Технология контейнерной виртуализации для платформы Android

Virtualization is widely used in desktop and server systems and in several years it will come into the mobile world. The paper suggests an approach for the container virtualization for the *Android operating system. Analogues (like Cells, VMware Horizon Mobile, TrustDroid, EmbeddedXen) are described and their advantages and disadvantages are considered. The approach suggested by the authors is based on the Linux containers (LXC) mechanism. It provides the virtualization of process identifiers, network resources and can also be used for resource management based on cgroups. A new supervisor (AndCont) has been developed to manage a multi-container environment. The suggested solution also includes a modified Android Binder driver for inter-process communications, components for multiplexing user input, a GPU and frame buffer virtualization scheme, a proxy-based solution for handling incoming and outcoming messages and phone calls. The notion of the active container is introduced to describe the Android OS instance that holds user input queues and is able to output a graphical content to the physical screen. Other (inactive) Android instances output into virtual buffers that are not visible to the user and they are able to run applications in the background. A special proxy-based layer has been developed for telephony virtualization. It includes a wrapper around native (proprietary) modules for the radio interface management and a set of rules that define the call routing scheme. Sound can be mixed from several applications running in different containers but during a call the audio tract is monopolized by telephony. The paper suggests two solutions for the power management: wake_lock() and wake_unlock() virtualization to prevent a container from turning a smartphone into the low-power state; wait_for_fb_sleep event emulation to prevent an inactive Android from rendering to the screen. Most fixes in regular Android drivers (e.g. Alarm, Binder, evdev, etc.) have been made by adding a separate state context for each container and device-wide event and data multiplexor. The performance testing strategy and scenarios are described, results are discussed. Adding containers has low impact on battery power consumption but the approach should be improved to reach better memory usage. It's possible to run two Androids with traditional applications (games, players) but the set of devices is limited. Performance tuning will be continued by the authors.*

Mobile virtualization; Android; linux containers; LXC; AndCont

***Аннотация* — Виртуализация широко применяется для настольных и серверных решений и начинает завоевывать область мобильных устройств. В статье рассматривается реализация контейнерной виртуализации для устройств на базе Android. Рассмотрены аналоги (Cells, VmWare Horizon Mobile, TrustDroid, EmbeddedXen). Для построения контейнеров применен механизм linux containers (LXC). Предлагается оригинальный подход к виртуализации телефонии, аудио- и видео- устройств. Предлагаются политики для маршрутизации входящих и исходящих вызовов, а также схема мультиплексирования звука. Вводится понятие активного контейнера Android, который принимает весь пользовательский ввод, осуществляет вывод графики на видимую область экрана, принимает входящие и отправляет исходящие сообщения и звонки. Неактивные контейнеры выполняют приложения в фоне, звук может быть микширован. Пользователю предоставляется интерфейс переключения активного контейнера. Описывается стратегия тестирования производительности и проводится анализ потребляемых ресурсов.**

Виртуализация мобильных устройств; Android; контейнерная виртуализация в linux; LXC; AndCont

I. ВВЕДЕНИЕ

В последние несколько лет наблюдается заметное движение технологий виртуализации от персональных компьютеров и серверов в сторону мобильных устройств. В отчете агентства IDC за 2011[1] год отмечается ожидаемый рост использования мобильной виртуализации в ближайшие три года. Выделяется несколько направлений, по которым будет происходить развитие: виртуализация мобильных клиентов и виртуализация мобильных устройств. Данная статья посвящена технологии виртуализации мобильных устройств на платформе Android.

Виртуализация мобильных устройств подразумевает несколько основных сценариев исппользования: создание персонального виртуального устройства (концепция Bring Your Own Device); использование изолированного окружения для ненадежного программного обеспечения или наоборот изоляция критических компонентов (таких как биллинг, сервисы мобильного оператора и пр.); упрощение обновления и отладки.

A. Bring Your Own Device

Концепция Bring Your Own Device (или BYOD) подразумевает возможность использования персональных мобильных телефонов сотрудников для доступа к корпоративным ресурсам (почта, календарь, различные средства автоматизации). Компании могут добиться существенного снижения расходов на покупку техники, переведя корпоративные приложения на мобильные устройства сотрудников. С другой стороны, отсутствие контроля мобильного устройства сотрудника, на котором установлено корпоративное ПО, влечет риски утечки данных компании как по вине пользователя устройства, так и по вине злоумышленников.

Таким образом, возникает конфликт интересов компании и сотрудника. Подходящим решением проблемы безопасности становится технология виртуализации,

[1] May 2011, IDC #228088, Volume: 1 Mobile Enterprise Software: Technology Assessment

позволяющая запускать несколько пользовательских окружений на одном устройстве. Значительную работу в этом направлении проделали компании Vmware [1] и CellRox.

B. Создание изолированного окружения («песочницы»)

Известно множество случаев, когда приложения из официальных магазинов приложений проявляли вирусную активность: собирали личные данные, отправляли платные SMS и пр. Виртуализация является признанным лидером для локализации такого программного обеспечения (ПО) и позволяет решить проблему мобильных вирусов и «троянских коней». Для этого пользователю достаточно создать специальное окружение с ограниченными ресурсами, – карантинную зону, в которой возможен безопасный запуск непроверенных приложений. В этом окружении должны отсутствовать личные данные. Порча этого окружения не должна повлечь порчу или утечку пользовательских данных.

C. Защита критических компонентов системы

Зачастую мобильные компьютеры совмещают в себе функции устройства управления какой-либо системой в реальном времени (RT) и другие функции, не являющиеся RT. Примером может служить бортовой компьютер современного автомобиля, на котором запущено сразу две ОС: небольшая ОС реального времени, обеспечивающая управление узлами и агрегатами автомобиля и ОС предназначенная для реализации пользовательских функций, таких как прием звонков, проигрывание музыки, навигация. Изоляция компонентов таких систем необходима для обеспечения безопасности и виртуализация активно используется в этой сфере [5].

D. Виртуализация для целей разработки

Виртуализированная платформа может предоставлять дополнительные возможности для тестирования приложений, выполняющихся в ней, и упрощения отладки и обновления ПО. Например, в виртуальном окружении возможно генерировать события устройств ввода/вывода по сценарию, что позволяет построить инструменты автоматизированного тестирования [7]. Также виртуализация может использоваться для обновления программного обеспечения на мобильных устройствах (такого как прошивки, системные библиотеки, прикладное ПО) без риска потери данных и поломки телефона.

II. Аналогичные проекты

Первые известные попытки виртуализации мобильных устройств были начаты в 2008 году и к настоящему моменту имеется несколько успешных подходов и проектов, которые рассматриваются в данном разделе.

A. Cells

Cells [1] является первым проектом, в рамках которого была создана технология контейнерной виртуализации для платформы Android. Технология обеспечивает совместный доступ одновременно работающих окружений Android к устройствам ввода, графическому ускорителю, телефонии и сетевым устройствам, а также позволяет ограничивать доступ контейнера к каждому системному устройству. Измерения показывают, что требования этой технологии к вычислительным ресурсам, а также расход заряда батареи устройства компонетами этой технологии незначительны.

B. VMware Horizon Mobile

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, и является процессом в основной (host) ОС (например, в Android, поставляемый вместе с устройством). Очевидно, что в данном случае виртуализация всего аппаратного обеспечения требует достаточно много вычислительных ресурсов, поэтому запуск более чем одного гипервизора нецелесообразен [2].

Подход VMware выглядит более сложным для реализации, чем подход Cells.

C. TrustDroid

Проект TrustDroid является прототипом системы изоляции приложений для платформы Android, основанной на доменах доверия. 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. Такое решение не использует виртуализацию аппаратного обеспечения или пространства пользователя. Для поддержки политики доменов изменения вносятся в ядро и в стандартные системные приложения Android.

Данная система является самой нетребовательной к ресурсам по сравнению с приведенными выше. Реализация сценариев использования BYOD и «создание песочницы» различны: в случае Cells и VMware 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.

Также TrustDroid имеет недостатки с точки зрения безопасности: предполагается, что стандартные системные приложения Android и ядро ОС никогда не скомпрометированы, но это может быть неверно, т.к. при получении прав суперпользователя их можно подменить. Эти недостатки, можно преодолеть, значительно доработав TrustDroid [3]. Тем не менее, TrustDroid решает только задачу обеспечения безопасности на устройстве, в то время как пользователю также важна возможность изоляции данных, находящиеся в разных окружениях.

D. EmbeddedXEN

Проект по портированию инфраструктуры XEN на платформу ARM, на момент написания статьи, находится в

стадии активной разработки и не может рассматриваться как законченное решение. В настоящее время в проекте уже адаптированы версии ядер Linux (с патчами проекта Android для машины Goldfish) и HTC Desire HD для работы в качестве Dom0[2].

III. Архитектура

В данной статье рассматривается технология контейнерной виртуализации для мобильных устройств на платформе Android, разработанная авторами. В основе решения (общая архитектура которого приведена на рис. 1) лежит набор утилит LXC (Linux Containers) являющийся готовым интерфейсом к инфраструктуре пространств имен ядра Linux для изоляции идентификаторов процессов, виртуализации сети, а также к инфраструктуре управления ресурсами cgroup. Решено использовать модель с одним активным Android, что означает: отрисовка на экране, а также получение ввода от пользователя принадлежат одному контейнеру, который называется *активным*.

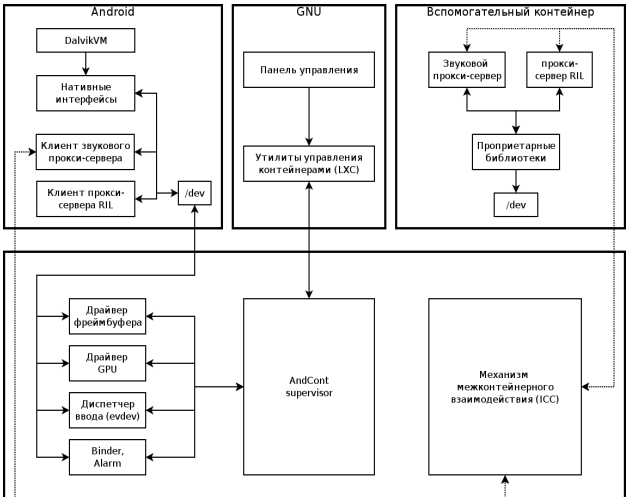

Рис.1

Кратко опишем назначение каждого компонента:

- *Панель управления* — графический интерфейс к LXC; позволяет управлять контейнерами (запуск, останов, переключение активного контейнера).
- *Супервизор ядра* (на рисунке AndCont supervisor) — инициализирует виртуальное состояние драйверов, перехватывая момент запуска контейнера, реализует переключение между контейнерами.
- *Механизм межконтейнерного взаимодействия* (*ICC*) — средство коммуникации между контейнерами на основе Netlink, обходящее проверки в прикладных сетевых протоколах, и на основе разделяемой памяти.
- *Диспетчер ввода* обеспечивает совместное использование устройств ввода Android, работающими параллельно.
- *Драйвер виртуального фреймбуфера* обеспечивает возможность одновременного использования экрана устройства.
- *Драйвер GPU* — обеспечивает возможность одновременного использования GPU всеми контейнерами.
- *binder* — драйвер IPC, реализованный в рамках проекта Android Open Source Project (AOSP).[3]
- *alarm* — интерфейс к часам реального времени.
- *Прокси-сервер механизма управления радиоинтерфейсами (Radio Interface Layer, RIL)* — обеспечивает совместный доступ нескольких Android к оборудованию доступа к мобильным сетям.
- *Прокси-клиент RIL* принимает запросы к сервису телефонии от приложений в контейнере и передает их прокси-серверу RIL.
- *Прокси-клиент аудио* получает контроль над звуковыми потоками контейнера и передает их прокси-серверу аудио.
- *Прокси-сервер аудио* воспроизводит звуковые потоки контейнеров.
- *Вспомогательный контейнер* предназначен для запуска системных демонов и библиотек Android, по разным причинам вынесенных из гостевых Android.

A. Механизм управления контейнерами

Для обечпечения работоспособности LXC потребовались следующие доработки:

- Блокировка сброса флага CAP_SYS_BOOT, отвечающего за возможность перезагрузки системы, который используется виртульной java-машиной Dalvik при инициализации.
- Устранение проверок открытых файловых дескрипторов, отличных от стандартных в lxc_start.
- Добавление нотификаций супервизора уровня ядра о запуске нового контейнера перед запуском процесса init.

Далее подробно рассматриваются стандартные компоненты системы Android, которые потребовали доработок.

B. Binder

Драйвер ядра binder — это механизм IPC, разработанный в рамках проекта AOSP. Наличие оригинального, отличного от стандартного IPC до конца не ясно, однако детали реализации binder были одним из препятствий для запуска нескольких виртуальных окружений Android. Драйвер содержит две статические переменные binder_context_mgr_node и binder_context_mgr_uid, значения которых можно установить системным вызовом ioctl(BINDER_SET_CONTEXT_MGR). Этот системный вызов выполняет программа servicemanager, запускаемая в процессе инициализации Android.

Драйвер binder проверяет, инициализированы ли эти переменные, и если да, то возвращает ошибку, что приводит к завершению программы servicemanager и,

[2] http://sourceforge.net/projects/embeddedxen

[3] http://source.android.com

поскольку эта программа необходима для работы Android, к прекращению инициализации Android.

Для устранения нежелательного завершения Android потребовалось создавать экземпляры упомянутых выше переменных для каждого контейнера и обращаться к экземпляру виртуального состояния, принадлежащему контейнеру, в контексте которого был сделан системный вызов ioctl().

C. Alarm

Alarm — драйвер, позволяющий перевести процесс в состояние ожидания, а также реализующий доступ к часам реального времени. Его интерфейс экспортируется через ioctl-вызовы устройства /dev/alarm.

При запуске второго виртуального окружения Android некоторые ioctl-вызовы этого драйвера возвращаются ошибки, вызванные наличием следующих статических переменных: alarm_opened, alarm_pending, alarm_enabled, wait_pending, alarms[ANDROID_ALARM_TYPE_COUNT].

Решение этой проблемы аналогично решению для драйвера binder, -- были созданы копии переменных для каждого контейнера.

D. Подсистема ввода

Android использует подсистему evdev для сбора событий ввода ядра.

Для каждого процесса, открывшего файл /dev/input/eventX, этот драйвер создает очередь событий ввода, поступающих из ядра подсистемы ввода. Таким образом, поступающие события ввода могут быть доставлены во все контейнеры. Функция evdev_event() является диспетчером событий ввода, поэтому для того чтобы предотвратить доставку сообщений ввода в неактивные контейнеры, эта функция была модифицирована так, чтобы все поступающие события ввода добавлялись только в очереди процессов активного контейнера.

E. Телефония

Программный стек управления оборудованием для доступа к мобильным сетям на Android содержит следующие компоненты:

- API пакета com.android.internal.telephony;
- демон rild, мультиплексирующий запросы пользовательских приложений в аппаратуре доступа к мобильным сетям;
- проприетарная библиотека доступа к оборудованию;
- драйвер GSM-модема.

При одновременном запуске нескольких виртуальных окружений Android возникают следующие проблемы:

- проприетарные реализации RIL пытаются повторно инициализацилизировать оборудование;
- не определен способ маршрутизации входящих звонки и SMS;
- не определен механизм управления исходящими звонками и SMS.

Здесь также необходимо отметить, что реализация драйверов GSM-модема существенно различается на разных устройствах, поэтому существование универсального решения виртуализации этого устройства в ядре невозможно. Кроме того, проприетарная библиотека RIL использует недокументированный протокол для взаимодействия с GSM-модемом, что является препятствием для его виртуализации даже для конкретной модели смартфона. Потребовалось разработать механизм абстракции доступа к оборудованию и модему (описан в разделе V).

F. WiFi

Некоторые приложения (например, AndroidMarket) требуют активного WiFi-подключения для нормального функционирования. Однако, по следующим причинам, невозможно перенести в контейнер сетевой интерфейс, соответствующий физическому WiFi-адаптеру:

- все контейнеры не могут использовать корневое сетевое пространство имен, поскольку это приводило бы к конфликтам за порты TCP и имена абстрактных Unix-сокетов;
- в случае, когда каждый контейнер работает в своем сетевом пространстве имен, беспроводной интерфейс должен быть перемещен из корневого сетевого пространства имен в пространство имен первого запущенного контейнера, т.е. перемещение этого интерфейса в другой контейнер выполнить уже невозможно.

Менеджер WiFi в Android использует библиотеку libhardware_legacy.so для управления беспроводным интерфейсом. Интересующий нас интерфейс представляет собой обертку над интерфейсом демона wpa_supplicant и предназначен для передачи текстовых команд этому демону. Таким образом, для виртуализации WiFi достаточно возвращать корректные ответы на запросы менеджера WiFi (запросы для получении информации о состоянии беспроводного соединения и списка доступных точек доступа), — в этом случае не требуется запуск wpa_supplicant и может быть использована описанная ранее схема виртуализации (см. рис. 2) сети с помощью драйвера veth: для каждого контейнера создается пара соединенных виртуальных Ethernet-адаптеров, один из которых перемещается в сетевое пространство имен контейнера, а другой подключается к мосту.

G. Wake-блокировки

Wake-блокировка — объект ядра, разработанный в рамках проекта AOSP и являющийся частью механизма управления энергопотреблением. В ядре существует очередь запросов на переход в состояние пониженного энергопотребления (workqueue suspend), которая активируется при снятии wake-блокировки функцией wake_unlock(), либо при истечении ее таймаута.

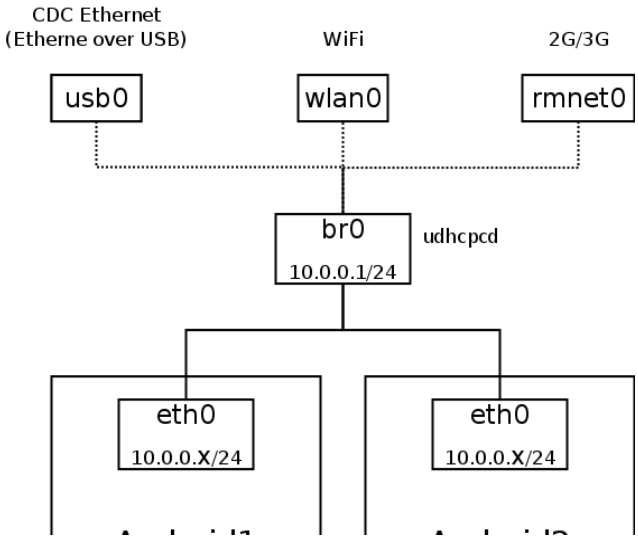

Рис. 2

Ядро экспортирует интерфейс управления wake-блокировками через файлы power/wake_lock и power/wake_unlock файловой системы sysfs. Запись строки S в первый из них приводит к созданию новой wake-блокировки с именем S, если ее не существует, и ее захвату. Аналогично, запись строки S в файл power/wake_unlock приведет к освобождению wake-блокировки с именем S.

Таким образом, несколько одновременно запущенных виртуальных окружений Android будут пытаться захватывать и отпускать одни и те же wake-блокировки, что не является ожидаемым поведением: для каждого виртуального окружения должен быть свой набор именованных wake-блокировок.

IV. Виртуализация периферийных устройств

Большой задачей виртуализации Android является виртуализация переферийных устройств и мультиплексирование потоков данных. В данном разделе рассмотрен предлагаемый подход.

A. Виртуализация устройств

Android предполагает, что использует периферийные устройства монопольно. Многие драйверы в своей реализации опираются на это предположение. Из-за этого запуск нескольких Android на одном устройстве приводит к ошибкам в периферийных устройствах их драйверах, связанным с конкурентным доступом к данным. Для работы нескольких Android в контейнерах требуется обеспечить возможность одновременого использования периферийных устройств и обеспечить невозможность Android использовать периферийные устройства произвольно.

Предлагаемый метод виртуализации устройств Android опирается на следующие требования:

- Физическое устройство подменяется на виртуальное, благодаря чему все запросы Android к физическому устройству направляются в виртуальное устройство.
- Виртуальное устройство выполняет часть запросов Android на физическом устройстве, а часть виртуально, например, изменяя виртуальное состояние в соответствии со своей политикой.
- Виртуальное устройство имеет тот же интерфейс и поведение, что и физическое.

Дополнительными требованиями к реализации интерфейсов виртуальных устройств являются:

- Интерфейс должен быть максимально совместимым со всеми доступными версиями Android;
- Интерфейс должен быть единственным и использоваться для всех задач доступа к устройству.

Далее будет рассмотрена виртуализация нескольких периферийных устройств выполненная авторами.

B. Экран

Необходимость в разделении доступа контейнеров к экрану очевидна: Каждый Android отображает свое содержимое, что в случае нескольких контейнеров приводит к смешению изображений. Для виртуализации экрана физическое устройство подменяется виртуальным. Подходящим интерфейсом к экрану устройства является драйвер Linux framebuffer (согласно Android porting guide). Он имеет простой интерфейс и используется из пространства пользователя.

Кроме драйвера фреймбуфера к физическому экрану имеет доступ GPU.

1) Перепрограммирование MMU GPU

Для того чтобы неактивные Android не отображали свой UI на экране, требуется подменить операцию отображения их кадров на экране. В случае с Google Nexus S, Samsung Galaxy SII и Goldfish (на которых проводилось тестирование) экраны смартфонов копируют кадр для отображения из RAM при помощи DMA. Драйвер фреймбуфера сообщает экрану адрес, по которому находится кадр, предназначенный для отображения (память экрана). Таким образом, операция отображения кадра на экране выглядит как операция записи кадра в RAM устройства по определенному адресу.

Чтобы подменить операцию записи в RAM, требуется установить, кем она может выполняться. В случае с обработкой графики запись кадра могут выполнять GPU или CPU. Современные GPU и CPU имеют в своем составе MMU. Для того чтобы кадры неактивных Android не записывались в память экрана, достаточно подменить отображение адресов в MMU так, чтобы адреса неактивных Android, отображаемые в память экрана, фактически указывали на обычный буфер в RAM устройства (теневой буфер).

2) Теневой фреймбуфер

Android отрисовывает свой UI по мере его изменения, а не, например, периодически по таймеру. Поэтому если просто перенаправить MMU на память экрана при переключении активного Android, то активный Android

может не перерисовать кадр на экране и изображение на нем окажется от предыдущего активного Android.

Если выделить теневой буфер для каждого Android, то при переключении активного Android достаточно скопировать кадр на экране в теневой буфер предыдущего активного Android. Чтобы отобразить Android, ставший активным, требуется скопировать содержимое его теневого буфера в память экрана. Но теневой буфер занимает от 2 до 4 Мб физической памяти. Имея возможность вызывать перерисовку кадра при переключении активного Android, можно избавиться от выделения теневого буфера для каждого Android, оставив единственный теневой буфер.

Указанная возможность предоставляется механизмом fb_early_suspend. Теневой буфер теперь становится не местом хранения кадра Android, а местом, куда все неактивные Android пишут свою картинку. Кроме того, благодаря наличию MMU можно сократить размер теневого буфера до одной страницы физической памяти отобразив все 2−4 Мб адресов в единственную страницу.

3) Виртуальный драйвер Linux framebuffer

Типичный сценарий использования драйвера Linux framebuffer заключается в отображении памяти экрана в адресное пространство пользовательского процесса системным вызовом mmap() и вызовах стандартных и специальных IOCTL. Так как физический Linux framebuffer был подменен на виртуальный, то mmap() и IOCTL вызываются у драйвера виртуального Linux framebuffer.

Все вызовы mmap() обрабатываются полностью в драйвере виртуального Linux framebuffer без использования драйвера физического Linux framebuffer. Процессы активного Android при вызове mmap получают доступ к памяти экрана. Процессы неактивных Android при вызове mmap() получают доступ к теневому буферу. Для всех запущенных Android создается виртуальное состояние драйвера Linux framebuffer. Для активного Android все вызовы в драйвер виртуального Linux framebuffer немедленно перенаправляются в драйвер физического Linux framebuffer, который изменяет состояние физического драйвера Linux framebuffer. Для неактивных Android стандартные IOCTL изменяют их виртуальные состояния Linux framebuffer. Нестандартные IOCTL'ы для неактивных Android возвращают ошибку.

4) Переносимость

Подмена стандартного для всех Android устройств интерфейса доступа к экрану Linux framebuffer и использование в качестве бэкенда драйвера физического Linux framebuffer со стандартным интерфейсом, делает решение хорошо переносимым. Необходимость в перепрограммировании MMU GPU и доступе к информации об отображении адресов требует определенного вмешательства в драйверы GPU. Задача упрощается тем, что, как правило, работа с MMU локализована в определенном наборе функций драйвера GPU, в которые можно внедрять функции обратного вызова для сбора всей необходимой информации. Работа с MMU GPU по сравнению с частотой отображения кадров на экране выполняется редко, поэтому добавленные функции не вносят ощутимых накладных расходов.

C. Управление питанием

Описанное выше решение по виртуализации экрана обладает значительным недостатком: хотя неактивные Android и не видны на экране, их UI продолжает отрисовываться и в них продолжают работать многие невидимые пользователю приложения. Устройство выполняет бесполезную вычислительную работу, тратится заряд его батареи.

Данный недостаток удалось преодолеть благодаря изменениям в подсистеме управления питанием ядра ОС, заключающимся в эмуляции событий wait_for_fb_sleep и события нажатия кнопки Power. Благодаря их генерации неактвиный Android переходит в режим ожидания, как это происходит на обычных телефонах с Android при нажатии кнопки Power. В режиме ожидания Android останавливает отрисовку UI и приостанавливает многие пользовательские приложения, что позволяет избавиться от приведенного недостатка.

Для Android, становящегося активным, требуется генерация событий, побуждающих завершение режима ожидания. Для этого для него генерируется событие wait_for_fb_wake и событие нажатия кнопки Power. Эти события также заставляют Android перерисовать свой кадр, в результате чего на экране будет изображен кадр ставшего активным Android.

D. GPU

Виртуализированное GPU должно обеспечить возможность использования GPU всеми запущенными Android.

OpenGL — единственный API, используемый Android для отрисовки графики. Он предоставляет каждому приложению графический контекст, благодаря чему приложения не замечают работы друг друга с GPU. Таким образом, в основном GPU уже виртуализировано.

Эксперименты на смартфонах Google Nexus S (GPU PowerVR SGX 540) и Samsung Galaxy SII (GPU ARM MALI-400 MP) показали, что для успешной работы нескольких Android, использующих GPU, требуется исключить многократную инициализаци GPU каждым Android. Для этого, как правило, требуется выполнить небольшие изменения в драйверах GPU или инфраструктуре GPU в пространстве пользователя.

Также необходимо исключить попытки перевода GPU в режим пониженного энергопотребления неактивными Android. Для перевода GPU в режим пониженного энергопотребления, как правило, используется стандартный для Android механизм нотификации о событиях управления питанием early_suspend, обработчики которого для драйвера GPU нужно изменить с учетом данного сценария.

V. Виртуализация телефонии и аудио

Аудио и телефония были виртуализированы в пространстве пользователя.

Общая идея такой виртуализации заключается в реализации прокси (сервера и клиента) на уровне аппаратно-независимого интерфейса в стеке Android. Прокси-сервер размещается в отдельном контейнере с оригинальным программным стеком Android для выполнения запросов клиентов на физических устройствах. Прокси-клиент размещается в каждом контейнере с Android пользователя.

A. Аудио-устройство

Типичный сценарий использования аудио-устройства при запуске нескольких Android, который был использован, — прослушивание музыки в одном контейнере и воспроизведение звука игры, запущенной в другом. Таким образом, пользователю, как правило, хотелось бы слышать звук, всех запущенных Android, но фактически физическое звуковое устройство не расчитано на это. Без модификаций Android сообщают об ошибке при попытке его конкурентного использования и иногда не могут воспроизводить звук до следующей перезагрузки. Виртуализация аудио устройства обеспечивает одновременное прослушивание звука всех Android. Вторым сценарием использования звукового устройства является взаимодействие с телефонией. Во время разговора по телефону аудио устройство должно записывать и передавать в сотовую сеть голос пользователя и воспроизводить получаемый обратно голос собеседника.

a) Прокси-клиент

Подменяемым аппаратно-независимым интерфейсом к звуковому устройству является интерфейс AudioHardwareInterface. Android Platform Developer's Guide[4] говорит о том, что он аппаратно независим. Кроме того он является самым низкоуровневым и простым аппаратно-независимым аудио интерфейсом. По сути, он представляет из себя поток для записи звука на воспроизведение. Кроме звукового потока, воспроизводимого Android, в его реализации можно получить и другую полезную информацию.

Можно было бы выполнить подмену на более высоком уровне, например, подменить часть аудио фреймворка Android, но анализ показал, что аудио фреймворк значительно сложнее AudioHardwareInterface. В нашем случае прокси-клиент является реализацией AudioHardwareInterface. Его основная обязанность — отправлять звуковой поток Android и другую полезную информацию прокси-серверу, который будет воспроизводить звуковые потоки всех Android. Звуковые потоки воспроизводятся в несжатом виде, поэтому их передача с использованием копирования в прокси-сервер может давать заметные накладные расходы. В качетве транспорта используется межконтейнерная разделяемая память. Информационные и управляющие сообщения передаются с использованием механизма межконтейнерной передачи сообщений.

2) Прокси-сервер

Основная задача прокси-сервера — микшировать получаемые от прокси-клиентов звуковые потоки и воспроизводить результирующий звуковой поток на физическом аудио устройстве. Чтобы обеспечить максимальную переносимость прокси-сервера, микширование и воспроизведение звука должны осуществляться независимо от устройства. Интерфейсы аудио фреймворка Android являются аппаратно-независимыми. Кроме того, в аудио фреймворке уже заложены возможности по микшированию звуковых потоков. Аудио фреймворк Android не предоставляет доступа к своему микшеру извне, поэтому для его использования требуется либо внедриться в аудио фреймворк, либо использовать микшер неявно.

Самым простым решением, которое позволяет микшировать и воспроизводить звук в Android, является использование аудио классов из Android SDK. В нашем решении звуковой поток каждого Android пользователя воспроизводится отдельным объектом класса AudioTrack из Android SDK. Благодаря использованию классов из Android SDK, аудио фреймворк Android микширует все воспроизводимые звуковые потоки и отправляет результирующий звуковой поток на воспроизведение физическим аудио устрйоством.

Нужно отметить, что классы из Android SDK являются Java классами, и для их использования требуется запуск виртуальной Java-машины Dalvik. Запуск Dalvik в случае с Android означает запуск всего Android, который занимает 200–250 Мб RAM и потребляет вычислительные ресурсы. Это непозволительные накладные расходы для задачи воспроизведения и микшировани звуковых потоков, но в данной работе их удалось избежать путем использования нижлежащей написанной на C++ реализации.

Для того чтобы Android-сервисы смогли использовать звуковой прокси-сервер, который в силу его низкоуровневости невозможно зарегистрировать в системе, пришлось удалить ряд проверок, что не повлияло на безопасность, т.к. в контейнере с прокси-сервером никогда не запускаются посторонние программы.

3) Регулирование уровня звука

Пользователь, вероятно, хочет, чтобы громкость звука разных Android можно было регулировать. В Android 2.x отсутсвует абстракция аппаратного микшера и микширование звуковых потоков внутри каждого Android полностью независимо, следовательно общий уровень громкости каждого Android можно настроить в каждом контейнере.

4) Устройства вывода звука

Мобильные устройства, как правило, имеют множество устройств вывода звука. Например, большой динамик, трубка, наушники, bluetooth-гарнитура. Запущенные Android одновременно могут требовать воспроизведения своих звуковых потоков на разных устройствах вывода. В некоторых случаях их требования будут несовместимы. Например, не имеет смысла проигрывать один звук на большом динамике, а другой в наушниках или трубке. Также неприемлемо если во время звонка в трубке будет слышен не только голос собеседника, но и игравшая до звонка музыка. Поэтому была разработана политика роутинга звуковых потоков на различные устройства вывода, реализующая стратегию «наименьшего удивления» пользователя. В рамках данной стратегии, в

[4] http://www.kandroid.org/online-pdk/guide/audio.html

частности, иногда требуется имитировать воспроизведение звука Android, для чего прокси-клиент переводится в режим фиктивного проигрывания, в котором синхронные функции блокируются на время воспроизводимого участка звукового потока.

5) Взаимодействие с сервисом телефонии

В прокси-клиенте, являющемся реализацией AudioHardwareInterface, мы можем получить информацию о том, существует ли в текущий момент активное телефонное соединение. Если существует, то требуется настроить аудио подсистему для записи и воспроизведению голоса и взаимодействия с сервисом телефонии. Фактически для этого достаточно перевести реализацию AudioHardwareInterface, поставляемую с устройством, в режим звонка при помощи вызова метода setMode(MODE_IN_CALL) и установить роутинг звука в трубку, иначе голос собеседника будет воспроизводиться текущим устройством вывода звука. Таким образом, за реализацию взаимодействия с сервисом телефонии отвечает поставщик устройства.

6) Задержки воспроизведения звука

Дополнительный этап микширования звуковых потоков Android, очевидно, увеличивает среднее время доставки звука в буфер аудио устройства, из-за чего буфер может быть опустошен, а воспроизведение резко прервано. При тестированиии же никаких артефактов в звуке замечено не было. Вероятно, это связано с тем, что размер буфера аудио устройства на Android, как правило, вмещает от 30 до 100 мс звука, а это является огромным запасом времени для CPU, чтобы в случае нехватки вычислительных ресурсов все-таки предоставить смикшированный звуковой поток для воспроизведения.

7) Переносимость

Виртуализированное аудио устройство работает на всех Andorid версии 2.x, т.к. не зависит от аппаратного обеспечения, но зависит от интерфейсов в программном стеке Android, которые не зафиксированы и могут меняться. На Android 4.x данное решение не проверялось.

B. Телефония

1) Клиентская часть

Демон rild загружает библиотеку RIL, имя которой указано в системном свойстве rild.libpath, и вызывает функцию RIL_Init. Значения системных свойств rild.libpath и rild.libarg задаются в файле /system/build.prop, который загружает в базу данных системных свойств процесс init. Таким образом, для подмены реализации RIL достаточно изменить значение системного свойства rild.libpath.

Структура RIL_RadioFunctions описывает интефейс RIL в направлении «демон rild−проприетарная реализация RIL» и содержит следующие поля:

- void (*onRequest)(int request, void* data, size_t datalen, RIL_Token t) — обработчик запросов RIL, здесь request — тип запроса, data — данные, t — токен, идентифицирующий конкретный запрос;
- RIL_RadioState (*onStateRequest)() возвращает состояние подключения к беспроводной сети;
- int (*supports)(int requestCode) возвращает 1, если реализация RIL поддерживает запрос типа requestCode, в противном случае, 0;
- void (*onCancel)(RIL_Token t) отменяет запрос, ожидающий обработки;
- const char* (*getVersion)(void) возвращает строку-описание реализации RIL.

В функцию инициализации библиотеки RIL первым аргументом (env) передается указатель на структуру, описывающую интерфейс RIL в направлении «библиотека RIL−демон rild»:

- void (*OnRequestComplete)(RIL_Token t, RIL_Errno e, void *response, size_t responselen) уведомляет rild о завершении запроса, обработанного функцией onRequest;
- void (*OnUnsolicitedResponse)(int unsolResponse, const void *data, size_t datalen) — асинхронное уведомление демона rild;

Есть три особенности архитектуры демона rild и интерфейса RIL, которые повлияли на дизайн прокси-клиента:

1. обработчик OnRequestComplete может вызываться в любой момент;
2. обработчики onStateRequest, supports и getVersion библиотеки~RIL являются синхронными, что в контексте прокси-клиента означает необходимость дожидаться ответа от сервера;
3. обработчик OnRequestComplete может вызвать onStateRequest, supports и getVersion.

Это привело к следующему дизайну заглушки:

- процедура чтения ответов от прокси-сервера вынесена в отдельный поток; при получении сообщения эта процедура его в одну из четырех очередей в зависимости от типа ответа;
- диспетчер сообщений от прокси-сервера, вынесен в отдельный поток.

Совместно два этих решения позволили избежать самоблокировок, которые возникают при вызове синхронного запроса в контексте обработчика onRequestComplete.

2) Серверная часть

Серверная часть реализована как однопоточное приложение, которое загружает проприетарную реализацию RIL и с момощью механизма межконтейнерного взаимодействия принимает сообщения от прокси-клиентов, работающих в пользовательских контейнерах.

В обработчике OnRequestComplete нам неизвестен тип запроса, но способ упаковки ответа проприетарной реализации RIL зависит от него, поэтому при поступлении запроса сервер запоминает соответствие токен−номер запроса, чтобы в этом обработчике вызвать нужный маршаллер ответа.

Прокси-сервер RIL должен обрабатывать ситуацию, когда из разных клиентов приходят запросы с одинаковыми токенами, поэтому при поступлении

очередного запроса прокси-сервер просматривает список токенов запросов, обрабатываемых проприетарной библиотекой RIL, и если оказывается, что запрос с таким токеном сейчас обрабатывается, то сервер генерирует новый токен для поступившего запроса. Таким образом, прокси-сервер для всех поступающих запросов хранит их реальный и фиктивный токены.

3) Политика маршрутизации запросов

Для каждого типа запросов, обрабатываемых функцией OnRequest, определена политика его обработки. В настоящее время определены три политики:

- безусловное блокирование запроса — эта политика установлена для всех запросов, которые ни разу не встречались в процессе разработки прокси-сервера;
- безусловная передача запроса проприетарной библиотеке — эта политика установлена для всех запросов, не изменяющих состояние проприетарной библиотеки и GSM-модема;
- передача запроса от активного контейнера проприетарной библиотеке — эта политика установлена для запросов, связанных с отправкой SMS и совершением звонков.

Для каждого типа асинхронных уведомлений также определены политики его маршрутизации:

- безусловное блокирование уведомления;
- маршрутизация в активный контейнер — эта политика установлена для уведомлений, связанных с принятием SMS и звонков;
- маршрутизация во все контейнеры — эта политика установлена для информационных уведомлений, например, уведомлений об изменении уровня сигнала сотовой сети.

VI. Тестирование

Целью настоящего тестирования являлись

- демонстрация готовности разработанной технологии виртуализации для платформы Android,
- выявление утечек памяти в разработанных компонентах,
- определение энергопотребления разработанной технологии.

A. Методика

Тестирование проводилось на смартфоне Samsung Galaxy S II по следующим сценариям использования смартфона:

1. простой;
2. проигрывание музыки с включенной подсветкой экрана;
3. игра (Angry Birds) и одновременное проигрывние музыки.

Тестовые сценарии исполнялись на трех конфигурациях:

- оригинальное окружение CyanogenMod 7 для Samsung Galaxy S II (1),
- один контейнер с CyanogenMod 7 (2),
- два контейнера с CyanogenMod 7: в одном контейнере проигрывалась музыка, в другом запущена игра (3).

В каждом тестовом прогоне измерялись:

- объем свободной памяти и размер кеша файловой системы (по информации /proc/mempinfo),
- уровень заряда аккумулятора (по информации /sys/class/power_supply/battery/capacity).

Каждый тестовый прогон длился 30 минут, измерения делались с частотой 2 с. В таблице приведены результаты измерений.

Сценарий	**CyanogenMod 7**	**1 контейнер**	**2 контейнера**
Использование памяти (Mb)			
1	174	98	198
2	175	99	198
3	235	150	290
Использование батареи (в %/30 мин**)**			
1	0	1,2	1,2
2	4	4	4
3	8	13	14

Тестирование показало приемлемость разработанных решений и продемонстрировало незначительное потребление памяти и аккумулятора смартфона разработанной технологией за исключением случая (последняя строка таблицы), в котором наблюдается повышение энергопотребления двумя контейнерами почти в два раза по сравнению с немодифицированным Android. В настоящее время ведется работа по выяснению причин такого поведения.

Кроме того, как видно из значений потребления памяти следует что при запуске второго контейнера оно возрастает более чем в 2 раза, что является препятствием для запуска нескольких контейнеров на одном смартфоне: эксперименты показывают, что смартфоне Google Nexus S, имеющем 380 Мб, невозможен запуск более чем одного контейнера без активации файла подкачки.

VII. Модель использования

После запуска смартфона на экране отображается панель управления, представляющая собой список контейнеров, имеющихся на смартфоне. Пользователь может запустить любой из этих контейнеров и переключиться в один из запущенных контейнеров.

В каждом контейнере с Android установлено приложение, позволяющее переключиться из контейнера обратно в панель управления. Это приложение выполнено в виде кнопки на рабочем столе.

Образ контейнера в настоящее время необходимо вручную размещать на внутреннем флеш-накопителе смартфона с помощью программы adb и регистрировать с помощью утилиты lxc_create.

Источники

[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 Universit y, 2011

[2] Ken Barr, Prashanth Bungale, Stephen Deasy, Viktor Third Gyuris, Perry Hung, Craig Newell, Harvey Tuch, and Bruno Zoppis. The VMWare mobile virtualizatio n 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 Conference. NICTA and University of Automation , 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. Department of Computer Science Columbia University, 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.