Android Process Management

Computers. Data Science and Engineering, UPC spring 2020.

Àlex Batlle Casellas

1 Introduction

Android is an operating system developed by Google, based on the Linux kernel and mainly intended for handheld, touch-sensitive devices. It has also developed in this last years towards other kind of devices: television, computers, IoT, ... It has also spawned different community-developed mobile-phone OS based on it, as it is open-source.
The main objective of this work is to look into Android process management, and we are going to adress several questions in order to structure the document:

1. How do applications run in Android? How are they represented logically (processes, execution threads, ...)?
2. What are the system calls available in Android?
3. Is Android capable of running parallel programs/applications?

2 Background

Android is a **multiprogrammed OS**, which means that multiple programs can run at once. In this sense, we are going to explore what drives this OS schedule regarding application execution policies. Android runs apps using what's called a **runtime environment**. A **Runtime Environment** is the execution environment provided to an app by the operating system. This means that the RE provides the apps with system calls, RAM access and virtual memory spaces. As of currently maintained versions of Android (8.0 Oreo, 8.1.0 Oreo, 9.0 Pie and Android 10), the system uses the **Android Runtime** (ART), an application runtime environment. ART uses under the cover **Dalvik executables** (`.dex` files), a type of interpretable bytecode files converted mainly from Java `.class` files. Android versions 4.4 KitKat and below used only Dalvik for application and process management. As this versions are far behind the current ones, we will not explore much about Dalvik.

3 Process Management in Android Devices

We should start by addressing the first question. So, we will first explain what the ART is.

Android Runtime (ART)

The Android Runtime is a managed runtime environment, quite similar to the Java Runtime Environment. ART translates an application's Dalvik bytecode (contained in a file named `classes.dex` inside an `.apk`) into **native machine code**, which the device can then rapidly run. This compilation is done ahead of time (AOT), upon installation of an app. Unlike ART, Dalvik used to only convert an app into bytecode and compile some frequently run sections into machine code just in time when the application was run (JIT). The former was interpreted, while the latter were executed directly on the machine, providing some time optimization. As of Android 7.0 Nougat, JIT compilation steps were re-introduced with a new interpreter (without completely dropping AOT compilation) to further increase optimizations upon runtime and space occupied by the apps. Both ART and Dalvik use Java Virtual Machines, a means to provide a **platform-agnostic** code development pipeline. When we say platform-agnostic, we mean that within devices with the same type of hardware components, an app will run "regardless" of the specifications of each component.

The *Zygote* process

As Android is based on the Linux kernel, its apps are represented as **Process Control Blocks** just as Linux processes are. Android uses a very common way of bringing applications into life, and that is by using a process named **Zygote**. This is the same procedure as in normal Linux systems (these use the `systemd` process, for example), but with some particularities. Zygote is a constantly running process started at boot time, and as its biology namesake, it is the "initial cell formed when a new organism is produced". It is initialized as an incomplete process: its memory space contains all the common classes and core libraries that are needed by any app, but it does not contain any code specific to an app. It runs a Java VM and listens to a socket for incoming commands.

Process creation from Zygote

Each time a "create another process" command is received through the socket, Zygote forks (which it can do fast and reliably, as this is a Linux kernel OS) and creates a child process that runs the app that sent the command. This newly created child contains its own Java VM (different from Zygote's), and when created loads from the application package file (`.apk`) the `classes.dex` file, and places inside the VM not a copy of the bytecode, but the ART machine code translation of it. This ART translation is an *ELF shared object*, called an **OAT library**. Now, this OAT library contains the activities, services and content provider components of the application in case. But, only Zygote contains all the libraries and core services of the device: each child is granted a map, not a copy, of this libraries. Hence, the core libraries are shared between processes, and processes are far more lightweight than they would be using static libraries (as it was done before). This speeds up a lot application start up and loading, as they are (only) responsible of loading their own private libraries. From all of this we can see that **apps will run in only one process** by default, and in only one execution flow/software thread. This can be tweaked on the app manifest but is not recommended in general.

Process states in Android

Process can be in five different states in Android. Whereas in Linux we had Ready, Running, Blocked (by the scheduler) and Zombie states, in Android the paradigm is different. The different states are:

- **Foreground process**: the process is running in front of the screen or interacting with the process in front of the screen, and then active system resources are needed to maintain them. Only a few processes are in foreground state at a time.
- **Visible process**: a visible process is a process that is not foreground but that is still affecting (not interacting) the process in the foreground. For example, a dialog over another application.
- **Service process**: this isn't tied to any app we can see on the screen, but is tied to a running app. Service processes can be further catalogued into:
  - Foreground service: displays a notification and is noticeable to the user. For example, playing an audio track, or downloading a file.
  - Background service: an operation not noticeable by the user. For example, compacting an app's storage.
  - Bound service: a service is called bound when a component or activity of an app binds to it by calling an API method named `bindService()`. A bound service has a client-server interface between it and the app(s) it is bound to, and only runs as long as another app is bound to it. Multiple apps can bind to the same service at once. For example, an app downloading source files or resource files from the internet.
- **Background process**: these are not visible to the user and do not affect the screen or the user context. At a time there may be several background processes running. For example, apps that

are "paused" belong to this category; they are cached (on cache memory or on RAM) and can be resumed at any time. They do not use CPU time.

- **Empty process**: this is a process that has ended. This would be analogous to the Zombie state in usual Linux machines, but in this case, the PCB no longer contains any of the app code or information. It is practically a Zygote clone, and may be kept around for faster process creation, or it may be killed.

When a process needs more memory or resources, empty processes go first in the death corridor. Then, background processes follow them in order of priority, and afterwards, services. A remarkable thing of Android is that **it uses RAM to cache apps**. There is no point in leaving RAM memory empty, because empty RAM doesn't give faster writing or allocation to Android. This OS uses RAM with respect to its apps' requirements at any point in time, emptying unused (full) chunks and storing background apps' information for resuming.

System Calls in Android

System calls in Android **are not different from the ones in Linux**, and in fact they are a subset of them. In particular, with a small Google-search, we can find the Android source code, and inside its directories, the system calls for Android 10. In particular, within its source code we can find a file named `SYSCALLS.TXT`, in `platform/bionic/android10-release/libc/`. This file provides us with the list of syscalls available from C, and they are the typical Linux syscalls. To name a few,

- `fork` in Linux → `fork` in Android
- `getpid`, `waitpid`, `getppid` in Linux → `getpid`, `waitpid`, `getppid` in Android
- `read`, `write` in Linux → `read`, `write` in Android
- `open`, `close`, `pipe`, `mknod` in Linux → `open`, `close`, `pipe`, `mknod` in Android
- `getrusage`, `clock_gettime`, `gettimeofday` in Linux → `getrusage`, `clock_gettime`, `gettimeofday` in Android

The fact that the syscalls are the same in Android and in Linux is due to Android being *based* on the Linux kernel. System calls are essentially **kernel functionalities**, and it is reasonable that what works in Linux should work the same way in Android.

Parallel programs in Android

Last but not least, we should (briefly) answer the last question. By default, an Android process will run only on one execution flow. But, we can tweak this while programming the application, by creating a `ThreadPoolExecutor` and a collection of `Thread` and `Runnable` objects. This is recommended when an app has to perform some performance-impacting calculations; an app is mainly run in the so-called `UI Thread`, which should not be blocked by time-consuming operations. If an app should use this kind of operations, the Android developers recommend us to create a `ThreadPool`. Also, we can run some (pretty simple) parallel programs by `ssh`-ing into an Android device (for example, with the app Termux installed) and coding a C parallel program making use of the `omp.h` library of OpenMP. This has been tried by the author on a Xiaomi Mi A2 Lite (mobile phone) and on a Mi Box 4 (Android TV), if it is of interest, and it works.

4 References

1. @ How Android Manages Processes, by Chris Hoffman. https://www.howtogeek.com/161225/htg-explains-how-android-manages-processes/
2. Google I/O 2014 - The ART runtime, by Google Developers. https://youtu.be/EBlTzQsUoOw.
3. Processes and threads overview, by Android Developers. https://developer.android.com/guide/components/processes-and-threads
4. Services overview, by Android Developers. https://developer.android.com/guide/components/services
5. @ Android Versions Guide: Everything You Need to Know, by Molly McLaughlin. https://www.lifewire.com/android-versions-4173277
6. How apps are built and run on the Android Runtime (ART), by @HeadFirstDroid. https://medium.com/@HeadFirstDroid/how-apps-are-built-and-run-on-the-android-runtime-art-c027f73edb09
7. The Zygote process, by Rasmus Nørgaard. https://medium.com/masters-on-mobile/the-zygote-process-a5d4fc3503db
8. Closer Look At Android Runtime: DVM vs ART, by Ankit Sinhal. https://android.jlelse.eu/closer-look-at-android-runtime-dvm-vs-art-1dc5240c3924
9. https://stackoverflow.com/questions/40336455/difference-between-aot-and-jit-compiler-in-the-art
10. https://stackoverflow.com/questions/9153166/understanding-android-zygote-and-dalvikvm
11. SYSCALLS.TXT. https://android.googlesource.com/platform/bionic/+/refs/heads/android10-release/libc/SYSCALLS.TXT
12. Sending operations to multiple threads, by Android Developers. https://developer.android.com/training/m threads