ANALYSIS AND DEBUGGING OF OEM'S

A Thesis
Presented to the
Faculty of
San Diego State University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
in
Computer Science

by
Mohit Joshi
Spring 2012
SAN DIEGO STATE UNIVERSITY

The Undersigned Faculty Committee Approves the

Thesis of Mohit Joshi:

Analysis and Debugging of OEM's

[Signature]
Joseph Lewis, Chair
Department of Computer Science

[Signature]
Carl Eckberg
Department of Computer Science

[Signature]
Robert Malouf
Department of Linguistics and Asian/Middle Eastern Languages

April 26, 2012
Approval Date
Copyright © 2012
by
Mohit Joshi
All Rights Reserved
DEDICATION

I dedicate this Master’s Thesis to my one and only one my Teacher my Guru Dr. Joseph Lewis, who helped me in this project with his ocean like patience and calm nature to achieve me this great moment.
ABSTRACT OF THE THESIS

Analysis and Debugging of OEM’s
by
Mohit Joshi
Master of Science in Computer Science
San Diego State University, 2012

Application layer, Middle layer, Radio Interface layer and Modem are the majorly most common terminologies we come across on smart phone’s platform. The thesis here delves into such layers to analyze and debug panics caused due to kernel panics, Bus lock up issue, Memory leak Android, Dalvik abort with Excessive JNI references, GPU Hang issue, Page Faults, ASIC chip corruption and many more.

The core logic here is not just to restrict the end user to know about some application layer dis-functionalities causing weird activities happening in the handset, but to make them to the core where we have all kinds of framework logic of subsystems, runtime framework and supported libraries which enhances the knowledge domain and educates the user to differentiate the bugs by layer specification.

We also show real time issues due to the Dalvik Virtual machine and how the supported libraries can also be the culprits.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
</tr>
<tr>
<td>CHAPTER</td>
</tr>
<tr>
<td>1  TARGETING ANDROID AND PLATFORM SPECIFIC LAYERS</td>
</tr>
<tr>
<td>- Research Objective</td>
</tr>
<tr>
<td>- Android Licensing</td>
</tr>
<tr>
<td>- Introducing Android</td>
</tr>
<tr>
<td>- The UI Subsystem</td>
</tr>
<tr>
<td>- The Connectivity</td>
</tr>
<tr>
<td>- The Shell Environment</td>
</tr>
<tr>
<td>- The Complete Phone (Application and Modem Layers)</td>
</tr>
<tr>
<td>- Activities</td>
</tr>
<tr>
<td>- Services</td>
</tr>
<tr>
<td>- Content Providers</td>
</tr>
<tr>
<td>- Broadcast Receivers</td>
</tr>
<tr>
<td>- Libraries</td>
</tr>
<tr>
<td>- Runtime Android</td>
</tr>
<tr>
<td>- Kernel – Linux</td>
</tr>
<tr>
<td>- Modem and Radio Interface Layer</td>
</tr>
<tr>
<td>2  PLATFORM LOGGING OVERVIEW</td>
</tr>
<tr>
<td>- Android Logging Levels</td>
</tr>
<tr>
<td>- The Log Class</td>
</tr>
<tr>
<td>- Filtering Log Output</td>
</tr>
<tr>
<td>- Some Key Points</td>
</tr>
<tr>
<td>- Controlling Log Output Format</td>
</tr>
<tr>
<td>- Android Logging Architecture</td>
</tr>
<tr>
<td>- The Android Logging System</td>
</tr>
</tbody>
</table>
The Basic Logcat ...................................................................................................11
Debugging - Userspace ..........................................................................................12
  Adb ...................................................................................................................12
  Dalvik Debug Monitor Server .........................................................................12
  Device or Android Virtual Device ...................................................................12
  Debugging Tips ................................................................................................12
  Dump the Stack Trace ......................................................................................13
Display Useful Info on the Emulator Screen .....................................................13
Get Application and System State Information from the Emulator .................13
Get Wireless Connectivity Information ...........................................................13
Log Trace Data ................................................................................................13
Log Radio Data ................................................................................................13
Capture Screenshots .........................................................................................13

3 DEBUGGING MECHANISM ..............................................................................14
  Classification of Logs ........................................................................................14
  Types of Logs .......................................................................................................14
    Logcat Logs: From Various Apps and System Level ......................................14
    Main Logs: The Main Application Log ...........................................................15
    Radio Logs .......................................................................................................16
    Kernel Logs ......................................................................................................16
    Event Logs .......................................................................................................16
    Bug Report: Combination of Several Logs .....................................................17
    Tombstones .....................................................................................................17
    Traces.txt ........................................................................................................17
    Serial Logs ......................................................................................................18
Commonly Seen Android Errors ...........................................................................18
Diving in One More Level of Logging ...............................................................18

4 THE ANDROID FUELING ............................................................22
  Dalvik Virtual Machine .....................................................................................22
  Why Has Google Chosen a Register-based VM? .............................................22
  Why Only Dalvik VM? ......................................................................................22
  Runtime Framework and Supported Libraries .................................................23
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Figure telling us that watchdog bite happened during specific sensors image loading period that’s caused modem rebooted and re-started whole system.</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>Represents the anatomy of the android RIL architecture.</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>Android logging architecture.</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>Figure showing a graphics distortion due to GPU hang.</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>Figure shows a force close seen while accessing the Hello Android app.</td>
<td>21</td>
</tr>
<tr>
<td>6</td>
<td>ANR.</td>
<td>21</td>
</tr>
<tr>
<td>7</td>
<td>Figure explains the android power cycle anatomy.</td>
<td>28</td>
</tr>
</tbody>
</table>
CHAPTER 1
TARGETING ANDROID AND PLATFORM
SPECIFIC LAYERS

RESEARCH OBJECTIVE

We always see some kind of unusual activities in our mobile phones; now these activities can be said as any kind of panic. In general words if we say my phone is not working smoothly that mean there could be potential issue to corruption in software binaries. The basic research objective of my thesis is to make the end user make aware of the unusual activities happening. To be more elaborate what is a force close? Why did it happen? What events were in the background? Why did it come only once? To answer all of these questions with evidence, we do triaging and analysis through logs.

The logs are basically a blue print of our cellular phone, If we have the logging on in our cell phones then we can know and keep track of every single activity in our cellular phone, but this does not seem to be as simple. As we know there are several features in a phone, related to GPS, Bluetooth accelerometer, Apps, Modem etc.

The analysis of a log varies accordingly. This thesis focuses on such areas and defines how logging can be useful for the feature developer to know what hidden bug is in its code. We are trying to explain user-space, kernel-space, driver-space, network level, ASIC level, chip level and modem specific issues as well as triaging technique to narrow down the root cause. We believe this will help in gaining understanding of areas not only with specific to android-java framework, but more in depth of all layers starting from the core kernel layer comprising of modem, baseband processor activities to user mode perspective i.e. the application layer.

Lastly this thesis does not end up just giving an analysis of the logs but also educates the user with the modem, network, driver, libraries and framework related event activities, by stating which layer of software stack the info could be grabbed. Also we are trying to prove for the user that bugs, issues, panics are not only found because of Java code errors, but also
due to other low level Assembly language C, C++. VLSI issues are also responsible for it and mostly written in software, firmware and hardware. In a broader view the framework errors are mostly due to major errors in code that do not align correctly with the hardware.

One should not forget the basic framework would remain the same but the prototype taken by OEM vendors would change by some additions or by tweaking it in such a way so that it is somewhat different from other OEM’s in terms of software, firmware and obviously hardware.

**ANDROID LICENSING**

This topic provides a brief history of Android, as we know it is open-source software created for mobile phones and other devices led by Google. Now very few people know that it was first developed by a small California-based firm called Android Inc. In July 2005, Google bought Android Inc. Basically, Android is relatively simple: it is a version of a Linux kernel. Google is still working on this and is continuing the platform’s development. On the 5th of November 2007, distribution of Google’s version was announced with the founding of the Open Handset Alliance [1]. Most of the Android platform is released under Apache 2.0 license. Google Android released the entire source code under an Apache license. With the Apache license means the software released under it can be freely downloaded and used for personal and commercial purposes. It allows user to make change to original software without having to contribute to the open source community.

**INTRODUCING ANDROID**

Android is a complete software stack for mobile devices which includes a new operating system, Linux kernel, middleware and have all those modern applications, which are available in any of modern mobile handsets (developed by mobile industry's giants like Nokia, Samsung, Sony Ericsson, etc.). Android also includes a set of core libraries that provide most of the functionality available in the core libraries of the Java programming language. Every Android application runs in its own process, with its own instance of the Dalvik virtual machines.
The UI Subsystem

Windows Views Widgets are used for displaying common elements such as edit boxes, lists, and drop-down lists. Android includes an embeddable browser built upon Web Kit, the same open source browser engine powering the iPhone's Mobile Safari browser.

The Connectivity

Android boasts a healthy array of connectivity options, including Wi-Fi, Bluetooth, and wireless data over a cellular connection (for example, GPRS, EDGE, and 3G).

The Shell Environment

Android’s Linux kernel–based OS does not come with a sophisticated shell environment, but because the platform is open, shells can be written and installed on a device. Likewise, multimedia codecs can be supplied by third-party developers and do not need to rely on Google or anyone else to provide new functionality. That is the power of an open source platform brought to the mobile market [2].

THE COMPLETE PHONE (APPLICATION AND MODEM LAYERS)

As mentioned, Android runs atop a Linux kernel. Android applications are written in the Java programming language, and they run within a virtual machine (VM). It's important to note that the VM is not a JVM as we might expect, but is the Dalvik Virtual Machine, an open source technology. Each Android application runs within an instance of the Dalvik VM, which in turn resides within a Linux-kernel managed process.

An Android application consists of one or more of the following classifications.

Activities

An application that has a visible UI is implemented with an activity, typically more self-contained in the communication with other programs as well as in time. When a user selects an application from the home screen or application launcher, an activity is started.
Services

A service should be used for any application that needs to persist for a long time, such as a network monitor or update-checking application. One might think of these as “running in the background,” in contrast with activities [3].

Content Providers

We can think of content providers as a database server. A content provider’s job is to manage access to persisted data, such as a SQLite database. Some sort of front-end may exist in any given app that uses content from that provider.

Broadcast Receivers

An Android application may be launched to process an element of data or respond to an event, such as the receipt of a text message [4].

An Android application, along with a file called AndroidManifest.xml, is deployed to a device. AndroidManifest.xml contains the necessary configuration information to properly install it to the device. It includes the required class names and types of events the application is able to process, and the required permissions the application needs to run.

Google usually refers to the Android OS as a software stack. Each layer of the stack groups together several programs that support specific operating system functions.

The base of the stack is the kernel. Google used the Linux version 2.6 OS to build Android's kernel, which includes Android's memory management programs, security settings, power management software and several hardware drivers.

Drivers are programs that control hardware devices. For example, the Nexus One has a camera. The Android kernel includes a camera driver, which allows the user to send commands to the camera hardware.

The next level of software includes Android's libraries. We can think of libraries as a set of instructions that tell the device how to handle different kinds of data. For example, the media framework library supports playback and recording of various audio, video and picture formats. Other libraries include a three-dimensional acceleration library (for devices with accelerometers) and a Webkit library for web browser application [5].
**Libraries**

This layer consists of Android libraries written in C, C++, and used by various
systems. These libraries tell the device how to handle different kinds of data that are exposed
to Android developers via Android Application framework. Some of these libraries includes
media, graphics, 3d, SQLite, web browser library etc.

The Android runtime layer that includes set of core Java libraries and DVM (Dalvik
Virtual Machine) is also located in same layer.

**Runtime Android**

This layer includes set of base libraries that are required for java libraries. Every
Android application gets its own instance of Dalvik virtual machine. Dalvik has been written
so that a device can run multiple VMs efficiently and it executes files in executable (.dex)
optimized for minimum memory [6].

**Kernel – Linux**

This layer includes Android’s memory management programs, security settings,
power management software and several drivers for hardware, file system access, networking
and inter-process-communication. The kernel also acts as an abstraction layer between
hardware and the rest of the software stack.

**MODEM AND RADIO INTERFACE LAYER**

Modems, which use a mobile telephone system (1x, CDMA, WCDMA, GSM, LTE,
GPRS, HSPA, EVDO, WiMax etc.), are known as wireless modems (sometimes also called
cellular modems). Wireless modems can be embedded inside a laptop or appliance or
external to it. The above network technologies are one of the foremost used in today’s
wireless industry [7]. Figure 1 shows a short snippet of a modem crash seen during dialing a
call. The mdm stands for the modem processor below, rmnet is the data channel and as we
see the kernel panic is not syncing with the subsystem or we can say a sub component of the
kernel, which fails to respond. Other useful aspects of these types of logs will be discussed
further.

Now lets start with some quick info on RIL (Figure 2).
Figure 1. Figure telling us that watchdog bite happened during specific sensors image loading period that’s caused modem rebooted and re-started whole system. We can flag lot of features and tag while collecting logs like process id, offset, time, timestamp, cycle, data address, data value, module and execution information to grab more meaningful information.

Android's Radio Interface Layer (RIL) provides an abstraction layer between Android telephony services (android.telephony) and radio hardware. The RIL is radio agnostic, meaning that its functioning does not depend on the particular details of the signal, and includes support for Global System for Mobile communication (GSM)-based radios.
Figure 2. Represents the anatomy of the android RIL architecture.
CHAPTER 2

PLATFORM LOGGING OVERVIEW

This chapter gives in depth knowledge of logging w.r.t Android. In this chapter we will see how the Android logging architecture is defined. This will let us see what are logs, what are the level of logs, what do the logs contain, what to look for in the logs, how to extract the logs, what info could be missing in a log, how to filter a log and many other answers to such questions will be resolved in this chapter.

ANDROID LOGGING LEVELS

The Android logging system provides a mechanism for collecting and viewing system debug output. Logcat dumps a log of system messages, which include things such as stack traces when the emulator throws an error and messages that we have written from were application by using the log class [8]. We can run logcat through ADB or from DDMS, which allows we to read the messages in real time.

The Log Class

Log is a logging class that we can utilize in our code to print out messages to the logcat. Common logging methods include:

- V (String, String) (verbose)
- D (String, String) (debug)
- I (String, String) (information)
- W (String, String) (warning)
- E (String, String) (error)

Filtering Log Output

Every Android log message has a tag and a priority associated with it. The tag of a log message is a short string indicating the system component from which the message originates. The priority is one of the following character values, ordered from lowest to highest priority:
V — Verbose (lowest priority): As the name suggest, the verbose a basic zero level
            can be considered that enables also the end user to understand what the log contains, this
            means in terms of event activities residing in the log.

D — Debug: This resembles to one higher version to view debugging info in the log.
This info really helps the developer to figure out the issue or identify the bug in short.

- I — Info
- W — Warning
- E — Error
- F — Fatal

The above options are more highly used to figure out issues for driver related bugs and
            merely into the lower layers.

S — Silent: (highest priority, on which nothing is ever printed).

Some Key Points

- Verbose should never be compiled into an application except during development.
- Debug logs are compiled in but stripped at runtime
- Error, warning and info logs are always kept

Controlling Log Output Format

Log messages contain a number of metadata fields, in addition to the tag and priority.
We can modify the output format for messages so that they display a specific metadata field.
To do so, we use the -v option and specify one of the supported output formats listed below.

1. Brief — Display priority/tag and PID of the process issuing the message (the default format).
2. Process — Display PID only.
3. Tags — Display the priority/tag only.
4. Raw — Display the raw log message, with no other metadata fields.
5. Time — Display the date, invocation time, priority/tag, and PID of the process
            issuing the message.
6. Thread time — Display the date, invocation time, priority, tag, and the PID and TID
            of the thread issuing the message.
7. Long — Display all metadata fields and separate messages with blank lines.
**ANDROID LOGGING ARCHITECTURE**

Figure 3 shows us briefly the information of the logging levels in Android. As we can see the Android utility involved, along with the exchange of the middle man (ADB) in the shell environment. Looking carefully at the screen shot we see the filters applied as discussed above for the main, radio, events etc. that happens at the core kernel level.

![Android Logging Architecture](image)

**Figure 3. Android logging architecture.**

- **android.util.Log**

  This API is basically used for sending log output Log.v() Log.d() Log.i() Log.w(), Log.e(), Log.f() and Log.s().

  The ADBD here refers to the daemon process involved in exchange of the shell commands between android logcat and the debugging bridge, this we see happening at run time.

**THE ANDROID LOGGING SYSTEM**

The Android logging system provides a mechanism for collecting and viewing system debug output.
Logs from various applications and portions of the system are collected in a series of circular buffers:

- /dev/log/main - default
- /dev/log/radio - radio/telephony related messages
- /dev/log/events - events-related messages
- /dev/log/system – system-related messages

`adb shell ls -s /dev/log:`

```
• crw-rw--w- root     log       10,  34 2011-02-24 08:40 system
• crw-rw--w- root     log       10,  35 2011-02-24 08:40 radio
• crw-rw--w- root     log       10,  36 2011-02-24 08:40 events
• crw-rw--w- root     log       10,  37 2011-02-24 08:40 main
```

The above command checks to see whether or not a log for the specified tag is loggable at the specified level. Before we make any calls to a logging method we should check to see if our tag should be logged [9].

We can change the default level by setting a system property:

- `setprop log.tag.<OUR_LOG_TAG> <LEVEL>`

A short example: Let us turn our attention to a command to our android shell:

- `adb shell setprop log.tag.ATCMD VERBOSE ATD#8581231234`

As we see above the input is: ATD#8581231234 and

The Output from log is:

```
02-24 18:06:00.328  4554  4944 D ATCMD:
```

In the above snippet example we see the use of atd (Attention Dial Command), this atd corresponds to the attention command given to the Device; as an attention command it instructs to attend or dial the call specified as 8581231234.

**THE BASIC LOGCAT**

The Android tool can view and filter the Android logging message

The format of log messages is as follows:

- `tv_sec tv_nsec pid tid priority tag  Message` 02-24 17:59:09.968  164  164 I usbd : main(): Phone was started up in normal mode
The above snippet provides us info about the phone’s activity. To be precise at what period of time it was powered up in the idle home screen mode. Again this message varies from OEM’s.

**DEBUGGING - USERSPACE**

The Android SDK provides most of the tools that we need to debug applications. We need a JDWP-compliant debugger if we want to be able to do things such as step through code, view variable values, and pause execution of an application [10]. If we are using Eclipse, a JDWP-compliant debugger is already included and there is no setup required. If we are using another IDE, we can use the debugger that comes with it and attach the debugger to a special port so it can communicate with the application VMs on devices. The main components that comprise a typical Android debugging environment are:

**Adb**

Adb acts as a middleman between a device and development system. It provides various device management capabilities, including moving and syncing files to the emulator, running a UNIX shell on the device or emulator, and providing a general means to communicate with connected emulators and devices.

**Dalvik Debug Monitor Server**

DDMS is a graphical program that communicates with devices through adb. DDMS can capture screenshots, gather thread and stack information, spoof incoming calls and SMS messages, and has many other features [11].

**Device or Android Virtual Device**

Application must run in a device or in an AVD so that it can be debugged. An adb device daemon runs on the device or emulator and provides a means for the adb host daemon to communicate with the device or emulator.

**Debugging Tips**

While debugging, keep these helpful tips in mind to help we figure out common problems with applications:
Dump the Stack Trace

To obtain a stack dump from emulator, we can log in with adb shell, use ps to find the process we want, and then kill -3. The stack trace appears in the log file.

Display Useful Info on the Emulator Screen

The device can display useful information such as CPU usage or highlights around redrawn areas. Turn these features on and off in the developer settings window as described in Debugging with the Development Tools App.

Get Application and System State Information from the Emulator

We can access dumpstate information from the adb shell commands. See dumpsys and dumpstate on the adb topic page.

Get Wireless Connectivity Information

We can get information about wireless connectivity using DDMS. From the Device menu, select Dump radio state.

Log Trace Data

We can log method calls and other tracing data in an activity by calling startMethodTracing(). See Profiling with Traceview and dmtracedump for details.

Log Radio Data

By default, radio information is not logged to the system (it is a lot of data). However, we can enable radio logging using the following commands [12]:

- adb shell
- logcat -b radio

Capture Screenshots

The Dalvik Debug Monitor Server (DDMS) can capture screenshots from the emulator. Select Device > Screen capture.
CHAPTER 3

DEBUGGING MECHANISM

CLASSIFICATION OF LOGS

In the above chapter we looked at the basic terminology of the log collection, but the question arises what kind of logs to look into. How will the analyzer approach an issue or a bug to track down to know its root cause. The answer to this is obviously like a never ending ocean, because it can depend on such a variety of features and that too indirectly indicates which layers we want to explore in the logs. Let us see at least a basic type of logging approach with some few examples.

TYPES OF LOGS

1. Logcat logs
2. Radio logs
3. Kernel logs
4. Event logs
5. Bug report
6. Tombstones
7. Traces.txt for ANRs (Application Not Responding)
8. Serial Logs (Optional)
9. Ram dump (cases when ‘adb shell’ doesn’t work)
10. Main logs
11. System Log

A point to be considered very carefully here is that each log can display different info in terms of the architectural differences because of variety of different OEM vendors incorporating different chipsets.

Logcat Logs: From Various Apps and System Level

The Android logging system provides a mechanism for collecting and viewing system debug output. Logs from various applications and portions of the system are collected in a
series of circular buffers, which then can be viewed and filtered by the logcat command. We can use logcat from an ADB shell to view the log messages [13].

- Application Not responding (/data/anr/traces.txt)
- Java crash (exceptions)
- Native Crash (/data/tombstones)
- Android frameworks reboot (eof)

**Main Logs: The Main Application Log**

Main — View the main log buffer (default). The main log, as the name suggests, is the first main basic log, which is extracted to view the event activity happening inside the phone. To explain this consider a small example: a user taps on the dialer app and exits back. During that event any basic info can be recorded through a main log, below is a small snippet explaining it.

Below is the short snippet of the main log explaining the phone status when it was kept in an ideal condition:

- 03-16 08:19:22.023 D/ComprehensiveCountryDetector(24656): onServiceStateChanged
- 03-16 08:19:22.023 D/StatusBar.NetworkController(24745): onServiceStateChanged state=0 home AT&T 310410 HSPA:11 CSS not supported -1 -1 mDataState=0 RoamInd=-1 DefRoamInd=-1 EmergOnly=false
- 03-16 08:19:22.023 D/StatusBar.NetworkController(24745): Combining data service state0for signal
- 03-16 08:19:22.033 D/StatusBar.NetworkController(24745): refreshViews connected={ data } level=4

We see above the format it displays the timestamp in respect to date and time [14]. Also we see the intent activities running, the state changes of which we get information regarding the Bluetooth tether icon and all other network stats.

This kind of powerful recording can help us to know at what time and at what instance some unusual activity happened, but this has some limitations also, as data or ore likely to say info regarding panics due to kernel or due to modem or say because of a GPS in standalone or in assisted mode cannot be captured. For that lets move on further to view other supported logging types.
To capture a basic Main Log we need to just type in:

- `adb logcat -v time > logname.txt`

The `-v` option as stated above is given for the log level output to be displayed verbosically, followed by redirecting to a file name where all the events are captured.

### Radio Logs

Radio — View the buffer that contains radio/telephony related messages. The radio log is one step advance compared to main log, this logs provides some more info related to the baseband side of the phone or in more simpler words to say events and activities related with the messaging, telephony, and other modem related areas can be captured.

Here's an example of how to view a log buffer containing radio and telephony messages:

- `adb logcat -b radio`

### Android Telephony – RIL messages: System Log - a log for low-level system messages and debugging: Many classes in the Android framework utilize the system log to keep their messages separate from (possibly noisy) application log messages. These programs use the `android.util.Slog` class, with its associated messages [15].

In all cases, eventually a formatted message is delivered through the C/C++ library down to the kernel driver, which stores the message in the appropriate buffer.

### Kernel Logs

- These helps in debugging target crash.
- Kernel logs are usually associated with a log level, 0, 1, 2, 3 etc.
- Any log with log level of 0,1,2 needs to be looked at.
- RAM Dumps —collection of faulty process and thread
  - Collection of traces of faulty threads and processes

### Event Logs

Events Log - for system event information: Event logs messages are created using `android.util.EventLog` class, which create binary-formatted log messages. Log entries consist of binary tag codes, followed by binary parameters. The message tag codes are stored on the
system at: /system/etc/event-log-tags. Each message has the string for the log message, as well as codes indicating the values associated with (stored with) that entry.

Event logs – which activity is launched/finished etc.

The Android framework reboots can be caused by Watchdog kills. A watchdog timer is a computer hardware or software timer that triggers a system reset or other corrective action (like process killer) if the main program, due to some fault condition, such as a hang, neglects to regularly service the watchdog. The intention is to bring the system back from the unresponsive state into normal operation [16].

- 12-02 11:14:12.210 I/watchdog_proc_pss(1252): [system,1252,45383680]
- 12-02 11:14:12.210 I/watchdog_hard_reset(1252): [system,1252,20971520,45383680]
- 12-02 11:14:12.210 I/watchdog_proc_pss(1252): [com.android.phone,1338,22220800]
- 12-02 11:14:12.210 I/watchdog_hard_reset(1252): null

**Bug Report: Combination of Several Logs**

Bug report: Prints dumpsys, dumpstate, and logcat data to the screen, for the purposes of bug reporting. Or we can say it a tool to collect logcat, event, radio and kernel logs in a single file.

To collect the bug report we need to type in

- Adb logcat > bugreport.txt

In other words it is a Tool to collect above logs in a single file

**Tombstones**

- Helps in debugging native user space crash
- User space segmentation fault will have SIGSEGV in the tombstone and the stack trace of the faulting thread
- The stack trace is a set of addresses which is parsed by the stack tool

**Traces.txt**

- Caused when an application doesn’t respond to key press within 5 seconds (ANR)
- The stack trace of the application is save in/data/anr/traces.txt
• Traces.txt holds java level stack traces

**Serial Logs**

• If we have a serial cable, please collect at least kernel logs over serial console
• We will need Putty/SecureCRT
• By default, it will give us dmesg logs
• We can press enter and issue any adb command to get other logs over the command prompt
• Very useful to debug issues where adb is not up

**COMMONLY SEEN ANDROID ERRORS**

Framework Reboot causes:

• Watchdog killing system process
• Fatal exception in one of system server’s threads
• Excessive JNI references
• Watchdog killing system process
• Stack trace from traces.txt
• "android.server.ServerThread" prio=5 tid=8 NATIVE
group="main" sCount=1 dsCount=0 s=N obj=0x2fb1df28 self=0x1fece8

Fatal Exception in system process

05-06 23:16:16.799 W/dalvikvm(  201): threadid=21: thread exiting with uncaught exception (group=0x40018560)
05-06 23:16:16.809 E/AndroidRuntime(  201): *** FATAL EXCEPTION IN SYSTEM PROCESS: WindowManagerPolicy
android.app.LoadedApk.forgetReceiverDispatcher(LoadedApk.java:610)
We can get the idea that system server is throwing fatal exception. We can conclude that user-space was corrupted by faulty thread while running specific application.

**DIVING IN ONE MORE LEVEL OF LOGGING**

The list below of issues describes the various types of panic and the reasons that are the most common in all OEM devices.
1. **Kernel panic**: Panics, which cause device instabilities, are certainly of concern. To more go deeper into this, it could be a soft kernel panic or a hard one. A soft kernel panic gets the device to reboot and go in a stable state where as a hard kernel panic causes a continuous reboot.

2. **Modem reset**: As the name suggests this causes the baseband of the device to reset causing a loss in network, this could be not alone due to some other panic but also due to instability issues of the kernel [17].

3. **Application not responding**: This is the most commonly seen problem in Android OEM’s. We should not consider this as an issue or a panic, rather this is somewhat of a wait period for the application to respond back, as it needs some time to recover from the background activity it is doing.

4. **Excessive Java native interface**: The Java Native Interface (JNI) is a programming framework that enables Java code running in a Java Virtual Machine (JVM) to call and to be called by native applications (programs specific to a hardware and operating system platform) and libraries written in other languages such as C, C++ and assembly.

5. **Application framework reboots**: Due to bad framework settings in either the OS or the OEM’s code can cause this type of panic to be seen, usually the end user experiences a random reboots coming across while on a call or sending an email or any such instances which requires handoffs with the framework.

6. **Memory leaks**: Memory leaks these are again some of the major issues seen on the performance side of the phone causing a sluggish behavior. This usually happens under certain identifiable circumstances as outlined below.

   Common culprits of memory leaks:
   - References to activities, context, view, drawable
   - Non-static inner classes (such as Runnable)
   - Caches to keep memory alive

7. **Memory corruption**: Memory corruption is usually caused due to sudden unintentional changes or one can say due to modifications due to some programatically errors. When these corrupted data is used in the computer program it crashes or does strange behaviors. In windows mobile this can be seen as heap corruption.

8. **Bus lock up issue**: There are a lot of dependencies between the channel and handshake signals that can be helpful to track the issue and prevent the deadlock situations and here bus plays a very vital role but there should be a good synchronization between the number of cycles, watchdog timer and signals.

   The bus types are usually platform-dependent but common buses are control bus, data bus, address bus and system bus etc.

9. **UI freeze**: Severe lock ups seen while usage of the phone, while swiping the phone during charging the battery or various sudden instances causes lockups to arrive.
10. **Watch Dog Byte**: The watchdog byte typically is set by a timer that triggers a system reset or any corrective action. This is due to some hang or any faulty condition. The basic intention is to bring back the device or the system from any unresponsive state to the regular operational mode.

11. **GPU hang**: Some complex calculation is going by the GPU while a high definition game or a video is being executed causing the Graphics processor to hang [18]. For example Figure 4.

   ![Image](image.jpg)

   **Figure 4. Figure showing a graphics distortion due to GPU hang.**

12. **Page Faults**: A page fault is a trap to the software raised by the hardware when a program accesses a page that is mapped in the virtual address space, but not loaded in physical memory. This is similar to page faults in a normal PC kind of memory environment [6].
13. **Unlocked register access**: In application processor’s registers a large amount of data is load from a larger memory to the registers, where it is used for all kind of arithmetic, manipulations or in a broader term we say some machine instructions are exchanged, there by causing sluggishness in performance there by losing the real meaning of caches by making it un meaningful in retrieving data in hot state (data which is accessed previously).

14. **Force close**: Figure 5 shows a screenshot of a force close seen in a “hello world” application. Debugging this issue can be difficult because of many symptoms as stated above that can cause such a force close.

![Figure 5](image)

**Figure 5.** Figure shows a force close seen while accessing the Hello Android app.

15. **ANR (Application Not Responding)**: One of the most widely seen panics in many phones (Figure 6).

![Figure 6](image)

**Figure 6.** ANR.
CHAPTER 4

THE ANDROID FUELING

DALVIK VIRTUAL MACHINE

A Dalvik Virtual Machine (DVM) is a software environment acting as an emulator, an operating system or a complete hardware virtualization, that has an implementation of resources without the actual hardware being present [19]. As an emulator it allows applications and operating systems to run on hardware that has a different processor architecture than the present one, while as an operating system VM virtualizes the server on operating system, and in-case of hardware virtualization two or more different operating systems can run simultaneously on the same hardware. Thus the main idea of virtual machine is to provide an environment that can execute instructions other than those associated with the host environment regardless of the hardware and software.

WHY HAS GOOGLE CHOOSEN A REGISTER-BASED VM?

Apart from the licensing issues with Sun [20], there are a number of technical advantages for selecting a registered-based VM:

1. A register VM is likely to have an intrinsic performance advantage over a stack VM when hosted on a pipelined processor [21].
2. Byte code verification is likely to be faster on a register VM (i.e., faster startup times) because stack height integrity checks will be greatly simplified.
3. A register VM will be more forgiving of incorrect code (in the VM, generated by the compiler, code corrupted during program transmission or storage attacked by malware) than a stack VM [22].

WHY ONLY DALVIK VM?

Mobile systems often offer limited RAM, a low performance CPU, slow internal flash memory, and limited battery power (though some of this is changing). Therefore, a need was felt for a VM that could provide better performance with limited resources. So came Dalvik, designed to run on Linux kernel, which provides process threading, pre-processing for faster application execution, User ID based security procedures and inter-process communication [11]. Dalvik works on low resourced ARM devices (Advanced RISC Machines), is 32- bit
processor architecture based on Reduced instruction set computer developed by ARM limited). ARM processors are used because of their simple architecture making it suitable for low power devices such as cell phones. Dalvik can also be ported to run on x86 systems.

**RUNTIME FRAMEWORK AND SUPPORTED LIBRARIES**

Runtime framework is a layer where a programmer starts building their applications. It consists of a virtual machine that execute Dalvik byte code (that is built out of the Java byte code and a core API). The cell phones of today don't have sufficient resources as that of a PC, so traditional Java (as requires more resources) cannot be used for mobile systems, in general. It is therefore required that a set of core functionality libraries optimized for these limited capabilities devices be deployed.

In case of Dalvik, the supported libraries includes:

1. dalvik/libcore (written in C/C++)
2. dalvik/vm/native (written in C/C++)
3. OpenSSL (for encryption)
4. zlib (free, general-purpose, data-compression library)
5. ICU (for character encoding)
6. Java packages (including java.nio, java.lang, java.util)
7. Apache Harmony classlib (including Apache HttpClient)

Dalvik VM itself is written in portable C. However it has one non-portable component of the runtime called JNI Call Bridge.

**DALVIK INTERNALS**

System memory & overhead: We see our phones sometimes unresponsive and may have wondered why this occurs. One of the reasons is **overhead to the system**. This could be more well understood by the minimum size of the memory for the virtual machine should, which should be set based on the recommendations of the operating system. So sharing of byte-code (also known as class-data) is important to avoid unnecessary usage of system memory [23].

System overhead refers to the processing time required by system software, which includes the operating system and any utility that supports application programs.
Redundancy and space: In our previous chapter we talked about memory leaks, fragmentation issues, page faults, redone, and OOM (out of memory) issues, below is one of the answer to all these issues.

Redundancy is important as it provides fault tolerance, but it can consume a lot of memory as it requires duplication of data set or error-correcting code that has to be stored in memory. Normally, class data is stored in individual files, causing too much memory usage and may result in lots of redundancy, example, in storing strings.

Dalvik overcomes the issue of memory space (consumed by redundancy, and separate files for each class) by aggregating the multiple classes into single .dex file. This saves a lot of memory for the system.

System memory: To minimize the system memory usage, dex files are mapped read-only (for security purpose, that is discussed later in this report) and also sharing is allowed between processes [8]. This avoids unnecessary repetition of data, and reduces memory usage.

**COMMON DALVIK ISSUES**

Logging examples: As above discussed about memory, low space issues, lets see some examples which we collected real time.

Logs:

1. **Dalvik running out of memory:**

   The below log snippet tells us about If we run out of memory, what could be the reason one of it is try first to remove everything in /data/dalvik-cache then reboot immediately. This is because the necessary .dex files will be recreated automatically.

   6r6r6ZZ1dalvikvm-heapOut of memory on a 20-byte allocation.(3r6r6ZZ1dalvikvm"main" prio=5 tid=1 RUNNABLE J3r6r6ZZ1dalvikvm | group="main" sCount=0 dsCount=0
   >Yr6r6ZZ1dalvikvm at dalvik.system.NativeStart.main(Native Method)

2. **Dalvik-heap out of memory we are seeing Dalvik-heap Out of memory errors during allocations from Sensors:**

   There is plenty of heaps available but that doesn't seem to matter. The GC does not seem to be able to clean up this "external memory"

   01-02 13:10:09.939 376 492 E dalvikvm-heap: Out of memory on a 52-byte allocation.
REAL TIME ISSUE AND ITS ANALYSIS

Template for Post Mortem Analysis

What was the test scenario? This test scenario involves threads, similar to what we have seen in other OS, the test comprises of executing two threads say A and B. According to the log from the successful test scenario, the scheduler runs the threads "A" and "B" in the same order as the "Thread.Start" function call. However in the failed scenario, the scheduler is running them in a different order.

Scheduler in dual core architecture is not guaranteed to run the threads in the same order. Hence run-test script can be changed to sort the contents of the expected output and the generated output.

050-sync-test: In these tests, multiple threads are started and the expected output suggests that the threads start in the same order in which they are called. But this is not possible in the dual-core architecture. Hence the run-test is modified accordingly.

However prints within synchronization should be correctly printed in order as only one thread can access the protected code at a particular time. This is done correctly.

Case 1-sync-test: In these tests, multiple threads are started and the expected output suggests that the threads start in the same order in which they are called. But this is not possible in the dual-core architecture. Hence the run-test is modified accordingly [24].

However prints within synchronization should be correctly printed in order as only one thread can access the protected code at a particular time. This is done correctly.

Case 2-dexfile: The main objective of this test is to test the functionality of the class file conversion from a dex file. In a normal Dalvik scenario, a JAVA application consisting of different classes are converted into different class files (.class), which are then optimized
to form a single DEX file. The DEX file generated here is done in /sdcard directory but since Sdcard directory can be used only when there is an SD card, we moved it to /data directory.

Case 3 finalizer: "system.gc" calls does not guarantee that garbage collection will be done immediately. (http://www.sap-img.com/java/garbage-collection-cannot-be-forced.htm ). So object finalizer can be called either by the gc or by the runFinalizer routine. And this timing Variance is OK. Hence a small delay can be added just before the System.Gc call so that System.Gc does the garbage collection. It might be a really good idea to call System.gc() at those points where the system is less busy and kind of suggests to the garbage collector to execute.
CHAPTER 5

REAL TIME ISSUES POST-MORTEM

To start with this chapter focuses on Run Time failures and Issues seen let’s start with Excessive JNI issue,

**EXCESSIVE JNI ISSUE**

Excessive JNI global reference causing system_server to crash.

The Java Native Interface, JNI, is just that; an interface. It's an API that allows Java code to interact with code written in another language, typically C or C++. Because the JNI is an interface, JVM vendors are free to implement the virtual machine as they see fit. As long as the JVM follows the specification of the JNI, all native code written to the specification should work with that JVM [25].

The Java Native Interface (JNI) is a programming framework that enables Java code running in a Java Virtual Machine (JVM) to call and to be called by native applications (programs specific to a hardware and operating system platform) and libraries written in other languages such as C, C++ and assembly.

How Power Management works and JNI scenario (Figure 7): Android supports its own power management designed with the premise that the CPU shouldn't consume power if no applications or services require power. Android requires that applications and services request CPU resources with "wake locks" through the Android application framework and native Linux libraries. If there are no active wake locks, Android will shut down the CPU.

**PAGE FAULTS ERROR**

A page fault is a trap to the software raised by the hardware when a program accesses a page that is mapped in the virtual address space, but not loaded in physical memory. In the typical case the operating system tries to handle the page fault by making the required page accessible at a location in physical memory or kills the program in the case of an illegal access. The hardware that detects a page fault is the memory management unit in a processor.
Figure 7. Figure explains the android power cycle anatomy.
The exception handling software that handles the page fault is generally part of the operating system.

Contrary to what the name 'page fault' might suggest, page faults are not errors and are common and necessary to increase the amount of memory available to programs in any operating system that utilizes virtual memory, including Microsoft Windows, Unix-like systems.

Consider one of the most common problems when dealing with virtual memory – the page fault. A page fault occurs when a program requests an address on a page that is not in the current set of memory resident pages. What happens when a page fault occurs is that the thread that experienced the page fault is put into a Wait state while the operating system finds the specific page on disk and restores it to physical memory.

When a thread attempts to reference a nonresident memory page, a hardware interrupt occurs that halts the executing program. The instruction that referenced the page fails and generates an addressing exception that generates an interrupt. There is an Interrupt Service Routine that gains control at this point and determines that the address is valid, but that the page is not resident. The OS then locates a copy of the desired page on the page file, and copies the page from disk into a free page in RAM. Once the copy has completed successfully, the OS allows the program thread to continue on. One quick note here – if the program accesses an invalid memory location due to a logic error an addressing exception similar to a page fault occurs. The same hardware interrupt is raised [26]. It is up to the memory manager’s Interrupt Service Routine that gets control to distinguish between the two situations.

It is also important to distinguish between hard page faults and soft page faults. Hard page faults occur when the page is not located in physical memory or a memory-mapped file created by the process (the situation we discussed above). The performance of applications will suffer when there is insufficient RAM and excessive hard page faults occur. It is imperative that hard page faults are resolved in a timely fashion so that the process of resolving the fault does not unnecessarily delay the program’s execution. On the other hand, a soft page fault occurs when the page is resident elsewhere in memory. For example, the page may be in the working set of another process. Soft page faults may also occur when the
page is in a transitional state because it has been removed from the working sets of the processes that were using it, or it is resident as the result of a prefetch operation.

To summarize then, a page fault is nothing more than the computer hardware reporting to the operating system (OS) that it either is not allowed to access the virtual page as requested by the running code (because of the access) or it cannot translate the virtual page to a physical page. In either case, it is the responsibility of the OS to "do the right thing" and allow the system to continue running.

When a page fault occurs, the hardware cannot do anything else with the instruction that caused the page fault and thus it must transfer control to an operating system routine (this is the page fault handler). The page fault handler must then decide how to handle the page fault. It can do one of two things:

- It can decide the virtual address is just simply not valid. In this case, Windows will report this error back by indicating an exception has occurred (typically STATUS_ACCESS_VIOLATION)
- It can decide the virtual address is valid. In this case, the OS will find an available physical page, place the correct data in that page, update the virtual-to-physical page translation mechanism and then tell the hardware to retry the operation. When the hardware retries the operation it will find the page translation and continue operations as if nothing had actually happened.

An interrupt can occur when a program requests data that is not currently in real memory. The interrupt triggers the operating system to fetch the data from a virtual memory and load it into RAM. An invalid page fault or page fault error occurs when the operating system cannot find the data in virtual memory. This usually happens when the virtual memory area, or the table that maps virtual addresses to real addresses, becomes corrupt.

**THREAD SCHEDULING**

We introduced the thread scheduler, part of the OS (usually) that is responsible for sharing the available CPUs out between the various threads. On multiprocessor systems, there is generally some kind of scheduler per processor, which then need to be coordinated in some way. A thread of higher priority (which is a function of base and local priorities) will preempt a thread of lower priority.
JAVA Thread Scheduling Features

1. The JVM schedules using a preemptive, priority-based scheduling algorithm.
2. All Java threads have a priority and the thread with the highest priority is scheduled to run by the JVM.
3. In case two threads have the same priority a FIFO ordering is followed.

Scheduling Threads

Threads can be scheduled, and the threads library provides several facilities to handle and control the scheduling of threads. It also provides facilities to control the scheduling of threads during synchronization operations such as locking a mutex [27]. Each thread has its own set of scheduling parameters. These parameters can be set using the thread attributes object before the thread is created. The parameters can also be dynamically set during the thread's execution.

Controlling the scheduling of a thread can be a complicated task. Because the scheduler handles all threads system wide, the scheduling parameters of a thread interact with those of all other threads in the process and in the other processes. The following facilities are the first to be used if we want to control the scheduling of a thread.

The threads library allows the programmer to control the execution scheduling of the threads in the following ways:

- By setting scheduling attributes when creating a thread
- By dynamically changing the scheduling attributes of a created thread
- By defining the effect of a mutex on the thread's scheduling when creating a mutex (known as synchronization scheduling)
- By dynamically changing the scheduling of a thread during synchronization operations (known as synchronization scheduling)

RED ZONE

Red zone is a term designating the fixed size area in memory beyond the stack pointer that has not been "allocated". This region of memory is not to be modified by interrupt/exception/signal handlers. This allows the space to be used for temporary data without the extra overhead of modifying the stack pointer.
**MEMORY LEAK ANDROID**

Android is the most happening platform that is available now and is used extensively by the offshore android development. There are a number of Android mobile applications that are built on this platform. But despite its immense popularity Android development is not devoid of issues. One of the main issues that are faced by the offshore Android development teams is memory-leaks. This is because the maximum heap for mobile applications is 16GB. If we come to think of it for a phone the memory space is really high. But then for offshore Android development team who wants to achieve more, this much space is not sufficient [21]. The main advantage of the Android is multi-tasking and hence it can take a lot of applications in the memory space. The memory leaks happen mainly because the term context is used for a lot of operations within Android development. But then the main use of this is to access or load resources for the Android applications [5]. The widgets are constructed with the help of this context parameter. Application and activity are the two types of contexts that are available. Normally the activity is passed.

**GARBAGE COLLECTION**

An Android application will run on a mobile device with limited computing power and storage, and constrained battery life. Because of this, it should be efficient. Battery life is one reason we might want to optimize application even if it already seems to run "fast enough". Battery life is important to users, and Android's battery usage breakdown means users will know (through the use of available service apps) if an application is responsible for draining their battery.

Note that although this document primarily covers micro-optimizations, these will almost never make or break software. Choosing the right algorithms and data structures should always be priority but is outside the scope of this document.

**Optimize Judiciously:**

- Android-specific micro-optimization, so it assumes that we've already used profiling to work out exactly what code needs to be optimized, and that we already have a way to measure the effect (good or bad) of any changes we make. We only have so much engineering time to invest, so it's important to know we're spending it wisely.

- We made the best decisions about data structures and algorithms, and that we've also considered the future performance consequences of API decisions. Using the
right data structures and algorithms will make more difference than any of the advice here, and considering the performance consequences of API decisions will make it easier to switch to better implementations later (this is more important for library code than for application code).

- One of the trickiest problems we'll face when micro-optimizing an Android app is that the application is pretty much guaranteed to be running on multiple hardware platforms. Different versions of the VM run on different processors running at different speeds. It's not even generally the case that we can simply say "device X is a factor F faster/slower than device Y", and scale results from one device to others. In particular, measurement on the emulator tells us very little about performance on any device. There are also huge differences between devices with and without a JIT: the "best" code for a device with a JIT is not always the best code for a device without.

**AVOID CREATING UNNECESSARY OBJECTS**

Object creation is never free. A generational GC with per-thread allocation pools for temporary objects can make allocation cheaper, but allocating memory is always more expensive than not allocating memory. If we allocate objects in a user interface loop, we will force a periodic garbage collection, creating little "hiccups" in the user experience. The concurrent collector introduced in Gingerbread helps, but unnecessary work should always be avoided. Thus, we should avoid creating object instances we don't need to. Some examples of things that can help:

- If we have a method returning a string, and we know that its result will always be appended to a String Buffer anyway, change the signature and implementation so that the function does the append directly, instead of creating a short-lived temporary object.

- When extracting strings from a set of input data, try to return a substring of the original data, instead of creating a copy. We will create a new String object, but it will share the char with the data [28]. (The trade-off being that if we're only using a small part of the original input, we'll be keeping it all around in memory anyway if we go this route.)

A somewhat more radical idea is to slice up multidimensional arrays into parallel single one-dimension arrays:

- An array of ints is a much better than an array of Integers, but this also generalizes to the fact that two parallel arrays of ints are also a lot more efficient than an array of (int,int) objects. The same goes for any combination of primitive types. Generally speaking, avoid creating short-term temporary objects if we can. Fewer objects created mean less-frequent garbage collection, which has a direct impact on user experience.
Generally speaking, in the presence of a garbage collector, it is never good practice to manually call the GC. A GC is organized around heuristic algorithms, which work best when left to their own devices. Calling the GC manually often decreases performance.

Occasionally, in some relatively rare situations, one may find that a particular GC gets it wrong, and a manual call to the GC may then improve things, performance-wise. This is because it is not really possible to implement a "perfect" GC, which will manage memory optimally in all cases. Such situations are hard to predict and depend on many subtle implementation details. The "good practice" is to let the GC run by itself; a manual call to the GC is the exception, which should be envisioned only after an actual performance issue has been duly witnessed.
CHAPTER 6

CONCLUSION AND FUTURE WORK

We saw above issues coming in from different layers, whether is the core kernel or the most everyday used the application, but this does not limits to this, as there still many more issues which are beyond the scope to express until and unless one gets an hands on opportunity to deal with it. We have tried to provide a one step ahead to educate the developer and to some degree possibly certain end users just how deep we can dig in finding the root cause by logging, faulty process, corrupted threads, stack call, traces and also how to find an inference for it.

We can also automate these procedure in some form of a GUI tool, so that it’s reduces effort to capture each log in through the command line, also one can think of parsing those logs based on a particular pattern and exporting it to a CSV file to make it easier.

On concluding this is a broad overview platforms like windows mobile and Linux based also approach the same way but the basic concept is the same.
REFERENCES


