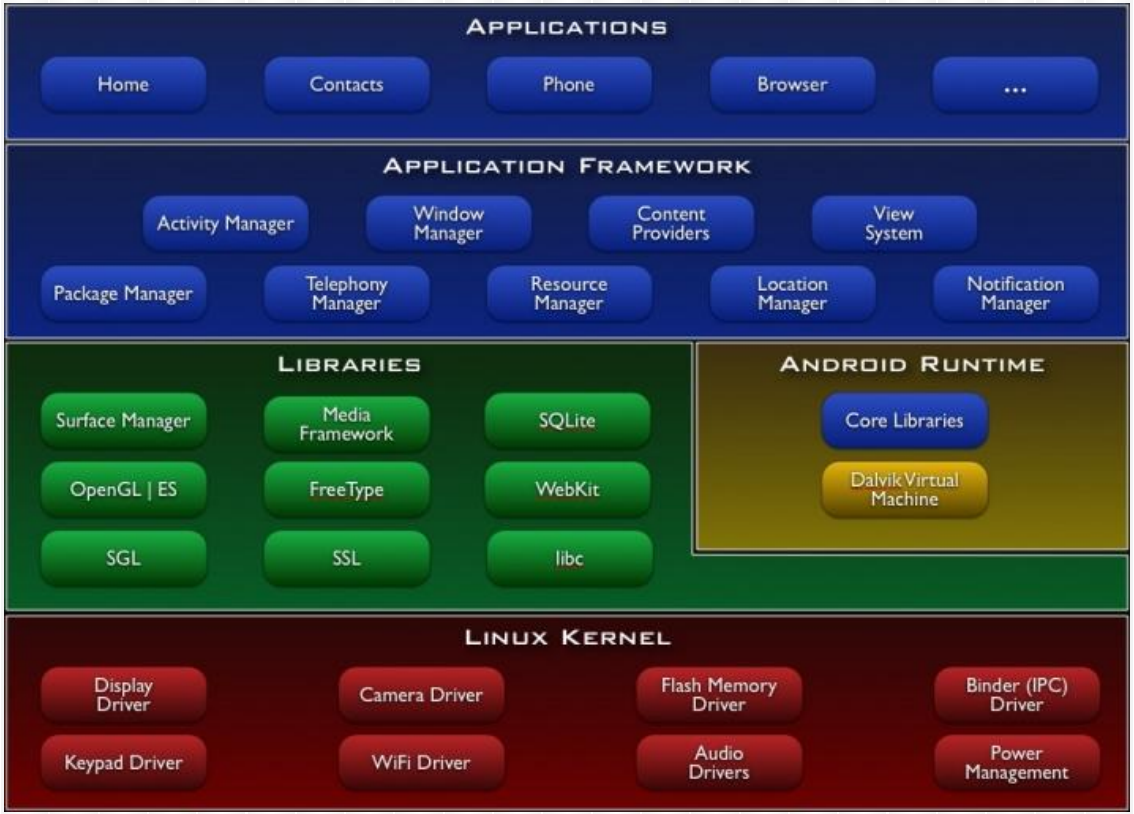
The Android System

Phil Lopreiato

Android System Architecture



Android App Components

- Activity:
  Single screen with user interface

```
<activity
    android:name="com.thebluealliance.androidclient.activities.HomeActivity"
    android:launchMode="singleTop"
    android:theme="@style/AppThemeNoActionBarTranslucentStatus" />
```

- Service:
  Component thats runs in the background

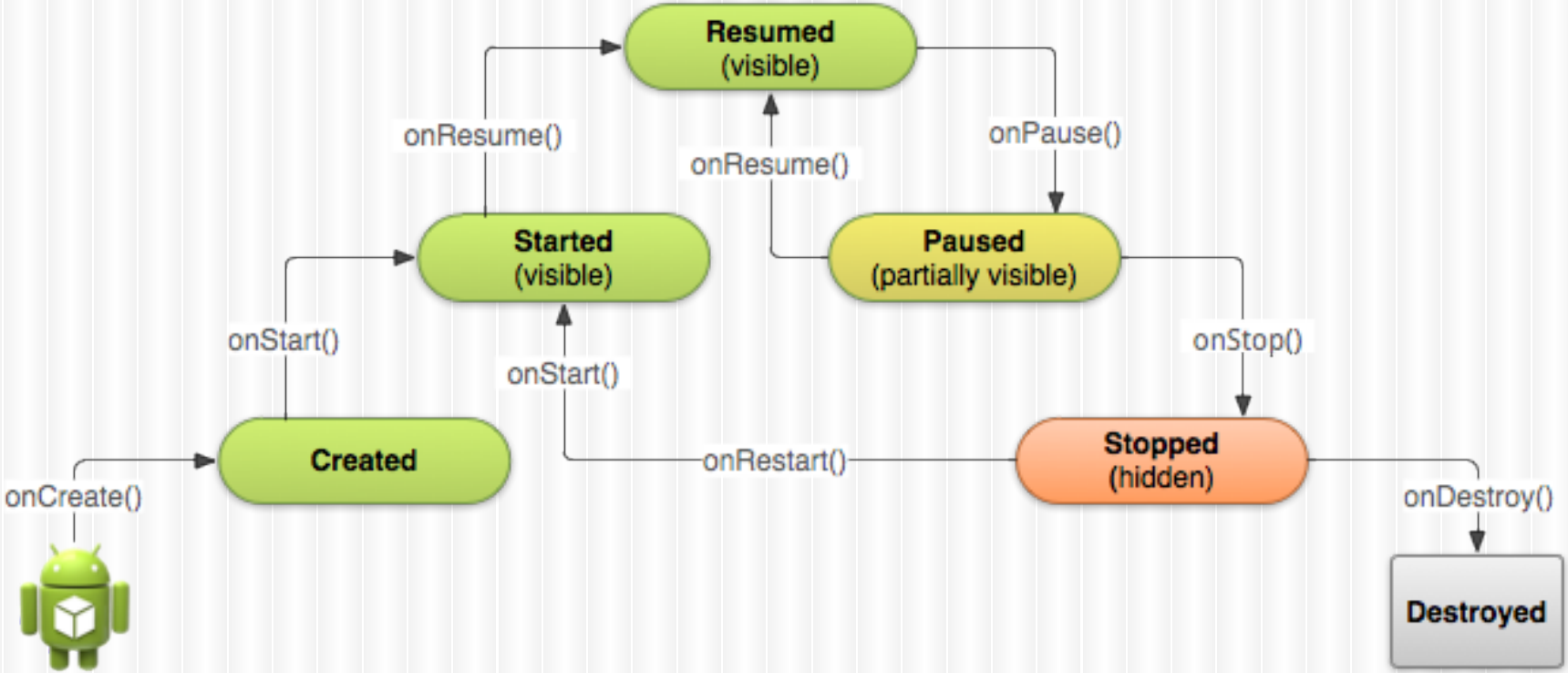
```
<service android:name="com.thebluealliance.androidclient.gcm.GCMMessageHandler" />
```

- Content Provider:
  Manage a set of app data shared with other apps

- Broadcast Receiver:
  Listens for and responds to system broadcasts

```
<receiver
    android:name="com.thebluealliance.androidclient.listeners.NotificationDismissedListener"
    android:exported="false" />
```

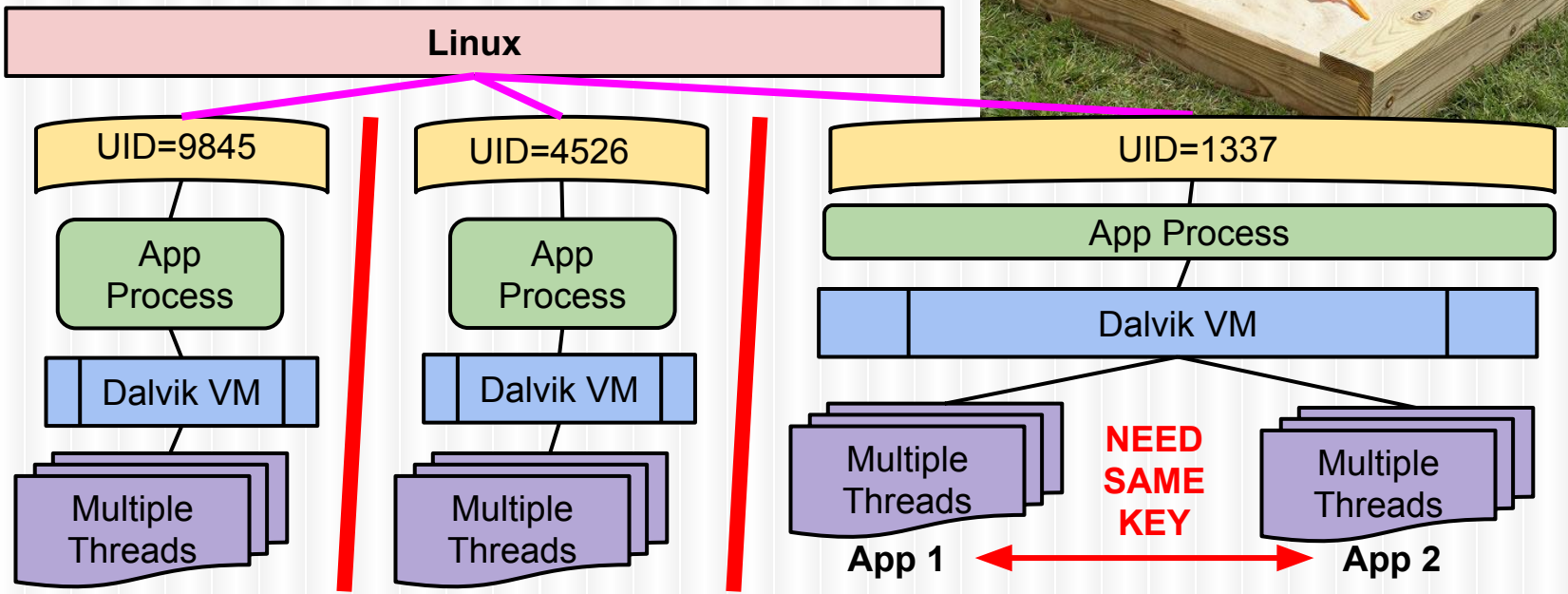
The Activity Lifecycle



See: http://developer.android.com/training/basics/activity-lifecycle/starting.html

An App's Sandbox

- Each app on the system lives in its own sandbox



“Linux” Kernel

- Android runs a modified Linux kernel, currently based on v3.4
  - Contains some additional modules:
    - *Alarm, Ashmem, Binder,* Power Management, *Low Memory Killer,* Kernel Debugger, Logger
  - Open source because GPL

Android v. Linux Kernel

Some Differences (Not Exhaustive)

- Wake Locks & PowerManager
  - Applications can prevent the system from sleeping
  - Adds syscalls for creating, destroying, using wakelocks
- Improved memory management
  - Tailored for systems with small amounts of RAM
  - ASHMEM: named memory block, shared across processes
  - PMEM: allocate named physically contiguous memory
  - Low Memory Killer
    - Terminate unused processes when system has low memory

Moving Up to User Space

Icky?

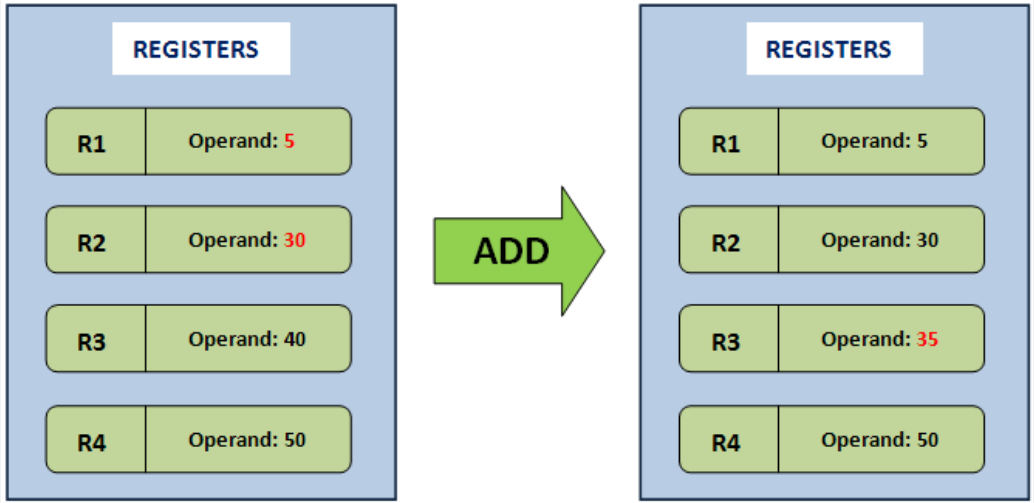
... Actually, Awesome

Bionic C Library

- Android does not use *glibc*
  - Uses *bionic*, an adaption of the BSD C Library optimized for embedded usage
  - Keep GPL out of userspace
- *bionic* is about half the size of *glibc* (at 200 kb)
- Fast *pthread* implementation
  - Uses 4 byte mutex, instead of standard 12
- Has some special Android parts built in

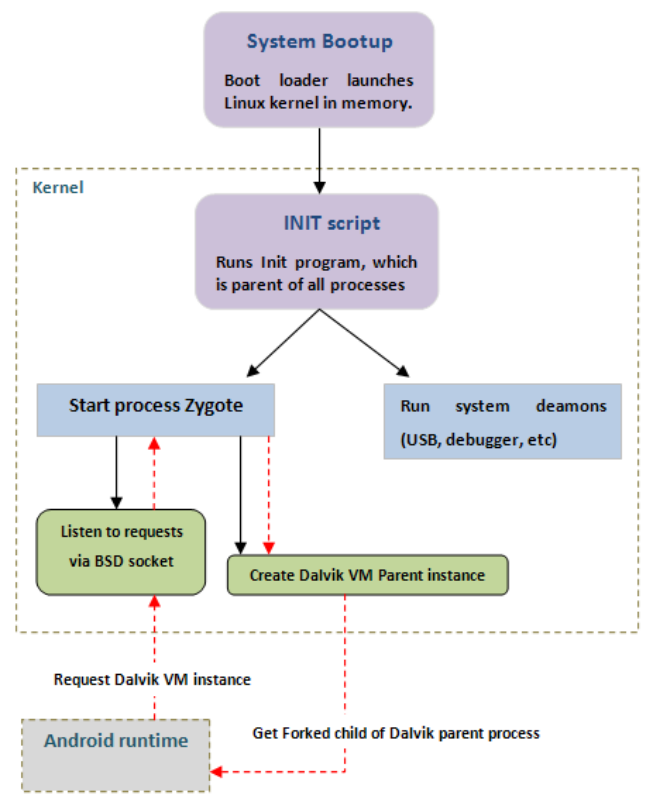
Dalvik Virtual Machine

- Is basically a JVM - allows “write once, run anywhere” code
- Executes “Dalvik Executable” files (.dex)
  - Optimized for low memory footprint
- Is a *register-based* VM - instruction operands are stored in registers



Dalvik Bootup

- Remember: every app runs in its own VM instance, so instances need minimal startup time and memory costs
- The *Zygote* is a process started at boot (think *init*), which preloads a Dalvik instance and some shared libraries
  - New apps open a socket to the Zygote, which *forks* a new VM for it
  - Core libraries are shared across VMs
  - Each VM is garbage collected independently



Just… How?

https://github.com/android/platform_frameworks_base/blob/lollipop-release/

- core/java/com/android/internal/os/Zygote.java
- core/jni/com_android_internal_os_Zygote.cpp

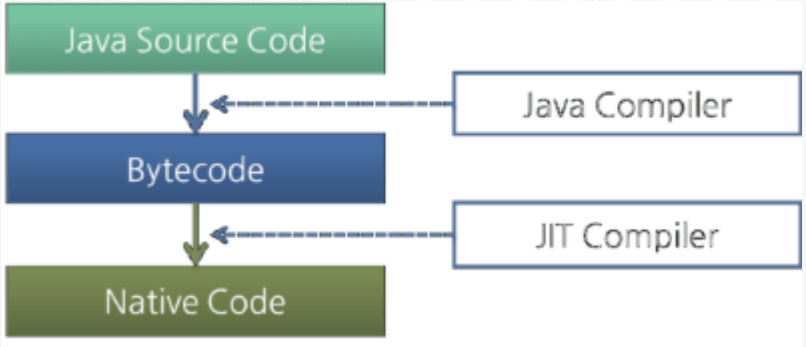
```
87      public static int forkAndSpecialize(int uid, int gid, int[] gids, int debugFlags,
88            int[][] rlimits, int mountExternal, String seInfo, String niceName, int[] fdsToClose,
89            String instructionSet, String appDataDir) {
90          long startTime = SystemClock.elapsedRealtime();
91          VM_HOOKS.preFork();
92          checkTime(startTime, "Zygote.preFork");
93          int pid = nativeForkAndSpecialize(
94                    uid, gid, gids, debugFlags, rlimits, mountExternal, seInfo, niceName, fdsToClose,
95                    instructionSet, appDataDir);
96          checkTime(startTime, "Zygote.nativeForkAndSpecialize");
97          VM_HOOKS.postForkCommon();
98          checkTime(startTime, "Zygote.postForkCommon");
99          return pid;
100     }
101
```

JIT All The Things

- DEX bytecode is compiled to Dalvik bytecode “Just In Time” for use
- Only “hot” (soon to be used) code is compiled
- Trace length of 100 opcodes
- Threads in the same process share a trace cache
- Consecutive traces can be chained together to increase performance
- JIT generates Dalvik bytecode, usually

NOT USED ANYMORE

- Starting in Android 4.4, ART (Android Runtime) replaced Dalvik
- ART will compile Ahead Of Time (AOT) at system install time



Disadvantages of Dalvik

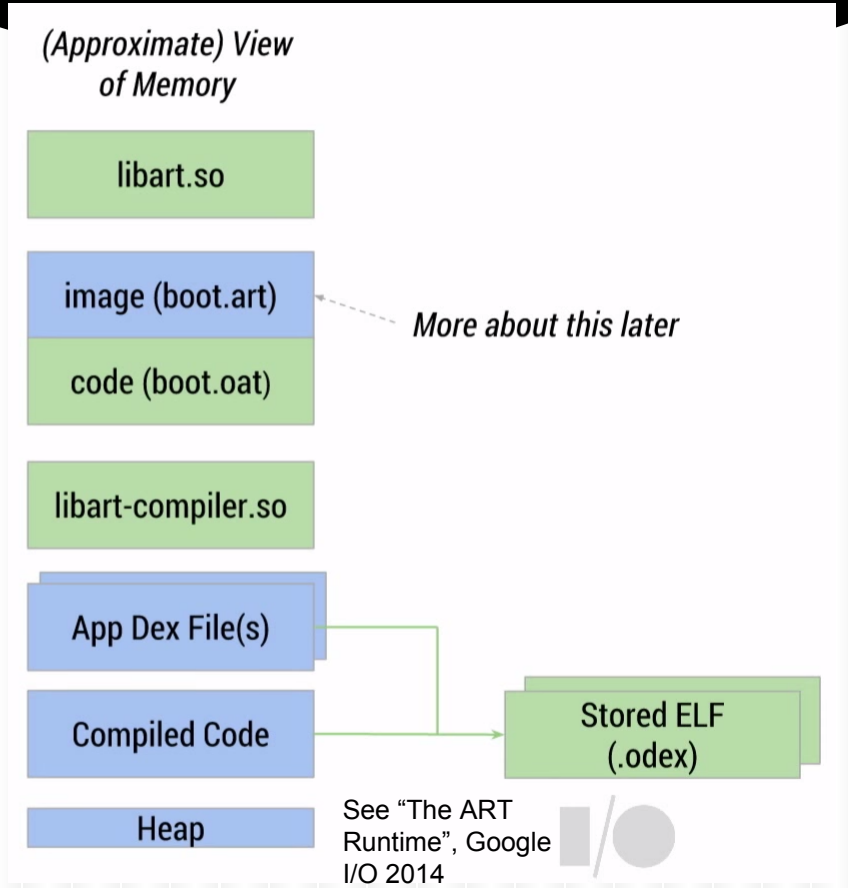
- Whenever a new application launches…
  - It needs to be JIT compiled and loaded into RAM
  - Sometimes cached (less effective as app sizes grew)

- These compilations happen while the user is engaged
  - Waiting for app to load
  - Can cause stutters if the system is low on memory

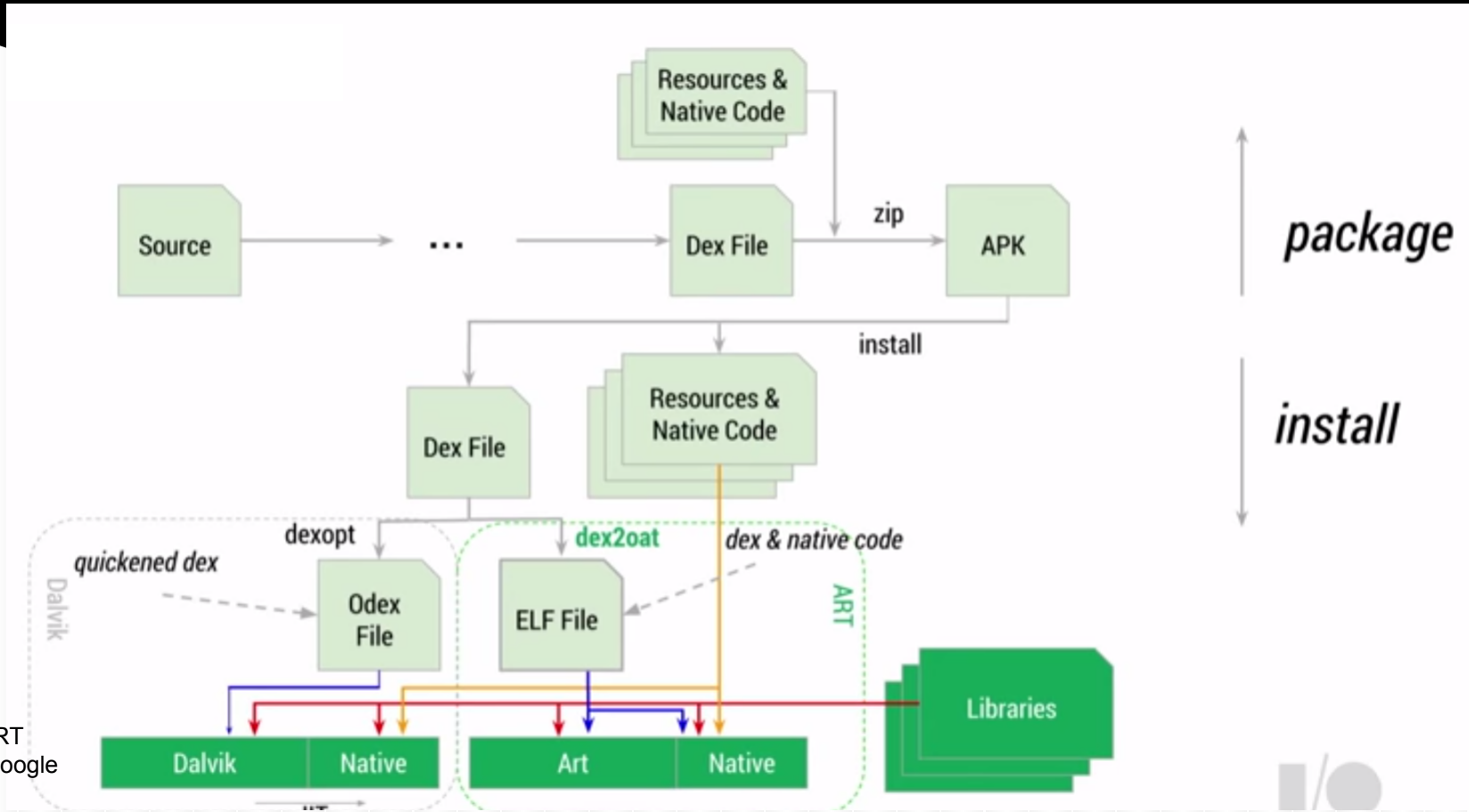
Why was this the original design choice, then?

Android Runtime (ART)

- Acts as Ahead of Time (AOT) compiler
  - Apps are compiled to *native code* **before** they're being used
    - At app install time
- Improves garbage collection (one pause instead of two)
  - Lower GC time - shouldn't be noticeable by user
- Optimize for Object Oriented Programs
- System performance improvements
  - Better battery life
  - Better memory footprint

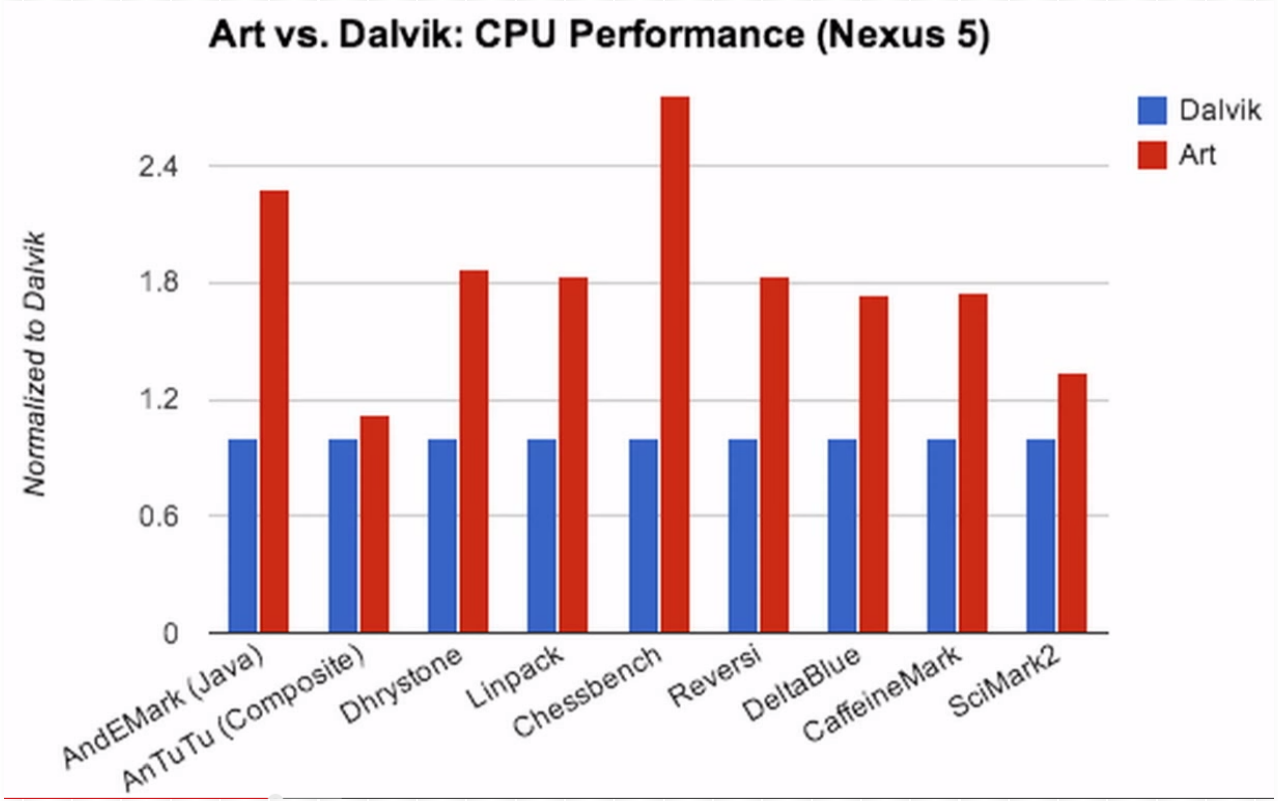


Life of an APK



See “The ART Runtime”, Google I/O 2014

Performance Boosting Thing



See “The ART Runtime”, Google I/O 2014

Security

- Android security is heavily dependent on Linux's multi-user features
- Because applications run in VMs, Dalvik controls code execution and resource accesses
  - Provides a security benefit, since the VM acts as a "gatekeeper" to the full system
  - All external requests are denied, unless the app requests the proper permission (enforced at runtime)
  - With new systems running ART, SELinux picks up the slack
- All app files are private to its assigned UID (at install time)
- Individual *components* can also be restricted
  - Only certain sources can start Activity, Service, or send broadcast to a receiver

Three Layers of Security

Divide Persistent Memory

- System partition is mounted as Read-Only
- Applications store data on its own R/W partition

Linux Discretionary Access Control (DAC)

- Each app's UID and GID can only access their files
- Only apps signed by the same author can run under the same UID
  - System sets up and enforces this

Android Permissions

- More granular access rights
- Mandatory Access Control (MAC) policy
- Requested at application install time, enforced at runtime
- Apps whitelist *Intents* they're allowed to receive

SELinux

- Linux Kernel security module, enforces Mandatory Access Control (MAC)
- Kernel enforces a set of “rules” on object accesses

Benefits to Android:

SELinux can confine the privileged system daemons, limiting the damage they can cause

SELinux can monitor and control interactions between apps and the kernel & system resources

SELinux provides a centralized, system-wide security policy configuration

So how can apps do IPC (Inter-Process Communication)?

IPC: Binders

Binders

- Binders provide access to functions and data between execution environments
- Android uses a custom implementation of *OpenBinder*, which extends the traditional IPC mechanisms
- Allow an app developer to call methods on *remote objects* as if they existed within *local objects*
- Binders are managed by an Android system service
  - Processes can *link-to-death* (get informed when a Binder is terminated)
- Each Binder is uniquely identifiable (can be used as security token)

Binder Kernel Driver

- Kernel Driver is written in C
- Has operations: *open*, *mmap*, *release*, *poll*, *ioctl*
- *ioctl* syscall is the main point of interface with higher layers
  - Takes a series of flags to define operations

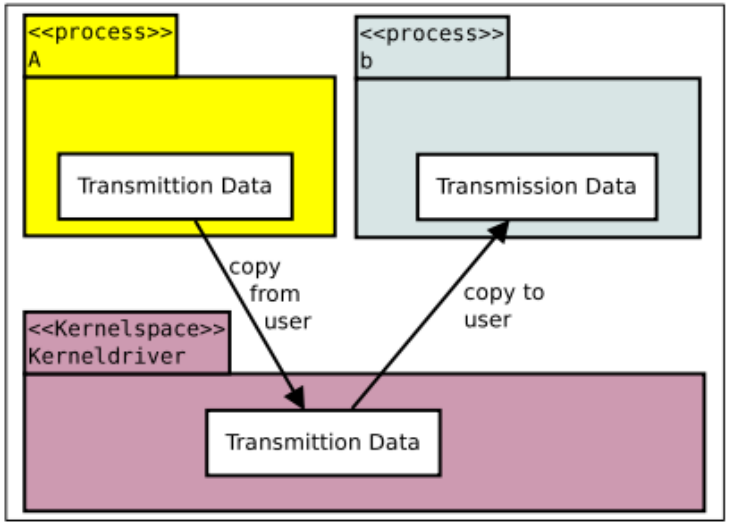
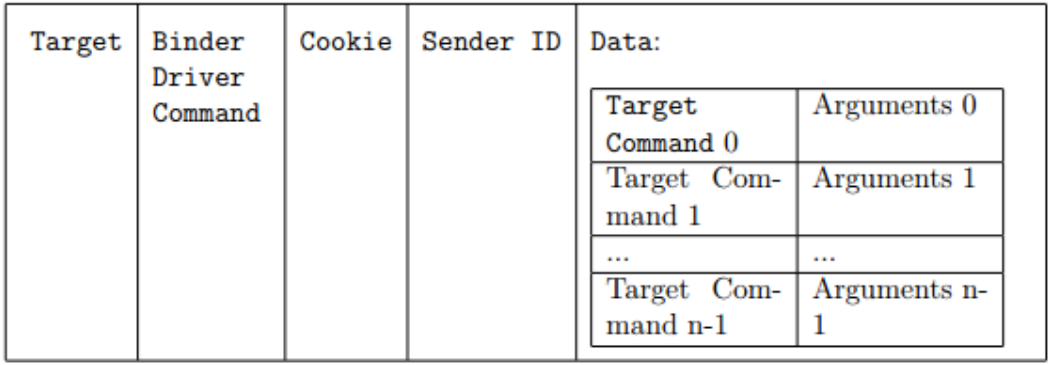


Figure 5.4.: Data Transaciton

Figure from Schreiber's paper

Binder Middleware

- Middleware written in C++
- Implements user space functionality for Binder framework
- Allow transformation of object for transmission
- Contains a JNI Wrapper so Java classes can use it



Target	Binder Driver Command	Cookie	Sender ID	Data:

Target Command 0	Arguments 0
Target Command 1	Arguments 1
...	...
Target Command n-1	Arguments n-1

Figure 4.3.: Transmission Data

Figure from Schreiber's paper

Binder API

- Android system provides a Java API for interacting with Binders
- Wraps the functionality of the middleware layer so that apps can communicate with Binders
- Introduces some additional abilities into the Binder framework
  - Namely intents

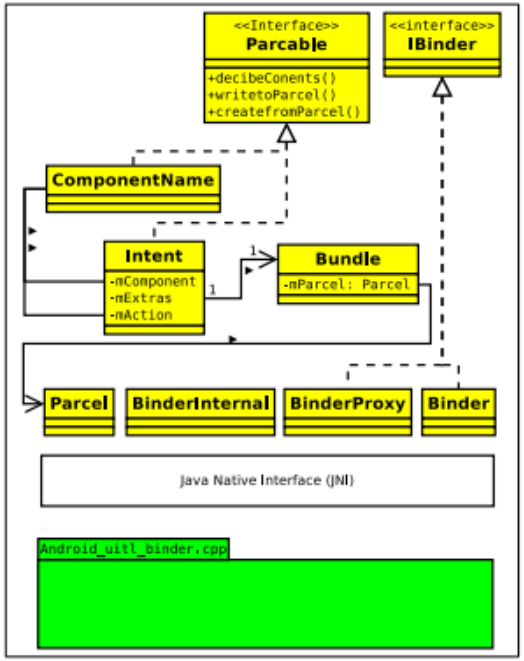


Figure 5.2.: Java System

Figure from Schreiber's paper

Questions

This presentation can be found at http://phil-lopreiato.github.io/droidsys/

References

- Patrick Brady, Google I/O 2008, "Anatomy and Physiology of an Android
- Stefan Brähler, Analysis of the Android Architecture (2010)
- Brian Carlstrom, Anwar Ghuloum, Ian Rogers, Google I/O 2014, "The ART Runtime"
- David Ehringer, The Dalvik Virtual Machine Architecture (2010)
- Thorsten Schreiber, Android Binder: Android Interprocess Communication (2011)
- Mark Sinnathamby, Stack based vs Register based Virtual Machine Architecture, and the Dalvik VM
- Stephen Smalley and Robert Craig, Security Enhanced (SE) Android: Bringing Flexible MAC to Android (2013)