

论文题目 基于 LXC 的 Android 系统虚拟化关键技术设计与实现

作者姓名 吴佳杰

指导教师 陈文智教授

学科(专业) 计算机应用技术

所在学院 计算机学院

提交日期 2014 年 1 月 6 日

独创性声明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得 浙江大学 或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在我论文中作了明确的说明并表示谢意。

学位论文作者签名: 吴佳杰 签字日期: 2014 年 3 月 9 日

学位论文版权使用授权书

本学位论文作者完全了解 浙江大学 有关保留、使用学位论文的规定，有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人授权 浙江大学 可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

(保密的学位论文在解密后适用本授权书)

学位论文作者签名: 吴佳杰

导师签名:

签字日期: 2014 年 3 月 9 日

签字日期: 2014 年 3 月 9 日

学位论文作者毕业后去向:

工作单位:

电话:

通讯地址:

邮编

密级：_____

浙江大学

硕士 学位 论文



论文题目 基于 LXC 的 Android 系统虚拟化关键技术设计与实现

作者姓名 吴佳杰
指导教师 陈文智教授
学科(专业) 计算机应用技术
所在学院 计算机学院
提交日期 2014 年 1 月 6 日

A Dissertation Submitted to Zhejiang
University for the Degree of
Master of Engineering



TITLE: The Design and Implementation of
Key Technology For Android System
Virtualization Based On LXC

Author: Wu Jiajie
Supervisor: Prof. Chen Wenzhi
Subject: Computer Application
College: Computer Science
Submitted Date: January 6th 2014

摘要

随着 Android 移动设备的普及, Android 系统的开放性吸引了越来越多的硬件与软件开发厂商, Android 应用市场也随之繁荣, 各具特色的应用软件满足了不同人的软件消费需求。同时 Android 系统的开放性也导致了恶意应用软件的泛滥, 威胁 Android 用户的隐私安全。为了增强 Android 系统的安全性, 本文在 Android 系统中引入虚拟化技术, 通过 Android 虚拟化后带来的隔离性来增强用户隐私的安全。

然而, 在虚拟化技术在高性能的服务器上得到了广泛应用并且日趋成熟, 但在计算处理能力较弱的 Android 移动设备上, 目前的虚拟化方案较少。因此, 针对 Android 系统特点, 本文提出了一种基于 lxc 的 Android 系统虚拟化方案, 该方案将 lxc 工具移植到 Android 系统中, 然后将 Android 客户机运行于 lxc 创建的容器下, 从而实现了多个 Android 系统共用一个 Linux 内核, 完成了 Android 系统的操作系统级虚拟化, 同时还设计了共享服务用于降低 Android 虚拟化带来的内存消耗。为了实现本方案, 本文完成了如下工作:

- (1)研究并分析了当前 Android 系统虚拟化技术, 调研了业界内相关实现, 为本文的 Android 系统虚拟化设计和实现提供参考;
- (2)分析了 Android 系统虚拟化所面临的问题, 介绍了使用 lxc 工具进行操作系统级虚拟化的优势, 同时也分析了 lxc 工具的实现原理;
- (3)设计并实现了 Android 虚拟化原型系统, 主要完成了 lxc 工具的移植, Binder 驱动的虚拟化, 以及输入与显示显示输出设备的复用;
- (4)通过功能测试, 验证了本虚拟化系统的可行性与功能完整性, 可通过虚拟化隔离多个 Android 系统增强安全性, 同时也通过性能测试统计了 Android 系统虚拟化后内存使用情况, 分析数据说明了本设计方案的优势。

关键词: Android, 操作系统级虚拟化, LXC

Abstract

With the popularity of Android mobile devices, the openness of the Android system has attracted more and more hardware manufacturers and software developers. Android application market is also prosperous, the distinctive applications meet different consumer's demand, but openness of Android system also led to the spread of malicious software applications, this threats Android user's privacy. In order to enhance the security of the Android system, we introduce virtualization technology in the Android system, Android system virtualization bring isolation which can protect user's privacy.

However, the current virtualization technology mainly used in server with high performance, server virtualization technology become matures, but there are less virtualization solutions on the Android device with weak computing power hardware. Therefore, for the Android system, we develop a virtualization solution based on LXC. We port LXC tools to the Android system, then Android virtual machine can run on the container which is created by LXC, this solution implements a number of Android system share one Linux kernel, which called operating system level virtualization. We also design sharing service to reduce the memory consumption. For this program, we complete flowing tasks:

- (1) We research the current Android system virtualization technology and research related works of the industry, which provide a reference for the Android system virtualization design and implementation;
- (2) We analyze the problems faced in Android system virtualization, introduce the use of LXC tools for operating system-level virtualization advantages, we also analyze the realization of the principle LXC tools;
- (3) We design and implement Android virtualization system, we mainly complete the transplant LXC tools, Binder driven virtualization, and input and output devices for reuse;
- (4) We verify the feasibility and functional integrity of the virtualization system

through functional testing, it prove that multiple Android systems can be isolated to enhance security. Also through performance testing, we get the memory usage statistics which prove the advantage of our design.

Keywords: Android, OS level virtualization, LXC, virtualized driver

目录

摘要	i
Abstract.....	ii
第 1 章 绪论	1
1. 1 课题背景	1
1. 2 本文工作内容	2
1. 3 本文的工作与创新点	3
1. 4 本文的内容组织	3
1. 5 本章小结	4
第 2 章 Android 系统与虚拟化技术概述	5
2.1 Android 系统架构介绍	5
2.2 Android 系统安全技术介绍	7
2.2.1 Android 系统和内核级安全	7
2.2.2 Android 应用程序安全	8
2.2.3 Android 安全隐患	9
2.3 虚拟化技术介绍	10
2.3.1 指令级虚拟化	11
2.3.2 硬件级虚拟化	11
2.3.3 操作系统级虚拟化	12
2.3.4 编程语言级虚拟化	12
2.4 Android 虚拟化现状	13
2.5 本章小结	15
第 3 章 Android 系统虚拟化关键技术研究	16
3.1 Android 系统虚拟化设计思路	16
3.1.1 Android 虚拟化面临的问题	16

3.1.2 Android 虚拟化技术的选择	17
3.2 LXC 工具研究	18
3.2.1 LXC 命令介绍	18
3.2.2 LXC 容器隔离实现机制研究	19
3.3 本章小结	27
第 4 章 Android 系统虚拟化设计与实现	28
4.1 总体设计	28
4.1.1 设计目标	28
4.1.2 总体设计方案	28
4.2 Android 下 LXC 工具移植	30
4.3 输入设备与显示输出设备复用	33
4.3.1 Linux 控制终端 tty 简介	33
4.3.2 显示输出设备复用	34
4.3.3 输入设备复用	37
4.4 Binder 驱动复用	39
4.4.1 Binder 驱动简介	39
4.4.2 Binder 驱动的设计	40
4.4.3 Binder 驱动的实现	42
4.5 本章小结	44
第 5 章 系统功能测试与性能演示	46
5.1 测试环境部署	46
5.2 功能测试	47
5.3 性能测试	51
5.4 本章小结	53
第 6 章 总结与展望	54
6.1 本文工作总结	54
6.2 工作展望	54

参考文献	56
攻读硕士学位期间主要的研究成果	58
致谢	59

图目录

图 2-1 Android 系统架构.....	5
图 2-2 Cells 系统架构.....	13
图 2-3 MVP 系统架构	14
图 3-1 命名空间层次关系.....	21
图 3-2 cgroup 层级关系.....	24
图 3-3 Cgroup 实现代码结构关系	26
图 3-4 Cgroup 文件创建流程.....	27
图 4-1 Android 系统虚拟化总体架构.....	29
图 4-2 Android 文件系统视图.....	32
图 4-3 输入与显示输出设备工作流程图	34
图 4-4 显示输出设备复用设计原理图	35
图 4-5 Android 启动部分流程.....	36
图 4-6 屏幕切换流程.....	37
图 4-7 输入系统结构图	38
图 4-8 getEvent 函数流程图.....	39
图 4-9 Binder 驱动架构	40
图 4-10 新的 Binder 驱动架构	41
图 4-11 服务名过滤转换流程图	43
图 4-12 修改服务名称流程	44
图 5-1 lxc-checkconfig 运行效果	48
图 5-2 Android 宿主机系统信息.....	49
图 5-3 Android 客户机系统信息.....	49
图 5-4 Android 宿主机和客户机程序运行效果.....	50
图 5-5 RE 管理器访问文件系统效果	51

图 5-6 Android 系统内存使用示意图 52

图 5-7 Android 应用 cpu 使用率 53

表目录

表 2-1 Android 应用框架层组件	6
表 3-1 LXC 常用命令	18
表 3-2 Cgroup 各子系统作用	24
表 4-1 支持 LXC 的 Linux 编译配置选项	30
表 4-2 Binder 驱动接口函数	42
表 5-1 硬件参数配置	46

第1章 绪论

1.1 课题背景

随着个人电脑市场的成熟饱和，个人计算机已经进入到后 PC 时代，这一时代的重要特点就是移动计算设备的普及，尤其是近几年，智能手机销量已经远远超过 PC 的销量，同时移动设备的硬件能力越来越强大，比如一款普通的智能机已经配备 1.3GHz 的双核处理器，1GB 内存和 16GB 的存储空间，这样的硬件配置可以与十年前的 PC 相当。在智能手机中的操作系统中，Android 系统装机量最高，占据了 80% 左右的市场。随着 Android 智能机增长^[1]，Android 应用也开始爆发式增长用于满足不同用户的需求。

Android 系统平台^[2]具有自己的独特优势，最重要的就是开放性，任何移动设备厂商都可以加入 Android 的联盟，各个硬件厂商的竞争使得 Android 硬件配置越来越高而价格却越来越低，这样吸引了更多的消费者去购买 Android 智能机，反过来促进了 Android 的普及。同时，平台的开放性吸引了更多的应用开发者，Android 用户获得了更丰富的软件资源。

现在，通过 Android 手机除了进行基本的打电话和发短信，还可以上网，发邮件，甚至可以转账付款，这些操作都可能涉及到个人的隐私问题，而很多恶意软件厂商利用 Android 系统的开放性特点，利用漏洞制造了很多恶意应用软件，用户下载安装这些软件后会使 Android 手机感染，这些恶意软件有的通过植入了恶意的代码或广告偷偷扣取通信费，有的偷偷收集用户的通信录名单后卖给第三方公司，进而更广泛的影响其它用户的通信安全，比如垃圾短信和骚扰电话，所以越来越多的用户关注 Android 的隐私安全性。

虽然 Android 系统本身设置了相应的安全规则，比如用户可以在安装应用时查看应用需要的权限来选择是否安装该应用，但对于一般用户而言并没建立起安全权限设置的概念，这使得 Android 系统设定的安全规则无法得到应用。通过

第三方的安全软件检测可以起到安全预防的作用，但是安全软件具有滞后性，无法防御最新的恶意应用，所以安全软件也并不能全面防范恶意应用。

针对上述 Android 的安全问题，本课题引入了虚拟化，利用虚拟化中隔离性特点来解决 Android 系统的安全问题，但传统的虚拟化方案主要针对服务器，服务器硬件具有较高的运算能力和内存，但 Android 系统对应的硬件能力与之相比弱很多，所以针对 Android 系统需要设计一种新的 Android 虚拟化方案，这就是本课题的背景。

1.2 本文工作内容

根据课题研究的背景，本文主要工作内容是实现一个 Android 虚拟化系统，具体内容如下：

(1)Android 系统架构与安全机制研究

实现 Android 系统之前，本文需要全面了解 Android 系统的架构，分析了 Android 系统的架构层次，同时也研究了目前 Android 平台使用的安全机制，指出了 Android 系统安全方面的不足之处。

(2)相关虚拟化技术研究

本文调研了目前在虚拟化技术相关的技术方案，从不同的层次介绍了虚拟化的优缺点。同时介绍了目前 Android 虚拟化业界内的实现方案，了解目前业界内实现方案的特点。

(3)lxc 工具实现原理分析

本文使用的了 lxc 工具作为本方案的基础工具，我们分析了 lxc 工具的使用方式，从内核层和应用层介绍了 lxc 工具的实现方式。

(4)设计和实现了 Android 系统的虚拟化系统

本文设计并实现了一个基于 lxc 的 Android 虚拟化系统，主要包括 lxc 工具的移植，Binder 驱动的复用，输入与显示输出设备的复用。

(5)测试并验证了 Android 虚拟化系统

测试主要是功能测试，也进行了简单的性能测试，验证了系统功能的完整性

与本设计方案的优点。

1.3 本文的工作与创新点

在本课题中，我们设计了一种新型的 Android 虚拟化方案，与目前其它 Android 虚拟化方案相比有明显的优势，本文的主要创新点包含以下方面：

- 设计并实现了 Android 系统操作系统级虚拟化

本课题使用 lxc 工具创建容器，将 Android 客户机系统运行于容器中，使多个 Android 系统共享同一个 Linux 内核，实现了轻量级的操作系统级虚拟化，简化了 Android 系统虚拟化的层次结构，减少了虚拟化带来的性能损失。该设计方案非常适用于 Android 智能手机这种计算资源有限的设备上。

- 输入与显示输出设备的复用

本课题通过修改 Android 系统，实现了多个 Android 系统共享使用底层输入与显示输出驱动设备。该方法的特点是不需要修改当前输入与显示输出设备驱动用于满足多个 Android 系统的使用，而是修改 Android 系统上层的输入与显示输出模块，减少了虚拟化后底层硬件驱动的开发工作，屏蔽了不同硬件之间的差异。

- Android 系统 Binder 驱动

本课题修改了 Binder 驱动，保证了虚拟化后多个 Android 系统的正常通信，确保了不同 Android 系统之间的服务能够正常使用。

1.4 本文的内容组织

本文的组织架构如下：

第1章是本文绪论

本章介绍了课题的研究背景，分析了本课题的意义，然后介绍了本文工作的主要内容，最后总结了本文的创新点。

第2章是相关Android虚拟化技术概述

本章介绍了 Android 系统的架构层次和安全机制，然后对当前使用的虚拟化技术进行了介绍分析，最后介绍了 Android 系统虚拟化业界实现情况。

第3章是Android虚拟化关键技术研究分析

本章分析了Android虚拟化需要解决的问题，比较了不同虚拟化技术的优缺点，确立了使用lxc进行虚拟化的方案，最后分析了lxc工具的实现原理。

第4章是Android虚拟化系统的设计与实现

本章首先做了Android系统的总体设计，介绍了系统的框架，然后对lxc移植，binder驱动和输入显示设备复用三个方面给出了设计方案和实现方法。

第5章是测试部分

本章首先介绍了Android虚拟化测试平台的部署，然后重点介绍了功能测试内容，功能测试用于验证虚拟化系统功能的完整性，最后是性能测试并分析了测试数据。

第6部分是本文总结与展望

本章对本文的工作进行了总结，同时指出了目前设计方案与工作的不足之处，为后续改进工作指明了方向。

1.5 本章小结

本章介绍了课题的研究背景，介绍了本文的主要工作内容，然后介绍了本课题的创新点，最后介绍了本论文组织结构。

第2章 Android 系统与虚拟化技术概述

Android 系统和虚拟化技术是研究本文课题最重要的基础知识，在本章概述中对这两项技术做详细的介绍，对于 Android 系统，主要介绍 Android 系统架构和系统安全方面的设计，虚拟化是本论文的重点，主要介绍目前使用的虚拟化技术，最后本章介绍 Android 系统虚拟化在业界内的实现。

2.1 Android 系统架构介绍

实现 Android 虚拟化需要对 Android 的系统的架构^[3]有所了解，本节将对 Android 系统架构进行介绍，主要介绍 Android 系统架构中各模块层次的功能及关系。下图 2-1 所示是 Android 系统的架构。

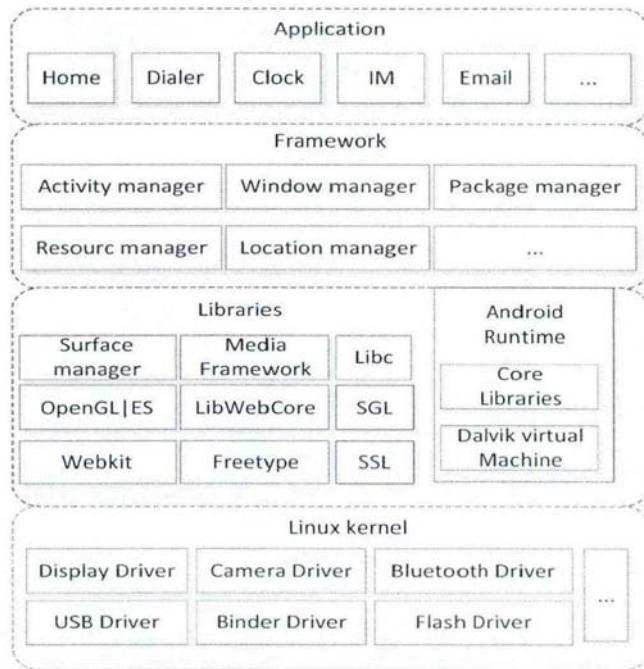


图 2-1 Android 系统架构

从上图 2-1 可以知道，Android 系统分为四层，下面我们将对这四层模块进行介绍：

1) 应用层。Android 系统一般安装后会再带一些应用程序, 如联系人管理程序, google 地图, 日历, 拨号, 短消息程序等, 用户自己可以选择相关的 Android 应用安装, 所有的应用都是使用 java 语言编写, 通过调用下一层应用程序框架提供的 API 完成相应功能, 应用层处于整个系统的最上层。

2) 应用程序框架层^[4]。这一层是为上层软件开发提供了相应的 API 框架, 该层模块以服务组件的形式存在, 这样的设计减少了代码的重用, 同时应用程序还可根据自己的需求替换组件, 实现自己的定制, 该框架组件主要包括以下几类:

表 2-1 Android 应用框架层组件

名称	功能
视图	可构建一个应用程序, 包括列表, 文本框, 按钮等基本图形
内容提供器	使应用程序可访问另一个应用程序数据, 如联系人数据库, 或者共享应用程序自身的数据
资源管理器	提供资源给应用程序使用, 如字符串资源, 音频图像资源等
通知管理器	使应用程序在状态栏中能够显示自定义的提示信息
活动管理器	管理应用程序生命周期并提供导航回退功能, 如退出应用等

3) 系统运行库层。该层包括了程序库和 Android 运行库, 前者包含一些 C/C++ 库, 这些库提供给应用框架层的不同组件使用, 为上层应用框架的开发者提供了服务, 主要包括了音视频相关的 Media Framework, 轻型数据库 SQLite, 3D 图形引擎 OpenGL ES, 2D 图形引擎 SGL, Webkit, Libc 库等; 后者包括了 Dalvik 虚拟机与核心库, 它依赖于底层 Linux 系统的内存和线程管理机制。核心库提供 java 语言相关的功能, 通过 JNI 方式向上层应用程序框架层提供了丰富的接口。

4) Linux 内核层。Linux 内核是软件和硬件之间的中间层, 用于驱动管理, 内存管理, 进程管理等, 在 Android 中, Linux 内核针对移动设备的硬件特性进行了优化, 主要包括电源管理, Android IPC 机制, 低内存管理, 内存分配管理, 目前最新的 Android4.4 系统采用的 Linux 3.8 内核。

通过上述介绍, 我们对 Android 整个系统的架构有所了解, 整个架构层次清

晰，每个功能模块明确，如果开发者进行 Android 系统虚拟化工作，能够较快的定位到相应的代码进行修改。

2.2 Android 系统安全技术介绍

Android 移动操作系统是开源的，Android 的开放性可以使各大手机厂商可以设计研发出最新的硬件，并且能够保证在 Android 平台上的应用能够兼容，但是开放性同时会带来安全问题，所以在设计时 Android 系统为保护消费者隐私，数据及硬件设备的可靠性做了一套安全保障环境。Android 设计了一套强大的安全架构及严格的安全应用，采用了多层次的安全设计，该设计保护其平台的开发性的同时也提供了灵活性。Android 系统的安全设计为开发者减少了负担，开发者能够灵活的使用其安全控制机制。

对于 Android 系统，为了实现保护用户数据，保护系统资源和提供应用程序隔离三个目标，Android 提供了如下五个关键的安全功能^[5]：

1. Linux 内核本身在操作系统级别的强大的安全
2. 强制应用程序使用沙盒技术
3. 安全的进程间通信
4. 应用程序数字签名
5. 用户授予应用程序定义的权限

2.2.1 Android 系统和内核级安全

在底层操作系统级别，Android 系统使用了安全的 Linux 内核，以及一个安全的进程间通信(IPC)机制。基于 Linux 内核主要使用了沙箱技术对应用程序进行限制，可以防止流氓应用程序损害其他应用程序。

- Linux 内核安全技术^[6]，Android 系统内核是 Linux 系统，使用的是安全增强型 SELinux^[7]，Linux 系统本身得到了广泛使用及不断的升级完善，已经成为一个稳定和安全的内核。Linux 内核对 Android 提供了下面四方面的关键安全功能，第一，基于用户的权限模型；第二，进程隔离；第三，

可扩展的安全的进程间通信机制；第四，移除 Linux 内核中对 Android 系统构成安全威胁的部分。作为多用户操作系统，Android 中 Linux 内核基本的安全目标是彼此间隔离用户资源，保护用户资源。

- 应用程序使用了内核级的沙盒技术，从两方面进行了隔离，第一，用户 ID 权限控制，每个 Android 应用都有一个 Linux 用户 ID(UID)，该 UID 会有相应的权限。第二，通过对创建的文件加入用户的标识进行文件访问控制，一个应用程序文件正常情况下不能被其它程序访问，比如进程 A 无法访问进程 B 的文件。权限决定了 Android 系统用户或程序的执行操作是否被允许，只有拥有相应的权限程序才能够访问数据文件、通信和调用 Android 系统组件等。
- 进程间通信，进程可以使用任何传统的 UNIX 风格的机制进行通信，例如 sockets，信号量，消息队列等。Android 也提供了新的 IPC 机制：1. Binder^[8]:轻量级的远程调用通信机制，Binder 使不同进程间进行调用通信时能够高效运行，Binder 的实现基于常用的 Linux 驱动；2. Services:提供使用 Binder 进行通信时对应的服务接口；3. Intents:一个简单的消息对象，表示想要做的事情，可以用于在系统范围内广播感兴趣的事件。

2.2.2 Android 应用程序安全

在应用程序层，Android 设计了数字签名机制用于标志开发者和应用程序之间的关系，保护开发者利益，也确保应用程序的安全。另外在应用程序访问特殊的资源时，需要通过相应的权限检查机制才能够完成。

- 数字签名机制。每个安装到 Android 系统中的应有程序都有一个数字证书，如果应用程序没有可用的数字签名，系统不许安装此程序。数字签名由应用程序开发者在程序完成时生成，不需要通过指定的机构获得，是应用程序包的自我认证。数字签名机制是针对应用开发者而不是用户，比如有恶意应用想替换现有的应用程序，需要通过已安装应用程序的数字证书才能安装，所以说数字证书从 Android 应用开发时就开始保证

Android 系统的安全。

- 用户授予应用程序权限。Android 应用程序默认只能访问有效范围的系统资源，任何应用程序访问特殊资源前需要向 Android 系统发出申请，通过定义受限的 API 函数接口可以实现敏感资源的保护，受保护 API 包括：1. 摄像功能；2. 本地数据(GPS)；3. 蓝牙功能；4. SMS/MM 功能；5. 网络/数据连接。这些资源只能通过底层 Linux 系统访问，应用程序调用这些敏感 API 需要在它的 Manifest 文件中定义需要的能力，安装应用程序时，系统向用户提供应用程序的权限要求并询问是否继续安装，用户继续安装则表明用户已经授权，应用程序就可以访问相应的资源。应用程序安装之后，用户还可以通过设置查看权限，可以关闭部分功能，比如 GPS，Wifi 等。

2.2.3 Android 安全隐患

Android 系统从底层 Linux 内核到上层应用都加入了各种安全措施，保证 Android 手机的安全性，看似完美的系统，但还是存在着安全问题^[9]。

- 基于 Linux 内核的攻击。由于 Android 系统是开源的，任何人都可以得到其源代码并进行编译产生镜像，即 ROM 文件，该文件可以安装到手机中，就像系统重装一样，即刷机。由于刷机可以改变系统界面，增加新的功能，获得不一样的用户体验，越来越多的人开始刷机并且获取 Android 系统的最高权限，现在市场上的 ROM 越来越多，随意刷机存在很大的风险，一些厂商会在制作 ROM 时，修改内核，植入恶意软件推送广告甚至病毒应用，获取自己的利益。程序获得 ROOT 权限是在对 Linux 内核最大的威胁，程序可以以 ROOT 身份访问系统文件，自动安装各种恶意程序，获取个人隐私。同时 Linux 内核本身存在着一些漏洞，比如整数溢出漏洞，经常会出现 Linux 的漏洞列表中，该漏洞可能会导致程序权限的提升，任意代码的执行。
- 基于应用程序的攻击。Android 软件市场多样，所以用户自己安装的程序

无法验证安全和可靠性。在安装的过程中，一般用户无法分辨出恶意程序，忽略权限的检查，这样就弱化了上述其中一个核心安全机制—应用程序权限用户授权。恶意程序还可以通过伪造数字签名，获得更高的权限，获取自己的商业价值，比如恶意扣费，损害用户的利益。

综上所述，Android 系统在安全方面做了很多的设计，从 Linux 内核层，Android 系统层，应用层再到用户授予权限，多方位的保护用户的隐私和利益，但 Android 系统在智能手机 80%的覆盖率及其开源的特点，恶意程序越来越多，用户的隐私，数据等受到越来越大的威胁，如果想更好的增强 Android 系统安全性，增强现有的安全机制已经非常困难，且上一节所描述的五个关键安全机制任何一点出现问题，都有可能被恶意代码利用，所以我们需要一种新的技术方案增强 Android 手机的安全性，这里我们想到了虚拟化，下一节将介绍现有虚拟化的方案，认识虚拟化的作用。

2.3 虚拟化技术介绍

计算机经过这么多年的发展，软件和硬件种类越来越多，整个系统也越来越难管理，于是虚拟化技术出现。虚拟化技术^[10]是对计算机系统进行了抽象，即加入了虚拟化层，该层将计算机系统的硬件资源进行管理然后向上层提供统一的接口。加入虚拟化技术后，不同系统之间的硬件差异被屏蔽，比如安装的操作系统可以不用关心真正的处理器指令集，按照自己系统的方式执行后由虚拟层转换，即上层软件可以直接运行在相应的虚拟平台上。虚拟化技术的引入，打破了计算机系统中软硬件紧密耦合的关系，为计算机带来了新的活力，解决了很多问题。比如在同一个物理机上能够同时运行不同发行版本的 Linux 系统，这样就可以屏蔽了操作系统的差异，同时也充分利用了物理资源。

计算机系统发展至今，本身采用了分层结构，所以目前虚拟化技术解决方案可以在不同层次上实现，不同的实现方式和抽象层次，虚拟化有不同的特性，从实现的技术上主要分为四种^[11]，指令级虚拟化，硬件级虚拟化，操作系统级虚拟化和编程语言级虚拟化，下面将对这四种实现方式进行介绍。

2.3.1 指令级虚拟化

指令级虚拟化是指通过软件模拟出客户端操作系统运行所需要的物理硬件的方法，我们一般称采用这种方法的虚拟机为模拟器^[12]。从计算机的组成原理来说，模拟器需要模拟整个计算机系统，这主要包括处理器、内存、硬盘、总线、定时器、I/O 设备等一系列部件。模拟器的基本原理就是将客户机执行时的指令与当前物理级的指令做映射，即模拟器将客户机指令转换后在本地物理机上执行。

指令级虚拟化的优点是虚拟机层次结构简单，健壮性好，能够在不同平台上执行，即客户平台架构发生改变，它也可以很容易适应平台的变化。例如模拟器可以完成原来运行在 X86 平台的操作系统运行在 ARM 系列的处理器上，缺点是效率低，实现复杂。

2.3.2 硬件级虚拟化

硬件级虚拟化^[13]是基于在客户操作系统运行的虚拟的物理机硬件指令大部分和真实的硬件的指令非常相似这一前提下实现的。该虚拟化技术可以将虚拟资源与物理资源进行关联，所以客户虚拟机运行时会使用本地的硬件资源。目前很多的虚拟化解决方案都使用了这一技术，比如 intel 公司 X86 系列使用的 VT 技术^[14]就是为硬件虚拟化准备，这说明这种虚拟化技术的可行性和实用性。

硬件级虚拟化主要完成的工作是构造的虚拟机需要处理指令集中的某些特殊指令，这些指令一般会引起处理器状态的切换，切换状态在多用户多任务系统中经常发生的，我们也称这些指令为特权指令。上层客户操作系统执行实现的命令传递给下层的虚拟机监控器(VMM)，当特权指令执行时该指令发给 VMM，VMM 能够截获这些敏感指令，在实际的处理器上执行该命令，并将结果返回给虚拟机。这样就可以实现各虚拟机之间的隔离。硬件级虚拟化的优点^[15]是不但实现的结构简单而且运行也是高效率，缺点是所虚拟的硬件体系平台比较有限，而且需要底层具体硬件支持。相比于其它虚拟化技术，硬件级虚拟化研究较多，应用也广泛，具有较多的商业软件实现，其中业界最具影响力的 VMware 和 Xen 就是属于硬件级虚拟化的范畴，还有其它的系统如在 Linux 已经集成的 KVM，微

软的 VirtualPC 等。

2.3.3 操作系统级虚拟化

操作系统级虚拟化不同于指令级和硬件级虚拟化技术。操作系统级的虚拟化不用创建出可运行系统的虚拟机，而是在一个操作系统之上创建出给多个应用程序运行的独立的环境，即这些相互独立的运行环境共用同一个操作系统内核。操作系统级的虚拟化相当于内核的一种功能，而不是独立的一层软件抽象。它的关键思想在于，操作系统加入相应的虚拟化功能后可以给每个客户虚拟机提供一个独立的与当前运行操作系统一样的环境，即相当于当前操作系统的一份拷贝，这样就实现虚拟机之间的隔离。

因为不需要指令的翻译或者 vmm 层的硬件管理，所以操作系统级虚拟化可以降低虚拟化带来的性能开销，效率非常高，大多数实现都能达到接近原本的性能，共用同一个内核也减少了多个操作系统的安装次数，减少软件冗余。也正因为其优点，操作系统级虚拟化不支持多种类型的操作系统，不具备跨平台的移植性。最典型的使用操作系统级虚拟化的就是 OpenVZ 系统^[16]，该系统能够在一个物理机上创建多个虚拟专用服务器，即相当于虚拟机，实现隔离，最大限度的共享硬件资源。

2.3.4 编程语言级虚拟化

编程语言级虚拟化是进程级的虚拟化方案，使用这种技术的虚拟机并不对底层硬件或者操作系统进行抽象，而是将程序运行的环境进行抽象，因为与系统中硬件环境指令的执行类似，应用程序也是由一系列的指令组成，所以编程语言级虚拟化只是抽象的层次不一样，虚拟机作为一个进程运行在操作系统之上，在这样的虚拟机中用户只能运行相应的语言编写的应用程序。比如目前最流行的 JVM^[17]是 java 语言使用的虚拟机，java 程序运行时相当于在一台物理机上运行，其它的还有 CLR, Erlang VM 等。

2.4 Android 虚拟化现状

随着 Android 系统的不断发展普及，越来越多的研究机构和商业公司开始研究并实现 Android 系统的虚拟化，学术界如美国哥伦比亚大学实现了一个 Cells 系统，用于智能手机系统的虚拟化，工业界如 VMware 公司实现了 MVP 系统用于 Android 系统虚拟化，下面主要介绍几个典型的可用于 Android 系统虚拟化的技术。

- 哥伦比亚大学 Cells 系统^[18]

Cells 系统是由哥伦比亚大学设计并实现的用于 Android 虚拟化的系统。该系统采用一个虚拟前台系统和多个虚拟后台系统的模式实现多个操作系统共用一个物理手机。Cells 系统功能包括 3D 图形加速，电源管理功能，完整的电话功能和为每个虚拟 Android 单独分配电话会号码等。

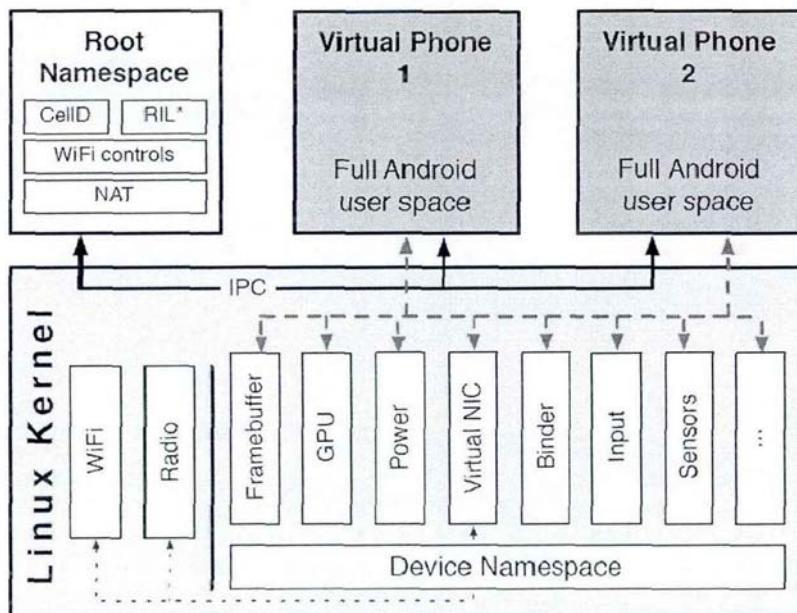


图 2-2 Cells 系统架构

图 2-2 是参考文献[18]所示的 Cells 系统的架构图，从该图中可以看到，每个 VP(virtual phone)都有自己的独立运行空间，多个 VP 能够同时运行并共享了一个 Linux 内核，为了实现虚拟化隔离功能，该系统将操作系统中的各类资源重新虚

拟化映射，这些资源包括文件系统，进程 PID，PIC 通信，网络接口，UID 等。对于设备的虚拟化，Cells 系统分别从内核态和用户态实现，内核态主要是关键性的硬件设备如输入，GPU，传感器，用户态实现了 wifi 控制，这样设计时为了更好的扩展性。该系统能够运行在 Google 的 Nexus 1 和 Nexus S 手机上，能够跨越多种硬件设备和不同 Android 系统版本，其测试结果也显示在 UI 显示和触摸屏操作方面，用户不会感觉到当前 Android 系统的延迟，说明该系统效率很高。目前该系统已经进行商业化，所以无法拿到相关的源代码进行分析。

- VMware 公司的移动虚拟化平台 MVP^[19]

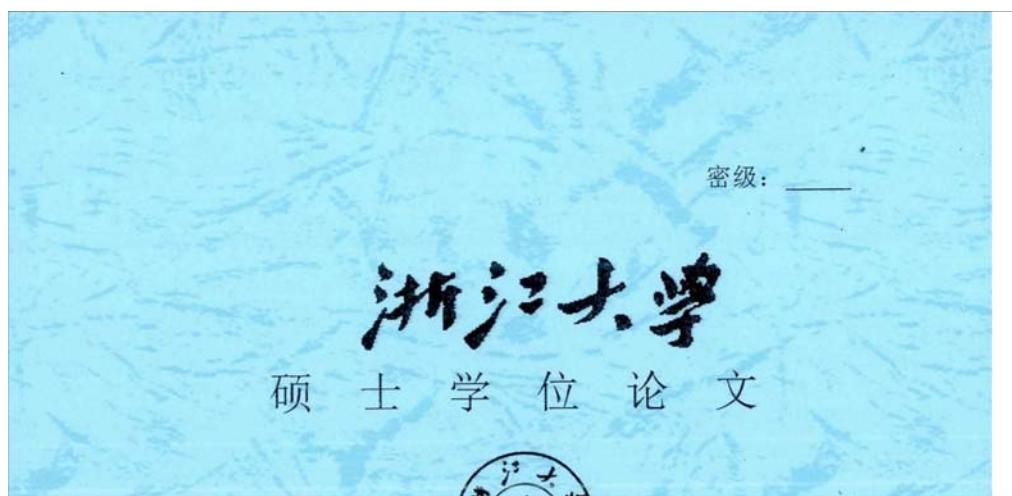
MVP (Mobile Virtualization Platform) 是 VMware 公司推出的移动虚拟化平台，该系统用户目标就是移动设备，即手机和平板。该系统实现了一个对硬件的抽象层 Hypervisor，在 MVP 上可以运行多个类型的操作系统，主机系统标准支持



系统启动时, mvpd 加载 gust 客户机的镜像到内存并启动 VMM, 而 VMM 与 host 主机之间的切换就是通过 MVPkm 模块实现, 而其它的模块为客户机提供了一套虚拟的设备, 在客户机系统中可通过内核 mksck 进行通信获取相应的功能。对于使用了 MVP 的手机厂商不需要担心硬件之间的差异, 只需要安装了该系统能够将自己相同的软件安装到不同的手机上, 不需要针对不同手机开发不同的驱动程序, 进而减少了手机移植的成本。作为商业软件, MVP 系统的兼容性, 安全性以及性能都很出色。

- 开源平台 Xen-ARM^[20]

Xen-ARM 是基于 Xen 而开发产生的, Xen 是由 XenSource 公司所主导, 而 Xen-ARM 的开发者是以三星为主体, 而近几年来有关 Xen-ARM 的论文多由三星公司所发表。Xen-ARM 架构与 Xen 相仿, 在物理机硬件上架构了一层 Hypervisor



第3章 Android 系统虚拟化关键技术研究

本章首先将对 Android 系统虚拟化需要解决的问题进行分析，说明 Android 虚拟化的特殊性，然后比较了第 2 章介绍的虚拟化技术并选择了操作系统级虚拟化作为 Android 系统虚拟化的关键技术，最后介绍了实现 Android 操作系统级虚拟化使用的工具 LXC，分析了 LXC 实现的原理。

3.1 Android 系统虚拟化设计思路

3.1.1 Android 虚拟化面临的问题

Android 系统一般都运行在智能手机平台上，有时可能会运行在更小的嵌入式平台上，与通用的 PC 虚拟化或者服务器虚拟化相比，面临着许多问题，主要体现在下面几个方面：

- 内存资源限制

Android 系统运行的硬件平台资源有限^[21]，比如智能手机的内存为 512M，如果虚拟化的两个操作系统，平均每个操作系统只能得到 256M 内存，还要保障通话和短信这些实时性较高的应用的正常运行，如何高效使用内存是 Android 系统虚拟化必须要考虑的问题。

- 计算处理能力有限

与个人电脑和服务器通常采用的 intel 处理器不同，intel X86 设计的目标是性能优先，而 Android 运行的硬件一般为 ARM 架构处理器，为了降低功耗，ARM 架构的计算处理能力比 X86 弱。虚拟化需要保证多个 Android 系统中的应用程序能够流畅运行。

- 设备功耗限制^[22]

智能手机都用锂电池供电，需要采用有效的手段进行电源管理。Android 系统本身具有电源管理模块，比如待机的时候关闭不再使用的设备，无触屏按键操

作超时后关闭背光等。如果采用的虚拟化技术加入 VMM 层，则 VMM 需要管理客户操作系统 Android 系统，原有的电源管理不再适用，VMM 即使有电源管理，需要针对 Android 内部机制进行特别设计，会比较复杂。

- Android 设备的多样性

Android 设备种类繁多，主要包括智能手机和平板电脑，但每个硬件厂商都会针对自己产品的特点做特殊的硬件配置，所以每个设备都有自己定制修改的 Android 系统，Android 系统虚拟化需要整合各个系统的差异性。

3.1.2 Android 虚拟化技术的选择

为了解决 3.1.1 小节中提出的 Android 虚拟化所面临的四个问题，我们需要选择最合适的虚拟化方案。

指令级虚拟化需要模拟每一条指令的执行，指令翻译非常占用 CPU 计算资源，会导致系统运行不流畅。比如 Bochs 系统^[23]，能够模拟大多数版本的 X86 系统，从加电开机到定制的外围设备启动都能够模拟，但是效率非常低。

硬件级虚拟化需要硬件 CPU 的支持，目前 Android 系统主要运行在 ARM 平台上，在 ARM 系列 CPU 中，目前支持硬件辅助虚拟化的是 Cortex-A15 架构，该系列 CPU 在 Android 移动设备还不够广泛。硬件级虚拟化一般需要加入 VMM 管理层，比如 XEN 虚拟化中的 Hypervisor 层，对于虚拟化 Android 的设备来说多一个管理层需要更多的内存和 CPU，所以硬件级虚拟化也不是最佳的 Android 虚拟化方案。

编程语言级虚拟化已经在 Android 系统中使用，Android 运行的 Apk 包一般就是用 Java 编写运行的，虽然针对每个应用程序使用了 Dalvik 虚拟机，但是还是无法解决恶意应用程序直接读取通讯录或通话记录的问题。

本课题使用了操作系统级虚拟化方案，操作系统级虚拟化不需要加入 VMM 层进行管理，而且几个并行运行的 Android 系统能够共用一套内核代码，可以减少 CPU 和内存的消耗，操作系统虚拟化还保留了原来 Android 系统的许多特性，可以继续使用原来的电源管理模块减少 Android 设备的功耗。总之，操作系统级

虚拟化适合于 Android 设备的虚拟化,充分使用 Android 设备的计算和内存资源。

本课题中,我们使用了 Linux 自带的工具 LXC,该工具提供了轻量级的虚拟化即操作系统级虚拟化,没有其他虚拟化方法的复杂性。下面 3.2 节中将对该工具进行研究介绍。

3.2 LXC 工具研究

LXC(Linux Container)^[24]来自于 Sourceforge 网站上的开源项目,LXC 给 Linux 用户提供了用户空间的工具集,用户可以通过 LXC 创建和管理容器,在容器中创建运行操作系统就可以有效的隔离多个操作系统,实现操作系统级的虚拟化。

3.2.1 LXC 命令介绍

目前从 Linux 内核 2.6.27 版本开始已经支持 LXC,只需要在 Linux 用户态安装相应的用户态工具 liblxc 即可。下表 3-1 是 LXC 的常用命令接口:

表 3-1 LXC 常用命令

命令	使用方法	参数含义
lxc-create 创建容器	lxc-create -n name [-f config_file]	-n 创建的容器名字 -f 容器配置文件的路径
lxc-start 在容器中执行命令	lxc-start -n name [-f config_file] [-c console_file] [-d] [-s KEY=VAL] [command]	-d 将容器守护进程执行 -f 后面跟配置文件 -c 指定一个文件作为容器 console 的输出,如果不指定,将输出到终端 -s 指定配置
lxc-execute 在容器中执行程序	lxc-execute -n name [-f config_file] [-s KEY=VAL] command	-n 容器名字 -f 容器配置文件的路径,如果没有使用默认 -s 后面跟配置键值对 command 为要执行的命令 例如:

		/bin/bash
lxc-kill 发送信号给容器中的第一个用户进程	lxc-kill -n name SIGNUM	-n 容器名字 SIGNUM 信号（此参数可选， 默认 SIGKILL）
lxc-stop 停止容器中所有的进程	lxc-stop -n name	-n 停止的容器名字
lxc-destroy 销毁容器	lxc-destroy -n name	-n 销毁的容器名字
lxc-monitor 监控一个容器状态的变换	lxc-monitor -n name	-n 监控的容器名字
lxc-info 用户获取一个容器的状态	lxc-info -n name	-n 获取的容器名字
lxc-cgroup 用于获取或调整与 cgroup 相关的参数	lxc-cgroup -n name subsystem value	-n 调整的容器名字 subsystem cgroup 子系统类型 value 子系统类型值
lxc-ls	lxc-ls	列出当前系统所有的容器
lxc-ps	lxc-ps	列出特定容器中运行的进程
lxc-checkconfig	lxc-checkconfig	判断 Linux 内核是否支持 LXC

3.2.2 LXC 容器隔离实现机制研究

从前面的介绍中我们可以了解到，LXC 能够创建容器用于 Android 系统的虚拟化，而 LXC 作为用户层管理工具主要提供了管理容器的接口，对实现容器的机制进行了封装隐藏，本小节将对 LXC 容器的实现机制进行分析。

LXC 内部采用了 Linux 内核 Namespace 和 Cgroup 两个特性，下面我们将对这两种机制进行介绍。

3.2.2.1 Namespace 命名空间实现机制研究

Namespace 命名空间机制^[25]给虚拟化提供了轻量级形式，即操作系统级虚拟化。该机制和 FreeBSD 的 jail 机制和 OpenVZ 类似。传统上，Linux 系统中所有进程通过 PID 进行标识，内核只需要管理一个 PID 列表，而且用户通过 uname 系统调用获取的系统相关信息也全部相同。在 Linux 系统中，用户的管理方式通过 UID 编号，即通过全局唯一的 UID 列表进行标识。全局 ID 能够使内核很好的管理系统，选择允许或拒绝某些特权。如 UID 为 0 的 root 用户能够允许做任何事情，但是其他 UID 的用户就会受到限制；对于用户 x 无法杀死另一个用户 y 的进程，但是用户 x 可以看到用户 y 的活动状态，而这种状态并不适用于一些场景，比如对于隐私性要求较高的服务。

为了解决上述类似的问题，Namespace 机制为进程 ID，用户 ID 等系列资源的隔离提供了一种占用资源极少的解决方案，而其它虚拟化方案一般需要一台物理机运行多个内核进行上述资源的隔离，命名空间可以在一台物理机上运行一个内核，通过命名空间机制将上述全局资源进行抽象。它可以使一组进程放置于容器中，而各个容器间彼此隔离，也可以设置容器之间进行共享。从用户的角度看，命名空间将全局资源控制进行了分割放置到相应的容器中，在容器中进程只能看到本容器中的成员而无法看到其他容器的成员。

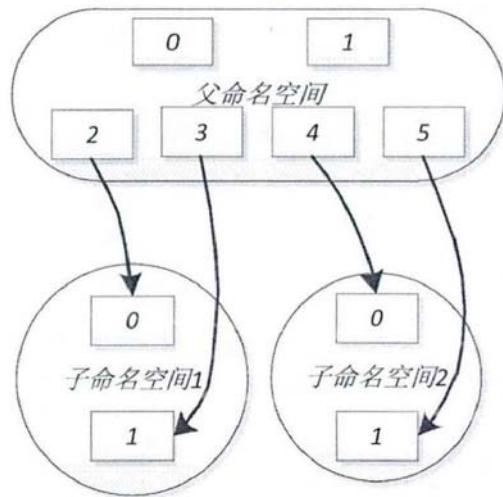


图 3-1 命名空间层次关系

在上图 3-1 中，我们可以看到三个命名空间，父命名空间记录管理了 6 个 PID 值，而两个子命名空间父命名空间知道子命名空间的存在，而两个子命名空间不知道对方的存在，对于进程各个子命名空间中属性的改变无法将影响传播到其它命名空间(包括父命名空间)，这样在两个子命名空间中运行 Linux 内核就可以很好的隔离两个系统。

- 命名空间在 Linux 内核中的表示^[26]

命名空间分别对 UTS，进程间通信(IPC)，文件系统视图，进程 PID，UID 和网络六个属性进行封装，内核中的数据结构如下：

```
struct nsproxy {
    atomit_t count; //指向同一个 nsproxy 的进程个数
    struct uts_namespace *uts_ns; //运行的内核
    struct ipc_namespace *ipc_ns; //进程通信 ipc
    struct pid_namespace *pid_ns; //进程 pid
    struct mnt_namespace *mnt_ns; //文件系统
    struct user_namespace *user_ns; //用户的资源限制信息
    struct net *net_ns; //网络
}
```

在每个 Task 任务结构中包含了该 nsproxy 结构，这样对于每个创建的进程都有对应的命名空间，通过该结构可以建立进程与命名空间之间的关系，代码如下。

```
struct task_struct{  
    ...  
    struct nsproxy *nsproxy;//命名空间指针，指向具体的对象  
    ...  
}
```

对于命名空间的支持，需要在 Linux 内核编译的时候进行启动设置。如果没有设置，会使用系统默认的命名空间即整个内核就一个命名空间，全局可见。

对于每个具体的子命名空间实现形式，在此不再具体展开。

- 命名空间在用户空间的表示

命名空间机制在用户层通过调用系统调用 clone^[27]实现，clone 系统调用与 fork 系统调用的最大区别就是通过传入众多的参数选择性的继承父进程的资源，而 fork 系统调用就会复制父进程的资源。clone 系统调用接口如下：

```
int clone(int (*fn)(void *), void *child_stack, int flags, void *arg);
```

参数：

fn：指向程序代码的指针

child_stack：子进程分配的堆栈空间

flags：标识子进程需要从父进程继承的资源

arg：传给子进程的参数

通过设置 flags 参数就可以创建新的命名空间，选择性的继承父进程的资源。

flags 设定为 CLONE_NEWPID 时会创建一个新的 PID 命名空间，即该新的命名空间为进程提供了一个新的独立的 PID 环境，调用 clone 系统的进程的 ID 号为 1，就成为该命名空间的第一个进程，相当于 Linux 系统的 Init 进程，由于是起始 PID 命名空间中的祖先进程，如果该进程结束在这个命名空间中的所有进程都会被结束。PID 的命名空间具有层次性，在父命名空间中的进程可以创建出子命名空间，而子命名空间对父命名空间可见。

flags 设定为 CLONE_NEWIPC 时会创建一个新的 IPC 命名空间，如果该标记没有设定，进程会根据父进程的参数进程设置。针对 system V 对象和 POSIX 信号队列的进程间通信机制，IPC 命名空间可以进行隔离，即当前进程的通信对象或队列在本命名空间中对其他进程可见，而在不同命名空间中的进程进行通信类似于不同 Linux 系统间的通信。

flags 设定为 CLONE_NEWUTS 时会创建一个新的 UTS 命名空间，初始结构会以调用进程的 UTS 进行初始化，通过调用 setdomainname 函数和 sethostname 函数可以分别设置域名和主机名，调用 uname 函数可以获得 UTS 信息。

flags 设定为 CLONE_NEWNS 时会创建新的 mount 挂载点命名空间，建立的挂载点命名空间就是进程可见的文件结构视图，所以通过挂载命名空间可以很好的进行文件系统的隔离。

flags 设定为 CLONE_NEWWNET 时会创建一个新的网络命名空间，网络命名空间对网络栈进行了视图上的隔离，包括 IPV4, IPV6, 栈协议, ip 路由表, 防火墙规则等。一个物理网络设备只能一次对应一个网络命名空间，但通过创建虚拟网络设备创建的隧道可以与实际的物理网络设备进行通信，这样就可以实现多个网络设备的通信。

以上所述的 flags 标识可以组合使用，LXC 调用该接口可以很方便的创建独立的运行环境，设定相应的参数，完成操作系统级的虚拟化隔离。

3.2.2.2 Cgroup 实现机制研究

LXC 通过命名空间机制实现了资源的隔离，而对于物理资源的限制则通过 cgroup(control group)机制实现。

cgroup 系统定义了以下的概念：

1)控制群(control group):控制群就是一组进程组，是 cgroup 系统中的控制的基本单位。

2)子系统：子系统是一类资源控制系统，比如 CPU，是对 cgroup 中进程的控制具体方法，通过子系统可以针对每个 cgroup 进行限制。目前 cgroup 机制支持的子系统如下表 3-2 所示：

表 3-2 Cgroup 各子系统作用

子系统类型	作用
cpu 子系统	用于管理 cgroup 中进程对 cpu 使用，可以分配 cpu 权重值给进程。
cpuset 子系统	在多核系统中用户 cpu 的分配和内存节点的分配。
cpuacct 子系统	用于自动生成 cgroup 中进程使用 cpu 的报告
memory 子系统	管理 cgroup 中进程对内存的使用，可以设定参数值限定进程使用的内存量
devices 子系统	管理进程对设备的访问，如设定访问权限
freezer 子系统	挂起或恢复 cgroup 中的进程组
net_cls 子系统	用于 cgroup 中进程对网络带宽的使用

3)层级：层级是各个控制群以树形的方式进行排列，子节点的控制权继承父节点的树形。linux 系统中一个进程在其中的一个控制群中，同时也在其中的一个层级下。一个层级通过 cgroup 系统的虚拟文件系统相对应。

它们之间的层级关系如下图 3-2 所示：

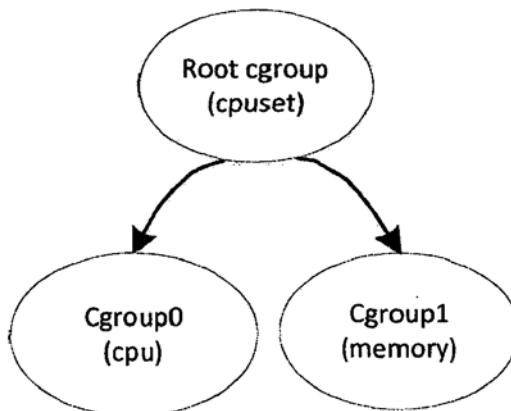


图 3-2 cgroup 层级关系

在 Linux 系统中第一个被用户建立的 cgroup 是根 cgroup，包含了系统中所有的进程，位于第一层级。在上图 3-2 中根 cgroup 又被分为两个子的 cgroup 系统，即 cgroup0 和 cgroup1，位于 cgroup 系统的第二层。一个子系统只能在一个层级

上，即在第一层级的 cpuset 子系统不同加入第二层级中的 cgroup0 和 cgroup1，因为前面已经说过层级中子节点需要继承父节点的属性，如果两个层级中都有则会出现重复。每个层级可以有多个子系统，如第二层级有 cpu 和 memory 子系统。对于进程来说，可以位于不同层级的 cgroup 中。

● cgroup 在 linux 内核层的表示

cgroup 相关的数据结构和命名空间类似，也是从进程管理结构开始，如下

```
struct task_struct
{
    ...
    #ifdef CONFIG_CGROUPS
        /* Control Group info protected by css_set_lock */
        struct css_set __rcu *cgroups;
        /* cg_list protected by css_set_lock and tsk->alloc_lock */
        struct list_head cg_list;
    #endif
    ...
}
```

从代码上可见，如果需要使用 cgroup 机制需要在 Linux 内核编译的时候开启 CONFIG_CGROUPS 宏。在 css_set 结构中存储了与进程相关 cgroups 信息，在 cg_list 将在同一个 css_set 的进程组链接起来。css_set 结构如下

```
struct css_set {
    atomic_t refcount; // 引用计数
    struct hlist_node hlist; // css_set 组成的 hash 表，可用于内核快速查找
    struct list_head tasks; // css_set 中进程组链表
    struct list_head cg_links; //
    struct cgroup_subsys_state *subsys[CGROUP_SUBSYS_COUNT];
    // 指针数组指向具体的子系统，进程可以获得控制信息
    struct rcu_head rcu_head; //
```

```
};
```

`css_set` 结构中最重要成员就是 `cgroup_subsys_state` 指针数组，在进程和 `cgroup` 之间没有直接的联系，需要通过 `cgroup_subsys_state` 间接指向 `cgroup` 结构，这样做主要是因为有的时候读取子系统状态会比较频繁而对 `cgroup` 赋值等操作较少，`struct cgroup` 中有一个成员 `cgroupfs_root` 结构，就是用于我们前面所说的层级关系的具体实现，这几个数据结构之间的关系如下图 3-3 所示。

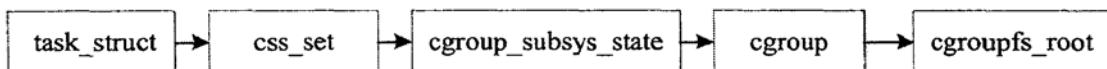


图 3-3 Cgroup 实现代码结构关系

对于子系统的管理，则通过 `cgroup_subsys` 数据结构进行管理。在该结构中定义了一组操作的接口函数指针，相当于 c++ 的基类，给出了接口函数具体的行为需要针对每个子系统的特点进行自行定义，比如 `cgroup_subsys_state` 接口每个子系统返回的信息是不同的，需要不同的实现，这样的设计就可以很好的完成多种类型的兼容。

● cgroup 在 Linux 用户层的表示

`cgroup` 在 linux 用户层通过 `cgroup` 文件系统方式表示出来。比如在用户空间可以通过执行如下命令 `:mount -t cgroup -o cpu, memory /cpu_mem /mycgroup/cpu_mem` 就可以创建名字为 `cpu_mem` 的层级，这个层级中有 `cpu` 和 `memory` 两个子系统，这两个系统可以是挂载到 `/mycgroup/cpu_mem` 文件下。这一过程就如同创建新的文件夹的过程，只是创建的是 `cgroup` 特殊文件系统。

如果在 `cpu_mem` 系统中想要创建一个新的 `cgroup`，只要进入 `cpu_mem` 文件目录下，执行命令 `mkdir newcgroup` 就可以创建一个叫 `newcgroup` 的控制群，这样也表示新创建的 `cgroup` 是属于 `cpu_mem` 这一个层级。进入 `newcgroup` 文件目录后，会有相关子系统的控制文件，可以读取或者修改这些控制文件，比如向的 `tasks` 文件中写入当前系统中某个进程的 `pid` 号就相当于将进程加入了新创建的

cgroup 控制群中。整个过程如下图 3-4 所示：



图 3-4 Cgroup 文件创建流程

3.3 本章小结

本章首先介绍了进行 Android 虚拟化需要面临的问题，然后分析了从不同层次进行虚拟化的优缺点，说明采用操作系统级虚拟化优势，最后介绍了用于轻量级虚拟化的 LXC 工具，详细介绍了用于实现 LXC 的背后技术-Namespace 机制和 Cgroup 机制。

第4章 Android 系统虚拟化设计与实现

在本章，我们将详细介绍 Android 系统虚拟化的设计方案，首先介绍了总体设计方案，给出了设计目标和总体框架设计，然后详细介绍了 lxc 工具的移植工作，最后是介绍输入与显示输出设备的复用和 binder 驱动实现方式。

4.1 总体设计

4.1.1 设计目标

通过前面几个章节的介绍，Android 系统虚拟化主要实现以下几个设计目标：

(1) 实现操作系统级的资源隔离。通过虚拟化技术，在一个物理设备上实现多个 Android 系统之间的资源隔离，保证每个 Android 系统中的应用程序能够安全的运行。

(2) 降低虚拟化带来的性能开销。实现操作系统级的虚拟化，减少虚拟化后系统性能的损失。针对 Android 系统的特点，优化虚拟化方案。

(3) 支持多个 Android 系统能够同时运行。Android 系统虚拟化需要支持多个 Android 系统，选择合适的虚拟化技术能够使多个 Android 系统同时运行并且流畅运行。

(4) Android 虚拟化能够正常输入输出。由于时间和能力有限，本原型系统在使用 Android_x86 开源代码实现，运行于 virtualbox 虚拟机下，主要实现 Android 虚拟化正常的输入与输出功能，对于其它功能如打电话等还需要进一步的工作。

4.1.2 总体设计方案

通过上一章 3.1 节我们可以已经知道 Android 系统虚拟化所要面临的问题，同时在 3.2 节我们也介绍了 LXC 工具，了解了 LXC 使用方法和 LXC 的实现原理。LXC 工具内核代码支持的资源隔离的功能，即通过 namespace 机制进行资源的隔离和 cgroup 机制进行资源的限制，使用这些功能可以实现 Android 系统之间资源

隔离，并且 LXC 对外接口简单，能够方便的使用 LXC 工具进行管理虚拟化后的管理，比如启动虚拟的 Android 系统，只需要调用一条命令即可，因此将 LXC 作为 Android 虚拟化的工具完全满足需求。

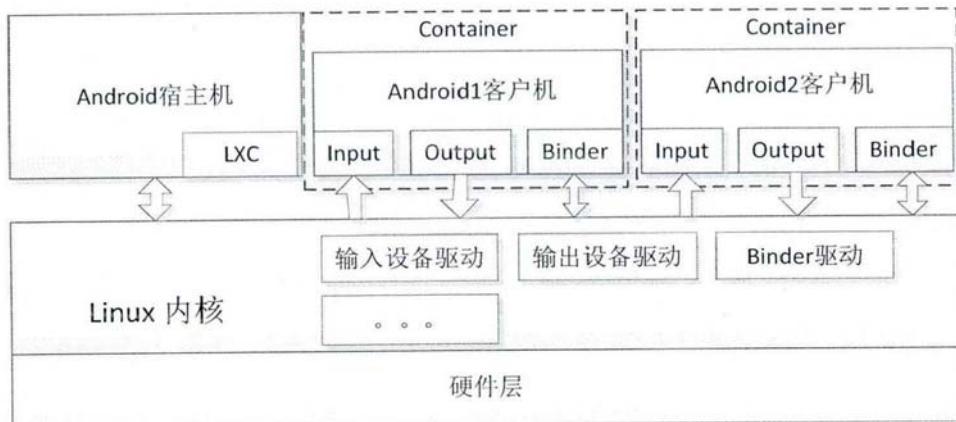


图 4-1 Android 系统虚拟化总体架构

上图 4-1 是本课题提出的总体设计方案，在该方案中，主要分为三层，最上层是我们运行的 Android 系统的用户态层序，包括一个 Android 宿主机和多个 Android 客户机，及用户操作系统级虚拟化的 LXC 工具，其中每个 Android 客户机通过 LXC 工具启动运行在各自的 Container 中，并且我们针对 Android 客户机特点对输入设备，显示输出设备两个模块进行优化，实现多个 Android 系统复用 Linux 内核输入设备和显示输出设备。中间层就是我们多个 Android 系统共享的 Linux 内核，在该层中我们首先需要开启内核支持 LXC 的 namespace 命名空间机制和 cgroup 机制，然后修改了 Binder 驱动，该驱动可以被多个 Android 系统复用，保证 Android 系统内的通信。最下面一层就是硬件层，该层就是 Linux 内核所驱动的硬件设备，该层没有任何的修改，也不需要硬件对虚拟化特性的支持。

从上述的总体设计方案中，我们实现基于 lxc 的 Android 虚拟化主要完成三方面的工作，第一，移植 LXC 工具到 Android 系统中，这是最基础的工作；第二，完成系统中输入设备和显示输出设备的复用，这是多个 Android 系统能够交互运行的基础；第三，修改 Linux 内核中 Binder 驱动代码，这是 Android 系统中各个应用程序能够正常通信的基础。本章后面三小节将分别给出这三方面实现的具体

方案。

4.2 Android 下 LXC 工具移植

Android 系统并不支持 lxc 工具，所以进行基于 lxc 的 Android 系统虚拟化之前需要在宿主 Android 系统中加入 lxc 工具，第一，需要底层的 Linux 系统支持 namespace 和 cgroup 机制，这是安装 lxc 工具的前提；第二，移植 lxc 工具，使其在 Android 系统环境下能够正常运行；第三，构建 lxc 工具运行 Android 客户机时能够使用的文件系统。

- Linux 内核编译开启支持 lxc 特性

一般 Android 系统中的 Linux 内核并没有开启 lxc 需要使用的 namespace 和 cgroup 机制，所以需要对 Linux 内核重新进行编译，编译前需要开启如下表 4-1 所示的编译选项：

表 4-1 支持 LXC 的 Linux 编译配置选项

Namespace 相关	
CONFIG_NAMESPACES	开启命名空间功能
CONFIG_UTS_NS	开启 UTS 命名空间功能
CONFIG_IPC_NS	开启 IPC 通信命名空间功能
CONFIG_PID_NS	开启 PID 命名空间功能
CONFIG_NET_NS	开启网络命名空间功能
Cgroup 相关	
CONFIG_CGROUPS	开启 cgroup 系统功能
CONFIG_CGROUP_NS	开启 ns 子系统功能
CONFIG_CGROUP_DEVICE	开启 device 设备子系统功能
CONFIG_CGROUP_SCHED	开启 cpu 调度子系统功能
CONFIG_CGROUP_CPUACCT	开启 cpu 报告子系统功能
CONFIG_CGROUP_CPUSETS	开启 cpusets 子系统功能

其它

CONFIG_POSIX_MQUEUE	开启 posix 消息传递功能
---------------------	-----------------

- lxc 工具的修改与编译

libc 库是 Linux 等类 Unix 系统的基础函数库, Android 系统自带的 libc 库是 bionic, 而不是 lxc 工具使用的 Glibc, 因为 bionic 库的代码量和运行速度上都比 glibc 小很多, 大约只有 200kb, 而且 bionic 库给 Android 系统提供了一些特定的函数, 如 getprop 函数。与 bionic 库相关的函数如下:

- a) 修改 bionic 库中的 setenv 函数。在 lxc 中的 lxc_init 函数初始化 lxc 的时候调用了 setenv 函数进行环境变量的设置。函数原型如下所示:

```
int setenv(const char *name, const char *value, int overwrite);
```

其中输入参数 value 值可能为 NULL, 对于这种情况, 在 glibc 中执行是没有问题的但调用 Android 的 bionic 库会发生段错误, 所以需要修改 bionic 库的 setenv 函数, 添加判断 NULL 值代码逻辑。

- b) 修改 bionic 库中的 tmpfile 函数。在 lxc 启动的时候会调用 tmpfile 函数创建一些临时的文件, 执行 bionic 库中的 tmpfile 函数时会发生错误, 无法创建文件, 所以我们重写了 tmpfile 函数, 使其能够正常创建文件。

除了 Android 本身 c 库对 lxc 工具的支持问题, lxc 本身代码在 Android 系统下运行也存在问题, 当 lxc 构建的 container 运行 Android 客户机时, 需要将根文件系统重新挂载, 调用 pivot_root 系统调用时, Android 文件系统的挂载结构不能通过 pivot_root 系统调用的检查, 在这里, 我们将该系统调用改为 chroot 解决该问题。

修改完成 lxc 源码后, 需要进行编译链接。由于 lxc 是运行在 Android 系统下, 需要使用 Android 系统相关的编译工具进行编译, 通过如下步骤就可以完成 lxc 工具的编译, 生成可在 Android 系统运行的代码:

- 1) 提取 Android NDK 工具中的 toolchain 编译工具;
- 2) 设置 tool chain 环境变量;
- 3) 设置 lxc 中 makefile 文件的编译工具为 toolchain;

4) 执行 make, 和 make install ;

编译好 lxc 之后可以放入/system/local 下, 使用 lxc 工具中的 lxc-checkconfig 检查运行环境, 全部显示为可行时说明 lxc 工具移植成功。

● Android 客户机及文件系统的设置

Android 系统运行于 Linux 系统之上, 当启动 Linux 内核后, 运行的第一个进程就是 init, 是后续启动所有进程的祖先进程。在我们 Android 虚拟化系统中, 首先启动的是 Linux 内核, 然后是 Android 宿主机, 最后通过 lxc 工具启动 Android 客户机。lxc 工具启动客户机的第一个进程也是 init, 在启动参数时我们需要修改 Android 客户机的挂载目录, 因为客户机和宿主机不能够使用同一套文件系统目录, 通过以下两个方面我们完成了 Android 客户机的顺利启动。

1) 多个 Android 文件系统的隔离。由于宿主机和客户机的 Android 目录相同, 我们将整个宿主机的目录结构拷贝到 Linux 系统根目录下 client 子目录下, 如图 4-2 所示, root1 目录和 root2 目录下的文件拷贝自宿主机中的目录, root1 和 root2 是客户机系统运行的文件系统环境, 文件目录/是宿主机运行的文件系统环境, 运行的客户机系统运行时相互之间无法感知到对方文件系统的存在。

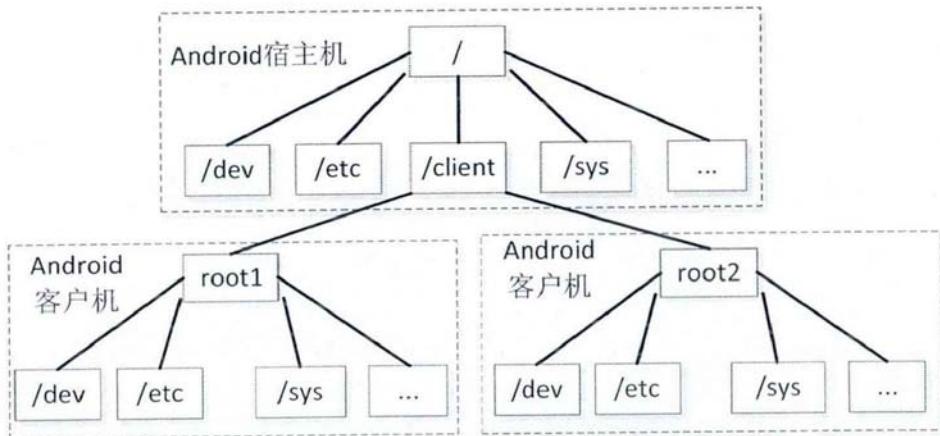


图 4-2 Android 文件系统视图

2) 修改 Android 客户机 init 启动进程和 init.rc 启动配置。修改 init.c 中的 main 函数, 修改挂载文件系统相关代码, 将我们自己创建的客户机文件系统挂载

使用。修改 init 启动时使用的 init.rc 文件，关闭服务 ServiceManager 的启动，原因在 4.3 节 Binder 驱动虚拟化中给出。

通过上述步骤，使用 lxc 工具的 lxc-start 命令就可以启动 Android 客户机系统，lxc 工具在 Android 系统上正常运行，而客户机系统的正常运行还需要修改相关 Linux 内核驱动和客户机代码。

4.3 输入设备与显示输出设备复用

4.3.1 Linux 控制终端 tty 简介

在 Linux 下，终端是一种字符型设备，一般都会用 tty 简称各种类型的终端，在 Linux 下终端主要包括三类，串口终端，伪终端 pty 和控制终端 tty。控制终端在文件系统的 /dev/tty 下，是进程控制终端的特殊文件。在 Linux 系统中，我们一般称显示设备为控制台终端(console)，通过特殊的设备文件与之关联，即 /dev 目录下的 tty0, tty1 等。Linux 启动后控制台登陆时使用的是 tty1，在 Linux 使用 Alt+(F1~F6) 可以分别切换到 tty1~tty6 下，我们称 tty1~tty6 为虚拟字符终端，对于图形交互的终端，在一般的 Linux 系统中对应的是 tty7 设备，所以与图形相关的交互都是通过该设备进行。而 tty0 是当前进程所使用的虚拟终端的别名，即在当前前台的终端，比如当前进程使用 tty2 作为控制终端，则 tty0 对应的就是虚拟终端 tty2，所以 Linux 系统产生的信息都会发送到 tty0 终端上。tty 设备有对应的 tty 驱动，在一个典型的 Linux 控制终端中，输入和显示输出设备的工作流程如下图 4-3 所示

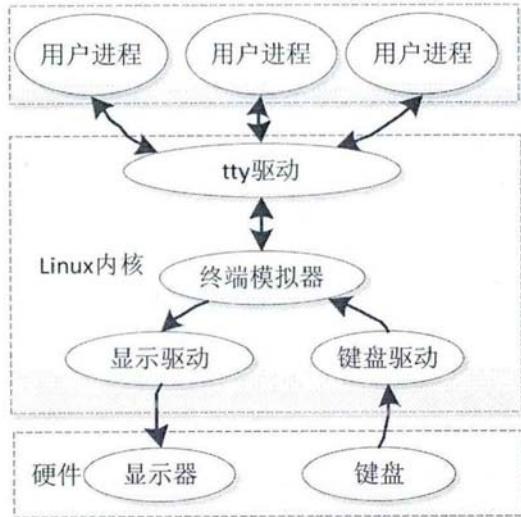


图 4-3 输入与显示输出设备工作流程图

从上图 4-3 我们可以看出，tty 驱动在 Linux 内核层，对上层应用层来说，tty 设备接收进程处理后的信息，对于下层来说，tty 驱动调用终端模拟器模块，所以 tty 驱动能够屏蔽具体硬件的细节，按照所需要的功能进行消息的处理传递，可以很好的将多样的输入设备和显示输出设备整合，完成终端任务的控制。

4.3.2 显示输出设备复用

4.3.2.1 显示输出设备复用设计

从 4.3.1 小节中的介绍，Android 系统虚拟化后，宿主机和客户机都会共同访问显示输出设备，而对于硬件设备只有唯一的一块显示屏，多个 Android 的访问必然带来冲突。该问题可以有两种方法解决，第一，从 Linux 的输出设备的 tty 驱动和显示驱动层修改，使 linux 支持多个 Android 系统的显示输出；第二，从运行在 Linux 内核层之上的 Android 系统修改，即修改 Android 宿主机和客户机中显示输入相关代码，保持原有驱动不变。Linux 内核中显示相关驱动复杂，需要实现功能多，与硬件相关性大，而从 Android 系统的显示输出相关框架清晰，其中与 tty 相关代码较少，可作为显示输出设备修改点完成设备的复用，实现方法简单有效，所以我们选择第二种方法解决。

Android 系统的会打开 tty 设备作为控制终端，而 tty 设备分为图像模式和文本

模式，所以通过不同 Android 系统打开不同的虚拟 tty 设备并设为图像模式就可以将当前需要显示的 Android 系统输出到屏幕上，我们设计的基本框架原理如下图 4-4 所示：

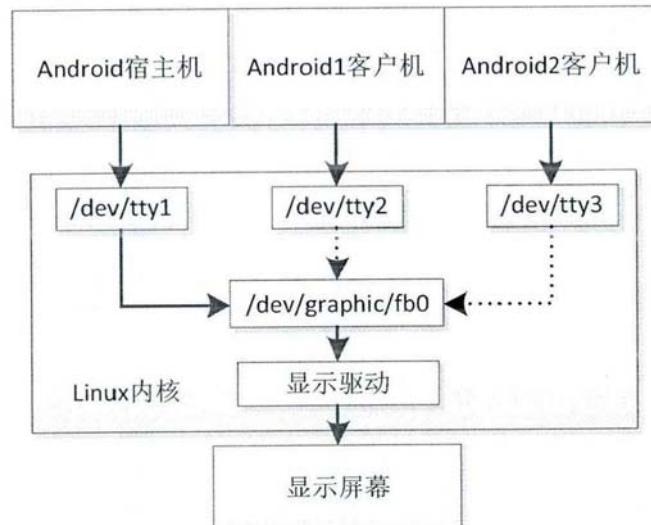


图 4-4 显示输出设备复用设计原理图

从上图 4-4 可见，每个 Android 系统都会选择打开一个空闲的 tty 设备，并让自己的 tty 设备独占显示帧缓存设备 fb0，即 framebuffer，图中当前占用 fb0 的是 Android 宿主机，则宿主机的图像输出会写入到 framebuffer 中，最终通过显示驱动显示到屏幕上，而其它的不占用 fb0 的 tty 对应的 Android 系统就不会显示到屏幕上。

4.3.2.2 显示输出设备复用实现

从上一节中的分析，我们实现显示输出设备的复用需要修改从 Android 系统启动到最终显示到屏幕的某些模块，通过对 Android 系统的代码分析，其中显示输出相关的代码在 surfaceFlinger 服务中，其启动后会通过 tty 打开 framebuffer 设备进行显示操作。我们需关注的基本步骤是从 Android 启动到 tty 设备的设置如下图 4-5 所示



图 4-5 Android 启动部分流程

从图 4-5 流程可知，我们只需要修改显示设备使用的 tty 对应的 framebuffer 就可以使多个系统复用一个显示输出设备即可，所以在该模块中我们加入自己的功能代码，具体实现如下：

- Android 系统 tty 设备的打开。首先需要建立起 tty 与对应 Android 系统之间的关联，我们修改 surfaceflinger 目录下的 DisplayDevice.cpp 文件，将初始化 tty 设备过程替换为我们实现的 get_new_tty() 函数，该函数主要完成空闲 tty 的查找，并设置为图形模式，设置 tty 的切换操作方式为“进程管理”，最后通过该 tty 打开 framebuffer 终端。
- Android 系统屏幕的切换。加入两个信号响应函数，open_screen 函数和 close_screen 函数分别针对获取屏幕和释放屏幕的操作，这两个函数实际调用的是 surfaceFlinger 服务中 screenAcquired 和 screenReleased 函数。对于不同 Android 系统之间的切换操作，可以通过系统自带的 tty 切换快捷键，即通过 Alt+Fn 的形式，当按下对应系统的快捷键时前台系统就会调用 close_screen 函数释放显示输出设备，而被切换的系统则调用 open_screen 函数获取显示输出设备。

4.3.3 输入设备复用

4.3.3.1 输入设备复用设计

由于多个 Android 系统使用同一套 Linux 内核，多个输入设备，如触摸屏，按键等，会被多个 Android 系统所共用，每个输入设备的事件都会发送到所有的 Android 系统中，但是对于虚拟化后的多个 Android，在同一个时刻运行在前台的系统只有一个，输入设备的事件消息只需要运行在前台的 Android 系统处理就可以。Android 系统是通过读取 Linux 输入设备文件来获取输入的信息事件，这些设备文件位于/dev/input 目录下，比如键盘设备对应的是 event2。用户输入系统的结构如下图所示：

37

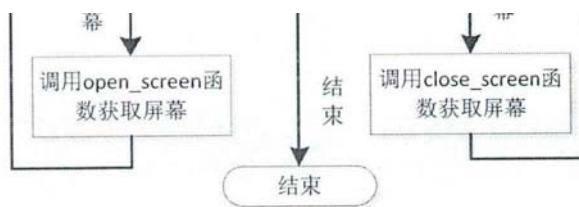


图 4-6 屏幕切换流程

4.3.3 输入设备复用

4.3.3.1 输入设备复用设计

由于多个 Android 系统使用同一套 Linux 内核，多个输入设备，如触摸屏，按键等，会被多个 Android 系统所共用，每个输入设备的事件都会发送到所有的 Android 系统中，但是对于虚拟化后的多个 Android，在同一个时刻运行在前台的系统只有一个，输入设备的事件消息只需要运行在前台的 Android 系统处理就可以。Android 系统是通过读取 Linux 输入设备文件来获取输入的信息事件，这些设备文件位于/dev/input 目录下，比如键盘设备对应的是 event2。用户输入系统的结构如下图所示：

37

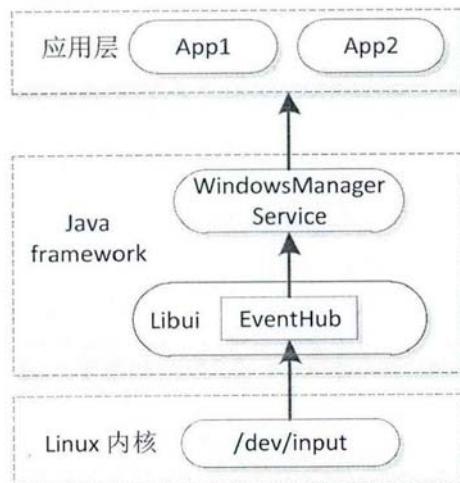


图 4-7 输入系统结构图

由上图 4-7 我们可以看出，Android 系统输入操作交给上层应用程序前是由 WindowsManagerService 服务处理，该服务收到的信息来自于下一层 EventHub 模块，EventHub 是 libui 的一部分，它实现了 input 输入设备驱动的控制，并且从中获取输入操作事件和信号，所以我们可以修改 EventHub 模块判断当前收到的输入事件是否需要向上传递，如果当前输入事件不是发给本 Android 系统的，则直接将输入事件丢弃，这样上层 Android 应用就不会收到输入事件进行处理。输入设备复用的方法就是通过 Android 系统底层过滤输入事件消息。

4.3.3.2 输入设备复用实现

从上一小节的分析可以看出，我们需要修改 Android 系统中 eventHub 模块相关的代码，通过对该模块代码逻辑的整理分析，我们定位到消息处理相关的函数位于 getEvents 函数中，该函数首先会查找所有的输入设备并打开，然后获取各输入设备中的输入事件最后将各输入事件向上传递。因此我们在 getEvents 函数中加入自己的相关代码，使 Android 系统能够过滤掉非本系统相关的输入事件，具体实现如下：

- a) 在查询设备本系统设备之前，打开虚拟 tty0 设备，通过 ioctl 函数定位到对应的 tty 设备，即本系统对应的控制终端。
- b) 获取输入显示模块中的 surfaceFlinger 正在使用的 tty 设备，然后与步骤 a

中获取的 tty 设备对比，如果相同则当前系统占据着显示输出，输入的事件是发给本系统，继续查询设备和转发输入事件；如果 tty 设备不相同，说明不是本系统的事件，后续不处理，即丢弃设备中输入事件。

加入上述代码逻辑后，原来的 getEvents 函数流程如下图 4-8 所示：

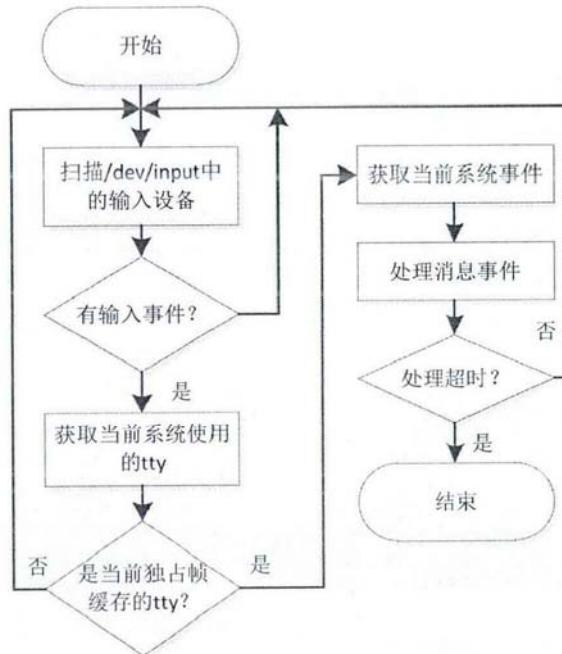


图 4-8 getEvents 函数流程图

4.4 Binder 驱动复用

4.4.1 Binder 驱动简介

Binder 驱动是 Android 系统中最重要的通信机制，Binder 驱动采用的架构是 C/S 架构，基于 Binder 通信的各组件包括 Binder 驱动，Service Manager，服务端和客户端等，其中客户端，服务端和 Service Manager 是运行在用户空间的进程，Binder 驱动运行在内核空间，Service Manager 和 Binder 驱动是 Android 系统中固有的程序，开发者根据规范实现客户端和服务端，就可以实现基于 Binder 的进程间通信。

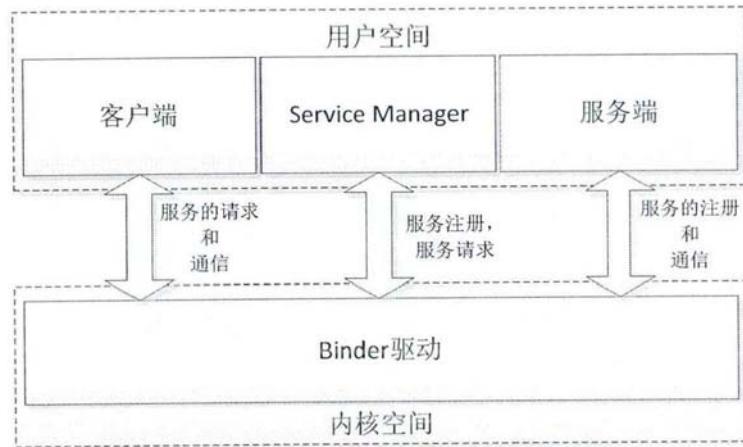


图 4-9 Binder 驱动架构

上图 4-9 是 Binder 驱动, Service Manager, 客户端和服务端四组件之间的关系, 首先服务端进程将自己的 Service 通过 Binder 驱动注册到 Service Manager 中, 客户端要使用某个服务时, 通过 Binder 驱动从 Service Manager 中获取该服务相关信息, 然后客户端根据获得的服务信息和服务端建立通信, 然后客户端和服务端就可以通过 Binder 驱动进行正常的通信交互。由此可见, Binder 驱动是 Service Manager、服务端和客户端三者之间的桥梁, 三者都是通过打开 Binder 设备进行 I/O 操作来传递和相应消息。

4.4.2 Binder 驱动的设计

在基于 LXC 的 Android 虚拟化中, 多个 Android 系统使用同一个 Linux 内核, 而 Binder 驱动属于 Linux 内核, 即多个 Android 系统会使用 Binder 驱动, 但是 Android 系统中 Binder 驱动实现的时候未考虑多个系统使用的情况, 如果直接让多个系统访问, 原生的 Binder 驱动只能注册一个 Android 宿主机的 Service Manager, 即句柄为 0 的标号, 其它 Android 客户机如果再以 0 号句柄进行注册就会失败, 如果以其它句柄号注册则无法标识为 Service Manager, 所以宿主机和客户机系统中只能通过 Binder 驱动注册一个 Service Manager。如果整个系统中只存在一个 Service Manager, 客户机和宿主机中的服务都会注册到该 Service Manager 中, 但是宿主机和客户机中很多服务名字是相同的, 注册时会产生冲突, 导致后

续的服务无法注册或者 Service Manager 无法区分这些服务是来自哪个 Android 系统，如哪些服务属于客户机，哪些服务属于宿主机，所以我们需要在 Binder 驱动中将不同 Android 系统的服务进行区分。

基于上述几个问题，我们设计了虚拟 Binder 驱动，其基本框架如下图 4-10 所示：

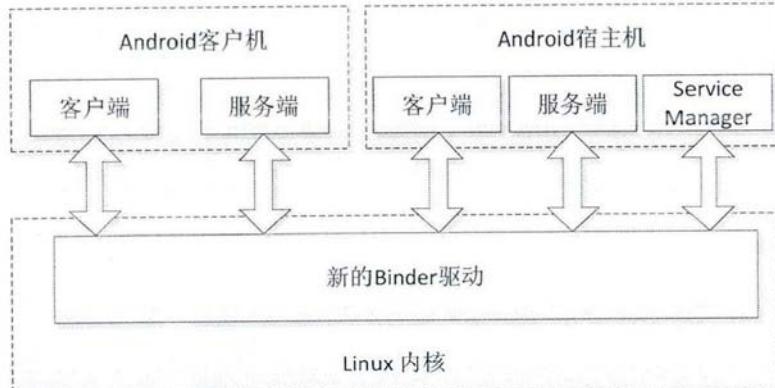


图 4-10 新的 Binder 驱动架构

在设计的框架中，我们主要的工作是在 Linux 内核中修改了原来的 Binder 驱动，该驱动能够被多个 Android 客户机使用。在 Android 客户机文件系统中创建 /dev/binder 设备空文件，并将该空文件和 Linux 内核创建的新 Binder 设备文件绑定，这样 Android 客户机访问 Binder 设备时打开该空文件相当于打开我们实现的新 Binder 设备文件，即访问新的 Binder 设备驱动中的函数进行操作。

在整个系统中只启用了一个 Service Manager 服务，该服务在宿主的 Android 系统中启动，管理了 Android 宿主系统和 Android 客户系统中的所有服务，在新的 Binder 驱动中，我们主要加入了一个新的功能，使 Binder 驱动能够区分来自上层的不同 Android 系统的服务，将不同 Android 系统中相同命名的服务进行区分，使得不同的 Android 系统的服务能够正确的注册到 Service Manager 服务中，并且不同的 Android 系统的客户端能够通过新的 Binder 驱动获取自己系统的中对应的服务。

4.4.3 Binder 驱动的实现

按照 Linux 系统定义的驱动开发框架, Binder 设备驱动属于 misc 设备, misc 设备就是简单的字符设备驱动, Linux 将这些具有共同特性的驱动抽象出共同的 API 接口, 用于简化设备驱动的初始化工作, 所有 misc 设备主设备号都为 10。在 Linux 一切皆文件的思想下, 驱动也是通过文件的形式定义, 首先我们需要创建一个 `file_operations` 类型的数据结构体, 变量名定义为 `mybinder_fops`, 在该结构体中, 主要定义了驱动的多种操作, 即与新的 Binder 设备的操作一一对应。我们需要实现的操作接口函数集合如下表 4-2 所示:

表 4-2 Binder 驱动接口函数

函数名	对设备操作的作用	实现方法
open	打开 binder 设备	使用原始 Binder 驱动的 <code>binder_open</code> 函数
poll	查询 binder 设备, 判断是否可以进行非阻塞地读	使用原始 Binder 驱动的 <code>binder_poll</code> 函数
unlocked_ioctl	向 binder 设备发送控制命令	自定义实现 <code>mybinder_ioctl</code> 函数
compat_ioctl	用于 64 位系统中的 32 位程序运行, 作用同上	自定义实现 <code>mybinder_ioctl</code> 函数
mmap	用于映射 binder 设备的内存空间到进程的地址空间	使用原始 Binder 驱动的 <code>binder_open</code> 函数
flush	刷新已经写入 buffer 的 I/O 操作	使用原始 Binder 驱动的 <code>binder_open</code> 函数
release	释放 binder 设备	使用原始 Binder 驱动的 <code>binder_open</code> 函数

从上表中我们可以看出, 对虚拟 binder 驱动大部分操作都保持了原生 binder 驱动中的操作, 需要修改的的函数接口就是 `mybinder_ioctl`, 该函数主要实现的功能是过滤和修改相应的 Android 系统服务名称, 而剩下的操作我们的实现方式还

是直接调用原生驱动中的 ioctl 函数，直接调用的方式比较简单，在此不再详细描述。在我们自定义的 mybinder_ioctl 中，我们需要解析出所有的命令操作，主要包括命令号和命令参数，然后从命令参数中解析出对于 binder 操作的子命令，通过分析子命令的含义，判断各个子命令是否需要修改。

服务名过滤转换流程如下图 4-11 所示：

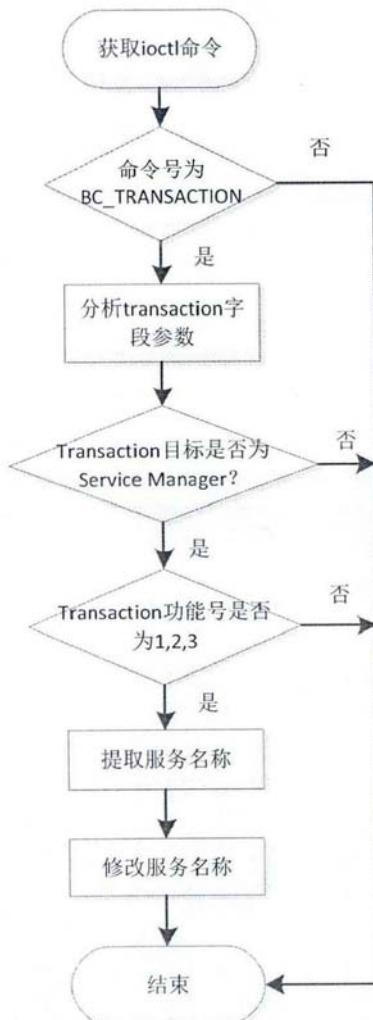


图 4-11 服务名过滤转换流程图

binder 事务命令结构 binder_transaction_data 有一个 code 字段，该字段描述了对 binder 设备的执行操作，检查的操作有以下三种：

- SVC_MGR_ADD_SERVICE 向 Service Manager 增加服务

- b) SVC_MGR_GET_SERVICE 向 Service Manager 获取服务
- c) SVC_MGR_CHECK_SERVICE 向 Service Manager 查询服务

对于这三种类型的操作，我们加入了 replace_name(old_name)函数对服务名进行转化，即新的服务名 new_name = replace_name(old_name)，其中 old_name 参数就是原来的服务名称。

在函数 replace_name 中，我们首先通过 tty 设备号确定当前 Android 系统，流程与输入设备复用相似。首先打开当前系统的虚拟的 tty0 设备，然后同过 ioctl 函数获取对应的 tty 设备，比如 tty0 对应的 tty 是 tty7，则参数 n 的值为 7，即当前系统的 id 为 7，在老的服务名称中加入 id 号 7，即重新生成该系统中的注册的服务名经过 replace_name 函数转换后会加入 id 号 7，这样不同的 Android 系统中的服务名就不会重复。replace_name 函数执行过程如下所示：



图 4-12 修改服务名称流程

4.5 本章小结

在本章我们给出了 Android 系统虚拟化的详细设计与实现方法，本章主要分为四个部分，第一部分介绍了本论文设计的 Android 虚拟化的总体设计方案及需

要解决的问题；第二部分为 LXC 工具移植到 Android 系统的方法，介绍了遇到的问题的解决方法，使我们虚拟化依赖的 lxc 工具能够在 Android 系统上正常运行；第三部分是输入设备和显示输出设备复用设计与实现，修改 Android 客户机中相关代码，我们实现了多个 android 系统对输入与显示输出设备的复用；第四部分详细介绍了 Binder 驱动的设计和实现，给出了我们解决多个 Android 系统共同使用一个 Binder 驱动的具体方法。

第5章 系统功能测试与性能演示

Android 系统虚拟化的目标是完成系统之间的隔离，应用程序能够在各自的系统中独立运行而互不影响，同时也要减少虚拟化带来的系统开销。本章将对基于 lxc 的 Android 系统虚拟化方案进行测试，主要验证整个系统各功能能否正常运行，而对于性能测试，由于测试环境 virtualbox 虚拟机的影响，只做了简单的内存和 cpu 使用情况测试。

5.1 测试环境部署

为了方便调试和开发，我们使用了 Android_x86 项目作为我们 Android 系统虚拟化系统。Android 系统基本都是运行在 ARM 平台上，如果进行 Android 开发需要在实际的配备 arm 芯片的物理机上运行，开发测试都很不方便，而在 X86 平台下的模拟器又很慢，所以有了 Android_x86 开源项目，该系统不需要模拟 arm 指令，直接可以在普通 pc 或虚拟机中运行。我们使用 virtualbox 作为我们的虚拟机。

测试环境的硬件参数配置如下表 5-1 所示：

表 5-1 硬件参数配置

	部件	参数
PC 物理机	CPU	英特尔 Core i5-2500K @3.3GHz 四核四线程
	内存	DDR3 SDRAM 8G 1333MHz
	硬盘	希捷 ST31000524AS 1TB 7200RPM
	显卡	英伟达 GTX550 Ti 950MHz
Virtualbox	CPU	单核
	内存	1G
	硬盘	16G

Android_x86 系统的编译和安装步骤如下：

1) 安装 repo 工具，用于下载 Android_x86 源码

2) 下载 Android_x86 源码，执行如下命令：

```
repo init -u git://git.android-x86.org/platform/manifest.git
```

3) 按照第 4 章表 4.1 所示，开启 linux 内核相关配置，按照第四章设计实现修改相关代码，编译 android 系统，生成可安装的 iso 镜像。

4) 按照 virtualbox 虚拟机的安装步骤，配置好 virtualbox 参数，生成虚拟硬盘后，将步骤 3 中的 iso 镜像安装到 virtualbox 中。

通过上述步骤就可以将完成 Android 系统测试环境相关的硬件和软件配置。

5.2 功能测试

功能测试是 Android 虚拟化方案验证的最重要部分，主要包括在 virtualbox 下启动 Android 宿主机，在宿主机中验证 lxc 工具功能，启动 Android 系统客户机，两个 Android 系统之间的切换，应用程序在各虚拟机之上的运行情况及 Android 系统之间的隔离情况。

(1) Android 宿主机启动及 lxc 工具功能验证测试。

我们运行的 Android 系统安装到 virtualbox 后就可以直接启动，进入启动图像界面后我们可以按键 Alt+[F1] 进入字符界面，在这 shell 环境下，用户可以使用一些命令验证当前宿主机 lxc 功能的支持及使用情况。首先使用 lxc-checkconfig 命令可以查看当前 Linux 内核是否已经开启 namespace 和 cgroup 机制，从图 5-1 可以看出所有 lxc 工具需要使用的内核特性都已经开启；lxc-ls 命令可以查看当前系统中使用 lxc 已经启动的容器的名字，当使用 lxc-start 启动一个容器时，会显示该名字；使用 lxc-monitor 命令可以检测当前运行的某个容器中程序的运行情况；测试完所有的 lxc 工具中的命令后，可以确认 lxc 工具已经移植到 Android 系统下，而且都能够正常运行。

```
--- Namespaces ---
Namespaces: enabled
Utsname namespace: enabled
Ipc namespace: enabled
Pid namespace: enabled
User namespace: enabled
Network namespace: enabled
Multiple /dev/pts instances: enabled

--- Control groups ---
Cgroup: enabled
Cgroup clone_children flag: enabled
Cgroup device: enabled
Cgroup sched: enabled
Cgroup cpu account: enabled
Cgroup memory controller: enabled
Cgroup cpuset: enabled

--- Misc ---
Veth pair device: enabled
Macvlan: enabled
Vlan: enabled
File capabilities: enabled
```

图 5-1 lxc-checkconfig 运行效果

2) Android 客户机的启动功能测试。

在启动 Android 客户机前，使用 mount 命令将 cgroup 文件系统进行挂载。然后在 Android 宿主机的控制台下执行 lxc-start -n Android4.2 命令启动客户机，客户机启动时会处于进入前端桌面，而宿主机将进入后台，屏幕显示出“ANDROID”启动画面，这代表 Android 客户机正在初始化，由于不需要启动 linux 内核，客户机很快就会初始化完毕进入用户操作界面。下图 5-2 和 5-3 分别是 Android 宿主机和 Android 客户机系统信息，可以看出我们的宿主机系统是 Android4.3，并且是中文版设置，而客户机系统是 Android4.2，是英文版设置。从上述测试结果来看，在 Android 虚拟化环境下，使用 lxc 工具能够正常启动 Android 客户机并且能够正常运行。

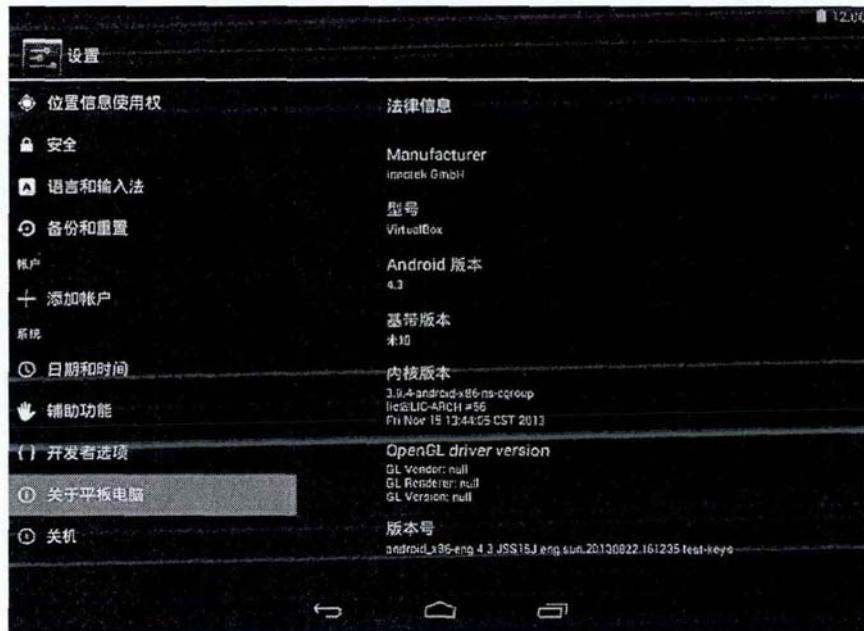


图 5-2 Android 宿主机系统信息

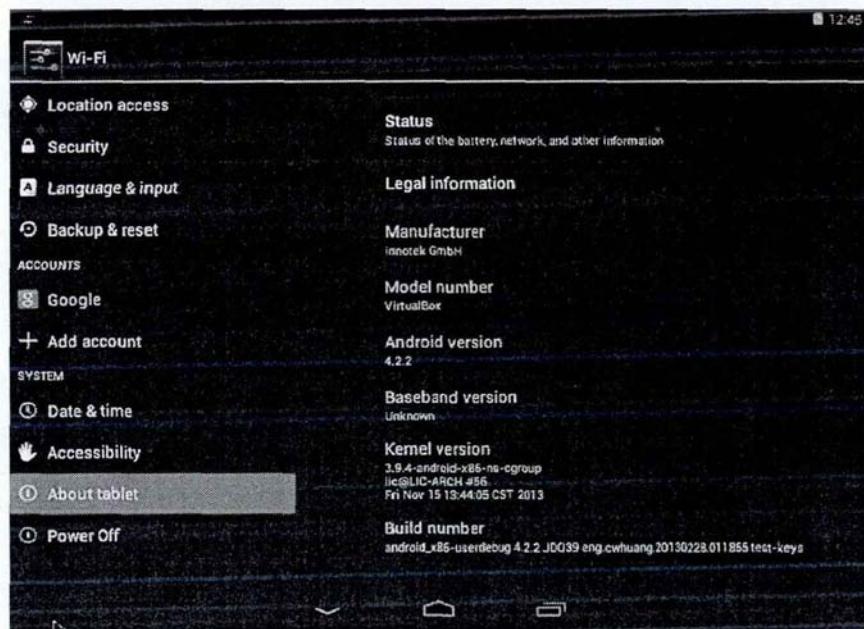


图 5-3 Android 客户机系统信息

3) Android 系统之间的切换功能测试。

启动 Android 客户机后，客户机在前台显示，现在所有的输入操作是针对该

Android 客户机的，Android 宿主机则运行在后台。按下组合按键 Alt+[F4]就可以进入 Android 宿主机中，即宿主机切换至前台显示，当前所有的输入操作针对该 Android 宿主机，组合按键 Alt+[F7]则有可以将 Android 客户机切换至前台，切换至宿主机命令下，运行 lxc-stop 命令停止 Android 客户机系统的运行。经过测试，Android 系统之间的切换快速，没有延时，并且不同的 Android 系统能够正常响应输入操作，如滑屏解锁，字符输入，点击应用等，这说明我们输入设备和输入设备复用成功，多个 Android 系统之间能够正常使用同一个设备，前台系统能够正常处理输入与输入时间消息而后台系统则屏蔽相应的输入与输出消息。

4) 应用程序的运行。

由于 Android X86 系统是无法使用 Android 市场的 app，因为该系统授权机制不一样，需要在设置->安全->未知来源中设置，允许未知来源的应用，然后通过系统自带的浏览器下载，就可以正常安装该应用程序。如下图 5-4 所示，分别在 Android 宿主机和客户机中打开浏览器，在宿主机中使用百度搜索“元旦”，在客户机中使用 google 搜索“元旦”，都能够正确返回结果，该测试说明多个 Android 能够复用网络设备，进行正常的网络通信。同时，还可以运行各种游戏，能够流畅运行并且通过组合按键，在不同的 Android 系统间进行切换。经测试表明，Android 系统虚拟化后应用程序能够正常运行，并且两个系统直接能够进行正常的前后台显示切换。



图 5-4 Android 宿主机和客户机程序运行效果

5) 隔离性测试

Android 采用虚拟化技术之后,不同的系统之间是完全隔离的,运行在 Android 之上的应用程序无法感知到对方的存在。隔离性测试上,我们自己写了 Test app 应用程序,该程序主要是获取系统中运行的应用和可访问文件系统,并且给该程序所有的用户访问权限,由于 Android 系统运行在 lxc 启动的 container 中,Test 程序根本无法感知对方的存在,即无法获取对方运行的 pid 等一切信息,也无法知道对方的文件系统视图。同时,使用第三方应用 Root Exploer 文件管理器访问文件系统视图,RE 管理器无法知道不同 Android 系统的文件目录结构,如下图 5-5 所示。



图 5-5 RE 管理器访问文件系统效果

通过上述几个方面的功能测试验证,使用 lxc 工具能够实现 Android 系统操作系统的虚拟化,并且加入 Binder 虚拟驱动,输入设备与显示输出设备复用的设计后,Android 系统能够共同使用 Linux 内核层的设备并且正常运行,同时也验证了 Android 虚拟化之后系统之间的隔离性,说明我们设计的 Android 虚拟化方案可行,Android 系统中基本功能能够正常使用。

5.3 性能测试

在本课题中,Android 虚拟化在性能上并没有过多的考虑,只是设计了共享服务一点来优化虚拟化带来的性能影响。从方案原理来说,使用 lxc 工具进行操作系统级虚拟化已经减少了虚拟化 vmm 层的开销,客户机在本方案中相当于一

个进程在运行，所以虚拟化本身几乎没性能上的损耗。本节将从 Android 操作系统级虚拟化后的内存使用情况进行性能上的统计与分析。

(1) Android 系统内存使用情况测试

测试方法及结果：在 Android 系统中，我们通过启动不同的 Android 系统个数来统计 Android 系统的内存使用情况，分析 Android 虚拟化所带来的性能开销。在系统中，我们连续启动了三个 Android 系统，每个系统启动后只运行系统所需的进程服务，我们没有运行任何其他应用程序，如游戏，视频等，通过/proc/meminfo 文件信息统计，我们得出如下图 5-6 所示结果。

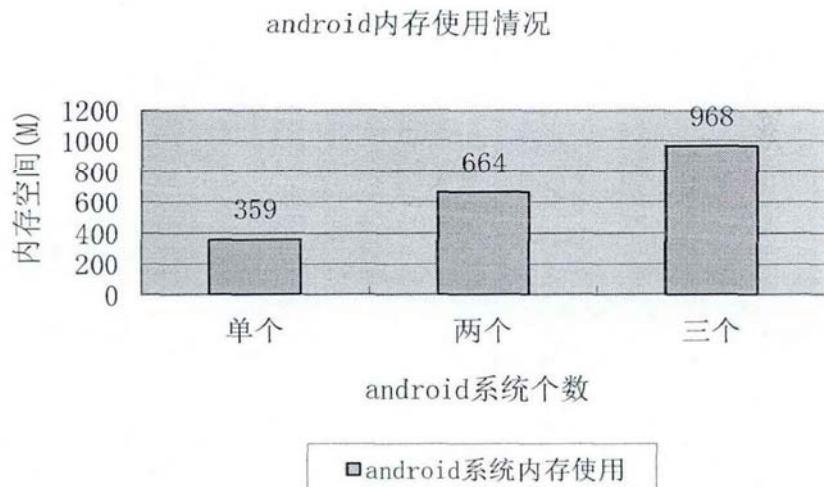


图 5-6 Android 系统内存使用示意图

测试结果分析：当单个 Android 宿主机启动后，系统占用的内存为 359M，而第二个 Android 系统启动后内存大小为 664M，即增加了 305M 内存，说明客户机 Android 系统使用的内存比宿主机小，而第二个 Android 客户机系统也使用了 304M 的内存。从中可以看出，lxc 工具启动 Android 客户机共享使用了 Linux 内核，节省了相应的内存空间。

(2) Android 应用 cpu 使用率测试

测试方法及结果：我们分别在非虚拟化的 Android 系统和虚拟化的系统中安装了我们常用的豌豆荚应用，然后统计豌豆荚从启动到应用下载安装过程中 cpu 的使用率情况，统计命令使用 top -d 1 | grep wandoujia 每隔 1 秒钟获取豌豆荚应

用cpu使用率情况，统计如图5-7所示。



图 5-7 Android 应用 cpu 使用率

测试结果分析：从统计图中我们可以看出，时间段1~4是豌豆荚应用的启动时间，总体而言非虚拟化系统中启动较快，虚拟化后损耗了部分性能，但在时间点3上虚拟化后的系统较快，可能是因为我们统计的时间间隔为1秒，不够精确；时间段5~8中是我们使用豌豆荚操作的时间，两种场景使用率基本相似；时间段9~12是下载应用时的使用率，使用率在8%左右，整体也接近。从豌豆荚使用过程中cpu的使用率可见，Android系统虚拟化后基本不影响应用的运行，性能与原生Android系统较接近。

5.4 本章小结

本章我们介绍了测试环境的软硬件配置及部署方式，然后重点测试了整个Android系统虚拟化后基本功能是否正常运行，主要验证的功能有Android宿主机中lxc工具，Android客户机的启动，不同Android系统的切换和隔离性，上述功能经测试后都能够正常运行。最后做了简单的内存性能测试，统计了多个Android系统下内存使用情况。从测试结果上可知，我们设计的虚拟化方案能够做到系统间的隔离，减少虚拟化带来的内存开销。

第6章 总结与展望

6.1 本文工作总结

随着 Android 智能手机的不断普及，手机硬件配置的不断升级，Android 虚拟化技术也会被越来越多的公司所采用，Android 虚拟化可以充分利用硬件资源，虚拟化后隔离的多个 Android 系统在安全领域也可以广泛应用。本文针对 Android 虚拟化提出了一种新的虚拟化方案---基于 lxc 的操作系统级虚拟化，相比于其它 Android 虚拟化技术，本文设计的方案优点是多个 Android 虚拟化系统使用同一个 Linux 内核，这样可以充分利用 Android 宿主机与客户机之间共用的资源，能够降低虚拟化带来的性能开销，同时该方案不用太多的修改 Android 和 Linux 内核的源代码，减少不同硬件平台的 Android 移植工作。为了实现本课题提出的设计方案，我们完成了以下的工作：

- 调研 Android 系统架构和 Android 系统采用的安全技术，分析本课题 Android 系统虚拟化的必要性。
- 研究了虚拟化技术，主要调研了从不同层次的虚拟化方案的优缺点，同时调研了目前业界内 Android 系统虚拟化的解决方案和产品。
- 研究了与本课题相关的关键技术，重点分析了使用的 lxc 工具的实现原理。
- 设计并实现了基于 lxc 的 Android 系统虚拟化原型，主要实现了 lxc 工具 Android 平台的移植，重新修改实现 Binder 驱动，及输入与显示输出设备的复用。

6.2 工作展望

由于时间有限，也因本人能力的限制，本文中 Android 虚拟化系统中还有些功能还未完善或开发，在此，我们对这些问题说明并对未来工作进行展望：

- 完善输入与显示输出设备复用支持，实现设备虚拟化。由于 Android 输入设备与显示输出设备较多，本文设计的方案需要修改 Android 客户机系统代码，在上层输入与显示输出接口实现了设备复用。但通过实现底层设备的虚拟化可以减少或不修改上层 Android 系统的修改，更好的解决设备复用问题。
- 优化多个 Android 系统中内存使用。Android 宿主机与客户机系统很多模块功能相同，可通过共享不影响隔离性的模块达到内存优化，进而提升系统性能。
- 将本系统移植到不同硬件设备上。由于时间有限，本方案只在 virtualbox 环境下进行了测试验证，真正被用户使用需要移植到不同的智能手机上，这需要针对不同的手机厂商 rom 进行修改，可能会碰到更多的问题，完成更多的工作。

参考文献

- [1] Margaret Butler. Android: Changing the Mobile Landscape[J]. *Pervasive Computing*, IEEE, 2011-1:4-7
- [2] About Android[EB/OL]. <http://developer.android.com/about/index.html>
- [3] Stefan B. Analysis of the android architecture[M]. Karlsruhe institute for technology, 2010-06
- [4] 杨丰盛. Android 技术内幕-系统卷[M]. 机械工业出版社, 2011-11:1-33
- [5] Android Security Overview[EB/OL].
<http://source.android.com/devices/tech/security/#introduction>
- [6] Enck W, Ongtang M, McDaniel P. Understanding android security[J]. *Security & Privacy*, IEEE, 2009-6:50-57
- [7] Shabtai A, Fledel Y, Elovici Y. Securing Android-powered mobile devices using SELinux[J]. *Security & Privacy*, IEEE, 2010, 8(3): 36-44.
- [8] Rosa T. Android Binder Security Note: On Passing Binder Through Another Binder[EB/OL]. <http://crypto.hyperlink.cz/files/xbinder.pdf>, 2011-11-11
- [9] Shabtai A, Fledel Y, Kanonov U, Elovici Y. Google Android: A Comprehensive Security Assessment[J]. *Security & Privacy*, IEEE, 2010-3:35-44
- [10] 虚拟化技术漫谈[EB/OL]. <http://www.ibm.com/developerworks/cn/linux/l-cn-vt/>
- [11] 金海等.计算系统虚拟化-原理与应用[M].清华大学出版社, 2010-9
- [12] 严迎建, 徐劲松, 陈韬等. 基于指令集模拟器的处理器建模与验证[J]. 计算机工程, 2008, 34(5): 248-250
- [13] Doorn L. Hardware virtualization trends[C]. VEE. 2006, 6: 45-45.
- [14] Uhlig R, Neiger G, Rodgers D, et al. Intel virtualization technology[J]. Computer, 2005, 38(5): 48-56
- [15] Menascé D A. Virtualization: Concepts, applications, and performance modeling[C]. Int. CMG Conference. 2005: 407-414.
- [16] Kolyshkin K. Virtualization in Linux[J]. White paper, OpenVZ, 2006.

- [17]Lindholm T, Yellin F, Bracha G, et al. The Java virtual machine specification[M]. Addison-Wesley, 2013.
- [18]Andrus J, Dall C, Hof A V, et al. Cells: a virtual mobile smartphone architecture[C]. Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. ACM, 2011: 173-187.
- [19]Barr K, Bungale P, Deasy S, et al. The vmware mobile virtualization platform: is that a hypervisor in your pocket? [J]. ACM SIGOPS Operating Systems Review, 2010, 44(4): 124-135.
- [20]Hwang J Y, Suh S B, Heo S K, et al. Xen on ARM: System virtualization using Xen hypervisor for ARM-based secure mobile phones[C]. Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE. IEEE, 2008: 257-261.
- [21]Heiser G. The role of virtualization in embedded systems[C]. Proceedings of the 1st workshop on Isolation and integration in embedded systems. ACM, 2008: 11-16.
- [22]Paul K, Kundu T K. Android on mobile devices: an energy perspective[C]. Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on. IEEE, 2010: 2421-2426.
- [23]Lawton K P. Bochs: A portable pc emulator for unix [J]. Linux Journal, 1996, 1996(29es): 7.
- [24]LXC: Linux container tools[EB/OL]. 2009-02-03.
<http://www.ibm.com/developerworks/library/l-lxc-containers/>
- [25]Biederman E, Networx L. Multiple instances of the global linux namespaces[C]. Proceedings of the Linux Symposium. 2006.
- [26]Mauerer W. Professional Linux kernel architecture [M]. Wiley. com, 2010.
- [27]clone (2) - Linux man page[EB/OL]. <http://linux.die.net/man/2/clone>

攻读硕士学位期间主要的研究成果

- [1] 参与“HA 虚拟化集群高可用”项目，主要负责系统的开发与测试工作，时间范围从 2011.10 至 2012.3
- [2] 参与“DXVA 视频重定向”项目，主要负责系统的设计，WDDM 驱动开发，时间范围从 2012.6 至 2013.5
- [3] 参与“Android 系统虚拟化”项目，主要负责相关技术调研与设计，lxc 工具研究，时间范围从 2013.6 至今

致谢

时光飞逝，两年多的研究生学习生涯就要过去，在这几年中的学习过程中，在知识和做事方式上都收获很多。

首先我要真诚的感谢我的导师陈文智教授，在研究生期间，在他的指导下我能够从项目实践中不断的学习新的知识，了解当前学术界和工业界最新最前沿的计算机技术。陈老师建立的实验室严谨的学习氛围，使我学会了良好的学习与工作习惯，最重要的是培养了自我学习的能力，养成一个良好的学习与生活心态。感谢陈老师的悉心教导和培养，衷心祝福陈老师身体健康，事事顺心，阖家欢乐！

同时，我也要感谢王总辉老师，在完成实验室课题项目时得到了王老师的很多指导，当遇到问题时，从与王老师的讨论中能够获取快速解决问题的方法。

本论文在完成过程中得到了实验室相应项目的组长徐磊博士的指导，同时也感谢其他项目成员孙伟杰，李川，李国玺同学支持，在系统的搭建与测试上给了我无私的帮助，论文写作过程中也给了许多意见，在此表示由衷感谢。

然后感谢实验室的各位同学，两年多的朝夕相处，共同的学习讨论，让我的学习生涯充满欢乐，谢谢你们的帮助和鼓励。

最后，我要感谢我的家人，感谢你们的关心与支持，给予了我学生生涯中的物质与精神支持，给予了我最无私的帮助。

署名 吴佳杰

当前日期：2014年1月6日