控阅

中国科学技术大学

工程硕士学位论文

基于 Driver 命名空间的 Android 操作系统虚拟化设计与实现

作者姓名：　　刘博文

专业领域：　　计算机技术

导师姓名：　　顾乃杰 教授

企业导师：　　林传文 高工

完成时间：　　二〇一七年六月

Controlled reading

University of Science and Technology of China

A Dissertation for Master's Degree of Engineering

Design and Implementation of Android OS-level Virtualization Based on Driver Namespace

Author's Name: Bowen Liu

Speciality: Computer Technology

Supervisor: Prof. Naijie Gu

Senior Eng. Chuanwen Lin

Finished time: June, 2017

中国科学技术大学学位论文原创性声明

本人声明所呈交的学位论文,是本人在导师指导下进行研究工作所取得的成果。除已特别加以标注和致谢的地方外，论文中不包含任何他人已经发表或撰写过的研究成果。与我一同工作的同志对本研究所做的贡献均已在论文中作了明确的说明。

作者签名：_____　　签字日期：_____

中国科学技术大学学位论文授权使用声明

作为申请学位的条件之一，学位论文著作权拥有者授权中国科学技术大学拥有学位论文的部分使用权，即：学校有权按有关规定向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅，可以将学位论文编入《中国学位论文全文数据库》等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。本人提交的电子文档的内容和纸质论文的内容相一致。

保密的学位论文在解密后也遵守此规定。

□公开　□保密（____年）

作者签名：_____　　导师签名：_____

签字日期：_____　　签字日期：_____

摘 要

随着移动设备的硬件配置不断提高、用户使用场景的多样性与日俱增、安全与隐私问题日益凸显，移动平台上对于虚拟化技术的需求也愈演愈烈。虚拟化技术当前已广泛应用在 PC 端和高性能服务器领域，基于移动平台的研究成果却十分匮乏。传统的 Hypervisor 虚拟化架构需要在物理硬件层和虚拟机间进行繁复的指令翻译，为系统带来大量的性能损耗。而移动平台各类硬件高度集成，对性能要求严苛。

针对上述问题，本文将 Android 系统的优点与虚拟化技术相结合，基于移动 ARM 平台提出了一种轻量级的操作系统虚拟化架构。该架构在保证多个虚拟 Android 系统同时运行的基础上，实现了虚拟系统间对一套硬件设备的隔离复用。本文的主要工作如下：

(1) 针对当前虚拟化技术解决方案的不足并提出一种改进方案。分析了各类虚拟化技术的优点与不足，并深入研究了操作系统级虚拟化技术领域具有代表性的解决方案。结合研究背景与项目需求，提出一种将操作系统级虚拟化技术运用在移动 ARM 平台上的改进方案。

(2) 在内核中设计 Driver 命名空间框架。本文重点研究了 Linux 内核现有的 MNT 命名空间和 PID 命名空间资源隔离机制在用户空间和内核中的实现原理。通过对内核的修改，引入 Driver 命名空间框架，为每个虚拟客户系统提供了一个隔离的硬件访问视图。同时设计了 active-inactive 模型，保证多个虚拟客户系统对一套硬件设备的隔离复用。

(3) 设计并实现基于 ARM 平台的 Android 操作系统虚拟化解决方案。此方案由系统级启动和硬件设备隔离复用两部分组成。利用各类命名空间的集合在宿主系统上以共享内核的方式构造出两个虚拟客户系统。在共享单个内存、CPU 资源的前提下，深入研究 Android 各个硬件设备子系统，利用本文设计的 Driver 命名空间框架实现了显示设备、输入设备、背光设备以及 LEDs 设备等最核心的硬件设备虚拟化方案。该方案是一种轻量级、通用的 Android 系统虚拟化解决方案。

(4) 对虚拟化方案进行实验和分析。基于 AC8317 Cortex Dual-Core 开发板，在功能测试上对宿主系统启动、虚拟客户系统启动、系统间切换、系统间隔离性进行了完整的测试工作，测试结果显示本方案基本满足所有的 Android 特性；在性能测试上从 CPU 利用率以及内存使用量两个角度来衡量虚拟化的引入所带来的性能开销。实验结果结果表明，本文提出的虚拟化架构在 CPU 利用率上和原

生 Android 系统基本持平，内存使用量比原生 Android 系统节约了 6.7%，说明该虚拟化架构的引入没有带来额外的性能开销，具有良好的应用前景。

关键词：操作系统虚拟化　命名空间机制　Driver 命名空间框架　硬件隔离复用　Android 子系统　轻量级

ABSTRACT

As the hardware configuration of mobile devices continues to increase, the diversity of users' scenes is increasing, security and privacy issues are becoming increasingly prominent, and the demand for virtualization technology on mobile platforms is becoming more and more intense. Virtualization technology is currently widely used in the PC and high-performance server field, but the researches based on the mobile platforms are very scarce. The traditional Hypervisor virtualization architecture requires complex instruction translation between the physical hardware layer and the virtual machine, resulting in significant performance loss for the system. All kinds of hardware are highly integrated on mobile platform, which demands performance.

In view of the above problems, combines the advantages of Android system with virtualization technology, this thesis proposed a lightweight operating system virtualization architecture based on mobile ARM platform. The architecture ensures the isolation and multiplexing of a set of hardware devices on the basis of ensuring that multiple virtual Android systems run at the same time. The main work of this thesis is listed as follows:

(1) In view of the shortcomings of the current virtualization technology solutions and propose an improved solution. Analyzes the advantages and disadvantages of all kinds of virtualization technology, and deeply studies the representative solution of the operating system level virtualization technology. Combined with the research background and project requirements, an improved solution is put forward in this thesis that puts operating system level virtualization technology into the mobile ARM platform.

(2) Design the Driver namespace framework in the kernel. This thesis focuses on the realization of the existing MNT namespace and PID namespace resource isolation mechanism in Linux kernel in user space and kernel space. Introducing the Driver namespace framework by modifying the kernel provides an isolated hardware access view for each virtual Android system. At the same time, the active-inactive model is designed to ensure the isolation and multiplexting of a set of hardware devices by multiple virtual client systems.

(3) Design and implement Android operating system virtualization solutions based on ARM platform. This solution consists of system-level start-up and hardware device isolation and multiplexing. Using the collection of various namespaces to create two virtual Android systems on the host system in the form of shared one single kernel. Under the premise of sharing a single memory and CPU resources, this thesis deeply researched each hardware device subsystem of Android, and realized the core hardware subsystem virtualization such as display device, input device, backlight device and LEDs device by using the Driver namespace framework designed in this thesis. This architecture is a lightweight, universal Android system virtualization solution.

(4) Experimenting and analyzing the virtualization solution. Based on the AC8317 Cortex Dual-Core development board, a complete functional test is performed on the host system startup, virtual Android system startup, inter-system switching, and system isolation. The test results show that this solution basically meets all the Android features. In the performance test from two variables, including CPU utilization and memory usage, to measure the introduction of virtualization brought about by the performance overhead. The results show that the virtual architecture proposed in this thesis has the same performance in CPU utilization as native system, and memory usage is 6.7% less than that of the native Android system, indicating that the introduction of the virtualization architecture did not bring additional performance overhead, it has a good application prospects.

Key Words: OS-level virtualization, namespace mechanism, driver namespace framework, hardware isolation and multiplexing, Android subsystem, lightweight

目 录

图 目 录

第一章　绪论

虚拟化技术是云计算研究领域的核心。通过将各类系统资源进行模拟并统一管理，从而提高了资源的利用率，被广泛应用在高性能服务器中。虚拟化技术在 PC 领域和服务器领域的研究日趋成熟，涌现出大量优秀的虚拟化产品。随着移动终端设备需求的不断增加，将虚拟化技术应用在移动平台正在成为当前研究的一大热点。本章主要介绍本文的研究背景及意义、国内外研究现状、本文的研究内容及组织结构等。

1.1　研究背景及意义

移动终端设备当前种类繁多，用户数量激增。诸如智能手机、平板电脑等典型终端设备变得越来越普及，智能设备已占据当下移动计算领域，市场份额已超过 PC 领域[1]。

而在移动设备上，对虚拟化技术的需求正在逐渐增加[2]，原因如下：(1) 移动设备销量的巨大增长推动着移动终端设备核心技术的飞速发展，持续革新。移动设备配置越来越高，一些智能设备已和桌面设备接近，而有些高端配置的移动设备甚至已经超过 PC 设备[3]。以华为 Mate9 手机为例，采用 8 核海思处理器，4GB 的内存配上 64G 的存储空间，这样的配置完全可以与一台 PC 电脑相当。科技的飞速发展为虚拟化奠定了技术基础；(2) 使用移动终端的用户对于设备使用场景的多样性与日俱增[4]。和以往移动设备仅用来接听电话、收发信息、日常娱乐等常见功能相比，当前的移动设备还经常用于工作办公、商务洽谈、科研任务等场景。使用场景的多样化为虚拟化技术提供了平台基础；(3) 多用户需求的出现。对于使用智能手机、平板电脑的用户来说，单一的用户模式已不能满足其需求。比如对于普通职员来说，忙碌的工作与日常生活需要切换不同的相互隔离的使用环境；而对于大多数家长来说，希望小孩能在一个特定且受限的使用环境中合理使用移动设备；(4) 安全与隐私问题的日益凸显[5]。用户依赖于多个移动设备来适应工作和个人对于灵活性的需求，不间断的访问将为移动设备引入新的安全和隐私问题，包括软件账密码、邮箱密码、理财软件的支付密码等。因此，在一个独立、隔离的运行环境中运行个性化的软件是当前更为安全且合理的做法。

虚拟化技术作为计算机技术发展的一个重要分支，由于其自身开销小、轻量级等优点，成为当前研究的热点[6]。传统的虚拟化技术通过在计算机系统中引入

虚拟层，用来屏蔽上层应用程序与底层硬件平台的不兼容性。与此同时，一个宿主系统与多个虚拟系统的共存也使得物理资源得到了充分的利用，为计算机领域带来了新的活力。目前，虚拟化技术在实现原理上主要分为硬件分片虚拟化技术、全虚拟化技术、半虚拟化技术（也称作准虚拟化技术）以及操作系统级虚拟化技术四大类[7]。

硬件分片技术将硬件资源分割成若干区域，每个区域可以运行独立的操作系统，不具有良好的灵活性；全虚拟化技术和半虚拟化技术则采用传统的 Hypervisor 架构，硬件层和虚拟机之间需要进行大量的指令翻译，效率低；而操作系统级虚拟化技术则是基于宿主系统以共享内核的方式构造出多个隔离的运行环境，这些运行环境保证了应用程序完整而隔离的运行[8]。操作系统级别的虚拟化解决方案不需要进行指令翻译并且没有虚拟层的管理开销，和原生系统性能基本保持一致。减少了软件冗余，最大限度的共享硬件资源[9]。

传统虚拟化技术的研究方向集中在 PC 端和服务器上，近年来随着移动设备的飞速发展，将虚拟化技术与移动设备相结合成为一个研究热点，它可以解决如下几个问题：(1) 解决了个人的隐私数据使用问题。操作系统级虚拟化构造出的各个可运行环境相互隔离独立，因此宿主系统与虚拟系统之间的隔离性保障了独立环境中的数据隐私性；(2) 解决了多用户使用角色、多样使用环境的需求。由于每次在不同虚拟操作系统之间切换时开销较小，用户可以定制出具有特殊需求的环境，针对不同的工作、生活环境切换到不同的虚拟系统中，满足了多用户多环境的需求[10]；(3) 降低了成本。基于一套硬件设备运行多个系统实例，既节约了成本也提高了竞争力。

当前虚拟化技术在移动领域的研究成果较为匮乏，且传统的 Hypervisor 虚拟化架构开销大、效率低[11]，针对上述问题，本文基于移动 ARM 平台，设计并实现了一种轻量级操作系统虚拟化架构。通过在 Linux 内核命名空间机制中引入 Driver 命名空间框架，支持在一套硬件设备上同时运行多个虚拟 Android 系统。同时提出 active-inactive 模型，允许多个虚拟系统实例隔离复用硬件设备。该架构是一种操作系统级虚拟化领域研究的新方案。

1.2　国内外研究现状

桌面与服务器端虚拟化领域一直是科学研究的重点，技术已趋近成熟，衍生出大量成熟的虚拟化产品[12]，其中典型的代表就是 2007 年由 Andrew、Avi Kivity 等人完成的 KVM(Kernel-based Virtual Machine)[13]。作为一种内核模块的实现方

案，它针对运行在 x86 体系上的硬件设备，内核只要加载该模块就会成为一个虚拟管理层。KVM 目前已广泛应用于各类数据中心，显著提高了服务器的效率。随着移动终端的爆炸式增长，近年来对于虚拟化领域的研究也逐渐转移到了移动端虚拟化上面来，目前国内外关于移动虚拟化的研究有如下成果。

1.2.1　OKL4项目

OKL4 Microvisor 由 Open Kernel Labs 实验室开发完成，是一个基于移动平台的先进且安全的虚拟化架构[14]。作为专门致力于嵌入式系统的微监管程序，OKL4 当前支持包括 ARMv5、ARMv6、ARMv7、ARMv7ve、ARMv8 在内的各类 ARM 处理器虚拟化实现方案[15]。

OKL4 将系统中的硬件设备、虚拟机、应用程序等资源分割成隔离的分区，被称为若干安全单元。Microvisor 实现在内核层，用来管理每个安全单元。与此同时，OKL4 设计了一种针对性强、通用性的进程间通信机制用来实现分区间、应用程序与硬件设备间的高效合作[16]。OKL4 Microvisor 提供高性能，高度安全和灵活的平台，是一种基于功能的访问控制、安全通信和高级驱动程序共享的移动平台虚拟化解决方案。但是，目前该方案的实现还需要硬件设备的支持与仿真的同步配合，由于移动设备面对当前层出不穷的硬件设备来说是一种繁重的工作。

1.2.2　Condroid项目

LXC（Linux Container）是 Sourceforge 网站上的开源项目，从内核 2.6.27 版本开始支持[17]。它利用 Linux 内核的 Namespace 隔离机制和 Cgroup 资源控制方案，在用户态下通过创建容器并定制每个容器中的配置文件，可在容器中定制不同的操作系统[18]。

2012 年，浙江大学的陈文智团队为了解决 Android 操作系统虚拟化中面临的内存资源限制、计算处理能力有限、设备功耗限制以及 Android 设备多样性等问题[19]，首先通过将 LXC 工具移植到 ARM 平台，利用 LXC 容器的特性创建出多个容器，并在每个容器中以修改原生 Android 源码的方式运行完整的 Android 系统，其架构如图 1.1 所示[20]。通过修改 Android 系统源码，使得多个系统共用显示子系统和进程通信 Binder 驱动，实现了设备复用。这种方案是一种基于移动平台的操作系统级虚拟化解决方案。同时支持多个隔离的 Android 用户空间实

例，使得多个完全独立的用户角色能够在一个设备上无缝共存[21]。但是该方案需要修改每个虚拟 Android 系统的源代码，不具有良好的通用性与移植性。

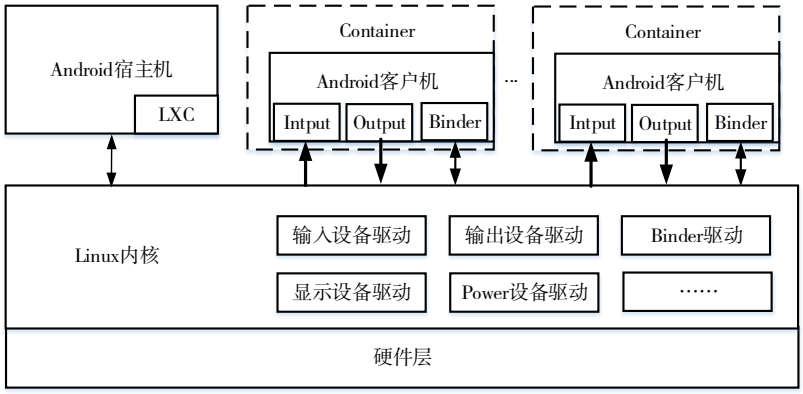

图1.1　Condroid架构图

1.2.3　Cells项目

哥伦比亚大学的软件与操作系统联合实验室于 2013 年提出了一种将操作系统级虚拟化技术运用在 Google Nexus 手机的虚拟化方案，取名为 Cells[22]。它摒弃了传统的 Hypervisor 设计架构，在终端设备中仅运行单独而又完整的 Android 系统实例，通过该团队封装好的工具 CellD 来创建出多个虚拟 Android 实例。在 Cells 中首先构造出一个 root 操作系统实例，通过该 root 系统实例构造出新的虚拟实例[23]。每个实例会在初始化的时候被分配特定的权限、特定的获取 Android 底层设备驱动的权限。构造出的各个实例之间可以自由切换，每个实例中的数据相对独立。

此方案的优势就是系统实际中只运行了一个 Android OS 实例，从而降低了很多系统开销，该方案的设计思想更接近于操作系统级别的虚拟化。但是系统中的 CellD 实例却引入了一定的管理开销。

1.3　本文的研究内容

针对上述方案存在的问题，达到轻量级、通用性的目的，本文将虚拟化技术与 Android 系统的优点相结合，基于 ARM 平台设计并实现了一种轻量级移动操作系统虚拟化解决方案。主要做了以下几个方面的工作：

(1) 针对当前各类虚拟化技术解决方案的不足并提出一种改进方案。

深入研究了全虚拟化技术、半虚拟化技术、操作系统级虚拟化技术各种成熟

的解决方案实现架构。通过调研其研究背景、实现原理、方案架构，分析方案的优点与不足，并结合研究背景与项目需求，提出一种将操作系统级虚拟化技术的优势运用在移动 ARM 平台上的改进方案。

(2) 研究 Linux 命名空间资源隔离机制并引入 Driver 命名空间框架。

Linux 内核提供了一种轻量级命名空间资源隔离机制，使得各类系统资源不再具有全局性而是属于某个特定的命名空间。本文重点研究了实现操作系统虚拟化解决方案所用到的 MNT 命名空间和 PID 命名空间，前者为每个虚拟客户系统提供了一个隔离的文件系统挂载点的视图，后者则提供了一个独立的进程信息视图，研究了这两种命名空间机制用户空间的使用方法和内核中的实现原理。同时通过对内核的修改引入 Driver 命名空间框架，该框架的实现为每个虚拟客户系统提供了一个隔离的硬件设备使用视图，并在该框架中构造 active-inactive 模型，保证了多个虚拟客户系统对一套硬件设备的隔离复用。

(3) 设计基于 ARM 平台的 Android 操作系统虚拟化架构。

Android 系统虽然采用 Linux 内核，但在其基础上扩展了大量的应用软件与硬件设备。通过运用 MNT 和 PID 命名空间机制，在宿主系统中构造出两个虚拟客户系统，以共享同一个内核的方式同时运行在系统中。由于宿主系统与虚拟客户系统同样共享一套硬件设备，通过本文设计的 Driver 命名空间框架来实现对设备的隔离复用。本文在共享单个内存、CPU 资源的前提下，深入研究了 Android 系统中各个设备子系统，设计了对显示设备子系统、输入设备子系统、背光设备子系统以及 LEDs 设备子系统等最核心硬件设备的虚拟化解决方案。

(4) 实现 Android 操作系统虚拟化方案。

本文实现的操作系统级虚拟化方案主要有两部分组成：系统级启动、硬件设备复用。首先修改内核编译选项定制支持命名空间功能的内核。接着在宿主系统中构造出两个虚拟客户系统，保证系统级的正常启动。最后运用 Driver 命名空间框架，结合显示设备、输入设备、背光和 LEDs 设备子系统的自身架构原理和对于虚拟化的要求实现了一个轻量级的 Android 系统虚拟化解决方案。

(5) 实验与分析。

本文实验平台选择 AC8317 Cortex Dual-Core 平台，在功能测试上对宿主系统启动、虚拟客户系统启动、系统间正常切换、系统间隔离性等进行了完整的测试工作；在性能测试上通过对 CPU 利用率以及内存使用度两个角度来衡量虚拟化的引入所带来的性能开销。

1.4　本文的组织结构

本文主要设计并实现了一个轻量级的移动操作系统虚拟化解决方案，并通过实验测试了虚拟化所带来的性能开销。本文总共有六个章节，具体每个章节的组织结构如下：

第一章是绪论。首先介绍虚拟化技术存在的问题，给出将操作系统虚拟化运用在移动平台的研究意义，接着介绍国内外在移动虚拟化领域的研究现状和典型的研究成果，最后概括了本文的主要工作和各个章节的组织结构。

第二章阐述了与本文相关的关键技术。首先介绍了虚拟化技术的分类，并从框架结构、设计原理、代表性方案等方面分析了优劣，确定了本文的研究对象为基于 Android 系统的操作系统级虚拟化。接着介绍了 Android 系统整体架构，阐述了它与 Linux 的异同。最后介绍了操作系统虚拟化技术中最核心的命名空间机制，并从用户空间、内核空间详细分析了 MNT 和 PID 命名空间实现原理。

第三章详细介绍了 Driver 命名空间框架的设计与实现。传统的命名空间资源隔离机制不能满足 Android 中大量硬件设备的隔离复用，本文设计的 Driver 命名空间可以解决该问题。本章首先介绍了 Driver 命名空间的设计内容，然后介绍了该框架的工作原理。包括数据结构设计、与框架相关的 API 接口设计，最后详细阐述了 active-inactive 模型，该模型依托于内核通知链机制和/proc 目录，通过接口的设计来实现多个系统对一套硬件设备的隔离复用。

第四章介绍了本文设计的操作系统级虚拟化架构。首先阐述了设计方案的特点，并介绍了本文的整体方案设计。通过分析 Android 各个硬件设备子系统，阐述了设计显示设备子系统、输入设备子系统、背光和 LEDs 设备子系统虚拟化方案的原因，并结合上述设备子系统的自身特点设计了隔离复用方案。

第五章详细介绍了本文操作系统级虚拟化架构的实现与具体的实验设计方法与测试数据结果分析。首先介绍了内核的定制，接着阐述了系统整体启动流程，包括 init 进程的工作原理与通过修改 init.rc 文件来实现虚拟客户系统的启动方法，然后详细介绍了显示设备、输入设备与背光 LEDs 设备的隔离复用实现原理，最后简要介绍了对该架构的内存优化。在实验部分，通过宿主系统启动、虚拟客户系统启动、系统间正常切换、系统间隔离性等方面进行了完整的功能测试工作；最后通过对 CPU 利用率以及内存使用度两个指标来衡量虚拟化的引入所带来的性能开销。

第六章是总结与展望。本章总结了本文提出的轻量级 Android 操作系统级虚拟化解决方案，并从三个方面提出了对下一步工作的改进和展望。

最后是参考文献、致谢以及在读期间发表的学术论文与科研成果。

第二章　关键技术及分析

本章主要介绍 Android 操作系统虚拟化方案中所涉及的关键技术。首先介绍主流虚拟化技术的分类、框架结构、技术原理与优缺点，概括操作系统级别的虚拟化优势；接着分析 Android 操作系统的分层架构，并阐述和 Linux 系统的异同；最后介绍 Linux 内核现有的轻量级命名空间资源隔离机制，并详细分析与操作系统虚拟化方案密切相关的 MNT 命名空间与 PID 命名空间的工作原理。

2.1　虚拟化技术

计算机在设计的初期，软硬件之间采用紧密耦合的设计思想，底层的硬件设备提供了完整且具有针对性的接口供用户空间内的应用软件调用[24]。随着当前计算机的不断发展，软件与硬件从种类上到数量上都具有着日新月异的发展，因此操作系统也越来越难应对软硬件的多样化所带来性能、功耗上的改变。

虚拟化技术，实质上是一种资源管理技术，它通过在计算机系统中加入了虚拟化层，将计算机的所有实体资源（如 I/O 设备、存储设备、网络资源、内存等）进行管理并予以转化后向需要使用资源的进程提供统一的应用接口来达到对计算机系统的抽象[25]。虚拟化技术的引入，打破了计算机体系传统的软硬件紧密耦合的设计理念，打破了各种实体结构间以往不可分割的障碍。对于操作系统而言，不同系统之间的硬件平台差异被有效的屏蔽。

虚拟化技术经历了十几年的发展已形成了多种研究分支，从功能上来说，可以分成硬件虚拟化技术、网络虚拟化技术、数据库虚拟化技术、内存虚拟化技术、服务虚拟化技术、存储虚拟化技术和桌面虚拟化等[26]。按照虚拟化技术的实现层面上，可以分为硬件分区虚拟化、完全虚拟化、半虚拟化、操作系统虚拟化四大类[27]。

2.1.1　硬件分区虚拟化技术

硬件分区虚拟化技术(硬件分片技术)将硬件资源分割成若干自定义的部分，并允许在各部分中自主的独立安装特定的操作系统。各个部分相互独立，享有独立的硬件资源[28]。这种方法的优点是可以同时运行多个操作系统，缺点则是各个系统对于硬件的管理有限，灵活性不好，也不具备对系统资源调度的能力。由 IBM

公司研制的 AIX 系统硬件资源（主要指 CPU）分割成最多 10 个部分，该产品已经成功运用在银行的数据中心中，是主流虚拟化方案的雏形[29]。

2.1.2　完全虚拟化技术

完全虚拟化技术（Full virtualization）也称为原始虚拟化技术[30]。该方法不需要对硬件资源进行分片划分，而是引入 Hypervisor 虚拟监管层，从而建立了在虚拟实例和底层硬件中的抽象层。

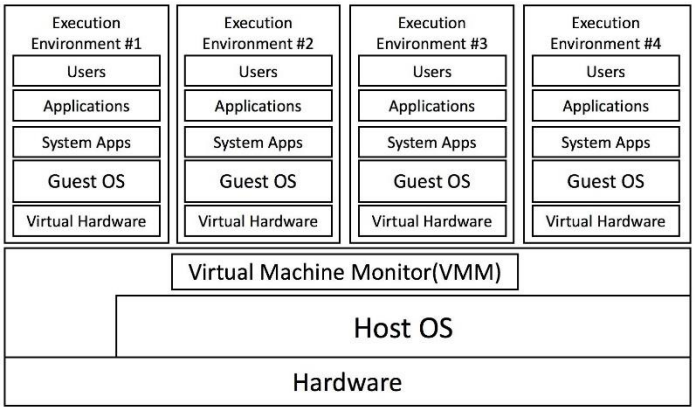

图2.1　完全虚拟化技术架构图

完全虚拟化技术架构如图 2.1 所示，使用二进制翻译和直接指令执行相结合，在 Hypervisor 虚拟监管层中加装了 VMM（Virtual Machine Monitor）层作为用户空间下的软件存在[31]，VMM 层负责翻译所有 CPU 指令，使用一系列作用于虚拟化硬件可达到所需效果的新指令序列替换那些不可虚拟化的指令，是连接硬件设备和虚拟机之间的纽带，使得虚拟机系统和底层的物理硬件彻底解除耦合。每个虚拟机需要利用 VMM 层来实现整套硬件设备的虚拟化工作。由于虚拟机中的系统没有意识到它是被虚拟化的，因此不需要虚拟系统做任何的修改[32]。该方案的核心依托于 Hypervisor 层的指令翻译，也因此属于指令级别的虚拟化技术。

完全虚拟化技术具有如下优点：(1) Hypervisor 将操作系统的指令翻译后供虚拟系统使用，用户级指令无需修改就可以直接在物理资源上运行，具有和物理机一样的执行速度，从而保证了虚拟化的高效；(2) 在宿主系统上允许构建若干个不同的虚拟操作系统；(3) 提供了最佳的隔离性，资源的分配、调度均由 Hypervisor 层进行管理。

但该方案的缺点也很明显：由于 Hypervisor 监管层直接运行在硬件上，管理虚拟运行的操作系统给处理器带来了很大的开销；同时，虚拟硬件等设备由于大量代码需要被翻译执行也要消耗资源，造成了性能的损耗。当前 VMware 和 Microsoft 公司都研制成功各自的商用全虚拟化产品[33]，如 VMware 系列的

VMware workstation 和微软的 Virtual PC、Virtual Server 系列。基于内核的虚拟机有 KVM，是面向 Linux 系统的开源产品；除此之外，还有 IBM 的 Z/VM。

2.1.3　半虚拟化技术

半虚拟化技术是一种硬件级别的虚拟化技术。由于完全虚拟化技术需要虚拟完整的硬件设备资源，未经修改的虚拟系统并不知道自己已经被虚拟化，半虚拟化的提出则实现了一种更低代价的虚拟化方案[34]（如图 2.2）。对于宿主操作系统，需要修改内核来将不可虚拟化的指令替换为直接与虚拟化层交互的超级调用；而对于虚拟操作系统，需要针对不同的系统做出相应的修改，简化了 Hypervisor 层的管理工作，极大改善了性能和效率。Hypervisor 监管层只是为其他关键的系统操作如内存管理、中断处理、计时等提供了超级调用接口[35]。

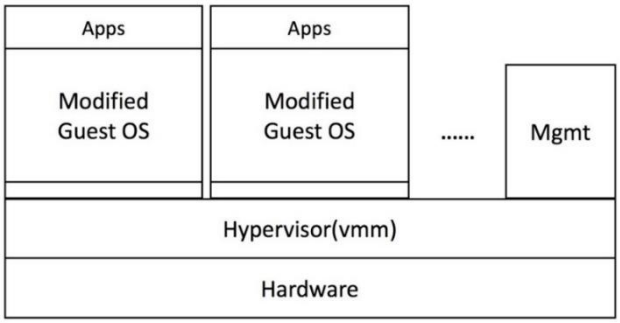

图2.2　半虚拟化技术架构图

Xen 作为一款开源的商用软件[36]，是半虚拟化技术的代表。通过修改操作系统内核，允许在一套硬件平台上支持多个系统的运行，同时引入的 Xen Hypervisor 监管层实现系统资源调度。它使用一个经过修改的 Linux 内核来虚拟化处理器，而用另外一个定制的虚拟机系统设备驱动来虚拟化 I/O。使用二进制翻译来实现虚拟化更复杂更困难，因此 Xen 中的虚拟客户系统也需要做相应的修改。

和完全虚拟化技术相比，由于半虚拟化技术在实现上需要对宿主系统和客户系统都要修改，分担了一部分 Hypervisor 层的压力，因此具有更好的性能。然而半虚拟化不支持未经修改的操作系统(如微软 Windows 系列，Windows 2003/XP 等)，因此它的兼容性和可移植性较差。由于半虚拟化需要系统内核的深度修改，在生产环境中，半虚拟化在技术支持和维护上会有很大的问题。Xen 目前仅可以实现将开源的操作系统作为虚拟客户系统，却无法实现对 MacOS 等闭源操作系统的虚拟化工作，具有一定的局限性。

2.1.4　操作系统级虚拟化

随着虚拟化领域相关研究的不断深入，一种轻量级的虚拟化解决方案被提出。与传统的虚拟化技术在实现上需要引入 Hypervisor 监管层不同，操作系统级别的虚拟化技术以共享内核的方式，利用宿主系统为应用程序提供多个完整、隔离的运行环境，每个独立的运行环境和宿主系统具有相同的功能与特性，相当于内核的一个新功能，从而实现了虚拟化。

高性能服务器领域与移动终端领域的飞速发展对于传统的虚拟化技术提出了新的挑战，而操作系统级别的虚拟化技术由于自身的显著优势近年来逐渐被各大科研机构和企业关注并涌现出一些较为成功的产品。操作系统虚拟化虚拟化的代表就是 SWsoft 的 Virtuozzo/OpenVZ 产品，该方案在每个 Virtuozzo 服务器中引入操作系统虚拟化技术，在一个宿主系统实例上构造出最多 10 个独立环境，具有良好的应用性。另一个典型代表则是 Sun 公司的 Container 技术[37]，通过 Linux 内核提供的 Namespace 与 Cgroup 特性，不虚拟任何硬件设备实现了在唯一系统的内核上运行多个独立的虚拟操作系统。多个虚拟系统共享一个内核、一个文件系统，使得性能得到了大幅度提高。

操作系统级别的虚拟化技术没有 Hypervisor 层，减少了指令翻译工作和虚拟层的管理开销。宿主系统为每个虚拟客户系统分配硬件访问权限并提供完整、独立、隔离的运行环境。这种虚拟化技术实现方案效率极高，虚拟化的引入并没有带来明显的开销。

2.2　Android操作系统架构

Android 是一个针对于智能设备的移动操作系统，拥有着大量的应用软件开发社区与丰富的硬件设备，具有显著的优势[38]。目前，Android 已经成为全球最大的智能手机操作系统。本文对 Android 操作系统进行了系统级别的虚拟化，因此首先介绍一下 Android 操作系统的架构。

2.2.1　Android系统架构

Android 系统架构如图 2.3 所示，应用层中运行着 Android 系统的各类应用程序；框架层中包含了所有核心服务，为应用层提供 API；核心层为框架层提供了库支持；而 Linux 内核层则是 Android 系统的基础。下面具体介绍如下：

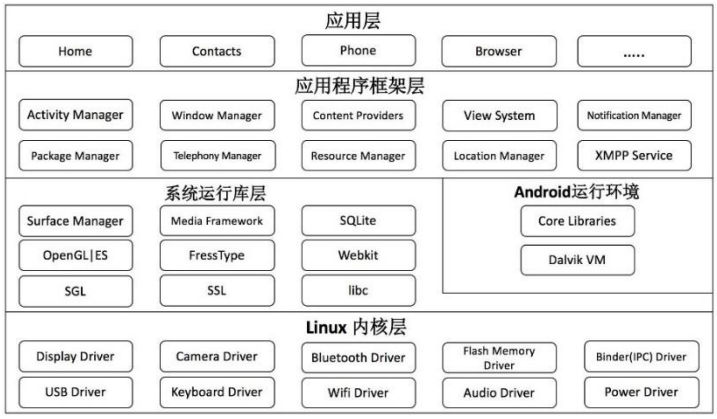

图2.3　Android系统架构示意图

(1) 应用层

和 Windows 操作系统类似，安装 Android 系统会同时安装一些应用软件。比如联系人软件、拨打电话软件、短信收发软件、google 地图软件、浏览器软件、相册图库软件等。所有应用软件均由 Java 语言开发，通过应用程序框架层提供的 API 完成所有所需的功能，方便灵活。

(2) 应用程序框架层

应用程序框架层为应用层提供 API，囊括了完整的各种功能 API 实现接口。由于每个页面在 Android 中被定义成一个活动(Activity)，该层包含了诸如活动管理器、包管理器在内的各类系统核心服务。

(3) 系统运行库

Google 公司基于 JVM 编写了专门为 Android 应用程序运行的 Dalvik 虚拟机。在 Android 系统中，每个应用程序都维系着一个 Dalvik 虚拟机实例，不同应用程序之间的虚拟机实例互不干扰。Dalvik 虚拟机内部进行了内存资源使用的优化，通过 SDK 中 的 "dx"工具转换成 Dalvik 的.dex 可执行文件。

系统运行库层为框架层提供必要的库支持，并为 Linux 内核层提供接口，从而连接应用程序框架层与 Linux 内核层的。系统运行库核心部分主要由负责管理显示的 SurfaceManager、支持 AAC 等多种格式录制回放的 Media Framework、轻量级关系型数据库 SQLite、web 浏览器 LibWebCore、3D 绘图函数库 OpenGL|ES、标准 C 系统函数库 Libc 等组成。

(4) Linux 内核层

Android 系统在设计初期基于 Linux 内核版本 2.6，将 Linux 内核的强大优点作为 Android 系统的核心与基础，用来连接底层硬件设备与 Android 用户空间。

2.2.2　Linux与Android的区别与联系

Android 系统与 Linux 系统具有着千丝万缕的联系。从联系上看，这两种系统都具有开源的特点，都具有着良好的特性，从而在各自领域中占据了一席之地。而从区别上看，Linux 系统主要应用在桌面与服务器领域且硬件设备较少的环境中。与 Linux 系统相比，Android 系统具有大量的应用软件，极大的丰富了移动设备市场[39]。针对智能设备对硬件设备的需求，Android 基于 Linux 系统的各类优点，并在此基础上增加了包括报警器、低内存管理、日志驱动在内的专有硬件设备驱动程序。因此对硬件设备的虚拟化是本文 Android 系统虚拟化架构的设计重点内容。

2.3　命名空间机制

从内核版本 2.4.19 开始，Linux 内核提供了轻量级的命名空间(Namespace)资源隔离方案。通过对全局的系统资源进行封装隔离，在特定类型的命名空间中可以看到诸如 UTS hostname、IPC 进程通信、PID 进程信息、NET 网络设备等系统资源。不同命名空间抽象的系统资源都是透明且不可见的[40]。改变一个命名空间中的系统资源只会影响当前命名空间里的进程，对其他命名空间中的进程没有影响。

如表 2.1 所示，目前 Linux 内核中实现了六种不同类型的命名空间，每一个命名空间是一类全局系统资源的抽象集合，这种抽象使得在进程的各类命名空间中可以看到隔离的全局系统资源。内核规定只需在 clone()系统调用时指定相应的 flag 即可创建新的命名空间[41]。

表2.1　内核六种命名空间机制

命名空间名称	宏定义 flag	隔离内容
IPC	CLONE_NEWIPC	隔离 IPC 进程通信机制，消息队列机制
Network	CLONE_NEWNET	隔离网络设备，包括端口、网络协议栈、防火墙
MNT	CLONE_NEWNS	隔离文件系统挂载点信息
PID	CLONE_NEWPID	隔离进程 ID 号
User	CLONE_NEWUSER	隔离 User ID 以及 Group ID 号
UTS	CLONE_NEWUTS	隔离 Hostname 以及 NIS domain name

IPC 命名空间用来隔离进程通信机制，其中包含消息队列、信号量集合和共享内存段，使得不同命名空间之间的进程不能直接通信；Network 命名空间用来

隔离网络设备、IP 地址、端口信息等，为进程提供了一个完全独立的网络协议栈的视图；USER 命名空间用来隔离用户 ID 与组 ID；UTS 命名空间则用来隔离系统的 hostname 以及 NIS 域名等。本文在实现操作系统级虚拟化技术的过程中主要用到 MNT 命名空间与 PID 命名空间，具体介绍如下。

2.3.1　MNT命名空间

MNT 命名空间隔离了文件系统的挂载点，从 Linux2.4.19 内核率先支持该命名空间特性，内部进程组能够看到与其他 MNT 命名空间隔离的一套文件系统挂载点信息[42]。不同 MNT 命名空间内的进程看到的文件系统是相互透明的，且 mount 和 unmount 操作不会相互影响。MNT 命名空间层级关系如图 2.4 所示，在系统根文件系统的 container 文件夹下存在两个独立的 MNT 命名空间视图，对于 container1 和 container2 来说，内部拥有一套和根文件系统相同的挂载点信息，相互独立且隔离。

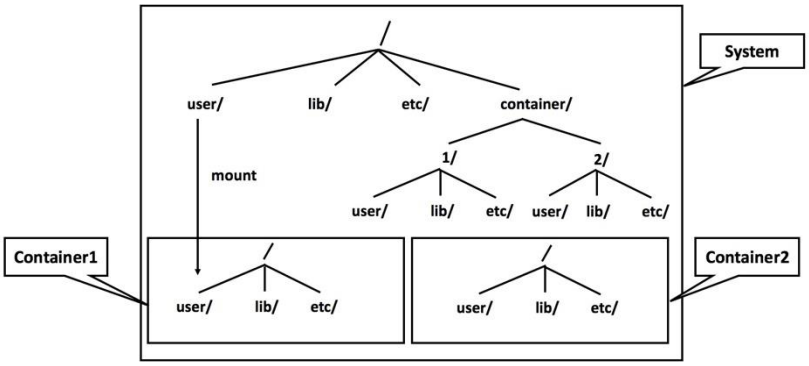

图2.4　MNT命名空间层级示意图

当父进程通过系统调用 clone，指定传入的参数，子进程就会处在一个全新的 MNT 命名空间中。这个新创建的 MNT 命名空间拥有一个全新的文件系统挂载点视图，与父进程所在的 MNT 命名空间独立且隔离。否则，子进程将和父进程共用 MNT 命名空间。在此基础上利用 pivot_root 命令，将参数 root_dir 个性化指定，从而可以构造出隔离的文件挂载点。该命名空间在内核中的实现主要有如下三个部分组成：

(1) init 进程创建子进程

内核中 fs_struct 结构体表示一个进程所属的全部文件系统挂载点信息，它是进程信息 task_struct 结构体中的成员之一，当 init 进程在内核启动时会默认指定好自己所在的根路径。

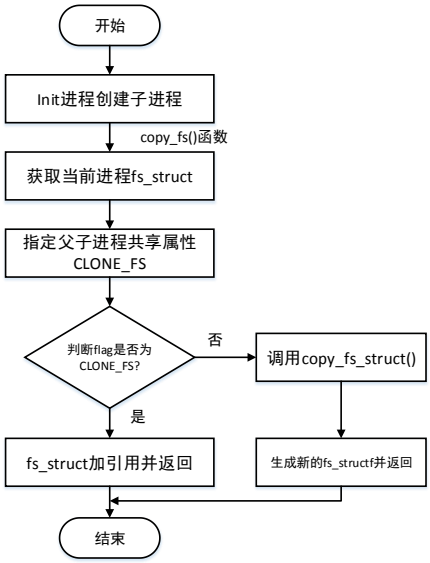

图2.5　init进程创建子MNT命名空间流程图

如图 2.5 所示，当 init 进程 fork 子进程时 copy_fs()函数首先会通过宏 CLONE_FS 初步判断子进程是否继承父进程的全部文件系统挂载点信息。如果子进程指定的 flag 不是 CLONE_FS，则通过 copy_fs_struct()返回给子进程一个新的 fs_struct 结构体；否则将当前 fs_struct 结构体引用值加 1 后直接返回[43]。

(2) 指定目录的创建 MNT 命名空间。将一个目录置于一个新的命名空间中而使得内部进程把该目录当成根目录。

```
struct mnt_namesapce {
        atomic_t count;
        struct vfsmount *root;  /* VFS文件系统根目录 */
        struct list_head list;
};
```

图2.6　MNT命名空间内核结构体

表示 MNT 命名空间的结构体如图 2.6 所示，mnt_namespace 结构表示了一个 MNT 命名空间，成员 root 表示该 MNT 命名空间的 root 目录，成员 list 则将系统中所有 MNT 命名空间串联在链表中。create_mnt_ns()函数[44]用来实现将指定目录作为新创建的 MNT 命名空间的 root 目录。首先，alloc_mnt_ns()分配一个 mnt_namespace 结构并完成最核心的 root 成员的初始化工作，接着将一个 vfsmount 类型的参数传给新创建的 root 成员，从而实现将该文件挂载点信息设置为新 MNT 命名空间的根目录。

(3) 不制定目录的创建 MNT 命名空间。

图 2.7 展示了 copy_mnt_ns()函数用来完成一个进程复制自身不指定目录的创建新的 MNT 命名空间的流程。

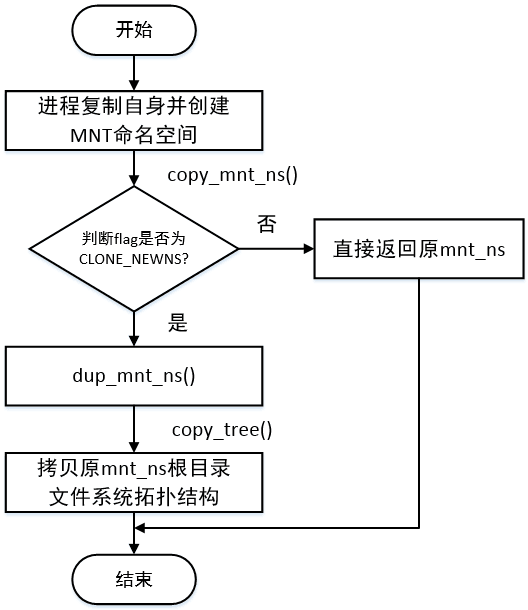

图2.7　不指定目录创建子MNT命名空间流程图

copy_mnt_ns()函数函数首先判断 clone 系统调用的参数是否为 CLONE_NEWNS，如果匹配成功则利用 dup_mnt_ns()函数来完成创建新的 MNT 命名空间工作。此函数利用 copy_tree()函数将父进程 mnt_namespace 结构中的 root 成员进行遍历，将 root 成员的文件系统全部 copy 给新创建 MNT 命名空间中的 root 成员，最后再把子进程 fs_struct 结构中的所有成员标记到新的 MNT 命名空间中。

2.3.2　PID命名空间

PID 命名空间是当前实现原理最复杂的命名空间，用来隔离进程 ID 空间。同一个进程在不同的 PID 命名空间中可拥有多个进程 ID 号，不同 PID 命名空间里的进程 ID 号可以重复[45]。与 MNT 命名空间的创建方法类似，将 clone 传入的参数改成 CLONE_NEWPID 即可创建出一个 PID 命名空间，为子进程提供了一个隔离独立的 PID 环境。如果在这个新创建的命名空间中通过 ps 查询进程列表，会发现进程号也从 1 开始，类似于根系统中的 init 进程一样，所有在该进程号 pid＝1 之后被创建的子进程都以该进程为父进程。该父进程对该命名空间下的子进程进行管理，只有当该进程被 kill 掉后，所在的命名空间内的所有进程都会被结束。

PID 命名空间在内核中以一种层级关系存在，具体由 parent 指针构建。父 PID 命名空间可以管理并看到子 PID 命名空间中的进程树，但子 PID 命名空间却看不到祖先或者兄弟命名空间里的进程信息。Linux 启动时会创建一个进程号等于 1 的 init 进程，该进程所在的 PID 命名空间是所有子 PID 命名空间的祖先，因此系统所有的进程在该命名空间都是可见的。当前，Linux 下的每个进程都有一个对应的/proc/PID 目录，该目录包含了所有有关当前进程的信息。

(1) PID 命名空间的内核实现

对于一个进程而言，由于它可以映射在多个 PID 命名空间中，在不同的命名空间下其进程 ID 号不同，因此传统的 task_struct 结构和 pid 结构已不再一一对应了。同一个 pid 结构允许关联到多个 PID 命名空间结构中，由于 PID 命名空间的引入，除了获取到 pid 结构以外，还需要标记该 pid 结构关联的 PID 命名空间。一个 PID 命名空间的数据结构如图 2.8 所示：

```
struct pid_namespace {
        ……
        struct task_struct *child_reaper;
        int level;
        struct pid_namespace *parent;
        ……
};
```

图2.8　PID命名空间内核结构体

其中 child_reaper 字段指向了本命名空间下的祖先进程，和全局的 init 进程功能一样；level 表示该命名空间所处的深度，内核目前支持最多嵌套 32 层，由内核中的宏 MAX_PID_NS_LEVEL 来定义；parent 成员则指向了本命名空间的父命名空间，因此通过该成员可以构建层次结构。

pid 结构如图 2.9 所示，count 表示引用计数，level 代表了对应的命名空间的层次数目，成员 tasks[]数组中的每个元素是是一个哈希链表表头类型，连接着所有使用此 pid 结构的进程，numbers 是一个 upid 结构的数组，长度至少为 1。

```
struct pid {
        atomic_t count;
        int level;
        struct hlist_head tasks[PIDTYPE_MAX];
        struct upid numbers[1];
};
```

图2.9　pid结构体示意图

upid 就是对应在一个命名空间中进程 ID 的信息，如图 2.10 所示。nr 是在某个 PID 命名空间下使用 ps 查询到的进程 ID 值；ns 成员指向关联的 PID 命名空间；pid_chain 则将所有的 upid 保存在一个散列表中供内核快速查找。

```
struct upid {
        int nr;
        struct pid_namesapce *ns;
        struct hlist_node pid_chain;
};
```

图2.10　upid结构体示意图

与此同时在 task_struct 中的 pids 字段为 struct pid_link pids[PIDTYPE_MAX]。该数组的每个元素标记一个 pid 结构，用来将 task_struct 结构和 pid 结构关联在一起。同时 pid 结构中又存在 numbers 成员，使得 PID 命名空间与 pid 结构同样连接起来。PID 命名空间中相关的结构体关系如图 2.11 所示：

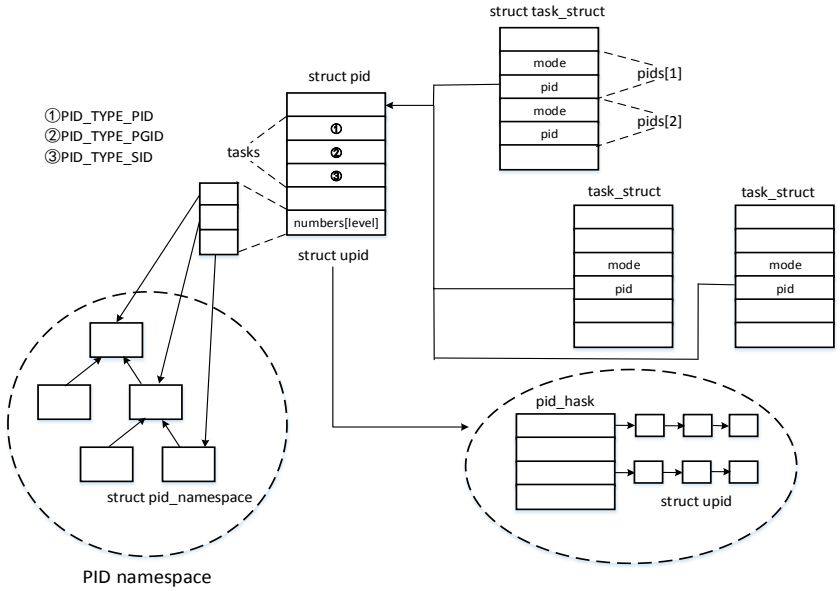

图2.11　PID命名空间各类结构体关系图

PID 命名空间的层次结构由 parent 维系。pid 结构中的 numbers 字段保存系统中所有 upid 成员供内核查询。同一个 pid 结构可被多个 task_struct 关联，说明 pid 结构可由不同的进程使用。pid 结构作为进程与 PID 命名空间的桥梁，首先通过 upid 类型的 numbers 成员关联到 pid_namespace 结构体，从而关联到 PID 命名空间；同时通过 tasks 成员可以索引到关联该 pid 结构的所有进程的 task_struct 结构。task_struct 结构、pid 结构、pid_namespace 结构就是这样关联在一起。

和 MNT 命名空间类似，调用 copy_pid_ns()完成复制或者创建 PID 命名空间的工作。该函数首先对 flag 进行检查，判断是否要创建一个新的 PID 命名空间。

如果不需要创建，则将父 PID 命名空间结构加引用并返回；否则通过调用 create_pid_namespace()函数创建一个新的 PID 命名空间。

(2) PID 命名空间用户空间的使用

进程的/proc/PID 文件中包含了该进程的所有信息[46]。对根 PID 命名而言，/proc 目录包含所有子命名空间里的进程信息。通过 readlink /proc/PID/ns/pid 命令可以查询到当前 PID 命名空间号，通过指定 CLONE_NEWPID 来创建出新的 PID 命名空间，父命名空间可以包含它的所有子命名空间里的进程的信息，用户空间可以通过两种方式来查询每个进程的 PID 命名空间信息：

方式 1:在根命名空间的/proc/PID/ns/目录下包含着所有进程的 PID 命名空间信息，通过 pstree –pl 命令可以查询并锁定到该父命名空间下特定的进程信息来查询。方式 2:由于 ps 命令依赖于系统的/proc 目录，通过在子命名空间中执行命 mount -t proc proc /proc 来挂载隔离的/proc 目录，这样就可以在每个子 PID 命名空间里隔离的查询到该命名空间下的进程信息。

同一个进程在各个命名空间里的映射关系可以通过/proc/PID/status 查询，该文件中的 NSpid 变量会返回若干字段，比如 NSpid：32212 232 35 1。该字段说明 init 进程存在于 4 个 PID 命名空间中，在各个命名空间中对应的进程号分别为 32212、232、35 和 1。

除了在 init 进程里指定 handler 的信号外，内核会帮 init 进程屏蔽掉其他信号，这样可防止其他进程不小心 kill 掉 init 进程导致系统挂掉[47]。不过有了 PID 命名空间后，可以通过在父命名空间中发送 SIGKILL 或者 SIGSTOP 信号来终止子命名空间中的 ID 为 1 的进程，从而实现了父命名空间对子命名空间的管理。

2.4　本章小结

本章首先从技术背景、基本框架以及实现原理等角度分析了硬件分区虚拟化、完全虚拟化与半虚拟化技术。虽然运用传统的 Hypervisor 架构具有良好的安全性，但由于引入的 VMM 层需要进行大量的指令翻译和管理开销，因此操作系统级虚拟化技术具有共享内核、轻量级的显著优势，是当前虚拟化领域研究的热点。接着详细介绍了 Android 系统架构，虽然采用 Linux 作为内核，但该系统具有丰富的应用软件与硬件设备，这也是 Android 系统虚拟化的难点。最后介绍了内核轻量级命名空间资源隔离机制，并从用户态与内核角度着重研究了 MNT 与 PID 命名空间，这两种命名空间分别为本文的虚拟化方案提供隔离的文件系统视图和独立完整的进程信息视图。

第三章　Driver命名空间框架设计

在移动平台上，由于处理器对于虚拟化的支持没有 PC 端那样完善，且移动平台中 Android 系统具有大量各式各样的硬件设备，如传感器、照相设备、显示设备、触摸屏等输入设备、LED、电话等。因此，解决移动平台中繁多硬件设备的隔离复用成为 Android 操作系统虚拟化研究的一大难点。

本章阐述 Driver 命名空间框架的设计内容与工作原理。通过对内核的修改引入该框架，将宿主系统构造出的每个虚拟客户系统关联到一个 Driver 命名空间。和现有命名空间(如 PID 命名空间)的设计目标一样，为每个虚拟客户系统提供了一个隔离的硬件设备访问环境。首先介绍 Driver 命名空间框架的设计内容，然后介绍该框架的工作原理，包括数据结构的设计、API 的设计以及核心的 active-inactive 模型。

3.1　设计内容

由第二章命名空间介绍可知，将 MNT 命名空间与 PID 命名空间结合可以构造出隔离的文件系统与进程空间，初步构造出虚拟操作系统运行环境。但是基本的操作系统虚拟化是不足以运行一个完整的移动设备用户环境。传统的操作系统级虚拟化方案主要被用在服务器环境和设备相对较少的使用场景上，移动设备中的应用程序经常直接与硬件设备进行交互，而这些硬件设备在设计之初是不支持复用的。此外，移动设备上的硬件设备是紧密集成的，且通常被硬件厂商封装好对外的控制接口，因此目前尚无较为完善的操作系统级虚拟化方案实现对设备的复用。

Driver 命名空间框架的提出，实现了在宿主系统构造出多个虚拟客户系统的基础上，允许多个虚拟系统同时复用一套硬件设备。该框架实现的本质是对 Linux 内核中现有命名空间机制的一种扩展，对硬件设备的访问进行了隔离，每个虚拟客户系统关联一个 Driver 命名空间，不同 Driver 命名空间下的进程能够隔离地访问硬件设备。同时提出 active-inactive 模型，该模型使得多个虚拟客户系统中有且仅有一个保持 active 状态，只有 active 状态的 Driver 命名空间中的进程对硬件设备访问是有效的，所有虚拟客户系统由宿主系统统一管理和切换。每个硬件设备会在内核初始化时在该框架中注册用于状态切换时的回调函数，当发生状态切换时通过内核通知链机制告知已注册的各个硬件设备，触发先前注册的回调函

数。该模型需要硬件设备时刻注意当前虚拟客户系统的状态以及改变状态时如何响应切换，是 Driver 命名空间框架的核心部分。

通过在内核中引入 Driver 命名空间框架以及 active-inactive 模型的实现，不同虚拟客户系统中的应用程序对于硬件的访问是完全透明的，保障了多个虚拟客户系统隔离且高效的复用硬件资源，实现了完整的操作系统级虚拟化方案。

Driver 命名空间框架的实现包含三个部分：

(1) 设计与 Driver 命名空间相关的数据结构以及 API。数据结构的设计包含两种类型：一种是实现在 Driver 命名空间框架中的数据结构，使得硬件设备可以在 Driver 命名空间中被标记；另一种则是实现在具体的硬件设备子系统中，实现对硬件设备的操作。API 的设计使得硬件设备能够感知 Driver 命名空间框架，包括定义在 Driver 命名空间框架中的接口函数以及定义在具体硬件设备中的回调函数。前者用于当硬件设备访问时被调用，后者则用于当回调发生后硬件设备创建相应的数据结构。

(2) 在 active-inactive 模型的实现中，设计 Driver 命名空间切换处理的 API。此模型的实现需要依赖于 Linux 内核通知链机制，是 Driver 命名空间框架的核心部分，与该模型相关的 API 设计包括状态切换 API、响应 API 以及封装内核通知链的 API。

(3) 为了实现基于 Driver 命名空间的操作系统级虚拟化，还需要修改具体的硬件设备子系统的源代码。本文完成了 Android 系统中最核心的硬件设备子系统虚拟化工作，包括输入子系统、显示子系统、电量背光灯子系统等。该部分将在第五章系统的实现部分详细介绍。

3.2　工作原理

3.2.1 数据结构设计

内核中在表示进程全部信息的 task_struct 结构体中存在成员 nsproxy 结构，该结构指向进程所属的各类命名空间。该结构体相当于进程的命名空间代理，是连接一个进程和它所指向各类命名空间的纽带。由于每个命名空间的成员都以结构体指针的形式存在，因此进程和命名空间之间存在着多对多的关系，即多个进程可以共享同一个 nsproxy 结构体实例[48]，一个 nsproxy 结构也可指向多个命名空间。task_struct 结构和 nsproxy 结构关系如图 3.1 所示：

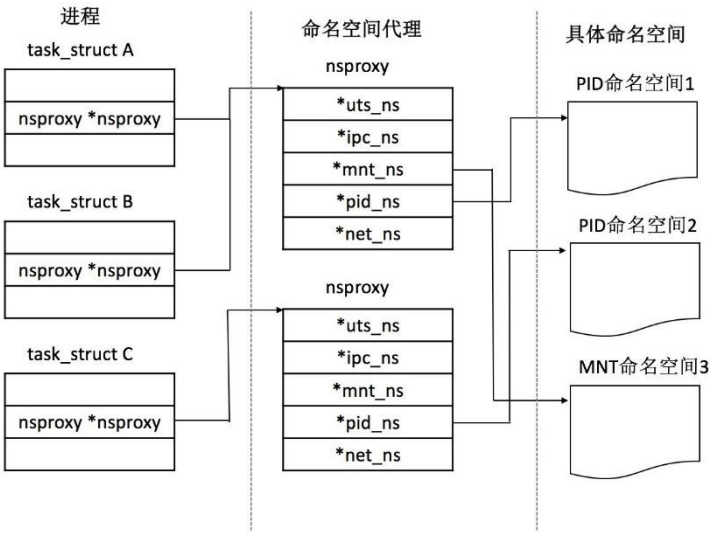

图3.1　task_struct、nsproxy关系图

在 Linux 内核中，用一个*_namespace 结构体标示一种类型的命名空间，本文对 Driver 命名空间数据结构的定义如图 3.2 所示：

```
struct driver_namespace {
        bool active;
        struct pid_namespace *pid_ns;
        struct driver_info *driver_info[DRIVER_MAX];
        struct blocking_notifier_head notifiers;
};
```

图3.2　Driver命名空间数据结构设计图

在设计该数据结构时，依赖内核现有的 PID 命名空间机制，每个 Driver 命名空间结构与 PID 命名空间结构具有一对一的关系。通过 init 进程获取到所属的 PID 命名空间后，将该信息共享给该进程所属的 Driver 命名空间；成员 active 代表着它是否是激活状态下的 Driver 命名空间，为了让 active-inactive 模型识别状态信息；成员 driver_info 代表在某个 Driver 命名空间中注册的某个硬件设备（在 Android 系统中也可称为子系统），一个 driver_info 代表一种硬件设备，如输入设备、传感器设备等，它在 Driver 命名空间中的某个进程使用设备时被创建；成员 notifiers 是一个 notifier_block 类型的链表，它把 driver_info 结构里的成员 notifier 串起来。当 Driver 命名空间的 active 状态发生切换时，通过内核通知链机制通知每个 Driver 命名空间中注册的硬件设备调用相应的切换回调函数。

引入全局变量 active_driver_ns，该变量标记了当前激活状态的 Driver 命名空间。初始默认值为 init 进程所在的 Driver 命名空间。在定义了最核心的 driver_namespace 结构体后，通过在 nsproxy 结构中添加该成员的指针*driver_ns 使得进程可以获取到自己所属的 Driver 命名空间。

在 driver_namespace 结构中保存着一个 driver_info 数组，该数组中的每个元素代表着 Driver 命名空间中注册的某个硬件设备子系统，如图 3.3 所示：

```
struct driver_info {
        struct driver_namespace *driver_ns;
        struct list_head list;
        struct notifier_block notifier;
};
```

图3.3　driver_info数据结构设计图

其中 driver_ns 成员指向了所属的 Driver 命名空间；成员 list 元素用于把不同 Driver 命名空间中的同一种硬件设备设备串在一起，最终标记全局的结构体数组 driver_global 中；notifier 成员是一个 notifier_block 结构，被 driver_namespace 中的 notifiers 成员串连起来。

最后，设计了一个内核中的全局结构体数组 driver_global，每个元素表示一个注册在 Driver 命名空间的硬件设备(如图 3.4 所示)，通过定义宏 DRIVER_MAX 最多可支持 32 种硬件设备。成员 name 表示了硬件设备名称，成员 ops 结构表示 Driver 命名空间具体操作硬件设备的接口，它提供了 create()与 release()两种接口，具体的实现则在硬件设备驱动中定义。每个硬件设备在初始化时会在 driver_global 数组里面注册一项。

```
struct driver_global {
        char *name;
        struct list_head head;
        struct driver_ops *ops;
};
```

图3.4　driver_global数据结构设计图

上述三个 Driver 命名空间的核心数据结构关系如图 3.5 所示：

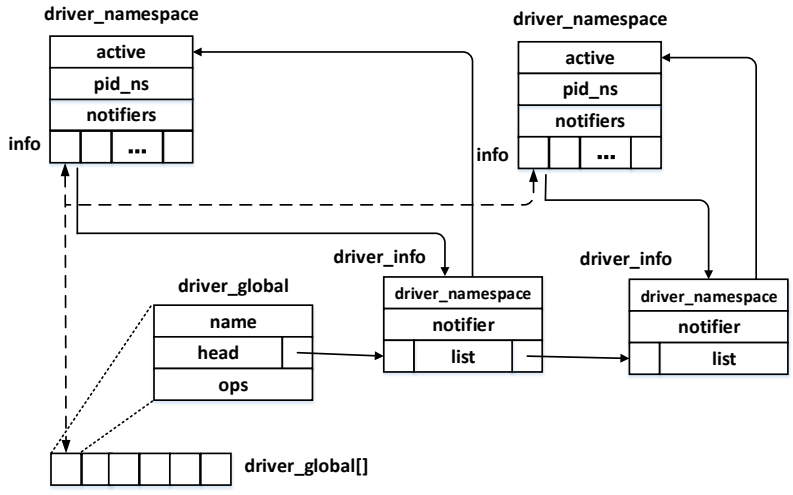

图3.5　Driver命名空间数据结构关系图

至于硬件设备在 driver_global 数组和 driver_namespace 的 info 数组中对应的索引值如何分配问题，通过定义全局变量*_id（如输入子系统为 evdev_id）标记了该设备元素的索引值，它是在硬件初始化时*_init()函数中赋值的。

3.2.2 硬件设备注册

数据结构的设计使得全局变量 driver_global 与每个硬件设备关联在一起，为了让每个硬件设备子系统注册在 Driver 命名空间框架中，需要 A_driver_ns(下文中用 A 代指某个具体的硬件设备)结构体来实现，该结构如图 3.6 描述。

```
struct A_driver_ns {
        struct A_clients *clients;
        struct driver_info *driver_info;
};
```

图3.6　A_driver_ns结构体设计图

该结构是 Driver 命名空间框架与底层硬件设备连接的纽带。该结构中有两个成员，成员 driver_info 将该结构连接到 Driver 命名空间框架中去；成员 A_client 是在每次进程打开 A 硬件设备时创建的结构，不同 Driver 命名空间下的所有 A_client 会被串在一起。

Driver 命名框架通过宏定义 DEFINE_DRIVER_INFO(A)定义了一系列供硬件设备使用的 API。每个注册在 Driver 命名空间框架中的硬件设备中定义该宏后，如显示设备子系统则需要定义 DEFINE_DRIVER_INFO(fb)，编译时便会为该硬件设备子系统生成相应的 API 接口。这些接口主要分为三种类型：

类型(1): 通过正在访问该硬件设备子系统的进程结构信息，找到其所属的 Driver 命名空间。

类型(2): 通过 Driver 命名空间信息与 A_id 序号，找到特定的 driver_info 结构，即找到 Driver 命名空间中的某个硬件设备。

类型(3): 判断当前 Driver 命名空间是否处于 active 状态的处理函数。

当 A 硬件设备在系统中通过 A_init 函数初始化的时候，本质上会调用 register_driver_ops 接口在全局数组 driver_global 中注册 A 设备。这个函数在初始化时会进行如下步骤：遍历搜索出第一个空位置，将该硬件设备名称赋给该位置索引值的 name 成员；接着把与硬件设备相关的结构 driver_ops 注册到 driver_global 中去。这个 driver_ops 接口的模版定义在 Driver 命名空间框架中，实现则在具体的 A 硬件设备中，用于日后让 Driver 命名空间的框架回调 A 硬件设备。

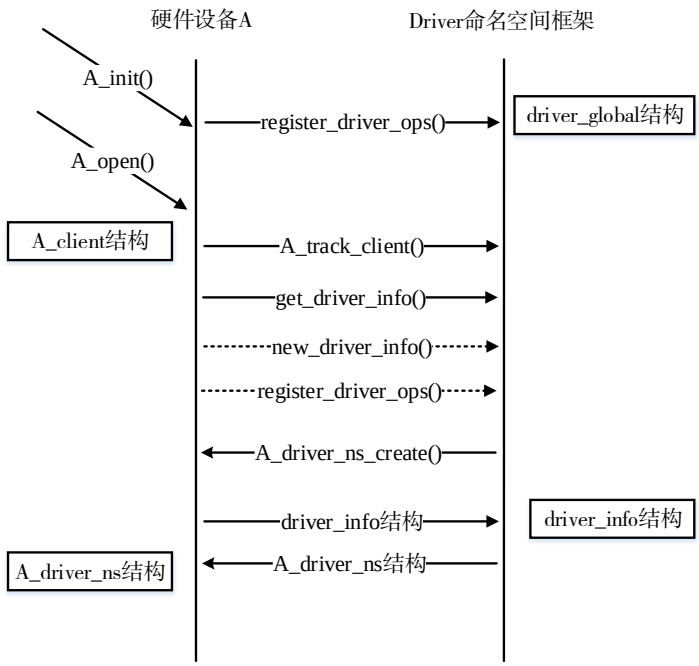

图3.7　硬件设备注册在Driver命名空间框架原理图

虽然硬件设备子系统已经初始化完成，但还没有被真正使用，所以相应的 driver_info 结构也没有创建。当系统中某个进程访问 A 硬件设备时 A_open()被调用，这个函数会创建 A_driver_ns 结构。前文提到过，每个注册在 Driver 命名空间中的硬件设备都要定义这个结构，它是连接 Driver 命名空间与具体硬件设备的纽带。A_driver_ns 中包含了 driver_info 结构。每次打开 A 硬件设备会创建一个 A_client 对象。在依次调用 A_track_client、get_driver_info 接口的过程中，首先会检查访问硬件设备的进程所属的 Driver 命名空间中是否已注册该硬件设备，有的话就直接返回 driver_info 结构体，否则就调用 new_driver_info()新建一个该结构体，新建该结构体的过程是通过调用 register_driver_ops 函数注册的回调函数 A_driver_ns_create 来完成。流程如图 3.7 所示，其中右向箭头表示定义在 Driver 命名空间框架中的通用接口，左向箭头表示定义在各个硬件设备子系统中的回调函数。

A_driver_ns_create 函数注册在 Driver 命名空间框架中，具体实现则在每个硬件设备程序中定义，此函数首先会创建连接 Driver 命名空间框架与具体硬件设备的 A_driver_ns 结构，然后注册状态切换时的 notifier 函数。notifier 函数链如图 3.8 所示，driver_namespace 结构中的 notifiers 成员链接了注册在该 Driver 命名空间中所有硬件设备 driver_info 结构中的 notifier 成员。

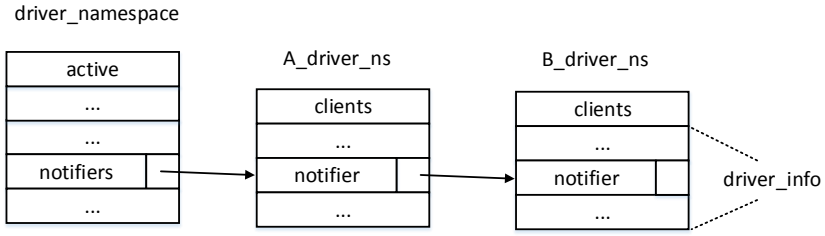

图3.8　notifiers、notifier关系示意图

通过数据结构的设计与硬件设备注册的 API 设计后，假设当前系统中构造出 Driver 命名空间 1 与 Driver 命名空间 2，其中 Driver 命名空间 1 注册了 A 和 B 两个硬件设备，而 Driver 命名空间 2 则只注册了 A 设备。他们的关系如图 3.9 所示：

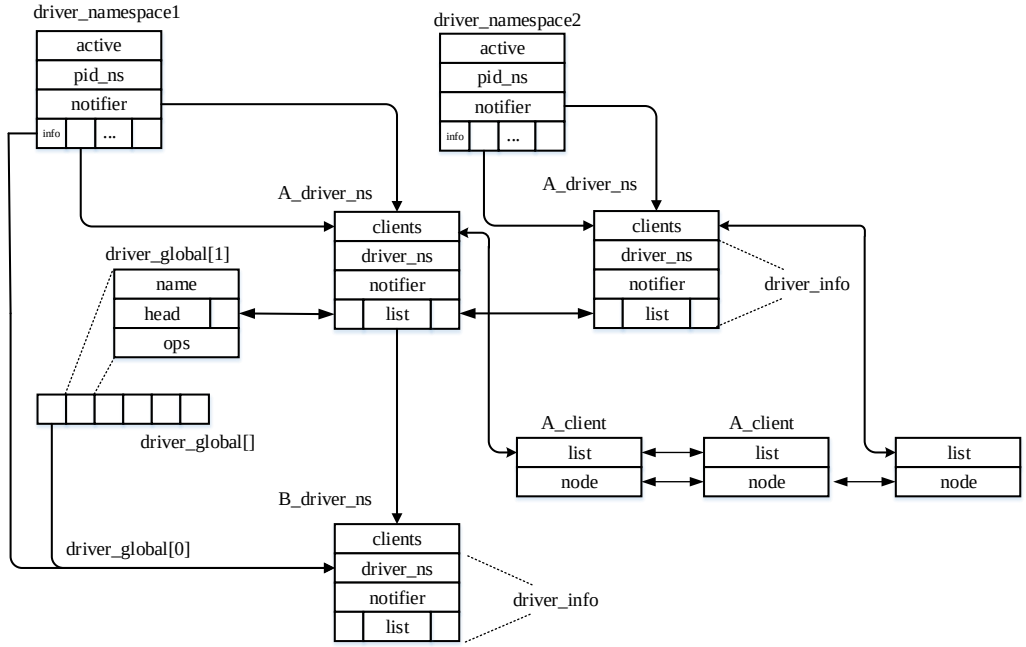

图3.9　Driver命名空间完整数据结构关系图

3.2.3 active-inactive模型

每个 Driver 命名空间中都可以注册多个硬件设备，同一个硬件设备可同时注册在多个 Driver 命名空间中。为了实现多个 Driver 命名空间能够隔离地访问设备，在此基础上设计并实现了 active-inactive 模型。整个系统由一个 active 的 Driver 命名空间与若干 inactive 的 Driver 命名空间组成，只有 active 的 Driver 命名空间中的进程对硬件设备的访问是有效的，每个硬件设备则会在初始化时注册用于状态切换发生时的回调函数。

该模型的实现除了要设计出支持该模型所需的 API，还需要依赖于内核现有的通知链机制(Linux kernel notification mechanism)[49]与/proc 虚拟文件系统两部分。内核通知链也被称为事件通知链，用来实现 Linux 中的某个子系统的状态发生变化时告知与该子系统关联的其它子系统。内核通知链由子系统列表和事件通知链两条主线组成，子系统列表注册了所有需要使用内核通知链的子系统，而每个事件通知链则是一个 notifier_block 结构(如图 3.10)，注册了特定的事件。

```
struct notifier_block {
    int (*notifier_call)(struct notifier_block *, unsigned long, void *);
    struct notifier_block *next;
    int priority;
};
```

图3.10　notifier_block结构体示意图

每当内核中发生特定的事件时，就会调用各个子系统的回调处理函数来响应相应的处理过程，而其中 notifier_call 就是当事件发生时应该调用的函数，实现在每个硬件设备子系统程序中。

Driver 命名空间的切换是通过/proc 虚拟文件系统[50]进行的。在 Linux 中，/proc 目录中包含着各类用来描述内核状态的文件，如/proc/PID/描述了当前进程的所有状态信息、/proc/fs/描述了当前系统中文件系统类型的信息，是连接用户空间与内核空间的桥梁。选用/proc 文件系统的原因有 3 点：第一，某个进程的各类命名空间信息是通过/proc 文件系统显示的；第二，系统启动时会初始化一个根命名空间，可以通过/proc 文件目录读取或管理所有子命名空间的信息；第三，内核为/proc 目录提供了 proc_mkdir()、proc_create()、proc_write()等接口，可以安全、简洁、方便的操作该目录。Driver 命名空间状态切换是由 set_state_switch() 接口实现，和其他命名空间一样，每个 Driver 命名空间初始化时会创建 /proc/driver_namespace 目录，当该目录捕获写事件时，会触发 set_state_switch 接口被调用。关键代码如图 3.11 所示：

```
void set_state_switch(struct driver_namespace *following_ns)
{
    ……
    current_ns = active_driver_ns;
    (void) blocking_notifier_call_chain(& current_ns->notifier,INACTIVE,current_ns);
    ……
    following_ns->active = true;
    active_driver_ns =  following_ns;
    ……
    (void) blocking_notifier_call_chain(& following_ns->notifier,ACTIVE, following_ns);
    ……
};
```

图3.11　set_state_switch函数核心代码

该函数需要调用已注册的 notifiers 函数，比如 Driver 命名空间 A 切换到 Driver 命名空间 B 时会先发 INACTIVE 事件到 Driver 命名空间 A 中注册的所有硬件硬件设备，通知其切换成 inactive 状态；然后将 Driver 命名空间 B 设为 active 状态；最后发送 ACTIVE 事件到 Driver 命名空间 B 中注册的所有设备。而对于每个硬件设备来说都会定义切换时的回调函数来响应切换过程。综合上述三个部分的工作原理介绍，Driver 命名空间框架工作原理如图 3.12 所示：

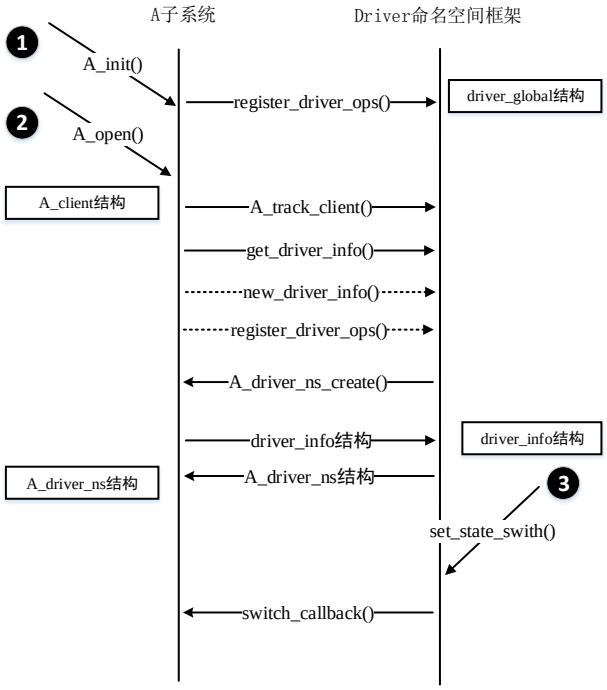

图3.12　Driver命名空间工作原理示意图

3.3　本章小结

本章详细阐述了 Driver 命名空间框架的设计方法与工作原理。宿主系统构造出的每个虚拟客户系统会关联一个 Driver 命名空间，和内核中各类命名空间机制类似，目的是为每个系统提供一个隔离、独立的访问硬件设备的视图。首先介绍了 Driver 命名空间框架的设计内容，然后详细介绍了该框架的工作原理，包括将硬件设备在 Driver 命名空间中标记的相关内核数据结构设计、将硬件设备注册在 Driver 命名空间框架中的 API 设计以及 active-inactive 模型的设计。整个系统中由一个 active 和若干个 inactive 状态的虚拟客户系统组成，只有处于 active 虚拟客户系统中的进程对设备的访问是有效的，宿主系统通过内核通知链机制与 /proc 目录来管理和切换虚拟客户系统。该框架保证了多个系统基于一套硬件设备的隔离复用，高效、简洁。

第四章　Android系统虚拟化方案设计

本章介绍 Android 系统虚拟化方案的整体架构设计和若干子系统的虚拟化架构设计。首先介绍本文设计的 Android 操作系统虚拟化方案的特点，其次阐述系统架构总设计方案，通过分析 Android 系统的硬件设备类别，最后介绍显示设备、输入设备、Backlights-LEDs 等核心硬件子系统的虚拟化方案设计。

4.1　设计目标及特点

本文基于移动 ARM 开发平台，将 Android 系统具有丰富的应用程序、多样化硬件设备的特点与虚拟化技术相结合，提出了一种轻量级的虚拟化架构，通过在宿主 Android 系统上构造出两个完整的虚拟 Android 系统运行环境，实现了一种操作系统级的移动虚拟化架构。本文设计并实现的轻量级虚拟化架构具有如下特点：

(1) 实现了一种操作系统级别的虚拟化架构。基于移动 ARM 平台，支持在宿主系统上同时运行两个虚拟客户系统。每个虚拟客户系统相当于一个独立隔离的使用场景和用户角色，满足了使用场景的多样性以及多用户角色问题。

(2) 系统在初始时只需运行一个最基本的系统实例，称之为宿主系统。由宿主系统来构造出两个虚拟客户系统，这些虚拟客户系统可同时运行在宿主系统实例中，与宿主系统共享内核。通过内核提供的轻量级命名空间资源隔离机制，为每个虚拟客户系统提供了一个独立隔离的运行环境且虚拟客户系统之间不允许相互检查、访问、篡改数据。

(3) 改变了传统硬件级虚拟化需要虚拟所有硬件设备的方式，通过将通用的 Driver 命名空间框架引入到内核现有的命名空间机制中，使得两个虚拟客户系统可隔离地访问 Android 系统中多个核心硬件设备子系统。

(4) 提出了 active-inactive 模型。该模型将两个虚拟客户系统关联的 Driver 命名空间分别被标记为 active 与 inactive 状态。所有虚拟客户系统均同时运行在系统中，inactive 的虚拟客户系统运行在后台，能够接收系统事件和执行任务，但不在屏幕上呈现内容，当且仅当 active 虚拟客户系统中的进程对硬件设备的访问是有效的。当进程需要访问硬件设备时，首先会先判断该进程所处的虚拟客户系统是否为激活状态：若是，则可以直接对设备进行操作；否则，系统需要先将当前 active 的虚拟客户系统设为 inactive，再将此虚拟客户系统设置为 active 状态

后该进程对设备的操作才是有效的。这种简单而强大的模型使得系统可以有效的跨越多个虚拟客户系统来实现硬件设备的多路复用。

(5) 由于总是处在激活状态下的虚拟客户系统直接访问硬件设备，使得额外的开销非常小，应用程序也总是运行在激活状态下的虚拟客户系统，因此应用程序也可以直接访问硬件，他们和在原生 Android 系统运行时一样快。实验结果证明虚拟化的引入并没有带来额外的系统开销。

和传统的 Hypervisor 架构通过引入 VMM 层来管理虚拟机的方法相比，本方案具有更少的虚拟化引入开销，是一种较完善且可以支持包括输入设备子系统、显示子系统、背光灯、电量指示灯子系统等设备在内 Android 操作系统虚拟化新方案。

4.2　系统架构设计

4.2.1　Android硬件设备概述

Android 系统除了具有丰富多样的应用软件以外，还集成了大量各式各样的硬件设备，占据着移动设备领域的最大市场份额。表 4. 1 展示了当前 Android 系统所支持的硬件设备列表，其中带*的设备表示 Android 特有的硬件驱动设备。

表4. 1　Android系统硬件驱动类型

硬件设备名称	设备描述
Alarm*	Android 系统的铃声子系统，包括来电铃声、短信铃声、系统提醒铃声等
Audio	Android 系统的音频子系统，包括音量管理、音频策略管理等功能
Bluetooth(闭源)	Android 系统的蓝牙子系统，用于短距离通信协议
Camera(闭源)	Android 系统的照相功能子系统
Input	Android 系统的输入设备子系统，包括了触摸屏、输入按键、输入传感器
Network(闭源)	Android 系统的 Wi-Fi 以及蜂窝数据子系统
Logger*	Android 系统的日志子系统，用于提供一个轻量级的日志记录驱动设备
Radio(闭源)	Android 系统无线通信子系统，如智能手机的 GSM、CDMA 等通信协议栈
Framebuffer	Android 系统的显示子系统
Backlight	Android 系统的背光灯 LCD 子系统
LEDs	Android 系统的 LED 指示灯子系统

列表中的硬件设备主要分成如下 3 类：

(1) 依赖于 Linux 内核的硬件设备子系统：LEDs 指示灯子系统、Audio 音频子系统、 Backlight 背光灯子系统、Framebuffer 显示子系统以及 Input 输入设备子系统。

(2) Android 系统特有的硬件设备子系统：Alarm 铃声子系统、Logger 日志文件子系统。

(3) 闭源的硬件设备子系统：Bluetooth 蓝牙子系统、Camera 照相子系统、Network 网络子系统以及 Radio 无线通信子系统[51]。

由于无法让闭源的硬件设备子系统感知 Driver 命名空间框架，因此本文选取了前两类中对于 Android 系统最核心的硬件设备子系统进行了虚拟化方案的研究与实现，在系统虚拟化的基础上完成了包括 LEDs 指示灯子系统、Backlight 背光灯子系统、Framebuffer 显示子系统以及 Input 输入设备子系统在内的硬件设备的虚拟化方案的实现。

4.2.2　系统架构

本系统完整的框架如图 4.1 所示：

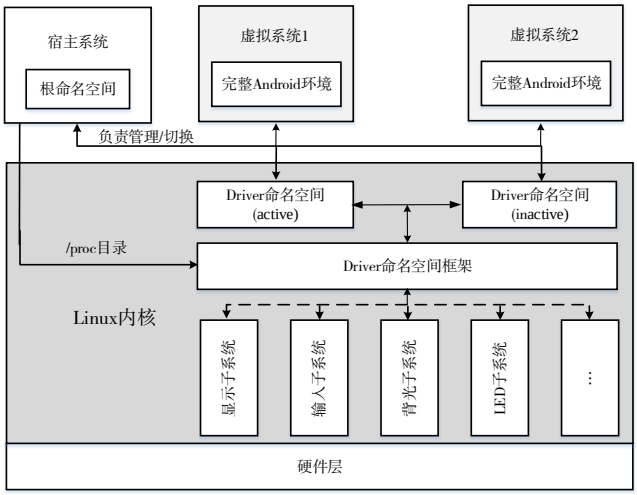

图4.1　系统完整架构图

整个系统启动时会先启动一个最小的初始化环境，是一个最基本的 Android 运行环境，称之为根命名空间。由 init 进程构建了 Android 的基本服务、文件系统以及各类命名空间。它类似于 Xen 里的 dom0，真正负责管理和切换两个虚拟客户系统，虚拟客户系统之间进行切换时都需要通过/proc 目录来向根命名空间申请。

选择 Android 作为虚拟客户系统原因有两点：第一，和宿主系统保持一致；第二，Android 系统具有开源、硬件接口良好封装性、应用软件多样性等优点，

极大丰富了移动虚拟化的应用场景。在根命名空间成功启动后，通过为宿主系统的 init 进程构造各类新的子命名空间，初始化出两个虚拟客户 Android 系统运行环境。MNT 命名空间提供了一个隔离、完整的文件系统视图；PID 命名空间提供了一个独立的进程视图；Driver 命名空间则提供了一个隔离的硬件设备访问视图。通过各类命名空间提供的各类系统资源隔离视图的集合，为每个虚拟客户系统创建出了一个完整而独立的 Android 操作系统运行环境。

两个虚拟客户系统可同时运行在一套硬件平台上，每个虚拟客户系统都具有访问硬件设备的权限。为了能够让不同的虚拟客户系统隔离的访问硬件资源，本文将 Driver 命名空间框架引入到内核现有的命名空间机制中，在不同的 Driver 命名空间中可独立注册各类硬件设备，如 A 虚拟客户系统的 Driver 命名空间中注册了输入设备子系统和显示设备子系统表示该系统可以访问输入设备以及显示设备，B 虚拟客户系统的 Driver 命名空间中注册了输入设备子系统表示该系统可以访问输入设备子系统，这种机制初步实现了不同虚拟客户系统对硬件资源的隔离访问。

在将 Driver 命名空间框架引入到内核的基础上，提出了 active-inactive 模型用来保证两个虚拟客户系统对硬件资源的隔离复用机制。对于某个硬件驱动设备来说，可能存在着多个虚拟客户系统中的进程在同时访问该设备资源，该模型的提出使得当且仅当处于 active 的 Driver 命名空间的虚拟客户系统中的进程对于真实硬件设备的访问是有效的。而对于 inactive 下的进程只是对于设备虚拟的访问。目前实现了包括显示设备子系统、输入设备子系统、LEDs 以及 Backlight 背光灯子系统在内的 Android 最核心的硬件设备的隔离复用。

每个虚拟 Android 客户系统运行在用户空间下，系统层面利用内核的轻量级操作系统隔离机制，包括文件系统隔离、进程空间隔离；硬件驱动层面利用 Driver 命名空间框架实现了对硬件资源的隔离复用。每个虚拟客户系统系统都有自己私有的空间，可以在各自虚拟系统中并发的使用操作系统的资源。不同虚拟客户系统之间相互独立、隔离，且不可以篡改、访问其他虚拟客户系统。只有根命名空间可以通过/proc 文件目录的读写来进行对虚拟客户系统的管理与切换。

4.3　显示子系统虚拟化方案设计

显示设备子系统主要由 Android 系统上层的 SurfaceFlinger、硬件抽象层(Hardware Abstract Layer)的 Gralloc、内核层中的 Framebuffer 显示驱动程序以及硬件层的驱动四部分组成，它是 Android 系统最重要的子系统之一[52]。系统上层

使用一个被称为 SurfaceFlinger 的进程来合成应用程序的窗口到屏幕上，内核层则提供一个标准封装好的 FrameBuffer(下文统称为 FB)显示驱动程序为底层物理硬件提供了抽象显示，一个进程可以读写显示缓冲区以及控制显示设备。这种可以直接映射内存方式和显示子系统的性能要求为移动虚拟化提出了新的挑战。和 Android 系统采用分层的架构一样，显示子系统的框架如图 4.2 所示：

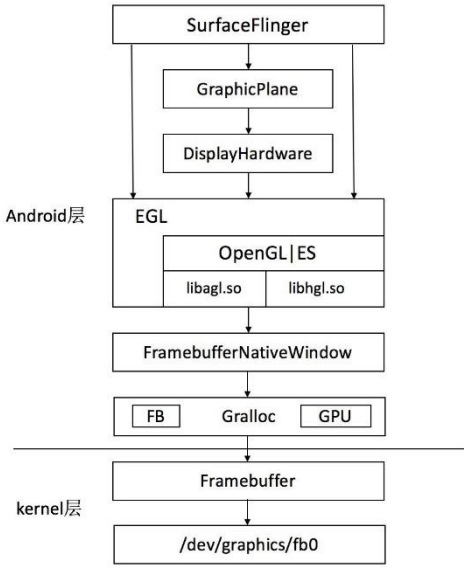

图4.2　显示子系统框架图

SurfaceFlinger 中的 GraphicPlane 类首先会创建一个对象用来标记 Android 系统的屏幕。在 SurfaceFlinger::readyToRun 后，完成对 GraphicPlane 类中包含的 DisplayHardware 对象实例的初始化。DisplayHardware 利用 EGL 来实现 OpenGL|ES 设置“本地化”所需的窗口。OpenGL|ES 通过 egl.cfg 配置文件来提示 EGL 所需的软硬件加载方式，确定后会通知 OpenGL|ES 加载两种库文件，其中 libhgl.so 用于硬件加速，libagl.so 用于软件加速。OpenGL|ES 在不同平台上需要与具体平台上的窗口系统建立起关联，而 FramebufferNativeWindow 就是将 OpenGL|ES 本地化的中介。

Android 中不是所有的硬件设备都有标准的内核接口，由于驱动涉及到 GPL 版权问题，因此制造商并不愿意公开自己的硬件驱动。Android 针对某些硬件需要有自己定制的要求，因此引入硬件抽象层。该层位于用户空间，向上为应用程序提供必要的接口，向下屏蔽底层复杂的驱动。Gralloc 作为显示子系统的抽象，为应用程序提供访问 FB 的接口，负责打开内核中的 FB 字符设备；同时也负责管理 FB 结构的初始化分配和释放操作。

Android 系统中，FB 提供的设备文件节点是/dev/graphics/fb*，理论上支持多个屏幕显示，规定 fb0 是主显示屏幕，必须存在。FB 将显示缓冲区抽象，是内

核中为显示设备提供的一个接口，使得开发者不必关心物理显示缓冲区的具体位置及存放方式。默认的缓冲区大小是整屏数据的两倍。应用程序可以把此设备文件映射到进程的虚拟地址空间中再进行操作。具体步骤如下：

(1) 打开/dev/graphics/fb0 设备文件。FB 是一个字符设备，主设备号为 29，使用的不是整个内存区域，而是显存部分。它可以看成显存的一个映射，在内核中用一个 fb_info 结构表示，进程通过操作此 fb_info 结构可以实现直接对显存的读写访问。

(2) 成功打开设备文件后，利用一系列标准的 ioctl 操作获取当前显示屏幕的 fb_fix_screeninfo 和 fb_var_screeninfo 等参数信息。

(3) 通过系统调用 mmap 将打开的 FB 设备文件映射到进程的虚拟地址空间中。此系统调用在内核中对应着一个 file_operations 结构的 mmap 文件操作函数，在将打开的 FB 设备成功映射到进程的虚拟地址空间后，进程对映射后的这段虚拟地址空间的读写就等价于对显存的读写，因为 FB 设备关联着一段显存的物理地址。

为了实现显示设备子系统的虚拟化，使得真实的 FB 能够被多个虚拟系统访问，本文在内核级 Driver 名称空间和 active-inactive 模型基础上，通过一个被称为 virtual_fb 的虚拟 FB 设备来实现对 FB 驱动设备的复用。virtual_fb 作为一个标准的 FB 设备被注册，可以和真实的物理设备进行通信，被每个虚拟客户系统访问。每个虚拟客户系统都维系着一个 virtual_fb 结构，active 状态的虚拟客户系统中的 virtual_fb 可以直接与真实的 FB 进行通信，独占屏幕显存以及底层的显示硬件；而每个 inactive 状态的虚拟客户系统则维护一个虚拟硬件状态，将显示的数据放在内存中的虚拟 FB 中。

允许 active 状态的虚拟客户系统中的 virtual_fb 直接访问真实的 FB 硬件设备，这就允许应用程序能够充分利用物理设备。当应用程序运行在 active 状态下的虚拟客户系统时，在成功 mmap 一个已经打开的 virtual_fb 设备文件后，该 virtual_fb 虚拟结构仅仅要做的就是将地址映射到真实的 FB 硬件提供的显存地址。这种方案保证了与原生系统相同的方式零开销的访问显存。同时不允许 inactive 状态的虚拟客户系统下的 virtual_fb 直接访问 FB 驱动程序，确保了 active 状态的虚拟客户系统能够拥有专属的硬件访问。运行在 inactive 状态的虚拟客户系统中的应用程序 mmap 一个 FB 设备文件时，virtual_fb 仅仅是将自己的缓冲区映射到进程的虚拟地址空间，不做实际的显示。显示设备子系统的虚拟化实现架构如图 4.3 所示：

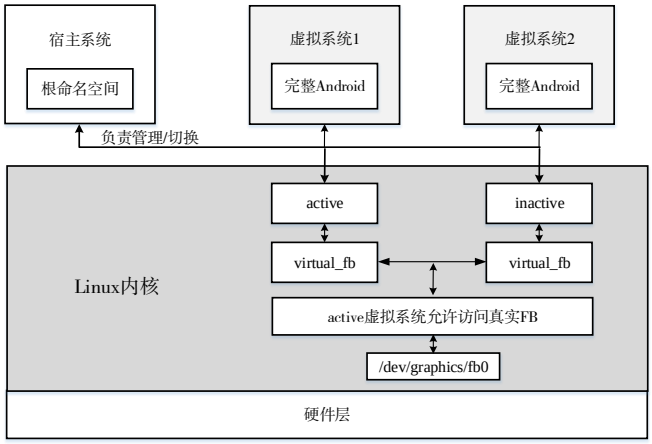

图4.3　显示子系统虚拟化方案架构图

在实现了显示子系统的框架设计之后，保证虚拟客户系统间切换时能正常显示就成了此框架实现的重点。在每次虚拟客户系统切换的时候需要如下三个步骤。

(1) 屏幕显存重映射。每个将要运行在 inactive 状态虚拟客户系统的进程要将他们的虚拟地址重新映射到 virtual_fb 中的显存中；而将要运行在 active 状态的进程则把虚拟地址重新映射到实际的物理设备显存中去。

(2) 屏幕显存的深拷贝。将当前屏幕显存的内容复制到之前 active 状态的虚拟客户系统的 virtual_fb 缓冲区，并且将新 active 状态的虚拟客户系统的 virtual_fb 缓冲区内容复制到屏幕的显存中。

(3) 硬件状态同步。通过将当前硬件状态保存到之前 active 的虚拟客户系统的虚拟硬件状态，然后设置当前硬件状态到新的 active 的虚拟客户系统的虚拟硬件状态中去。具体实现将在第五章介绍。

4.4　输入子系统虚拟化方案设计

Linux 内核针对输入提供了标准的接口，Android 系统输入设备子系统可处理各种输入设备，如触摸屏、导航轮、GPS、接近传感器、光传感器、耳机输入控件和输入按键等，和 Linux 原有的输入子系统架构保持一致。该类子系统主要由输入子系统核心层(Input Core)、硬件驱动层以及输入事件处理程序(Event Handler)三部分组成[53]。

输入子系统核心层作为整个输入子系统的中间层位于内核中。该层为硬件驱动层中各类输入设备封装接口并同时收集输入事件处理程序中的统一处理模版。通过提供 input_handle 结构，将输入设备驱动和输入事件处理程序连接在一起。

输入事件处理程序直接连接到用户空间的输入设备并且创建相应的处理流程，同一个输入事件处理程序允许处理多个设备，通过一个 input_handler 结构体来实现输入事件的处理工作。该结构中具体实现了如何处理、响应输入事件，根据一定的规则对事件进行处理。可根据具体的需要实现其中的一些函数，当一个新的 input_handler 处理器需要被添加进输入子系统的时候，input_register_handler()函数被调用来完成该功能。内核中规定 input_handler 处理器和具体的输入硬件设备允许一对多的关系，通过宏定义允许一个输入事件处理程序最多处理 32 个输入设备。输入子系统数据结构体关系如图 4.4 所示：

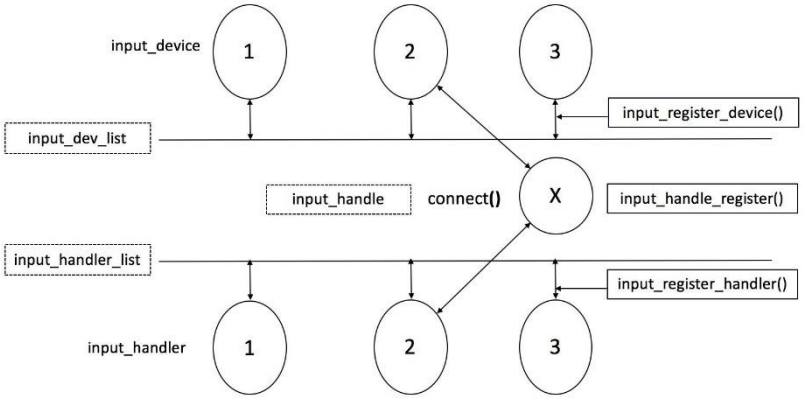

图4.4　输入子系统核心数据结构关系图

硬件设备通过 input_register_device 函数将各个 input_device 结构的输入设备注册在 input_dev_list 上；同理，输入事件处理程序通过 input_register_handler 将各个 input_handler 结构的输入事件处理程序注册在 input_handler_list 上；最终通过输入核心层的 connet 函数将输入设备与输入事件处理程序连接起来。

通常，输入事件在默认的情况下会被发送到所有监听该输入事件的进程中，但这个机制并不支持多个虚拟客户系统提供所需的隔离。由上文分析可知，为了让输入子系统使用 Driver 命名空间，只需要修改输入事件处理程序即可。对于每个进程监听的输入事件，输入事件处理程序首先检查是否在 active 状态下的对应的 Driver 名称空间，如果是，则相应该输入事件；如果不是，则不唤醒特定的处理流程。

evdev 是输入事件处理程序的核心组成部分，硬件设备通过输入核心层传给 evdev 处理程序后，该处理程序可以响应硬件设备并提供时间处理流程。在输入子系统之中添加 Driver 命名空间逻辑可以实现在不同的虚拟客户系统中隔离复用输入事件。通过追踪每一个 Driver 名称空间中注册的输入设备的每一个客户(clients)，输入事件只在那些 active 的虚拟客户系统中的某个 clients 被响应。而 inactive 虚拟客户系统中的 clients 是无法察觉到相应的输入事件的。

evdev 事件处理程序通过维护着一个 grab 状态来判断该事件处理程序是否成功捕获输入事件。grab 状态取决于 Driver 命名空间框架上下文，只有处在一个 active 的虚拟客户系统中的 clients 才能够捕获一个输入事件。而在 inactive 的虚拟客户系统中，输入事件的 grab 过程是虚拟的：如果在相同的虚拟客户系统中没有其它的 clients 获取 grab 状态，该捕获操作成功并且 clients 被标记为 grab 成功，但不采取任何实质的处理流程。当另一个命名空间变成 active 状态后，进行 grab 状态交换。当前真正捕获的设备解除，成为虚拟；而当前虚拟状态的 grab 成为真正的捕获状态。

4.5　Backlight和LEDs子系统虚拟化框架设计

Android 系统中 Backlight-LCD 背光子系统和 LEDs 子系统紧密相关，背光子系统通过对 LCD 显示屏的设置来实现对 Android 系统亮度、电量以及饱和度的调节；LEDs 子系统则为背光子系统提供了硬件驱动程序，是亮度调节的真实硬件设备。除了参与背光屏亮度调节机制，LEDs 子系统还存在于移动设备中的指示灯中[54]。通过将 Driver 命名空间框架引入到该子系统中，实现了对上述两个子系统的虚拟化。

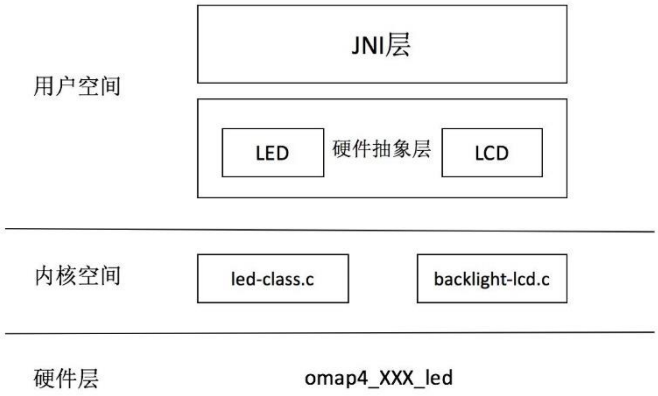

图4.5　Backlights和LEDs子系统框架图

背光与 LEDs 子系统框架如图 4.5 所示，背光 JNI 层是背光子系统位于用户空间的最上层，通过调用硬件抽象层的方法，为应用程序实现一个设置亮度、饱和度的接口。Android 设备中的 LEDs 广泛存在，不只是有 LCD 背光屏，还有其他很多类型的 LED，如指示灯 LEDs。背光硬件抽象层实现了这些 LED 的抽象，向下屏蔽硬件差异，向上提供对 LEDs 的操作接口。内核层提供两个驱动注册的接口, led-class.c 文件中的 leds_init()函数在初始化时通过 led_classdev_register 函数在背光子系统中创建/sys/class/leds/子目录并在该子目录下创建属性文件亮度

brightness、电量 power、对比度 contrast。当 LCD 驱动调用 led_classdev_register 函数时，则会在目录/sys/class/leds 创建子目录 lcd-backlight 和相关的属性文件。背光驱动层中的 omap4_XXX_display_probe 函数通过上层的 led_classdev_register 函数调用来注册 LED 与 LCD 设备。

当背光和 LEDs 子系统在用户空间内的相关属性文件创建好后，对属性文件读操作通过 led_brightness_show 函数完成；写操作则调用 led_contrast_store 函数。对于电量的设置则通过对 led_power_store 的修改，该修改会通过电源管理器将亮度、对比度等数值反馈给 Android 的电源管理 service。对于 Backlight 背光子系统和 LEDs 子系统，通过将 Driver 命名空间框架引入上述子系统来实现虚拟化遵循如下规则：

(1) 在 active 状态的虚拟客户系统和 inactive 状态的虚拟客户系统均可注册 LEDs 设备和 LCD 背光设备。

(2) 对于 LEDs 子系统而言，仅允许处于 active 状态的虚拟客户系统设置亮度、电量、对比度。

(3) 对于 Backlight 子系统而言，仅允许处于 active 状态的虚拟客户系统设置亮度与对比度。

4.6　本章小结

本章详细设计了系统的整体架构设计和 Android 系统最核心的若干硬件设备子系统虚拟化方案设计。首先介绍了本文设计的 Android 操作系统虚拟化方案的特点，其次详细阐述了系统架构总设计方案，并对 Android 系统的硬件设备进行分类。除去闭源模块，选取系统中最核心的显示设备、输入设备、背光 LEDs 设备作为研究对象，完成了上述硬件设备子系统的虚拟化方案设计。

第五章　虚拟化方案的实现与验证实验

本章阐述本文提出的轻量级 Android 操作系统虚拟化方案的实现原理以及验证实验。为了实现 Android 操作系统级移动虚拟化解决方案，基于 ARM 平台需要进行如下工作：

(1) 定制 Linux 内核。需要在编译内核时在.config 文件中开启 Namespace 机制的编译选项；由于本方案在内核中扩展了 Driver 命名空间框架，需要修改相应的 Kconfig 文件与 Makefile 文件。

(2) 基于宿主 Android 系统启动两个虚拟客户系统。通过为 Android 的 init 进程构造出各类子命名空间以及对 init.rc 文件的修改构造出两个虚拟客户系统实例；利用命名空间机制为虚拟客户系统提供隔离的视图(主要包括 MNT 文件系统视图、PID 进程视图、Driver 硬件设备命名空间视图)。

(3) 在将 Driver 命名空间框架引入内核的基础上，实现不同虚拟客户系统对显示设备的隔离复用，完成 Android 系统最重要的显示子系统的虚拟化。

(4) 实现不同虚拟客户系统对输入设备的隔离捕获，完成 Android 系统输入子系统的虚拟化。

(5) 同时需实现 Android 系统中若干较为关键的子系统的虚拟化工作。如 Backlight 背光子系统与 LEDs 指示灯子系统。

(6) 利用 Android 系统的 LMK(Low Momery Killer)机制，对该虚拟化方案从内存使用量上进行初步的优化。

(7) 在实验与分析部分，从宿主系统启动、虚拟客户系统启动、系统间正常切换、Android 基本特性以及系统间隔离性进行完整的功能测试工作；在性能测试上通过对 CPU 利用率以及内存使用度两个参数来衡量虚拟化的引入所带来的性能开销。

5.1　定制Linux内核

一般情况下，Android 系统的 Linux 内核是不支持 Namespace 命名空间机制的，为了使用内核现有命名空间机制，需要在编译内核前打开相关的配置选项。

表 5. 1 显示了 Namespace 的编译选项，执行 make menuconfig 图形化内核配置命令后，会在内核的目录下生成相应的.config 文件，该文件里包含了编译支持的选项。通过查找 Namesapce support 对应的选项，从而确定命名空间机制是否

真正被打开。除了.config 文件以外，在定制 Linux 内核时还需要对 Makefile 和 Kconfig 两个文件进行配置[55]。Makefile 文件用来让 make 工具编译时读取完整、详细的编译规则；而 Kconfig 文件通常分布在内核代码的子目录中，每个子目录中的 Kconfig 文件用来描述所对应模块的编译配置规则，所有子目录中 Kconfig 文件的配置规则最终会导入到内核根目录下的 kernel/.config 文件中。

表5.1　内核命名空间编译选项

名称	描述	.config 文件表示
Namespace Support	Kconfig 中开启命名空间功能	CONFIG_NAMESPACES=y
MNT Namespace	MNT 命名空间	CONFIG_MNT_NS=y
PID Namespace	PID 命名空间	CONFIG_PID_NS=y
UTS Namespace	UTS 命名空间	CONFIG_UTS_NS=y
IPC Namespace	IPC 命名空间	CONFIG_IPC_NS=y

为了将 Driver 命名空间框架编译进内核，需要对内核的两个文件进行修改：

(1) 图 5.1 描述了对 kernel/init/Kconfig 文件的修改记录。

```
config DRIVER_NS
    bool "Driver Namespace" /* 定义Driver命名空间框架 */
    default n   /* 为了不对内核造成影响，默认为不开启 */
    depends on PID_NS  /* Driver命名空间框架和PID具有一对一关系 */
help
    Muptiplex to drivers and depend on PID namespace.
```

图5.1　Kconfig文件修改记录

(2) 对 kernel/Makefile 文件的修改只需增加 obj-$(CONFIG_DRIVER_NS) += driver_namespace.o 即可。

5.2　系统启动过程

5.2.1　Android系统启动流程

当内核启动成功后，Android 系统会创建进程号等于 1 的 init 进程。而此用户空间下的 init 进程在启动时会根据 init.rc 初始化配置文件来启动 Android 系统用户空间的所有服务等[56]。图 5.2 展示了 Android 系统的启动流程。首先根据 start_kernel 函数的启动规则将内核启动，该函数的主要功能为内核各个模块的初始化工作。接着通过调用 rest_init 函数启动第一个进程(init 进程，其 PID＝1)。该函数也会同时生成内核进程 Kthreadd，该进程可以创建 Android 系统内核空间

驱动线程(其 PID=2)。在创建完成 init 进程后，kernel_init 函数会将内核初始化时保存的函数进行驱动模块初始化，之后直接调用 init_post()函数进入用户空间，执行 init 进程代码。默认的 init 进程代码会放在/init 中，通过 run_init_process 函数中的系统调用 do_execve，运行/init 程序的 Main()函数。

init 进程用于启动本地进程，首先会创建 Zygote 进程，该进程是一个用于孵化 Java 进程的本地进程，所创建的都是 Java 进程。接着 init 进程会 fork 出 Android 系统所需要的各个核心服务，其中最重要的就是 ServiceManager 服务，该服务是系统所有服务的管理者，而系统调用 fork 的规则是从 init.rc 文件的描述中获取。

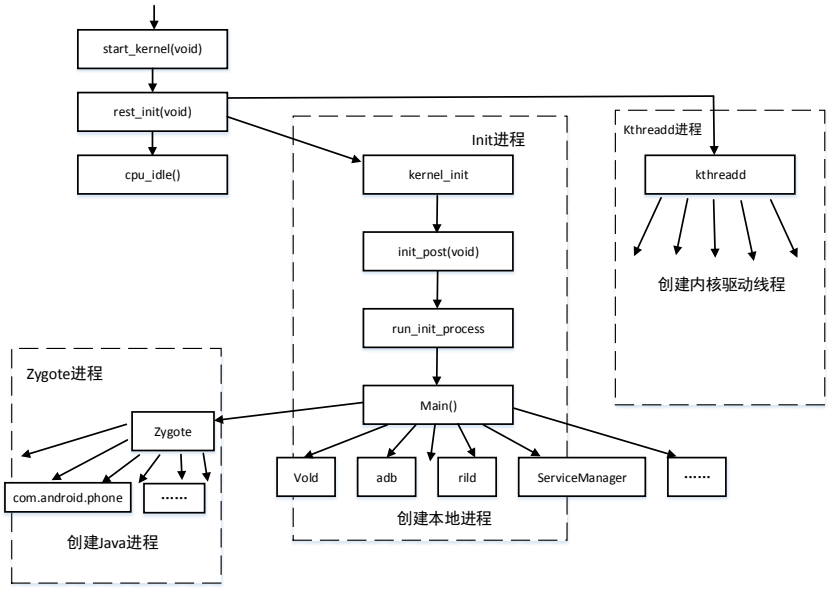

图5.2　Android系统启动流程图

Android 系统的 init 进程可以做如下事情：(1) 根据 init.rc 文件中的语法规则与内容进行分析并运行该文件；(2) 当一些关键进程死亡后，需要通过发送 SIGNAL 的方式来重启相应的进程；(3) 提供 Android 系统的属性服务。init 初始化过程中分别挂载了 tmpfs、devpts、proc、sysfs 四类文件系统。而 proc 文件系统与 Driver 命名空间框架的实现紧密相关。

init.rc 文件与 init 的关系类似于 Makefile 文件与 make 命令的关系，该文件具有特殊的语法规则。其基本语法是以行为单位、以空格作为间隔的、以#开始代表注释行。init.rc 文件主要包含 Action、Service、Command 与 Options 四种类型的命令。Action 是以 on 开头的语句，通过 trigger 关键字来决定何时执行相应的 Service；Service 以 service 开头，由 init 进程启动，一般运行于 init 进程 fork 出的子进程，所以启动 service 前需要判断对应的可执行文件是否存在。init 进程需要生成的子进程会定义在 init .rc 文件中，其中每一个 service 在启动时会通过

fork 方式生成子进程；Command 包含常用的命令，如实现文件目录的挂载等功能；Options 是 Services 的可选项，与 Service 配合使用。当 init 进程执行完 init.rc 配置文件的所有语法命令之后，一个完整 Android 系统便启动成功。

5.2.2　虚拟Android客户系统的启动

在 ARM 平台实现操作系统级虚拟化方案时，整个系统首先启动一个最小的初始化环境，称为根命名空间(宿主系统)。根命名空间启动一个最基本的 Android 环境，由 init 进程构建了 Android 的基本服务、文件系统以及各类命名空间。在根命名空间启动成功后，启动虚拟客户系统需要如下 3 个步骤：

步骤(1)：编写创建虚拟客户系统的脚本文件。

在根命名空间成功启动之后，为宿主系统的 init 进程构造各类新的子命名空间。MNT 命名空间提供了一个隔离、完整的文件系统视图；PID 命名空间提供了一个独立的进程视图；Driver 命名空间和 PID 命名空间具有一对一的关系，在每次创建新的子 PID 命名空间时被初始化，提供了一个隔离的硬件访问视图。这些命名空间的组合为子 init 进程提供了各类系统资源隔离的视图，子 init 进程完成和根 init 进程同样的功能，新生成的文件系统、关键服务会绑定在各类子命名空间中，从而构造出多个完整而独立的虚拟 Android 客户系统。命名空间机制在用户空间的创建由 clone 系统调用完成，通过在调用 clone 时候指定相应的 flag，进程可以复制自身并创建一个新的命名空间。clone 用法如下所示：

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

其中 fn 指向程序代码的指针；child_stack 参数代表为子进程分配的堆栈空间；flag 标示了子进程需要从父进程需要继承的资源，如 CLONE_NEWNS 为进程创建一个新的文件系统命名空间、CLONE_NEWPID 为进程创建一个新的进程空间；参数 arg 则代表传给子进程的参数。图 5.3 展示了 clone 调用时创建新的命名空间的流程，首先通过调用内核文件 fork.c 文件中的 do_fork 函数来检查相应的标志是否正确并完成 task_struct 的一部分复制工作。之后便将 clone 的参数传给 copy_process 进行进程的复制，该函数中提供了 copy_namespaces 来完成命名空间的复制和创建。当该函数检查 flag 成功后便调用 copy_new_namespaces 函数来完成具体类型的命名空间创建。根据 clone 函数提供的 flag 来找到对应的 copy_*_ns 函数从而完成对进程创建一个新的子命名空间。有关 copy_*_ns 函数本文已在 2.3 章节中详细介绍过。

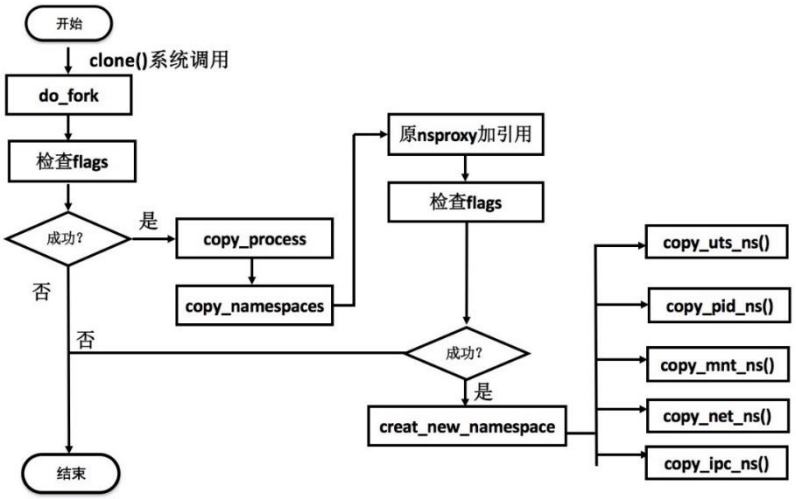

图5.3　clone创建新命名空间流程图

宿主系统的 init 进程通过 clone 系统调用指定特定的标识，从而以共享内核的方式构造出不同的虚拟客户系统，本文通过编写“new.sh”脚本来完成该工作。该脚本具有如下功能：

①在宿主系统（根命名空间）启动后，该脚本通过 clone 系统调用为 Android 的/init 进程创建一个新的 MNT 命名空间和 PID 命名空间。由于 Driver 命名和 PID 具有一对一的映射关系，因此也构造出了一个新的 Driver 命名空间，并将该 Driver 命名空间设为 active 状态。

②该脚本继续通过 clone 调用为/init 进程指定相同的 flag 标识构造出又一个新的命名空间视图，将此 Driver 命名空间设为 inactive 状态。

③对于构造出的两个独立的命名空间视图而言，利用子 init 进程读取 init.rc 配置文件构造出两个隔离完整的 Android 虚拟客户系统。通过 Driver 命名空间框架的引入，这两个虚拟客户系统一个是 active 状态，另一个为 inactive 状态。根命名空间用来管理和切换这两个虚拟客户系统。

步骤(2)：将该脚本文件添加进宿主系统启动流程。

出于安全考虑将 new.sh 脚本文件拷贝至 Android 系统 system 分区下的/system/etc/目录中，需要在交叉编译生成 ARM 平台的 Android 系统镜像文件时将该脚本拷贝进/system/etc/路径下。之后需要修改根命名空间中的 init.rc 文件使得 init.rc 可以运行该脚本。具体修改如图 5.4 所示：

```
service MNT-usbfs /system/etc/new.sh
    class main
    user root
    group root
oneshot
```

图5.4　启动脚本修改记录

步骤(3)：修改执行路径权限。

init.rc 中存在 mount ext4 ext4@system /system ro 语句，该语句说明系统的 system 分区是以只读的形式挂载的，通过 android_filesystem_config.h 配置文件修改特定目录下的权限{00550,AID_ROOT,AID_SHELL,"system/etc/new.sh"},

除了对根命名空间中的 init.rc 需要修改以外，由于每个新建的虚拟客户系统中的 init 进程通过读取各自的 init.rc 配置文件来创建虚拟 Android 系统的各个服务，虽然各个虚拟客户系统具有独立的文件系统视图，但对于根命名空间而言仍然是一个文件系统，因此需要建立每个虚拟客户系统 init.rc 文件的映射，这样有两个好处：(1) 可以使系统间实现隔离；(2) 可以个性化定制 init 进程创建的服务。为了让根命名空间可以通过 proc 文件来使用 Driver 命名空间框架，还需要在两个虚拟客户系统中禁止挂载 proc 目录。经过上述步骤，实现了在宿主系统上构造出了两个同时运行的虚拟客户系统。

5.3　显示子系统的虚拟化实现

在 Linux 中，当一个进程通过 mmap 系统调用可以将一个打开的 FB 字符设备文件映射到该进程的虚拟地址空间中，而 FB 则可以申请系统的内存作为与自己关联的显存，这样的映射方式就允许进程可以直接操作显存，从而将屏幕缓冲区的内容显示在 ARM 平台的显示屏上。FB 框架如图 5.5 所示：

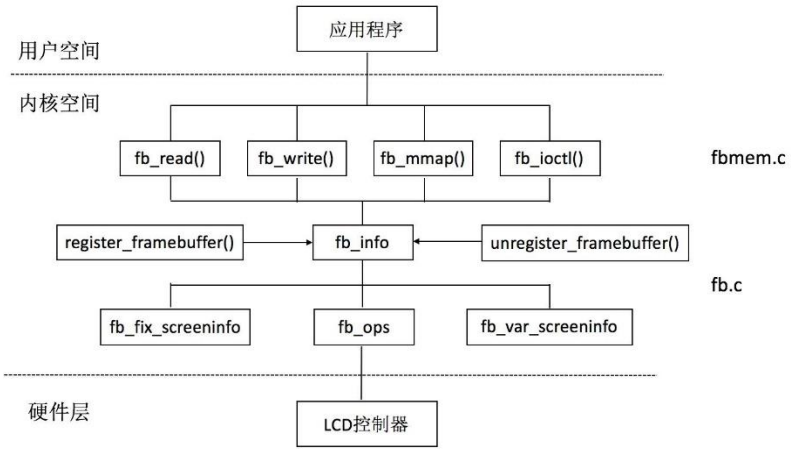

图5.5　FB架构图

在用户空间中，由 4.3 节分析可知，应用程序通过 SurfaceFlinger 调用 libgl.so 进行图像处理，并将待显示的数据交给 Gralloc，在 gralloc_device_open 函数中通过指定 FB 驱动以及 GPU 驱动将需要待显示的数据刷新在显示设备中。当用户空间的某个进程使用 FB 时，通过加载 Gralloc 模块中的 private_handle_t 结构来

描述该进程打开的 FB 设备，成员 fd 指向打开的字符设备缓冲区的文件描述符，通过 gralloc_alloc_buffer 函数完成对 FB 的初始化工作。在 Gralloc 层中，fb_context_t 结构是对 FB 设备的统一描述，它是 FB 内部使用的一个类，包含了众多信息（如实际物理地址）。它仅有唯一成员就是 framebuffer_device_t，该结构描述缓冲区信息，当 Gralloc 层中调用 fb_device_open 函数打开 FB 时，会触发 mapFramebufferLocked()完成初始化工作，通过一系列 ioctl 将缓冲区的信息保存在 fb_var_screeninfo 和 fb_fix_screeninfo，并将当前 FB 显存缓冲区映射到当前进程的虚拟地址空间中，这样 FB 设备就完成了初始化工作。

在内核空间中，fbmem.c 和 fb.c 这两个文件是 FB 设备驱动的中间层，为用户空间提供系统调用，同时为底层硬件提供接口[57]。fbmem.c 定义了一组文件操作，包括 fb_read、fb_write、fb_ioctl 和 fb_mmap 为用户空间提供操作 FB 设备的接口。一个 FB 对应一个 fb_info 结构，它包括了一个 FB 帧缓冲区的属性和操作的完整集合。fb_ops 结构用来实现对 FB 设备的操作，比如 fb_save_state 保存硬件状态操作等，这些操作允许设备使用者自主编写。fb_fix_screeninfo 结构保存了当前 FB 缓冲区的固定参数，如缓冲区物理地址、缓冲区的长度等。fb_var_screeninfo 包含了 FB 的可变参数，如屏幕分辨率、透明度等。

通过对 Android 显示子系统框架的分析，结合虚拟化 FB 设备时所面临的挑战，实现了一个完整的方案。在两个虚拟客户系统中对 FB 设备初始化时分别创建一个虚拟的 FB 结构 virtual_fb，由于一个 FB 对应着一个 fb_info 结构，实际上就是创建一个 fb_info 结构，只是将该结构中的 flag 成员设为虚拟状态。active 状态的虚拟客户系统中的应用程序通过 mmap 调用，将 virtual_fb 映射到进程的虚拟地址空间时，把 virtual_fb 的物理地址直接指向真实 FB 的地址，保证了 active 状态的虚拟客户系统中进程可以直接访问真实的 FB 设备；inactive 状态的虚拟客户系统中的应用仅仅将它的 virtual_fb 映射到进程的虚拟地址空间，不会在屏幕上显示数据。为了实现该方案，首先需要在显示子系统中增加相应的 API 来为 Driver 命名空间框架回调；其次需要在 fbmem.c 中添加逻辑，在 fb_read、fb_write、fb_ioctl 和 fb_mmap 等文件操作中首先判断 fb_info 是否位于 active 的虚拟客户系统，如果是，则将已映射的虚拟 FB 结构直接指向真实的 FB 设备；否则仅仅实现 mmap 映射。显示子系统的虚拟化实现需要如下步骤：

(1) 在内核现有的 FB 框架中为两个虚拟客户系统各注册一个虚拟的 FB 结构。由于每个虚拟 FB 结构要作为标准的 fb_info 结构被注册，因此当 register_framebuffer 被调用时需要提供一个接口来完成创建虚拟 FB 结构的功能。该接口需要完成如下功能①调用标准 FB 设备的 framebuffer_alloc 函数为虚拟 FB 分配空间；②在内核中设定宏 VIRT_FB 用于将虚拟 FB 结构的 flag 成员设为该

宏，用于标记虚拟 FB 结构；③创建虚拟 FB 结构中的 fbops 成员，该成员包含了对 FB 设备的操作函数集合，对这些函数集的创建可以使得虚拟 FB 设备可以像真实的 FB 设备一样提供接口；④需要对表 5.2 中的 4 个参数进行初始化。

表5.2　FB中4个核心参数

参数名称	描述
fix.smem_start	FB 缓冲区的物理地址起始位置，通过 vmalloc_user(fix.smem_len)获取
fix.smem_len	FB 缓冲区长度，和标准 fb_info 相同
screen_base	虚拟基地址，和 fix.smem_start 地址相同
screen_size	虚拟内存大小，和标准 fb_info 相同

(2) 设计与 Driver 命名空间框架交互的 API。为了将显示子系统与 Driver 命名空间框架结合来实现 FB 设备的虚拟化，首先需要针对该子系统设计数据结构，显示子系统的数据结构关系如图 5.6 所示：

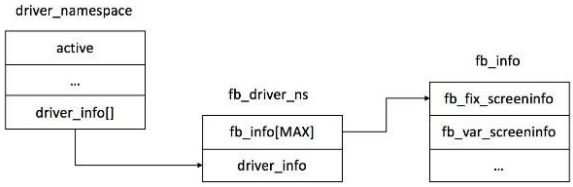

图5.6　显示子系统数据结构关系图

由第三章可知，每个 fb_driver_ns 结构是显示硬件设备与 Driver 命名空间框架的纽带，成员 fb_info 指向了该 Driver 命名空间下所有打开的 FB 设备，成员 driver_info 则用来关联到某个 Driver 命名空间中。

对于 API 的设计主要分为如下 3 类：第一类，设计了一系列判断所给定的 fb_info 是否为虚拟 FB 结构，该结果用于 fbmem.c 中对于 FB 设备的不同处理；第二类，设计实现了用于 Driver 命名空间框架来注册设备的回调函数，此类回调函数用于将 FB 设备注册在 Driver 命名空间框架中、用于两个虚拟客户系统进行切换时找到所属的设备；第三类，设计当两个虚拟客户系统状态切换时的回调函数。当两个虚拟客户系统状态切换时，通过内核通知链机制通知在 Driver 命名空间框架中注册的显示设备触发回调函数，实现 active 与 inactive 状态的切换。

(3) 修改 fbmem.c 中对 FB 字符设备文件的 file_operations 函数操作集合。通过步骤 2 设计的第 1 类 API，在该文件的 fb_read、fb_write、fb_ioctl 和 fb_mmap 文件操作函数中引入判断虚拟 FB 与真实 FB 的逻辑。对于处于 active 状态虚拟客户系统中的进程操作虚拟 FB 设备时，fbmem.c 直接将该虚拟 FB 结构指向真实的 FB；而对于 inactive 的进程操作虚拟 FB 设备时，fbmem.c 只是做简单映射操作，从而实现了两个系统对 FB 设备的复用。

(4) 实现两个虚拟客户系统切换时显示子系统的正常显示。由于 FB 设备的工作原理是将该字符设备缓冲区的地址映射到进程的虚拟地址空间来实现应用程序对显存的操作，当两个系统发生状态切换时会发生复杂的映射操作。因此首先介绍一下 mmap 原理[58]。

Linux 提供 4G 字节的虚拟地址空间。1G 供内核使用，剩余 3G 供用户态的各个进程使用，称为用户空间。通过 mmap 系统调用，可以将一个 FB 字符设备文件映射到进程的虚拟地址空间中。该系统调用采用的是简洁的共享内存方式，不需要大量的数据拷贝。当成功执行该系统调用后会返回被映射区的指针，用户进程通过对这段虚拟地址空间的操作实现了对 FB 字符设备文件的读写。首先需要寻找一段连续的虚拟地址空间为映射做准备，通过建立 vm_area_struct 结构体用来描述一段连续的虚拟内存空间来传给内核中的 FB 字符设备文件使用。接着通过修改进程页表，由 fbmem.c 中的 fb_mmap 文件操作来建立虚拟地址空间 vma 和 FB 字符设备文件具体的物理地址之间的映射。该文件操作方法如下所示:int(*mmap)(struct file*, struct vm_area_struct)。对于不连续的物理页面则使用 remap_vmalloc_range(struct vm_area_struct *vma, void *addr,unsigned long pgoff)来实现了将物理内存与 vma 建立了映射关系，如图 5.7 所示：

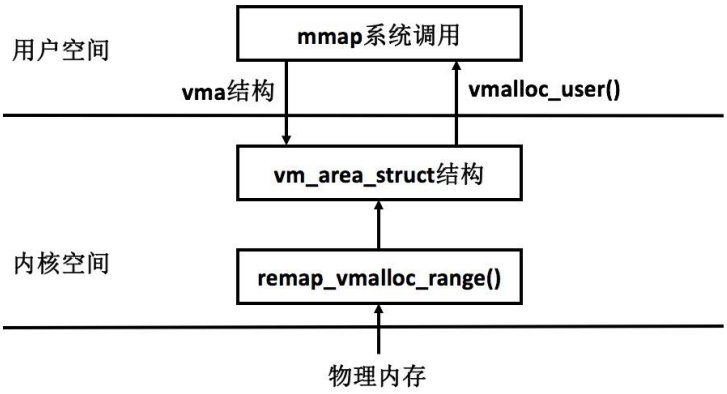

图5.7　进程虚拟地址空间与物理内存的映射关系

经过上述分析，在两个虚拟客户系统之间进行切换时，显示子系统的切换会发生如下 3 个事件：

①FB 显存缓冲区的重映射。

马上要变成 inactive 的虚拟客户系统中的所有进程需要将他们的虚拟地址重新映射到到虚拟 FB 显示缓冲区的物理地址中；而即将要运行在 active 虚拟客户系统的所有进程则将他们的虚拟地址空间重新映射到到真实的 FB 物理地址中去。首先，当前虚拟客户系统中可能存在多个进程的虚拟地址空间映射到 FB 的缓冲区物理地址中，通过 vma_prio_tree_foreach 函数来遍历出当前所有映射进程

的 vma 结构，并通过链表串联在 mmlist 结构中，该 mmlist 结构表示了当前所有映射到 FB 设备的进程。其次，通过逐个遍历 mmlist 中的每个 vma 成员，调用 zag_page_range 函数解除页面的映射，释放所映射的内存页面，然后通过调用 fb_mmap 函数将该 vma 进行重新映射。此处的 fb_mmap 函数会存在两种情况的调用规则：当 active 状态的进程则映射到真实 FB 缓冲区的物理地址中，反之则只映射到虚拟 FB 缓冲区的地址中。

②FB 显存缓冲区的深度拷贝。

通过把真实 FB 缓冲区的内容复制到即将转变为 inactive 状态虚拟客户系统的虚拟 FB 缓冲区，同时把即将转变为 active 状态的虚拟客户系统的 FB 缓冲区中的内容复制到真实 FB 缓冲区，实现了切换虚拟 FB 与真实 FB 时候内容的交换。每个 FB 设备的 fb_info 结构中维系着一个 screen_base 成员指向该 FB 设备的虚拟基地址，通过对 memcpy 函数灵活运用，往真实 FB 中拷贝缓冲区内容是通过 memcpy(info->screen_base, virtual->screen_base, size)；往虚拟 FB 中拷贝内容时则 memcpy(virtal->screen_base, info->screen_base, size)。其中 size 为 fb_info 中的 fix.smem_len 成员表示缓冲区的大小。

③硬件状态同步。

由于 Android 系统的显示设备会根据最新的硬件状态来将数据输出在屏幕中。硬件状态同步通过保存当前硬件状态到之前 active 的虚拟客户系统，再将当前硬件状态更新到现在 active 虚拟客户系统的硬件状态中去。该操作的实现很简单，只需要调用 fb_ops 中的 fb_save_state 接口即可。

5.4　输入子系统虚拟化实现

Linux 中输入事件在默认的情况下会被发送到所有监听该输入事件的进程中，但这个机制并不支持多个虚拟客户系统提供所需的隔离。为了让输入子系统使用 Driver 命名空间, 只需要修改输入事件处理程序即可。对于每个进程监听的输入事件，输入事件处理程序首先检查是否处在 active 状态下的 Driver 命名空间，如果是，则响应该输入事件；如果不是，则不唤醒特定的处理流程。不需要改变设备驱动程序以及输入核心层。evdev 是输入事件处理程序的核心组成部分，硬件设备通过输入核心层传给 evdev 处理程序后，该处理程序可以响应硬件设备并提供事件处理流程。输入子系统的虚拟化实现除了依托于 Driver 命名空间框架以外，还需要对 drivers/input/evdev.c 做相应的修改。具体修改如下：

(1) 结构体的增加如图 5.8 所示：

```
struct evdev_driver_ns {
        struct list_head clients;
        struct driver_info driver_info;
};
```

图5.8　输入子系统中增加的数据结构示意图

由第三章可知，evdev_driver_ns 结构包含了一个 driver_info 结构，是输入事件处理程序与 Driver 命名空间框架的纽带。每个在 Driver 命名空间框架中注册的输入设备都要定义这个结构，而当每次打开 evdev 设备都会创建一个 evdev_client 对象。所有同一个 Driver 命名空间下的 evdev_client 被串到该 Driver 命名空间中表示 evdev 结构 evdev_driver_ns 的成员 clients 中。

(2) 增加用来支持 Driver 命名空间框架的 API

Driver 命名空间框架中定义了若干函数模版、结构体模版，针对不同子系统的特性进行具体的实现；同时该框架中也注册着各个子系统的回调函数便于使用。在 evdev 事件处理程序的源码中增加了如下 API 来支持 Driver 命名空间框架。

①evdev_client_is_active(evdev_client)：该函数通过调用链 evdev_client 结构->evdev_driver_ns 结构->driver_info->driver_namespace->active 来判断出某个使用输入设备的 clients 结构是否处于 active 状态下的 Driver 命名空间。

②switch_callback()：当 active 与 inactive 状态发生切换时，Driver 命名空间框架利用内核通知链机制来对注册在该 Driver 命名空间中的输入设备进行通知，做出相应的处理。该函数是一个.notifier_call 类型的回调函数，当命名空间的状态发生切换时会触发该函数进行处理。当处理 ACTIVE 时，只需要把当前已 grab 输入事件的 client 响应输入事件即可；当处理 INACTIVE 时则需要将当前 evdev_driver_ns 串联的所有 clients 状态设置为 ungrab 即可。

③evdev_driver_ns_create(driver_namespace)：Driver 命名空间框架中为每个子系统定义一个 driver_ops 结构，该结构中的 create 成员在输入设备初始化时注册在 driver_global 结构中。进程打开一个输入设备时会检查所在 Driver 命名空间中是否已注册该设备，没有的话就调用该函数。该函数具有 2 个功能：创建 driver_info 结构、将 switch_callback 函数连接在 driver_info 的 notifier 成员中并将该回调函数注册在内核通知链中。

(3) 修改 evdev.c 现有的 API

首先在 evdev_grab()中引入 client->grab=true 的逻辑，同理在 evdev_ungrab 函数中加入 client->grab=false。当一个进程打开一个 evdev 设备文件的 evdev_open 函数中加入 evdev_track_client 调用关系。最后，在 evdev 设备初始化 evdev_init

中加入 REGISTER(evdev, "event driver")语句，该宏定义在 Driver 命名空间框架中定义，实质上是注册了一个 driver_global 全局结构。

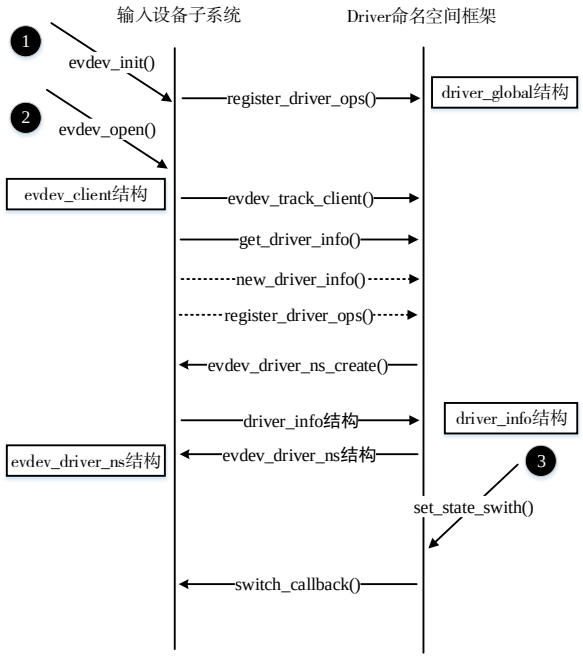

图5.9　输入设备子系统虚拟化实现原理图

介绍了对输入子系统所做的修改后，输入子系统虚拟化实现的完整流程如图 5.9 所示。首先，当输入设备在系统中初始化的时候子系统，通过调用 Driver 命名空间框架中的 register_driver_ops 接口将输入设备子系统注册在 driver_global 全局数组中，该注册过程主要将输入设备的 evdev_driver_ns_create 接口注册在框架中用于回调功能。此时输入设备仅仅是初始化完成，由于还没有进程真正的访问它，该子系统在 Driver 命名空间中相应的 driver_info 结构还没有创建。

其次，当某个进程打开输入设备子系统中的一个设备时 evdev_open 接口被调用。随后 evdev_track_client 被调用，每个进程使用输入设备就会生成一个 evdev_clients 结构，get_driver_info 函数中首先会检查使用设备进程所在的 Driver 命名空间中是否已注册该设备(是否已经存在 driver_info)。有的话就直接返回，否则调用 new_driver_info 新建一个 driver_info 结构体。由于 driver_info 包含在 evdev_driver_ns 结构中，因此调用输入子系统中的 evdev_driver_ns_create 来创建 evdev_driver_ns 结构，同时将切换处理函数注册在当前 Driver 命名空间的内核通知链上。创建成功后把 driver_info 结构返回给 Driver 命名空间框架，连接在该设备 driver_global 数组中的链表中；而 evdev_driver_ns 返回给输入设备子系统。

最后，当 active 状态与 inactive 状态的 Driver 命名空间进行切换时，通过在 /proc 文件中写数据触发 proc_active_ns_write 函数，最终 set_state_switch 接口被

调用。通过内核通知链处理函数 swtich_callback 先根据当前 Driver 命名空间找到相应的 evdev_driver_ns 结构，该结构上串连着多个使用输入设备的会话 clients。如果处理的是 ACTIVE 说明该 Driver 命名空间将由 inactive 即将变成 active 状态，如果该会话之前就是 grab 状态，就调用 evdev_grab 接口让其处理输入事件；如果处理 INACTIVE，表示该 Driver 命名空间将由 active 切换到 inactive，如果当前会话是真正的 grab 状态，则调用 evdev_ungrab 取消状态。由此对于输入设备子系统的虚拟化工作就完成了。

5.5　Backlight与LEDs子系统虚拟化实现

Android 系统中 Backlight 背光子系统通过 LCD 背光屏幕可以对亮度、电量以及饱和度进行调节设置，而背光子系统对于上述参数的调节还需要 LED 配合完成；LEDs 子系统除了为背光子系统提供驱动以外，自身还包含了智能设备指示灯的亮度调节，如短信提示灯、手电筒指示灯等。这两个子系统具有密切的联系，在实现了 Android 系统最核心的显示子系统和输入设备的隔离复用后，由于运用 Driver 命名空间框架来实现上述两个子系统的原理类似，因此本文对这两个子系统一起讨论。

表5.3　背光-LEDs-LCD子系统调节参数表

子系统名称	亮度 brightness	电量 power	对比度 contrast
Backlight	是	是	否
LCD	否	是	是
LEDs	是	是	否

表 5.3 显示了 Backlight 和 LEDs 子系统调节参数的权限，由 4.5 节介绍可知，用户空间中的某个应用程序需要对亮度调节时，首先进行背光子系统和 LEDs 子系统的初始化工作，该初始化会在用户空间创建 Backlights 与 LED 的设备节点。应用程序通过对设备文件的读写来响应内核空间相应的函数*show 和*store。对于 Backlight 背光和 LEDs 子系统，通过将 Driver 命名空间框架引入上述子系统来实现虚拟化遵循如下规则：(1)在 active 状态的虚拟客户系统和 inactive 状态的虚拟客户系统均可注册 LEDs 设备和 LCD 背光设备。(2)对于 LEDs 子系统而言，仅允许处于 active 状态的虚拟客户系统设置亮度、电量、对比度。(3)对于 Backlight 子系统而言，仅允许处于 active 状态的虚拟客户系统设置亮度与对比度。实现上述两个子系统虚拟化时需要修改如表 5.4 的三个文件：

表5. 4　背光-LEDS子系统需要修改的文件

修改文件名	修改函数名
backlight.c	backlight_store_power、backlight_store_brightness
lcd.c	lcd_store_power、lcd_store_contrast
led-class.c	led_brightness_store、led_power_store

在上表的三个文件中只需要在每个函数中增加相应的 Driver 命名空间逻辑(如图 5.10 所示)，即可实现对 Backlight 和 LEDs 子系统的虚拟化。

```
if (!is_active_driver_ns(current_driver_ns())) {
    printk("%s: not setting %s power to %ld from inactive namespace\n",
                                                __func__, driver_name(dev), power);
    error = -ENXIO;
    return error;
};
```

图5.10　Backlights-LEDs子系统核心函数修改记录

5.6　性能优化

本文使用如下两个技术来实现两个虚拟客户系统之间、虚拟客户系统与宿主系统之间隔离。第一，通过 MNT 命名空间为两个虚拟客户系统提供独立的文件系统视图，两个虚拟客户系统互相不能看到并访问文件系统，只有宿主系统可以管理完整的文件系统。第二，通过在内核中引入 Driver 命名空间框架，两个虚拟客户系统分别位于不同的 Driver 命名空间中，实现了不同虚拟客户系统对硬件设备的隔离访问与复用。

由于宿主系统与两个虚拟客户系统共享内存，本文使用 LMK(Low Memory Killer)机制[59]来保证两个虚拟客户系统同时启动时整个系统内存的正常使用。LMK 可以杀死 inactve 虚拟客户系统中大量消耗内存的进程。Android 启动这些进程纯粹作为一种优化来减少应用程序的启动时间，所以这些进程可以被杀死并重新启动，没有任何损失的功能。核心进程是不会选择死亡的，即使用户需要的核心进程服务被杀死也会重新启动。

用户在使用 Android 设备时，如果仅仅通过普通的退出按键来跳出当前的应用程序，这种方法并不能够真正在系统中退出该应用程序。随着运行时间的增加，应用程序的开启数量也会逐渐增加，系统的内存会变得不足，这就需要杀掉一部分进程以释放内存空间。LMK 机制会在系统内存不足的情况下，选择一个进程并将其 kill 掉，该机制实质是利用内核中 OOM(Out Of Memory)来完成这个任务，至于是否需要杀死一些进程和哪些进程需要被杀死，则由 LMK 判断。

```
static int lowmem_adj[4] =
{
        0,
        1,
        6,
        12
};
```

```
static size_t lowmem_minfree[4] =
{
        3*512, //6MB
        2*1024, //8MB
        4*1024, //16MB
        16*1024 //64MB
};
```

图5.11　lowmem_adj和lowmem_minfress结构体示意图

如图 5.11 所示，LMK 通过在 adj 文件中配置 lowmem_adj 结构体和在 minfree 文件中配置 lowmem_minfree 结构体来描述进程使用内存的临界值。其中 lowmem_adj 指定了进程的临界值优先级 oom_adj，lowmem_minfree 则指定了当前系统储存空闲页面的数量。这两个结构体中的数据是一一对应的关系，即当系统中空闲页面的数量下降到 3*512 个页面时，oom_adj 大于等于 0 的进程将被杀死，以此类推。在内核空间通 task_struct->signal_struct->oom_adj 来指定优先级的数值，在用户名空间/proc/PID/oom_obj 目录下会纪录该数值，内核规定此参数最小为-17(通常不设为此值)，最大为 15。当内存中的空闲页面的数量不足时，内核会向数值大的进程发送 SIGKILL 信号将其杀死。本文将 init 进程的 oom_adj 设为-16 并在 init.rc 文件中修改(如下描述)，可以根据需要定制需要杀死的进程。

```
on init
    # set init and its oom_adj.
    write /proc/1/oom_adj -16
```

5.7　实验与分析

Android 系统虚拟化的目标就是为了实现在宿主系统上同时构造出多个虚拟客户系统实例，各个系统之间互相隔离、不能访问并操作其他虚拟客户系统的文件系统与数据，同时也要尽可能的减少虚拟化所带来的系统开销问题。本文在功能测试上对宿主系统启动、虚拟客户系统启动、系统间正常切换、Android 基本特性以及系统间隔离性等进行了完整的测试工作；在性能测试上通过对 CPU 利用率以及内存使用度两个参数来衡量虚拟化的引入所带来的性能开销。

5.7.1　测试环境部署

作为移动设备的主流操作系统，Android 系统主要运行在 ARM 平台，本文基于 AC8317 ARM 开发板对该虚拟化方案进行测试，硬件平台测试环境如表 5.5 所示：

表5.5　测试平台参数

内容	类别	参数
硬件	CPU	ARM Cortex A7 dual-core
	内存	DRAM 1GB
	存储	eMMC 8GB
操作系统	内核版本	Linux kernel 3.4
	Android	AOSP Android 4.2.2

操作系统选用从 AOSP(Andorid Open Source Project)下载的较为稳定、特性较为全面的 Android4.2.2 版本，对应内核版本为 kernel 3.4 进行测试。采用 AC8317 ARM 开发板配套的 armv6z-mediatek-linux-gnueabi-gcc 交叉编译工具，通过如表 5.6 中的步骤将 Android 源码生成可烧写到板子上的 Image 镜像文件。

表5.6　交叉编译Android源码步骤

步骤	描述
source ./selfenv	设置编译环境
./selfbuild uboot	编译 uboot
./selfbuild driver	编译 AC8317 的 driver
./selfbuild kernel	编译 linux 内核
./selfbuild arm2	编译 arm2
./selfbuild android	编译 android
./selfbuild image	生成最后需要烧写到板子的 Image

将生成的 Image 文件烧写到 AC8317 开发板，经过串口连接到 ARM 开发板中，利用 Android 提供的 adb 工具来连接到宿主系统的 shell 中运行相关的命令。

5.7.2　功能测试

功能测试是对于该操作系统虚拟化方案重要的部分，保证了最基本的功能完整性，在功能测试上从如下 5 个方面进行测试。

(1) 宿主系统启动测试

宿主 Android 系统在初始启动的时候构建出两个虚拟客户系统，同时用来管理两个虚拟客户系统的切换。通过对 init.rc 文件的修改，宿主系统初始运行着一个最原始的 Android 系统，用来生成文件系统、生成各类根命名空间等。通过 adb 工具连接到系统的命令行可以显示宿主系统的文件系统视图，Android 系统的文

件系统如图 5.12 所示：

图5.12　宿主Android系统文件视图

同时通过 readlink /proc/1/ns/可以查询到当前根系统创建的各类根命名空间，如图 5.13 所示。

```
root@android: ~ # readlink /proc/1/ns/
uts: [4026531833]
mnt: [4026531836]
pid: [4026531837]
ipc: [4026531839]
```

图5.13　各类根命名空间信息图

(2) 虚拟客户系统启动测试

MNT 命名空间与 PID 命名空间为两个虚拟客户系统提供了独立的文件系统与进程视图，每个虚拟客户系统运行着一个完整而独立的 Android 环境，虚拟客户系统之间不允许访问，从宿主系统角度可以看到根命名空间下的所有各个子命名空间视图。通过 pstree 工具可以查询到两个虚拟客户系统的 PID 视图，图 5.14 显示了查询 init 进程的关系树。

```
root@android: ~ # pstree -pl | grep init
|-sshd(955)-sshd(17810)-sshd(17930)-init(1)-init(13331)-init(14212)-pstree(13667)
```

图5.14　pstree工具查询init进程树关系图

为了测试需要，通过在虚拟客户系统中重新挂载/proc 目录可以看到每个虚拟客户系统中的完整进程视图。由此可以判断虚拟系统已经完全启动。

(3) 虚拟客户系统切换测试

当宿主系统与两个虚拟客户系统都启动成功时，当前显示在屏幕的是 active 状态下的虚拟客户系统，而 inactive 状态的客户系统则运行在后台。在每个虚拟客户系统中安装了一个负责切换的应用程序，当运行该应用程序时会触发/proc 文件目录的写事件，通过调用 Driver 命名空间框架中的切换回调函数，实现了不同虚拟客户系统之间的正常切换。

(4) Android 特性测试

根据 Sanity Test 特性测试表格从 Android 特性上对本虚拟化实现方案进行测试，如表 5.7 所示：

表5.7　Android Sanity Test特性测试表

序号	特性	总次数	通过次数	失败次数	N/A	描述
1	Touch panel	10	10	0	0	触摸屏功能正常
2	Display	10	10	0	0	显示功能正常
3	Kernel	20	20	0	0	内核启动成功
4	SD	10	10	0	0	SD 存储正常
5	USB	10	10	0	0	USB 正常
6	APP	10	10	0	0	应用程序
7	Image	10	10	0	0	图片显示正常
8	Brightness	10	10	0	0	亮度功能正常
9	Contrast	10	10	0	0	饱和度功能正常

(5) 隔离性测试

为了实现对两个虚拟客户系统隔离性的验证，采用文件系统读写和进程列表两方面来测试。在 A 虚拟客户系统中创建记事本文件 A.txt，并运行闹钟、播放器 APP 后，切换至 B 虚拟客户系统中同样的方法创建记事本文件 B.txt 并运行指南针、音乐播放器。A 系统中看不到 B.txt，且通过 ps 命令查看不到 B 系统中运行的进程列表；同理 B 系统中也看不到 A 系统的所有信息，只有通过宿主系统才可查看，从而完成了隔离性测试。

在宿主系统上运行两个虚拟客户系统，它们共享内核、对硬件设备复用，通过上述 5 个方面的测试得知本方案在保障隔离性的同时，Android 系统基本特性也能够运行。

5.7.3　性能测试

在实现 Android 操作系统虚拟化方案时通过 LMK 机制对内存使用情况上做了简单的优化，虽然操作系统级虚拟化相比于传统 Hypervisor 架构减少了 VMM 的管理开销，但随着两个虚拟客户系统的同时运行，需要从 CPU 使用率和内存使用量两方面来分析虚拟化的引入所带来的性能变化。

(1) CPU 使用率分析

分别在非虚拟化的 Android 系统和虚拟化的系统中安装“videos”应用软件，

然后统计从启动到播放视频的过程中 CPU 使用率的问题，通过命令 top –d 1|grep videos 每隔 1 秒获取 CPU 使用率情况。

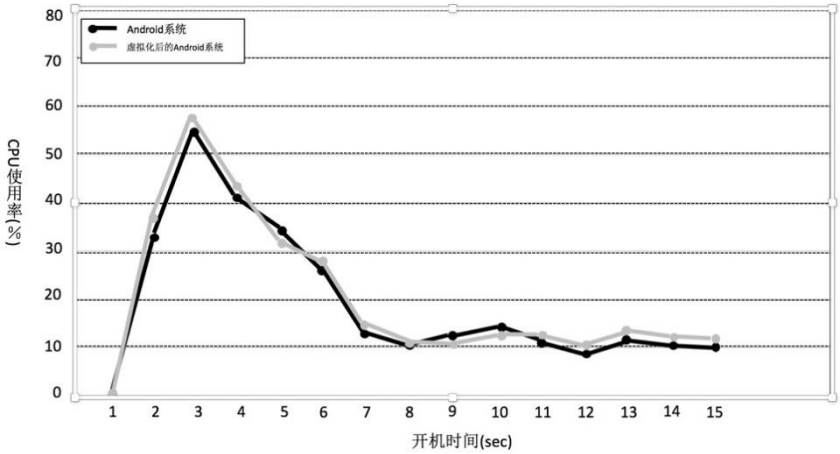

图5.15　videos应用软件CPU使用率示意图

从图 5.15 中可以看出，对于 videos 应用程序从启动初期到完全启动状态，虚拟化后的 Android 系统和原生 Android 系统并无明显差别，平均有 0.3%的性能差异。按照同样的方法，分别对 camera、photos、clock、calculator 等系统软件的 CPU 使用率进行测试，测试结果如表 5.8 所示。

表5.8　不同应用软件CPU使用率变化表

应用软件	完全启动耗时(s)	CPU 使用率差异(%)
Videos	15	0.3%
Camera	18	0.1%
Photos	13	0.2%
Clock	8	0.3%
Calculator	9	0.1%

表 5.8 的测试结果此虚拟化架构对于 Android 系统软件启动过程的 CPU 使用率有 0.2%的差异，说明虚拟化的引入并没有带来大量额外的 CPU 开销。

(2) 内存使用情况分析

在内存使用情况中，通过统计只启动宿主系统、启动宿主系统和一个虚拟客户系统、启动宿主系统和两个虚拟客户系统三种情况下的内存使用情况。同时在时间上从系统启动初期、启动 15min、启动 30min、启动 45min 分别对上述三种情况进行了统计，如图 5.16 所示：

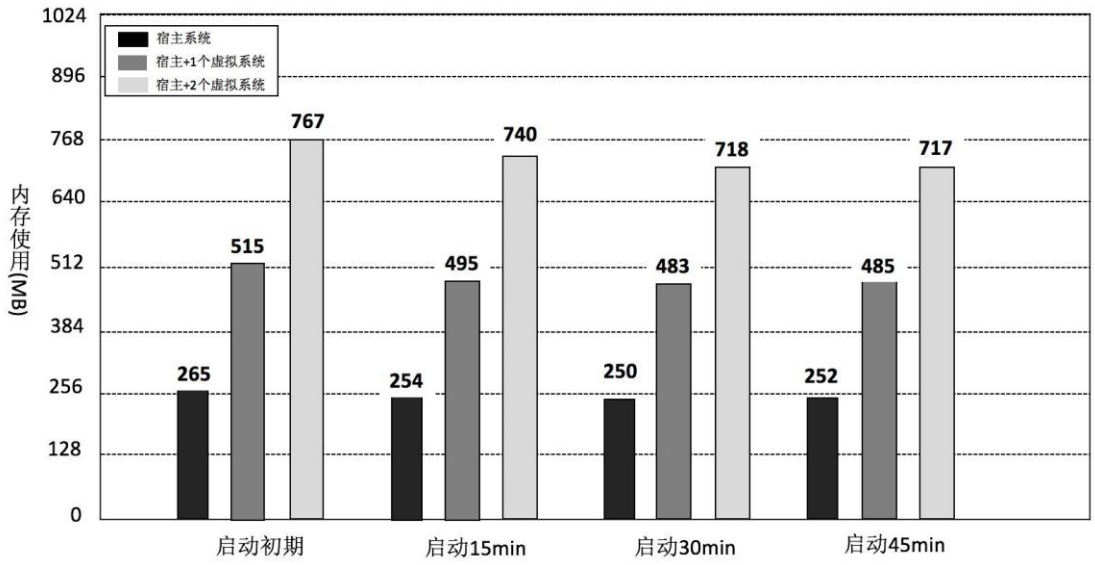

图5.16　内存使用量测试结果

统计结果表明，随着启动时间的持续，由于 LMK 内存优化的原因，虚拟客户系统所带来的内存开销逐渐减少。从启动 30min 后的稳定状态得出，由于共享内核的缘故，每个虚拟客户系统所带来的内存开销比原生系统约减少了 6.7%。

5.8　本章小结

本章详细阐述了本文提出的轻量级 Android 操作系统虚拟化方案的实现原理和验证实验。首先定制 Linux 内核，开启命名空间的编译选项；同时修改 Kconfig 与 Makefile 文件将 Driver 命名空间添加进内核。其次为宿主系统的 init 进程创建各类子命名空间，提供隔离的 MNT、PID、Driver 视图，各类子命名空间的集合形成了两个独立完整的虚拟客户系统运行环境。利用 Driver 命名空间实现了不同虚拟系统对显示设备、输入设备、背光 LEDs 设备的隔离复用，完成了 Android 操作系统级虚拟化方案。最后，利用 LMK 机制对该虚拟化方案进行了简单的内存使用优化。在实验部分，从宿主系统启动、虚拟客户系统启动、系统间正常切换、Android 基本特性以及系统间隔离性等进行了完整的功能测试工作。在性能测试上通过 CPU 使用率和内存使用度两个参数来衡量虚拟化的引入所带来的性能开销。实验表明随着启动时间的持续，由于 LMK 的优化，虚拟客户系统所带来的系统开销逐渐减少。增加每个虚拟客户系统所带来的内存开销小于原生系统，由于共享内核的缘故，每个虚拟客户系统所带来的内存开销比原生系统约减少了 6.7%；而 CPU 使用率与原生系统基本持平。

第六章 总结和展望

6.1 本文总结

随着当前移动终端设备的飞速发展，为了满足使用场景的多样性带来以及多用户隐私访问等问题，虚拟化技术在移动设备上的需求也在逐渐增加。移动平台上虚拟化的相关研究比较匮乏，传统的 Hypervisor 架构存在一定性能损耗，而移动平台各类硬件高度集成，对性能要求严苛。为了解决此问题，本文基于 ARM 平台，将 Android 丰富的应用程序、多样化硬件设备的特点与虚拟化技术相结合，在宿主 Android 系统上构造出两个完整的虚拟客户系统实例。利用 MNT 和 PID 命名空间为每个虚拟客户系统提供一个隔离的文件系统视图与进程信息视图，同时通过在内核中引入 Driver 命名空间框架使得虚拟客户系统间对硬件设备隔离复用，实现了一种操作系统级移动虚拟化的新方案。

本文的主要工作和贡献如下：

(1) 分析各类虚拟化技术解决方案的不足并提出一种改进方案。分析了各类虚拟化技术的优点与不足，并深入研究了操作系统级虚拟化研究领域较为成熟的解决方案。结合研究背景与项目需求，提出一种将操作系统级虚拟化技术的优势运用在移动 ARM 平台上的改进方案。

(2) 研究 Linux 命名空间资源隔离机制并提出 Driver 命名空间框架。现有内核提供了一种轻量级命名空间资源隔离机制，本文重点研究了实现操作系统虚拟化解决方案过程中所用到的 MNT 和 PID 命名空间，研究了这两种命名空间机制用户空间的使用方法和内核中的实现原理。同时在内核中设计了 Driver 命名空间框架，该框架为每个虚拟客户系统提供一个隔离的硬件设备使用视图，并实现 active-inactive 模型，允许仅当 active 的虚拟客户系统中的进程对硬件设备操作是有效的，保证了多个虚拟客户系统对一套硬件设备的隔离复用。

(3) 设计基于 ARM 平台的 Android 操作系统虚拟化架构。基于 MNT、PID、Driver 等各类命名空间设计了基于宿主系统构造出两个虚拟客户系统的整体架构。由于宿主系统与虚拟客户系统共享一套硬件设备，利用本文设计的 Driver 命名空间框架来实现对设备的隔离复用。在共享单个内存、CPU 资源的前提下，深入研究了 Android 系统中各个硬件设备子系统，并针对不同子系统对于隔离复用的需求设计了包括显示设备子系统、输入设备子系统、背光设备子系统以及 LEDs 设备子系统等在内的最核心硬件设备的虚拟化解决方案。

(4) 实现 Android 操作系统虚拟化解决方案。本文实现的操作系统级虚拟化方案主要由两部分组成：系统级启动、硬件设备复用。首先修改内核编译选项定制支持各类命名空间功能的内核。接着在保证宿主系统正常启动的基础上，通过配置 init.rc 文件来构造出两个虚拟客户系统，其中一个为 active 状态另一个为 inactive 状态，由宿主系统进行管理和状态的切换。最后利用 Driver 命名空间框架，在系统完全启动的基础上，实现了显示设备、输入设备、背光和 LEDs 设备的隔离复用，完成了一种轻量级的 Android 操作系统虚拟化解决方案。

(5) 实验和分析。通过搭建实验测试平台，选择 AC8317 Cortex Dual-Core 开发板，在功能测试上对宿主系统启动、虚拟客户系统启动、系统间正常切换、系统间隔离性等进行了完整的测试工作，测试结果显示本方案基本满足所有的 Android 特性；在性能测试上从 CPU 使用率、内存使用度两个参数衡量虚拟化的引入带来的性能开销。内存使用量通过统计只启动宿主系统、宿主系统和一个虚拟系统、宿主系统和两个虚拟系统三种情况，并在每种情况中从启动初期、启动 15min、启动 30min、启动 45min 进一步统计。实验表明由于 LMK 的优化，虚拟客户系统所带来的系统开销逐渐减少。增加每个虚拟客户系统所带来的内存开销小于原生系统，由于共享内核的缘故，每个虚拟客户系统所带来的内存开销比原生系统约减少了 6.7%；而 CPU 使用率与原生系统基本一致。

6.2　下一步工作与展望

本文提出了一种轻量级的移动平台 Android 操作系统级虚拟化方案，但该方案仍有一些方向可以继续研究，未来关于本文的研究工作可以从以下展开：

(1) 本文虽然实现了显示设备、输入设备、背光设备和 LEDs 等 Android 系统最核心的硬件设备的虚拟化隔离复用工作，但仍有一些闭源设备，如 Wi-Fi 模块、电话拨打模块等未完成虚拟客户系统间的隔离复用问题，未来的工作重点可以放在继续实现上述硬件设备的隔离复用机制上，实现一套完整的移动智能设备虚拟化方案。

(2) 虽然操作系统级虚拟化具有轻量级、虚拟化带来的资源开销少等优势，但和传统的 Hypervisor 架构相比，由于宿主系统和虚拟客户系统共用内核，因此数据安全性上做的还不够好。

(3) 本文在实现该虚拟化解决方案的过程中仅仅通过 LMK 机制对内存的使用做了简单的优化工作，未来为了能在宿主系统中构造出更多虚拟实例，可以通过共享那些不影响隔离复用的硬件设备来实现内存的优化。

参考文献

[1] Xu Lei , Li Guoxi, Li Chuan, et al. Condroid: A Container-Based Virtualization Solution Adapted for Android Devices[C]// 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud 2015), San Francisco, CA,2015:81-88.

[2] Kolin Paul, Tapas Kumar Kundu. Android on Mobile Devices: An Energy Perspective[C]//Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on. 2010:2421-2426.

[3] 文雨，孟丹，詹剑锋. 面向应用服务级目标的虚拟化资源管理[J]. 软件学报，2013(02):358-377.

[4] Margaret Butler. Android: Changing the Mobile Landscape[J]. IEEE Pervasive Computing, 2011(1):4-7.

[5] Kang J, Hu C, Wo T, et al. MultiLanes: Providing Virtualized Storage for OS-Level Virtualization on Manycores[J]. ACM Transactions on Storage (TOS), 2016, 12(3): 12.

[6] IBM虚拟化与云计算小组. 虚拟化与云计算[M]. 北京. 电子工业出版社. 2010:26-54.

[7] Kumar K, Kurhekar M. Economically Efficient Virtualization over Cloud Using Docker Containers[C]//Cloud Computing in Emerging Markets (CCEM), 2016 IEEE International Conference on. IEEE, 2016: 95-100.

[8] Boyd-Wickizer S, Clements A T, Mao Y, et al. An Analysis of Linux Scalability to Many Cores[C]//OSDI. 2010, 10(13): 86-93.

[9] Vaughan-Nichols S J. Virtualization sparks security concerns[J]. Computer, 2008, 41(8): 13-15.

[10] Detken K.-O., Oberle A., Kuntze N., et al. Simulation environment for mobile virtualized security appliances[C]// 2012 IEEE 1st International Symposium on Wireless Systems within the Conferences on Intelligent Data Acquisition and Advanced Computing Systems (IDAACS-SWS 2012), Offenburg, Germany, 2012:113-18.

[11] Xu L, Chen W, Wang Z. Research about virtualization of ARM-based mobile smart devices[M]//Multimedia and Ubiquitous Engineering. Springer Berlin Heidelberg, 2014: 259-266.

[12] 汪志刚. 复合桌面虚拟化技术研究[D]. 上海:上海交通大学, 2010.

[13] Dall Christoffer, Nieh, Jason. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor[J]. ACM SIGPLAN NOTICES, 2014, 49(4):333-347.

[14] 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.

[15] 赵杭君. 基于OKL4的RTEMS虚拟化及其应用[D]. 大连:大连理工大学, 2012.

[16] Kang H Y. Study and Implementation of Virtualization Technology in Embedded Domain Using OKL4[J]. Advances in Future Computer and Control Systems, 2012, 1: 221.

[17] Sorceforge. Infrastructure for container projests[EB/OL]. (2016-12-12). https://linuxcontainers.org

[18] Tihfon G M, Park S, Kim J, et al. An efficient multi-task PaaS cloud infrastructure based on docker and AWS ECS for application deployment[J]. Cluster Computing, 2016, 19(3): 1585-1597.

[19] 吴佳杰. 基于LXC的Android系统虚拟化的关键技术设计与实现. [D]. 杭州:浙江大学, 2014.

[20] Chen Wenzhi, Xu Lei, Li Guoxi, et al. A Lightweight Virtualization Solution for Android Devices[J]. IEEE Transactions on Computers, 2015, 64(10):2741-2751.

[21] Chen X. Smartphone virtualization: Status and challenges[C]//Electronics, Communications and Control (ICECC), 2011 International Conference on. IEEE, 2011: 2834-2839.

[22] Dall C, Andrus J, Van't Hof A, et al. The design, implementation, and evaluation of cells: A virtual smartphone architecture[J]. ACM Transactions on Computer Systems (TOCS), 2012, 30(3): 9.

[23] 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.

[24] 姚远. Pcanel/V2-基于Intel VT-x技术的VMM架构[D]. 杭州:浙江大学, 2007.

[25] 孙鹏. 基于虚拟化技术的智能手机移动办公室的研究与应用[D]. 上海:复旦大学, 2010.

[26] 周刚. 云计算环境中面向取证的现场迁移技术研究[D]. 武汉:华中科技大学, 2011.

[27] Liang H, Li M, Xu J, et al. vmOS: A virtualization-based, secure desktop system[J]. Computers & Security, 2016(65):329-43.

[28] 周睿. 面向安全关键的虚拟化与分区操作系统研究与实现[D]. 兰州:兰州大学, 2010.

[29] Cilardo A, Gallo L. Improving multibank memory access parallelism with lattice-based partitioning[J]. ACM Transactions on Architecture and Code Optimization (TACO), 2015, 11(4): 45.

[30] Spink T, Wagstaff H, Franke B. Hardware-Accelerated Cross-Architecture Full-System Virtualization[J]. ACM Transactions on Architecture and Code Optimization (TACO), 2016,

13(4): 36.

[31] Liu J, Huang W, Abali B, et al. High Performance VMM-Bypass I/O in Virtual Machines[C]//USENIX Annual Technical Conference, General Track. 2006: 29-42.

[32] 陈晓. 基于LinuxContainer的移动终端虚拟化[D].广州:华南理工大学, 2013.

[33] 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.

[34] 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.

[35] Rossier D. EmbeddedXEN: A Revisited Architecture of the XEN hypervisor to support ARM-based embedded virtualization[J]. White paper, Switzerland, 2012.

[36] Xen Project. Architecture for split drivers within Xen[EB/OL]. 2011. http: //wiki.xensource.com/xenwiki/XenSplitDrivers.

[37] Barlev S, Basil Z, Kohanim S, et al. Secure yet usable: Protecting servers and Linux containers[J]. IBM Journal of Research and Development, 2016, 60(4): 12: 1-12: 10.

[38] AOSP community. Android open source projest[EB/OL]. (2012). https://source.android.com.

[39] 侯竑炤. 基于 Android 的车载物联信息终端研究与设计[D]. 复旦大学, 2012.

[40] Tihfon G.M., Jinsul Kim, Kim K.J. A New Virtualized Environment for Application Deployment Based on Docker and AWS[C]// International Conference on Information Science and Applications (ICISA) 2016. LNEE 376, 2016:1339-49.

[41] Michael Kerrisk. clone()-Linux Programmer's Manual[EB/OL]. (2016-12-12).http://man7.org/linux/man-pages/man2/clone.2. html.

[42] Tao L, Huang P. Automatic software install/update for embedded Linux[J]. Journal of Shanghai Jiaotong University (Science), 2008, 13(1): 107-109.

[43] Bacis E, Mutti S, Capelli S, et al. DockerPolicyModules: mandatory access control for docker containers[C]//Communications and Network Security (CNS), 2015 IEEE Conference on. IEEE, 2015: 749-750.

[44] Rathore M S, Hidell M, Sjodin P. PC-based router virtualization with hardware support[C]//Advanced Information Networking and Applications (AINA), 2012 IEEE 26th International Conference on. IEEE, 2012: 573-580.

[45] 徐贤仲. 专用隔离系统的设计与实现[D]. 北京邮电大学, 2014.

[46] 林杰. 面向服务监控的可控云关键技术研究[D]. 北京邮电大学, 2015.

[47] Thangaraju D B. Linux signals for the application programmer[J]. Linux Journal, 2003,

2003(107): 6.

[48] Rosen R. Linux containers and the future cloud[J]. Linux J, 2014, 2014(240).

[49] 谢婉君, 贾濡. Linux 内核移动性支持机制与实现[J]. 计算机技术与发展, 2015, 25(3): 103-107.

[50] Socala Arkadiusz, Cohen Michael. Automatic profile generation for live Linux Memory analysis[J]. Digital Investigation, 2016(16):S11-S24.

[51] Nagahara Y, Oyama H, Azumi T, et al. Distributed intent: Android framework for networked devices operation[C]//Computational Science and Engineering (CSE), 2013 IEEE 16th International Conference on. IEEE, 2013: 651-658.

[52] Jusung Kim, Oh-Chul Kwon, Chang-Gun Lee. Development of frame buffer structure for automatic dynamic resolution switching on Android mobile platform[J]. Journal of KISS: Computing Practices, 2010, 16(12):1209-13.

[53] Zhejun Fang, Qixu Liu, Yuqing Zhang, et al. A static technique for detecting input validation vulnerabilities in Android[J]. Science China Information Sciences, 2017, 60(5):052111(16 pp.).

[54] Carroll A, Heiser G. An Analysis of Power Consumption in a Smartphone[C]//USENIX annual technical conference. 2010, 14: 21-21.

[55] Buffenbarger J. Adding automatic dependency processing to makefile-based build systems with amake[C]//Proceedings of the 1st International Workshop on Release Engineering. IEEE Press, 2013: 1-4.

[56] Shabtai A, Fledel Y, Elovici Y. Securing Android-powered mobile devices using SELinux[J]. IEEE Security & Privacy, 2010, 8(3): 36-44.

[57] Yoon C, Kim D, Jung W, et al. AppScope: Application Energy Metering Framework for Android Smartphone Using Kernel Activity Monitoring[C]//USENIX Annual Technical Conference. 2012, 12: 1-14.

[58] Tianhua L, Hongfeng Z, Guiran C, et al. The design and implementation of zero-copy for linux[C]//Intelligent Systems Design and Applications, 2008. ISDA'08. Eighth International Conference on. IEEE, 2008, 1: 121-126.

[59] Singh Atikant, Agrawal Aakriti V., Kanukotla Anuradha. A Method to Improve Application Launch Performance in Android Devices[C]//2016 International Conference on Internet of Things and Applications (IOTA), 2016:112-115.

致　谢

在科大度过了难忘的研究生生涯，这三年，相比本科阶段我更专注的学习并研究了专业知识，培养了良好的做研究的态度，并找到了自己的兴趣点；这三年，我遇到了很多良师益友，科大的老师都具有很好的专业背景，并能给我耐心的答疑解惑，科大的同学们都能在生活与学习上以真诚的态度来交往；这三年，我感受到了父母对我的爱，也同时收获了爱情。感谢中国科学技术大学对我这三年的培养，让我能够严格要求自己，为走出象牙塔做好了准备。

首先要感谢我的导师顾乃杰教授。顾老师严谨的科研态度潜移默化的影响着我，让我每次在遇到一个棘手的难题时，从确定研究方向、理清思路与逻辑、优化现有研究成果等步骤指导我进行科学探索，受益匪浅。同时，顾老师在算法理论上具有扎实的学术背景，能给予我耐心且中肯的制导，让我能够系统的了解并钻研专业知识，跟踪本领域最新的研究成果，激励着我不断进步，不断学习。

感谢实验室的张旭、杜云开、曹华雄、张明、林传文、张孝慈、黄增士、钟旭东、江国荐、潘涛、冯光辉、王小乐、王少萍、陈露、王裕民、刘思婷、周文博等师兄师姐对我的指导和帮助，让我在学习过程中避免了很多弯路。感谢谷德贺、苏俊杰、郑晓松、郝建林、王岩、李燚、陈悟、王小强等同学，在一起做项目、研读论文的过程中能够相互讨论，勇于阐述出自己鲜明的观点，互相激励，共同进步。感谢实验室的陈思润、丁世举、朱方祥等师弟，在我写论文期间能够提出建设性的意见，督促我精益求精。实验室大家庭营造了一个良好的学习氛围。

感谢在科大的三年中所有向我传授过知识的老师们。感谢许胤龙、郑启龙、田野、张信明、黄刘生、熊焰、李京、岳丽华、曾凡平等老师，他们一流的教学水平让我学到了大量的专业知识。感谢班主任李胜柏老师，在日常生活和学业要求上都对我和蔼的关心和指导。

感谢我的室友张必红、杨金星、刘杰、许杨、刘阵、马顶，在学习上能够与你们一起刷题、相互督促进步、交流学习心得；在业余生活上也为三年的研究生生活增添了很多乐趣，留下了美好的回忆。

感谢我的父母和家人，感谢他们从我出生到现在的养育之恩，感谢他们在我每个人生阶段的坚定支持，感谢他们能陪伴着我度过近二十年的求学生涯。

感谢我的女朋友，虽然她在遥远的新加坡，但能够默默的在身后理解和支持我，分享我的快乐也能分担我的低谷，她的出现是我不断奋斗的动力。

最后，再一次发自内心的感谢关心和帮助过我的每一个人。

在读期间发表的学术论文与取得的研究成果

研究生期间参与的科研项目：

1. ARM 平台双操作系统项目，杰发科技公司

研究生期间论文发表情况：

1. **Bowen Liu,** Naijie Gu, Dehe Gu. A lightweight OS-level virtualization architecture based on Android.2017 2nd International Conference on Computer, Network Security and Communication Engineering(CNSCE 2017), 2017. **(EI)**

2. **刘博文,** 顾乃杰, 谷德贺, 苏俊杰. 移动平台 Android 操作系统虚拟化技术的实现[J]. 计算机工程与应用，**已录用**.