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 методов контейнерной виртуализации для платформы Android»

Направление: 010600.68 — Прикладные математика и физика

Магистерская программа: «Математические и информационные технологии»

Выполнил студент Карташов А. 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 4 Гб 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, мы могли 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 нам будет интересна *виртуализация 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.
- 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 платформах

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

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

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 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 времени 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].

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

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 поддерживает ограниченный набор эмулируемых платформ на базе ARM и оборудования[4]. Медленная работа и большое энергопотребление на мобильных устройствах.

- KVM/ARM — находится в стадии разработки[5], реализован 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 технологии предназначены для использования IT-специалистами. Кроме того, поскольку большинство перечисленных технологий появились недавно, то вызывает сомнения стабильность их работы. Видимо, только LXC и QEMU можно считать достаточно надежными, т. к. они довольно легко портируемы на другие архитектуры.

[3] http://en.wikipedia.org/wiki/X86
[4] http://qemu.weilnetz.de/qemu-doc.html#ARM-System-emulator
[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

2.1 Цель

В настоящей работе мы ограничимся реализацией сценария использования «создание песочницы». Поэтому целью настоящей работы является разработка технологии виртуализации пользовательского окружения Android, позволяющей:

- создавать изолированные пользовательские окружения Android из специально подготовленных шаблонов,
- настраивать доступ к системным устройствам и высокоуровневым сервисам смартфона (телефония, звук) для каждого виртуального окружения,
- запускать и останавливать виртуальные окружения,
- переключать активное (отображаемое в текущий момент) виртуальное окружение.

Отметим также, что реализация сценария использования «создание песочницы» является необходимой для реализации «Bring Your Own Device».

2.2 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 незначительны.

- 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.

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

- TrustDroid [3]

  Является прототипом системы изоляции приложений, основанной на доменах доверия. 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 аппаратного 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 скомпрометированы, но это может быть неверно, т. к. при получении прав суперпользователя их можно подменить. Эти недостатки, 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, находящиеся в разных Android'ах, а также независимость действий, производимых в каждом из них.

- EmbeddedXEN[10]

  Проект по портированию инфраструктуры XEN на платформу ARM. В настоящее время (5 мая 2012 г.) находится в стадии активной разработки. Тем не менее, уже адаптированы версии ядер Linux (с патчами проекта Android для машин Goldfish и HTC Desire HD для работы в качестве Dom0.

Таким образом, подход проекта Cells имеет лучшее соотношение достоинств и недостатков. Поэтому был выбран подход проекта Cells — подход виртуализации уровня операционной системы.

2.3 Анализ необходимости виртуализации компонентов ОС

2.3.1 Изолированное пользовательское окружение

В системах контейнерной виртуализации традиционно изолируют следующие компоненты исполнительной среды:

1. идентификаторы процессов в разных виртуальных окружениях,
2. корень файловой системы: каждое виртуальное окружение не должно иметь доступ к файлам хостового окружения и других виртуальных окружений;
3. сеть: нужно иметь сетевой адаптер в каждом виртуальном окружении;
4. общесистемные файловые системы (sysfs, proc): необходимо ограничение доступа к некоторым частям этих файловых систем;
5. дисковые разделы: необходимо предотвратить монтирование одних и тех же разделов запоминающего устройства в разных виртуальных окружениях.

В ядре Linux первые три задачи решаются с помощью инфраструктуры пространств имен. Пространство имен — набор объектов ядра, являющийся уникальным с точки зрения пользовательского окружения.

Пятую задачу можно решить, используя подсистему cgroup для ограничения доступа к системным устройствам.

Четвертую задачу можно решить только путем модификации логики работы с этими подсистемами; в частности, в OpenVZ каждый контейнер владеет своей копией дерева kobject'ов.

В следующих разделах будут описаны специфичные для Android компоненты, требующие разработки специальных методов изоляции или совместного использования несколькими контейнерами.

2.3.2 Binder

Драйвер ядра Binder — это механизм IPC, разработанный в рамках проекта AOSP. Мотивы и цели разработки нового механизма IPC пока не ясны, но Binder был одним из первых драйверов, который оказался препятствием для запуска нескольких виртуальных окружений Android.

Проблема с драйвером Binder состоит в следующем. В следующем листинге представлен исходный код обработчкика ioctl-запросов этого драйвера:

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

```
[drivers/staging/android/binder.c]
[..]
  48 static struct binder_node *binder_context_mgr_node;
  49 static uid_t binder_context_mgr_uid = -1;
[..]
2679 static long binder_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
2680 {
2681         int ret;
[..]
2702         switch (cmd) {
[..]
2754         case BINDER_SET_CONTEXT_MGR:
2755                 if (binder_context_mgr_node != NULL) {
2756                         binder_debug(BINDER_DEBUG_TOP_ERRORS,
2757                                 "binder: BINDER_SET_CONTEXT_MGR already set\n");
2758                         ret = -EBUSY;
2759                         goto err;
2760                 }
2761                 if (binder_context_mgr_uid != -1) {
2762                         if (binder_context_mgr_uid != current->cred->euid) {
2763                                 binder_debug(BINDER_DEBUG_TOP_ERRORS,
2764                                         "binder: BINDER_SET_"
2765                                         "CONTEXT_MGR bad uid %d != %d\n",
2766                                         current->cred->euid,
2767                                         binder_context_mgr_uid);
2768                                 ret = -EPERM;
2769                                 goto err;
2770                         }
2771                 } else
2772                         binder_context_mgr_uid = current->cred->euid;
2773                 binder_context_mgr_node = binder_new_node(proc, NULL, NULL);
2774                 if (binder_context_mgr_node == NULL) {
2775                         ret = -ENOMEM;
2776                         goto err;
2777                 }
2778                 binder_context_mgr_node->local_weak_refs++;
2779                 binder_context_mgr_node->local_strong_refs++;
2780                 binder_context_mgr_node->has_strong_ref = 1;
2781                 binder_context_mgr_node->has_weak_ref = 1;
2782                 break;
```

Системный вызов `ioctl(BINDER_SET_CONTEXT_MGR)` выполняется при запуске окружения `Android`: он вызывается при запуске процесса `servicemanager`:

```
[frameworks/base/cmds/servicemanager/service_manager.c]
259 int main(int argc, char **argv)
260 {
261     struct binder_state *bs;
```

```
262     void *svcmgr = BINDER_SERVICE_MANAGER;
263
264     bs = binder_open(128*1024);
265
266     if (binder_become_context_manager(bs)) {
267         LOGE("cannot become context manager (%s)\n", strerror(errno));
268         return -1;
269     }
270
271     svcmgr_handle = svcmgr;
272     binder_loop(bs, svcmgr_handler);
273     return 0;
```

```
[frameworks/base/cmds/servicemanager/binder.c]
137 int binder_become_context_manager(struct binder_state *bs)
138 {
139     return ioctl(bs->fd, BINDER_SET_CONTEXT_MGR, 0);
140 }
```

Это означает, при запуске первого окружения переменная `binder_context_mgr_node` будет проинициализирована на строке 2773, а при запуске второго окружения проверка на строке 2755 не пройдет и будет возвращена ошибка. Эта ошибка является фатальной для окружения `Android` — процесс его инициализации прекращается, поскольку это приведет завершению процесса `servicemanager` (строки 266–269 файла `service_manager.c`), поскольку этот процесс помечен как критический в файле инициализации `Android`'а:

```
service servicemanager /system/bin/servicemanager
    user system
    critical
    onrestart restart zygote
    onrestart restart media
```

то завершение этого процесса приведет к перезапуску Java-машины.

Кроме этого, в драйвере `Binder` есть функция `binder_deferred_release`, которая обращается к однократно инициализируемому состоянию состоянию (переменной `binder_context_mgr_node`):

```
2945 static void binder_deferred_release(struct binder_proc *proc)
2946 {
2947         struct hlist_node *pos;
2948         struct binder_transaction *t;
2949         struct rb_node *n;
2950         int threads, nodes, incoming_refs, outgoing_refs, buffers,
                 active_transactions, page_count;
2951
2952         BUG_ON(proc->vma);
2953         BUG_ON(proc->files);
2954
2955         hlist_del(&proc->proc_node);
2956         if (binder_context_mgr_node && binder_context_mgr_node->proc == proc) {
2957                 binder_debug(BINDER_DEBUG_DEAD_BINDER,
2958                              "binder_release: %d context_mgr_node gone\n",
```

```
2959                                proc->pid);
2960                 binder_context_mgr_node = NULL;
2961         }
```

Особенностью этой функции является то, что она вызывается из контекста потока ядра — она вызывается из функции `binder_deferred_func`, которая периодически помещается в workqueue.

2.3.3 Подсистема ввода

`Android` использует подсистему `evdev` для сбора событий ввода ядра. Добавление события ввода в очередь, читаемую пользовательским процессом, происходит в функции `evdev_event` (`drivers/input/evdev.c`):

```
 73 static void evdev_event(struct input_handle *handle,
 74                         unsigned int type, unsigned int code, int value)
 75 {
 76         struct evdev *evdev = handle->private;
 77         struct evdev_client *client;
 78         struct input_event event;
 79         struct timespec ts;
 80
 81         ktime_get_ts(&ts);
 82         event.time.tv_sec = ts.tv_sec;
 83         event.time.tv_usec = ts.tv_nsec / NSEC_PER_USEC;
 84         event.type = type;
 85         event.code = code;
 86         event.value = value;
 87
 88         rcu_read_lock();
 89
 90         client = rcu_dereference(evdev->grab);
 91         if (client)
 92                 evdev_pass_event(client, &event);
 93         else
 94                 list_for_each_entry_rcu(client, &evdev->client_list, node)
 95                         evdev_pass_event(client, &event);
 96
 97         rcu_read_unlock();
 98
 99         wake_up_interruptible(&evdev->wait);
100 }
```

Поле `client_list` структуры `evdev` содержит список процессов, открывших устройство `/dev/input/eventX` (представленное здесь переменной `evdev`). Поскольку устройство идентифицируется по минорному номеру файла `/dev/input/eventX`, который одинаков во всех запущенных контейнерах, то потоки `InputReader` читающие события ввода из подсистемы `evdev` в параллельно работающих окружениях `Android`, будут получать одни и те же события ввода ядра.

Схема взаимодействия подсистемы ввода ядра с пространством пользователя показана на рис. 1.

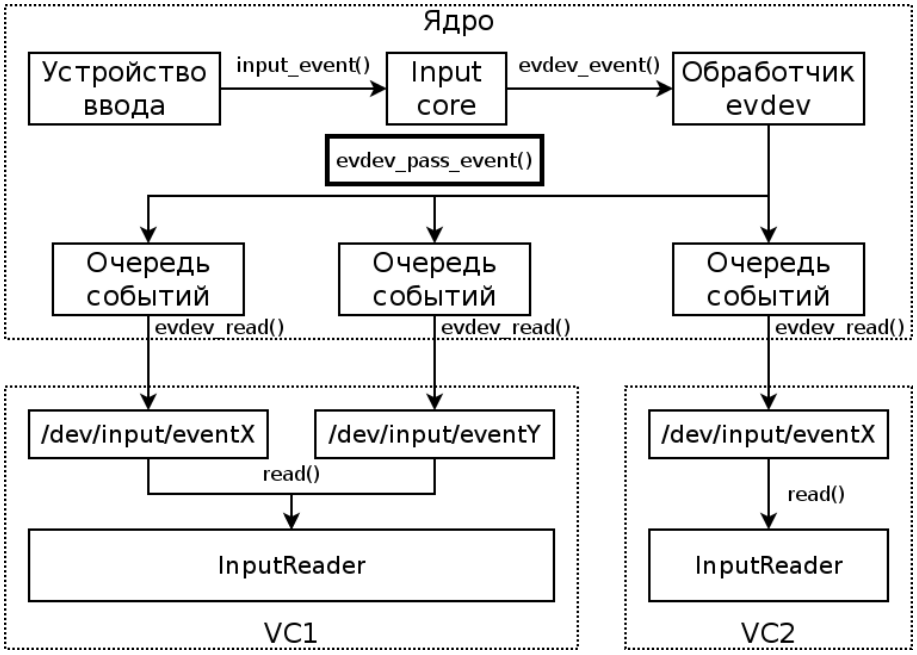

Рис. 1: Архитектура подсистемы ввода ядра

2.3.4 Телефония

Программный стек платформы `Android` управления оборудованием для доступа к мобильным сетям устроен следующим образом:

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

Интерфейс между компонентами (2) и (3) документирован и называется `RIL`[11]. При одновременном запуске нескольких виртуальных окружений `Android` необходимо:

- предотвращать повторную инициализацию оборудования,
- маршрутизировать входящие звонки и SMS,
- управлять исходящими звонками и SMS.

2.3.5 Сетевая подсистема

Некоторые приложения, в частности, Android Market, требуют активного беспроводного соединения[12], поэтому в каждом виртуальном окружении должен присутствовать виртуальный беспроводной интерфейс, который бы предотвращал попытки виртуального окружения конфигурировать реальный беспроводной сетевой интерфейс, а также предоставлял доступ к физической беспроводной сети.

2.3.6 Alarm

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

[11] `development/pdk/docs/porting/telephony.jd`
[12] причины такого поведения не ясны

При запуске второго виртуального окружения Android некоторые ioctl-вызовы этого драйвера возвращаются ошибки, вызванные наличием статического состояния:

```
[drivers/rtc/alarm-dev.c]
50 static int alarm_opened;
51 static DEFINE_SPINLOCK(alarm_slock);
52 static struct wake_lock alarm_wake_lock;
53 static DECLARE_WAIT_QUEUE_HEAD(alarm_wait_queue);
54 static uint32_t alarm_pending;
55 static uint32_t alarm_enabled;
56 static uint32_t wait_pending;
57
58 static struct alarm alarms[ANDROID_ALARM_TYPE_COUNT];
```

Вот начало обработчика ioctl-запросов этого драйвера:

```
[drivers/rtc/alarm-dev.c]
60 static long alarm_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
61 {
62         int rv = 0;
63         unsigned long flags;
64         struct timespec new_alarm_time;
65         struct timespec new_rtc_time;
66         struct timespec tmp_time;
67         enum android_alarm_type alarm_type = ANDROID_ALARM_IOCTL_TO_TYPE(cmd);
68         uint32_t alarm_type_mask = 1U << alarm_type;
69
70         if (alarm_type >= ANDROID_ALARM_TYPE_COUNT)
71                 return -EINVAL;
72
73         if (ANDROID_ALARM_BASE_CMD(cmd) != ANDROID_ALARM_GET_TIME(0)) {
74                 if ((file->f_flags & O_ACCMODE) == O_RDONLY)
75                         return -EPERM;
76                 if (file->private_data == NULL &&
77                     cmd != ANDROID_ALARM_SET_RTC) {
78                         spin_lock_irqsave(&alarm_slock, flags);
79                         if (alarm_opened) {
80                                 spin_unlock_irqrestore(&alarm_slock, flags);
81                                 return -EBUSY;
82                         }
83                         alarm_opened = 1;
84                         file->private_data = (void *)1;
85                         spin_unlock_irqrestore(&alarm_slock, flags);
86                 }
87         }
```

Видно, что строки 76–81 запрещают выполнение set-запросов на файле /dev/alarm, если его уже открыл другой процесс. Это означает, что драйвер не будет обслуживать никакие виртуальные окружения, кроме первого запущенного.

2.3.7 Wake-блокировки

Wake-блокировка — объект ядра, разработанный в рамка проекта AOSP и являющийся частью еще одного механизма управления энергопотреблением ядра. В ядре есть workqueue `suspend`, которая активируется при снятии wake-блокировки функцией `wake_unlock`[13] либо при истечении ее таймаута.

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.[14] Аналогично, запись строки `S` в файл `power/wake_unlock` приведет к отпусканию 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.

Кроме того, в статье Cells указывается на необходимость виртуализировать не только wake-блокировки, экспортируемые через `sysfs`, но и wake-блокировки, используемые только в ядре. Однако в наших экспериментах ни разу не наблюдался переход машины в состояние пониженного энергопотребления, вызванный wake-блокировками, поэтому было принято решение не реализовывать методику виртуализации wake-блокировок, изложенную в статье Cells.

2.3.8 Доступ к устройствам

Файловая система `cgroup` реализует возможность задавать список устройствам, к которым имеет доступ процесс.

2.4 Постановка задачи

Таким образом, в настоящей работе будут решены следующие задачи:

- адаптация утилит управления пространствами имен ядра Linux для работы на смартфоне,
- обеспечение совместного доступа из одновременно работающих контейнеров к
  - драйверам `Binder` и `Alarm`,
  - драйверу `evdev` подсистемы ввода ядра Linux,
  - сервису телефонии,
  - WiFi.

[13] `kernel/power/wakelock.c`
[14] `kernel/power/userwakelock.c`

3 Реализация

3.1 Архитектура

Архитектура разрабатываемого решения представлена на рис. 2.

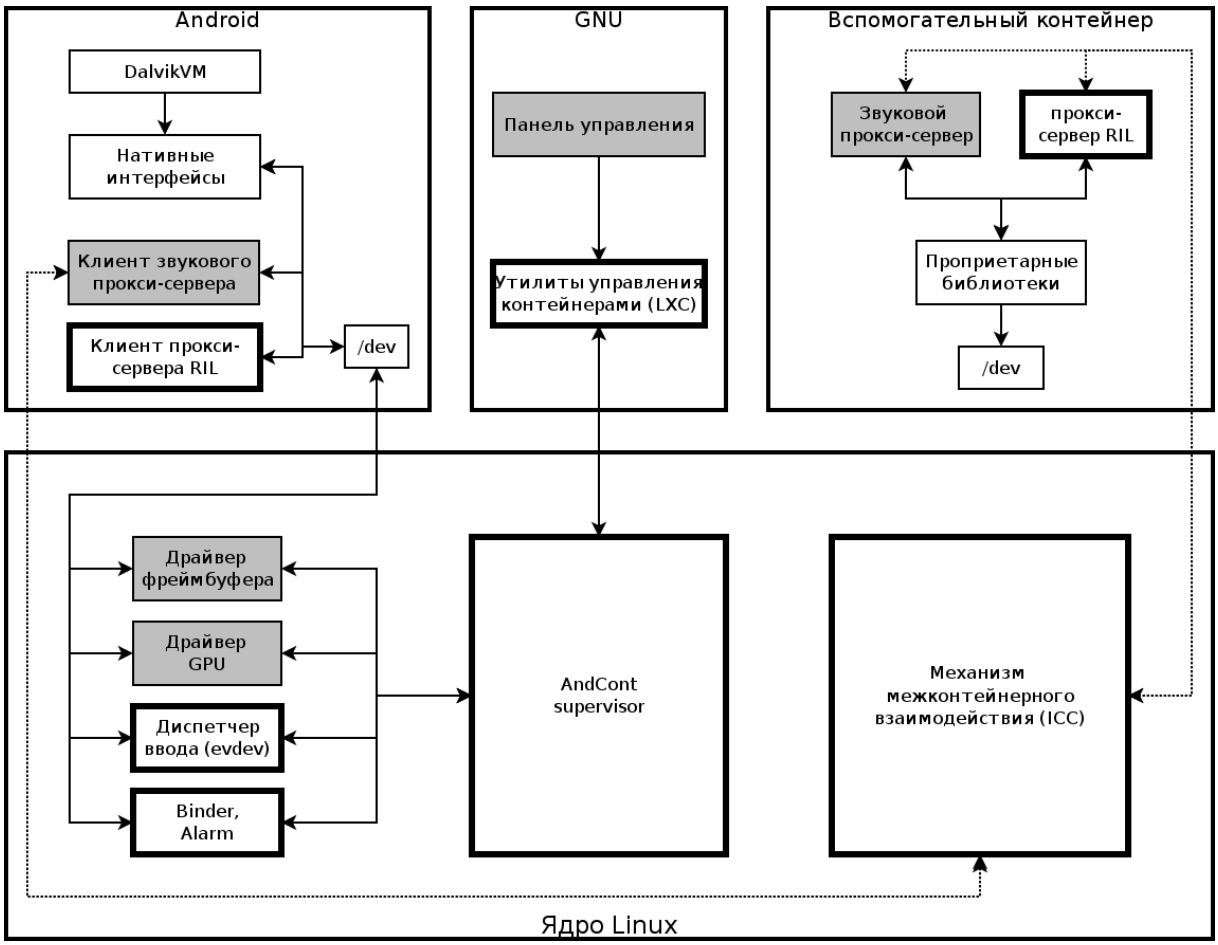

Рис. 2: Архитектура системы виртуализации

На рис. 2 жирной рамкой отмечены компоненты, которые были упомянуты в предыдущем разделе и о которых пойдет речь в настоящем разделе. Затемненным прямоугольником отмечены компоненты, которые не входят в область задач, решаемых в настоящей работе, но которые зависят от компонентов, описанных в настоящей работе.

Кратко опишем назначение каждого компонента:

- Панель управления — графический интерфейс к LXC; позволяет
  - выводить список контейнеров, имеющихся на смартфоне,
  - запускать выбранный контейнер,
  - переключаться в выбранный контейнер.
- LXC — набор утилит управления виртуальными контейнерами; эти утилиты позволяют
  - запускать и останавливать контейнер,
  - управлять ограничениями на ресурсы, доступные контейнеру,
  - управлять доступом к системным устройствам.

  Кроме того, утилита, предназначенная для запуска контейнера (`lxc_start`), уведомляет супервизор ядра о запуске контейнера.

- Супервизор ядра (компонент `AndCont Supervisor` на диаграмме)
  - инициализирует виртуальное состояние драйверов, перехватывая момент запуска контейнера,
  - реализует переключение между контейнерами.
- Механизм межконтейнерного взаимодействия (ICC) — средство коммуникации между контейнерами на базе Netlink, обходящее проверки в прикладных сетевых протоколах.
- Диспетчер ввода обеспечивает совместное использование устройств ввода `Android`'ами, работающими параллельно.
- `Binder` — драйвер IPC, реализованный в рамках проекта AOSP.
- `Alarm` — интерфейс к часам реального времени.
- Прокси-сервер RIL обеспечивает совместный доступ нескольких `Android`'ов к оборудованию доступа к мобильным сетям.
- Прокси-клиент RIL принимает запросы к сервису телефонии от приложений в контейнере.
- Вспомогательный контейнер предназначен для запуска системных демонов и библиотек `Android`, по разным причинам вынесенных из гостевых `Android`'ов.

3.2 Инфраструктура AndCont, встраиваемая в ядро

Для управления виртуальными контейнерами и их состоянием в ядро была добавлена группа модулей, которую будем называть `AndCont`. `AndCont` можно разделить на две части:

- *супервизор* — компонент, отвечающий за создание, уничтожение и переключение виртуальных контейнеров, а также хранение их виртуального состояния;
- *драйвер виртуального устройства* — управляет частью состояния виртуального контейнера.

Виртуальный смартфон (мы его также называем контейнером) представлен в инфраструктуре `AndCont` следующей структурой:

```
struct andcont_virt_phone {
  vcid_t id;
  spinlock_t lock;
#ifdef CONFIG_ANDCONT_SUPERVISOR_POWER_MANAGEMENT
  unsigned int suspended;
#endif
//driver specific structures
#ifdef CONFIG_ANDCONT_VALARM_DEV
  alarm_state_t alst;
#endif
#ifdef CONFIG_ANDCONT_VBINDER
  binder_state_t bs;
#endif
#ifdef CONFIG_ANDCONT_VIRTUAL_FB
  vfb_phone_data_t vfb_pd;
#endif
#ifdef CONFIG_ANDCONT_VPM_SGS2
  vpmsgs2_state_t vpmsgs2;
#endif
```

```
#ifdef CONFIG_ANDCONT_DEMO_DRIVER
  my_phone_data_t mpd;
#endif
}
```

Ее поля имеют следующий смысл:

- `id` — идентификатор контейнера. Мы будем использовать поле `nsproxy` структуры `task_struct`, на которую указывает `current`, в качестве идентификатора текущего контейнера. Единственная проблема этого решения — выполнение системного вызова `clone()`, который скопирует хотя бы одно пространство имен; такая ситуация, видимо, маловероятна. Конечно, лучшим следует признать решение `OpenVZ`, в котором под идентификатор виртуального окружения резервируется поле в структуре `task_struct`, но тогда нам бы пришлось модифицировать код функции `do_fork`, для того чтобы она корректно наследовала идентификаторы контейнеров.
- `alst` — виртуальное состояние драйвера `Alarm`.
- `bs` — виртуальное состояние драйвера `Binder`.
- `vfb_pd` — виртуальное состояние драйвера фреймбуфера.
- `vpmsgs2` — виртуальное состояние драйвера управления питанием для смартфона Samsung Galaxy SII ([будет] описан в докладе Евгения).

3.2.1 Супервизор

Супервизор предназначен для решения следующих задач:

- создание виртуального состояния контейнера при получении специального сообщения от пользовательского приложения;
- переключение активного контейнера;
- уничтожение виртуального состояния контейнера при его остановке и уведомление пользовательских приложений об этом событии.

В качестве транспорта для взаимодействия с супервизором был выбран протокол семейства `Netlink` с номером 17 (`NETLINK_SUPERVISOR`). Каждое сообщение, адресованное супервизору или посылаемое им, начинается стандартным заголовком протоколов `Netlink` — структурой `struct nlmsghdr`. Поле `nlmsg_type` этого заголовка делится на две части: верхняя часть интерпретируется как номер *подсистемы супервизора*, а нижняя может выбираться произвольно. Сейчас в супервизоре имеется две подсистемы:

- `ANDCONT_SUBSYS_KERNEL` (1) — подсистема управления виртуальными контейнерами и уведомления об изменении их состояния (будет описана в этом разделе);
- `ANDCONT_SUBSYS_ROUTER` (2) — подсистема межконтейнерного взаимодействия (будет описана в следующем разделе).

Для супервизора резервируется первая широковещательная группа протокола `NETLINK_SUPERVISOR`, в которую подсистема управления контейнерами рассылает уведомления об изменении состояния виртуальных контейнеров.

В сообщении подсистемы управления контейнерами после заголовка `Netlink` идет заголовок, описываемый следующей структурой:

```
enum andcont_supervisor_event_t {
  ANDCONT_SUPERV_EVENT_PHONE_ADDED,
  ANDCONT_SUPERV_EVENT_PHONE_REMOVED,
  ANDCONT_SUPERV_EVENT_FOREGROUND_CHANGED
};

typedef struct {
  enum andcont_supervisor_event_t event;
  void* arg;
} andcont_msg_event_t;
```

Значение перечисления `andcont_supervisor_event_t` описывают три типа изменения состояния виртуального контейнера:

- `ANDCONT_SUPERV_EVENT_PHONE_ADDED` — это сообщение отправляется программой инициализации контейнера перед запуском демона `init` контейнера; значение поля `arg` в этом случае не определено. После создания виртуального состояния для запускаемого контейнера супервизор ретранслирует это сообщение в свою группу широковещания, проинициализировав поле `arg` номером первого процесса в контейнере — здесь используется одно из свойств реализации `Netlink` — обработчик входящего сообщения `Netlink` вызывается в контексте процесса, пославшего это сообщение.
- `ANDCONT_SUPERV_EVENT_PHONE_REMOVED` — супервизор отправляет это сообщение в свою группу широковещания при уничтожении всех пространств имен контейнера. Фактически перехватывается вызов функции `free_nsproxy`, которая вызывается при уничтожении последнего процесса в контейнере.
- `ANDCONT_SUPERV_EVENT_FOREGROUND_CHANGED` — пользовательская программа отправляет этого сообщение, чтобы переключить активный контейнер; после того как все виртуализированные драйверы обработают это событие, сообщение ретранслируется в группу широковещания супервизора.

3.2.2 Механизм межконтейнерного взаимодействия

Разрабатываемое решение содержит клиент-серверные компоненты (прокси-RIL, прокси-аудио), клиентская и серверная части которых работают в разных сетевых пространствах имен. Были предприняты использовать следующие существующие в ядре `Linux` механизмы в качестве транспорта между клиентской и серверной частями таких компонент:

- локальные Unix-сокеты — попытка оказалась неуспешной, поскольку ядро запрещает установку соединения между двумя такими сокетами, существующими в разных сетевых пространствах имен: эта политика реализована в функции `unix_find_socket_byinode`, которая, в частности, вызывается при установке соединения:

```
[net/unix/af_unix.c]
285 static struct sock *unix_find_socket_byinode(struct net *net, struct inode *i)
286 {
287         struct sock *s;
288         struct hlist_node *node;
289
290         spin_lock(&unix_table_lock);
291         sk_for_each(s, node,
292                     &unix_socket_table[i->i_ino & (UNIX_HASH_SIZE - 1)]) {
293                 struct dentry *dentry = unix_sk(s)->dentry;
```

```
294
295                 if (!net_eq(sock_net(s), net))
296                         continue;
297
298                 if (dentry && dentry->d_inode == i) {
299                         sock_hold(s);
300                         goto found;
301                 }
302         }
303         s = NULL;
304 found:
305         spin_unlock(&unix_table_lock);
306         return s;
307 }
```

- `Netlink` — попытка также оказалась неуспешной по той же причине: в частности, такая политика реализована в функции `netlink_lookup`, которая ищет `Netlink`-сокет в указанном сетевом пространстве имен:

```
[net/netlink/af_netlink.c]
229 static inline struct sock *netlink_lookup(struct net *net, int protocol,
230                                            u32 pid)
231 {
232         struct nl_pid_hash *hash = &nl_table[protocol].hash;
233         struct hlist_head *head;
234         struct sock *sk;
235         struct hlist_node *node;
236
237         read_lock(&nl_table_lock);
238         head = nl_pid_hashfn(hash, pid);
239         sk_for_each(sk, node, head) {
240                 if (net_eq(sock_net(sk), net) && (nlk_sk(sk)->pid == pid)) {
241                         sock_hold(sk);
242                         goto found;
243                 }
244         }
245         sk = NULL;
246 found:
247         read_unlock(&nl_table_lock);
248         return sk;
249 }
```

Эта функция вызывается функцией `netlink_getsockbypid`, которая, в свою очередь, вызывается функцией `netlink_unicast`, причем в параметре `net` оказывается указатель на сетевое пространство имен, которому принадлежит сокет, через который отсылается сообщение. Проверка в строке 240 не позволит пройти сообщению между сетевыми пространствами имен.

Аналогичная политика реализована в функции `do_one_broadcast`, которая осуществляет широковещательную рассылку сообщения `Netink`:

```
[net/netlink/af_netlink.c]
985 static inline int do_one_broadcast(struct sock *sk,
986                                     struct netlink_broadcast_data *p)
987 {
988         struct netlink_sock *nlk = nlk_sk(sk);
989         int val;
990
991         if (p->exclude_sk == sk)
992                 goto out;
993
994         if (nlk->pid == p->pid || p->group - 1 >= nlk->ngroups ||
995             !test_bit(p->group - 1, nlk->groups))
996                 goto out;
997
998         if (!net_eq(sock_net(sk), p->net))
999                 goto out;
```

Проверка на строке 998 снова не позволит пройти сообщению между сетевыми пространствами имен.

- Unix FIFO — оказалось, что этот механизм подходит для нашей задачи, но у него есть следующие недостатки:
  - канал `Unix` является однонаправленным, поэтому для дуплексного соединения необходимо создавать по два канала;
  - необходим промежуточный агент, который бы соединял клиентские и серверный части, поскольку канал нельзя разместить в файловой системе так, чтобы он был доступен в разных контейнерах.
- Использование инфраструктуры виртуализации сети опять же требует наличия промежуточного агента в корневом пространстве имен.

По этой причине было принято решение создать собственный механизм, который мы будем называть роутером, на основе `Netlink`. Поскольку супервизор тоже использует `Netlink`, то протокол роутера стал расширением протокола супервизора. Сообщения роутера имеют номер подсистемы 2, заголовок такого сообщения определен следующим образом:

```
typedef struct {
  pid_t dest;
  pid_t src_pid;
  uint32_t dst_group;
  uint32_t type;
} andcont_router_msg_t;
```

Поля этого заголовка имеют следующий смысл:

- `dest` — PID первого процесса контейнера в корневом пространстве имен;
- `src_pid` — PID первого процесса контейнера, отправившего это сообщение, заполняется ядром.
- `dst_group` — группа широковещания, в которую должно быть доставлено сообщение;

- `type` — поле, зарезервированное для пользовательского протокола.

Общая схема использования роутера выглядит следующим образом: для каждого виртуализируемого в пространстве пользователя сервиса резервируется группа широковещания `Netlink`, затем клиентская часть прокси этого сервиса открывает сокет `Netlink` в этой группе. Затем в заголовках всех запросов, отправляемых на серверную часть прокси, значение поля `dest` устанавливается равным 1 — серверная часть должна работать в корневом сетевом пространстве имен. Далее, серверная часть прокси запоминает значение поля `src_pid` в сообщение запроса и помещает его в поле `dest` сообщения-ответа.

3.3 Изолированное окружение исполнения

3.3.1 LXC

Для организации изолированных окружений был выбран набор утилит LXC, поскольку он предоставляет готовое решение для изоляции идентификаторов процессов, виртуализации сети, а также управления ресурсами, потребляемыми каждым виртуальным окружением.

Рассматривался также вариант использования для этих целей инфраструктуры `OpenVZ`, но этот вариант был отвергнут по следующим причинам:

- Текущая экспериментальная версия патча `OpenVZ` разрабатывается для ядра версии 2.6.32, но на смартфонах, на которых тестировалось наше решение, работают ядра версии 2.6.35.
- При попытке наложить патч `OpenVZ` на одно из используемых ядер не удалось пропатчить 166 файлов из 1682; кроме того, патч модифицирует 34 файла, специфичных для архитектуры `x86`.

Портирование такого количества изменений на ядро 2.6.35 было сочтено нецелесообразным.

Следует отметить, что, отказавшись от `OpenVZ`, мы лишились готового решения по виртуализации `sysfs`: при создании нового виртуального окружения `OpenVZ` копирует дерево kobject'ов хостового окружения.

3.3.2 Модификации LXC

Набор утилит LXC пришлось адаптировать для наших целей.

Сброс полномочия CAP_SYS_BOOT Рассмотрим файл `lxc/src/start.c`

```
 421 static int do_start(void *data)
 422 {
 423         struct lxc_handler *handler = data;
[...]
 455         if (prctl(PR_CAPBSET_DROP, CAP_SYS_BOOT, -1, 0, 0)) {
 456                 SYSERROR("failed to remove CAP_SYS_BOOT capability");
 457                 return -1;
 458         }
```

`do_start` — функция, которая подготавливает контейнер к запуску, ей передается управления после завершения системного вызова `clone()`. На 455 строке эта функция сбрасывает полномочие `CAP_SYS_BOOT` у запускаемого контейнера. Однако впоследствии процесс `zygote` пытается выставить разрешенные полномочия `07C13C20` (130104352), т. е. пытается разрешить полномочие `CAP_SYS_BOOT` (`0x400000`). Происходит это следующим образом[15]:

1. `init` запускает процесс `app_process` с параметрами:

[15] как это было обнаружено на самом деле: `http://osll.spb.ru/issues/3031#note-11`

```
/system/bin/app_process -Xzygote /system/bin --zygote --start-system-server
```

2. Процесс `app_process` запускает виртуальную машину `Dalvik`, а в ней — класс `com.android.internal.os.ZygoteInit`.

3. Метод `main` класса `com.android.internal.os.ZygoteInit` вызывает метод `startSystemServer`:

```
[frameworks/base/core/java/com/android/internal/os/ZygoteInit.java]
 533    private static boolean startSystemServer()
 534            throws MethodAndArgsCaller, RuntimeException {
 535        /* Hardcoded command line to start the system server */
 536        String args[] = {
 537            "--setuid=1000",
 538            "--setgid=1000",
 539            "--setgroups=1001,1002,1003,1004,1005,1006,1007,1008,1009,1010,1018,
                            1300,3001,3002,3003",
 540            "--capabilities=130104352,130104352",
 541            "--runtime-init",
 542            "--nice-name=system_server",
 543            "com.android.server.SystemServer",
 544        };
 545        ZygoteConnection.Arguments parsedArgs = null;
 546
 547        int pid;
 548
 549        try {
[...]
 562            pid = Zygote.forkSystemServer(
 563                    parsedArgs.uid, parsedArgs.gid,
 564                    parsedArgs.gids, debugFlags, null,
 565                    parsedArgs.permittedCapabilities,
 566                    parsedArgs.effectiveCapabilities);
```

4. Метод `Zygote.forkSystemServer` является нативным:

```
[dalvik/vm/native/dalvik_system_Zygote.c]
491 static void Dalvik_dalvik_system_Zygote_forkSystemServer(
492         const u4* args, JValue* pResult)
493 {
494     pid_t pid;
495     pid = forkAndSpecializeCommon(args, true);
```

5. Функция `forkAndSpecializeCommon` пытается установить полномочия (строка 445):

```
 357 static pid_t forkAndSpecializeCommon(const u4* args, bool isSystemServer)
 358 {
 359     pid_t pid;
```

```
 360
 361     uid_t uid = (uid_t) args[0];
 362     gid_t gid = (gid_t) args[1];
 363     ArrayObject* gids = (ArrayObject *)args[2];
 364     u4 debugFlags = args[3];
 365     ArrayObject *rlimits = (ArrayObject *)args[4];
 366     int64_t permittedCapabilities, effectiveCapabilities;
 367
 368     if (isSystemServer) {
[...]
 376         permittedCapabilities = args[5] | (int64_t) args[6] << 32;
 377         effectiveCapabilities = args[7] | (int64_t) args[8] << 32;
 378     } else {
 379         permittedCapabilities = effectiveCapabilities = 0;
 380     }
[...]
 445         err = setCapabilities(permittedCapabilities, effectiveCapabilities);
```

6. Наконец, функция `setCapabilities` делает системный вызов `capset` (строка 347):

```
331 static int setCapabilities(int64_t permitted, int64_t effective)
332 {
333 #ifdef HAVE_ANDROID_OS
334     struct __user_cap_header_struct capheader;
335     struct __user_cap_data_struct capdata;
336
337     memset(&capheader, 0, sizeof(capheader));
338     memset(&capdata, 0, sizeof(capdata));
339
340     capheader.version = _LINUX_CAPABILITY_VERSION;
341     capheader.pid = 0;
342
343     capdata.effective = effective;
344     capdata.permitted = permitted;
345
346     LOGV("CAPSET perm=%llx eff=%llx\n", permitted, effective);
347     if (capset(&capheader, &capdata) != 0)
348         return errno;
```

Из приведенных фрагментов кода видно, что значение разрешенных полномочий определяется значением строкой 540 файла `ZygoteInit.java`. Поскольку мы хотим вносить минимальное количество изменений в `Android` и, тем более, в его часть, написанную на Java, то было принято решение удалить строки 455–458 из файла `lxc/src/lxc_start.c`.

Открытые файловые дескрипторы Еще раз посмотрим файл `src/lxc/start.c`:

```
139 int lxc_check_inherited(int fd_to_ignore)
140 {
```

```
 141         struct dirent dirent, *direntp;
 142         int fd, fddir;
 143         DIR *dir;
 144         int ret = 0;
 145
 146         dir = opendir("/proc/self/fd");
[...]
 152         fddir = dirfd(dir);
 153
 154         while (!readdir_r(dir, &dirent, &direntp)) {
 155                 char procpath[64];
 156                 char path[PATH_MAX];
[...]
 167                 fd = atoi(direntp->d_name);
[...]
 174                 /*
 175                  * found inherited fd
 176                  */
 177                 ret = -1;
 178
 179                 snprintf(procpath, sizeof(procpath), "/proc/self/fd/%d", fd);
 180
 181                 if (readlink(procpath, path, sizeof(path)) == -1)
 182                         ERROR("readlink(%s) failed : %m", procpath);
 183                 else
 184                         ERROR("inherited fd %d on %s", fd, path);
 185         }
 186
[...]
 189         return ret;
```

Функция `lxc_check_inherited` вызывается функцией `lxc_start`, которая вызывается утилитой `lxc_start` для запуска контейнера:

```
 625 int lxc_start(const char *name, char *const argv[], struct lxc_conf *conf)
 626 {
[...]
 630
 631         if (lxc_check_inherited(-1))
 632                 return -1;
```

Из приведенных фрагментов кода видно, что функция `lxc_start` вернет ошибку, если функция `lxc_check_inherited` обнаружит унаследованный файловый дескриптор. Выяснилось[16], что некоторые файловые дескрипторы «утекают» из демона `adbd`, что делает невозможным использование LXC вообще.

Причина такого поведения не очень ясна, но для того чтобы удовлетворить требование отсутствия унаследованных файловых дескрипторов, строки 179–184 была заменены на строку

```
close(fd);
```

[16] `http://osll.spb.ru/issues/3004#note-32`

Уведомления супервизора уровня ядра Супервизор уровня ядра необходимо уведомить о запуске нового контейнера для того, чтобы он создал экземпляр виртуального состояния драйверов, описанных в следующих разделах.

Было реализовано два варианта решения этой задачи:

- с помощью файла под каталогом `/dev`,
- с помощью синхронного сообщения протокола `Netlink`.

Сейчас реализован второй вариант через механизм межконтейнерного взаимодействия.

3.4 Binder

Из сказанного в разделе 2.3.2 следует, что необходимо иметь изолированные экземпляры переменных `binder_context_mgr_node` и `binder_context_mgr_uid` для каждого виртуального окружения. С другой стороны, преследовалась цель внести минимальное количество изменений в существующий код. Поэтому процедуры доступа к виртуальному состоянию были реализованы следующим образом:

```
struct binder_node;

typedef struct binder_state {
  struct binder_node* v_binder_context_mgr_node;
  uid_t v_binder_context_mgr_uid;

  vcid_t vc;
} binder_state_t;

// ============================================================================

binder_state_t* andcont_lookup_binder_state(void);

// ----------------------------------------------------------------------------

#define VBINDER_STATE                                                  \
  binder_state_t* virtual_binder_state = andcont_lookup_binder_state(); \
  if (!virtual_binder_state) {                                         \
    panic("Failed to lookup the current virtual state!\n");            \
  }

#define binder_context_mgr_node virtual_binder_state->v_binder_context_mgr_node

#define binder_context_mgr_uid  virtual_binder_state->v_binder_context_mgr_uid
```

Макрос `VBINDER_STATE` был добавлен в начало всех функций, обращающихся к виртуализированному состоянию драйвера:

- `binder_inc_node`,
- `binder_get_ref_for_node`,
- `binder_transaction`,
- `binder_thread_write`,

- `binder_ioctl`.

Определение имен виртуализированных переменных `binder_context_mgr_node` и `binder_context_mgr_uid` как имен макросов позволило не заменять все вхождения этих переменных на обращения к их виртуальным версиям.

Кроме того, в разделе 2.3.2 отмечалась, что необходимо модифицировать функцию `binder_deferred_release`:

```
2945 static void binder_deferred_release(struct binder_proc *proc)
2946 {
2947         struct hlist_node *pos;
2948         struct binder_transaction *t;
2949         struct rb_node *n;
2950         int threads, nodes, incoming_refs, outgoing_refs, buffers,
                 active_transactions, page_count;
2951
2952         BUG_ON(proc->vma);
2953         BUG_ON(proc->files);
2954
2955         hlist_del(&proc->proc_node);
2956         if (binder_context_mgr_node && binder_context_mgr_node->proc == proc) {
2957                 binder_debug(BINDER_DEBUG_DEAD_BINDER,
2958                              "binder_release: %d context_mgr_node gone\n",
2959                              proc->pid);
2960                 binder_context_mgr_node = NULL;
2961         }
```

Судя по смыслу проверки на строке 2956, в виртуализированной версии драйвера необходимо сделать действие в строках 2957–2960 для всех активных виртуальных окружений, поэтому строки 2956–2961 были переписаны следующим образом:

```
struct list_head* it;
vphone_list_t* list;

list = andcont_get_vphone_list();
spin_lock(&list->lock);

list_for_each(it, &list->head.list_entry) {
  vphone_t *phone = &list_entry(it, vphone_entry_t, list_entry)->phone;
  binder_state_t* virtual_binder_state = &phone->bs;

  if (binder_context_mgr_node && binder_context_mgr_node->proc == proc) {
    binder_debug(BINDER_DEBUG_DEAD_BINDER,
                 "binder_release: %d context_mgr_node gone\n",
                 proc->pid);
    binder_context_mgr_node = NULL;
  }
}

spin_unlock(&list->lock);
```

3.5 Подсистема ввода

В драйвере `evdev` процессу, открывшему файл в каталоге `/dev/input`, ставится в соответствие структура

```
struct evdev_client {
        struct input_event buffer[EVDEV_BUFFER_SIZE];
        int head;
        int tail;
        spinlock_t buffer_lock; /* protects access to buffer, head and tail */
        struct fasync_struct *fasync;
        struct evdev *evdev;
        struct list_head node;
        struct wake_lock wake_lock;
        char name[28];
}
```

В эту структуру было добавлено поле

```
struct task_struct* owner
```

чтобы информация об этом процессе не терялась. Это поле теперь инициализируется в функции `evdev_open` — обработчике системного вызова `open` устройств `evdev`:

```
static int evdev_open(struct inode *inode, struct file *file)
                      dev_name(&evdev->dev), task_tgid_vnr(current));
        wake_lock_init(&client->wake_lock, WAKE_LOCK_SUSPEND, client->name);
        client->evdev = evdev;

        client->owner = current;
```

Далее, в функцию `evdev_pass_event`, занимающуюся помещением событий ввода в очередь событий каждого процесса, читающего `evdev`, мы добавили строки

```
if (!task_vcid(client->owner) == andcont_get_foreground_virtual_phone_id()) {
  return;
}
```

Так мы предотвратили передачу событий ввода в неактивные контейнеры.

3.6 Alarm

Подход к виртуализации этого драйвера аналогичен подходу к виртуализации драйвера `binder`: все статические переменные в драйвере `alarm-dev` были собраны в структуру, экземпляр которой создается при запуске контейнера:

```
typedef struct alarm_state {
  int v_alarm_opened;
  uint32_t v_alarm_pending;
  uint32_t v_alarm_enabled;
```

```
  uint32_t v_wait_pending;

  struct alarm v_alarms[ANDROID_ALARM_TYPE_COUNT];
} alarm_state_t;

#define VALARM_STATE                                                    \
  alarm_state_t* virtual_alarm_state = andcont_lookup_alarm_state();    \
  if (!virtual_alarm_state) {                                           \
    panic("Failed to lookup the current virtual state!\n");             \
  }

// Virtual state accessors

//#define alarm_wait_queue virtual_alarm_state->v_alarm_wait_queue
#define alarm_opened     virtual_alarm_state->v_alarm_opened
#define alarm_pending    virtual_alarm_state->v_alarm_pending
#define alarm_enabled    virtual_alarm_state->v_alarm_enabled
#define wait_pending     virtual_alarm_state->v_wait_pending
#define alarms           virtual_alarm_state->v_alarms

// Externals

extern alarm_state_t* andcont_lookup_alarm_state(void);
```

3.7 Телефония

Драйверы GSM-модема реализованы по-разному даже на смартфонах, на которых мы тестировали наше решение, поэтому найти универсальное решение для виртуализации этого устройства на уровне ядра 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. Поэтому было принято решение подменить проприетарную библиотеку RIL, заменив ее заглушкой. Сама же проприетарная библиотека RIL запускается в отдельном виртуальном окружении под управлением демона, фильтрующего запросы от заглушки RIL.

Текущее решение для виртуализации RIL использует механизм межконтейнерного взаимодействия и резервирует пятую широковещательную группу `Netlink` в протоколе `NETLINK_SUPERVISOR` для взаимодействия между серверной и клиентской частями.

3.7.1 Клиентская часть

Демон `rild` загружает библиотеку RIL, имя которой указано в системном свойстве `rild.libpath` и вызывает функцию `RIL_Init`, которая имеет следующий прототип:

```
const RIL_RadioFunctions* RIL_Init(const struct RIL_Env* env, int argc, char** argv);
```

В качестве аргумента `argv` передается разбитое на слова значение системного свойства `rild.libarg`.

Значения системных свойств `rild.libpath` и `rild.libarg` задаются в файле `/system/build.prop`[17], который загружает в базу данных системных свойств процесс `init`. Поэтому для подмены реализации RIL достаточно изменить значение системного свойства `rild.libpath`.

Структура `RIL_RadioFunctions` содержит следующие поля:

[17] по крайней мере, это верно для смартфонов, с которыми мы работали

- `void (*onRequest)(int request, void* data, size_t datalen, RIL_Token t)` — обработчик запросов RIL;
- `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 передается указатель на следующую 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;
- `void (*OnUnsolicitedResponse)(int unsolResponse, const void *data, size_t datalen)` — асинхронное уведомление демона `rild`;
- `void (*RequestTimedCallback) (RIL_TimedCallback callback, void *param, const struct timeval *relativeTime)` — вызывает функцию `callback` по истечении временного интервала, описанного параметром `relativeTime`.

Есть три особенности архитектуры демона `rild` и интерфейса RIL, которые повлияли на дизайн клиентской заглушки:

1. обработчик `OnRequestComplete` может вызываться в любой момент;
2. обработчики `onStateRequest`, `supports` и `getVersion` библиотеки RIL являются синхронными, что в контексте заглушки 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 сервера вынесена в отдельный поток; при получении сообщения эта процедура кладет его в одну из очередей в зависимости от типа ответа;
- диспетчер сообщений типа `PROXY_MSG_RESPONSE` и `PROXY_MSG_UNSOL_RESPONSE` (это сообщения от прокси-сервера, см. следующий раздел) вынесен в отдельный поток.

Совместно два этих решения позволили избежать самоблокировок из-за п. 3.

Заглушка RIL упаковывает эти вызовы в сообщения со следующим заголовком:

```
struct client_to_server_msg_header {
  size_t size;
  int type;
  RIL_Token tok;
  int req_id;
}
```

Каждый из вызовов обрабатывается следующим образом:

- `onStateRequest` — поле `type` инициализируется значением `PROXY_MSG_STATE_REQUEST` (2);
- `supports` — поле `type` инициализируется значением `PROXY_MSG_SUPPORT_REQUEST` (3), поле `req_id` — параметром `requestCode`;

- `onCancel` — поле `type` инициализируется значением `PROXY_MSG_CANCEL_REQUEST` (4), поле `tok` — параметром `t`;
- `getVersion` — поле `type` инициализируется значением `PROXY_MSG_VERSION_REQUEST` (5);
- `onRequest` — поле `type` инициализируется значением `PROXY_MSG_REQUEST` (1), поле `req_id` — параметром `request`, поле `tok` — параметром `t`, за заголовком следует сериализованное представление параметров запроса, на которые указывает параметр `data`,

Процедуры сериализации/десериализации параметров запросов генерируются автоматически по описаниям структур в файле `hardware/ril/include/telephony/ril.h`. Структуры сериализуются следующим образом: копия структуры помещается в буфер сериализации, поля, являющееся указателем, сериализуются отдельно и помещаются в буфер последовательно после копии структуры; затем все указатели заменяются смещениями относительно начала буфера сериализации.

Особенности маршаллинга Интерфейс RIL определяет 103 типа запросов (определяемых аргументом `request` обработчика `onRequest`), но только 31 из них потребовал нетривиального маршаллинга аргументов; ниже перечислены запросы, корректность обработки которых была проверена в ходе тестирования решения:

- `RIL_REQUEST_SEND_SMS`,
- `RIL_REQUEST_QUERY_FACILITY_LOCK`,
- `RIL_REQUEST_DIAL`,
- `RIL_REQUEST_SIM_IO`.

3.7.2 Серверная часть

Серверная часть реализована как однопоточное приложение, которое загружает проприетарную реализацию RIL и ожидает сообщения от заглушки. Все исходящие сообщения сервера имеют следующий заголовок:

```
struct server_to_client_msg_header {
  size_t size;
  int type;
  RIL_Token tok;
  char* body;
  char* orig_body;

  union {
    RIL_Errno ril_errno;
    RIL_RadioState radio_state;
    int supports;
    int resp_id;
  };
};
```

Ответы проприетарной реализации RIL обрабатываются следующим образом:

- `OnRequestComplete(RIL_Token t, RIL_Errno e, void* response, size_t responselen)` — инициализирует поле заголовка `type` значением `PROXY_MSG_RESPONSE` (6), поле `tok` — значением параметра `t`, поле `ril_errno` — значением `e`; далее размещается сериализованный ответ, на который указывает параметр `response`.

- `OnUnsolicitedResponse(int unsolResponse, const void *data, size_t datalen)` — инициализирует поле заголовка `type` значением `PROXY_MSG_UNSOL_RESPONSE` (7), поле `resp_id` — значением параметра `unsolResponse`; далее размещается сериализованный ответ, на который указывает параметр `data`.
- `RequestTimedCallback` реализует диспетчер на основе POSIX-таймеров[18] и очереди с приоритетами.

Сообщения от заглушки RIL обрабатываются следующим образом:

- при получении сообщение типа `PROXY_MSG_STATE_REQUEST` сервер вызывает функцию `onStateRequest` проприетарной библиотеки и отправляет заглушке заголовок, в котором поле `type` инициализировано значением `PROXY_MSG_STATE_RESPONSE` (7), поле `radio_state` — значением, возвращенным из этой функции;
- при получении сообщение типа `PROXY_MSG_SUPPORT_REQUEST:` сервер вызывает функцию `supports` проприетарной библиотеки и отправляет заглушке заголовок, в котором поле `type` инициализировано значением `PROXY_MSG_SUPPORT_RESPONSE` (8), поле `supports` — значением, возвращенным из этой функции;
- при получении сообщение типа `PROXY_MSG_CANCEL_REQUEST` сервер вызывает функцию `onCancel` проприетарной библиотеки, которой в качество аргумента передается значение поля `tok` входящего сообщения;
- при получении сообщение типа `PROXY_MSG_VERSION_REQUEST` сервер вызывает функцию `getVersion` проприетарной библиотеки и отправляет заглушке заголовок, в котором поле `type` инициализировано значением `PROXY_MSG_VERSION_RESPONSE` (10), за которым следует строка, возвращенная из этой функции.

Следует отметить, что в обработчике `OnRequestComplete` нам неизвестен тип запроса, поэтому при поступлении запроса сервер запоминает соответствие токен–номер запроса, чтобы в этом обработчике вызвать нужный маршаллер ответа.

Особенности маршаллинга Только 17 из 103 запросов RIL требуют нетривиального маршаллинга ответов; ниже перечислены только те запросы, корректность обработки которых была проверена в ходе тестирования решения:

- `RIL_REQUEST_REGISTRATION_STATE`,
- `RIL_REQUEST_GPRS_REGISTRATION_STATE`,
- `RIL_REQUEST_OPERATOR`,
- `RIL_REQUEST_GET_CURRENT_CALLS`,
- `RIL_REQUEST_GET_NEIGHBORING_CELL_IDS`,
- `RIL_REQUEST_GET_SIM_STATUS`,
- `RIL_REQUEST_SIGNAL_STRENGTH`,
- `RIL_REQUEST_SEND_SMS`,
- `RIL_REQUEST_SIM_IO`.

Более того, только 4 из 30 типов асинхронных уведомлений потребовали нетривиального маршаллинга ответов:

[18] POSIX.1-2001

- RIL_UNSOL_ON_USSD,
- RIL_UNSOL_DATA_CALL_LIST_CHANGED,
- RIL_UNSOL_SUPP_SVC_NOTIFICATION,
- RIL_UNSOL_CDMA_CALL_WAITING.

3.7.3 Подмена токенов

Прокси-сервер RIL должен обрабатывать ситуацию, когда из разных клиентов приходят запросы с одинаковыми токенами. Для решения этой проблемы было предложено следующее решение: 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.

3.7.4 Дефекты, обнаруженные в реализации RIL

В процессе отладки решения для виртуализации RIL была обнаружена следующая ошибка в существующей реализации: когда вызывается callback `onRequest` для обработки запроса `RIL_REQUEST_SEND_USSD`, принимающего в качестве аргумента `data` строку, содержащую USSD-запрос, арумент `datalen` содержит некорректную длину этой строки, что приводило к тому, что клиент прокси-RIL отправлял не весь USSD-запрос на сервер. Поэтому для этого запросу пришлось написать специальный маршаллер, исправляющий длину строки-запроса — поскольку без прокси этот зарос отрабатывает корректно, то правдоподобно выглядит такое предположение, что проприетарная библиотека RIL определяет длину USSD-запроса с помощью функции `strlen`, а значит, мы можем использовать эту функцию для исправления длины запроса на стороне клиента-RIL.

Помимо этой, обнаружены также отступления от спецификации в проприетарной библиотеке RIL: в частности, большинство запросов, ответом которых, согласно спецификации, является массив строк фиксированной длины, возвращают массив строк, количество элементов в котором не соответствует спецификации. Подсказка для решения этой проблемы была найдена в исходных текстах библиотеки `libril.so`:

```
[hardware/ril/libril/ril.cpp]
1298 static int responseStrings(Parcel &p, void *response, size_t responselen,
                              bool network_search) {
1299     int numStrings;
1300
1301     if (response == NULL && responselen != 0) {
1302         LOGE("invalid response: NULL");
1303         return RIL_ERRNO_INVALID_RESPONSE;
1304     }
1305     if (responselen % sizeof(char *) != 0) {
1306         LOGE("invalid response length %d expected multiple of %d\n",
1307             (int)responselen, (int)sizeof(char *));
1308         return RIL_ERRNO_INVALID_RESPONSE;
1309     }
1310
1311     if (response == NULL) {
1312         p.writeInt32 (0);
1313     } else {
1314         char **p_cur = (char **) response;
```

```
1315
1316         numStrings = responselen / sizeof(char *);
1317 #ifdef NEW_LIBRIL_HTC
1318         if (network_search == true) {
1319             // we only want four entries for each network
1320             p.writeInt32 (numStrings - (numStrings / 5));
1321         } else {
1322             p.writeInt32 (numStrings);
1323         }
1324         int sCount = 0;
1325 #else
1326         p.writeInt32 (numStrings);
1327 #endif
```

Т. е. обработчик ответа на solicited-запрос не анализирует содержимое ответа (строка 1316) — эта же идея была использована и при маршаллинге таких запросов.

3.8 WiFi

В ходе работы над проектом было обнаружено, что некотрые приложения (например, AndroidMarket) требуют активного WiFi-подключения для нормального функционирования. Однако мы не можем пробросить в контейнер сетевой интерфейс, соответствующий физическому WiFi-адаптеру, поскольку это предполагает удаление этого интерфейса из корневого сетевого пространства имен, т. е. пробросить этот интерфейс в другой контейнер уже не получится.

Была предпринята попытка пробрасывать в контейнеры виртуальные беспроводные сетевые интерфейсы, создаваемые драйвером `mac80211_hwsim`, но у этого решения обнаружились следующие недостатки:

- этот драйвер без модификаций не может создавать виртуальные сетевые интерфейсы по запросу — число этих интерфейсов фиксировано и передается как параметр модуля драйвера;
- в сборки `Android`'а входит модифицированный `wpa_supplicant`, который использует нестандартный интерфейс драйвера `bcm4329` для сканирования радиоэфира — этот интерфейс не является частью стека `CFG80211`, на котором построен драйвер `mac80211_hwsim`

По этим причинам от этого решения пришлось отказаться.

Оказалось, что менеджер WiFi использует библиотеку `libhardware_legacy.so` для управления беспроводным интерфейсом. Интересующий нас интерфейс реализован в файле `hardware/libhardware_legacy/wifi.h`:

- `int wifi_load_driver(void)` — загружает модуль драйвера беспроводного интерфейса в ядро;
- `int wifi_unload_driver(void)` — выгружает модуль драйвера беспроводного интерфейса из ядра;
- `int wifi_start_supplicant(void)` — запускает демон `wpa_supplicant`;
- `int wifi_connect_to_supplicant(void)` — устанавливает соединение с управляющим интерфейсом демона `wpa_supplicant`;
- `int wifi_stop_supplicant(void)` — уничтожает демон `wpa_supplicant`;
- `int wifi_wait_for_event(char *buf, size_t buflen)` — ожидает асинхронных событий на управляющем интерфейсе демона `wpa_supplicant`;
- `int wifi_command(const char *command, char *reply, size_t *reply_len)` — посылает синхронный запрос демону `wpa_supplicant`.

Из этого описания видно, что менеджер WiFi напрямую взаимодействует с `wpa_supplicant`'ом вызывая функции `wifi_command()` и `wifi_wait_for_event()`. Таким образом, для виртуализации WiFi достаточно возвращать «правильные» ответы на запросы менеджера WiFi — тогда нам даже не потребуется запускать сам `wpa_supplicant`.

Все функции этого интерфейса, кроме двух упомянутых выше, были заменены тривиальными заглушками (`return 0;`). С функцией `wifi_command()` несколько сложнее — набор команд `wpa_supplicant`'а довольно велик, однако менеджер WiFi в «благоприятных» условиях (имя точки доступа известно заранее, она доступна и `wpa_supplicant` настроен так, чтобы ассоциировать беспроводной интерфейс с ней) выполняет запросы шести типов:

- `STATUS` — возвращает состояние беспроводного соединения; фиктивный обработчик возвращает следующий ответ:

  ```
  bssid=11:11:11:11:11:11
  ssid=fake
  id=1
  pairwise_cipher=NONE
  group_cipher=NONE
  key_mgmt=NONE
  wpa_state=COMPLETED
  ```

- `AP_SCAN N` — запускает сканирование радиоэфира на интерфейсе с идентификатором `N`; фиктивный обработчик всегда отвечает `OK`.

- `BLACKLIST` — управление «черным» списком (беспроводных сетей?); фиктивный обработчик всегда отвечает `OK`.

- `DISCONNECT` — отключает беспроводной интерфейс от точки доступа; фиктивный обработчик всегда отвечает `OK`.

- `SCAN_RESULTS` — запрашивает список обнаруженных точек доступа; фиктивный обработчик всегда возвращает одну обнаруженную точку доступа:

  ```
  bssid / frequency / signal level / flags / ssid
  11:11:11:11:11:11  2437  194  fake
  ```

- `DRIVER` — введенное проектом AOSP расширение набора команд `wpa_supplicant`'а для прямого взаимодействия с драйвером беспроводного интерфейса[19]. Менеджер WiFi использует следующие команды из этого набора:

  - `POWERMODE` — смысл не ясен; фиктивный обработчик всегда возвращает `OK`.
  - `RSSI-APPROX` — возвращает уровень сигнала точки доступа, с которой в настоящий момент ассоциирован беспроводной интерфейс; фиктивный обработчик всегда отвечает `fake rssi -50`.
  - `RXFILTER-ADD X` — смысл не ясен; фиктивный обработчик всегда возвращает `OK`.
  - `RXFILTER-START` — смысл не ясен; фиктивный обработчик всегда возвращает `OK`.
  - `GETPOWER` — смысл не ясен; фиктивный обработчик всегда возвращает `powermode = 0`.
  - `POWERMODE N` — смысл не ясен; фиктивный обработчик всегда возвращает `OK`.
  - `BTCOEXMODE N` — устанавливает режим совместимости с адаптером Bluetooth; фиктивный обработчик всегда возвращает `OK`.
  - `BTCOEXSCAN-STOP` — смысл не ясен; фиктивный обработчик всегда возвращает `OK`.

[19] `bcm4329` через его проприетарный интерфейс

– SCAN-PASSIVE — активирует режим пассивного сканирования радиоэфира на беспроводном интерфейсе; фиктивный обработчик всегда возвращает OK.

– LINKSPEED — возвращает пропускную способность беспроводного канала передачи данных; фиктивный обработчик всегда возвращает LinkSpeed 72.

Помимо синхронных запросов необходимо имитировать асинхронные ответы демона wpa_supplicant; оказалось, что достаточно имитировать только два события в следующей последовательности:

1. CTRL-EVENT-STATE-CHANGE — уведомление об изменении состояния беспроводного соединения; при первом вызове функции wifi_wait_for_event() возвращается

   ```
   CTRL-EVENT-STATE-CHANGE id=1 state=7 BSSID=00:00:00:00:00:00
   ```

2. CTRL-EVENT-CONNECTED — уведомление о завершении процедуры инициализации беспроводного соединения; при втором вызове функции wifi_wait_for_event() возвращается

   ```
   CTRL-EVENT-CONNECTED - Connection to 11:11:11:11:11:11 completed (auth) [id=1 id_str=]
   ```

Следующий вызов функции wifi_wait_for_event() блокируется на условной переменной.

После получения уведомления CTRL-EVENT-CONNECTED менеджер WiFi запускает DHCP-клиент на беспроводном интерфейсе. Поскольку LXC позволяет задать имя сетевого интерфейса, пробрасываемого в контейнер, то нам не потребуется модифицировать сценарии инициализации контейнера для настройки сети — все это сделает менеджер WiFi. С другой стороны, поскольку такое поведение Android'а невозможно модифицировать конфигурационными файлами, то в корневом пространстве пользователя мы вынуждены запускать DHCP-сервер. Далее, для предотвращения влияния этого DHCP-сервера на физические сети, для создания виртуального интерфейса, пробрасываемого в контейнер, мы вынуждены использовать драйвер veth, который создает пару «соединенных» сетевых интерфейсов, один из которых подключается к виртуальному мосту — таким образом, мы изолируем виртуальную сеть от физической.

3.9 Управление доступом к устройствам

Попытки реализации управления доступом к устройствам путем пометки kobject'ов провалились, поскольку оказалось, что по kobject'у нельзя сказать, какому устройству, видимому под каталогом /dev, он соответствует, даже зная тип тип структуры, в которую внедрен этот kobject. Это означает, что нельзя универсальным способом ограничить доступ как интерфейсам, экспортируемым через файлы под каталогом /dev, так и соответствующим им записям в файловой системе sysfs. Поэтому было принято отказаться от реализации ограничений доступа к sysfs и остановиться на реализации ограничения доступа к файлам блочных и символьных устройств.

Описанная выше задача имеет стандартное решение: инфраструктура ядра cgroup содержит подсистему Device, активируемую опцией CONFIG_CGROUP_DEVICE конфигурационного файла ядра. Доступ к устройствам настраивается с помощью следующих файлов в файловой системы cgroup:

- devices.list — содержит список устройств, к которым группа процессов имеет доступ (доступен только для чтения);
- devices.allow — запись строки вида [a|c|b] <MAJOR>:<MINOR> r?w?m? разрешает доступ к символьному (c), блочному (b) или символьному и блочному (a) устройствам с номером <MAJOR>:<MINOR>.
- devices.deny — запись строки вида [a|c|b] <MAJOR>:<MINOR> r?w?m? исключает символьное (c), блочное (b) или символьное и блочное (a) устройства с номером <MAJOR>:<MINOR> из списка устройств, к которым имеет доступ группа процессов.

Работа этой подсистемы основана на том, что операции открытия/создания файла вызывают функцию `inode_permisson` (`fs/namei.c`, строка 252) после того, как путь к файлу был разрешен, т. е. найдена `inode`'а, соответствующая этому файлу. Эта функция в качестве одной из проверок вызывает функцию `devcgroup_inode_permission` (`security/device_cgroup.c`, строка 478), которая проверяет, входит ли `inode`'а в «белый» список устройств вызывающего процесса и имеет ли этот процесс к запрошенный тип доступа.

Конфигурационный файл LXC может содержать строки вида `lxc.cgroup.<x> = <value>`[20]: при запуске контейнера строка `<value>` будет записана в файл `<CGROUP>/<CONTAINER>/<x>`, где `<CGROUP>` — точка монтирования файловой системы `cgroup`, `CONTAINER` — имя запускаемого контейнера[21]. Поэтому, на наш взгляд, работы по этой задаче сводятся к вычитыванию списка зарегистрированных в системе устройств из каталогов `/sys/dev/char` и `/sys/dev/block` и созданию графического интерфейса, позволяющего задавать ограничения доступа к устройствам и формирующего опции `lxc.cgroup` конфигурационного файла контейнера.

3.10 Настройка сети

Наиболее естественной для нашей задачи выглядит настройка сети, показанная на рис. 3:

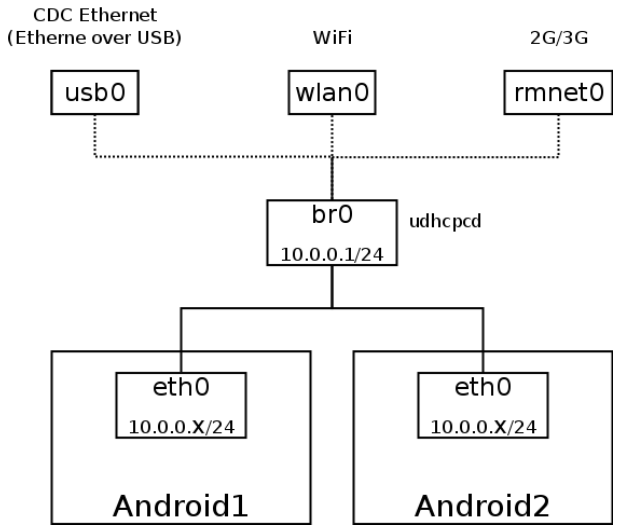

Рис. 3: Схема настройки сети

Скрипт инициализации корневого пользовательского окружения создает виртуальный мост (интерфейс `br0` на рисунке) следующим образом:

```
brctl addbr br0
ip link set br0 up
ip addr add 10.0.0.1/24 dev br0
```

и запускает на нем DHCP-сервер. Мы используем DHCP-сервер `udhcpd`, входящий в состав busybox и конфигурируем его следующим образом:

```
start     10.0.0.10
end       10.0.0.254
interface br0

opt       dns    8.8.8.8
option    subnet 255.255.255.0
```

[20] `lxc/confile.c`, строка 92
[21] `lxc/conf.c`, строка 851, функция `setup_cgroup`

```
opt       router 10.0.0.1
option    lease  864000
```

Для доступа к физической сети могут использоваться USB Gadget (с драйвером, в котором активирована USB-функция RNDIS или USB CDC Ethernet), WiFi-адартер или GSM-модем.

Часть конфигурационного файла LXC, отвечающая за настройку сети, выглядит следующим образом:

```
lxc.network.type=veth
lxc.network.link=br0
lxc.network.flags=up
lxc.network.name=eth0
```

В этом случае LXC при запуске контейнера создаст пару соединенных между собой виртуальных Ethernet-интерфейсов с помощью драйвера `veth`, один из которых будет подключен к виртуальному мосту `br0`, а другой будет перемещен в сетевое пространство имен создаваемого контейнера и ему будет присвоено имя `eth0`.

4 Тестирование и анализ результатов

4.1 Цели тестирования

Основным критерием возможности практического использования разработанной технологии вирутуализации является количество потребляемой ей памяти. Поэтому было принято решение

- определить количество памяти, потребляемой системой при запуске нескольких пользовательских окружений `Android`,
- оценить максимальное количество параллельно работающих контейнеров на смартфоне,
- проанализировать использование памяти программи в контейнере.

Кроме того, необходимо проверить корректность реализации мультиплексирования в сервисе прокси-RIL и диспетчере событий ввода.

4.2 Сценарий тестирования

Измерение количества потребляемой памяти было проведено по следующему сценарию:

1. измерение потребляемой памяти до запуска контейнеров,
2. измерение потребляемой памяти после запуска вспомогательного контейнера — это и предыдущее измерения позволят оценить количество памяти, потребляемой системными компонентами,
3. измерение потребляемой памяти после запуска одного контейнера — это измерение позволит определить количество памяти, потребляемой одним контейнером,
4. измерение потребляемой памяти после запуска второго контейнера — это контрольное измерение, позволяющее выявить аномалии в поведении параллельно работающих контейнеров и в поведении системных компонетов.

В каждом случае статистика потребления памяти пользовательскими программами фиксировалась программой `top`, а также фиксировалось содержимое файла `/proc/meminfo` для определения количества памяти, потребляемой подсистемами ядра.

Измерения проводились на смартфоне `Samsung Galaxy S II`.

4.3 Протоколы тестирования

4.3.1 Потребление памяти до запуска контейнеров

- Вывод программы `busybox top`:

```
Mem: 43332K used, 887640K free, 0K shrd, 2404K buff, 22284K cached
CPU:  0.0% usr  0.7% sys  0.0% nic 99.2% idle  0.0% io  0.0% irq  0.0% sirq
Load average: 0.00 0.00 0.00 1/77 2611
  PID  PPID USER     STAT   VSZ %VSZ CPU %CPU COMMAND
 2611  2609 root     R     3484  0.3   0  0.7 /root/busybox top
 2590  1555 root     S    25948  2.7   0  0.0 avmcp -qws
 2609  2591 root     S     3932  0.4   0  0.0 /system/bin/sh -
 2591  1555 root     S     3384  0.3   0  0.0 /adbd
 1555     1 root     S     1596  0.1   0  0.0 {init} /bin/busybox sh /init
    1     0 root     S     1556  0.1   1  0.0 {init} /bin/busybox sh /init
 1567  1555 root     S      292  0.0   0  0.0 /ueventd
```

- Статистика `/proc/meminfo`:

```
MemTotal:         930972 kB
MemFree:          887624 kB
Buffers:            2404 kB
Cached:            22284 kB
SwapCached:            0 kB
Active:             5516 kB
Inactive:          22084 kB
Active(anon):       2928 kB
Inactive(anon):       52 kB
Active(file):       2588 kB
Inactive(file):    22032 kB
Unevictable:           0 kB
Mlocked:               0 kB
HighTotal:        409600 kB
HighFree:         383052 kB
LowTotal:         521372 kB
LowFree:          504572 kB
SwapTotal:             0 kB
SwapFree:              0 kB
Dirty:                 0 kB
Writeback:             0 kB
AnonPages:          2968 kB
Mapped:             6604 kB
Shmem:                56 kB
Slab:               7836 kB
SReclaimable:       3704 kB
SUnreclaim:         4132 kB
KernelStack:         616 kB
PageTables:          112 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:      465484 kB
Committed_AS:      14220 kB
VmallocTotal:     139264 kB
VmallocUsed:       20100 kB
VmallocChunk:     110460 kB
```

4.3.2 Потребление памяти после запуска вспомогательного контейнера

- Вывод `busybox top`:

```
Mem: 67952K used, 863020K free, 0K shrd, 3596K buff, 40144K cached
CPU:  0.1% usr  0.7% sys  0.0% nic 99.0% idle  0.0% io  0.0% irq  0.0% sirq
Load average: 0.00 0.00 0.00 1/97 3697
  PID  PPID USER     STAT   VSZ %VSZ CPU %CPU COMMAND
 3631  2607 root     S    27176  2.9   0  0.0 /system/bin/mediaserver
 2590  1552 root     S    26792  2.8   0  0.1 avmcp -qws
 3630  2607 root     S    10612  1.1   0  0.0 /ril/proxyserver-ril -d /dev/ttyS0
 3627  2607 root     S <   9764  1.0   0  0.0 /system/bin/audiod
 2602  2591 root     S     3932  0.4   0  0.0 /system/bin/sh -
 3696  2602 root     R     3484  0.3   0  0.5 ./busybox top
 2591  1552 root     S     3380  0.3   0  0.1 /adbd
 2606     1 root     S     1796  0.1   0  0.0 lxc-start -n android3 -d
 1552     1 root     S     1596  0.1   0  0.0 {init} /bin/busybox sh /init
    1     0 root     S     1556  0.1   1  0.0 {init} /bin/busybox sh /init
 3628  2607 1000     S      828  0.0   0  0.0 /system/bin/servicemanager
 3629  2607 root     S      692  0.0   0  0.0 /system/bin/debuggerd
 2607  2606 root     S      356  0.0   0  0.0 /sbin/init
 2608  2607 root     S      304  0.0   0  0.0 /sbin/ueventd
 1564  1552 root     S      292  0.0   0  0.0 /ueventd
```

- Статистика `/proc/meminfo`:

```
MemTotal:         930972 kB
MemFree:          863004 kB
Buffers:            3596 kB
Cached:            40144 kB
SwapCached:            0 kB
Active:             9368 kB
Inactive:          40148 kB
Active(anon):       5788 kB
Inactive(anon):      104 kB
Active(file):       3580 kB
Inactive(file):    40044 kB
Unevictable:           0 kB
Mlocked:               0 kB
HighTotal:        409600 kB
HighFree:         362380 kB
LowTotal:         521372 kB
LowFree:          500624 kB
SwapTotal:             0 kB
SwapFree:              0 kB
Dirty:                 0 kB
Writeback:             0 kB
AnonPages:          5808 kB
Mapped:            12588 kB
Shmem:               124 kB
Slab:               9264 kB
SReclaimable:       4268 kB
SUnreclaim:         4996 kB
KernelStack:         776 kB
PageTables:          680 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:      465484 kB
Committed_AS:      47532 kB
VmallocTotal:     139264 kB
VmallocUsed:       23292 kB
VmallocChunk:     108420 kB
```

4.3.3 Потребление памяти одним контейнером

PID `init`'а — 3741.

- Вывод `busybox top`:

```
Mem: 290444K used, 640528K free, 0K shrd, 10908K buff, 118948K cached
CPU:  0.3% usr  1.5% sys  0.0% nic 98.0% idle  0.0% io  0.0% irq  0.0% sirq
Load average: 0.43 0.33 0.14 1/381 5362
  PID  PPID USER     STAT   VSZ %VSZ CPU %CPU COMMAND
 4907  4830 1000     S     198m 21.7   0  0.0 system_server
 5058  4830 10017    S     147m 16.1   0  0.0 {d.process.acore} android.process.
 5015  4830 1001     S     142m 15.6   0  0.0 {m.android.phone} com.android.phon
 5001  4830 10009    S     141m 15.4   0  0.0 {ndroid.launcher} com.android.laun
 5207  4830 10022    S     139m 15.2   0  0.0 com.android.mms
 5009  4830 10016    S     136m 14.9   0  0.0 {ndroid.musicvis} com.android.musi
 5010  4830 10038    S     134m 14.7   0  0.0 {putmethod.latin} com.android.inpu
 4991  4830 1000     S     132m 14.4   0  0.2 {ndroid.systemui} com.android.syst
 5174  4830 10007    S     131m 14.4   0  0.0 {d.process.media} android.process.
 5261  4830 10003    S     130m 14.2   0  0.0 {.cooliris.media} com.cooliris.med
 5249  4830 10044    S     130m 14.2   0  0.0 {m.android.email} com.android.emai
 5222  4830 10023    S     128m 14.0   0  0.0 {droid.deskclock} com.android.desk
 5079  4830 1000     S     128m 14.0   0  0.0 {ogenmod.cmparts} com.cyanogenmod.
 5178  4830 10013    S     128m 14.0   0  0.0 {viders.calendar} com.android.prov
 5117  4830 10024    S     127m 13.9   0  0.0 {.quicksearchbox} com.android.quic
 5155  4830 10032    S     127m 13.9   0  0.0 {roid.dspmanager} com.bel.android.
 5126  4830 10014    S     127m 13.9   0  0.0 {m.android.music} com.android.musi
 5166  4830 10000    S     127m 13.9   0  0.0 {droid.bluetooth} com.android.blue
 5232  4830 10025    S     126m 13.8   0  0.0 {oid.voicedialer} com.android.voic
 5106  4830 10033    S     126m 13.8   0  0.0 {t.cactii.flash2} net.cactii.flash
```

- Статистика `/proc/meminfo`:

```
MemTotal:         930972 kB
MemFree:          640544 kB
Buffers:           10908 kB
Cached:           118948 kB
SwapCached:            0 kB
Active:           128636 kB
Inactive:         111656 kB
Active(anon):     110488 kB
Inactive(anon):      420 kB
Active(file):      18148 kB
Inactive(file):   111236 kB
Unevictable:           0 kB
Mlocked:               0 kB
HighTotal:        409600 kB
HighFree:         172832 kB
LowTotal:         521372 kB
LowFree:          467712 kB
SwapTotal:             0 kB
SwapFree:              0 kB
Dirty:                 0 kB
Writeback:             0 kB
AnonPages:        110468 kB
Mapped:            49952 kB
Shmem:               480 kB
Slab:              17644 kB
SReclaimable:       9288 kB
SUnreclaim:         8356 kB
KernelStack:        3048 kB
PageTables:         8788 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:      465484 kB
Committed_AS:    1504860 kB
VmallocTotal:     139264 kB
VmallocUsed:       48988 kB
VmallocChunk:      80772 kB
```

4.3.4 Потребление памяти двумя контейнерами

PID `init`'а первого контейнера — 3741, PID `init`'а второго контейнера — 5375.

- Вывод `busybox top`:

```
Mem: 503068K used, 427904K free, 0K shrd, 16192K buff, 209228K cached
CPU:  1.3% usr  2.7% sys  0.0% nic 95.8% idle  0.0% io  0.0% irq  0.0% sirq
Load average: 1.44 0.62 0.28 2/643 6976
  PID  PPID USER     STAT   VSZ %VSZ CPU %CPU COMMAND
 6565  6470 1000     S     199m 21.7   0  0.5 system_server
 4907  4830 1000     S     198m 21.7   0  0.1 system_server
 6665  6470 10001    S     143m 15.6   0  0.0 {droid.wallpaper} com.android.wall
 6661  6470 10009    S     142m 15.6   0  0.0 {ndroid.launcher} com.android.laun
 5015  4830 1001     S     142m 15.5   0  0.0 {m.android.phone} com.android.phon
 6673  6470 1001     S     141m 15.5   0  0.0 {m.android.phone} com.android.phon
 6598  4830 10009    S     140m 15.3   0  0.0 {ndroid.launcher} com.android.laun
 5207  4830 10022    S     139m 15.2   0  0.0 com.android.mms
 6857  6470 10022    S     137m 15.0   0  0.0 com.android.mms
 5174  4830 10007    S     136m 14.9   0  0.1 {d.process.media} android.process.
 5009  4830 10016    S     136m 14.9   0  0.0 {ndroid.musicvis} com.android.musi
 5010  4830 10038    S     134m 14.7   0  0.0 {putmethod.latin} com.android.inpu
 6666  6470 10038    S     134m 14.7   0  0.0 {putmethod.latin} com.android.inpu
 5261  4830 10003    S     134m 14.7   0  0.0 {.cooliris.media} com.cooliris.med
 4991  4830 1000     S     133m 14.5   0  0.1 {ndroid.systemui} com.android.syst
 6817  6470 10007    S     132m 14.5   0  0.0 {d.process.media} android.process.
 6652  6470 1000     S     132m 14.4   0  0.3 {ndroid.systemui} com.android.syst
 6741  6470 1000     S     131m 14.4   0  0.0 {ndroid.settings} com.android.sett
 6910  6470 10003    S     130m 14.2   0  0.0 {.cooliris.media} com.cooliris.med
 5249  4830 10044    S     130m 14.2   0  0.0 {m.android.email} com.android.emai
```

- Статистика `/proc/meminfo`:

```
MemTotal:         930972 kB
MemFree:          428012 kB
Buffers:           16200 kB
Cached:           209204 kB
SwapCached:            0 kB
Active:           231056 kB
Inactive:         186348 kB
Active(anon):     192116 kB
Inactive(anon):      712 kB
Active(file):      38940 kB
Inactive(file):   185636 kB
Unevictable:           0 kB
Mlocked:               0 kB
HighTotal:        409600 kB
HighFree:          12116 kB
LowTotal:         521372 kB
LowFree:          415896 kB
SwapTotal:             0 kB
SwapFree:              0 kB
Dirty:                 0 kB
Writeback:             0 kB
AnonPages:        192064 kB
Mapped:            83972 kB
Shmem:               812 kB
Slab:              25956 kB
SReclaimable:      14652 kB
SUnreclaim:        11304 kB
KernelStack:        5144 kB
PageTables:        16524 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:      465484 kB
Committed_AS:    2881380 kB
VmallocTotal:     139264 kB
VmallocUsed:       73776 kB
VmallocChunk:      51972 kB
```

4.4 Анализ результатов измерений

Из приведенных в предыдущем разделе измерений можно сделать следующие выводы:

- Объем памяти, занимаемой сервисными программами — 50 Мб, причем панель управления и вспомогательный контейнер потребляют примерно одинаковый объем памяти.
- Один контейнер после инициализации пользовательского окружения (CyanogenMod 7) потребляет примерно 210 Мб, причем наибольшее количество памяти потребляет процесс `system_server` (200 Мб), в контексте которого работают все нативные службы `Android`'а. Результат анализа памяти этого процесса приведен в таблице

Параметр	Размер
исполняемый код	23 Мб
секции данных	1 Мб
анонимные отображения, доступные для чтения и записи	62 Мб
прочие анонимные отображения	212 Кб
прочие отображения	116 Мб

Анализ памяти, занятой прочими отображениями, приведен в следующей таблице:

Параметр	Размер
отображенные jar-файлы	16 Мб
память устройств	95 Мб
в т. ч. ashmem	80 Мб

Отображения ashmem, размеры которых превосходит 1 Мб:

Файл	Размер
`/dev/ashmem/dalvik-heap (deleted)`	56 Мб
`/dev/ashmem/dalvik-heap (deleted)`	7 Мб
`/dev/ashmem/dalvik-LinearAlloc (deleted)`	2 Мб
`/dev/ashmem/dalvik-LinearAlloc (deleted)`	2 Мб

Таким образом, наибольшее количество памяти в Android'е резервируется под кучу виртуальной машины (56 Мб) и под, предположительно, нативную кучу (62 Мб).

- Технически на смартфоне `Samsung Galaxy S II` можно запустить до четырех контейнеров. Однако на смартфонах, на которых установлено меньше 512 Мб ОЗУ (в частности, Google Nexus S), невозможен запуск более одного контейнера без активации swap'а.

5 Заключение

5.1 Выводы

В настоящее время в рамках проекта AndroidVM получен рабочий прототип решения для виртуализации пользовательского окружения Android, позволяющего запускать несколько пользовательских окружений на одном устройстве и реализующее одновременный доступ из разных контейнеров к устройствам ввода и телефонии.

5.2 Недостатки и направление дальнейших работ

Текущее решение имеет следующие недостатки:

- Решение не позволяет задавать список устройств, доступных контейнеру. Эта возможность уже заложена в ФС cgroup. Доработка панели управления контейнерами позволит получить удобный интерфейс для задания списка доступных устройств.
- Решение не позволяет динамически ограничивать доступ к сервисам телефонии, предоставляемым проприетарной реализацией RIL. Для преодоления этого недостатка потребуется серьезно доработать прокси-сервер RIL, в частности, необходимо:
  - расширить внутренне представление клиента RIL так, чтобы в нем можно было хранить политики доступа к определенным группам функций интерфейса RIL;
  - разработать протокол управления этими политиками;
  - разработать графический интерфейс для доступа к этому протоколу.
- Процесс в контейнере может вызвать перезагрузку системы из-за того, что он имеет полномочие CAP_SYS_BOOT. Поэтому системный вызов sys_reboot, сделанный из контейнера, должен приводить к уничтожению первого процесса в этом контейнере, а не к перезагрузке всей системы.
- Не реализована инициализация подключения к мобильной сети передачи данных через интерфейс RIL — в настоящий момент на все такие запросы прокси-сервер RIL возвращает ошибку.
- Не выяснена причина критической ошибки в ядре при остановке контейнера — эта проблема не воспроизводится на Android-эмуляторе, а записи об ошибке не удалось прочитать из лога ядра — сетевая подсистема, по-видимому, отказывает раньше, чем ядро перезагружает машину[22].
- Не найдено удовлетворительное решение проблемы с нехваткой памяти на смартфоне Google Nexus S:
  - Удалось задействовать swap-файл, но он размещен на разделе, к которому имеют доступ пользовательские контейнеры — переразметить внутреннюю MMC-карту смартфона не удалось из-за того, что загрузчик второй ступени[23] восстанавливает таблицу разделов на накопителе; найти делающий это код в загрузчике не удалось.
  - Не удалось задействовать демон слияния страниц в ядре Linux для уменьшения потребления памяти — ему необходимо указывать области, которые следует сканировать в поисках страниц для слияния, но эту информацию ему никто не может предоставить, поскольку системные библиотеки разных Android-контейнеров расположены в разных частях файловой системы.

    Решением может служить использование единого каталога с библиотеками и системными программами для всех пользовательских контейнеров. Это позволит сэкономить как минимум 23 Мб оперативной памяти.
- Не решена проблема с настройкой WiFi в корневом пользовательском окружении: пока даже не удается вручную загрузить драйвер WiFi-адаптера в ядро[24]

[22] http://osll.spb.ru/issues/3212
[23] SBL — second boot loader
[24] http://osll.spb.ru/issues/2938

- Не смотря на то что SD-карта примонтирована в контейнере и, например, файловый менеджер ее использует, некоторые другие приложения сообщают о ее отсутствии.
- Не виртуализирована подсистема Bluetooth.
- Необходимо выработать политику доступа к часам реального времени через драйвер Alarm. В данный момент контейнеры могут устанавливать время глобально для всей системы.

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.