

分类号

密级

UDC

编号

成都理工大学

硕士学位论文

题名和副题名 基于 Linux 的虚拟化技术研究和应用

作者姓名 李绍

指导教师姓名及职称 罗省贤 教授

申请学位级别 硕士 专业名称 计算机软件与理论

论文提交日期 2011.4 论文答辩日期 2011.5

学位授予单位和日期 成都理工大学 (年 月)

答辩委员会主席 李旭瑞

评阅人 李旭瑞 董世杭

2011 年 5 月

分类号_____

学校代码: 10616

UDC _____

密级_____ 学号: 2008020450

成都理工大学硕士学位论文

基于 Linux 的虚拟化技术研究和应用

李绍

指导教师姓名及职称 _____ 罗省贤 教授

申请学位级别 _____ 硕士 _____ 专业名称 _____ 计算机软件与理论

论文提交日期 _____ 2011.4 _____ 论文答辩日期 _____ 2011.5

学位授予单位和日期 _____ 成 都 理 工 大 学 (_____ 年 _____ 月)

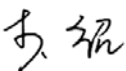
答辩委员会主席 _____

评阅人 _____

2011 年 5 月

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，也不包含为获得成都理工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论文中作了明确的说明并表示谢意。

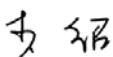
学位论文作者签名： 


2011年 5 月 26 日

学位论文版权使用授权书

本学位论文作者完全了解成都理工大学有关保留、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。本人授权成都理工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： 

学位论文作者导师签名： 

2011 年 5 月 26 日

基于 Linux 的虚拟化技术研究及应用

作者简介：李绍，男，1984 年 10 月出生，师从成都理工大学罗省贤教授，2011 年 7 月毕业于成都理工大学计算机软件与理论专业，获得工学硕士学位。

摘 要

虚拟化技术从 20 世纪 60 年代诞生到现在，经历了几十年的低潮后，在 90 年代随着计算机技术的发展再一次发展起来。操作系统领域一直以来面临的一个挑战来自于应用程序间存在的相互独立性和资源互操作性之间的矛盾，即每个应用程序都希望能够运行在一个相对独立的系统环境下，不受其他应用程序的干扰，同时又能够以方便有效的方式和其他应用程序共享和交换系统资源。传统的计算机操作系统缺乏对程序间独立性的有效支持，因此发展了虚拟化技术来解决这些问题，达到同一个计算机平台上运行多个相互独立的操作系统的目标。

虚拟技术在最近几年得到了飞速发展，能够有效地提高处理器的利用率，并且处理器厂商相继推出了硬件支持虚拟化的处理器、芯片组等。现在虚拟化技术正逐步在系统软件中得到广泛应用。

本文首先研究了计算机虚拟化技术的基本理论和机制，从虚拟化技术分类出发，研究完全虚拟化实现原理，以及全虚拟化中处理器虚拟化、内存虚拟化的实现技术；然后以基于硬件辅助虚拟化技术为目标，研究虚拟机 VMX 操作模式，并对基于 Linux 的 KVM 虚拟化方案进行深入探讨。

在研究和总结基于 Linux 的 KVM 虚拟机实现原理与关键技术基础之上，本文着重研究了内存虚拟化技术以及基于 Intel VT-x 虚拟化技术的 EPT 机制，结合 EPT 页表地址转换方法，在特定需求下，提出并实现了一种内存虚拟化中 EPT 机制的优化方案，并将该优化方案应用到实际项目中。本文在研究过程中，分别对 Linux 分页地址转换方法和虚拟机 EPT 页表地址转换方法进行了分析测试。实验和分析表明，通过对 EPT 实现机制优化，进行页表的预分配可以达到为客户机分配固定内存大小的效果。

基于处理器虚拟化的技术，本文还进行了 Intel VT-x 的 VMCS 技术的研究，并以此为出发点研究了虚拟处理器的创建、初始化与运行的具体实现，并依赖于 VMCS 技术和 Linux 进程绑定技术研究了虚拟机与处理器核心绑定的技术，设计并实现了一种虚拟机与处理器核心绑定的方案。

本文在研究中对提出的优化方案和实现方案分别进行了大量的测试，其中包括对内存虚拟化中 EPT 页表的优化、虚拟机绑定实现和客户机内存申请的测试并对测试数据进行统计分析。测试结果的分析表明，本文提出的 EPT 页表优化方案和虚拟机绑定取得效果，通过长时间运行以及 Benchmark、自写测试程序的验证，系统稳定，具有实际应用意义。

关键词： 虚拟机 EPT 内存虚拟化 进程绑定 Intel VT-x

Research and Application of Virtualization Technology based on Linux

Introduction of the author: Li Shao, male, was born in October, 1984, whose tutor was Professor Luo Shengxian. He graduated from Chengdu University of Technology in Computer Application major and was granted the Master Degree in July, 2011.

ABSTRACT

Gone through decades of doldrums from the birth of 1960s to now, virtualization technology once developed with the development of computer technology after 1990s. Operating systems has always faced a challenge from the conflict between the independence and interoperability of resources for the application from the interaction, that is, each application would like to run the system in a relatively independent environment, free from interference from other applications, while allowing to share and exchange system resources with other applications for easy and effective way. Conventional computer operating system lack of effective support for independence between applications, so the goals to run multiple independent operating systems on a computer platform is the development of virtualization technology.

Virtualization technology has been rapidly developed in recent years, it can improve processor utilization and processor vendors have introduced hardware support for virtualization, the processor, chipset and so on. Virtualization technology is now becoming widely used as a software layer.

This thesis studies the basic theory and mechanisms of virtualization technologies, studies the implementation principle of full virtualization, and the implementation techniques of processor and memory virtualization; and see the Hardware Virtualization technology as the target, study VMX virtual machine operating mode, and discuss Linux-based KVM virtualization solutions in depth.

Based on the study and summarize the principle and key technologies of the Linux-based KVM virtual machine implementation, this thesis focuses on the memory virtualization technology and the EPT mechanism based Intel VT-x virtualization technology, combined with the methods of EPT page table address translation, with the specific demand, proposed and implemented a virtual memory

optimization program in the EPT mechanism, and the optimization scheme is applied to actual projects. Based on the research process, do the analysis and testing for the conversion process on the Linux page addresses and virtual machine EPT page table address translation. Experiments and analysis show that the pre-allocation of the page table can be achieved to allocate fixed memory size effect for the client through the optimization of EPT.

Based on the processor virtualization technology, the thesis also carried out the research of the VMCS Intel VT-x technology, and as a starting point for research of the creation of a virtual processor, a concrete realization of initialization and operation, and research the binding technology between virtual machine and processor core rely on VMCS technology and the process binding technology of Linux, designed and implemented a binding solution between a virtual machine and the processor core.

In this thesis, extensive testing were carried out for the solutions the optimization and implementations, including the EPT optimization of the memory virtualization, virtual machine bound implementation and require memory for client machine and test data for statistical Analysis. The results of the analysis showed that the proposed optimization scheme EPT page tables and virtual machine bound achieved results, with practical significance.

Key words: virtual machine, virtual memory, the process binding, EPT, Intel VT-x

目 录

第 1 章 前言	1
1.1 研究背景及意义	1
1.2 研究现状	2
1.2.1 虚拟机软件研究现状	2
1.2.2 基于 Linux 内核的虚拟机 KVM 研究现状	3
1.3 研究内容	3
1.4 论文结构	4
第 2 章 计算机虚拟化技术的机制与原理	5
2.1 计算机虚拟化基本概念	5
2.1.1 计算机虚拟化结构层次	5
2.1.2 虚拟化技术的分类	6
2.2 基于软件的完全虚拟化	7
2.2.1 完全虚拟化的基本原理	8
2.2.2 全虚拟化具体实现技术	8
2.3 硬件辅助虚拟化	9
2.3.1 Intel VT 虚拟化技术	10
2.3.2 虚拟机操作模式简介	10
2.4 基于 Linux 的 KVM 虚拟机实现原理	12
2.4.1 KVM 虚拟化方案简介	12
2.4.2 KVM 执行流程分析	13
第 3 章 内存虚拟化及 EPT 机制优化方法研究	15
3.1 功能需求	15
3.2 内存虚拟化的解决方案	15
3.2.1 Linux 地址转换	15
3.2.2 Linux 分页机制	15
3.2.3 页表虚拟化思想	16
3.2.4 内存隔离的实现方法	17
3.3 基于 EPT 页表的地址转换方法	17
3.3.1 EPT 机制工作原理及实现	17

3.3.2 基于 EPT 技术的地址转换过程.....	18
3.3.3 EPT 页表地址转换机制的优化方法	20
3.4 详细设计	20
3.4.1 主要的数据结构	20
3.4.2 实现步骤	23
第 4 章 基于 KVM 的处理器虚拟化及虚拟机绑定研究	26
4.1 虚拟机与客户机间的模式切换分析	26
4.1.1 VMCS 数据结构	26
4.1.2 VM_Exit 流程	27
4.1.3 VM_Entry 流程	28
4.2 虚拟 CPU 的结构及运行机制	28
4.2.1 VCPU 结构体分析	28
4.2.2 VCPU 的创建过程	29
4.2.3 VCPU 的运行	30
4.3 虚拟机绑定的设计方案	31
4.3.1 研究目标	31
4.3.2 设计思路	31
4.3.3 虚拟机绑方法设计	33
4.4 实现方法和步骤	33
4.4.1 主要数据结构	33
4.4.2 进程绑定的实现	34
4.5 实现效果	36
第 5 章 测试及实验结果分析	37
5.1 内存虚拟化中 EPT 页表优化测试	37
5.1.1 测试目标	37
5.1.2 测试方法和步骤	37
5.1.3 效果分析及结论	38
5.2 虚拟机绑定测试	41
5.2.1 测试目标	41
5.2.2 测试方法和步骤	41
5.2.3 效果分析与结论	42
5.3 虚拟机内存申请测试	46

5.3.1 测试项目目标	46
5.3.2 测试方法描述	46
5.3.3 效果分析与结论	47
结 论	49
致 谢	51
参考文献.....	53
攻读学位期间取得学术成果.....	55

第1章 前言

虚拟机技术是当前学术界和工业界共同研究的热点，近年来多核系统、集群、网络甚至云计算的广泛部署，使得虚拟化技术在商业应用上的优势日益显著，不仅降低了 IT 业成本，而且能够有效地提高处理器应用的灵活性、安全性和可扩展性。目前虚拟化技术是计算机操作系统和系统结构领域的重要研究方向，并且已经在 Linux 操作系统中得到支持和应用。

1.1 研究背景及意义

随着计算机技术的发展，人们对计算机处理速度的要求也在不断提高，但是仅仅提高单核芯片的速度会产生过多热量且无法带来相应的性能改善，除了热量带来的问题，单核处理器的性价比也难以让人们接受，快速的单核处理器价格要高出很多。多核处理器可以带来性能上的明显提高，通过对核心进行逻辑处理器的区分和控制，同时在操作系统的内核进行任务划分，可实现特定时钟周期内的多任务执行。但是如果要充分利用多核新技术的优势，有些优秀的单核遗产操作系统的移植成本会很高，如何在多核 CPU 上实现多操作系统同时运行，目前流行的虚拟化技术提供了有效的解决方案，通过虚拟化技术，可实现相互隔离的客户单核系统运行在同一 CPU 的不同核心上，让现有的单核商业操作系统仅作出很少的修改就可利用性能不断提高的多核新技术处理器，使得处理器的性能在软件研发成本最小化的条件下得到充分的利用。

目前计算机厂商正在大力发展支持虚拟技术的硬件，如英特尔公司的处理器虚拟化技术 VT-x，通过按照纯软件虚拟化的要求消除虚拟机监视器(VMM)代表客户操作系统听取、中断与执行特定指令的需要，不仅能够有效减少 VMM 干预，还为 VMM 与客户操作系统之间的传输平台控制提供了有力的硬件支持，这样在需要 VMM 干预时，可实现更加快速、可靠和安全的切换；支持连接的虚拟化技术 VT-c，提高了交付速度，减少了 VMM 与服务器处理器的负载；以及支持设备虚拟化的 VT-d 技术，通过减少 VMM 参与管理 I/O 流量的需求，不但加速了数据传输，而且消除了大部分的性能开销。本文所依托的虚拟化与多核技术项目是在以上硬件支持的虚拟技术基础上对 Linux 内核 2.26.33.7 进行裁剪，并针对客户机的需求进行处理器隔离、内存隔离等研究和性能优化，通过研究对内存、网卡、串口等设备进行虚拟化，使得客户系统不需要或只做少量修改，就能充分利用多核处理器技术和性能，实现在一个双核处理器平台上相互独立地运行两个单核系统。

1.2 研究现状

伴随着硬件的发展,上世纪 90 年代后期开始,台式机的性能逐渐达到支持多个系统同时运行的水平,小型机和微机领域的虚拟化技术开始迅速升温。1997 年,在斯坦福大学开发的 Disco 系统中探索了在共享内存的大规模多处理器系统上运行普通的桌面操作系统^[1]。

在个人计算机领域广泛使用的 x86 体系结构设计中,存在对系统虚拟化的支持缺陷,因此 x86 体系结构的虚拟化技术需要使用软件的方法来弥补体系结构设计的不足。虽然基于软件的方法可以实现系统虚拟化,但是在发展过程中存在着不可回避的问题:性能的下降以及一些兼容性的损失。为了从根本上解决体系结构的缺陷,Intel 和 AMD 两家公司都发展了带有硬件处理器支持的虚拟化技术,即 Intel 公司的 VT (Virtualization Technology)技术和 AMD 公司的 SVM (Secure Virtual Machine) 技术。

为了使虚拟化解方案更加高效,计算机的各个层次都在逐渐加入对虚拟化的硬件支持,逐渐形成一个对虚拟化更好支持的虚拟化环境系统,如在 Intel 的处理器采用 VT 技术外,芯片组中开始提供针对 I/O 虚拟化功能的 VT-d 技术,网卡中也开始提供更好的网络虚拟化支持的多序列 VMDq 技术等。PCI 标准组织在制定 PCI 设备级对虚拟化进行支持的单根 PCI 桥 IOV (Single Root IOV, SR-IOV) 和多根 PCI 桥 IOV (Multi-Root IOV, MR_IOV) 标准。

1.2.1 虚拟机软件研究现状

近些年发展推出了以下重要的虚拟化软件^[2]:

(1) Bochs(模拟器)

Bochs 是一个仿真 X86CPU 的计算机模拟器,它具有可移植性,可运行在多种平台上,如 x86, PowerPC, Alpha, SPARC, Bochs, 其优点是不仅能模拟处理器,而且能模拟整台计算机,包括键盘、鼠标、视屏图像硬件、网卡等。

(2) 库级虚拟化

这种虚拟化技术通过库来模拟操作系统的一部分,如 Wine。

(3) QEMU(模拟器)

QEMU 支持两种操作模式,一种是全系统模拟模式,该模式模拟整个计算机系统,而且能在合理的速度下使用动态翻译模拟一些处理器架构;另一种模式是用户模式模拟,该模式能寄存在 Linux 上,并且支持不同平台的二进制程序运行。

(4) VMWare 全虚拟化

VMWare 是商业级的全虚拟化技术,以 Hypervisor 为客户操作系统和硬件

之间的抽象层，该抽象层允许任何其它客户操作系统运行在主机操作系统上。

（5）Z/VM 全虚拟化

Z/VM 是 System Z 操作系统的 Hypervisor。核心是控制程序 CP，向客户机操作系统提供硬件资源虚拟，允许在多个客户操作系统上虚拟多个处理器和其他资源。

（6）Xen 半虚拟化

Xen 由 XenSource 开发，是一种开源免费的操作系统级准虚拟技术。Xen 需要与系统共同协作，客户操作系统需要修改，从系统支持的角度来看，这种虚拟化技术会使用户付出一定的开发成本。

1.2.2 基于 Linux 内核的虚拟机 KVM 研究现状

Linux 内核虚拟机 KVM (Kernel-based Virtualization Machine)，最早由 Qumranet 公司开发，它是基于 GPL 授权方式的开源虚拟机软件，并在 2007 年 2 月被集成进入 Linux2.6.30 内核代码树中。

KVM 是采用基于 Intel 技术的硬件虚拟化方法，并结合 QEMU 来提供设备虚拟化。KVM 的特点在于和 Linux 内核结合得很好，它继承了 Linux 的大部分功能，可以充分利用 Linux 内核一直在不断优化和改进的技术优势，另外 KVM 做为 Linux 内核的一个模块加入内核代码树，该项目的发起人和维护人倾向于认为 KVM 是 Hypervisor 模型。

1.3 研究内容

通过对当前国内外计算机虚拟化技术研究现状的分析，本文基于 Intel-VT 技术研究内存虚拟化原理、EPT 机制应用的优化技术、处理器虚拟化原理、VMCS 和虚拟 CPU 的构成和运行机制以及虚拟机绑定实现方法。通过研究提出一套将 Guest OS 绑定到单个 CPU 核心的解决方案，利用 EPT 页表进行内存虚拟化、减少 Host 与 Guest 的切换频率，提高系统运行效率的优化方案。

本论文的主要研究内容如下：

- （1）研究 Linux 内核的内存管理、内存分页访问机制和 EPT 页表技术；
- （2）深入分析内存虚拟化技术原理和 KVM 内核虚拟化解决方案，研究内存虚拟化实现技术和优化方案；
- （3）基于全虚拟化技术，研究硬件支持的处理器虚拟化技术；
- （4）基于 VT-x 技术，研究 VMCS 和虚拟处理器的运行机制和应用；
- （5）在研究处理器虚拟化技术基础上，研究实现单个 CPU 核心与客户机绑定的方法。

1.4 论文结构

第 1 章 前言。介绍研究背景，国内外计算机虚拟化技术的研究现状以及虚拟机软件开发现状，本文研究内容与研究目标。

第 2 章 计算机虚拟化技术的基础。讨论计算机虚拟化技术的基本概念、计算机虚拟化技术基本原理。

第 3 章 内存虚拟化研究及 EPT 机制优化实现。本章是全文的核心实现之一，从项目和实际的需求出发，设计并实现了符合运行两个虚拟机时为虚拟机分配固定内存需求的优化方案。讨论了 Linux 的内存管理和页表虚拟化思想以及 KVM 虚拟化方案中内存虚拟化的实现原理，并详细讨论了 Intel 硬件加速内存虚拟化中 EPT 页表优化方案的设计思路与实现技术。

第 4 章 处理器虚拟化研究及虚拟机绑定实现。本章是全文的核心实现之二。从项目和实际需求出发，设计并实现了多核处理器环境下单个虚拟机与一个指定处理器核心绑定的方案。讨论了处理器虚拟化的基本原理，并重点分析了结合 Intel 硬件支持虚拟化 VT-x 技术中 VMCS 技术与 Linux 进程绑定技术实现指定 CPU 与指定虚拟机绑定的设计思路与实现技术。

第 5 章 测试及实验结果分析。本章论述了对第 3 章内存虚拟化和 EPT 机制优化方案、第 4 章虚拟机绑定方案进行测试用例设计，进行测试，并对测试结果进行了相应的分析。

第 2 章 计算机虚拟化技术的机制与原理

计算机虚拟化技术发展了几十年，虚拟化思想从最初的部分虚拟化，然后产生了基于软件的完全虚拟化思想，直至近些年为解决硬件体系结构在虚拟化方面设计存在的缺陷而并推出的硬件支持虚拟化，虚拟化性能越来越高。本章将主要介绍计算机虚拟化技术的机制原理。首先介绍计算机虚拟化的基本概念，包括虚拟化技术的分类等；其次介绍具体的虚拟化技术，主要从虚拟化原理、处理器虚拟化以、内存虚拟化以及硬件辅助虚拟化几方面阐述；最后介绍了基于 Linux 的 KVM 虚拟机的实现。

2.1 计算机虚拟化基本概念

虚拟化是一个广义的术语。对于计算机科学来说，虚拟化代表着对计算资源的抽象。概括来说，虚拟化就是改变事物的形式状态，从一种形式变为另外一种形式。计算机的虚拟化是使单个计算机看起来像多个相同的或者不同的计算机。

2.1.1 计算机虚拟化结构层次

现代计算机系统是一个庞大、复杂、密切协调的整体，对其自下而上进行层次划分，每一层都向上一层呈现一个抽象，并且每一层只看见下一层的抽象借口，下层的内部具体实现机制被封闭^[3]，如图 2-1 所示。因此，把资源抽象为层的方式，每一层只考虑本层的设计以及向上层提高接口，与下层进行交互。

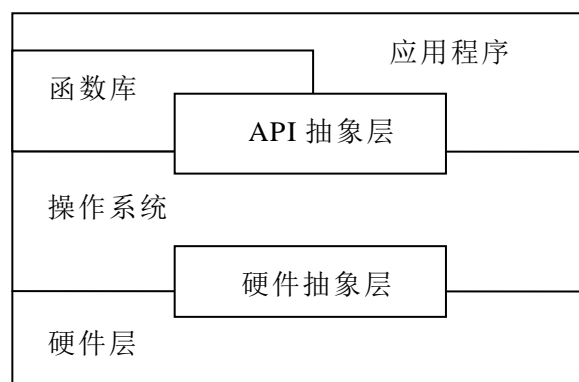


图 2-1 计算机系统的抽象层

因此从本质上看，虚拟化就是处于下一层的软件模块利用向上一层的软件模块提供一个接口的方法（该方法提供的接口与它原先所期待的运行环境完全一致的），抽象出一个经过虚拟的硬件或软件接口，从而使上层软件可以像运行

在真实物理环境上一样直接运行在虚拟的环境上。

在虚拟化中，真实的物理环境一般称为宿主机（Host），而经过虚拟化建立的环境称为客户机（Guest）。在宿主机上运行的系统称宿主机操作系统 Host OS，在客户机上运行的操作系统称为客户机系统 Guest OS。计算机系统的每一个层次都可以进行虚拟化，不同层次的虚拟化可以带来不同的虚拟化效果。在虚拟化环境中，利用虚拟化技术可以将一台真实的物理计算机系统变为一台或者多台虚拟的计算机系统，这样经过虚拟化的计算机系统都有自己的虚拟机执行环境，包括处理器、内存、各种设备等虚拟硬件。在计算机虚拟化结构层次中，负责虚拟化的这层称为虚拟机监控器 VMM（Virtual Machine Monitor）。

2.1.2 虚拟化技术的分类

虚拟化技术可以有不同的划分方法，一般的可以划分为以下几类^[2]：

（1）平台虚拟化（Platform Virtualization），针对操作系统和计算机的虚拟化，即对硬件层和操作系统层的虚拟化。

（2）资源虚拟化（Resource Virtualization），针对特定的计算机系统资源的虚拟化，例如内存、存储、网络资源等，即对硬件层的虚拟化。

（3）应用程序虚拟化（Application Virtualization），包括仿真、模拟、解释技术等，即对应用程序层的虚拟化。

以上每种划分方法根据具体的应用又可以划分为不同的类别，平台虚拟化技术又可以细分为如下几个子类：

（1）全虚拟化（Full Virtualization）

全虚拟化是指虚拟机模拟了完整的底层硬件，包括处理器、物理内存、时钟、外设等，使得为原始硬件设计的操作系统或其它系统软件完全不做任何修改就可以在虚拟机中运行。操作系统与真实硬件之间的交互可以看成是通过一个预先规定的硬件接口进行的。全虚拟化 VMM 以完整模拟硬件的方式提供全部接口。比如 x86 体系结构中，对于操作系统切换进程页表的操作，真实硬件通过提供一个特权 CR3 寄存器来实现该接口，操作系统只需执行 "mov pgtable, %%cr3" 汇编指令即可。全虚拟化 VMM 必须完整地模拟该接口执行的全过程。如果硬件不提供虚拟化的特殊支持，那么这个模拟过程将会十分复杂：一般而言，VMM 必须运行在最高优先级来完全控制主机系统，而 Guest OS 需要降级运行，从而不能执行特权操作。当 Guest OS 执行前面的特权汇编指令时，主机系统产生异常（General Protection Exception），执行控制权重新从 Guest OS 转到 VMM 手中。VMM 事先分配一个变量作为影子 CR3 寄存器给 Guest OS，将 pgtable 代表的客户机物理地址（Guest Physical Address）填入影

子 CR3 寄存器，然后 VMM 还需要将 pgtable 翻译成主机物理地址（Host Physical Address）并填入物理 CR3 寄存器，最后返回到 Guest OS 中。随后 VMM 还将处理复杂的 Guest OS 缺页异常（Page Fault）。

（2）超虚拟化（Paravirtualization）

这是一种修改 Guest OS 部分访问特权状态的代码以便直接与 VMM 交互的技术。在超虚拟化虚拟机中，部分硬件接口以软件的形式提供给客户机操作系统，这可以通过 Hypercall（VMM 提供给 Guest OS 的直接调用，与系统调用类似）的方式来提供。例如，Guest OS 把切换页表的代码修改为调用 Hypercall 来直接完成修改影子 CR3 寄存器和翻译地址的工作。由于不需要产生额外的异常和模拟部分硬件执行流程，超虚拟化可以大幅度提高性能。

（3）硬件辅助虚拟化（Hardware-Assisted Virtualization）

硬件辅助虚拟化是指借助硬件（主要是主机处理器）的支持来实现高效的全虚拟化。例如有了 Intel-VT 技术的支持，Guest OS 和 VMM 的执行环境自动地完全隔离开来，Guest OS 有自己的“全套寄存器”，可以直接运行在最高级别。因此在上面的例子中，Guest OS 能够执行修改页表的汇编指令。Intel-VT 和 AMD-V 是目前 x86 体系结构上可用的两种硬件辅助虚拟化技术。

（4）部分虚拟化（Partial Virtualization）

VMM 只模拟部分底层硬件，因此客户机操作系统不做修改是无法在虚拟机中运行的，其它程序可能也需要进行修改。在历史上，部分虚拟化是通往全虚拟化道路上的重要里程碑，最早出现在第一代的分时系统 CTSS 和 IBM M44/44X 实验性的分页系统中。

（5）操作系统级虚拟化（Operating System Level Virtualization）

在传统操作系统中，所有用户的进程本质上是在同一个操作系统的实例中运行，因此内核或应用程序的缺陷可能影响到其它进程。操作系统级虚拟化是一种在服务器操作系统中使用的轻量级的虚拟化技术，内核通过创建多个虚拟的操作系统实例（内核和库）来隔离不同的进程，不同实例中的进程完全不了解对方的存在。比较著名的有 Solaris Container [2]，FreeBSD Jail 和 OpenVZ 等。

2.2 基于软件的完全虚拟化

在 Intel、AMD 虚拟化技术出现之前，传统硬件体系结构在设计方面不能很好的支持虚拟化，因此在对硬件层和操作系统层进行系统虚拟化时，并不能直接有效地实现虚拟化需求。为了解决这些硬件上的缺陷，只能在基于软件的虚拟化技术上寻找解决方案：模拟执行和直接源代码改写。其中模拟执行就是

基于软件的完全虚拟化。

2.2.1 完全虚拟化的基本原理

传统的 x86 体系结构缺乏必要的虚拟化硬件支持，任何虚拟机监控器都无法直接利用硬件虚拟化的优势，虽然这种传统 x86 体系架构不支持虚拟化，但是我们可以使用纯软件实现的方式构造虚拟机监控器。

虚拟机是对真实计算环境的抽象和模拟，VMM 需要为每个虚拟机分配一套数据结构来管理它们状态，包括虚拟处理器的全套寄存器，物理内存的使用情况，虚拟设备的状态等等。VMM 调度虚拟机时，将其部分状态恢复到主机系统中。并非所有的状态都需要恢复，例如主机 CR3 寄存器中存放的是 VMM 设置的页表物理地址，而不是 Guest OS 设置的值。主机处理器直接运行 Guest OS 的机器指令，由于 Guest OS 运行在低特权级别，当访问主机系统的特权状态（如写 GDT 寄存器）时，权限不足导致主机处理器产生异常，将运行权自动交还给 VMM。此外，外部中断的到来也会导致 VMM 的运行。VMM 可能需要先将该虚拟机的当前状态写回到状态数据结构中，分析虚拟机被挂起的原因，然后代表 Guest OS 执行相应的特权操作。最简单的情况，如 Guest OS 对 CR3 寄存器的修改，只需要更新虚拟机的状态数据结构即可。一般而言，大部分情况下，VMM 需要经过复杂的流程才能完成原本简单的操作。最后 VMM 将运行权还给 Guest OS，Guest OS 从上次被中断的地方继续执行，或处理 VMM 注入的虚拟中断和异常。这种经典的虚拟机运行方式被称为 Trap-And-Emulate，虚拟机对于 Guest OS 完全透明，Guest OS 不需要任何修改，但是 VMM 的设计会比较复杂，系统整体性能受到明显的损害。

基于软件的完全虚拟化技术可以实现对计算机系统各个层面的虚拟化，但是也存在着一些影响性能的问题，比如：降优先级（deprivilege）的方法存在 Ring Compression 问题，在处理器虚拟化实现中存在难以处理自修改代码以及自参考代码的问题，另外由于硬件架构的限制，一些功能虽然可以用软件方法实现，但是实现方法非常复杂，如内存虚拟化中的影子页表等等，这些都限制了计算机虚拟化技术的发展，为了解决这些问题，最近几年 Intel、AMD 等公司相继发布了支持虚拟化技术的硬件平台。

2.2.2 全虚拟化具体实现技术

一般来说，所有的虚拟化都可以通过虚拟来实现。作为管理者 VMM 对物理资源的虚拟可以划分为三个部分^[3]：处理器虚拟化、内存虚拟化和 I/O 设备虚拟化。在完全虚拟化技术中采用了借助模拟执行的软件来实现。

2.2.2.1 处理器虚拟化简介

硬件支持虚拟化出现之前，为了解决传统虚拟化的漏洞，在实际应用中已经采取了一些模拟技术，提供了平台虚拟化的能力。完全虚拟化的处理器虚拟化是基于软件的处理器虚拟化，它的本质就是软件模拟。

在计算机系统中，从资源的角度看，处理器呈现给软件的接口就是一系列的指令或指令集和一些寄存器，包括通用运算的寄存器和用于控制处理器行为的状态和控制寄存器。这些资源中影响处理器状态和行为的寄存器被称为特权资源，而读写这些资源的指令被称作敏感指令。绝大多数的敏感指令是特权指令，特权指令只能在处理器的最高特权级别（内核态）执行，如果执行特权指令的处理器不在内核态，就会引发一个异常。软件模拟的主要工作就是要处理虚拟机执行时遇到的这些敏感指令。模拟的好处在于，VMM 可以控制整个虚拟机的执行过程，可以对所执行的每一条指令进行模拟，不会导致遗漏需要模拟的敏感指令。

处理器虚拟化主要有以下几种实现方法^[15]：

- （1）解释执行，即取一条指令，模拟出这条指令，再继续取下一条指令，往复循环。
- （2）扫描与修补，让大部分的指令直接在物理 CPU 上执行，而把操作系统代码中的敏感指令替换为跳转指令或陷入到 VMM 中去执行的指令，使其运行到敏感指令时出控制流就会进入 VMM 中，通过 VMM 来模拟执行。
- （3）二进制代码翻译，在 VMM 中开辟一块代码缓存，存放翻译好的代码，虚拟机操作系统代码不会直接被物理 CPU 执行。

2.2.2.2 内存虚拟化简介

内存虚拟化要实现两个目的：

- （1）为虚拟机提供一个从零开始连续的物理内存空间。
- （2）在各虚拟机之间以及虚拟机与宿主机之间有效隔离、调度并共享内存资源。

完全虚拟化中采用影子页表来实现内存虚拟化。所谓影子页表是由 VMM 维护的一张页表，并保持和客户机页表一致，动态修改。

2.3 硬件辅助虚拟化

所谓硬件辅助虚拟化技术，就是在处理器、芯片组以及 I/O 设备等硬件中添加了虚拟化支持，可以让计算机软件非常有效、直接的实现虚拟化功能。现在主要有 Intel 公司的 Intel VT（Intel Virtualization Technology）以及 AMD 公

公司的 AMD-V (AMD Virtualization) 虚拟化技术, 本文基于 Intel VT 环境进行虚拟化技术研究。

2.3.1 Intel VT 虚拟化技术

Intel 虚拟化技术平台包括对处理器、内存和 I/O 设备的虚拟化支持, 其中提供了 VT-x(Intel VT for x86)来支持处理器虚拟化, 并提供了 EPT (Extended Page Table), 支持内存虚拟化。

在 Intel VT 技术实现的计算机虚拟化系统中, 上层是资源管理、系统调度等通用功能, 下层是 Intel VT 实现的处理器、内存等虚拟化与平台相关的部分, 其结构如图 2-2 所示。

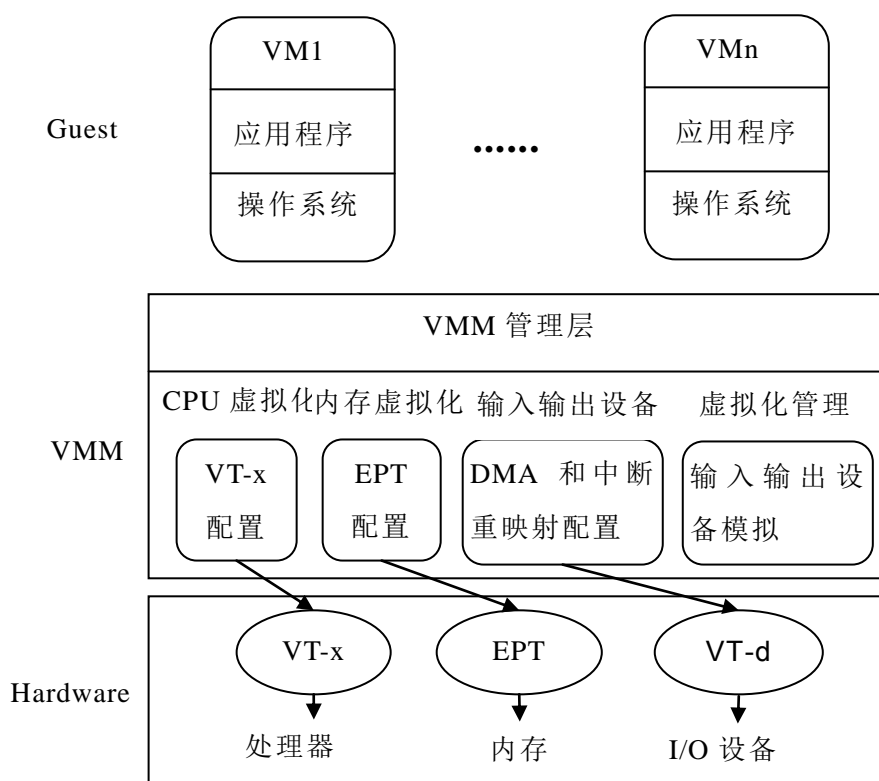


图 2-2 Intel 虚拟化结构图

2.3.2 虚拟机操作模式简介

传统的 IA32 处理器架构的 Linux 操作系统, 在系统保护角度存在着两种模式: 内核模式和用户模式。Intel VT 技术扩展了传统处理器架构, 其中的 VT-x 提供了处理器虚拟化的硬件支持。VT-x 引入了一种概念, 称为 VMX 操作模式, 它又包括两种模式: 一种是根操作模式 VMX Root Operation, 对应着 Linux 操作系统中的用户模式和内核模式, 也即是 VMM 运行的模式; 另外一种是非根模式 VMX Non-Root Operation, 也称为客户模式, 即客户机运行的模式。如图 2-3 所示是 VT-x 的系统架构。VMX 的两种操作模式类似 Linux, 也有对应的特权

级^[3]。

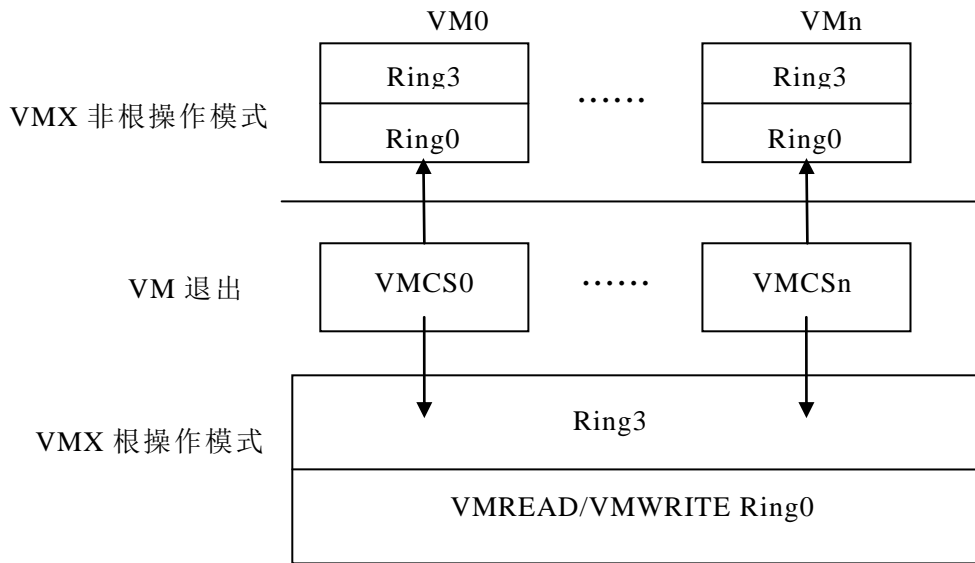


图 2-3 Intel VT-x 结构图

VT-x 技术中添加了一组新的命令供 VMM 控制客户机，包括 VMLAUCH、VMRESUME、VMREAD、VMWRITE 等，作为传统计算机平台架构的扩展，传统的操作系统没有虚拟化功能，默认情况下 VMX 操作模式功能是关闭的。可以通过 VMM 使用 VT-x 扩展指令开启 VMX 操作模式，其中 VMXON、VMXOFF 命令分别可以打开或关闭 VMX 操作模式。图 2-4 所示描述的是 VMX 模式下的操作，具体操作流程如下：

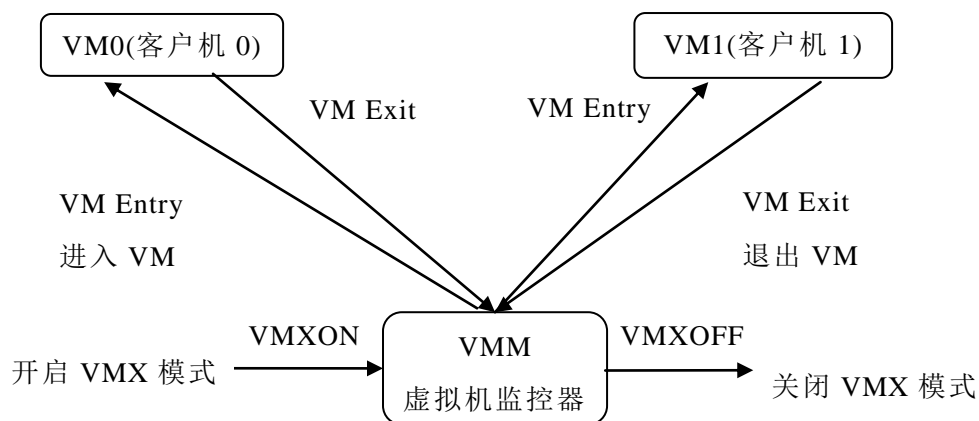


图 2-4 VMX 操作模式

(1) VMM 执行 VMXON 指令进入到 VMX 操作模式，CPU 处于 VMX 操作模式，VMM 软件开始执行；

(2) VMM 执行 VMLAUNCH 或 VMRESUME 指令进入虚拟机(VM-Entry)，客户机软件开始执行，此时 CPU 进入非根模式；

(3) 当客户机执行特权指令，或者当客户机运行时产生了中段或者异常，

触发 VM 退出 (VM-Exit) 而陷入到 VMM, CPU 切换到根模式。VMM 根据 VM-Exit 的原因作相应处理, 然后转到 (2) 继续运行客户机。

2.4 基于 Linux 的 KVM 虚拟机实现原理

内核虚拟机 Linux KVM 是全虚拟解决方案, 它的特点是系统内核通过添加内核模块使内核自身称为一个虚拟机管理程序。本章节主要介绍了 KVM 虚拟化方案的设计原理, 并详细分析了 KVM 虚拟机的执行流程。

2.4.1 KVM 虚拟化方案简介

KVM 虚拟化解决方案的底部是要进行虚拟化的硬件, 这些硬件有些部分直接支持虚拟化, 也有些部分不直接支持虚拟化, KVM 模块负责设备虚拟以及内核与 QEMU 进行 I/O 的逻辑处理。图 2-5 为使用 KVM 的虚拟化组件示意图。

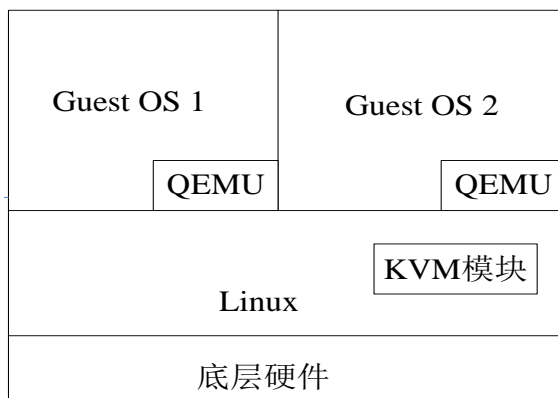


图 2-5 KVM 的虚拟化组件图

图 2-5 中底层是支持虚拟化的硬件, 硬件之上是加载了 KVM 模块的 Linux 内核, 最上层是可以直接运行在硬件上的 n 个客户机。

在 KVM 虚拟化解决方案中, 处理器直接提供虚拟化支持, 其中内存通过 KVM 进行虚拟化, 而 I/O 则通过一个用户空间的 QEMU 进程来虚拟化。

加载了 KVM 内核模块时会创建 `/dev/kvm` 设备来提供内存虚拟化。在 VMM 创建 VM, 并加载 VM 客户机操作系统 Guest OS 时, 会将 Guest 作为一个进程运行并为它们映射相应的地址空间。为客户机操作系统映射物理内存实际上就是给这个进程映射虚拟内存, VMM 维护了一组影子页表 SPT (shadow page table), 来处理客户机物理地址到宿主机物理地址的映射。

KVM 向 Linux 中引入了一种除现有的内核模式和用户模式之外的新进程模式: 客户模式, 用来执行客户操作系统代码。根据一般 Linux 系统的执行模式, 客户模式可以执行客户操作系统的代码, 但是只能执行非 I/O 的代码, 而 I/O 的代码是由 QEMU 提供的。QEMU 可以虚拟化一个计算机平台, 它允许对一个

完整的 PC 环境进行虚拟化，在客户机操作系统中所有的 I/O 请求都会被 VMM 截获，然后转发给用户空间的 QEMU 进程来模拟 I/O 操作。Guest OS 有两种运行模式，在客户模式下运行时可以支持标准的内核，而在用户模式下运行时则支持自己的内核和用户空间应用程序。

2.4.2 KVM 执行流程分析

KVM 是一个典型的 Linux 字符设备，通过 `/dev/kvm` 设备节点在用户空间可以调用一组 `ioctl()` 函数来创建和运行虚拟机。`/dev/kvm` 提供的操作有以下几种：

- (1) 创建一个新的虚拟机；
- (2) 为新虚拟机分配内存；
- (3) 设定 VCPU（虚拟 CPU）寄存器；
- (4) 注入中断到 VCPU 中；
- (5) 运行一个 VCPU。

在 KVM 中，一个虚拟处理器 VCPU 并不由它自己来调度，而是由 VMM 来决定的。一个客户机有自己的内存空间，独立于创建它的用户程序，和 Linux

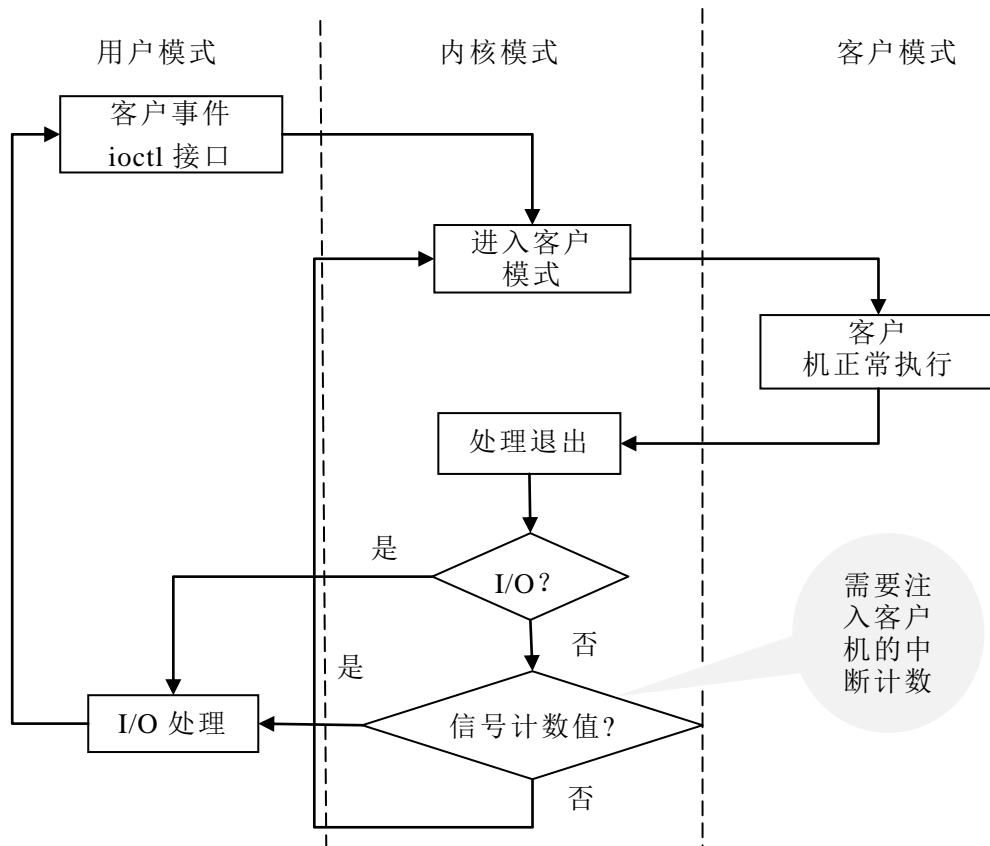


图 2-6 客户机执行流程图

中用户使用内存一样，VMM 内核通过分配不连续的页来构成客户机地址空间。

另外，在用户空间里使用 `mmap()` 函数映射并直接访问客户机内存。

Guest OS 在运行过程中将涉及到三种模式：内核模式、客户模式以及新增加的的客户模式，在 KVM 中它的执行流程如图 2-6 所示。

根据图 2-6 所示，Guest OS 执行流程的特点如下：

（1）在最外层，用户空间调用内核函数来执行客户模式，在遇到 I/O 指令或者有信号触发的外部事件时退出客户模式。

（2）在内核层，内核触发硬件机制进入客户模式。如果由外部中断或者影子页表缺页这类事件引起处理器退出客户模式，内核会进行对应的处理，并重新进入客户机执行。如果退出原因是 I/O 指令或者进程调度信号，则内核会退出到用户空间执行。

（3）在硬件层，处理器遇到错误处理、外部中断等指令时会退出客户模式。

第3章 内存虚拟化及 EPT 机制优化方法研究

针对 KVM 虚拟机优化要求支持 8G 内存及实现虚拟机系统间的内存隔离，研究 Linux2.6.33 内核中内存寻址、分页机制、内存管理以及 KVM 虚拟化解决方案中的内存虚拟化，在此基础上设计与实现 KVM 虚拟机系统内存的隔离方法，通过虚拟化技术实现在双核处理器上同时独立运行两个客户操作系统，达到充分利用多核计算机系统资源的目标。

3.1 功能需求

VMM 虚拟机系统内存管理应满足以下的功能需求及性能指标：

- (1) 系统能够寻址 8GB 物理内存地址空间，地址采用 36bits。
- (2) 两个客户操作系统相互隔离。实现内存分区，每个客户系统占有固定的 3.5G 物理内存，寻址空间相互隔离。
- (3) 没有硬盘，缺页处理时不支持换入换出。
- (4) 虚拟机地址转换的时间花费不影响系统性能。

3.2 内存虚拟化的解决方案

3.2.1 Linux 地址转换

在内存使用中涉及三种不同的地址：逻辑地址，线性地址，物理地址。逻辑地址由一个段和偏移量组成，偏移量指明了从段起点地址到实际地址之间的距离；线性地址即虚拟地址，是一个 32 位无符号整数，可以表达 4G 的地址，其取值范围为 0x00000000~0xffffffff；物理地址用于内存芯片级内存单元寻址，本项研究涉及的物理地址由 36 位无符号整数表示^{[13][14]}。

内存控制单元 MMU 通过分段单元的硬件电路把一个逻辑地址转换为线性地址，然后通过分页单元的硬件电路把线性地址转换成物理地址。图 3-1 为地址转换的示意图^{[4][7]}。

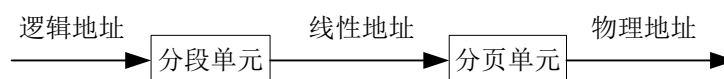


图 3-1 两级地址转换机制

3.2.2 Linux 分页机制

在分页单元中，把 32 位线性地址分成不同的域，每个域若干位，代表一个索引项，指向一个下级索引表或页面。线性地址的转换都是基于一种转换表，称为页目录表或者页表。

Linux2.6.11 版本开始采用四级分页模型，共有 4 种页表：页全局目录 PGD、页上级目录 PUD、页中间目录 PMD、页表 PT。启用了物理地址扩展的 32 位系统则使用三级页表，其中页全局目录对应 x8086 的页目录指针表 PDPT，取消页上级目录，页中间目录对应页目录，页表保留。图 3-2 为 Linux 分页模式。

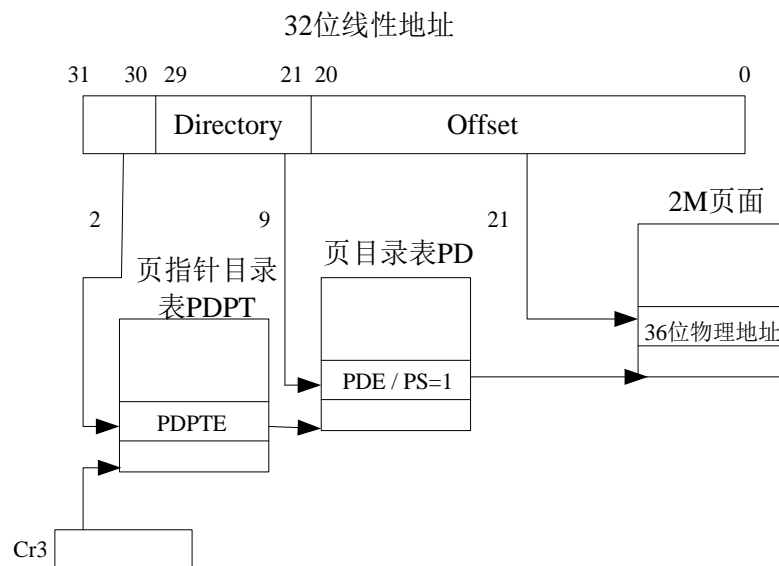


图 3-2 Linux 分页模式

在这种分页模式中，启用了物理地址扩展 PAE 分页机制，页表项需要包含 12 位标志位和 24 位物理地址位，其大小扩展到 64 位，一个 4K 页表包含 512 个表项。页目录指针表的页表，由 4 个 64 位表项组成，第四个表项对应内核空间的 1G 线性空间。在页目录表的表项中，打开标志位中的 PS 位设置为 1，这样每个表项指向一个 2M 页面。

采用这种分页模式，VMM 可以使用高达 64G 的物理内存，达到支持 8G 物理内存的目标。

3.2.3 页表虚拟化思想

由于 VMM 管理所有系统资源，因此 VMM 也管理整个内存资源，它负责页式内存管理，维护虚拟地址到机器地址的映射关系。而 Guest OS 本身也有页式内存管理机制，因此有 VMM 的系统会比正常系统多一种地址从而多一层地址映射。进行地址映射时涉及到的三种地址：①客户机虚拟地址(GVA)；②客户机物理地址(GPA)；③主机物理地址(HPA)。映射关系如下^[16]：

- (1) Guest OS: $GPA = f(GVA)$ ，VMM: $HPA = g(GPA)$ 。
- (2) Guest OS 维护一套页表，负责 GVA 到 GPA 的映射；
- (3) VMM 维护一套页表，负责 GPA 到 HPA 的映射。

实际运行时，用户程序访问 GVA，经 Guest OS 的页表转换得到 GPA，再

由 VMM 介入，使用 VMM 的页表将 GPA 转换为 HPA。

由上所述，内存虚拟化的主要任务是实现地址空间的虚拟化，即客户机到主机的两次地址转换。其中 GVA->GPA 的转换是由客户机来维护，通常是客户操作系统通过 VMCS 中客户机状态域 CR3 指向的页表来指定；GPA->HPA 的转换是由 VMM 维护，在 VMM 将物理内存分配给客户机时 GPA->HPA 的转换就确定了，这个映射关系由 VMM 内部数据结构（即为 EPT 表）来记录。

3.2.4 内存隔离的实现方法

根据 VMM 系统的需求分析，VMM 内存隔离的实现可以分为：

- （1）创建 VM 时分配固定物理内存区域；
- （2）客户系统在内存寻址时采用 EPT 页表地址转换机制；
- （3）使用 TLB/VPID 缓存器。

通过在创建 VM 时进行内存空间分区，即在为 VM 进行 malloc() 申请线性空间时，对申请的空间立即进行地址映射，可以使 VM 的线性地址空间与固定的物理地址空间绑定，从而实现在内存区间上对不同客户系统的隔离；在虚拟机运行时需要对内存页进行修改，但是物理内存是多个虚拟机共享的，因此不能让虚拟机直接访问物理地址，所以采用虚拟机客户系统物理地址，利用 EPT 页表实现虚拟机物理地址到主机物理地址的转换过程，而 EPT 页表是由 VMM 来维护的，从而在虚拟机内存寻址上实现了内存隔离的目的；在 TLB 中加入 VPID 的标识，当进行地址转换时，TLB 中缓存的转换表可以直接定位到对虚拟机 VPID 的条目，如果 TLB 命中失败，只需要刷新对应 VPID 的条目，这样提高地址转换时间。

3.3 基于 EPT 页表的地址转换方法

在原有的 CR3 页表地址映射的基础上，Intel VT-x 技术引入了 EPT 页表来实现 GPA->HPA 映射，这样 GVA->GPA->HPA 两次地址转换都由 CPU 硬件自动完成。EPT 页表和普通的 Linux 页表具有一样的结构，不同的是 EPT 表由宿主机创建维护，由 CPU 硬件来查询使用。

3.3.1 EPT 机制工作原理及实现

内存虚拟化解决方案是通过两次地址转换来支持地址空间的虚拟化^[1]，即要实现客户机虚拟地址 GVA->客户机物理地址 GPA->宿主机物理地址 HPA 的转换。在这两次映射关系中，第一次 GVA->GPA 映射发生在客户机内，第二次 GPA->HPA 映射发生在宿主机内，其映射结构有宿主机来记录管理^[17]。

EPT（Extended Page Table，扩展页表）技术是内存虚拟化技术中硬件设备

提供的支持。在没有硬件内存虚拟化支持的情况下，KVM 实现内存虚拟化的方法是影子页表技术。影子页表技术为 Guest OS 的每个页表维护一个“影子页表”，并将合成后的映射关系写入到“影子”中，Guest OS 的页表内容则保持不变。最后，宿主机将影子页表交给内存管理模块（Memory Management Unit, MMU）进行地址转换。而 EPT 机制通过使用硬件支持内存虚拟化技术，使其能在原有的页表的基础上增加一个 EPT 页表，通过这个页表能够将 Guest OS 的物理地址直接翻译为主机的物理地址，从而减少整个内存虚拟化所需的代价。

EPT 页表存在于 VMM 内核空间中，由 VMM 维护。EPT 页表的基地址是由 VMCS“VM-Execution”控制域的 Extended page table pointer 字段指定的，它包含了 EPT 页表的宿主机系统物理地址。EPT 页表各级表项的格式相同，如图 3-3 所示^[19]。



图 3-3 页表项格式图

EPT 扩展页表技术解决了使用影子页表存在的实现复杂、开发调试困难的问题，它直接在硬件上支持 GVA->GPA->HPA 的两次地址转换，降低了内存虚拟化的难度，并提高了系统性能。图 3-4 是 EPT 的工作原理图^[9]。

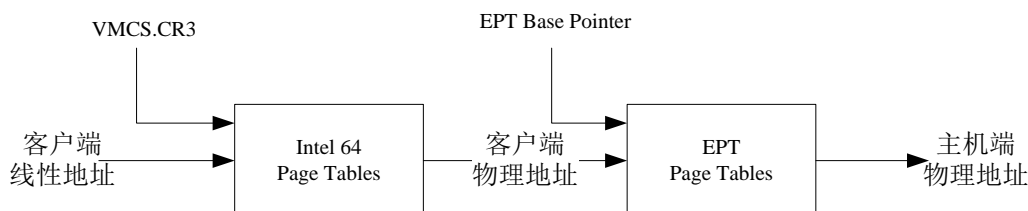


图 3-4 EPT 工作原理图

3.3.2 基于 EPT 技术的地址转换过程

本文结合一般的 Linux 分页地址转换过程，分析了 EPT 表的地址转换过程，如图 3-5 所示。由该图可知，CPU 完成一次客户机寻址的基本过程如下：

CPU 首先查找客户机 CR3 指向的一级页表。由于客户机 CR3 指向的是 GPA，因此 CPU 需要通过 EPT 页表进行客户机 CR3 GPA->HPA 的转换。如果在 EPT 页表中没有查找结果，则 CPU 抛出 EPT Violation 异常，由客户机退出到宿主机环境来处理，完成对页表项对应物理内存页的申请和 EPT 表页表项填充工作。处理完后，由宿主机再次陷入客户机中，CPU 再次对同一个 GPA 进行地址转换，通过这次转换可以获得客户机一级页表的地址。然后 CPU 根据 GVA 和一级页表表项内容，得出客户机二级页表的 GPA。同样，CPU 通过查

询 EPT

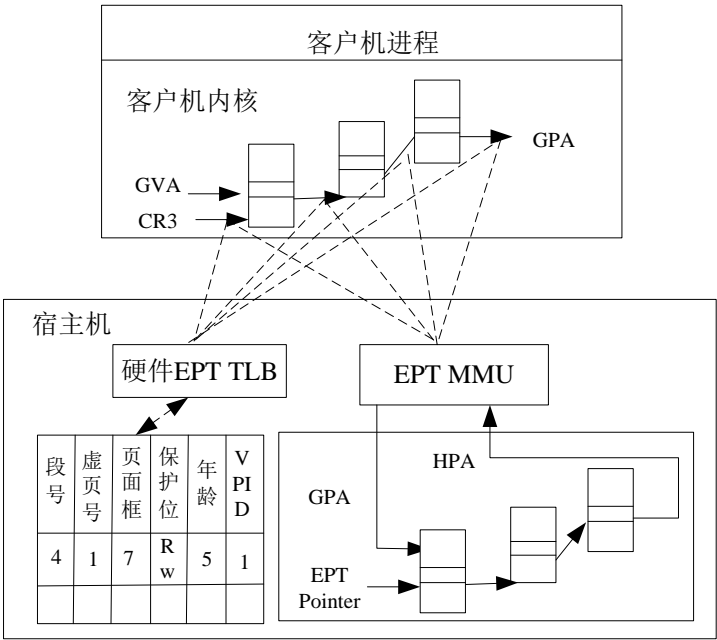


图 3-5 使用 EPT 进行 GPA-HPA 地址转换的过程

页表得出客户机二级页表 GPA->HPA 的地址转换。按照上述查找过程，CPU 会依次查找每一级页表的地址，最后获得 GVA 对应的 GPA，然后通过查询 EPT

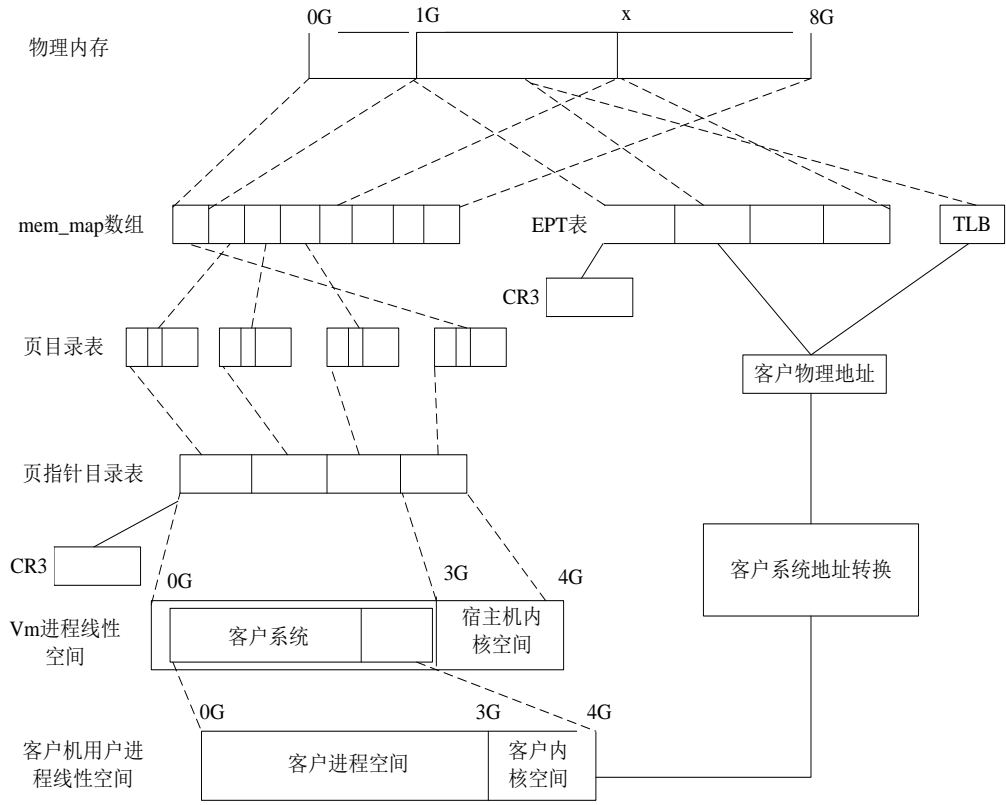


图 3-6 客户机进程与物理内存关系图

页表获得 HPA。图 3-6 描述了是通过物理内存与宿主机、客户机进程地址空间的关系，展示了客户机进程访问物理内存的地址转换过程。

3.3.3 EPT 页表地址转换机制的优化方法

通过上面分析可以得知，客户机的每个 GPA 都需要通过查询 EPT 表来完成 GPA->HPA 的地址转换。而 Linux 页表采用懒惰式建表，即 EPT 页表是动态创建的，客户机运行开始时 EPT 是空页表，第一次使用发生 EPT Violation 时才开始建立映射。当访问一个地址时，如果这个地址处于一个新的页地址范围内，客户机就要退出，进入宿主机中进行相应处理。若需要不断访问新地址，这种客户机与宿主机的切换就会频繁发生。在切换中 CPU 要做大量的工作：保存客户机上下文的各种状态值、加载宿主机上下文、分析客户机退出原因等，这些处理会产生很大的时间开销。本文采用为客户机预分配固定的内存（如 3.5GB）的方法，可以一次性为客户机建立完整的 EPT 表，减小因查询无结果导致客户机与宿主机切换的发生频率。

实现方法如下：修改 KVM 模块，优化 EPT 表机制。首先通过调用内核 `__VCPU_run()` 函数启动一个虚拟 CPU（VCPU），接着该 VCPU 运行客户操作系统。在客户系统运行过程中，总会有事件导致发生退出异常（fault），如果造成 fault 的原因是 EPT 异常（`ept_violation`），那么下一步就要在宿主机中进行 `ept_violation` 处理，本文对 `ept_violation` 处理进行优化，添加对首次缺页处理的判断。如果是第一次 EPT 的缺页处理，即客户机运行初期，利用内核函数功能获取预设的客户机物理内存大小，然后在该地址空间范围内对每个相隔 4K（一页）的客户物理地址进行缺页模拟，以完成 EPT 页表的填充，进一步实现地址转换；如果不是第一次缺页处理，则进行一次一页地址范围的缺页处理，最后返回客户机。EPT 表机制优化实现流程如图 3-7 所示。

3.4 详细设计

根据第 3.3.3 节提出的 EPT 页表优化方法，首先分析了在使用 EPT 进行地址转换过程中出现的主要数据，然后根据虚拟机的具体实现给出优化方案中 EPT 页表的初始化函数调用关系，最后给出采用优化方案后客户机创建过程中为客户机分配固定物理内存的流程，以及介绍调用的主要函数。

3.4.1 主要的数据结构

实现 EPT 机制优化方案要依赖 Linux 和 KVM 中与内存、EPT、虚拟 CPU、进程有关的数据结构，如 `PAGE`、`VMCS`、`MM_STRUCT` 等，这些结构包含了页面、VCPU、进程的具体控制和数据信息。优化方案是基于对这些数据结构

的操作实现的。

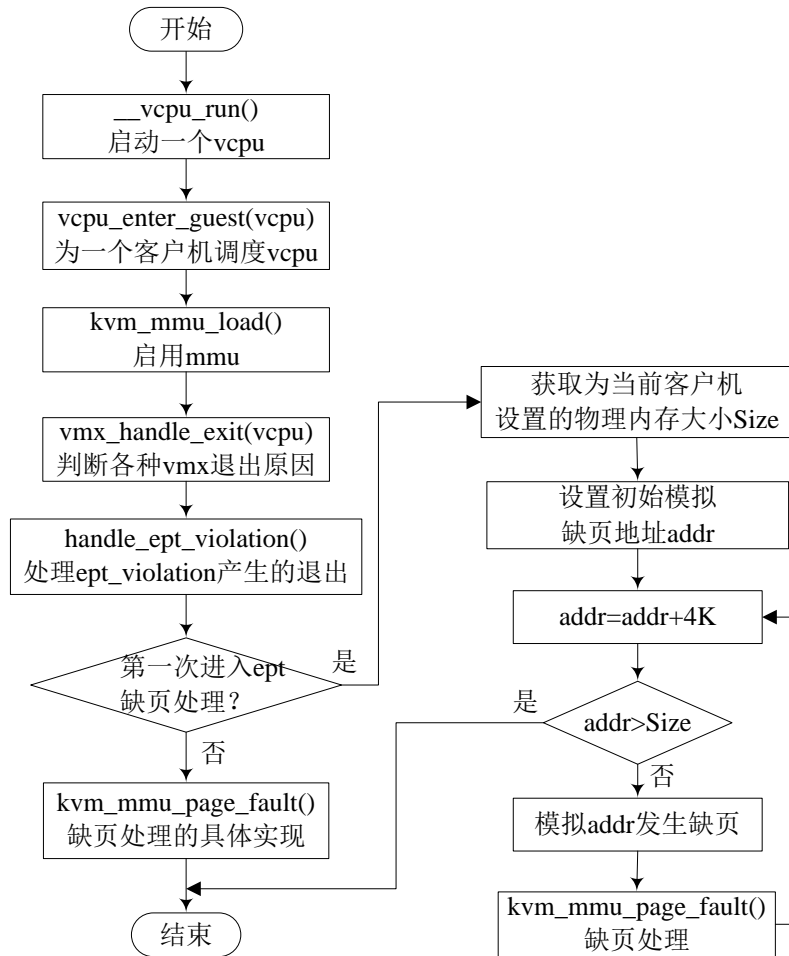


图 3-7 EPT 表机制优化实现流程

3.4.1.1 页面描述结构

页面描述结构与页面、物理页框对应，是 Linux 中对内存管理的重要数据结构，保存 Linux 内存管理中与一个页框对应的控制信息，应用于申请、分配内存操作，本文实现内存申请使用 4K 的页面大小。页面描述结构成员如下：

```

struct page {
    page_flags_t flags; 页标志字
    atomic_t _count;    页引用计数器
    atomic_t _mapcount;  页映射计数器
    unsigned long private; 私有数据指针
    struct address_space *mapping; 该页所在地址空间描述结构指针，用于内容为文件的页帧
    pgoff_t index;      该页描述结构在地址空间 radix 树 page_tree 中的对象索引号即页号
    struct list_head lru; 最近最久未使用 structslab 结构指针链表头变量

```

```
};
```

3.4.1.2 虚拟内存状态描述结构

虚拟 CPU 相关状态的数据结构，保存虚拟 CPU 相关寄存器和内容和虚拟 CPU 相关的控制信息，本文涉及到结构中的 EPT_POINTER，GUEST_PHYSICAL_ADDRESS，EXIT_QUALIFICATION 等成员。VMCS 描述结构的成员如下：

```
struct vmcs {
    u32 revision_id;
    u32 abort;
    char data[0];
};

enum vmcs_field {
    VIRTUAL_PROCESSOR_ID                = 0x00000000,
    EPT_POINTER                          = 0x0000201a, //EPT 表的入口地址，
创建表时该值修改
    EPT_POINTER_HIGH                    = 0x0000201b,
    GUEST_PHYSICAL_ADDRESS              = 0x00002400,
    GUEST_PHYSICAL_ADDRESS_HIGH        = 0x00002401,
    VMCS_LINK_POINTER                  = 0x00002800,
}
```

每个 VMCS 对应一个虚拟 CPU，主要供 CPU 使用。CPU 在发生 VM-Exit 和 VM-Entry 时都会自动查询和更新 VMCS，EPT 页表的基地址是由 VMCS “VM-Execution” 控制域的 Extended page table pointer 字段指定的，它包含了 EPT 页表的主系统物理地址。

3.4.1.3 进程地址空间的总控制结构

mm_struct 用来描述一个进程的虚拟地址空间，在 Linux 中每个进程对应一个该结构，它包含了进程的所有控制信息以及进程占用物理内存空间相关的关联信息，主要用到的成员是 mmap。mm_struct 结构的成员如下：

```
struct mm_struct {
    struct vm_area_struct * mmap;      vmarea 列表
    struct vm_area_struct * mmap_cache; /* last find_vma result */
    pgd_t * pgd; //指向页表的 寻址时装入 cr3 中
    atomic_t mm_users;                /* How many users with user space? */
}
```

```

    atomic_t mm_count;           /* How many references to "struct mm_struct"
(users count as 1) */
    int map_count;               /* number of VMAs */
};

```

3.4.1.4 KVM 中用户空间内存区描述结构

KVM 模块中描述 KVM 用户空间区域，通过该结构为 VM 分配物理内存，结构中用到了成员 slot、guest_phys_addr、memory_size、userspace_addr，来获取客户机的物理内存大小、地址范围等。kvm_userspace_memory_region 描述结构成员如下：

```

struct kvm_userspace_memory_region {
    __u32 slot;
    __u32 flags;
    __u64 guest_phys_addr; //客户系统物理地址的起始地址
    __u64 memory_size; //客户系统物理内存大小
    __u64 userspace_addr; /* start of the userspace allocated memory */
};

```

3.4.2 实现步骤

在 VM 创建流程中完成 EPT 页表的初始化步骤。首先实现初始化 EPT 页表的实现机制分析，然后在创建 VM 并为虚拟机申请宿主机虚拟空间时完成对 EPT 页表的初始化工作。

3.4.2.1 初始化 EPT 页表

根据 EPT 的机制原理，EPT 页表存放了客户机物理地址和宿主机物理地址的映射关系。在 KVM 实现方案中 EPT 页表是动态写入的，只有在客户机运行过程中客户机物理地址没有对应宿主机物理地址，地址无法访问且合法的情况下才对该地址所在的页进行映射。优化后，需要在虚拟机运行前为客户机分配物理内存，建立完整的 EPT 页表地址映射关系，也即初始化 EPT 页表。

图 3-8 是初始化 EPT 表的函数调用关系。

TLB entries 通过加入虚拟处理器 ID (Virtual Processor ID, VPID) 而产生微妙的改变。每一个 TLB entry 都为内存页面缓存一个虚拟到物理地址的转换，该转换具体到一个既定的进程和虚拟机 (Virtual Machine, VM)。当处理器在虚拟客户端和主机环境之间切换时，Intel 过去的处理器都会不断刷新 TLB 以确保进程只访问被允许访问的内存。而 VPID 会追踪 TLB 中某个转换 entry 所联合是哪个虚拟机，这样当 VM exit 和 VM re-entry 发生时，TLB 即使

不刷新也能够保证安全度。如果一个进程试图访问不属于自己的地址转换，则会直接简单地在 TLB 发生命中失败而不是非法访问^[3]。

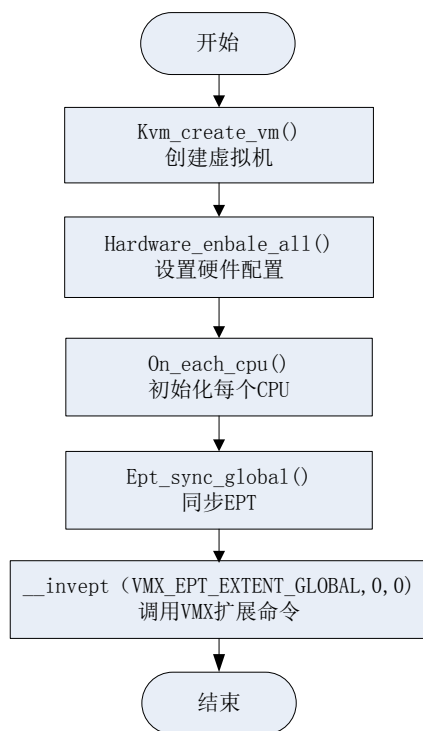


图 3-8 初始化 EPT 表过程的函数调用关系

3.4.2.2 创建客户机 VM

创建 VM，为客户机申请主机虚拟内存空间。首先要获取用户空间中为客户机设定的物理内存大小，然后在宿主机的 userspace_memory_region 中申请虚拟地址空间，调用 Linux 内核函数 do_mmap()进行虚拟内存的具体实现，最后

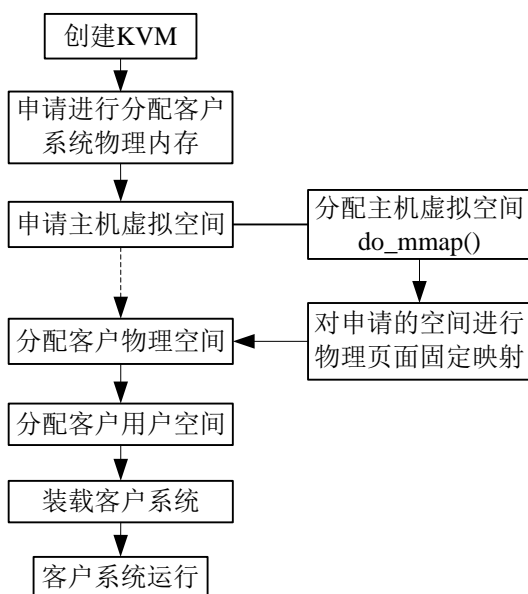


图 3-9 创建 vm 内存分配流程图

完成虚拟空间的物理页面映射。上图 3-9 为创建虚机时进行内存分配的流程图。

创建 VM 并分配虚拟机内存过程中需要调用的相关函数如表 3-1 所示。

表 3-1 创建 VM 相关的函数

函数名	函数原型	功能描述
kvm_create_phys_mem	void *kvm_create_phys_mem(kvm_context_t kvm, unsigned long phys_start,unsigned long len, int log, int writable)	创建 VM 时申请内存分配入口函数
__kvm_set_memory_region	int __kvm_set_memory_region(kvm, mem, user_alloc);	为客户机申请虚拟内存空间入口函数
do_mmap	do_mmap(NULL, 0, npages * PAGE_SIZE,PROT_READ PROT_WRITE, MAP_PRIVATE MAP_ANONYMOUS, 0);	申请主机虚拟空间的最终实现函数，但不进行实际物理映射
handle_ept_violation	static int handle_ept_violation(struct kvm_vcpu *vcpu)	KVM 对 ept 缺页退出逻辑的处理函数

第 4 章 基于 KVM 的处理器虚拟化及虚拟机绑定研究

Intel VT-x 技术扩展了传统的 IA32 处理架构，提供了处理器虚拟化的硬件支持。Intel VT-x 为了更好地支持 CPU 虚拟化，引入虚拟机控制结构 VMCS(Virtual-Machine Control Structure)。VMCS 主要由 CPU 操作，它是保持在内存中的数据结构，一个虚拟 CPU 对应一个 VMCS，即一个 VMCS 保存的内容代表了一个虚拟 CPU (VPCU) 的存在。

根据需求，虚拟机绑定要实现创建一个客户机并与指定物理 CPU 绑定，并且在客户机运行过程中，CPU 不发生迁移。以虚拟机为目标，分别对处理器虚拟化的实现原理、VMCS 描述符、VPCU 描述符以及 VM-Entry 和 VM-Exit 等进行研究分析。

4.1 虚拟机与客户机间的模式切换分析

虚拟机运行过程中所有的内容在 VMM 的根模式和客户机的客户模式不断的切换中完成，根模式要频繁处理在客户模式运行时遇到中断、缺页等操作。VMCS 类似于虚拟寄存器，它包含了虚拟 CPU 的相关寄存器的内容和虚拟 CPU 相关的控制信息，可以通过指令配置。修改 VMCS 可以实现对虚拟 CPU 的控制。

4.1.1 VMCS 数据结构

从计算机系统资源的角度看，处理器呈现给软件的接口就是一系列的指令或指令集和一些寄存器，包括通用运算的寄存器和用于控制处理器行为的状态和控制寄存器。

通用计算机平台处理器包括的寄存器有^[3]：

- (1) 控制寄存器 CR0, CR3, CR4。
- (2) 调试寄存器 DR7。
- (3) RSP、RIP 和 RFLAGS。
- (4) CS、SS、DS、ES、FS、GS、LDTR、TR 以及影子段描述符寄存器。
- (5) GDTR、IDTR。
- (6) MSR: IA32_SYSENTER_CS、IA32_SYSENTER_ESP
和 IA32_SYSENTER_EIP。

CPU 在发生 VM-Exit 和 VM-Entry 时都会自动查询和更新 VMCS，VMM 通过指令来配置 VMCS，进而影响 CPU 的行为。VT-x 引入的新指令，包括 VMLAUCH/VMRESUME 用于发起 VM-Entry，VMREAD/VMWRITE 用于配置

VMCS 等。

VMCS 做为一个保持 VCPU 相关信息的数据结构，有具体的格式和内容。在 Intel VT-x 技术中，它是一个最大不超过 4KB 的内存块，并且是与 4KB 对齐。VMCS 内容格式中各个域描述如下：

(1) 位 0-3: VMCS 版本标识，即 VMCS 数据格式的版本号。

(2) 位 4-7: VMX 中止指示，VM-Exit 执行不成功时，VMX 中止，处理器会将 VMX 中止的原因写入该位置。

(3) 位 8~:VMCS 数据域，该域格式和处理器相关，不同型号处理器有不同格式，具体格式由 VMCS 版本标识决定。

VMCS 数据域是 VMCS 最重要的域，也是处理器寄存器的主要数据，它主要包括 6 大类信息^[15]：

(1) 客户机状态域：保存客户机运行时即客户模式时的 CPU 状态。当 VM-Exit 发生时，CPU 把当前状态存入客户机状态域；当 VM-Entry 发生时，CPU 从客户机状态域恢复状态。

(2) 宿主机状态域：保存 VMM 运行时，即根模式时的 CPU 状态。当 VM-Exit 发生时，CPU 从宿主机状态域恢复 CPU 状态。

(3) VM-Entry 控制域：控制 VM-Entry 的过程。

(4) VM-Execution 控制域：控制处理器在 VMX 非根模式下的行为。

(5) VM-Exit 控制域：控制 Vm-Exit 的过程。

(6) VM-Exit 信息域：保存 VM-Exit 原因和其他信息。

4.1.2 VM_Exit 流程

在客户机执行过程中若发生中断、或者执行了敏感指令等事件，将会引发 VM-Exit，导致 CPU 从客户模式切换到根模式，使处理器从客户机退出到 VMM 中执行。在 VM-Exit 流程中同样要设置 VMCS 的各个域，首先是要记录产生 VM-Exit 的原因，作为进入 VMM 后进行 handle_exit 的依据，然后 CPU 状态通过硬件机制保存到 VMCS 客户机状态域，一些 MSR 寄存器也会保存到 MSR-store 区域里。最后会从 VMCS 宿主机状态域中将宿主机状态恢复到对应的寄存器，进行模式切换。从客户机退出到 VMM 的具体流程如图 4-1 所示。

在 VMM 运行过程中，处理完 VM-Exit 后，会再次通过执行 VMLAUNCH/VMRESUME 命令引发 VM-Entry 过程，进入客户机执行。这一过程频繁发生并不断重复，在这个切换过程中 VMM 完成了虚拟机管理机的各个功能。

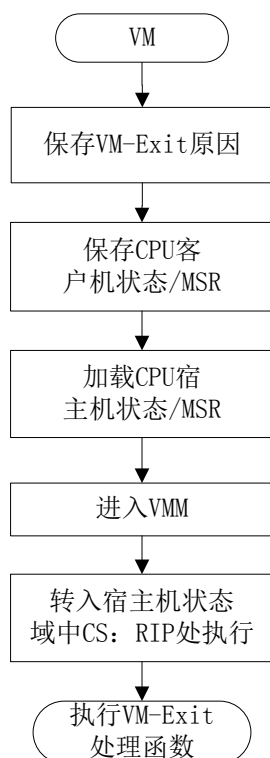


图 4-1 VM-Exit 流程图

4.1.3 VM_Entry 流程

VM-Entry 是指物理 CPU 从根模式转换到客户模式，即是 CPU 从 VMM 进入客户机执行。VMM 在设置好 VMCS 各个域的内容后，通过执行 VMX 扩展命令 VMLAUNCH/VMRESUME 来完成状态切换过程。

在处理器执行 VMLAUNCH/VMRESUME 进行 VM-Entry 时，要进行一系列的处理才能完成这个过程，具体实现过程如图 4-2 所示。

4.2 虚拟 CPU 的结构及运行机制

4.2.1 VCPU 结构体分析

硬件虚拟化采用 VCPU (Virtual CPU)，描述符来描述虚拟 CPU，它在实现上是一个结构体 (struct kvm_VCPU VCPU)。VCPU 一般可以划分为两个部分：①VMCS 数据结构中存储的由硬件使用和更新的内容，②VMCS 没有保存而由 VMM 使用和更新的内容。

由 VMCS 保存的内容一般包括以下几个重要的部分：

(1) VCPU 标识信息：标识 VCPU 的一些属性。

(2) 虚拟寄存器信息：虚拟的寄存器资源，开启 Intel VT-x 机制时，虚拟寄存器的数据存储在 VMCS 中。

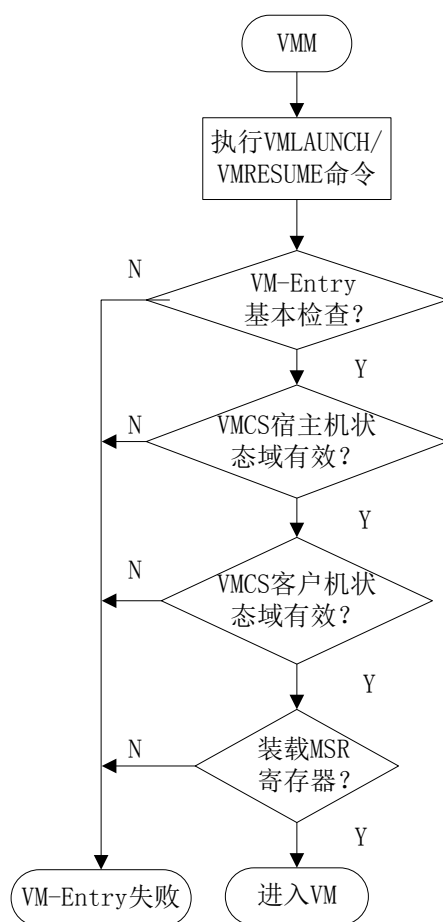


图 4-2 VM-Entry 流程图

(3) VCPU 状态信息：标识 VCPU 当前的状态。

(4) 额外寄存器/部件信息：存储没有 VMCS 中没有保存的一些寄存器或者 CPU 部件。

(5) 其他信息：存储 VMM 进行优化或者额外信息的字段。

在具体实现中，VMM 创建 VM 时，首先为虚拟机创建 VCPU，然后由 VMM 来调度运行。

4.2.2 VCPU 的创建过程

创建一个 VCPU 即是创建 VCPU 描述符，VCPU 描述符本质是一个结构体，创建 VCPU 描述符就是分配这个结构对应大小的内存。在 Linux 中为了提高数据的索引速度，减小内存占用空间，一般描述符有多级结构体来构成。在初始化一个描述符时，它的每一级结构体都要进行相应的初始化，分配相应大小的内存。在 VCPU 创建过程时，需要对 VCPU 描述符的每个组成部分进行初始化，具体过程如图 4-3 所示。

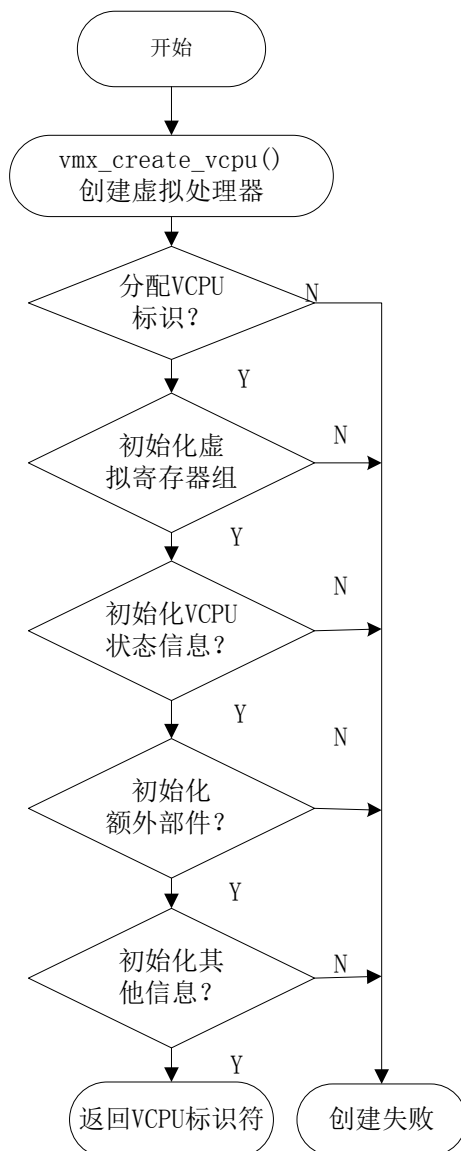


图 4-3 VCPU 创建流程图

4.2.3 VCPU 的运行

VCPU 的运行是通过 VMM 的调度程序来实现的。VCPU 在创建并初始化后，即可进入等待状态，调度程序按照一定的调度策略算法选择 VCPU 运行，实际过程是把选择的 VCPU 切换到物理 CPU 上执行，即将 VCPU 与物理 CPU 的处理器上下文切换。

虚拟机处理器上下文切换包括以下步骤：

- (1) VMM 保存自己的上下文，主要保存 VMCS 不保存的寄存器，宿主机状态域以外的内容。
- (2) VMM 把保存在 VCPU 中由软件处理的上下文加载到物理处理器中。
- (3) VMM 执行 VMLAUNCH/VMRESUME 指令，引发 VM-Entry，物理处

理器自动将 VCPU 上下文中的 VMCS 部分加载到处理器。

VMM 与 VCPU 进行处理器上下文切换的具体流程如图 4-4 所示。

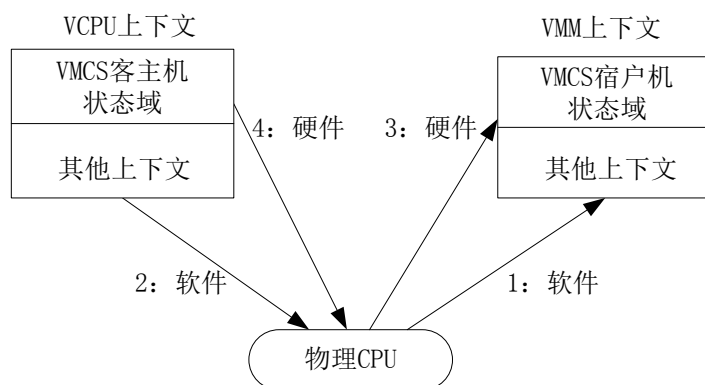


图 4-4 VMM 与 VCPU 上下文切换

4.3 虚拟机绑定的设计方案

在创建虚拟机时，处理器虚拟过程中用加入控制代码的方式将一个 VMCS 与一个物理 CPU 核心绑定在一起，并结合 Linux 进程绑定的方法可以实现一个单核客户系统与一个 CPU 核心绑定的效果。

4.3.1 研究目标

根据需求，要在多核计算机平台下，实现每个 Guest OS 在一个虚拟的单核处理器上运行的效果，具体的设计目标如下：

- (1) 使 VMM 创建的 Guest OS 进程在指定的核上运行；
- (2) 实现 Guest OS 进程的绑定，使两个 Guest OS 独立运行在指定的核上，并在整个运行期间不发生迁移。
- (3) 当两个 Guest OS 独立运行在虚拟的单核平台上时，互不影响，实现 CPU 核的隔离。

4.3.2 设计思路

VMM 采用 KVM 虚拟化解决方案实现系统的虚拟化。本文采用的 Linux 内核版本是 2.6.33.7，该版本已经把 KVM 集成到内核里。而 QEMU 是运行在 User 模式的应用程序，因此，首先应该分析清楚 KVM 和 QEMU 的工作方式。

在 Intel VT 技术根模式/非根模式下，KVM 虚拟机充分利用硬件的支持，实现了系统的虚拟化。根据 KVM 的运行级别，可把 KVM 分为两部分，分别是运行于 Kernel 模式的 KVM 内核模块和运行于 User 模式的 QEMU 模块。这里的 Kernel 模式和 User 模式实际上分别指的是 VMX 根模式下的特权级 0 和特权级 3。另外，KVM 将客户机所在的运行模式称为客户（Guest）模式，即 VMX

的 VMX 非根模式。

利用 VT-x 技术的支持, KVM 中的每个虚拟机可具有多个虚拟处理器 VCPU, 每个 VCPU 对应一个 QEMU 的线程, 正是 QEMU 线程支持着 KVM 虚拟机的运行。没有硬件支持的纯 QEMU 虚拟机和利用硬件支持实现虚拟化的 KVM 虚拟机的差异也就在这里。为了更加形象地说明 KVM 和 QEMU 间的关系, 可称它为 KVM 线程, 而创建客户机的应用程序 QEMU, 称为 QEMU 进程。QEMU 应用程序在创建或者说加载一个 Guest OS 后, 就是 KVM 线程在运行。VCPU 的创建、初始化、运行以及退出处理都在 KVM 线程上下文中进行, 这些需要 Kernel、User 和 Guest 三种模式相互配合。

KVM 线程与 KVM 内核模块间以 ioctl 的方式进行交互, 指示 KVM 内核模块进行 VCPU 的创建和初始化等操作, 主要包括 VMM 创建 VCPU 运行所需的各种数据结构以及初始化, 如 VMCS。而 KVM 内核模块与客户机之间通过 VM Exit 和 VM entry 操作进行切换。

初始化工作完成之后, KVM 线程以 ioctl 的方式向 KVM 内核模块发出运行 VCPU 的指示, 后者执行 VM entry 操作, 将处理器由 kernel 模式切换到 Guest 模式, 转而运行客户机。但此时仍处于 KVM 线程上下文中, 且正在执行 ioctl 系统调用的 kernel 模式处理程序。

客户机在运行过程中, 如发生异常或外部中断等事件, 或执行 I/O 操作, 可能导致 VM exit, 将处理器状态由 Guest 模式切换回 Kernel 模式。KVM 内核模块检查发生 VM exit 的原因, 如果 VM exit 是由 I/O 操作导致的, 则执行系统调用返回操作, 将 I/O 操作交给处于 User 模式的 QEMU 进程来处理。这时 QEMU 会创建一个新的 QEMU 线程来完成这些 I/O 处理, 之后再次执行 ioctl, 指示 KVM 切换处理器到 Guest 模式, 恢复客户机的运行。如果 VM exit 由于其它原因导致, 则由 KVM 内核模块负责处理, 并在处理后切换处理器到 Guest 模式, 恢复客户机的运行^[27]。

由以上分析可知, Guest OS 虚拟单核平台的实现, 与 QEMU 应用程序、KVM 线程、异步 IO 事件处理这三个方面有着密切的联系。根据对 KVM 虚拟机初始化、VCPU 创建和执行过程的研究, 要实现虚拟机进程与指定处理器核心绑定, 可以从 VMCS 描述符、KVM 进程两方面入手:

(1) VMCS 是与 VCPU 一一对应的, 将 VMCS 与指定处理器核心绑定, 可以实现 VCPU 与指定 CPU 绑定;

(2) KVM 进程在 VMM 中, 与普通 Linux 程序是一样的, 采用进程绑定的技术手段可以实现 KVM 进程与指定 CPU 绑定。

4.3.3 虚拟机绑方法设计

由前面的分析可知,要实现 Guest OS 独自在一个虚拟的单核平台上运行,就需要从 VMCS、QEMU 应用程序、KVM 线程、异步 IO 事件处理着手。如果这四方面的操作和处理能在指定的核上完成,那么虚拟的单核平台也就得以实现。通过研究表明,VMX 的扩展指令 VMPTRLD 等可以实现 VMCS 与物理 CPU 绑定;CPU 的硬亲和性(Affinity)机制可以解决剩余三个问题。只要对 VMCS 进行控制,完成 KVM 进程 Affinity 属性的正确设置,就可将 Guest OS 进程绑定到指定的核上,使其在整个运行期间不发生迁移,如图 4-5 所示^[18]。

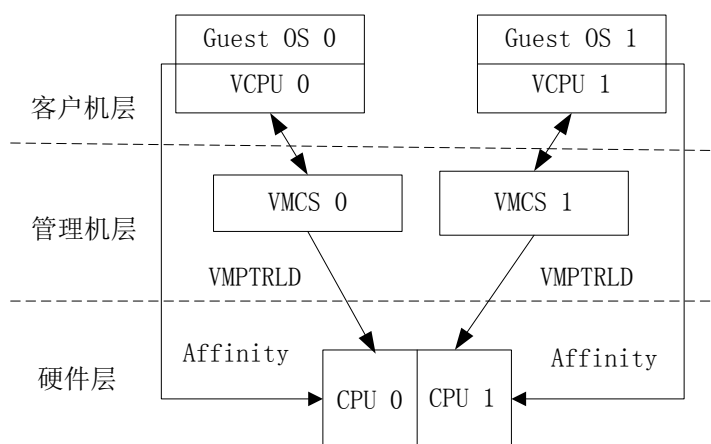


图 4-5 Guest OS 与 CPU 核的绑定

根据 KVM 原理、机制及相关技术的分析,可以通过修改、优化 QEMU 源代码,实现特定的 KVM 虚拟机进程与指定处理器核心绑定的目标。在创建客户机的时候实现对客户机的绑定,使客户机只能在指定核上的运行,不发生迁移。

4.4 实现方法和步骤

根据设计方案,需要对 KVM、QEMU 源代码进行修改,以完成增加-cpuid 的命令选项、参数解析,以及相关的绑定操作。

4.4.1 主要数据结构

源代码的修改主要涉及以下数据结构和选项变量^[10]:

(1) 选项枚举变量

该选项变量主要是列出 QEMU 命令的所有选项以及选项的属性,如选项名称、参数、帮助等。枚举变量的成员如下:

```
enum {
#define DEF(option, opt_arg, opt_enum, opt_help)  opt_enum,
#define DEFHEADING(text)
```

```
#include "QEMU-options.h"
#undef DEF
#undef DEFHEADING
#undef GEN_DOCS
};
```

(2) 选项结构体变量

该变量主要是保存选项枚举变量中定义的选项，包括选项名称、选项是否有参数，以及选项参数的索引。QEMUOption 结构体的成员如下：

```
typedef struct QEMUOption {
    const char *name;           //选项名称
    int flags;                  //选项是否有参数
    int index;                  //选项参数索引
} QEMUOption;
static const QEMUOption QEMU_options[] = {
    { "h", 0, QEMU_OPTION_h },
#define DEF(option, opt_arg, opt_enum, opt_help) \
    { option, opt_arg, opt_enum },
#define DEFHEADING(text)
#include "QEMU-options.h"
#undef DEF
#undef DEFHEADING
#undef GEN_DOCS
    { NULL },
};
```

在文件 vl.c 下的 main() 函数中，有一个处理选项的入口：switch(popt->index)，它将在执行时索引该参数的位置，并进行对应的参数处理。如果要添加新的选项，这两个选项变量都必须做出相应的修改。

4.4.2 进程绑定的实现

4.4.2.1 VMCS 的绑定实现

实现 VMCS 绑定需要重新加载 VCPU，并从用户空间程序获取 VMCS 要绑定的目标处理器序号，然后进行加载 VCPU 的安全条件检查，在 VMX 模块中获取 VCPU 对应的 VMCS 描述符，最后调用汇编语言执行 VMX 扩展指令，实现 VMCS 与物理 CPU 的绑定，并加载 VCPU。具体流程见图 4-6。

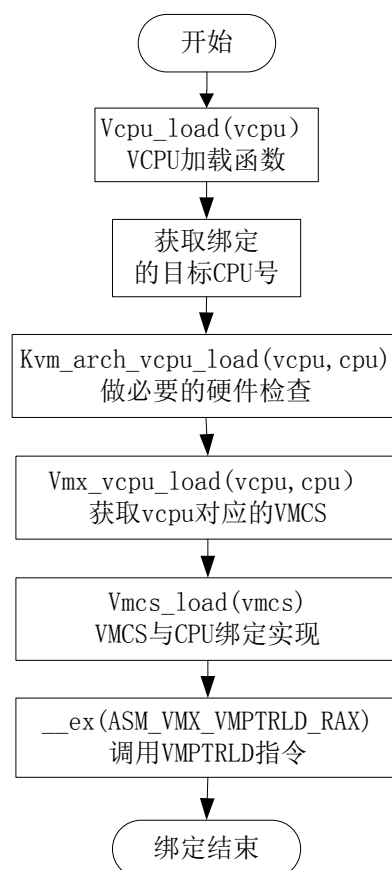


图 4-6 VMCS 绑定实现流程图

4.4.2.2 KVM 进程及异步 I/O 事件绑定

设计实现主要分为两部分：①从启动 QEMU 的命令行获取 cpuid 参数；②根据获得的 cpuid 参数，对处理器虚拟化设计分析中的三个方面进行绑定。主要实现过程如下：

- (1) 在选项枚举变量中添加选项参数：QEMU_OPTION_cpuid；
- (2) 在选项结构体变量 QEMU_options[] 中添加一项：
{"cpuid", HAS_ARG, QEMU_OPTION_cpuid}；
- (3) 在主函数中添加 case 分支：case QEMU_OPTION_cpuid，解析命令选项 cpuid，获取 cpuid 参数；
- (4) 根据获得的 cpuid 参数，在 ap_main_loop() 函数中，将 QEMU 进程绑定到指定的核上；
- (5) 根据获得的 cpuid 参数，在 ap_main_loop() 函数中，将 KVM 线程绑定到指定的核上；
- (6) 根据获得的 cpuid 参数，在 aio_thread() 函数中，将 aio 线程绑定到指定的核上。

4.5 实现效果

通过实现效果的测试和验证，在客户机的创建、客户机运行，以及客户机的异步 IO 事件处理中已实现了以下功能：

（1）在启动 Guest OS 时，指定创建 Guest OS 进程的 CPU 参数，可使创建客户机的进程在指定的核上运行；

（2）通过指定创建 Guest OS 进程的 CPU 参数，可实现对 Guest OS 的绑定，使其独立运行在指定的核上，并在整个运行期间不发生迁移。

第 5 章 测试及实验结果分析

本文在双核处理器、8G 内存的环境下进行研究和测试；分别设计了 EPT 页表优化方案、虚拟机绑定实现方案以及虚拟机内存申请三组测试内容，并进行了详细的测试分析和效果验证。

5.1 内存虚拟化中 EPT 页表优化测试

5.1.1 测试目标

验证 VMM 中能够运行两个单核 Guest OS，每个 Guest OS 能够独立分配 3.5GB 的物理内存空间。

5.1.2 测试方法和步骤

5.1.2.1 测试方法描述

1. 测试环境和前置条件

(1) 系统映像存放在 U 盘中，通过 U 盘引导系统，没有硬盘等外存储设备；

(2) HostOS: Debian (Linux2.6.33.7)，(测试原生单核系统性能时，按 f2 进入 BIOS，设置关闭 MutilCore, HT)；

(3) GuestOS vm0/vm1: Debian (Linux2.6.33.7)，两个客户机系统和 Host 系统配置相同；

(4) 测试目标程序: Binutils-2.20.1；

(5) 需要的磁盘空间大小: 148MB；

(6) 需要的软件工具: GNU make 3.81、gcc 4.4.5。

2. 测试方法

Host OS 与 Guest OS 配置相同，使用 HostOS 作为原生系统，编译软件的时间做为性能对比的参考值。同时运行两个 Guest OS，在这两个 Guest OS 中同时对相同的软件进行编译，记录测得编译时间。计算原生系统中编译软件的时间效率与 Guest OS 中编译软件的时间效率，进行对比，分析 VMM 的性能。

5.1.2.2 测试步骤

1. 测试命令

```
#apt-get install make
```

```
#apt-get install gcc
```

```
#cd binutils-2.20.1
```

Binutils-2.20.1#./configure

Binutils-2.20.1#time make

2. 测试过程

(1) 在 Host 中，执行测试命令，记录每次编译 binutils 的时间 t ;

(2) 在 Host 中/home 路径下执行./kvmboot.

脚本包括启动两台虚拟机的命令:

```
QEMU-system-x86-_64 -hda /home/g1/guest1.img -hdb /home/g1/live-rw -m
3584 -cpuid 0 -chardev pipe,id=ptest,path=/home/serial/path -device isa-serial,
chardev=ptest -net none -pcidevice host=${A_arr[0]} -pcidevice
host=${A_arr[65]} -vnc :0&
```

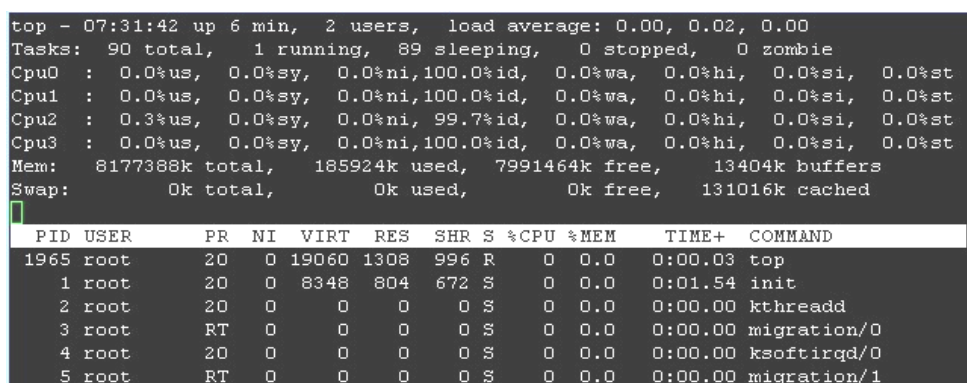
```
QEMU-system-x86-_64 -hda /home/g2/guest2.img -hdb /home/g2/live-rw -m
3584 -cpuid 1 -chardev pipe,id=ptest,path=/home/serial/path -device isa-serial,
chardev=ptest -net none -pcidevice host=${A_arr[5]} -pcidevice
host=${A_arr[60]} -vnc :1&
```

(3) 在两台虚拟机 VM0, VM1 中同时执行测试命令对 binutils 进行编译, 纪录编译时间 t_0 , t_1 ;

(4) 重复 (1), (2), (3) 步骤, 记录多次编译时间。

5.1.3 效果分析及结论

(1) 启动虚拟机前, Host 中的内存使用情况如图 5-1 所示 (top 命令截图)。



```
top - 07:31:42 up 6 min, 2 users, load average: 0.00, 0.02, 0.00
Tasks: 90 total, 1 running, 89 sleeping, 0 stopped, 0 zombie
Cpu0  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu1  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu2  :  0.3%us,  0.0%sy,  0.0%ni, 99.7%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu3  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   8177388k total, 185924k used, 7991464k free, 13404k buffers
Swap:   0k total,    0k used,    0k free, 131016k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
1965	root	20	0	19060	1308	996	R	0	0.0	0:00.03	top
1	root	20	0	8348	804	672	S	0	0.0	0:01.54	init
2	root	20	0	0	0	0	S	0	0.0	0:00.00	kthreadd
3	root	RT	0	0	0	0	S	0	0.0	0:00.00	migration/0
4	root	20	0	0	0	0	S	0	0.0	0:00.00	ksoftirqd/0
5	root	RT	0	0	0	0	S	0	0.0	0:00.00	migration/1

图 5-1 启动虚拟机前 Host 中内存的使用情况

(2) 在 Host 中进行编译测试, 编译 Binutils 的时间如图 5-2 所示。

执行命令: time make, 使用 time 工具统计编译时间, 最后得到 real、user、sys 三个时间值, 其中 real 时间是编译程序从开始运行到编译结束使用的时间, user 时间是编译程序在用户空间执行代码的时间, sys 时间是编译程序处于内核态执行代码的时间。

```

make[4]: Leaving directory `/home/test/decomp/binutils-2.20.1/ld'
make[3]: Leaving directory `/home/test/decomp/binutils-2.20.1/ld'
make[2]: Leaving directory `/home/test/decomp/binutils-2.20.1/ld'
make[1]: Nothing to be done for `all-target'.
make[1]: Leaving directory `/home/test/decomp/binutils-2.20.1'

real    1m14.459s
user    0m47.171s
sys     0m5.420s
root@debian:/home/test/decomp/binutils-2.20.1#

```

图 5-2 Host OS 中编译 Binutils 的时间

(3) 启动两个虚拟机后，Host 中的内存使用情况如图 5-3 所示。

```

top - 07:40:28 up 15 min, 2 users, load average: 0.01, 0.04, 0.01
Tasks: 92 total, 1 running, 91 sleeping, 0 stopped, 0 zombie
Cpu(s): 0.0%us, 0.1%sy, 0.0%ni, 99.9%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 8177388k total, 8127968k used, 49420k free, 59488k buffers
Swap: 0k total, 0k used, 0k free, 529408k cached

```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
26998	root	20	0	3697m	3.5g	2352	S	0	45.4	0:05.70	qemu-system-x86
26999	root	20	0	3697m	3.5g	2356	S	0	45.4	0:05.70	qemu-system-x86
1	root	20	0	8348	804	672	S	0	0.0	0:01.55	init
2	root	20	0	0	0	0	S	0	0.0	0:00.00	kthreadd
3	root	RT	0	0	0	0	S	0	0.0	0:00.00	migration/0
4	root	20	0	0	0	0	S	0	0.0	0:00.02	ksoftirqd/0

图 5-3 启动虚拟机后 Host 中内存的使用情况

(4) 优化前启动虚拟机后，虚拟机内存使用情况如图 5-4 所示。

```

Tasks: 89 total, 1 running, 88 sleeping, 0 stopped, 0 zombie
Cpu(s): 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 8179420k total, 813672k used, 7365748k free, 117624k buffers
Swap: 0k total, 0k used, 0k free, 452252k cached

```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
3042	root	20	0	3695m	173m	2312	S	0	2.2	0:02.23	qemu-system-x86
1	root	20	0	8348	808	676	S	0	0.0	0:01.52	init
2	root	20	0	0	0	0	S	0	0.0	0:00.00	kthreadd
3	root	RT	0	0	0	0	S	0	0.0	0:00.00	migration/0
4	root	20	0	0	0	0	S	0	0.0	0:00.02	ksoftirqd/0
5	root	RT	0	0	0	0	S	0	0.0	0:00.00	migration/1
6	root	20	0	0	0	0	S	0	0.0	0:00.02	ksoftirqd/1
7	root	RT	0	0	0	0	S	0	0.0	0:00.00	migration/2

图 5-4 优化前客户机内存的分配情况

(5) 在 Guest OS1 与 Guest OS2 上分时进行编译测试，二者分别编译 Binutils 的时间如图 5-5、图 5-6 所示。

对 Guest OS1 和 Guest OS2 分时测试，验证两个 Guest OS 相互之间的影响，测试结果显示分时测试时，两个 Guest OS 的编译时间接近，并比 Host OS 的编译时间多花费 5 秒钟。

```

make[4]: Leaving directory `/home/test/binutils-2.20.1/ld'
make[3]: Leaving directory `/home/test/binutils-2.20.1/ld'
make[2]: Leaving directory `/home/test/binutils-2.20.1/ld'
make[1]: Nothing to be done for `all-target'.
make[1]: Leaving directory `/home/test/binutils-2.20.1'

real    1m19.735s
user    0m48.835s
sys     0m7.784s
root@Guest OS1:/home/test/binutils-2.20.1# _

```

图 5-5 Guest OS1 单独编译 Binutils 的时间

```

make[4]: Leaving directory `/home/test/binutils-2.20.1/ld'
make[3]: Leaving directory `/home/test/binutils-2.20.1/ld'
make[2]: Leaving directory `/home/test/binutils-2.20.1/ld'
make[1]: Nothing to be done for `all-target'.
make[1]: Leaving directory `/home/test/binutils-2.20.1'

real    1m19.482s
user    0m48.863s
sys     0m7.572s
root@Guest OS2:/home/test/binutils-2.20.1# _

```

图 5-6 Guest OS2 单独编译 Binutils 的时间

(6) 在 Guest OS1 和 Guest OS2 中同时进行编译测试，二者编译 Binutils 的时间如图 5-7、图 5-8 所示。

```

make[4]: Leaving directory `/home/test/binutils-2.20.1/ld'
make[3]: Leaving directory `/home/test/binutils-2.20.1/ld'
make[2]: Leaving directory `/home/test/binutils-2.20.1/ld'
make[1]: Nothing to be done for `all-target'.
make[1]: Leaving directory `/home/test/binutils-2.20.1'

real    1m26.884s
user    0m54.687s
sys     0m8.345s
root@localhost:/home/test/binutils-2.20.1# _

```

图 5-7 Guest OS1 编译 Binutils 的时间

```

make[4]: Leaving directory `/home/test/binutils-2.20.1/ld'
make[3]: Leaving directory `/home/test/binutils-2.20.1/ld'
make[2]: Leaving directory `/home/test/binutils-2.20.1/ld'
make[1]: Nothing to be done for `all-target'.
make[1]: Leaving directory `/home/test/binutils-2.20.1'

real    1m27.280s
user    0m54.583s
sys     0m8.777s
root@localhost:/home/test/binutils-2.20.1# _

```

图 5-8 Guest OS2 编译 Binutils 的时间

(7) 测试数据统计分析

各种配置情况下测试数据的统计结果如表 5-1 所示。该表显示对 Host、Guest OS1 和 Guest OS2 分别进行了 10 次测试，并得到了 10 组数据。测试数据包括 real、user 和 sys 三类时间，对 real 时间进行求平均值，并求出 Host:

(Guest1+Guest2)/2、Host: Guest1、Host: Guest2 三组比值验证虚拟机的系统性能。

测试结论：通过测试图例及数据分析说明，VMM 中可以正常地运行两个独立的 Guest OS，并实现了为每个 Guest OS 分配 3.5G 物理内存。

表 5-1 测试数据统计分析表

system	CPU 份额	测试次数序号							平均时间	Host/Guest 百分比
		1	2	3	4	5	6	7		
Host	real	78.499	83.519	74.718	74.348	71.769	71.893	72.698	72.12	82.64%
	user	48.299	48.343	46.767	47.067	46.875	46.539	46.943		
	sys	5.676	5.872	6.136	5.496	4.444	4.812	5.132		
Guest1	real	87.156	90.212	89.179	87.822	87.359	87.369	87.317	87.35	82.564%
	user	54.135	56.716	56.400	54.835	54.567	54.847	54.895		
	sys	8.973	9.949	8.617	8.441	8.413	8.529	8.385		
Guest2	real	88.67	88.852	89.262	87.692	87.000	87.269	87.301	87.19	82.716%
	user	55.295	55.739	55.315	54.611	54.691	54.619	54.467		
	sys	8.977	8.789	9.309	8.721	8.389	8.261	8.629		

5.2 虚拟机绑定测试

5.2.1 测试目标

验证 VMM 实现了能够正确绑定客户机到指定处理器核心上,绑定在处理器核上的进程不因 CPU 负载平衡而发生迁移。

5.2.2 测试方法和步骤

5.2.2.1 测试方法描述

1. 测试进程绑定的前置条件

在启动 Guest OS 时,指定创建 Guest OS 进程的 CPU 参数,实现对 Guest OS 的绑定,使 Guest OS 的进程在指定的单核上运行。

2. 测试方法

在其他环境相同的条件下,分别比较未指定 cpuid 和指定 cpuid 时,启动客户机,观察执行客户机的物理 CPU 负载。通过 CPU 利用率,测试 Guest OS 绑定效果。测试程序:计算费波拉数列 n 项和。

5.2.2.2 测试步骤

测试步骤如下：

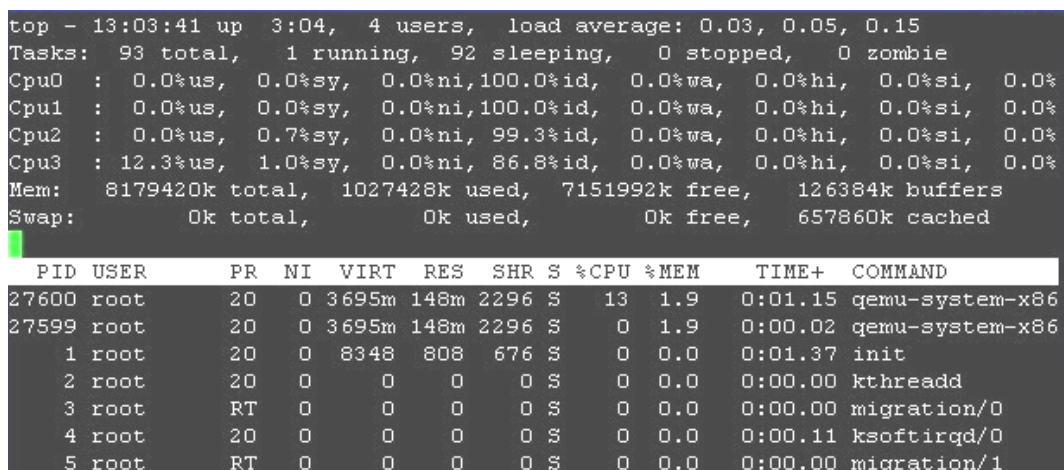
- (1) 未设置 cpuid, 第一次启动 Guest OS 时, 观察 Guest OS 在哪个核上运行;
- (2) 设置 cpuid=0, 启动 Guest OS 后, 观察 Guest OS 在哪个核上运行;
- (3) 设置 cpuid=3, 启动 Guest OS 后, 观察 Guest OS 在哪个核上运行;
- (4) 未绑定进程情况下, 在 Guest OS 中运行测试程序, 观察不同时刻 Guest OS 在哪个核上运行及其 CPU 的利用率;
- (5) 未绑定进程情况下, 在 Guest OS 中运行测试程序, Host OS 运行两个测试程序, 观察不同时刻 Guest OS 在哪个核上运行及其 CPU 的利用率;
- (6) 绑定进程情况下, 在 Guest OS 中运行测试程序, 观察不同时刻 Guest OS 在哪个核上运行及其 CPU 的利用率;
- (7) 绑定进程情况下, 在 Guest OS 中运行测试程序, Host OS 运行两个测试程序, 观察不同时刻 Guest OS 在哪个核上运行及其 CPU 的利用率。

5.2.3 效果分析与结论

5.2.3.1 Guest OS 在指定核上运行的测试

- (1) 未指定 CPU 参数情况下, 第一次启动 Guest OS 时, Guest OS 启动后自动分配在 CPU 核 3 上运行, 如图 5-9 所示。

运行命令: QEMU-system-x86_64 guest0.img -m 3584



```
top - 13:03:41 up 3:04, 4 users, load average: 0.03, 0.05, 0.15
Tasks: 93 total, 1 running, 92 sleeping, 0 stopped, 0 zombie
Cpu0  : 0.0%us, 0.0%sy, 0.0%ni,100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Cpu1  : 0.0%us, 0.0%sy, 0.0%ni,100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Cpu2  : 0.0%us, 0.7%sy, 0.0%ni, 99.3%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Cpu3  : 12.3%us, 1.0%sy, 0.0%ni, 86.8%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Mem:   8179420k total, 1027428k used, 7151992k free, 126384k buffers
Swap:   0k total, 0k used, 0k free, 657860k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
27600	root	20	0	3695m	148m	2296	S	13	1.9	0:01.15	qemu-system-x86
27599	root	20	0	3695m	148m	2296	S	0	1.9	0:00.02	qemu-system-x86
1	root	20	0	8348	808	676	S	0	0.0	0:01.37	init
2	root	20	0	0	0	0	S	0	0.0	0:00.00	kthreadd
3	root	RT	0	0	0	0	S	0	0.0	0:00.00	migration/0
4	root	20	0	0	0	0	S	0	0.0	0:00.11	ksoftirqd/0
5	root	RT	0	0	0	0	S	0	0.0	0:00.00	migration/1

图 5-9 Guest OS 启动后在核 3 上运行

- (2) 启动 Guest OS 在指定的核 0 上运行, 如图 5-10 所示。

运行命令: QEMU-system-x86_64 guest0.img -cpuid 0


```

top - 13:08:45 up 3:09, 4 users, load average: 0.00, 0.02, 0.10
Tasks: 93 total, 2 running, 91 sleeping, 0 stopped, 0 zombie
Cpu0 : 10.2%us, 13.8%sy, 0.0%ni, 76.1%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Cpu1 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Cpu2 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Cpu3 : 0.7%us, 0.0%sy, 0.0%ni, 99.3%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Mem: 8179420k total, 987500k used, 7191920k free, 126384k buffers
Swap: 0k total, 0k used, 0k free, 657860k cached

```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
27608	root	20	0	3695m	111m	2300	R	24	1.4	0:00.72	qemu-system-x86
27607	root	20	0	3695m	111m	2300	S	1	1.4	0:00.02	qemu-system-x86
1	root	20	0	8348	808	676	S	0	0.0	0:01.37	init
2	root	20	0	0	0	0	S	0	0.0	0:00.00	kthreadd
3	root	RT	0	0	0	0	S	0	0.0	0:00.00	migration/0
4	root	20	0	0	0	0	S	0	0.0	0:00.11	ksoftirqd/0
5	root	RT	0	0	0	0	S	0	0.0	0:00.00	migration/1

图 5-10 Guest OS 启动后在指定的核 0 上运行

(3) 启动 Guest OS 在指定的核 3 上运行，如图 5-11 所示。

运行命令：QEMU-system-x86_64 guest0.img -cpuid 3

```

top - 14:27:22 up 4:27, 5 users, load average: 0.14, 0.03, 0.15
Tasks: 95 total, 1 running, 94 sleeping, 0 stopped, 0 zombie
Cpu0 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Cpu1 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Cpu2 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Cpu3 : 22.2%us, 4.3%sy, 0.0%ni, 73.5%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Mem: 8179420k total, 1036276k used, 7143144k free, 126520k buffers
Swap: 0k total, 0k used, 0k free, 658428k cached

```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
27726	root	20	0	3695m	152m	2300	S	26	1.9	0:01.23	qemu-system-x86
11	root	20	0	0	0	0	S	0	0.0	0:11.95	events/0
27597	root	20	0	19056	1320	1000	R	0	0.0	0:03.38	top
27725	root	20	0	3695m	152m	2300	S	0	1.9	0:00.03	qemu-system-x86
1	root	20	0	8348	808	676	S	0	0.0	0:01.38	init
2	root	20	0	0	0	0	S	0	0.0	0:00.00	kthreadd
3	root	RT	0	0	0	0	S	0	0.0	0:00.00	migration/0

图 5-11 Guest OS 启动后在指定的核 3 上运行

效果分析：通过上述的测试结果，可清晰看到，在启动 Guest OS 时，通过设置 CPU 参数，可使 Guest OS 的进程在指定的核上运行。

5.2.3.2 Guest OS 进程绑定效果测试

测试目标：Guest OS 上运行 CPU 占用率比较高的测试程序； Host OS 上运行两个测试程序，Host OS 进程和 Guest OS 进程竞争 CPU，从而引起进程的调度。在是否绑定进程的两种情况下，通过测试分析进程的迁移情况。

1. 未绑定进程时，在 Guest OS 中运行测试程序。

运行命令：qemu-system-x86_64 guest0.img -m 3584

(1) t1 时刻，Guest OS 运行在核 1，如图 5-12 所示（top 命令截图）。

在截图数据中观察四个 CPU 核的使用率，在 t1 时刻可以发现 CPU 核 1 的利用率达到了 87.4%，其他核的利用率为 0，说明 Guest OS 当前是运行在 CPU 核 1 上。

```
top - 13:35:47 up 3:36, 5 users, load average: 0.36, 0.38, 0.34
Tasks: 94 total, 2 running, 92 sleeping, 0 stopped, 0 zombie
Cpu0 : 0.0%us, 0.0%sy, 0.0%ni,100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Cpu1 : 87.4%us, 0.0%sy, 0.0%ni, 12.6%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Cpu2 : 0.0%us, 0.0%sy, 0.0%ni,100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Cpu3 : 0.0%us, 0.3%sy, 0.0%ni, 99.7%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Mem: 8179420k total, 1065964k used, 7113456k free, 126404k buffers
Swap: 0k total, 0k used, 0k free, 657892k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
27673	root	20	0	3697m	179m	2376	R	87	2.2	0:46.27	qemu-system-x86
1	root	20	0	8348	808	676	S	0	0.0	0:01.37	init
2	root	20	0	0	0	0	S	0	0.0	0:00.00	kthreadd
3	root	RT	0	0	0	0	S	0	0.0	0:00.00	migration/0
4	root	20	0	0	0	0	S	0	0.0	0:00.12	ksoftirqd/0
5	root	RT	0	0	0	0	S	0	0.0	0:00.00	migration/1
6	root	20	0	0	0	0	S	0	0.0	0:00.14	ksoftirqd/1

图 5-12 Guest OS 运行在核 1 上 (t1 时刻)

(2) t2 时刻, Guest OS 仍然运行在核 1 上, 未发生迁移, 如图 5-13 所示。

```
top - 13:36:41 up 3:37, 5 users, load average: 0.52, 0.42, 0.36
Tasks: 94 total, 1 running, 93 sleeping, 0 stopped, 0 zombie
Cpu0 : 0.0%us, 0.0%sy, 0.0%ni,100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Cpu1 : 87.7%us, 0.0%sy, 0.0%ni, 12.3%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Cpu2 : 0.0%us, 0.0%sy, 0.0%ni,100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Cpu3 : 0.0%us, 0.0%sy, 0.0%ni,100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Mem: 8179420k total, 1065964k used, 7113456k free, 126404k buffers
Swap: 0k total, 0k used, 0k free, 657892k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
27673	root	20	0	3697m	179m	2376	S	88	2.2	1:33.51	qemu-system-x86
1	root	20	0	8348	808	676	S	0	0.0	0:01.37	init
2	root	20	0	0	0	0	S	0	0.0	0:00.00	kthreadd
3	root	RT	0	0	0	0	S	0	0.0	0:00.00	migration/0
4	root	20	0	0	0	0	S	0	0.0	0:00.12	ksoftirqd/0
5	root	RT	0	0	0	0	S	0	0.0	0:00.00	migration/1
6	root	20	0	0	0	0	S	0	0.0	0:00.14	ksoftirqd/1

图 5-13 Guest OS 运行在核 1 上 (t2 时刻)

(3) t3 时刻, Host OS 上运行两个测试程序, 此时 Guest OS 迁移到核 0 上运行, 如图 5-14 所示。

```
top - 13:37:17 up 3:37, 5 users, load average: 0.59, 0.45, 0.37
Tasks: 96 total, 2 running, 94 sleeping, 0 stopped, 0 zombie
Cpu0 : 87.7%us, 0.0%sy, 0.0%ni, 12.3%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Cpu1 : 48.0%us, 0.0%sy, 0.0%ni, 52.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Cpu2 : 0.0%us, 0.0%sy, 0.0%ni,100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Cpu3 : 48.8%us, 0.0%sy, 0.0%ni, 51.2%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Mem: 8179420k total, 1066228k used, 7113192k free, 126404k buffers
Swap: 0k total, 0k used, 0k free, 657892k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
27673	root	20	0	3697m	179m	2376	R	88	2.2	2:04.97	qemu-system-x86
27678	root	20	0	3708	380	300	S	49	0.0	0:03.24	fi
27679	root	20	0	3708	380	300	S	49	0.0	0:02.10	fi
1	root	20	0	8348	808	676	S	0	0.0	0:01.37	init
2	root	20	0	0	0	0	S	0	0.0	0:00.00	kthreadd
3	root	RT	0	0	0	0	S	0	0.0	0:00.00	migration/0
4	root	20	0	0	0	0	S	0	0.0	0:00.12	ksoftirqd/0

图 5-14 Guest OS 迁移到在核 0 上 (t3 时刻)

(4) t4 时刻, Guest OS 再次发生迁移, 分配到核 2 上运行, 如图 5-15 所示。

```
top - 13:37:59 up 3:38, 5 users, load average: 0.79, 0.52, 0.39
Tasks: 96 total, 3 running, 93 sleeping, 0 stopped, 0 zombie
Cpu0 : 0.0%us, 0.0%sy, 0.0%ni,100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Cpu1 : 38.7%us, 0.0%sy, 0.0%ni, 61.3%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Cpu2 : 87.4%us, 0.0%sy, 0.0%ni, 12.6%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Cpu3 : 38.1%us, 0.3%sy, 0.0%ni, 61.6%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Mem: 8179420k total, 1066352k used, 7113068k free, 126404k buffers
Swap: 0k total, 0k used, 0k free, 657892k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
27673	root	20	0	3697m	179m	2376	R	88	2.2	2:41.75	qemu-system-x86
27679	root	20	0	3708	380	300	S	39	0.0	0:19.50	fi
27678	root	20	0	3708	380	300	R	39	0.0	0:20.51	fi
27597	root	20	0	19056	1320	1000	R	0	0.0	0:01.36	top
1	root	20	0	8348	808	676	S	0	0.0	0:01.37	init
2	root	20	0	0	0	0	S	0	0.0	0:00.00	kthreadd
3	root	RT	0	0	0	0	S	0	0.0	0:00.00	migration/0

图 5-15 Guest OS 再次发生迁移（t4 时刻）

测试结果分析：

通过上述测试的 CPU 利用率可看出，t3 时刻、t4 时刻 Guest OS 在两个核之间发生了迁移，并且这个迁移是动态的过程。这是由于在（3）和（4）的情况下，Host OS 的两个测试程序和 Guest OS 竞争 CPU，引起了进程的调度。

2. 将 Guest OS 绑定到指定的核 0，在 Guest OS 中运行相同的测试程序。

运行命令：qemu-sysytem-x86_64 guest0.img -m 3584 -cpuid 0

（1）t1 时刻，Guest OS1 运行在核 0，如图 5-16 所示。

```
top - 13:40:03 up 3:40, 5 users, load average: 0.37, 0.43, 0.37
Tasks: 94 total, 2 running, 92 sleeping, 0 stopped, 0 zombie
Cpu0 : 87.3%us, 0.3%sy, 0.0%ni, 12.3%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Cpu1 : 0.0%us, 0.0%sy, 0.0%ni,100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Cpu2 : 0.0%us, 0.0%sy, 0.0%ni,100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Cpu3 : 0.0%us, 0.0%sy, 0.0%ni,100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Mem: 8179420k total, 1068072k used, 7111348k free, 126404k buffers
Swap: 0k total, 0k used, 0k free, 657892k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
27682	root	20	0	3697m	180m	2376	R	87	2.3	0:21.80	qemu-system-x86
27597	root	20	0	19056	1320	1000	R	0	0.0	0:01.44	top
1	root	20	0	8348	808	676	S	0	0.0	0:01.37	init
2	root	20	0	0	0	0	S	0	0.0	0:00.00	kthreadd
3	root	RT	0	0	0	0	S	0	0.0	0:00.00	migration/0
4	root	20	0	0	0	0	S	0	0.0	0:00.12	ksoftirqd/0
5	root	RT	0	0	0	0	S	0	0.0	0:00.00	migration/1

图 5-16 Guest OS 运行在核 0 上（t1 时刻）

（2）t2 时刻，Guest OS 未发生迁移，仍然运行在核 0 上，如图 5-17 所示。

```
top - 13:41:06 up 3:41, 5 users, load average: 0.44, 0.44, 0.37
Tasks: 94 total, 2 running, 92 sleeping, 0 stopped, 0 zombie
Cpu0 : 87.3%us, 0.0%sy, 0.0%ni, 12.7%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Cpu1 : 0.0%us, 0.0%sy, 0.0%ni,100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Cpu2 : 0.0%us, 0.0%sy, 0.0%ni,100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Cpu3 : 0.0%us, 0.0%sy, 0.0%ni,100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Mem: 8179420k total, 1068072k used, 7111348k free, 126404k buffers
Swap: 0k total, 0k used, 0k free, 657892k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
27682	root	20	0	3697m	180m	2376	R	87	2.3	1:16.77	qemu-system-x86
27681	root	20	0	3697m	180m	2376	S	0	2.3	0:00.18	qemu-system-x86
1	root	20	0	8348	808	676	S	0	0.0	0:01.37	init
2	root	20	0	0	0	0	S	0	0.0	0:00.00	kthreadd
3	root	RT	0	0	0	0	S	0	0.0	0:00.00	migration/0
4	root	20	0	0	0	0	S	0	0.0	0:00.12	ksoftirqd/0
5	root	RT	0	0	0	0	S	0	0.0	0:00.00	migration/1

图 5-17 Guest OS1 仍然运行在核 0 上（t2 时刻）

(3) t3 时刻，主机运行两个测试程序，Guest OS 未发生迁移，如图 5-18 所示。

```
top - 13:46:03 up 3:46, 5 users, load average: 0.86, 0.67, 0.48
Tasks: 96 total, 3 running, 93 sleeping, 0 stopped, 0 zombie
Cpu0 : 89.3%us, 0.0%sy, 0.0%ni, 10.7%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Cpu1 : 38.6%us, 0.0%sy, 0.0%ni, 61.4%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Cpu2 : 47.4%us, 0.0%sy, 0.0%ni, 52.6%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Cpu3 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Mem: 8179420k total, 1068584k used, 7110836k free, 126404k buffers
Swap: 0k total, 0k used, 0k free, 657892k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
27682	root	20	0	3697m	180m	2376	R	89	2.3	5:38.36	qemu-system-x86
27687	root	20	0	3708	376	300	R	48	0.0	0:25.65	fi
27688	root	20	0	3708	376	300	S	39	0.0	0:20.56	fi
27597	root	20	0	19056	1320	1000	R	0	0.0	0:01.68	top
1	root	20	0	8348	808	676	S	0	0.0	0:01.37	init
2	root	20	0	0	0	0	S	0	0.0	0:00.00	kthreadd
3	root	RT	0	0	0	0	S	0	0.0	0:00.00	migration/0

图 5-18 Guest OS1 未发生迁移 (t3 时刻)

(4) t4 时刻，Guest OS 仍然未发生迁移，运行在核 0 上，如图 5-19 所示。

```
top - 13:48:57 up 3:49, 5 users, load average: 0.57, 0.68, 0.52
Tasks: 96 total, 2 running, 94 sleeping, 0 stopped, 0 zombie
Cpu0 : 89.7%us, 0.0%sy, 0.0%ni, 10.3%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Cpu1 : 38.9%us, 0.0%sy, 0.0%ni, 61.1%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Cpu2 : 46.9%us, 0.0%sy, 0.0%ni, 53.1%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Cpu3 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Mem: 8179420k total, 1068708k used, 7110712k free, 126404k buffers
Swap: 0k total, 0k used, 0k free, 657892k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
27682	root	20	0	3697m	180m	2376	R	89	2.3	8:13.54	qemu-system-x86
27687	root	20	0	3708	376	300	S	48	0.0	1:48.90	fi
27688	root	20	0	3708	376	300	S	39	0.0	1:32.39	fi
27681	root	20	0	3697m	180m	2376	S	0	2.3	0:00.51	qemu-system-x86
1	root	20	0	8348	808	676	S	0	0.0	0:01.37	init
2	root	20	0	0	0	0	S	0	0.0	0:00.00	kthreadd
3	root	RT	0	0	0	0	S	0	0.0	0:00.00	migration/0

图 5-19 Guest OS1 仍然未发生迁移 (t4 时刻)

测试结果分析：经过一段时间的运行测试可看出，绑定 Guest OS 进程后，Guest OS 只能在指定的核上运行，并且 Host OS 的运行不会使其发生产生进程迁移。

5.3 虚拟机内存申请测试

5.3.1 测试项目目标

Host OS 为 Guest OS 分配了 3.5GB 的物理内存，Guest OS 在实际运行过程中应表现得如同在一台单核 CPU 物理机上一样，因此需要测试 Guest OS 对 3.5GB 大小内存的申请和使用能力，以及普通用户进程能够使用的最大内存。

5.3.2 测试方法描述

1、测试前置条件

(1) 已在 Host OS 上启动两个虚拟机 VM；

(2) 已将两个虚拟机 VM 分别绑定在 CPU 的两个核上；

(3) 客户机已分配物理内存：3.5GB。

2、测试方法

设置不同大小的字符型数据块，一次测试调用 malloc()函数连续申请 n 次某一相同大小的数据块，分析 Guest OS 能够申请的最大内存容量。用相同的方法进行多次测试，最后对测试结果进行统计分析。

5.3.3 效果分析与结论

运行测试程序进行多次不同数据块大小和不同申请次数的测试，得到以下测试结果：

(1) **测试一**：设定数据块为 300M，连续申请 12 次，未填充数据，部分测试数据如图 5-20 和图 5-21 所示。

```
root@GuestOS2:~/T# ./test
0    use 300M memory    at 7fec71fa8010
1    use 600M memory    at 7fec5f3a7010
2    use 900M memory    at 7fec4c7a6010
3    use 1200M memory   at 7fec39ba5010
4    use 1500M memory   at 7fec26fa4010
5    use 1800M memory   at 7fec143a3010
6    use 2100M memory   at 7fec017a2010
7    use 2400M memory   at 7febeeba1010
8    use 2700M memory   at 7febdbfa0010
9    use 3000M memory   at 7febc939f010
10   use 3300M memory   at 7febb679e010
11   use 3600M memory   at 7feba3b9d010
```

图 5-20 申请 300*12M 内存

```
Free 300M memory
Free 600M memory
Free 900M memory
Free 1200M memory
Free 1500M memory
Free 1800M memory
Free 2100M memory
Free 2400M memory
Free 2700M memory
Free 3000M memory
Free 3300M memory
Free 3600M memory
root@GuestOS2:~/T#
```

图 5-21 释放已申请的内存

(2) **测试二：**设定数据块为 100M，连续申请 30 次，并填充数据，部分测试数据如图 5-22 所示。

```
root@GuestOS2:~/T# ./test
0    use  100M memory    at  7ff5e3a20010
1    use  200M memory    at  7ff5dd61f010
2    use  300M memory    at  7ff5d721e010
3    use  400M memory    at  7ff5d0e1d010
4    use  500M memory    at  7ff5caa1c010
5    use  600M memory    at  7ff5c461b010
6    use  700M memory    at  7ff5be21a010
7    use  800M memory    at  7ff5b7e19010
8    use  900M memory    at  7ff5b1a18010
9    use 1000M memory    at  7ff5ab617010
10   use 1100M memory    at  7ff5a5216010
11   use 1200M memory    at  7ff59ee15010
12   use 1300M memory    at  7ff598a14010
```

(a)

```
20   use 2100M memory    at  7ff566a0c010
21   use 2200M memory    at  7ff56060b010
22   use 2300M memory    at  7ff55a20a010
23   use 2400M memory    at  7ff553e09010
24   use 2500M memory    at  7ff54da08010
25   use 2600M memory    at  7ff547607010
26   use 2700M memory    at  7ff541206010
27   use 2800M memory    at  7ff53ae05010
28   use 2900M memory    at  7ff534a04010
29   use 3000M memory    at  7ff52e603010
Free 100M memory
Free 200M memory
```

(b)

图 5-22 申请 100*30M 内存

(3) **测试结论：**经过测试，实验一测试结果说明在客户机内只申请内存不写入数据（即没有实际分配物理内存）可以达到 3.5G 的需求。实验二测试结果说明通过设定不同大小的字符型数据块，连续申请不同次数，在客户机中一个用户程序每次最大可申请 3GB 的物理内存（由于 Linux 进程的限定，每个进程可占用 4G 内存，其中系统地址空间为 1G，用户地址空间只能占用 3G）。

结 论

本文在研究目前流行的计算机虚拟化技术和基于 Linux 的 KVM 虚拟化方案的基础上，以实际项目为依托，从传统的全虚拟化技术入手，深入研究了计算机内存虚拟化技术和处理器虚拟化技术。总结了对宿主机和客户机之间地址转换的实现流程，设计与实现了一种基于 KVM 的 EPT 页表优化方案，达到为客户机分配固定内存的目标。本文深入研究硬件支持虚拟化技术，结合 Linux 与 KVM 虚拟化方案的实现，对 VCPU、VMCS 描述符和 VCPU 的创建、初始化和加载执行进行分析研究，并结合 Linux 进程绑定技术，实现了虚拟机在指定物理 CPU 核上运行的目标。本文在支持硬件虚拟化的计算机平台上进行 VMM 的研发与测试，经过长时间运行以及 Benchmark、自写测试程序的测试，系统稳定，获得了较好的系统性能。

1. 研究成果

本文的研究目标是根据项目需求，在对计算机虚拟化技术基本原理研究分析的基础上，详细分析内存虚拟化、处理器虚拟化，研究并提出 EPT 页表优化方案、虚拟机与处理器绑定方案。为了达到此目标，对计算机硬件支持虚拟化、内存管理等方面进行了深入研究，主要取得了以下成果：

- (1) 在研究当前全模拟内存虚拟化的基础上，总结了基于 KVM 的硬件支内存虚拟化的实现技术；
- (2) 提出了基于 EPT 的宿主机和客户机地址转换的流程优化方案；
- (3) 在软件实现处理器虚拟化技术基础上，总结了 KVM 中 VMCS 的具体应用以及 VCPU 的创建运行的实现技术；
- (4) 研究 Intel VT-x 技术中的 VMCS 及 Linux 进程绑定技术，实现了一个客户机与指定物理处理器核的绑定，使得 Guest OS 可在指定的核上独立运行，并在运行期间不发生迁移，该项技术支持仅能运行在单核上的遗留单核操作系统，充分利用多核处理器的资源。

2. 建议

通过分析和总结已完成的研究和工作成果，针对本文提出的优化方法、虚拟机绑定方案，提出以下几点建议：

- (1) 本文所提出的 EPT 页表机制优化方法在项目测试中取得了一定效果，并得到应用。下一步应研究扩展该优化方案应用范围的方法。
- (2) 虚拟机绑定方案可以实现虚拟机与指定物理处理器核的绑定，应在此

基础上研究不需参数自动绑定的方法；

虚拟化技术及其应用是计算机科学技术领域的一项重要研究课题。现代社会的发展对计算机虚拟化的需求是相当广泛的，如在网络服务器、科学计算与仿真、工业设计与金融等领域都存在着巨大的应用潜力，但这些领域所涉及的大规模数值计算和海量数据检索存在一个不可避免的计算能力瓶颈，因此进一步深入研究和发 展高性能计算技术以及虚拟化技术都可谓势在必行，并具有广阔的发展和应用前景。

致 谢

“穷究于理，成就于工”，在成都理工大学的三年时间里，经历着校园里栀子花开，嗅着散发的淡淡幽香，感受着轻轻地飘荡微风，走过熙攘的人群，穿过葱郁的慧园，拂过砚湖平静的水面，透过安静的图书馆，回味着校园里的每一个角落。三年过去了，发生的一件件事情，闪现在脑中，有欢笑，有泪水，有着大家相互间的那种情谊，在这里遇到困难了，在这里战胜困难了，在这里成熟了。难以忘记那些一路陪伴我成长的良师益友，是你们让我的生活丰富多彩，充满阳光。我想衷心的说一声谢谢，谢谢你们在我孤独时陪在我身边，谢谢你们在我彷徨时为我照亮前进的道路，谢谢你们在我失落时对我的鼓励，谢谢你们在我生命中写下绚丽的一页。

感谢我的导师罗省贤教授，感谢您对我的悉心指导与无微不至的关怀，您严谨的治学态度和对学生的奉献精神让我受益匪浅，我感受到了教师职业的光荣与神圣。您引领我进入学术的殿堂并让我感受到了研究的乐趣与魅力。感谢您一直以来对我的支持与鼓励，给我提供了一个锻炼发展成长的平台，谢谢罗老师。

感谢与我一起参加项目的涂得志与姚峰老师，感谢你们在项目中为指明方向、提出指导和建议，让我有了一个自我提升的机会，让我可以在学术的海洋里汲取营养与积累能量，感谢同学黄煜在项目上对我的合作、帮助与建议，从您身上我学到了坚持与投入，感谢您对我的支持与研究课题上的参与和帮助。感谢林龙增、车翔、龙从海等同学对我的关心与照顾，让我更快地融入到这个集体，和你们在一起很开心，在你们身上我学到了很多优秀的品质，能够跟你们一起度过在研究生最后一年的时光是我莫大的荣幸。

感谢实验室的同学们，我们像家人一样工作生活在一起，感谢你们对我的关心与照顾。感谢师兄李军、彭武杰、赵小盼、张士恒、刘刚国、洪振刚等，感谢一同陪我成长的石红霞、游左勇、葛辉、程志、刘扬、王光辉、张新菊、吴保来等，感谢师弟师妹们对我的支持与信任，感谢你们三年中对我的帮助。

感谢成都理工大学与信息工程学院的老师，感谢你们教授我知识，教我做人，培养我成长，让我快乐充实地在理工度过了大学和研究生时光。

感谢那些在我生命中驻足的同学们，感谢你们让我的生活变得丰富多彩，我们从相识走到了相知，我们一起努力学习，为了理想而共同奋斗。特别感谢研究生会文艺部的同学们，在我们合作的日日夜夜里建立了深厚的友谊，为了我们共同的兴趣爱好而努力。

感谢各位评审专家在百忙之中审阅我的文章，感谢答辩委员会各位专家在

百忙之中出席我的论文答辩会。

还要感谢我的爸爸、妈妈和弟弟，你们的辛勤工作使我可以更专注在学业上，你们的理解与鼓励让我在软弱时坚定信念，你们的宽容与肯定让我大胆地去追寻梦想。你们是我强有力的后盾，是我永远的归属，拥有你们是我此生最大的幸福。

最后，向被本文引用的文献和图件的版权所有者表示诚挚的谢意，也要感谢我阅读过的所有论文的作者，你们的研究成果和无声的交流为本文的研究提供了有益的基础和思路。

参考文献

- [1] 虚拟化技术基础 http://blog.chinaunix.net/u3/110913/showart_2185421.html
- [2] 探索 Linux 内核虚拟机 <http://www.ibm.com/developerworks/cn/Linux/l-Linux-kvm/>
- [3] 英特尔开源软件技术中心 复旦大学并行处理研究所著 系统虚拟化 清华大学出版社.2009
- [4] 徐磊 Linux 系统下 C 程序开发详解 电子工业出版社.2008
- [5] 纪纯杰 贺晓能 Linux 内核分析及常见问题解答 著人民邮电出版社 2000
- [6] Naba Barkakati Red Hat LINUX 核心技术精解 北京：中国水利水电出版社.1999
- [7] 李善平 李文峰等 著 Linux 内核 2.4 版源代码分析大全 机械工业出版社.2002
- [8] BOVET&CESATI 著 深入理解 Linux 内核中国电力出版社.2010
- [9] KVM 开源项目 官方网址 http://www.Linux-kvm.org/page/Main_Page
- [10] QEMU 开源项目 官方网址 http://wiki.QEMU.org/Main_Page
- [11] The Intel® 64 and IA-32 Architectures Software Developer's Manual.2009
- [12] Claudia Salzberg Rodriguez 等 Linux 内核编程 机械工业出版社.2006
- [13] 吴国伟 李张等 Linux 内核分析及高级编程 电子工业出版社.2008
- [14] 陈莉君 深入分析 Linux 内核源码 <http://www.kerneltravel.net/>
- [15] 鲁松主编 计算机虚拟化技术及应用 机械工业出版社.2008
- [16] Mel Gorman. 深入理解 Linux 虚拟内存管理[M]. 北京：航空航天大学出版社.2006
- [17] 纪纯杰 贺晓能 Linux 内核分析及常见问题解答.2000
- [18] 虚拟机中 Guest OS 时钟 (TIMEKEEP) 问题探讨 <http://www.ibm.com/developerworks/cn/Linux/l-Linux-kvm/>
- [19] http://en.wikipedia.org/wiki/Comparison_of_virtual_machines
- [20] Robin JS,Irvine CE.Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor,Proc.9th USENIX Security Symposium,2000
- [21] 李庆华, 罗犀劲. 基于检查点进程迁移机制的改进[J].计算机仿真, 2003,20(5): 50-52
- [22] 周全, 卢显良, 任立勇, 等. 基于 Linux 的进程迁移机制设计[J]. 计算机应用, 2003,23(8): 58-60
- [23] 吴松, 金海. 存储虚拟化研究[J]. 小型微型计算机系统, 2003,24(4): 728-732
- [24] 虚拟化 IT 效率最大化[J]. 信息系统工程, 2007(3)
- [25] 王迪 . 分析：虚拟化技术对企业应用带来的优势 [EB/OL]. <http://tech.sina.com.cn/smb/2008-11-26/0615888394.shtml>,2008-11-26
- [26] 龚哎斐, 张文静. 基于虚拟化架构的软件开发与测试环境自动化[J]. 自动化与信息工程,2008(2)

- [27] 张萧, 祝明发, 肖利民. 分布式 I/O 资源虚拟化技术的研究[J]. 微电子与计算机, 2008,25(100)
- [28] Martin F. Maldonado. 虚拟化概述: 模式的观点 [EB/OL].
<http://www.ibm.com/developworks/cn/grid/grvirt/>
- [29] IBM.. Using the Virtual I/O Server, 2007,9
- [30] User-mode Linux. <http://user-mode-linux.sourceforge.net/>
- [31] Aloni D. Cooperative linux. Proceedings of the Linux Symposium. 1: 23-32,2004
- [32] Linux_VServer. <http://linux-vserver.org/>
- [33] Kivity A, Kamay Y, Laor D, Lublin U, Liguori A. kvm: the Linux Virtual machine monitor, Proceedings of the 2007 Ottawa Linux Symposium,2007
- [34] Russel R. lguest: Implementing the little Linux hypervisor. Proceedings of the 2007 Ottawa Linux Symposium, 2007

攻读学位期间取得学术成果

- [1] 李绍, 罗省贤. 一种基于内存虚拟化技术的 EPT 机制优化方法[J]. 电脑与电信, 2011(02): 52-54.
- [2] 参加校企合作项目《基于 Intel 多核技术的虚拟机研究和开发》, 主要负责内存虚拟化部分, 参与处理器虚拟化研究。