# Эффективное использование потоков в операционной системе Android

Технологии асинхронной обработки данных

*Андерс Ёранссон*

ДМК Пресс издательство

O'REILLY®

**УДК 004.451.9Android:004.451.2**
**ББК 32.972.11**
**Е69**

**Ёранссон А.**
Е69 Эффективное использование потоков в операционной системе Android / пер. с англ. А. В. Снастина. – М.: ДМК Пресс, 2015. – 304 с.: ил.

**ISBN 978-5-97060-168-6**

Чтобы написать действительно полезное и удобное приложение для Android, то без многопоточности никак не обойтись, но как узнать о технологиях и методах, которые помогут решить такую задачу? Книга с практической точки зрения описывает несколько асинхронных механизмов, доступных в программной среде Android SDK, а также рассматривает основные принципы и правила выбора одного из них, лучше всего подходящего для создаваемого приложения.

Издание предназначено для программистов разной квалификации, уже работающих под Android и желающих улучшить качество создаваемых программ.

УДК 004.451.9Android:004.451.2
ББК 32.972.11



ISBN 978-1-449-36413-7 (анг.) 
ISBN 978-5-97060-168-6 (рус.) 

# Содержание

# Глава 1

# Компоненты ОС Android и необходимость параллельных вычислений

Прежде чем углубляться в многопоточный мир, для начала следует поближе познакомиться с платформой Android, с архитектурой её приложений и с особенностями выполнения приложений в этой среде. Данная глава закладывает основы, необходимые для дальнейшего обсуждения многопоточности в остальной части книги. Всё же следует отметить, что более полную информацию о платформе Android можно получить из официальной документации (**https://developer.android.com**) или из многочисленных книг по программированию в ОС Andriod, имеющихся в продаже.

## Стек программной среды ОС Android

Приложения Android запускаются на верхнем уровне стека программной среды[1], в основе которого находится ядро Linux, уровнем выше – системные библиотеки на языках C/C++, и среда времени выполнения (runtime), отвечающая за выполнение кода приложения (рис. 1.1).

---

[1] Термин «стек программной среды» введён здесь по аналогии с термином «стек сетевых протоколов» во избежание путаницы, поскольку программным стеком уже достаточно давно называют область памяти, выделяемую программе для организации собственного внутреннего стека. – *Прим. перев.*

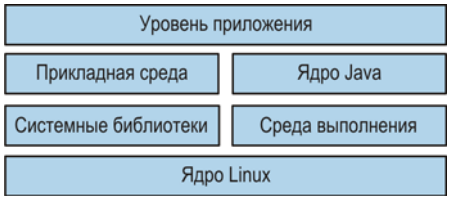


**Рис. 1.1** ❖ Стек программной среды ОС Android

Главные составные части стека программной среды операционной системы (ОС) Android перечислены ниже:

- *Приложения* – приложения для платформы Android, написанные на языке Java, которые используют библиотеки языка Java и прикладной среды Android.
- *Ядро Java* – библиотеки ядра языка Java, используемые приложениями и прикладной средой. Это ядро не является полностью совместимым с реализацией Java SE или Java ME, а представляет собой некоторое подмножество более ранней, уже вышедшей из употребления реализации Apache Harmony, основанной на версии Java 5. Библиотеки ядра Java предоставляют основные механизмы работы с потоками: класс `java.lang.Thread` и пакет `java.util.concurrent`.
- *Прикладная среда* – классы платформы Android для работы с оконной системой (оконным интерфейсом), с компонентами графического пользовательского интерфейса, с ресурсами и т. п., то есть практически со всем, что необходимо для написания Android-приложения на языке Java. Прикладная среда определяет жизненные циклы компонентов Android и управляет этими циклами, а также взаимодействием компонентов и обменом данными между ними. Кроме того, прикладная среда определяет набор специализированных для платформы Android механизмов асинхронного выполнения, которыми приложения могут пользоваться для упрощения управления потоками: `HandlerThread`, `AsyncTask`, `IntentService`, `AsyncQueryHandler` и `Loaders`. Все эти механизмы будут подробно описаны в данной книге.
- *Системные библиотеки* – библиотеки на C/C++, которые работают непосредственно с графикой, системными ресурсами, базами данных, шрифтами, программным интерфейсом OpenGL и т. д. Обычно приложения на языке Java не взаимодействуют

напрямую с системными библиотеками, поскольку прикладная среда предоставляет функции-обёртки на Java, скрывающие обращения к этим библиотекам.

- *Среда выполнения* – надёжно изолированная и защищённая среда, выполняющая в виртуальной машине код Android-приложения, скомпилированный во внутренний байт-код. Каждое приложение работает в собственной среде выполнения – это либо Dalvik, либо ART (Android Runtime). Последняя была добавлена в версию KitKat (API level 19) как дополнительная. Ее можно активировать или отключить, но на момент написания данной книги Dalvik остаётся средой выполнения по умолчанию.
- *Ядро Linux* – заложенное в основу платформы ядро операционной системы Linux, позволяющее приложениям использовать аппаратные функции устройств: звуковых, сетевых, видео и т. п., а также управляющее процессами и потоками. Процесс порождается для каждого приложения, и каждый процесс предоставляет запускаемому приложению собственную среду выполнения. Внутри процесса код приложения может выполняться несколькими потоками. Ядро ОС распределяет доступное процессорное время, необходимое для выполнения кода, между процессами и потоками в них с помощью *механизма планирования* (*scheduling*).

# Архитектура приложения

Основными элементами любого приложения являются объект `Application` и компоненты программной платформы Android: `Activity`, `Service`, `BroadcastReciever` и `ContentProvider`.

## Приложение

В языке Java выполняющееся приложение представляет объект `android.app.Aplication`, экземпляр которого создаётся при запуске и уничтожается при завершении приложения (то есть время существования экземпляра класса `Application` совпадает со временем существования соответствующего процесса Linux). Когда процесс завершается и перезапускается, создаётся новый экземпляр класса `Application`.

## Компоненты

Основными составными частями любого Android-приложения являются компоненты, управляемые средой выполнения: `Activity`, `Service`,

`BroadcastReceiver` и `ContentProvider`. Конфигурация и взаимодействие этих компонентов определяют поведение приложения. Перечисленные выше компоненты имеют разные области ответственности и жизненные циклы, но все они являются точками входа, посредством которых можно запускать приложения. После начала работы любой компонент может активировать другой компонент и т. д. на протяжении всего времени выполнения приложения. Запуск компонента в текущем либо в другом приложении производится с помощью объекта `Intent`[1]. Объект `Intent` определяет операции, запрашиваемые у принимающей стороны или получателя (receiver), например отправка электронной почты или фотографирование, и может передавать данные от отправителя (sender) получателю. Объект `Intent` может быть явным или неявным:

- *явный* объект `Intent` – определяет полностью классифицированное имя компонента, известное внутри приложения во время компиляции;
- *неявный* объект `Intent` – среда выполнения создаёт связь с компонентом, для которого определён набор характеристик в `IntentFilter`. Если `Intent` обнаруживает совпадение характеристик компонента с `IntentFilter`, такой компонент может быть активирован.

Компоненты и их жизненные циклы относятся к терминологии, специализированной для ОС Android, поэтому нельзя установить однозначное соответствие с объектами языка Java, лежащими в их основе. Объект Java может существовать дольше соответствующего ему компонента, а среда выполнения может содержать несколько Java-объектов, относящихся к одному и тому же активному компоненту. Из-за этого могут возникать некоторые недоразумения, и, как мы увидим в главе 6, такая ситуация увеличивает вероятность утечек памяти.

Реализация компонента в приложении осуществляется созданием производного класса, и все компоненты в приложении должны быть зарегистрированы в файле *AndroidManifest.xml*.

### *Activity*

Компонент `Activity` – это образ экрана, который почти всегда полностью занимает площадь реального экрана устройства. Здесь выво-

---

[1] В некоторых книгах об ОС Android на русском языке объекты типа Intent обозначены термином «намерения». – *Прим. перев.*

дится информация, обрабатывается ввод пользователя и т. д. Здесь же располагаются элементы пользовательского интерфейса – кнопки, текстовые фрагменты, изображения и др., – отображаемые на экране и содержащие ссылки на объекты в иерархии, включающей все экземпляры класса `View`. Следовательно, объём памяти, потребляемой компонентом `Activity`, может увеличиваться весьма существенно.

Когда пользователь перемещается между экранами, экземпляры компонента `Activity` образуют стек. При переходе к новому экрану предыдущий экземпляр `Activity` помещается в стек, а возврат в обратном направлении вызывает извлечение очередного экземпляра из стека.

На рис. 1.2 пользователь начал работу с экземпляром A компонента `Activity`, завершив его, перешёл к экземпляру B, затем к C и D. Экземпляры A, B и C являются полноэкранными компонентами, а D занимает лишь часть дисплея устройства. В результате экземпляр A будет удалён, B полностью невидим, C отображается частично, а D показан полностью, поскольку находится на вершине стека. Следовательно, D получает фокус и принимает данные, вводимые пользователем. Положение в стеке определяет состояние каждого экземпляра `Activity`:

- активный, расположенный на переднем плане: D;
- приостановлен и отображается частично: C;
- остановлен и невидим: B;
- неактивен и удалён: A.

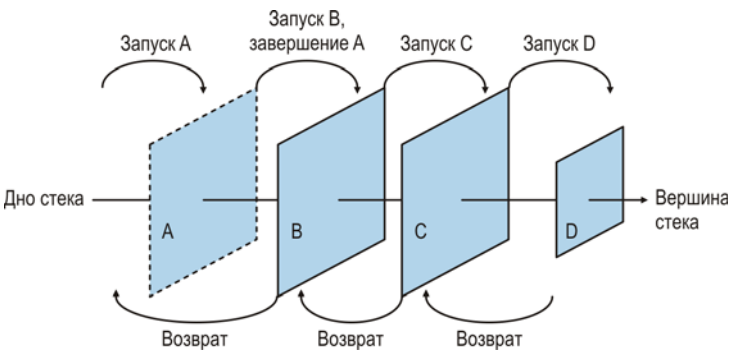


**Рис. 1.2** ❖ Стек экземпляров Activity

Состояние экземпляра `Activity`, находящегося на вершине стека, определяет системный приоритет соответствующего приложения (системный приоритет также обозначается термином *«ранг процесса»*

(*process rank*)), который, в свою очередь, влияет на возможность завершения приложения (см. раздел «Завершение приложения» ниже) и на планирование выполнения потоков приложения (глава 3).

Жизненный цикл экземпляра компонента `Activity` завершается, когда пользователь даёт команду возврата к предыдущему экрану, например нажимает кнопку «Назад» (Back), или когда `Activity` явно вызывает метод `finish()`.

### *Service*

Компонент `Service` может незаметно выполняться в фоновом режиме без прямого взаимодействия с пользователем. Обычно он используется для частичного снятия чрезмерной нагрузки с других компонентов, когда длительности операций превосходят длительности их существования. Компонент `Service` может работать в двух режимах: в режиме *запускаемой службы* (*started*) и в режиме *подключаемой службы* (*bound*):

- *в режиме запускаемой службы* – компонент `Service` запускается вызовом метода `Context.startService(Intent)`, с явным или неявным объектом `Intent`, и завершается вызовом метода `Context.stopService(Intent)`;
- *в режиме подключаемой службы* – несколько компонентов могут подключиться к компоненту `Service`, вызвав метод `Context.bindService(Intent, ServiceConnection, int)` и передав ему явный или неявный объект `Intent`. После подключения компонент может взаимодействовать с `Service` через интерфейс `ServiceConnection` и в любой момент разорвать установленную связь вызовом метода `Context.unbindService(ServiceConnection)`. Компонент `Service` автоматически удаляется, когда последний компонент разорвёт связь с ним.

### *ContentProvider*

Если возникает необходимость использовать большие объёмы данных внутри одного приложения или совместно с другими приложениями, можно воспользоваться компонентом `ContentProvider`. Он способен обеспечить доступ к любому источнику данных, но наиболее часто применяется в связке с базами данных SQLite, которые всегда принадлежат только одному приложению. С помощью компонента `ContentProvider` любое приложение может предоставлять свои данные другим приложениям, выполняющимся в удалённых процессах.

***BroadcastReceiver***

Этот компонент имеет узкую специализацию: он принимает сообщения, отправляемые другими компонентами внутри приложения, удалёнными приложениями или программной платформой. Входящие сообщения фильтруются, чтобы выделить предназначенные для данного экземпляра `BroadcastReceiver`. Компонент `BroadcastReceiver` должен зарегистрироваться, чтобы начать прием сообщений, и отменить регистрацию по завершении. Если этот компонент статически зарегистрирован в файле *AndroidManifest.xml*, он автоматически будет получать сообщения, пока приложение установлено в системе. То есть `BroadcastReceiver` может активировать связанное с ним приложение, если `Intent` обнаружит совпадение в отфильтрованных сообщениях.

## Выполнение приложения

Android – это многопользовательская, многозадачная система, способная одновременно выполнять несколько приложений и позволяющая пользователю переключаться между ними без сколько-нибудь существенных задержек. Ядро Linux обеспечивает многозадачность, а выполнение приложений основано на концепции процессов Linux.

### Процесс Linux

В Linux каждому пользователю присваивается уникальный идентификатор (user ID) – номер, с помощью которого операционная система отличает пользователей друг от друга. Каждому пользователю разрешён доступ только к собственным ресурсам, и ни один из пользователей (за исключением суперпользователя *root*, который нас не интересует в контексте данной книги) не сможет получить доступа к личным ресурсам другого пользователя. Для изоляции пользователей друг от друга создаются так называемые «песочницы» (sandboxes). В ОС Android пакет каждого приложения имеет не повторяющийся в системе идентификатор пользователя, то есть приложение в Android соответствует отдельному пользователю в Linux и не имеет доступа к ресурсам других приложений.

Android добавляет в каждый процесс среду выполнения, например виртуальную машину Dalvik, для каждого экземпляра любого приложения. На рис. 1.3 показаны отношения между моделью процесса Linux, виртуальной машиной и конкретным экземпляром приложения.

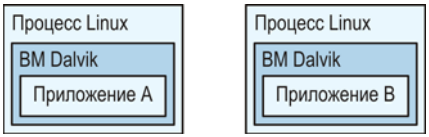


**Рис. 1.3** ❖ Приложения выполняются в разных процессах и в разных виртуальных машинах

По умолчанию между приложением и процессом устанавливается отношение «один к одному», но при необходимости приложение можно запустить в нескольких процессах или организовать выполнение нескольких приложений в одном процессе.

## Жизненный цикл

Жизненный цикл приложения ограничен рамками соответствующего процесса Linux, которому в Java соответствует класс `android.app.Application`. Объект `Application` запускается для каждого приложения, когда среда выполнения вызывает его метод `onCreate()`. Теоретически среда выполнения должна завершать приложение вызовом его же метода `onTerminate()`, но полагаться на это не следует. Процесс Linux может быть экстренно завершён раньше, чем соответствующий экземпляр среды выполнения получит возможность вызвать метод `onTerminate()`. Объект `Application` должен быть самым первым экземпляром компонента, создаваемым в процессе, а удаляться он должен самым последним.

### *Запуск приложения*

Приложение запускается, когда один из его компонентов инициализируется для выполнения. Любой компонент может быть точкой входа в приложение, и как только первый компонент активируется, создаётся процесс Linux – если он уже не находился в рабочем состоянии, – и выполняется следующая процедура запуска:

1. Запускается процесс Linux.
2. Создается среда выполнения.
3. Создается экземпляр класса `Application`.
4. Создается компонент точки входа в приложение.

Настройки нового процесса Linux и среды выполнения являются достаточно длительными операциями. Они могут ухудшить производительность и принести немало отрицательных эмоций пользовате-

лю. Поэтому система пытается сократить время запуска приложений Android, активируя особый процесс с именем *Zygote* во время загрузки самой системы. Zygote содержит полный набор заранее загруженных системных библиотек ядра. Новые процессы приложений создаются (fork) как клоны процесса Zygote, без копирования системных библиотек, которые совместно используются всеми приложениями.

### *Завершение приложения*

Процесс создаётся при запуске приложения и завершается, когда системе потребуется освободить ресурсы. Поскольку пользователь может снова обратиться к приложению позднее, среда выполнения старается избегать полного уничтожения всех его ресурсов, пока активные приложения, которых может быть достаточно много, не исчерпают доступных ресурсов системы. То есть приложение не завершается автоматически, даже если все его компоненты были уничтожены.

Когда система испытывает недостаток ресурсов, она обращается к среде выполнения, чтобы определить, какой процесс следует завершить. Для принятия такого решения в системе существует ранжирование процессов по степени важности, в зависимости от видимости приложения на экране и его компонентов, выполняемых в текущий момент. В соответствии с приведённой ниже классификацией процессы с более низким рангом завершаются раньше, чем процессы с более высоким рангом. Процессы могут иметь следующие ранги, перечисленные в порядке убывания:

- *Foreground* (*Активный, работающий в основном режиме*) – приложение представлено видимым компонентом на переднем плане экрана, или компонент `Service` связан с компонентом `Activity` в удалённом активном процессе, или выполняется компонент `BroadcastReceiver`.
- *Visible* (*Видимый*) – приложение представлено частично видимым компонентом (то есть часть этого компонента скрыта).
- *Service* (*Служба*) – компонент `Service` выполняется в фоновом режиме и не связан с каким-либо видимым компонентом.
- *Background* (*Фоновый режим*) – невидимый компонент `Activity`. Этот ранг получают процессы большинства приложений.
- *Empty* (*Пустой*) – процесс без активных компонентов. Пустые процессы сохраняются для ускорения запуска, но они являются первыми кандидатами на удаление, когда системе требуются ресурсы.

Подобная система ранжирования действительно гарантирует, что ни одно видимое (активное) приложение не будет принудительно завершено программной платформой в случае исчерпания системных ресурсов.

**Жизненные циклы двух взаимодействующих приложений**

Данный пример иллюстрирует жизненные циклы двух процессов, P1 и P2, взаимодействующих обычным образом (рис. 1.4). P1 – приложение-клиент, вызывающее компонент `Service` в приложении-сервере P2. Процесс клиента P1 начинает работу объектом `Intent` при получении сообщения. На запуске данный процесс активирует экземпляры компонентов `BroadcastReceiver` и `Application`. Через некоторое время запускается компонент `Activity`, и в течение всего периода существования этого компонента процесс P1 имеет наивысший ранг: Foreground.

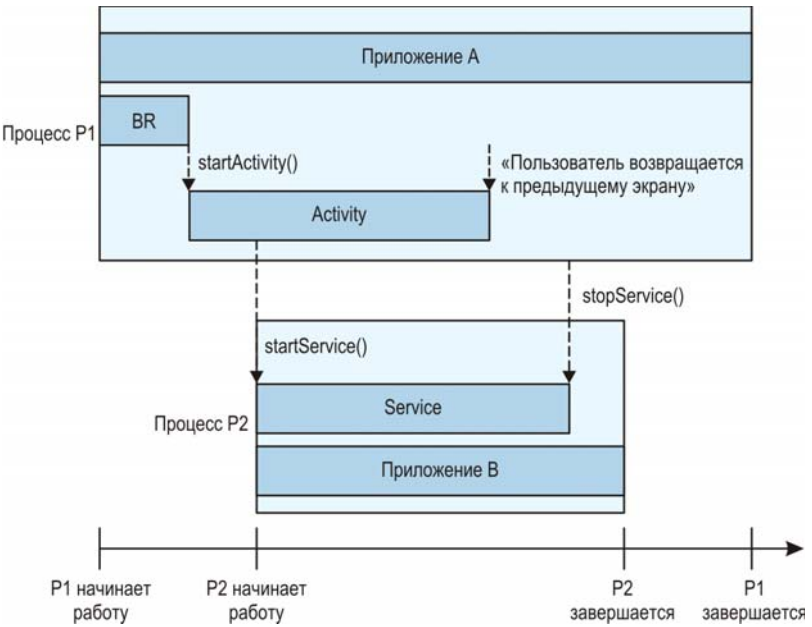


**Рис. 1.4** ❖ Приложение-клиент запускает компонент `Service` в другом процессе

Компонент `Activity` перекладывает работу на компонент `Service` в процессе P2, который активировал экземпляр этого компонента и соответствующий экземпляр компонента `Application`. То есть приложение-клиент разделило работу между двумя разными процессами. Компонент `Activity`

в процессе P1 может завершиться, а компонент `Service` в процессе P2 будет продолжать выполнение.
Если все компоненты приложения-клиента завершат работу, например когда пользователь покинет `Activity` в процессе P1, а компонент `Service` в процессе P2 получит запрос от какого-нибудь другого процесса или будет остановлен средой выполнения, оба процесса получат ранг Empty (пустой) и станут наиболее вероятными кандидатами на удаление системой при нехватке ресурсов.
В табл. 1.1 приводится порядок изменения ранга процессов во время выполнения.

***Таблица 1.1. Изменение ранга процессов***

| Состояние приложения | Ранг процесса P1 | Ранг процесса P2 |
|---|---|---|
| Запуск процесса P1 с точки входа `BroadcastReceiver` | Foreground | – |
| Процесс P1 запускает компонент `Activity` | Foreground | – |
| Процесс P1 активирует компонент `Service` – точку входа в процесс P2 | Foreground | Foreground |
| В процессе P1 уничтожается компонент `Activity` | Empty | Service |
| В процессе P2 останавливается работа компонента `Service` | Empty | Empty |

Следует отметить, что фактический и предполагаемый (кажущийся) жизненные циклы приложения, определяемого процессом Linux, отличаются. В системе может существовать множество функционирующих процессов приложений, даже после завершения их пользователем. Если системные ресурсы позволяют, такие пустые процессы продолжают существовать для сокращения времени запуска при повторных обращениях к соответствующим приложениям.

## Структурирование приложений для улучшения производительности

Android-устройства представляют собой многопроцессорные системы, способные одновременно выполнять несколько операций, но каждое приложение само должно позаботиться о разделении на операции и их параллельном выполнении для оптимизации производительности. Если приложение не предусматривает деления на операции и выполняется как одна длинная операция, такое приложение сможет использовать только один процессор (CPU), но в этом случае