View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by eEast-UkrNUIR (Electronic Volodymyr Dahl East Ukrainian National University Institutional...

УДК 681.3.06

АНАЛИЗ ЭФФЕКТИВНОСТИ МНОГОПОТОЧНОЙ ОБРАБОТКИ В ANDROID-ПРИЛОЖЕНИЯХ

Щербакова М. Е., Щербаков Е. В.

ANALYSIS OF MULTI-THREADED PROCESSING EFFICIENCY IN ANDROID-APPLICATIONS

Shcherbakova M. E., Shcherbakov E. V.

Рассмотрены особенности использования модели параллельных вычислений на основе потоков при разработке приложений для платформы Android. Кратко проанализированы составляющие эту платформу программные средства многопоточного распараллеливания в операционной системе Linux, используемые при этом средства языка программирования Java, а также объектно-ориентированные возможности разработки многопоточных приложений для операционной системы Android. Исследована эффективность рассмотренной модели распараллеливания вычислений при разработке Android-приложений, требующих длительных вычислений и интенсивного отображения результатов вычислений на экране мобильного устройства, функционирующего под управлением ОС Android.

Ключевые слова: *Android, Linux, Java, XML, процесс, поток, многопоточность, пользовательский интерфейс, виртуализация, фрактал.*

1. Введение. Фундамент, на котором построена ОС Android[1], - это аккуратно закодированная и хорошо протестированная многопользовательская ОС Linux [2]. Каждое приложение работает как отдельный пользователь в своем собственном процессе Linux. Linux и ее основные сервисы управляют техническими средствами смартфонов, планшетов, читалок электронных книг, умных часов, интерактивного TV (iTV), а также предоставляют Android-приложениям полный доступ к функциям каждого устройства, в том числе к процессорам, памяти, сенсорному экрану, хранилищам данных и многому другому.

Одной из причин быстрого завоевания рынка системой Android является также тот факт, что Android использует язык Java [3] в качестве основного языка программирования. Платформа разработки приложений Android является многослойным набором программных компонентов, включающих в себя тысячи библиотек функций на Java, которые работают сверху ядра Linux. Java-библиотеки упрощают задачу общения приложений с ядром Linux, которое является весьма сложным по своей сути.

В частности, одним из таких сложных вопросов программирования является параллельная обработка информации с помощью нескольких потоков. Потоки являются краеугольным камнем любой многозадачной операционной системы и могут рассматриваться как мини-процессы, запущенные в рамках основного процесса с целью параллельного выполнения нескольких программных ветвей в пределах приложения. Большинство нетривиальных приложений Android используют несколько потоков, поэтому многопоточное программирования имеет важное значение для разработчиков на базе платформы Android.

Целью работы является сравнительный анализ средств многопоточного распараллеливания вычислений, предоставляемых платформой Android, а также исследование эффективности этих средств при разработке Android-приложений, требующих длительных вычислений и отображения большого количества данных.

2. Многопоточность в ОС Linux. Многопоточность – это создание и управление несколькими единицами выполнения внутри одного процесса. Процесс – это набор компьютерных ресурсов, выделяемых операционной системой для выполнения откомпилированных программных файлов приложения, таких как код исполнительных файлов, оперативная память, объекты ядра ОС Linux и др. Потоки представляют собой последовательности выполнения кода в рамках процесса, каждой из которых назначаются виртуальный процессор, стековая память и текущее программное состояние. Другими словами, процессы выполняют двоичные файлы, а потоки - это наименьшая единица для планирования выполнения кода программ процессорами вычислительного устройства.

Каждый процесс содержит один или несколько потоков. Если процесс содержит один поток, то есть только одну последовательность выполнения программного кода, то в процессе кроме этой последовательности выполнения ничего больше не может выполняться до самого завершения всего процесса. Такие процессы называются однопоточными и представляют собой классические процессы ОС Unix. Если процесс содержит более чем один поток, то более чем одна последовательность кода может выполняться одновременно. Такие процессы называются многопоточными, и в ОС Android, базирующейся на ОС Linux, подавляющее большинство приложений запрограммированы для выполнения с использованием многопоточных процессов.

Практически все современные операционные системы предусматривают на уровне пользователей две основные виртуализированные абстракции: виртуальная память и виртуализированный процессор. Вместе они дают иллюзию для каждого запущенного процесса, что только он потребляет ресурсы компьютерного устройства. Виртуальная память предоставляет каждому процессу линейное адресное пространство памяти, которое плавно отображается в физические оперативную и внешнюю память за счет подкачки. Хотя оперативная память в действительности может в одно и то же время содержать данные до 100 различных выполняющихся процессов, но для каждого процесса создается иллюзия, что вся виртуальная память находится в его распоряжении. Виртуализированный процессор дает возможность процессу действовать так, как будто он один выполняется операционной системой, скрывающей от него тот факт, что она одновременно выполняет несколько процессов в режиме многозадачности на (возможно) нескольких процессорах ядрах.

Виртуальная память ассоциируется с процессом, а не потоком, т. е., каждый процесс имеет свое пространство виртуальной памяти, а все потоки данного процесса пользуются этой памятью сообща. Наоборот, виртуализированный процессор ассоциируется с потоками, а не с процессами. Каждый поток представляет собой независимую единицу для выполнения операционной системой, что дает возможность одному процессу выполнять более чем одну программную нить приложения в одно и то же время.

Основное преимущество многопоточных процессов таково: когда процесс должен выполнить много работы, потоки могут быть запущены одновременно на нескольких процессорах, что потенциально ускоряет вычисления. Хотя одновременные вычисления можно организовать и с помощью нескольких процессов, потоки запускаются быстрее процессов и потокам часто бывает проще и/или эффективнее взаимодействовать между собой при совместном использовании ресурсов по сравнению с процессами, которые взаимодействуют через сетевое соединение или канал.

Основным недостатком является то, что многопоточность является существенным источником ошибок программирования из-за возникающих при этом гонок и взаимоблокировок параллельно выполняющихся потоков.

3. Программирование потоков в Java. Java обеспечивает надежную встроенную поддержку для разработки многопоточных приложений, а создание новых потоков является относительно простым. Каждый поток представляется объектом класса, производным от базового класса java.lang.Thread, и, чтобы создать новый поток, надо просто определить класс, расширяющий класс Thread, или же реализующий интерфейс java.lang.Runnable.

При разработке многопоточных приложений часто хочется создать класс с некоторой наследуемой функциональностью, код которого должен работать в отдельном потоке. Но этого нельзя сделать, так как этот новый класс должен наследовать и базовый класс Thread, а язык Java не допускает множественного наследования. Расширение класса Thread не дает никаких функциональных либо программных преимуществ по сравнению с реализацией новым классом интерфейса Runnable, так что второй вариант (реализация интерфейса Runnable) обычно оказывается предпочтительней.

Единственным методом, определяемым в интерфейсе Runnable, является метод run(), который вызывается для выполнения потока. Как только поток выходит из метода run() (по завершению работы или вследствие необработанного исключения), он считается недействительным и не может быть перезапущен или повторно использован. Фактически метод run() выполняет ту же роль для потока, что и метод main() для Java-приложения: он является начальной точкой входа в код, выполняемый потоком. Как и в случае с методом main(), как правило, программа не должна вызывать метод run() напрямую. Вместо этого реализация интерфейса Runnable передается конструктору класса Thread, а поток вызовет метод run() автоматически в момент своего запуска. Например, чтобы выполнить какую-то подзадачу приложения в отдельном потоке, другом, чем основной поток приложения, нужно создать отдельный класс (скажем, Worker), реализующий интерфейс Runnable:

```
class Worker implements Runnable {
        public void run() {
                // код, выполняющий подзадачу;
        }
}
```

Для того, чтобы использовать этот класс, достаточно создать новый объект класса Thread, передав его конструктору в качестве аргумента объект класса Worker, и, чтобы начать выполнение, вызвать метод start() класса Thread. Вызов start() задает, что вновь созданный поток должен начать выполнять код, вызвав метод run(), как было отмечено ранее:

```
Thread t = new Thread(new Worker());
t.start();
```

4. Программирование многопоточных приложений в ОС Android. Когда один из компонентов Android-приложения, такой как активность, запускается на выполнение и приложение не имеет каких-либо других работающих в данное время компонентов этого приложения, операционная система Android создает новый Linux-процесс только с одним потоком выполнения, называемом основным потоком приложения или UI-потоком [4]. Главное назначение основного потока – это управление пользовательским интерфейсом (UI), заключающееся в обработке системных событий и событий, вызванных взаимодействиями пользователя с элементами UI, а также в динамическом изменении представления элементов UI в ответ на эти события. Любые дополнительные компоненты, которые запускаются в рамках приложения, также, по умолчанию, будут работать в рамках главного потока выполнения.

Любой компонент внутри Android-приложения, выполняющий трудоемкую по времени задачу в основном потоке, вызывает блокировку всего приложения до завершения этой задачи. Как правило, это приводит к отображению операционной системой предупреждения для пользователя «Приложение не отвечает». Очевидно, это далеко не желаемое поведение для любого приложения. Этого можно избежать, запуская трудоемкие задачи в отдельных рабочих потоках выполнения, давая возможность основному потоку беспрепятственно выполнять все остальные задачи приложения.

Из вышесказанного вытекает одно из главных правил разработки для Android - никогда не выполнять трудоемкие операции в основном потоке приложения. Второе, не менее важное, правило заключается в том, что код в отдельном рабочем потоке никогда, ни при каких обстоятельствах не должен непосредственно обновлять любой элемент пользовательского интерфейса. Любые изменения в пользовательском интерфейсе всегда должны выполнятся из основного потока. Причиной последнего является то, что набор программных средств для построения пользовательского интерфейса Android-приложений не является потокобезопасным. Попытки выполнять потоконебезопасный код из нескольких потоков приводят, как правило, к непонятным ошибкам и непредсказуемому поведению приложения.

В том случае, если коду, выполняющемуся в рабочем потоке, необходимо взаимодействовать с пользовательским интерфейсом, он должен делать это, синхронизуя свою работу с основным потоком. Это достигается путем создания обработчиков в основном потоке, которые, в свою очередь, получают сообщения из других потоков и соответствующим образом обновляют пользовательский интерфейс.

Для создания новых потоков можно использовать упомянутый выше класс Thread из базовой библиотеки Java. Но использование этого класса в Android-приложениях имеет ограничения – нельзя из вторичного потока изменять пользовательский интерфейс. Для решения этой проблемы в Android SDK имеется абстрактный параметризуемый класс AsyncTask:

```
abstract class AsyncTask<Params, Progress,
        Result> { }
```

Чтобы использовать этот абстрактный класс, необходимо сделать следующее:

- Создать класс, производный от AsyncTask, и реализовать его метод doInBackground(), который выполняется в пуле рабочих потоков. Для этого, как правило, создается внутренний класс в активности или фрагменте приложения:

```
public class MyTask extends AsyncTask<String,
        Float, Boolean> { }
```

- Для изменения пользовательского интерфейса необходимо реализовать метод onPostExecute() класса AsyncTask, который получает результаты, возвращенные методом doInBackground(), и выполняется в основном потоке, так что в нем можно надежно обновлять пользовательский интерфейс.
- После этого, чтобы выполнить в рабочем потоке задачу, реализованную в методе doInBackground(), нужно из основного потока вызвать метод execute() объекта класса, производного от AsyncTask.

Кроме перечисленных выше, класс AsyncTask содержит еще несколько методов для взаимодействия основного и рабочего потоков, которые можно переопределять по мере необходимости:

- Метод onPreExecute(), который вызывается из основного потока перед запуском метода doInBackground().
- Метод onProgressUpdate(), выполняющийся в основном потоке и обрабатывающий данные о выполнении рабочего потока, передаваемые ему методом publishProgress().
- Метод publishProgress(), вызываемый из выполняющегося в рабочем потоке метода doInBackground() для передачи данных, вычисленных или полученных в рабочем потоке, методу основного потока onProgressUpdate().

Используя класс AsyncTask, основной поток может запустить несколько одновременно выполняющихся рабочих потоков, переопределяя описанные выше методы в производных от AsyncTask классах. Но такое выполнение порождает ряд проблем, связанных с конкурентным использованием потоками общих ресурсов, а также с отсутствием гарантий, что из-за конкуренции потоки будут завершаться в том же порядке, в котором они были запущены.

Как следствие, начиная API level 11 в реализацию класса AsyncTask были внесены

изменения, в результате которых все задачи, запускаемые методом execute(), по умолчанию выполняются последовательно одним рабочим потоком, а для выполнения задач параллельными потоками был добавлен новый метод:

```
public final AsyncTask<Params, Progress, Result>
executeOnExecutor(Executor exec, Params…
        params)
```

Реализации используемого в этом методе интерфейса Executor из пакета java.util.concurrent могут выполнять задачи последовательно, используя единственный рабочий поток, параллельно с использованием ограниченного по числу пулом потоков или же непосредственно создавая новый поток для каждой задачи.

Таким образом, оба представленные ниже вызовы методов:

```
task.execute(params);
task.executeOnExecutor(AsyncTask
        .SERIAL_EXECUTOR, params);
```

делают одно и то же, последовательно выполняя все запускаемые задачи на одном рабочем потоке.

Параллельное же выполнение задач пулом потоков обеспечивает вызов метода:

```
task.executeOnExecutor(AsyncTask
        .THREAD_POOL_EXECUTOR, params);
```

Пул потоков в классе AsyncTask конфигурируется таким образом, чтобы в нем было по крайней мере 5 параллельно работающих потоков с дальнейшим расширением по мере необходимости до 128, что вполне достаточно для подавляющего большинства приложений, реализуемых на платформе Android.

5. Разработка и анализ многопоточного Android-приложения. Как уже говорилось ранее, наиболее распространенное применение многопоточной обработки – это решение требующих длительной обработки задач без торможения визуального интерфейса пользователя, для чего обычно достаточно одного – двух параллельно работающих потоков.

Но в более сложных вычислительных приложениях для эффективного решения их задач может понадобиться несколько десятков параллельно работающих рабочих потоков. Естественно, при таком распараллеливании серьезного ускорения обработки и отображения данных можно достигнуть при решении задачи на компьютере с многоядерным процессором. На рис. 1 представлено изображение множества Мандельброта [5] - фрактала, определённого как множество точек на комплексной плоскости, для которых не уходит в бесконечность итеративная последовательность:

Z0 = 0

Zn+1 = Zn2 + C

Если получать приведенное на рисунке изображение последовательно с помощью одного потока, то на его получение и отображение на экране устройства понадобиться порядка 10 секунд. Так как итерации для получения изображения ведутся построчно, вычисления можно распределить между группой параллельно работающих потоков, итерирующих соседние строки изображения. В результате время получения множества в заданных границах и его отображения на экране может уменьшиться в несколько раз, если Android-устройство построено на базе кристалла с несколькими процессорными ядрами.

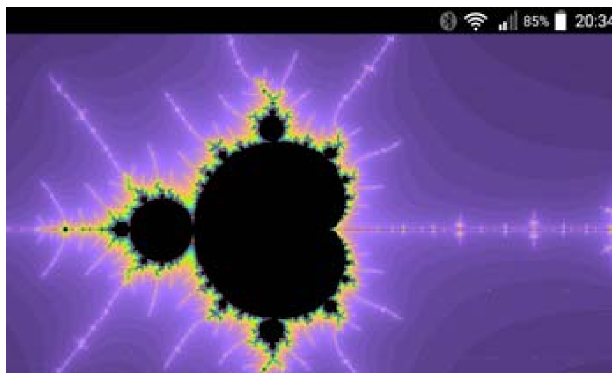

Рис. 1. Представление на экране Android-устройства изображения множества Мандельброта

В реализованном авторами Android-приложении предусмотрено автоматическое масштабирование изображения в зависимости от размеров окна браузера. Кроме того, после касания какой-то точки изображения устанавливаются новые границы множества Мандельброта с установленным коэффициентом масштабирования, равным 8, и с помощью той же группы параллельно работающих потоков вычисляется новый фрактал, поддерживающий соотношение сторон текущего размера окна. На рис. 2 представлен один из таких фракталов.

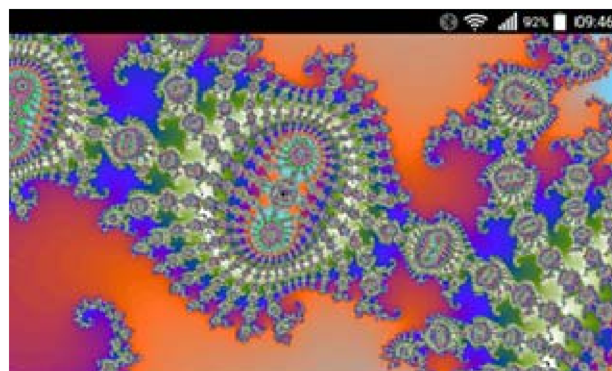

Рис. 2. Изображение фрактала, полученного масштабированием одного из прямоугольных участков на границе множества Мандельброта

Модифицируя данное Android-приложение, был проведен ряд экспериментов по распараллеливанию вычислений на отдельные потоки, используя бюджетный смартфон LG D690 G3 Stylus на базе системы-на-чипе MediaTek MT6582. В ее состав входят четыре процессорных ядра Cortex-A7, работающих на

частоте 1,3 ГГц, и видеоядро Mali-400MP. В смартфоне применен 5,5-дюймовый IPS дисплей с разрешением 960 на 540 точек, объем оперативной памяти смартфона равен 1 ГБ. Эксперименты проводились с использованием операционной системы Android 5.0.1 в штатном режиме работы смартфона без отключения его основных сервисов, поэтому все приведенные ниже данные по времени выполнения приложения являются усредненными.

В табл. приведены времена получения изображений фракталов в зависимости от количества одновременно используемых рабочих потоков при параллельном вычислении строк изображения.

Таблица

Времена получения фракталов в зависимости от количества используемых рабочих потоков

Количество потоков	Время (мс) получения фрактала	Количество потоков	Время (мс) получения фрактала
1	14137	7	2880
2	7630	8	2705
3	5452	9	2614
4	4138	10	2562
5	3442	11	2498
6	3108	12	2447

Для большей наглядности на рис. 3 данные табл. 1 представлены в виде гистограммы.

Рис. 3. Гистограмма времен получения фракталов в зависимости от количества используемых рабочих потоков

В приложении, помимо отмеченных в таблице и на гистограмме рабочих потоков, всегда использовался основной поток, последовательно отображающий строки, вычисленные одним либо несколькими рабочими потоками, с перерисовкой экрана после каждой добавленной строки.

Как можно увидеть из приведенных данных, время получения фрактала существенно уменьшается при подключении к вычислениям, кроме основного и 1-го рабочего потока, 2-го (на 6507 мс), 3-го (на 2177 мс) и 4-го (на 1313 мс) рабочих потоков, что объясняется наличием 4-х процессорных ядер в используемом для экспериментов устройстве LG D690 G3. Уменьшение времени выполнения приложения, достигаемое подключением к вычислениям 5-го и последующих потоков, происходит из-за того, что уменьшается нагрузка на основной поток из-за сокращения в нем количества циклов на перепланировку потоков, а также потому, что сокращается суммарное время простоев процессорных ядер за счет большего количества готовых к выполнению потоков в каждый момент их назначения на выполнение процессорными ядрами, производимого диспетчером потоков ОС Linux.

6. Выводы. Модель параллельных вычислений на основе потоков, используемая в приложениях для платформы Android, базируется на свойствах и методах параметризуемого класса AsyncTask. В этой модели все длительные по времени вычисления выполняются одним либо несколькими рабочими потоками, что дает возможность выполнять обработку параллельно несколькими процессорными ядрами. Вся работа по синхронизации рабочих потоков, исполнению потоконебезопасных участков приложения, а также визуализация результатов обработки на экране устройства выполняются основным потоком, с которого начинается работа при запуске приложения.

Для проверки эффективности этой модели распараллеливания обработки было выбрано представительное Android-приложение, требующее длительных вычислений рабочими потоками и порождающее интенсивный поток обмена данными с основным потоком для отображения результатов вычислений. Проведенные измерения времени работы приложения при различном количестве используемых параллельно работающих потоков показали высокую эффективность модели многопоточных вычислений Android. На четырехъядерном процессоре Cortex-A7 ускорение работы приложения составило: при 2-х рабочих потоках – в 1.85 раза, при 3-х рабочих потоках – в 2.59 раза, при 4-х рабочих потоках – в 3.42 раза по сравнению с временем выполнения приложения с использованием только одного рабочего потока.

Л и т е р а т у р а

1. Dave MacLean Pro Android 5 / Dave MacLean, Satya Komatineni, Grant Allen. - Springer Science+Business Media, New York, 2015. – 813 p.
2. Robert Love. Linux System Programming, Second Edition / Robert Love. - O'Reilly Media, Sebastopol, 2013. – 456 p.
3. Brett Spell. Pro Java 8 Programming / Brett Spell. - Springer Science+Business Media, New York, 2015. – 695 p.
4. Anders Göransson. Efficient Android Threading / Anders Göransson. - O'Reilly Media, Inc., Gravenstein Highway North, Sebastopol, 2014. – 279 p.

5. Мандельброт Б. Фрактальная геометрия природы / Мандельброт Б.; [пер. с англ]. - Москва: Институт компьютерных исследований, 2002. – 656 с.

References

1. Dave MacLean Pro Android 5 / Dave MacLean, Satya Komatineni, Grant Allen. - Springer Science+Business Media, New York, 2015. – 813 p.
2. Robert Love. Linux System Programming, Second Edition / Robert Love. - O'Reilly Media, Sebastopol, 2013. – 456 p.
3. Brett Spell. Pro Java 8 Programming / Brett Spell. - Springer Science+Business Media, New York, 2015. – 695 p.
4. Anders Göransson. Efficient Android Threading / Anders Göransson. - O'Reilly Media, Inc., Gravenstein Highway North, Sebastopol, 2014. – 279 p.
5. Mandelbrot B. Fractalnaja geometrija pryrody / Mandelbrot B. - Moskva: Institut komputernych issledovanij, 2002. – 656 p.

Щербакова М. Є., Щербаков Є. В. Аналіз ефективності багатопотокової обробки в Android-додатках.

Розглянуті особливості використання моделі паралельних обчислень на основі потоків при розробці додатків для платформи Android. Коротко проаналізовані складові цієї платформи, такі як програмні засоби багато-потокового розпаралелювання в операційній системі Linux, використовувані при цьому засоби мови програмування Java, а також об'єктно-орієнтовані можливості розробки багато-потокових додатків для операційної системи Android. Досліджена ефективність розглянутої моделі розпаралелювання обчислень при розробці Android-додатків, які потребують тривалих обчислень і інтенсивного відображення результатів обчислень на екрані мобільного пристрою, який функціонує під управлінням ОС Android.

Ключові слова: *Android, Linux, Java, XML, процес, потік, багато-потоковість, інтерфейс користувача, віртуалізація, фрактал.*

Shcherbakova M. E., Shcherbakov E. V. Analysis of multi-threaded processing efficiency in Android-applications.

Considered the features of the use of parallel computing model based on threads in developing applications for the Android platform. Briefly analyze the components of the platform software multithreaded parallelization on Linux operating system used with the Java programming language tools and object-oriented features of the development of multi-threaded applications for the Android OS. Were analyzed the efficiency of this model of parallel computing in the development of Android-applications that require long calculation and intensive display computing results on the screen of the mobile device that operates on Android OS.

Keywords: *Android, Linux, Java, XML, process, thread, multithreading, user interface, virtualization, fractal.*

Щербакова Марина Евгеньевна – к.т.н., доцент, доцент кафедры компьютерной инженерии, Восточноукраинский национальный университет им. В. Даля, ms432@mail.ru

Щербаков Евгений Васильевич - к.т.н., доцент, доцент кафедры компьютерной инженерии, Восточноукраинский национальный университет им. В. Даля, gkvarc@gmail.com

Рецензент: д.т.н., професор ***Суворін О.В.***

Стаття подана 05.09.2016