Учреждение Российской Академии наук
Санкт-Петербургский академический универитет –
Научно-образовательный центр нанотехнологий РАН

На правах рукописи

Диссертация допущена к
защите
Зав. кафедрой

_____

« » _____ 2012 г.

Диссертация
на соискание ученой степени
магистра

Тема «Разработка технологии виртуализации периферийных устройств на смартфонах с ОС Android»

Направление: 010600.68 — Прикладные математика и физика

Магистерская программа: «Математические и информационные технологии»

Выполнил студент Баталов Е. А.

Руководитель
к.т.н., доцент Кринкин К. В.

Рецензент
д.ф.-м.н, профессор Тормасов А. Г.

Санкт-Петербург
2012

Содержание

1 Введение

1.1 Виртуализация

Можно дать несколько определений виртуализации:

- *Виртуализация* — создание объекта, подобного другому объекту. Причем второго не существует
- *Виртуализация* — предоставление чему-либо ожидаемого вычислительного контекста, в то время как такого контекста в действительности не существует или он сильно отличается от ожидаемого

Широко распространенным примером виртуального контекста в вычислительных системах является понятие процесса. Рассмотрим основные свойства процесса:

- Процесс не видит изменения состояния процессора, вызванные параллельно работающими процессами.
- Процессу доступны 4Гб памяти на машинах с 32-х разрядной шиной адреса.
- Процесс не имеет доступа к памяти другого процесса.

Перечисленные свойства процесса не реализуются распространенными типами процессоров и памяти.

Обычно управление процессами берет на себя операционная система (ОС). Процесс не существует, он является программной моделью, реализация которой во многом не ограничена. Например, вместо того чтобы делить процессор одной машины между всеми процессами, можно было бы запускать каждый процесс на отдельной машине и заниматься лишь выбором того на какой машине запустить следующий процесс. Пример с процессами показывает, насколько глубоко вошла виртуализация в нашу жизнь. Обычно понятие виртуализации связывают с большими решениями, реализующими вычислительные машины, ядра или пользовательские окружения ОС, абстрактные машины для исполнения байт-кодов и т. п.

Но есть и множество мелких примеров виртуализации. Например, в ОС нередко требуется разделение одного устройства между несколькими потребителями. Программы-пользователи считают, что владеют устройством монопольно. Каждый пользователь получает для себя виртуальное устройство, которое ведет себя как настоящее. На самом деле, ни одно из виртуальных устройств не существует.

Среди всех типов виртуализации рассмотрим *виртуализацию платформы*. Она включает в себя виртуализацию ОС и виртуализацию физической машины. Можно выделить несколько типов такой виртуализации[1]:

- Полная виртуализация — виртуализация всего аппаратного обеспечения, позволяющая запускать немодифицированную гостевую ОС.

[1] `http://en.wikipedia.org/wiki/Virtualization#Hardware`

- Частичная виртуализация — часть целевого окружения виртуализирована. Многие гостевые программы, кроме ОС, могут работать без модификации.

- Паравиртуализация — интерфейс целевого окружения модифицируется и виртуализируется. Гостевая ОС модифицируется для работы с модифицированным окружением.

- Виртуализация уровня операционной системы — виртуализация пользовательского окружения ОС. Позволяет запускать несколько изолированных пользовательских окружений на одном ядре ОС.

1.2 Применение виртуализации на мобильных устройствах

1.2.1 Bring your own device

Широкое распространение мобильных устройств постепенно меняет стратегию компаний по выделению сотрудникам ПК для работы в офисе. Практически каждый сотрудник имеет в личном использовании мобильное устройство, позволяющее выполнять те же задачи, что и ПК на его рабочем месте. Наиболее распространенные сценарии использования такого ПК: работа в корпоративной почте, корпоративном Web-портале, календаре.

Компания может сократить расходы на покупку техники переведя, корпоративные приложения на мобильные устройства сотрудников. Таким образом, если раньше корпоративные приложения исполнялись на ПК, который обслуживается IT-специалистами компании и находится в ее офисе, то теперь корпоративные приложения исполняются на устройстве пользователя, который, как правило, не согласится, чтобы его устройство контролировали и ограничивали в функциях, как это делается с ПК в офисах компании. Отсутствие контроля мобильного устройства сотрудника, на котором установлено корпоративное ПО, в свою очередь, означает высокие риски утечки данных компании как по вине пользователя устройства, так и по вине злоумышленников. Таким образом, возникает конфликт интересов компании и сотрудника.

Хорошим решением проблемы безопасности могла бы стать технология виртуализации, позволяющая запускать несколько пользовательских окружений на одном устройстве. Значительную работу в этом направлении проделали компании `VMware` [2] и `CellRox`[2].

1.2.2 Создание песочницы

Известно множество случаев, когда приложения из официальных магазинов приложений проявляли вирусную активность: собирали личные данные, отправляли платные SMS, получали права администратора, скачивали другие программы и выполняли их.

Виртуализация ОС могла бы решить проблему мобильных вирусов: пользователь мог бы использовать изолированное окружение для запуска всех непроверенных приложений. В этом окружении отсутствовали бы личные данные, т. к. пользователь не использует

[2] `http://cellrox.com`

его для социальной активности. Порча этого окружения не влияла бы на другие, так как все они изолированы.

1.2.3 Смена прошивок

Виртуализация позволила бы легко и быстро менять прошивки устройств без риска потери данных и поломки телефона. Для установки прошивок, меняющих ядро ОС, понадобилось бы решение по полной виртуализации устройства. Если прошивка меняет только пространство пользователя, то достаточно решения по виртуализации уровня ОС.

1.2.4 Защита важных компонентов системы от ошибок в других компонентах

Последние несколько лет мобильные компьютеры совмещают в себе функции устройства управления какой-либо системой в реальном времени (RT) и другие не-RT-функции. Надежное исполнение RT-функций, как правило, очень важно для системы, в то время как не-RT-функциональность не является критичной.

Виртуализация могла бы защитить RT-функциональность от не-RT. Примером описанной системы может служить бортовой компьютер современного автомобиля, на котором запущено сразу две ОС: небольшая ОС реального времни и большая ОС, не являющаяся таковой. RTOS защищается гипервизором от не-RT. Этот вариант применения виртуализации хорошо описан в [5].

1.2.5 Виртуализированное окружение для тестирования приложений

Виртуализированная платформа может предоставлять дополнительные возможности для тестирования приложений, выполняющихся в ней. Например, устройства ввода могут генерировать события ввода по сценарию, что не умеют делать невиртуализированные устройства. Подробно этот вариант использования виртуализации описан в [7].

1.3 Проблемы виртуализации на мобильных устройствах

Существует большой набор решений виртуализации для персональных компьютеров и серверов. Преобладающей архитектурой центральных процессоров на таких устройствах являются `IA 32` и `Intel 64`.[3]

На мобильных устройствах сейчас преобладает архитектура `ARM` [4].

Большинство решений виртуализации зависит от архитектуры центрального процессора. Портирование таких решений может представлять из себя как простую, так и очень сложную задачу, требующую, по сути, создать решение заново.

В данный момент доступны следующие решения виртуализации для мобильных устройств:

- `QEMU без KVM` — ограниченный набор эмулируемых платформ на базе `ARM` и оборудования [4]. Медленная работа и большое энергопотребление на мобильных устройствах

[3] `http://en.wikipedia.org/wiki/X86`

[4] `http://qemu.weilnetz.de/qemu-doc.html#ARM-System-emulator`

- KVM/ARM — находится в стадии разработки[5], только для процессоров архитектуры ARM с аппаратной поддержкой виртуализации (например, ARM Cortex-A15, ARM Cortex-A7). Сейчас такие процессоры еще не получили распространения.[6]

- Xen ARM [6] — поддерживает процессоры архитектуры ARMv7 с аппаратной поддержкой виртуализации и без нее. Это решение требует портирования для каждого мобильного устройства. В данный момент поддерживается только nVidia Tegra250 Development Board.

- VMware Horizon Mobile [2] — проприетарный гипервизор второго типа[7] от VMware. На основе него компания VMware выпустила решение, реализующее сценарий использования Bring Your Own Device.

- LXC — технология контейнерной виртуализации (виртуализация на уровне операционной системы), позволяющая запускать несколько изолированных пользовательских окружений на одном устройстве. Все окружения используют общее ядро. LXC не зависит от аппаратной платформы, поэтому портирование на ARM не требуется.

- Cells [1] — технология контейнерной виртуализации, предназначенная для запуска нескольких пользовательский окружений Android на одном устройстве. Судя по [1], виртуализация в Cells основывается на использовании пространств имен (namespaces) в ядре Linux. В Cells используются как стандартные пространства имен, так и специально созданные в рамках проекта. Cells, в первую очередь, занимается мультиплексированием устройств и обеспечением удобства работы пользователя.

  Текущая реализация Cells очень сильно зависит от предположения, что одновременно используется только один Android. Компания CellRox[8] лицензировала Cells для создания коммерческих продуктов.

Как мы видим, набор доступных продуктов достаточно небольшой. Наиболее близки к готовности только проприетарные коммерческие продукты. Открытые технологии либо не могут быть использованы на многих устройствах (Xen, KVM), либо находятся в разработке (KVM), либо неудобны в использовании для конечного пользователя, т. к. все перечисленные технологии предназначены для использования IT-специалистами. Кроме того, большинство перечисленных технологий появились недавно, поэтому вызывает сомнение стабильность их работы. Видимо, только LXC и QEMU можно считать достаточно надежными, т. к. они довольно легко портируемы на другие архитектуры.

[5] http://www.virtualopensystems.com/
[6] http://en.wikipedia.org/wiki/ARM_Cortex-A15_MPCore#Implementations
[7] http://en.wikipedia.org/wiki/Hypervisor#Classification
[8] http://cellrox.com

1.4 Android как платформа для исследований в области виртуализации

Android — основанная на Linux операционная система. Ядро этой ОС является ядром Linux с небольшим количеством модификаций.

Большая часть кода Android из пространства ядра и пространства пользователя полностью открыты.

Производители устройств вместе с Android для своих устройств поставляют библиотеки, специфичные для этого устройства. Исходных кодов к этим библиотекам обычно нет. Часть драйверов также поставляются в бинарном виде.

На данный момент доля рынка Android составляет 50–60% и постепенно растет [9]. Android является единственной открытой ОС среди основных мобильных ОС с высокой долей рынка.

Благодаря тому что Android основан на Linux, для его исследования и модификации можно применять стандартные инструменты. Таким образом, ОС Android хорошо подходит для исследований в области виртуализации на мобильных устройствах.

[9] http://en.wikipedia.org/wiki/Android_(operating_system)#Market_share

2 Постановка и анализ задачи

Виртуализация ОС `Android` позволила бы реализовать сразу несколько сценариев использования:

- «Bring Your Own Device» реализуется в полном объеме без ограничений.
- «Создание песочницы» реализуется в полном объеме.
- «Смена прошивок» реализуется в случае если:
  - Виртуализация реализована для прошивки, которую пользователь хочет использовать (переносимость системы виртуализации зависит от способа ее реалзации).
  - Изменения, вносимые прошивкой в компоненты системы виртуализации, разделяемые между всеми экземплярами `Android`, (например, ядро ОС) не конфликтуют между собой.
- «Виртуализированное окружение для тестирования приложений» реализуется при дополнительной доработке системы виртуализации под этот usecase. Доработка предполагает добавление возможности выполнения виртуальными устройствами сценария, заданного пользователем или записи состояния устройств в каждый момент времени (например записи содержимого экрана).

2.1 Анализ существующих продуктов виртуализации Android

- `Cells` [1]

  Использование контейнерной виртуализации и общего ядра для всех пользовательских окружений `Android` делает решение `Cells` хорошо применимым на практике, т. к. дополнительные требования к вычислительным ресурсам и дополнительный расход заряда батареи устройства незначтельны [1].

- `VMware Horizon Mobile` [2]

  `VMware` для своего решение создала полноценный гипервизор второго типа, виртуализирующий аппаратное обеспечение абстрактной машины. Для этой машины было создана ядро `Linux` с патчами `Android`, поверх которого запускается созданное `VMware` пользовательское окружение Android. Гипервизор работает на распространенных типах процессоров ARM, не имеющих расширений для аппаратной виртуализации. Гипервизор работает как процесс в хостовой ОС (например, в `Android`, который поставляется вместе с устройством).

  Понятно, что виртуализация всего аппаратного обеспечения требует достаточно много вычислительных ресурсов, поэтому запуск более чем одного гипервизора, видимо, будет нецелесообразен. По нашему мнению, подход `VMware` является более сложным для реализации, чем подход `Cells`.

- `TrustDroid` [3]

  Является прототипом системы изоляции приложений, основанной на доменах. Целевая ОС этой системы — `Android`. Каждому приложению (в том числе стандартным приложениям `Android`) назначается домен. Приложения из разных доменов не могут взаимодействовать между собой и не могут работать с данными, опубликованными приложениями из других доменов.

  Эта система не использует виртуализацию аппартного обеспечения или пространства пользователя. Для поддержки политики доменов изменения вносятся в ядро и в стандартные системные приложения `Android`. Эта система является самой нетребовательной к ресурсам, по сравнению с приведенными выше. Реализация сценариев использования «bring your own device» и «создание песочницы» несколько отличается: в предыдущих случаях политики настраиваются на уровне пользовательских окружений `Android`, при использовании же `TrustDroid` пользователю необходимо настраивать политики и домены вручную для каждого приложения, что выглядит значительно сложнее для использования, т.к. количество пользовательских окружений, как правило, исчисляется единицами.

  Также `TrustDroid` имеет недостатки с точки зрения безопасности: он предполагает что стандартные системные приложения `Android` и ядро ОС никогда не скомпроментированы, но это может быть неверно, т. к. при получении прав суперпользователя их можно подменить. Эти недостатки, вобщем-то, можно преодолеть, значительно доработав `TrustDroid`.

  Тем не менее, `TrustDroid` решает только задачу обеспечения безопасности на устройстве. Пользователю же в некоторых случаях важна возможность изоляции данных, находящися в разных `Android`'ах, а также независимость действий, производимых в каждом из них.

Вывод Видимо, подход `Cells` имеет лучшее соотношение достоинств и недостатков. Было решено использовать подход `Cells`.

2.2 Цель работы

Цель работы, во многом, заключается в решении обозначенной далее проблемы. Пользовательское окружение `Android` (далее просто `Android`) как часть ОС, преоплагает, что использует все периферийные устройства монопольно (ни с кем их не разделяет). Драйверы периферийных устройств чаще всего написаны, исходя из предположения, что их использует только один `Android`. Таким образом, при запуске нескольких `Android`'ов на одном устройстве, в периферийных устройствах и их драйверах возникают критические ошибки. Целью данной работы является обеспечить возможность одновременного использования периферийных устройств несколькими `Android`'ами, запущенными на одном устройстве.

2.3 Задачи работы

Для обеспечения возможности одновременного использования периферийных устройств несколькими запущенными Android'ами, требуется научиться управлять доступом Android'ов к периферийным устройствам. Для этого достаточно подменить физические устройства на виртуальные. Виртуальные устройства будут перехватывать все запросы Android'ов к физическим устройствам, благодаря чему получат контроль над доступом Android'ов к физическим устройствам. Данный метод работает в общем случае. Требуется учитывать, что создание виртуального периферийного устройства может быть сложнее, чем внесение небольших изменений в его драйвер, позволяющих обеспечить его одновременное использование. Тем не менее, и при явном создании виртуального периферийного устройства и при внесении небольших изменений в его драйвер, выполняется виртуализация периферийного устройства - для каждого Android'а создается видимость монопольного использования физического периферийного устройства.

Задачей данной работы является обеспечить возможность одновременного использования:

1. Аудио устройства

   Пользователю, как правило, хотелось бы слышать звук, всех запущенных Android'ов, но фактически одновременно использовать физическое аудио устройство может только один Android. Остальные сообщают об ошибке при попытке его использования и иногда не могут воспроизводить звук до следующей перезагрузки. Кроме того, требуется, чтобы аудио устройство работало с сервисом телефонии, а именно, чтобы во время разговора аудио устройство позволяло записывать голос пользователя и воспроизводить голос собеседника. Реализация данного требования сильно зависит от реализации сервиса телефонии при его использовании несколькими Android'ами.

2. Фреймбуфера

   Фреймбуфер, по сути, является абстракцией экрана. Экран является одним из основных способов взаимодействия пользователя и устройства, поэтому требуется обеспечить возможность его одновременного использования запущенными Android'ами. Кроме необходимости устранить все критические ошибки при работе Android'ов с экраном, требуется выработать политику по разделению к нему доступа, т.к. Android'ы одновременно пишут свою картинку на экран, где она смешивается и пользователь не понимает что изображено.

3. GPU

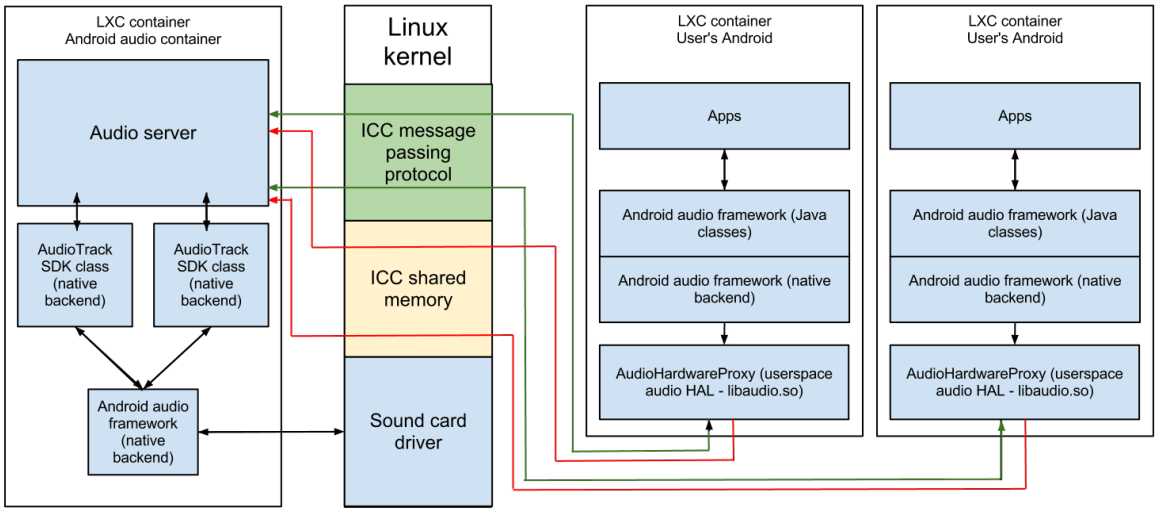
   Большая часть игр не работает при осутствии GPU. Стандартные меню и рабочий стол многих Android прошивок также не работают при отсутствии GPU. Эти факты говорят о том, что GPU является важным устройством, доступ к которому требуется иметь из каждого запущенного Android'а.

3 Виртуализация периферийных устройств

3.1 Виртуализация аудио устройства

3.1.1 Архитектура

Схема виртуализации аудио устройства:



Виртуальное аудио устройство должно позволять:

- Проигрывать звук всех запущенных `Android`'ов
- Взаимодействовать с сервисом телефонии для записи голоса пользователя и воспроизведения голоса собеседника
- Воспроизводить звук на разных устройствах вывода звука (динамик, трубка, наушники, bluetooth гарнитура)

Виртуальное аудио устройство, при помощи подмены библиотеки `libaudio.so` в `Android`'ах пользователя, получает контроль над воспроизводимыми ими звуковыми потоками. Эти звуковые потоки отправляются при помощи механизма межконтейнерного взаимодействия в `аудио сервер`, где воспроизводятся при помощи нативного бэкэнда `Android SDK`. `Аудио сервер` запускается в невиртуализированном пользовательском окружении `Android`, поэтому воспроизведение при помощи `Android SDK` осуществляется на физическом аудио устройстве. Проигрывание звука на различных устройствах вывода и интеграция с сервисом телефонии, во многом, заложены в аудио подсистеме `Android`, которая используется для этого `аудио сервером`.

3.1.2 Получение контроля над звуковыми потоками `Android`

Для исключения конфликтов между `Android`'ами при доступе к физическому аудио устройству требуется сделать так, чтобы `Android`'ы не работали с этим устройством напрямую. Для этого необходимо найти уровень абстракции звукового оборудования (интерфейс), который:

- Используется на всех Android устройствах.
- Имеет фиксированный и простой интерфейс.
- Ответственен за работу с физическим устройством и никакой другой компонент Android не делает этого.
- Позволяет в его реализации получить звуковые потоки, проигрываемые Android'ом, использующим этот уровень абстракции.

Так как Android основан на Linux, было решено проверить используется ли в Android стандартный для Linux уровень абстракции звукового оборудования ALSA.

В исходных кодах ядер смартфонов Samsung Galaxy SII и Google Nexus S был найден драйвер их звукового устройства (sound/soc/s3c24xx/). Для обоих смартфонов он одинаковый. Эти драйверы используют для своей работы подсистему Alsa System on Chip (ASoC) - реализацию ядерной компоненты подсистемы ALSA более подходящую для SoC устройств. Описание этой подсистемы можно найти в стандартной документации ядра Linux:
Documentation/sound/alsa/soc/overview.txt.

В логах смартфона Samsung Galaxy SII присутствуют записи говорящие об использовании ALSA:
I/ALSAModule(3700): Initialized ALSA PLAYBACK device hifi
D/AudioFlinger(3700): setParameters(: io 1, keyvalue routing=2, tid 98, calling tid 68
I/AudioFlinger(3700): AudioFlinger's thread 0x5e7e0 ready to run
D/AudioHardwareYamaha(3700): AudioStreamOut.setParameters(keyValuePairs ="routing=2")
D/AudioHardwareYamaha(3700): doRoutingVOIP(0x2,0)
I/AudioHardwareYamaha(3700): AudioHardware.setDevices(devices=00000002h, mask=0000BFFF
W/AudioPolicyManagerBase(3700): getDeviceForStrategy(unknown strategy: 4
I/ALSAModule(3700): Terminated ALSA PLAYBACK device hifi)

В пользу версии об использовании ALSA говорило и то, что после запуска Android и проигрывания им какого-либо звука можно было успешно запустить программу speaker-test, использующую стандартную ALSA библиотеку пространства пользователя libasound.so и проигрывающую с ее помощью шум.

Полученные данные достаточно убедительно говорили об использовании ALSA на Android. Оставалось невыясненным почему на устройстве Google Nexus S нет библиотеки ALSA (libasound.so). Было решено поискать информацию на эту тему в интернете. В списке рассылки
http://music.columbia.edu/pipermail/andraudio/2011-February/000176.html и в Android PDK
(http://www.kandroid.org/online-pdk/guide/audio.html) было найдено что ALSA может использоваться на устройстве, но лежит она на уровне ниже стандартизированного уровня абстракции звукового оборудования
AudioHardwareInterface. В тех же источниках было установлено какой интерфейс C++

имеет
AudioHardwareInterface. Файл с ним расположен в дереве исходников Android по пути hardware/libhardware_legacy/include/hardware_legacy/AudioHardwareInterface.h. Приведем его важный фрагмент:

```
/**
 * AudioHardwareInterface.h defines the interface to the audio hardware abstraction layer.
 * The interface supports setting and getting parameters, selecting audio routing
 * paths, and defining input and output streams.
 * AudioFlinger initializes the audio hardware and immediately opens an output stream.
 * You can set Audio routing to output to handset, speaker, Bluetooth, or a headset.
 * The audio input stream is initialized when AudioFlinger is called to carry out
 * a record operation.
 */
class AudioHardwareInterface
{
public:
    /**
     * setMode is called when the audio mode changes. NORMAL mode is for
     * standard audio playback, RINGTONE when a ringtone is playing, and IN_CALL
     * when a call is in progress.
     */
    virtual status_t    setMode(int mode) = 0;
    // set/get global audio parameters
    virtual status_t    setParameters(const String8& keyValuePairs) = 0;
    virtual String8     getParameters(const String8& keys) = 0;

    /** This method creates and opens the audio hardware output stream */
    virtual AudioStreamOut* openOutputStream(
                                uint32_t devices,
                                int *format=0,
                                uint32_t *channels=0,
                                uint32_t *sampleRate=0,
                                status_t *status=0) = 0;
    virtual    void         closeOutputStream(AudioStreamOut* out) = 0;
    static AudioHardwareInterface* create();
};
// ----------------------------------------------------------------------------
extern "C" AudioHardwareInterface* createAudioHardware(void);
```

AudioHardwareInterface является сервисом, который инициализирует аудио устройства и позволяет открыть поток воспроизведения аудио AudioStreamOut, назначить audio_mode, сконфигурировать себя при помощи параметров.

Выше приведено объявление C функции `createAudioHardware(void)` назначением которой является создание экземпляра интерфейса `AudioHardwareInterface` специфичного для устройства на котором запущен `Android`. Эту функцию вызывает один из основных и самых крупных компонентов аудио подсистемы `Android - AudioFlinger`. Функция `createAudioHardware(void)` обычно находится в библиотеке `libaudio.so`, которая находится в зависимостях у `libaudioflinger.so`.

Аудио параметры и `audio_mode` описываются в файле `frameworks/base/include/media/AudioSystem.h`. Приведем его фрагмент:

```
enum audio_mode {
  MODE_INVALID = -2,
  MODE_CURRENT = -1,
  MODE_NORMAL = 0,
  MODE_RINGTONE,
  MODE_IN_CALL,
  MODE_IN_COMMUNICATION,
  NUM_MODES  // not a valid entry, denotes end-of-list
};
class AudioParameter {
  public:
    AudioParameter() {}
    AudioParameter(const String8& keyValuePairs);
    virtual ~AudioParameter();
    // reserved parameter keys for changing standard parameters with
       setParameters() function.
    // Using these keys is mandatory for AudioFlinger to properly monitor audio
       output/input
    // configuration changes and act accordingly.
    //  keyRouting: to change audio routing, value is an int in AudioSystem.
       audio_devices
    //  keySamplingRate: to change sampling rate routing, value is an int
    //  keyFormat: to change audio format, value is an int in AudioSystem.
       audio_format
    //  keyChannels: to change audio channel configuration, value is an int in
       AudioSystem.audio_channels
    //  keyFrameCount: to change audio output frame count, value is an int
    //  keyInputSource: to change audio input source, value is an int in
       audio_source
    //      (defined in media/mediarecorder.h)
    static const char *keyRouting;
    static const char *keySamplingRate;
    static const char *keyFormat;
    static const char *keyChannels;
    static const char *keyFrameCount;
    #ifdef HAVE_FM_RADIO
    static const char *keyFmOn;
    static const char *keyFmOff;
    #endif
```

```
34      static const char *keyInputSource;
};
```

При переходе в обычный режим работы `AudioFlinger` вызывает `AudioHardwareInterface::setMode(MODE_NORMAL)`. При разговоре по телефону `AudioFlinger` вызывает `AudioHardwareInterface::setMode(MODE_IN_CALL)`. При поступлении звонка нужно проигрывать звук, поступающий из сотовой сети, а также отправлять звук с устройства аудио захвата в сотовую сеть. Эксперименты показали, что существенная часть этих обязанностей исполняется аудио устройством, его драйвером и библиотекой, поставляемой с устройством и реализующей интерфейс `AudioHardwareInterface`. Кроме того при исследовании аудио подсистемы `Android` и радио интерфейса `Android` не было найдено других мест где могли бы исполняться подобные функции. В каких случаях используются остальные `audio_mode` неизвестно, т.к. их использование не наблюдалось.

В классе `AudioParameter` описаны стандартные параметры, при помощи которых можно конфигурировать `AudioHardwareInterface` или получить информацию о нем. Параметры `keySamplingRate`, `keyFormat`, `keyChannels`, `keyFrameCount` можно считать параметрами только для чтения, так как они, как правило, задаются ограничениями аудио оборудования и не меняются. Параметры `keyFmOn` и `keyFmOff` предназначены для работы с `FM` радио и для прототипа, реализуемого в данной работе, не представляют интереса. Остается единственный важный для параметр - `keyRouting`. Обычно у мобильного устройства есть несколько устройств воспроизведения звука - большой динамик, малый динамик для разговоров по телефону, `mini jack` выход на наушники, `bluetooth` гарнитура. Параметр `keyRouting` отвечает за то на каком из этих устройств будет проигрываться звуковой поток `Android`'а. Так как сценарии использования этих устройств воспроизведения очень разные, требуется выработать политику, которая будет определять звуковые потоки каких `Android`'ов воспроизводить и на каких устройствах. Об этом будет рассказано в одно из следущих подпунктов.

Рассмотрим фрагмент интерфейса `AudioStreamOut` - аудио потока предназначенного для воспроизведения:

```
1  /**
    * AudioStreamOut is the abstraction interface for the audio output hardware.
3   * It provides information about various properties of the audio output hardware
        driver.
    */
5  class AudioStreamOut {
   public:
7      /** write audio buffer to driver. Returns number of bytes written */
       virtual ssize_t      write(const void* buffer, size_t bytes) = 0;
9  };
```

В нем объявлен метод write, который принимает буфер, содержащий “сырой” аудио поток в формате `pcm 16 bit stereo`. Этот аудио поток представляет из себя все смикшированные аудио потоки `Android`'а. Таким образом, внутри этого метода получается доступ к аудио потоку `Android`'а.

Рассмотрев все интерфейсы из `AudioHardwareInterface.h`, мы видим, что данный уровень абстракции аудио оборудования удовлетворяет всем требованиям к уровню абстракции аудио оборудования, определенным в начале раздела. Следовательно, для получения контроля над аудио потоками `Android`'ов и исключения конфликтов при использовании физических звуковых устройств требуется реализовать интерфейсы из `AudioHardwareInterface.h`.

В качестве основы для реализации был использован `frameworks/base/services/audioflinger/AudioHardwareGeneric.cpp`, который позволяет писать звук `Android`'а в файл. Реализация `AudioHardwareInterface` быда названа `AudioHardwareProxy`, т.к. ее обязанностью является подмена настоящего `AudioHardwareInterface` и передача управляющих комманд и аудио потока в другой компонент виртуализации аудио - `аудио сервер`. Кроме того, `AudioHardwareProxy` позволяет имитировать проигрывание аудиопотока `Android`'а без настоящего проигрывания. Для этого, в первую очередь, требутеся имитировать временные задержки в зависимости от размера проигрываемого буфера.

3.1.3 Аудио сервер

Обязанностью `аудио сервера` является микширование аудипотоков `Android`'ов и воспроизведение получившегося аудиопотока на физическом аудио устройстве. `Аудио сервер` получает от `AudioHardwareProxy`, работающих в каждом запущенном `Android`'е, звуковой поток, текущий затребованный routing (текущее значение параметра `keyRouting`), текущий `audio_mode`. Этой информации достаточно `аудио серверу`, чтобы принять решение звуковые потоки каких `Android`'ов воспроизводить и на какое устройство вывода аудио. Решение принимается на основе заданной политики.

В рамках этой политики каждому routing'у присваивается число - приоритет. Сама политика работает в соответствии со следующими правилами:

- В каждый момент времени воспроизводятся звуковые потоки только тех контейнеров, у которых текущий приоритет routing'а равняется максимальному среди текущих приоритетов.
- Если в данный момент контейнер не воспроизводит звук, то его routing не учитывается при определении routing'а с максимальным приоритетом.
- На физическом устройстве в каждый момент времени выставляется routing, который в данный момент имеет максимальный приоритет.

Приведем код задающий приоритеты routing'ов и промоделируем некоторые сложные ситуации, в которых эта политика показывает себя хорошо:

```
//order relation for routing values by "importance"
//so if container routingOrder is less then order of curHWRouting
//then we don't play it
int routingOrder(int value) {
    switch(value) {
        case AudioSystem::DEVICE_OUT_EARPIECE:
            return 5; //small speaker
        break;
        case AudioSystem::DEVICE_OUT_SPEAKER:
            return 0; //big speaker
        break;
        case AudioSystem::DEVICE_OUT_WIRED_HEADPHONE:
        case AudioSystem::DEVICE_OUT_BLUETOOTH_SCO:
            return 10; //Headset
        break;
        case 3:
        case 4:
        case 5:
        case 6:
        case 7:
        case 9:
            LOGW("Assigning -1 order to %d routing", value);
            return -1; //unknown
        break;
        default:
            LOGW("Assigning -1 order to %d routing", value);
            return -1;
    }
}
```

Как мы видим для наушников (headset) задан максимальный приоритет, для маленького динамика (small speaker, handset) задан средний и для большого динамика задан минимальный приоритет. Рассмотрим несколько сценариев работы:

1. Пользователь вставляет наушники в устройство.
2. Пользователь запускает воспроизведение аудио в контейнере 1.
3. Контейнер 1 устанавливает максимальный приоритет равный 10 и начинает воспроизведение.
4. Пользователь запускает воспроизведение аудио в контейнере 2.
5. Контейнер 2 устанавливает максимальный приоритет равный 10 и начинает воспроизведение.
6. Пользователь останавливает воспроизведение в контейнере 1. Контейнер 2 продолжает воспроизведение с приоритетом 10 на наушники.

7. Пользователь достает наушники из устройства. Контейнер 2 меняет routing на большой динамик с приоритетом 0. Контейнер 1, в котором остановлено воспроизведение не меняет routing (это делается только во время воспроизведения), его routing остается с приоритетом 10. Контейнер 2 воспроизводится т.к. нет контейнеров звук которых воспроизводится и приоритет которых выше 0.

8. Ожидания пользователя не нарушены.

1. Пользователь слушает музыку в контейнере 1 на большом динамике с приоритетом 0.

2. В контейнер 2 поступает телефонный звонок. Контейнер 2 устанавливает routing на малый динамик с приоритетом 5. `AudioHardwareProxy` в контейнере 1 эмитирует воспроизведение аудио контейнером 1. Текущий routing на физическом устройстве устанавливается в routing на малый динамик. Максимальный приоритет равен 5.

3. Пользователь заканчивает телефонный разговор в контейнере 2. Контейнер 2 перестает играть звук, максимальный приоритет среди контейнеров воспроизводящих звук становится равным 0. Контейнер 1 снова воспроизводится на физическом устройстве, т.к. максимальный приоритет равен приоритету контейнера 1.

4. Ожидания пользователя не нарушены.

Остается невыясненным вопрос как аудио сервер микширует звуковые потоки и воспроизводит их на физическом устройстве. Т.к. хотелось бы сделать аудио сервер максимально переносимым, то для воспроизведения и микширования аудио нужно использовать компоненты, идущие в составе `Android` и имеющие интерфейс не меняющийся на разных устройствах. Понятно, что запустить такие компоненты можно только в пользовательском окружении, которое может разрешить все зависимости этих компонент. Т. к. эти компоненты обычно работают в `Android`, то и запускать их придется не в обычном `GNU` окружении, а в окружении с библиотеками `Android` (в т.ч. `libc`), для чего понадобится отдельный контейнер.

Первый прототип аудио сервера использовал для воспроизведение реализацию `AudioHard`
поставляемую с устройством, а для микширования использовались возможности библиотеки `GStreamer`
(`http://gstreamer.freedesktop.org/`). Реализация `AudioHardwareInterface` была выбрана для воспроизведения из-за своей простоты и хорошей изученности во время работы, описанной в разделе «Получение контроля над звуковыми потоками `Android`». Использование `GStreamer`, являющегося достаточно большой библиотекой имеющей много зависимостей, а также большое потребление процессора при микшировании (стабильно 20% одного ядра) побудило к поиску другого решения.

Известно, что `Android` каким-то образом осуществляет микширование своих аудио потоков. Также известно, что для программистов `Android` приложений в `SDK` есть классы для работы со звуком. Естественно, что звуковой поток, воспроизводимый при помощи

таких классов, микшируется с остальными звуковыми потоками, вопроизводящимися в данный момент. Из этих соображений следует, что для воспроизведения и микширования можно использовать Java классы из Android SDK. Для этого понадобится Android с невиртуализированным AudioHardwareInterface. Можно запустить такой Android в отдельном контейнере. SDK классы в нем воспроизводили бы звук на физическом устройстве. Далее в этом контейнере можно запустить аудио сервер (уже теперь написанный на Java). Классы из SDK позволили бы воспроизводить звуковые потоки контейнеров и микшировать их. Но реализовывать такое решение в чистом виде не представляется возможным, т. к. каждый экземпляр Android занимает около 250 Мб оперативной памяти. Кроме того он занимает процессорное время и место в постоянной памяти устройства.

Этот подход удалось реализовать с минимальными дополнительными затратами памяти и процессорного времени. Получилось запустить всю аудио подсистему Android и получить доступ к ней используя Android IPC не запуская виртуальную машину Java, не смотря на то, что классы из SDK это Java классы, а без запуска виртуальной машины не работает более 95 % Android.

Запустить подсистему аудио удалось достаточно просто, т.к. выяснилось что она и так запускается отдельно от большей части Android'а. Важным здесь является то, что она написана на C++, а не на Java.

Java классы из SDK удалось использовать без запуска JVM благодаря следующей их особенности: во многом эти классы являются обертками над нативными C++ классами, интерфейс которых похож на интерфейс Java классов. Это даже позволило использовать документацию из SDK для работы с ними. Эксперименты показали что нативные классы без своей Java части работают именно так как ожидается (в соответствии с документацией из SDK для Java классов). Для воспроизведения аудиопотока каждого контейнера используется класс AudioTrack. AudioTrack вызывает callback когда нужно предоставить аудио буфер для воспроизведения. В callback'е аудио сервер отправляет запрос к AudioHardwareProxy на получение нового аудио буфера определенного размера. AudioHardwareProxy удовлетворяет запрос. Воспроизведение осуществляет плавно, без задержек, "голоданий" или переполнений буфера аудио карты. Аудио подсистема Android микширует звуковые потоки всех AudioTrack, работающих в данный момент. Тестирование такого решения показало, что микширование теперь занимает гораздо меньше процессорного времени (от 1-2 до 10% одного ядра). Кроме того данное решение избавлено от зависимости - библиотеки GStreamer.

Как уже говорилось, удалось достаточно легко запустить звуковую подсистему Android в отдельном контейнере. Тем не менее в нее пришлось внести небольшие изменения. Сервис AudioFlinger при запросе действий с аудио системой осуществляет проверку разрешений на работу со звуком для приложения сделавшего запрос. Аудио сервер эту проверку не проходил, т.к. по умолчанию в Android у приложений нет никаких разрешений. В качестве аудио сервера используется нативное не Java приложение. Полностью нативное приложение не может пройти процедуру установки и закрепления за собой нужных разрешений. Если бы использовалось Java приложение, то понадобилось бы запустить

весь Android (так устроен процесс его запуска). Для того чтобы обойти данную проблему были внесены изменения в аудио подсистему Android. Были отключены проверки разрешений в файлах AudioFlinger.cpp и AudioPolicyService.cpp. Для этого в функциях recordingAllowed(), settingsAllowed() и checkPermission() нужно всегда возвращать true.

3.1.4 Взаимодействие AudioHardwareProxy и аудио сервера

В предыдущих подпунктах говорилось что AudioHardwareProxy передает звуковой поток контейнера аудио сервер'у, а также уведомляет его о событиях изменения routing'а, старта/останова воспроизведения и изменения audio_mode. Стоит заметить, что подобное межконтейнерное взаимодействие невозможно при клонировании сетевых пространств имен контейнеров (а это происходит при запуске контейнеров). Сокеты в разных контейнерах не видят друг друга. В проекте, частью которого является данная работа, был реализован механизм межконтейнерного взаимодействия на основе протокола Netlink. По сравнению с протоколом TCP он получился менее удобным, что усложнило компоненты виртуализации аудио, использующие этот механизм. Например, в рамках одного процесса можно отсылать сообщения из любого потока (thread'а), но принимать можно только из одного. Потокам приходится ждать сообщения используя pthread condition variable. Поток слушатель сообщений будит поток получатель сообщения и предоставляет последнему информацию о полученном сообщений.

Так как достаточно большие потоки несжатого аудио передаются из контейнера с AudioHardwareProxy в контейнер с аудио сервер'ом и таких потоков несколько (по одному на каждый пользовательский Android), то хотелось бы делать это эффективно, максимально исключая копирование аудио потоков. Для исключения копирования обычно используется разделяемая память. Но из-за клонирования пространств имен IPC, контейнеры не видят стандартную System V IPC разделяемую память друг друга. Поэтому был написан свой драйвер межконтейнерной разделяемой памяти. За выделение и освобождение буферов в разделямой памяти отвечает аудио сервер.

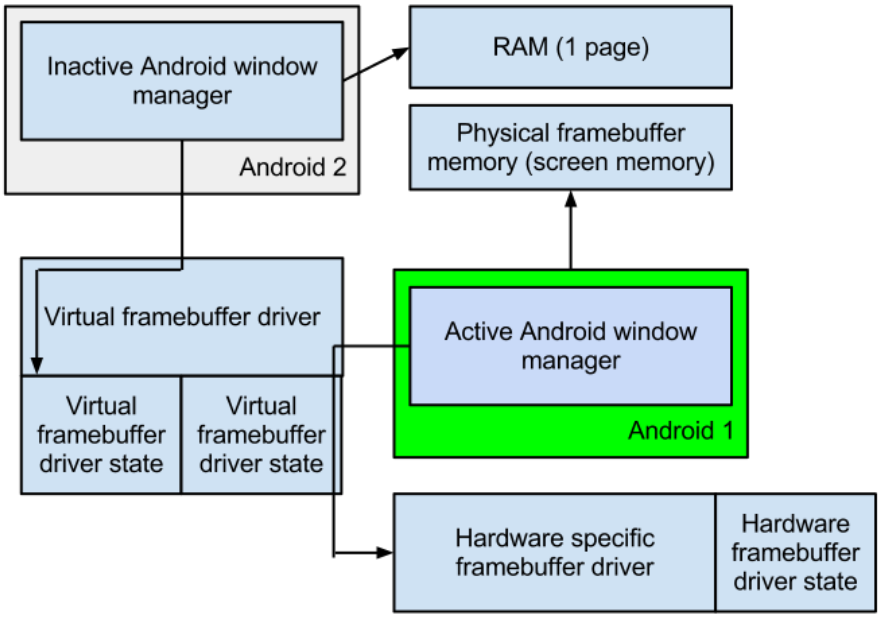
3.1.5 Анализ полученного решения

Данное решение работает на версиях 2.x Android. На Android 4.0 не проверялось от части по причине все еще малой его распространенности. Тестирование решения заключалось в долговременном использовании аудио в разных контейнерах. Ошибок не выявленно. Также отметим что виртуализация аудио не была выполнена в проекте Cells [1] и дизайн виртуального аудио устройства не находился под его влиянием.

3.2 Виртуализация фреймбуфера

3.2.1 Архитектура

Схема виртуализации `фреймбуфера`:



Использование виртуального `фреймбуфера` `Android`'ами должно устранить критические ошибки, возникающие при использовании физического фреймбуфера. `Фреймбуфер` с аппаратной точки зрения можно рассматривать как устройство вывода графики, например, экран устройства.
Виртуальный `фреймбуфер` должен предоставить возможность разделения экрана устройства, чтобы картинка `Android`'ов на нем не смешивалась. Для этого виртуальный `фреймбуфер` позволяет отображать одновременно только один `Android`, называемый активным. При этом виртуальный `фреймбуфер` позволяет в любой момент времени переключить активный `Android` на другой. Для всех `Android`'ов в виртуальном `фреймбуфере` выделяется виртуальное состояние, но только виртуальное состояние активного `Android`'а применяется к драйверу физического фреймбуфера.

3.2.2 Анализ методов виртуализации фреймбуфера

При виртуализации `фреймбуфера` требуется разграничить к нему доступ так, чтобы кадры, которые пишут в него `Android`'ы не смешивались на экране. Можно выполнить эту задачу двумя способами:

- Ввести понятие активного в данный момент `Android`'а и отображать только его кадры. При этом предусмотреть возможность менять активный `Android` в любой момент времени.

- Отображать на экране устройства сразу несколько Android'ов при этом выделив определенную область экрана для каждого из них.

Для реализации был выбран первый способ. Этому способствовали следущие соображения:

- Экраны наиболее распространенных мобильных устройств с Android (смартфонов) слишком малы чтобы удобно работать сразу с несколькими Android'ами.
- Реализация второго способа выглядит сложнее.
- С точки зрения задачи создания прототипа виртуализации пользовательского окружения Android не важно какой из способов будет реализован.

Аналогично со случаем виртуализации аудио устройства требуется найти такой уровень абстракции графического оборудования, который:

- Используется на всех Android устройствах.
- Имеет фиксированный и желательно простой интерфейс.
- Ответственен за работу с физическим устройством и никакой другой компонент Android не делает этого.
- Позволяет в его реализации получить кадры, отображаемые Android'ом, использующим этот уровень абстракции.

В графической подсистеме Android имеются стандартизированные интерфейсы для вывода графики: gralloc и OpenGL ES различных версий. Более подробно о них написано в презентации
http://www.vivantecorp.com/Khronos.pdf. Gralloc используется для выделения буферов разделяемой памяти, обладающих следующим свойством: реализация OpenGL ES работающая на устройстве должна иметь возможность осуществлять рендеринг изображения в эти буферы. Буферы должны относиться к классу разделяемой памяти для того, чтобы каждое пользовательское приложение могло рисовать в них графику, а затем оконный менеджер SurfaceFlinger мог осуществлять двумерную композицию растровых изображений каждого приложения. Для последней операции требуется чтобы память буферов была видима в разных процессах, то есть была разделяемой. Вторым сценарием использования gralloc является выполнение при помощи него операции swap buffers
(http://en.wikipedia.org/wiki/Multiple_buffering#Description).
И OpenGL ES и gralloc удовлетворяют требованиям, выдвинутым для уровней абстракции графического оборудования которые имело бы смысл виртуализировать. OpenGL ES имеет большой и сложный интерфейс, поэтому производить виртуализацию на этом уровне было бы крайне сложно. Интерфейс gralloc является достаточно простым, но виртуализация на его уровне имеет значительные недостатки:

- Требование чтобы выделяемая gralloc память могла быть использованна реализацией OpenGL ES, поставляемой с устройством для рендеринга в нее графики делает реализацию gralloc зависимой от OpenGL ES устройства.

- Использование специальных классов памяти, в которые может рендерить OpenGL ES устройства требует написания драйвера, позволяющего разделять между контейнерами память такого класса. Если этого не делать, то потребуется выполнять копирование кадров из контейнера с пользовательским Android в контейнер, в котором запущен графический сервер (по аналогии с виртуализацией аудио).

- Для каждого пользовательского Android потребуется выделить свой фиктивный фреймбуфер, в котором SurfaceFlinger будет собирать кадр. На устройствах с типичным разрешением фреймбуфер занимает 3-6 Мб. Это достаточно большие накладные расходы, учитывая что на мобильных устройствах, как правило, отключен swapping, а запуск нескольких Android'ов существенно уменьшает количество свободной физической памяти.

Проект Cells [1] не подменял gralloc в их решении по виртуализации графики. Почему они не уточняют. Вместо этого ими был написан свой драйвер виртуального Linux framebuffer. Кроме того в Android porting guide
(http://www.kandroid.org/online-pdk/guide/display_drivers.html)
говорится, что при портировании Android на новое устройство нужно написать свой драйвер Linux framebuffer. Написание такого драйвера будет иметь следущие приемущества перед подменой gralloc:

- Не важно каким образом получен кадр, записываемый в фреймбуфер. Не важно какого класса память использовалась для рендеринга графики приложений.

- Виртуализированный драйвер Linux framebuffer для выполнения операций с аппаратным фреймбуфером сможет полагаться на драйвер Linux framebuffer поставляемый с устройством. Этот драйвер имеет стандартизированный интерфейс, т. к. относится к драйверам класса Linux framebuffer.

- Первые 2 пункта делают решение независимым от устройства.

- Так как драйвер Linux framebuffer работает в ядре, то в нем можно изменять виртуальные адресные пространства SurfaceFlinger'ов так чтобы SurfaceFlinger активного Android'а писал кадр в настоящий фреймбуфер, а SurfaceFlinger'ы неактивных Android'ов писали свои кадры в единственную общую для всех страницу памяти и не замечали подмены. Таким образом, удается избежать копирования кадров и дополнительных затрат памяти характерных для способа с подменой gralloc.

Перечиленные приемущества позволяют сделать вывод, что виртуализация графики через создание драйвера виртуального Linux framebuffer является лучшим решением.

3.2.3 Виртуальный Linux framebuffer

Драйвер Linux framebuffer имеет множество стандартных функций, описанных в структуре fb_ops, определенной в файле include/linux/fb.h исходных текстов ядра Linux. Это функции вызывающиеся при открытии и закрытии файла фреймбуфера (обычно /dev/fb0), функции установки видеорежима, проверки значения параметров, которые пытается установить приложение, функции обработки стандартных и нестандартных для Linux framebuffer IOCTL'ов, функции копирования областей кадра и т.п. Всего в ядре 2.6.35 их 21.

Кроме того у каждого драйвера Linux framebuffer есть свое стандартизированное состояние. В нем хранится такая информация как текущее разрешение экрана, физический и виртуальный адрес памяти фреймбуфера, информация о back buffer'ах и т. п.

С точки зрения разграничения доступа к физическому фреймбуферу интересна стандартная функция mmap драйвера виртуального Linux framebuffer. Назначение этой функции во многом соответствует назначению стандартного системного вызова mmap - отобразить в адресное пространство пользовательского процесса память фреймбуфера. После этого пользовательский процесс может изменять состояние памяти фреймбуфера и тем самым вызывать отрисовку нужного изображения на экране устройства. Также достаточно логичным выглядит факт, что единственным пользовательским процессом, который выполняет mmap фреймбуфера в Android является оконный менеджер SurfaceFlinger.

В сценарии использования требуется, чтобы активный Android имел возможность писать свой кадр в память физического фреймбуфера, а неактивные Android'ы писали свои кадры в страницу памяти не относящуюся к физическому фреймбуферу и не замечали при этом подмены. Являясь частью ядра Linux драйвер виртуального Linux framebuffer может производить любые действия с адресным пространством пользовательских процессов. Эта возможность и используется. На практике оказалось, что в интерфейсе подсистемы управления памятью ядра Linux нет стандартных функций для изменения отображения пользовательских страниц на физическую память, если эти страницы уже отображены. Это связано с тем, что ядро само освобождает все страницы отображенные в файл, когда тот закрывается или когда пользовтельской приложение выполняет системный вызов munmap. Видимо, именно поэтому такая функциональность не экспортируется из подсистемы управления памятью для использования драйверами. На основе кода обработчика системного вызова munmap была написана функция, выполняющая отмену отображения страниц пользовательского процесса на физическую память. Важным ее свойством является использование переносимых и безопасных высокоуровневых операций. Приведем код этой функции с комментариями:

```
1  //Clears mapping of old_vma and returns new_vma - vma without mapping
   //to physical memory but with the same virtual adresses as old_vma has
3  static struct vm_area_struct* clone_and_unmap_old_vma(
     struct vm_area_struct *old_vma,
5    struct task_struct* task)
   {
7    struct vm_area_struct* new_vma;
```

```
        struct vm_area_struct* prev;
        struct rb_node** rb_link, *rb_parent;
        //Allocate new_vma using kernel caching allocator
        new_vma = kmem_cache_zalloc(vm_area_cachep, GFP_KERNEL);
        if (!new_vma) {
          LOG_ERROR(``OOM!'');
          return NULL;
        }
        //lock task's virtual address space
        down_write(&task->mm->mmap_sem);
        //clone main fields of old_vma
        new_vma->vm_mm = old_vma->vm_mm;
        new_vma->vm_start = old_vma->vm_start;
        new_vma->vm_end = old_vma->vm_end;
        new_vma->vm_flags = old_vma->vm_flags;
        new_vma->vm_page_prot = old_vma->vm_page_prot;
        new_vma->vm_pgoff = old_vma->vm_pgoff;
        INIT_LIST_HEAD(&new_vma->anon_vma_chain);
        new_vma->vm_file = old_vma->vm_file;
        //do_munmap decreases file node ref counter
        //increase it before this happens
        get_file(new_vma->vm_file);
        //realease old_vma's mapping to physical memory
        //doing all low level job
        do_munmap(old_vma->vm_mm, old_vma->vm_start,
                  old_vma->vm_end - old_vma->vm_start);

        //put new_vma into rb tree which is used to quickly search for task's
        //pages by virtual address
        find_vma_prepare(vma->vm_mm, vma->vm_start,
                         &prev, &rb_link, &rb_parent);
        vma_link(vma->vm_mm, vma, prev, rb_link, rb_parent);

        up_write(&task->mm->mmap_sem);

        return new_vma;
      }
```

Далее очищенные от отображения на физическую память диапазоны виртуальных адресов можно отобразить на произвольную физическую память стандартными функциями, экспортированными из подсистемы управления паматью ядра `Linux`.

Полученные возможности по изменению отображения адресного пространства оконного менеджера на память `фреймбуфера` используются при переключении активного `Anroid`'а следующим образом:

- Адресное пространство оконного менеджера активного `Android`'а отображается на память физического `фреймбуфера`.

- Адресные пространства оконных менеджеров неактивных Android'ов отображаются в обычную страницу RAM устройства.

Кроме правильного отображения адресных пространств оконных менеджеров драйверу виртуального Linux framebuffer нужно эмулировать и другие аспекты поведения, чтобы Android'ы не замечали, что кроме них с этим драйвером работает кто-то еще. Это достигается следующим способом:

- Каждому Android'у выделяется виртуальное состояние драйвера Linux framebuffer.
- Все неактивные Android'ы в процессе своей работы изменяют только виртуальное состояние.
- Действия активного Android'а отражаются как на драйвере физического Linux framebuff так и на его виртуальном состоянии.
- В случае если Android становится активным, то его виртуальное состояние применяется к драйверу физического Linux framebuffer.

Последнее, что необходимо сделать, это заставить gralloc'и, поставляемые с устройством в виде бинарной динамической библиотеки, использовать виртуальный Linux framebuffe вместо Linux framebuffer'а, поставляемого с устройством. Для этого в kobject'е драйвера физического Linux framebuffer'а нужно подменить имя "fb0" на другое, а имя в kobject драйвера виртуального Linux framebuffer'а подменить на "fb0". Такая подмена позволяет получить файловую ноду /dev/fb0 виртуального Linux framebuffer в каждом контейнере с Android. После таких действий gralloc'и начинают использовать драйвер виртуального Linux framebuffer.

3.2.4 Анализ полученного решения

Полученное решение работает в Android для машины goldfish - наиболее простом дистрибутиве Android. Android для goldfish использует программную эмуляцию OpenGL ES 1.0 для отрисовки графики. Этот дистрибутив использовался на ранних стадиях, когда нужно было научиться запускать на одном устройстве несколько самых простых окружений Android. В дистрибутивах, использующих GPU, данное решение применяется только для скрытия и показа панели управления, написанной на Qt. Она не будет рассмотрена в данной работе. Для скрытия неактивных Android'ов, использующих GPU, применется другой механизм. О нем будет рассказано в разделе «Виртуализация GPU». Кроме того реализация виртуального Linux framebuffer обладает заметным недостатком: хотя неактивные Android'ы и не отображаются на экране, они продолжают рисовать графику, тратя на это вычислительные ресурсы и заряд батареи устройства. О том как удалось избавиться от этого недостатка будет рассказано в подпункте «Виртуализиация управления питанием».

3.3 Виртуализация GPU

3.3.1 Архитектура

Виртуализированное GPU должно обеспечить возможность использования GPU всеми запущенными Android'ами. Интерфейс OpenGL, используемый для работы с GPU в Android подразумевает, что приложения не замечают работы друг друга с GPU. Таким образом, в основном все уже виртуализировано. Для устранения критических ошибок, возникающих при запуске нескольких Android'ов, необходимы небольшие изменения в драйверах GPU или скриптах инициализации Android.

3.3.2 Анализ методов виртуализации GPU

Как уже отмечалось в подпункте «Виртуализация фреймбуфера» на устройствах с GPU решение по виртуализации фреймбуфера не работает. Это происходит из-за того что SurfaceFlinger не использует фреймбуфер для построения кадра, хотя и выполняет его mmap. Этот факт можно доказать следующим образом:

- В обработчике mmap драйвера виртуального Linux framebuffer ничего не делаем.
- В результате страницы, которые должны были быть отображены в память фреймбуфера, становятся невалидными и при первом обращении к ним SurfaceFlinger получает сигнал SIGSEGV.
- Но этого не происходит.
- Таким образом SurfaceFlinger при использовании GPU никогда не обращается к памяти фреймбуфера.

Это в свою очередь означает, что решение, описанное в подпункте «Виртуализация фреймбуфера», не работает, т.к. оно основывается на предположении что SurfaceFlinger пишет кадр на экран, используя память фреймбуфера отображенную в свое адресное пространство.

Первое, что необходимо выяснить - возможен ли запуск нескольких Android'ов, использующих GPU на одном устройстве. Это возможно, т. к.:

- У проекта Cells [1] получилось это сделать.
- По их словам, конфликтов при одновременном доступе к GPU нет, т.к. отрисовка графики при помощи OpenGL предполагает использование независимых OpenGL контекстов. Это означает что приложение при работе с библиотекой OpenGL не видит, что вместе с ним OpenGL используют другие приложения.

Cells [1] также утверждает, что для корректной работы нескольких Android'ов с GPU нужно исключить его многократную инициализацию каждым Android'ом.

Одновременная работа с GPU возможна, но при этом остается нерешенной проблема: запущенные Android'ы одновременно пишут свою графику на экран, что неприемлимо для пользователя.

Восстановление трасс исполнения в драйверах GPU смартфонов Google Nexus S и Samsung Galaxy SII позволило обнаружить общий алгоритм построения кадра на экране устройства при использовании GPU. Он заключается в следующем:

- В RAM выделяются буферы, в которые рендерится графика каждого приложения. Рендерингом занимается GPU, поэтому ее MMU программируется драйвером для того чтобы инструкции, выполняющиеся в GPU имели доступ к буферам в RAM.

- Окончательный кадр собирается тоже GPU. Здесь используется аппаратно ускоренная 2D композиция - Blitting (http://en.wikipedia.org/wiki/Blitter). Все задачи по копированию графики отдельных приложений в окончательный кадр берет на себя GPU. Логично предположить что этот процесс управляется оконным менеджером SurfaceFlinger'ом. Чтобы GPU могла собрать окончательный кадр в памяти фреймбуфера, эта память должна быть доступна инструкциям, выполняемым в GPU. Для этого драйвер GPU программирует MMU GPU.

Используя полученную информацию практически сразу можно построить решение по разделению экрана между Android'ами при использовании GPU:

- Сценарий использования остается тем же что и в драйвере виртуального Linux framebuffer: активный Android пишет свой кадр на экран, неактивные в страницу RAM устройства.

- Подобно тому как выполняется подмена отображения пользовательских виртуальных адресов в драйвере виртуального Linux framebuffer, выполняется перепрограммирование MMU GPU, чтобы адреса GPU участвующие в обработке графики активного Android'а указывали на память физического фреймбуфера, а адреса GPU, участвующие в обработке графики неактивных Android'ов указывали на страницу в RAM устройства.

Далее рассмотрим детали реализации этого решения для GPU PowerVR серии SGX смартфона Google Nexus S и GPU ARM серии MALI смартфона Samsung Galaxy SII.

3.3.3 Виртуализация GPU PowerVR серии SGX

Драйвер для этой серии GPU находится в каталоге drivers/gpu/pvr/ дерева исходных текстов ядра Linux смартфона Google Nexus S.

В драйвере была найдена функция, отвечающая за программирование MMU. Она находится в файле sgx/mmu.c и называется MMU_MapPages. Она была модифицирована следующим образом: в случае если MMU GPU программируется на адреса физического фреймбуфера, вызывается функция
andcont_fb_mmaped_callback, которая запоминает аргументы, переданные в функцию MMU_MapPages. Далее немного модифицировав эти аргументы можно вызвать MMU_MapPages и перепрограммировать MMU так как нужно. Из-за того что Android'ы не знают друг о

друге, каждый запущенный Android выполняет отображение адресов GPU в память физического фреймбуфера независимо. Адреса GPU при этом разные, а адреса физического фреймбуфера одинаковые. Зная для каждого Android'а виртуальные адреса GPU, которые должны были быть отображены в память физического фреймбуфера и имея функцию, позволяющую перепрограммирвоать MMU GPU, можно реализовать решение по разделению экрана между несколькими Android'ами. Далее приведены изменения, внесенные в функцию MMU_MapPages (изменения помечены директивой препроцессора #ifdef CONFIG_ANDCONT_PVRSGX_VFB):

```
2101 IMG_VOID
2102 MMU_MapPages (MMU_HEAP *pMMUHeap,
2103                IMG_DEV_VIRTADDR DevVAddr,
2104                IMG_SYS_PHYADDR SysPAddr,
2105                IMG_SIZE_T uSize,
2106                IMG_UINT32 ui32MemFlags,
2107                IMG_HANDLE hUniqueTag)
2108 {
//...
2149 #ifdef CONFIG_ANDCONT_PVRSGX_VFB
2150     {
2151       unsigned int fb_base = registered_fb[0]->fix.smem_start;
2152       unsigned int fb_buff_size = registered_fb[0]->fix.smem_len / 4;
2153
2154       if (SysPAddr.uiAddr >= fb_base && SysPAddr.uiAddr < fb_base +
    fb_buff_size * 2) {
2155         andcont_fb_mmaped_callback(pMMUHeap,
2156             DevVAddr, &SysPAddr, uSize, ui32MemFlags, hUniqueTag);
2157         DevPAddr = SysSysPAddrToDevPAddr(PVRSRV_DEVICE_TYPE_SGX, SysPAddr);
2158       }
2159     }
2160 #endif
```

Для исключения многократной инициализации GPU требуется обеспечить однократный запуск демона pvrsrvinit, а также сделать изменения в файле s3c_lcd/s3c_displayclass.c, исправляющее многократное создание swap chain'ов (http://en.wikipedia.org/wiki/Swap_Chain):

```
    static PVRSRV_ERROR CreateDCSwapChain(IMG_HANDLE hDevice,
                                          IMG_UINT32 ui32Flags,
                                          DISPLAY_SURF_ATTRIBUTES *
                                              psDstSurfAttrib,
                                          DISPLAY_SURF_ATTRIBUTES *
                                              psSrcSurfAttrib,
                                          IMG_UINT32 ui32BufferCount,
                                          PVRSRV_SYNC_DATA **ppsSyncData,
                                          IMG_UINT32 ui32OEMFlags,
                                          IMG_HANDLE *phSwapChain,
                                          IMG_UINT32 *pui32SwapChainID)
```

```
    {
11  //...
        if(psDevInfo->psSwapChain)
13      {
    #ifdef CONFIG_ANDCONT_PVRSGX_VFB
15          *phSwapChain = (IMG_HANDLE)psDevInfo->psSwapChain;
            *pui32SwapChainID =(IMG_UINT32)psDevInfo->psSwapChain;
17          return PVRSRV_OK;
    #else
19          return (PVRSRV_ERROR_FLIP_CHAIN_EXISTS);
    #endif
21      }
       //...
23  static PVRSRV_ERROR DestroyDCSwapChain(IMG_HANDLE hDevice,
                                           IMG_HANDLE hSwapChain)
25  {
        S3C_SWAPCHAIN *sc = (S3C_SWAPCHAIN *)hSwapChain;
27      S3C_LCD_DEVINFO *psLCDInfo = (S3C_LCD_DEVINFO*)hDevice;

29      if(!hDevice
          || !hSwapChain)
31      {
            return PVRSRV_ERROR_INVALID_PARAMS;
33      }
    #ifdef CONFIG_ANDCONT_PVRSGX_VFB
35      if(!psLCDInfo->psSwapChain) {
          return PVRSRV_OK;
37      }
    #endif
```

3.3.4 Виртуализация GPU ARM серии MALI

Драйвер для этой серии GPU находится в каталоге drivers/media/video/samsung/mali/ дерева исходных текстов ядра Linux смартфона Samsung Galaxy SII.

В драйвере была найдена функция mali_address_manager_map. Ее назначение в точности совпадает с назначением функции MMU_MapPages в драйвере GPU серии PowerVR SGX. mali_address_manager_map отображает адреса GPU в адреса физической памяти устройства. Наличие такой функции позволяет полностью повторить решение для разделения экрана между
Android'ами на устройствах с GPU серии PowerVR SGX.

Для GPU ARM серии MALI на смартфоне Samsung Galaxy SII ошибок при запуске нескольких Android'ов не обнаружено. То есть ошибок при многократной конфигурации GPU либо нет, либо конфигурацию GPU однократно выполняет его драйвер во время инициализации ядра. Также в пространстве пользователя отсутствует демон, предназначенный

для конфигурации GPU.

3.3.5 Анализ полученного решения

Описанное решение имеет те же достоинства и недостатки, что и решение описанное в подразделе «Виртуализиация фреймбуфера». Принцип их работы очень похож. Обеспечение доступа к GPU для всех Android'ов позволило запускать приложения с 3D графикой, большая часть которых без GPU не работает.

3.4 Виртуализиация управления питанием

3.4.1 Архитектура

Подсистема управления питанием предназначена для обеспечения эффективного расходования заряда батареи устройства.

Необходимость в виртуализации управления питанием возникла из-за того, что в процессе работы с Android'ом, экран неожиданно мог погаснуть, а при переключении Android'ов он сильно мелькал. Это случалось из-за того, что неактивные Android'ы имели возможность управлять состоянием экрана. Кроме того хотелось бы, чтобы неактивные Android'ы не тратили вычислительные ресурсы и энергию на рендеринг своего UI, ведь он не виден пользователю.

Подсистема управления питанием выполняет команды, отдаваемые ей Android'ом посредством записи в файлы sysfs. В основном они заключаются в переводе устройства в один из режимов энергопотребления. Виртуализация управления питанием происходит в обработчиках чтения и записи sysfs файлов. Например, попытка неактивного Android'а перевести устройство в режим пониженного энергопотребления будет проигнорирована именно в обработчике записи в sysfs файл.

3.4.2 Реализация

Для полноценной виртуализации подсистемы управления питанием потребовалось изучить ее устройство. Она содержит следующие компоненты:

- Sysfs power management. Это обычный для Linux интерфейс управления питанием в sysfs, позволяющий задать ядру текущий режим энергопотребления путем записи его названия в файл /sys/power/state

- Framebuffer early suspend. Это механизм уведомления пространства пользователя о состоянии экрана устройства (включен или выключен). Он позволяет пространству пользователя остановить рисование графики пока экран устройства выключен и возобновить рисование при его включении. Эксперименты показали что Android всегда следует этому сценарию. В Android уведомления Framebuffer early suspend реализованы с использованием двух файлов в sysfs: /sys/power/wait_for_fb_sleep и /sys/power/wait_for_fb_wake. Когда пользовательский поток читает из одного из

этих файлов, системный вызов `read` блокируется до тех пор пока экран не погаснет или включится соответственно.

- `Wakelock`'и. Не будут описаны в данной работе.

Кроме того было замечено, что на невиртуализированном устройстве при нажатии кнопки `Power` `Android` перестает рисовать свой `UI` и гасит экран. Если требуется принудительно перевести `Android` в режим пониженного энергопотребления, то эмуляция нажатия кнопки `Power` именно то что нужно. Такая необходимость возникает когда требуется перевести `Android` из активного состояния в неактивное. Событие нажатие кнопки `Power` является инициатором всех действий `Android`'а по управлению питанием. Именно с эмуляции нажатия кнопки `Power` начинается процесс перевода `Android`'а в неактивное состояние.

Виртуализация `Framebuffer early suspend` осуществляется следующим способом:

- Неактивный `Android` всегда видит что экран выключен.
- Активный `Android` видит настоящее состояние экрана.

Такой подход позволяет остановить рендеринг графики для неактивных `Android`'ов тем самым сэкономив вычислительные ресурсы и заряд батареи устройства. Остановка рендеринга графики неактивных `Android`'ов означает еще и то что на экране устройства показывается `UI` только активного Android'а - картинка на экране не смешивается. `Framebuffer early suspend` является стандартным и видимо всегда используемым механизмом. Таким образом, можно разграничить доступ к экрану не используя методы описанные в подразделах «Виртуализация фреймбуфера» и «Виртуализация GPU». Это решение имеет следующие ограничения:

- Виртуализация `Framebuffer early suspend` останавливает рендеринг графики неактивных `Android`'ов не во всех случаях. Например, в случае поступления входящего звонка, `Android` всегда включает экран и рисует графику.
- При сценарии использования когда экран устройства делится на области, в каждую из которых рисуются кадры разных `Android`'ов это решение не работает. Для данного сценария может хорошо подойти решение, описанное в предыдущих подразделах, т. к. при помощи подмены адресов можно “смещать” кадры каждого `Android`'а по экрану устройства.

Виртуализация `Sysfs power management` осуществляется следующим способом:

- Если режим энергопотребления изменяет активный `Android`, то операция выполняется.
- Если режим энергопотребления изменяет неактивный `Android`, то операция незаметно игнорируется.

Этот подход не нарушает семантику `Sysfs power management` т.к.:

- `Android` использует только 2 режима энергопотребления: `mem` и `on`.
- Наблюдения показали что неактивные `Android`'ы всегда устанавливают режим `mem`.
- Активный `Android` может устанавливать любой из двух режимов.
- Режим `on` является надмножеством режима `mem`, т.е. в режиме `on` включены по крайней мере те же сервисы что и в режиме `mem`.
- Наблюдения показали что работа устройства в режиме `on` не нарушает работу `Android`'ов, запросивших режим `mem`.
- Переключение режимов активным `Android`'ом не нарушает ожидания неактивных.

Выяснилось, что `Android` при нажатии кнопки `Power` может работать не только с подсистемой управления питанием, но и выполнять другие действия, набор которых не фиксирован и зависит от устройства. Повсеместно распространенным примером такого действия является явное управление подсветкой экрана используя файл в `sysfs` специфичный для устройства.

Таким образом, процесс переключения активного `Android`'а выглядит следующим образом:

1. Эмулируем нажатие кнопки `Power` для каждого `Android`'а, участвующего в переключении (текущий активный `Android` и будущий активный `Android`).
2. Считаем что каждый из них сменил свое состояние на противоположное (активный - не активный).

Заметим что неизвестно в какой момент каждый из `Android`'ов, выполнит включение или отключение экрана. Это означает что последним может выполниться отключение. Отключение экрана после переключения активного `Android`'а конечно же удивит пользователя. Но это не единственная проблема. Как уже говорилось `Android` может выполнять и другие действия с устройством. Точно не известно какие, поэтому имеющаяся ситуация гонки может не просто удивить пользователя, но и спровоцировать критическую ошибку. Эта проблема была решена следующим образом: `Android`'у, становящимуся неактивным, дается достаточно времени чтобы выполнить все действия с устройством - перевести его в сотояние пониженного энергопотребления. Только после этого другой `Android` получает возможность перевести устройство в обычное состояние. Для пользователя дополнительная временная задержка незаметна. Теперь не требуется заботиться о том, что на самом деле делают `Android`'ы при нажатии кнопки `Power`. Первый `Android` выполняет все нужные действия, а затем обратные к ним выполняет второй `Android`.

В итоге при переключения активного `Android`'а работает следующий сценарий:

1. Переключение начинается с эмуляции нажатия кнопки `Power` для активного `Android`'а.
2. Он записывает в файл `/sys/power/state` значение “mem”и гасит подсветку экрана.

3. Ядро отвечает на эти действия активному `Android`'у событием `wait_for_fb_sleep`.

4. Активный `Android` останавливает отрисовку своего `UI`, начинает ждать события `wait_for_fb_wake`, выполняет другие действия по переводу устройства в режим пониженного энергопотребления.

5. Активный `Android` получает статус неактивного.

6. Неактивный `Android`, на который пользователь выполнил переключение получает статус активного.

7. Для него эмулируется событие нажатия кнопки `Power`.

8. Активный `Android` включает подсветку экрана, записывает в файл `/sys/power/state` значение "on", выполняет другие действия по переводу устройства в нормальный режим энергопотребления.

9. Ядро доставляет ему событие `wait_for_fb_wake`,

10. Активный `Android` начинает отрисовку своего `UI` на экране и ждет события `wait_for_fb_s`

Приведем код с комментариями, который координирует описанный сценарий:

```
static void manage_power_on_fg_change(struct foreground_changed_event_params*
    params)
{
  unsigned int hphone_suspended;
  vphone_t* old_phone;
  vphone_t* new_phone;

  old_phone = andcont_find_phone_by_id(andcont_get_vphone_list(),
      id_to_vcid(params->old_id));
  new_phone = andcont_find_phone_by_id(andcont_get_vphone_list(),
      id_to_vcid(params->new_id));

  if (old_phone == new_phone) {
    LOG_INFO("Old and new phones are the same");
    //simply make screen shine and return
    if (new_phone->suspended) {
      LOG_INFO("Making foreground phone awake by power button");
      input_event(vinput_device, EV_KEY, KEY_POWER, 1);
      input_event(vinput_device, EV_KEY, KEY_POWER, 0);
    }
    return;
  }
  //Suspend old vphone by power button emulation
  //old fg_vc can be NULL if we do switch for the first time
  //We put msleeps to allow Android to process events and make this
  //code "synchronous"
```

```
26    if (old_phone &&
          !old_phone->suspended) {
28      LOG_INFO("Making old suspended by power button");
        input_event(vinput_device, EV_KEY, KEY_POWER, 1);
30      input_event(vinput_device, EV_KEY, KEY_POWER, 0);
        msleep(5);
32      old_phone->suspended = 1;
    }
34  //work in context of new
    supervisor->fg_vc = id_to_vcid(params->new_id);
36  early_suspend_wake_queue(); //give old fb_sleep event
    msleep(500); //wait for old container to perform all jobs
38  LOG_INFO("Making new awake by power button");
    input_event(vinput_device, EV_KEY, KEY_POWER, 1);
40  input_event(vinput_device, EV_KEY, KEY_POWER, 0);

42  new_phone->suspended = 0;
    early_suspend_wake_queue(); //give new fb_wake event
44  LOG_INFO("Waiting all power events to be processed");
    msleep(100); //waiting all power events to be processed
46 }
```

3.4.3 Анализ полученного решения

Решение по виртуализации управления питанием позволило:

- Избавиться от рендеринга `UI` неактивных `Android`'ов тем самым экономя заряд батареи и вычислительные ресурсы.
- Избежать использования более сложных решений по разделению доступа `Android`'ов к экрану устройства.
- Избавиться от неожиданных отключений экрана и лишних мельканий при переключении.

4 Тестирование и анализ результатов

4.1 Цели тестирования

Целью тестирование является выявить изменения в поведении виртуализированных устройств или подтвердить их отсутствие с точки зрения пользователя и Android'ов. С точки зрения пользователя, виртуализированные устройства не должны вносить заметных задержек или артефактов в наблюдаемую пользователем информацию (звук, картинку на экране). С точки зрения Android'ов, виртуализированные устройства должны себя вести также как физические.

4.2 Сценарии тестирования

Тестирование производилось на двух и более одновременно работающих Android'ах. Для тестирования виртуализированного аудио устройства в одном Android'е проигрывалась музыка, в то время как в другом запускались игры, проигрывание музыки или совершались телефонные звонки.

Для тестирования виртуализированного экрана проверялся сам факт одновременной работы нескольких Android'ов и возможность переключения активного Android'а. Возможность использования GPU каждым Android'ом проверялась посредством запуска в них 3D игр.

Изменения в поведении устройств с точки зрения пользователя оценивались при помощи наблюдения картинки на экране и прослушивания звука. Изменения в поведении устройств с точки зрения Android'ов проверялись на основе записи в глобальном логе Android, в который записываются предупреждения и ошибки.

4.3 Результаты тестирования

В глобальном логе Android не встречаются новые записи об ошибках или предупреждениях, которых не было при использовании не виртуализированных устройств.

С точки зрения пользователя не замечено задержек или артефактов при проигрывании звука. Отзывчивость игр и пользовательского интерфейса также субъективно не изменилась.

Иногда во время переключения активного Android'а возникают критические ошибки в ядре ОС.

4.4 Анализ результатов

Редкое проявление критических ошибок при переключении активного Android'а позволяет выдвинуть предположение о возникновении условия гонок. В целом, процесс переключения активного Android'а является переходным процессом, во время которого желательно останавливать любые изменения в виртуальных устройствах. Для этого можно было бы ввести глобальный для всех виртуальных устройств семафор типа “read-write”,

который будет захватываться на чтение в каждой функции-интерфейсе к виртуальным устройствам, а во время переключения активного Android'а захватываться на запись. Таким образом во время переключения все внешние вызовы в виртуальные устройства будут отложены до окончания переключения. В данной работе этот механизм не был реализован.

С точки зрения производительности виртуализация фреймбуфера и GPU практически не вносит дополнительных затрат вычислительных ресурсов, т.к. операция записи кадров в память выполняется обычным способом. Кроме того рендеринг графики неактивных Android'ов останавливается, и активный Android практически использует экран и GPU монопольно. Вероятно, поэтому пользователь не замечает изменений в отзывчивости графики.

Передача звукового потока Android'ов микшеру и само микширование требуют, вероятно, заметного количества вычислительных ресурсов. Для оценки влияния этих операций на энергозатраты и производительность требуется выполнить сравнение загрузки CPU и времени работы устройства с и без виртуализации аудио устройства. В данной работе этого проделано не было. Не смотря на то, что виртуализированное проигрывание звука явно требует больше вычислительных ресурсов и изменяет распределение случайных задержек при доставке звука к физическому аудио устройству, отсутствие артефактов можно объяснить использованием механизмов буферизации, которые эффективно сглаживают случайные задержки.

5 Заключение

Данная работа продемонстрировала примеры виртуализации устройств для ОС `Android` на уровне библиотек в пространстве пользователя и на уровне ядра ОС. Полученные результаты являются основой для выработки подходов по виртуализации других устройств.

6 Библиография

[1] Jeremy Andrus, Christoffer Dall, Alexander Van't Hof, Oren Laadan, and Jason Nieh. Cells: A virtual mobile smartphone architecture. Technical report, Department of Computer Science Columbia University, 2011.

[2] Ken Barr, Prashanth Bungale, Stephen Deasy, Viktor Gyuris, Perry Hung, Craig Newell, Harvey Tuch, and Bruno Zoppis. The vmware mobile virtualization platform: is that a hypervisor in your pocket? Technical report, VMware, Inc, 2010.

[3] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Stephan Heuser, Ahmad-Reza Sadeghi, and Bhargava Shastry. Practical and lightweight domain isolation on android. Technical report, Technische Universitat Darmsdtadt, 2011.

[4] Jason Fitzpatrick. An interview with steve furber. *Communications of the ACM*, 54(5):34–39, 2011.

[5] Gernot Heiser. Virtualizing embedded systems — why bother. In *DAC'11 Proceedings of the 48th Design Automation Conference*. NICTA and University of New South Wales, ACM, 2011.

[6] Joo-Young Hwang, Sang Bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim. Xen on arm: System virtualization using xen hypervisor for arm-based secure mobile phones. Technical report, Software Laboratories, Corporate Technology Operations, Samsung Eletronics Co. Ltd, 2008.

[7] Jae-Ho Lee, Yeung-Ho Kim, and Sun ja Kim. Design and implementation of a linux phone emulator supporting automated application testing. In *Third 2008 International Conference on Covergence and Hybrid Information Technology*. Electronics and Telecommunications Research Institute, IEEE, 2008.