String callingPackage, String number);	Place a call to the specified number.
statusbar 1	`IStatusBarService`	expandNotificationsPanel()	Brings up notifications
statusbar 9		expandSettingsPanel()	Brings up settings
statusbar 2		collapsePanels()	Hides all panels
dream 1	`IDreamManager`	dream()	Screensaver (if configured)
power 10 (< 4.4.1) power 11 (> 4.4.2)	`IPowerManager`	isScreenOn()	Returns 0 if screen is off, else 1

The low level call numbers assigned to methods can change between Android versions - even within the same API version (For example, `IDisplayManager` and `IPowerManager` within KitKat). It's rare, but could happen. Beware. In general, it's a bad idea to rely on hard coded numbers - if creating a tool or app to use these private APIs, compile them alongside the updated .aidl files

Invoking calls in this way will return a result in a Parcel (the Binder term for a message). Each parcel contains, at a minimum, a 32-bit return value (0x00000000 indicating success, otherwise some error value, commonly `0xffffffff` or `0xffffffb6 ("not a data message")` if a call number is outside the defined range). Depending on the AIDL definition, what follows is either an integer value (i32), or a length specification, followed by an opaque object (usually, but not necessarily, a string). Because `service`, like Binder, has no idea of what the opqaue object is, it will display the result in a manner not unlike the `od` command, with a hex dump of the message contents, alongside an ASCII dump of it.

Experiment: Using the `service` command to call services (cont.)

Only services with a published interface (specified in [brackets]) can be invoked. Note, not all services will blindly lend themselves to this type of invocation: Depending on the security policy, which is implemented differently by individual services, your service call request may be denied. If that is the case, the output of `service call` will contain a unicode error message, like so:

Output 6-2: Error messages returned from `service call`

```
# Attempt to call cancelMissedCallsNotification(), which requires MODIFY_PHONE_STATE
# (You can get past this, as well as most other permission checks, by running as root)
#
shell@htc_m8wl:/ $ service call phone 13
Result: Parcel(
  0x00000000: ffffffff 00000050 0065004e 00740069 '....P...N.e.i.t.'
  0x00000010: 00650068 00200072 00730075 00720065 'h.e.r. .u.s.e.r.'
  0x00000020: 00320020 00300030 00200030 006f006e ' .2.0.0.0. .n.o.'
  0x00000030: 00200072 00750063 00720072 006e0065 'r. .c.u.r.r.e.n.'
  0x00000040: 00200074 00720070 0063006f 00730065 't. .p.r.o.c.e.s.'
  0x00000050: 00200073 00610068 00200073 006e0061 's. .h.a.s. .a.n.'
  0x00000060: 00720064 0069006f 002e0064 00650070 'd.r.o.i.d...p.e.'
  0x00000070: 006d0072 00730069 00690073 006e006f 'r.m.i.s.s.i.o.n.'
  0x00000080: 004d002e 0044004f 00460049 005f0059 '..M.O.D.I.F.Y._.'
  0x00000090: 00480050 004e004f 005f0045 00540053 'P.H.O.N.E._.S.T.'
  0x000000a0: 00540041 002e0045 00000000          'A.T.E.......
```

Once you get past permissions, however, (for example, by running as root), the possibilities of using `service call` in this manner are nearly endless, spanning all the features and capabilities of the Android frameworks. As we cover the framework services in this work one by one, we'll be showing their respective AIDL definitions, and number the calls accordingly.

The Binder

The discussion so far has mentioned the Binder several times, but kept it a very high level overview. Indeed, at a high level, suffice it to consider the Binder as a special type of a file descriptor, which - through a dedicated kernel driver - is connected to the service. This is also how Linux sees it, when the process is viewed through the `/proc/`*pid*`/fd` directory. Virtually every process in the system (With the exception of a few native processes) opens a handle to `/dev/binder`.

Much of Binder's inner workings, however, are shrouded in darkness - probably because, for most developers, ignorance is bliss. For those who want to know the details, there is, after all, always the source. For the scope of this work, however, it's beneficial to elucidate some of these dark corners and provide a closer view of Binder, explaining its functionality without going into the (not so well documented) source.

A little history

The Android Binder mechanism traces its root back to the Binder of another mobile operating system, BeOS. Binder served as the underlying support interconnecting BeOS's rich set of frameworks. Once heralded as the "next generation operating system", BeOS never gained much traction save for a few fans, and was eventually acquired by Palm. If the name doesn't ring a bell, that's fine - Palm Pilots were all the rage back at the end of the last millenium, catapulting 3COM to great heights before Palm was split off and spiraled back to earth. Palm was eventually acquired by HP, and its OS served as the basis for "WebOS", another venture that fell far short of its promise.

Binder, however, survived. Besides being ported to PalmOS (and integrated into their Cobalt architecture), it was also ported to other operating systems - including, of course, Linux. The Linux port was open sourced (at http://openbinder.org/, and though the website seems to have died since, some mirrors[1] survived). The original developers left Palm to join Android, and brought Binder with them. Chief amongst them was Dianne Hackborn, a well renowned developer and still one of the major figures driving Android today. An interview she gave to OSNews[2] back in 2006< explained the fundamentals of OpenBinder.

Android's implementation of Binder is more specific than OpenBinder, and - just like as originally intended in BeOS - serves as the fulcrum for all of its frameworks.

So, what, exactly, is Binder?

Binder is a Remote Procedure Call mechanism, allowing applications to communicate programmatically, but without having to worry about how to send and receive messages. From the application's perspective - server or client - all it needs to do is either call a method (client) or provide a method (service). When the client calls the method, the corresponding method is magically invoked in the service, with all the "details" handled transparently by Binder. These "minutiae" include:

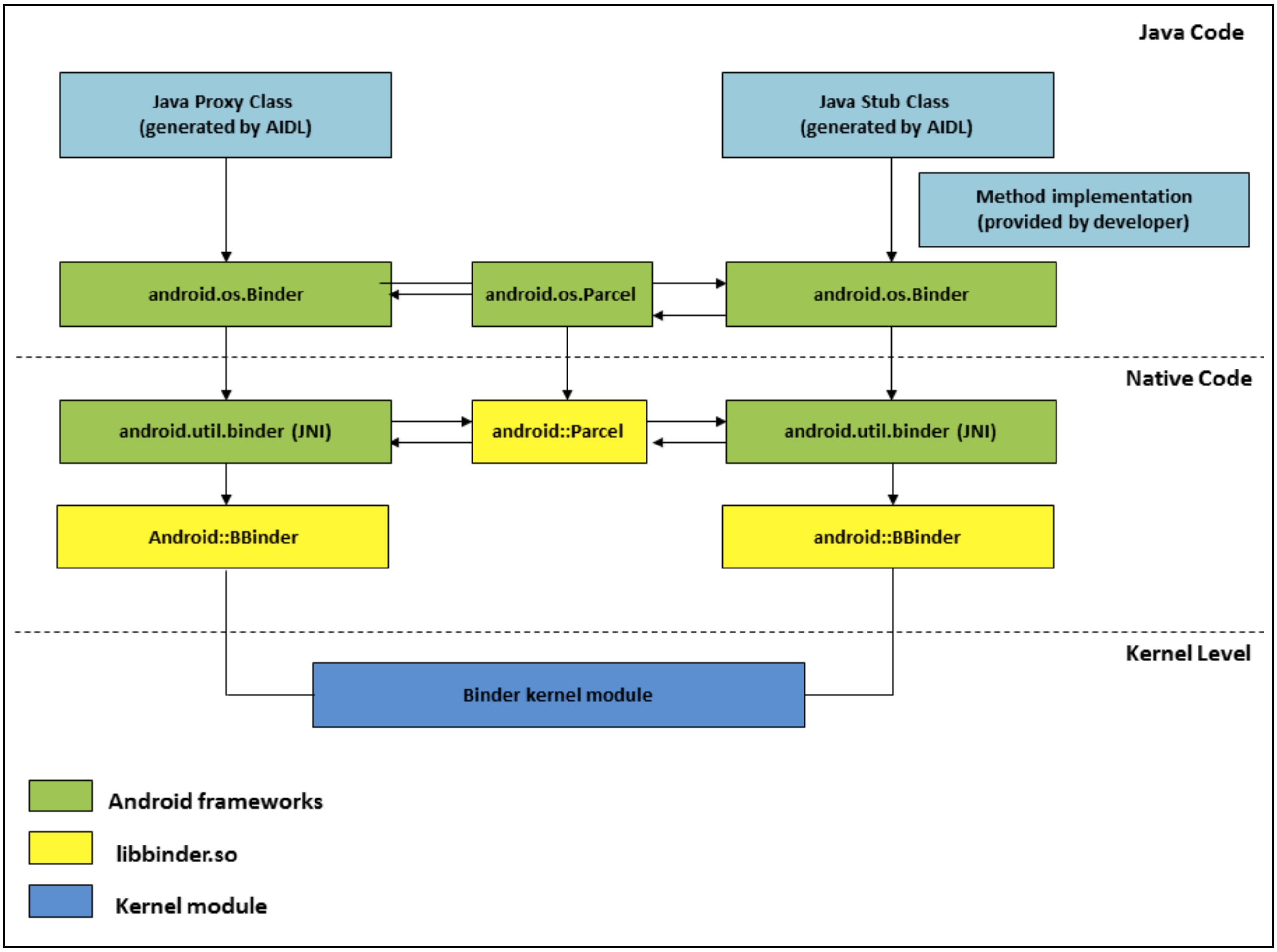
- **Locating the service process:** In most cases, the client and the service are two different processes (`system_server` notwithstanding). The Binder needs to locate the service process for the client, so as to be able to deliver the message. This "location service" (also known as "endpoint mapping") is technically handled by `servicemanager`, as explained previously, but the `servicemanager` is only responsible for maintaing the service directory, mapping an interface name to a Binder handle. The "handle" is an opaque identifier, which was given to the `servicemanager` by Binder, and which only Binder knows the "true" meaning of - that is, the underlying PID wherein the service is located.
- **Delivering the message:** As discussed previously, AIDL is used to generate the code which takes the parameters of the called method and serializes them (i.e. packs them into a structure in memory), or deserializes them (unpacks the structure back to individual parameters). The passing of the serialized structure from one process to another, however, is handled by Binder itself. Clients call the `BINDER_WRITE_READ ioctl(2)`, which sends the message over Binder, and blocks until a reply is returned (hence, the code - first write, then read).

- **Delivering objects:** Binder can be used to pass around objects - the service handles mentioned previously are one such type of an object, but so are **file descriptors** (just like UNIX Domain sockets). Passing around descriptors is an especially important feature, as it allows a trusted process (such as `system_server`) to natively open a device or socket for an untrusted process (such as a user app) - assuming the untrusted process has the required permission (as specified in the App's manifest).
- **Supporting credentials:** Inter process communication naturally has significant security aspects. A recipient of a message has to be able to verify the identity of the sender, so as not to be tricked into compromising overall system security. Binder is aware of its users' credentials - PID and UID - and securely embeds them in messages, so peers can operate with a reasonable level of security.

Using Binder

Binder is used in all applications, whether or not the developers themselves realize it. The code involved in binder operates on no less than three levels, as shown in Figure 6-3:

Figure 6-3: Message flow between client and server using Binder



In an effort to be true to the power user's view adopted in this work, Figure 6-3 is as far as we go - for now. More detail on the various levels - from the Java objects, through AIDL, native, and kernel - can be found in Volume II.

Tracing Binder

The /dev/binder connection multiplexes any number of service connections over the same file descriptor. This means that a process will hold that descriptor irrespective of whether it is connected to one service, or to many. Indeed, a process can hold this descriptor and not be connected (yet) to any services at all.

It follows, then, that there's no simple way to see exactly which services a given handle is connected to. If the Binder debug functionality is enabled through the Linux debugfs filesystem (/sys/kernel/debug/binder), however, you can use the `bindump` tool (on the book's companion website) to figure out who's connected to what, as shown in the following experiment:

Experiment: Using the `bindump` tool to view open binder handles

The `bindump` tool, which you can find on the Book's companion website is nothing more than a simple derivative of the `service` command, which obtains a handle to the system service of choice (as does `service check`), and then inspects its own entry in the /sys/kernel/debug/binder/proc directory. Each process using binder has a pseudo-file containing various statistics, and the `node` entries contained therein reveal the PIDs connected on the other end. Because all the binder debug data is world readable, you can run this tool on unrooted devices as well.

Output 6-3: Revealing binder endpoints using the `bindump` utility

```
#
# Inquire about wallpaper service
shell@htc_m8wl:/ $ /data/local/tmp/bindump wallpaper
Service: wallpaper node ref: 2034
User:  PID  1377        com.htc.launcher
User:  PID  1194        com.android.systemui
Owner: PID  1008        system_server
User:  PID   368        /system/bin/servicemanager
#
# Who owns the batterypropreg service?
shell@htc_m8wl:/ $ /data/local/tmp/bindump owner batterypropreg
Service: batterypropreg node ref: 105785
Owner: PID  8153        /sbin/healthd
```

The book's companion website also provides a special version of `strace()`, the Linux system call tracing tool, with augmented functionality that includes parsing of Binder messages (i.e. deciphering `ioctl(2)` codes and payloads).

system_server

Android devices have dozens of services, and along with vendor and user-installed apps, this number can exceed one hundred. Fortunately, the vast majority of framework services are simple enough that they do not require their own process, and can instead run as threads. These threads, however, need a host process to run in - and that is exactly what system_server provides.

Similar to Windows' svchost.exe, the `system_server` provides nothing more than a shell - a container process. The two can also be compared in the sense that svchost.exe loads services through dynamically linked libraries (DLLs), whereas `system_server` loads Java classes. In Android, however, this is even more important a function: Though the Dalvik VM is optimized for sharing, running services alongside one another in the same VM provides an even greater savings in resources. This does not come without a bit of risk, however, as a misbehaving service can thus affect its siblings. For the most part, though, this isn't much of a concern, as only Android's system services, and not those of the vendor or additional apps, are allowed to run inside system_server.

The `system_server` is not a native app: It is implemented mostly in Java, with some JNI calls in places where it must start native services. The services it loads are similarly implemented in Java, though a great deal of them also rely on JNI to escape the virtual machine and interact with hardware components. Zygote automatically starts the system_server when it itself is started by the /init.rc (q.v. Figure 5-22) with the `--start-system-server` switch. The switch makes Zygote invoke `startSystemServer()`, in which are hardcoded the arguments - capabilities, group memberships (`--setgroups`), the "nice name" (system_server), and the class to load - `com.android.server.SystemServer`. The `system_server` **does not** execute with root privileges, but comes pretty close - uid:gid of `system:system`, enhanced capabilities, and a host of secondary group memberships. The security perspective of `system_server` - GIDs and capabilties - is shown in Chapter 8.

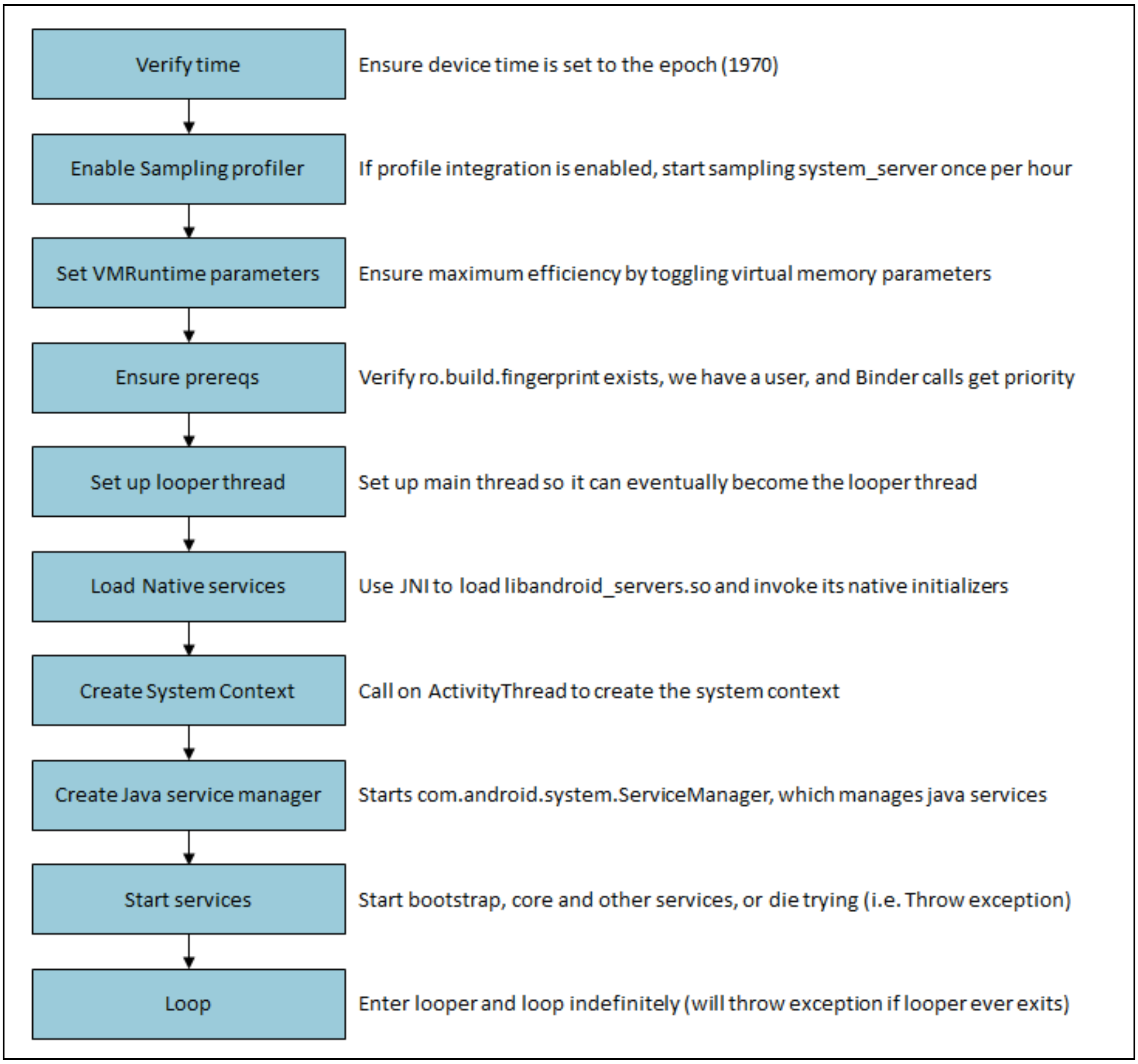
Startup and Flow

For such an important fulcrum of the entire system, system_server has a rather simple flow. Once it has forked off from Zygote, the child process drops its privileges, and toggles the capabilities as discussed above. It then proceeds to load the class, whose `main()` performs basic initialization (notably lifting its VM limits and loading the `libandroid_servers.so` to perform JNI component initialization), before instantiating the framework services. Once all services have been created (and their corresponding threads spun), with nothing else to do the main thread enters a looper, to loop (hopefully) endlessly (unless the system is shut down). The high level flow is shown in Figure 6-4, on the next page.

There are numerous system services to start, however, and `system_server` needs to instantiate them one by one. Android L takes great steps in refactoring this flow. Even though much work remains, the flow is significantly simplified from previous version by grouping services of similar classification. There are currently three "classes":

- **Bootstrap services**: These include the `Installer`, `ActivityManagerService`, `PowerManagerService`, `DisplayManagerService`, `PackageManagerService` and `UserManagerService`. Additionally, a check is performed if the device's /data partition is encrypted or in the process of encryption - which affects startup by starting only apps designated as "core apps".
- **Core services**: These include the `LightsService`, `BatteryService`, `UsageStatsService`, and the `WebViewUpdateService`. The last is a new service in L which periodically checks the browser component for any updates.
- **"Other" services**: basically, everything else. There are dozens of services in this class (which the source admits is "a miscellaneous grab bag of stuff that has yet to be refactored and organized").

Figure 6-4: The flow of system server


Verify time
Ensure device time is set to the epoch (1970)
Enable Sampling profiler
If profile integration is enabled, start sampling system_server once per hour
Set VMRuntime parameters
Ensure maximum efficiency by toggling virtual memory parameters
Ensure prereqs
Verify ro.build.fingerprint exists, we have a user, and Binder calls get priority
Set up looper thread
Set up main thread so it can eventually become the looper thread
Load Native services
Use JNI to load libandroid_servers.so and invoke its native initializers
Create System Context
Call on ActivityThread to create the system context
Create Java service manager
Starts com.android.system.ServiceManager, which manages java services
Start services
Start bootstrap, core and other services, or die trying (i.e. Throw exception)
Loop
Enter looper and loop indefinitely (will throw exception if looper ever exits)

Not all the services are visible to applications: Some, like the `Installer` are internal, and thus invisible both to apps as well as `service list`. We discuss all the services - internal and app facing - one by one in the next chapters.

Once the services are started, `SystemServer` has nothing more to do in its main thread. The thread therefore enters its looper, which hopefully loops indefinitely. We say "hopefully", since the looper is not expected to exit, and will throw a runtime exception if it does. Internally, the loop blocks, polling its file descriptors (and in particular, its Binder handle) for incoming messages. When messages arrive, they are dispatched to their respective targets.

Modifying startup behavior

The flow of `system_server` and the classes of services it starts can be modified by setting certain system properties.

A key parameter is the `ro.factorytest` system property, which defines whether or not the device is configured for a "factory test" mode, affecting the startup of `SystemServer` according to the following values:

Table 6-5: Factory test values and their impact on startup

value	#define	Implies
0 (default)	FACTORY_TEST_NONE	Normal startup.
1	FACTORY_TEST_LOW_LEVEL	No bluetooth, input, accessibility, lock settings
2	FACTORY_TEST_HIGH_LEVEL	Uid 0 for factory test applications

Another important parameter is the `ro.headless` system property, which - if set - disables the WallPaper service, and the System UI services. The `config` family of properties can also be used to selectively disable subsystems, as shown in Table 6-6:

Table 6-6: `config` properties affecting system services

`config` Property	Disables
disable_storage	MountService
disable_media	AudioService,WiredAccessoryManager, CommonTimeManagementService
disable_bluetooth	BluetoothManagerService
disable_telephony	Unused
disable_location	LocationManagerService, CountryDetectorService
disable_systemui	StatusBarManagerService
disable_noncore	UpdateLockService, LockSettingsService, TextServicesManager, SearchManagerService, WallpaperManagerService, DockObserver, UsbService
disable_network	NetworkStatsService,NetworkPolicyManagerService,WifiP2pService, WifiService,ConnectivityService, NsdService,NetworkTimeUpdateService,CertBlacklister

Thanks to the Linux `/proc` filesystem, you can examine system_server and its many threads. Looking at its file descriptors is somewhat futile - it's impossible to tell which descriptors belong to which thread - and most of them are sockets and pipes anyways. Enumerating the threads, however, can be useful. This is shown in the following experiment.

Experiment: Unraveling the threads of system_server

Dalvik's thread objects may be named when created. Naming a thread calls the underlying `prctl(2)` system call - a little known but highly useful API which allows the renaming of threads and processes at the kernel level. The name is then visible through the /proc filesystem in the status proc entry of the thread. The method is not perfect, as it allows for only 16 characters in a name - but it sure beats rummaging through random thread identifiers, trying to figure out which does what.

Using a basic script (which even Android's limited shell supports) you can easily enumerate the threads, and get their individual names (this works on any process, so as long as the `for` iterates over its task/ subdirectory, which contains a directory entry for each thread). Binder threads and thread pools are omitted from this output:

Output 6-4: Iterating through threads

```
root@flounder:/proc/507/task # for t in *; do echo Thread $t  `grep Name: $t/status`; done
507: Name:  system_server      # The main thread (same as PID)
512: Name:  Heap thread poo    # L: ART Heap thread pool
514: Name:  Signal Catcher     # Dalvik signal catcher
515: Name:  ReferenceQueueD    # Dalvik Reference Queue Daemon
516: Name:  FinalizerDaemon    # Dalvik object finalizer
517: Name:  FinalizerWatchd    # Dalvik finalizer watchdog
518: Name:  HeapTrimmerDaem    # L: ART Heap Trimmer Daemon
519: Name:  GCDaemon           # Garbage Collector (L: "GCDaemon", for ART)
524: Name:  SensorService
525: Name:  SensorEventAckR
526: Name:  android.bg
527: Name:  ActivityManager
529: Name:  FileObserver       # FileObserver$Thread
530: Name:  android.fg
531: Name:  android.ui
532: Name:  android.io
533: Name:  android.display
534: Name:  CpuTracker         # Created by ActivityManager
535: Name:  PowerManagerSer    # Created by PowerManagerService
537: Name:  BatteryStats_wa
562: Name:  PackageManager     # Created by
594: Name:  PackageInstalle    #     PackageManage
596: Name:  AlarmManager       # Created by AlarmManagerService
597: Name:  InputDispatcher    # Started by InputManager
598: Name:  InputReader        # Started by InputManager
599: Name:  MountService       # Created by MountService
600: Name:  VoldConnector      # Created by MountService
602: Name:  NetdConnector      # Created by ConnectivityManager
603: Name:  NetworkStats
604: Name:  NetworkPolicy
605: Name:  WifiP2pService
606: Name:  WifiStateMachin
607: Name:  WifiService
608: Name:  ConnectivitySer    # Created by ConnectivityManager
609: Name:  NsdService         # Neighbor Services Discovery (State Machine Thread)
610: Name:  mDnsConnector      # Created by NsdService
611: Name:  ranker             # Created by NotificationManagerService
613: Name:  AudioService       # Created by AudioService$AudioSystemThread
622: Name:  UEventObserver     # Kernel uevent observer (shared by many services)
623: Name:  backup             # Created by BackupManagerService
626: Name:  WifiWatchdogSta
627: Name:  WifiManager
628: Name:  WifiScanningSer
629: Name:  WifiRttService
630: Name:  EthernetService
634: Name:  LazyTaskWriterT    # ActivityManager's TaskPersister
635: Name:  UsbService host
844: Name:  watchdog
845: Name:  SoundPool          # AudioService$SoundPoolListenerThread
846: Name:  SoundPoolThread    # AudioService$SoundPoolListenerThread
906: Name:  NetworkTimeUpda    # NetworkTimeUpdateService's HandlerThread
984: Name:  IPC Thread
1009: Name:  WifiMonitor
1507: Name:  SyncHandler-0
1513: Name:  UsbDebuggingMan
```

TIDs aren't normally predictable, but a large part of system_server's are started incrementally, and so looking at the IDs can give you a sense as to the system's framework startup.

Summary

This chapter discussed the Android framework service architecture, explaining the underlying mechanisms of Inter Process Communication (IPC) through Remote Procedure Call (RPC) in Android, focusing on the role of the `servicemanager` and the `service` utility. It then focused on the `system_server` process, which serves as a host to Android's myriad frameworks, all implemented in Java.

This naturally begs much more discussion - specifically, of the dozens of services, and of Binder - the transport that facilitates RPC. This discussion, however, is left for Volume II.

Files discussed in this chapter

Component	File	Contains
ServiceManager	f/native/cmds/servicemanager/service_manager.c	Body of service manager
	frameworks/native/cmds/servicemanager/binder.[ch]	Binder interface
SystemServer	f/b/services/java/com/android/server/SystemServer.java	The `SystemServer` class

References

1. OpenBinder documentation (mirror): http://www.angryredplanet.com/~hackbod/openbinder/docs/html/
2. OSNews, Interview with Dianne Hackborn: http://www.osnews.com/story/13674/

VII: Android Through a Linux Lens

Android developers are accustomed to thinking about their applications in terms of the Android lifecycles described in the previous chapter. From the Linux perspective, however, Android applications are **Linux processes**, and aren't much different from any other process on the system.

This chapter puts that very perspective in focus. We first discuss the facilities Linux provides for process monitoring and tracing, through the /proc filesystem, which was touched on briefly in Chapter 2, but is now explored in the detail it deserves. We discuss the per-process and per-thread entries in /proc/*pid* which allow you to poll for real-time, on-the-fly statistics. First, we discuss the symbolic links that report working directories. We next focus on the highly useful fd/ and fdinfo/ subdirectories, which provides accurate representations of open file descriptors. Next up is the status entry, which gives a high level view of the process - and in particular, thread state and virtual memory.

Virtual memory is an important metric for diagnosing performance, so the next section focuses on the basics of user memory management, going into theoretical concepts, but also presenting the smaps proc entry, and two tools - procrank and librank - which you can use to get accurate memory statistics. We then explain the dreaded Out-Of-Memory condition, the bane of Android's application lifecycle, forcing the app to live in the shadow of ever-looming, and very unpredictable death.

Lastly, we explain system calls, showing the `toolbox ps` tool, the proc entries of wchan and syscall, and the all-powerful strace tool, which you can use for active tracing.

The chapter relies heavily on concepts from operating system theory, and is full of hands-on experiments meant to further elucidate these concepts, which are far from trivial. The methods and experiments shown in this chapter are all based on Linux kernel features, which makes them just as applicable on a Linux system, as they are in Android - so you might consider referring to this chapter for Linux Debugging tips, a subject on which there is a surprising dearth of books on.

/proc, revisited

The /proc filesystem was touched on in Chapter 2. The touch was hardly a graze, however, as it has not begun to scratch the surface of this extremely important filesystem. In particular, the per-process (and thread) directories in /proc, with their plethora of real-time diagnostic information about the inner workings of applications.

To understand the per process directories, think of a process as in object-oriented terms: A process can be thought of as an instance of a process *class*, all instances of which have the same *properties* - though naturally property values may differ. The pseudo-files in the per-process directory simply show you the values of the properties, and - in some cases, if they are writable - allow you to modify these properties.

Output 7-1 shows the entries the Android shell would see in its own directory (using `$$` as the process ID of the current shell - note we don't use /proc/self, because `ls` would see itself!). The `cut` filter is optional, and is used only to improve readability.

Output 7-1: Annotated per-process entries in L (3.10 kernel).

```
shell@flounder: /data$ ls -l /proc/$$ | cut -c1-10,55-
dr-xr-xr-x attr                  # Attr
-r-------- auxv                  # ELF AUXilliary Vectors, in binary form
-r--r--r-- cgroup                # Control Group membership of process
--w------- clear_refs            # Clear page referenced bits in smaps
-r--r--r-- cmdline               # Command line (argv[]), NUL separated
-rw-r--r-- comm                  # 16-byte argv[0]
lrwxrwxrwx cwd -> /data          # Process current working directory, as symlink
-r-------- environ               # Environment (as per set or getenv()), NUL separated
lrwxrwxrwx exe -> /system/bin/sh # Full path of executable, as symlink,
dr-x------ fd                    # Directory containing open file descriptors, as symlinks
dr-x------ fdinfo                # Directory containing metadata on open file descriptors
-r--r--r-- limits                # Process hard/soft limits, as per ulimit or setrlimit(2)
-r--r--r-- maps                  # Process address space memory map
-rw------- mem                   # Process virtual memory, as a pseudo-file
-r--r--r-- mountinfo             # Mounted file systems, as process sees them (mount namespace)
-r--r--r-- mounts                # Mounted file systems, slightly different format
-r-------- mountstats            # Mounted file systems, slightly different format
dr-xr-xr-x net                   # Network statistics of process network namespace
dr-x--x--x ns                    # Namespaces of this process, as symlinks (commonly, mnt)
-rw-r--r-- oom_adj               # Out-Of-Memory (old) score adjustment. Used by ActivityManager
-r--r--r-- oom_score             # Out-Of-Memory score. Determines process killability on OOM
-rw-r--r-- oom_score_adj         # Out-Of-Memory (new) socre adjustment. Used by ActivityManager
-r--r--r-- pagemap               # Page map of process
-r--r--r-- personality           # OS Personality. Commonly, this is 0000000 (Linux)
lrwxrwxrwx root -> /             # Process root directory. Always symlinks to /, unless chroot(2)ed
-rw-r--r-- sched                 # CFS statistics for this proecess, in human-readable form
-r--r--r-- schedstat             # More CFS statistics for this process
-r--r--r-- smaps                 # q.v maps, but with more detailed information per memory region
-r--r--r-- stack                 # kernel stack of process (technically, main thread)
-r--r--r-- stat                  # Statistics (from task_struct), in machine-readable form
-r--r--r-- statm                 # More statistics, in machine-readable form
-r--r--r-- status                # Statistics (from task_struct), in human-readable form
-r--r--r-- syscall               # Current system call and arguments
dr-xr-xr-x task                  # Subdirectory containing entries for process threads
-r--r--r-- wchan                 # Wait Channel in kernel
```

Remember - *none of these are actually files*. This means two things:

- The exact listing of the file may change, according to your kernel version. In general, the newer the kernel, the more likely you are to have more pseudofiles, though support for some of them may be disabled when compiling the kernel.

- **The files aren't really there, until you ask for them:** which means that every time you display the files, you're likely to get different content. When using `ls`, the kernel doesn't even bother reporting file sizes, which is why (if you try the above command without the `cut` filter, all file sizes are shown as 0.

The last point is a very important one to consider: Because the files are purely virtual, there is no overhead in maintaining the /proc entries - the kernel maintains all these statistics anyway during normal operation. All it takes is "faking" the existence of these files, and - when the user asks for any - collecting the statistics in real time, and providing them in pseudo-file form. This makes the /proc filesystem an extremely powerful mechanism for system and process tracing, provided the method used is that of **polling**.

Tracing by polling means that the tracing program or script has to keep on explicitly asking for specific /proc entries periodically, because /proc entries do not support on-change callbacks (at least, not yet). This does have certain disadvantages - if the polling granularity is too coarse, you may end up missing the exact event you were trying to intercept. But the advantage - zero overhead - clearly outweighs the disadvantage. The human-readability and ease of parsing of the pseudofiles is another clear advantage, as we demonstrate in this chapter.

Because kernels change so frequently, this unfortunately has the side effect of leaving the documentation (`proc(5)` on Linux systems with `man` installed) somewhat outdated, and not all these are properly documented. We next turn our attention to some of the more important of these pseudo-files, which you can readily use when profiling or debugging the system.

The symlinks: cwd, exe, root

Looking at output 7-1, three entries immediately stand out - those of cwd, exe and root. The reason they are different is because they are shown as symbolic links, whereas other entries are shown as pseudo-files.

The rationale behind displaying these entries as symbolic links is readability. All three entries point to files or directories, and by using symbolic links, it makes it easier for the user to apply a file operation (e.g. `cat(1)`, `ls(1)`) on the target of the link (which most commands follow automatically), rather than have to first display the contents of a pseudo-file, then embed the output into the next command.

The three entries give you the most important high-level statistics for the process, namely:

- **cwd** - which displays the current working directory. In output 7-1, you could see from the prompt that the shell's present working directory is /data - and that is exactly what the cwd link is pointing to. A fun experiment is to `cd` to any directory of your choice, then repeat the `ls -l /proc/$$/cwd`. You will see that you can run, but youcan't hide - Any time you use the query the cwd entry, the kernel retrieves the working directory **at that moment**, which means you will always get the right directory.
- **exe** - which displays the full path to the executable used to start this process (that is, the one loaded by the `execve(2)` system call. This is useful because many processes can change their name, as displayed in `ps` during their lifetime, but they cannot change this entry.
- **root** - which displays the root directory. Normally, this will be the real root directory - (/). If an application is `chroot(2)`ed, that is, confined to a subdirectory which is defined as its new root, this will be clearly visible from this entry.

The cwd and root entries are used by tools such as fuser and lsof, which find open files and directories by patname (fuser) or by process (lsof). Knowing this, it becomes a simple matter to implement both these utilities as shell scripts, which could become quite useful on systems which do not have these tools pre-installed. The following experiment shows how a similar shell script trick can be used with the seemingly less useful exe entry.

Experiment: Determining the 32/64-bitness of Android apps

At first glance, an entry such as `exe` seems somewhat useless - after all, in most cases the executable name isn't really expected to change during the process lifetime.. or is it?

It turns out, that's not always the case. A good example in Android are the various Dalvik apps, spawns of Zygote, all changing their name to that of the loaded class they are executing. This is done by changing the value of argv[0], using the `prctl(PR_SET_NAME..)` system call. The real name of all these apps is still /system/bin/app_process, which is the "true" instance of the VM which loaded them. In L this is even more useful, because 32-bit apps will be clearly visible as app_process32, whereas 64-bit ones will be app_process64. The following example shows how you can use that to your advantage in a shell script:

Output 7-2: Using the `exe` proc entry to figure out 32/64-bitness of an app

```
root@flounder:/#  cd proc; for p in [0-9]*; do
> if ls -l $p/exe | grep app_process32 > /dev/null; then # isolate 32-bit, but ignore output
> echo `cat $p/cmdline` \(PID $p\) is a 32-bit app
> fi
> if ls -l $p/exe | grep app_process64 > /dev/null; then # isolate 64-bit, but ignore output
> echo `cat $p/cmdline` \(PID $p\) is a 64-bit app
> fi
> done
com.google.process.location (PID 10446) is a 64-bit app
com.google.android.inputmethod.latin (PID 10702) is a 64-bit app
... etc.. etc..
com.google.android.apps.maps (PID 14610) is a 32-bit app
com.google.android.talk (PID 15219) is a 32-bit app
zygote64 (PID 211) is a 64-bit app
zygote (PID 212) is a 32-bit app
system_server (PID 511) is a 64-bit app
com.android.systemui (PID 691) is a 64-bit app
com.android.server.telecom (PID 930) is a 64-bit app
com.android.phone (PID 988) is a 64-bit app
```

To understand the script better, note the *pattern* used:

1. **`cd` to the `/proc` directory:** Because everything starts here.
2. **Iterate over `[0-9]*` entries:** The root of /proc contains additional files, alongside the per-process entries. We want *just* the per-process directories, so we isolate only those entries beginning with a digit. This will run a loop with `$p` set to the PID iterator.
3. **Perform check by looking at the `exe` `/proc` entry**: Note the use of `grep`, with output discarded (`> /dev/null`). We're only interested in whether or not there was output - `grep`'s implicit return value, which is what the `if` will branch by. Because there are actually three cases here (`app_process32`, `app_process64` or neither), we don't use and `if/else` construct, but two separate `if` statements.
4. **Print out the user-facing output:**: Using `cmdline`, taking advantage of its NUL-separation, when employing `cat(1)`, only argv[0] is printed. This is better than the comm entry, since the latter is truncated at 16-bytes. Note also the use of `$p` - our iterator, which holds the PID (which shows the initial `cd(1)` was like Chekhov's Gun).

At first glance, the idea of running a script inline might deter some readers. That's understandable, especially when considering the rigid syntax of shell scripting (which is why this is an experiment - you're urged to try this at least once for yourself!). Remember, however, that every inline script like that can easily be put into a file, `chmod(2)`ed `+x` - thereby becoming a new tool for you to add to your arsenal. The exact same pattern - iterating and `grep(1)`ping - albeit with different /proc entries - can be adapted to create custom, reusable tools, which will work correctly both on Android and Linux systems.

fd

A process performs all of its I/O through **file descriptors**. The files, pipes and sockets opened - irrespective of language, Java C or other, all map to numbered file descriptors, with three default ones - standard input (`stdin`, `stdout` and standard error (`stderr`) - numbered 0, 1 and 2, respectively. When a process opens or creates a file (or socket, pipe, etc), the created object is linked to the next available descriptor. What appears to the process as a number is, in fact, a handle to an opaque object, that only has true meaning in the kernel, wherein that handle is deciphered as an index to an array of objects.

It is therefore of the utmost importance to be able to figure out which descriptors a process is using in real time. There are quite a few tools for that - most notably, `lsof` (**lis**t **o**pen **f**iles), which dumps open files per process along with other mappings. Rather than rely on `lsof`, however (which may or may not be present in a given distribution), it often makes sense to get the information straight from the horse's mouth - that is, from /proc/*pid*/fd.

The fd/ subdirectory follows the same symlink convention as the cwd, exe and root entries described in the previous section. This makes it extremely useful to just `ls -l` a given PID's fd/ directory in order to figure out which files are in use. So useful, in fact, that this work demonstrated the technique several times by now in previous chapters. It couldn't be simpler, but be aware that listing descriptors does require root privileges if you are not the owner of the process (and, by corollary, the /proc/*pid*/fd directory):

Output 7-3: Showing the file descriptors of a process (Zygote) through /proc/*pid*/fd:

```
root@flounder:/proc/151/fd # ls -l
lrwx------ root     root     2015-01-04 10:19 0 -> /dev/null
lrwx------ root     root     2015-01-04 10:19 1 -> /dev/null
lrwx------ root     root     2015-01-04 10:19 10 -> socket:[11955]
lrwx------ root     root     2015-01-04 10:19 11 -> socket:[6639]
lr-x------ root     root     2015-01-04 10:19 12 -> /dev/alarm
lrwx------ root     root     2015-01-04 10:19 13 -> socket:[11986]
lrwx------ root     root     2015-01-04 10:19 2 -> /dev/null
lrwx------ root     root     2015-01-04 10:19 3 -> socket:[10409]
l-wx------ root     root     2015-01-04 10:19 4 -> /sys/kernel/debug/tracing/trace_marker
lr-x------ root     root     2015-01-04 10:19 5 -> /system/framework/framework.jar
lr-x------ root     root     2015-01-04 10:19 6 -> /system/framework/core-libart.jar
lrwx------ root     root     2015-01-04 10:19 7 -> socket:[11323]
lr-x------ root     root     2015-01-04 10:19 8 -> /system/framework/framework-res.apk
lr-x------ root     root     2015-01-04 10:19 9 -> /dev/__properties__
```

For regular files, this works perfectly. The convention isn't as useful, however, when it gets to sockets. Since sockets have no filesystem representation (some UN*X sockets notwithstanding), there is nothing to symlink to. It would be trivial to add a fake symlink, which would contain a string of the IP or domain socket in question, but at the time of writing the Linux kernel opts instead to take the path of least resistance, and simply spit out the inode number associated with the socket. You can see the socket numbers above, for descriptors 7, 10, 11 and 13. But where do these sockets connect to?

Fortunately, there are other pseudofiles in procfs which will resolve this data for you. The following experiment shows how to figure out sockets - both UN*X and IP:

Experiment: Resolving inodes to socket names through /proc/net

It's always possible to "cheat" and use a tool like `lsof(1)` (but not the toolbox tool) to automatically resolve all descriptors, including sockets, for you. But with a little bit more knowledge of Linux procfs files, it's not that hard to do so on your own. The sockets in Linux and Android are usually one of the following types*:

- **UN*X domain sockets:** Used for local only communication. Some of these sockets are named, i.e. they have a filesystem representation. In practice, these are not really files - domain sockets are in-memory kernel constructs, and the filename is used to ensure system-wide uniqueness. In Android these sockets are located in /dev/socket, with an additional type of sockets using "@" naming conventions, which do not appear on the filesystem. Other sockets may remain unnamed. The kernel keeps the domain socket statistics in /proc/net/unix
- **IP based sockets:** Over IPv4 and/or IPv6. Linux (and Android) differentiates between the two address families, and further differentiates by protocol type - udp or tcp. As a consequence, there are thus no less than four files to consult - tcp6, udp6, tcp and udp.
- **Netlink sockets:** Used as an efficient kernel-user space notification mechanism. These are unique to Linux, and are also favored because of their multicast capabilities - i.e. it is possible to share a socket between members of a group, sending messages to all of them at once. Statistics are kept at /proc/net/netlink.

For IP-based sockets, a simple method is to look for the inode number in the various /proc/net** statistics files. Since there are four files, it's quicker to do so by using `grep(1)`, as shown in the following output:

Output 7-4: Figuring out IP sockets from /proc/net

```
shell@flounder:/ $ grep 471470 /proc/net/*
/proc/net/tcp6:  19: 0000000000000000FFFF00006E01000A:E0AE 0000000000000000FFFF00000E7BC2AD:01BB 01
                     00000000:00000000 00:00000000 00000000 10060    0 471450 1 0000000000000000
                     23 4 32 10 -1 com.google.android.apps.maps 14610 droid.apps.maps
```

Tools such as `busybox netstat` or `busybox lsof` (but not those of `toolbox` can parse the output - but if you do it manually, all it takes is a bit of hex-juggling: The format of the lines is:

```
##: Loc_v6/v4:Port Rem_v6/v4 state .. inode# .. pname pid comm
```

which gives you the details you need.

Mappings for UN*X domain sockets are unfortunately not always this easy. Sometimes using `grep` will yield the socket name from /proc/net/unix, but often times the socket is unnamed, which makes it difficult to figure out which peer is connected to it. In some cases, it's possible to recover the other end by trying one number lower or higher, which may still be an unnamed socket, but using `ls -lR /proc/[0-9]*/fd` and looking for it often reveals the other end point's holder. *This is not a fool-proof method*, because at times sockets are not created in pairs, but it's the simplest way of deducing the number in absence of kernel symbols and /proc/kcore.

* - There are less often encountered types, such as raw sockets, which naturally maintain statistics in other files.
** - Technically, it's more accurate to read /proc/*pid*/net/*family*, since sockets may be contained in a namespace. /proc/net offers the global namespace, however, so this works well too.

fdinfo

At first glance, the fdinfo directory looks unimpressive - just like fd, but without symbolic links (or fancy colors). The information contained in fdinfo, however, is just as important as fd/, if not more so.

For every open file descriptor, the corresponding fdinfo/## entry holds metadata about the file. Device drivers and filesystem implementors may use this file to convey information about the current state of the file descriptor back to user space, though in practice few do, leaving only the default information maintained by the kernel itself, specifically:

- **flags:** The flags used in the open(2) system call, when creating or opening the file. These are defined in the <fcntl.h> header file.
- **pos:** The current position of the "file pointer": i.e. the offset of the next byte to be read or written from the file.

The **pos** statistic in a real gem, because it can let you monitor a process and figure out roughly where along its timeline it presently is - all entirely *unobtrusively*. With a little bit of shell scripting, you can harness this functionality to create custom tools to conditionally operate on a process, as shown in the following listing:

Listing 7-1: A small script to watch and act on a file position in a process

```
PID=$1                     # PID is first argument
FD=$2                      # FD to watch is second argument
OFFSET=$3                  # OFFSET to monitor is third argument
CUT_COMMAND='busybox cut'  # needed because toolbox doesn't have cut built-in

# This isolates just the numerical offset from the fdinfo entry of $FD

#      Get the data              | isolate pos line   | isolate numerical value
CUROFF=`cat /proc/$PID/fdinfo/$FD |     grep pos       | $CUT_COMMAND -d':' -f2`

if [[ $CUROFF -gt $OFF ]]; then
  echo Do something
 # Insert command to execute on trigger here
else
  echo Nothing to do.
fi
```

The one drawback of the above script is that it relies on polling, rather than notification. Simply put, the results will potentially change in between executions, and - depending on when you choose to execute it - you might end up missing the precise offset you were looking for. This can be assuaged by running the script at regular intervals, and/or changing the OFFSET parameter to allow for more leeway (i.e. set the offset to a little bit before the actual required offset, and use conventional debugging from the point on).

The script shown above is really an example of a *pattern*, which can be used all over procfs to harvest data, and perform operations based on values collected. Because procfs exports its data as pseudo-files, it just takes knowing the right filename, and a mastery of the UN*X filters (cut, grep, sort and their ilk) to create any number of customized tools. In fact, most of the tools you probably know and love (or at least, respect?) can be implemented in script form by iterating over procfs, and the useful per-process/thread entries (especially stat or status, shown next).

status

The status proc entry is a one-stop shop for all the things you'd want to know at a high level on the process being inspected. And not just what you would like to know, so much as what the kernel would: The /status is effectively a human-readable dump of the `task_struct`, which is a mammoth structure in the Linux kernel serving as the process control block (PCB). This is what the kernel sees, at a glance, when handling a process:

Output 7-5: The annotated /proc/*pid*/status entry

```
shell@flounder:/proc $ cat /proc/511/status
Name:   system_server  # Same as /proc/511/comm
State:  S (sleeping)   # or R - running, T - Stopped, D - Uninterruptible (deep) sleep
Tgid:   511            # Thread Group id: The real process id
Pid:    511            # Thread, not process id
PPid:   211            # Parent Thread Group id
TracerPid:      0      # Any ptrace(2) attached process, like strace, gdb or debuggerd
Uid:    1000    1000    1000    1000  # Real, Effective, Set and File-System UIDs
Gid:    1000    1000    1000    1000  # Real, Effective, Set and File-System GIDs
FDSize: 2048                          # Maximum # of file descriptors allowed
Groups: 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1018 1032 3001 3002 3003 3006 3007
VmPeak:  2367448 kB      # Virtual Memory high-water mark
VmSize:  2258636 kB      # Virtual Memory size, present
VmLck:         0 kB      # Memory locked by mlock(2) APIs
VmPin:         0 kB      # Pinned memory
VmHWM:    178100 kB      # RSSPeak - i.e. Resident memory footprint high-water mark
VmRSS:    151600 kB      # Resident memory footprint, present
VmData:   230448 kB      # Size of data segment (heap memory)
VmStk:      8192 kB      # Size of process thread stacks
VmExe:        16 kB      # Size of executable
VmLib:    120016 kB      # Memory used by shared library (.so) files
VmPTE:       968 kB      # Memory used by Page Table Entries
VmSwap:        0 kB      # Memory used by process in swap (If no swap, always 0)
Threads:        82       # Number of threads. If > 1, this is a multi-threaded process
SigQ:   3/6826           # Handled Signals/Size of signal queue
SigPnd: 0000000000000000 # Bitmask of pending signals (for thread)
ShdPnd: 0000000000000000 # Shared pending signals for process
SigBlk: 0000000000001204 # Bitmask of signals blocked (by SIG_BLOCK)
SigIgn: 0000000000000000 # Bitmask of signals ignored (by SIG_IGN)
SigCgt: 00000002000094f8 # Bitmask of signals caught (by handlers)
CapInh: 0000000000000000 # Bitmask of inherited capabilities
CapPrm: 0000001007813c20 # Bitmask of permitted capabilities
CapEff: 0000001007813c20 # Bitmask of effective capabilities
CapBnd: 0000000000000000 # Bitmask of capabilities bounding set
Cpus_allowed:   3        # Bitmask of CPUs allowed (3 = 0011)
Cpus_allowed_list: 0-1   # List of CPUs allowed, for the hex-challenged
voluntary_ctxt_switches:        40684 # Voluntary (system call induced) context switches
nonvoluntary_ctxt_switches:     13471 # Nonvoluntary (preemption induced) context switches
```

There is copious output here, so it makes sense to go over the non-obvious fields step by step:

Sorting out the pid, tid, tgid, and ppid confusion

It's tempting to think that "pid" would stand for Process ID. Right? Well, tough. It doesn't. Originally, Linux did actually use PID as Process IDs, but ever since the turn of the millenium Linux joined other modern operating systems, in scheduling **threads** and not processes. As such, `Pid` correctly describes the **thread, and not the process id** of the entry being inspected. A process is, therefore, a group of threads sharing the same resources (virtual memory, file descriptors, etc), and that is what is shown by the `Tgid` field.

Some readers may first balk at this, especially when in the above example the `Tgid` fields and `Pid` fields hold the same value. There's no contradiction here, though: For the `main thread` of a process, the `Pid` and the `Tgid` values will always match. This is, in fact, how one can easily determine this is the main thread of the process - in other words, the first thread in the thread group. For child threads, however, **The Pid will change, while the Tgid will remain constant**.

By the same reasoning, the Ppid field - Parent Process Identifier - is therefore more correctly the Parent Thread Group IDentifier. Somehow, PTGID just doesn't sound as catchy, so it's still referred to as PPID. It's important to know one's lineage, because parents are responsible for collecting their children's return code (returned from `main()` or a call to `exit()`). Some parents also drag all their offspring when they die, killing the entire process group.

Most UN*X tools are "conditioned" to only show statistics for main threads, and so the notion (or illusion) or processes still works pretty well - and thus merits this section for explanation. The `ps` command, in an effort to maintain backward compatibility with days of yore, maintains the lie by calling `Tgid` as `PID`, and (when displaying threads, as in Linux's `ps -L`) will actually refer to the `Pid` as `LWP`. Android's `ps`, which displays threads with the `-t` switch, doesn't bother, and still calls the field `PID`. The threads in any thread group can be seen by looking at the task/ subdirectory, as shown in the following experiment:

Experiment: Viewing threads and processes in /proc

It often benefits an administrator or debugger to sift through the threads of multi-threaded processes. The /proc filesystem offers per-thread statistics. Unlike Linux `ps`, which offers the 'l' state to denote multi-threaded processes (in BSD mode), Android's `ps` tool only provides `-t` to list all threads. You can use the Android `top(1)` tool to display the number of threads:

Output 7-6: Using `toolbox top` to show number of threads:

```
shell@flounder:/$ top
User 0%, System 1%, IOW 0%, IRQ 0%
User 3 + Nice 0 + Sys 8 + Idle 609 + IOW 0 + IRQ 0 + SIRQ 0 = 620

  PID PR CPU% S  #THR     VSS     RSS PCY U        Name
19213  0   1% R     1   4088K   1340               top
    1  1   0% S     1   1004K    544               /init
..
  154  1   0% S    14  82016K  37568K  fg system   /system/bin/surfaceflinger
  198  0   0% S     5   4764K    320K     shell
  202  0   0% S     9  12872K   1852K     root
  203  0   0% S     1   1732K    732K     root
  204  1   0% S     1   3556K   1532K     root
  205  0   0% S     2  13932K   5460K  fg drm
  206  0   0% S    11 113916K  24788K  fg media
  211  0   0% S     6 2080384K  81204K     root
  212  1   0% S     6 1488620K  59788K     root
  511  1   0% S    82 2258636K 150116K  fg system   system_server
  691  0   0% S    25 2166460K 141436K  fg u0_a20   com.android.systemui
19129  1   0% S    29 2128380K  57220K  bg u0_a16   com.android.vending
..
```

You can extend the pattern of iterating over processes, from the previous experiment, to also iterate over a given thread group's threads. You can `cd` to /proc/*tgid*/task to find numbered subdirectories corresponding to all threads in the group (including the main one). If you then `cd` to the individual task/ subdirectories, you'll see they are similar to the main thread's entry (/proc/*tgid*/task/*tgid* in fact being the same). The per-process and per-thread entries are essentially the same (recall Linux sees threads, not processes), with nearly all process level attributes (maps, fd, etc) remaining the same, but a few (syscall, wchan, and a few others) potentially different per thread.

Experiment: Viewing threads and processes in /proc (cont.)

The status entry can be particularly confusing, because most of its entries apply to the thread group (and are thus identical across threads) whilst others do change on a per thread basis. The Tgid: for example, can indeed be corroborated to be the real process ID by the following:

Output 7-7: TGID vs. PID, hands-on

```
shell@flounder:/proc/211/task $ for t in *; do
> echo -n "PID $t: "; grep Tgid: $t/status;
> done
PID 19130: Tgid:        211
PID 19131: Tgid:        211
PID 19132: Tgid:        211
PID 19133: Tgid:        211
PID 19134: Tgid:        211
PID 211: Tgid:  211
```

Thanks to Android's best practice of naming individual threads, you can iterate over individual threads of most multi-threaded processes and actually tell them apart. This is especially useful for Dalvik apps (such as system server, q.v. Output 6-4), or even for Zygote itself - For example :

Output 7-8: Showing named threads

```
shell@flounder:/proc/211/task $ for t in *; do
> echo -n "PID $t: "; grep Name: $t/status;
> done
PID 19130: Name:        ReferenceQueueD
PID 19131: Name:        FinalizerDaemon
PID 19132: Name:        FinalizerWatchd
PID 19133: Name:        HeapTrimmerDaem
PID 19134: Name:        GCDaemon
PID 211: Name:  main
```

A little known fact is that you can cd directly into a thread. While listing /proc will only show main threads (or kernel threads), invoking cd with a valid TID will simply switch into the per-thread statistics, which are the same as what you would get through /proc/*tgid*/task/*tid*. When you perform the list, procfs gets picky and filters out child threads. When you cd directly, procfs doesn't care - if you specified a valid thread, child, main or kernel - you got it.

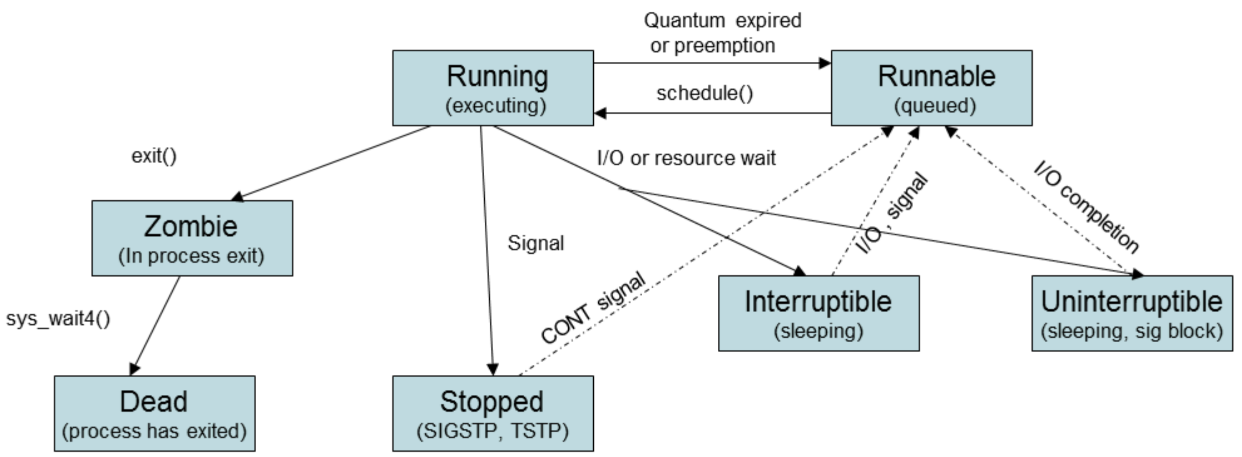
Thread states and context switches

While a thread would, optimally, want to always be running, more often than not it doesn't need to. Threads spend their lifecycle executing every now and then, but more often than not they are waiting. For an event, for user input, for I/O, maybe a mutex, or maybe just a chance to execute, because all CPUs or cores are presently occupied by other threads. At any given time, the kernel maintains the list of threads, and for each, it records the state.

The process itself is not a runnable entity. Thus, the state and context switching statistics you see in the per-process entry is in reality that of the main thread.

The `State:` field in /proc/status shows the same state shown in Android's `ps` tool (or Linux's `ps`, in BSD syntax). The states used are quite similar to states in other UN*X (such as Darwin and other BSD), and in fact not unlike those of all operating systems, including Windows (though the nomenclature is obviously different). The following state diagram depicts the transitions between states:

Figure 7-1: The Linux thread state machine



Kernel State Constant	ps	Meaning
TASK_RUNNING(0)	R	Process is in running state, or runnable
TASK_INTERRUPTIBLE(1)	S	Process is sleeping, may be interrupted (by signals)
TASK_UNINTERRUPTIBLE(2)	D	Process is sleeping, may NOT be interrupted
TASK_STOPPED(4)	T	Process stopped (as per SIG_STOP, or TSTP)
TASK_TRACED(8)	---	Process is being traced (single-step)
EXIT_ZOMBIE(16)	Z	Process has exited, waiting for RC collection
EXIT_DEAD(32)	---	Process has exited, cleanup pending

As shown in the diagram, there is no clear distinction between Running (i.e. presently executing in a core or hyperthread) and Runnable (that is, on the run queue, but waiting for an available CPU) - both are, from the kernel's perspective, the same state. A thread will actually run for as long as it can, until one of two occur:

- **Preemption:** occurs when, due to an external interrupt, the kernel realizes that either the thread quantum (allotted timeslice) has expired, or some higher priority thread has become runnable. In both these cases, while the thread would no doubt benefit from prolonging execution, it is kicked out in favor of another thread which takes its place, in what's known as a **context switch**. This is obviously contrary to what the thread would have wanted (if it had a will or a say), and is therefore considered **nonvoluntary**.
- **Sleep/Wait:** occurs when the thread simply has nothing to do at the present moment. This can occur because of one of several reasons, namely:
  - **The user stopped the thread:** by using the `STOP` signal. UN*X users are likely familiar with the CTRL-Z combination, which causes the terminal driver to send the signal to the main thread, thereby stopping the entire thread group - or what they know as the process. A thread or group thus stopped can only be resumed with the `CONT` signal, which is usually what `fg` or `bg` send. You can, of course, stop and resume threads manually by using `kill -STOP` ***pid*** and `kill -CONT` ***pid***, respectively.
  - **The terminal driver stopped the thread:** because of an attempt to run a full-screen command (e.g. vi, more) in the background, or any background command on input, or on output when the `stty +tostop` setting is set. The signal sent here is `TSTP`, but otherwise behaves similarly to `STOP`.
  - **The thread actually yielded the CPU:** which occurs when the thread calls `sleep(2)` or other forms of delayed execution, or - more commonly - when the thread is waiting for an IPC object (e.g. a mutex). This can also occur implicitly, when the thread makes an I/O call that cannot be immediately serviced (i.e. is not present in the buffer or page caches). Such I/O requires storage or human user input, both of which are orders of magnitude slower than the CPU. The I/O system call therefore chooses the greater good, which is to suspend the thread and put it on an I/O wait queue*. When the I/O is complete (via an interrupt), the thread can be rescheduled, possibly preempting another thread. In any of these cases, however, the thread "agrees" (or, at least, acquiesces) the context switch, which is why it is referred to as a **voluntary** context switch.

The distinction between voluntary and nonvoluntary context switches is an important one, which is why you can find those statistics as fields in the status entry. A thread with an unusually high number of nonvoluntary context switches keeps getting "kicked out" of the CPU when it still needs it - which implies it might benefit from an increased priority (or is just a plain CPU hog).

The last two states shown in the diagram - Zombie and Dead - are non-states. A UN*X process has a very clear raison d'etre - its return code, which it is supposed to provide in an `exit(2)` system call, or as a return value from its `main()`. This code, however, must be picked up by the parent process. This requires responsible parenting, in the form of calling one of the `wait#(2)` system calls (usually after obtaining the child's SIGCHLD death notification) to collect the return code. A main thread briefly enters the Zombie state when it exits (or returns from `main(2)`), in the hopes of being put out of its misery by its parent. If the parent fails to live up to its obligations, however (by ignoring the signal or forgetting `wait#(2)` after `fork()`, as some programmers do), the child is condemned to a Walking Dead state. Fortunately, UN*X Zombies are quite benign, and don't actually consume any resources - memory, CPU, or other - aside from a process table entry. The Zombies may also find peace when their recalcitrant parent dies (or is killed), leading to an adoption by `init()` (PID 1), which is always happy to call `wait#(2)` and provide Requiem.

* - The choice of whether to enter `TASK_INTERRUPTIBLE` (which still accepts signals) or `TASK_UNINTERRUPTIBLE` (which pends signals) is left up to the system call, or more accurately the driver in charge of servicing said call.

High-Level memory statistics

The /proc/*pid*/status entry also offers valuable high-level insights into process memory utilization (note here, we use "process" rather than "thread", because resources are handled at the process, not thread level). The various statistics are shown in Table 7-1:

Table 7-1: High Level memory statistics in /proc/*pid*/status

Metric	Meaning
VmPeak	Virtual Memory high-water mark: The highest value obtained by VmSize over the lifetime of this process
VmSize	Virtual Memory size, at the present moment.
VmLck	Memory locked by mlock(2) APIs. For most applications, this is 0.
VmPin	Pinned memory. For most applications, this is 0.
VmHWM	Resident memory footprint high-water mark: The highest value obtained by VmRSS over the lifetime of this process
VmRSS	Resident memory footprint, at the present moment.
VmData	Size of data segment - This is the size of the process heap memory
VmStk	Size of process thread stacks
VmExe	Size of executable
VmLib	Memory used by shared library (.so) files
VmPTE	Memory used by Page Table Entries
VmSwap	Memory used by process in swap (In Android - no swap, ergo always 0, unless using swap to ZRAM)

Looking at the high level statistics can often provide a quick diagnosis as to memory hogging problems - particularly high levels of `VmRss:`. To gain more insights into memory problems, however, we need to consider the more fine grained statistics of /proc/smaps - and memory management in general.

User mode memory management

Programmers don't normally pay attention to memory. It's a given that each process gets its own address space, wherein it can allocate memory freely according to need, and be assured that the kernel will handle all the minutiae. The address space is **private** - that is, belonging only to this process, and **virtual** - i.e. abstracted from the actual RAM by the kernel and memory management unit.

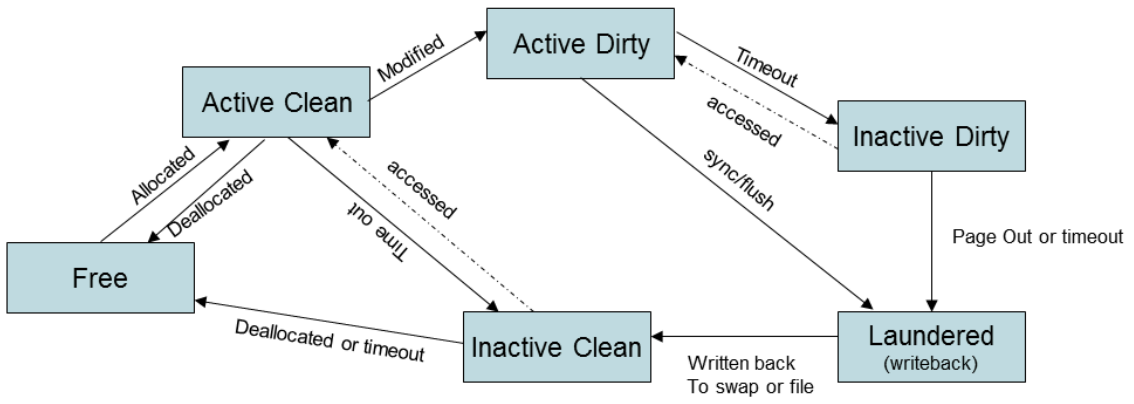
In practice, however, memory is one of the most critical bottlenecks an application can face. Improper memory management not only has an adverse effect on the mismanaging process, but also on the entire system. The effect is further exacerbated on Android: whereas on Linux memory allocations which would deplete the RAM could be backed by swap (leading to excessive swapping and performance degradation but still being satisfied), Android has no swap - and, as a consequence, it's simply not possible to deplete the RAM without triggering a global out-of-memory condition. At that point, the only way to recover RAM would involve killing a victim process to recover its RAM.

Android does a remarkable job of optimizing the available memory, to compensate for the lack of swap. The very design of the Dalvik Virtual Machine emphasized sharing as much virtual memory as possible. Indeed, multiple instances of the traditional Java VM (such as that of Sun's J2ME) simply could not be satisfied without 100+ MB of RAM per instance. Dalvik VMs instances, by contrast, are nearly all shareable, resulting in a fairly low memory footprint for individual apps.

Virtual Memory classification and lifecycle

It's tempting to think that "all virtual memory is the same", but that is far from true. Virtual memory can be classified by no less than four types, which impact its usage - and, even more importantly, its release. Pages have their own lifecycle, depicted in the following figure:

Figure 7-2: The page lifecycle



State	Meaning
Free	Page is unallocated
Active Clean	Page is claimed by one or more processes, but not modified
Active	Page is claimed by one or more processes, and modified
Inactive Dirty	Page has aged since last access, and needs to be commited
Laundered	Page is selected for committing (writeback pending)
Inactive Clean	Page has been committed (or not modified). May be reused or freed

Named (mmapped) vs. Anonymous

The first classification of a memory page denotes its source - **mapped** pages are pages taken from files on the storage (disk, flash, or network file system). The pages are loaded from a file, (via the kernel's page cache), contiguously into the process' virtual memory. The file name serves as a name for the memory itself, which is why mapped pages are often also referred to as **named** memory.

By contrast, some memory is not file backed, and is created ad-hoc for the process' immediate need. This includes memory used by the stack or the heap, when the program sets up a stack frame or calls `malloc(3)`. This memory has no backing in a file, and therefore has no name - which is why it is often referred to as **anonymous** memory.

The `maps` per-process proc entry will show you all memory mappings. Named mappings are easy to see because the device and inode number, along with the corresponding filename, are clearly listed. Anonymous file mappings have a no numbers, but "special" anonymous mappings - such as the stack or heap, are clearly shown.

Dirty vs. Clean

Once a page is loaded into memory, it may remain unaltered - used as read-only memory, or it may be modified by the process for whatever reason. Unaltered memory is referred to as **clean** memory, whereas modifying memory makes it **dirty**.

This distinction is more than just cleanliness - The system needs to know if a page is dirty or clean for several reasons, including knowing what to do when the page has spent too much time in memory, and how it affects sharing - which brings us to the next classifications:

Active (referenced) vs. Inactive

Virtual Memory pages , when mapped to RAM, have an "age". The kernel and MMU work together to maintain a reference mechanism, tracking the Least Recently Used (**LRU**) pages. When a page is accessed, it is immediately marked as Active. If left unused for more than a given period, the status is changed to inactive.

The activity indication is important for purposes of **purging** and **writeback**. Purging refers to the process of discarding a page, when it is no longer required. If the page is both inactive and clean, it implies that it has either a zeroed page, or it has been backed up to storage. In either case, it can be rid of immediately, to make room for another process' virtual memory paging request.

Writeback refers to the process of taking dirty pages, and writing them back to storage - commonly, the files whence they came, but - in the case of anonymous memory - to swap. Mapped pages are especially important to write back within reasonable time, because the system could face power loss (or a crash) unpredictably. Mapped pages not written will result in data loss. The kernel therefore maintains page expiration parameters in `/proc/sys/vm`. Anonymous pages are written to swap (compressed RAM, in Android systems which support it). If there's no swap (or compressed RAM) to write back to, the system faces an Out-Of-Memory condition (as we describe later).

Private vs. Shared

Memory that is unique to the process is considered **private**. This implies that only the process has access to the memory page, and no other process can physically access the memory. Normally, this is when a process requests a file mapping using `mmap(2)` with the `MAP_PRIVATE` flag, or (more commonly) allocates anonymous memory using `malloc(3)` or `new`.

Memory can also be **shared**, in between two or more processes. This is usually the result of a deliberate sharing done by the programmer, using memory sharing calls such as `mmap(2)` with the `MAP_SHARED` flag, or other mechanisms (such as System V `shm*` APIs, or - in Android - `/dev/ashmem`). This is known as `explicit sharing`, because the process specifically tells the kernel - I want this memory to be shareable with others.

Shared memory may be mapped in different virtual addresses in process A and process B, but both will eventually get to the same physical page. This means that there is only one physical copy of shared memory in the system - after all, why waste two RAM pages with exactly the same content? This also has the upside of relieving the kernel from having to maintain the sharing by updating multiple copies of memory on change. If there's only one copy mapped to all processes, any change in that copy is instantly reflected among all processes.

Things get a little bit more complicated: Sometimes, a process may request private memory, but the system may decide to share it anyway, without informing the process - in what's known as **implicit sharing**. Examples of this abound - in fact, most memory is implicitly shared, unless otherwise stated. For example, consider libraries and frameworks: Each process certainly needs a copy of them, but the vast majority of the processes use the copies verbatim, not making any changes. Library and framework code, for example, is mapped read-only, and therefore, by definition, *cannot change*. It doesn't make sense, then, to map individual copies of libraries (especially commonly used ones, like Bionic), when all copies are the same. In these cases, even though the process may request `MAP_PRIVATE`, the kernel basically says "sure, sure", but performs sharing anyway, outright lying to the process, which remains entirely oblivious.

To maintain this elaborate ruse, the kernel does require a little bit more overhead: If memory is implicitly shared, and one of the parties tries to modify it anyway, the kernel will have to allow that party to modify the memory, without affecting any of the others. This is when the kernel employs **copy-on-write**, which involves intercepting the write attempt (by a page fault), then creating a new copy which can be written to, and remapping that copy to the process virtual memory, instead of the original page, which is left unmodified and mapped to all other parties' address space.

Experiment: Examining address space mappings through /proc/*pid*/maps

The per-process procfs maps entry provides a full layout of the process address space, enabling you to quickly determine which files have been mapped, alongside the anonymous memory regions. Output 7-9 demonstrates the entry for the shell (on a 64-bit system), with annotations:

Output 7-9: Examining a process address space (e.g. the shell) through /proc/*pid*/maps

```
shell@flounder:/ $ cat /proc/$$/maps
# Most executables are mapped in three segments:
#                      Write
#                  Read|Exec
# Address Range       |||   Off
5579579000-55795bc000 r-xp 00000000 103:0d  495                    # Code (text), read-only
55795cb000-55795cd000 r--p 00042000 103:0d  495                    # Data, read-only
55795cd000-55795ce000 rw-p 00044000 103:0d  495                    # Data, read-write
55795ce000-55795cf000 rw-p 00000000 00:00 0                        # Anonymous: Reservation
559e8f6000-559e90c000 rw-p 00000000 00:00 0                        # Anonymous: Heap
..
7f8298b000-7f82a12000 r-xp 00000000 103:0d  1275   /system/lib64/libc.so  # Bionic code
7f82a12000-7f82a22000 ---p 00000000 00:00 0                       # Guard (traps overflows)
7f82a22000-7f82a26000 r--p 00087000 103:0d  1275   /system/lib64/libc.so  # Bionic constants
7f82a26000-7f82a29000 rw-p 0008b000 103:0d  1275   /system/lib64/libc.so  # Bionic data
..
7ff66d9000-7ff66fa000 rw-p 00000000 00:00 0                              [stack]
..                       |
                     Mapping type (p = MAP_PRIVATE, s = MAP_SHARED)
```

You can see even more detail by examining the smaps per process entry. This provides the same information as maps did, but with the additional breakdown by classification for every region - but that will be demonstrated shortly, in the next experiment.

To view memory statistics on a system-wide level, you can consult the /proc/meminfo file. The file (used by utilities like top and vm_stat uses the same nomenclature and classifications just defined:

Output 7-10: Examining system wide memory utilization with /prc/meminfo

```
shell@flounder:/ $ cat /proc/meminfo
MemTotal:                     # Total physical RAM in system (2GB, minus reservations)
MemFree: Buffers: 177452 kB   # RAM still free
Cached:              336 kB   # Buffer cache (used to cache raw device blocks)
SwapCached:       488156 kB   # Page cache (used to cache filesystem based I/O)
Active: Inactive:     52 kB   # Anonymous memory also in swap
Active(anon):     570104 kB   # Total active (broken further to file/anon below)
                  338552 kB   # Total inactive  - broken further to file/anon below -+
                  303228 kB   # Total active anonymous memory
Inactive(anon):   130684 kB   # Total inactive anonymous memory
Active(file):     266876 kB   # Total active mmap(2)ed memory
Inactive(file):   207868 kB   # Total inactive mmap(2)ped memory
Unevictable:        1836 kB   # Memory locked and unremovable/swappable
Mlocked:               0 kB   # Memory locked by processes using mlock(2)
SwapTotal:        520908 kB   # Swap: This is a 64-bit L, swapping to compressed RAM
SwapFree:         496780 kB   # Available swap
Dirty:               324 kB   # Total dirty pages
Writeback:             0 kB   # Total scheduled to be written back to files/swap
AnonPages:        422024 kB   # Total presently anonymous memory
Mapped:           227664 kB   # Total presently memory mapped
Shmem:             11912 kB   # Total explicitly shared memory in the system
# ... The rest of the file shows kernel-related memory statistics
```

Memory Metrics

To calculate memory statistics, one has to take into account quite a few factors, such as whether or not memory is resident, shared, and other parameters. We can start off with the following simple formula:

```
VmSize = VmRSS + VmFileMapped + VmSwap + VmLazy
```

In plain English, this means that the virtual memory of a process may be classified into four disjoint categories:

- **VmRSS:** The **R**esident **S**et **S**ize - are those pages of virtual memory which are presently backed by physical RAM pages. This may be because they have been recently active, or - in some cases - because the process (or kernel) requires them to be locked in memory. The resident memory may further be subclassified as Unique (private to this process) or Shared (in between one or more processes).
- **VmFileMapped:** Pages which were retrieved from files, by means of the mmap(2) system call, may be written back to the files at any time to free memory. In fact, in most cases if the pages remain clean (unmodified), they can simply be discarded as - if need arises - they can always be reloaded from the files, which are still on flash/disk. The size of pages in this category is not directly reported in /proc/*pid*/status, but can be figured out from /proc/smaps, and - on a system-wide level - from /proc/meminfo.
- **VmSwap:** Pages which resulted from memory allocation (i.e. malloc(3) or similar) do not have any file backing. These are also known as **anonymous** pages, since they have no name (read: filename) to back them up. It follows that there is no way to write them back out. In Linux, swap space comes to the rescue, as a portion of storage set aside to back anonymous pages. In Android, however, there is no swap. This value is therefore almost always 0, unless the system swaps to compressed RAM (ZRAM).

- **VmLazy:** Programmers are a greedy lot, often allocating far more memory than they actually need. The kernel takes a lazy approach to allocation, preferring to set aside pages "on paper" until they are actually required, or written to. These pages are allocated in the process Page Table Entries (visible in /proc/*pid*/status as `VmPTE:`), but the actual allocation is deferred until a pointer to the page is actually dereferenced. The kernel then experiences a *page fault* from the MMU, which reports - correctly - that the page does not exist. The kernel then proceeds to actually allocate the page. Using lazy allocation saves a great deal of memory, but does impact performance marginally. A worse scenario occurs when the page fault cannot be satisfied due to no available physical pages, and no way to write back any to disk. That's when an Out-of-Memory (OOM) condition occurs, which we discuss later in this chapter.

It's tempting to sum up VmRSS over all processes in order to calculate the overall RAM footprint. Doing so, however, would be wrong - because Some of the resident memory of the process may in fact be shared with others, in which case a simple summation would end up overcounting the shared regions. A more accurate measure is needed, and this is what Linux offers with the **PSS** - **P**roportional **S**et **S**ize statistic. In mathematical speak, PSS would be defined as:

$$PSS = USS + \sum_{i=1}^{s} \frac{Shared_i}{p_i}$$

Where:

s = # of shared regions

$Shared_i$ = Size of shared region i

p_i = # of processes sharing region i

If, like most non-math-types, you find the equation a tad alarming (or promised yourself to not ever use Sigma notation again), this can be put in words, like so:

- PSS will count 1K for every 1K of private memory (USS). That is, if a process has a private memory page, it counts in full for purposes of calculating the PSS footprint
- PSS will only count 1/n K for every 1K of shared memory, with "n" dependent on the number of processes sharing this region. Because there may be more than one regions, we need the Sigma notation - which basically says, add $1/n_1$ for the first region found, then add another $1/n_2$ for the second region found (assuming n_2 sharers), etc.

This might seem odd at first, but when you take the sum of PSS over all processes in the system - mathematical magic adds up the shares in a way that every shared region is counted in full, and exactly once, for the purposes of determining the footprint (If you want a slight challenge, you can work out the (double) sigma notation required to prove the correctness of this claim).

Fortunately, PSS measurements prove far more useful than the algebraic equations backing them, and are readily obtainable (with virtually no need for math) directly from the /proc/smaps file. This is best exemplified in the following experiment:

Experiment: Observing RSS, USS and PSS through /proc/*pid*/smaps

The per-process smaps entry breaks down memory regions from the maps entry and provides detailed information on each. For this experiment, find a binary that isn't concurrently executing as another process, and doesn't terminate quickly. A good candidate for that is `ping`, which is an actual binary (not a `toolbox` tool), and will run indefinitely.

Start `ping` with some address - it doesn't even matter if the address is reachable - what matters is that `ping` will run. If you choose another binary, that's fine too - the choice of binary is entirely inconsequential for this experiment - what matters is that the binary loads, and creates a process instance, which you can then suspend. Once the binary is running (possibly paused for input), hit `CTRL-Z` to suspend it and go back to a prompt, or leave it running in the background. Then, inspect the first 10 or so lines of its smaps entry. Using `ping`, it will look something like this:

Output 7-11(a): Examining the USS, RSS and PSS of a single instance of a given binary

```
# Start the binary in the background - collect PID
shell@flounder:/system/bin $ ping 1.1.1.1 > /dev/null &
[1] 20117
shell@flounder:/system/bin $ more /proc/20117/smaps # Inspect ping's smaps entry
55705e3000-55705ec000 r-xp 00000000 103:0d 463      /system/bin/ping
Size:                 36 kB
Rss:                  32 kB
Pss:                  32 kB
Shared_Clean:          0 kB
Shared_Dirty:          0 kB
Private_Clean:        32 kB
Private_Dirty:         0 kB
Referenced:           32 kB
Anonymous:             0 kB
AnonHugePages:         0 kB
Swap:                  0 kB
KernelPageSize:        4 kB
MMUPageSize:           4 kB
Locked:                0 kB
VmFlags: rd ex mr mw me
dw ..
```

What do we see in the output?

- The first memory region is loaded from /usr/bin/ping on disk (no surprise here). The region is readable, executable, and (seemingly) private (`r-xp`). It was loaded from device `103,0d`, inode #463.
- The VmSize of this region is 36Kb. Of which, 4k have been immediately freed - because the Rss is 32k. This amounts to a portion of the ELF header, which has no practical use in memory during runtime.
- The 32k of RSS are all private, and clean: Private, implies unique to this process (i.e. USS), and clean implies that they have not been modified since their loading. All 32k are also recently active (Referenced), which again is no surprise, since ping is executing. The pages are not anonymous (because they are mapped to a file).
- Consequentially, the PSS is 32K. With no shared memory, every 4k of USS map to 4K of PSS.

So far (hopefully), so good. But what happens when we start another instance of `ping` (or our process)? Doing so, we then inspect the smaps entry of the first instance (not the second!), and see that it has changed!

Experiment: Observing RSS, USS and PSS through /proc/*pid*/smaps (cont.)

Output 7-11(b): Examining the USS, RSS and PSS of the first of two concurrent instances of a given binary

```
# Once again, start the binary in the background - collect PID
shell@flounder:/system/bin $ ping 1.1.1.1 > /dev/null &
[2] 20130
shell@flounder:/system/bin $ more /proc/20117/smaps
55705e3000-55705ec000 r-xp 00000000 103:0d 463        /system/bin/ping
Size:                 36 kB  # VmSize unchanged
Rss:                  32 kB  # RSS unchanged
Pss:                  16 kB  # PSS drops by half because
Shared_Clean:         32 kB  # All 32k are now shared!
Shared_Dirty:          0 kB
Private_Clean:         0 kB
Private_Dirty:         0 kB
Referenced:           32 kB
Anonymous:             0 kB
AnonHugePages:         0 kB
Swap:                  0 kB
KernelPageSize:        4 kB
MMUPageSize:           4 kB
Locked:                0 kB
VmFlags: rd ex mr mw me
dw ..
```

Comparing the two outputs you can see that most metrics in the original process have not changed - The VmSize is still 36k with a 32k RSS. The RSS, however, is now all shared - between the two instances of the binary - and therefore the PSS has dropped by half, to 16k.

Running this example with another instance of the process will bring down the PSS to 10k (technically 10.6k, but rounded down), and with four instances - to 8k (32k divided cleanly by four). Killing instances reduces the number of sharers, and brings up the PSS value.

Note, that throughout the example, the region remained seemingly private (`r-xp` in both outputs). This is obviously false, since smaps clearly shows the region becomes shared. This paradox is resolved by explaining 'p' not as private, but as `MAP_PRIVATE` - i.e. the argument to `mmap(2)` when mapping the region. This is the same "ruse" that was previously discussed when explaining private/shared memory. Stated otherwise, the process mapped this region as private, and the operating system plays along - but if another process maps this same region, the kernel reserves the right to make this **implicitly shared** between the processes, so long as neither process actually writes to (dirties) the region. If a write occurs, the writing process triggers a **copy-on-write** fault, which forces the kernel to actually allocate another copy of the page(s) written to, so as not to violate the existing copy. This is in contrast to 's' in the permission flags, which means **explicitly shared** (i.e. `MAP_SHARED` in the `mmap(2)` arguments) - denoting that the region can be dirtied and still remain shared as a single copy.

The experiment hopefully served as a simple example of how PSS is calculated. 'Simple', because this was a simple case where all memory was shared, thereby reducing USS to zero, and making the PSS calculation straightforward. Other regions may be mixed - both private and shared, which makes PSS calculation a bit more challenging, but fortunately smaps does that automatically.

If you're not a fan of parsing /proc/smaps manually, there's a tool for that - two, in fact. This is shown in the next experiment.

Experiment: Observing RSS, USS and PSS through procrank and librank

The AOSP provides two useful tools to show memory statistics - procrank and librank. These are not present in most production devices, but it's a simple enough matter to copy them from the emulator image to the device, along with their dependency, /system/lib/libpagemap.so. This is shown in the following output:

Output 7-12: Moving binaries from the emulator to a real device

```
morpheus@forge (~/tmp)$ adb -s emulator-5554 pull /system/xbin/procrank
morpheus@forge (~/tmp)$ adb -s emulator-5554 pull /system/xbin/librank
morpheus@forge (~/tmp)$ adb -s emulator-5554 pull /system/lib/libpagemap.so
# kill emulator or use -s with serial number of device from adb devices
morpheus@forge (~/tmp)$ adb push librank /data/local/tmp
morpheus@forge (~/tmp)$ adb push procrank /data/local/tmp
morpheus@forge (~/tmp)$ adb push libpagemap.so /data/local/tmp
```

On the device, you'll need to make the binaries executable (by using chmod(1)), and then execute them. Because the library dependency is also in /data/local/tmp, and libraries are searched for in /system/lib[64], you'll need to alter the library load path. On a rooted device, this is not necessary if you drop the library dependency in /system/lib.

Output 7-13: Moving binaries from the emulator to a real device

```
# On device:
shell@htc_m8wl:/ $ chmod 755 /data/local/tmp/procrank
shell@htc_m8wl:/ $ /data/local/tmp/procrank
CANNOT LINK EXECUTABLE: could not load library "libpagemap.so" needed by
"/data/local/tmp/procrank" caused by library "libpagemap.so" not found
shell@htc_m8wl:/data $ LD_LIBRARY_PATH=/data/local/tmp /data/local/tmp/procrank
  PID       Vss      Rss      Pss      Uss  cmdline
  234   331600K   89211K   61314K   49124K  system_server   ..
```

Once you have procrank and librank copied (or, if you just run them on the emulator), you can turn to analyzing their output. Both tools operate by iterating over the per-process smaps statistics (newer versions work with the pagemap entry), but they differ in how they output the statistics: procrank does so by sorting processes in descending order of memory usage, while librank sorts regions of memory by the processes using them. There's a lot to be learned from the output about memory utilization (and optimization) in Android. Starting with pagemap:

Output 7-14: The output of procrank from the L Emulator

```
root@generic:/ # procrank
  PID       Vss      Rss      Pss      Uss  cmdline
  354   631600K   99212K   60712K   48784K  system_server
  709   578240K   91200K   45218K   24512K  com.android.systemui
581   565940K   72876K   43295K   38940K  com.android.launcher
52   102852K   47784K   24356K   2132K  /system/bin/surfaceflinger
538   540284K   44108K   16408K   13284K  com.android.phone
66   508532K   46416K   14799K   8948K  zygote
955   531912K   30760K  6756K        4700K  com.android.calendar
   843   522168K   30364K6264K        4396K  com.android.providers.calendar
  1077   521044K   28308K5499K        3668K  com.android.browser
   978   520144K   29004K5311K        3336K  com.android.deskclock
  1037   520732K   27380K5082K        3484K  com.android.exchange
..
```

Experiment: Observing RSS, USS and PSS through procrank (cont.)

As the output shows, processes are ranked by descending order of Vss (as per `-v`, the default; you can also sort by `-p`ss, `-r`ss, or `-u`ss). More advanced options will show only cached (`-c`) or non-cached (`-C`) pages - try `-h` for more options.

In the output, however, something stands out very quickly - though Vss sizes are humongous (565M for the Launcher, really?), the actual RSS sizes are small (indicating plenty of discarded mappings), and the PSS sizes are smaller still. As you can see, the average unique footprint of apps is no more than a few MB! About 85%-90% of the RSS of the average app is all shared, reducing its PSS dramatically. This shareability is owed to the structure of Zygote and the Dalvik VM (ART included) which maximize shared memory in ways Java never could (but still not as efficiently as iOS, some would argue).

You can work back to see how much memory is shared by subtracting the USS measure from the RSS. Subtracting the USS from the PSS will give you the weighted average of the shared regions, and dividing the shared memory size by this amount will give you a rough idea of how many processes are sharing the same regions. This is a rough idea only, because summing up PSS loses some granularity - different regions likely have a different number of sharers.

The `librank` tool's output is slightly different, as it is sorted by memory region. Otherwise, however, the terminology remains the same. The following output demonstrates the usage of the boot.oat shared region, which holds precompiled framework classes in ART:

Output 7-15: The output of librank demonstrating sharing on the ART precompiled classes

```
root@generic:/ # librank
 RSStot      VSS      RSS      PSS      USS  Name/PID
..
27179K                                      /data/dalvik-cache/arm/system@framework@boot.oat
          48556K   19576K    6468K    3268K    system_server [354]
          48556K   13488K    3982K    2112K    zygote [66]
          48556K   13984K    3802K    2192K    com.android.phone [538]
          48556K   14800K    3164K     744K    com.android.systemui [709]
          48556K   11880K    1933K     324K    com.android.launcher [581]
          48556K   10544K    1302K     160K    com.android.inputmethod.latin [500]
          48556K    9268K     958K     104K    android.process.media [766]
          48556K    9048K     861K      64K    com.android.email [997]
          48556K    8200K     826K      96K    com.android.server.telecom [532]
          48556K    8628K     759K      20K    com.android.calendar [955]
          48556K    7960K     655K      12K    com.android.providers.calendar [843]
          48556K    7280K     588K      24K    com.android.deskclock [978]
          48556K    7220K     566K      20K    com.android.browser [1077]
          48556K    6772K     474K       0K    com.android.exchange [1037]
          48556K    6132K     429K       0K    com.android.dialer [1061]
          48556K    5796K     405K       4K    com.android.sharedstoragebackup [1096]
```

`librank` shows, yet again, the efficiency of sharing - The VSS associated with the boot.oat is some 48MB. In practice, however, less than half is resident, and - in most processes unique memory footprint of the oat is in the low KB.

Newer versions of Android make use of an even more clever forms of sharing, through the Linux kernel's Kernel Samepage Merging (**KSM**) mechanism. This features lets the kernel auto-detect identical physical pages in memory (by comparing hashes), even if they are not memory mapped. If an identity is detected, the pages can be merged, subject to the usual copy-on-write restrictions. KSM has been a feature in the Linux kernel as of 2.6.27 or so, but has only recently entered Android.

Out of Memory conditions

Despite all the extensive memory sharing in Android, along with tricks like KSM or ZRAM, lack of real swap space is an inherent problem. Android is not at fault here - swap and flash simply do not go well together, due to flash memory's limited Program/Erase (P/E) cycles. As a consequence, running out of memory at any given time is a clear and present danger.

The Linux kernel has long had a mechanism to deal with memory shortage. This mechanism - called OOM (Out-Of-Memory) is triggered when a memory request cannot be satisified. In Linux, this happens rarely - if the system is low on RAM there is usually ample swap space to fall on. It's only when the system is both out of RAM *and* swap space that OOM is triggered.

OOM isn't a thread - it's implemented as a code path following the page fault which occurs. The code looks through the list of processes, and attempts to find the most suitable candidate, whose sacrifice will result in the best memory gain for the system. All processes are candidates on this "death row" , sorted by their `oom_score` - a heuristically devised score which evolved as did the kernel. This score is visible in the per-process /proc/*pid*/oom_score as a read-only pseudofile.

The problem with the heuristic is, that - as will all heuristics - it doesn't always reliably work. Often times, innocent processes are sacrificed just for being with the wrong score at the wrong time. Execution is imminent and swift, with no saving throw - essentially a `kill -9` - and there is nothing the victim can do about it.

It is for this reason that the Android application lifecycle exists with the perpetual fear of untimely death. Applications are not guaranteed persistence in any way, and are instead given callbacks to save their state (as an opaque `android.os.Bundle`), with the only promise that, if they are killed, they will be reincarnated with that bundle. An application has no way to predict when and even if it may be terminated. As opposed to iOS's jetsam (a mechanism designed for a similar functionality), the application doesn't even leave a tombstone (though some detail to the kernel log is saved).

In an effort to bring a bit more determinism to the heuristic, Linux offered a method to adjust the score from user space. First, as /proc/*pid*/oom_adj and (in later kernels) as /proc/*pid*/oom_score_adj. These files enable a user space process to add a modifier to the score - a negative modifier to reduce the score (thus making the process less killable) or a positive modifier to increase the score (effectively giving the process suicidal tendencies).

Android's system processes use this mechanism to make themselves unkillable. /init and its cohorts from the various .rc files give them an `oom_adj` of -16 or -17 (which completely disables OOM for the task). In newer kernels, setting the `oom_score_adj` to -1000 achieves a similar result, which effectively makes their score close to (if not) 0.

In the wrong hands, this could have also been abused by apps (after all, who wouldn't resist the temptation for immortality?) but Android's `ActivityManager` automatically resets the score adjustment along various stages of the application's lifecycle (as we discussed in Volume II). As of Android L, The `ActivityManager` relies on the `lmkd` (discussed in Chapter 5), because the adjustment files are owned by (and writable to) root only.

Android takes another precautionary measure, in the form of the LowMemoryKiller (lmk). This is an Androidism which enhances OOM by preemptively killing processes before a real OOM condition is triggered. In previous Android versions, init would set the module's parameters via sysfs on startup. With L, init merely ensures file permissions on the sysfs pseudofiles, leaving the task to lmkd instead.

Experiment: OOM memory adjustments in action.

You can observe the OOM score adjustments in real time by examining the procfs entries during application lifecycle. For this experiment, open an ADB shell while using an app. For example, if you're using the Chrome web browser, you'll see:

Output 7-16(a): Viewing an active application's OOM scores

```
root@flounder:/ # ps | grep chrome
u0_a34    12079 211   2295800 167140 ffffffff a7058b1c S com.android.chrome
root@flounder:/ # cat /proc/12079/oom_score_adj
0
root@flounder:/ # cat /proc/12079/oom_adj
0
root@flounder:/ # cat /proc/12079/oom_score
70
```

Moving the application to the background (by simply pressing the home button) will automatically reflect in OOM. The oom_adj increases, and the score shoots up accordingly

Output 7-16(b): Viewing an active application's OOM scores

```
root@flounder:/ # cat /proc/12079/oom_adj
6
root@flounder:/ # cat /proc/12079/oom_score_adj
411
root@flounder:/ # cat oom_score
470
```

In Android L, you can also attach a trace to `lmkd` during the application lifecycle events, to see incoming messages from `ActivityManager` to `lmkd`, in order to adjust the scores. This was shown in the Experiment in Chapter 5 (specifically, output 5-6). Suspending `lmkd` (by `kill -STOP`) will prevent any OOM modifications.

Tracing System Calls

Virtually any "meaningful" operation performed by a user-mode thread requires some kernel-involvement. Whether it is dealing with a file, opening a socket, or handling any type of resource outside one's own previously allocated virtual memory, a user-mode thread must request that service from the kernel, by means of a *system call*.

System calls require the user mode process to first traverse into kernel mode. The method of doing so differs with each architecture, but always involves a special machine instruction - ARM's `SVC` (a.k.a `SWI`), or Intel's `SYSENTER` (or `SYSCALL`). These instructions set the processor mode to privileged (supervisor) mode, and are setup by the kernel upon boot to transfer control to a predefined kernel entry point - `system_call`. All system calls are thus funneled to one function. The system call number (passed in ARM's `r12` or Intel's `EAX`) is used to redirect execution to the specific system call implementation, by consulting an internal table.

Given all the above, it should be clear why system calls deserve special focus, when it comes to debugging and tracing processes. Most of the time, the internal operations inside a process - changing this or that variable - aren't of too much interest, if only because they are so plentiful and hard to trace. Operations on files or sockets, however, are especially interesting, and tracing system calls provides a simple way to trace these operations, among others.

The `toolbox ps` tool

Toolbox's `ps` tool, however crude, does offer two valuable fields pertaining to system calls: `WCHAN` and `PC`. The first denotes the "Wait Channel", which is the kernel address the entry is presently in, or -1 (0xffffffff) if this cannot be determined (Recall each line in `ps` refers to a kernel thread or the main thread of a process, unless `ps -t` is used). The second is the return address (in user space), where execution resumes after the system call. Resolving the kernel address requires some manual work, as shown in the following experiment:

Experiment: Manually resolving `toolbox ps`'s `WCHAN` value

When faced with a `WCHAN` address - or any kernel address - you can follow the simple method shown here to use /proc/kallsyms and resolve it to a more readable symbol. You start at the exact address - which never produces a match, since entries in kallsyms are only for entry points, and the `WCHAN` is inside a function. You then go back by removing the least significant digits, taking advantage of `grep`'s ability to match the prefix. At some point, `grep` will match one or more addresses - and the closest one to the one checked is the name of the function the kernel is in.

Output 7-17: Resolving a kernel address using /proc/kallsyms

```
# Prerequisite - disable kernel pointer hiding (0 - fully disable, 1 - root only)
root@generic:/# echo 0 > /proc/sys/kernel/kptr_restrict
# Now, attempt to get the address. Note the first few attempts fail
1|root@generic:/data # grep c0029d4 /proc/kallsyms
1|root@generic:/data # grep c0029d /proc/kallsyms
1|root@generic:/data # grep c0029 /proc/kallsyms
c00294f0 T exit_signals
...
# Too many results - go back by 1 hex digit (i.e. d - 1 = c)
root@generic:/data # grep c0029c /proc/kallsyms
c0029c14 T do_sigtimedwait    # Got it!
```

One caveat to keep in mind - make sure to find the closest symbol *before* and not *after* the address you're looking for. Sometimes (like in this example) the closest symbol may wrap, and other times `grep` might return matches which are after your symbol (and therefore incorrect).

The toybox in Android M's preview automatically maps the WCHAN address to a symbol, by looking at /proc/<pid>/wchan (as shown next). The experiment is just as relevant, however, since it shows the *technique* for resolving kernel addresses.

wchan and syscall

The /proc filesystem also offers system call tracing mechanisms. The wchan per-thread entry, like the `toolbox ps` output, shows the location in kernel mode where a thread is sleeping (or 0, if the thread is presently active), but also resolves it to the closest symbol, saving you the hassle of the previous experiment. What more, it works even if the `/proc/kallsyms` file restricts addresses.

In some kernels, the syscall per-thread entry offers even more detail: It captures the system call number, along with arguments, that the thread is in at the time of polling. The format of this is demonstrated through Output 7-18:

Output 7-18: The syscall and wchan procfs entries

```
root@flounder:/proc/12079 # cat syscall
# num   Arg1      Arg2      Arg3  Arg4    Arg5     Stack Pointer  Program Counter
22      0x10 0x7fc139b110 0x10 0x2324a 0x0 0x8 0x7fc139b040   0x7fa7058b1c
# syscall 22 in ARM64 is epoll_wait, as confirmed by wchan:
root@flounder:/proc/12079 # cat wchan
SyS_epoll_wait
```

You can resolve the *program_counter* value - which is also the value quoted by `toolbox top`'s `PC` - using the method shown in the previous experiment. A caveat with system call numbers, however, is that they are not guaranteed to remain constant across architectures. The system call numbers of Intel and ARM are understandably different, but more surprisingly those of 32-bit and 64-bit are sometimes different. You will need the specific system call file for your architecture, which you can find in the Android NDK, under platforms/android-*APIversion*/arch-*arch*/usr/include/asm/unistd.h, replacing *arch* with arm, arm64, x86 or x86_64. Fortunately, kernels with syscall procfs entry normally have wchan as well, so you can resolve the syscall number via wchan, as demonstrated above. Most kernels also have a stack entry, which details the kernel stack.

The `strace` tool

The methods shown so far all used polling - i.e. you could get an exact reading on a system call, but were responsible for initiating the reading, and could only capture one result at a time. This is useful in case of diagnosing a hanging or unrespnsive process. Most system call tracing, however, is best performed as an on-going operation, attaching to the process as unobtrusively as possible, and getting notifications on every system call it performs.

This is where `strace` comes into play. This powerful binary, which has been used several times by now in this book to trace and explain the internals of processes, is utterly invaluable as a tracing tool. A complete example of its usage would likely take up a chapter by itself, but table 7-2 summarizes some of the more useful switches:

Table 7-2: The more useful switches of `strace`

Switch	Use
`-i`	Print instruction pointer at time of syscall
`-t[t[t]]`	Print timestamp, with/without usecs
`-f`	Follow the clone() syscall, auto-attaching to child processes/threads
`-o` ***`file`***	Save output to *file*
`-v[v]`	Verbose mode for various syscall arguments

`strace` is exceptionally good at understanding the system call arguments (even more so when `-v`/`-vv` is used. At the time of writing, there is no Android-aware version of the tool, nor is there an ARM64 compatible version. The `jtrace` tool, from the book's companion website, provides an `strace` clone which addresses both these issues.

Summary

This chapter focused on the usage of the /proc file system - in particular, the per-process entries in /proc/*pid* and per-thread entries in /proc/*pid*/task/*tid* - and the plethora of information they provide, to allow for powerful native-level debugging and tracing of processes. The methods demonstrated apply to mainline Linux in the same ways, because procfs is an integral part of the Linux kernel.

References and Files Discussed in this Chapter

Reference	Provides
/proc/*pid*/fd /proc/*pid*/fdinfo	Information about open file descriptors for process
/proc/*pid*/maps	Address space of process, as list of mapped and anonymous regions
/proc/*pid*/smaps	As per /proc/*pid*/maps, but with per-region statistics
/proc/*pid*/status	Information from process or thread's control block (kernel's `task_struct`)

1. www.kernel.org/doc/Documentation/filesystems/proc.txt Documentation about the procfs filesystem entries.

VIII: Android Security

As with other operational aspects, Android relies on the facilities of Linux for its basic security needs. For most apps, however, an additional layer of security is enforced by the Dalvik Virtual Machine. Android Security is therefore an amalgam of the two approaches - VM and native - which allows for defense in depth.

This chapter starts by providing a brief insight into *threat modeling*: A practice taken by security experts to try and analyze the possible attack vectors and threats which may compromise a device. Malicious apps, and theft are just two of the possible threats considered, as mobile security must address all the "traditional" faults of desktop security, and then some.

We continue by exploring the Linux user model, and its adaptation to the Android landscape. Starting with the native Linux permissions, and the clever usage of IDs for Apps and group membership. We then proceed to highlight capabilities, an oft overlooked feature of Linux used extensively in Android to work around the inherent limitation using the almighty root uid in the classic model. Next is a discussion of SELinux, a Mandatory Access Control (MAC) framework introduced in 4.3 and enforced in 4.4. Lastly, we consider various protections against code injection, the bane of application security.

At the Dalvik level, we consider the simple, yet effective permission model enforced by the Virtual Machine and the package manager, as well as the bindings to the Linux level. But up to this point, both Linux and Dalvik can be thought of as aspects of *application level* security.

We therefore next consider *user-level* security: protecting the device against human users by locking the device. No longer the domain of simple PINs and patterns, device locking methods get ever more innovative, and have expanded to include biometrics as well. As of JB, Android allows multiple users to coexist, each with his or her own private data, and set of installed applications, and so the implementation of multiple users is covered as well.

At this point, we turn to a discussion of encryption on Android. Beginning with aspects of key management, we explain the inner workings of the keystore service, and the maintenance of cetificates on the device. We then touch on Android's storage encryption feature (introduced in HoneyComb) and filesystem authentication using Linux's **dm-verity** (as introduced in KitKat).

Last, but in no way least, is a focus on device rooting, without which no discussion about security would be complete. Rooting brings with it tremendous advantages to the power user (and is one of the reasons Android's popularity has exploded in hacker and modder circles), but also woeful, dire implications on application and system security. The two primary methods - boot-to-root and "one-click" are detailed and contrasted.

Threat Modeling Mobile Security

If one considers the evolution of hacking, a logical progression can be seen: At first, the main targets were servers. It was much easier to hack into a server, a "sitting duck" in terms of being always connected to the internet, than try to hack into a desktop, which only sporadically, if at all, was ever connected - and even then, through a low bandwidth modem.

This changed with the proliferation of broadband connections, and the rise of local area networks. Suddenly, millions of new potential targets emerged on the Internet. As desktop machines, the security posture was off to a much weaker start than a server. Insecure defaults and the overly user-friendly (and complex) operating system that was Windows provided a ripe breeding ground for hackers, and brought on waves of worms and malware.

Attack vectors

Mobile devices, while similar in some respects to desktops, have an entirely different threat landscape. Unlike the latter, their very mobility exposes them to far more risks, as they may be accidentally misplaced, or deliberately stolen. This effectively negates the aspects of digital security one could enforce on a desktop, by restrictring access at the lock and key (or keycard) level, opening up a slew of attacks an adversary could try once physical access to a device is obtained.

But that, alas, is only half of it: Unlike desktops, mobile devices - being far more personal - are more likely to contain personal user data, which makes them more lucrative a target for hacking. The attack profile has also changed - rather than obtain full control of the device remotely (what hackers call "pwning"), it often suffices to just get access to user data, and - using a likely always on Internet connection - smuggle it out to a remote server.

The Rogue App

The primary attack vector on a mobile device is from within: That of a **rogue application**. Users are eager to expand the functionality of their devices by installing more and more apps. But a misbehaving or deliberately malicious app, could attempt to access the user's information, or even take over phone functionality, for example by sending premium SMS messages for outrageous prices. Generally, this is classified as **local privilege escalation**, as an application is already installed and running on the local device, but with a restricted set of privileges, which is wishes to elevate.

To prevent this, Android must treat all applications as suspect. By default, applications are given a minimal set of permissions, but are otherwise restricted. The minimal set, however, does not include anything which might be potentially sensitive - even if it is vital. Accessing the network, for example, could be used maliciously to funnel out information from the device. For this reason, any permission outside the minimal set must be explicitly requested by the application, in its manifest. Each application is given its own UID, which isolates it from others, and - needless to say - root access for applications is out of the question.

Android took a step up in application restrictions in Jellybean, with the introduction of SELinux, a mandatory access control framework which effectively sandboxes all processes except the very trusted ones. In Android L, the frameworks have also been extended to support package restrictions.

That, however, is not enough - Android must also protect *itself*, as it is likely that a malicious application could try - within the limited subset of permissions it does have - to attack vulnerable components of the operating system which houses it. This is not without precedent. It's possible to exploit such vulnerabilities and trick more privileged components of the operating system - particularly those running as root - to perform an operation on behalf of the application. Due to the vast amount of code in the Android frameworks, and even more code in the underlying Linux kernel, this is a serious threat. Most past vulnerabilities have in fact done just that in order to elevate their privilege.

The Rogue User

It's hard to think of the device user as an actual threat (although iOS certainly seems to do so). The potential of device theft, however, makes it unclear as to just who the valid user is. The system must therefore be secure at all times, especially when outside the user's reach.

The first line of defense is the lock screen, which must balance the need for strong authentication credentials with an easy to use (and quick) unlock operation. After all, you wouldn't want to type in a 20 character, case-sensitive password every time your screen blanks! It therefore falls upon the user to decide what is "acceptable" security, in choosing the authentication mechanism, as well as the timeout to enforce it.

Android introduced face unlock as a method for quick (albeit not too safe) unlock, and (in Lollipop) has followed iOS with built-in support for fingerprint authentication as well. Lollipop also brings unlocking via paired devices (over Bluetooth, when the paired device - usually an Android Wear device) is near.

There is also the potential of a device being stolen, turned off, and rebooted. For this, Android must ensure its boot process is secure. Otherwise, someone could override the boot loader and restart the device in an alternate configuration, which could be less secure. This is why boot loaders are often locked by default, and if unlocked - will first efface the entire `/data` partition.

Finally, the user's data should be encrypted - else a sophisticated attacker can simply pry it open and access the raw flash storage. Android offered encryption as early as Honeycomb, but once again trailed iOS as it only enabled it by default beginning with Lollipop. The encryption key must not rest anywhere on the device, and be derived from the user's unlock code for maximum usability.

Remote Code Injection

Last, but not least (if all the above weren't bad enough), mobile devices are still subject to the very same attack vector servers and desktops were - remote code injection. The same class of vulnerabilities which plague desktop can also affect mobile devices, as attackers seek to target devices over the Internet, either as random "drive-by" (malware spam or malicious banners), or through targetted attacks (usually socially engineered email).

Webkit, which served as the basis for Android's browser and webviews, has proven to be an inexhaustible font for vulnerabilities. These were often carried out by a combination of malforming HTML, CSS, Javascript, or all of the above. Google has now moved to Chrome as the default browser, but the potential of a vulnerability in such a frequently used code based is so great, that Lollipop checks and automatically updates Chrome indepedently of the rest of the OS.

It's worth noting that code injection can also exist in the boot loader phase. Such a vulnerability could offer the same effect as unlocking the bootloader - i.e. booting into any configuration desired - but without effacing data, and thereby compromising the user's data.

The Android approach to security

In security, the union of two elements does not necessarily make them secure. Quite the contrary, in fact, as it suffices that one of the elements contains a vulnerability, in order for the entire system to be compromised. Android has learned this oh-so-well over its relatively short existence, as its security has been broken time and time (and time) again, despite significant improvements with each version. Sometimes, the vulnerability lay in Android itself, and other times in the underlying Linux. It follows, therefore, that Android security must incorporate both worlds - Linux and its own - and combine them together as efficiently and as securely as possible.

Security at the Linux Level

Android builds a rich framework on top of the Linux substrate, but at its core, relies on Linux for virtually all operations. The Linux inheritance also binds Android to use the same security features as those offered by Linux - the permissions, capabilities, SELinux, and other low-level security protections.

Linux Permissions

The security model of Linux is a direct port of the standard UN*X security model. This model, which has remained largely unchanged since its inception some 40 years ago, provides the following primitives:

- **Every user has a numeric user id**: The actual user name doesn't matter, though some usernames are reserved for system users (which are designated the owners of configuration files and directories). Two users may share the same user id, but this in effect means that, as far as the system is concerned, this represents a single user with two username/password combinations.
- **Every user has a numeric primary group id**: Much like the username, the group name doesn't matter, and some GIDs are reserved for system use.
- **Users may hold memberships in additional groups**: Traditionally, additional group memberships is maintained by the `/etc/group` file. It lists the group names, group ids, and any members who are not already in a group by virtue of the primary GID.
- **Permissions on file are granted for a specified user, group, and "other":** This is the familiar output of "`ls -l`", which maps the permissions (read, write or execute) to the user and group, and the "rest of the world". Both files and directories follow this extremely limited model, for which UN*X has been duly criticized. Because of its limitations, file access requirements basically force the creation of specialized groups
- **(Almost) everything in UN*X can be accessed as files**: It thus follows that access to system resources - named IPC objects, UNIX domain sockets, and devices - is a corrolary of file permissions. In other words, since the resources have a filesystem representation they can be `chown/chgrp/chmod`ed just as files can be, and have the same type of permissions.
- **UID 0 is omnipotent**: Because of the way permission checks are implemented, "0" effectively short circuits the checks and grants access to all files, or resource. What follows is that uid 0 (the "root" user) wields power absolute over the system.
- **SetUID or SetGID binaries allow assuming another uid (or joining another group) during their execution**: with no questions asked. Having execute permission to a Set[ug]id binary will automatically bestow those special permissions. This mechanism, which rightfully looks like a gaping design flaw, is actually a feature, used to work around privileged operations, such as changing one's uid (`su`) or password (`passwd`). Such operations - by definition - are only possible for uid 0, but can be enabled if the root user empowers specific binaries (by `chmod 4`*`xxx`* and `2`*`xxx`*, for SetUID and SetGID, respectively). As a precaution, copying or moving the binaries will strip those bits.

Android takes the classic model - which it obtains for free from the underlying Linux system - and naturally employs it, but offers a different, somewhat novel interpretation: In it, the "users" are granted to individual applications, not human users. Suddenly, much in the same way as human users sharing the same UN*X server were comparmentalized from one another, applications enjoy (and are limited by) the same seclusion. A user cannot access another user's files, directories, or processes - and this exact isolation enables applications to run alongside eachother, but with no power to influence one another. This approach is quite unique to Android - iOS runs all applications under one uid (mobile, or 501) and relies on kernel-enforced sandboxing to isolate applications from one another.

When an application is installed for the first time, the PackageManager assigns it a unique user id - which is understandably referred to as an *application id*. This id is taken from the range of 10000-90000, and bionic - the Android C runtime library - automatically maps this to a human readable name - app_XXX or u_XXXX.

Android can't get rid of SetUID support entirely - because this requires recompilation of the kernel and other modifications. Beginning with JB 4.3, however, no SetUID binaries are installed by default, and the /data partition is mounted with the nosuid option.

System defined AIDs

Android maintains the lower range of user ids - 1000-9999 - exclusive for system use. Only a subset of this range is actually used, and it is hardcoded in android_filesystem_config.h. Table 8-1 shows the UIDs defined and used by Android. Most of these are used as GIDs as well: By joining secondary groups, system processes like system_server, adb, installd and others gain the ability to access system files and devices, which are owned by these groups - a simple yet effective strategy.

Table 8-1:: Android AIDs and their default holders

GID	#define	Members	Permits
1001	AID_RADIO	system_server	/dev/socket/rild (To Radio Interface Layer Daemon) Access net.*, radio.* properties
1002	AID_BLUETOOTH	system_server	Bluetooth configuration files
1003	AID_GRAPHICS	system_server	/dev/graphics/fb0, the framebuffer
1004	AID_INPUT	system_server	/dev/input/*, the device nodes for input devices.
1005	AID_AUDIO	system_server	/dev/eac, or other audio device nodes access /data/misc/audio, read /data/audio
1006	AID_CAMERA	system_server	Access to camera sockets
1007	AID_LOG	system_server	/dev/log/*
1008	AID_COMPASS	system_server	Compass and location services
1009	AID_MOUNT	system_server	/dev/socket/vold, on the other side of which is the VOLume Daemon
1010	AID_WIFI	system_server	WiFi Configuration files (/data/misc/wifi)
1011	AID_ADB	(reserved)	Reserved for ADBD. Owns /dev/android_adb.
1012	AID_INSTALL	installd	Owns some application data directories
1013	AID_MEDIA	mediaserver	Access /data/misc/media, and media.* service access
1014	AID_DHCP	dhcpcd	Access /data/misc/dhcp Access dhcp properties
1015	AID_SDCARD_RW		Group owner of emulated SDCard
1016	AID_VPN	mtpd racooon	/data/misc/vpn, /dev/ppp
1017	AID_KEYSTORE	keystore	Access /data/misc/keystore (system keystore)
1018	AID_USB	system_server	USB Devices
1019	AID_DRM		Access to /data/drm
1020	AID_MDNSR	mdnsd	Multicast DNS and service discovery
1021	AID_GPS		Access /data/misc/location
1023	AID_MEDIA_RW	sdcard	Group owner of /data/media and real SDCard
1024	AID_MTP		MTP USB driver access (***not*** related to mtpd)

Table 8-1 (cont):: Android AIDs and their default holders

1026	AID_DRMRPC		DRM RPC
1027	AID_NFC	com.android.nfc	Near Field Communication support: /data/nfc, and nfc service lookup
1028	AID_SDCARD_R		external storage read access
1029	AID_CLAT		CLAT (IPv6/IPv4)
1030	AID_LOOP_RADIO		Loop Radio devices
1031	AID_MEDIA_DRM		DRM plugins. Access to /data/mediadrm.
1032	AID_PACKAGE_INFO		Package information metadata
1033	AID_SDCARD_PICS		PICS folder of SD Card
1034	AID_SDCARD_AV		Audio/Video folders of SD Card
1035	AID_SDCARD_ALL		All SDCard folders

Android system properties also rely on UIDs for access control - init's property_service limits access to several property namespaces, as was shown in Chapter 4. It likewise falls on the servicemanager, as the crux of all IPC, to provide basic security. Though the Binder eventually provides security through a uid/pid model, servicemanager can restrict the lookup of well known service names to given uids, though uid 0 or SYSTEM are always allowed to register. Up to and including KitKat, this was in a hard-coded allowed array, as shown in Listing 8-1:

Listing 8-1: Hard-coded service permissions (from service_manager.c) on KK

```
/* TODO:
 * These should come from a config file or perhaps be
 * based on some namespace rules of some sort (media
 * uid can register media.*, etc)
 */
static struct {
    unsigned uid;
    const char *name;
} allowed[] = {
    { AID_MEDIA, "media.audio_flinger" },
    { AID_MEDIA, "media.log" },
    { AID_MEDIA, "media.player" },
    { AID_MEDIA, "media.camera" },
    { AID_MEDIA, "media.audio_policy" },
    { AID_DRM,   "drm.drmManager" },
    { AID_NFC,   "nfc" },
    { AID_BLUETOOTH, "bluetooth" },
    { AID_RADIO, "radio.phone" },
    { AID_RADIO, "radio.sms" },
    { AID_RADIO, "radio.phonesubinfo" },
    { AID_RADIO, "radio.simphonebook" },
/* TODO: remove after phone services are updated: */
    { AID_RADIO, "phone" },
    { AID_RADIO, "sip" },
    { AID_RADIO, "isms" },
    { AID_RADIO, "iphonesubinfo" },
    { AID_RADIO, "simphonebook" },
    { AID_MEDIA, "common_time.clock" },
    { AID_MEDIA, "common_time.config" },
    { AID_KEYSTORE, "android.security.keystore" },
};
  // ........
int svc_can_register(unsigned uid, uint16_t *name)
{
    unsigned n;

    if ((uid == 0) || (uid == AID_SYSTEM)) return 1;

    for (n = 0; n < sizeof(allowed) / sizeof(allowed[0]); n++)
        if ((uid == allowed[n].uid) && str16eq(name, allowed[n].name))
            return 1;
    return 0;
}
```

With the introduction of SE-Linux, and the slow but steady migration of Android to it, the hard-coded method has been finally abandoned, in favor of integration with an SE-Linux policy, much in the same way as init's properties have. At any rate, it's important to note this is but one layer of security: servicemanager refuses to allow untrusted AIDs to register well known names. As we discuss later, the Binder allows both client and server to perform additional permission checks, and an additional layer of Dalvik-level permissions is also employed.

Paranoid Android GIDs

Android GIDs of 3000 through 3999 are also recognized by the kernel, when the CONFIG_PARANOID_ANDROID is set. This restricts all aspects of networking access to these GIDs only, by enforcing additional gid checks in the kernel socket handling code. Note that netd overrides these settings, because it is running as root. Table 8-2 shows the known network ids

Table 8-2: Android Network-related AIDs and their holders

GID	#define	Members	Permits
3001	AID_BT_ADMIN	system_server	Creation of AF_BLUETOOTH sockets
3002	AID_NET_BT	system_server	Creation of sco, rfcomm, or l2cap sockets
3003	AID_NET_INET	system_server	/dev/socket/dnsproxyd, and AF_INET[6] (IPv4, IPv6) sockets
3004	AID_NET_RAW	system_server,mtpd, mdnsd	Create raw (non TCP/UDP or multicast) sockets
3005	AID_NET_ADMIN	racoon, mtpd	Configure interfaces and routing tables
3006	AID_NET_BW_STATS	system_server	Reading bandwidth statistics accounting
3007	AID_NET_BW_ACCT	system_server	Modifying bandwidth statistics accounting

Isolated Services

As of Jelly Bean (4.1) Android introduces the notion of isolated services. This feature is a form of compartmentalization (similar to iOS's XPC) which enables an application to run its services in complete separation - in a different process, with a separate UID. Isolated services use the UID range of 99000 through 99999 (AID_ISOLATED_START through _END), and the servicemanager will deny them any request. As a consequence, they cannot lookup any system services, and are effectively limited to in memory operations. This is primarily useful for applications such as web browsers, and indeed Chrome is a prime example of using this mechanism. As shown in output 8-1, isolated services are marked as u##_i##:

Output 8-1: Chrome's isolated services

```
shell@htc_m8wl:/$ ps | grep chrome
u0_a114   4577  384   1178728 118528 ffffffff 4007941c S com.android.chrome
u0_i0     5510  384   1283624  89788 ffffffff 4007941c S com.android.chrome:sandboxed_process0
#
# Pulling the Chrome.apk to the host and dumping its manifest:
#
morpheus@Forge (/tmp)$ /aapt d xmltree Chrome.apk  AndroidManifest.xml
....
  E: service (line=285)
    A: android:name(0x01010003)="org.chromium.content.app.SandboxedProcessService0"
    A: android:permission(0x01010006)="com.google.android.apps.chrome.permission.CHILD_SERVICE"
    A: android:exported(0x01010010)=(type 0x12)0x0
    A: android:process(0x01010011)=":sandboxed_process0"
    A: android:isolatedProcess(0x010103a9)=(type 0x12)0xffffffff   0xffffffff="true", so isolated
  # ... 12 more entries
  E: service (line=298)
    A: android:name(0x01010003)="org.chromium.content.app.PrivilegedProcessService0"
    A: android:permission(0x01010006)="com.google.android.apps.chrome.permission.CHILD_SERVICE"
    A: android:exported(0x01010010)=(type 0x12)0x0
    A: android:process(0x01010011)=":privileged_process0"
    A: android:isolatedProcess(0x010103a9)=(type 0x12)0x0         0x0="false", so not isolated
...
```

Root-owned processes

As with Linux, the root user - uid 0 - is still just as omnipotent - but far from omnipresent: Its use is limited to the absolute bare minimum, and that minimum is shrinking from one Android release to another. Quite a few previous Android exploits targetted root-owned processes (with vold being a perennial favorite), and the hope is that by reducing their number, the attack surface could be greatly reduced. The `installd` is an example of such a process, whose root privileges have been removed beginning with JellyBean.

It is likely impossible to remove all root owned processes: At the very least, init needs to retain root capabilities, as does Zygote (whose fork() assume different uids, something only uid 0 can do). You can see the root owned processes on your device by typing

```
ps | grep ^root | grep -v " 2 "
```

(The `grep -v` ignores kernel threads, whose PPID is 2).

Table 8-3 shows the services which still run as root by default in KitKat (but note your device may have more, as added by the device vendor)

Table 8-3: Android services still running as root

Service	Rationale
init	Somebody *has* to maintain root privileges in the system and launch others - might as well be PID 1
uevendt (init)	Minimal operation
healthd	Minimal operation
zygote[64]	Requires setuid() to change into AID when loading APKs, retains capabilities for `system_server`
debugger[64]	Requires root privileges to use `ptrace(2)`, in order to read process memory when generating tombstones
adb	Developers may need legitimate root access; system trusts ADB to immediately drop privileges to `shell` if `ro.debuggable` is 0 or `ro.secure` is 1
vold	[Un/]Mounting filesystems, and more.
netd	Configuring interfaces, assigning IPs, DHCP and more
lmkd	Adjusting OOM settings, possibly killing other processes

As stated back in Chapter 2, the vendor binaries greatly increase the attack surface of Android, especially when they are run as root. What exacerbates the matter is that, whereas the AOSP binaries remain open source and therefore easy to analyze for security by all, the vendor binaries are closed source - and some vendors sacrifice security in favor of functionality. When you hear of a specific vulnerability in a device (e.g. HTC One M8), rather than a version of Android, it is very likely the cause lies within a vendor binary.

Eventually, it is expected that Android will leave only those services which absolutely **must** have root, and others will follow in the steps of `installd`. To do so, Android will have to increase its usage of another important Linux security feature - Capabilities.

Linux Capabilities

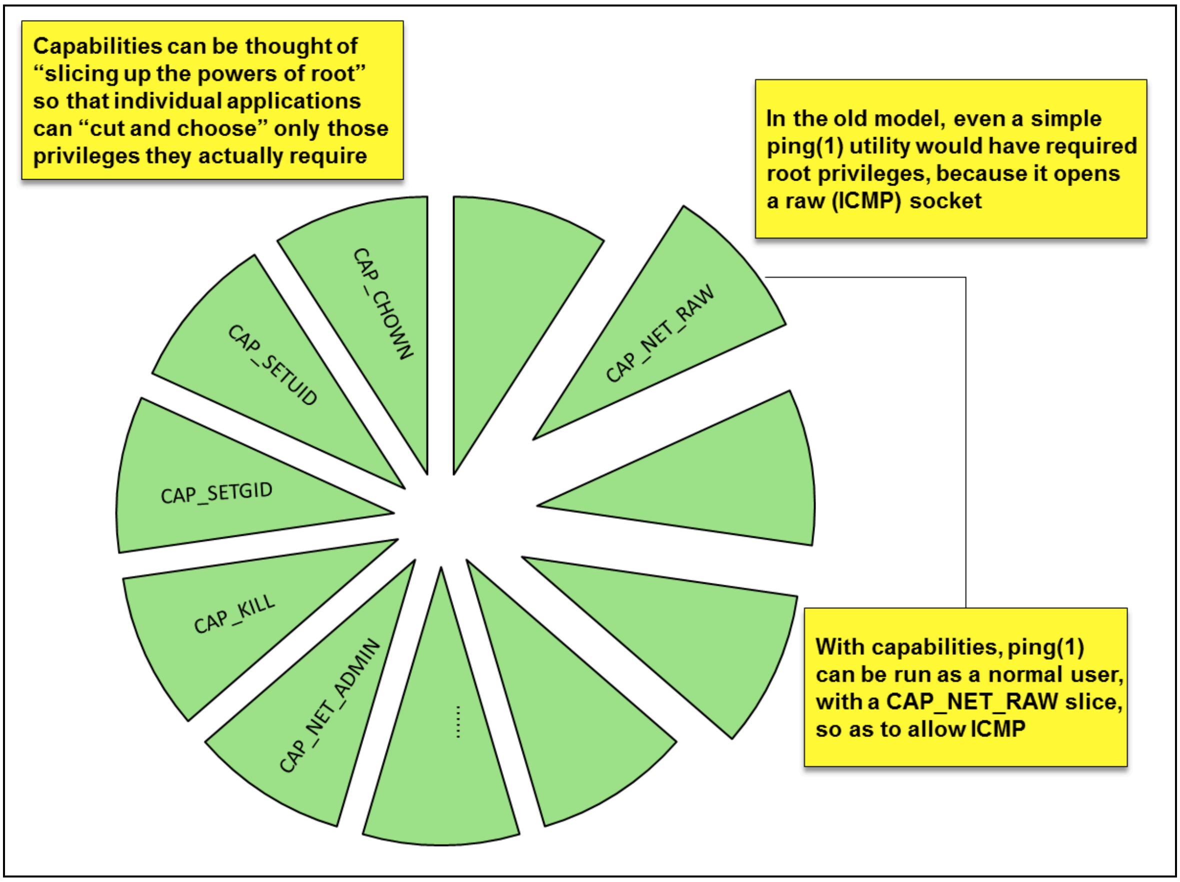
Originally part of the POSIX.1e draft (and thus meant to be incorporated as a standard for all UN*X), capabilities were an early adoption into the 2.2 line of kernels. Though the POSIX draft was eventually withdrawn, capabilties remained implemented in Linux, and have since been expanded and improved on. Distributions of Linux don't make use of capabilities all that often, but Android makes extensive use of them.

The idea behind capabilities is to break the "all-or-nothing" model of the root user: The root user is fully omnipotent, whereas all other users are, effectively, impotent. Because of this, if a user needs to perform some privileged operation, the only standard solution is to resort to SetUID - become uid 0, for the scope of the operation, then yield superuser privileges, and revert to a non-privileged user. This holds true for even relatively simple operations: Setting the system time, binding privileged (< 1024) network ports, mounting certain filesystems, and more. As a result, UN*X systems traditionally contained a very large number of SetUID binaries.

If a SetUID binary can be trusted, then - in theory - the model should work. In practice, however, SetUID poses inherent security risks: If a SetUID binary is somehow exploited, it could be tricked into compromising root. Common tricks include symlinks and race conditions (diverting the binary to overwrite system configuration files), and code injection (forcing the binary to execute a root shell - hence the term "shellcode" for injected code).

Capabilities offer a solution to this problem, by "slicing up" the powers of root into distinct areas, each represented by a bit in a bitmask, and allowing or restricting privileged operations in these areas only, by toggling the bitmask. This makes them an implementation of the *principle of least privilege*, a tenet of security which dictates that an application or user must not be given any more rights than are absolutely required for its normal operation. You can see a logical view of capabilities in Figure 8-1:

Figure 8-1: A logical representation of capabilities


Capabilities can be thought of "slicing up the powers of root" so that individual applications can "cut and choose" only those privileges they actually require
In the old model, even a simple ping(1) utility would have required root privileges, because it opens a raw (ICMP) socket
CAP_CHOWN
CAP_SETUID
CAP_NET_RAW
CAP_SETGID
CAP_KILL
CAP_NET_ADMIN
.....
With capabilities, ping(1) can be run as a normal user, with a CAP_NET_RAW slice, so as to allow ICMP

Restricting a subset of allowed privileges to only those absolutely required, while revoking the rest, increases security significantly. Even if a given application or user ends up being malicious (or cajoled to the dark path by code injection), its scope of damage is *compartmentalized*. Capabilities are like a sandbox, allowing only those operations which an app, by design, requires - while at the same time preventing it from running amuck and compromising system security. In fact, a nice side effect of capabities is that they can be used to restrict the root user itself, in cases where the user behind the uid is not fully trustworthy.

`init` still starts most of Android's server processes as root, and these processes have the full capabilities bitmask (`0xffffffffffffffff`) as they launch. Before these processes actually do anything, however, they drop their privileges, and retain only the capabilities they need. A good example of adhering to the principle of least privilege can be seen in `installd`, which makes sure to drop all but the privileges it needs for package installation:

Listing 8-2: Installd's usage of capabilities

```
static void drop_privileges() {

  // Ask the kernel to retain capabilities, since we setgid/setuid next
  if (prctl(PR_SET_KEEPCAPS, 1) < 0) {
      ALOGE("prctl(PR_SET_KEEPCAPS) failed: %s\n", strerror(errno));
      exit(1);
  }

  // Switch to gid 1012
  if (setgid(AID_INSTALL) < 0) {
      ALOGE("setgid() can't drop privileges; exiting.\n");
      exit(1);
  }

  // Switch to uid 1012
  if (setuid(AID_INSTALL) < 0) {
      ALOGE("setuid() can't drop privileges; exiting.\n");
     exit(1);
  }

  struct __user_cap_header_struct capheader;
  struct __user_cap_data_struct capdata[2];
  memset(&capheader, 0, sizeof(capheader));
  memset(&capdata, 0, sizeof(capdata));
  capheader.version = _LINUX_CAPABILITY_VERSION_3;
  capheader.pid = 0;


  // Request CAP_DAC_OVERRIDE to bypass directory permissions
  // Request CAP_CHOWN to change ownership of files and directories
  // Request CAP_SET[UG]ID to change identity
  capdata[CAP_TO_INDEX(CAP_DAC_OVERRIDE)].permitted |= CAP_TO_MASK(CAP_DAC_OVERRIDE)
  capdata[CAP_TO_INDEX(CAP_CHOWN)].permitted        |= CAP_TO_MASK(CAP_CHOWN);
  capdata[CAP_TO_INDEX(CAP_SETUID)].permitted       |= CAP_TO_MASK(CAP_SETUID);
  capdata[CAP_TO_INDEX(CAP_SETGID)].permitted       |= CAP_TO_MASK(CAP_SETGID);

  capdata[0].effective = capdata[0].permitted;
  capdata[1].effective = capdata[1].permitted;
  capdata[0].inheritable = 0;
  capdata[1].inheritable = 0;

  if (capset(&capheader, &capdata[0]) < 0)
     { ALOGE("capset failed: %s\n", strerror(errno)); exit(1); }
}
```

The heaviest user of capabilties is, unsurprisingly, `system_server`, since it is a `system` owned process, but still needs root privileges for many of its normal operations. Table 8-4 shows the Linux capabilities, and the Android processes known to use them:

Table 8-4: Linux capabilities used by Android processes

capability	#define	Users	Permits
0x01	CAP_CHOWN	installd	Change file and group ownership
0x02	CAP_DAC_OVERRIDE	installd	Override Discretionary Access Control on files/dirs
0x20	CAP_KILL	system_server	Kill processes not belonging to the same uid
0x40	CAP_SETGID	installd	allow setuid(2), seteuid(2) and setfsuid(2)
0x80	CAP_SETUID	installd	allow setgid(2) and setgroups(2)
0x400	CAP_NET_BIND_SERVICE	system_server	Bind local ports at under 1024
0x800	CAP_NET_BROADCAST	system_server	Broadcasting/Multicasting
0x1000	CAP_NET_ADMIN	system_server	Interface configuration, Routing Tables, etc.
0x2000	CAP_NET_RAW	system_server	Raw sockets
0x10000	CAP_SYS_MODULE	system_server	Insert/remove module into kernel
0x800000	CAP_SYS_NICE	system_server	Set process priority and affinity
0x1000000	CAP_SYS_RESOURCE	system_server	Set resource limits for processes
0x2000000	CAP_SYS_TIME	system_server	Set real-time clock
0x4000000	CAP_SYS_TTY_CONFIG	system_server	Configure/Hangup tty devices
0x400000000	CAP_SYSLOG	dumpstate	Configure kernel ring buffer log (dmesg)
0x1000000000	CAP_MAC_OVERRIDE	system_server	Override MAC policies (actually ignored)

Note, that table 8-4 provides a limited (albeit large) subset of the Linux capabilities. It is likely that over the evolution of both Linux and Android more capabilities will be added. The following experiment demonstrates how you can see capabilities used by processes:

Experiment: Viewing capabilities and group memberships

You can easily view system_server's capabilities and group memberships (or those of any other process, for that matter), by looking at /proc/${PID}/status, replacing ${PID} with the pid of the process in question:

Output 8-2: Viewing system_server's capabilities and group memberships

```
root@generic:/ # cat /proc/${SS_PID}/status
Name:   system_server
State:  S (sleeping)
Tgid:   372
Pid:    372
PPid:   52
TracerPid:      0
Uid:    1000    1000                # AID_SYSTEM
Gid:    1000                        # AID_SYSTEM
...     # Note the secondary group memberships, below
Groups: 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1018 1032 3001 3002 3003 3006 3007
...
CapInh: 0000000000000000            # Inherited
CapPrm: 0000000007813c20            # Permitted
CapEff: 0000000007813c20            # Effective
CapBnd: fffffe000000000             # Bounding set
```

In the above, you can see four bitmasks for capabilities: Those inheritable by child process, those potentially permitted for this process, those actively in effect (as in, permitted and also explicitly required by the process), and the bounding set. The bounding set (added in Linux 2.6.25) is a bitmask which limits the usage of capset(2)).

Experiment: Viewing capabilities and group memberships (cont.)

By looking over PIDs in /proc, you can single out the processes which use capabilities. This requires a bit of shell scripting, as shown in the following output:

Output 8-3: Processes with capabilities

```
root@htc_m8wl:/proc # for p in [0-9]*;         \ # Iterate over all PIDs
              do CAP=`grep CapPrm $p/status | \ # Get permitted capabilities
                      grep -v -v ffffff     | \ # Rule out root processes
                      grep -v 0000000000000`; \ # Rule out incapable processes
                 if [[ !  -z $CAP ]]; then    \ # If capabilities were found,
                       grep Name $p/status;     #    Print the process name
                       echo PID $p - $CAP;      #    PID, and capabilities mask
                 fi; \
            done
Name:   system_server
PID 13662 - CapPrm: 0000000007813c20
Name:  wpa_supplicant
PID 13907 - CapPrm: 0000000000003000
Name:  rild
PID 368 - CapPrm: 0000000000003000
Name:  netmgrd
PID 375 - CapPrm: 0000000000003000
Name:  installd
PID 387 - CapPrm: 00000000000000c3
Name:  dumpstate
PID 389 - CapPrm: 0000000400000000
Name:  dumpstate
PID 390 - CapPrm: 0000000400000000
Name:  qseecomd
PID 398 - CapPrm: 0000000000222000
Name:  qseecomd  # A secondary thread of qseecomd, therefore same capabilities
PID 510 - CapPrm: 0000000000222000
```

As the above shows, the capabilities are in line with Table 8-4. Note that some vendors (above, HTC) may add their own processes (above, qseecomd) with additional capabilities.

Beginning with JB (4.3), Zygote calls prctl(PR_CAPBSET_DROP) and prctl(PR_SET_NO_NEW_PRIVS), to ensure that no further capabilities can be added to its child processes (i.e. the user apps). It is likely that, going forward, vold and netd will both drop their privileges and rely on capabilities, rather than retain their root privileges. This is especially important considering vold's history of vulnerabilities.

SELinux

Figure 8-2: The SELinux logo

SELinux - Security Enhanced Linux - marks a step further in the evolution of Linux beyond standard UN*X. Originally developed by the NSA, SELinux is a set of patches which have long since been incorporated into the mainline kernel, with the aim of providing a Mandatory Access Control (MAC) framework, which can restrict operations to a predefined policy. As with capabilities, SELinux implements the principle of least privilege, but with much finer granularity. This greatly augments the security posture of a system, by preventing processes from operating outside strictly defined operational bounds. So long as the process is well behaving, this should pose no problem. If the process misbehaves, however (as most often is the case of malware, or the result of code injection), SELinux will block any operation which exceeds those bounds. The approach is very similar to iOS's sandbox (which builds on the TrustedBSD MAC Framework), though the implementation is quite different.

Though long included in Linux (and, like capabilities, not always implemented by default), SELinux was introduced into Android with JellyBean (4.3). The initial introduction was gentle - setting SELinux in permissive mode, wherein any violations of the policy are merely audited. With KitKat (4.4), however, SELinux now defaults to enforcing mode for several of Android's services (specifically, `installd`, `netd`, `vold` and `zygote`), though still permissive for all other processes. In general, it is considered a good practice to use the per-domain permissive mode, in order to test a policy before setting it to enforcing, and it is likely that enforcement will expand with the next version of Android.

SELinux's port to Android - commonly referred to as SEAndroid - was first described in a paper[1a] and a presentation[1b] by Smalley and Craig of the NSA (who have followed up on SEAndroid with an excellent presentation in the 2014 Android Builders Summit[1c]). Google provides basic documentation in the Android Source site[2]. Of the mainline Linux distributions RedHat has been an early adopter, and provides a comprehensive guide[3].

SEAndroid follows the same principle of the original, but extends it to accommodate Android specific features - such as system properties, and (naturally) the Binder (via kernel hooks). Samsung further extends SEAndroid, and uses it as a foundation for their "KNOX" secure platform (currently in v2.0). KNOX (referred to by some as "obKNOXious" :-) boasts a stronger security policy, enforcing and confining all processes (except init and the kernel threads). In the following discussion, "SELinux" refers to those features found in both Linux and Android, whereas "SEAndroid" refers only to the latter.

The main principle of SELinux (and, in fact, most MAC frameworks) is that of *labeling*. A label assigns a type to a resource (object), and a security **domain** for a process (subject). SELinux can then enforce so as to allow only processes in the same domain (likewise labeled) to access the resource (Some MAC Frameworks go as far as to make resources with different labels invisible, somewhat akin to the Linux concept of namespaces, although SELinux does not go that far). Depending on the policy, domains can also be made **confined**, so that processes cannot access any resource but those allowed. The policy enforcement is performed independently of other layers of permissions (e.g. file ACLs). The policy may also allow relabeling for some labels (relabelto and relabelfrom, also called a **domain transition**) in some cases, which is a necessary requirement if a trusted process (e.g. Zygote) spawns an untrusted one (virtually any user application).

An SELinux label is merely a 4-tuple, formatted as a string of the form `user:role:type:level`. All processes with the same label (i.e. in the same domain) are equivalent. SEAndroid (presently) only defines the *type* - i.e. the label is always in the form `u:r:domain:s0`. As of KitKat, the SEAndroid policy defines individual domains for all daemons (i.e. each daemon gets its own permissions and security profiles), along with the domains shown in table 8-5, for application classes.

Table 8-5: The application class domains in Android 4.4

Label (domain)	Apps	Restrictions
`u:r:kernel:s0`	Reserved for kernel threads	Unconfined (God Mode)
`u:r:isolated_app:s0`	Isolated processes	Previously connected anonymous UNIX sockets, read/write
`u:r:media_app:s0`	signed with media key	Allowed to access network
`u:r:platform_app:s0`	signed with platform key	
`u:r:shared_app:s0`	signed with shared key	
`u:r:release_app:s0`	signed with release key	
`u:r:untrusted_app:s0`	All other	Access ASEC, SDCard, TCP/UDP sockets, PTYs

The keys referred to in table 8-5 are defined in /system/etc/security/mac_permissions.xml, which is part of the **middleware MAC** (MMAC) implementation: The Package Manager recognizes the keys used for signing apps, and labels the applications accordingly (using a call to `SELinuxMMAC.assignSeinfoValue`. This is done during package scanning (part of the package installation, as described in Volume II). Note the term *middleware* here applies to labeling performed strictly in user mode by the Android system components.

All the `_app` domains inherit from the base `appdomain`, which allows the basic application profile, including actions such as using the binder, communicating with zygote, sufraceflinger, etc. You can find the type enforcement (.te) files, which contain the detailed definitions for all domains, in the AOSP's external/sepolicy directory. The syntax used in those files is a mixture of keywords and macros (from temacros), which allow or deny operations in the domain, as shown in Listing 8-3:

Listing 8-3: Sample te file (debuggerd.te)

```
# debugger interface
type debuggerd, domain;
permissive debuggerd;
type debuggerd_exec, exec_type, file_type;

init_daemon_domain(debuggerd)   # force automatic transition when init spawns us
unconfined_domain(debuggerd)    # Leaves debuggerd unconfined, at present
relabelto_domain(debuggerd)     # Allow domain transition to this domain
allow debuggerd tombstone_data_file:dir relabelto;  # For tombstone files
```

The files in external/sepolicy form the baseline, which all devices are meant to automatically inherit from. Rather than modify them, vendors are encouraged to add four specific variables in their BoardConfig.mk file, specifying `BOARD_SEPOLICY_[REPLACE|UNION|IGNORE]`, to override, add or omit files from the policy, and `BOARD_SEPOLICY_DIRS` to provide the search path for the directories containing their files. This mitigates the risk of an accidental policy change due to file error, which may result in security holes. The directory also contains the mac_permissions.xml template, which is populated with keys in keys.conf.

The stock type enforcement files are all concatenated and compiled into the resulting /sepolicy file, which is a binary file placed on the root file system. Doing so offers further security, because the root filesystem is mounted from the initramfs, which is itself part of the `bootimg`, that is digitally signed (and therefore hopefully tamperproof). The compilation is performed merely as an optimization, and the resulting file can be easily decompiled, as is shown in the experiment sec-dispol. The binary policy file can be loaded through /sys/fs/selinux (though init most commonly does so through `libselinux.so`).

Experiment: Decompiling an Android /sepolicy file

If you have a Linux host, decompiling an /sepolicy can be performed with the `sedispol` command, which is part of the `checkpolicy` package. Assuming Fedora or a similar derivative, this first involves getting the package, if you don't already have it:

Output 8-4: Obtaining the `checkpolicy` package

```
root@Forge (~)# yum install checkpolicy
Loaded plugins: langpacks, refresh-packagekit
--> Running transaction check
---> Package checkpolicy.x86_64 0:2.1.12-3.fc19 will be installed
--> Finished Dependency Resolution
..
Installed:
  checkpolicy.x86_64 0:2.1.12-3.fc19
root@Forge (~)# rpm -ql checkpolicy
/usr/bin/checkmodule
/usr/bin/checkpolicy
/usr/bin/sedismod
/usr/bin/sedispol # This is the policy disassembler
/usr/share/man/man8/checkmodule.8.gz
/usr/share/man/man8/checkpolicy.8.gz
```

Once you have the command, all you need is to transfer the policy to the host, and start examining it (the command is an interactive one). Though the policy is usually the one defined in /sepolicy, you can get the actively loaded policy through sysfs, as well. The /sys/fs/selinux/ directory will contain many interesting entries used for configuring (and potentially disabling) SELinux, of which one is the actively loaded policy. This will require you to do something similar to the following:

Output 8-5: Decompiling/Disassembling the active policy

```
root@htc_m8wl:/ # ls -l /sys/fs/selinux/policy /sepolicy
-r-------- root     root         74982 1970-01-01 01:00 policy
-rw-r--r-- root     root         74982 1970-01-01 01:00 sepolicy
root@htc_m8wl:/ # cp /sys/fs/selinux/policy /data/local/tmp
root@htc_m8wl:/ # chmod 666 /data/local/tmp/sepolicy
#
# Back on the host (as any user)
#
morpheus@Forge (~)$ adb pull /data/local/tmp/sepolicy
2750 KB/s (74982 bytes in 0.026s)
morpheus@Forge (~)$ sedispol sepolicy
Reading policy...
libsepol.policydb_index_others: security:  1 users, 2 roles, 287 types, 1 bools
libsepol.policydb_index_others: security: 1 sens, 1024 cats
libsepol.policydb_index_others: security:  84 classes, 1333 rules, 1 cond rules
binary policy file loaded

Select a command:
1)  display unconditional AVTAB
...
Command ('m' for menu): 1
allow qemud installd : udp_socket { ioctl read write create getattr setattr lock relabelfrom
 relabelto append bind connect listen accept getopt setopt shutdown recvfrom sendto recv_msg
 send_msg name_bind node_bind };
allow system installd : udp_socket { ioctl read write create getattr setattr lock relabelfrom
 relabelto append bind connect listen accept getopt setopt shutdown recvfrom sendto recv_msg
  send_msg name_bind node_bind };
allow keystore ctl_dumpstate_prop : property_service { set };
allow keystore ping : peer { recv };
... # Probably more output than your terminal can buffer - consider "f" for file output..
```

What remains, then, is to define the process of assigning the labels to resources, through **contexts**. The resources recognized by SELinux are Linux file objects (including sockets, device nodes, pipes, and other objects with a file representation), and SEAndroid extends this further to allow for properties.

Application Contexts

The /seapp_contexts file provides a mapping of applications (in the form of UIDs) to domains. This is used to label processes based on the UID, and the seinfo field (as set by the package manager, according to the package signature as it correlates with /system/etc/security/mac_permissions.xml). You can see the labeling of processes with the toolbox's ps -Z:

Output 8-6: SELinux process contexts with ps -Z

```
shell@htc_m8wl:/ $ ps -Z | grep platform_app
u:r:platform_app:s0   u0_a42    7343  7129  com.android.systemui
u:r:platform_app:s0   smartcard 7717  7129  org.simalliance.openmobileapi.service:remote
u:r:platform_app:s0   u0_a42    30131 7129  com.android.systemui:recentapp
u:r:platform_app:s0   fm_radio  30405 7129  com.htc.fmservice
#
# Compare with seapp_contexts
#
shell@htc_m8wl:/ $ grep platform_app /seapp_contexts
user=_app seinfo=platform domain=platform_app type=platform_app_data_file
user=smartcard seinfo=platform domain=platform_app type=platform_app_data_file # PID 7717
user=felicarwsapp seinfo=platform domain=platform_app type=platform_app_data_file
user=irda seinfo=platform domain=platform_app type=platform_app_data_file
user=fm_radio seinfo=platform domain=platform_app type=platform_app_data_file  # PID 30405
```

File Contexts

SE-Linux can associates every file with a security context. The /file_contexts file provides all the contexts for protected files, and the -Z switch of toolbox's ls can display them, as shown in the following:

Output 8-7: SELinux file contexts with ps -Z

```
shell@htc_m8wl:/ $ ls -Z /dev | grep video_device
drwxr-xr-x root     root              u:object_r:video_device:s0 video
crw-rw---- system   camera            u:object_r:video_device:s0 video0
crw-rw---- system   camera            u:object_r:video_device:s0 video1
crw-rw---- system   camera            u:object_r:video_device:s0 video2
crw-rw---- system   camera            u:object_r:video_device:s0 video3
crw-rw---- system   camera            u:object_r:video_device:s0 video32
crw-rw---- system   camera            u:object_r:video_device:s0 video33
crw-rw---- system   camera            u:object_r:video_device:s0 video34
crw-rw---- system   camera            u:object_r:video_device:s0 video35
crw-rw---- system   camera            u:object_r:video_device:s0 video38
crw-rw---- system   camera            u:object_r:video_device:s0 video39
#
# Compare with /file_contexts definitions
#
shell@htc_m8wl:/ $ grep video_device /file_contexts
/dev/nvhdcp1            u:object_r:video_device:s0 #  Left over from the external/sepolicy/file_contexts,
/dev/tegra.*            u:object_r:video_device:s0 #  even though this is not an NVidia device
/dev/video[0-9]*        u:object_r:video_device:s0 # note regular expressions gets all the above
```

Property Contexts

As discussed in Chapter 4, init's property service restricts access to certain property namespaces by a hard coded uid table. This is a very rigid mechanism, and hardly scalable as new properties and namespaces are added in between Android releases.

Since SELinux already provides the notion of execution contexts, it is trivial to extend them to properties, as well. As of JellyBean, /init protects access to properties by a check_mac_perms() boolean. The function loads the property contexts from two files - /data/security/property_contexts (when present), and /property_contexts.

Output 8-8: SELinux Property contexts

```
shell@htc_m8wl:/ $ cat /property_contexts
#line 1 "external/sepolicy/property_contexts"
##########################
# property service keys
#
#
net.rmnet0              u:object_r:radio_prop:s0
..
..
sys.usb.config          u:object_r

ril.                    u:object_r:rild_pro

net.                    u:object_r:system_pro
dev.                    u:object_r:system_pro
runtime.                u:object_r:system
hw.                     u:object_r:system_prop
sys.                    u:object_r:system_pro
sys.powerctl            u:object_r:powe
service.                u:object_r:system
wlan.                   u:object_r:system_pr
dhcp.                   u:object_r:system_pr
bluetooth.              u:object_r:bluetoo

debug.                  u:object_r:shell_p
log.                    u:object_r:shell_pro
service.adb.root        u:object
..
```

If you go back to Chapter 4, you'll see that the property contexts essentially mirror the definitions in the table. The main difference, however, is that providing the contexts in an external file provides a far more extensible way of changing and modifying properties - all without a need to recompile /init.

init and toolbox commands

Recall from Chapter 4 that the Android /init has a rich variety of commands, which may be used in its .rc files. With the introduction of SELinux, additional commands have been added to allow for SELinux contexts. Toolbox has likewise been modified to allow SELinux modifications from the shell. Table 8-6 shows these commands

Table 8-6: init and toolbox commands for SELinux

init	Toolbox	Usage
N/A	`getenforce`	Get SELinux Enforcement status
`setcon` ***`SEcontext`***	N/A	Set (change) SELinux context. Init uses `u:r:init:s0`
`restorecon` ***`path`***	`restorecon [-nrRv] pathname...`	Restore SELinux context for path
`setenforce` *`[0/1]`*	`setenforce [Enforcing|Permissive|1|0]`	Toggle SELinux enforcement on/off
`setsebool` ***`name value`***	`setsebool` ***`name value`***	Toggle boolean *value* (0/false/off or 1/true/on)

Note you can achieve most of the functionality of the SELinux commands by accessing files in /sys/fs/selinux (which is, in fact, what some of these commands do), though this would require **both** root access and an unconfined domain. /init, which remains unconfined, can also relabel processes (as it does for services with the `seclabel` option, and additionally provides the `selinux.reload_policy` property trigger to reload the policy. Disabling SELinux altogether can be accomplished through /sys/fs/selinux/disable, or through the kernel command line argument `selinux=0`.

Other noteworthy features

Linux has some additional settings which Android enables, which aim to improve security by hardening otherwise insecure defaults. This section discusses them briefly.

AT_SECURE

The Linux Kernel's ELF loader uses an auxilliary vector to provide metadata for the images it loads. This vector can be accessed through the /proc filesystem (as /proc/*pid*/auxv. One of its entries, AT_SECURE is set to a non-zero value for set[ug]id binaries, programs with capabilities, and programs which force an SELinux domain traversal. In those cases, Bionic's linker (/system/bin/linker) is configured to drop "unsafe" environment variables (a hard coded list in the __is_unsafe_environment_variable function, in bionic/linker/linker_environ.cpp. Chief amongst the variables are LD_LIBRARY_PATH and LD_PRELOAD, a favorite technique for library injection.

Address Space Layout Randomization

Code injection attacks use the target process' address space as their playing field, and their success often depends on intimate knowledge of its details - addresses, regions and protections. This is because injection attacks either directly add code into an existing program, or subvert its execution so as to jump to already existing regions. In both cases, knowledge of the layout is vital, because jumping to an incorrect address will lead to a crash. Normally, since process launch deterministically into a private address space, a hacker can (to paraphrase an old java motto) "debug once, hack everywhere".

Address Space Layout Randomization (ASLR) attempts to make injection attacks harder by introducing randomness - shuffling the layout of memory regions, making their addresses less predictable. This increases the chance a targetted piece of code will be "shifted" in memory, and basically trade a crash in place of compromise by malicious code - a lesser evil, by all counts.

Linux offers randomization capabilities through /proc/sys/kernel/randomize_va_space (or sysctl kernel.randomize_va_space). The value "0" specifies no randomization, "1" specifies stack randomization, and "2" specifies both stack and heap, which is the default. Executables can also be compiled with the PIE (Position-Independent-Executable) option (the -pie switch), which is mandatory as of Android L (defined as APP_PIE in the Android.mk files).

Experiment: Testing ASLR

To see the effects of ASLR, you can use the following shell script over /proc. The script iterates over all processes, finds the location of libc.so in it (only the text section, as filtered by the grep r-x), and displays it along with the PID if found:

Output 8-9: Showing the effects of ASLR

```
root@htc_m8wl:/# cd /proc
root@htc_m8wl:/proc # for x in [0-9]*; do \
                           lc=`grep libc.so /proc/$x/maps| grep r-x`; \
                           if [[ ! -z "$lc" ]]; then echo $lc in PID $x; fi; \
                        done | sort
400c6000-40111000 r-xp 00000000 b3:2e 1492 /system/lib/libc.so in PID 28686
.. # All Android apps share shame memory space
400f7000-40142000 r-xp 00000000 b3:2e 1492 /system/lib/libc.so in PID 9615
b6de0000-b6e2b000 r-xp 00000000 b3:2e 1492 /system/lib/libc.so in PID 7470
b6e5a000-b6ea5000 r-xp 00000000 b3:2e 1492 /system/lib/libc.so in PID 375
```

As the output shows, the library is often randomized, yet some processes still share the same location for libc - those are the spawns of the zygote, which fork() to load a class, but does not call exec() - and hence remains with the same address space layout.

Kernel-space ASLR has yet (at the time of writing) to make it into Android. Introduced for the first time in iOS 6.0, it eventually made it into Linux with version 3.14 (which is actually the most recent at the time these lines are being typed). It is quite likely to be introduced into Android with the version to follow KitKat.

> ⚠ ASLR provides a layer of defense only against code injected through an input vector. If an adversary already has execution privileges, s/he can invoke the powerful `ptrace(2)` APIs to read the address space of other processes, and even inject remote threads. Thankfully, one has to obtain root privileges first. SELinux can (and should!) be used to prevent access to `ptrace(2)` altogether.

Kernel hardening

Unlike mainline Linux, Android kernels export no /proc/kcore by default, as this entry allows kernel read-only memory access from user mode (by root). The /proc/kallsyms is still present in most devices (and actually world readable by default), but protected by the kernel.kptr_restrict sysctl, which is set by default to 2, to prevent any addresses from being displayed. Kernel ring-buffer access (via the dmesg) is likewise protected by kernel.dmesg_restrict.

Stack protections

As sophisticated as attacks can get, they still (for the most part) rely on overwriting a function pointer, which - when called - causes a subversion of the program flow. Not all programs use function pointers, but all utilize the return address, which is stored on the stack during a function call.

As a countermeasure to this, most modern compilers offer automatic stack protection, by means of a *canary*. Like the proverbial canary in the coal mine, a stack canary is a random value written to the stack upon function entry, and verified right before the function returns. If the value cannot be verified, the stack is deemed corrupt, and the program voluntarily aborts, rather than potential trigger malicious code.

This form of protection has been available in Android since its early days, with gcc's `-fstack-protector`. Note that it does not provide a panacea, since code can still be injected via function pointers aside from the return address (C++ methods make good candidates).

Data Execution Prevention

Code injection attacks rely on embedding malicious code inside input - whether direct from the user or from other sources. Input, however, is data - and memory used for data (the heap and the stack) can be flagged as non-executable. This complicates attacks somewhat, because just using the classic trampoline technique (overwriting a pointer or the stack return address with the address of the injected code) won't work if the injected code is in the data segment.

Unfortunately (for most), while making data non-executable complicates the simple attacks, attacks have considerably evolved. The current counterattack is Return-Oriented-Programming (ROP), a fairly old technique (introduced by Solar Designer in a '00 paper as return-to-libc), which strings together "gadgets" of calls back into existing portions of code in the program, simulating function calls on the stack. Because these are calls into code, there's nothing to make non-executable, and thus the protection can be fairly reliably circumvented.

Compiler-level protections

All the above protections are, in a way, treating the symptoms, rather than the disease. At the end of the day, the only proper ways to combat code injection attacks which exploit memory corruption is to exercise defensive coding, which involves input validation and strict bounds checking on memory operations. Newer versions of Android have taken that to heart, with the source compiled with enhanced checks, most notably `FORTIFY_SOURCE` and `-Wformat-security`, which add additional checks on memory copying functions, and prevent format string attacks.

Security at the Dalvik Level

Dalvik Level Permissions

Working at the level of a virtual machine, rather than native code, brings with it tremendous advantages for monitoring operations and enforcing security. At the native level, one would have to monitor system calls for any significant resource access. The problem with system calls, however, is that their granularity is inaccurate. File access is straightforward (open/read/write/close), but other operations, (e.g. a DNS lookup) are a lot harder to monitor, as they involve multiple system calls. Therein lies the advantage of the Virtual Machine - most operations are carried out by means of pre-supplied packages and classes, and those come built-in with permission checks.

Android actually takes this a step further: Whereas in a normal Java class a malicious developer could ostensibly import other classes, implement functionality from scratch or use JNI (to break out of the VM), in order to avoid permission checks, though this is next to impossible in Android: The user application is entirely powerless, devoid of all capabilities and permissions at the Linux level, so any access to the underlying system resources should be blocked right there. In order to carry out any operation which has an effect outside the scope of the application, one has to involve `system_server`, by calling `getSystemService()`.

While any app can freely invoke a call to `system_server`, none has access to its defined permissions - which `system_server` will check. This check is performed outside the application's process, so the application has no plausible avenue by means of which it may somehow obtain those permissions, unless they were a priori assigned to it. The assignment is performed when the application is loaded and installed - meaning that the user has been notified of the application's requested permissions,(has hopefully read through the very long list), and approved them (again, hopefully knowing the ramifications of hitting "OK")*. If the permission requested during runtime has been revoked (for example, through the AppOps service or through `pm revoke`), a security exception will be thrown (normally, this will crash the application, unless the developer braced for such an exception, in which case it may handle the exception, usually popping up an explanation on what permission was required, or at other times failing silently).

What follows is that the permissions themselves need no special data structures or complicated metadata. A permission in Dalvik is nothing more than a simple constant value, which is granted to an application in its manifest, as it declares it `<uses-permission>`. An application can likewise define its own constants (as `<permission>` tags in the Manifest). When the Package Manager installs an app, it adds the permissions of said app to the "permissions database", which is in effect part of the package database, /data/system/packages.xml. This database contains a lot more valuable information (including public keys) than just permissions (which is why it is discussed in detail in Volume II), but the pertinent portions of it are shown in Table 8-7:

Table 8-7: The elements pertaining to permissions in the package database

Element	Contains
permission-trees	An array of tree `items`, specifying permission namespaces, and the packages which define them
permissions	An array of permission `item`s, each of which defines: • `name` - The permission constant name, as defined in its original `permission` element • `package` - The package which defined this permission (with "android" for SDK permissions) • `protection` - which defines the permission protection level and flags from the `PermissionInfo` class. Permissions levels are 0 (`PROTECTION_NORMAL`), 1 (`.._DANGEROUS`), 2 (`.._SIGNATURE`) or 3 (`..SIGNATURE_OR_SYSTEM`), with flags for `SYSTEM` (0x10) and `DEVELOPMENT` (0x20). Note the value is printed as a decimal integer, when in fact it should be hexadecimal.

* - Android M finally fixes this naive and broken model, following iOS's design of enforcing permissions during runtime, prompting the user during the action, with the help of an out-of-process entity (in iOS this is handled by the TCC daemon).

Table 8-7 (cont.): The elements pertaining to permissions in the package database

Element	Contains
package	Each installed application is identified by its `name` attribute (reverse DNS name of package) and assigned an AID via the `userId` attribute. Specific permissions granted to the application are listed as items in the `<perms>` child element.
shared-user	AIDs shared between two or more applications are specified by the `userId` attribute, and once more specific permissions are granted - this time to the AID (i.e. all applications sharing it) as items in the `<perms>` child element.

If you inspect the package database (as root), you will find that the `<permissions>` element contains both custom permissions (i.e. those declared by installed Apps) and system ones. The built-in system permissions, along with protected broadcasts, are specified in the /system/framework/framework-res.apk, which can be examined using `aapt`, as shown in the following output:

Output 8-10: Dumping the /system/framework/framework-res.apk from a Nexus 9

```
morpheus@Forge (~/tmp) % adb pull /system/framework/framework-res.apk
6343 KB/s (19250841 bytes in 2.963s)
morpheus@Forge (~/tmp) % aapt d xmltree framework-res.apk AndroidManifest.xml | more
N: android=http://schemas.android.com/apk/res/android
 E: manifest (line=20)
  A: android:sharedUserId(0x0101000b)="android.uid.system" (Raw: "android.uid.system")
  A: android:versionCode(0x0101021b)=(type 0x10)0x15
  A: android:versionName(0x0101021c)="5.0-1573874" (Raw: "5.0-1573874")
  A: android:sharedUserLabel(0x01010261)=@0x1040104           # link to resources.arsc
  A: package="android" (Raw: "android")
  A: coreApp=(type 0x12)0xffffffff (Raw: "true")
  E: uses-sdk (line=0)
   A: android:minSdkVersion(0x0101020c)=(type 0x10)0x15      # 21, For Android L
    A: android:targetSdkVersion(0x01010270)=(type 0x10)0x15   # 21, For Android L
   E: eat-comment (line=27)
   E: protected-broadcast (line=29)
    A: android:name(0x01010003)="android.intent.action.SCREEN_OFF"
   E: protected-broadcast (line=30)
    A: android:name(0x01010003)="android.intent.action.SCREEN_ON"
    ...
  # Permission groups, as returned by pm list permission-groups
  E: permission-group (line=315)
   A: android:label(0x01010001)=@0x1040109                    # For UI or pm .. -s
   A: android:icon(0x01010002)=@0x1080532                     # For UI display
   A: android:name(0x01010003)="android.permission-group.MESSAGES"
   A: android:priority(0x0101001c)=(type 0x10)0x168
   A: android:description(0x01010020)=@0x104010a              # for pm list permissions -s
   A: android:permissionGroupFlags(0x010103c5)=(type 0x11)0x1 # FLAG_PERSONAL_INFO
  E: permission (line=323)
   A: android:label(0x01010001)=@0x1040161
   A: android:name(0x01010003)="android.permission.SEND_SMS"  # For UI or pm .. -s
   A: android:protectionLevel(0x01010009)=(type 0x11)0x1      # NORMAL (0), DANGEROUS(1), etc
   A: android:permissionGroup(0x0101000a)="android.permission-group.MESSAGES"
   A: android:description(0x01010020)=@0x1040162              # for pm list permissions -s
   A: android:permissionFlags(0x010103c7)=(type 0x11)0x1      # FLAG_COSTS_MONEY
   ...
```

As the above shows, permission are bundled into groups, with flags* defined both at the group level (through `android.content.pm.PermissionGroupInfo`) and at the individual level (through `android.content.pm.PermissionInfo`). This bundling and categorizing comes in handy for the power user, who is expected to use the `pm` upcall script to display or manage permissions.

* - Make that "flaG", since at the present time only `FLAG_PERSONAL_INFO` (group) and `FLAG_COSTS_MONEY` (permission) are used. But this scheme does allow for future extension

Experiment: Using the `pm` command

You can use `pm list permissions` to display permissions, both of the Android frameworks, and of third party applications. To do so, try:

Output 8-11: Listing permissions with `pm`

```
root@htc_m8wl:/# pm list permissions -f | more
All Permissions:
+ permission:android.permission.GET_TOP_ACTIVITY_INFO
  package:android
  label:get current app info
  description:Allows the holder to retrieve private information about the current application
            in the foreground of the screen.
  protectionLevel:signature
.. # Application declared permissions, as imported from their AndroidManifest.xml
+ permission:com.facebook.system.permission.READ_NOTIFICATIONS
  package:android
  label:null
  description:null
  protectionLevel:signature
..
```

Other useful switches include `-s` (verbose human readable output in your locale), `-g` (permission groups). The `pm` command can also be used to grant and revoke optional permissions (`pm [grant|revoke] PACKAGE PERMISSIONS`) and even toggle permission enforcement (i.e. `pm set-permission-enforced PERMISSION [true|false]`). The full syntax of this command, including some notable changes made for Android M, is explained in Volume II.

The AppOps service (detailed in Volume II) provided a powerful GUI by means of which users could track and fine-grain tune application permission usage. The GUI has been removed as part of KitKat's 4.4.2 "security update", but the service is alive and well. In fact, Lollipop introduces the `appops` upcall script, which can be used to allow, deny, ignore or reset an application's permissions. Unfortunately, the command line only allows a small subset of operations (`android:[coarse|fine|monitor]_location` and `android:get_usage_stats` and `android:activate_vpn`), but those could be extended to the full set of (presently) 48 operations by recompiling `android.app.AppOpsManager`. Note, however, that `AppOps` is another layer on top of the permissions - and uses a separate database (/data/system/appops.xml). This is shown in Output 8-12:

Output 8-12: Demonstrating the `appops` upcall script in L

```
shell@flounder:/ $ appops
usage: adb shell appops set <PACKAGE> <OP> <allow|ignore|deny|default> [--user <USER_ID>]
  <PACKAGE> an Android package name.
  <OP>      an AppOps operation.
  <USER_ID> the user id under which the package is installed. If --user is not
            specified, the current user is assumed.
shell@flounder:/ $ appops set com.android.musicfx android:get_usage_stats allow
# To see changes reflected in the AppOps database, you need root:
shell@flounder:/ $ su
root@flounder:/  # cat /data/system/appops.xml | grep -A 4 musicfx
<pkg n="com.android.musicfx">
<uid n="10014" p="true">
<op n="0" />
<op n="43" m="0" />  # android.apps.AppOpsManager.OP_GET_USAGE_STATS = 43
</uid>
```

Mapping permissions to Linux UIDs

The /system/etc/permissions/platform.xml file acts as a "glue" between Dalvik level permissions and those of Linux. The file is included in the AOSP sources, and is well documented so that vendors can (carefully) add any specific permissions or AIDs. The mapping works both ways - that is, a given <permission> can be set to grant membership to a <group>, and vice versa by using <assign-permission> to a given named permission to a uid. Listing 8-4 shows a sample of this file:

Listing 8-4: An example /system/etc/permissions/platform.xml file

```
...

  <!-- This file is used to define the mappings between lower-level system
   user and group IDs and the higher-level permission names managed
   by the platform.

    Be VERY careful when editing this file!  Mistakes made here can open
    big security holes.
-->
<permissions>

   <!-- The following tags are associating low-level group IDs with
        permission names.  By specifying such a mapping, you are saying
        that any application process granted the given permission will
        also be running with the given group ID attached to its process,
        so it can perform any filesystem (read, write, execute) operations
        allowed for that group. -->

   <permission name="android.permission.BLUETOOTH_ADMIN">
       <group gid="net_bt_admin" />
   </permission>

   <!-- =================================================================== -->

   <!-- The following tags are assigning high-level permissions to specific
        user IDs.  These are used to allow specific core system users to
        perform the given operations with the higher-level framework.  For
        example, we give a wide variety of permissions to the shell user
        since that is the user the adb shell runs under and developers and
        others should have a fairly open environment in which to
        interact with the system. -->

   <assign-permission name="android.permission.MODIFY_AUDIO_SETTINGS" uid="media"/>
      ...

    <!-- This is a list of all the libraries available for application
         code to link against. -->

    <library name="android.test.runner"
            file="/system/framework/android.test.runner.jar" />

</permissions>
```

If you check the /system/etc/permissions/ directory on your device, you will likely find several more XML files - android.hardware.* and android.software.*, copied during the build process from the AOSP files, as well as possibly some vendor provided files.

Dalvik Code Signing

Permissions by themselves are somewhat useless - after all, any app can declare whatever permissions it requires in its AndroidManifest.xml, and the unwitting user will probably click "ok" when prompted. To bolster security, Google requires digital signatures on applications uploaded to the Play store, so as to identify the developer(s) behind them, and add accountability.

Thus, all Android applications must be signed (with the process explained in Volume II. What's not so clear is - by whom. As Google was playing catch-up to Apple and opened the Play Store, it wanted to offer an advantage to developers, in the form of a simpler process. As opposed to Apple's lengthy validation process - all apps must be vetted by Apple, and digitally signed by them, Google offered anyone the ability to just create a key pair, publish their public key, and use the private key to sign their APK file. The rationale was that this achieves a similar level of identifying the APK's source, while at the same time greatly simplifying the process of submitting applications to the Store.

In practice, this led to an explosion of Malware in the Play Store. The Google approach was that any malware found and reported would be removed from the Store, and the corresponding public keys blacklisted. From the malware author's side, this was a case of "better to beg forgiveness than ask permission" - as the malware by then would have likely propagated by the time it was detected, thus achieving its purpose. This, coupled with the fact that a malware developer could always generate a new key pair, hollowed out the entire security model. A recent study published in RSA 2014[4] found that "malicious apps have grown 388 percent from 2011 to 2013, while the number of malicious apps removed annually by Google has dropped from 60% in 2011 to 23% in 2013", and that effectively one out of every 8 apps in the store is, in fact, malicious.

The Android "Master Key" vulnerability

One of the most serious vulnerabilities discovered in Android (in 2013) is what came to be known (somewhat erroneously) as the "Master Key Vulnerability". The vulnerability (discovered by BlueBox security[5a], and refined (among others) by Saurik[5b], the noted creator of iOS Cydia) occurred in of mishandling of APK files which contained files with duplicate names. APKs are ZIP files, and normally most utilities - aapt included - would not allow duplicate file names in the same zip. Technically, however, it *is* possible, and introduced a peculiar vulnerability: File signature verification was performed on the first entry in the APK, whereas extraction was performed on the second! This oddity was due to two different libraries - Java's and Dalvik's native implementation - being used for the tasks. As a consequence, it followed that anyone could take a validly signed APK file, and just add additional files with the same names as the original (including classes.dex, of course). This effectively bypassed Android's signature validation on APKs. Though fixed, the bug is a great example of oftentimes gaping vulnerabilities which need little to no technical knowledge in order to exploit.

The Android "Fake ID" vulnerability

The 2014 counterpart of the "Master Key" vulnerability became known as the "Fake ID" vulnerability. This time, a fault in Android's certificate validation allows the forgery of an application's identity, by supplying a deliberately broken certificate chain: Packing a malicious app (with a fake certificate) along with a real (though unrelated) one, or even several. As a consequnce, a malicious app could inherit the permission sets given to trusted apps (the example commonly given was impersonating Adobe's components and becoming a WebKit plugin).

The vulnerability (also discovered by BlueBox[6]) generated a big buzz at the Black Hat conference of that year, especially considering it was exploitable for almost four years - since Eclair(!) - at the time affecting all devices on the market - up to and including KitKat. Google eventually patched this, and it is no longer an issue with L - but (along with numerous other examples) it just comes to show that security vulnerabilities do abound*.

* - As an anecdote, Apple's iOS 6.x-7.0.4 all suffered a similarly embarassing bug - the so called SSL "goto fail" - which was the result of code accidentally(?) left behind that effectively bypassed SSL certificate validation. Apple was ridiculed by Andro-philes.. demonstrating that people in glass houses shouldn't throw stones.

User Level Security

So far, the discussion in this chapter focused on application level security. Android also needs to offer security at the user-level, allowing only the legitimate device user access to it, and in particular its sensitive data. Beginning with JellyBean, Android supports multiple users, which complicates matters a little.

The Lock Screen

The lock screen is a device's first and only real line of defense against theft or physical interception by malicious entities. It is also the screen most often seen by the user, when the device awakens from its frequent slumber. As such, it must be made resilient, on the one hand, but also natural and quick, on the other. As with most Android features, vendors may customize this screen, though Android provides an implementation which is often used as is.

Passwords, PINs and Patterns

The default Android lock screen allows either passwords, PINs or "patterns". Patterns are, in effect, PINs, but instead of remembering actual digits, the user simply has to swipe a grid (usually 3x3). The user can opt for an actual PIN instead, which is technically stronger than a pattern in that its length may be up to 16 characters, and it may repeat digits. A password provides a further enhancement over a PIN in that it allows a mix of different case letters and numbers.

The lock screen is, in effect, just an activity, implemented by the `com.android.keyguard` package. The package contains all the primitives for the system supplied lock screens and methods, and includes the following classes:

Table 8-8: The classes in `com.android.keyguard`

Class	provides
`BiometricSensorUnlock`	Interface used for biometric methods, e.g. FaceUnlock
`Keyguard[PIN|SimPin|Password]View`	Default views to prompt for PIN or password credentials
`KeyguardSecurityView`	Implemented by keyguard views (emulates activity lifecycle)
`KeyguardService`	Keyguard Service implementation
`KeyguardSecurityCallback`	Interface implemented by KeyguardHostView
`KeyguardViewMediator`	Mediates events to the Keyguard view

The lock screen invocation begins when the power manager wakes up the display, and notifies the implementation of the `WindowPolicyManager`. This calls the `KeyguardServiceDelegate`'s `onScreenTurnedOn`, which waits for the keyGuard. From there, it falls on the keyGuard to draw the lock screen (via some activity), and handle whatever lock credentials mechanism was chosen by the user. The lock screen can also be invoked from the `DevicePolicyManager`'s `lockNow` method, when the system policy enforces automatic locking.

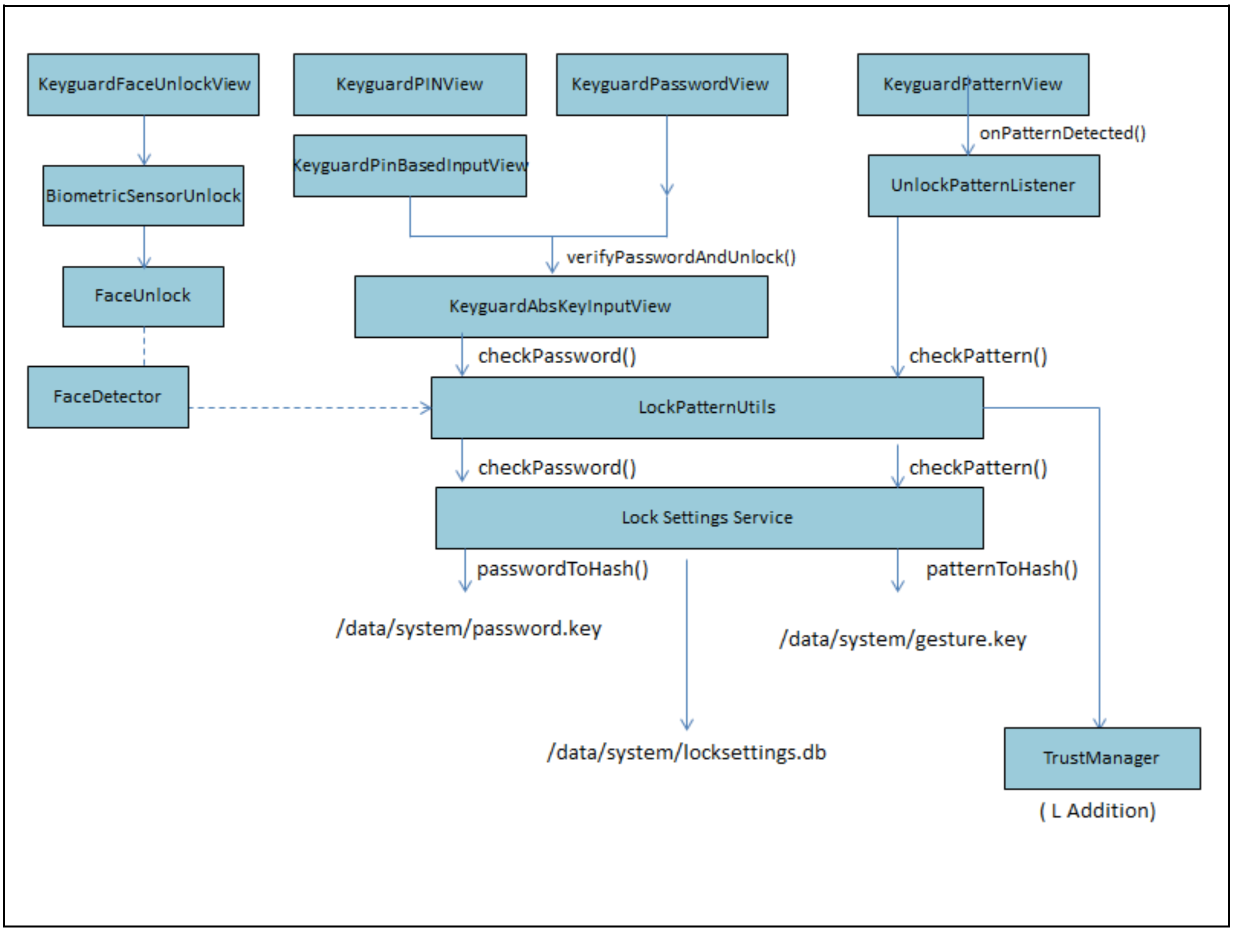
The actual logic of handling the lock is performed by `LockPatternUtils`, which calls on the `LockSettingsService`, a thread of `system_server`. The service, in turn, verifies the input against `LOCK_PATTERN_FILE` (gesture.key) or `LOCK_PASSWORD_FILE` (password.key, for PINs and passwords alike). In both cases, neither pattern nor passwords are actually saved in the file, but their hashes are. The service additionally uses the locksettings.db file, which is a SQLite database which holds the various settings for the lock screen. Those are shown in table 8-9:

Table 8-9: The locksettings.db database keys

LockPatternUtils constant	Key name (lockscreen.*)
LOCKOUT_PERMANENT_KEY	lockedoutpermanently
LOCKOUT_ATTEMPT_DEADLINE	lockedoutattempteddeadline
PATTERN_EVER_CHOSEN_KEY	patternneverchosen
PASSWORD_TYPE_KEY	password_type
PASSWORD_TYPE_ALTERNATE_KEY	password_type_alternate
LOCK_PASSWORD_SALT_KEY	password_salt
DISABLE_LOCKSCREEN_KEY	disabled
LOCKSCREEN_BIOMETRIC_WEAK_FALLBACK	biometric_weak_fallback
BIOMETRIC_WEAK_EVER_CHOSEN_KEY	biometricweakeverchosen
LOCKSCREEN_POWER_BUTTON_INSTANTLY_LOCKS	power_button_instantly_locks
LOCKSCREEN_WIDGETS_ENABLED	widgets_enabled
PASSWORD_HISTORY_KEY	passwordhistory

Putting these components together, Figure 8-3 demonstrates a slightly simplified flow through which the device is unlocked:

Figure 8-3: Unlocking the device



The TrustManager is an L addition, which helps unlock the device without a pattern - but by alternate lock methods, such as a paired BlueTooth dongle or device.

Experiment: Viewing the locksettings.db

If your device is rooted and you have the SQLite3 binary installed, you can inspect the locksettings.db file. You can also use adb to pull the locksettings.db to your host.

Output 8-13 Viewing the lock settings Database

```
root@htc_m8wl:/data # sqlite3 /data/system/locksettings.db
SQLite version 3.7.11 2012-03-20 11:35:50
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .dump
PRAGMA foreign_keys=OFF;
BEGIN TRANSACTION;
CREATE TABLE android_metadata (locale TEXT);
INSERT INTO "android_metadata" VALUES('en_US');
CREATE TABLE locksettings (_id INTEGER PRIMARY KEY AUTOINCREMENT,
                           name TEXT,user INTEGER,value TEXT);
INSERT INTO locksettings VALUES(2,'lockscreen.options',0,'enable_facelock');
INSERT INTO locksettings VALUES(3,'migrated',0,'true');
INSERT INTO locksettings VALUES(4,'lock_screen_owner_info_enabled',0,'0');
INSERT INTO locksettings VALUES(5,'migrated_user_specific',0,'true');
INSERT INTO locksettings VALUES(9,'lockscreen.patterneverchosen',0,'1');
INSERT INTO locksettings VALUES(11,'lock_pattern_visible_pattern',0,'1');
INSERT INTO locksettings VALUES(12,'lockscreen.password_salt',0,'-3846188034160474427');
INSERT INTO locksettings VALUES(81,'lockscreen.disabled',0,'1');                # No Lock
INSERT INTO locksettings VALUES(82,'lock_fingerprint_autolock',0,'0');
INSERT INTO locksettings VALUES(83,'lockscreen.alternate_method',0,'0');
INSERT INTO locksettings VALUES(84,'lock_pattern_autolock',0,'0');
INSERT INTO locksettings VALUES(86,'lockscreen.password_type_alternate',0,'0');
INSERT INTO locksettings VALUES(87,'lockscreen.password_type',0,'131072');      # PIN
INSERT INTO locksettings VALUES(88,'lockscreen.passwordhistory',0,'');
DELETE FROM sqlite_sequence;
INSERT INTO "sqlite_sequence" VALUES('locksettings',88);
COMMIT;
```

The columns in the locksettings table includes "user" (to support Android Multi-User login, as of JB). The values are usually boolean (0/1), but not always - there are some flag combinations, and a salt for the .key file. You can use SQL statements to change the lock settings from within SQLite3. though they will be cached by the lock settings service. You can also just rename the file - if you do so and restart system_server, it will be recreated with the defaults (and also have the nice side effect of resetting your password or pattern).

Alternate lock methods

Ice Cream Sandwich introduced face recognition as an alternative to the traditional methods. This was touted to much fanfare, as a potential differentiator against iOS. Unfortunately, the recognition rates are far from perfect - figures range from as low as 60% to 90%. Face recognition can also easily be defeated - by holding up a picture to the phone. Interestingly, people who have tried this method found it works with greater accuracy than the user's actual face...

The Motorola Atrix 4G was the first Android device to implement fingerprint scanning as an alternative method. This also suffered poor recognition rates. Apple's acquisition of Authentec in 2012 suggested fingerprint authentication was coming to iOS and, indeed, it made its debut in the iPhone 5S. Samsung initially slammed this as a poor, uninnovative feature, but nonetheless (and unsurprisingly) went on to introduce it to their "next big thing", the Galaxy S5. Other Android vendors are quickly following, and it seems this will become a standard feature, with L offering built-in support through its fingerprint service.

Another important addition in L is the notion of unlocking the device using another device - a paired BlueTooth device such as Android Wear, which works by proximity alone - leaving the device unlocked so long as the user is nearby. TrustManager, fingerprint, and the internals of LockSettings are discussed in volume II.

Multi-User Support

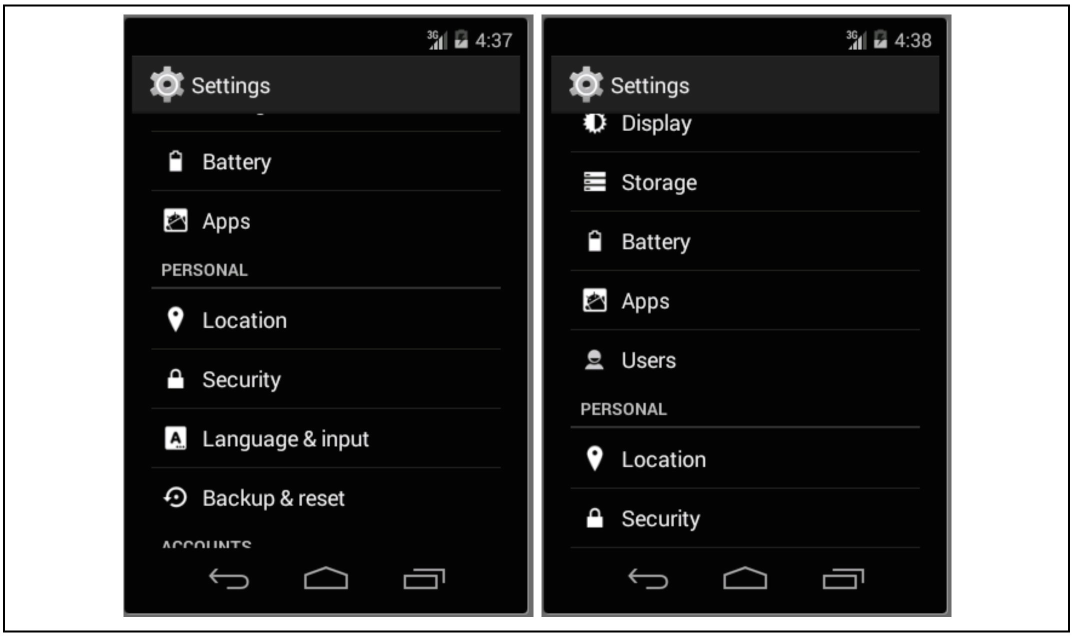
For the majority of its existence, Android has operated under the assumption that the device only has one user. Unlike desktop systems, which have long allowed user login and switching, this feature was only introduced into Android with JellyBean (4.2), and has been initially introduced only into tablets.

Android already uses the user IDs for the individual applications, as explained previously. To implement multi-user support, it builds on the same concept, by carving up the AID space into non-overlapping regions, and allocating one of every human user. Application IDs are thus renamed from `app_###` to `u##_a###`, and users are created with separate directories in `/data/user`. Application data directories are moved to `/data/user/##/`, with the primary user being user "0". The legacy `/data/data` thus becomes the primary user's directory (symlinked from `/data/user/0`). The user profiles themselves are stored in `/data/system/users`. This is shown in the following experiment:

Experiment: Enabling multi-user support on API 17 and later

On tablets, multi-user support will be enabled by default as of JellyBean (API 17). A little known feature, however, is that you can enable it on phones as well. All it takes is setting a system property - `fw.max_users` to any value greater than 1. Doing so on the Android emulator will bring up the "Users" option to settings, as shown in the following screenshot:

Screenshot 8-1: Before and After `fw.max_users` property modification



Adding a user is straightforward, though the system will force you to set a lock screen, in order to differentiate between the two users on login. The process should look something like Output 8-14 (next page)

Experiment: Enabling multi-user support on API 17 and later (cont)

Output 8-14: Listing multiple user profiles in /data/system/users

```
root@generic:/data/system/users # ls -F
0/
0.xml
userlist.xml
root@generic:/data/system/users # setprop fw.max_users 3
#
# Add another user through settings.. (shell stop/start might be necessary) then ls again
#
root@generic:/data/system/users # ls -F
0/
0.xml
10/
10.xml
userlist.xml
root@generic:/data/system/users # cat userlist.xml
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<users nextSerialNumber="11" version="4">
    <user id="0" />
    <user id="10" />
</users>
root@generic:/data/system/users # cat 0.xml # Display details for user 0
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<user id="0" serialNumber="0" flags="19" created="0" lastLoggedIn="1400702272027"
      icon="/data/system/users/0/photo.png">
    <name>Owner.com</name>
    <restrictions />       # restrictions, if any, go in this element
</user>
root@generic:/data/system/users # ls -l 0   # Show settings for user 0
-rw-rw---- system system  ... accounts.db                # External (POP3, IMAP, etc) accounts
-rw------- system system  ... accounts.db-journal        # SQLite3 journal
-rw------- system system  ... appwidgets.xml             # Installed widgets
-rw-rw---- system system  ... package-restrictions.xml # Package Restrictions
-rw------- system system  ... photo.png                  # User selected photo
-rwx------ system system  ... wallpaper                  # Selected wallpaper
-rw------- system system  ... wallpaper_info.xml         # MetaData
```

From the command line, the same effect can be achieved by using `pm create-user`, which connects to the user manager (`IUserManager.Stub.asInterface(ServiceManager.getService("user"))`) and invokes its `createUser()` method. The `pm remove-user` likewise removes a user.

Depending on how you create the user (as a separate user or a restricted user, which shares the original user's apps), the `restrictions` element may be populated with the following boolean attributes, all defined in the `android.os.UserManager` class. The actual handling of the files (above) and the restrictions is performed by `com.android.server.pm.UserManagerService`.

Table 8-10: User Restrictions

Restriction
no_modify_accounts
no_config_wifi
no_install_apps
no_uninstall_apps
no_share_location
no_install_unknown_sources
no_config_bluetooth
no_config_credentials
no_remote_user

Key Management

Android relies extensively on cryptographic keys, for system internal use (validating installed packages) and for application use. In both cases, the keystore service (discussed in Chapter 4) plays an integral part in abstracting and hiding the implementation.

Certificate Management

Public Key Infrastructure is the de-facto fulcrum of all Internet security. Encryption rests on several key assumptions which relate to the algorithms and methods behind public keys, the most important of which is a *trust*. Simply put, this means that if you know a subject's public key, the key can be used not just for encrypting messages to it, but also authenticating messages from it. This, in turn, means that if this subject vouches for another public key by authenticating it (which is, in effect, what a certificate is), then that public key's ownership can be established. In this way, a *trust hierarchy* can be formed.

This principle, while powerful, does lead to a chicken and egg problem - you can authenticate a public key only if some other public key has been a priori used to authenticate it. The way around this predicament is to hard code the initial public keys in the operating system. These keys are encoded in the form of *root certificates* - public keys authenticating themselves. When passed over the network, they are of no value (as they are trivial to spoof). When hard-coded, however, they can be trusted and provide the basis for the trust hierarchy.

Android hard-codes root certificates in /system/etc/security/cacerts. The certificates are encoded in their PEM (Privacy-Enhanced-Mail) form, which is a Base64 encoding of the certificate between delimiters. Some devices will also have the plain ASCII form of the certificate before or after the PEM encoding. If not, it's a simple matter to display it using the `openssl` command line utility, which is built-in to Linux or Mac OS, shown in output 8-15:

Output 8-15: Using `openssl` to decode a PEM certificate

```
morpheus@Forge (/tmp)$ adb pull /system/etc/security/cacerts
pull: building file list...
pull: /system/etc/security/cacerts/ff783690.0 -> ./cacerts/ff783690.0
..
morpheus@Forge (/tmp)$ openssl x509 -in ff783690.0 -text | more
Certificate:
    Data:
        Version: 3 (0x2)    # Denotes the X.509v3 format
        Serial Number:      # Used to refer to certificate when revoking
            44:be:0c:8b:50:00:24:b4:11:d3:36:2a:fe:65:0a:fd
        Signature Algorithm: sha1WithRSAEncryption
        Issuer: # Issuer in LDAP notation: C=country, ST=state, L=location,
                # O=Organization, OU= Organizational Unit, CN=Common Name
        Validity
            Not Before: # Usually coincides with certificate issue date
            Not After : # Usually set to 2-10 of years from issue date
        Subject: # Certificate Owner, in same LDAP notation
                 # ...
        Subject Public Key Info:
         .. # Modulus and Exponent (usually 65537)
        X509v3 extensions:
            X509v3 Key Usage:
                Digital Signature, Non Repudiation, Certificate Sign, CRL Sign
            X509v3 Basic Constraints: critical
                CA:TRUE
            X509v3 Subject Key Identifier:
                A1:72:5F:26:1B:28:98:43:95:5D:07:37:D5:85:96:9D:4B:D2:C3:45
            X509v3 CRL Distribution Points:
                URI:http://crl.usertrust.com/UTN-USERFirst-Hardware.crl
            X509v3 Extended Key Usage:
                TLS Web Server Authentication, IPSec End System, IPSec Tunnel, IPSec User
  Signature Algorithm: sha1WithRSAEncryption
        # .. SHA-1 hash of certificate, signed with RSA private of issuer
-----BEGIN CERTIFICATE-----
MIIEdDCCA1ygAwIBAgIQRL4Mi1AAJLQR0zYq/mUK/TANBgkqhkiG9w0BAQUFADCB
lzELMAkGA1UEBhMCVVMxCzAJBgNVBAgTAlVUMRcwFQYDVQQHEw5TYWx0IExha2Ug
...  Base 64 (original PEM) encoding of the certificate
KqMiDP+JJn1fIytH1xUdqWqeUQ0qUZ6B+dQ7XnASfxAynB67nfhmqA==
-----END CERTIFICATE-----
```

Of special importance are the Over-The-Air (OTA) update certificates, stored in the /system/etc/security/otacerts.zip archive. The archive usually contains one (rarely, more) certificates which are used for validating OTA updates (described in Chapter 3). The RecoverySystem class parses this file (hardcoded as DEFAULT_KEYSTORE), in its getTrustedCerts() method using a CertificateFactory. Once again, any certificates would be encoded in PEM (usually, without human readable text), but you can use the method shown in output 8-13 to decode them. Removing this file is a good method to "combat" auto-updates in some Android distributions (such as FireOS), which may cause you to lose root access post-update.

Certificate Pinning

JellyBean (API 17) introduces *certificate pinning*, which has become a common add-on to SSL certificate validation. Pinning involves hard-coding the expected public key of a host (via its certificate), so that if the host presents a certificate which does not match the pin (or one of the pins in a pin set) it is rejected.

Unlike the certificates discussed previously, which are in /system/etc/security (and therefore cannot be modified), pins are maintained in /data/misc/keychain/pins, which is a file that can be replaced. The CertPinInstallReceiver class registers a broadcast receiver for the UPDATE_PINS intent, and - when such an intent is received, its extras are expected to contain the following:

- **EXTRA_CONTENT_PATH**: The file name containing the new pins.
- **EXTRA_VERSION_NUMBER**: Which is expected to be greater than the current version.
- **EXTRA_REQUIRED_HASH**: Of the current pins file.
- **EXTRA_SIGNATURE**: Signature of the file supplied, its version and hash of current pins file

The CertPinInstallReceiver's onReceive (inherited from ConfigUpdateInstallerReceiver) gets the values from the broadcast intent, ensures the version number is indeed greater than the current version of the pins file (in /data/misc/keychain/metadata/version), and that the current file's hash matches the hash specified in the intent. It then verifies the signature, using the certificate stored in the system settings database under config_update_certificate (the UPDATE_CERTIFICATE_KEY). If everything is in order, the filename from the intent is copied over the existing pins file, and the metadata/version is updated to reflect the new version number.

Google pins all of its (many) certificates by default, and the vendor may pin additional ones. A quick way of looking at pins is shown in Output 8-16:

Output 8-16: Displaying the pinned domains

```
root@htc_m8wl:/ # cat /data/misc/keychain/pins | cut -d"=" -f1
*.spreadsheets.google.com
*.chart.apis.google.com
appengine.google.com
*.google-analytics.com
*.doubleclick.net
*.chrome.google.com
*.plus.google.com    # largely unused ;-)
# ......
*.youtube.com
*.profiles.google.com
*.mail.google.com
www.googlemail.com
gmail.com
```

The Android Explorations Blog[7] contains a sample application demonstrating the creation of a pins file and its update operation through the intent.

Certificate Blacklisting

Android provides the `CertBlacklister` class to handle black listing (effectively, revocation) of certificates. The class (instantiated as a service of `system_server`, as discussed in Chapter 5) registers an observer for two content URIs:

- content://settings/secure/pubkey_blacklist: Stores known compromised or revoked public keys or certificates. Content written here ends up written to /data/misc/keychain/pubkey_blacklist.txt.
- content://settings/secure/serial_blacklist: Stores known compromised or revoked serial numbers of certificates. Serial numbers written here are saved to /data/misc/keychain/serial_blacklist.txt.

Both values are also in the system's secure settings, as can be seen in the following output:

Output 8-17: Viewing the serial and pubkey blacklists

```
root@htc_m8wl:/ # sqlite3 /data/data/com.android.providers.settings/databases/settings.db \
                  "select * from secure" | grep black
95|serial_blacklist|827,864
99|pubkey_blacklist|5f3ab33d55007054bc5e3e5553cd8d8465d77c61,783333c9687df63377efceddd82efa..
root@htc_m8wl:/ # cat /data/misc/keychain/serial_blacklist.txt
827,864
root@htc_m8wl:/ # cat /data/misc/keychain/pubkey_blacklist.txt
5f3ab33d55007054bc5e3e5553cd8d8465d77c61,783333c9687df63377efceddd82efa9101913e8e
```

Secret and Private Key Management

Storing secrets - symmetric keys or the private part of a public-key pair - poses serious challenges for any security infrastructure. If one assumes that file permissions are a strong enough layer of security, the secrets can be placed in a file and appropriately locked down. The underlying file permissions of Linux, however, are inflexible, and configuration errors could lead to secret leakage. Likewise, there is the problem of obtaining root access - which effectively voids all permissions, leaving everything in the clear.

Android provides access to secrets via the `keystore` service. This service has already been discussed in Chapter 4. Keystores for applications are maintained on a per-user basis, in the /data/misc/keystore/user_## directory, but applications have no direct access to that directory, and must go through the keystore service, which is the sole owner of the directory (permissions 0700). The service also provides public key functions - `generate`, `sign` and `verify` - without allowing applications any access to the underlying private keys. This allows the key storage to be potentially implemented in hardware.

Indeed, Android offers hardware backed secure storage, on those devices which support it, as of JellyBean. As discussed in Chapter 11, the `keymaster` HAL abstraction provides both a uniform interface for encryption operations, and allows its implementation in both software and hardware. Thus, supporting devices implement a hardware backed keymaster module, whereas those which do not use a `softkeymaster` instead.

Storage Security

/data Encryption

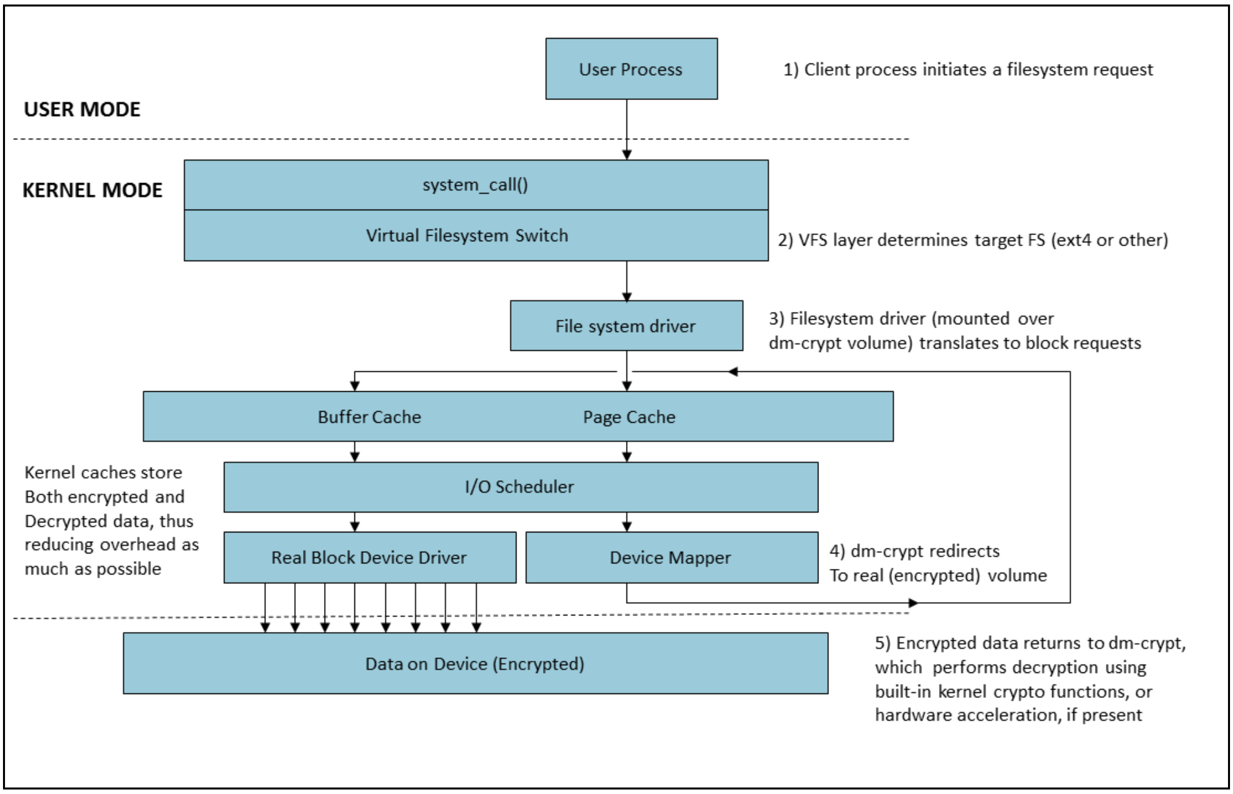
While most users remain oblivious to the need for encryption on their devices, corporate users certainly fear the compromising of data which would ensue should a device be lost or stolen. iOS provided transparent encryption as of iOS 4, and coincidentally, so has Android as of Honeycomb. By using the very same dm-crypt mechanism utilized by OBBs and ASEC, Honeycomb extends the notion of encryption to the full filesystem layer. The term "full disk encryption" is therefore somewhat inaccurate here, since it is only the /data partition which is normally encrypted. This actually makes more sense, because /system contains no sensitive data (and would be impacted from the latency incurred by crypto-operations.

Android's documentation provides a detailed explanation of encryption, which has been revised for Android L[8] . As with ASECs and OBBs, the volume manager is responsible for performing both the filesystem encryption and decryption. The former is performed when selected by the user, and is a rather lengthy operation. The latter is performed transparently, when the encrypted filesystem is mounted as a block device using the device mapper.

Note, that unlike obb and asec - the decryption keys for which are stashed somewhere on the device in plaintext, albeit readable only by root - the key for the /data partition encryption does not actually reside on the device, but requires the user to interact during boot, and supply it (or, more accurately, the password from which this key is derived). This requires modifications to the Android boot process, as well as an interaction between init and vold, which we describe in Chapter 4

Prior to the dm-crypt solution, there were several proposed alternatives for file system encryption (most notably EncFS by Wang et Al.[9]), but the dm-crypt one is the de facto standard, now that L has enabled it by default. The architecture is shown in Figure 8-4:

Figure 8-4: The DM-Crypt Architecture


User Process
1) Client process initiates a filesystem request
USER MODE
KERNEL MODE
system_call()
Virtual Filesystem Switch
2) VFS layer determines target FS (ext4 or other)
File system driver
3) Filesystem driver (mounted over dm-crypt volume) translates to block requests
Buffer Cache
Page Cache
Kernel caches store Both encrypted and Decrypted data, thus reducing overhead as much as possible
I/O Scheduler
Real Block Device Driver
Device Mapper
4) dm-crypt redirects To real (encrypted) volume
Data on Device (Encrypted)
5) Encrypted data returns to dm-crypt, which performs decryption using built-in kernel crypto functions, or hardware acceleration, if present

Android M (PR1) further employs dm-crypt with a new feature called "adoptable storage", which enables the user to extend Android filesystem encryption to external storage (e.g. USB drivers). As usual, this is handled by vold, who maintains the encryption keys in /mnt/vold, and mounts the decrypted volumes under /mnt/expand.

Performance Impact

An oft asked question pertains to the potential performance impact of encryption. Encryption naturally requires more processing by the CPU (to decrypt and re-encrypt the data), which can impact performance and, to an extent, power consumption. While there have been differing accounts, the overall view is that the performance impact ranges from negligible to manageable. This is corroborated by the following:

- **Access to the storage device is already inherently slow:** While not as slow as hard drives, flash devices run at significantly slower rates than the CPU. Adding the overhead of an encryption or decryption routine adds several more microseconds per access, but when viewed percentage-wise, this accounts for a fractional gain, at best.
- **The Linux kernel optimizes access with caching:** As can be seen in figure 8-4, the Linux kernel helps optimize data access by caching device data. Because dm-crypt appears as a block device, it exists under the caches, and therefore can benefit from it: Data is decrypted only once, and read/write operations can occur on the cached (decrypted) copy. When the data is flushed back to the underlying device, it can be re-encrypted, and then find its way to the underlying physical device.
- **To begin with, access to /data isn't as often as to /system:** Unlike access to the /system partition, which stores Android's vast frameworks and static configuration, access to /data occurs only when an app is loaded, or some runtime configuration change occurs.

Secure Boot

KitKat introduced a new feature for securing the boot process, using the kernel's device mapper. This feature, known as **dm-verity** originated in Chromium OS, and has been ported into Linux (and thus Android), beginning with kernel version 3.4.

Recall from Chapter 3, that a chain of trust (also known as the *verified boot path*) has been established from the ROM, via the boot loader, and onto the kernel and the root file system (i.e. the boot partition). While the bootloader actually does verify /system, it does so only when flashing the entire partition - which leaves open the avenue for a root owned process (be it "rooting" or malware) to make persistent changes in /system, by remounting it as read-write, and modifying files in it. Using dm-verity effectively extends the boot chain of trust one more level, onto /system.

Verifying the integrity of a partition is the simple matter of hashing all of its blocks (DM-Verity uses SHA-256), and comparing that hash against a stored, digitally signed hash value. To do so effectively, however, one has to avoid the lengthy process of reading the entire partition, which can delay boot. To get around this limitation, dm-verity reads the entire partition only once,and records the hash value of each 4k block in the leaf nodes of the tree. Multiple leaf nodes are rehashed in the second level of the tree, and then onward to the third, until a single hash value is calculated for the entire partition - this is known as the **root hash**. This hash is digitally signed with the vendor's private key, and can be verified with its public key. Since disk operations are performed in full blocks, it is a straightforward to add an additional hash verification on the block as it is placed into the kernel's buffer/page cache, and before it is returned to the requester. If the hash check fails, an I/O error occurs, and the block is known to be corrupted.

The dm-verity feature is touted for malware prevention, since it effectively prevents any modification of /system, but does have the side effect of preventing unauthorized persistent rooting, as well. Malware could definitely attempt to make modifications to /system, but Android would detect them, potentially refusing to boot - yet the same would apply for any "persistent root" back door, e.g. dropping a SetUID /system/xbin/su. From the vendor's perspective, this is fine - most vendors would only provide root via bootloader unlocking, which breaks the chain of trust at its very first link. Further, dm-verity requires only a subtle modification to the update process (discussed in Chapter 3) - namely, that the vendor regenerate the signature when /system is modified during an update. Otherwise, /system remains read only throughout the device's lifetime, and the signature must therefore remain intact.

The kernel mode implementation of dm-verity is rather small - a 20k file of drivers/dm/dm-verity.c, which plugs into the Linux Device Mapper (as discussed in Volume III). Google details the verified boot process in the Android Documentation[10]. The Android Explorations Blog[11] once more provides further detail, including using the veritysetup during the building of the image.

Rooting Android

Most vendors provide ADB functionality on their devices and leave the operating system relatively open for developers, but few (if any) provide root access to the device. There is a strong rationale not to do so, considering that obtaining root access to a UNIX system brings with it virtual omnipotence - and Android is no different. Leaving behind open access to root would also potentially provide an attack vector for malware (which Android knows no shortage of). With root access, any file on the system could be read, or - worse - overwritten, which would give an attacker both access to private data, as well as the ability to hijack control of the device.

The same can be said for Apple's iOS (also a UNIX system, based on Darwin), but herein lies the significant difference between the two. Apple's developers have engineered the system from the ground up, literally, from the very hardware to the uppermost layers of software, to be rock solid and not to allow root access (in fact, not to allow *any* access aside from a sandboxed app model) at all costs. Android is built on Linux, which itself is a mix of code strains from different contributors, not all of which adhere to the strictest security standards. Additionally, several vendors leave an avenue, which can be exploited (by a human user in possession of the device) to gain root access - redirecting the system to boot an alternate configuration. Another way of looking at it is, Android considers the application to be the enemy - whereas iOS considers the user itself to be one.

Boot-To-Root

When Android devices boot, they normally do so by the process described in Chapter 3. It is possible, however, to divert the boot process to an alternate boot, for "safe" boot, system upgrade, or recovery. This can usually be done by pressing a physical button combination (usually one or both of the volume buttons, and the home button, if it exists), or by a fastboot command, when the device is connected over USB. Once the boot flow is diverted, the boot loader can be directed to load an alternate boot image - either the on-flash recovery image, an update supplied on the SD-card, or (over USB) an image supplied through fastboot.

If a device's bootloader can be unlocked (as explained in Chapter 3) the device can be rooted. It's that simple. As previously mentioned, unlocking the boot loader will cause /data to be effaced, in an effort to prevent the user's sensitive data from falling into the wrong hands. Additionally, some boot loaders will permanently set a flag indicating that the loader has been tampered with, even if it is re-locked at some point. This is to note that the boot loader basically shirks all responsibility for system security, as it will no longer enforce digital signatures on images flashed.

All it takes to "root" the device is really just one part of the device image - the init RAM disk (initramfs). Because the kernel mounts the initrd as the root filesystem and starts its /init with root privileges, supplying an alternate /init - or even just a different /init.rc file - suffices to obtain root access. From that point onwards, it's a simple matter of convenience: It's straightforward to simply have ADB maintain root privileges (by setting `ro.secure=0`*) or replace adb to a version which doesn't drop privileges. Most rooting tools, however, usually drop a su binary into /system/bin or /system/xbin, and use `chmod 4755` to toggle the setuid bit, so when it is invoked from the shell, the setuid effect will kick in, and automatically bestow root permissions. The code for such a binary (pre Kit-Kat) is so simple it can be summarized in three functional lines:

Listing 8-5: A simple implementation of **su**, for non SE-Linux enforced devices

```
#include <stdio.h>
void main(int argc, char **argv)
{
        setuid(0);
        setgid(0);
        system("/system/bin/sh");

}
```

* - In recent builds, adb is conditionally compiled (#ifdef ALLOW_ADB_ROOT) so as to ignore this property.

You can find a similar implementation (with command line options) in the AOSP's /system/extras/su/su.c. As of KitKat, however, the introduction of SE-Linux in enforcing mode makes the binary less trivial, in that its parent (the shell) is already confined to a restricted execution context (`u:r:shell:s0`), which it cannot break out of. This requires the `su` binary to make an IPC call to a process in the `u:r:init:s0` (or `u:r:kernel:s0`) unrestricted context, to then spawn a shell (e.g. the WeakSauce exploit (with DaemonSu), as explained on the book's companion website[12]).

If you've rooted a KitKat (or later) device with SE-Linux in enforcing mode, you can likely see this for yourself, as shown in the following output:

Output 8-18: viewing an `su` implementation, accommodating for SELinux

```
#
# Starting off with a non-privileged shell, get PID and UID:
shell@htc_m8wl:/ $ echo $$
6498
shell@htc_m8wl:/ $ id
uid=2000(shell) gid=2000(shell) groups=1003(graphics),1004(input),1007(log),1009(mount),
1011(adb),1015(sdcard_rw),1028(sdcard_r),3001(net_bt_admin),3002(net_bt),3003(inet),
3006(net_bw_stats) context=u:r:shell:s0
#
# Switch to a root shell, do same (i.e. get PID and UID):
shell@htc_m8wl:/ $ su
root@htc_m8wl:/ # echo $$
6503
root@htc_m8wl:/ # id
uid=0(root) gid=0(root) context=u:r:init:s0
# Use the toolbox specific -Z flag to ps, to show SELinux contexts
root@htc_m8wl:/ # ps -Z
# Note the "su" is the child of the shell (in this case, 6498), but has no children itself.
# the actual shell spawned by su is started from a daemonsu instance, which gets the u:r:init
# unrestricted SE-Linux context from daemonsu. eu.chainfire.supersu is the GUI app.

u:r:shell:s0                   shell     6498  601   /system/bin/sh
u:r:shell:s0                   shell     6503  6498  su
u:r:init:s0                    root      6506  5319  daemonsu:0:6503
u:r:init:s0                    root      6510  6506  tmp-mksh
u:r:untrusted_app:s0           u0_a140   6528  575   eu.chainfire.supersu
u:r:init:s0                    root      6578  6510  ps
```

The practice of rooting is so popular that there are quite a few "SuperUser" applications, which provide a GUI interface to administer root access, once the device is rooted. The applications actually offer a programmatic API (via permissions and intents) to allow other applications access to root. One noteable example is chainfire's SuperSU, which defines its own Dalvik level permissions (`android.permission.ACCESS_SUPERUSER` and `eu.chainfire.supersu.permission.NATIVE`) and enables applications to broadcast intents in order to obtain superuser privileges. This application also cleverly works around SE-Linux, as can be seen from the output above.

Rooting via Exploiting

Whether or not a vendor has left the boot-root backdoor open, often there exist additional backdoors. These, unlike the former, are quite unintentional, and all rely on some form of system vulnerability exploitation. The ways to do so are myriad, and often unpredictable until discovered, but they all share the same common denominator: Find some insecure configuration setting or software component, and trigger some code path, by means of which root access can be obtained. As mentioned in the threat modeling section of this chapter, the security jargon for these attack types is **privilege escalation**, as it refers to the process wherein a lower privilege process (that is, some app), can increase its privileges, usually first to those of the system user, and then root.

There is a strong similarity between exploit-based rooting methods and "jailbreaking" for iOS. In both cases, it takes the discovery and exploitation of software bugs, and both methods *should not* be possible in a perfect world (at least, according to Google and Apple). Once these methods are discovered, their days are numbered: The operating system is fairly quickly patched, and suggested to the user for download and updated (or even auto-updated, as for example with the Amazon Kindle). One prominent example was in Gingerbread, wherein Google itself pushed an update for a vulnerability in the Linux kernel, known at the time to have been actively exploited by malware.

A thorough discussion of exploitation techniques is thus beyond the scope of this work, and quite frankly is pointless, since all known exploits at this time have been patched. Exploits generally obtain root by passing crafted input to a process already running as root (vold has been a perennial favorite..), corrupting its memory (stack or heap) and usually overwriting a function pointer (or, commonly, a return address) to subvert its execution, and direct it at the attacker-controlled input. An additional trick - Return Oriented Programming (ROP) is often used to direct execution to snippets of code which already exist in the program, but run in an attacker controlled manner. This method, which is somewhat like biological DNA splicing and recombination, defeats data execution prevention methods, such as ARM's XN bits. A lengthy discussion of past exploits and ROP methods can be found in the Android Hacker's Handbook.

It should be noted that not all exploits necessarily involve code injection - some are much more simple and elegant (for example, the "WeakSauce" exploit for HTC One phones, discssued in the book's companion website[12]). Similarly, the latest vulnerability in Android at the time of writing was not really due to Android - but to the Linux kernel. Geohot's clever "Towelroot" exploit[14] used a well known kernel bug in handling fast mutexes (CVE-2014-3153) to gain root. While TowelRoot itself is not malware per se but a rooting utility, malware could use the exact same bug to surreptitiously gain root access, without the user's knowledge or consent.

To paraphrase a quote attributed to Donald Rumsfeld - there are "known unknowns" - those are essentially the 0-days which were unknown, but have been discovered - and patched - but there are also "unknown unknowns". The latter are the 0-days which are likely to exist, but have not been discovered yet, or - worse - have been discovered, but not publicized yet. Any hacker uncovering a 0-day in effect obtains a skeleton key to all Android devices vulnerable to that particular issue. A malicious hacker can incorporate this into powerful malware, or not even bother, and directly sell it on the open market. Though not as lucrative as iOS exploits, Android 0-days can fetch anywhere between $50,000 and $500,000 dollars - depending on vector (local/remote) and impact.

Security Aspects of Rooting

Because a boot-based rooting method requires user intervention, and/or connecting the device to a host, it is generally not considered to be an insecurity of the Android system. It does, however, leave a clear attack vector for an adversary who gains possession of the device. This could be an issue if the device is lost, stolen, or just left outside one's reach for a sufficient amount of time. It would take a skilled attacker no more than 10-20 minutes to root a device, steal all the personal data from it, and leave a backdoor or two. This is why most bootloaders are often locked, and while an unlock of the bootloader is possible, it will force a factory reset and erasure of all personal data - Once the bootloader is unlocked, however, the device *is* vulnerable (unless the bootloader is locked again).

Exploitation attacks are even simpler in the sense that they do not require the user to manually divert the system boot process. In fact, these attacks require no user intervention at all. Therein lies their advantage (for those looking for a simple "1-click" root method), but also their great risk, as they can be carried out without the user's knowledge, often when installing a seemingly innocuous app, which like the proverbial Trojan horse compromises the entire system.

Explotation attacks are even more dangerous when they are HTTP-borne. When the vulnerability exploited, or part thereof, involves the browser, it suffices to visit a malicious website - or inadvertently access some content from it (for example, through an ad network), for malicious payload to target the browser, and gain the initial foothold on the device. Indeed, sophisticated malware consists of multiple payloads injected over several stages, initially obtaining remote execution, then followed by obtaining remote root.

What follows is that rooting the device can, in fact, be dangerous, if not carried out through trusted sources: When an eager user downloads a rooting utility, whether one-click or tethered, if the download source is not a trusted one, it could be hard - virtually impossible - to detect additional payloads or backdoors which may be injected by such utilities. Less than proper tools may jump on the chance to also change system binaries or frameworks, for example disabling the Dalvik permission mechanism for malware purposes. Malware could possibly inject a rootkit all the way down to the Linux kernel, though most would probably not put that much effort when it's fairly trivial to hack the higher layers. Somewhat ironically, some of the SuperUser applications themselves had vulnerabilities in the past, which enabled rogue applications to detect a rooted device, and escalate their own privileges through the applications (q.v. CVE-2013-6774).

The last, but hardly least impact of rooting a device one has to consider is that on applications - Android's Application content protections disintegrate on a rooted device: OBBs can be read by root, as can the keys to ASEC storage. Application encryption likewise fails, and though hardware backed credential storage offers some resistance, its client processes' memory can easily be read (via `ptrace(2)` methods and the like). DRM solutions also fail miserably. Unfortunately, there's no foolproof way of detecting a rooted device from a running application, and refusing to execute on one.

Arguably, the same can be said for Jailbroken iOS - after all, Apple's fairplay protections and application encryptions, though stronger than Android's, are equally frangible. Yet one has to keep in mind that iOS only has an exploitation vector (with an ever increasing level of difficulty in between releases), whereas most Android devices do allow Boot-to-Root. Coupled with the ease of Dalvik bytecode decompilation, this poses a serious concern for application developers.

Summary

This chapter attempted to provides an overview of Android's myriad security features, both those inherited from Linux, and those which are specific to Android and mostly implemented in the Dalvik level. Special attention has been given to the Android port of SELinux - which, though currently not in full effect, is already adopted by Samsung in KNOX, and is likely to play a larger part in upcoming releases of Android.

While trying to be as detailed as possible, this review is by no means comprehensive. The interested reader is referred to Android Security specific books, such as Nikolay Elenkov's Android Security Internals[15], which devotes full chapters to what was covered here in sections.

References

1. a. SEAndroid - A Paper, Smalley/Craig: http://www.internetsociety.org/sites/default/files/02_4.pdf
   b. SEAndroid - A Presentation, Smalley/Craig: http://www.internetsociety.org/sites/default/files/Presentation02_4.pdf
   c. SEAndroid at ABS - Smalley/Craig: http://events.linuxfoundation.org/sites/events/files/slides/abs2014_seforandroid_smalley.pdf
2. Android Developer, SELinux: http://source.android.com/devices/tech/security/se-linux.html
3. RedHat, RHEL6 and SELinux: https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Security-Enhanced_Linux/
4. RiskIQ, Mobile Apps on Google Play: http://www.riskiq.com/company/press-releases/riskiq-reports-malicious-mobile-apps-google-play-have-spiked-nearly-400
5. a. BlueBox, Android "Master Key Vulnerability": https://bluebox.com/technical/uncovering-android-master-key-that-makes-99-of-devices-vulnerable/
   b. Saurik, Android "Master Key Vulnerability": www.saurik.com/id/17
6. BlueBox, Android "Fake ID Vulnerability": http://bluebox.com/technical/android-fake-id-vulnerability
7. Android Explorations, Certificate Pinning in 4.2: http://nelenkov.blogspot.com/2012/12/certificate-pinning-in-android-42.html
8. Android Documentation, Device Encryption: https://source.android.com/devices/tech/encryption/index.html
9. EncFS, by Wang et Al: http://cs.gmu.edu/~astavrou/research/Android_Encrypted_File_System_MDM_12.pdf
10. Android Documentation, DM-Verity: https://source.android.com/devices/tech/security/dm-verity.html
11. Android Explorations, KitKat Verified Boot: http://nelenkov.blogspot.com/2014/05/using-kitkat-verified-boot.html
12. NewAndroidBook.com, Analyzing the WeakSauce Exploit: http://newandroidbook.com/Articles/HTC.html">
13. Android Hacker's Handbook, Wiley 2014, by Joshua Drake and others
14. Towelroot.com, http://www.towelroot.com
15. Android Security Internals, No Starch 2014, by Nikolay Elenkov

Had enough? If this has only begun to whet your appetite for more on Android - stay tuned for Volume II - coming soon, which picks up where this leaves off, and discusses the true internals of the system: The framework services, graphics, audio, and multimedia, and much more - from the programmer's view!

Feel free to drop me a line and let me know what you liked, and what you hated! Also remember to check out NewAndroidBook.com for more updates, and tons of bonus material! Hope to see you next volume!

Android may be an open source system, but how many people can actually sit down and sift through millions lines of Java, C, C++ and XML, just to figure out how it works?

Android Internals::A Confectioner's Cookbook is the first time the inner workings of the world's most popular operating system have been documented! Without going into the lengthy code, it presents the logic and flow of Android's various components using detailed illustrations, verbose annotations and hands-on experiments! The companion website - http://NewAndroidBook.com/ - offers plenty of bonus material, in the form of special tools for free download, articles, and code samples.

Volume I takes the power user's point of view - the utilities and functionality accessible through `adb shell`. In particular, we explore:

- Partitions and Filesystems
- The Boot Process
- Init and its configuration files
- The native daemons in /system/bin
- The framework service architecture and `servicemanager`
- Monitoring through Linux interfaces
- Security

All versions of Android - up to and including Lollipop - are covered, with examples taken from the wide gamut of Android Devices - Nexi, Samsung Galaxy S series, NVidia Shield, Amazon Kindle, HTC One M8, and the Android Emulator.

This is the first in a multi-volume series, aiming to explore Android down to its last class. Stay tuned for Volume II - The Programmer's View - which picks up where the Power User's View ends, and dives deeper still into the frameworks, input, audio, video and network architecture... wading through the inevitable quagmire of code.

"The most detailed and up-to-date description of Android's architecture yet."

Nikolay Elenkov, "Android Security Internals" and Android Explorations Blog

Jonathan Levin is a longtime trainer and consultant specializing in the system and kernel levels of the "Big Three" - Windows, Linux and Mac OS X, as well as their mobile derivatives. He is the founder and CTO of Technologeeks.com, a partnership of experts offering training and consulting on system/kernel programming, debugging and more.

Fresh after his take on iOS in "Mac OS X and iOS Internals" (Wiley, 2012, with a 2nd edition coming in 2015), Jonathan turns his attention to the "Other Operating System" - and brings an even greater level of detail to the operating system that is to Mobile what Windows was to the Desktop. And this time, it's personal - this entire work is self-published.

http://www.NewAndroidBook.com
http://www.Technologeeks.com