Von Geysiren und Kaffeebohnen

Android-Laufzeitumgebung – Teil 1: Die (virtuelle) Maschine

Jörg Pleumann

Android-Applikationen werden überwiegend in Java implementiert. Die Laufzeitumgebung macht Android „weitgehend" zu einem gewöhnlichen Java-System. Die Einschränkung „weitgehend" signalisiert es schon: Es existieren einige subtile Unterschiede zwischen einer Java-Umgebung von Sun (jetzt Oracle) und der in Android. Für Entwickler sind diese durchaus interessant, insbesondere, wenn man Code zwischen beiden Plattformen portieren möchte. In einer zweiteiligen Artikelserie soll daher die Laufzeitumgebung von Android beleuchtet werden.

Abbildung 1 verortet die Laufzeitumgebung in der gesamten Architektur der Plattform. Sie setzt sich aus zwei wesentlichen Komponenten zusammen:

- der virtuellen Maschine (VM), die in Android auf den Namen *Dalvik* hört, und
- einer umfangreichen Laufzeitbibliothek, den sogenannten *Core Java Libraries*.

Die Dalvik VM

Die Dalvik VM, in der jeglicher Android-Bytecode ausgeführt wird, gehört zu den Komponenten der Plattform, die komplett neu entwickelt wurden. Der Name der VM leitet sich von der isländischen Hafenstadt Dalvik ab, für die sich der Google-Ingenieur Dan Bornstein während des Projekts als Urlaubsziel interessierte. Entgegen anderslautender Gerüchten, die es von einem Wikipedia-Scherz immerhin bis in verschiedene Fachbücher geschafft haben, stammt jedoch keiner seiner Verwandten von dort.

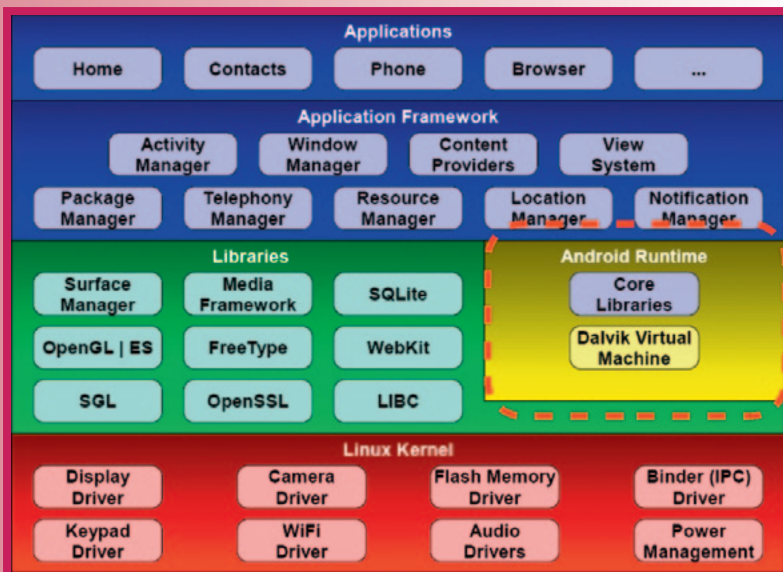

Abb. 1: Die Laufzeitumgebung innerhalb der Android-Architektur

Die minimalen Eckdaten eines Systems, auf dem die Dalvik VM lauffähig ist, lesen sich ein wenig wie die Beschreibung eines Desktop-PCs aus den späten 90er Jahren, nur dass dieser PC heute nicht mehr unförmig, laut und beige daherkommt, sondern in der Form eines – je nach Modell mehr oder minder eleganten – Mobiltelefons unser ständiger Begleiter ist:

- Taktfrequenz der CPU mindestens 250 – 500 MHz,
- Hauptspeicher minimal 64 MByte,
- keine Auslagerungsdatei, also kein virtueller Speicher,
- Batteriebetrieb.

Es können mehrere Instanzen der Dalvik VM in getrennten Linux-Prozessen ausgeführt werden. Dieser Mechanismus wird in Android genutzt, um die einzelnen Anwendungen gegeneinander zu isolieren, trägt also zur Sicherheit und Stabilität des Systems bei. Die Dalvik VM besitzt zudem – wie wir im zweiten Teil des Artikels sehen werden – eine reichhaltige Laufzeitbibliothek, die zusammen mit dem Android -Framework den Baukasten bildet, aus dem sich Anwendungsentwickler bedienen.

Schaut man sich diese Eigenschaften – speziell den Batteriebetrieb – an, dann wird relativ schnell klar, dass Effizienz bei der Dalvik VM eine hohe Bedeutung zukommt. Die VM muss zum einen dafür sorgen, dass die CPU so wenig wie nötig genutzt wird, da jeder Taktzyklus der CPU die Batterie belastet. Zum anderen muss der Wunsch nach einer reichhaltigen Laufzeitbibliothek in Einklang mit dem begrenzten Hauptspeicher und den mehreren Instanzen der VM gebracht werden.

Vermehrung durch Zellteilung

Zur Erzeugung zusätzlicher Instanzen und zur Lösung des Speicherproblems bedient sich die Dalvik VM eines Modells, das dem traditionellen `init`-Prozess eines Unix-Systems ähnelt und dieses in die Java-Welt überträgt. Beim Hochfahren des Systems wird zunächst die sogenannte „Zygote" gestartet, quasi die Mutter aller Java-Prozesse. Die Zygote enthält eine vollständig initialisierte Instanz der Dalvik VM inklusive solcher Klassen aus sowohl der Laufzeitbibliothek als auch dem Framework, die von praktisch al-

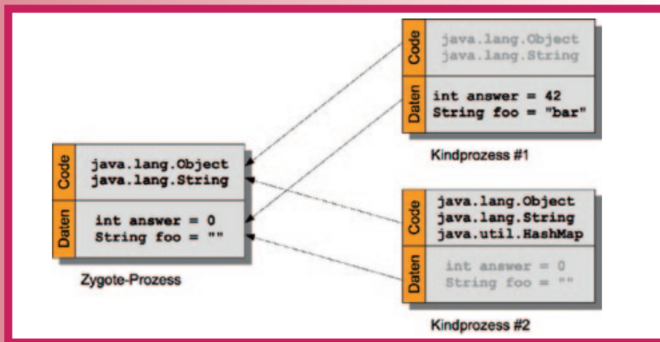

Abb. 2: Die Zygote und ihre Kinder

len Java-Prozessen benötigt werden (ein triviales Beispiel wäre `java.lang.Object`).

Die Zygote kann über einen speziellen Aufruf geklont werden. Beim Klonen entsteht ein unabhängiger Prozess mit eigenem logischen Adressraum, der aber mittels der Linux-Speicherverwaltung zunächst auf die gleichen physikalischen Speicherbereiche zugreift wie die Zygote. Erst wenn der Kindprozess eine dieser Speicherseiten verändert, erhält er seine unabhängige Kopie („copy-on-write"-Technik). Da große Teile des Speichers eines Java-Prozesses aber nie wieder verändert werden (z. B. die VM selbst oder der Bytecode der einzelnen Java-Klassen), können sich die Zygote und ihre Kinder dauerhaft Speicher teilen und kommen so gut mit dem begrenzten Hauptspeicher aus.

Abbildung 2 verdeutlicht das Prinzip für zwei Kindprozesse. Die hellgrau beschrifteten Speicherbereiche entsprechen denen der Zygote und belegen daher keinen eigenen physikalischen Speicher.

Register-Architektur

Das Problem der CPU-Effizienz wird teilweise dadurch gelöst, dass die Dalvik VM von der üblichen Architektur einer JVM abweicht. Traditionell basiert eine JVM auf einer Stack-Architektur, d. h. der „simulierte Prozessor" verfügt über einen Stack, über den die einzelnen Bytecode-Instruktionen ihre Argumente und Ergebnisse miteinander austauschen. In dieser Architektur werden relativ viele zusätzliche Instruktionen benötigt, um den Stack zu bedienen.

Die Dalvik VM hingegen basiert auf einer Register-Architektur. Argumente und Ergebnisse werden nicht über einen Stack, sondern über eine endliche Anzahl von Registern – nicht unähnlich lokalen Variablen – ausgetauscht. Dies ist zum einen näher an der tatsächlichen Prozessorarchitektur (speziell im Fall des ARM-Prozessors). Zum anderen werden bei der Dalvik VM meist weniger Instruktionen für die gleiche Aufgabe benötigt, weil die zu verwendenden Register jeweils in die Instruktion codiert sind. Dafür ist die Breite einer Codeeinheit bei Dalvik nicht 8, sondern 16 Bit.

Instruktionssatz und Binärformat

Der Instruktionssatz der Dalvik VM, also die Menge der unterstützten Bytecode-Instruktionen, ähnelt vom Umfang und den Möglichkeiten her sehr dem, was bei der JVM seit langem Standard ist. Die geänderte Architektur findet natürlich ihren Niederschlag in den einzelnen Instruktionen, aber wie Listing 1 für den Klassiker

```
System.out.println("Hallo, Welt!");
```

zeigt, erkennt man die JVM in Dalvik durchaus wieder.

JVM-Bytecode

```
0: getstatic java.lang.System.out Ljava/io/PrintStream;
3: ldc "Hallo Welt!"
5: invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
8: return
```

Dalvik-Bytecode

```
0: sget-object v0, java.lang.System.out Ljava/io/PrintStream;
2: const-string v1, "Hallo Welt!"
4: invoke-virtual {v0, v1}, java/io/PrintStream/println(L...;)V
7: return-void
```

Listing 1: Bytecode-Vergleich zwischen einer JVM und Dalvik

Gleichzeitig existieren jedoch auf Bytecode-Ebene in der Dalvik VM einige neue Instruktionen, die eine JVM nicht kennt und die an einigen Problemstellen für erheblich effizienteren Code sorgen können. Ein Beispiel ist eine neue Instruktion, die ein Array der Art

```
public static final char[] DATA = {
  "J", "a", "v", "a", "S", "p", "e", "k", "t", "r", "u", "m"
};
```

aus einer Tabelle initialisieren kann. Eine JVM muss für jedes Element des Arrays mehrere Instruktionen ausführen, was den Code unnötig aufbläst. Listing 2 zeigt den entstehenden Bytecode für beide Plattformen im direkten Vergleich. Die JVM braucht über 70 Bytes, während Dalvik mit 26 Einheiten zu je 16-Bit auskommt und bei größeren Arrays weniger stark wächst. Wie groß der Unterschied bei z. B. einer CRC-Tabelle ausfällt, kann man sich leicht überlegen.

JVM-Bytecode

```
 0: bipush 12
 2: newarray char
 4: dup
 5: iconst_0
 6: bipush 75   // J
 8: castrore
     ...
64: dup
65: bipush 11
67: bipush 109  // M
69: castore
70: putstatic DATA
73: return
```

Dalvik-Bytecode

```
 0: const/16 v0, #int 12
 2: new-array v0, v0, [C
 4: fill-array-data v0, 0a
 7: sput-object v0, DATA: [C
 9: return-void
10: array-data (12 units)
         'J', 'a', 'v', 'a', ...
```

Listing 2: Bytecode zum Initialisieren eines Arrays

Einher mit dem geänderten Instruktionssatz geht ein eigenes Binärformat. Die Dalvik VM ist nicht in der Lage, die üblichen `.class`-Dateien auszuführen, die der Java-Compiler für die JVM erzeugt. Stattdessen findet ein neues Format namens *Dalvik Executable* (DEX) Verwendung. Auch hier sind einige Verbesserungen eingeflossen. Die wesentliche ist, dass eine DEX-Datei die Verschmelzung mehrerer Java-Klassen ist. Im Unterschied zu einer `.jar`-Datei findet die Verschmelzung jedoch bereits auf Ebene des Binärcodes statt. Redundante Informationen – etwa Import-/Export-Informationen und

Strings – werden nur einmal gespeichert. Als Ergebnis ist eine unkomprimierte `.dex`-Datei im Normalfall immer noch kleiner als die äquivalente, komprimierte `.jar`-Datei mit Code für eine JVM.

Verifikation von Bytecode

Die Dalvik VM führt Überprüfungen des Bytecodes durch, die äquivalent zu denen einer JVM sind. Die Verifikation von Bytecode spielt jedoch auf einem Android-System nicht die gleiche Rolle wie auf üblichen Java-Systemen. Grund dafür ist die Tatsache, dass sich Android stärker auf das Sicherheitsmodell von Linux – speziell die Isolation der Prozesse und die Zugriffsrechte im Dateisystem – verlässt als auf die Sandbox von Java. Trotzdem ist es sinnvoll, den Bytecode beim (ersten) Start zu überprüfen. Wenn einmal sichergestellt ist, dass jede Instruktion auf allen möglichen Pfaden durch den Code „gutartig" sein muss, kann diese Überprüfung bei deren Ausführung entfallen, was die Ausführungsgeschwindigkeit erhöht.

Interpreter versus Compiler

In ihrer ursprünglichen – und bis zur Version 2.1 gültigen – Form war die Dalvik VM ein reiner Interpreter. Es existierte kein Just-In-Time (JIT) oder Ahead-Of-Time (AOT) Compiler, der den Bytecode in Maschinencode für die jeweilige Architektur (z. B. den ARM-Prozessor) umwandelt. Damit war Dalvik von der Ausführungsgeschwindigkeit gegenüber einer aktuellen JVM von Sun, die auf vergleichbarer Hardware läuft, klar im Nachteil, benötigte aber auch wesentlich weniger Speicher als diese. Seit der Version 2.2 von Android besitzt Dalvik einen JIT-Compiler für die ARM-Architektur. Google gibt für diesen einen Geschwindigkeitszuwachs um den Faktor 5 an. In einzelnen Benchmarks liegt der Zuwachs jedoch noch deutlich höher. Auch im täglichen Android-Leben wirkt sich der JIT-Compiler positiv aus: Anwendungen starten schneller und die Benutzungsoberfläche reagiert merklich zügiger.

Fehlende Abfallpresse

Die Dalvik VM führt derzeit eine nicht kompaktierende Garbage Collection durch, d. h. sie ist nicht in der Lage, belegte Speicherblöcke so zusammenzuschieben, dass sich ein großer freier Bereich ergibt. Als Resultat kann der Heap über die Zeit so stark fragmentieren, dass Speicheranforderungen nicht mehr befriedigt werden können, obwohl in der Summe genügend Speicher vorhanden ist. Dieses Problem ist jedoch bei Android eher theoretischer Natur, da die meisten Anwendungen durch das Aktivitätsmodell (siehe [GrHi08] und [GrPl08]) eher kurzlebig sind. Trotzdem kann es natürlich nicht schaden, sparsam mit dem Speicher umzugehen.

Java Native Interface

Obwohl Java derzeit die primäre Sprache zur Entwicklung von Anwendungen für Android ist, besteht die Möglichkeit, nativen, also z. B. in C oder C++ entwickelten Code in eine Anwendung einzubetten. Die Dalvik VM unterstützt das übliche Java Native Interface (JNI) für die Kommunikation zwischen nativem Code und interpretiertem Bytecode und ein komplettes Native Development Kit (NDK), um derlei gemischte Anwendungen zu entwickeln. Dem Zugewinn an Ausführungsgeschwindigkeit steht jedoch die Aufgabe der Plattformunabhängigkeit gegenüber, sodass man als Entwickler jeweils im Einzelfall entscheiden muss, ob man diese Möglichkeit nutzen möchte. Der JIT-Compiler hat das Geschwindigkeitsverhältnis von Bytecode und nativem Code zusätzlich verschoben.

Kommandozeilenschnittstelle

Dieses Feature dürfte weitgehend unbekannt sein: Obwohl Android-Anwendungen meist eine grafische Oberfläche besitzen und über den Home Screen gestartet werden, ist die Dalvik VM eine eigenständig ausführbare Komponente. Sie kann über eine Kommandozeile (z. B. ein Terminal-Programm mit einer `adb shell`) aufgerufen werden und versteht viele der Parameter, die auch bei einer Sun JVM Verwendung finden. Insbesondere kann über `-classpath` festgelegt werden, wo der Bytecode zu finden ist, der natürlich hier im DEX-Format vorliegen muss. Auch akzeptiert die VM den Namen einer Klasse, deren `main()`-Methode ausgeführt werden soll.

Zusammenfassung und Ausblick

Dieser erste Artikel zur Android-Laufzeitumgebung hat sich mit der Dalvik VM beschäftigt, die an einigen Stellen geschickt mit Traditionen aus dem Java-Umfeld bricht und dadurch an für mobile Geräte wichtiger Effizienz bezüglich Speicher und Batterie gewinnt. Insbesondere besitzt sie einen neuen Befehlssatz, der einige Java-Konstrukte kompakter auszudrücken vermag als die klassische JVM.

Im zweiten Artikel soll die Laufzeitbibliothek diskutiert werden. Diese bestimmt aus Sicht eines Einwicklers unmittelbar, wie viel Java in Android „steckt" und wo die Fallstricke bei der Portierung von existierendem Code verborgen sind.

Literatur

[GrHi08] Th. Grothaus, B. Hinüber, Android im Überblick, in: JavaSPEKTRUM, 2/2008,
s. a. `http://www.sigs-datacom.de/wissen/artikel-fachzeitschriften/artikelansicht.html?show=2244`

[GrPl08] D. Gruntz, J. Pleumann, Programmierkonzepte von Android am Beispiel, in: JavaSPEKTRUM, 4/2008,
s. a. `http://www.sigs-datacom.de/wissen/artikel-fachzeitschriften/artikelansicht.html?show=2244`

Jörg Pleumann verfügt über langjährige Erfahrung im Bereich mobiles und eingebettetes Java. Er leitet die Android-Entwicklungsgruppe bei der Noser Engineering AG in Winterthur, Schweiz. Noser Engineering ist Gründungsmitglied der Open Handset Alliance, war unmittelbar an der Entwicklung von Android beteiligt und hat z. B. die Core Java Libraries und den größten Teil der offiziellen Compatibility Test Suite (CTS) beigesteuert.
E-Mail: joerg.pleumann@noser.com