Mobile Devices - An Introduction to the Android Operating Environment

Design, Architecture, and Performance Implications

1.0 Introduction

With the worldwide proliferation of mobile devices, reliability, availability, connectivity, as well as performance related concerns, similar to the once encountered on traditional IT server systems, became paramount. On the smartphone and internet tablet side, one of the fastest growing solutions are Android based products (source digitimes 2010). While Android based systems get a lot of exposure in the media, there is a lot of myth surrounding the actual implementation details. Some people label Android as a Linux solution, which really does not reflect the facts. Ergo, this report discusses the major components that comprise the Android operating environment, elaborating on the Android design and architecture (the building blocks), as well as addressing the Android verses Linux question.

Keywords: Android, Linux, Architecture, Kernel, Design, Performance

2.0 Background & History

Android is described as a mobile operating system, initially developed by Android Inc. Android was sold to Google in 2005. Android is based on a modified Linux 2.6 kernel. Google, as well as other members of the Open Handset Alliance (OHA) collaborated on Android (design, development, distribution). Currently, the Android Open Source Project (AOSP) is governing the Android maintenance and development cycle [8].

To reiterate, the Android operating system is based on a modified Linux 2.6 kernel [6]. Compared to a Linux 2.6 environment though, several drivers and libraries have been either modified or newly developed to allow Android to run as efficiently and as effectively as possible on mobile devices (such as smart phones or internet tablets). Some of these libraries have their roots in open source projects. Due to some licensing issues, the Android community decided to implement their own c library (Bionic), and to develop an Android specific Java runtime engine (Dalvik Virtual Machine – DVM). With Android, the focus has always been on optimizing the infrastructure based on the limited resources available on mobile devices [2]. To complement the operating environment, an Android specific application framework was designed and implemented. Therefore, Android can best be described as a complete solution stack, incorporating the OS, middle-wear components, and applications. In Android, the modified Linux 2.6 kernel acts as the hardware abstraction layer (HAL). To summarize, the Android operating environment can be labeled as:

- An open platform for mobile development
- A hardware reference design for mobile devices
- A system powered by a modified Linux 2.6 kernel
- A run time environment
- An application and user interface (UI) framework
3.0 Android Architecture

Figure 1 outlines the current (layered) Android Architecture. The modified Linux kernel operates as the HAL, and provides device driver, memory management, process management, as well as networking functionalities, respectively. The library layer is interfaced through Java (which deviates from the traditional Linux design). It is in this layer that the Android specific libc (Bionic) is located. The surface manager handles the user interface (UI) windows. The Android runtime layer holds the Dalvik Virtual Machine (DVM) and the core libraries (such as Java or IO). Most of the functionalities available in Android are provided via the core libraries.

The application framework houses the API interface. In this layer, the activity manager governs the application life cycle. The content providers enable applications to either access data from other applications or to share their own data. The resource manager provides access to non-code resources (such as graphics), while the notification manager enables applications to display custom alerts. On top of the application framework are the built-in, as well as the user applications, respectively. It has to be pointed out that a user application can replace a built-in application, and that each Android application runs in its own process space, within its own DVM instance. Most of these major Android components are further discussed (in more detail) in the next few sections of this report.

Figure 1: Android Architecture

Note: Figure 1 courtesy of the OHA

3.1 Dalvik Virtual Machine

Android based systems utilize their own virtual machine (VM), which is known as the Dalvik Virtual Machine (DVM) [4]. The DVM uses special byte-code, hence native Java byte-code cannot directly be executed on Android systems. The Android community provides a tool (dx) that allows converting Java
class files into Dalvik executables (dex). The DVM implementation is highly optimized in order to perform as efficiently and as effectively as possible on mobile devices that are normally equipped with a rather modest (these days normally a dual, or quad) CPU subsystem, limited memory resources, no OS swap space, and limited battery capacity. The DVM has been implemented in a way that allows a device to execute multiple VM’s in a rather efficient manner. It also has to be pointed out that the DVM relies on the modified Linux kernel for any potential threading and low-level memory management functionalities. With Android 2.2, some major changes to the JVM infrastructure were implemented. Up to version 2.2, the JVM was an actual interpreter, similar to the original JVM solution deployed with Java 1.0. While the Android solution always reflected a very efficient interpreter, it was still an interpreter and hence, no native code was generated. With the release of Android 2.2, a just-in-time (JIT) compiler was incorporated into the solution stack, which translates the Dalvik byte-code into much more efficient machine code (similar to a C compiler). Currently, Android version 4 (Ice Cream Sandwich) and 4.1/4.2 (Jelly Bean) is deployed on some devices. It has to be pointed out though that currently, only a few devices are actually running version 4.1/4.2 or 4.0, while most devices are still operating on older Android versions. Down the road, additional JIT and garbage collection (GC) features will be deployed with Android, further busting aggregate systems performance.

3.2 Target Platform - Instruction Set

To simplify the discussion, the statement made here is that most of the Linux 2.6 based devices are x86 based systems, whereas most mobile phones are ARM based products. While ARM represents a 32-bit reduced instruction set computer (RISC) instruction set architecture, x86 systems are primarily based on the complicated instruction set computer (CISC) architecture. In general, the statement can be made that ARM (RISC) is executing simpler (but more) instructions compared to an x86 (CISC) system. As already discussed, memory is at a premium in mobile devices due to size, cost, and power constraints. ARM addresses these issues by providing a 2nd 16-bit instruction set (labeled thumb) that can be interleaved with regular 32-bit ARM instructions. This additional instruction set can reduce the code size by up to 30% (at the expense of some performance limitations). Ergo, from an overall systems perspective, the incorporation of the thumb instruction set can be considered as an exercise in compromises. Compared to x86 processors, the ARM design reveals a strong focus on lower power consumption, which again makes it suitable for mobile devices [1]. As with any other computer, the processor has a significant impact on aggregate systems performance. In the early days of Android, most devices featured the same ARM Qualcomm processor and hence, their performance behavior was pretty comparable. With the distribution of the Motorola Droid, the next generation of chipsets was introduced (supporting enhanced graphics processors). Today, there are basically 3 major chipsets being deployed in Android devices. To illustrate, HTC is utilizing the Qualcomm Snapdragon, Motorola uses the Texas Instruments OMAP, whereas Samsung designed their own Hummingbird chipset. It has to be pointed out though that all 3 processors discussed here are based on the ARM Cortex-A8 architecture (with vendor specific tweaks to offer unique features).

3.3 Kernel and Startup Process

It is paramount to reiterate that while Android is based on Linux 2.6, Android does not utilize a standard Linux kernel [6],[7]. Hence, an Android device should not be labeled a Linux solution per se. Some of the Android specific kernel enhancements include:

- alarm driver (provides timers to wakeup devices)
- shared memory driver (ashmem)
- binder (for inter-process communication)
- power management (which takes a more aggressive approach than the Linux PM solution)
- low memory killer
- kernel debugger and logger

During the Android boot process, the Android Linux kernel component first calls the init process (compared to standard Linux, nothing unusual there). The init process accesses the files init.rc and init.device.rc (init.device.rc is device specific). Out of the init.rc file, a process labeled zygote is started. The zygote process loads the core Java classes, and performs the initial processing steps. These Java classes can be reused by Android applications and hence, this step expedites the overall startup process. After the initial load process, zygote idles on a socket and waits for further requests.

Every Android application runs in its own process environment. A special driver labeled the binder allows for (efficient) inter-process communications (IPC). Actual objects are stored in shared memory. By utilizing shared memory, IPC is being optimized, as less data has to be transferred. Compared to most Linux or UNIX environments, Android does not provide any swap space. Hence, the amount of virtual memory is governed by the amount of physical memory available on the device [7].

3.4 The Bionic Library

Compared to Linux, Androids incorporates its own C library (Bionic) [3]. The Bionic library is not compatible with the Linux glibc. Compared to glibc, the Bionic library has a smaller memory footprint. To illustrate, the Bionic library contains a special thread implementation that 1st, optimizes the memory consumption of each thread and 2nd, reduces the startup time of a new thread. Android provides run-time access to kernel primitives [2]. Hence, user-space components can dynamically alter the kernel behavior. Only processes/threads though that do have the appropriate permissions are allowed to modify these settings. Security is maintained by assigning a unique user ID (UID) and group ID (GID) pair to each application. As mobile devices are normally intended to be used by a single user only (compared to most Linux systems), the UNIX/Linux /etc/passwd and /etc/group settings have been removed. In addition (to boost security), /etc/services was replaced by a list of services (maintained inside the executable itself). To summarize, the Android C library is especially suited to operate under the limited CPU and memory conditions common to the target Android platforms [2]. Further, special security provisions were designed and implemented to ensure the integrity of the system.

3.5 Storage Media & File System

When it comes to configuring and setting-up mobile devices, traditional hard drives are in general too big (size), too fragile, and consume too much power to be useful. In contrast, flash memory devices normally provide a (relative) fast read access behavior as well as better (kinetic) shock resistance compared to hard drives. Fundamentally, two different types of flash memory devices are common, labeled as NAND and NOR based solutions [5]. While in general, NOR based solutions provide low density, they are characterized as (relative) slow write and fast read components. On the other hand, NAND based solutions offer low cost, high density, and are labeled as (relative) fast write and slow read IO solutions. Some embedded systems are utilizing NAND flash devices for data storage, and NOR based components for the code (the execution environment). From a file system perspective, as of Android version 2.3, the (well-known) Linux ext4 file system is being used [9]. Prior to the ext4 file system, Android normally used YAFFS (yet another flash file system). The YAFFS solution is known as the first NAND-optimized Linux flash file system. Some Android product providers (such as Archos with ext3 in Android 2.2) replaced the standard Archos file system with another file system solution of their choice. As of the writing of this report, the maximum size of any Android application equals to a low 2-digit MB number, which compared to actual Linux based systems has to be considered as being very small. This
implies that the memory and file system requirements (from a size perspective – not from a data integrity perspective) are vastly different for Android based devices compared to most Linux systems.

3.6 Power Management

In the mobile device arena, power management is obviously paramount. That does not imply though that power management should be neglected on any other system. Hence, power management in any IT system, with any operating system, is considered a necessity due to the ever increasing power demand of today’s computer systems. To illustrate, to reduce and manage power consumption, Linux based systems provide power-saving features such as clock gating, voltage scaling, activating sleep modes, or disabling memory cache. Each of these features reduces the system’s power consumption (normally at the expense of an increased latency behavior) [9]. Most Linux based systems manage power consumption via the Advanced Configuration and Power Interface (ACPI). Android based systems provide their own power management infrastructure (labeled PowerManager) that was designed based on the premise that a processor should not consume any power if no applications or services actually require power. Android demands that applications and services request CPU resources via wake locks through the Android application framework and native Linux libraries. If there are no active wake locks, Android will shutdown the processor.

4.0 Android Applications

Android applications are bundled into an Android package (.apk) via the Android Asset Packaging Tool (AAPT). To streamline the development process, Google provides the Android Development Tools (ADT). The ADT streamlines the conversion from class to dex files, and creates the .apk during deployment. In a very simplified manner, Android applications are in general composed of:

- Activities (needed to create a screen for a user application – classes with a UI)
- Intents (used to transfer control from one activity to another)
- Services (classes without a UI, so they can be executed in the background)
- Content Providers (allows the application to share information with other applications)

5.0 Android and Linux – Comparison

Figure 2 discloses the major differences between the Android and the Linux 2.6 operating environment. First of all, the Android kernel was derived from Linux, but has been significantly altered outside the mainline Linux kernel distribution. To further illustrate that point, Android is neither equipped with a native X-Windows setup, nor does it support the full set of standard GNU libraries. Hence, it is (more than just) a daunting task to port any existing GNU/Linux application or library to Android (support for X-Windows would be possible in Android though).

The biggest difference between Linux and Android revolves around the Java abstraction layer embedded into Android. As depicted in Figure 2, the Android design is based on a deeper implementation stack than Linux. In other words, the Android applications are farther removed from the actual kernel than in Linux (have a longer code path down into the OS layer). The core of Linux applications are developed in c and c++, hence c and c++ code represents the predominant Linux application environment. In Linux, the user applications (via the libraries and the system call subsystem) have direct kernel access, not so with Android (see Figure 2) [7]. In Android, the kernel is almost hidden deep inside the Android operating environment. Under Linux, the make process for (c, c++) applications can directly be optimized via special compiler flags, further boosting application performance [7]. Further, the Linux operating setup
natively incorporates a very rich infrastructure of libraries, debuggers, and development tools that are not accessible by Android. While the Android design is based on a deeper implementation stack, and hence the applications are farther removed from the kernel compared to Linux, Android kernel performance is still important and has to be quantified and understood. As in Linux, aggregate application performance is still impacted by the efficiency of the implemented kernel primitives. Compared to Linux, only a few Android performance, stress-testing, and benchmarking tools (such as DHTDroid) are available today. Based on the rapid development and deployment cycle of Android based systems, the need for actual Android application and kernel-level performance tools will increase rather significantly over the near future.

Figure 2: The Android vs. the Linux 2.6 Environment

Note: Figure 2 courtesy of OHA. The Java Native Interface (JNI) represents the interface between the Java code (running in a JVM) and the native code running outside the JVM.

Summary

Elaborating on the major components that comprise the Android operating environment, this report focused on providing a comprehensive overview of the status quo. The very impressive, rapid evolution of Android resembles the great work done by the Linux community over the years. As discussed in this report, Android is not a Linux solution per se, but does utilize a modified Linux 2.6 kernel that is incorporated into the Android operating environment.

References

2. Liang, “System Integration for the Android Operating System”, National Taipei University, 2010