

系统虚拟化

—— 原理与实现

Intel® Corporation 2008

英特尔开源软件技术中心

著

复旦大学并行处理研究所

清华大学出版社
北京

内 容 简 介

本书深入而又系统地介绍了以软件完全虚拟化、硬件辅助虚拟化及类虚拟化为核心的各种系统虚拟化技术。全书共 9 章,第 1 章概述性地介绍了虚拟化技术;第 2 章介绍计算机系统知识;第 3 章从 CPU 虚拟化、内存虚拟化和 I/O 虚拟化三大块对系统虚拟化技术进行概述,并介绍虚拟机监控器(VMM)的组成与分类,而且对市场上流行的虚拟化产品进行了简单介绍;第 4~6 章分别从基于软件的完全虚拟化、硬件辅助的完全虚拟化和类虚拟化三种实现技术角度深入介绍系统虚拟化方法;第 7 章介绍虚拟机的性能评测和调试技术;第 8 章介绍系统虚拟化的应用实例;最后在第 9 章对虚拟机和系统虚拟化技术的发展作一个展望。

本书是系统虚拟化技术实现原理的全面展示,也是作者这些年在虚拟化学术和工业研究领域开发的经验总结。本书理论与实践相结合,用通俗易懂的语言描述系统虚拟化技术原理,其中不乏具有代表性和普遍意义的实例和技术细节,是学习系统虚拟化技术的宝贵资料。本书不仅可以作为教材,供计算机相关专业的大学高年级学生和研究生阅读;而且可以作为一本参考手册,供大学或企业里与系统相关领域的研究开发人员以及对虚拟机及虚拟化核心技术有兴趣的研究者和开源工作者阅读。

本书封面贴有清华大学出版社防伪标签,无标签者不得销售。

版权所有,侵权必究。侵权举报电话: 010-62782989 13701121933

图书在版编目(CIP)数据

系统虚拟化: 原理与实现/英特尔开源软件技术中心,复旦大学并行处理研究所著. —北京: 清华大学出版社, 2009. 3

ISBN 978-7-302-19372-2

I. 系… II. ①英… ②复… III. 虚拟技术 IV. TP391. 9

中国版本图书馆 CIP 数据核字(2009)第 011521 号

责任编辑: 索 梅 徐跃进

责任校对: 梁 毅

封面设计: 崔爽纯

责任印制: 杨 艳

出版发行: 清华大学出版社

<http://www.tup.com.cn>

社 总 机: 010-62770175

投稿与读者服务: 010-62776969, c-service@tup.tsinghua.edu.cn

质 量 反 馈: 010-62772015, zhiliang@tup.tsinghua.edu.cn

地 址: 北京清华大学学研大厦 A 座

邮 编: 100084

邮 购: 010-62786544

印 刷 者: 北京市清华园胶印厂

装 订 者: 三河市溧源装订厂

经 销: 全国新华书店

开 本: 185×230 **印 张:** 16 **字 数:** 353 千字

版 次: 2009 年 3 月第 1 版 **印 次:** 2009 年 3 月第 1 次印刷

印 数: 1~3000

定 价: 29.00 元

本书如存在文字不清、漏印、缺页、倒页、脱页等印装质量问题,请与清华大学出版社出版部联系调换。联系电话: (010)62770177 转 3103 产品编号: 029201-01

P(R)E F A C E

前

言

虚拟化技术在近期成为了学术界和产业界的一大焦点，并且被认为是在将来的一段时间内最具影响力的技术之一，它可能会改变现有系统软件的整个样子，为系统软件带来一场新的革命。

虚拟化技术正在成为系统软件中广泛存在的一层，它的普及可以从三个角度来看待。从硬件平台来讲，虚拟化技术被用于企业级服务器、桌面平台（例如台式计算机和笔记本式计算机）以及嵌入式系统中；从用途来讲，虚拟化技术被用于系统资源管理、容错、软硬件维护、增强系统安全、提升性能和节能等领域；从趋势来讲，虚拟化技术正在广泛地与其他技术结合，并且得到更多硬件上的支持，其性能损失不断降低，部分固化到硬件中。

虚拟化技术的含义很广泛。将任何一种形式的资源抽象成另一种形式的技术都是虚拟化。在常用的操作系统中就存在某种意义上的“虚拟化技术”，例如虚拟内存空间和进程。如果把内存看作是一个设备，虚拟内存就是将物理内存虚拟成多个内存空间。虚拟内存的容量可以少于或多于物理内存。进程的概念实际是对于物理硬件执行环境的一个抽象，每一个进程都享有一个完整的硬件执行环境，并且与其他进程相隔离。

相对于进程级的虚拟化，虚拟机是另外一个层面的虚拟化，即系统级虚拟化。与虚拟单个进程的执行环境所不同，系统级虚拟化所抽象的环境是整个计算机，其抽象出的环境称为虚拟机，包括 CPU、内存和 I/O。在每个虚拟机中都可以运行一个操作系统，在一台计算机上可以虚拟出多个虚拟机。

本书尝试将当前主要的虚拟机和系统级虚拟化原理梳理出来，从一个系统设计者的角度来介绍。从基本的原理出发，本书结合主流的 x86 体系结构和硬件上对虚拟化的支持来介绍系统级虚拟化是如何实现的。除介绍虚拟机与系统级虚拟化原理之外，本书力图加入学术界对于虚拟化技术或利用虚拟化技术的最新研究、产业界的最新应用和将来可能的发展趋向。

1. 面向的读者

系统虚拟化是一门跨领域的学科，涉及操作系统、编译和体系结构等学科知识，并延展到资源管理、性能和系统安全等问题。

本书定位的读者包括计算机相关专业的高年级学生、研究生、研究开发人员及对虚拟机

及虚拟化核心技术有兴趣的学者。

2. 全书结构

本书的结构安排尽可能使每章的内容自包含, 尽力让对于某一章节感兴趣的读者不需检索其他章节的内容。

第1章从虚拟化技术的历史开始讲起, 将现有的虚拟化技术作一个分类。

第2章介绍了一个缩略版的计算机系统, 帮助读者温习这些知识。其内容主要包括硬件抽象层、操作系统的硬件管理机制以及进程等与后续章节有关的操作系统概念。对于这些内容已经了解的读者可以直接跳过这一章。

第3章介绍典型系统级虚拟化的技术以及VMM的组成和分类, 最后还介绍一些目前市场比较流行的虚拟化产品。

第4章介绍基于软件的完全虚拟化技术。

第5章介绍硬件辅助的完全虚拟化技术。

第6章以Xen为例介绍类虚拟化技术的实现原理。

第7章介绍虚拟机的性能评测和调试技术。

第8章介绍系统虚拟化的应用实例。

第9章对虚拟机和系统虚拟化技术的发展作一个展望。

本书的第1章由复旦大学张逢喆、英特尔公司董耀祖、李少凡合作撰写; 第2章由复旦大学俞捷和英特尔公司张鑫合作撰写; 第3章由英特尔公司田坤和余珂撰写; 第4章由复旦大学张逢喆和黄弋简撰写; 第5章由英特尔公司余珂、李欣、蒋运宏和徐雪飞撰写; 第6章由复旦大学张逢喆、刘鹏程和黄弋简撰写; 第7章由英特尔公司董耀祖和杨晓伟撰写; 第8章由英特尔公司余珂、王庆和复旦大学吴曦、袁立威合作完成; 第9章由复旦大学刘鹏程、周亦勋、宋翔和英特尔公司董耀祖合作完成。英特尔公司李少凡和董耀祖对本书的每一版草稿均作了细致的审阅工作, 余珂、张鑫、王庆以及复旦大学的张逢喆对全书的统编和修改作了大量的工作。

3. 如何阅读本书

对于虚拟机和系统虚拟化基本原理可以阅读第1、3、4、5、6章。

希望单独了解基于软件的完全虚拟化、硬件辅助的完全虚拟化或类虚拟化的读者可以单独阅读对应的章节。

希望了解系统虚拟化性能评测和优化技术的读者可以阅读第7章。

希望了解系统虚拟化技术背景、应用和发展的读者可以阅读第1、8、9章。

4. 感谢

在这里, 首先要感谢英特尔公司副总裁王文汉博士、英特尔亚太研发有限公司总经理兼首席研发官梁兆柱博士、英特尔亚太研发有限公司创新中心总监黄波博士和英特尔开源技术中心高级经理冯晓焰先生, 以及复旦大学软件学院院长臧斌宇教授, 他们是本书的发起人, 并一直鼓励我们完成本书。

也要感谢所有在英特尔开源技术中心工作的同事以及所有在复旦大学软件学院学习工

作的同事和同学们,感谢他们不仅在工业界还在学术界推动虚拟化技术向前发展所做的努力,同时也感谢他们对本书草稿进行了一遍又一遍的阅读,并提出了许多宝贵的修改意见。在此,特别感谢英特尔公司辛晓慧、崔得暄、韩伟东、贺青和单海涛等,他们为本书提供了大量技术资料。还要特别感谢复旦大学陈海波、陈榕、杨子夜、王慧红和陈诚等,他们为本书的编撰提供了许多帮助。

最后,感谢您在茫茫书海中选择了本书,并衷心祝愿您能从中受益。

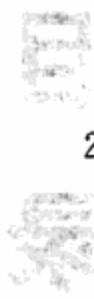
虚拟化专题写作组
2008年9月

CONTENTS

目
录

第1章 开篇	1
1.1 形形色色的虚拟化	2
1.2 系统虚拟化	3
1.3 系统虚拟化简史	5
1.4 系统虚拟化的好处	9
第2章 x86 架构及操作系统概述	12
2.1 x86 的历史和操作系统概要	12
2.1.1 x86 的历史	12
2.1.2 操作系统概述	13
2.2 x86 内存架构	13
2.2.1 地址空间	13
2.2.2 地址	14
2.2.3 x86 内存管理机制	15
2.3 x86 架构的基本运行环境	23
2.3.1 三种基本模式	23
2.3.2 基本寄存器组	23
2.3.3 权限控制	24
2.4 中断与异常	25
2.4.1 中断架构	25
2.4.2 异常架构	29
2.4.3 操作系统对中断/异常的处理流程	30
2.5 进程	30
2.5.1 上下文	31

第1章 开篇	1
1.1 形形色色的虚拟化	2
1.2 系统虚拟化	3
1.3 系统虚拟化简史	5
1.4 系统虚拟化的好处	9
第2章 x86 架构及操作系统概述	12
2.1 x86 的历史和操作系统概要	12
2.1.1 x86 的历史	12
2.1.2 操作系统概述	13
2.2 x86 内存架构	13
2.2.1 地址空间	13
2.2.2 地址	14
2.2.3 x86 内存管理机制	15
2.3 x86 架构的基本运行环境	23
2.3.1 三种基本模式	23
2.3.2 基本寄存器组	23
2.3.3 权限控制	24
2.4 中断与异常	25
2.4.1 中断架构	25
2.4.2 异常架构	29
2.4.3 操作系统对中断/异常的处理流程	30
2.5 进程	30
2.5.1 上下文	31



2.5.2 上下文切换	32
2.6 I/O 架构	33
2.6.1 x86 的 I/O 架构	33
2.6.2 DMA	33
2.6.3 PCI 设备	34
2.6.4 PCI Express	38
2.7 时钟	39
2.7.1 x86 平台的常用时钟	40
2.7.2 操作系统的时钟观	41

第 3 章 虚拟化概述 42

3.1 可虚拟化架构与不可虚拟化架构	42
3.2 处理器虚拟化	44
3.2.1 指令的模拟	44
3.2.2 中断和异常的模拟及注入	47
3.2.3 对称多处理器技术的模拟	48
3.3 内存虚拟化	51
3.4 I/O 虚拟化	55
3.4.1 概述	55
3.4.2 设备发现	56
3.4.3 访问截获	57
3.4.4 设备模拟	58
3.4.5 设备共享	60
3.5 VMM 的功能和组成	60
3.5.1 虚拟环境的管理	61
3.5.2 物理资源的管理	62
3.5.3 其他模块	63
3.6 VMM 的分类	63
3.6.1 按虚拟平台分类	63
3.6.2 按 VMM 实现结构分类	65
3.7 典型虚拟化产品及其特点	68
3.7.1 VMware	68
3.7.2 Microsoft	69
3.7.3 Xen	70
3.7.4 KVM	72

3.8 思考题	72
---------------	----

第4章 基于软件的完全虚拟化 74

4.1 概述	74
4.2 CPU 虚拟化	74
4.2.1 解释执行	75
4.2.2 扫描与修补	76
4.2.3 二进制代码翻译	79
4.3 内存虚拟化	84
4.3.1 概述	85
4.3.2 影子页表	85
4.3.3 内存虚拟化的优化	93
4.4 I/O 虚拟化	95
4.4.1 设备模型	95
4.4.2 设备模型的软件接口	97
4.4.3 接口拦截和模拟	98
4.4.4 功能实现	101
4.4.5 案例分析：IDE 的 DMA 操作	102
4.5 思考题	103

第5章 硬件辅助虚拟化 104

5.1 概述	104
5.2 CPU 虚拟化的硬件支持	105
5.2.1 概述	105
5.2.2 VMCS	107
5.2.3 VMX 操作模式	109
5.2.4 VM-Entry/VM-Exit	110
5.2.5 VM-Exit	112
5.3 CPU 虚拟化的实现	116
5.3.1 概述	116
5.3.2 VCPU 的创建	117
5.3.3 VCPU 的运行	118
5.3.4 VCPU 的退出	123
5.3.5 VCPU 的再运行	125
5.3.6 进阶	126

5.4 中断虚拟化	128
5.4.1 概述	128
5.4.2 虚拟 PIC	129
5.4.3 虚拟 I/O APIC	130
5.4.4 虚拟 Local APIC	131
5.4.5 中断采集	131
5.4.6 中断注入	132
5.4.7 案例分析	133
5.5 内存虚拟化	135
5.5.1 概述	135
5.5.2 EPT	136
5.5.3 VPID	139
5.6 I/O 虚拟化的硬件支持	140
5.6.1 概述	140
5.6.2 VT-d 技术	141
5.7 I/O 虚拟化的实现	147
5.7.1 概述	147
5.7.2 设备直接分配	147
5.7.3 设备 I/O 地址空间的访问	148
5.7.4 设备发现	149
5.7.5 配置 DMA 重映射数据结构	149
5.7.6 设备中断虚拟化	150
5.7.7 案例分析：网卡的直接分配在 Xen 里面的实现	150
5.7.8 进阶	151
5.8 时间虚拟化	151
5.8.1 操作系统的时间概念	151
5.8.2 客户机的时间概念	152
5.8.3 时钟设备仿真	153
5.8.4 实现客户机时间概念的一种方法	154
5.8.5 实现客户机时间概念的另一种方法	155
5.8.6 如何满足客户机时间不等于实际时间的需求	156
5.9 思考题	157
第 6 章 类虚拟化技术	158
6.1 概述	159



6.1.1	类虚拟化的由来	159
6.1.2	类虚拟化的系统实现	160
6.1.3	类虚拟化接口的标准化	161
6.2	类虚拟化体系结构	162
6.2.1	指令集	162
6.2.2	外部中断	163
6.2.3	物理内存空间	164
6.2.4	虚拟内存空间	165
6.2.5	内存管理	165
6.2.6	I/O 子系统	166
6.2.7	时间与时钟服务	167
6.3	Xen 的原理与实现	167
6.3.1	超调用	167
6.3.2	虚拟机与 Xen 的信息共享	168
6.3.3	内存管理	168
6.3.4	页表虚拟化	169
6.3.5	事件通道	171
6.3.6	授权表	172
6.3.7	I/O 系统	172
6.3.8	实例分析：块设备虚拟化	175
6.4	XenLinux 的运行	176
6.5	思考题	177
第 7 章 虚拟环境性能和优化		178
7.1	性能评测指标	178
7.2	性能评测工具	179
7.2.1	重用操作系统的性能评测工具	179
7.2.2	面向虚拟环境的性能评测工具	180
7.3	性能分析工具	181
7.3.1	Xenoprof	182
7.3.2	Xentrace	184
7.3.3	Xentop	185
7.4	性能优化方法	185
7.4.1	降低客户机退出事件发生频率	185
7.4.2	降低客户机退出事件处理时间	186

7.4.3 降低处理器利用率	187
7.5 性能分析案例	187
7.5.1 案例分析: Xenoprof	187
7.5.2 案例分析: Xentrace	189
7.6 可扩展性	192
7.6.1 宿主机的可扩展性	192
7.6.2 客户机的可扩展性	193
7.7 思考题	194

第8章 虚拟化技术的应用模式 195

8.1 常用技术介绍	195
8.1.1 虚拟机的动态迁移	195
8.1.2 虚拟机快照	196
8.1.3 虚拟机的克隆	197
8.1.4 案例分析: VMware VMotion 和 VMware 快照	197
8.2 服务器整合	200
8.2.1 服务器整合技术	200
8.2.2 案例分析: VMware Infrastructure 3	201
8.3 灾难恢复	202
8.3.1 灾难恢复与虚拟化技术	202
8.3.2 案例分析: VMware Infrastructure 3	203
8.4 改善系统可用性	203
8.4.1 可用性的含义	203
8.4.2 虚拟化技术如何提高可用性	204
8.4.3 虚拟化技术带来的新契机	204
8.4.4 案例分析: VMware HA 和 LUCOS	205
8.5 动态负载均衡	206
8.5.1 动态负载均衡的含义	206
8.5.2 案例分析: VMware DRS	206
8.6 增强系统可维护性	207
8.6.1 可维护性的含义	207
8.6.2 案例分析: VMware VirtualCenter	208
8.7 增强系统安全与可信任性	208
8.7.1 安全与可信任性的含义	208
8.7.2 虚拟化技术如何提高系统安全	209

8.7.3 虚拟化技术如何提高可信任性	210
8.7.4 案例分析: sHyper、VMware Infrastructure 3 和 CoVirt	210
8.8 Virtual Appliance	211
第9章 前沿虚拟化技术	213
9.1 基于容器的虚拟化技术	213
9.1.1 容器技术的基本概念和发展背景	214
9.1.2 基于容器的虚拟化技术	216
9.2 系统安全	219
9.2.1 基于虚拟化技术的恶意软件	219
9.2.2 虚拟机监控器的安全性	221
9.3 系统标准化	224
9.3.1 开放虚拟机格式	224
9.3.2 虚拟化的可管理性	225
9.3.3 虚拟机互操作性标准	226
9.4 电源管理	227
9.5 智能设备	228
9.5.1 多队列网卡	229
9.5.2 SR-IOV	229
9.5.3 其他	231
索引	233
参考文献	237

CHAPTER 1

第 1 章

开 篇

虚拟化(Virtualization)以各种形式存在已经有四十多年的时间了。它对于不同的人来说可能意味着不同的东西,这要取决于他们所从事工作领域的环境。有经验的程序员可能还记得,曾有一段时间他们担心是否有可用内存来存放自己的程序指令和数据,于是出现了虚拟内存。后来,为了更好地时分共享(Time-sharing)昂贵的大型机系统,出现了虚拟服务器。然而,虚拟化技术的内涵远远不止于虚拟内存和虚拟服务器。目前,已经有了网络虚拟化、微处理器虚拟化、文件虚拟化和存储虚拟化等技术,如果在一个更广泛的环境中或从更高级的抽象(如任务负载虚拟化和信息虚拟化)来思考虚拟化技术,虚拟化技术就变成了一个非常强大的概念,可以为最终用户、应用程序和企业提供很多优点。抽象来说,虚拟化是资源的逻辑表示,它不受物理限制的约束。具体来说,虚拟化技术的实现形式是在系统中加入一个虚拟化层,虚拟化层将下层的资源抽象成另一形式的资源,提供给上层使用。通过空间上的分割、时间上的分时以及模拟,虚拟化可以将一份资源抽象成多份。反过来,虚拟化也可以将多份资源抽象成一份。总的来说,虚拟化可以把一个纷繁复杂、无计划性的世界改造成一个似乎是为人们的特定需求而度身订造的世界。

系统虚拟化是虚拟化技术中的一种,其抽象的粒度是整个计算机。早在 20 世纪 60 年代这个名称就已经诞生,从这个程度上来说,这是一个和操作系统有着同样悠久历史的领域。在虚拟化技术发展的几十年历程中,它经历了数次大幅度的起落,人们不断被虚拟化技术潜在的功能所吸引,然后又因客观技术上的限制而放弃。但是,随着近年来处理器技术和性能的迅猛发展,虚拟化技术成熟的时机真正到来。尤其是硬件虚拟化技术的诞生(例如 Intel VT 和 AMD SVM 技术),极大地扩展了虚拟化技术的应用范围。本章先简单介绍一些常见的虚拟化概念,然后着重介绍系统虚拟化,包括它的发展历史、特点以及系统虚拟化会带来什么好处。

1.1 形形色色的虚拟化

现代计算机系统是一个庞大的整体,整个系统的复杂性是不言而喻的。因而,计算机系统被分成了多个自下而上的层次。图 1-1 所示的是一种常见的计算机系统中的抽象层。每一个层次都向上一层次呈现一个抽象,并且每一层只需知道下层抽象的接口,而不需要了解其内部运作机制。例如,操作系统所看到的硬件是一个硬件抽象层,而不需要理解硬件的布线或者电气特性等。这样,以层的方式抽象资源的好处是每一层只需要考虑本层设计以及与相邻层间的相互交互,从而大大降低了系统设计的复杂性,提高了软件的移植性。

如图 1-1 所示,在硬件与操作系统之间的是硬件抽象层,在操作系统与应用程序或函数库之间的是 API 抽象层。硬件抽象层(Hardware Abstraction Layer, HAL)是计算机中软件所能控制的硬件的抽象接口,通常包括 CPU 的各种寄存器、内存管理模块、I/O 端口和内存映射的 I/O 地址等。API 抽象层抽象的是一个进程所能控制的系统功能的集合,包括创建新进程、内存申请和归还、进程间同步与共享、文件系统和网络操作等。

本质上,虚拟化就是由位于下层的软件模块,通过向上一层软件模块提供一个与它原先所期待的运行环境完全一致的接口的方法,抽象出一个虚拟的软件或硬件接口,使得上层软件可以直接运行在虚拟的环境上。虚拟化可以发生在现代计算机系统的各个层次上,不同层次的虚拟化会带来不同的虚拟化概念。因此,在学术界和工业界里,也先后出现了各种形形色色的虚拟化概念。下面介绍一些常见的虚拟化概念。

在介绍各种虚拟化概念之前,先介绍虚拟化中的两个重要名词。在虚拟化中,物理资源通常有一个定语称为宿主(Host),而虚拟出来的资源通常有一个定语称为客户(Guest)。根据资源的不同,这两个名词的后面可以接不同的名词。例如,如果是将一个物理计算机虚拟为一个或多个虚拟计算机,则这个物理计算机通常也被称为宿主机(Host Machine),而其上运行的虚拟机被称为客户机(Guest Machine)。宿主机上如果运行有操作系统,通常称为宿主机操作系统(Host OS),而虚拟机中运行的操作系统被称为客户机操作系统(Guest OS)。

1. 硬件抽象层上的虚拟化

硬件抽象层上的虚拟化是指通过虚拟硬件抽象层来实现虚拟机,为客户机操作系统呈现和物理硬件相同或相近的硬件抽象层。由于客户机操作系统所能看到的是硬件抽象层,因此,客户机操作系统的行为和在物理平台上没有什么区别。通常来说,宿主机和客户机的

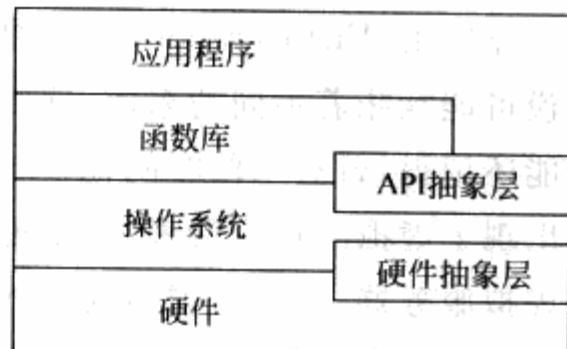


图 1-1 计算机系统的各个抽象层

ISA(Instruction Set Architecture, 指令集架构)是相同的, 客户机的大部分指令可以在宿主处理器上直接运行。只有那些需要虚拟化的指令才会由虚拟化软件进行处理, 从而大大降低了虚拟化开销。另外, 客户机和宿主机的硬件抽象层的其他部分如中断控制器、设备等, 可以是完全不同的, 当客户机对硬件抽象层访问时, 虚拟化软件需要对此进行截获并模拟。比较知名的硬件抽象层上的虚拟化有 VMware 的系列产品、Xen 等。

2. 操作系统层上的虚拟化

操作系统层上的虚拟化是指操作系统的内核可以提供多个互相隔离的用户态实例。这些用户态实例(经常被称为容器)对于它的用户来说就像是一台真实的计算机, 有自己独立的文件系统、网络、系统设置和库函数等。从某种意义上说, 这种技术可以被认为是 UNIX 系统 chroot 命令的一种延伸。因为这是操作系统内核主动提供的虚拟化, 因此操作系统层上的虚拟化通常非常高效, 它的虚拟化资源和性能开销非常小, 也不需要有硬件的特殊支持。但它的灵活性相对较小, 每个容器中的操作系统通常必须是同一种操作系统。另外, 操作系统层上的虚拟化虽然为用户态实例间提供了比较强的隔离性, 但其粒度是比较粗的。因为操作系统层上虚拟化的高效性, 它被大量应用在虚拟主机服务环境中。比较有名的操作系统级虚拟化解决方案有 Parallels 的 Virtuozzo, Solaris 的 Zone 和 Linux 的 VServer 等。

3. 库函数层上的虚拟化

操作系统通常会通过应用级的库函数提供给应用程序一组服务, 例如文件操作服务、时间操作服务等。这些库函数可以隐藏操作系统内部的一些细节, 使得应用程序编程更为简单。不同的操作系统库函数有着不同的服务接口, 例如 Linux 的服务接口是不同于 Windows 的。库函数层上的虚拟化就是通过虚拟化操作系统的应用级库函数的服务接口, 使得应用程序不需要修改, 就可以在不同的操作系统中无缝运行, 从而提高系统间的互操作性。例如, WINE 系统是在 Linux 上模拟了 Windows 的库函数接口, 使得一个 Windows 的应用程序能够在 Linux 上正常运行。

4. 编程语言层上的虚拟化

另一大类编程语言层上的虚拟机称为语言级虚拟机, 例如 JVM (Java Virtual Machine) 和微软的 CLR(Common Language Runtime)。这一类虚拟机运行的是进程级的作业, 所不同的是这些程序所针对的不是一个硬件上存在的体系结构, 而是一个虚拟体系结构。这些程序的代码由虚拟机的运行时支持系统首先翻译为硬件的机器语言, 然后再执行。通常一个语言类虚拟机是作为一个进程在物理计算机系统中运行的, 因此, 它属于进程级虚拟化。

1.2 系统虚拟化

系统虚拟化是指将一台物理计算机系统虚拟化为一台或多台虚拟计算机系统。每个虚拟计算机系统(简称为虚拟机)都拥有自己的虚拟硬件(如 CPU、内存和设备等), 来提供一

个独立的虚拟机执行环境。通过虚拟化层的模拟，虚拟机中的操作系统认为自己仍然是独占一个系统在运行。每个虚拟机中的操作系统可以完全不同，并且它们的执行环境是完全独立的。这个虚拟化层被称为虚拟机监控器（Virtual Machine Monitor, VMM）。如图 1-2 所示。

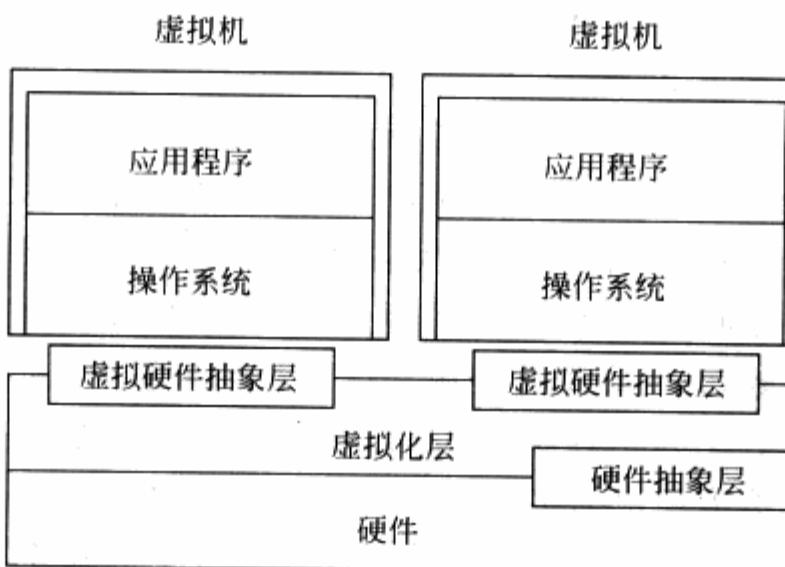


图 1-2 系统虚拟化

从本质上来说，虚拟计算机系统和物理计算机系统可以是两个完全不同 ISA 的系统。例如，可以在一个 x86 的物理计算机上运行一个安腾的虚拟计算机。但是，不同的 ISA 使得虚拟机的每一条指令都需要在物理机上模拟执行，从而造成性能上的极大下降。当然，相同体系结构的系统虚拟化通常会有比较好的性能，VMM 实现起来也会比较简单。虚拟机的大部分指令可以在处理器上直接运行，只有那些需要虚拟化的指令才会由 VMM 进行处理。

1974 年，Popek 和 Goldberg 定义了虚拟机可以看作是物理机的一种高效隔离的复制。上面的定义里蕴涵了三层含义（同质、高效和资源受控），这也是一个虚拟机所具有的三个典型特征。

所谓同质是指虚拟机的运行环境和物理机的环境在本质上需要是相同的，但是在表现上能够有一些差异。例如，虚拟机所看到的处理器个数可以和物理机上实际的处理器个数不同，处理器主频也可以与物理机的不同，但是物理机上和虚拟机中看到的处理器必须是同一种基本类型的。

所谓高效是指虚拟机中运行的软件需要有接近在物理机上直接运行的性能。为了做到这一点，软件在虚拟机中运行时，大多数的指令需要直接在硬件上执行，只有少量指令需要经过 VMM 处理或模拟。

所谓资源受控是指 VMM 需要对系统资源有完全控制能力和管理权限，包括资源的分配、监控和回收。

系统虚拟化最早出现于 20 世纪 60 年代的 IBM System360 上。经过一段时间的沉寂

后,相同 ISA 的系统虚拟化从 20 世纪 90 年代开始再次取得了长足的发展,不论从硬件还是软件上都涌现出了很多的技术和解决方案。本书将围绕相同 ISA 的系统虚拟化展开,以众所周知的 x86 架构为例,介绍系统虚拟化的分类、VMM 的实现原理、系统虚拟化的应用模式以及将来的发展方向。在本书的后续章节中,如果没有特指的话,则所指的虚拟化就是相同 ISA 的系统虚拟化。

1.3 系统虚拟化简史

历史上第一个虚拟机是 1965 年左右 IBM 公司开发的 System/360 Model 40 VM。其最初的设计目的是将当时最先进的虚拟内存的概念延展到计算机的其他子系统,搭建一个时分共享的系统,运行多个单用户的操作系统,以实现多个用户对昂贵物理计算机资源的共享。之后,随着时分多用户操作系统的发展,虚拟化技术真正成熟是在 15 年后的 IBM VM/370 系统中。作为一个标志性的系统,VM/370 的许多原理至今还在 IBM 的 z 系列大型机上使用。VM/370 运行在 IBM System/370 大型机上,并虚拟出同体系结构的 System/370 虚拟机。由于 System/370 硬件的优秀设计,对 VM/370 的实现提供了完整而高效的支持,以 VM/370 为代表的虚拟机和系统虚拟化技术完整实现了虚拟机的思想。但是,由于其应用于大型机上,并且面临的问题与背景和目前的服务器与个人计算机环境有所不同,因此与本书主要介绍的系统虚拟化技术不同。为了区分两者,把前者称为传统系统虚拟化或经典系统虚拟化。传统系统虚拟化是完全虚拟化(Full Virtualization),它所抽象的虚拟计算机具有完全的物理计算机特性。在这里不得不提一下另一个出现在同一时代,事实上还略早的概念——半虚拟化(Partial-virtualization)。半虚拟化最早出现在 IBM M44/44X 系统中,它提供了对底层硬件的部分模拟,以满足某些专门的软件的执行环境,但是并不能运行所有可能运行在物理机上的软件。半虚拟化技术的出现直接导致其后虚拟化技术的出现。

伴随着硬件的发展,从 20 世纪 90 年代后期开始,台式计算机的性能逐渐达到支持多个系统同时运行的水平。在大型机上沉寂一时的虚拟化技术,在小型机和微机领域开始迅速升温。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。基于 Disco 系统的研究经验,Disco 开发者们继续进行了个人计算机上虚拟化技术的研究,之后就有了 1998 年 VMware 公司的诞生。举棋止搏

在个人计算机领域广泛使用的 x86 体系结构的先天设计,存在对系统虚拟化的支持缺陷或虚拟化漏洞(Virtualization Hole)。在 x86 体系结构上的虚拟化技术,都需要用软件的方法来弥补体系结构设计上的不足。例如,采用代码扫描与修补方法(Scan-and-patch)或二进制代码翻译(Binary Translation)技术来实现基于软件的完全虚拟化。但这样做势必带来性能上的损失以及非常大的软件复杂度。

在这种情况下,学术界提出了另一种思路来克服体系结构上的缺陷,叫做类虚拟化技术(Para-Virtualization)。其主要思想是通过客户机操作系统与虚拟化管理层的协同设计,由

虚拟化管理层软件提供一个近似于原物理系统,但又不完全相同(与原系统)的虚拟平台,以避免虚拟化漏洞和实现更高的虚拟化效率。虚拟化技术需要修改操作系统的源代码来与下层的虚拟化管理层软件协同工作,从而避免体系结构上的缺陷。美国华盛顿州大学的 Denali 项目和源自英国剑桥大学的 Xen 项目都支持类虚拟化。在部分国内书籍和文章上,类虚拟化也被翻译成半虚拟化或部分虚拟化,但这是不确切的翻译,会与前面真正的半虚拟化混淆。

虽然上述两种基于软件的方法都能够实现系统虚拟化,但是它们各自存在不可回避的问题。基于软件的完全虚拟化方法不可避免地会导致性能上的下降,同时伴随着一些兼容性上的损失。而修改操作系统的方法对于现有系统的移植和伴随着内核升级的维护提出了要求,并且它对于非开源的操作系统也有局限性。

从根本上解决体系结构上的缺陷,最好的方法是从体系结构本身入手。Intel 公司和 AMD 公司在 2006 年以后都逐步推出了带有硬件虚拟化支持的处理器,如 Intel 公司的 Virtualization Technology(VT)技术和 AMD 公司的 Secure Virtual Machine(SVM) 技术,从根本上保证了 x86 架构是一个可虚拟化的架构。Intel 公司和 AMD 公司采用的是硬件辅助的完全虚拟化策略。Intel 公司的 VT 技术和 AMD 公司的 SVM 技术弥补了 x86 体系结构上的漏洞。操作系统不需要做任何修改就可以运行在虚拟机中。VMM 软件可以利用这些硬件虚拟化技术,这样软件的实现就可以极大地简化,并且也更为高效和安全。

另外,值得一提的是,除了 x86 处理器架构中加入了硬件的虚拟化支持外,其他处理器也先后纷纷在硬件中加入了虚拟化支持,以提供一个对虚拟化友好的体系结构。IBM 公司于 2001 年在 POWER4 处理器中加入了虚拟化的支持,2004 年在 POWER5 系列中推出了增强的虚拟化支持。2005 年,Sun 公司在 SPARC 处理器中第一次推出了虚拟化支持。有意思的是,作为系统厂商,IBM 公司和 Sun 公司都采用了一种整体的类虚拟化策略,除了在硬件里加入对虚拟化的支持外,还在固件层加入了相应的类虚拟化 VMM 层。图 1-3 给出了 IBM POWER5 虚拟化架构的示意图。在固件中的 POWER Hypervisor VMM 向上呈现了一个称为 PFW 的接口,上层的操作系统(如 AIX、Linux 等)需要根据这个类虚拟化接口进行修改。由此可见,这种虚拟化策略需要从处理器层到固件层,再到操作系统层对虚拟化的全面协同支持,而这对 IBM 和 Sun 这类系统厂商并不是难事。对于系统厂商来说,他们不仅能轻而易举地控制自己的处理器和固件层,而且也很容易修改它们的操作系统,例如 IBM 就拥有自己的 AIX 操作系统,Sun 也拥有自己的 Solaris 操作系统。因此,要修改它们对于 IBM 和 Sun 来说是件很容易的事。

今天的大部分服务器和台式机处理器中都已经有了对虚拟化的支持,但这只是解决了处理器层如何更好地支持虚拟化的问题。为了使虚拟化解决方案更加高效(例如 I/O 虚拟化),计算机系统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生态系统。以 Intel 为例,除了处理器中的 VT 技术之外,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队列

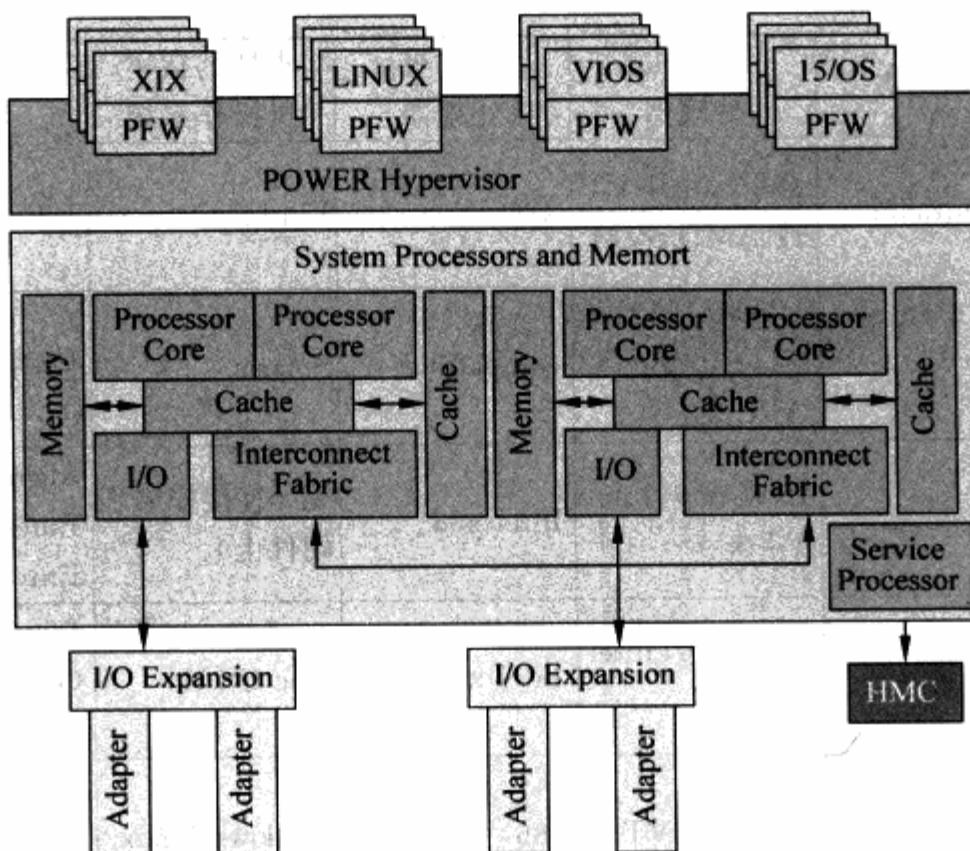


图 1-3 IBM POWER5 虚拟化架构

的 VMDq 技术等。与此同时,PCI 标准组织也在积极地制定在 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。

从 20 世纪 90 年代发展至今,虚拟化软件已经取得了长足的发展,呈现出一派百花齐放的繁荣景象。除了上面提到的 VMware、Denali 和 Xen 之外,还有非常多的新兴虚拟机软件涌现出来,例如 KVM、VirtualBox、微软的虚拟化系列产品线(如 VirtualPC、Hyper-V)、Parallels 的虚拟化系列产品线(如 Virtuozzo、Parallels Desktop for Mac)、Citrix 的 XenServer、Sun 的 xVM、Oracle 的 Oracle VM 和 VirtualIron 等。表 1-1 中列出了一些知名的虚拟化软件及相关信息。

表 1-1 知名的虚拟化软件

虚拟化软 件名称	创建者	宿主机 处理器	客户机 处理器	支持的宿主 机操作系统	支持的客户 机操作系统
Denali	美国华盛顿州大学	x86	x86	Denali	Iwaco, NetBSD
Hyper-V	Microsoft	x86 + Intel VT 或 AMD SVM	x86	Windows 2008 w/ Hyper-V Role	Supported Drivers for Windows 2000, Windows 2003, Windows 2008, Windows XP, Windows Vista, Linux

续表

虚拟化软件名称	创建者	宿主机处理器	客户机处理器	支持的宿主机操作系统	支持的客户机操作系统
KVM	Qumranet	x86 + Intel VT 或 AMD SVM	x86	Linux	Linux, Windows
Logical Domains	Sun	UltraSPARC T1, UltraSPARC T2	compatible	Solaris	Solaris, Linux 和 FreeBSD
Oracle VM	Oracle	Intel x86, Intel VT-x	Intel x86	无(直接安装在硬件上)	Microsoft Windows, Oracle Enterprise Linux, Red Hat Enterprise Linux
Parallels Desktop for Mac	Parallels	Intel x86, Intel VT-x	Intel x86	Mac OS X	Windows, Linux, FreeBSD, OS/2, eComStation, MS-DOS, Solaris
Parallels Workstation	Parallels	x86, Intel VT-x	x86	Windows, Linux	Windows, Linux, FreeBSD, OS/2, eComStation, MS-DOS, Solaris
Sun xVM	Sun	x86, SPARC	与宿主机处理器相同	无(直接安装在硬件上)	Windows XP & 2003 Server, Linux, Solaris
VirtualBox	Sun	x86	x86	Windows, Linux, Mac OS X, Solaris	DOS, Windows, Linux, OS/2, FreeBSD, Solaris
Virtual Iron Virtual Iron 3.1	Virtual Iron	x86、Intel VT 或 AMD SVM	x86	无(直接安装在硬件上)	Windows, Linux
Virtual PC 2007	Microsoft	x86	x86	Windows Vista, XP Pro, XP Tablet PC Edition	DOS, Windows, OS/2, Linux, OpenSolaris
Virtual PC 7 for Mac	Microsoft	PowerPC	x86	Mac OS X	Windows, OS/2, Linux
VirtualLogix VLX	VirtualLogix	ARM, TI DSP C6000, Intel x86, Intel VT-x 和 VT-d, PowerPC	与宿主机处理器相同	无(直接安装在硬件上)	Linux, Windows XP, C5, VxWorks, Nucleus, DSP/BIOS 和 proprietary OS
Virtual Server 2005 R2	Microsoft	x86	x86	Windows 2003, XP	Windows NT, 2000, 2003, Linux
VMware ESX Server 3.0	VMware	x86, AMD64	x86, AMD64	none(bare metal install)	Windows, Red Hat, SuSE, Netware, Solaris
VMware Fusion	VMware	x86, Intel VT-x	x86	Mac OS X	Windows, Linux, Netware, Solaris, others

续表

虚拟化软件名称	创建者	宿主机处理器	客户机处理器	支持的宿主机操作系统	支持的客户机操作系统
VMware Server	VMware	x86	x86	Windows, Linux	DOS, Windows, Linux, FreeBSD, Netware, Solaris, Virtual appliances
VMware Workstation 6.0	VMware	x86	x86	Windows, Linux	DOS, Windows, Linux, FreeBSD, Netware, Solaris, Darwin, Virtual appliances
VMware Player 2.0	VMware	x86	x86	Windows, Linux	DOS, Windows, Linux, FreeBSD, Netware, Solaris, Darwin, Virtual appliances
Xen	英国剑桥大学	x86, 安腾, Power PC	与宿主机处理器相同	NetBSD, Linux, Solaris	Linux, Solaris, Windows XP & 2003 Server, Plan 9
z LPARs	IBM	z/Architecture	z/Architecture	系统内置	Linux, z/OS, z/VSE, z/TPF, z/VM, VM/CMS, MUSIC/SP

小型机和微机领域的虚拟化经过十多年的发展,今天已经形成了一个良好的生态系统,包括从各有特色的各种虚拟化软件,到各个层次的硬件(如处理器、芯片组和设备等)对虚拟化的支持等。可以说,虚拟化 1.0 时代已经非常成功了。随着整个信息产业界的不断发展,年轻的虚拟化将会迎来更多的发展机遇和进一步的需求。例如,云计算 (Cloud Computing)将是人类使用计算资源方式的一个重大变革方向。云计算的一个核心思想就是在服务器端提供集中的计算资源,同时这些计算资源要独立地服务于不同的用户,也就是在共享的同时,为每个用户提供隔离、安全、可信的工作环境。虚拟化技术将是云计算的一个基础架构。通俗地说,云计算实际上是一个虚拟化的计算资源池,用来容纳各种不同的工作模式,这些模式可以快速部署到物理设施上。虚拟化的资源按照来自用户的需求多少动态调动资源,每个用户都有一个独立的计算执行环境。虚拟化如何为云计算的发展提供一个自适应、自管理的灵活基础架构将是一个富有挑战性的话题。

1.4 系统虚拟化的好处

系统虚拟化提供了多个隔离的执行环境,这种将运行环境完整地封装起来带来的好处是很多的。如图 1-4 所示,以虚拟机为粒度的抽象提供了优秀的封装性,使得一台计算机上能够运行多个虚拟机,虚拟机之间有很强的隔离性,虚拟机与硬件没有直接的关联。另外,虚拟化层作为特权层能够提供前所未有的功能。

虚拟机抽象				
封装	多实例	虚拟机间隔离	与硬件脱离	特权功能
虚拟机克隆、虚拟机快照、虚拟机挂起与恢复	优化资源调度	故障隔离	虚拟机迁移 兼容旧系统	事件记录与回放 入侵检测与防护
灾难恢复、便利软件测试和调试	服务器合并、节能	硬件维护、负载平衡		

图 1-4 系统虚拟化的功能

1. 封装性

以虚拟机为粒度的封装使得虚拟机运行环境的保存非常便捷。虚拟机的优秀封装性使得以下应用模式可以很方便地实现。

(1) 虚拟机快照(Snapshot)。将运行中的一个虚拟机的某个时间点的状态抓取下来，就像抓拍一张照片一样。

(2) 虚拟机克隆(Clone)。从一个虚拟机的执行环境复制出一个或多个相同的虚拟机。

(3) 虚拟机挂起(Suspend)。指暂停一个运行中的虚拟机，将其运行环境保存在磁盘上。与之相反，虚拟机恢复(Resume)是指将保存在磁盘上的虚拟机运行环境恢复到内存中以继续运行的操作。

优秀的封装性使得虚拟机的保存更容易，从而在灾难恢复中发挥更大的作用。虚拟机快照和克隆等使得部署各种软件运行环境更容易，从而使软件开发的测试和调试更为快捷方便。虚拟化最早起源于服务器虚拟化，目前桌面虚拟化也逐渐开始在企业环境中发展起来。优秀的封装性使得企业中可以方便地进行桌面软件维护管理，从而大大降低了IT的管理维护成本。

2. 多实例

在一个计算机上运行多个虚拟机使得资源的调度更为优化。不同的虚拟机有不同的繁忙和空闲时段，忙闲交错使得单个计算机的系统资源利用率大大提高。出于上述原因，产业界大力推广了服务器整合(Server Consolidation)，使得多个物理服务器合并到少数几个计算机上，作为虚拟机来运行。这可以用更少的服务器获得同样的整体性能，并大大提高计算机性能的利用率。伴随着处理器性能的不断提升，特别是多核时代到来后，虚拟化的服务器整合比也在不断提高。由于虚拟化技术拥有高效、节能省电和节省空间等多种优势，无论是大型企业数据中心整合还是中小型企业的经济型服务器选型，虚拟化都在其中扮演着重要的角色。事实上，实施服务器虚拟化可以让客户获得更大的收益。虚拟化技术能够为公司节约大量的成本，降低了系统管理成本，节约人力，提高旧业务系统的性能，还降低了新系统的开发部署成本。

3. 隔离

尽管不使用虚拟化也可以对不同的工作负载进行整合，但是虚拟机所具有的隔离性可以提供一些重要的特性。使用虚拟机，每个应用程序可以在自己的操作系统环境中独立地

运行,而不会影响到其他的工作负载。例如,如果一个虚拟机的操作系统由于故障或受到恶意破坏崩溃了,其他虚拟机中的应用程序仍然可以继续正常运行。故障或破坏被天然地封闭在一个虚拟机中。这种隔离性支持多个用户在同一台物理服务器上对不同的应用程序进行独立的操作。例如,虚拟机所提供的隔离特性非常适合于测试场景,支持并行地和隔离地执行多项测试。同时,这种隔离性对于系统安全十分重要,也是安全软件公司利用虚拟机作为诱饵吸引攻击的原因之一。

4. 硬件无关性

正如前面谈到的,虚拟化是资源的逻辑表示,它不受物理限制的约束。由于虚拟化层的抽象,虚拟机与底层的硬件没有直接的绑定关系。因此,尽管目前计算机体系结构呈现出很大的异构性,但只要另一台计算机提供相同的虚拟硬件抽象层,一个虚拟机就能够无缝地迁移过去。虚拟机迁移是虚拟化技术中的亮点之一。

有了虚拟机的迁移,硬件的维护就不必每次都关闭计算机了。在一台计算机需要硬件维护时,其中运行的虚拟机能够迁移到其他计算机上,等维护工作结束后再迁移回来。在计算机集群的环境下,虚拟机的迁移能够提供集群以负载平衡的功能。在集群中一些计算机工作量较大而一些计算机比较空闲的情况下,一部分运行在繁忙的计算机上的虚拟机能够迁移到较空闲的计算机上。

另外,虚拟硬件抽象层不需要与物理硬件相匹配,因而还可以虚拟出较旧的硬件来兼容旧系统软件。

5. 特权功能

由于虚拟化层处于客户机及客户机操作系统的下面,其具有更高的特权级。在这个层中添加新的功能有如下两个优势。

- (1) 新的功能有高特权级,不能被客户机操作系统绕过。
- (2) 新的功能不需要了解客户机内部的语义,使其实现上更容易。

目前,在研究和工业界领域已经有大量基于虚拟化特权功能特性的研究和应用。

事件记录与回放(Log and Replay)在虚拟化层中能够很方便地实现,因为虚拟化层截获并传递所有系统级事件。

入侵检测(Intrusion Detection)也是在虚拟化层中添加的功能之一。利用虚拟化层所在的特权级别,入侵检测系统能够保证其检测不会被客户机绕过,同时客户机也不会知道这个服务的存在。

病毒检测等安全防范机制同样也能够在虚拟化层中实现,其好处是不需要担心检测系统自身的安全,客户机中的病毒很难攻击到下层,甚至不会知道自身是运行在客户机中。另一个好处是检测不会被上层客户机绕过。

x86 架构及操作系统概述

在读者深入奇妙的虚拟技术世界开始探索前,了解一些 CPU 架构和操作系统的知识是必须的。本章的内容可以帮助读者更好地理解后面的相关章节,如果读者对这些内容已经有所了解,可以直接跳过本章进行阅读。

CPU 架构和操作系统的知识是浩瀚的,限于篇幅,这里无法对所有部分都深入讲解,但会尽力将它们的思想传达给读者。

2.1 x86 的历史和操作系统概要

2.1.1 x86 的历史

作为世界上最流行的处理器架构,Intel 的 x86 体系结构历史悠久,从 1978 年 8086/8088 处理器的问世到现在的 Core 2 Duo 和 Core 2 Quad,以及 Xeon 5300 和 7300 系列处理器,Intel x86 体系接口已经经历了整整 30 年的历史。

最早的 x86 处理器 8086 和 8088 是 16 位的处理器。8086 处理器拥有 16 位的寄存器和 16 位的外部数据总线,使用 20 位地址寻址(拥有 1MB 的地址空间)。1982 年,Intel 公司发布了 80286 处理器,引入了保护模式的概念。3 年后,Intel 公司发布了 x86 体系结构下的第一款 32 位处理器 80386,并且引入了虚拟内存。1989 年,80486 发布,相比于其前身,80486 采用了 5 级流水线机制,并且引入了片上一级缓存和能量管理。1993 年,Intel 公司的第一款奔腾处理器发布(Pentium),此款处理器在 80486 的基础上,进一步增大一级缓存,并将其分成指令缓存和数据缓存两个部分,进一步加快了处理器对主存的访问时间。同时,奔腾处理器还引入了 MMX 技术,使得处理器对多媒体处理的支持进一步增强。从 1995 年到 1999 年,Intel 公司发布了一系列基于 x86 体系结构的处理器,这一系列处理器被称为 P6 家族处理器,包括奔腾 Pro(Pentium Pro)、奔腾 2(Pentium II)、奔腾 2 至强(Pentium II Xeon)、赛扬(Celeron)、奔腾 3(Pentium III)以及奔腾 3 至强(Pentium III Xeon)处理器。P6

家族处理器采用了超标量(superscalar)技术,以乱序执行的方式进一步增强了处理器的处理速度。从 2002 年开始到 2006 年,奔腾 4(Pentium 4)家族处理器占据了主导地位。奔腾 4 家族的处理器基于 NetBurst 微处理结构,在提升性能的同时,进一步增强了对多媒体处理的支持,并且引入了超线程的概念(Hyper-Threading),引领单处理器的性能走向极致。与此同时,在奔腾 4 的 672 和 662 处理器上,Intel 还首次加入了虚拟化支持,即 Intel VT 技术。从 2006 年起,处理器进入多核时代(Multicore),Intel 相继发布了 Core Duo 和 Core 2 Duo 系列处理器。至此,x86 体系结构走过了其 30 年的历程。

2.1.2 操作系统概述

操作系统作为硬件平台上最重要的软件,对下负责管理平台硬件,对上向应用程序提供标准接口。操作系统中最重要的部分称为操作系统内核,运行在 CPU 最高的特权级上,可以访问系统的一切资源,称操作系统内核运行的状态为内核态。应用程序通常运行在 CPU 最低的特权级上,只能访问部分资源,此种状态称为用户态。

操作系统利用平台架构提供的各种功能,使用硬件资源,其实现和平台架构是紧密相关的。在后面几节的内容中会对 x86 架构提供给操作系统的各种功能,以及操作系统通常是如何使用它们的进行介绍。

2.2 x86 内存架构

内存架构往往是硬件架构中最为复杂的部分。可以毫不夸张地说,理解了内存架构,就理解了现代计算机体系架构的大部分内容。下面介绍 x86 为操作系统提供了什么样的内存架构,以及操作系统是如何使用它们的。

2.2.1 地址空间

很多教科书把内存(这里特指安插在主板上的 RAM)比作一个大数组,地址就是这个数组的索引。与之类似,地址空间则是个更大的数组,它是所有可用资源的集合,同样,地址是这个数组的索引。地址空间可以划分成如下两种类型。

1. 物理地址空间

硬件平台可以粗略地划分成三个部分:CPU、内存和其他硬件设备。其中,CPU 是平台的主导者,从 CPU 的角度来看,内存和其他硬件设备都是可以使用的资源。这些资源整合在一起,分布在 CPU 的物理地址空间内。如同名字的暗示,CPU 使用物理地址索引这些资源。物理地址空间的大小,由 CPU 实现的物理地址位数所决定,物理地址位数和 CPU 处理数据的能力(即 CPU 位数)没有必然的联系,例如 16 位的 8086 CPU 具有 20 位地址空间。

前面提到,内存和其他硬件设备分布在物理地址空间内。用一个例子可以很容易地说

明这个问题。假设一个平台,CPU 的物理地址空间为 4GB,有 512MB 内存,其他硬件设备的 I/O 寄存器被映射到 512MB 的 I/O 地址内,则该平台的物理地址空间可能是如图 2-1 所示划分的。

从图 2-1 中可以看出,512MB 内存和 I/O 地址只占用物理地址空间的一部分,还有大部分处于空闲。数组再次显示了它的作用,有一个 4GB 大小的数组,其中 1GB 的元素具有有效值(512MB 内存、512MB I/O 地址),其他元素不存在。

2. 线性地址空间

一个平台只有一个物理地址空间,但每个程序都认为自己独享整个平台的硬件资源,为了让多个程序能够有效地相互隔离和使用物理地址空间的资源,线性地址空间的概念被引入了。和物理地址空间一样,线性地址空间的大小取决于 CPU 实现的线性地址位数,例如实现了 32 位线性地址的 CPU 具有 4GB 大小的线性地址空间。需要注意的是,线性地址空间的大小和物理地址空间的大小没有必然联系。例如,Intel 的 PAE 平台就具有 4GB 的线性地址空间,64GB 的物理地址空间。

线性地址空间会被映射到物理地址空间某一部分或整个物理地址空间。CPU 负责将线性地址转换成物理地址,使程序能够正确访问到该线性地址空间所映射到的物理地址空间。一个平台上可以有多个线性地址空间,在现代操作系统中,每个进程通常都拥有自己的私有线性地址空间。一个典型的线性地址空间构造如图 2-2 所示。

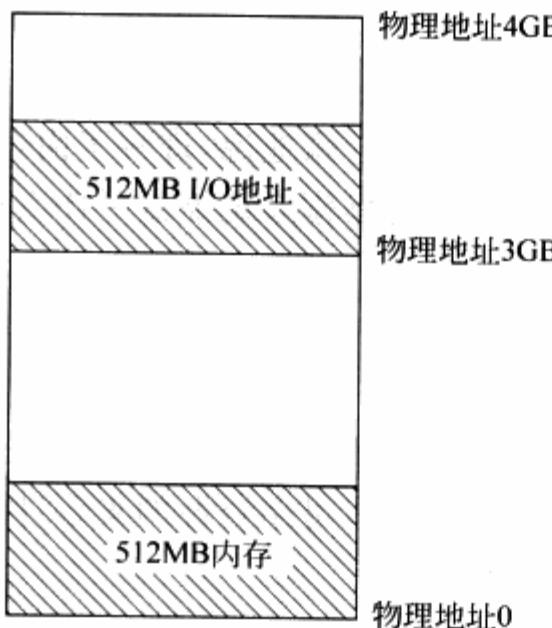


图 2-1 物理地址空间

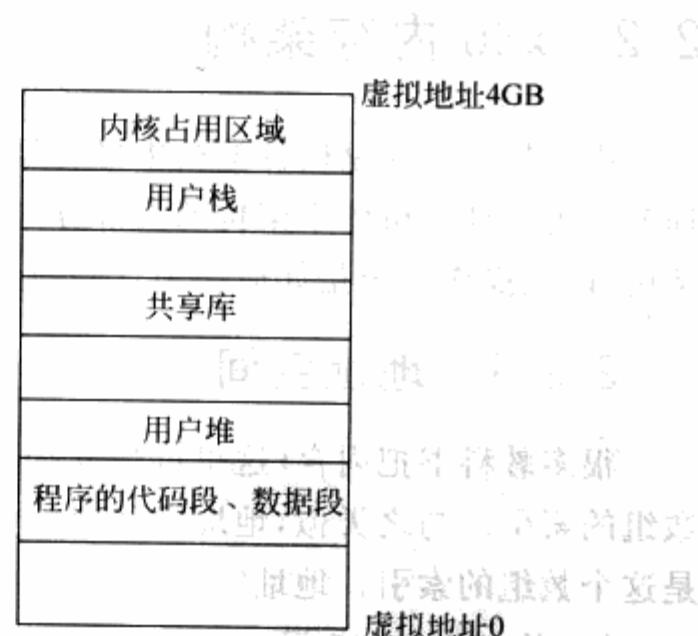


图 2-2 线性地址空间构造

2.2.2 地址

上一节已经讲到,地址是访问地址空间的索引。根据访问的地址空间不同,索引可以分为线性地址和物理地址。但由于 x86 特殊的段机制,还有一种额外的地址——逻辑地址。

1. 逻辑地址

该地址即程序直接使用的地址(x86 架构无法禁用段机制,逻辑地址一直存在)。例如

下面这个程序：

```
int a = 1;  
int * p = &a;
```

这里，指针变量 p 中存储的即是一个逻辑地址。逻辑地址由一个 16 位的段选择符和一个 32 位的偏移量(32 位平台)构成。逻辑地址的转换过程在后面介绍。在上面这个例子中，指针变量 p 实际上存储的是逻辑地址的偏移部分，该偏移对应的段选择符位于段寄存器中，并没在程序中反映出来。

2. 线性地址

又称虚拟地址。线性地址是逻辑地址转换后的结果，用于索引线性地址空间。当 CPU 使用分页机制时，线性地址必须转换成物理地址才能访问平台内存或硬件设备；当分页机制未启用时，线性地址等于物理地址。

3. 物理地址

该地址索引物理地址空间，是 CPU 提交到总线用于访问平台内存和硬件设备的最终地址。它和上面两个地址有如下关系。

(1) 分段机制启用，分页未启用：逻辑地址 → 线性地址 = 物理地址

(2) 分段、分页机制同时启用：逻辑地址 → 线性地址 → 物理地址

在某些书籍中还有“总线地址”的提法，这是因为给设备寄存器分配的物理地址和寄存器在设备上的地址是不同的(通常设备的寄存器都认为自己是从地址 0 开始的)，两者之间存在一个映射关系，由设备的电子线路负责转换并对 CPU 透明。由于 CPU 用于访问设备的物理地址是设备寄存器展现给总线的地址，所以在 x86 下有时也称物理地址为总线地址。

2.2.3 x86 内存管理机制

x86 架构的内存管理机制以复杂著称，这里面有着很多的历史原因。下面分别对 x86 的分段机制和分页机制进行介绍。

1. 分段机制

分段是一种朴素的内存管理机制，它将内存划分成以起始地址(Base)和长度(Limit)描述的块，这些内存块就称为“段”。段可以与程序最基本的元素联系起来，例如程序可以简单地分为代码段、数据段和栈，段机制中就有对应的代码段、数据段和栈段。

分段机制由 4 个基本部分构成：逻辑地址、段选择寄存器、段描述符和段描述符表。其核心思想是：使用段描述符描述段的基地址、长度以及各种属性(例如读写属性、访问权限)。当程序使用逻辑地址访问内存的某个部分时，CPU 通过逻辑地址中的段选择符索引段描述符表以得到该内存对应的段描述符，并检测程序的访问是否合法，如合法，根据段描述符中的基地址将逻辑地址转换为线性地址。

分段机制的流程可以用图 2-3 概括。

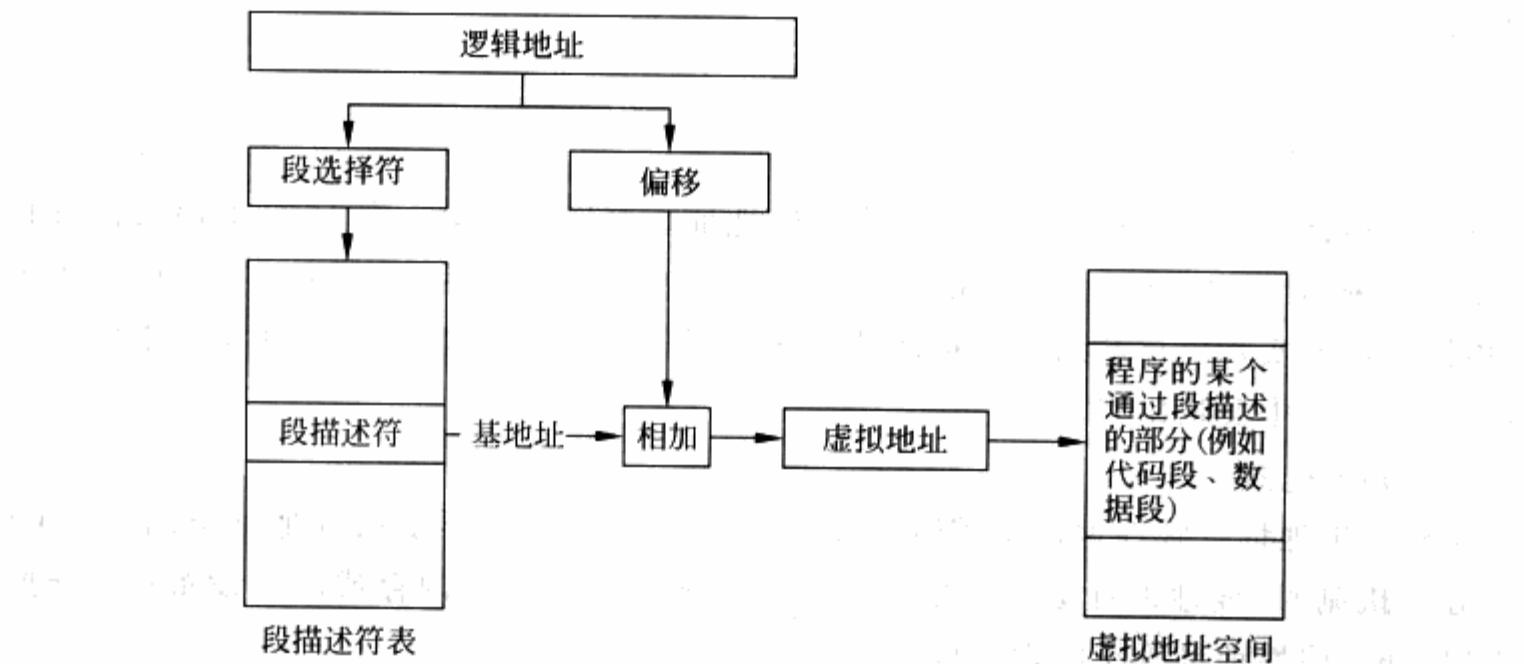


图 2-3 分段机制

(1) 段选择符(segment selector)

段选择符是逻辑地址的一个组成部分,共 16 位,用于索引段描述表以获得该段对应的段描述符,其结构如图 2-4 所示。

各字段的含义如下。

- Index: 段描述符表的索引。
- TI: 用于指明索引哪个段描述符表。0 为 GDT, 全局描述符表; 1 为 LDT, 本地描述符表。
- RPL: 具体内容见 2.3.3。

段选择符作为逻辑地址的一部分,对程序是可见的。但通常段描述符的修改和分配由连接器和加载器完成,而不是应用程序本身。

为了使 CPU 能够快速地获得段选择符,x86 架构提供了 6 个段寄存器(segment register)用于存放当前程序的各个段的段选择符。6 个段寄存器分别如下。

- CS(code-segment, 代码段): 存放代码段的段选择符。
- DS(data-segment, 数据段): 存放数据段的段选择符。
- SS(stack-segment, 栈段): 存放栈的段选择符。
- ES、FS、GS: 供程序自由使用,可以存放额外 3 个数据段的段选择符。

通常程序只使用 CS、DS、SS 三个段寄存器。

(2) 段描述符(Segment Descriptor)

段描述符描述某个段的基地址、长度以及各种属性。其结构如图 2-5 所示。

各个字段的详细解释,可查阅《Intel(R) 64 and IA-32 Architectures Software Developer's Manual Volume 3A》3.4.5 节。在这里,我们只关心其中的 Base 字段和 Limit 字段。前者

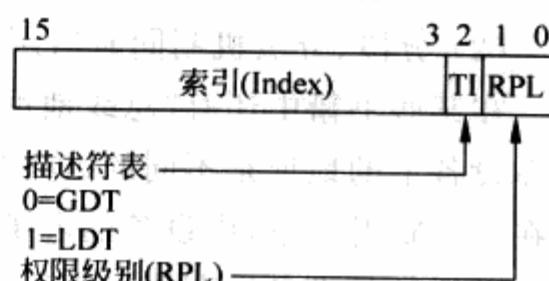
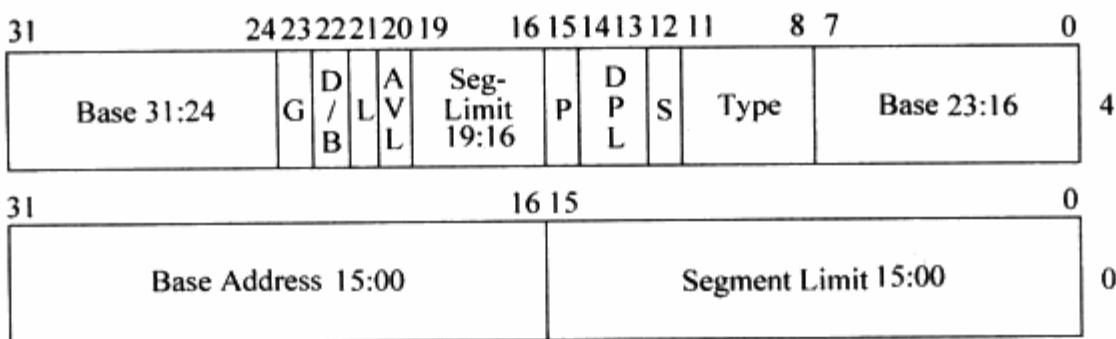


图 2-4 段选择符



L — 64-bit code segment (IA-32e mode only)
 AVL — Available for use by system software
 BASE — Segment base address
 D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
 DPL — Descriptor privilege level
 G — Granularity
 LIMIT — Segment Limit
 P — Segment present
 S — Descriptor type (0 = system; 1 = code or data)
 TYPE — Segment type

图 2-5 段描述符

描述了该段的基地址,后者描述了该段的长度。

当 CPU 通过一个逻辑地址的段选择符获得该段对应的段描述符后,会使用描述符中各种属性字段对访问进行检查,一旦访问被确认合法,CPU 将段描述符中的 32 位基地址和逻辑地址中的 32 位偏移量相加以获得该逻辑地址对应的线性地址。

为了加速段描述符的访问,x86 在段寄存器后增加了一个程序不可见的段描述符寄存器。当段寄存器被加载入一个新的段选择符后,CPU 自动将该段选择符索引的段描述符加载入这个不可见的符寄存器中,故 CPU 自需要在更新段寄存器时才索引段描述符表。

(3) 段描述符表

x86 架构提供两种段描述符表,它们是:全局段描述符表(Global Descriptor Table,下面简称为 GDT)和本地段描述符表(Local Descriptor Table,下面简称为 LDT)。

系统中至少有一个 GDT 可以被所有进程访问。相对的,系统中可以有一个或多个 LDT,可以被某个进程私有,也可以被多个进程共享。GDT 仅仅是内存中的一个数据结构,可以把它看成一个数组,由基地址(Base)和长度(Limit)描述。与之相反,LDT 是一个段,它需要一个段描述符来描述它。LDT 的段描述符存放在 GDT 中,当系统中有多个 LDT 时,GDT 中必须有对应数量的段描述符。

为了加速对 GDT 和 LDT 的访问,x86 提供了 GDTR 寄存器和 LDTR 寄存器。它们的描述如下。

- GDTR: 包括一个 32 位的地址(BASE)和一个 16 位长度(LIMIT)。
- LDTR: 结构同段寄存器(包括对程序不可见的段描述符寄存器)。

可以使用 LGDT/SGDT 指令对 GDTR 进行读取/存储,类似地,可以使用 LLDT/

SLDT 对 LDTR 进行同样的操作。通常在进程切换时,LDTR 中的值会被换成新进程对应的 LDT 的段描述符。

图 2-6 显示了通过段选择符索引 GDT/LDT 的过程。

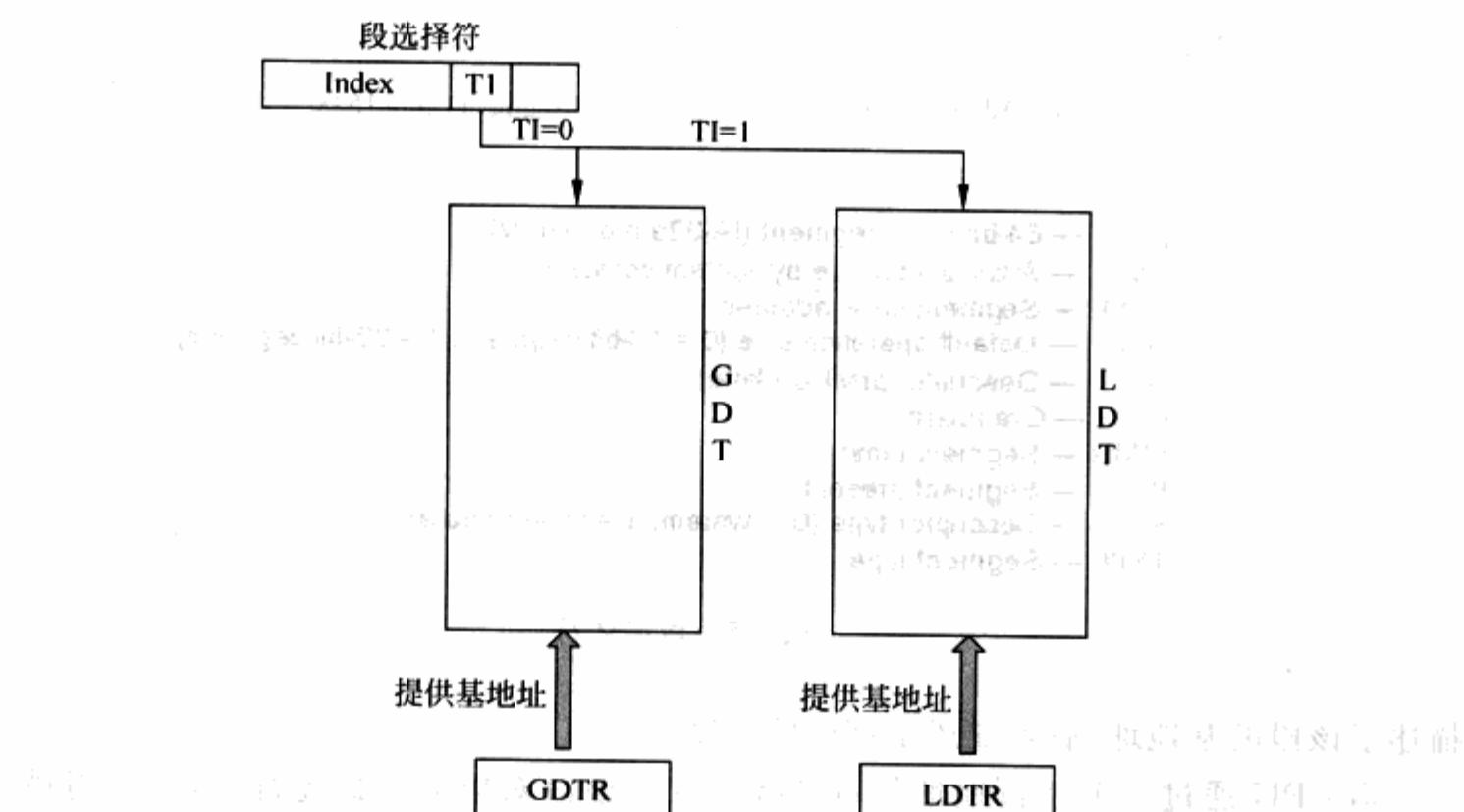


图 2-6 索引段描述符表

从图中可以看出,GDTR/LDTR 为 GDT/LDT 提供了地址,段选择符的 TI 位确定索引 GDT 还是 LDT。

(4) 逻辑地址转换总结

到此为止,读者应该已经清楚了逻辑地址的转换过程,下面用一个例子把这个过程梳理一下。

假设程序中某条语句访问了数据段,例如:

```
int a = 100 //全局变量
int func()
{
    int b;
    b = a;
}
```

程序从加载到变量 a 的逻辑地址转换为线性地址的过程如下。

(1) 程序加载。

① 通过该进程 LDT 的段选择符索引 GDT,获得 LDT 的段描述符,被加载到 LDTR 寄存器中。

② 该进程的 CS、DS、SS 被加载入相应的段选择符。同时，CPU 根据段选择符的 TI 字段，索引 GDT/LDT，获得相应的段描述符并加载入 CS、DS、SS 对应的不可见的段描述符寄存器。

(2) 程序执行到 $b=a$ ，需要从 a 所在的内存中取值，必须先把 a 的逻辑地址转换为线性地址。

- 1) 进行必要的属性、访问权限检查。
- 2) 从 DS 对应的段描述符寄存器获得该段的基址。
- 3) 将变量 a 的 32 位偏移量和描述符中的基地址相加，获得变量 a 的线性地址。

2. 分页机制

分页是更加粒度化的内存管理机制，与分段机制将内存划分成以基地址和长度描述的多个段进行管理不同，分页机制是用粒度化的单位“页”来管理线性地址空间和物理地址空间。架构下一个典型的页大小是 4KB，则一个 4GB 的虚拟地址空间可以划分成 1024×1024 个页面。物理地址空间的划分同理。架构允许大于 4KB 的页面大小(如 2MB、4MB)，限于篇幅关系，这里只介绍最为经典的 4KB 页面管理机制。

同时，分页机制让现代操作系统中的虚拟内存机制成为可能，感谢这种机制，一个页面可以存在于物理内存中，也可以存放在磁盘的交换区域(如 Linux 下的 Swap 分区，Windows 下的虚拟内存文件)中，程序可以使用比机器物理内存更大的内存区域。

分页机制的核心思想是通过页表将线性地址转换为物理地址，并配合旁路转换缓冲区(Translation Lookaside Buffer, TLB)来加速地址转换过程。操作系统在启动过程中，通过将 CR0 寄存器的 PG 位置 1 来启动分页机制。图 2-7 展示了分页机制的概要。

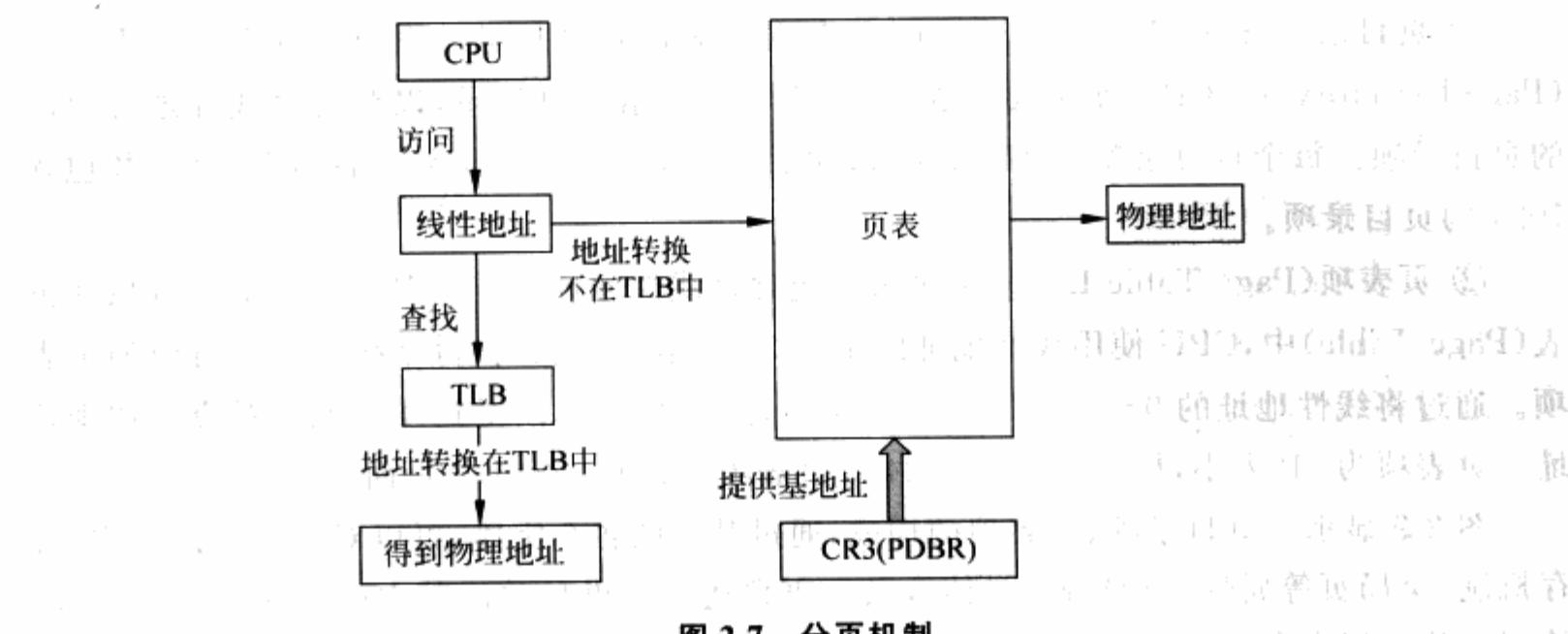


图 2-7 分页机制

从图中可以看出，分页机制主要由页表、CR3 寄存器和 TLB 三个部件构成。下面以 4KB 页大小为例，对各个部件进行讲解。

页表

页表(Page Table)是用于将线性地址转换成物理地址的主要数据结构。一个地址对齐到页边界后的值称为页帧号(或页框架),它实际是该地址所在页面的基地址。线性地址对应的页帧号即虚拟页帧号(Virtual Frame Number, VFN),物理地址对应的页帧号即物理页帧号(Physical Frame Number, PFN)或机器页帧号(Machine Frame Number)。故也可以认为,页表是存储 VFN 到 PFN 映射的数据结构。4KB 大小的页面使用两级页表,如图 2-8 所示。

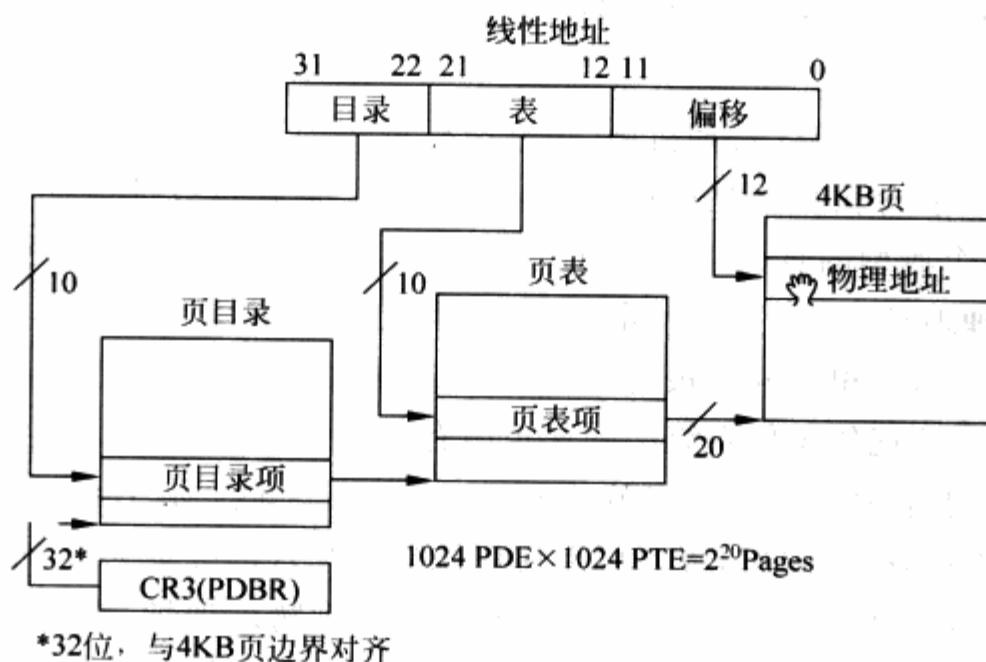


图 2-8 二级页表

① **页目录项(Page Directory Entry):** 包含页表的物理地址。页目录项存放在页目录(Page Directory)中,CPU 使用线性地址的 22~31 位索引页目录,以获得该线性地址对应的页目录项。每个页目录项为 4B 大小,故页目录占用一个 4KB 大小的物理页面,共包含 1024 的页目录项。

② **页表项(Page Table Entry):** 页表项包含该线性地址对应的 PFN。页表项存放在页表(Page Table)中,CPU 使用线性地址的 12~21 位索引页表,获得该线性地址对应的页表项。通过将线性地址的 0~11 位偏移量和基地址相加,就可以得到线性地址对应的物理地址。页表项为 4B 大小,故页表包含 1024 个页表项,占用一个 4KB 页面。

图 2-9 显示了页目录项、页表项的构成,通过其中的各个字段,可以对页面访问权限、缓存机制、全局页等属性进行控制,具体含义不再赘述。这里只关注其中的 P(Present)字段,该字段使虚拟内存的实现成为可能。P 字段根据其值不同,可以代表两种情况:

- 1) P=1: 物理页面存在于物理内存中,CPU 完成地址转换后,可以直接访问该页面。
- 2) P=0: 页面不在物理内存中,当 CPU 访问该页面时会产生一个缺页错误(Page Fault)并交由操作系统的缺页错误处理程序处理。通常操作系统会将存放在磁盘上的页面

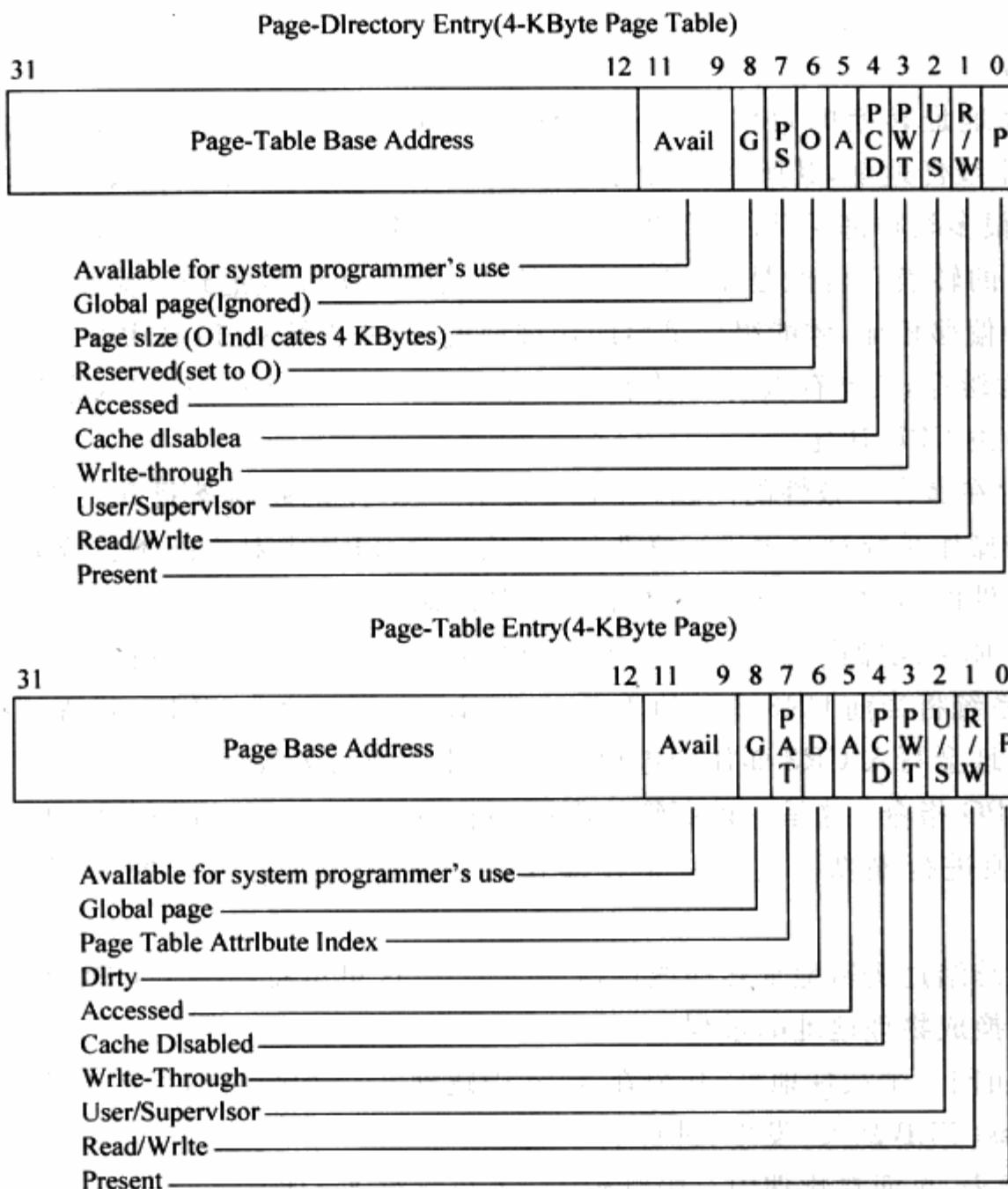


图 2-9 页面目录项和页表项

调入物理内存,使访问可以继续。P=0 时,页面目录项、页表项格式变为图 2-10 中的格式。此时 1~31 位供操作系统使用以记录物理页面在磁盘上的信息,通常是物理页面在磁盘上的位置。

31	Available to Operating System or Executive												0
0												0	

图 2-10 P=0 时的页面目录项、页表项

CPU 在索引页面前,必须知道页面目录所在的物理地址,该物理地址存放在 CR3 (Control Register 3)寄存器中,也称为页面目录基址寄存器(Page-directory base register, PDBR)。一个进程在运行前,必须将其页面目录的基地址存入 CR3。页面目录的基地址必须对

齐到 4K 边界。

3. TLB

为了提高地址转换的效率, x86 架构使用 TLB 对最近用到的页面映射进行缓存, 当 CPU 访问某个线性地址, 其所在页面的映射存在于 TLB 中时, 无须查找页表即可进行地址转换。注意, 很多教材都说 TLB 存放的是线性地址到物理地址的转换, 准确地说应该是: VFN 到 PFN 的转换。也就是说, CPU 从 TLB 获得一个线性地址对应的 PFN 后, 仍然要和线性地址的偏移相加, 才能得到最后的物理地址, 而非直接从 TLB 中获取物理地址。

TLB 作为缓存, 其能存放的映射条目是有限的, 当 TLB 中没有空闲条目可用时, 替换哪一个旧条目由 CPU 决定。

TLB 也存在缓存一致性的问题, 这主要是指 TLB 中的映射条目和页表中的映射条目的一致性。当操作系统对页表进行修改后, 要负责对 TLB 中对应的条目或者整个 TLB 进行刷新。从软件的角度来看, x86 提供了两种方式刷新 TLB:

1) 更新 CR3: 此操作可以导致 TLB 被整体刷新, TLB 中所有映射条目失效(全局 TLB 除外)。操作系统将当前 CR3 中的值重新写回 CR3 以刷新整个 TLB。进程切换时, 新进程的页目录基址会写入 CR3 而使老进程在 TLB 中的条目失效。

2) INVLPG 指令: 这是一种更细粒度的刷新, 操作系统可以用它对 TLB 中单独的页目录项、页表项进行刷新。这通常是在操作系统修改页表后进行的(例如分配/释放了页面)。

前面已经总结过逻辑地址转换线性地址的过程, 这里再总结一下 CPU 使用分页机制, 将线性地址转换成物理地址的过程:

1) CPU 访问一个线性地址, 映射在 TLB 中跳到 6)。如映射不存在于 TLB 中, 我们称一次 TLB Miss(TLB 缺失)发生, 进行下一步。

2) 查找页表, 页面在物理内存中跳到 4), 不再进行下一步。

3) 操作系统的缺页处理函数接管, 通常会进行如下操作: a) 将页面从磁盘复制到物理内存中; b) 更改对应页表项, 设置 P 位为 1, 并对其他字段进行相应设置; c) 刷新 TLB 中对应的页表项; d) 从缺页错误处理函数中返回。

4) 到这一步, 页面已经存在于物理内存中, 并且页表已经包含该映射。此时, 重新执行引发 TLB Miss 的指令。

5) TLB Miss 再次发生, CPU 重新查页表, 把对应的映射插入到 TLB 中。

6) 到这一步, TLB 已经包含了该线性地址对应的 PFN。通过将线性地址中的偏移部分和 PFN 相加, 就得到了对应的物理地址。

本节对 x86 的内存管理机制进行了宏观的介绍, 由于篇幅关系, 未能深入细节以及引入 PAE、Intel x86_64 平台的新技术, 读者朋友如果感兴趣, 请查阅《Intel(R) 64 and IA-32 Architectures Software Developer's Manual Volume 3A》一书。

2.3 x86 架构的基本运行环境

从 CPU 的角度来看,程序不过是一组指令并按编译时生成的顺序执行。执行的过程中会从内存中取值并在寄存器中操作,以得到期望的结果。此外,还有一些特殊的寄存器对 CPU 的状态和行为进行控制。本节介绍一下 x86 架构的基本运行环境。

2.3.1 三种基本模式

实际上,x86 有 4 种运行模式:实模式、保护模式、SMM 模式和虚拟 8086 模式。除 SMM 模式外,其他三种模式常见于各种教科书,对理解 x86 CPU 的工作极为重要,在此也对它们进行简要介绍。

(1) 实模式(Real Mode):当 CPU 加电并经历最初的混沌状态后,首先进入的就是实模式,它是早期 Intel 8086 处理器工作的模式。在该模式下,逻辑地址转换后即为物理地址,CPU 可以访问 1MB 的物理地址空间(实际上是 1MB+64KB)。操作系统或 BIOS 通常在该模式下准备必要的数据结构和初始化关键的寄存器,然后切换入保护模式。

(2) 保护模式(Protect Mode):操作系统运行时最常用的模式。在该模式下,CPU 的所有功能几乎都能得到使用,可以访问架构允许的所有物理地址空间(例如 x86 是 4GB)。本章所有的讲解,如无特殊说明,都是基于保护模式进行的。

(3) 虚拟 8086 模式(Virtual 8086 mode):为了使早期的 8086 程序能在保护模式下运行,x86 提供了虚拟 8086 模式。该模式可以让 CPU 在保护模式下为 8086 程序虚拟实模式的运行环境,使这些程序在执行时无须真正的从保护模式切换到实模式。

2.3.2 基本寄存器组

寄存器是软件操作 CPU 的最基本部件,x86 架构的寄存器可以粗略地分为以下几类。

(1) 通用寄存器:共有 8 个 32 位的通用寄存器,例如常见的 EAX、EDX 等,用来保存程序运行时的临时变量、栈指针等数据。

(2) 内存管理寄存器:包括段寄存器和描述符表寄存器。

(3) EFLAGS 寄存器:32 位的寄存器,用来保存程序运行中的一些标志位信息,如溢出、开启中断与否、分支跳转等信息。

(4) EIP 寄存器:32 位的寄存器,用来保存指向当前指令的地址。通常教科书中称该寄存器为 PC 指针。

(5) 浮点运算寄存器:对于浮点运算,x86 会通过一个浮点运算协处理器来处理。协处理器中包括 8 个 80 位的浮点数据寄存器,1 个 16 位的控制寄存器,1 个 16 位的状态寄存器,1 个 16 位的标志寄存器,1 个 11 位的指令码寄存器,1 个 48 位的浮点指令指针寄存器和 1 个 48 位的浮点数据指针寄存器。这些浮点运算寄存器为浮点运算提供了一个基本的

运行环境。

(6) 控制寄存器：x86 提供了 5 个控制寄存器，分别是 CR0~CR4 寄存器。这些控制寄存器决定了 CPU 运行的模式和特征等。

(7) 其他寄存器：x86 还提供了其他一些寄存器，包括 8 个调试寄存器(DR0~DR7)、内存区域类型寄存器(MTRR)、机器检查寄存器(Machine Check Registers)以及性能监控寄存器。

2.3.3 权限控制

权限控制是指 CPU 对资源进行分类，使不同权限的程序只能访问自身权限所允许访问的资源。操作系统的用户态和内核态之分就是最常见的权限控制，内核态程序具有最高权限，用户态程序具有最低权限。x86 架构提供两种权限控制机制——段保护和页保护。如它们名字所暗示，这两种机制对应内存管理中的段机制和分页机制，下面分别进行介绍。

1. 段保护

段保护引入了如下三种属性对权限控制进行控制。

(1) 当前权限级别(Current Privilege Level,CPL)：CPL 表示当前运行的代码的权限。通过 CS 的 0、1 位记录代码的 CPL 值，CPL 可以有 0~3 共 4 个级别，这就是常说的 Ring 级别(实际上，Ring 级别有更广阔的含义)。其中，Ring0 对应 CPL=0，具有最高权限，操作系统的内核运行在该权限；Ring3 对应 CPL=3，用户程序运行在 Ring3。CPL 值越高权限越低。

(2) 描述符权限级别(Descriptor Privilege Level,DPL)：DPL 表示段和门(Gate,参考 2.4 节)所具有的权限。它表示代码访问某个段或通过某个门时所需要的最低权限。例如，某个数据段描述符有 DPL=2，则只有 CPL=0、1、2 的代码可以访问该数据段，CPL=3 的不能访问。

(3) 所要求权限级别(Requested Privilege Level,RPL)：RPL 比较特殊，它存在于段寄存器的 0~1 位(注意，CS 寄存器的 0~1 位是 CPL)，用于程序在访问段时增加一级检查。其用途见后面的例子。

前面介绍了，程序访问一个段，要通过段寄存器得到段描述符，这样会产生 2 次检查，参与检查的 3 个属性分别是：程序本身的 CPL、段寄存器的 RPL、段描述符的 DPL。CPL、DPL、RPL 组合起来的情况有很多种，但只有当 $CPL \leq DPL$ 且 $RPL \leq DPL$ 时，访问才被允许，其余情况均被拒绝。通常，可以把 RPL 设置成 0 来简化检查，此时，满足 $CPL \leq DPL$ 访问即被允许。

2. 页保护

页保护的思想比段保护简单，它通过在页目录项、页表项中引入一个 User/Supervisor 位，将页面(或整个页目录项)分成 User 和 Supervisor 两个特权级。该位为 0 时表示 Supervisor 模式，对应 CPL=0、1、2 的情况；为 1 表示 User 模式，对应 CPL=3 的情况。

当程序运行在 CPL=0、1、2，也就是 Supervisor 模式下时，可以访问所有页面；运行在 CPL=3 下的程序处于 User 模式，只能访问 User 页面。

段保护和页保护是可以混用的，从而带来了更为灵活的保护机制。

在本节中，对 x86 架构的基本运行环境和程序运行所必须理解的概念进行了介绍。其中权限控制的部分属于极为复杂的内容，我们只进行了宏观上的概括，有兴趣的读者可以参考“Intel(R) 64 and IA-32 Architectures Software Developer's Manual Volume 3A”了解详细情况。

2.4 中断与异常

如果程序总是顺序执行，那么事情将变得非常简单。但事情往往和人们所期望的不太一样，中断和异常会打断顺序执行的程序流，转而进入一条完全不同的执行路径。操作系统的内核为什么那么难懂，很大一部分要归功于它们。本节将介绍现代 CPU 架构中的中断和异常机制，让事情变得简单一点。

2.4.1 中断架构

从某种意义上来说，现代计算机架构是由大量的中断事件驱动的。中断提供给外部硬件设备一种“打断 CPU 当前执行任务，并响应自身服务”的手段。

1. 可编程中断控制器

中断从设备发送到 CPU 需要由被称为“中断控制器”的部件转发(MSI 除外)。中断控制器发展至今，经历了 PIC(Programmable Interrupt Controller，可编程中断控制器)和 APIC(Advanced Programmable Interrupt Controller，高级可编程中断控制器)两个阶段。

2. PIC

8259A 芯片即常说的 PIC，它具有 IR0~IR7 共 8 个中断管脚连接外部设备。中断管脚具有优先级，其中 IR0 优先级最高，IR7 最低。PIC 有如下三个重要的寄存器。

(1) IRR(Interrupt Request Register，中断请求寄存器)：共 8 位，对应 IR0~IR7 这 8 个中断管脚。某位置 1 代表收到对应管脚的中断但还未提交给 CPU。

(2) ISR(In Service Register，服务中寄存器)：共 8 位。某位置 1 代表对应管脚的中断已经提交给 CPU 处理，但 CPU 还未处理完。

(3) IMR(Interrupt Mask Register，中断屏蔽寄存器)：共 8 位。某位置 1 对应的中断管脚被屏蔽。

除此之外，PIC 还有一个 EOI 位，当 CPU 处理完一个中断时，通过写该位告知 PIC 中断处理完成。PIC 向 CPU 递交中断的流程如下。

(1) 一个或多个 IR 管脚上产生电平信号，若对应的中断没有被屏蔽，IRR 中相应的位被置 1。

- (2) PIC 拉高 INT 管脚通知 CPU 中断发生。
- (3) CPU 通过 INTA 管脚应答 PIC, 表示中断请求收到。
- (4) PIC 收到 INTA 应答后, 将 IRR 中具有最高优先级的位清 0, 并设置 ISR 中对应的位。
- (5) CPU 通过 INTA 管脚第二次发出脉冲, PIC 收到后计算最高优先级中断的 vector, 并将它提交到数据线上。
- (6) 等待 CPU 写 EOI。收到 EOI 后, ISR 中最高优先级的位被清 0。如果 PIC 处于 AEOI 模式, 当第二个 INTA 脉冲收到后, ISR 中最高优先级的位自动清 0。

3. APIC

PIC 可以在 UP(单处理器)平台上工作, 但无法用于 MP(多处理器)平台。为此, APIC 应运而生。APIC 由位于 CPU 中的本地高级可编程中断控制器 (Local Advanced Programmable Interrupt Controller, LAPIC) 和位于主板南桥中 I/O 高级可编程中断控制器 (I/O Advanced Programmable Interrupt Controller, IOAPIC) 两部分构成。它们的关系如图 2-11 所示。

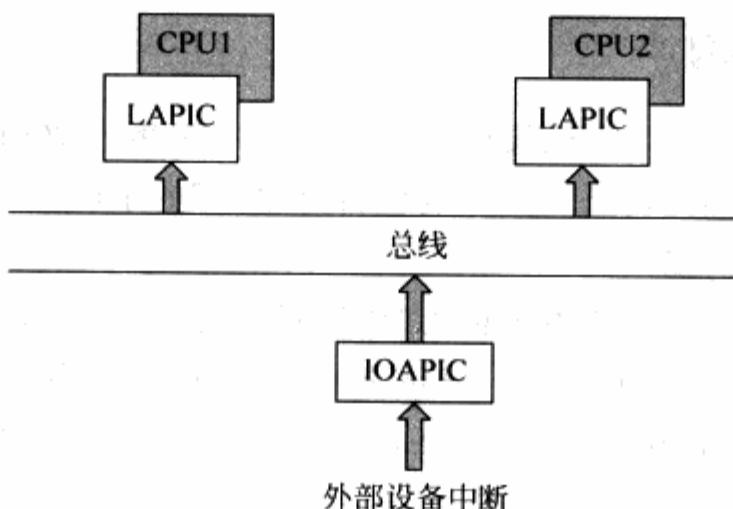


图 2-11 APIC 系统架构

其中, IOAPIC 通常有 24 个不具有优先级的管脚, 用于连接外部设备。当收到某个管脚的中断信号后, IOAPIC 根据软件 (通常是操作系统) 设定的 PRT (Programmable Redirection Table) 表, 查找到管脚对应的 RTE (Redirection Table Entry, PRT 的表项)。通过 RTE 的各个字段, 格式化出一条包含该中断所有信息的中断消息, 再由系统总线 (老式的通过专门的 APIC 总线) 发送给特定 CPU 的 LAPIC, LAPIC 收到该消息后择机将中断递交给 CPU 处理。

在 LAPIC 内部, 也有 IRR、ISR 和 EOI 寄存器, 其中 IRR、ISR 为 256 位, EOI 为 32 位, 它们的功能和 PIC 的大体类似。APIC 系统中, 中断的发起大致流程如下。

- (1) IOAPIC 收到某个管脚产生的中断信号。
- (2) 查找 PRT 表获得该管脚对应的 RTE。根据 RTE 各字段格式化出一条中断消息,

并确定发送给哪个(或多个)CPU 的 LAPIC。

- (3) 通过系统总线或 APIC 总线发送中断消息。
- (4) LAPIC 收到中断消息, 判断是否由自己接收。
- (5) 如确定接收, 将 IRR 中对应的位置置 1。同时确定此时是否将该中断交由 CPU 处理。
- (6) 如确定提交中断给 CPU 处理, 从 IRR 获取最高优先级的中断, 将 ISR 中对应的位置置 1, 并提交中断。对于 edge 触发中断, IRR 中对应位此时清 0。
- (7) CPU 处理完中断, 软件写 EOI 寄存器告知中断处理完成, 对于 level 触发中断, IRR 中对应位此时清 0。LAPIC 可提交下一个中断。

4. 处理器间中断

在 MP(多处理器)平台上, 多个 CPU 要协同工作, 处理器间中断(Inter-processor Interrupt, IPI)提供 CPU 之间相互通信的手段。CPU 可以通过 LAPIC 的 ICR(Interrupt Command Register, 中断命令寄存器)向指定的一个/多个 CPU 发送中断。

操作系统通常使用 IPI 来完成诸如进程转移、中断平衡和 TLB 刷新等工作。

5. 中断的重要概念

1) 中断的分类

中断可以从多个方面进行分类。从中断源的角度来看, 可以分为如下几类。

- ① 外部中断: 指连接在 IOAPIC 上设备产生的中断、LAPIC 上连接的设备或 LAPIC 内部中断源产生的中断以及处理器间中断。
- ② 可屏蔽中断: 指可以通过某种方式(例如 CLI 指令、TPR)进行屏蔽的中断。与之对应的概念是不可屏蔽中断(non-maskable interrupt)。
- ③ 软件产生中断: 指通过 INT n 指令产生的中断。

这样的分类并非绝对, 例如外部中断通常是可屏蔽中断, 但也可能属于不可屏蔽中断。通常, 根据外部中断的触发方式, 又把它们分为如下几类。

- ① edge 触发中断: 指中断边沿方式触发(例如上升沿)。ISA 设备、时钟设备多使用这种触发方式。
- ② level 触发中断: 指中断以电平方式触发, 在中断程序应答设备前, 该电平一直有效。PCI 设备使用这种触发方式。

2) 中断的优先级

在使用 PIC 的系统中, PIC 的管脚决定了中断的优先级, 连接 IR0 的设备具有最高优先级, 连接 IR7 的设备优先级最低。在 APIC 系统中, IOAPIC 的管脚不再具备优先级, 设备的中断优先级由它所连接管脚对应 RTE 中的 vector 字段决定。vector 是 x86 架构用于索引 IDT 表(在后面介绍)的下标, 范围从 0~255, 值越大优先级越高。其中, 32~255 可以供外部中断使用。

在现代操作系统中, 有几个概念和 vector 常联系在一起使用, 这里简单介绍一下。

- ① IRQ: PIC 时代的产物, 由于 ISA 设备通常是连接到固定的 PIC 管脚, 所以说一个设

备的 IRQ 实际是指它连接的 PIC 管脚号。IRQ 暗示着中断优先级,例如 IRQ0 比 IRQ3 有着更高的优先级。当前进到 APIC 时代后,或许是出于习惯,人们仍习惯用 IRQ 表示一个设备的中断号,但对于 16 以下的 IRQ,它们可能不再与 IOAPIC 的管脚对应。

② GSI(Global System Interrupt): ACPI 引入的概念,它为系统中每个中断源指定一个唯一的中断号。IRQ 和 GSI 在 APIC 系统中常常被混用,实际上对于 15 以上的 IRQ,它和 GSI 相等。

在这里,GSI 和 IRQ 可以看作等同的概念,表示某个设备的中断号。它们与 vector 的关系由操作系统决定,通常是在设备驱动注册中断处理程序时由操作系统分配。

3) 中断的屏蔽

无论是在 PIC 收到中断信号后,还是 LAPIC 收到中断消息后,并不一定都是马上交给 CPU 处理的,这还要取决于 CPU 当前是否屏蔽中断(不可屏蔽中断除外)。当 CPU 屏蔽中断时,中断会被依附(pending)在 PIC/LAPIC 的 IRR 寄存器中,一旦 CPU 开启中断,会在第一时间响应 PIC/LAPIC 所依附的中断。CPU 可以通过以下几种方法屏蔽/开启中断。

① CLI/STI 指令:这是操作系统最常用的屏蔽/开启中断的方法。CLI 指令将本 CPU 的 EFLAGS 寄存器的 IF 位清 0,阻止接收中断;STI 指令将 IF 位置一,允许接收中断。这两条指令都只对当前 CPU 起作用,而不影响平台上的其他 CPU。

② TPR 寄存器:根据该寄存器值代表的优先级,部分屏蔽外部中断。

③ PIC/IOAPIC 的中断屏蔽位:PIC 可以通过 IMR 寄存器屏蔽对应管脚。IOAPIC 可通过 RTE 中的 mask 位屏蔽对应管脚。该方法不会将中断依附(pending)到 IRR,而是直接忽略,对于 edge 触发中断可能导致中断丢失。

4) IDT 表

IDT 表实际就是个大数组,用于存放各种“门”(中断门、陷阱门、任务门),这些“门”是中断和异常通往各自处理函数的入口。当一个中断或异常发生时,CPU 用它们对应的 vector 号索引 IDT 表以获得对应的“门”。每个“门”占 8B,x86 最多有 256 个 vector,故 IDT 表长度最大为 $8 \times 256 = 2048B$ 。

IDT 表的基地址存放在 IDTR 寄存器中,该寄存器和 GDTR 类似,由一个地址(Base)字段和长度(Limit)字段构成,通过 LIDT/SIDT 指令可以加载和存储 IDTR 寄存器。IDT 表要求被对齐到 8B 边界以提高效率。

5) 中断门

“门”是入口,中断门就是中断的入口。中断门实际上是一种段描述符,称为系统描述符(System Descriptor),由段描述符的 S 位控制。中断门的格式如图 2-12 所示。

其中,段选择符、偏移量字段可以看成一个逻辑地址,通过索引 GDT 将该逻辑地址转换成中断处理函数入口的线性地址。这里特别要注意的是 DPL 字段,很多操作系统把门的 DPL 设置成 0,而在 2.3.3 节讲过,只有当 $CPL \leq DPL$ 、 $CPL \leq RPL$ 访问才被允许。这就引出一个问题:程序在用户态时($CPL=3$)发生中断,岂不是不能过一个 $DPL=0$ 的中

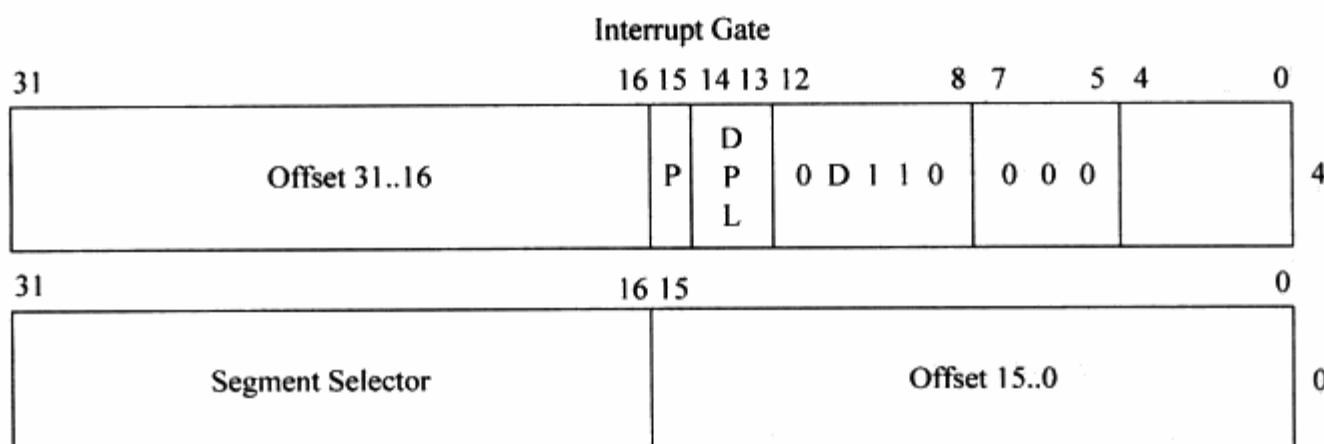


图 2-12 中断门

断门？实际上，中断门和陷阱门的 DPL 只在使用 INT n 指令引起中断/异常时才检查，硬件产生的中断/异常不检查。P 字段代表该中断门是否有效，清 0 无效。

虽然没有强行规定中断一定要使用中断门，但通常操作系统是这样做的。中断门和陷阱门（后面介绍）的唯一区别是程序通过中断门跳转后，EFLAGS 寄存器的 IF 位自动清 0，中断关闭。而陷阱门没有这样的效果。

2.4.2 异常架构

和中断相比，异常最大的不同在于它是在程序的执行过程中同步发生的。例如下面这个程序：

```
void main()
{
    int a = 10;
    a = a / 0;
}
```

程序运行到 $a=a/0$ 一句时必然引起一个除 0 异常，但不能预料该程序在执行时是否会中断。异常根据产生的原因和严重程度可以分为如下三类。

(1) 错误(Fault)：由某种错误情况引起，一般可以被错误处理程序纠正。错误发生时，处理器将控制权转移给对应的处理程序。例如，常见的缺页错误就属于此类。

(2) 陷阱(Trap)：指在执行了一条特殊指令后引起的异常。例如，Linux 中用于实现系统调用的 INT 80 指令就属于此类。

(3) 终止(Abort)：指严重的不可恢复的错误，将导致程序终止的异常。例如 MCA (Machine Check Architecture)。

和中断门一样，陷阱门存放在 IDT 表中。异常发生后，CPU 用该异常的 vector 号索引其对应的陷阱门。x86 架构将 vector 0 ~ 19 预留给各个异常。陷阱门的格式如图 2-13 所示。

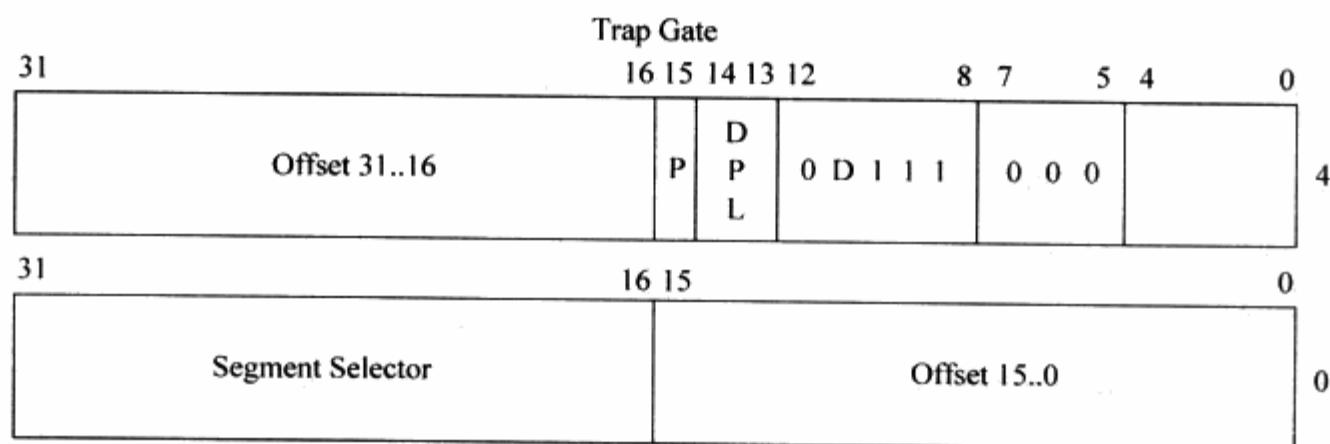


图 2-13 陷阱门

其各字段含义和中断门一样,在此不多做解释。

2.4.3 操作系统对中断/异常的处理流程

虽然各个操作系统对于中断/异常处理实现不同,但基本流程遵循如下的顺序。

一个中断/异常发生,打断当前正在执行的任务。

(1) CPU 通过 vector 索引 IDT 表得到对应的“门”,并获得其处理函数的人口地址。

(2) 程序跳转到处理函数执行,由于处理函数存放在 CPL=0 的代码段,程序可能会发生权限提升。处理函数通常执行下列几个步骤。

① 保存被打断任务的上下文(上下文的概念在第 3 章介绍),并开始执行处理函数。

② 如果是中断,处理完成后需要写 EOI 寄存器(伪中断不需要)应答,异常不需要。

③ 恢复被打断的任务的上下文,准备返回。

(3) 从中断/异常的处理函数返回,恢复被打断的任务,使其继续执行。

操作系统可能对上述执行路径进行封装,也会引入诸如软中断之类的机制,但总顺序如上所述,不会有太大的不同。

本节中对现代计算机的中断/异常架构做了宏观的介绍,需要说明的是,中断/异常属于处理器架构中比较复杂的部分,上面的内容仍然有很多细节没有涉及到。目前,新的中断方式——MSI(Message Signalled Interrupts)已经被广泛应用,限于篇幅的关系,我们没有介绍,感兴趣的朋友可以查阅后面的参考文献。

2.5 进程

进程有时被称为任务,有时又被称为程序,说法不一。在这里引用一段从 Windows 内核书籍摘抄过来的文字,这是迄今为止最准确的进程定义。

“尽管表面上看起来程序和进程非常类似,但本质上它们却是截然不同的。程序是指一个静态的指令序列,而进程则是一个容器,其中包含了当执行一个程序的特定实例时所用到

的各种资源。”——摘自“Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server 2003, Windows XP, and Windows 2000”一书。

进程就是各种资源的合集,通常一个进程包含下面几种资源。

(1) 私有的线性地址空间:这是进程可以使用的线性地址的总和,其中内核部分可能和其他进程是共享的。

(2) 可执行的程序:也就是前面说的二进制序列,包含代码和数据。

(3) 一些已经获得的其他资源,如打开的文件、管道等。

(4) 进程的权限:指进程的运行权限,例如在 Linux 中就有 root 用户和非 root 用户之分。

(5) 进程的描述符:有的操作系统称为控制块,包含操作该进程的一些必要的信息,例如进程的 ID 号。

进程是内核调度器调度的单元,进程的运行构成了操作系统工作的大部分内容,其机制较为复杂。限于篇幅,这里只介绍和虚拟技术关系最紧密的概念——上下文。

2.5.1 上下文

之所以标题不是“进程的上下文”,是因为还有一个“中断的上下文”存在。上下文也是个难以定义的概念,它实际上是从 CPU 的角度引出的。简单地说,上下文就是程序(进程/中断)运行时所需要的寄存器的最小集合。这些寄存器的后面可能代表着程序运行的一类资源,例如 CR3 寄存器就可以概括进程的私有线性地址空间(分页机制启动时)。

下面来看看 x86 架构下上下文包含哪些寄存器。

(1) 通用寄存器组:即 EAX、EBX、ECX、EDX、ESI、EDI 这 6 个,加上 ESP(栈指针)、EBP(框架指针)。

(2) 段相关寄存器组:CS、DS、SS,如果程序使用了 ES 等额外段寄存器,也要包括进来。

(3) 标志寄存器:主要是指 EFLAGS 寄存器。

(4) 程序指针寄存器:EIP。

(5) GDT 基地址:用于访问 GDT,GDTR 中的内容。

(6) LDT 段选择符:如果程序使用了私有的 LDT,LDTR 的内容。

(7) IDT 基地址:用于访问 IDT 表,IDTR 的内容。

(8) 控制寄存器组:CR 系列,表示当前程序运行时的 CPU 控制状态。

(9) 浮点相关寄存器组:用于浮点计算的一些寄存器组。

(10) 一些特殊用途的寄存器:例如 x86 架构下的 MSR(Model-Specific Registers)。

一个程序的上下文可能是上面列出内容的一个子集(例如进程),也可能是全部(例如虚拟机)。从程序员的观点来看,上下文的概念有些微改变,通常对于上下文切换时不需要改变的寄存器,也可以说它不是该程序的上下文。例如进程切换时,GDTR 中的内容不需要

改变,为了方便,通常在讲一个进程的上下文时不把 GDTR 算进去。在后面的内容中,包括虚拟机部分,提到上下文都是指在上下文切换时必须更改的寄存器的集合。

2.5.2 上下文切换

上下文切换是指程序从一种状态切换到另一种状态(例如用户态切换到内核态),或从一个程序切换到另一个程序(例如进程切换)时,导致上下文相关寄存器值的变化行为。这种变化是指旧程序(切换前的程序)上下文相关寄存器的值被保存到内存中,新程序(切换后的程序)上下文相关寄存器的值被加载到寄存器中。在操作系统中,通常只有三种情况会发生上下文切换。

(1) 用户态到内核态的切换:此时的上下文切换是因为进程的用户态和内核态运行在不同的 Ring 级别,对资源的访问权限不同,需要切换部分上下文。例如,从用户态的栈切换到内核态的栈。

(2) 进程切换:由于一个 CPU 在同一时刻只能有一个进程运行,所以在新的进程运行前,需要把上下文相关寄存器的值换成新进程的相关值,例如把 CR3 换成新进程页目录的地址,EIP 指向新进程运行的第一条指令。这通常是个全上下文的切换。

(3) 到中断上下文的切换:中断的处理函数运行在特殊的上下文环境,称为中断上下文。CPU 处理一个中断时,不管当前 CPU 在运行一个进程,还是本身就在一个中断上下文中,都要切换到新中断的上下文。例如,更改栈指针、EIP 变化等。这通常是部分上下文的切换,例如 CR3 寄存器的值就不需要更改。根据 x86 架构的特点,无论是 Linux 的硬中断机制,还是 Windows、Solaris 的中断线程化机制,处理中断必然经过一个中断上下文阶段,可能的情况如下。

① 进程上下文 → 中断上下文(处理中断) → 进程上下文(中断返回执行)。

② 进程上下文 → 中断上下文 → 新进程上下文(处理中断) → 进程上下文(最先被打断进程的上下文)。

③ 中断上下文 → 新中断上下文 →

x86 只有一种机制,即任务门(Task Gate)可以使中断的处理不经过中断上下文而直接进入进程上下文,但几乎没有操作系统使用它,读者可以忽略。

上下文切换通常有两个步骤。

(1) 保存旧上下文:将被切换出去的程序(如一个被新进程替代的旧进程)或将被切换出去的状态(如程序的用户态)的上下文相关寄存器的值保存在内存中。

(2) 加载新上下文:将要运行的程序(如新进程)或新状态(如程序的内核态)运行需要的上下文相关寄存器的值从内存中读入,加载入对应寄存器中。

需要注意的是,保存旧的上下文动作在 x86 架构下有时会由 CPU 自动完成一部分(例如中断发生时、使用 TSS),但在现代操作系统中,通常是由软件完成。

在本节中,对进程的基本概念,以及和后面虚拟技术联系紧密的上下文概念进行了介

绍。读者通过本节应该建立起强烈的上下文印象,上下文切换技术在虚拟技术中是非常重要并且频繁用到的。

2.6 I/O 架构

将计算机进行的任务进行一个粗略的分类,其实只有两种:CPU 运算和 I/O 操作。I/O 架构毫无疑问是现代计算机体系的重要组成部分,本节对 x86 的 I/O 架构进行介绍,为后面 I/O 虚拟化章节提供准备知识。

2.6.1 x86 的 I/O 架构

I/O(输入输出)是 CPU 访问外部设备的方法。设备通常通过寄存器和设备 RAM 将自身的功能展现给 CPU,CPU 读写这些寄存器和 RAM 即可以完成对设备的访问和操作。通过访问方式的不同,可以将 x86 架构的 I/O 分为如下两类。

(1) Port I/O(端口 I/O): 即通过 I/O 端口访问设备寄存器。x86 有 65 536 个 8 位的 I/O 端口,编号为 0x0~0xFFFF。如果把端口号看作访问设备端口的地址,那么这 65 536 个端口就构成了 64KB 的地址空间,称为 I/O 端口地址空间。与前面介绍的线性地址空间和物理地址空间不同,I/O 端口地址空间是独立的,也就是说它并不是线性地址空间或物理地址空间的一部分。使用 IN/OUT 指令访问端口时,CPU 通过一个特殊的管脚标识这是一次 I/O 端口访问,于是芯片组知道地址线上的地址是 I/O 端口号并进行相应操作。此外,2 个或 4 个连续的 8 位 I/O 端口,可以组成 16 位或 32 位的 I/O 端口。

(2) MMIO(Memory Map I/O,内存映射 I/O): 即通过内存访问的形式访问设备寄存器或设备 RAM。x86 架构下,MMIO 和 Port I/O 最大的不同是 MMIO 要占用 CPU 的物理地址空间。它把设备的寄存器或设备 RAM 映射到物理地址空间某段地址,使用 MOV 这样的访存指令访问此段地址即可访问到映射的设备。很多 CPU 架构都没有 Port I/O,采用统一的 MMIO 方式。由此可见,MMIO 是一种更加先进的 I/O 访问方式。

对于 Port I/O,由于编译器不能产生 IN/OUT 指令,操作系统通常把汇编指令封装成类似 inb()、outb()这样的函数。对于 MMIO,由于整个物理地址空间都会被映射到线性地址空间,程序访问 I/O 资源时也要做线性地址到物理地址的转换。与普通物理地址到线性地址的映射不同的是,MMIO 地址通常是不可缓存的(un-cacheable)。

2.6.2 DMA

DMA(直接内存访问)是将 CPU 从 I/O 操作中解放出来的一种技术。如果设备向内存复制数据都经过 CPU,则会消耗大量的 CPU 时间,不利于系统性能。通过 DMA,驱动程序可以事先(或在需要的时候)设定一个内存地址,设备就可以绕开 CPU 直接向内存中复制(或读取)数据。根据发起者不同,DMA 可以被分为两种。

(1) 同步 DMA：是指 DMA 操作由软件发起。一般的流程是设备驱动在设定好需要被 DMA 访问的内存地址后，写某个寄存器来通知设备发起 DMA。此时，设备会直接从该内存地址读取内容并操作。一个典型的例子是声卡，当播放一段音频时，驱动将该音频存放的地址通知声卡，设备从内存直接读取数据并播放，完成后以一个中断通知驱动操作完成。

(2) 异步 DMA：是指 DMA 操作由设备发起。一般的流程是设备将数据直接复制到一个事先设定好的内存地址，再通过一个中断通知驱动程序。典型的例子是网卡收包，当网卡接收到数据包后，会直接复制到驱动程序设定的内存地址去，并以中断的形式通知网络包的到来。

设备的 DMA 操作都是使用物理地址访问内存，不经过线性地址到物理地址的转换。但 IOMMU 出现后，这个情况就被改变了，这在后面 VT-d 技术相关章节介绍。从驱动的角度来看，它要提供一片内存区域供设备访问，DMA 要求这片内存区域在物理上是连续的。现代设备支持一种称为“分散—聚合(Scatter-gather)”DMA 的机制，允许驱动向设备提供不连续的物理内存。实际上，驱动是将一组以“起始地址—长度”为属性的内存描述符提供给设备，每个描述符描述了一块连续的物理内存，但连续两个描述符描述的内存不需要是连续的。从宏观上来看，通过这组内存描述符可以向设备提供一片不连续的内存区域；但从微观的角度看，DMA 操作访问的仍然是连续的物理内存。

2.6.3 PCI 设备

PCI 总线无疑是总线中的王者。在它之前，各种平台都拥有自己特定的总线，例如 x86 的 ISA 总线、Power PC 的 VME 总线。PCI 出现后，由于速度快、具有动态配置功能和独立于 CPU 架构等特点，迅速被各种平台接受，成为一种通用的总线架构。

1. PCI 总线架构

PCI 总线是一种典型的树结构。把北桥中 HOST-PCI 桥看作根，总线中其他 PCI-PCI 桥、PCI-ISA 桥(ISA 总线转 PCI 总线桥)等桥设备和直接接 PCI 总线的设备看作节点，整个 PCI 架构可以概括成图 2-14 所示。

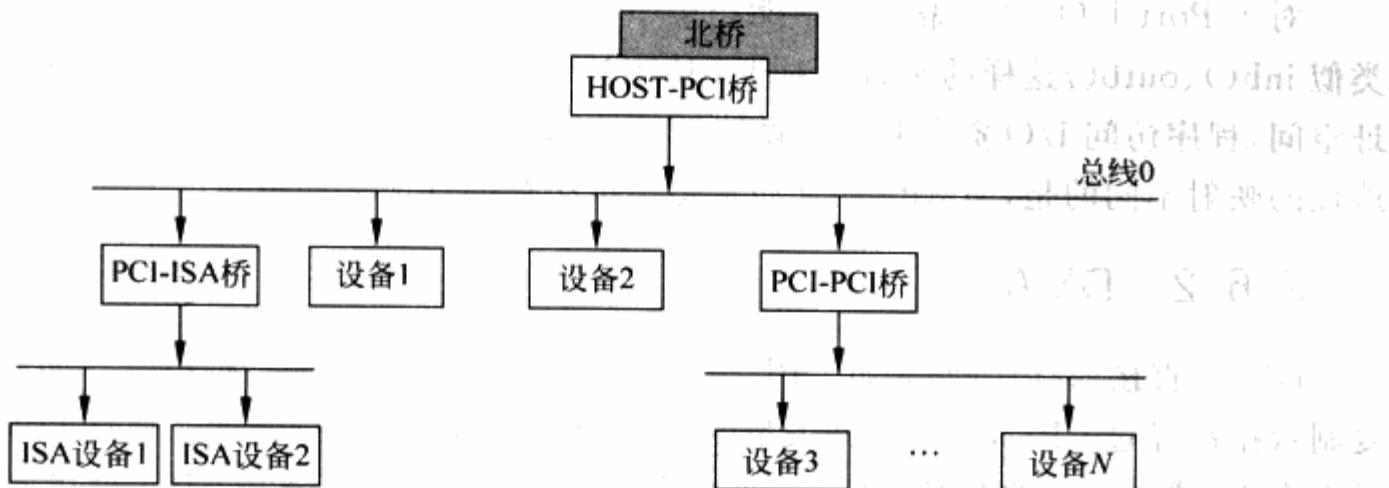


图 2-14 PCI 总线结构

通过桥,PCI 总线可以很容易地被扩展,并且与其他总线相互挂接,构成整个系统的总线网络。与 HOST-PCI 相连的总线被称为总线 0,其他层次总线的编号,是在 BIOS(或操作系统)枚举设备时确定的,这在后面的章节会介绍。

2. 设备标识符

设备标识符可以看作是设备在 PCI 总线上的地址,它的格式如图 2-15 所示。

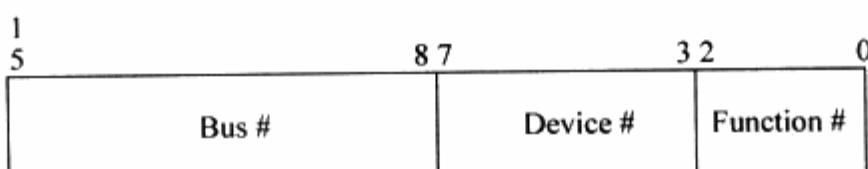


图 2-15 设备标识符

其中,8 位的 Bus 字段代表设备所在的总线号,故系统中最多有 256 条总线。Device 字段表示设备号,代表在 Bus 所表示总线上的某个设备。Function 字段表示功能号,标识具体设备上的某个功能单元。“功能单元”这样的说法很拗口,实际上可以称为逻辑设备。举一个简单的例子,一块 PCI 卡,它上面有两个独立的设备,这两个设备共享了一些电子线路,那么这两个设备就是这块 PCI 卡的两个功能单元。但从软件的角度来看,它们和两个独立接入 PCI 总线的设备无异。如同 Function 字段长度所暗示的,一个独立的 PCI 设备上最多有 8 个功能单元。Device 和 Function 两个字段一般合起来使用,表示一条总线上最多有 256 个设备。通常,用设备标识符三个字段的缩写 BDF 来代表它。

当程序通过 BDF 访问某个设备时,先通过 Bus 字段选定特定的总线,再根据 Device 字段选定特定的设备,最后通过 Function 字段就可以选定特定的功能单元(逻辑设备)了。

3. PCI 配置空间

对于程序员来说,不需要了解 PCI 设备电路上实现的细节,只需要了解操作它的接口即可。PCI 配置空间正是这么一个接口,其结构如图 2-16 所示。

PCI 设备规范规定,设备的配置空间最多为 256 个字节,其中前 64 个字节的格式和用途是统一的,如图 2-16 所示。各个字段的具体含义请参见“PCI Local Bus Specification Revision 3.0”的第 6 章,这里只关心对于程序员最重要的 Base Address Registers 和 Interrupt Pin、Interrupt Line。

(1) Base Address Registers: 基地址寄存器,也就是常说的 PCI Bar。它报告设备寄存器或设备 RAM 在 I/O 端口地址空间(或物理地址空间中)的地址。地址是由软件(BIOS 或操作系统)动态配置的,这就一改 ISA 设备通过跳线进行配置的不灵活的特点。通常枚举 PCI 设备的软件(BIOS 或操作系统)会在获得平台所有 PCI 设备后,根据设备数量,依照固定的算法为每个设备的 PCI Bar 分配 I/O 端口(或物理地址)。设备的电子线路负责把这些端口(或地址)映射到自身的寄存器(设备 RAM)上,这样,CPU 就可以通过端口号(Port I/O 方式)、物理地址(MMIO 方式)访问到设备了。使用哪种方式访问,由 PCI Bar 的最后一个位表示。当该位为 1 时,表示是 I/O 端口;该位为 0 时,表示是 MMIO 端口。某些架构

31	16 15	0	
Device ID	Vendor ID	00h	
Status	Command	04h	
Class Code		08h	
BIST	Header Type	Latency Timer	Cacheline Size
Base Address Registers			10h
			14h
			18h
			1Ch
			20h
			24h
Cardbus CIS Pointer			28h
Subsystem ID	Subsystem Vendor ID	2Ch	
Expansion ROM Base Address			30h
Reserved		34h	
Capabilities Pointer		38h	
Reserved			3Ch
Max_Lat	Min_Gnt	Interrupt Pin	Interrupt Line

图 2-16 PCI 配置空间

根本就没有 Port I/O 方式,全部采用 MMIO,根据访问目标性质不同,PCI Bar 又可以划分为如下两种类型。

① 可预取(Prefetchable)类型: 这主要是设备 RAM。由于 RAM 具有在每次读操作后内容不自动改变的性质,所以可以使用预读机制。例如,程序在读第 N 个字节的内容时,总线可能已经读出了第 $N+1$ 个字节的内容。当预读出的内容不需要时,只要简单地抛弃就行,不会有什么影响。

② 不可预取类型(Un-Prefetchable): 这主要指设备寄存器。寄存器和 RAM 有着不同的性质,有些寄存器本身就是设备的 FIFO 队列的接口。很可能当一次读操作完成后,寄存器的值就改变了。如果使用预读机制,例如程序本身只读了寄存器的第一个字节,而总线却连续读入了 4 个字节,那么后面 3 个字节的内容可能就会改变,下次程序真正访问它们时得到的就是错误的值。对于 PCI Bar 是否为可预取类型,可以根据该 PCI Bar 的第 3 个位判断,1 为可预取,否则为不可预取。

(2) Interrupt Pin: 中断针脚。PCI 中断线的标准设计是 4 条: INTA、INTB、INTC 和 INTD,分别对应值 0~3。该寄存器的值表示设备连接的是哪个中断针脚。具体信息请参考“PCI Local Bus Specification Revision 3.0”的 2.2.6 节。

(3) Interrupt Line: 设备的中断线。该寄存器只起一个保存作用, BIOS 和操作系统可以自由使用它。BIOS 通常用它来保存设备所连的 PIC/IOAPIC 的管脚号。

x86 架构把 I/O 端口地址空间中的 0xCF8~0xCFF 段预留给了 PCI 总线, 用于访问设备的配置空间。其中, 前 32 位的寄存器为“地址寄存器”, 后 32 位为“值寄存器”。软件通过把设备的 BDF 和要访问的配置空间的字节偏移写入“地址寄存器”中, 就可以通过“值寄存器”读写该配置空间了。

4. PCI 设备枚举过程

PCI 设备的枚举和资源分配(即配置 PCI 配置空间)通常是由 BIOS 完成的, 并提供特殊的 PCI 设备枚举接口供保护模式下的操作系统使用, 这些接口称为 PCIBIOS。由于某些平台, 例如嵌入式, 是没有 BIOS 的, 并且操作系统厂商对 BIOS 的可靠性也不信任, 故某些操作系统也实现了自己的 PCI 设备枚举接口。无论是 BIOS, 还是操作系统, 其枚举设备的过程都遵循着一般规律。

从前面的 PCI 总线概要图知道, PCI 设备和总线一起构成了树结构, 其中 PCI-PCI 桥(或 PCI-ISA 等其他桥, 这里只关心 PCI-PCI 桥)是子树的根节点, 设备枚举的过程就是要在内存中建立一棵和实际总线情况相符合的设备树。枚举过程中最关键的步骤是发现 PCI-PCI 桥, 这个可以通过 PCI 配置空间的 Header Type 字段判断, 该字段为 1 时表示为桥设备。PCI-PCI 桥主要有三个属性。

- Primary Bus: 表示该桥所属的根总线。
- Secondary Bus: 表示以该桥为根节点的子总线。
- Subordinate Bus: 表示该桥为根的子树中, 最大的总线号。

下面利用图 2-17 说明三者的关系。

如图 2-17 所示, 对于“PCI-PCI 桥 1”, 其 Primary Bus 是总线 0, Secondary Bus 是总线 1, 而以它为根的总线中最大的总线号为 2, 所以其 Subordinate Bus 为总线 2。

设备枚举从根节点 HOST-PCI 桥开始, 首先探测总线 0 上的各个设备。当探测到第一个桥设备时, 为其分配 Primary Bus 号和 Secondary Bus 号, 其中 Secondary Bus 号为 1(即当前系统中最大总线号加 1), Subordinate Bus 号暂设为和 Secondary Bus 相同, 当在子树中发现新总线后会动态调整该值。接着以该桥为根节点, 继续探测其下属总线, 其过程和前面的相同, 发现第一个桥设备后则以其为根往下探测, 如此反复直到所有的子树都探测完毕。

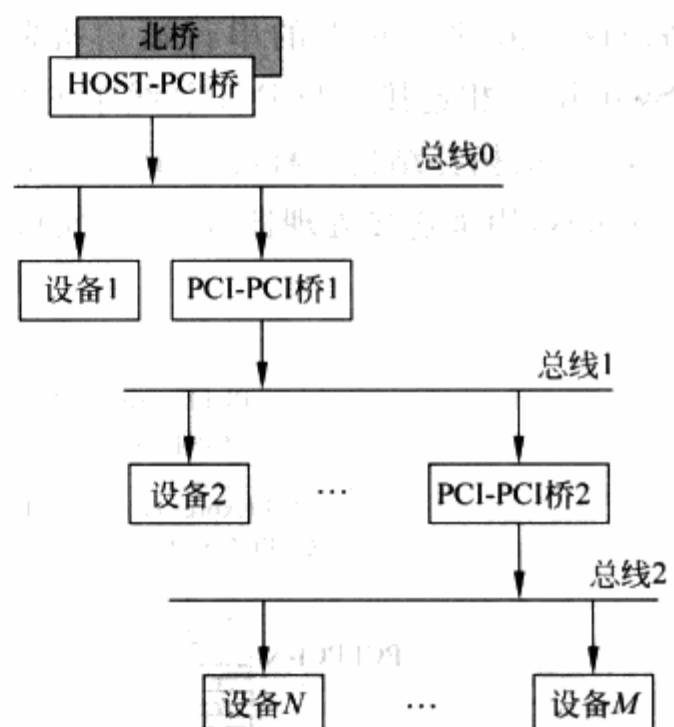


图 2-17 PCI 设备的枚举

当 PCI-PCI 桥收到写入 0xCF8 中的 BDF 后,会将 Bus 字段与自身的 Secondary Bus 相比,相符则在下属总线上搜寻设备;如果不相符,但 Bus 值落在 Subordinate Bus 范围内,则把该地址传递给下属总线中各桥,否则不予理睬。

通过这种方式, BIOS 或操作系统可以枚举出总线上所有设备并为之分配资源,一旦 PCI 配置空间设定好,软件就可以直接通过 PCI Bar 访问设备了。

2.6.4 PCI Express

PCI Express 的设计目标是用来替换当前广泛使用的 PCI、PCI-X 和 AGP 等总线标准,成为新一代通用、高速的 IO 互联标准,同时保持对 PCI 标准的软件兼容性。PCI Express 标准在 PCI-SIG 组织的推动下不断向前发展,在本书完书时已公布的最新版本是 2.0,感兴趣的读者可以从 PCI-SIG 组织的网站获得 PCI Express 的标准文本。

1. PCI Express 架构

PCI Express 抛弃了 PCI 所采用的多个设备共享的并行的总线结构,转而使用了与网络协议类似的点对点的串行通信机制。多个 PCI Express 设备(Endpoint)通过交换器(Switch)互相连接。与 PCI 总线中的桥设备类似,通过交换器,可以搭建一个树形的 PCI Express 的拓扑结构。标准的 PCI Express 拓扑结构如图 2-18 所示。树的根节点是 Root Complex,用来连接处理器、内存系统和 IO 系统,其作用类似 PCI 总线树中的 Host-PCI 桥。

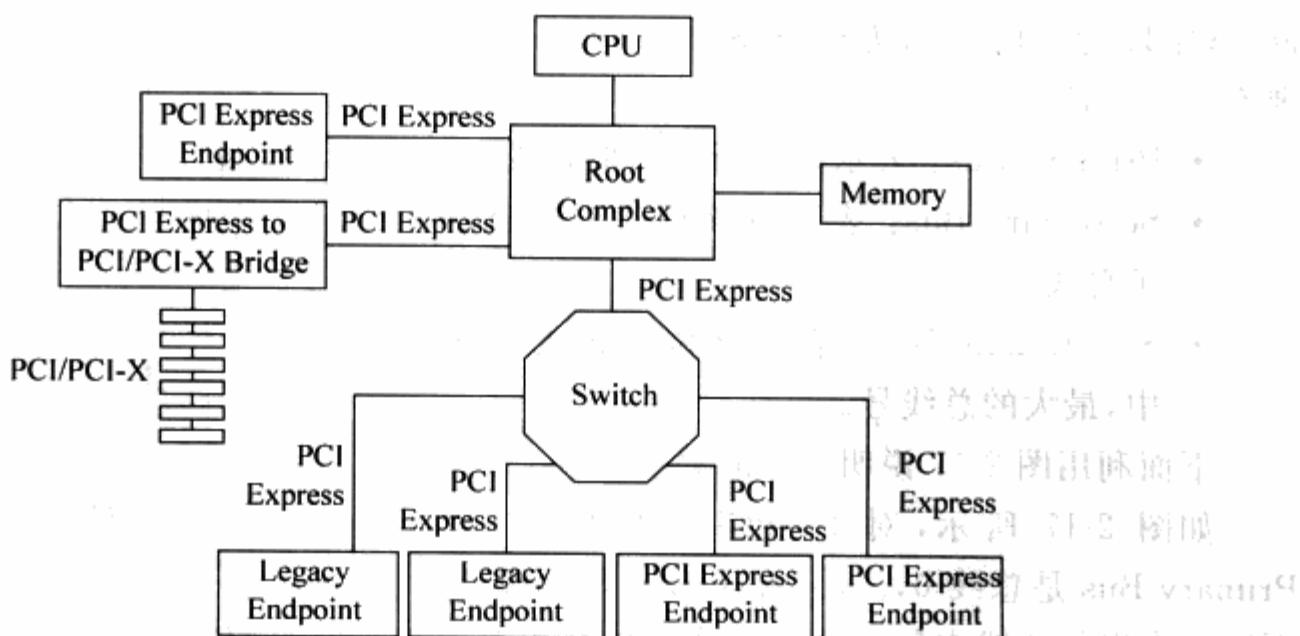


图 2-18 PCI Express 拓扑结构图

2. PCI Express 的优点

PCI Express 比 PCI 总线拥有更高的带宽。串行通信机制可以让物理链路工作在很高的频率,例如,一条 x1 的 PCI Express 链路每个方向上的通信带宽最多可达 5Gb/s。点对点的通信方式使得链路两端的设备可以独占通信带宽,而且多个链路可以并发传输数据。

PCI Express 在与 PCI 总线迥异的硬件基础之上,构建了与 PCI 总线完全兼容的软件接口。PCI Express 定义了基于数据包的分层通信协议,包括物理层(Physical Layer)、数据链路层(Data Link Layer)和事务层(Transaction Layer)。在事务层协议中,PCI Express 定义了内存读写、IO 读写、配置空间读写和消息事务。通过这些事务的定义,PCI Express 可以实现所有的 PCI 总线事务。

PCI Express 将 PCI 总线的配置空间大小从 256 字节扩展到 4KB 字节,解决了困扰 PCI 总线的配置空间过小的问题,可以容纳更多的设备功能配置。PCI Express 还增加了一种新的 MMIO 方法来访问扩充过的配置空间。为了保持兼容性,4KB 字节配置空间中的前 256 字节仍然可以使用原来的方式访问。

PCI Express 除了保留了 PCI 总线的优点以外,还增加了诸如 QoS 服务、高级错误报告(AER)等新的特性。软件可以通过 PCI Express 提供的软件接口来配置和使用这些新的功能。此外,PCI Express 标准具有良好的扩充性。PCI-SIG 的 SR-IOV 标准在 PCI Express 基础上做了扩展,支持该标准的设备可以动态地生成新的逻辑设备。DMA 重映射可以利用 PCI Express 内存读写事务数据包中所包含的设备标识符和地址信息,为每个逻辑设备提供独立的地址转换。

本节对 x86 架构的 I/O 架构做了介绍,并介绍了当前最流行的 PCI 总线。读者通过本章的内容,应该建立起操作系统通过什么方式访问/操作设备的概念,为后面的 I/O 虚拟化内容打下基础。

2.7 时钟

十几年前的教科书总喜欢把计算机比做人,CPU 常常被比作大脑或心脏。如果沿用这个过时的比喻,时钟中断无疑扮演着脉搏的角色。在现代计算机架构中,时钟有着重要的地位,操作系统中的很多事件都是由时钟驱动的,例如进程调度、定时器等。

时钟根据工作方式不同,可以分成如下两类。

(1) 周期性时钟(Periodic Timer): 这是最常见的方式,时钟以固定频率产生时钟中断。通常,周期性时钟会有一个计数器,要么以固定值递减到 0 产生中断,例如 PIT; 要么固定增长,当达到某个阈值时产生中断,同时自动将阈值增加一个固定值,计数器继续递增,例如 HPET。

(2) 单次计时时钟(One-shot Timer): 大多数时钟都可以配置成这种方式,例如 PIT、HPET。其工作方式和到达阈值产生中断的周期性时钟类似,不同的是产生中断后阈值不会自动增加,而是需要软件(通常是时钟中断处理函数)增加该阈值。这提供给软件动态调整下一次时钟中断到来时间的能力,使一些新技术,例如无滴答声内核(Tickless Kernel)的实现成为可能。

2.7.1 x86 平台的常用时钟

由于历史原因,x86 平台出现过多种时钟,目前仍在广泛使用的有以下几种。

(1) PIT(Programmable Interrupt Timer 或 Programmable Interval Timer,可编程中断/间隔时钟): 随 IBM PC 平台产生,被广泛应用,其频率为 1000Hz 左右,即每次中断间隔约为 1ms,通常接 IRQ0,软件可以通过 0x40~0x43 I/O 端口进行操作。PIT 是一种低精度的时钟,容易溢出(16 位),已渐渐被后来出现的高精度时钟取代。PIT 支持周期性和单次计时两种工作方式。

(2) RTC(Real Time Clock,实时时钟): 通常是和 CMOS 集成在一起的,由 CMOS 电池供电,故能在关机后继续计时。其频率范围在 2~8192Hz,通常接 IRQ8,软件可以通过 0x70~0x71 I/O 端口操作。RTC 支持周期性和单次计时两种方式,此外,它还可以配置成每秒产生一次中断,具有闹钟功能。由于具有关机继续计时的功能,RTC 常被用作为操作系统提供日期,即“年/月/日”。

(3) TSC(Time Stamp Counter,时间戳计数器): 和普通时钟不同,它可以看作一个单调递增的计数器(64 位),是由 x86 架构引入的。其时钟频率和 CPU 的频率相关,操作系统在使用前需要计算其频率,例如 1GHz 的 TSC,其值每纳秒增加 1。通过 rdtsc 指令,可以读取当前 TSC 的值。由于不产生时钟中断,故无所谓周期性和单次计时方式。

(4) LAPIC Timer: 该时钟是根据 LAPIC 所在总线(系统总线或 APIC 总线)频率产生的。为 32 位,有如下两个特点。

① 由于 LAPIC 是每个 CPU 一个,故其中断也是对于本地 CPU 的(Per CPU Interrupt)。

② 可以通过寄存器配置,对总线周期进行不同的分频而产生不同频率的时钟中断。

LAPIC Timer 可配置成周期性和单次计时两种工作方式。

(5) HPET(High Precision Event Timer,高精度时间时钟): 是 Intel 和微软共同开发的新型高精度时钟,其最低频率为 10MHz,可以作为 64 位或 32 位时钟使用。HPET 可以提供最多 8 个时钟,典型的实现至少有一个时钟可用。HPET 的时钟通过一个主计数器,和 32 个比较器、匹配器一起,又可以被配置成 32 个子时钟(又称为 channel),每个子时钟可以按不同频率产生不同的中断。例如,可以将一个子时钟配置成每毫秒产生一个 IRQ8 中断,另一个子时钟可以被配置成每微秒产生一个 IRQ0 中断。HPET 可用于替代传统的 PIT 和 RTC,此时平台的 IRQ0、IRQ8 中断被 HPET 占用。HPET 支持周期性和单次计时两种工作方式。

x86 平台提供如此多的时钟,操作系统可以根据不同的需要使用其中的一个或多个。同时使用多个时钟带来的一个明显的缺点是过多的时钟中断会影响系统的性能。所以,当有高精度时钟可用时,操作系统通常禁用低精度的时钟,并根据需要使用高精度时钟模拟低精度时钟。例如,可以用 HPET 代替 PIT,并模拟 RTC。

2.7.2 操作系统的时钟观

第2章

从操作系统的角度来看,时钟的作用可以分为如下两类。

(1) 提供统计值及驱动事件: 提供统计值是指操作系统用时钟来维护一些必需的数据, 例如一个进程在用户态/内核态的时间, 系统的日期、时间等。驱动事件是指驱动以时间为资源的程序, 典型的就是进程。例如, 分时操作系统为每个进程分配固定的时间片, 调度时间片耗尽的进程睡眠, 唤醒分配到新时间片的进程运行。

(2) 维护定时器(Timer): 定时器是程序中常用的组件, 用于在某个指定时间到达后执行特定的操作。定时器大量运用于操作系统中, 例如内核为 I/O 操作注册的超时定时器、操作系统提供给应用程序使用的定时器接口等, 参见图 2-19。

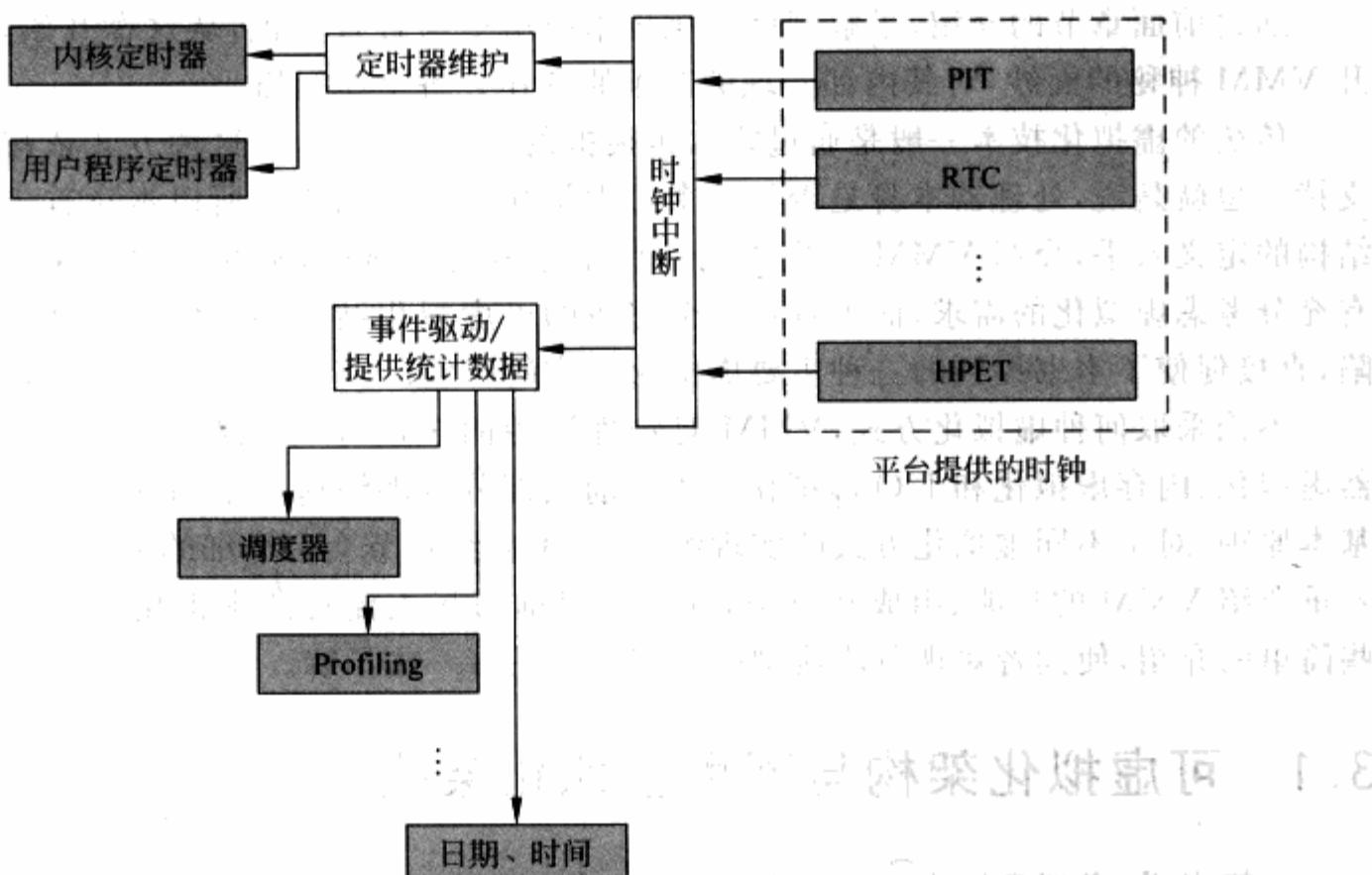


图 2-19 操作系统的时钟

从图 2-19 中可以看出, 操作系统使用时钟的功能, 是以时钟中断为基础的。要了解时钟的作用, 可以以时钟中断为脉络, 勾画出整个框图。

本节介绍了 x86 平台时钟的基本架构。需要指出的是, 操作系统往往会对时钟架构进行封装以方便维护和使用。但从硬件的角度来看, 时钟中断仍然是所有封装的基础, 故虚拟化中对时钟的处理主要是提供准确的时钟中断以模拟硬件的行为。

CHAPTER 3

第 3 章

虚拟化概述

通过前面章节的介绍,了解到虚拟化技术的历史与背景知识,从这章开始,将进一步揭开 VMM 神秘的面纱,对其内部实现的基本原理作一番全面扫描。

传统的虚拟化技术一般是通过陷入再模拟的方式实现的,而这种方式依赖于处理器的支持。也就是说,处理器本身是否是一个可虚拟化的体系结构。所以本章首先从可虚拟化结构的定义入手,介绍 VMM 实现中的一些基本概念。显然,某些处理器在设计之初并没有充分考虑虚拟化的需求,而不具备一个完备的可虚拟化结构。如何填补这些结构上的缺陷,直接促使了本书提到的三种主要虚拟化方式的产生。

不论采取何种虚拟化方式,VMM 对物理资源的虚拟可以归结为三个主要任务:处理器虚拟化、内存虚拟化和 I/O 虚拟化。本章前面部分就围绕这三个部分展开介绍虚拟化的基本原理,对于不同虚拟化方式的实现细节,本书后续章节会有详细的描述。本章后面部分着重介绍 VMM 的功能、组成和分类,并且对目前市场上流行的虚拟化产品及其特点做一些简单的介绍,使读者对现阶段典型的虚拟化产品有一些了解。

3.1 可虚拟化架构与不可虚拟化架构

一般来说,虚拟环境由三个部分组成:硬件、VMM 和虚拟机,如图 3-1 所示。在没有虚拟化的情况下,操作系统直接运行在硬件之上,管理着底层物理硬件,这就构成了一个完整的计算机系统,也就是下文所谓的“物理机”。在虚拟环境里,虚拟机监控器 VMM 抢占了操作系统的位置,变成了真实物理硬件的管理者,同时向上层的软件呈现出虚拟的硬件平台,“欺骗”着上层的操作系统。而此时操作系统运行在虚拟平台之上,仍然管理着它认为是“物理硬件”的虚拟硬件,俨然不知道下面发生了什么,这就是图 3-1 中的“虚拟机”。

虚拟机可以看作是物理机的一种高效隔离的复制,上面的定义里蕴含了三层含义(同质、高效和资源受控),这也是一个虚拟机所具有的三

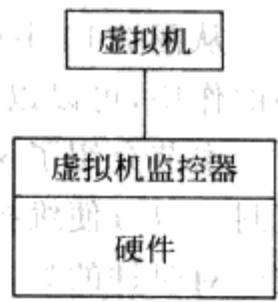


图 3-1 虚拟环境的组成

个典型特征。

关于这三个特点的含义，在第1章已经阐述。

或许硬件体系结构会有所限制，或许VMM的实现方式会有所不同，但如果符合不了上面的三个特点，那么可以说，这个虚拟机是失败的，这个VMM的骗术是不高明的。

上面提到的虚拟机必须具备的三个特点也就决定了不是在任何体系结构下都可以虚拟化的。给定一个系统，其对应的体系结构是否可虚拟化，就要看能否在该系统上虚拟化出具备上面三种特征的虚拟机。

为了进一步研究可虚拟化的条件，先从指令开始着手介绍。

大多数的现代计算机体系结构都有两个或两个以上的特权级，用来分隔系统软件和应用软件。系统中有一些操作和管理关键系统资源的指令会被定为特权指令，这些指令只有在最高特权级上能够正确执行。如果在非最高特权级上运行，特权指令会引发一个异常，处理器会陷入到最高特权级，交由系统软件来处理。在不同的运行级上，不仅指令的执行效果是不同的，而且也并不是每个特权指令都会引发异常。假如一个x86平台的用户违反了规范，在用户态修改EFLAGS寄存器的中断开关位，这一修改将不会产生任何效果，也不会引起异常陷入，而是会被硬件直接忽略掉。

在虚拟化世界里，还有另一类指令被称为敏感指令，简言之就是操作特权资源的指令，包括修改虚拟机的运行模式或者下面物理机的状态；读写敏感的寄存器或是内存，例如时钟或者中断寄存器；访问存储保护系统、内存系统或是地址重定位系统以及所有的I/O指令。

显而易见，所有的特权指令都是敏感指令，然而并不是所有的敏感指令都是特权指令。

为了VMM可以完全控制系统资源，它不允许直接执行虚拟机上操作系统（即客户机操作系统）的敏感指令。也就是说，敏感指令必须在VMM的监控审查下进行，或者经由VMM来完成。如果一个系统上所有敏感指令都是特权指令，则能够用一个很简单的方法来实现一个虚拟环境：将VMM运行在系统的最高特权级上，而将客户机操作系统运行在非最高特权级上，当客户机操作系统因执行敏感指令（此时，也就是特权指令）而陷入到VMM时，VMM模拟执行引起异常的敏感指令，这种方法被称为“陷入再模拟”。

总而言之，判断一个结构是否可虚拟化，其核心就在于该结构对敏感指令的支持上。如果在某些结构上所有敏感指令都是特权指令，则它是可虚拟化的结构；否则，如果它无法支持在所有的敏感指令上触发异常，则不是一个可虚拟化的结构，我们称其存在“虚拟化漏洞”。

我们已经知道，通过陷入再模拟敏感指令的执行来实现虚拟机的方法是有前提条件的：所有的敏感指令必须都是特权指令。否则，要么系统的控制信息会被虚拟机修改或访问，要么VMM会遗漏需要模拟的操作，影响虚拟化的正确性。如果一个体系结构上存在敏感指令不属于特权指令，那么其就存在虚拟化漏洞。有些计算机体系结构是存在虚拟化漏洞的，就是说它们不能很高效地支持系统虚拟化。

虽然虚拟化漏洞有可能存在,但是可以采用一些办法来填补或避免这些漏洞。最简单最直接的方法是,如果所有虚拟化都采用模拟来实现,例如解释执行,就是取一条指令,模拟出这条指令执行的效果,再继续取下一条指令,那么就不存在所谓陷入不陷入的问题,从而避免了虚拟化漏洞。这种方法不但能够适用于模拟与物理机相同体系结构的虚拟机,而且也能模拟不同体系结构的虚拟机。虽然这种方法保证了所有指令(包括敏感指令)执行受到VMM的监督审查,但是它对每条指令不区别对待,其最大的缺点很明显就是性能太差,是不符合虚拟机“高效”特点的,导致其性能下降为原来的十分之一甚至几十分之一。

既要填补虚拟化漏洞,又要保证虚拟化的性能,只能采取一些辅助的手段。或者直接在硬件层面填补虚拟化漏洞,或者通过软件的办法避免虚拟机中使用到无法陷入的敏感指令。这些方法都不仅保证了敏感指令的执行受到VMM的监督审查,而且保证了非敏感指令可以不经过VMM而直接执行,从而相比完全解释执行来说,性能得到了极大的提高。

3.2 处理器虚拟化

处理器虚拟化是VMM中最核心的部分,因为访问内存或者I/O的指令本身就是敏感指令,所以内存虚拟化与I/O虚拟化都依赖于处理器虚拟化的正确实现。

3.2.1 指令的模拟

VMM运行在最高特权级,可以控制物理处理器上的所有关键资源;而客户机操作系统运行在非最高特权级,所以其敏感指令会陷入到VMM中通过软件的方式进行模拟。从客户机操作系统的角度而言,无论一条指令是直接执行在物理处理器上,还是被VMM软件模拟,其期望的执行效果必须正确。所以,处理器虚拟化的关键在于正确模拟指令的行为。

在介绍指令模拟之前,我们理解三个概念:虚拟寄存器、上下文和虚拟处理器。它们有助于理解虚拟处理器模拟指令。

从某种程度上来说,物理处理器无非包括了一些存放数据的物理寄存器,并且规定了使用这些寄存器的指令集,然后按照一段预先写好的指令流,在给定时间点使用给定的部分寄存器来完成某种目的。

在没有虚拟化的环境里,操作系统直接访问物理处理器,处在最高的特权级别,可以控制系统中的所有关键资源,包括寄存器、内存和I/O外设等。但是,当VMM接管物理处理器后,昔日的操作系统成为了客户机操作系统而降级到非最高特权级别上,这时,其试图访问关键资源的指令就成为了敏感指令。VMM会通过各种手段,保证这些敏感指令的执行能够触发异常,从而陷入到VMM进行模拟,以防止对VMM自身的运行造成破坏。

所以,当客户机操作系统试图访问关键资源的时候,该请求并不会真正发生在物理寄存器上。相反,VMM会通过准确模拟物理处理器的行为,而将其访问定位到VMM为其设计

与物理寄存器对应的“虚拟”的寄存器上。当然,从 VMM 实现来说,这样的虚拟寄存器往往是在内存中。

图 3-2 是一个具体的访问控制寄存器 CR0 的例子。当处理器取下一条指令 MOV CR0,EAX 后,发现特权级别不符合,则抛出异常,VMM 截获这个异常之后模拟处理器的行为,读取 EAX 的内容并存放到虚拟的 CR0 中。由于虚拟的 CR0 存放在 VMM 为该虚拟机设计的内存区域里,因此该指令执行的结果并不会让物理的 CR0 的内容发生改变。等到下一次,当虚拟机试图读 CR0 时,处理器也会抛出异常,然后由 VMM 从虚拟的 CR0 而不是物理的 CR0 中返回内容给虚拟机。

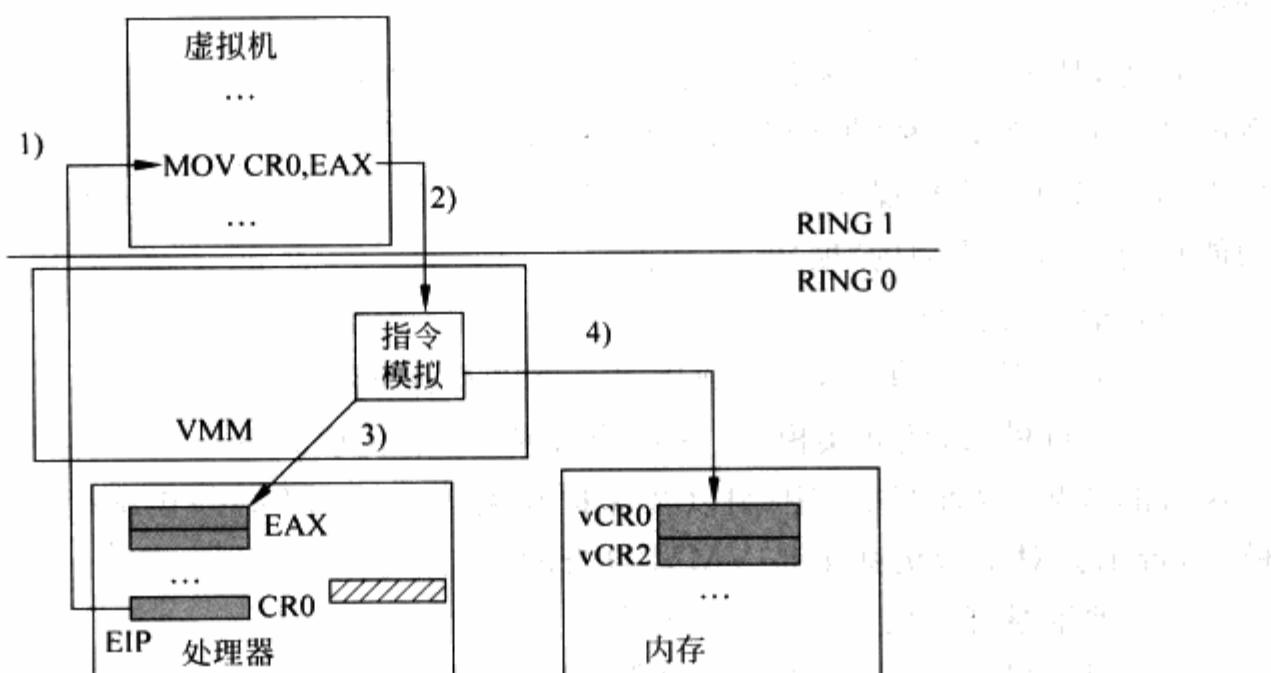


图 3-2 访问虚拟寄存器

在没有虚拟化的环境里,操作系统直接负责物理处理器管理,负责进程间调度和切换。但是,VMM 接管物理处理器后,客户机操作系统没有管理物理处理器的权利,可以说此时它已经运行在 VMM 为之设计的虚拟处理器之上,管理虚拟处理器,并在虚拟处理器上负责该虚拟机内进程间调度和切换。而 VMM 管理物理处理器,负责虚拟处理器的调度和切换,以保证在给定时间内,每个虚拟处理器上的当前进程可以在物理处理器上运行一段时间。但是,不管是何种调度切换,必然要涉及到保留现场,这个现场就是上下文状态,只不过前一种情况是进程上下文,后一种情况是虚拟处理器上下文。为了让读者更好地理解,我们从进程上下文开始类比地介绍。

通过第 2 章我们知道,在某个时刻,物理处理器中的寄存器状态构成了当前进程上下文状态。

进程上下文主要是与运算相关的寄存器状态,例如 EIP 寄存器指向进程当前执行的指令,ESP 存放着当前进程的堆栈指针等。当操作系统进行调度时,当前进程的上下文,即上述寄存器状态被保存在进程特定的内存区域中,而下一个进程的上下文被恢复到相应的寄

存器中,从进程角度看,就好像从未被中断一样。

虚拟处理器上下文比进程上下文更为复杂,因为客户机操作系统本身包含许多敏感指令,会试图访问和修改物理处理器上定义的所有寄存器,而这种访问和修改会被 VMM 重定位到虚拟处理器上。所以,对于虚拟处理器,其上下文包括了更多的系统寄存器,例如 CR0、CR3、CR4 和各种 MSR 等。当 VMM 在决定切换虚拟处理器的时候,为了让虚拟机看来好像也从未被中断过一样,VMM 需要考虑保存与恢复的上下文也更为复杂。

我们没有介绍虚拟处理器,但是上面已经谈到虚拟处理器,那什么是虚拟处理器呢?虚拟处理器其实也是一个虚拟的概念,一个逻辑上而非物理上的概念,可以从两个角度来理解。

首先,从客户机操作系统来说,其在运行的处理器需要具备与其“期望”的物理处理器一致的功能和行为,这种“期望”的前提条件甚至可以允许客户机操作系统的修改,例如 VMM 可以通过修改客户机操作系统的源代码,使客户机操作系统所“期望”的与 VMM 所呈现的功能集合一致。典型的“期望”包括:

- (1) 指令集合与执行效果。
- (2) 可用寄存器集合,包括通用寄存器以及各种系统寄存器。
- (3) 运行模式,例如实模式、保护模式和 64 位长模式等。处理器的运行模式决定了指令执行的效果、寻址宽度与限制以及保护粒度等。VMM 必须正确模拟虚拟机期望的运行模式,否则会对虚拟机甚至是 VMM 自身的运行产生严重影响。

- (4) 地址翻译系统,例如页表级数。
- (5) 保护机制,例如分段和分页等。
- (6) 中断/异常机制,例如虚拟处理器必须能够正确模拟真实处理器的行为,在错误的执行条件下,为虚拟机注入一个虚拟的异常。

其次,从 VMM 的角度来说,虚拟处理器是其需要模拟完成的一组功能集合。虚拟处理器的功能可以由物理处理器和 VMM 共同完成。对于非敏感指令,物理处理器直接解码处理其请求,并将相关的效果直接反映到物理寄存器上;而对于敏感指令,VMM 负责陷入再模拟,从程序的角度也就是一组数据结构与相关处理代码的集合。数据结构用于存储虚拟寄存器的内容,而相关处理代码负责按照物理处理器的行为将效果反映到虚拟寄存器上。

值得一提的是,VMM 已经可以为虚拟机呈现出与实际物理机不一致的功能和行为。例如,虚拟处理器的个数,可以与物理处理器的个数不一致。在有多个物理处理器的平台上,VMM 可以让虚拟机看到该平台好像只有一个物理处理器(即一个虚拟处理器),而在只有一个物理处理器的平台上,VMM 可以让虚拟机看到该平台好像有多个物理处理器(即多个虚拟处理器),这种效果完全取决于用户对虚拟环境的配置,以及 VMM 自身的策略。

在介绍完以上三个概念之后,基本上也就了解了在处理器虚拟化中,不论是定义虚拟寄存器和虚拟处理器,还是利用上下文进行虚拟处理器调度切换,其宗旨都是让虚拟机里执行的敏感指令陷入下来之后,能被 VMM 模拟,而不要直接作用于真实硬件上。

当然,模拟的前提是能够陷入。接下来,理解一下客户机操作系统执行时,是如何通知 VMM 的,也就是 VMM 的陷入方式。概括地说,VMM 陷入是利用了处理器的保护机制,利用中断和异常来完成的,它有以下几种方式。

(1) 基于处理器保护机制触发的异常,例如前面提到的敏感指令的执行。处理器会在执行敏感指令之前,检查其执行条件是否满足,例如当前特权级别、运行模式以及内存映射关系等。一旦任一条件不满足,VMM 得到陷入然后进行处理。

(2) 虚拟机主动触发异常,也就是通常所说的陷阱。当条件满足时,处理器会在触发陷阱的指令执行完毕后,再抛出一个异常。虚拟机可以通过陷阱指令来主动请求陷入到 VMM 中去。例如,在后面介绍的类虚拟化技术就是通过这种方式实现 Hypercall 的。

(3) 异步中断,包括处理器内部的中断源和外部的设备中断源。这些中断源可以是周期性产生中断的时间源,也可以是根据设备状态产生中断的大多数外设。一旦中断信号到达处理器,处理器会强行中断当前指令,然后跳到 VMM 注册的中断服务程序,所以这也为 VMM 的陷入提供了一种途径。例如,VMM 可以通过调度算法指定当前虚拟机运行的时间片长度,然后编程外部时钟源,确保时间片用完时触发中断,从而允许 VMM 进行下一次调度。

3.2.2 中断和异常的模拟及注入

中断和异常机制是处理器提供给系统程序的重要功能,异常保证了系统程序对处理器关键资源的绝对控制,而中断提供了与外设之间更有效的一种交互方式。所以,VMM 在实现处理器虚拟化时,必须正确模拟中断与异常的行为。

VMM 对于异常的虚拟化需要完全遵照物理处理器对于各种异常条件的定义,再根据虚拟处理器当时的内容,来判断是否需要模拟出一个虚拟的异常,并注入到虚拟环境中。

VMM 通常会在硬件异常处理程序和指令模拟代码中进行异常虚拟化的检查。无论是哪一条路径,VMM 需要区分两种原因:一是虚拟机自身对运行环境和上下文的设置违背了指令正确执行的条件;二是虚拟机运行在非最高特权级别,由于虚拟化的原因触发的异常。第二种情况是由于陷入再模拟的虚拟化方式所造成的,并不是虚拟机本身的行为。而对于第一种情况的检查,VMM 实际是在虚拟处理器的内容上进行的,因为它反映了虚拟机所期望的运行环境。错误的异常注入会让客户机操作系统做出错误的反应,后果无法预知。

物理中断的触发来自于特定的物理中断源,同样,虚拟中断的触发来自于虚拟设备的模拟程序。当设备模拟器(后面会讲到)发现虚拟设备状态满足中断产生的条件时,会将这个虚拟中断通知给中断控制器的模拟程序,例如模拟 LAPIC。最后,VMM 会在特定的时候检测虚拟中断控制器的状态,来决定是否模拟一个中断的注入。而这里的虚拟中断源包括:处理器内部中断源的模拟,例如 LAPIC 时钟、处理器间中断等;外部虚拟设备的模拟,例如 8254、RTC、IDE、网卡和电源管理模块等;直接分配给虚拟机使用的真实设备的中断,通常

来自于 VMM 的中断服务程序；自定义的中断类型。

不管怎样，当 VMM 决定向虚拟机注入一个中断或是异常时，它需要严格模拟物理处理器的行为来改变客户指令流的路径，而且还要包括一些必需的上下文保护与恢复。VMM 需要首先判断当前虚拟机的执行环境是否允许接受中断或是异常的注入，假如客户机操作系统正好通过 RFLAGS.IF 位禁止了中断的发生，这时 VMM 就只能把中断事件暂时缓存起来，直到某时刻客户机操作系统重新允许了中断的发生，VMM 才立即切入来模拟一个中断的注入。而当中断事件不能被及时注入时，VMM 还要进一步考虑如下因素。

(1) 该中断类型是否允许丢失中断，如果允许，VMM 则可以将其后到达的多个同类型中断合为一个事件；否则，VMM 必须要跟踪所有后续到达的中断实例，在客户指令流重新允许中断时，将每一个缓存的中断一一注入。

(2) 该中断在阻塞期间是否被中断源取消，这决定了 VMM 是否会额外地注入一个已经被取消的假中断。

(3) 当一次阻塞的中断实例比较多时，VMM 可能还需要考虑客户机操作系统能否处理短时间内大量同类型的中断注入，因为这在真实系统中可能并不出现。

当然，上面只是简单地列举了一些中断注入时需要考虑的因素，在实际的实现中基于正确性与效率方面还有更多需要考虑的因素，详细内容请参考后面的章节。

在模拟中断或异常的注入时，VMM 需要首先判断是否涉及到运行模式的切换。假如虚拟机可能运行在一个 64 位兼容模式，而其中断/异常处理函数运行在 64 位长模式，这时 VMM 就需要按照处理器的规定，将虚拟机的运行模式进行软件切换，对保存的客户上下文进行相应的修改。在可能的模式切换完成后，VMM 还需要根据真实处理器在该模式下的中断注入过程，完整地进行软件模拟。例如，将必需的处理器状态（指令地址、段选择子等）复制压入当前模式下对应中断/异常服务程序的堆栈；到中断模拟逻辑去查找发生中断的向量号；根据该向量号来查找相关的中断/异常服务程序的入口地址；最后修改虚拟机的指令地址为上述人口地址，然后返回到虚拟机去执行等。

总的来说，中断/异常的虚拟化由中断/异常源的定义、中断/异常源与 VMM 处理器虚拟化模块间的交互机制以及最终模拟注入的过程所组成。不同的 VMM 在这个方面的设计和实现都不尽相同，根据不同的客户操作系统的类型也会造成差异。同样，请详细阅读后面的章节来了解针对不同 VMM 模式的具体实现。

3.2.3 对称多处理器技术的模拟

在没有虚拟化的环境里，对称多处理器技术可以让操作系统拥有并控制多个物理处理器，它通过提供并发的计算资源和运算逻辑，允许上层操作系统同时调度多条基于不同计算目的的进程并发执行，从而有效地提高系统的吞吐率与性能。

同样道理，当物理计算资源足够多时，VMM 也可以考虑为虚拟机呈现出多个虚拟处理

器,也就是客户对称多处理器虚拟化技术,也称客户 SMP 技术。这样,当这些虚拟处理器同时被调度在多个物理处理器上执行时,也可以有效地提高给定虚拟机的性能。虽然从 VMM 调度的角度来说,因为 VMM 仍然在物理处理器上基于一定的策略来管理多个虚拟处理器,客户 SMP 功能的加入并没有带来整体性能改观。但是,对某个虚拟机来说,以前每个虚拟处理器属于各自不同的虚拟机,而现在客户 SMP 功能引入后,某些具有相同属性的虚拟处理器可以隶属于同一个虚拟机,从而使该虚拟机因为拥有多个计算资源而可以让其虚拟机性能较其他虚拟机得到提高,如图 3-3 所示。

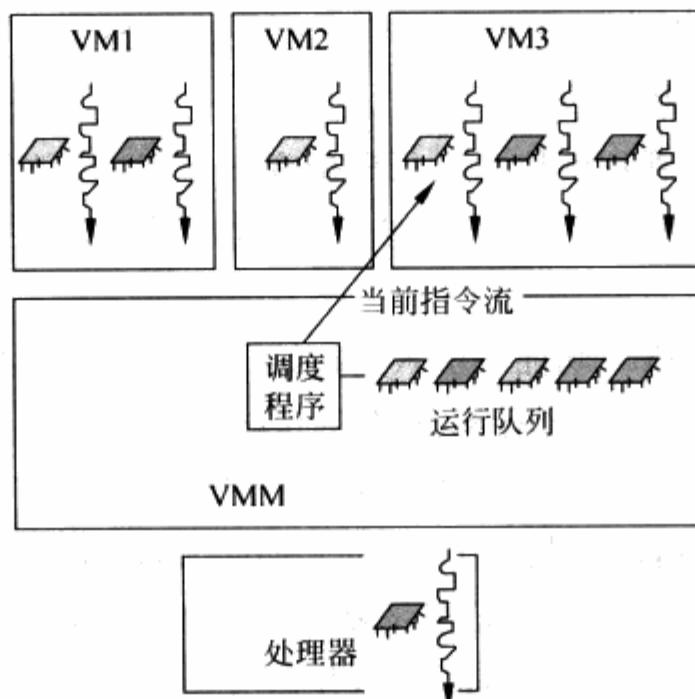


图 3-3 对称多处理器的模拟

因此可以说,客户 SMP 中虚拟处理器的数目与实际物理处理器数目之间没有必然联系,也就是说,客户 SMP 中虚拟处理器的个数可以小于、等于或是大于实际物理处理器个数。当然,客户 SMP 功能引入后,VMM 在虚拟环境的管理和责任上发生了一些变化。

首先,VMM 必须按照客户机操作系统期望的那样呈现客户 SMP 的存在,这样客户机操作系统才不会认为其运行在单一处理器上,才会试图初始化其他的虚拟处理器,并在其上运行调度程序。VMM 可以是模拟一个现实中的接口,例如通过 ACPI 表来表述;也可以是自定义一个接口协议,只要客户机操作系统被修改来配合 VMM 即可。

其次,我们知道,SMP 的并发执行能力虽带来了性能上的提升,但多个处理器竞争共享资源的情况也给软件实现带来了更多复杂性。为了保证 SMP 情况下多处理器访问共享资源的正确性,通常系统程序需要实现一套同步机制来协调处理器之间的步调,从而确保任何时候只有一个处理器能对共享资源进行修改,并且在释放修改权之前,确保修改的效果能够被每个处理器觉察到。在客户 SMP 机制引入后,实际上 VMM 面临着物理处理器之间(也被称为主机 SMP)以及虚拟处理器之间(即客户 SMP)的同步问题。

(1) 对于发生在 VMM 自身代码之间的同步问题,由 VMM 负责协调物理处理器之间

的步调来满足主机 SMP 的要求。

(2) 对于发生在同一个虚拟机内部,多个虚拟处理器间的同步问题,通常 VMM 并不需要参与,因为这是客户机操作系统自身的职责。VMM 只需要在客户机操作系统发起某种特权操作,例如刷新页表时,正确地模拟其效果即可。

(3) 对于 VMM 造成的虚拟处理器之间的同步问题,仍然需要 VMM 来负责处理。例如,VMM 可能将 N 个虚拟处理器在 $M(M > N)$ 个物理处理器之间进行迁移,客户机操作系统只知道自己有 N 个虚拟处理器,所以只会在这 N 个虚拟处理器的上下文内进行同步操作,但当 VMM 将这 N 个虚拟处理器迁移到 M 个物理处理器上运行时,VMM 就必须负责所有 M 个物理处理器上状态的同步。

最后,VMM 对虚拟机管理模块也必须根据客户 SMP 的存在做相应的修改。例如,挂起命令要区分挂起虚拟处理器还是挂起虚拟机,当挂起某个虚拟机就必然挂起该虚拟机内部所有指令流的执行。

下面来看一下在客户 SMP 功能被引入后初始化过程是如何模拟的。

通常,对称多处理器技术定义有标准的一套初始化过程。在没有虚拟化的环境里, BIOS 负责选取 BSP(主启动处理器)与 AP(应用处理器),把所有处理器都初始化到某种状态后, BIOS 在 BSP 上通过启动加载程序(Boot Loader)跳转至操作系统的初始化代码,同时所有的 AP 处于某种等待初始化硬件信号的状态。接下来,操作系统会在初始化到某个时刻时,发出某种初始化硬件信号给所有的 AP,并提供一段特定的启动代码,AP 在收到初始化硬件信号后,就会跳转到操作系统指定的启动代码中继续执行。通过这样一种方式,操作系统最终就成功地按自己的方式初始化了所有的处理器,最后在每个处理器上独立地运行调度程序。

那么,在虚拟环境里,客户 SMP 功能被引入后初始化过程是怎样的?注意,此时讨论的是 VMM 已经启动运行起来,而客户机操作系统正处在初始化阶段。VMM 选择第一个虚拟处理器作为 BSP,其他虚拟处理器为 AP,把所有虚拟处理器都初始化到某种状态。这里又分为两种情况:如果客户机操作系统是不能修改的,而它又期望看到虚拟处理器与物理处理器加电重设后一样的状态,VMM 就必须按照软件开发手册上对于处理器加电重设状态的描述,设置虚拟处理器的寄存器状态,包括虚拟控制寄存器和虚拟运行模式等;如果客户机操作系统可以修改,VMM 就可以使用一套自定义的协议而不必依照规范定义的那样,例如直接跳过实模式把虚拟处理器初始化为保护模式。接下来,当启动代码初始化到某个时刻时,AP 需要收到某种初始化信号被唤醒。这里还是相应地分为两种情况:如果客户机操作系统不能被修改,则 VMM 负责截获客户机操作系统发出的 INIT-SIPI-SIPI 序列,唤醒其他虚拟的 AP;如果客户机操作系统可以被修改,VMM 也可以自定义一套简单的唤醒机制。

3.3 内存虚拟化

在介绍完处理器虚拟化基本原理之后,接着来看一下内存虚拟化。首先从一个操作系统的角度,介绍其对物理内存存在的两个主要基本认识:物理地址从 0 开始和内存地址连续性。而内存虚拟化的产生,主要源于 VMM 与客户机操作系统在对物理内存的认识上存在冲突,造成了物理内存的真正拥有者——VMM,必须对客户机操作系统所访问的内存进行一定程度上的虚拟化。所以在这一节的最后一部分,会进一步介绍内存虚拟化相关的一些知识与方法。读者将会看到,内存虚拟化既满足了客户机操作系统对于内存和地址空间的特定认识,也可以更好地在虚拟机之间、虚拟机与 VMM 之间进行隔离,防止某个虚拟机内部的活动影响到其他的虚拟机甚至是 VMM 本身,从而造成安全上的漏洞。

下面先分析一下没有虚拟化的环境。在这种环境里,任何一个操作系统都认为自己完全控制处理器,相应的就完全拥有了内存的所有权。所以,操作系统总是按照一台物理计算机上内存的属性和特征对其进行管理。

那么,再来重温一下第 2 章对于内存和地址空间的介绍。指令对于内存的访问都是通过处理器来转发的,首先处理器会将解码后的请求发送到系统总线上,然后由芯片组来负责进一步转发。为了唯一标识,处理器采用统一编址的方式将物理内存映射成为一个地址空间,即所谓的物理地址空间。平时,我们把一根根内存条插到主板上的内存插槽中,每根内存条都需要被映射到物理地址空间中某个位置。一般来说,每根内存插槽在物理地址空间的起始地址可以在主板制造时就固定下来,也可以通过某种方式由 BIOS 加电后自动配置。一旦内存插槽的起始地址被固定下来,这根内存条上每个字节的物理地址就相应地确定下来了。总的来看,一根根内存条形成了一个连续的物理地址空间,而且这个物理地址空间一定是从 0 开始的。

例如有 4 个内存插槽的主板,每个插槽插上 256MB 的内存条,如果这 4 条插槽的起始地址分别固定为 0x00000000、0x10000000、0x20000000 和 0x30000000,那么在它们上面的物理内存就被映射成 0x00000000—0x0FFFFFFF、0x10000000—0x1FFFFFFF、0x20000000—0x2FFFFFFF、0x30000000—0x3FFFFFFF 这 4 段。总的来说,这 4 根内存条组成该系统 1GB 的内存,而且这 1GB 的内存是从 0 开始的连续空间,4 根内存条上每个字节都会对应到一个唯一的物理地址。处理器访问任何一个字节就是通过请求一个物理地址,芯片组收到处理器发出的内存访问请求后,就会检测内部维护的物理地址空间的分配表,当发现目标地址落在 0x00000000—0x3FFFFFFF 范围内时,处理器就会进一步把请求转发给内存控制器。

在没有虚拟化的环境里,操作系统也会假定物理内存是从物理地址 0 开始的。以 x86 处理器上运行 Linux 内核为例。在 x86 上,Linux 内核可执行文件头里定义了每个段的大小、期望在物理地址空间中被加载的位置即 1MB,以及加载后执行第一条指令的地址等,这些信息在编译连接阶段就确定下来了。由于加载的位置是 1MB,那么对于后面代码,其访

问的段都是基于 1MB 这个起始地址的,这也是在编译链接阶段就确定下来了的。通常,在加载内核时,启动加载程序(Boot Loader)就会通过对该文件格式的分析,将相应的段复制到期望的位置,然后跳转到内核文件指定的入口点。而系统所做的,必须保证在该指定位置存在可用内存。如果物理地址空间不是从 0 开始的,Boot Loader 将会因指定位置找不到可用内存而拒绝加载内核,即使是把内核加载到内存中了,由于内核代码在访问段时也会自身产生错误而造成整个系统的崩溃。

除此之外,现实中的操作系统基本上对内存连续性存在一定程度的依赖性,如 DMA。DMA 的目的就是允许设备绕过处理器来直接访问物理内存,从而保证 I/O 处理的高效。目前绝大多数设备都支持 DMA 功能,只是在实现上对驱动程序提出了不同的要求。如图 3-4 所示。

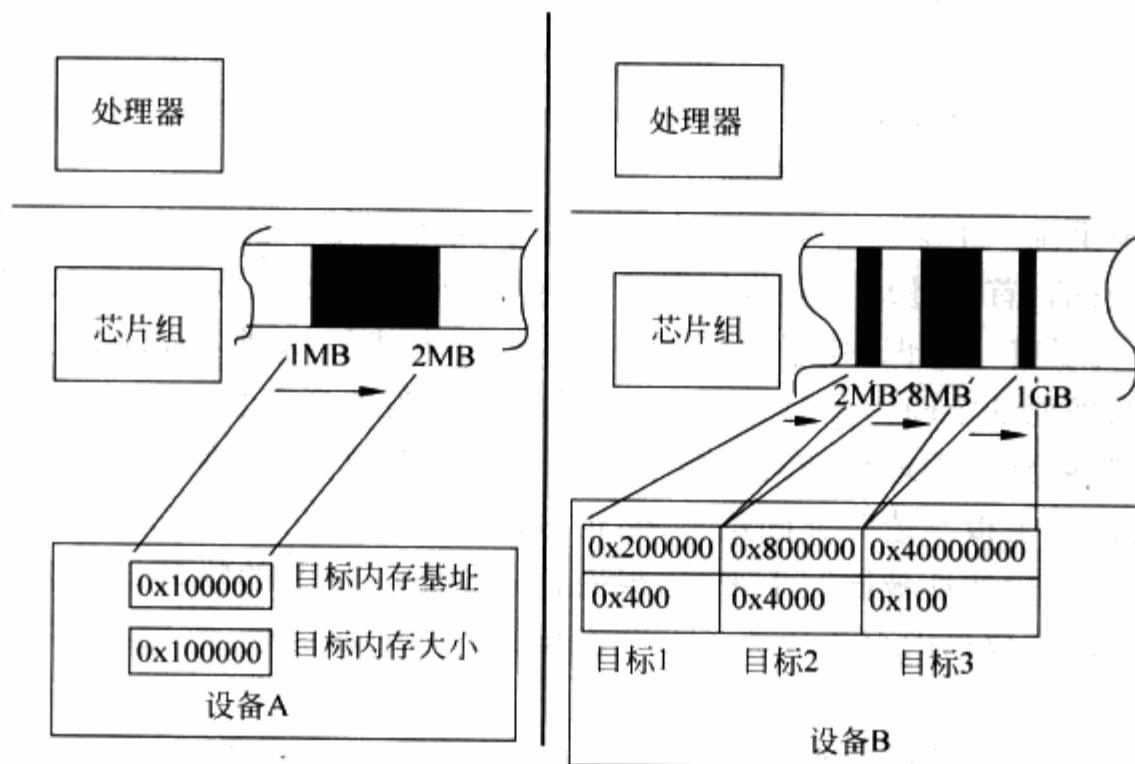


图 3-4 DMA 对内存连续性的要求

图 3-4 是一个简化的例子,现实中的设备在 DMA 的逻辑上要复杂得多。但是,这并不妨碍我们以这个简化的模型来说明问题。左边的设备使用了一种最直接的方式,即驱动程序提供 DMA 的目标内存地址 0x100000 以及大小 1MB,然后设备顺序地访问从 0x100000 到 0x200000 的内存。很容易看到,当一个内存页面大小小于 1MB 时,就需要请求几个在物理上连续的内存页面,以满足设备顺序访问内存的需求。而右边的设备则使用了一种更加灵活的方式,叫做分散-聚合(Scatter-Gather),它允许驱动程序一次提供多个物理上不连续的内存段,设备通过相关信息来离散地访问这些不连续的目标内存。

在实际设备中,这两种模式都非常普遍,而且即使在后一种模式中,设备允许的离散块是有限的,为了支持更大的 DMA 区域,驱动程序仍然会在每一个离散块中分配多个连续的内存页面,这就意味着驱动程序必须能够从操作系统中分配到足够多连续的空闲内存页来

满足 DMA 的要求。

除了满足上述设备 DMA 的需求外,操作系统还在其他方面存在对内存连续性的要求。例如,在某些情况下物理连续的页面可以简化程序的设计,并且带来性能上的提升。又比如,操作系统可以在处理器的帮助下,使用大页面映射的方式加速对一块连续内存页面的访问。

总而言之,在没有虚拟化的情况下,操作系统在对内存的使用与管理上已经达成了以下两点认识。

(1) 内存都是从物理地址 0 开始的。

(2) 内存都是连续的,或者说至少在一些大的粒度(如 256MB)上连续。

了解了操作系统对物理内存的假定和认识后,在虚拟环境里,VMM 的任务就是模拟使得虚拟出来的内存仍然符合客户机操作系统对内存的假定和认识。

因此,在虚拟环境里,内存虚拟化面临的问题是:物理内存要被多个客户机操作系统同时使用,但物理内存只有一份,物理起始地址 0 也显然只有一个,无法同时满足所有客户机操作系统内存从 0 开始的要求;由于使用内存分区方式,把物理内存分给多个客户机操作系统使用,客户机操作系统的内存连续性要求虽能得到解决,但是内存的使用效率非常不灵活。在面临这些问题的情况下,VMM 所要做的就是“欺骗”客户机操作系统,让这种谎言满足客户机操作系统对内存的那两点要求,这种欺骗过程,就是内存虚拟化。

内存虚拟化的核心,在于引入一层新的地址空间——客户机物理地址空间。

在图 3-5 中,VMM 负责管理和分配每个虚拟机的物理内存,客户机操作系统所看到的是一个虚构的客户机物理地址空间,其指令目标地址也是一个客户机物理地址。这样的地址在无虚拟化的情况下,其实就是实际物理地址。但是,在有虚拟化的情况下,这样的地址是不能被直接发送到系统总线上去的,需要 VMM 负责将客户机物理地址首先转换成一个实际物理地址后,再交由物理处理器来执行。

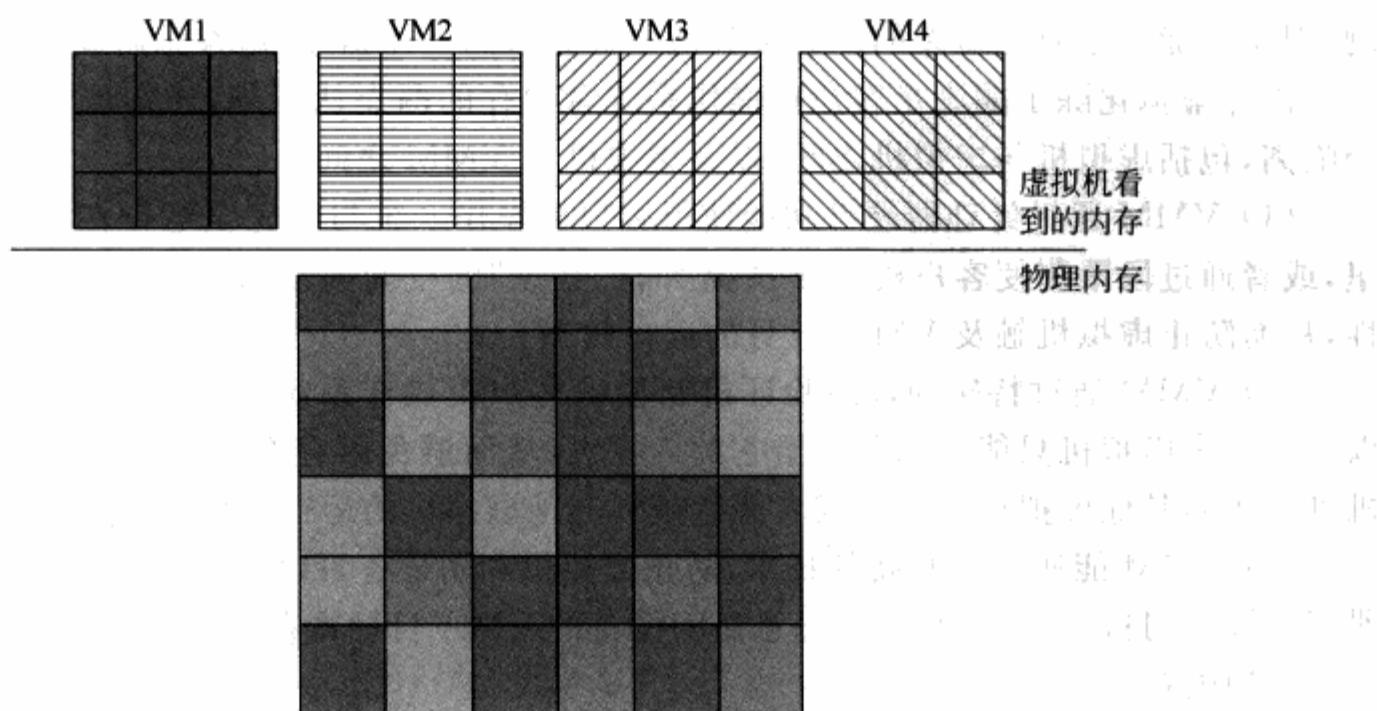


图 3-5 内存虚拟化示意图

这里值得一提的是,为了更有效地利用空闲的物理内存,尤其是系统长期运行后产生的碎片,VMM 通常会以比较小的粒度(如 4KB)进行分配,这就造成了给定一个虚拟机的物理内存实际上是不连续的问题,其具体位置完全取决于 VMM 的内存分配算法。

由于引入了客户机物理地址空间,内存虚拟化就主要处理以下两个方面的问题。

- (1) 给定一个虚拟机,维护客户机物理地址到宿主机物理地址之间的映射关系。
- (2) 截获虚拟机对客户机物理地址的访问,并根据所记录的映射关系,将其转换成宿主机物理地址。

第一个问题相对比较简单,因为这只是一个数据结构的映射问题。在实现过程中,客户机操作系统采用客户页表维护了该虚拟机里进程所使用的虚拟地址到客户机物理地址的动态映射关系,用一个简单的公式表示就是 $GPA = f1(GVA)$ 。这里,GVA 代表客户机虚拟地址,GPA 代表客户机物理地址。而 VMM 负责维护客户机物理地址到宿主机物理地址之间的动态映射关系,用一个简单的公式表示就是 $HPA = f2(GPA)$ 。这里,HPA 代表宿主机物理地址。虚拟机里一个进程所使用的客户机虚拟地址要变成物理处理器可以执行的宿主机物理地址,需要经过两层转换,即 $HPA = f2(f1(GVA))$ 。

VMM 内存虚拟化任务就是跟踪客户页表,当其发生变化时,及时地切入,构造一个有效的客户机虚拟地址到宿主机物理地址间的映射关系,加到物理处理器所遍历的真实页表上。具体如何实现,请参见后续章节。

第二个问题从实现上来说相对更加复杂和具有挑战性,也是衡量一个虚拟机的性能最重要的方面。再者,地址转换一定要发生在物理处理器处理目标指令之前,否则一旦客户机物理地址被直接发送到系统总线上,那会对整个虚拟环境,包括其他虚拟机以及 VMM 自身,造成严重的破坏和巨大的漏洞。

一个最简单的办法就是设法让虚拟机对客户机物理地址空间的每一次访问都触发异常,然后由 VMM 来查询地址转换表模拟其访问。这种方法的完备性和正确性没有任何问题,但是性能上绝对是最差的。有关其他更好方法的具体描述,请参见后续章节。

内存虚拟化除了能满足客户机操作系统对内存的两点认识外,还实现了整个系统的安全隔离,包括虚拟机与虚拟机之间,以及虚拟机与 VMM 之间。

(1) VMM 通过处理器硬件功能使得客户机操作系统与之完全运行在不同的地址空间里,或者通过段限制使客户机操作系统所能“看见”的空间大小,以保证 VMM 自身的安全性,从而防止虚拟机触及 VMM 自身的运行状态。

(2) VMM 通过特殊的权限验证机制使得客户机操作系统局限在给定的地址空间里,以保证一个虚拟机只能访问分配给它的内存页,从而避免因存在设计上的漏洞让某个虚拟机可以访问其他虚拟机的信息或者恶意破坏其他虚拟机的运行环境。

(3) VMM 能通过一些硬件技术,防止虚拟机利用设备可以通过 DMA 的方式绕过处理器而直接访问目标内存的特点,恶意访问设备的 DMA 目标寄存器,进而通过设备越权访问所有物理内存。

有关上面的详细描述,请参见后续章节。

3.4 I/O 虚拟化

现实中的外设资源也是有限的,为了满足多个客户机操作系统的需求,VMM 必须通过 I/O 虚拟化的方式来复用有限的外设资源。VMM 截获客户操作系统对设备的访问请求,然后通过软件的方式来模拟真实设备的效果。模拟软件本身作为物理驱动程序众多客户端中的一个,从而有效地实现了物理资源的复用。由于从处理器的角度看,外设是通过一组 I/O 资源(端口 I/O 或者是 MMIO)来进行访问的,所以设备相关的虚拟化又被称为 I/O 虚拟化。基于设备类型的多样化,以及不同 VMM 所构建的虚拟环境上的差异,I/O 虚拟化的方式和特点纷繁复杂,不一而足。

3.4.1 概述

首先分析一下以往没有虚拟化的情形。给定一个外设,定义有自己的一套供软件访问的接口,这些接口的属性可能是单向的,也可能是双向的。操作系统含有外设的驱动程序,它们接收来自其他模块(如用户进程)的请求,然后按照外设规定好的方式驱动外设完成特定的任务。驱动程序并不关心外设内部的逻辑电路是如何实现的,只要驱动程序按照定义好的接口使用外设,外设总会通过其内部逻辑电路完成期望的效果。由于处理器在计算机中的核心地位,因此外设的访问接口最终也会被映射到处理器所能认识的地址空间或者其他资源中。这样,当驱动程序通过指令的方式访问外设接口时,处理器才能正确识别目标对象,然后将相关请求发送到系统总线上,最终由芯片组转发给目标外设。图 3-6 是一个典型设备可能会具有的资源。

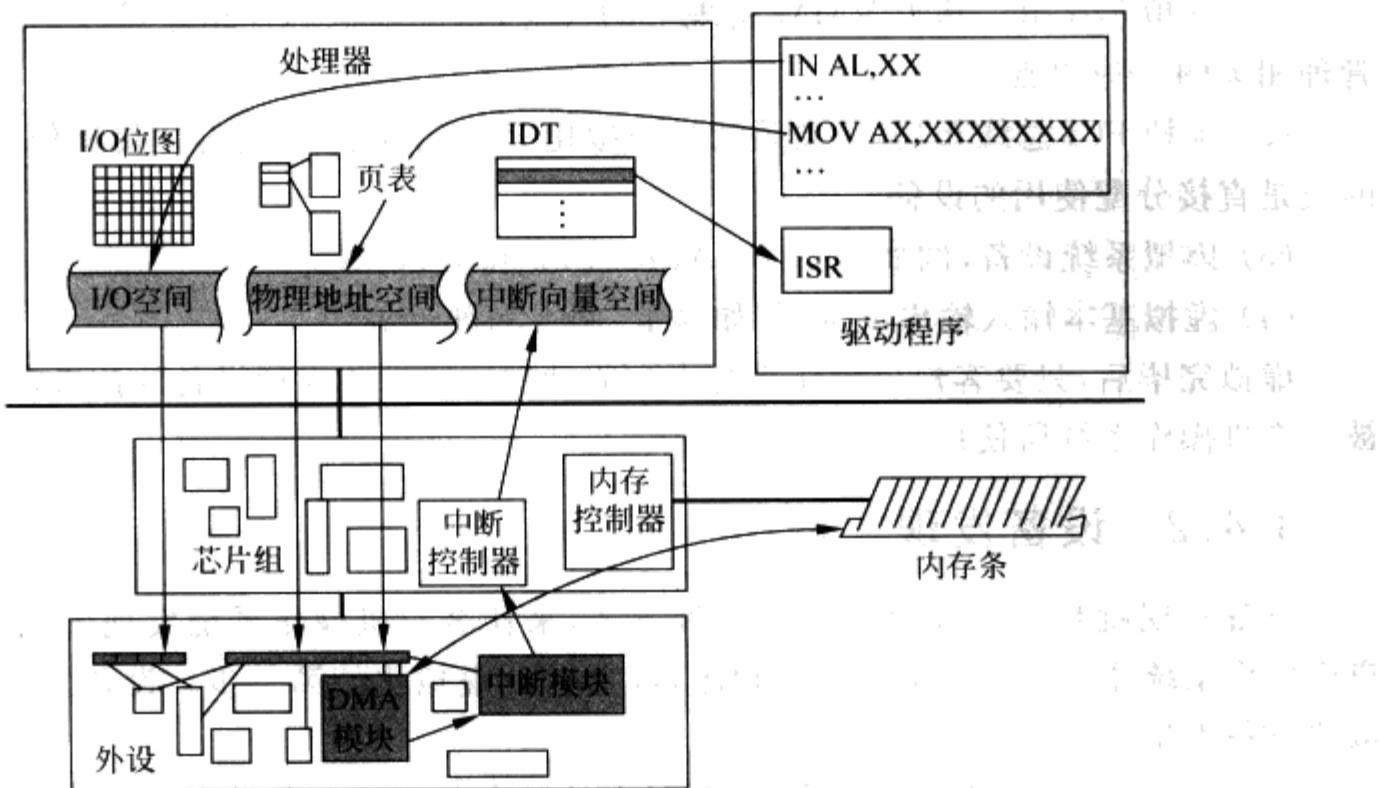


图 3-6 一个典型设备具有的资源

(1) I/O 端口寄存器。它被映射到 I/O 地址空间中,这个空间由特殊的指令如 IN/OUT 访问。如果指令流处于最高特权级别,整个 64KB 的 I/O 地址空间都可以自由访问,驱动程序就是这样一个例子;当指令流运行在其他特权级别时,只有 I/O 位图中允许的端口才可以访问。

(2) MMIO 寄存器。它被映射到物理地址空间中,通过页表的方式来控制访问权限。

(3) 中断。由于外设的物理构造,速度远低于处理器,往往需要异步事件通知的方式来完成延后的操作,这个通知机制通常通过中断模块实现。每个允许发生中断的外设会在处理器的中断向量空间中分配一个序号,中断模块发出中断消息,通过芯片组中的中断控制器通知给处理器,最后处理器会中断当前指令流,跳到中断描述符表 IDT 中对应该向量的中断服务程序执行。

I/O 端口、MMIO 与中断模块组成了一个典型外设呈现给软件的基本资源。图 3-6 中还额外标注了外设中的一个特殊模块——DMA 模块。如第 2 章所述,DMA 提供给设备不经处理器而直接访问内存的方式,从而特别适用于大批量数据的批量传输。从访问方式来说,DMA 模块被映射在 I/O 端口或者是 MMIO 中。

在虚拟环境里,I/O 面临的问题是:现实中外设资源是有限的,为了满足多个客户机操作系统对外设访问的需求,VMM 必须通过 I/O 虚拟化的方式复用有限的外设资源。在面临这种问题的情况下,VMM 所要做的还是模拟,即截获客户机操作系统对设备的访问请求,然后通过软件的方式模拟真实物理设备的效果,仍然是“欺骗”客户机操作系统。这种模拟过程,就是 I/O 虚拟化。

I/O 虚拟化并不需要完整地虚拟化出所有外设的所有接口,究竟怎样做完全取决于设备与 VMM 的策略以及客户机操作系统的需求。

(1) 虚拟芯片组。基于 VMM 实现上的考虑,这个虚拟芯片组还可以承担 ACPI 电源管理相关的一些功能。

(2) 虚拟 PCI 总线布局,主要是通过虚拟化 PCI 配置空间,为客户机操作系统呈现虚拟的或是直接分配使用的设备。

(3) 虚拟系统设备,例如 PIC、IO-APIC、PIT 和 RTC 等。

(4) 虚拟基本输入输出设备,例如显卡、网卡和硬盘等。

虚拟完毕后,只要客户机操作系统中有驱动程序遵守该虚拟设备的接口定义,它就可以被客户机操作系统所使用。

3.4.2 设备发现

设备发现就是要让 VMM 提供一种方式,来让客户机操作系统发现虚拟设备,这样客户机操作系统才能加载相关的驱动程序,这是 I/O 虚拟化的第一步。设备发现取决于被虚拟的设备类型。

(1) 模拟一个所处物理总线的设备,这其中又包括以下两种类型。

① 模拟一个所处总线类型是不可枚举的物理设备,而且该设备本身所属的资源是硬编码固定下来的。这类设备典型的例子就是 ISA 设备、PS/2 键盘、鼠标、RTC 及传统 IDE 控制器等。对于这类设备,驱动程序可能会通过设备特定的方式来检测设备是否存在,例如读取特定端口的状态信息。所以,只要 VMM 在给定端口进行了正确的模拟,客户机操作系统就能够成功地检测到虚拟设备的存在。

② 模拟一个所处总线类型是可枚举的物理设备,而且相关设备资源是软件可配置的。这类设备典型的例子就是 PCI 设备。由于 PCI 总线通过 PCI 配置空间定义了一套完备的设备发现方式,并且允许系统软件(BIOS 或操作系统)通过 PCI 配置空间的一些字段对给定 PCI 设备进行资源的配置,例如允许或禁止 I/O 端口和 MMIO,设置 I/O 和 MMIO 的起始地址等。所以,VMM 仅靠模拟设备自身的逻辑是不够的,它必须进一步地模拟 PCI 总线的行为,包括拓扑关系和设备特定的配置空间内容,以便让客户机操作系统发现这类虚拟设备。

(2) 模拟一个完全虚拟的设备,例如前面介绍的 FE/BE 模型。在这种情况下,因为没有一个现实中的规范与之对应,这种虚拟设备所处的总线类型将完全由 VMM 自行决定,VMM 可以选择将虚拟设备挂在 PCI 总线上,也可以完全自定义一套新的虚拟总线协议。因此,当遇到相对复杂的 PCI 总线需要模拟时,VMM 完全可以自定义并使用一套简化的、适用于该类型虚拟设备的总线,避免模拟的复杂性。当然,客户机操作系统里也必须加载一个特殊的总线驱动程序才行。因此,使用原来 PCI 总线的好处是兼容性,客户机操作系统可以重用已有的总线驱动来发现虚拟设备。而引入新的总线协议带来的问题就是维护性,而且,为了在不同的 VMM 上运行,同一个客户机操作系统可能需要加载许多不同的总线协议驱动程序。

3.4.3 访问截获

现在,虚拟设备已经被客户机操作系统发现了,客户机操作系统中的驱动程序就会按照接口定义访问这个虚拟设备。VMM 的问题又来了,它不仅需要知道那个虚拟设备相关的接口资源,而且它还得找到有效的办法来截获客户机操作系统对虚拟设备的访问,并进行模拟。可以毫不犹豫地说,关键点就在于处理器虚拟化。

对于一个非直接分配给客户机操作系统使用的设备,假如该设备可以具有端口 I/O 资源,那么我们知道处理器对于端口 I/O 资源的控制在于指令流所处的特权级别和相关 I/O 位图。由于客户机操作系统被降级而运行在一个非特权的环境里,客户机操作系统是否能够访问给定 I/O 端口就完全由 I/O 位图来决定。自然地,VMM 可以把设备的所有端口 I/O 从 I/O 位图中关闭,这样,当客户指令流在访问该 I/O 端口时,物理处理器就会及时地抛出一个保护异常,接着,VMM 就可以获得异常原因,然后将请求发送给设备模拟器进行模拟。相反,对于一个直接分配给客户机操作系统使用的设备,VMM 可以把该设备所属端口 I/O 从 I/O 位图中打开,这样,处理器就会把访问发送到系统总线,最终到达目标物理设备而不被模拟。

对于一个非直接分配给客户机操作系统使用的设备,假如该设备可以提供 MMIO 资源,那么我们知道 MMIO 本身也是物理地址空间的一部分,而物理地址空间的访问控制是通过页表来控制的,因此在物理处理器遍历的真实页表里,VMM 只要把映射到该 MMIO 的页表项设为无效,当客户指令流试图再访问目标地址时,物理处理器就会抛出一个缺页异常,接着 VMM 遍历客户页表,就可以发现设备所属的 MMIO 资源,然后将请求发送给设备模拟器进行模拟。相反,对于一个直接分配给客户机操作系统使用的设备,VMM 只需要按照客户页表的设置打开真实页表的映射即可,这样客户机操作系统对该设备的访问也不再被模拟。

对于一个非直接分配给客户机操作系统使用的设备,假如该设备可以产生中断,那么 VMM 只要提供一种机制,供设备模拟器在接收到物理中断并需要触发中断时,可以通知到虚拟中断逻辑,然后由虚拟中断逻辑模拟一个虚拟中断的注入;相反,对于直接分配给客户机操作系统的设备,VMM 物理中断处理函数在接收到物理中断后,辨认出中断源属于某个客户机,然后通知该客户机的虚拟中断逻辑。

假如某设备可以提供 DMA 或类似的共享内存机制,那么,我们知道 DMA 允许设备绕过处理器直接访问目标内存,而若客户驱动程序是未经修改的,则设备模拟器接收到的 DMA 目标地址是客户机物理地址。因此,VMM 只要提供一种机制,让设备模拟器可以了解各种地址之间转换关系,从而可以把客户机物理地址映射成自己的虚拟地址,就能真正做到对 DMA 目标地址的访问了。

3.4.4 设备模拟

至于虚拟化方式,虚拟现实可以与现实设备具有完全一样的接口定义,从而允许客户机操作系统中的原有驱动程序无须修改就能驱动这个虚拟设备。这时,VMM 的设备模拟器往往需要仔细研究现实设备的接口定义和内部设计规范,然后以软件的方式模拟真实的逻辑电路来满足每个接口的定义和效果,例如 PS/2 键盘、鼠标等。在这种情况下,现实设备具备哪些资源,设备模拟器需要呈现出同样的资源。

既然无论如何 VMM 需要在客户机操作系统中提供一个特定的驱动程序,那么还可以把这个驱动程序进一步简化,并称客户机操作系统中的驱动程序为前端(Front-End, FE)设备驱动,而 VMM 中的驱动程序为后端(Back-End, BE)设备驱动。简化就是前端程序将来自于其他模块的请求通过客户机之间的特殊通信机制直接发送给后端程序,而后端程序在处理完请求后再发回通知给前者,如图 3-7 所示。与传统设备驱动程序流程比较,传统设备驱动程序为了完成一次操作要涉及到多个寄存器的操作,使得 VMM 要截获每个寄存器访问并进行相应的模拟,也就导致多次上下文切换。但是,这种方式是基于请求/事务的,能在很大程度上减少上下文切换的频率,提供更大的优化空间,这种方式可以看作是上一种方式的衍生。

如果直接将物理设备分配给某个客户机操作系统,由客户机操作系统直接访问目标设备,VMM 不需要为这种方式提供模拟,客户机操作系统中原有的驱动程序也可以无缝地操作目

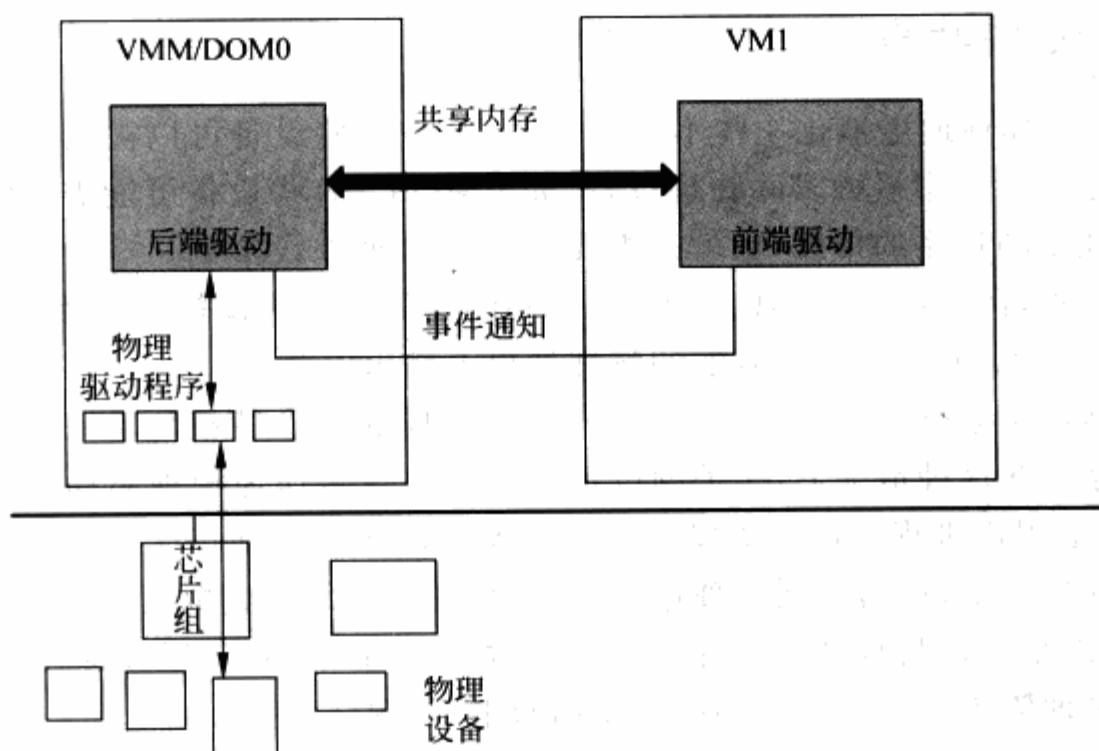


图 3-7 Xen FE/BE 模型的 I/O 虚拟化

标设备,这种 I/O 虚拟化的方式从性能上来说是最优的。但是,这种虚拟方式仍然会受到最大可用资源的限制。目前与此相关的技术有 IOMMU,如 Intel、VT-d 和 PCI-SIG 的 SR-IOV 等。

归纳一下以上三种 I/O 虚拟化方式,如表 3-1 所示。

表 3-1 不同 I/O 虚拟化方式对比

I/O 虚拟化方式	兼容性	性 能	成 本	扩 展性
设备接口完全模拟	重用已有驱动程序	纯粹的软件模拟引起太多的上下文切换	没有额外硬件开销	设备模拟器一端是瓶颈
FE/BE 模拟	需要加载特定的驱动程序	基于事务的通信机制简化了 FE/BE 之间的开销	没有额外硬件开销	BE 一端是瓶颈
直接分配之 VT-d	重用已有驱动程序	直接访问物理设备,减少了虚拟化的开销	需要购买较多额外的硬件	有局限性,例如主板上的扩展槽一般是有限的
直接分配之 SR-IOV	需要加载新的驱动程序	直接访问物理资源,减少了虚拟化的开销	需要购买较少额外的硬件	扩展性较好,因为 SR-IOV 设备本身支持多个虚拟机的同时访问

I/O 虚拟化的方式是非常灵活的,完全基于 VMM 本身对于所构建的虚拟环境的设计。而所虚拟出来的设备可以与某种真实物理设备一致,也可以截然不同,在这种情况下,VMM 需要在客户机操作系统中显式地安装特殊的驱动才能使之正常工作。在性能成为关键瓶颈时,VMM 甚至可以直接把物理设备控制权交给某个客户机操作系统。

3.4.5 设备共享

VMM 可以选择性地虚拟化一些不同的设备,其中有些设备可以被设备模拟器用软件的方式完全模拟掉而不用接触实际物理设备,如 CMOS。有些设备可能需要设备模拟器进一步去请求物理驱动程序的帮助,也就是需要利用到物理外设资源。一般输入输出类设备就属于后一类设备,例如鼠标、键盘、显卡、硬盘和网卡等。这些设备都涉及到或者从真实的物理外设上获取输入,或者需要往真实的物理外设上输出内容。

设备模拟器通常运行在一个 I/O 特权环境中,这样的 I/O 特权环境中有驱动物理外设的物理驱动程序。在这种情况下,相关的设备模拟器本身是作为物理驱动程序的一个客户而存在,例如一个用户进程。I/O 特权环境中的大多数物理驱动程序都是可以同时接收多个客户或是进程的请求,从而达到物理资源的复用。同样,每个虚拟机都有自己专属的设备模拟逻辑,也就是在 I/O 特权环境中存在一个相对应的用户进程。通过这种方式,I/O 虚拟化就有效地将物理资源在多个虚拟环境中复用起来,如图 3-8 所示。

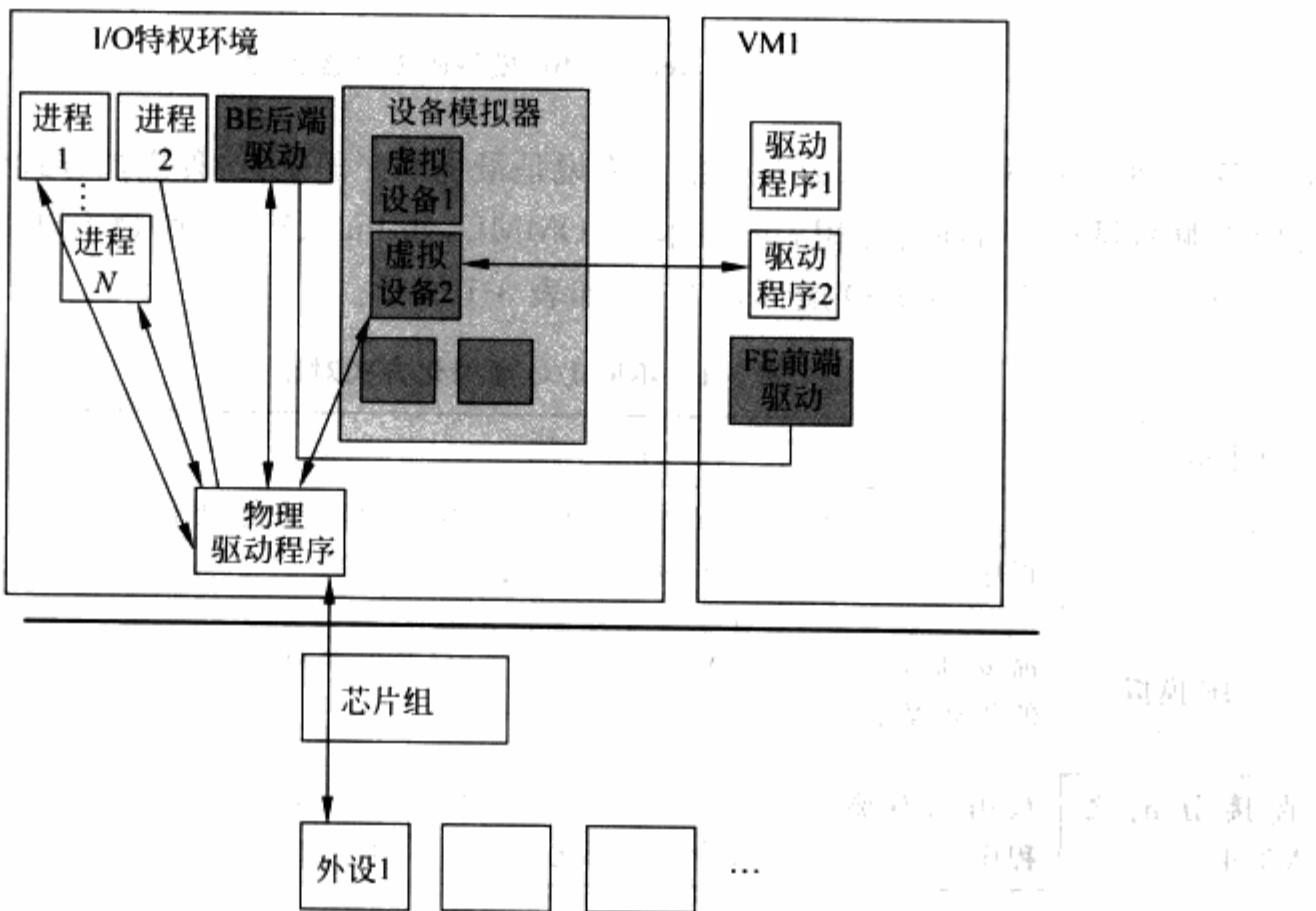


图 3-8 设备复用

3.5 VMM 的功能和组成

前面已经说到,一个成功的且骗术高明的 VMM 主要职责是构建符合那三个特点的虚拟机,物理机器上能够运行操作系统,虚拟机上也能够运行操作系统,只不过此时叫客户机

操作系统。从软件角度来看,物理机器是由处理器、内存和 I/O 设备等一组资源构成的实体。虚拟机也一样,由虚拟处理器、虚拟内存和虚拟 I/O 设备等组成。VMM 的主要功能是基于物理资源创建相应的虚拟资源,组成虚拟机,为客户机操作系统提供虚拟的平台。

顺理成章,VMM 基本上可以分为两部分:虚拟环境的管理和物理资源的管理。前一部分是所有 VMM 产品需要提供的基本功能,而后一部分根据实现结构的差异,其存在也各有差异。

3.5.1 虚拟环境的管理

1. 虚拟资源

通过截获客户机操作系统对处理器、内存和外设等资源的访问,来构建一个虚拟环境。在这个环境中,客户机操作系统认为自己运行在一台计算机上,并唯一地拥有这台“虚拟”机器上的所有资源,其中虚拟化的原理在本章前面部分已经介绍了,所以 VMM 需要提供如下基本模块。

- (1) 处理器虚拟化模块。为虚拟机提供虚拟处理器。
- (2) 内存虚拟化模块。为虚拟机提供虚拟内存。
- (3) 设备虚拟化模块。为虚拟机提供虚拟 I/O 设备。

2. 虚拟环境的调度

既然 VMM 可以同时构建多个虚拟环境,从而允许多个客户机操作系统并发执行,那么随之而来 VMM 必须实现一套策略来有效地调度。

VMM 的调度程序和操作系统的调度程序类似。在本章前面讲述虚拟处理器上下文时已经介绍过,操作系统调度程序的调度单位是进程/线程,VMM 调度程序的调度单位是虚拟处理器。当虚拟处理器被调度到时,VMM 调度程序负责将虚拟处理器上下文装载到物理处理器上,然后虚拟处理器所对应的客户机指令开始真正被执行。当时间片用完或者虚拟处理器主动让出,调度程序会被触发。调度程序根据调度策略,挑选下一个虚拟处理器继续运行。

与操作系统一样,VMM 的调度策略可以有多种,例如平均分配时间片来进行调度,或者按照虚拟机的权重来分配时间片进行调度等。

3. 虚拟机间通信机制

与操作系统中的进程间通信机制类似,虚拟环境下也存在虚拟机间通信机制。虚拟机间通信机制的作用,顾名思义,就是为虚拟机提供相互之间通信的手段。在某些情况下,虚拟机之间需要互相通信来完成特定功能。举例来说,后面介绍的类虚拟化 I/O 中是基于事务的模型,一个 I/O 事务需要特权虚拟机和正常虚拟机共同合作完成,中间就会大量用到虚拟机间通信。

虚拟机间通信机制从实现上来说可以多种多样。通常来说,VMM 实现虚拟机间的通信机制,并向虚拟机提供相应的 API。虚拟机的客户机操作系统通过调用这些 API 与其他

虚拟机进行通信。这些 API 可以是事件通知,也可以是内存共享等。在虚拟机间通信机制的实现上,严格的安全权限检查也是必需的,否则虚拟机之间的隔离就会受到影响。另外,VMM 除了提供虚拟机之间通信的 API 外,也提供虚拟机与 VMM 之间交互的 API。

4. 虚拟化环境的管理接口

虚拟机的管理功能也非常重要。可管理性是用户挑选虚拟化产品的重要指标之一。实现上来说,虚拟机的管理功能由上层的管理程序和 VMM 提供的管理接口组成。VMM 需要提供一组完备的管理接口,来支持虚拟环境的创建、删除、暂停、查询和迁移等功能。上层的管理程序则通过调用 VMM 提供的管理接口,为用户提供管理界面。

3.5.2 物理资源的管理

与操作系统一样,VMM 本身也承担了全部或者部分物理资源管理的角色。

1. 处理器管理

包括系统启动时检测并获取所有的处理器;对每个处理器进行初始化,如设置运行模式、设置页表、设置中断处理函数等;将所有的处理器纳入调度序列,由调度程序对处理器进行调度。有些 VMM 还支持对物理处理器的热插拔,当有处理器插入时,VMM 得到通知,初始化后将其纳入管理队列当中;当处理器拔出时,VMM 同样得到通知,将该处理器上的任务迁移到其他处理器上,并将其从管理队列中删除。有些 VMM 还具有高可靠性的支持,当收到处理器失效通知时,如 MCA(Machine Check Abort),VMM 将其做热拔出处理。

2. 内存管理

包括系统启动时 VMM 检测并且获得所有内存;对获得的内存进行初始化,包括分页并设置页表等;提供内存分配的接口,以便 VMM 的其他模块能够获得/释放内存;给虚拟机分配内存,并且维护虚拟机物理地址与实际物理地址的映射关系,以供 VMM 的其他模块查询使用。

3. 中断管理

VMM 负责初始化并设置中断相关的资源,如处理器中断向量表、Local APIC 和中断控制器(I/O APIC、8259 PIC)。当中断发生后,VMM 是接收者,它会根据中断的来源,或者直接处理,或者转发到相关特权虚拟机来处理。

4. 系统时间维护

VMM 拥有和时间相关的硬件资源,因此 VMM 负责维护系统时间,并且向各虚拟机提供虚拟化的时间。

5. 设备管理

在 Hypervisor 模型下,所有的外设都属于 VMM,因此,VMM 需要包含所有设备的驱动程序来管理这些设备。在混合模型下,大部分的外部设备属于特权客户操作系统,由特权客户操作系统的驱动程序来管理这些外设。VMM 也拥有少部分的设备,如用于调试的串

口,因此也需要包含这些设备的驱动程序。在本章后面,有关于 Hypervisor 模型和混合模型的详细描述。

3.5.3 其他模块

除了前两节所介绍的基本功能外,通常 VMM 还会包括以下功能模块。

1. 软件定时器

软件定时器为 VMM 的其他模块提供了一种方法,使其能够在未来指定的某时刻执行指定的动作。软件定时器通常是通过时钟中断处理函数来实现的,在 VMM 内部被广泛应用,如系统时间的维护等。

2. 多处理器同步原语(spinlock、rcu 等)

与操作系统一样,当多处理器共享同一个资源时,VMM 需要提供同步原语来同步多处理器的读写访问,保证资源的正确性。

3. 调试手段(包括系统级别和虚拟环境特定)

调试手段对于 VMM 的开发不可或缺。printk 是最简单有时也是最有效的调试手段。有些 VMM 也会开发出类似于 gdb 的调试工具。虚拟机还为客户机操作系统的调试提供了极好的环境,由于 VMM 能够控制虚拟机的一切,因此,VMM 级别上的调试工具能够轻松地获取客户机操作系统的实时信息,控制客户机操作系统的运行,极大地方便了用户的调试。

4. 性能采集与分析工具

VMM 通常也会提供 profiling 工具,用于性能数据的采集和分析。这些功能能够采集 VMM 全局的性能数据,也能够采集针对某个虚拟机的性能数据。

5. 安全机制

从基本功能上来说,VMM 需要保证各个虚拟机之间,以及虚拟机与 VMM 之间是隔离的,虚拟机上运行的恶意代码只能影响该虚拟机本身,不能影响 VMM 和其他虚拟机。此外,虚拟化技术的出现,为安全提供了新的平台。学术界也针对如何利用 VMM 提高安全性提出了很多想法。具体参见第 10 章的相关内容。

6. 电源管理

与操作系统类似,VMM 也支持电源管理,包括处理器电源管理、睡眠状态电源管理等。详细阐述请参见第 10 章的相关内容。

3.6 VMM 的分类

3.6.1 按虚拟平台分类

根据 VMM 所提供的虚拟平台类型可以将 VMM 分成两类:第一类 VMM 虚拟的是现实存在的平台,并且在客户机操作系统看来,虚拟的平台和现实的平台是一样的,客户机操

作系统察觉不到是运行在一个虚拟平台上。这样的虚拟平台可以运行现有的操作系统，无须对操作系统进行任何修改，因此这种方式被称为完全虚拟化(Full Virtualization)。第二类 VMM 虚拟的平台是现实中不存在的，而是经过 VMM 重新定义的，这样的虚拟平台需要对所运行的客户机操作系统进行或多或少的修改使之适应虚拟环境，因此客户机操作系统知道其运行在虚拟平台上，并且会去主动适应。这种方式被称为类虚拟化(Para-Virtualization)。另外，值得指出的是，一个 VMM 可以既提供完全虚拟化的虚拟平台，又提供类虚拟化的虚拟平台。

1. 完全虚拟化

如上所述，在客户机操作系统看来，完全虚拟化的虚拟平台和现实平台是一样的，客户机操作系统无须做任何修改就可以运行。这就意味着客户机操作系统会像操作正常的处理器、内存、I/O 设备一样来操作虚拟处理器、虚拟内存和虚拟 I/O 设备。从实现的角度来看，VMM 需要能够并且正确处理客户机所有可能的行为，进一步说，客户机的行为是通过指令反映出来的，因此 VMM 需要能够正确处理所有可能的指令。对于完全虚拟化来说，所有可能的指令是指所虚拟的处理器其手册规范上定义的所有指令。

在实现方式上，以 x86 架构为例，完全虚拟化经历了两个阶段：软件辅助的完全虚拟化和硬件辅助的完全虚拟化。

1) 软件辅助的完全虚拟化

在 x86 虚拟化技术的早期，x86 体系没有在硬件层次上对虚拟化提供支持，因此完全虚拟化只能通过软件实现。一个典型的做法是优先级压缩(Ring Compression)和二进制代码翻译(Binary Translation)相结合。

优先级压缩的原理是：由于 VMM 和客户机运行在不同特权级上，对应到 x86 架构上，通常是 VMM 运行在 Ring 0，客户机操作系统内核运行在 Ring 1，客户机操作系统应用程序运行在 Ring 3。当客户机操作系统内核执行相关特权指令时，由于处在非特权的 Ring 1，因此通常会触发异常，VMM 截获该特权指令并进行虚拟化。Ring Compression 能够正确处理大部分特权指令，但是由于 x86 指令体系在设计之初并没有考虑到虚拟化，因此有些指令还是不能通过 Ring Compression 正常处理，即在 Ring 1 中做特权操作的时候却没有触发异常，从而 VMM 不能够截获并做相应处理。

二进制代码翻译方法因此被引入来处理这些虚拟化不友好的指令。二进制代码翻译的思想也很简单，就是通过扫描并修改客户机的二进制代码，将难以虚拟化的指令转化为支持虚拟化的指令。VMM 通常会对操作系统的二进制代码进行扫描，一旦发现需要处理的指令，就将其翻译成为支持虚拟化的指令块(Cache Block)。这些指令块可以与 VMM 合作访问受限的虚拟资源，或者显式地触发异常让 VMM 进一步处理。此外，由于该技术可以修改客户机的二进制代码，因此也被广泛应用于性能优化，即将某些造成性能瓶颈的指令替换成更加高效的指令来提高性能。

优先级压缩和二进制代码翻译技术虽然能够实现完全虚拟化，但是这种打补丁的方式

很难在架构上保证其完整性,因此,x86 厂商在硬件上加入了对虚拟化的支持,从而在硬件架构上实现了虚拟化。

2) 硬件辅助完全虚拟化

很多问题,如果在本身的层次上难以解决,那么通过增加一个层次,在其下面一个层次就会变得容易解决。硬件完全虚拟化就是这样一种方式,既然操作系统已经是硬件之上的最后一层系统软件,如果硬件本身加入足够的虚拟化功能,就可以截获操作系统对敏感指令的执行或者对敏感资源的访问,从而通过异常的方式报告给 VMM,这样就解决了虚拟化的问题。Intel 的 VT-x 技术是这一方向的代表。VT-x 技术在处理器上引入了一个新的执行模式用于运行虚拟机。当虚拟机执行在这个特殊模式中时,它仍然面对的是一套完整的处理器寄存器集合和执行环境,只是任何特权操作都会被处理器截获并报告给 VMM。VMM 本身运行在正常模式下,在接收到处理器的报告后,通过对目标指令的解码,找到对应的虚拟化模块进行模拟,并把最终的效果反映在特殊模式下的环境中。

硬件虚拟化是一种完备的虚拟化方法,因为内存和外设的访问本身也是由指令来承载,对处理器指令级别的截获就意味着 VMM 可以模拟一个与真实主机完全一样的环境。在这个环境中,任何操作系统只要能够在现实中的等同主机上运行,也就可以在这个虚拟机环境中无缝地运行。

在第 6 章会对硬件辅助完全虚拟化的各方面技术进行详细阐述。

2. 类虚拟化

类虚拟化是通过在源代码级别修改指令以回避虚拟化漏洞的方式来使 VMM 能够对物理资源实现虚拟化。上面谈到 x86 存在一些难以虚拟化的指令,完全虚拟化通过 Binary Translation 在二进制代码级别上来避免虚拟化漏洞。类虚拟化采取的是另一种思路,即修改操作系统内核的代码(即 API 级),使得操作系统内核完全避免这些难以虚拟化的指令。操作系统通常会使用到处理器提供的全部功能,例如特权级别、地址空间和控制寄存器等。类虚拟化首先需要解决的问题就是如何插入 VMM。典型的做法是修改操作系统的处理器相关代码,让操作系统主动让出特权级别,而运行在次一级特权上。这样,当操作系统试图去执行特权指令时,保护异常被触发,从而提供截获点供 VMM 来模拟。

既然内核代码已经需要修改,类虚拟化进一步可以被用于优化 I/O。也就是说,类虚拟化不是去模拟真实世界中的设备,因为太多的寄存器模拟会降低性能。相反,类虚拟化可以自定义出高度优化的 I/O 协议。这种 I/O 协议完全基于事务,可以达到近似物理机的速度。

3.6.2 按 VMM 实现结构分类

前面介绍了 VMM 的组成部分,还可以将当前主流的虚拟化技术实现结构分为三类。

1. Hypervisor 模型

在 Hypervisor 模型中,VMM 首先可以被看做是一个完备的操作系统,不过和传统操作系统不同的是,VMM 是为虚拟化而设计的,因此还具备虚拟化功能。从架构上来看,首

先,所有的物理资源如处理器、内存和 I/O 设备等都归 VMM 所有,因此,VMM 承担着管理物理资源的责任;其次,VMM 需要向上提供虚拟机用于运行客户机操作系统,因此,VMM 还负责虚拟环境的创建和管理。

图 3-9 展示了 Hypervisor 模型的架构,其中处理器管理代码(Processor, P)负责物理处理器的管理和虚拟化,内存管理代码(Memory, M)负责物理内存的管理和虚拟化,设备模型(Device Model, DM)负责 I/O 设备的虚拟化,设备驱动(Device Driver, DR)则负责 I/O 设备的驱动,即物理设备的管理。VMM 直接管理所有的物理资源,包括处理器、内存和 I/O 设备,因此,设备驱动是 VMM 的一部分。此外,处理器管理代码、内存管理代码和设备模型也是 VMM 的一部分。

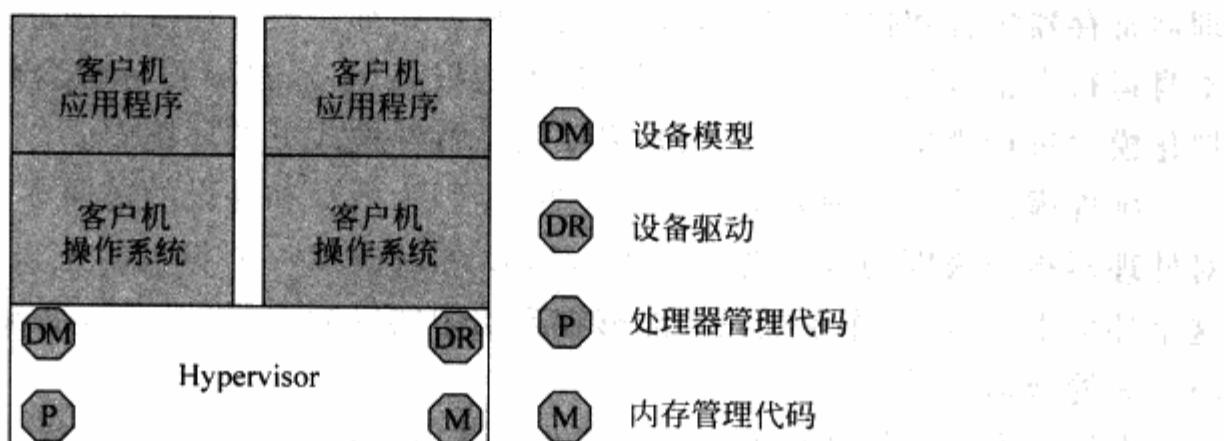


图 3-9 Hypervisor 模型 VMM

在 Hypervisor 模型中,由于 VMM 同时具备物理资源的管理功能和虚拟化功能,因此,物理资源虚拟化的效率会更高一些。在安全方面,虚拟机的安全只依赖于 VMM 的安全,不像下面将会提到的宿主模型,需要同时依赖于 VMM 和宿主机操作系统的安全。

有得必有失,Hypervisor 模型在拥有虚拟化高效率的同时也暴露出其缺点。由于 VMM 完全拥有物理资源,因此,VMM 需要进行物理资源的管理,包括设备的驱动。我们知道,设备驱动开发的工作量是很大的,因此,这对于 Hypervisor 模型来说是个很大的挑战。事实上,在实际的产品中,基于 Hypervisor 模型的 VMM 通常会根据产品定位,有选择地挑选一些 I/O 设备来支持,而不是支持所有的 I/O 设备。例如,如果是面向服务器市场的,那么会只挑选服务器上的 I/O 设备来开发设备驱动。此外,在基于 Hypervisor 模型中,很多功能必须在 VMM 中重新实现,例如调度和电源管理等,无法像宿主模型那样借助宿主机操作系统。

2. 宿主模型

与 Hypervisor 模型不同,在宿主模型中,物理资源由宿主机操作系统管理。宿主机操作系统是传统操作系统,如 Windows、Linux 等,这些传统操作系统并不是为虚拟化而设计的,因此本身并不具备虚拟化功能,实际的虚拟化功能由 VMM 来提供。VMM 通常是宿主机操作系统独立的内核模块,有些实现中还包括用户态进程,如负责 I/O 虚拟化的用户态设备模

型。VMM 通过调用宿主机操作系统的服务来获得资源,实现处理器、内存和 I/O 设备的虚拟化。VMM 创建出虚拟机之后,通常将虚拟机作为宿主机操作系统的一个进程参与调度。

图 3-10 展示了宿主模型的架构。由于宿主机操作系统控制所有的物理资源,包括 I/O 设备,因此,设备驱动位于宿主机操作系统中。VMM(图中的虚拟机管理内核模块)则包含了处理器虚拟化模块和内存虚拟化模块。图中的设备模型实际上也是 VMM 的一部分,在具体实现中,可以将设备模型放在用户态,也可以放在内核态中。

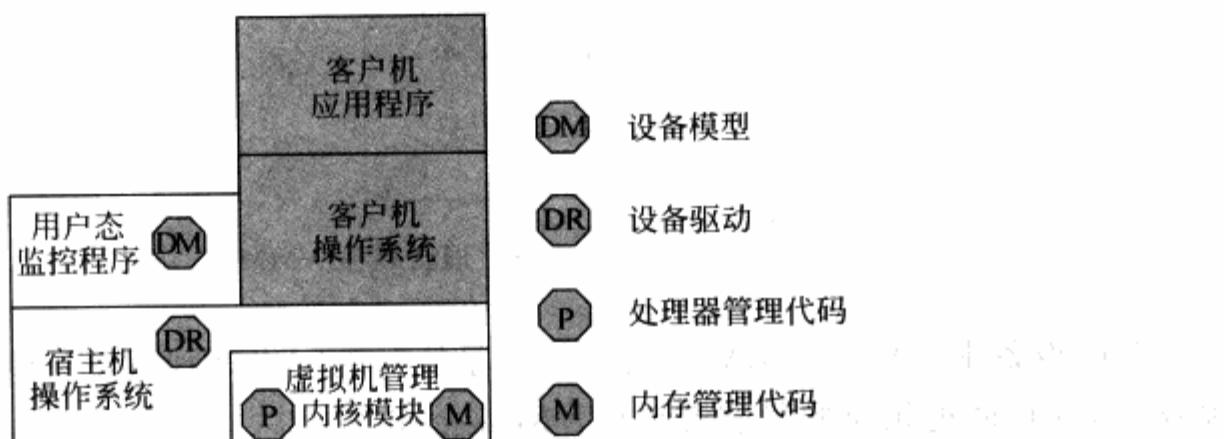


图 3-10 宿主模型 VMM

宿主模型的优缺点和 Hypervisor 模型恰好相反。宿主模型最大的优点是可以充分利用现有操作系统的设备驱动程序,VMM 无须为各类 I/O 设备重新实现驱动程序,可以专注于物理资源的虚拟化。考虑到 I/O 设备种类繁多,千变万化,I/O 设备驱动程序开发的工作量非常大,因此,这个优点意义重大。此外,宿主模型也可以利用宿主机操作系统的其他功能,例如调度和电源管理等,这些都不需要 VMM 重新实现就可以直接使用。

宿主模型当然也有缺点,由于物理资源由宿主机操作系统控制,VMM 需要调用宿主机操作系统的服务来获取资源进行虚拟化,而那些系统服务在设计开发之初并没有考虑虚拟化的支持,因此,VMM 虚拟化的效率和功能会受到一定影响。此外,在安全方面,由于 VMM 是宿主机操作系统内核的一部分,因此,如果宿主机操作系统内核是不安全的,那么,VMM 也是不安全的,相应的运行在虚拟机之上的客户机操作系统也是不安全的,相对容易被攻破。换言之,虚拟机的安全不仅依赖于 VMM 的安全,也依赖于宿主机操作系统的安全。与现有的操作系统架构相比,宿主模型在架构上并没有提高安全性。

3. 混合模型

混合模型是上述两种模式的汇合体。VMM 依然位于最底层,拥有所有的物理资源。与 Hypervisor 模式不同的是,VMM 会主动让出大部分 I/O 设备的控制权,将它们交由一个运行在特权虚拟机中的特权操作系统来控制。相应的,VMM 虚拟化的职责也被分担。处理器和内存的虚拟化依然由 VMM 来完成,而 I/O 的虚拟化则由 VMM 和特权操作系统共同合作来完成。

图 3-11 展示了混合模型的架构。I/O 设备由特权操作系统控制,因此,设备驱动模块

位于特权操作系统中。其他物理资源的管理和虚拟化由 VMM 完成,因此,处理器管理代码和内存管理代码处在 VMM 中。

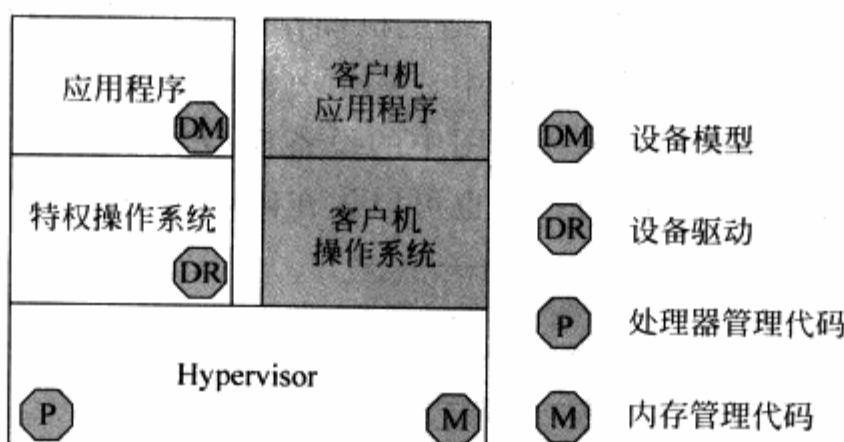


图 3-11 混合模型 VMM

I/O 设备虚拟化由 VMM 和特权操作系统共同完成,因此,设备模型模块位于特权操作系统中,并且通过相应的通信机制与 VMM 合作。

混合模型集中了上述两种模型的优点。VMM 可以利用现有操作系统的 I/O 设备驱动程序,不需要另外开发。VMM 直接控制处理器、内存等物理资源,虚拟化的效率也比较高。在安全方面,如果对特权操作系统的权限控制得当,虚拟机的安全性只依赖于 VMM。

当然,混合模型也存在缺点。由于特权操作系统运行在虚拟机上,当需要特权操作系统提供服务时,VMM 需要切换到特权操作系统,这里面就产生上下文切换的开销。当切换比较频繁时,上下文切换的开销会造成性能的明显下降。出于性能方面的考虑,很多功能还是必须在 VMM 中实现,无法借助特权操作系统,如调度程序和电源管理等。

3.7 典型虚拟化产品及其特点

虚拟化技术经过多年的发展,已经出现了很多成熟的产品,应用也从最初的服务器扩展到了桌面等更宽的领域。本节就向读者介绍几种典型的虚拟化产品及其特点。

3.7.1 VMware

VMware 是 x86 虚拟化软件的主流厂商之一。VMware 的 5 位创始人中的 3 位曾在斯坦福大学研究操作系统虚拟化,项目包括 SimOS 系统模拟器和 Disco 虚拟机监控器。1998 年,他们与另外两位创始人共同创建了 VMware 公司,总部位于美国加州 Palo Alto。VMware 在 2003 年被 EMC 收购,成为 EMC 的全资子公司。VMware 于 2007 年在纽约证交所上市。

VMware 提供一系列的虚拟化产品,产品的应用领域从服务器到桌面,产品可以运行在 Windows、Linux 和 Mac OS X 之上。此外,基于 Hypervisor 架构的 VMware ESX Server

则直接运行在物理硬件之上,无须操作系统。下面是 VMware 主要产品的简介,包括 ESX Server、VMware Server、VMware Workstation 和 VMware Fusion。

VMware ESX Server 3 是 VMware 的旗舰产品。ESX Server 基于 Hypervisor 模型,在性能和安全性方面都得到了优化,是一款面向企业级应用的产品。VMware ESX Server 支持完全虚拟化,可以运行 Windows、Linux、Solaris 和 Novell Netware 等客户机操作系统。VMware ESX Server 也支持类虚拟化,可以运行 Linux 2.6.21 以上的客户机操作系统。ESX Server 的早期版本采用软件虚拟化的方式,基于 Binary Translation 技术。在新版本 ESX Server3 中已经开始采用硬件虚拟化的技术,支持 Intel VT 技术和 AMD-V 技术。

VMware Server 之前叫 VMware GSX Server,是 VMware 面向服务器端的入门级产品。VMware Server 采用了宿主模型,宿主机操作系统可以是 Windows 或者 Linux。VMware Server 的功能与 ESX Server 类似,但是在性能和安全性上与 ESX Server 有所差距。VMware Server 也有自己的优点,由于采用了宿主模型,因此 VMware Server 支持的硬件种类要比 ESX Server 多。此外,VMware Server 是免费的。

VMware Workstation 是 VMware 面向桌面的主打产品。与 VMware Server 类似,VMware Workstation 也是基于宿主模型,宿主机操作系统可以是 Windows 或者 Linux。VMware Workstation 也支持完全虚拟化,可以运行 Windows、Linux、Solaris、Novell Netware 和 FreeBSD 等客户机操作系统。与 VMware Server 不同,VMware Workstation 专门针对桌面应用做了优化,如为虚拟机分配 USB 设备,为虚拟机显卡进行 3D 加速等。其中,3D 加速功能目前还是实验性的,今后还会进一步增强。

VMware Fusion 也是 VMware 面向桌面的一款产品,功能和 VMware Workstation 基本相同,主要区别在于 VMware Fusion 的宿主机操作系统是基于 Intel Mac 硬件平台的 Mac OS X,而 VMware Workstation 则运行在 Windows 和 Linux 上。

由于 VMware 起步比较早,因此 VMware 产品具有以下几个特点。

- (1) 功能丰富。几乎大部分的虚拟化功能都有相应的 VMware 产品对应。
- (2) 配置和使用方便。VMware 开发了非常易于使用的配置工具和用户界面。
- (3) 稳定,适合企业使用。目前,很多企业都选择了 VMware ESX Server。

3.7.2 Microsoft

微软在虚拟化产品方面起步比 VMware 晚,但是在认识到虚拟化的重要性之后,微软通过外部收购和内部开发,推出了一系列虚拟化产品,目前已经形成了比较完整的虚拟化产品线。微软的虚拟化产品涵盖了服务器虚拟化(Virtual Server, Windows Server 2008)、桌面虚拟化(Virtual PC)、应用虚拟化(SoftGrid)、表现虚拟化(Terminal Service)、存储虚拟化(Windows Storage Server)和网络虚拟化等领域。此外,微软还开发了集中式的管理工具 System Center 用于虚拟化的管理。

本书主要面向操作系统层面的虚拟化,因此,这里主要介绍操作系统虚拟化相关的产品

Virtual PC、Virtual Server 和 Windows Server 2008。

Virtual PC 是面向桌面的虚拟化产品,最早由 Connectix 公司开发,后来该产品被微软公司收购。Virtual PC 是基于宿主模型的虚拟机产品,宿主机操作系统是 Windows。早期版本也采用软件虚拟化方式,基于 Binary Translation 技术。新版本已经支持硬件虚拟化技术,支持 Intel VT 技术和 AMD-V 技术。2006 年,微软将该产品免费。

Virtual Server 是面向服务器的入门级虚拟化产品。与 Virtual PC 一样,Virtual Server 基于宿主模型,宿主机操作系统可以是 Windows XP 和 Windows Server 2003 等。Virtual Server 从 2005 R2 SP1 版本开始支持硬件虚拟化技术,包括 Intel VT 技术和 AMD-V 技术。2006 年,微软将该产品免费。作为入门级产品,有些功能目前 Virtual Server 还不支持,目前最新版的 Virtual Server 2005 R2 SP1 中,不支持 64 位的客户机操作系统,也不支持对称多处理器(SMP)的客户机操作系统。

Windows Server 2008 是微软推出的新一代服务器操作系统,其中一项重要的新功能是虚拟化功能。Server 2008 的虚拟化架构采用的是混合模型,其重要组件之一 Hyper-V 作为 Hypervisor 运行在最底层,Server 2008 本身作为特权操作系统运行在 Hyper-V 之上。Server 2008 采用硬件虚拟化技术,必须运行在支持 Intel VT 技术或者 AMD-V 技术的处理器上。Server 2008 的虚拟化功能是纯 64 位,只运行在 Server 2008 64 位版本中。Server 2008 在性能和功能上都优于 Virtual Server,功能上包括支持 32 位/64 位客户机操作系统,支持对称多处理器的客户机操作系统(一个虚拟机最多可以有 4 个虚拟处理器)等。

Microsoft 虚拟化产品的特点在于和 Windows 操作系统结合得非常好,在 Windows 下非常易于配置和使用。

3.7.3 Xen

Xen 是一款基于 GPL 授权方式的开源虚拟机软件。Xen 起源于英国剑桥大学 Ian Pratt 领导的一个研究项目,之后,Xen 独立出来成为一个社区驱动的开源软件项目。Xen 社区吸引了许多公司和科研院所的开发者加入,发展非常迅速。之后,Ian 成立了 XenSource 公司进行 Xen 的商业化应用,并且推出了基于 Xen 的产品 Xen Server。2007 年,Ctrix 公司收购了 XenSource 公司,继续推广 Xen 的商业化应用。Xen 开源项目本身则被独立到 www.xen.org,并且成立了 Xen AB(Xen Project Advisor Board,Xen 项目指导委员会)来管理 Xen 开源项目,主要工作包括管理 Xen 项目的路线图、Xen 商标的授权政策等。截至 2008 年,Xen AB 由 7 家公司组成:Ctrix、IBM、Intel、HP、Novell、Red Hat 和 Sun Microsystems。

从技术角度来说,Xen 基于混合模型,如图 3-12 所示。特权操作系统(Domain 0)可以是 Linux、Solaris 以及 NetBSD,理论上,其他操作系统也可以移植作为 Xen 的特权操作系统,例如,曾经就有开发者尝试移植过 Plan 9 操作系统。Xen 最初的虚拟化思路是类虚拟化,通过修改 Linux 的内核,实现处理器和内存的虚拟化,通过引入 I/O 的前端驱动/后端

驱动(front/backend)架构实现设备的类虚拟化。Xen 1.0 和 2.0 成功地实现了操作系统的类虚拟化,即图中的 Domain 0 和 Domain N。Xen 类虚拟化虚拟机的性能接近物理机。

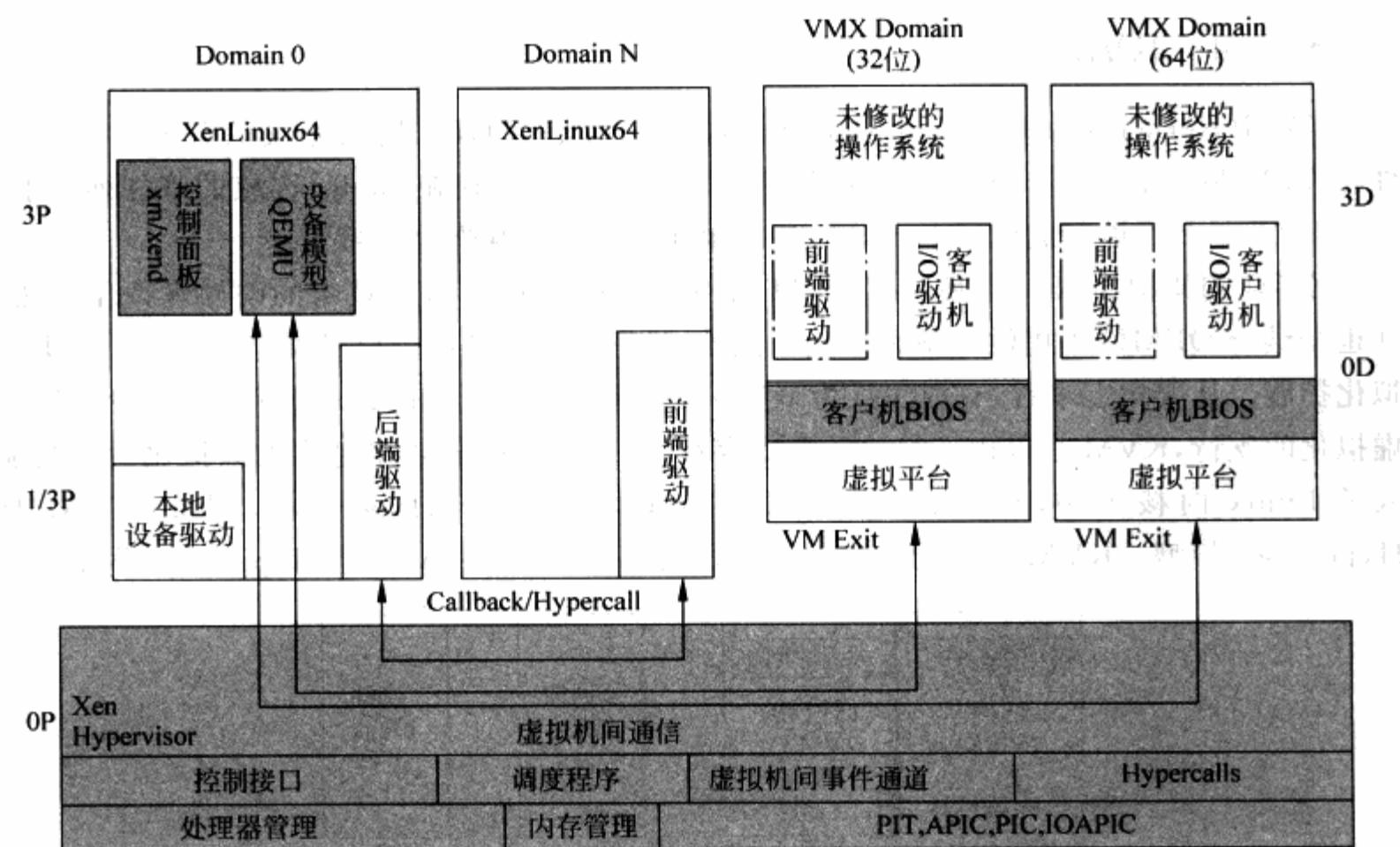


图 3-12 Xen 架构图

随着 Xen 社区的发展壮大,硬件完全虚拟化技术也被加入到 Xen 中,Xen 3.0 支持基于 Intel VT 和 AMD-V 硬件技术的完全虚拟化。图 3-12 中的 VMX Domain 是支持完全虚拟化的虚拟机。Hypervisor 通过硬件提供的功能实现处理器、内存和 I/O 的虚拟化,其中,I/O 虚拟化中的设备模型借用了另外一个开源项目 QEMU,利用 QEMU 的设备模拟代码来完成 I/O 设备的虚拟化。此外,类虚拟化中的前端驱动/后端驱动的架构也可以应用在 VMX Domain 中,用于提高 I/O 设备的性能。

Xen 支持多种硬件平台,官方的版本支持包括 x86_32、x86_64、IA64 和 PowerPC 架构。由于 Xen 是开放源代码的,Xen 也被开发者移植到其他架构上。例如,Embedded Xen 项目就将 Xen 移植到了 ARM 平台上。

Xen 目前已经比较成熟,基于 Xen 的虚拟化产品也很多,如 Citrix、VirtualIron、Redhat 和 Novell 等都有相应的产品。Xen 目前还在发展当中,一些新的技术和功能还在不断地被加入,如硬件 I/O 虚拟化技术 VT-d、SR-IOV 等。

作为开源软件,Xen 的特点如下。

(1) 可移植性非常强,开发者可以将其移植到其他平台,也可以将其修改用于项目研究等。

(2) 独特的类虚拟化支持,提供了接近于物理机的性能。

Xen 的易用性还有待加强。

3.7.4 KVM

KVM(Kernel-based Virtual Machine)也是一款基于 GPL 授权方式的开源虚拟机软件。KVM 最早由 Qumranet 公司开发,在 2006 年 10 月出现在 Linux 内核的邮件列表上,并于 2007 年 2 月被集成到了 Linux 2.6.20 内核中,成为内核的一部分。

KVM 的架构如图 3-13 所示。KVM 采用的是基于 Intel VT 技术的硬件虚拟化方法,并也是结合 QEMU 来提供设备虚拟化。此外,最近 Linux 社区中已经发布了 KVM 的类虚拟化扩展。从架构上来看,有说法认为 KVM 是宿主模型,因为 Linux 设计之初并没有针对虚拟化的支持,KVM 是以内核模块的形式存在的。但是,随着越来越多的虚拟化功能被加入了 Linux 内核当中,也有说法认为 Linux 已经是一个 Hypervisor,因此,KVM 是 Hypervisor 模型。KVM 项目的发起人和维护人倾向于认为 KVM 是 Hypervisor 模型。

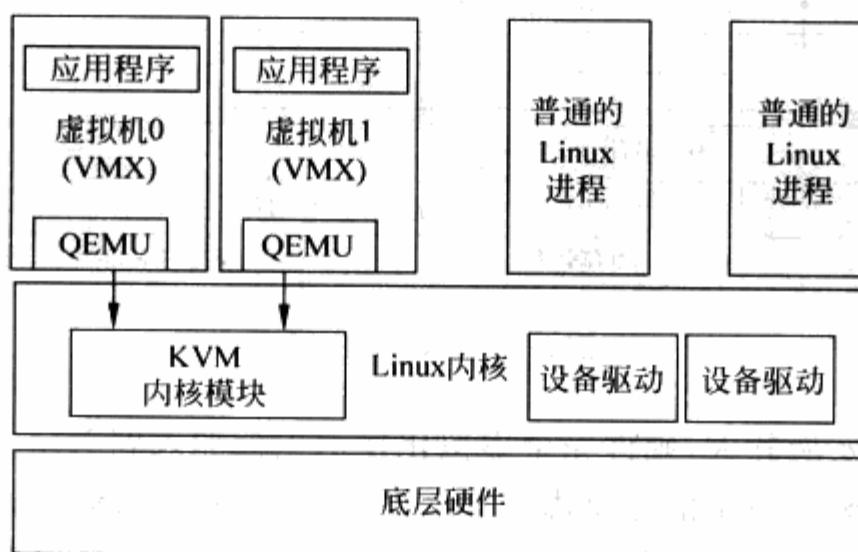


图 3-13 KVM 架构图

KVM 支持多种硬件平台,包括 IA32、IA64、S390 和 PowerPC。KVM 也可以移植到其他操作系统上,目前就有项目在将 KVM 移植到 FreeBSD 上。KVM 还处在发展阶段。目前,随着众多 Linux 内核开发者的加入,发展非常迅速。

KVM 的特点在于和 Linux 内核结合得非常好,因此 KVM 继承了 Linux 的大部分功能。当然,和 Xen 一样,作为开源软件,KVM 的移植性也很好。

3.8 思考题

(1) 如果一个处理器不是可虚拟化的结构,例如敏感指令 A 在非最高特权级别的执行不会触发异常,那么读者能否在深入后面章节之前,结合本章中的简要介绍,设想一下三种

主流的虚拟化方法在处理敏感指令 A 时的一个具体流程。

- (2) 内存分区与内存虚拟化相比较,各有什么优缺点?
- (3) 如果 VMM 希望将某个设备直接交付虚拟机使用,从资源的角度说,这个过程通常意味着哪些资源的直接交付? 直接交付的方式可能有哪些?
- (4) Hypervisor 模型、宿主模型和混合模型在 I/O 资源的管理上各不相同,请列举这三种模型在 I/O 资源管理上的不同之处,并分析这些不同对 I/O 虚拟化性能的影响。
- (5) VMM 和操作系统都有中断管理模块,请根据对操作系统的理解以及本章对 VMM 的描述,阐述 VMM 和操作系统在中断管理方面可能的相同之处与不同之处。
- (6) 根据本章对几个典型虚拟化产品的介绍,你认为未来的虚拟化产品还可以有哪些功能? 可以结合本章所介绍的产品加以说明。

卷附录

卷附录是关于本书学习过程中可能会遇到的一些问题的解答。对于一些常见的问题,本书会给出简要的解答;对于一些较为复杂的问题,则会给出一些参考文献,读者可以自行查阅。卷附录的内容包括:常见问题解答、常见故障排除、常见配置指南、常见命令行工具、常见脚本语言、常见编程语言、常见数据库系统、常见中间件系统、常见操作系统、常见网络协议、常见存储系统、常见备份恢复系统、常见虚拟化系统等。

CPU S.A.

卷附录是关于本书学习过程中可能会遇到的一些问题的解答。对于一些常见的问题,本书会给出简要的解答;对于一些较为复杂的问题,则会给出一些参考文献,读者可以自行查阅。

基于软件的完全虚拟化

4.1 概述

由于硬件体系结构在虚拟化方面设计存在缺陷,导致了第 3 章所指出的虚拟化漏洞,系统虚拟化因此不能直接而有效地实现。为了弥补虚拟化漏洞,在硬件还未提供足够的支持之前,基于软件的虚拟化技术就已经先给出了两种可行的解决方案:模拟执行和直接源代码改写。这里,模拟执行对应的就是基于软件的完全虚拟化技术,而直接源代码改写对应的是类虚拟化技术,将来第 6 章给予介绍。

我们知道,所有的虚拟化形式都可以用模拟来实现。解释执行就是最简单最直接的模拟实现方式,取一条指令出来,模拟出这条指令执行的效果,再继续取下一条指令,从某种程度上解决了陷入再模拟,也避免了虚拟化漏洞。模拟技术通常可以被用在不同体系结构的虚拟化中,也就是在一种硬件体系结构上模拟出另一种不同硬件体系结构的运行环境。而在同一种体系结构的模拟中,情况会变得更容易一些,因为大多数指令是可以不需要被模拟执行而直接放在真实的硬件上执行,于是一条指令再一条指令地解释执行在这里就没有必要了。可以采用改进的代码扫描与修补(Scan-and-patch)技术和二进制代码翻译技术,尽可能地提高虚拟化的性能。

本章将主要介绍借助于模拟执行的软件完全虚拟化。首先介绍基于软件的 CPU 虚拟化原理和实现,包括解释执行、扫描与修补以及二进制代码翻译,主要基于二进制代码翻译介绍 CPU 虚拟化是如何实现的;其次介绍基于影子页表的内存虚拟化技术;最后介绍 I/O 虚拟化。

4.2 CPU 虚拟化

对于传统虚拟化漏洞而言,在硬件设计对此问题进行改进之前,一些模拟技术就已经先被使用来弥补这个漏洞,提供平台虚拟化的能力。可以说,基于软件的 CPU 完全虚拟化,

其本质就是软件模拟。所有虚拟化的形式都可以用模拟来实现,模拟的强大之处在于,VMM 可以将虚拟机的整个执行过程置于控制中,VMM 执行每一条指令都有时机进行模拟,因而不会漏过需要模拟的敏感指令。

模拟技术早在现代虚拟化诞生之前就存在了。使用模拟器,人们可以在一种平台上运行另一种平台的应用程序或者操作系统,例如 DEC 开发的 FX! 32 能够在 ALPHA 平台上运行为 x86 平台编译的应用程序。图 4-1 就是一个模拟器架构图。

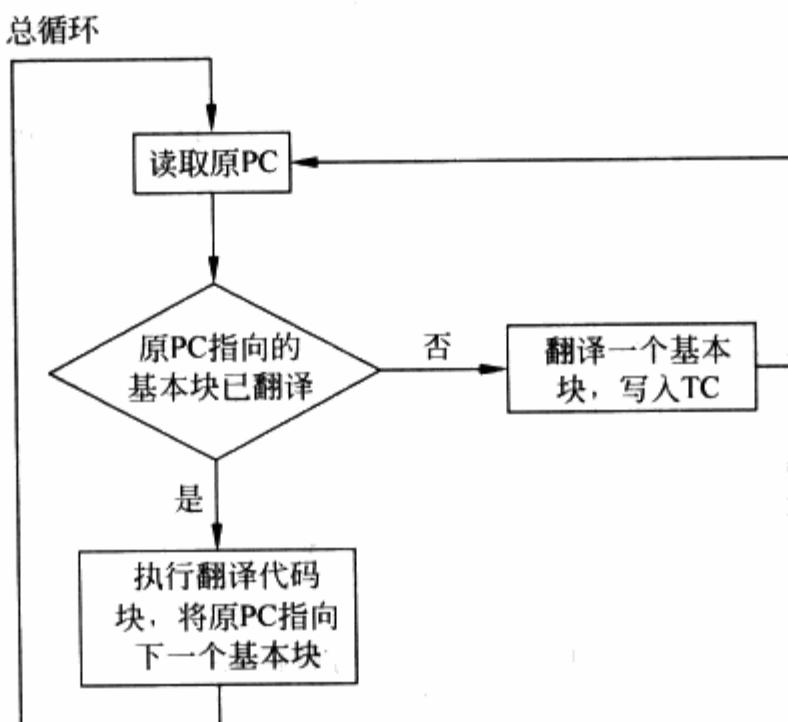


图 4-1 模拟器架构

模拟技术不仅能够用于应用程序级模拟,而且可以用于系统级模拟。它既能够用于不同硬件体系结构间的模拟,更可以用于相同硬件体系结构的模拟,只不过在相同硬件体系结构下模拟,情况变得比不同硬件体系结构下的模拟简单一些,这使得产生一些改进技术以提高虚拟化的性能。

4.2.1 解释执行

在模拟技术中,最简单最直接的模拟技术是解释执行,即取一条指令,模拟出这条指令执行的效果,再继续取下一条指令,周而复始。由于是一条一条取指令而不会漏过每一条指令,在某种程度上即每条指令都“陷入”了,所以解决了陷入再模拟的问题,进而避免了虚拟化漏洞。这种方法不仅适用于模拟与物理机相同体系结构的虚拟机,而且也适用于模拟与物理机不同体系结构的虚拟机。

图 4-2(a)所示为代码以正常执行的方式运行,图 4-2(b)所示为虚拟机的代码以解释执行的方式运行。图中灰色部分表示会被载入物理 CPU 执行的代码,白色部分表示不会被载入物理 CPU 执行的代码。正常执行的方式就是最常见的直接在物理 CPU 上运行编译好的代码;而在解释执行方式中,编译好的二进制代码是不会被载入物理 CPU 直接运行

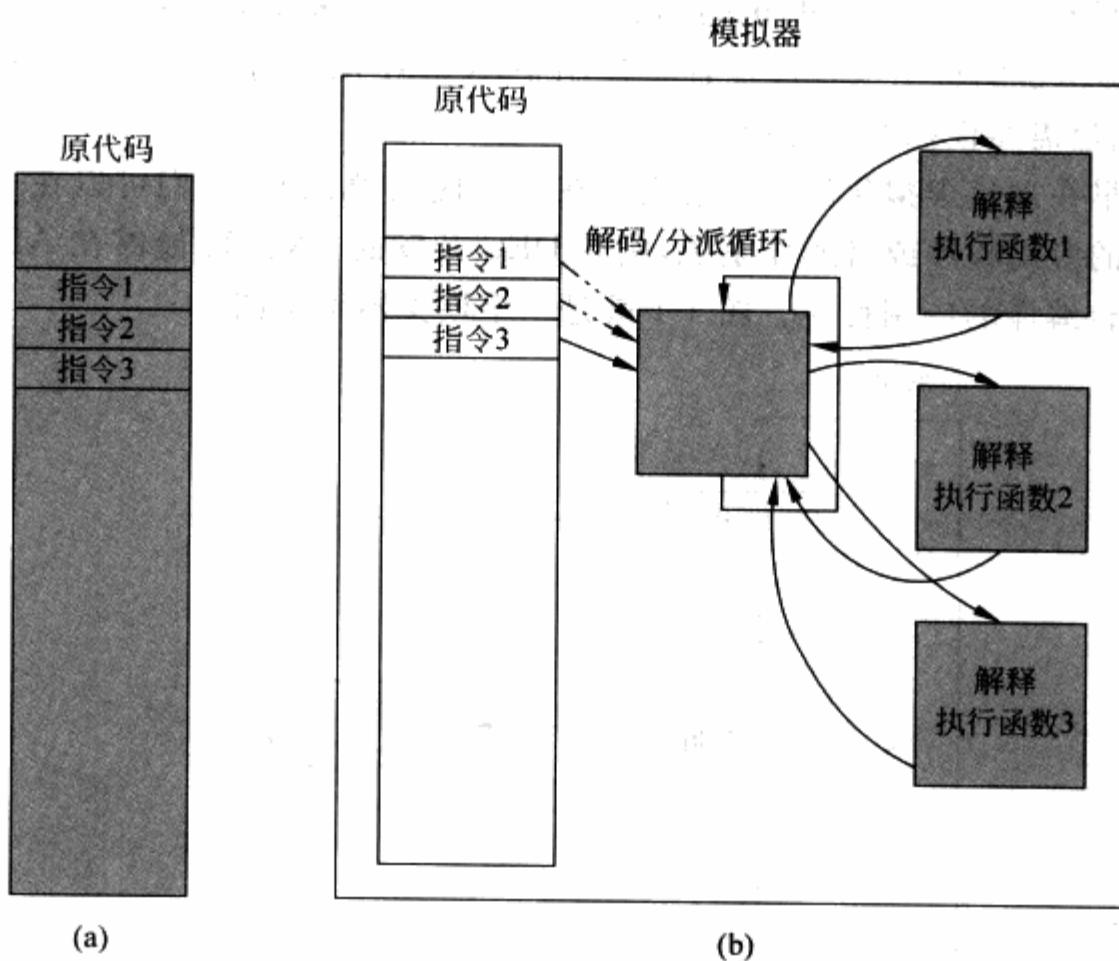


图 4-2 正常执行与解释执行

的,而是由解释器逐条解码,再调用对应的函数来模拟对应指令的功能。

虽然这种方法保证了所有指令执行受到 VMM 的监视控制,然而它对每条指令不区别对待,其最大的缺点就是性能太差。由于这里所说的虚拟化前提是模拟与物理机相同体系结构的虚拟机,那么至少有很多非敏感指令就不需要模拟而可以直接在物理 CPU 上运行,这便诞生了以下两种改进技术。

4.2.2 扫描与修补

由于解释执行有很大的性能损失,加上虚拟机中模拟的 CPU 和物理 CPU 的体系结构是相同的,这样大多数指令可以被映射到物理 CPU 上直接运行,因此,CPU 虚拟化过程中可以采用更优化的模拟技术来弥补虚拟化漏洞。

扫描与修补技术通过这样的方式,让大多数的指令直接在物理 CPU 上运行,而把操作系统代码中的敏感指令替换为跳转指令或会陷入到 VMM 中去的指令,使其一旦运行到敏感指令处控制流就会进入 VMM 中,由 VMM 代为模拟执行。

扫描与修补技术的流程如下。

- (1) VMM 会在虚拟机开始执行每段代码之前对其进行扫描,解析每一条指令,查找到特权指令和敏感指令。
- (2) 补丁代码会在 VMM 中动态生成,通常每一个需要修补的指令会对应一块补丁

代码。

(3) 敏感指令被替换成一个外跳转,从虚拟机跳转到 VMM 的空间里,在 VMM 中执行动态生成的补丁代码。

(4) 当补丁代码执行完后,执行流再跳转回虚拟机中的下一条代码继续执行。

需要注意的一点是,在补丁比被修补的指令长时,需要使用更巧妙的方法来完成修补。例如,在 x86-32 体系结构中,一个外跳转指令占 5 个字节,比有些特权或敏感指令长。可行的一个解决方法是使用更短的能够引起陷入的指令,例如 INT 3 指令等。在陷入后,VMM 由陷入发生的地址查表找出对应的原指令,然后进行模拟。与外跳转不同,陷入会引起特权级切换,因而性能开销更大。

图 4-3 是一个从 VirtualBox 的实际代码中提取出来的补丁代码例子,它对应的是 Intel IA32 关闭中断指令 CLI,其中一些非相关的代码已经略去。

mov dword[ss: PATM_INTERRUPTFLAG], 0	标记正在进入临界区
pushf	保存标志寄存器的内容
and dword[ss: PATM_VMFLAGS], ~X86_EFL_IF	清除虚拟 EFLAGS 中的 IF 位,关中断
popf	恢复标志寄存器的内容
mov dword[ss: PATM_INTERRUPTFLAG], 1	标记退出临界区
DB 0xE9	单字节 0xE9,无条件跳转 JMP
DD PATM_JUMPDELT A	4 字节占位符,在补丁生成时被替换为跳转
目标地址	

图 4-3 CLI 指令的补丁代码

可以把这段补丁代码看作一个模板,其中以 PATM 开头的标签都会在补丁代码生成时被替换成相应的变量的地址或者值。整个补丁代码含义如下。

(1) 第 1 行的指令是将一个变量赋值为 0。PATM_INTERRUPTFLAG 标签在补丁代码生成时会被替换为客户机上下文结构中的一个变量 fPIF 的地址。客户机上下文结构被保存在 VMM 的内存中,段选择符 SS 在这里的作用是让客户机能够访问 VMM 的地址空间。这条指令的作用是告诉 VMM 当前正在执行到生成的补丁代码,在这个临界区,发生异常是危险的。

(2) 第 2 行指令将 EFLAGS 寄存器的低 16 位内容保存在栈上。

(3) 第 3 行真正执行关中操作,但关中的效果也是通过修改客户机上下文中的一个变量来实现的。标志符 PATM_VMFLAGS 会被动态地替换为这个变量的地址。

(4) 第 4 行栈上保存的内容会被恢复回 EFLAGS 寄存器。第 2 行和第 4 行的目的是为了避免临界区中间的运算指令改变 EFLAGS 中的标志位。

(5) 第 5 行通过给 fPIF 赋值为 1 来标记退出临界区。

(6) 第 6 行是一个伪代码,0xE09 是 jmp 指令的机器码。

(7) 第 7 行是一个 4 字节的占位符。PATM_JUMPDELT A 在补丁代码生成时会被替

换为虚拟机中被打补丁的指令的下一条指令。如果下一条指令也是需要打补丁的,那么会将两个补丁代码块串联起来,从而减少了两次控制流在虚拟机和 VMM 之间的转移。

扫描与修补原理示意图如图 4-4 所示。在图 4-4 里,灰色部分仍然表示会被载入物理 CPU 执行的代码,白色部分仍然表示不会被载入物理 CPU 执行的代码。除了一些敏感指令被 VMM 替换成了外跳转外,其他指令都能够直接被物理 CPU 载入运行。对于那些被打上补丁的地方,外跳转将执行流转到了对应的补丁代码块,从而模拟该指令的功能。执行监控模块负责动态地对将要执行的原代码块进行扫描,找到需要打补丁的地方打补丁,并生成相应的补丁代码块。

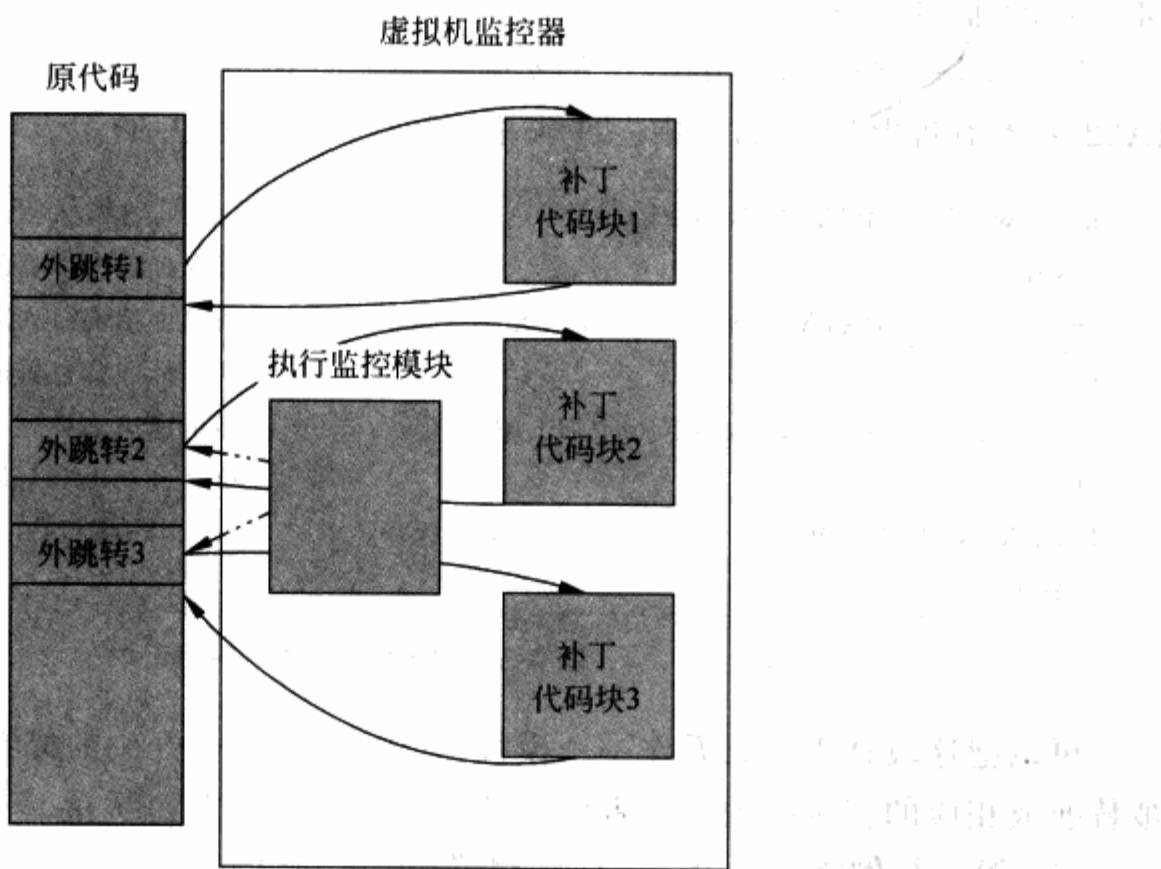


图 4-4 扫描与修补

值得一提的是,补丁代码块存放在 VMM 内存空间的代码缓存中。由于缓存的容量是有限的,所以随着虚拟机的运行,缓存会被填满,有一些补丁代码块可能会被逐出缓存。所以,VMM 中会记录一个 PC 到补丁代码块的对应关系(下面称 PC-补丁代码对)。当补丁代码块生成的时候,VMM 会记录下这个 PC-补丁代码对;当补丁代码被逐出缓存时,这个 PC-补丁代码对也会从相应记录中删除。这样,VMM 只需要查找记录就能够知道哪些 PC 对应的指令已经生成过补丁代码了,并且这些补丁代码块现在还存在代码缓存中。

在扫描与修补技术中,异常的处理也相对比较简单。由于指令是被一条一条打补丁的,原代码块相对的位置没有改变,因此,发生异常时可以很方便地找到异常指令对应的 PC,然后将这个异常交给客户机操作系统处理就可以了。

扫描与修补技术实现相对简单,在扫描与修补技术中,大多数客户机操作系统和用户代

码可以直接在物理 CPU 上运行,其性能损失也相对较小。当然,扫描与修补技术也有其缺点。

- (1) 由于特权指令和敏感指令都被模拟执行,各条指令的模拟执行时间可能会很短,但也可能会很长。
- (2) 由于每个补丁都引入了额外的跳转,这些跳转会降低代码的局部性。
- (3) 由于扫描与修补技术直接在虚拟机内存中进行代码修补,其须维护一份与补丁对应的原始代码的备份,以便在需要时将代码恢复原状。

4.2.3 二进制代码翻译

为了更好地提高性能,更为复杂的代码缓冲区技术也被用到了模拟技术中。二进制代码翻译技术(BT 技术)在 VMM 中开辟一块代码缓存,将代码翻译好放在其中。这样,客户机操作系统代码并不会直接被物理 CPU 执行,所有要被执行的代码都会在代码缓存中。相比较而言,BT 技术最为复杂,其在性能上同扫描与修补技术各有长短。

1. 基本概念

在详细介绍 BT 技术之前,先介绍几个基本概念。

在编译理论中,基本块是一个很重要的概念,它表示只有一个人口和一个出口的代码块,即这块代码只能从头进入,从尾退出。既不会有外界跳转进入到代码块中间的某个地方,也不会有代码块中间某个地方有外界跳转跳出该代码块。这里,基本块可以认为是静态基本块。

BT 技术的动态翻译也是以基本块为单位的,我们称之为动态基本块。与编译器不同的是,编译器在静态能够得到的源代码信息是不包含在编译生成的二进制代码中的,因而在运行时是无法获得这种源代码信息的。例如,源代码中的跳转标签在基本块分析会被作为划分基本块的分界,因为标签所在位置是一个可能的调整入口。但是,在动态运行时,二进制代码中是不包含这种信息的,所以,动态划分基本块时能够准确找到出口,但会遗漏一些人口。基于这个原因,动态基本块可能会比静态基本块要大一些。

例如,Fedora 6 系统在 QEMU-0.9.1 上启动时,整个过程进行了 1 322 514 次基本块翻译,在 VMware 上启动和关闭 64 位的 Windows XP 专业版共翻译了 259 936 个基本块。

BT 技术将源代码以基本块为粒度翻译代码,模拟器动态地、按需要地读入二进制代码进行翻译,将翻译好的目标代码存放在模拟器开辟的内存空间中,这块空间被称为代码缓存(translation cache)。这与扫描与修补技术的代码缓存概念是类似的。同样,由于代码缓存是在模拟器的内存空间分配的,因此其容量是有限的,在代码缓存用满的时候,部分缓存就需要被释放出来,因此,一个好的管理策略是很重要的。

源代码中的指令与翻译后的代码用某种映射关系联系起来,例如,最常用的是哈希表,即由源代码的 PC 值通过哈希函数计算查表得到其在代码缓存区中的位置。如果一个 PC 没有找到对应的表项,表示这块代码还未被翻译,或者在释放缓存空间时已被清理。

最后,介绍一下什么是翻译。模拟器对于读入的二进制代码不作限制,它们可以是应用程序代码,也可以是操作系统内核代码。读入的二进制代码可能包含所有的 x86 体系结构的指令,模拟器将其翻译输出为 x86 指令的一个安全的子集,即其中不包含特权指令和敏感指令,能够运行在用户态。

在原体系结构和目标体系结构相同的情况下,模拟器翻译方法大致可以分为两种:简单翻译和等值翻译。简单翻译比较直接,但指令数量会大大膨胀;等值翻译相对更为高效,但动态分析比前者困难。例如,QEMU 使用的是一种简单指令模板来进行翻译。图 4-5 给出了一个例子,其目的是用加法指令将寄存器 ECX 和 EDX 相加并存在 EDX 中。

```
add %ECX, %EDX  
翻译成  
mov REGS->ECX, temp1  
add temp1, REGS->EDX
```

图 4-5 简单翻译加法指令

经过简单翻译之后,可以看到,REGS 结构是模拟器中为每个虚拟 CPU 维护的一个数据结构,存有虚拟 CPU 所有寄存器的值,即相当于包含所有虚拟寄存器。在目标代码生成时,上面 REGS 会被替换成这个数据结构中内存中的地址,而 temp1 会用一个寄存器替换。

在同硬件体系结构的模拟中,很多指令是可以等值翻译的,即原代码和目标代码是一样的。理论上来讲,大多数指令是可以等值翻译的,除了以下几种例外。

(1) PC 相对寻址的指令。这类指令的寻址与 PC 相关,但由于原代码和目标代码的指令相对关系是不同的,因此不能直接使用。模拟器的翻译模块需要在目标代码中插入一些补偿代码来确保寻址的正确。这类翻译会导致目标代码少量增大,因而引起一些性能损失。

(2) 直接控制转换。原代码中的控制转换,例如函数调用和跳转指令,其目标地址需要被替换成存于代码缓存的目标代码地址。其中,直接调用和直接跳转可以被直接替换成代码缓存中的目标地址,因为它们是固定的,其引起的性能损失是可忽略的。

(3) 间接控制转换。间接调用、返回和间接跳转的目标地址是动态运行时得到的。由于目标地址不固定,代码翻译时就无法绑定跳转目标。跳转目标通常在动态时计算出来,例如通过查询哈希表。根据运行程序的不同,这种翻译的性能损失也不同,但通常在百分之几以内。

(4) 特权指令。对于特权指令的翻译分两种。对于简单的能够就地模拟的指令,例如 CLI,翻译的代码通常只要简单地设置一下模拟器中的某个标志位就可以完成对应效果了,例如 vcpu.flags.IF=0。而对于稍复杂的指令,需要做的就是用跳转从模拟环境跳到模拟器中进行深度模拟,而且这个动作会引起比较大的性能开销。

这里再提一下，在同体系结构下，等值翻译的一个潜在前提是虚拟机执行的代码可能会用到所有 CPU 寄存器。因而，在从模拟器运行时环境和虚拟机之间切换时，所有寄存器的内容都需要有一次切换。为了让虚拟机能够从模拟环境中跳到模拟器的环境，虚拟机需要用一个寄存器来存放跳转的目标地址，这个寄存器可以是暂时不再被使用的寄存器，也可以把一个寄存器的值临时保存到栈上以腾出空间。

2. 基于 BT 技术的 CPU 虚拟化

下面介绍 BT 技术如何被运用在 CPU 虚拟化中进行软件模拟的。本节以 QEMU 为例来说明。首先需要声明的是，在 QEMU 中，它为每个虚拟 CPU 都维护了一个数据结构 ENV，它保存的是当前虚拟 CPU 的运行环境，包括各种寄存器的参数和值。

图 4-6 所示的是 QEMU 翻译一个 Linux 代码基本块的过程。图 4-6 中的 4 条指令是一个基本块，是 QEMU 通过反汇编原代码，解码得到的 x86 指令。然后，QEMU 逐条指令地套用翻译模板，将其变成中间形式。在对中间形式的伪指令进行一些优化以后，QEMU 最终将其生成目标指令。

IN: 0xc075a4f8: mov \$0x1,%eax 0xc075a4fd: cpuid 0xc075a4ff: and \$0x2,%dh 0xc075a502: jne 0xc075a50d	OP: 0x0000: movl_T0_im 0x1 0x0001: movl_EAX_T0 0x0002: cpuid 0x0003: movl_T1_im 0x2 0x0004: movh_T0_EDX 0x0005: andl_T0_T1 0x0006: movh_EDX_T0 0x0007: update_1_cc 0x0008: set_cc_op 0x16 0x0009: jz_subb 0x0 0x000a: goto_tb0 0x000b: movl_eip_im 0xc075a50d 0x000c: movl_T0_im 0x837b8bc 0x000d: exit_tb 0x000e: goto_tb1 0x000f: movl_eip_im 0xc075a504 0x0010: movl_T0_im 0x837b8bd 0x0011: exit_tb 0x0012: end	OUT: [size=88] 0x08cbef40: mov \$0x1,%ebx 0x08cbef45: mov %ebx,0x0(%ebp) 0x08cbef48: call 0x80f2428 0x08cbef4d: mov \$0x2,%esi 0x08cbef52: mov 0x8(%ebp),%eax 0x08cbef55: mov %eax,%ebx 0x08cbef57: shr \$0x8,%ebx 0x08cbef5a: and %esi,%ebx 0x08cbef5c: mov %bl,0x9(%ebp) 0x08cbef5f: mov %ebx,0x2c(%ebp) 0x08cbef62: movl \$0x16,0x30(%ebp) 0x08cbef69: cmpb \$0x0,0x2c(%ebp) 0x08cbef6d: jne 0x8cbef74 0x08cbef6f: jmp 0x8cbef86 0x08cbef74: jmp 0xa789cc7 0x08cbef79: movl \$0xc075a50d,0x20(%ebp) 0x08cbef80: mov \$0x837b8bc,%ebx 0x08cbef85: ret 0x08cbef86: jmp 0xa789cc7 0x08cbef8b: movl \$0xc075a504,0x20(%ebp) 0x08cbef92: mov \$0x837b8bd,%ebx 0x08cbef97: ret
--	---	--

(a) 4条指令

(b) 反汇编

(c) 生成目标指令

图 4-6 QEMU 的一个基本块翻译

其中，敏感指令 CPUID 翻译后在目标代码中生成了一个函数调用。这个函数是 QEMU 在用户控件的一个辅助函数 helper_cpuid。它做的事情就是根据虚拟 CPU 的配置，将返回信息填写好，模拟出 CPUID 指令的执行效果。整个过程在用户态就能完成。

对于如 INT \$0x80 这样的系统调用，是不能在一个 QEMU 的辅助函数中完成模拟的，它的翻译过程略有不同。在虚拟机启动时，初始化 IDT 表的方法不会直接修改到硬件

的 IDT 表,而是会修改 ENV 结构中虚拟的 IDT 数据结构。这个数据结构会被 QEMU 用于查找虚拟机操作系统的中断或异常处理函数的入口,以及其权限设置等。如图 4-7 所示,输入代码块中的 INT \$0x80 指令被翻译为两条指令。一条是将发生中断的 EIP 保存在 ENV 环境变量中,然后调用 raise_interrupt 函数,并传入两个参数,前一个参数 0x80 指示的是当前中断的中断号,后一个参数 0x2 表示的是 INT 指令的长度。QEMU 能够用这个值计算出下一条指令所在的 EIP。在输出代码块中,中间形式的伪代码被逐条翻译成 x86 指令,寄存器和虚拟地址也都在这步被分配和确定。

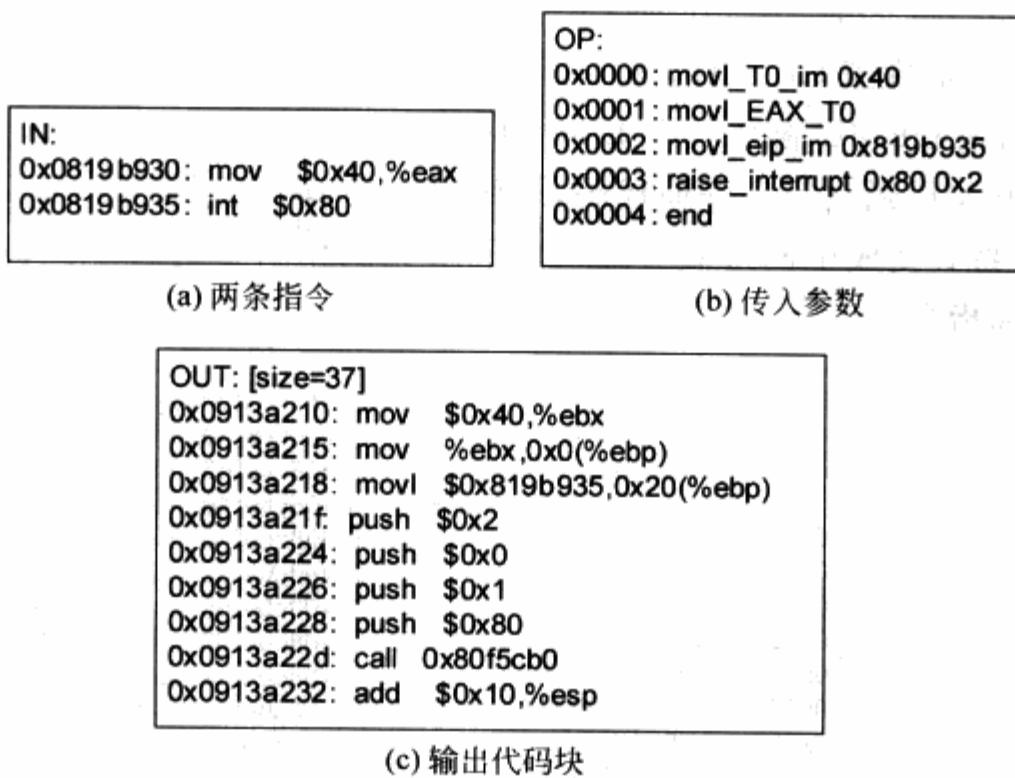


图 4-7 INT 80 的翻译

raise_interrupt 函数做的事情是使得 QEMU 从主运行循环跳出,并向虚拟机的操作系统传播中断。QEMU 首先将当前执行的 EIP 和寄存器等状态保存在 ENV 结构中,然后在 ENV 结构中找到系统启动时记录下的 IDT 表的值,从中得到系统调用的中断描述符。通过一些保护性检查后,QEMU 将当前 EIP 指向系统调用处理函数的入口,并装载虚拟机内核的代码/数据段,然后返回主循环继续执行。这样,执行就转入到虚拟机的内核,开始系统调用的处理。

与之相对的,在系统调用处理结束以后,中断返回指令会执行相反的操作,即载入用户态的代码/数据段,恢复用户态的寄存器的值,返回到中断指令的下一条指令继续运行。

除了系统调用外,其他虚拟机主动地陷入也是类似处理的,例如 x86 的 INT 3 和 into 等。

对于异常(Fault)和外部中断的处理和系统调用比较类似。不同的是,QEMU 从宿主机得到中断和异常的信号。例如,缺页异常是先由宿主机收到并处理的,宿主机会通过发送信号将异常通知 QEMU 进程。QEMU 进程的执行被打断,转而执行信号处理函数。信号

处理函数会用类似方法将中断或异常向上传播给客户机操作系统。另外一个不同是,在结束外部中断或异常的处理后,QEMU 返回到用户态被中断的那条指令继续执行,而不是被中断的指令的下一条指令。

可以总结一下,BT 技术在 VMM 中开辟一块代码缓存,将代码翻译好放在其中。原始的客户机操作系统代码并不会直接被物理 CPU 执行,它们以基本块的形式组织,模拟器先将即将执行的基本块翻译成目标代码块,再转入目标代码块执行,再翻译接下来要运行的原始基本块,如图 4-8 所示。

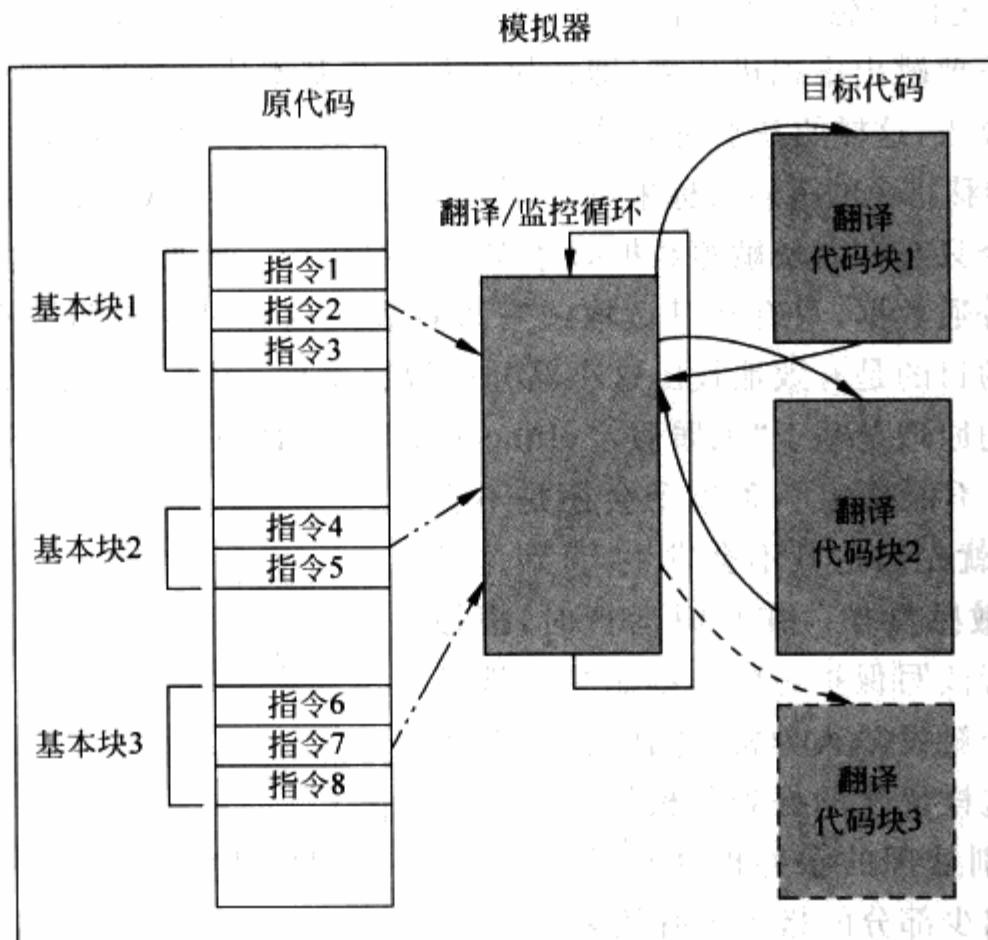


图 4-8 二进制代码翻译

3. BT 技术的难点

对于以下几种情况,BT 技术在处理过程中会遇到困难,需要特别处理。

(1) **自修改代码(Self Modifying Code)**。自修改代码值程序会修改自身代码段的内容。一旦发生自修改操作,模拟器需要将代码缓存中对应已翻译的代码清除掉,对新写的代码重新翻译。

(2) **自参考代码(Self Referential Code)**。指的是程序会从自己代码段中读取内容。在这种情况下,模拟器需要让程序读取原代码段的内容而不是代码缓冲区的内容。

(3) **精确异常(Precise Exceptions)**。精确异常指在翻译代码执行中发生了中断或异常,这时需要将运行状态对应到原代码执行到异常点时的状态,然后交给客户机操作系统去处理。精确异常问题对于 BT 技术来讲比较难解决,这主要是由于翻译的代码和原代码已

经失去了逐条对应的关系。一个可能的解决方法就是在发生异常时,模拟器回滚到基本块的开头,然后用解释执行的方式逐条执行原代码。

(4) 实时代码。对于实时性要求较高的代码,运行在模拟环境下会损失时间精确性。这个问题尚未得到很好的解决。

4. BT 技术的优化

在 BT 技术发展的时候,也积累了许多优化技术,可以提高整体性能。这些优化技术有基本块串联、自适应翻译和指令缓存布局优化等。

BT 技术的优化首先想到的就是减少模拟器环境和虚拟机环境的切换,一个方法就是使得运行尽可能不要跳出虚拟机环境,即让执行从一个基本块直接跳转到下一个基本块,而不需要模拟器的介入,这样的基本块串联起来以后就形成了超级块。基本块的串联可以通过修改直接控制流转移指令的跳转目标来完成,例如固定目标地址的 CALL 和 JMP 指令等。

一些敏感指令只有在涉及敏感数据时才需要模拟执行。例如,大多数时候 mov 指令只是在读取或写入普通数据,只有在其读取或写入页表页内容时才需要被模拟执行。BT 技术用自适应翻译的目的是有效地找出这小部分的敏感操作,而不影响敏感指令的非敏感操作。自适应翻译的原理是基于“无罪假定 (Innocent Until Proven Guilty)”。无罪假定的意思是,如果模拟器不能确定一条指令会还是不会进行敏感操作,那就先假定其不会发生,直到这条指令确实发生了敏感操作。在平时运行时,模拟器对敏感数据进行读写保护,这样,敏感指令涉及到这些数据时就会触发陷入异常,进而,模拟器才特别对其进行处理。这样优化的好处是模拟器不需要事先知道需要特别处理的敏感指令有哪些,这样 的方法对于筛选出少部分的指令很有效果。

合理地放置代码缓存能够加强执行时 CPU 中指令缓存的局部性,这一点对于性能优化有很大帮助。有时,一些虚拟执行的计算密集型程序会出现性能好于原代码执行,其原因就是指令/数据缓存有更好的局部性。但是,要刻意地做这种优化,其难度是非常大的。如图 4-9 所示,存放在代码缓存中的代码相对于原代码有更好的局部性。

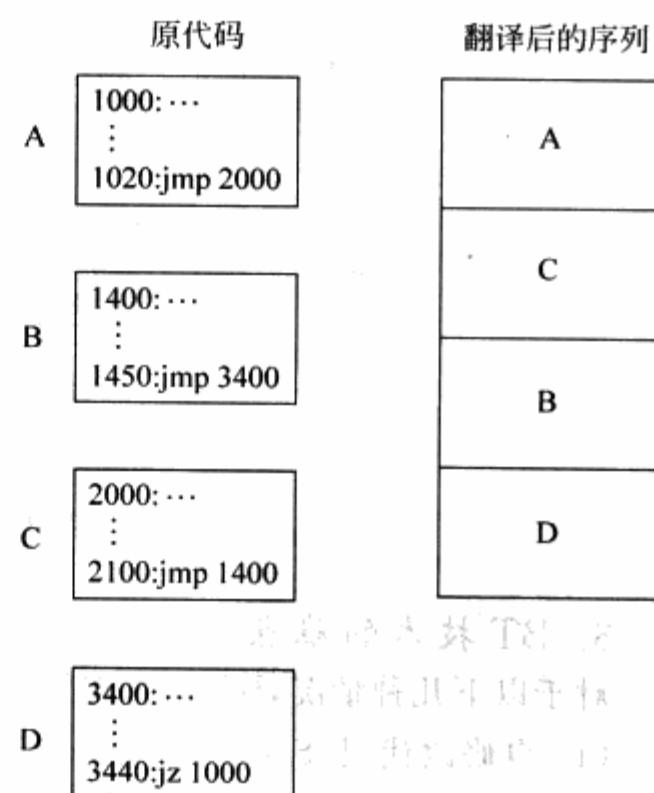


图 4-9 指令缓存布局优化

4.3 内存虚拟化

内存虚拟化的目的有如下两个。

- (1) 提供给虚拟机一个从零地址开始的连续物理内存空间。

(2) 在各虚拟机之间有效隔离、调度以及共享内存资源。

下面就来介绍一下软件完全虚拟化给出的解决方案。

4.3.1 概述

为了让客户机操作系统使用一个隔离的、从零开始且具有连续性的内存空间, VMM 引入一层新的地址空间, 即客户机物理地址空间。客户机物理地址空间是客户机操作系统所能“看见”和管理的物理地址空间, 这个地址空间不是真正的物理地址空间, 它和物理地址空间还有一层映射。有了客户机物理地址空间, 就形成了从应用程序所在的客户机虚拟地址(Guest Virtual Address, GVA)到客户机物理地址(Guest Physical Address, GPA), 再从客户机物理地址 GPA 到宿主机物理地址(Host Physical Address, HPA)的两层地址转换。前一个转换由客户机操作系统完成, 后一个转换由 VMM 负责。

为了实现从客户机物理地址 GPA 到宿主机物理地址 HPA 的地址翻译, VMM 为每个虚拟机动态地维护了一张从客户机物理地址到宿主机物理地址映射关系的表。

有了这张表之后, VMM 截获任何试图修改客户机页表或刷新 TLB 的指令, 根据这张表, 将修改从客户机虚拟地址到客户机物理地址映射的操作, 变成修改客户机虚拟地址到相应的宿主机物理地址映射的操作。

自从有了这张表之后, 虽然宿主机物理地址只有一个零起始地址, 但在不同客户机物理地址空间里, 可以各有一个零起始地址, 而且客户机操作系统看来连续的客户机物理内存空间, 其对应的宿主机物理内存空间可能是不连续的, 而这增加了 VMM 为多个虚拟机分配宿主机物理内存时的灵活性, 提高了宿主机物理内存的利用率。

VMM 还可以通过该表确保运行于同一宿主机上的不同客户机访问的是不同的物理内存, 即相同的客户机物理地址被映射到了不同的宿主机物理地址上。这样一来, 一个客户机只能访问 VMM 通过该表设置分配给它的宿主机物理内存, 而不能访问其他客户机拥有的宿主机物理内存。

有时, VMM 使用页共享技术以写时复制(Copy On Write)的方式让不同的客户机可以共享包含相同数据的宿主机物理页, 删除多余的页备份。这种页共享技术, 是通过将不同客户机的某些客户机物理地址映射到相同的宿主机物理地址上, 来实现共享这个宿主机物理地址对应的宿主机物理页。

除此之外, VMM 可以在客户机完全不知情的情况下, 将客户机所拥有的某一客户机物理页映射到另一张新的宿主机物理页上, 甚至可以将客户机所拥有的某一客户机物理页所对应的宿主机物理页换出到硬盘上, 而客户机仍然以为它访问的客户机物理页是普通的硬件内存资源。只有当它被真正访问时, VMM 才将换出的页再次换入到宿主机内存中。

4.3.2 影子页表

客户机操作系统所维护的页表负责传统的从客户机虚拟地址 GVA 到客户机物理地址 GPA 的转换。如果 MMU 直接装载客户机操作系统所维护的页表来进行内存访问, 那么由

于页表中每项所记录的都是 GPA, 硬件无法正确通过多级页表来进行地址翻译。

针对这个问题, 影子页表(Shadow Page Table)是一个有效的解决方法。如图 4-10 所示, 一份影子页表与一份客户机操作系统的页表对应, 其作的是由 GVA 直接到 HPA 的地址翻译。

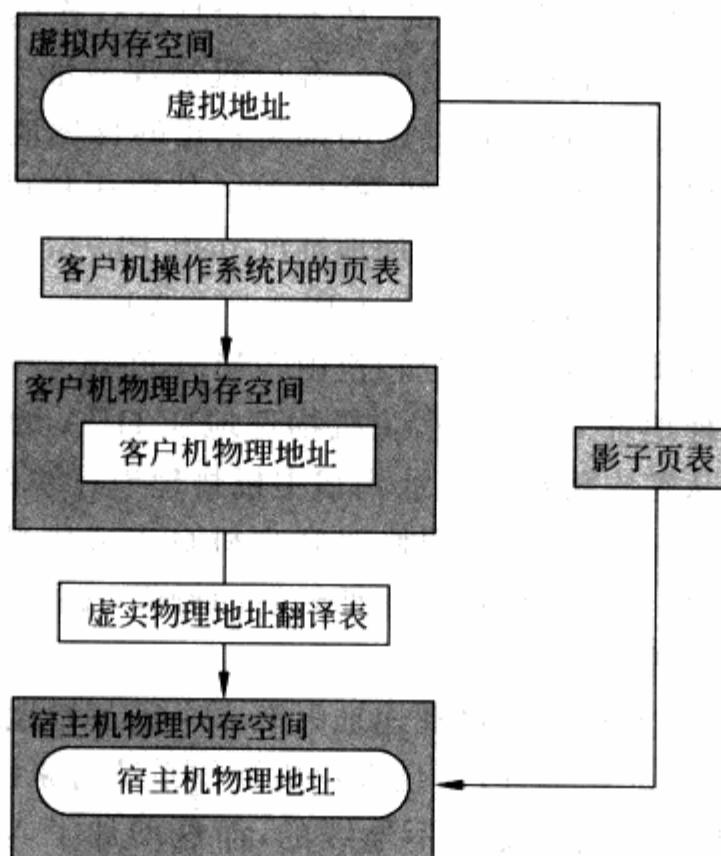


图 4-10 影子页表的作用

如图 4-11 所示, 为了使影子页表机制能够工作, VMM 需要对 MMU 实现虚拟化。客户机操作系统所能看到和操作的都是虚拟 MMU, 客户机操作系统所维护的页表只是被客户机操作系统载入到虚拟 MMU 中, 不能被物理 MMU 所直接利用来进行 MMU 硬件实现的地址翻译。因而, 真正被 VMM 载入到物理 MMU 中的页表是影子页表。

影子页表是被物理 MMU 所装载使用的页表, VMM 为每个客户机操作系统中的每一套页表都维护了一套相应的影子页表。有了影子页表, 普通的内存访问只需使用影子页表即可实现从客户机虚拟地址 GVA 到宿主机物理地址 HPA 的转换, 而不需要在每次访问内存时都进行客户机虚拟地址 GVA 到客户机物理地址 GPA 以及客户机物理地址 GPA 到宿主机物理地址 HPA 的两次转换。而且, 在 TLB 和 CPU 缓存上缓存的是来自影子页表的从客户机虚拟地址 GVA 到宿主机物理地址 HPA 的映射, 因而大大降低了额外的性能开销。

下面来看一下为什么在物理 MMU 中加载影子页表与客户机操作系统加载客户机页表等效。分页机制启动以后, 在访问存储器时若一个虚拟地址到物理地址的映射不在 TLB 缓存中, 就需要从页表基址寄存器中的物理地址所指向的物理页开始遍历页表。以 x86

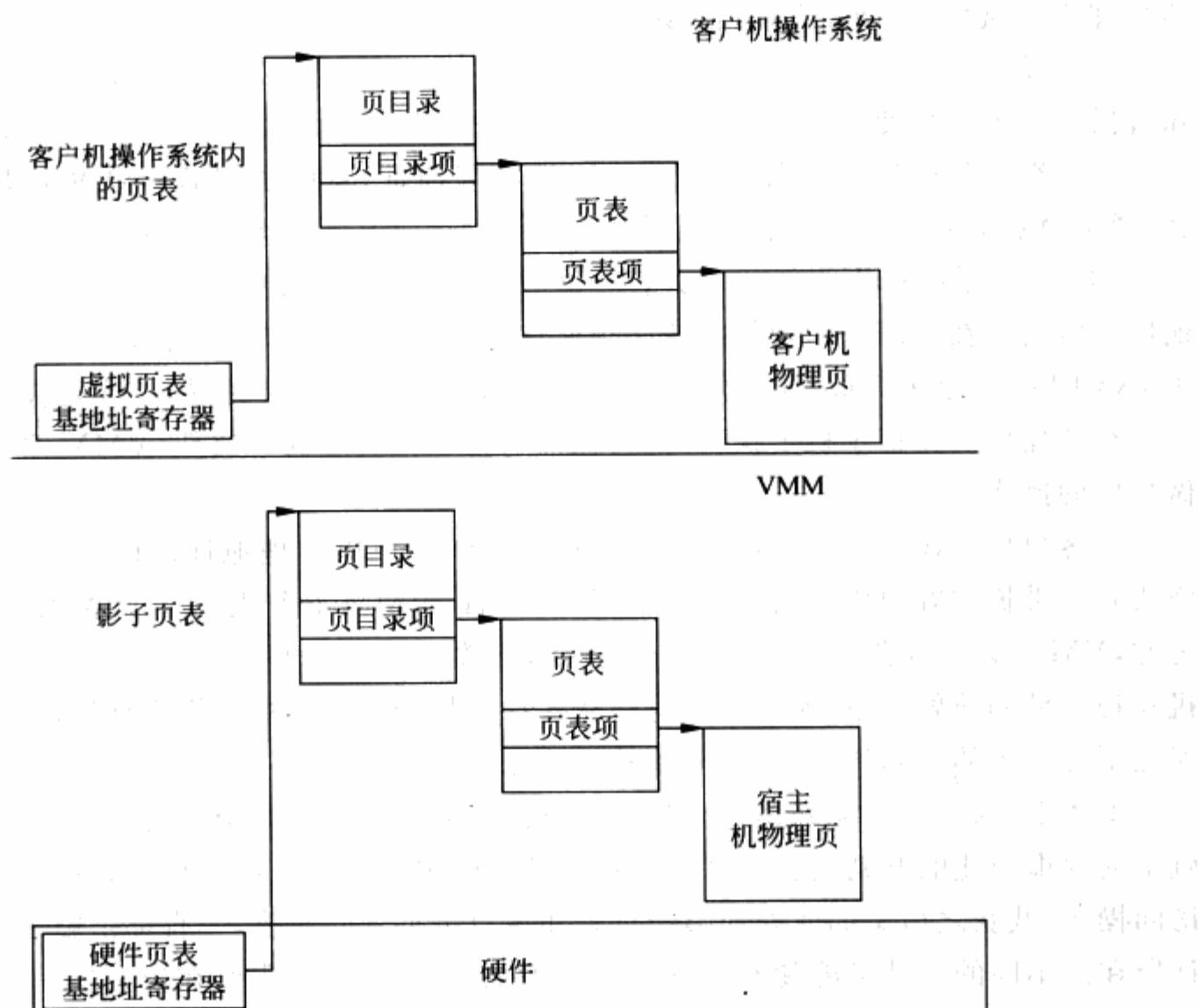


图 4-11 影子页表与客户机操作系统页表

架构的最简单的两级页表为例,CR3 寄存器所指向的是一个页目录表页,该页目录表中的每一项要么为空,要么指向一个页表页,而页表页中的每一项,要么为空,要么指向一个物理页。假设在遍历客户机页表时,客户机虚拟地址 0xc4567010 处在页目录项第 0x311 项,并处在该页目录项所指向的页表中第 0x167 项所指向的客户机物理页中偏移 0x010 字节处。那么,在宿主机上遍历影子页表时,从硬件 CR3 寄存器中的宿主机物理地址所指向的页目录表第 0x100 项指出的页表第 0x200 项指出的宿主机物理页中偏移 0x010 处的宿主机物理地址,就是该客户机虚拟地址 0xc4567010 所对应的宿主机物理地址。

实际上,在影子页表的实现中,影子页表的页表结构并不一定与客户机页表的页表结构完全一致,例如在 64 位的宿主机上,就可以运行 32 位的客户机,尽管客户机的页表结构只有两级。但这并无妨碍,必须要保证的是,相对于同一个虚拟地址,在影子页表中最后的那级页表的页表项所指向的宿主机物理页是,并且只能是客户机物理页在前文介绍的客户机物理地址与宿主机物理地址映射表中相对应的宿主机物理页。只有这样,客户机操作系统才能由影子页表访问到它想访问的客户机物理地址。这就是影子页表的基本原理。

客户机操作系统使用的页表不是静态的,客户机总在不时地修改客户机页表。在传统

的物理机上执行时,刷新 TLB 的场合大致有如下三种。

(1) 写 CR3 寄存器,若写入 CR3 寄存器的物理地址与 CR3 寄存器原来的内容相同,意味着操作系统只想使当前 TLB 的内容全部失效,要访问虚拟地址,硬件需要重新遍历页表。

(2) 若写入 CR3 寄存器的是个不同的物理地址,意味着操作系统想使用新的一套页表,通常就意味着另一个进程占用了处理器,自然 TLB 的内容也全部失效。

(3) 操作系统只修改部分的页表。这时,若原来的页表项所涉及到的虚拟地址到物理地址的转换已在 TLB 中,操作系统必须使该 TLB 项失效,即使与单个 TLB 项失效,这可以用 INVLPG 指令来完成,通常修改页表和 INVLPG 是伴随执行的。这是在传统的物理机上执行的操作。在宿主机上执行时,为保证与客户机页表的一致性,VMM 必须对影子页表做相应的操作。

当客户机操作系统修改从客户机虚拟地址到客户机物理地址的内存映射关系,即企图修改它所维护的客户机页表时,为了保证一致性,VMM 必须对影子页表也做相应的维护。为此,VMM 必须截获这样的内存访问操作,修改在影子页表中同一客户机虚拟地址到宿主机物理地址的映射关系,使之仍然符合从客户机物理地址到宿主机物理地址的映射关系,这保证了客户机的正确执行,也带来额外的性能开销。

事实上,这样的性能开销一直是客户机性能开销的热点所在。影子页表的性能开销也就是内存虚拟化的开销是影响客户机系统性能的关键因素。以何种方式截获客户机的内存访问操作,截获之后又如何处理影子页表和客户机页表的修改,一直都是影子页表的关键设计所在。不同的设计可能使整个客户机的性能相差很大。我们在较后的章节中再进一步介绍。

在前面所说的三种操作中,CR3 的写入和 INVLPG 都是属于特权指令,VMM 可以截获它们,并作相应的处理。比较复杂的是客户机操作系统修改客户机页表的操作。

由于页表页也是内存的一部分,其本身也必然作为普通数据页在页表中有一份映射。又由于客户机的页表是由其自身维护的,它本身具有对于页表页的读写权限。在与客户机维护的页表相对应的影子页表中,如果对于页表页的映射也是可写的,那么 VMM 将失去截获页表修改的机会,也就失去了对于客户机操作系统维护的页表的更新追踪。

因此,在影子页表中,对于页表页访问权限是只读的。任何写页表页的操作都会触发缺页异常,由 VMM 截获处理。在处理函数中,VMM 除了代客户机操作系统更新页表项之外,还会将更新的 GPA 翻译成 HPA,同时更新影子页表。

影子页表的建立与修改都是由 VMM 完成的,所以 VMM 总是可以控制影子页表的页访问权限的,由此,VMM 也总是可以控制何时和怎样截获客户机操作系统对客户机页表的修改。

需注意的是,某些时候客户机操作系统会在 VMM 无法感知的情况下,将客户机页表页回收,并将作为普通的数据页使用。例如,当一个进程退出时,其使用的页表页也被回收,但这一事件并非系统级事件,无法从 VMM 获知。也就是说,客户机操作系统回收页表的

操作通常不是由特权或敏感指令完成的,VMM 无从截获,只能凭经验推测。此时,VMM 应对这些页取消写保护,否则就会造成不必要的额外性能的开销。这也是影子页表的设计要考虑的因素之一。

从时间上看,由于提供了影子页表供物理 MMU 直接寻址使用,大多数的内存访问可以在不受 VMM 干预的情况下正常执行,没有额外的地址翻译开销。因此,总体而言,影子页表为完全内存虚拟化减少了时间上的开销,但与在非虚拟环境下相比仍有一定的差距。

从空间上看,影子页表的引入意味着 VMM 需要为每个客户机操作系统的每套页表结构都维护一套相应的影子页表,这会带来较大空间上的额外开销。考虑到宿主机上可能同时运行多个客户机,而多个客户机上也可能同时运行多个进程,同时维护的影子页表所占用的物理地址空间可能非常惊人。另外,客户机操作系统在进程中止时会回收进程页表,前文也提过,这类事件 VMM 无法直接获知,让 VMM 有机会回收已不再使用的影子页表。因此,影子页表的设计中对影子页表占用的物理空间也通常会作优化处理。

一方面,VMM 会根据客户机操作系统对客户机页表的修改得到一些暗示,从而推测出客户机操作系统回收页表页的事件,这些推测可能在多数情况下帮助 VMM 正确回收影子页表,释放宿主机的物理空间。另一方面,在影子页表的设计中,通常都会含有积极的影子页表的回收机制。下面会对此继续讲述。

1. 影子页表的结构

在影子页表中,每个页表项包含的都是宿主机物理地址 HPA。那么,这些宿主机物理地址是如何得到的,与客户机页表中对应的客户机物理地址 GPA 又是什么关系呢?

仍然以 x86 架构上最简单的二级页表为例,当客户机操作系统进入保护模式之前,会为第一个保护模式的进程先准备好客户机 CR3 寄存器的值,我们将该值除去最低 12 位偏移值的高 20 位称为客户机页帧号(Guest Frame Number,GFN)。每帧代表一个内存页。VMM 在为它建立影子页表时,会根据 GFN 找到与之相对应的映射在宿主机上的物理页帧号(Machine Frame Number,MFN)。客户机认为是储存在该 GFN 中的数据实际上储存在该 MFN 中。

VMM 要从宿主机的物理内存中新分配一个物理页,该物理页的起始地址右移 12 位后称为影子宿主机物理页帧号(Shadow Machine Frame Number,SMFN),VMM 拟将这个物理页的起始地址载入物理 CR3 寄存器,指向相应客户机进程的影子页表。客户机操作系统总会切换进程,当下次这个进程又重新被调度执行时,VMM 不需重新分配新的宿主机物理页,只需找到以前为它分配的可载入物理 CR3 寄存器的宿主机物理页即 SMFN 即可。为此,VMM 要在 GFN、MFN 和 SMFN 之间建立一定的联系,考虑到 GFN 和 MFN 是一对一的映射关系,只需建立 MFN 和 SMFN 的关系即可。最常用的算法就是 hash 表,以 MFN 的值和 SMFN 所对应的影子页表的类型 type(通常是指在影子页表中是第几级页表,也有其他特殊类型)为键值来索引 SMFN,即 $SMFN = \text{hash}(MFN, type)$ 。

现在载入物理 CR3 寄存器的宿主机物理页已经有了,对两级页表来说,每张影子页表都有 1024 个页表项,每个页表项对应客户机页表相应位置的页表项。客户机页表项的存在位(Present Bit)如果为 1,则 VMM 会为相应的影子页表的页表项填入宿主机物理地址。如果该页表项所处的页表不是页表结构中的最后一级页表,那么填入的过程则与上述相同。即根据客户机页表项所含物理地址右移 12 位得到 GFN,将之转换为相应的 MFN,若根据 hash(MFN, type)可以得到相应的 SMFN,则该 SMFN 就是应该填入该影子页表项的宿主机物理地址;反之,VMM 需为其新分配宿主机物理页,并为该宿主机物理页和客户机物理页映射的宿主机物理页在 hash 表中建立映射关系,以备下次使用。

现在已经知道影子页表的页表项如何建立,另一个问题就是这些影子页表在何时建立。实际上,影子页表的建立过程与修改过程交织在一起,贯穿于上文所述的 VMM 针对客户机操作系统修改客户机页表和刷新 TLB 所做的三种操作中。这就是 VMM 对客户机操作系统修改客户机 CR3 寄存器的截获与处理,VMM 对客户机操作系统 INVLPG 指令的截获与处理,以及 VMM 对因客户机页表和影子页表不一致而触发的缺页异常的截获与处理。通常,最后一种发生机率最高,处理也最复杂。

2. 影子页表的建立

开始时,VMM 中与客户机操作系统所拥有的页表相对应的影子页表是空的,不包含任何从客户机虚拟地址到宿主机物理地址的映射信息。随后,VMM 以按需调整的方式,根据客户机操作系统修改它所拥有的客户机页表相应地修改与之对应的影子页表。

一开始,影子页表是空的,而影子页表又是载入到物理 CR3 中真正为物理 MMU 所利用进行寻址的页表,因此开始时任何的内存访问操作都会引起缺页异常。如果客户机操作系统为所访问的客户机虚拟地址分配了客户机物理页,即客户机操作系统的当前页表中包含了从这个客户机虚拟地址到已经分配了的某一客户机物理页地址的映射,那么,它正是由于影子页表中相应从客户机虚拟地址到宿主机物理地址的映射尚未初始化造成了这种异常的发生。需要注意的是,这样的缺页异常在非虚拟环境中是不会出现的。在这种情况下,VMM 截获该缺页异常,在相应的影子页表中建立从这个客户机虚拟地址到与客户机物理地址相对应的宿主机物理地址的映射,从而完成了对缺页异常的处理,完成影子页表的初始化,而且这种异常不会再告之给客户机操作系统。

此外,如果 VMM 在客户机操作系统不知情的情况下将分配给客户机的宿主机物理页换出到了硬盘上,那么,也会出现与上述类似的情况,即虽然客户机操作系统为所访问的客户机虚拟地址分配了客户机物理页,但是 VMM 却没有在影子页表中为这个客户机虚拟地址建立相应的到宿主机物理地址映射,这样也会发生上述类型的缺页异常,由 VMM 处理。

当发生缺页异常时,如果客户机操作系统尚未给这个客户机虚拟地址分配客户机物理页,即相应的客户机操作系统页表中的确没有这个客户机虚拟地址到某一客户页的映射,那么,VMM 首先将缺页异常传递给客户机操作系统,由客户机操作系统为这个客户机虚拟地

址分配客户机物理页，而由于客户机操作系统分配客户机物理页需要修改其页表，因此这个操作又被 VMM 截获，VMM 更新影子页表中相应的页目录和页表项，增加从这个客户机虚拟地址到与新分配的客户机物理页相对应的宿主机物理页的映射。

3. 影子页表的缺页处理机制

当缺页异常发生时，首先由 VMM 截获。VMM 将发生异常的客户机虚拟地址在客户机页表中对应页表项的访问权限位与缺页异常的错误码进行比较，从而检查此缺页异常是否是由客户机本身引起的。对于由客户机本身引起的缺页异常，例如客户机所访问的客户页表项存在位(Present Bit)为 0，或者写一个只读的客户机物理页，VMM 将直接返回客户机操作系统，再由客户机操作系统的缺页异常处理机制来处理该缺页异常；如果缺页异常不是由客户机引起的，那么它必定是由客户机页表和影子页表不一致所引起的，这样的缺页异常又称为影子缺页异常(Shadow Page Fault)。对于影子缺页异常，VMM 尝试根据客户机页表同步影子页表。

首先，VMM 根据客户机页表项建立起相应的影子页目录和页表结构，这个过程就是上文影子页表的结构一节所述的内容。然后，VMM 根据发生缺页异常的客户机虚拟地址，在客户机页表的相应页表项中得到与之对应的客户机物理地址。最后，根据客户机物理地址在地址转换表中得到相应的宿主机物理地址，VMM 再把这个宿主机物理地址填入到影子页表项中。如前所述，此时该影子页表项一定在最后一级影子页表中。这样，就建立了从发生缺页异常的客户机虚拟地址到影子页表中相应的宿主机物理地址的映射。

在根据客户机页表项同步影子页表时，除了要建立起相应的影子页表数据结构、填充宿主机物理地址到影子页表的页表项中之外，VMM 还要根据客户页表项的访问位和修改位设置对应影子页表项的访问位和修改位，以保证影子页表项中这些位的语义最终能和客户机页表中的相同。

(1) 如果发生缺页异常，客户机页表页表项的存在位为 0，表示客户机物理页不存在，那么，VMM 将相应的影子页表项设为空值，以保证对应的宿主机物理页也不存在。在这里，VMM 如果只是将相应的影子页表项的存在位置 0 是不够的，因为 VMM 只将存在位置 0 还有其他用处，即下文中会说明的截获客户机将客户页表项访问位置 1 的情况。

(2) 如果客户机页表项的存在位为 1，即客户机物理页存在，并且它的访问位和修改位都被置为 1，那么 VMM 将相应的影子页表项的访问位和修改位置为 1。

(3) 如果客户机页表项的访问位为 0，表示客户机尚未访问相应的客户机物理页。在这种情况下，VMM 需要截获将来客户机页表项的访问位被置 1 的操作，从而将相应的影子页表项的访问位置 1。为此，VMM 将相应的影子页表项的存在位标记为 0，当下次客户机访问这个页时，无论是读还是写访问，由于影子页表项的存在位为 0，因此会发生缺页异常。VMM 截获这个异常，将相应的影子页表项的访问位置 1。注意，上面为了截获读访问这个页引起的客户机页表项的访问位置 1，需要将影子页表项的存在位置 0，所以只是将影子页表项的读写位设为只读是不够的。

(4) 对于客户机页表项的修改位,如果它为 0,表示客户机物理页不曾被写入过。在这种情况下,VMM 需要设法截获未来客户机写此客户机物理页,引起客户机页表项的修改位被置 1 的情况,从而设置相应的影子页表项的修改位和访问位。为此,VMM 会将影子页表项的读写位设为只读。这样,客户机要写客户机物理页时,会发生写相应的只读宿主机物理页而引起的缺页异常,这个异常会被 VMM 截获,VMM 将影子页表项的访问位和修改位设为 1,并将相应的客户机页表项的访问位和修改位设为 1。

正如上文所提到的,影子页表和客户机页表之间并不是时刻同步的,只有在需要的时候才进行同步。这时候,影子页表充当了客户机页表巨大的 TLB,称为虚拟 TLB。当客户机操作系统需要访问它的客户机页表时,物理 MMU 真正访问的是这个称为虚拟 TLB 的影子页表。

当客户机页表被修改时,若影子页表中对应该客户机页表的表项访问权限低于客户机页表表项的,VMM 会截获一个缺页异常,这就可以理解为 TLB 未命中,它表示尽管客户机页表中所访问的是合法的地址映射,但是影子页表中尚未建立起与之相对应的映射,即发生影子缺页异常。此时,VMM 根据客户机页表的客户机虚拟地址到客户机物理地址的映射,在影子页表中建立相应的客户机虚拟地址到宿主机物理地址的映射,并且置上相应的权限位,就相当于 TLB 填充。

当客户机试图修改客户机页表的表项,由于客户机试图执行敏感指令重写 CR3 或执行 INVLPG 敏感指令以刷新 TLB,VMM 截获这一操作,并对影子页表进行相应的修改,刷新影子页表这一相对于客户机页表的虚拟 TLB 中的全部或部分内容,这就相当于 TLB 刷新,如图 4-12 所示。

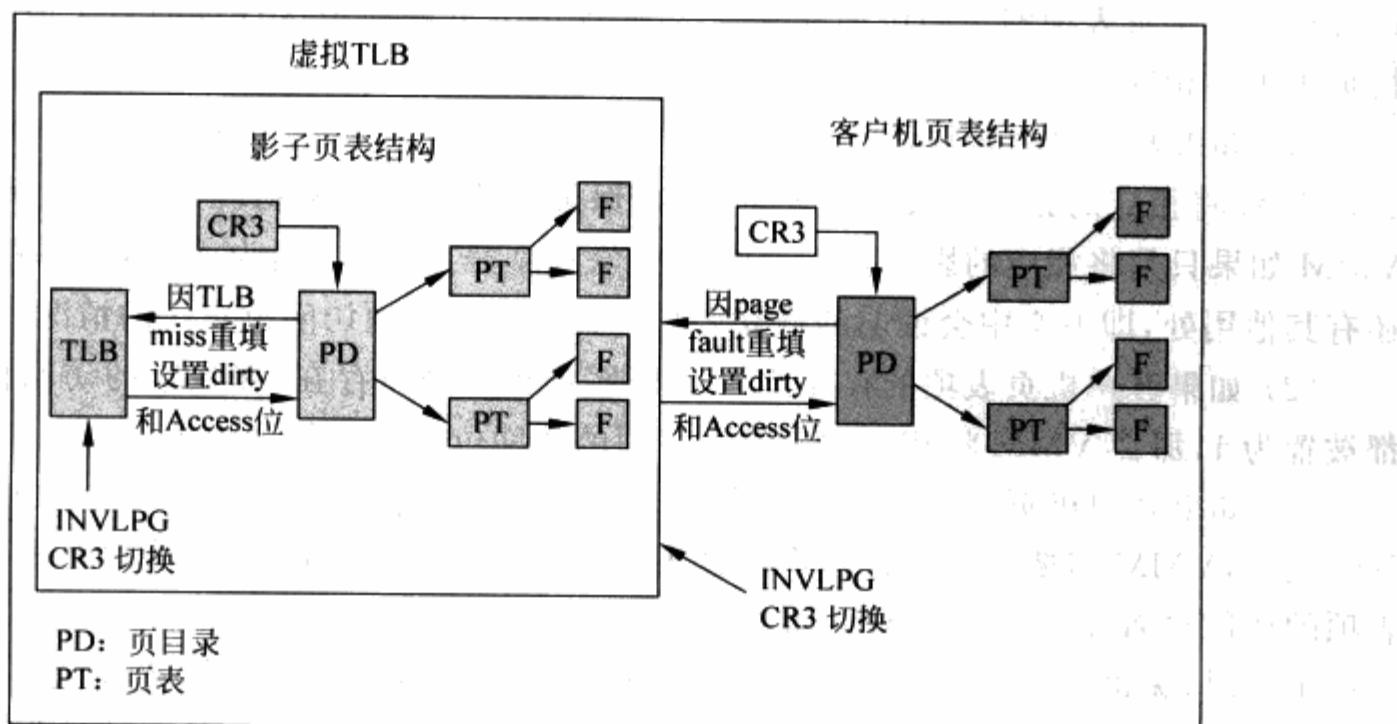


图 4-12 虚拟 TLB

4.3.3 内存虚拟化的优化

1. 自伸缩内存调节技术

自伸缩内存调节技术是一种 VMM 通过“诱导”客户机操作系统来回收或分配客户机所拥有的宿主机物理内存的技术。

在这种技术中,一个称为“气球”的模块作为一个伪设备驱动程序或内核服务载入到客户机操作系统之中。气球模块不提供操作系统或上层应用程序调用的接口,而是只为 VMM 提供了私有的交互接口,通过和 VMM 的交互,实现自伸缩的宿主机物理内存回收和分配。

当 VMM 需要从客户机回收宿主机物理内存时,它通知植人客户机操作系统的气球模块,由气球模块调用客户机操作系统本身的内存分配函数分配客户机物理内存;VMM 相应地把这些客户机物理内存所对应宿主机物理内存回收掉。在这个过程中,气球模块申请客户机物理内存,但并不真正有效地使用它们,而是通知 VMM 这些客户机物理内存对应的宿主机物理内存可以被回收,通过气球的“膨胀”实现了为 VMM “圈地”的功能;而 VMM 将气球圈得的“地”顺理成章地回收掉,挪作他用,从而实现了宿主机物理内存的回收。

气球的“膨胀”是气球模块向客户机操作系统申请更多的客户机物理内存,这使得客户机操作系统所拥有的客户机物理内存资源变得更为紧缺。这将导致客户机操作系统调用它自身的内存管理算法:当客户机物理内存足够时,客户机操作系统从其闲置客户机物理内存链表中返回客户机物理内存给气球;当客户机物理内存资源稀缺时,客户机操作系统必须回收一部分客户机物理内存,以满足气球申请客户机物理内存的需要。客户机操作系统自己决定回收哪些特定的客户机物理页,而且,如果必要的话,将一部分客户机物理页换出到硬盘上,再将它们供给气球。对每个分配给气球的客户机物理页,气球模块将它的客户机物理页号告诉 VMM,VMM 根据它就能将对应的宿主机物理页回收掉了。

当 VMM 需要为客户机分配宿主机物理内存时,也是类似。

由于被气球膨胀而“圈”起来的客户机物理内存所对应的宿主机物理内存为 VMM 所回收,因此,客户机操作系统应该不能访问这些分配给气球的客户机物理内存。然而,即使客户机操作系统企图访问这些被气球“圈”起来的所对应宿主机物理内存已经被 VMM 回收的客户机物理内存,VMM 仍有办法确保系统的正确性不被破坏。当一个客户机物理页被分配给气球时,系统会标记它,回收这个客户机物理页所对应的宿主机物理页号。这样,此后任何对这个客户机物理页的访问都将引起缺页错误,并被 VMM 所截获。从而,VMM 可以有效地重置这一页的状态,把这个客户机物理页从气球那儿还给客户机操作系统,并且将这个客户机物理页映射到新的一张宿主机物理页上。这样一来,在客户机操作系统看来,它可以正常地第一次访问原先被气球夺走的客户机物理页,其实是那页现在对应另

一张新的宿主机物理页，而客户机物理页先前所对应的宿主机物理页仍能顺利地被 VMM 回收。

2. 页共享技术

在客户机技术的许多应用场景中，都存在着在不同的客户机之间共享宿主机物理内存的可能性。例如，当多个客户机运行同一个操作系统的不同实例时，运行相同的应用程序的不同实例时，或者包含共享的数据时，都有共享包含相同数据的宿主机物理内存的机会。因此，如果在一台物理主机上运行多个客户机，VMM 通过实现页共享技术可以有效地节省宿主机物理内存资源。

页共享技术很早就在操作系统的层面上得以实现。例如，有的操作系统实现了对应用程序代码段、只读数据段等只读的物理内存的共享。不同的进程运行相同的应用程序时，可以共享这一应用程序的代码段和只读数据段。万一发生对共享页的写操作，利用写时备份机制加以处理。这种页共享只能识别出有限数量的可共享页，在现实应用中，很可能存在许多可写页实际并没有发生写操作，因而可以被共享。此外，虽然这样的页共享对应用程序是透明的，但是需要修改操作系统，并且可能改变应用程序的编程接口。

随着虚拟机技术的引入，在 VMM 层面上实现页共享，就避免了对操作系统的修改，保持了应用程序接口不变。更为重要的是，基于内容的页共享技术被引入。它通过比较宿主机物理页的数据，可以识别出所有包含相同数据的宿主机物理页，从而实现最大程度的页共享。对于包含相同数据的宿主机物理页，无论它们在客户机操作系统层面上的用途是什么，在 VMM 的层面上，它们都可以被无所顾虑地共享。这种虚拟机层面上实现的页共享有两大优点。

- (1) 它无须修改操作系统，甚至完全不必关心操作系统的逻辑。
- (2) 实现了最大程度的页共享，而不是只共享有限的只读页。

基于内容的页共享主要性能开销，来源于扫描宿主机物理内存数据以挖掘出相同数据的不同宿主机物理页。为了避免这种耗时的宿主机物理页扫描和两两宿主机物理页之间的数据比对，可以利用哈希表以索引的方式高效地描述宿主机物理页的共享情况。

对每一个宿主机物理页，系统为它的数据计算哈希值，作为哈希索引的键值。如果这个键值对应的哈希表单元已经存在，那么，这个宿主机物理页和先前已经加入到这个哈希表单元中宿主机物理页拥有相同的哈希值，这意味着它们有很高的概率包含相同的数据而可以共享。接着，对这些哈希值相同的宿主机物理页，系统进一步比较它们的数据，若数据相同，则确认它们可以共享。

对于可以共享的宿主机物理页，系统使用经典的写时备份机制实现共享，多余的备份被删除。任何尝试对共享页的写操作都会引起缺页错误，并被 VMM 所截获。VMM 在客户机操作系统完全不知情的情况下复制一份备份分配给需要进行写操作的客户机。

如果这个键值对应的哈希表单元不存在,那么就创建一个新单元,将这个宿主机物理页的描述符加入到这个单元中。直到未来系统发现有另外一个页和它包含相同的数据,这个引起新单元创建的宿主机物理页才会和新加入的宿主机物理页一起被标记为写时备份。与将引起新单元创建的页立即标记为写时备份相比,通过延后标记为写时备份避免了由不能共享的宿主机物理页被标记为写时备份而引起不必要的缺页异常。

4.4 I/O 虚拟化

在第3章中已经介绍了I/O虚拟化的基本原理与方法,这一节着重介绍I/O完全虚拟化的原理与实现。本节首先介绍设备模型的基本概念,然后介绍设备模型的原理、方法和一般实现,最后,以IDE设备的DMA操作为例,分析设备模拟的执行过程。由于PC上的设备主要是通过PCI总线连接的,因此下面的介绍将以PCI设备为主。

4.4.1 设备模型

虚拟机中侦测和驱动的设备一般不是直接对应于硬件设备,而是由VMM抽象出来的,其设备的种类和型号与真实设备可能比较接近,也可能完全不同。不同的VMM提供的虚拟设备种类和型号都是不同的。

虚拟设备的功能可以多于或少于真实硬件,甚至能够模拟出真实硬件不具备的一些特性,虚拟出不存在的硬件设备。例如,在VMware Workstation中,虚拟机可以拥有一个SCSI磁盘,而真实硬件上所用的可以是IDE磁盘,SCSI磁盘有一些特性是IDE磁盘不具备的,这些特性都是由设备模型模拟出来的。

在软件完全虚拟化系统中,一般使用I/O模拟的方法来虚拟化I/O设备。具体来说,在进行设备虚拟化时,VMM需要对某一目标设备进行模拟,为客户机提供一个虚拟的设备,使其可以透明地对这个虚拟设备进行操作;客户机操作系统发现虚拟的目标设备后,会使用目标设备的驱动程序来驱动该设备。客户机中的驱动程序会发出一些请求并等待设备的响应,这些请求被VMM拦截处理后,响应会返回给客户机操作系统。如果这些响应与真实物理设备的响应相似,客户机操作系统可以安全地认为自己运行于物理硬件平台之上,VMM就成功地给客户机提供了一个虚拟设备。

VMM中进行设备模拟,并处理所有设备请求和响应的逻辑模块,就是设备模型。

设备模型并不需要精确模拟目标设备。从操作系统的角度,设备可以分为软件可见部分和软件不可见部分。前者是操作系统控制操作硬件的所有接口,后者则包括硬件的内部逻辑以及与其他设备的连接等。设备模型在进行设备I/O模拟的时候,只需要正确模拟目标设备的软件接口就可以保证客户机操作系统观察到的虚拟设备与目标设备一致,而不必

考虑真实硬件的硬件构造及硬件接口,也不需要了解所运行的客户机操作系统的技术细节,如图 4-13 所示。

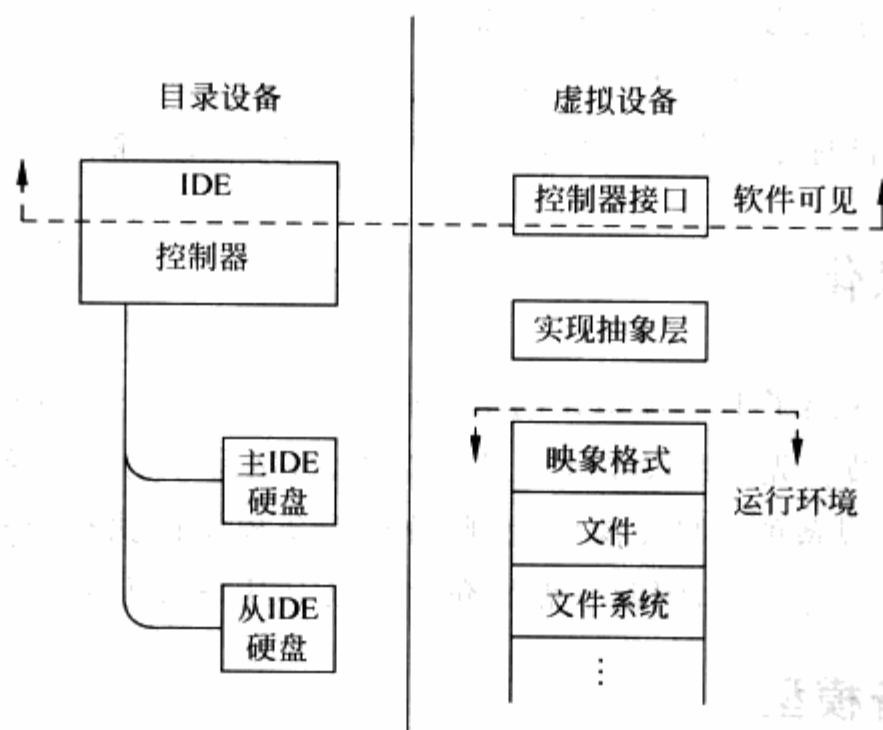


图 4-13 目标设备与虚拟设备

设备模型为了模拟目标设备软件接口,也需要同时实现目标设备的功能。这些功能也是基于软件实现的。因此,设备模型所模拟的目标设备与宿主机的硬件不存在直接的关联和对应关系,而是建立在一定的运行环境之上。例如,操作系统所提供的系统调用,这种方法使得设备模型可以完全独立于宿主机的硬件,进而实现跨平台的设备模拟。图 4-14 显示了设备模型的逻辑层次关系,对于不同构造的虚拟机,其逻辑层次都是类似的:VMM 拦截客户机的 I/O 操作,将这些操作传递给设备模型进行处理;设备模型运行在一个特定的运行环境下,这可以是操作系统,可以是 VMM 本身,也可以是另一个客户机。

图 4-15 所示的是宿主机模型中设备模型的一个可能的实现。在这个例子中,VMM 的主要部分是一个宿主机操作系统的内核模块,设备模型是一个用户态进程。当客户机发生 I/O 之后,VMM 作为内核模块将其拦截后,会通过宿主机内核态-用户态接口传递给用户态的设备模型处理。设备模型运行于宿主机操作系统之上,可以使用相应的系统调用以及所有运行库。宿主机操作系统及其上的运行库,就构成了设备模型的运行环境。

在 Hypervisor 模型中,设备模型的位置如图 4-16 所示。

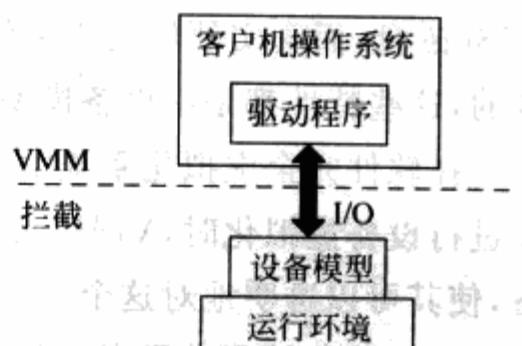


图 4-14 设备模型在 VMM 中的分层

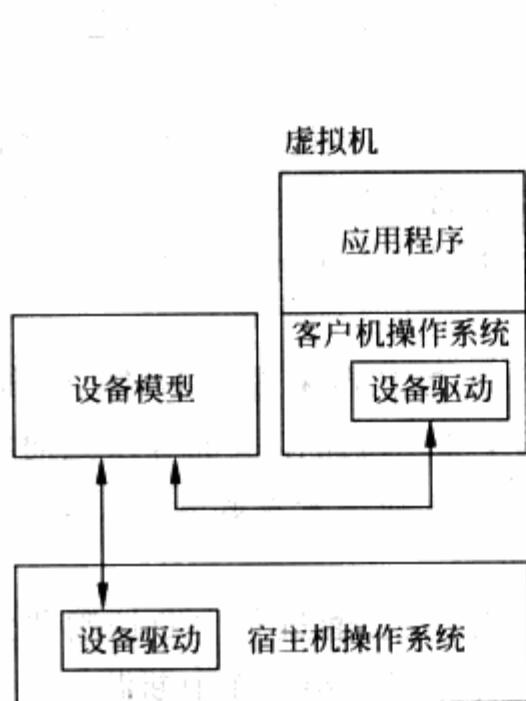


图 4-15 宿主机模型中的设备模型

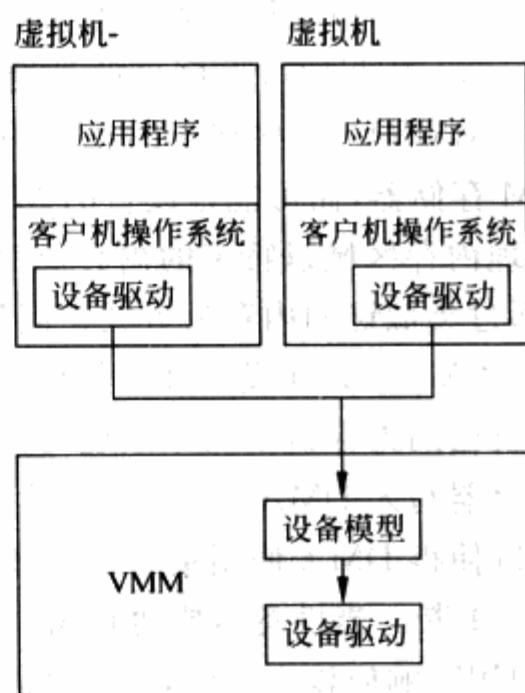


图 4-16 Hypervisor 模型中的设备模型

设备模型是位于虚拟机设备驱动程序与实际设备驱动之间的一个模块。由设备驱动所发出的 I/O 请求先通过设备模型模块转化为物理 I/O 设备的请求,再通过调用物理设备驱动来完成相应的 I/O 操作。反过来,设备驱动将 I/O 操作结果通过设备模型模块,返回给客户机操作系统的虚拟设备驱动程序。

4.4.2 设备模型的软件接口

由于设备多种多样,不同的设备其软件接口也差异巨大,一个完整的设备模型需要大量的代码分别对每个设备的接口和逻辑进行模拟。然而,还是可以看到,不同设备的软硬件交换信息的方法是有限的,对于一个典型的 PCI 设备,它可能包含以下种类的接口。

(1) PCI 配置空间。在第 2 章中提到,PCI 配置空间包含了设备的很多基本信息,最重要的包括设备标识符,它使 OS 可以发现并识别设备类型;基址寄存器,它使 OS 可以映射并寻址属于该设备的寄存器。PCI 配置空间通过平台相关的寄存器访问,可以是端口 I/O,也可以是 MMIO,一般由两个寄存器组成,一个用于指定设备和偏移,另一个用于读取或写入数据。PCI 配置空间的寻址方式如图 4-17 所示。另外,PCI 设备的发现也是通过客户机操作系统遍历 PCI 总线(总线号、设备号及功能号的组合),检查返回值的有效性来进行的。

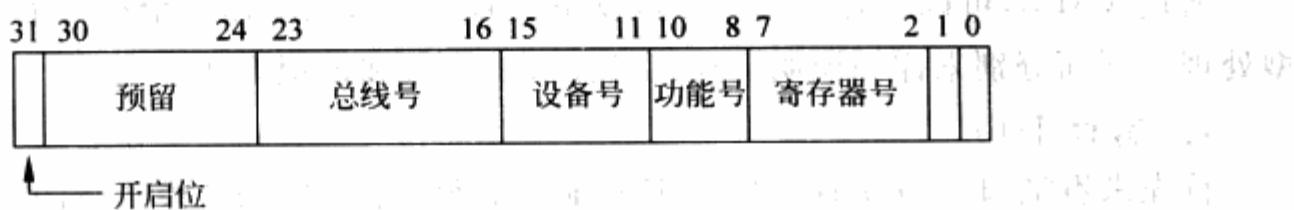


图 4-17 PCI 配置空间偏移寄存器

(2) 端口 I/O。操作系统通过特定指令访问 I/O 空间，在 x86 平台上，这包括 in、out、ins 和 outs。这些端口 I/O 一般是设备相关的寄存器。

(3) MMIO。某些特定的物理内存区域（如 0xf000000~0xffffffff）并不会映射到真正的 RAM 存储器，而可能是设备的 MMIO，其中包含设备的寄存器。操作系统通过页表将相应的物理内存区域以特定的内存类型（例如某些情况下不进行缓存）映射到虚拟地址空间内，并通过类似访问内存的方式访问设备寄存器。

(4) DMA。PCI 设备并不使用 ISA 设备所用的 DMA 控制器来进行 DMA 操作，而是通过其自己的寄存器使操作系统可以控制 DMA 的传输。例如，操作系统可以先向特定的硬件寄存器写入 DMA 的地址，然后向另一寄存器写入 DMA 命令来发起一个 DMA。这种方式可以使得 DMA 使用更大的物理地址空间。

(5) 中断。当设备需要通知操作系统处理某些中断时，它会通过其中断控制器发起中断。在 CPU 响应该中断时，一般会通过读写硬件设备的特定寄存器清除中断源。

在虚拟机中，当客户机通过这些接口与虚拟设备进行数据交换时，VMM 会截获这些访问，并将其重定向至设备模型，就可以进行设备模拟了，如图 4-18 所示。

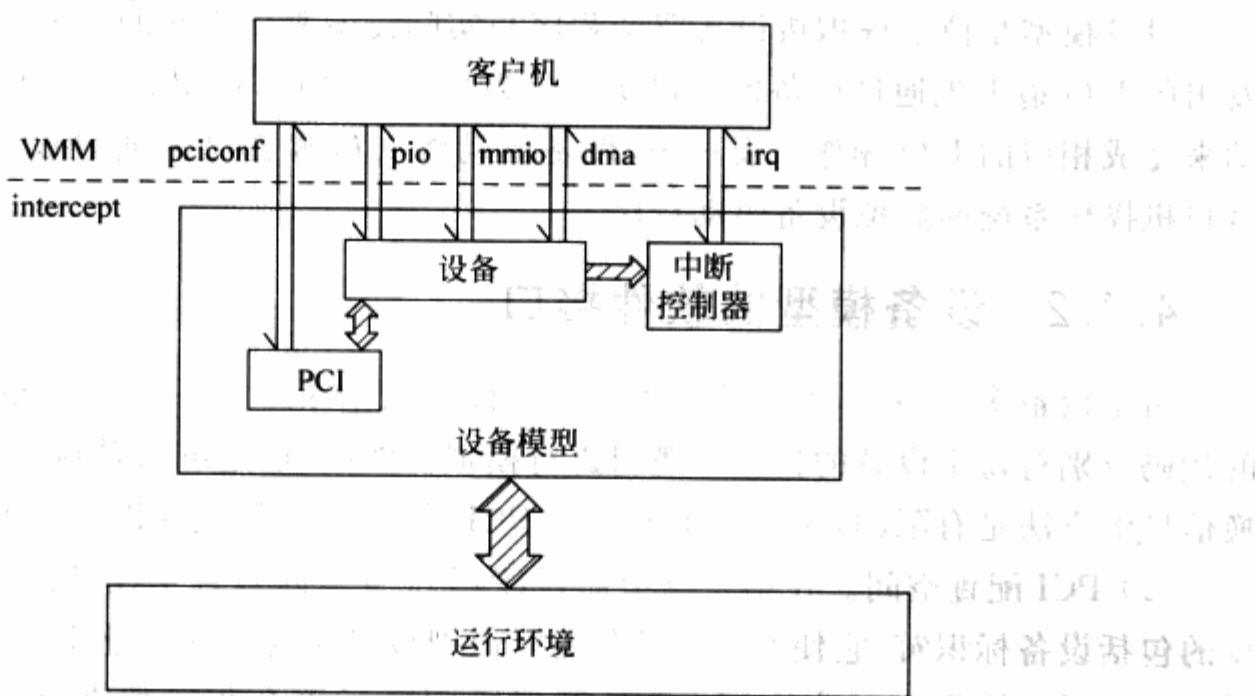


图 4-18 设备模型的软件接口

4.4.3 接口拦截和模拟

通过 VMM，可以将客户机对虚拟设备的软件接口的操作完全拦截下来，并交给设备模型处理。下面分别来详细说明上述几类接口的拦截和模拟方法。

1. 端口 I/O

首先来看端口 I/O 实现。以 IDE 控制器为例，传统的 PIO 模式下的 IDE 控制器使用 4 个不同的端口 I/O 范围，分别是 0x1f0-0x1f7、0x3f4-0x3f7、0x170-0x177 以及 0x374-0x377。

其中,前两组对第一条 IDE 电缆(Primary),后两组对应第二条 IDE 电缆(Secondary)。所有 IDE 命令和数据的读写都会通过前面所提到的 in、out、ins 和 outs 这 4 条指令由客户机发起。对于这 4 条敏感指令,VMM 可以通过修补、动态翻译或者直接陷入的方式拦截并执行端口 I/O 的处理函数。在初始化阶段,设备模型首先会将这些端口 I/O 在 VMM 中进行注册,客户机运行过程中当这些端口的访问发生时,VMM 会根据其端口号和访问的数据宽度(1、2、4 字节等)分发至相应的设备模型预先注册的端口 I/O 处理函数,相应的端口 I/O 处理函数可以由此用软件模拟所需逻辑。图 4-19 描述了当客户机试图在主 IDE 设备的 COMMAND 寄存器块的 DATA 寄存器上写入 4 个字节时,VMM 及设备模型的处理过程。

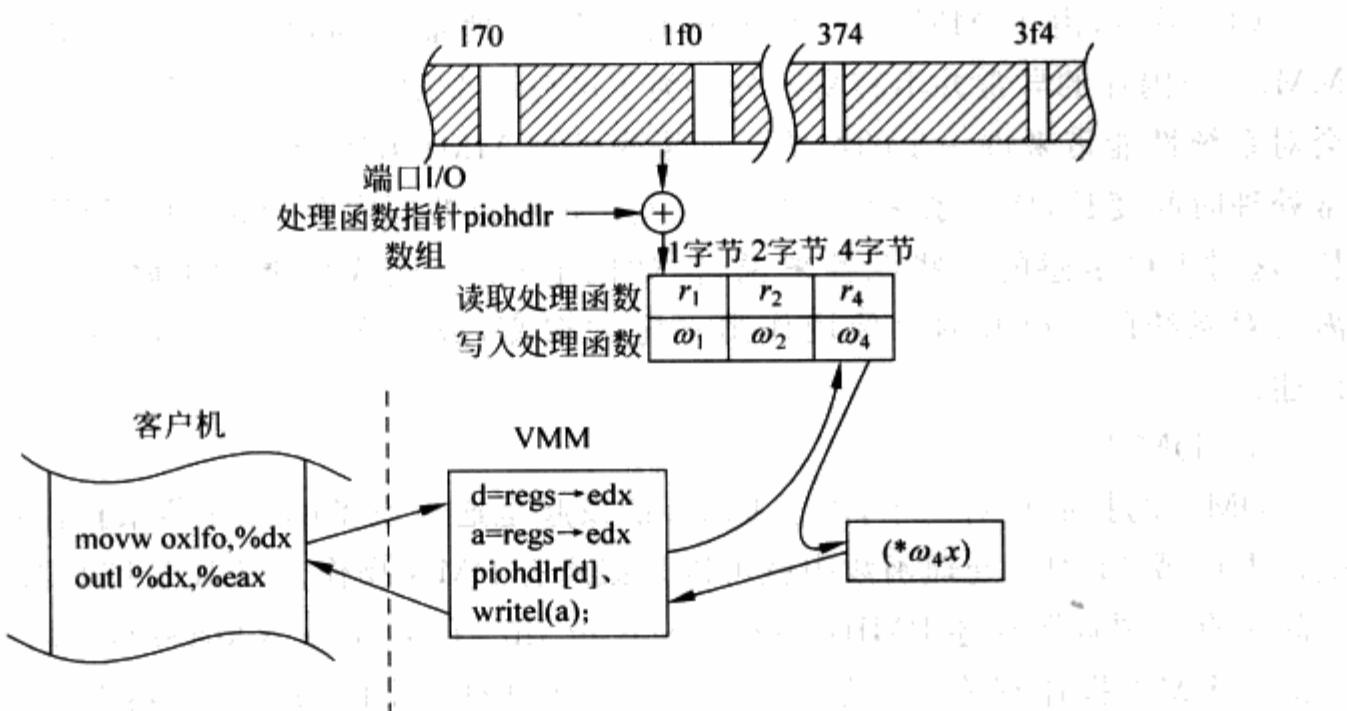


图 4-19 IDE 控制器设备模型的端口 I/O 处理

2. MMIO

需要较大寄存器空间的设备一般会使用 MMIO,即内存映射的 I/O,例如网络设备、显卡等。MMIO 与物理内存共用一个地址空间,例如 MMIO 可以放置在 3.75~4GB 的高地址上。对于 VMM、MMIO 的处理方法与端口 I/O 是类似的,也是基于拦截/分发/处理这一过程。但是,MMIO 的模拟与端口 I/O 相比也有一些显著的不同。

(1) 由于 MMIO 的访问不限于某些特定指令,因此不可能采用类似端口 I/O 的提前修补或翻译。为了使 MMIO 访问陷入,在初始化阶段客户机映射 MMIO 所属的物理地址范围时,VMM 不会建立相应的影子页表项。而当运行时,客户机的 MMIO 访问都会造成缺页异常,VMM 拦截这些异常后就可以将控制交由设备模型进行处理了。

(2) 一个 I/O 端口上可以进行多字节的访问,例如,对于 0x1f0 的 4 字节地址与 0x1f1~0x1f7 完全无关,在 MMIO 中则并非如此。一个 4 字节的 MMIO 寄存器一般会占用内存地址空间的 4 个字节。因此,对 MMIO 的处理要对访问宽度、越界和非对齐访问小心地检查和处理。

(3) 由于端口 I/O 的空间比较小,又不存在对齐问题,一般可以采用数组结构来存储各端口对应的处理函数,这会获得较高的性能。但是,在 MMIO 的情况下,由于所占范围较大(可能达上百 MB),使用数组结构会占用过大的内存,利用率也很差。所以,一般 MMIO 分发都是基于区域(Region)实现的,即设备模型向 VMM 指定其可以处理的 MMIO 区域(基地址和长度)及相应的处理函数。这样的方法减小了内存的使用,但却在一定程度上影响了性能。在 MMIO 陷入发生时,VMM 要根据异常地址查找其对应的区域。在 MMIO 较少时,可以简单地使用链表作为数据结构;当需要处理的 MMIO 区域非常多时,可能要考虑引入更加复杂的查找算法。

(4) 最后,由于 MMIO 与系统内存在同一地址空间,而且都是由缺页异常陷入的,MMIO 与内存的异常并不容易区分。为了区分一个缺页异常是 MMIO 还是系统内存可能会对系统性能带来进一步的影响。如果先处理 MMIO,则会导致每一个普通的影子页表异常处理时间变长,使得系统整体性能变差;如果先处理内存异常,则 MMIO 的处理时间变长,这对 I/O 敏感的一些设备来说(例如网卡)都会造成性能的下降。为解决这一矛盾,需要对系统的 I/O 和内存使用作出一定的权衡,或引入更快的方法进行区分,在此不再详述。

3. DMA

DMA 的拦截相对简单,由于 DMA 的发起是通过设备的寄存器来控制的,设备模型在端口 I/O 或 MMIO 处理函数中就可以拦截所有 DMA 操作。以 IDE 控制器为例,PCI 配置空间中有一项资源描述 BMIBA (Bus Master Interface Base Address),指向了一块 16 个端口的与 DMA 操作有关的寄存器块。客户机通过对其中的命令寄存器 BMICX 的第 0 位置 1 就可以发起 DMA 操作。

如前所述,设备模型并不需要了解具体设备上 DMA 的实现方法,而只需将数据从客户机所属内存中读出或写入即可,这需要通过内存管理模块的帮助将客户机用于 DMA 传输的缓冲区映射到设备模型的地址空间内。与一般的多线程程序类似,在这一过程中,设备模型需要注意缓存和可能出现的更新顺序问题,以确保客户机在知道 DMA 结束的时候(例如通过中断),DMA 内存中的数据是有效的。

4. PCI 配置空间

客户机发现和初始化设备的时候会首先访问 PCI 配置空间,其中的基地址寄存器和命令寄存器使得客户机可以使用该设备的其他 I/O 资源。由于客户机所有设备使用相同的 PCI 配置空间寄存器来访问所有设备的配置空间,而配置空间的头部又有统一的标准,设备模型通常可以使用统一的配置空间处理函数来处理设备的 I/O 资源分配和映射。另外,客户机 BIOS 或操作系统可能会通过写入基地址寄存器的方法重新配置 I/O 资源的基地址,设备模型在拦截到这些操作后也要将对应资源映射做相应的修改。

然而,还是会有一些设备使用 PCI 配置空间可能与其他设备不同,从而需要特别处理。例如,由于历史原因,在访问第一个 IDE 控制器(或者唯一的 IDE 控制器)时,部分客户机会

忽略基地址寄存器的前四项，而是使用特定的端口 I/O 0x1f0、0x3f4、0x170 和 0x374。但是，对于第二个 IDE 控制器（如果存在），客户机使用资源则会遵循其基地址寄存器。因此，为了正确运行这些客户机，设备模型需要在基址寄存器外额外注册这 4 组端口。再如，一部分设备会在配置空间中放置一些重要的设备相关寄存器（如与中断有关的寄存器等）。在处理这些设备的 PCI 配置空间时，就必须引入设备相关的逻辑。

5. 中断

中断的处理需要设备模型模拟的中断控制器来处理。作为 PCI 设备，只需要控制其到中断控制器的中断线即可，与物理设备的逻辑和处理方法类似。

4.4.4 功能实现

由于在功能实现时不必拘泥于目标设备的硬件结构和组成，实现虚拟设备的功能要灵活得多。在前述的 IDE 存储系统的例子中，真实设备一般是由 IDE 控制器以及挂在其下的具体 IDE 硬盘所组成：IDE 控制器是一个 PCI 设备，有一系列软件可控的接口；而硬盘本身则被控制器控制，没有独立的软件接口。而在虚拟 IDE 时，只需将 IDE 控制器的软件接口模拟并暴露给客户机使用，而并不一定需要遵从控制器——硬盘这一真实物理结构。事实上，为了加大灵活性，虚拟 IDE 往往会搭建一个专门的块设备抽象层，它的实现可以是一块硬盘，或者一个分区，也可以是不同文件格式的单一文件。通过在块设备的实现上使用文件格式，可以引入一些真实硬件所没有或较难实现的高级特性，如加密、增量存储和备份等。

在实现虚拟设备的功能时，一般要访问物理上的真实硬件，这是通过运行环境（如宿主机操作系统）的系统调用完成的。在这个过程中，设备模型和运行环境会一起给虚拟设备的 I/O 带来额外的时间和 CPU 的开销，这些与设备模型本身所采用的 I/O 模型，VMM 处理的及时性都有关。在对 I/O 效率要求较高的设备模拟时，例如网络设备，如何减少这些额外开销就显得尤其重要。

与一般的 I/O 密集型程序类似，I/O 模型是影响设备模型和整个 VMM 性能的一个重要部分。图 4-20 以 POSIX 运行环境为例介绍了经过简化的 3 种典型的 I/O 模型，它们被不同 VMM 广泛地使用。图 4-20(a) 将非阻塞查询置于主线程的主循环之中，每次循环都对等待的 I/O 操作进行非阻塞的查询，当无数据时则继续循环，否则读取并处理数据。当需要等待的 I/O 操作较多时，一般会用非阻塞的 select 代替。图 4-20(b) 所示为使用 POSIX aio 的设备模型，即异步 I/O，当需要发起 I/O 时，设备模型在主线程中指定缓冲区并调用 aio_read，然后继续循环，当数据结束时则自动调用信号处理函数完成通知过程。图 4-20(c) 则使用了独立的 I/O 线程，这允许设备模型使用阻塞 I/O，而将调度任务交给运行环境的调度器来完成。

这三种不同的 I/O 模型其优缺点与普通程序也是一致的。非阻塞 I/O 实现较为简单，但由于需要等待运行部分结束后才进行查询，在数据到达之后会引入额外的延时。异步

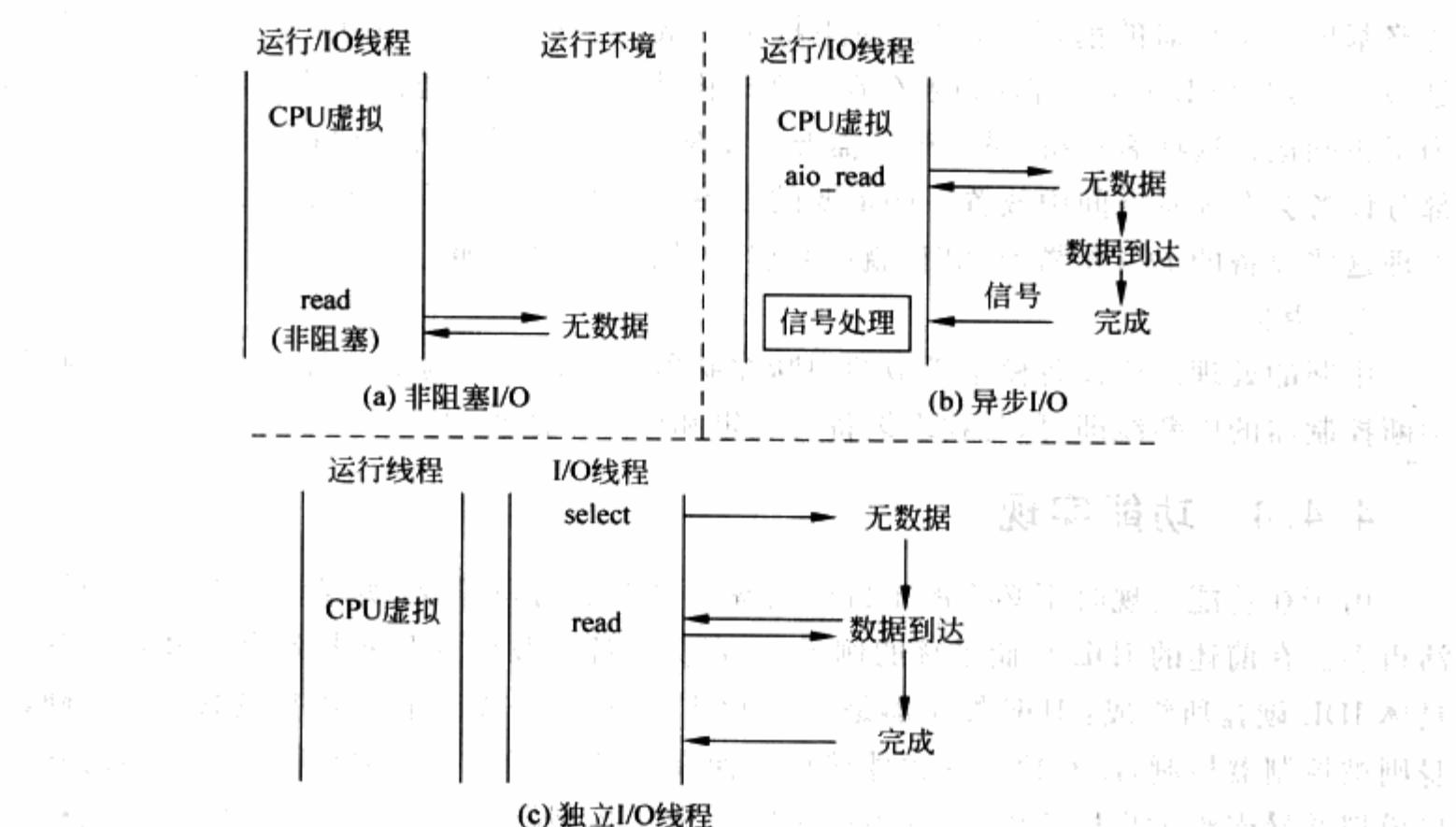


图 4-20 设备模型的 I/O 模型

I/O 在响应速度上略好,而且在主线程内减少了数据到达后重新调用 read 系统调用的时间,不过 aio 的跨平台性略差,而且需要健壮的信号处理程序来避免可能出现的丢失信号的问题。独立的 I/O 线程则较为稳定,可以依靠良好的调度器获得较好的性能。处理不同的 I/O 时,还可以使用多个 I/O 线程,进一步提高性能。

4.4.5 案例分析: IDE 的 DMA 操作

下面分析一个具体的例子:客户机的 IDE DMA 读操作。为行文方便,假定虚拟设备的端口 I/O 为 0x1f0-0x1f7(命令寄存器)、0x3f4-0x3f7(控制寄存器)和 0xc100-0xc10f(DMA 寄存器),设备模型使用 aio 方式。整个过程如图 4-21 所示。

- (1) 为了完成 DMA 操作,客户机驱动程序首先需要设置一个物理区域描述符表(Physical Region Descriptor Table, PRDT),并将其物理地址写入寄存器 BMIDTPX(0xc104)。
- (2) VMM 拦截了这个端口 I/O 写入,并调用 IDE 设备模型中处理这个端口的相应函数。
- (3) 设备模型用 VMM 所提供的内存管理功能将 PRDT 映射到自己的地址空间。
- (4) 返回客户机后,驱动程序将用于存放读取数据的缓冲区的物理地址及长度写入 PRDT。
- (5) 客户机驱动程序通过命令寄存器 0x1f0-0x1f7 指定需要读取的 IDE 扇区,并通过写

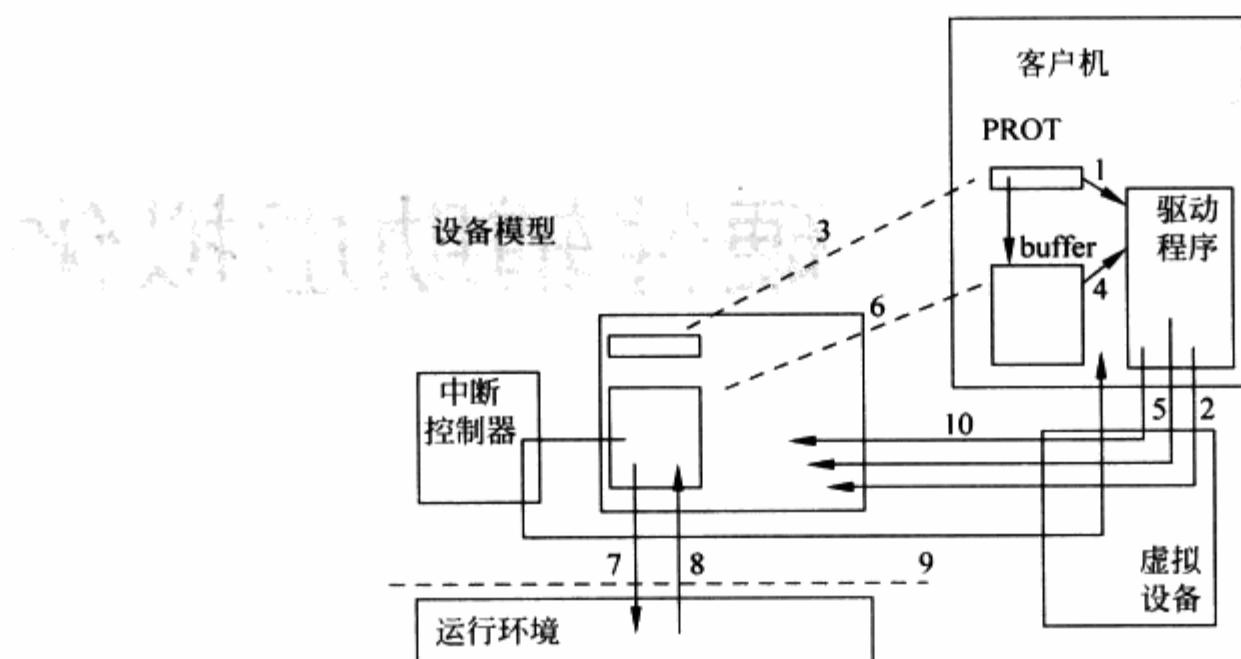


图 4-21 IDE 的 DMA 读操作

入 0x1~BMICX(0xc100)发起读 DMA 操作。

(6) 设备模型截获这个操作后,首先读取 PRDT 中的描述符,并将客户机缓冲区映射到自己的地址空间。然后通过扇区地址,计算出这些数据实际存在的位置,例如映像文件内的偏移。

(7) 设备模型使用 aio_read 系统调用发起读取操作,将实际数据读入缓冲区,然后返回客户机运行。

(8) 当异步 I/O 结束之后,运行环境通过信号通知设备模型。当设备模型的信号处理函数运行时,需要读取的数据已经被读入了缓冲区中。

(9) 设备模型通过虚拟中断控制器向客户机注入中断。

(10) 客户机响应中断,通过写寄存器 BMISX(0xc102)清除中断标志。至此,一次 IDE 的 DMA 读操作就结束了,在客户机缓冲区内出现的数据就可以被客户机继续使用。

4.5 思考题

(1) 客户机操作系统释放页表页时,VMM 可能并不知情而继续维持对该页的写保护。那么,VMM 如何才能尽早地发现并取消对客户机操作系统已经释放了的页表页的写保护,从而避免不必要的权限不足导致缺页异常,以提高运行效率?

(2) 借助 QEMU 系统的代码框架,实现一个简单的设备模型。设备的行为是将任意的字符串输入逆序输出,输入字符串的最大长度为 4096 字节。

CHAPTER 5

第 5 章

硬件辅助虚拟化

5.1 概述

通过前面几章，读者已经熟悉了 VMM 的结构和功能，也了解了如何通过软件虚拟化技术来实现 VMM。本章以 IA32 架构为主，介绍如何通过硬件辅助虚拟化技术来实现 VMM。很多硬件体系结构都有类似技术，如 IA64 的 VT-i^[15]。

硬件辅助虚拟化技术，顾名思义，就是在 CPU、芯片组及 I/O 设备等硬件中加入专门针对虚拟化的支持，使得系统软件可以更加容易、高效地实现虚拟化功能。

之所以需要在硬件中加入虚拟化的支持，原因是多方面的。首先，由于原有的硬件体系结构在虚拟化方面存在缺陷，例如虚拟化漏洞，导致单纯的软件虚拟化方法存在种种问题，如降优先级(deprivelege)的方法存在 Ring Compression 问题。BT 技术存在难以处理自修改代码及自参考代码的问题。PV 技术存在需要修改源码的总问题；其次，由于硬件架构的限制，某些虚拟化功能尽管可以用软件方法来实现，但是实现起来非常复杂，一个典型的例子是内存虚拟化的“影子页表”；最后，某些通过软件方法实现的虚拟化功能性能不佳，例如 I/O 设备的虚拟化。这些问题，都只有通过在 CPU 体系架构上增加相应的硬件支持，才能彻底解决。

这里以 Intel Virtualization Technology (Intel VT) 为例具体说明硬件辅助虚拟化技术所提供的支持。Intel VT 是 Intel 平台上硬件虚拟化技术的总称，包含了对 CPU、内存和 I/O 设备等各方面的虚拟化支持。图 5-1 中的 Physical Platform Resource 列举了 Intel VT 涵盖的内容。在 CPU 虚拟化方面，Intel VT 提供了 VT-x (Intel Virtualization technology for x86) 技术；在内存虚拟化方面，Intel VT 提供了 EPT(Extended Page Table)技术；在 I/O 设备虚拟化方面，Intel VT 提供了 VT-d (Intel Virtualization Technology for Direct I/O) 等技术。

图 5-1 展示了使用 Intel VT 技术实现的 VMM 的典型结构。上层是通用功能，如资源管理、系统调度等。下层是平台相关的部分，即使用 Intel VT 实现的处理器虚拟化、内存虚拟化和 I/O 虚拟化。

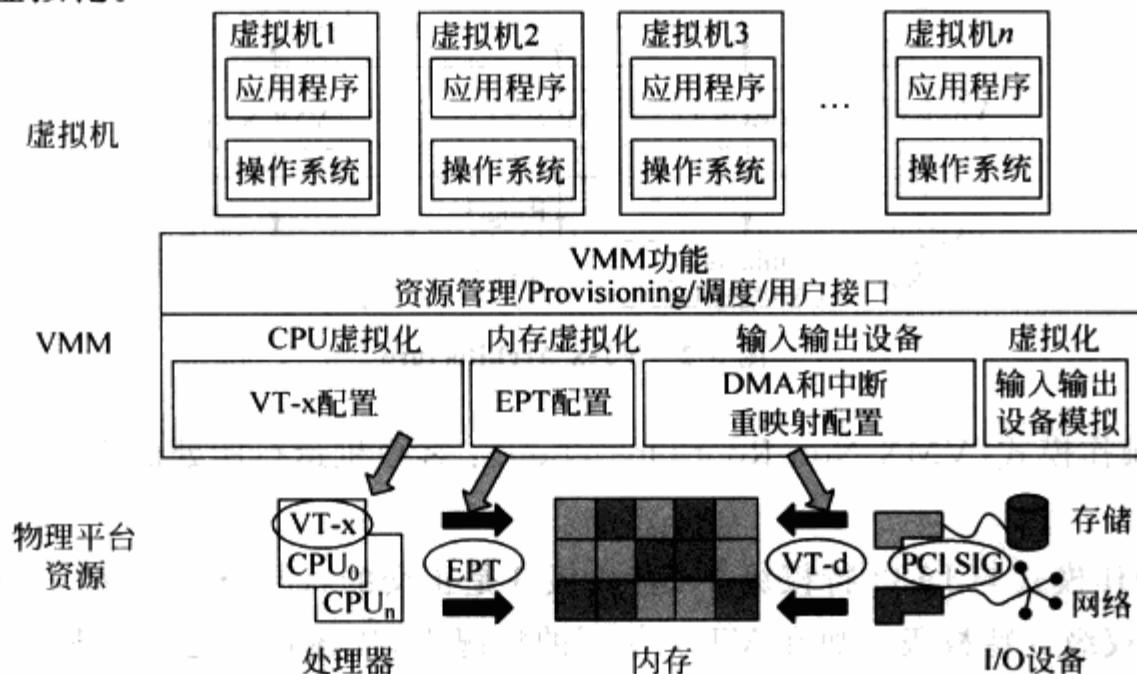


图 5-1 Intel 虚拟技术

本章下面的章节是对图 5-1 的展开，以 Intel VT 为例阐述硬件辅助虚拟化技术，并进一步描述如何在 VMM 中运用这些技术。5.2 节和 5.3 节是 CPU 虚拟化的相关内容，包括 VT-x 的原理介绍，以及如何利用 VT-x 实现 CPU 虚拟化；5.4 节介绍中断虚拟化的相关内容；5.5 节介绍 EPT 的原理和基于该技术的内存虚拟化实现；5.6 节和 5.7 节介绍 VT-d 技术以及基于该技术的 I/O 虚拟化的相关内容。最后，5.8 节对虚拟化技术中的难点——时间虚拟化加以介绍。

此外，AMD 平台也提供了类似的技术，即 AMD Virtualization (AMD-V)，其原理和使用方式与 Intel VT 类似，有兴趣的读者可以进一步阅读 AMD-V 的相关文档。

5.2 CPU 虚拟化的硬件支持

5.2.1 概述

Intel VT 中的 VT-x 技术扩展了传统的 IA32 处理器架构，为 IA32 架构的处理器虚拟化提供了硬件支持。^[17] 手册包含了 VT-x 具体的硬件规范，本节会依据该硬件规范展开描述，读者也可以阅读该手册进一步了解技术细节。

VT-x 的基本思想可以概括为图 5-2 所示。

首先，VT-x 引入了两种操作模式，统称为 VMX 操作模式。

- 根操作模式(VMX Root Operation)：VMM 运行所处的模式，以下简称根模式。

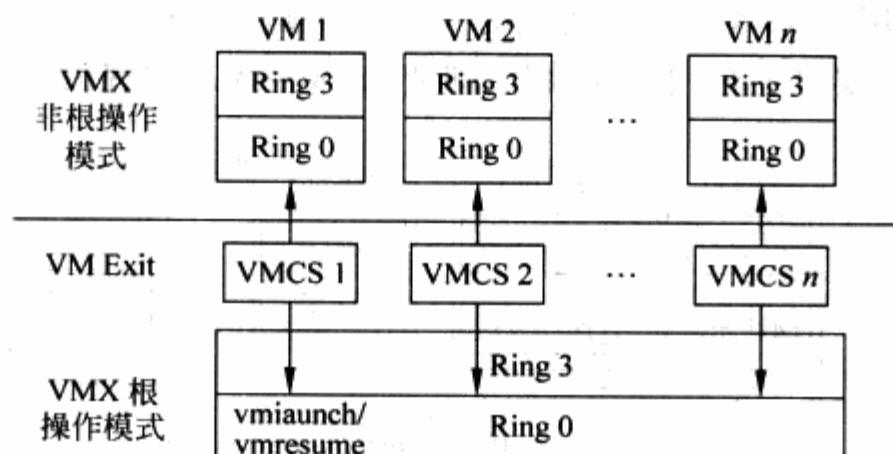


图 5-2 VT-x Architecture

- 非根操作模式(VMX Non-Root Operation): 客户机运行所处的模式,以下简称非根模式。

这两种操作模式与 IA32 特权级 0~特权级 3 是正交的,即每种操作模式下都有相应的特权级 0~特权级 3 特权级。故在 VT-x 使用的情况下,描述程序运行在某个特权级,例如特权级 0,还必须指出当前是处于根模式还是非根模式。

引入两种操作模式的理由很明显。回顾第 4 章 4.1 节关于虚拟化漏洞的阐述,我们知道指令的虚拟化是通过“陷入再模拟”的方式实现的,而 IA32 架构有 19 条敏感指令不能通过这种方法处理,导致了虚拟化漏洞。最直观的解决办法,是使得这些敏感指令能够触发异常。可惜这种方法会改变这些指令的语义,导致与原有软件不兼容,这是不可接受的。引入新的模式可以很好地解决问题。非根模式下所有敏感指令(包括 19 条不能被虚拟化的敏感指令)的行为都被重新定义,使得它们能不经虚拟化就直接运行或通过“陷入再模拟”的方式来处理;在根模式下,所有指令的行为和传统 IA32 一样,没有改变,因此原有的软件都能正常运行。

VT-x 中,非根模式下敏感指令引起的“陷入”被称为 VM-Exit。VM-Exit 发生时,CPU 自动从非根模式切换成为根模式。相应地,VT-x 也定义了 VM-Entry,该操作由 VMM 发起,通常是调度某个客户机运行,此时 CPU 从根模式切换成为非根模式。

其次,为了更好地支持 CPU 虚拟化,VT-x 引入了 VMCS(Virtual-Machine Control Structure,虚拟机控制结构)。VMCS 保存虚拟 CPU 需要的相关状态,例如 CPU 在根模式和非根模式下的特权寄存器的值。VMCS 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。VMM 可以通过指令来配置 VMCS,进而影响 CPU 的行为。

最后,VT-x 还引入了一组新的指令,包括 VMLAUNCH/VMRESUME 用于发起 VM-Entry,VMREAD/VMWRITE 用于配置 VMCS 等。

下面的章节中,会分别对 VT-x 引入的 VMCS、VMX 操作模式、虚拟化相关的新指令分别进行介绍。

5.2.2 VMCS

VMCS 的概念与第 4 章 4.2 节阐述的虚拟寄存器的概念类似,可以看作是虚拟寄存器概念在硬件上的应用。虚拟寄存器的操作和更改完全由软件执行,但 VMCS 却主要由 CPU 操作。VMCS 是保存在内存中的数据结构,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,每个 VMCS 对应一个虚拟 CPU。

VMCS 在使用时需要与物理 CPU 绑定。在任意给定时刻,VMCS 与物理 CPU 是一对一的绑定关系,即一个物理 CPU 只能绑定一个 VMCS,一个 VMCS 也只能与一个物理 CPU 绑定。VMCS 在不同的时刻可以绑定到不同的物理 CPU,例如在某个 VMCS 先和物理 CPU1 绑定,并在某个时刻解除绑定关系,并重新绑定到物理 CPU2。这种绑定关系的变化称为 VMCS 的“迁移(Migration)”。

VT-x 提供了两条指令用于 VMCS 的绑定与解除绑定。

- VMPTRLD <VMCS 地址>: 将指定的 VMCS 与执行该指令的物理 CPU 绑定。
- VMCLEAR: 将执行该指令的物理 CPU 与它的 VMCS 解除绑定。该指令会将物理 CPU 缓存中的 VMCS 结构同步到内存中去,从而保证 VMCS 和新的物理 CPU 绑定时,内存中的值是最新的。

VMCS 的一次迁移过程如下。

(1) 在 CPU1 上执行 VMCLEAR,解除绑定。

(2) 在 CPU2 上执行 VMPTRLD,进行新的绑定。

VT-x 定义了 VMCS 的具体格式和内容。规定它是一个最大不超过 4KB 的内存块,并要求是 4KB 对齐。描述了 VMCS 的格式,各域描述如下。

(1) 偏移 0 处是 VMCS 版本标识,表示 VMCS 数据格式的版本号。

(2) 偏移 4 处是 VMX 中止指示,VM-Exit 执行不成功时产生 VMX 中止,CPU 会在此处存入 VMX 中止的原因,以方便调试。

(3) 偏移 8 处是 VMCS 数据域,该域的格式是 CPU 相关的,不同型号的 CPU 可能使用不同格式,具体使用哪种格式由 VMCS 版本标识确定。

VMCS 块格式如表 5-1 所示。

表 5-1 VMCS 块格式

字节偏移	描 述
0	VMCS revision identifier
4	VMX-abort indicator
8	VMCS data(implementation-specific format)

VMCS 主要的信息存放在“VMCS 数据域”,VT-x 提供了两条指令用于访问 VMCS。

- VMREAD <索引>: 读 VMCS 中“索引”指定的域。

- VMWRITE <索引> <数据>：写 VMCS 中“索引”指定的域。

VT-x 为 VMCS 数据域的每个字段也定义了相应的“索引”，故通过上述两条指令也可以直接访问 VMCS 数据域中的各个域。

具体而言，VMCS 数据域包括下列 6 大类信息。

(1) 客户机状态域：保存客户机运行时，即非根模式时的 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 从该域恢复 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：提供 VM-Exit 原因和其他的信息。VM-Exit 信息域是只读的。

本节先介绍客户机状态域和宿主机状态域，其他域会在 5.2.4 节中介绍。

1. 客户机状态域

客户机状态域用于保存 CPU 在非根模式下运行时的状态。当发生 VM-Entry 时，CPU 自动将客户机状态域保存的状态加载到 CPU 中；当发生 VM-Exit 时，CPU 自动将 CPU 的状态保存回客户机状态域。

客户机状态域中首先包含了一些寄存器的值，这些寄存器是必须由 CPU 进行切换的，如段寄存器、CR3、IDTR 和 GDTR。CPU 通过这些寄存器的切换来实现客户机地址空间和 VMM 地址空间的切换。客户机状态域中并不包括通用寄存器和浮点寄存器，它们的保存和恢复由 VMM 决定，可提高效率和增强灵活性（见第 6 章 6.3.3 节关于上下文切换的三个例子）。客户机状态域包含的寄存器如下。

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

除了上述寄存器外，客户机状态域中还包含了一些 MSR 的内容。这些 MSR 既可以由处理器进行切换，也可以由 VMM 进行切换。由谁切换，可以通过 VMCS 的一些控制域设定。这些 MSR 包括 IA32_SYSENTER_CS、IA32_SYSENTER_ESP 和 IA32_SYSENTER_EIP 等。

除此之外，客户机状态域还包含了一些非寄存器内容，主要用于精确模拟虚拟 CPU，例如中断状态域等。

2. 宿主机状态域

宿主机状态域用于保存 CPU 在根模式下运行时的 CPU 状态。宿主机状态域只在 VM-Exit 时被恢复，在 VM-Entry 时不用保存。这是因为宿主机状态域的内容通常几乎不

需要改变,例如 VM-Exit 的入口 RIP 在 VMM 整个运行期间都是不变的。当需要改变时,VMM 可以直接对该域进行修改,别忘了,VMCS 是保存在内存中的。

宿主机状态域只包含寄存器值,具体内容如下。

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

与客户机状态域相比,宿主机状态域没有 LDTR,正如操作系统内核通常不使用 LDT 一样,VMM 只需要使用 GDT 就足够了。

此外,当 VM-Exit 发生时,宿主机状态域中的 CS: RIP 指定了 VM-Exit 的人口地址,SS、RSP 指定了 VMM 的栈地址。

5.2.3 VMX 操作模式

5.2.1 节介绍了 VMX 操作模式的基本概念,本节及下节进行详细论述。

作为传统 IA32 架构的扩展,VMX 操作模式这个功能在默认情况下是关闭的,因为传统的操作系统并不需要使用这个功能。当 VMM 需要使用这个功能时,可以使用 VT-x 提供的新指令来打开与关闭这个功能,参见图 5-3。

- VMXON: 打开 VMX 操作模式。
- VMXOFF: 关闭 VMX 操作模式。

描述了开启/关闭 VMX 的过程,以及 VMX 开启情况下,VMM 和客户软件的交互操作。

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

(2) VMM 执行 VMLAUNCH 或 VMRESUME 指令产生 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发生了中断或异常,VM-Exit 被触发而陷入到 VMM,CPU 切换到根模式。VMM 根据 VM-Exit 的原因做相应处理,然后转到步骤(2)继续运行客户机。

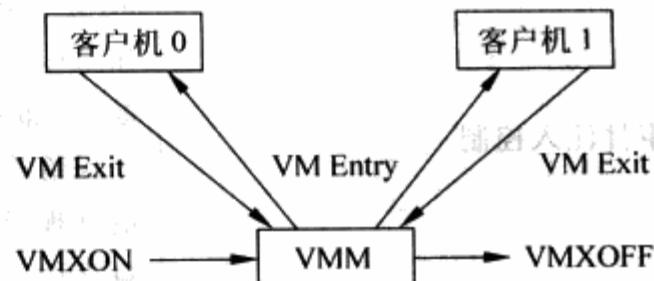


图 5-3 VMX 操作模式

(4) 如果 VMM 决定退出，则执行 VMXOFF 关闭 VMX 操作模式。

下面根据上述流程，介绍 VMX 操作模式的具体细节。

5.2.4 VM-Entry/VM-Exit

VMM 在机器加电引导后，会进行类似操作系统一样的初始化工作，并在准备就绪时通过 VMXON 指令进入根模式。在创建客户机时，VMM 会通过 VMLAUNCH 或 VMRESUME 指令切换到非根模式运行客户机，客户机引起 VM-Exit 后又切换回根模式运行 VMM。本节介绍这两种 VMX 操作模式切换的细节。

1. VM-Entry

VM-Entry 是指 CPU 由根模式切换到非根模式，从软件角度看，是指 CPU 从 VMM 切换到客户机执行。这个操作通常由 VMM 主动发起。在发起之前，VMM 会设置好 VMCS 相关域的内容，例如客户机状态域、宿主机状态域等，然后执行 VM-Entry 指令。

VT-x 为 VM-Entry 提供了两条指令。

- VMLAUNCH：用于刚执行过 VMCLEAR 的 VMCS 的第一次 VM-Entry。
- VMRESUME：用于执行过 VMLAUNCH 的 VMCS 的后续 VM-Entry。

VM-Entry 的具体行为由 VM-Entry 控制域规定，该域的具体定义如表 5-2 所示。

表 5-2 VMCS VM-Entry 控制域

域	描述
IA-32e mode guest	在支持 Intel 64 架构的处理器上，此位决定了在 VM-Entry 后处理器是否处于 64 位模式。当客户机处于 64 位模式时，需要打开这一位。
MSR VM-Entry 控制	在 VM-Entry 时，VMM 可以指示 CPU 在切换到客户机环境之前正确地装载 MSR 的值，这可以通过两个 VMCS 寄存器来指定：VM-Entry MSR-load count(指定了要装载的 MSR 的数目)和 VM-Entry MSR-load address(指定了要装载的 MSR 区域的物理地址)。
事件注入控制	在 VM-Entry 时，如果需要，VMM 可以操作 VMCS 中相关的字段(如 VM-Entry Interruption-Information 字段，VM-Entry exception error code 等)来向客户机虚拟 CPU 注入一个事件(Event Injection)。这里的“事件”包括同步的异常(Exception)和异步的中断(外部中断和 NMI)。例如，当客户机虚拟 CPU 意外地试图访问一个保留的虚拟 MSR 寄存器从而导致 VM-Exit 后，就可以在 VM-Exit 的处理函数中向 VM-Entry Interruption-Information 写入需要的信息来注入 #GP(General Protection fault)。再如，我们模拟的虚拟设备在一个 DMA 操作结束后需要向客户机注入虚拟中断时，也是通过写 VMCS 中这个字段来实现的。

VM-Entry 控制域中的“事件注入控制”用到了 VM-Entry Interruption-Information 字段,表 5-3 进一步列出了该字段的格式。每次 VM-Entry 时,在切换到客户机环境后即将执行客户机指令前,CPU 会检查这个 32 位字段的最高位(即 bit31)。如果为 1,则根据 bit10: 8 指定的中断类型和 bit7: 0 指定的向量号在当前的客户机中引发一个异常、中断或 NMI。此外,如果 bit11 为 1,表示要注入的事件有一个错误码(如 Page Fault 事件),错误码由另一个 VMCS 的寄存器 VM-Entry exception error code 指定。注入的事件最终是用客户机自己的 IDT 里面指定的处理函数来处理的。这样,在客户机虚拟 CPU 看来,这些事件就和没有虚拟化的环境里面对应的事件没有任何区别。

表 5-3 VM-Entry Interruption-Information 字段的格式

位	描 述
7: 0	中断或异常向量(Vector of interrupt or exception)
10: 8	中断类型(Interruption type) 0: 外部中断(External interrupt) 1: 保留(Reserved) 2: 非屏蔽中断(Non-maskable interrupt, NMI) 3: 硬件异常(Hardware exception) 4: 软件中断(Software interrupt) 5: 特权软件异常(Privileged software exception) 6: 软件异常(Software exception) 7: 保留(Reserved)
11	错误代码传递(0 = 不传递,1 = 传递)
30: 12	保留(Reserved)
31	合法(Valid)

2. VM-Entry 的过程

当 CPU 执行 VMLAUNCH/VMRESUME 进行 VM-Entry 时,处理器要进行下面的步骤。

- (1) 执行基本的检查来确保 VM-Entry 能开始。
- (2) 对 VMCS 中的宿主机状态域的有效性进行检查,以确保下一次 VM-Exit 发生时可以正确地从客户机环境切换到 VMM 环境。
- (3) 检查 VMCS 中客户机状态域的有效性;根据 VMCS 中客户机状态域区域来装载处理器的状态。
- (4) 根据 VMCS 中 VM-Entry MSR-load 区域装载 MSR 寄存器。
- (5) 根据 VMCS 中 VM-Entry 事件注入控制的配置,可能需要注入一个事件到客户机中。

第(1)~(4)步的检查如果没有通过,CPU 会报告 VM-Entry 失败,这通常意味着 VMCS 中某些字段的设置有错误。如果所有这些步骤都正常通过了,处理器就会把执行环

境从 VMM 切换到客户机环境,开始执行客户机指令。

5.2.5 VM-Exit

VM-Exit 是指 CPU 从非根模式切换到根模式,从客户机切换到 VMM 的操作。引发 VM-Exit 的原因很多,例如在非根模式执行了敏感指令、发生了中断等。处理 VM-Exit 事件是 VMM 模拟指令、虚拟特权资源的一大任务。在本节中,介绍 VM-Exit 相关的域以及 VM-Exit 的具体流程。

1. 非根模式下的敏感指令

当成功执行 VM-Entry 之后,CPU 就进入了非根模式。前面提到,敏感指令如果运行在 VMX 非根操作模式,其行为可能会发生变化。具体来说有如下三种可能。

(1) 行为不变化,但不引起 VM-Exit: 这意味着虽然是敏感指令,但它不需要被 VMM 截获和模拟,例如 SYSENTER 指令。

(2) 行为变化,产生 VM-Exit: 这就是典型需要截获并模拟的敏感指令。

(3) 行为变化,产生 VM-Exit 可控: 这类敏感指令是否产生 VM-Exit,可以通过 VM-Execution 域控制(见下节)。出于优化的目的,VMM 可以让某些敏感指令不产生 VM-Exit,以减小模式切换带来的上下文开销。

由此可见,使用 VT-x 技术实现的 VMM,并不需要对所有敏感指令进行模拟,这大大减小了 VMM 实现的复杂性。VM-Execution 域的存在又为 VMM 的实现带来了灵活性,下一节将对该域进行介绍。

2. VM-Execution 控制域

VM-Execution 控制域用来控制 CPU 在非根模式运行时的行为,根据虚拟机的实际应用,VMM 可以通过配置 VM-Execution 控制域达到性能优化等目的。VM-Execution 控制域主要控制三个方面。

(1) 控制某条敏感指令是否产生 VM-Exit,如果产生 VM-Exit,则由 VMM 模拟该指令。

(2) 在某些敏感指令不产生 VM-Exit 时,控制该指令的行为。

(3) 异常和中断是否产生 VM-Exit。

表 5-4 列举了一些典型的 VM-Execution 控制域,在 5.3 节读者可以看到该域是如何被使用到 CPU 的虚拟化实现中的。

表 5-4 VM-Execution 控制域

字 段	描 述
External-interrupt exiting	控制外部中断是否产生 VM-Exit: 1: 外部中断触发 VM-Exit。 0: CPU 查找客户机 IDT 表直接把中断递交给客户操作系统

续表

字 段	描 述
HLT exiting	控制 HLT 指令是否产生 VM-Exit: 1: 客户机执行 HLT 指令会触发 VM-Exit。客户机执行 HLT 指令通常意味着客户操作系统处于空闲状态,此时 VMM 可以暂时挂起客户机,去运行其他的客户机,直到某种条件(如虚拟中断)唤醒客户机为止。 0: 不引起 VM-Exit
INVLPG exiting	控制 INVLPG 指令是否产生 VM-Exit: 1: 客户机执行 INVLPG 指令产生 VM-Exit。 0: 不产生 VM-Exit
WBINVD exiting	控制 WBINVD 指令是否产生 VM-Exit: 1: 客户机执行 WBINVD 指令产生 VM-Exit。该位和 INVLPG exiting 一起用于帮助实现 MMU 的虚拟化。 0: 不产生 VM-Exit
RDPMC exiting	控制 RDPMC 指令是否产生 VM-Exit: 1: 客户机执行 RDPMC 指令产生 VM-Exit。此位用来帮助实现 performance monitor 虚拟化。 0: 不产生 VM-Exit
RDTSC exiting	控制 RDTSC 指令是否产生 VM-Exit: 1: 客户机执行 RDTSC 指令产生 VM-Exit。此位用来帮助实现 TSC 虚拟化。 0: 不产生 VM-Exit
CR8-load exiting	控制 CR8 LOAD 指令是否产生 VM-Exit: 1: 客户机装载 CR8 产生 VM-Exit。 0: 不产生 VM-Exit
CR8-store exiting	控制 CR8 STORE 指令是否产生 VM-Exit: 1: 客户机读取 CR8 产生 VM-Exit。 0: 不产生 VM-Exit
MOV-DR exiting	控制 MOV DR 指令是否产生 VM-Exit: 1: 客户机访问调试寄存器产生 VM-Exit。此位用来帮助实现调试寄存器的虚拟化。 0: 不产生 VM-Exit
Unconditional /O exiting	控制端口 I/O 访问是否产生 VM-Exit: 1: 客户机所有访问端口 I/O 指令触发 VM-Exit,如 IN、INS、INSB、INSW、INSD、OUT、OUTS、OUTSB、OUTSW 和 OUTSD。此位用来帮助实现输入输出设备虚拟化。 当 Use I/O bitmaps 为 1 时,此位被忽略

续表

字 段	描 述
Use I/O bitmaps	是否使用 I/O 位图来控制 I/O 指令： 1：客户机访问 I/O 端口时，只有当该端口在 I/O 位图对应位的值为 1 时才发生 VM-Exit。 0：不使用 I/O 位图
Use MSR bitmaps	是否使用 MSR 位图来控制 MSR 的访问： 1：客户机访问 MSR 时，只有当该 MSR 在 MSR 位图对应位的值为 1 时才发生 VM-Exit。 0：不使用 MSR 位图
Use TSC offset	当 RDTSC 为 1 时，提供客户机 TSC 和物理 CPU TSC 之间的偏移。见后面 5.3.3 节“VCPU 的硬件优化”
Exception bitmap	当客户机产生异常时，是否产生 VM-Exit。该字段为 32 位，某位置 1 表示该异常发生时引发一个 VM-Exit 陷入到 VMM。否则，直接由客户操作系统处理。缺页异常有特殊处理

3. VM-Exit 控制域

VM-Exit 控制域规定了 VM-Exit 发生时 CPU 的行为，表 5-5 描述了该域的内容。

表 5-5 VM-Exit 控制域

字 段	含 义
Host Address Space	在支持 Intel 64 架构的处理器上，此位决定了在下一次 VM-Exit 后处理器是否处于 64 位模式。64 位的 VMM 通常需要打开这一位
Acknowledge interrupt on exit	该位控制当一个外部中断引起 VM-Exit 时，是否应答中断控制器。 1：应答。 0：不应答
VM-Exit MSR-store count	指定 VM-Exit 发生时，CPU 要保存的 MSR 数目
VM-Exit MSR-store address	指定了要保存的 MSR 区域的物理地址
VM-Exit MSR-load count	指定 VM-Exit 发生时，CPU 要装载的 MSR 数目
VM-Exit MSR-load address	指定了要装载的 MSR 区域的物理地址

4. VM-Exit 信息域

VMM 除了要通过 VM-Exit 控制域来控制 VM-Exit 的行为外，还需要知道 VM-Exit 的相关信息（如退出原因）。VM-Exit 信息域满足了这个要求，其提供的信息可以分为如下 4 类。

(1) 基本的 VM-Exit 信息，包括如下内容。

① Exit Reason：提供了 VM-Exit 的基本原因（如表 5-6 所示）。

表 5-6 Exit Reason 字段格式

字 段	描 述
Basic exit reason	VM-Exit 的基本原因,如果 VM-Entry failure 为 1,该字段表示 VM-Entry 失败的原因
VM-Exit from VMX root operation	该位为 1,表示一次 VM-Exit 发生在 CPU 处于根模式时
VM-Entry failure	该位为 1,表示一次 VM-Entry 失败了

② Exit qualification: 提供 VM-Exit 的进一步原因。这个字段的值根据 VM-Exit 基本退出原因的不同而不同。例如,对于因为访问 CR 寄存器导致的 VM-Exit,Exit qualification 提供的信息包括:是哪个 CR 寄存器、访问类型是读还是写、访问的内容等。同样的,VT-x 规范也完整地定义了所有 VM-Exit 退出原因所对应的 Exit qualification。对于某些不需要额外信息的退出原因,没有相应的 Exit qualification 的定义。

(2) 事件触发导致的 VM-Exit 的信息。事件是指外部中断、异常(包括 INT3/INTO/BOUND/UD2 导致的异常)和 NMI。对于此类 VM-Exit, VMM 可以通过 VM-Exit interruption information 字段和 VM-Exit interruption error code 字段获取额外信息,例如事件类型、事件相关的向量号等。

(3) 事件注入导致的 VM-Exit 的信息。一个事件在注入客户机时,可能由于某种原因暂时不能成功,而触发 VM-Exit。此时, VMM 可以从 IDT-vectoring information 字段和 IDT-vectoring error code 中获取此类 VM-Exit 的额外信息,例如事件类型、事件向量号等。

(4) 执行指令导致的 VM-Exit 的信息。除了第一类中列出的信息外,客户机在执行敏感指令导致 VM-Exit 时,VMCS 中还有三个字段可以提供额外的信息。Guest linear address 字段给出了导致 VM-Exit 指令的客户机线性地址,VM-Exit instruction length 字段给出了该指令的长度,VM-Exit instruction information 字段给出了当该指令为 VMX 指令时的额外信息。

5. VM-Exit 的具体过程

了解 VM-Exit 需要的各种数据域、控制域后,介绍一下整个 VM-Exit 的大致流程。当一个 VM-Exit 发生时,依次执行下列步骤。

(1) CPU 首先将此次 VM-Exit 的原因信息记录到 VMCS 相应的信息域中,VM-Entry interruption-information 字段的有效位(bit31)被清零。

(2) CPU 状态被保存到 VMCS 客户机状态域。根据设置,CPU 也可能将客户机的 MSR 保存到 VM-Exit MSR-store 区域。

(3) 根据 VMCS 中宿主机状态域和 VM-Exit 控制域中的设置,将宿主机状态加载到 CPU 相应寄存器。CPU 也可能根据 VM-Exit MSR-store 区域来加载 VMM 的 MSR。

(4) CPU 由非根模式切换到了根模式,从宿主机状态域中 CS: RIP 指定的 VM-Exit 入

口函数开始执行。

在 VMM 处理完 VM-Exit 后,会通过 VMLAUNCH/VMRESUME 指令发起 VM-Entry 进而重新运行客户机。当下一次 VM-Exit 发生后,又会重复上述处理流程。虚拟化的所有内容就在 VMM→客户机→VMM→……的不断切换中完成。

5.3 CPU 虚拟化的实现

5.3.1 概述

通过前面章节的学习,读者应该了解了处理器虚拟化的概念。与软件虚拟化技术不同的是,使用 Intel VT-x 的 VMM 在处理器虚拟化的实现上更加简单和高效。

和软件虚拟技术用“CPU 执行环境处理器”来描述虚拟 CPU 类似,硬件虚拟化使用 VCPU(Virtual CPU)描述符来描述虚拟 CPU。VCPU 描述符类似操作系统中进程描述符(或进程控制块),本质是一个结构体,通常由下列几部分构成。

- ① VCPU 标识信息: 用于标识 VCPU 的一些属性,例如 VCPU 的 ID 号,VCPU 属于哪个客户机等。
- ② 虚拟寄存器信息: 虚拟的寄存器资源,在使用 Intel VT-x 的情况下,这些内容包含在 VMCS 中,例如客户机状态域保存的内容。
- ③ VCPU 状态信息: 类似于进程的状态信息,标识该 VCPU 当前所处的状态,例如睡眠、运行等,主要供调度器使用。
- ④ 额外寄存器/部件信息: 主要指未包含在 VMCS 中的一些寄存器或 CPU 部件。例如浮点寄存器和虚拟的 LAPIC 等。
- ⑤ 其他信息: 用于 VMM 进行优化或存储额外信息的字段,例如存放该 VCPU 私有数据的指针等。

由此可见,Intel VT-x 情况下的 VCPU 可以划分成两个部分,一个是以 VMCS 为主由硬件使用和更新的部分,这主要是虚拟寄存器;一个是除 VMCS 之外,由 VMM 使用和更新的部分,主要指 VMCS 以外的部分。图 5-4 展示了 VCPU 的构成。

当 VMM 创建客户机时,首先要为客户机创建 VCPU,整个客户机的运行实际上可以看作是 VMM 调度不同的 VCPU 运行。下面就以 VCPU 的创建—运行—退出为主线,介绍使用 Intel VT-x 技术的 CPU 虚拟化的实现。

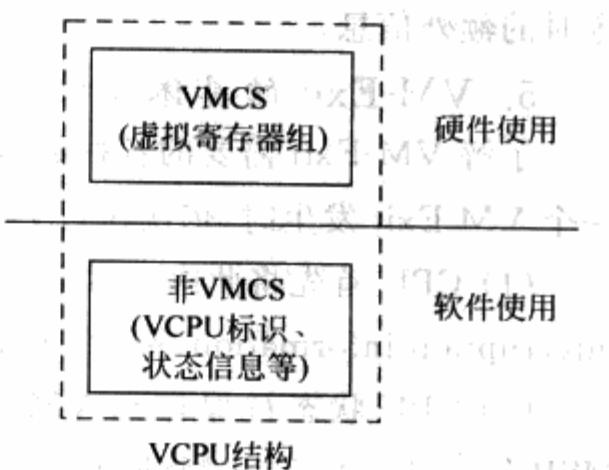


图 5-4 VCPU 结构

5.3.2 VCPU 的创建

创建 VCPU 实际上是创建 VCPU 描述符,由于本质上 VCPU 描述符是一个结构体,因此创建 VCPU 描述符简单来说就是分配相应大小的内存。VCPU 描述符包含的内容很多,通常会被组织成多级结构,例如第一级结构体可以是各个平台通用的内容,中间包含一个指针指向第二级结构体,包含平台相关的内容,如图 5-4 所示。对于这样的多级结构,需要为每一级结构体相应地分配内存。

VCPU 描述符在创建之后,需要进一步初始化才能使用。物理 CPU 在上电之后,硬件会自动将 CPU 初始化为特定的状态。VCPU 的初始化也是一个类似的过程,将 VCPU 描述符的各个部分置成可用的状态。通常初始化包含如下内容。

(1) 分配 VCPU 标识:首先要标识该 VCPU 属于哪个客户机,再为该 VCPU 分配一个在客户机范围内唯一的标识。

(2) 初始化虚拟寄存器组:主要指初始化 VMCS 相关域。这些寄存器的初始化值通常是根据物理 CPU 上电后各寄存器的值设定的。

(3) 初始化 VCPU 状态信息:设置 VCPU 在被调度前需要配置的必要标志。具体情况依据调度器的实现决定。

(4) 初始化额外部件:将未被 VMCS 包含的虚拟寄存器初始化为物理 CPU 上电后的值,并配置虚拟 LAPIC 等部件。

(5) 初始化其他信息:根据 VMM 的实现初始化 VCPU 的私有数据。

VMCS 的创建与初始化

VMCS 的创建与初始化是支持 VT-x 的 VCPU 创建的重要组成部分,这里再详细说明一下。

VMCS 在分配时,只需要分配一块 4KB 大小,并对齐到 4KB 边界的内存即可。初始化则需要根据 VT-x 的定义,对于前面所列的 VMCS 相关内容进行初始化,基本思想是根据物理 CPU 初始化的定义,提供一个和物理 CPU 初始后类似的状态。此外,根据 VMM 的 CPU 虚拟化策略,设置相应的 VMCS 控制位。

(1) 客户机状态域:这个状态域描述了 VCPU 运行时的状态,因此,初始化的取值基本上是参考物理 CPU 初始后的状态。例如,物理 CPU 加电后会通过复位地址跳转到 BIOS 执行,那么 Guest RIP 字段可直接设置为虚拟机 Guest BIOS 的起始指令地址。

(2) 宿主机状态域:这个状态域描述了发生 VM-Exit 时,CPU 切换到 VMM 时的寄存器的值,因此,初始化的取值是参考 VMM 运行时的 CPU 的状态。例如,HOST CS、HOST DS 等字段的取值是 VMM 运行时的段寄存器的值;HOST CR0、HOST CR3 等字段的取值是 VMM 运行时控制寄存器的值,通常是保护模式的、开页的。此外,HOST RIP 字段通常被设置为 VMM 中 VMX Exit 处理函数(VMX Exit Handler)的入口。

(3) VM-Execution 控制域:这个域控制 VCPU 运行时的一些行为,如执行某些敏感指

令是否发生 VM-Exit。因此,这个域的取值主要取决于 VMM 对于相应敏感指令的虚拟化策略。举例来说,对于 IN/OUT 指令,如果 VMM 允许客户机软件直接访问某些 I/O 端口,那么 VMM 就会将 Use I/O bitmaps 位置为 1,并且在 I/O bitmap 中将相应的 I/O 端口所对应的位置为 0,这样,客户机软件访问这些 I/O 端口时就不会发生 VM-Exit。VM-Execution 控制域给 VMM 带来了很大的灵活性,允许 VMM 做很多优化,后面还会做进一步的介绍。

(4) VM-Entry 控制域:这个状态域主要在每次 VM-Entry 之前设置,因此在 VCPU 初始化时不需要特别设置。

(5) VM-Exit 控制域:这个状态域有两个字段 VMM 通常有兴趣去设置,一个是 Acknowledge interrupt on exit,有助于更快地响应外部中断;另一个是 Host Address Space,用于支持 IA32e 模式。

(6) VM-Exit 信息域:这个域的值由硬件自动更新,因此不需要初始化。

5.3.3 VCPU 的运行

VCPU 创建并初始化好之后,就可以通过调度程序被调度运行。调度程序会根据一定的策略算法来选择 VCPU 运行。具体的调度策略内容超出了本节的描述范围,本节主要描述在选定 VCPU 之后,如何将 VCPU 切换到物理 CPU 上运行。

1. 上下文切换

从第 2 章已经知道了上下文实际是一个寄存器的集合。这里的寄存器包括通用寄存器、浮点寄存器、段寄存器、控制寄存器以及 MSR 等。前面已经提到,在 Intel VT-x 的支持下,VCPU 的上下文可以分为两部分。故上下文的切换也分为由硬件自动切换(VMCS 部分)和 VMM 软件切换(非 VMCS 部分)两个部分。其中,硬件切换部分可以更好地保证 VMM 与客户机的隔离,但缺乏灵活性。软件切换部分则可以由 VMM 自己选择性地切换需要的上下文(例如,浮点寄存器就无须每次都切换),从而有更大的灵活性并节省切换的开销。

图 5-5 描述了 VT-x 支持的 CPU 上下文切换的过程。可以归纳为下列几个步骤。

(1) VMM 保存自己的上下文,主要是保存 VMCS 不保存的寄存器,即宿主机状态域以外的部分。

(2) VMM 将保存在 VCPU 中的由软件切换的上下文加载到物理 CPU 中。

(3) VMM 执行 VMRESUME/VMLAUNCH 指令,触发 VM-Entry,此时 CPU 自动将 VCPU 上下文中 VMCS 部分加载到物理 CPU,CPU 切换到非根模式。

此时,物理 CPU 已经处于客户机的运行环境了,rip/eip 也指向了客户机的指令,这样 VCPU 就被成功调度并运行了。

上下文切换次数频繁会带来不小的切换开销,因此对上下文切换进行优化是很有必要的。和操作系统一样,VMM 也使用“惰性保存/恢复(Lazy Save/Restore)”的方法进行优化,其基本思想是尽量将寄存器的保存/恢复延迟到最后一刻,即其他 VCPU 或 VMM 需要

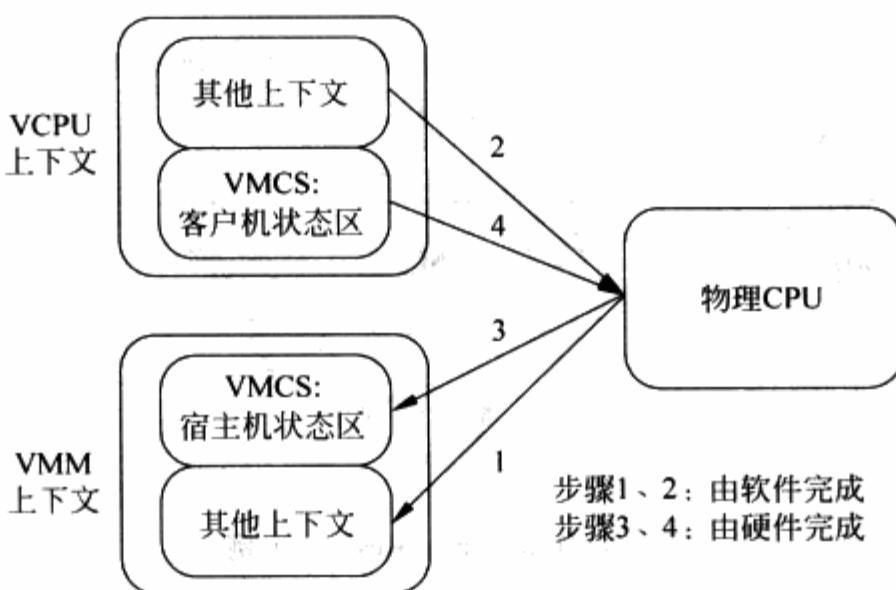


图 5-5 VM-Entry 中的上下文切换

用该寄存器的时候再保存/恢复。这种方法能够减少很多不必要的寄存器保存/恢复，提高上下文切换的效率。具体来说，VMM 通过考察资源的使用情况来实现“惰性保存/恢复”。

- (1) 对于 VMM 需要使用的寄存器，每次 VCPU 和 VMM 切换时都要保存/恢复。
- (2) 对于 VMM 没有使用的寄存器，如果 VMM 无法知道 VCPU 是否在最近的执行中曾经修改了这个寄存器(如扩展通用寄存器 DR6)，那么在 VCPU 和 VMM 切换时，不需要对这个寄存器进行保存和恢复。但是，当 VMM 进行不同的 VCPU 切换时，例如使一个 VCPU 睡眠并调度另一个 VCPU 运行，需要每次都保存和恢复这个寄存器。
- (3) 对于 VMM 没有使用的寄存器，如果 VMM 可以知道客户机是否在最近的执行中修改了这个寄存器(如浮点寄存器)，还可以做进一步的优化。不仅在 VCPU 和 VMM 切换时，不需要对这个寄存器进行保存和恢复，即使切换不同的 VCPU，也不需要每次都保存/恢复，而是根据需要进行。

举一个简单的例子来说明这种情况。如图 5-6 所示，VCPU1、VCPU2 和 VCPU3 按照顺序调度到物理 CPU 上执行，即 VCPU1 先执行，其次 VCPU2，最后 VCPU3。其中，VCPU1 和 VCPU3 在执行过程中会使用浮点寄存器，而 VCPU2 不会。VMM 了解到这种情况后，在从 VCPU1 调度到 VCPU2 时，只需保存 VCPU1 的浮点寄存器而无须加载 VCPU2 的；从 VCPU2 调度到 VCPU3 时，只需加载 VCPU3 的浮点寄存器而无须保存 VCPU2 的。这样就将原本两次保存/加载的工作减少为一次(保存 VCPU1 半次，加载 VCPU3 半次)。

2. VCPU 的硬件优化

相对于软件虚拟技术实现的 CPU 虚拟化，使用 Intel VT-x 技术的 VMM 可以采用多种方式对 VCPU 的实现进行优化。优化的目的，是尽可能少地在客户机和 VMM 之间切换，从而减少上下文切换的开销。Intel VT-x 提供的优化方法可以分为如下两种。

- (1) 无条件优化：指以往在软件虚拟技术下必须陷入到 VMM 中的敏感指令，通过 Intel

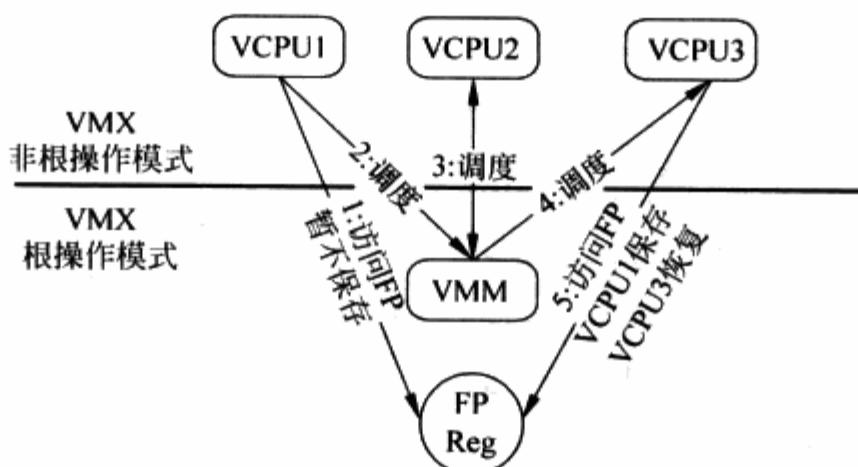


图 5-6 Lazy Save/Restore 示例

VT-x 已可以在客户机中直接执行。如后面将看到的 CR2 访问、SYSENTRY/SYSEXIT 指令。

(2) 条件优化：指通过 VMCS 的 VM-Execution 控制域，可以配置某些敏感指令是否产生 VM-Exit 而陷入到 VMM 中。如 CR0、TSC 的访问。

下面举几个例子来说明 Intel VT-x 带来的优化技术。

1. 访问 CR0

CR0 是一个控制寄存器，控制处理器的状态，如启动保护模式、打开分页机制。操作 CR0 的指令有 MOV TO CR0、MOV FROM CR0、CLTS 和 LMSW，这些指令必须在特权级 0 执行，否则产生保护异常。

在纯软件虚拟机中，客户操作系统是特权级 1、特权级 2 上执行 CR0 读写指令，因此所有的指令都产生保护异常，然后 VMM 模拟操作 CR0 指令的执行。

在硬件辅助的虚拟机中，虽然 CR0 的访问同样需要 VMM 模拟处理，但是 VT-x 提供了加速方法，能够减少因访问 CR0 所引起的 VM-Exit 的次数。首先，VMCS 的“VM-Execution 控制域”中的 CR0 read shadow 字段用来加速客户机读 CR0 的指令。每次客户机试图写 CR0 时，该字段都会自动得到更新，保存客户机要写的值。这样，客户机所有读 CR0 的指令都不用产生 VM-Exit，CPU 只要返回 CR0 read shadow 的值即可。其次，VMCS 的“VM-Execution 控制域”的 CR0 guest/host Mask 字段提供了客户机写 CR0 指令的加速。该字段每一位和 CR0 的每一位对应，表示 CR0 对应的位是否可以被客户软件修改。若为 0，表示 CR0 中对应的位可以被客户软件修改，不产生 VM-Exit；若为 1，表示 CR0 中对应的位不能被客户软件修改，如果客户软件修改该位，则产生 VM-Exit。

同样的机制被用于加速 CR4 的访问。该优化属于条件优化。

2. 访问 TSC

在纯虚拟机软件中，因为读取 TSC 可以在任何特权级别执行，VMM 必须想办法截获 TSC 读取指令。

在硬件辅助的虚拟机中，当“VM-Execution 控制域”中 RDTSC exiting 字段为 1 时，客

户软件执行 RDTSC 产生 VM-Exit,由 VMM 模拟该指令。客户读取 TSC 在某些操作系统里是一个非常频繁的操作,为了提高效率,VT-x 提供了下面的硬件加速。当 VMCS 中 RDTSC exiting 为 1 并且 Use TSC offset 为 1 时,硬件加速有效。VMCS 中 TSC 偏移量表示该 VMCS 所代表的虚拟 CPU TSC 相对于物理 CPU TSC 的偏移,即虚拟 TSC=物理 TSC+TSC 偏移量。当客户软件执行 RDTSC 时,处理器直接返回虚拟 TSC,不产生 VM-Exit。这样,对 TSC 的虚拟化只需在适时更新 VMCS 中 TSC 偏移量即可,不需要每次 TSC 访问都产生 VM-Exit,大大提高了 TSC 访问的效率。该优化属于条件优化。

3. GDTR/LDTR/IDTR/TR 的访问

在纯软件虚拟机中,客户操作系统是运行在特权级 1、特权级 2 上,执行 LGDT、LIDT、LLDT 和 LTR 指令,会产生保护异常,需要 VMM 模拟这些指令的执行。在模拟的过程中,对于不同的情况,还有很多复杂的处理。例如,客户操作系统在 GDT 中,为自身内核段设置的描述符的 DPL 是 0。由于它本身运行在非特权级 0 上,所以 VMM 要通过截获 LGDT 指令,对 GDT 中的描述符进行修改。同时,像 SGDT 这样的指令可以在任何特权级下执行,客户操作系统中的程序只需要读取 GDT 并判断描述符的 DPL 就知道自身运行在虚拟机环境下,这也是一个虚拟化的漏洞。

使用 Intel VT-x 技术,VMCS 为客户提供了一套 GDTR、IDTR、LDTR 和 TR,分别保存在客户机状态域和宿主机状态域中(宿主机状态域不包括 LDTR,前面说过,VMM 不需要使用它),由硬件切换。而客户机运行在非根模式的特权级 0,所以也无须对 GDT 表等做任何更改,客户机执行 LGDT 等指令也无须产生 VM-Exit。这样的优化大大降低了 VMM 的复杂度,使实现一个 VMM 变得简单。该优化属于无条件优化。

4. 读 CR2

在发生缺页异常时,CR2 保存产生缺页错误的虚拟地址。缺页错误处理程序通常会读取 CR2 获得产生该错误的虚拟地址。缺页错误是一个发生频率较高的异常,这决定了读取 CR2 是一个高频率的操作。读取 CR2 必须在特权级 0 上执行,否则产生保护错误。

在纯软件虚拟机中,客户操作系统是在特权级 1、特权级 2 上执行读取 CR2 指令,产生保护错误,需要 VMM 模拟该指令。

使用 Intel VT-x 技术,VM-Entry/VM-Exit 时会切换 CR2。并且,客户操作系统是在非根模式的特权级 0 执行读取 CR2 指令,不产生保护错误,故无须 VMM 模拟该指令。此外,如果客户机在特权级 0 以外的级别执行读 CR2 指令,会产生保护错误,该错误是否引发 VM-Exit 由 Exception bitmap 控制。该优化属于无条件优化。

5. SYSENTER/SYSEXIT

早期的系统调用是通过 INT 指令和 IRET 指令实现的。在当前主流的 IA32 CPU 中,Intel 推出了经过优化的 SYSENTER/SYSEXIT 指令以提高效率。现代操作系统都倾向于使用 SYSENTER/SYSEXIT 实现系统调用。

SYSENTER 指令要求跳转的目标代码段运行在特权级 0,否则产生保护错误。在软件

虚拟技术中,客户操作系统运行在特权级 1、特权级 2,当客户应用程序执行 SYSENTER 会产生保护错误,需要由 VMM 模拟 SYSENTER 指令。SYSEXIT 指令必须在特权级 0 执行,否则产生保护错误。和 SYSENTER 一样,SYSEXIT 在软件虚拟技术中必须由 VMM 模拟。

使用 Intel VT-x 技术,客户操作系统运行在非根模式的特权级 0,SYSENTER/SYSEXIT 都不会引起 VM-Exit,即客户操作系统的系统调用无须 VMM 干预而直接执行。该优化属于无条件优化。

6. APIC 访问控制

对于现代主流的支持 SMP 的操作系统来说,LAPIC 在中断的递交中扮演着一个非常重要的角色。LAPIC 里面有很多寄存器,通常操作系统会以 MMIO 方式来访问它们。在这些寄存器里,操作系统使用其中的 TPR(Task Priority Register)来屏蔽中断优先级小于或等于 TPR 的外部中断。

通过虚拟化客户机的 MMU,当客户机试图访问 LAPIC 时,会发生一个缺页异常类型的 VM-Exit,从而被 VMM 拦截到。VMM 经过分析,知道客户机正在试图访问 LAPIC 后,就会模拟客户机对 LAPIC 的访问。通常,对于客户机的每一个虚拟 CPU,VMM 都会分配一个虚拟 LAPIC 结构与之对应,客户机的 MMIO 操作不会真的影响物理的 LAPIC,而只是反映到相应的虚拟 LAPIC 结构里面。VMM 的这种模拟有相当大的开销,如果客户机的每一个 LAPIC 访问都导致一次缺页异常类型的 VM-Exit 并由 VMM 模拟的话,会严重影响到客户机的性能。

针对这种情况,VT-x 提供了硬件加速支持。可以设置 VMCS 中的 Use TPR shadow=1,Virtualize APIC accesses=1,设置 Virtual APIC page 为虚拟 LAPIC 结构的地址,同时修改 VCPU 页表,使得客户机访问 LAPIC 时不发生 Page Fault(这需要相应地设置 VMCS 中的 Virtual-APIC address 寄存器)。同时,对于那些暂时不能注入客户机的中断(如果有的话),还需要挑出优先级最高的那个(就是向量号最大的那个),将其优先级填入 VMCS 中的 TPR threshold 寄存器。

这样设置后,对于除了 TPR 以外的 LAPIC 寄存器的访问,客户机会直接发生 APIC-Access 类型的 VM-Exit。此时,CPU 可告知 VMM 客户机正试图访问哪个 LAPIC 寄存器,这可降低 VMM 对客户机此次访问的模拟开销;而客户机对 TPR 的读操作则可以直接从虚拟 LAPIC 结构中的相应偏移处读取而无须发生任何 VM-Exit。最后,客户机对 TPR 的写操作只在必要的时候(客户机把 TPR 减小到比 TPR threshold 还要小的时候)才发生 TPR-Below-Threshold 类型的 VM-Exit,这种情况下 VMM 可检测是否有虚拟中断可以注入客户机。

上面谈到 TPR 寄存器时,说是用 MMIO 方式来访问的,其实对于 64 位的 x86 平台,专门有一个特别的系统控制寄存器 CR8 被映射到了 TPR(读写 CR8 就等效于读写 TPR),64 位的客户机通常通过 CR8 寄存器来访问 TPR。当客户机试图访问 CR8 时,会发生一个

Control-Register-Accesses 类型的 VM-Exit。为了更快地模拟客户机对 CR8 的访问,除了上面提到的设置外,可以设置 VMCS 中的 CR8-load exiting=0 和 CR8-store exiting=0。这样,客户机读 CR8 时,CPU 可以从虚拟 LAPIC 结构中相应的偏移处直接返回正确的值而不会发生任何 VM-Exit; 当客户机写 CR8 时,只在必要的时候才发生 TPR-Below-Threshold 类型的 VM-Exit。

7. 异常控制

在软件虚拟化技术中,客户机产生的异常都会被 VMM 截获,由 VMM 决定如何处理,通常是注入给客户机操作系统。

使用 Intel VT-x 技术,可以用 Exception bitmap 配置哪些异常需要由 VMM 截获。对于不需要 VMM 截获的异常,可以将 Exception bitmap 中对应的位置 1,则异常发生时直接由客户机操作系统处理。这样的优化可以大大减少由客户机异常引起的 VM-Exit。该优化属于条件优化。

8. I/O 控制

在软件虚拟技术中,VMM 需要截获 I/O 指令来实现 I/O 虚拟化。但由于 I/O 指令通过设置可以在特权级 3 执行,截获 I/O 指令需要额外的处理。

使用 Intel VT-x 技术,可以通过 VMCS 的 Unconditional I/O exiting、Use I/O bitmaps、I/O bitmap 进行配置,选择性地让 I/O 访问产生 VM-Exit 而陷入 VMM 中。这样,对于不需要模拟的 I/O 端口,可以让客户机直接访问。该优化属于条件优化。

9. MSR 位图

x86 包括很多 MSR 寄存器,使用 Intel VT-x 和 I/O 控制一样,可以通过 use MSR bitmaps、MSR bitmap 来控制对 MSR 的访问是否触发 VM-Exit。该优化属于条件优化。

5.3.4 VCPU 的退出

上一节了解了 VCPU 如何被调度运行,也了解了针对 VCPU 运行时的一些优化。和进程一样,VCPU 作为调度单位不可能永远运行,总会因为各种原因退出,例如执行了特权指令、发生了物理中断等。这种退出在 VT-x 中表现为发生 VM-Exit。

对 VCPU 退出的处理是 VMM 进行 CPU 虚拟化的核心,例如模拟各种特权指令。本节介绍在使用 Intel VT-x 的情况下,VMM 是如何处理 VCPU 退出的。

图 5-7 描述了 VMM 处理 VCPU 退出的典型流程,可以归纳为下列几个步骤。

- (1) 发生 VM-Exit,CPU 自动进行一部分上下文的切换。见 6.2.7 节。
- (2) 当 CPU 切换到根模式开始执行 VM-Exit 的处理函数后,进行另一部分上下文的切换工作(见 6.3.3 节)。

根据 VM-Exit 信息域获得发生 VM-Exit 的原因,并分发到对应的处理模块处理。例如,原因是执行了特权指令,则调用相应指令的模拟函数进行模拟。

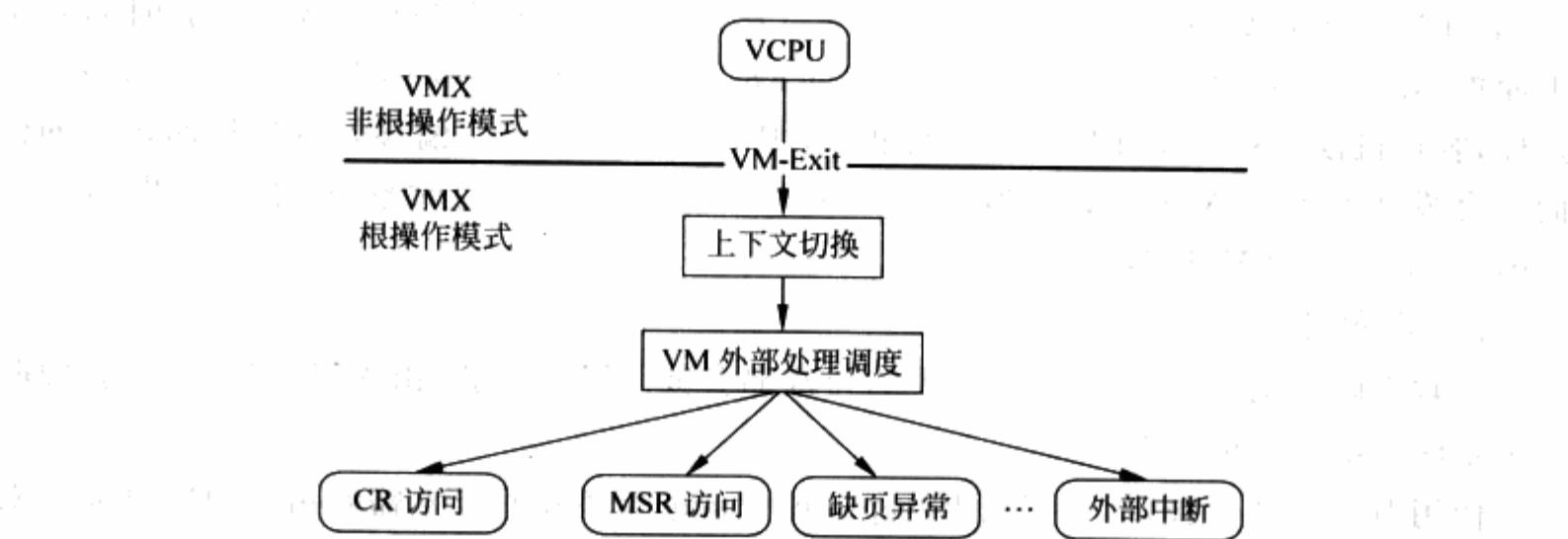


图 5-7 VM-Exit 的处理

图 5-7 列举了一些较为典型的 VCPU 退出的原因,总结起来,VCPU 退出的原因大体上有三类。

(1) 访问了特权资源,对 CR 和 MSR 寄存器的访问都属于这一类。

对于此类 VM-Exit, VMM 通过特权资源的虚拟化来解决。特权资源虚拟化的要点在于解决客户机与 VMM 在特权资源控制权的矛盾。即客户机认为自己完全拥有特权资源,可以自由读写,而特权资源的实际拥有者是 VMM,不能允许客户机自由读写。VMM 通过引入“虚拟特权资源”和“影子特权资源”来解决这个矛盾。“虚拟特权资源”是客户机所看到的特权资源,VMM 允许客户机自由地读写。“影子特权资源”是客户机运行时特权资源真正的值,通常是 VMM 在“虚拟特权资源”的基础上经过处理得到的,因此称其为“影子”。

图 5-8 以特权寄存器为例,展示了特权寄存器的虚拟化过程。当 VCPU 读特权寄存器时,VMM 将“虚拟寄存器”的值返回。例如,对于 MOV EAX,CR0 指令,VMM 将 Virtual CR0 的值赋给 EAX,然后 VM-Entry 返回。当 VCPU 写特权寄存器时,VMM 首先将值写入“虚拟寄存器”,然后根据“虚拟寄存器”的值以及虚拟化策略来更新“影子寄存器”,最后将“影子寄存器”的值应用到 VCPU 上,将值写入 VMCS“客户机状态域”的对应字段并且 VM-Entry 返回。这里的虚拟化策略是因特权虚拟器而异的,例如对于下面指令:

```

MOV EAX, 0x00000001
MOV CR0, EAX

```

假设原来 Virtual CR0=0x80000001,VMM 比较之后会发现客户机试图将 CR0 的第 31 位(CR0.PG: 页模式)清掉,即关掉 CPU 的页模式。为了实现内存的隔离,VT-x 是不允许客户机的页模式关掉的。因此,VMM 会将 Virtual CR0 按照客户机要求设置为 0x00000001,但是影子 CR0 依然设置为 0x80000001(相应 VMCS 中的 Guest CR0 字段也会被设置)。此外,VMM 会通知内存虚拟化模块有关客户机页模式的变化,内存虚拟化模块会做相关处理,如不再使用客户机的页表等。

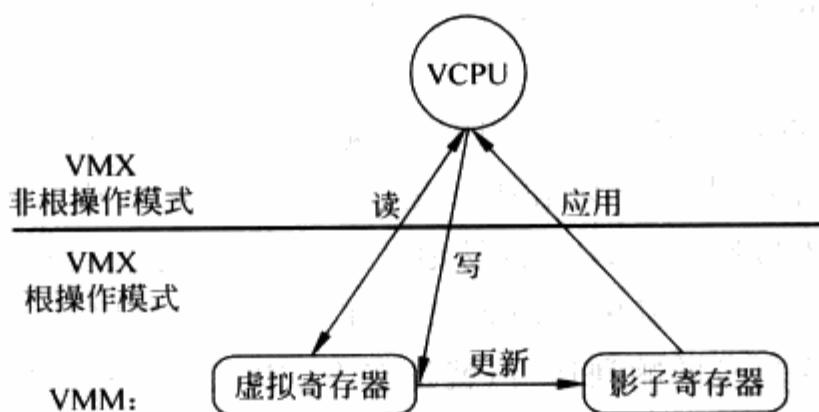


图 5-8 特权寄存器的虚拟化

(2) 客户机执行的指令引发了异常,例如缺页错误。

客户机指令导致的异常,很多是不需要虚拟化的,可以直接交由客户操作系统处理,例如“除 0 错误”、“溢出错误”和“非法指令”等异常。这些都可以通过前面提到的 Exception bitmap 设置。

对于需要虚拟化的异常,没有一个通用的方法,VMM 会对不同的异常做不同的处理。以缺页错误为例,VMM 会首先分析产生错误的原因:如果是因为访问 MMIO 地址导致的异常,则可以知道客户机在做 I/O 操作,VMM 会调用 I/O 虚拟化模块处理;如果是软件虚拟化章节提到的影子页导致的异常,VMM 会调用内存虚拟化模块处理;如果所有的原因都不是,那么就是客户机正常的缺页错误(即不需要 VMM 处理的缺页错误),该异常会被注入给客户机,由客户操作系统自己处理。

(3) 发生了中断。这可以分成两种情况,一种是发生了真正的物理中断;一种是客户机的虚拟设备发生了虚拟中断,并通过 VMM 提供的接口使客户机发生 VM-Exit。对于前者,VMM 首先读取 VMCS 的 VM-Exit interruption information 字段来获得中断向量号,然后调用 VMM 中对应的中断处理函数。对于后者,VMM 在感知到虚拟中断发生时,会用某种方法把该虚拟中断的目标 VCPU 拖到 VMM 中,常用的方法是发一个 IPI 给运行该 VCPU 的物理 CPU。然后,VMM 在 IPI 的处理函数中将该虚拟中断注入给客户机,由客户操作系统处理。

5.3.5 VCPU 的再运行

VMM 在处理完 VCPU 的退出后,会负责将 VCPU 投入再运行。从 VT-x 的角度来看,有几点需要额外考虑。

(1) 如果 VCPU 继续在相同的物理 CPU 上运行,可以用 VMRESUME 来实现 VM-Entry。VMRESUME 比 VMLAUNCH 更轻量级,执行效率更高。因此,作为优化,VMM 调度程序通常会尽量将 VCPU 调度在同一个物理 CPU 上。

(2) 如果由于某种原因(如负载均衡),VCPU 被调度程序迁移到了另外一个物理 CPU 上,那么 VMM 需要做如下几件事情。

① 将 VCPU 对应的 VMCS 迁移到另一个物理 CPU(见 6.2.2 节), 这通常可以由一个 IPI 中断实现。

② 迁移完成后, 在重新绑定的物理 CPU 上执行 VMLAUNCH 发起 VM-Entry。

此外, 在上一节中看到, 某些异常和中断是需要注入给客户机, 这也是在 VCPU 运行时进行的。通过“事件注入机制”, 可以很容易地让 VCPU 在运行后直接进入到相应的中断/异常处理函数中执行。整个虚拟化的内容就是在 VMM → 客户机 → VMM → …… 中完成的。这里再细化一下, 客户机的顺利运行, 就是在 VCPU 运行 → VCPU 退出 → VCPU 再运行 → …… 的过程中完成的。

5.3.6 进阶

前面几节介绍了 CPU 虚拟化的基本知识和流程, 本节引入一些高阶知识, 让读者对 CPU 虚拟化中一些较为复杂的部分有一个感性的认识。

1. CPU 模式的虚拟化

在第 2 章介绍了 CPU 的几种运行模式, 除此之外, 还有 64 位 CPU 用的 IA-32e 模式。其中, 保护模式又包括分页打开和分页关闭两种情况。在一个物理机器上, 可能会同时运行一个实模式的虚拟机、一个保护模式的虚拟机和一个 IA-32e 模式的虚拟机, 以及其他组合。为此, VMM 必须有模拟各种 CPU 运行模式的能力。

然而, 由于客户机物理地址空间和机器真实的物理地址空间并不相同, 且客户机物理地址空间占用的真实物理页面通常是不连续的, 因此, 目前 VT-x 技术要求物理 CPU 处于非根模式时, 分页机制必须是开启的, 而不考虑客户机当前运行的模式。也就是说, 即使客户机运行在实模式, 其所在物理 CPU 的分页机制也是开启的。由于实模式使用的内存访问模式和保护模式不同, VMM 需要大量的工作对客户机的实模式进行模拟。随着硬件技术的发展, 硬件很可能会直接支持客户机的实模式内存访问, 从而大大简化对客户机运行模式的虚拟化。

客户机看到 CPU 模式实际是 VCPU 中设置的模式, 即 VCPU 结构中 CR0 寄存器值反映的模式, 该模式和物理 CPU 的真正模式可能不同, 所以 CPU 模式的虚拟化包括如下两个方面。

- (1) 对标志 CPU 模式的控制寄存器(如 CR0 的 PE/PG 位)的虚拟化。
- (2) 对 CPU 运行环境的虚拟化, 如指令的操作数长度等。

当客户机运行在实模式时, 由于物理 CPU 运行在保护模式, 两者在指令、运行环境方面都不同, 通常 VMM 是对客户机的指令进行模拟执行。

当客户机运行在分页关闭的保护模式时, 同样, 物理 CPU 实际运行在分页开启的保护模式下。此时, VMM 要负责模拟客户机实模式的内存访问机制。

当客户机运行在分页开启的保护模式时, 和物理 CPU 的运行模式一样, 此时不需要就模式的虚拟化做额外的工作。

表 5-7 总结了客户机运行模式和物理 CPU 的实际模式之间的对应关系，并总结了模拟客户机运行模式的方法。需要指出的是，客户机运行在何种模式和 VMM 在何种模式运行没有必然关系。例如，基于对大内存支持的需求，可以将 VMM 运行在 IA32-e 模式下。在该 VMM 上，可以运行一个 IA-32 保护模式的虚拟机。当发生 VM-Exit 和 VM-Entry 引起客户机和 VMM 切换时，硬件会通过装载“客户机状态域”/“宿主机状态域”，而自动完成 CPU 模式的转换。

表 5-7 CPU 模式虚拟化总结

虚拟机认为的 CPU 模式	物理 CPU 实际运行模式	CPU 模式虚拟化手段
实模式	分页打开的保护模式	指令模拟
分页关闭的保护模式	分页打开的保护模式	VMM 提供额外的页表
分页打开的保护模式	分页打开的保护模式	不需要
IA-32e 模式	IA-32e 模式	不需要

2. 多处理器虚拟机

随着多核技术的发展，今天大部分的计算机都具备了多个 CPU。可以通过将客户机配置为多 CPU 的虚拟平台，来提高客户机的计算能力。所谓配置多 CPU 的虚拟平台，本质上就是给客户机分配多个 VCPU，并通过调度器让它们共享一个物理 CPU 分时执行或分散到多个物理 CPU 同时执行。这和操作系统中的多线程任务采用的是同样的思想。

与单 VCPU 客户机相比，多 VCPU 客户机在实现上还有几点需要注意。

(1) 多 VCPU 发现的问题。在物理机器上，操作系统需要知道平台所有 CPU 的信息，同样，需要让客户操作系统知道它所拥有的 VCPU 的信息，例如 VCPU 的个数、每个 VCPU 的 ID 号等。这些信息在物理平台上是通过 BIOS 提供的，例如 [18] 或者 [19] 中的 MADT Table。在虚拟环境下，客户机的虚拟 BIOS 负责这项工作。

(2) 多个 VCPU 初始化的问题。在物理平台上，多处理器的初始化通常会遵循一个规范，例如 IA32 手册 [16] 的 7.5 节 MULTIPLE-PROCESSOR (MP) INITIALIZATION 就定义了 IA32 平台上多处理器的初始化流程。硬件通常会选择一个 CPU 作为主 CPU，称为 BSP (Boot Strap Processor)，来执行 BIOS，其他的从 CPU，称为 AP (Application Processor)，处于 Wait-for-SIPI 状态，等待 BSP 发 SIPI (Start-up IPI) 唤醒它们之后再开始执行。客户机同样需要遵循这个规范，包括挑选一个 VCPU 作为 BSP 执行 BIOS，将其他 VCPU 置于 Wait-for-SIPI 状态等待 BSP 发 SIPI，只是这个挑选工作是由 VMM 来完成的。

(3) VCPU 的同步问题。在物理多 CPU 系统中，各个 CPU 都是同时在运行的。但是，对于拥有多个 VCPU 的客户机，在某一个时刻，可能一部分 VCPU 正在运行，一部分则处于睡眠或者阻塞的状态。这对于 VCPU 之间的通信和同步都造成了一些延时问题。例如，当客户机的两个 VCPU 都运行着竞争自旋锁的代码时，考虑下面这段两个 VCPU 通过自旋锁进行同步的代码：

```
acquire_spinlock(lock)
    critical section
release_spinlock(lock)
```

当 VCPU0 上运行的代码通过 `acquire_spinlock(lock)` 取得自旋锁, 进入临界区后, VCPU0 可能会被调度出去。其后, 当另一段运行在 VCPU1 上的代码尝试获取这个自旋锁时, 它将因为得不到自旋锁而不得不进行等待, 直到 VCPU0 被重新调度执行, 完成临界区并执行 `release_spinlock(lock)` 释放自旋锁。这样, VCPU1 因为 VCPU0 被调度出去的缘故, 额外增加了同步的时延。

为了解决这些 VCPU 间的通信和同步的延迟问题, 一种解决方案是对一个多 VCPU 客户机的多个 VCPU 进行群体调度(Gang Scheduling), 即它们要么同时在多个物理 CPU 上同时运行, 要么同时不运行。群体调度带来的限制是一个多 VCPU 客户机的 VCPU 个数不可以超过物理平台的物理 CPU 个数。

5.4 中断虚拟化

5.4.1 概述

在第 2 章, 已经介绍了现代计算机架构的中断系统, 对于今天拥有众多五花八门外设的计算机而言, 中断系统的作用是至关紧要的。与此对应, 在虚拟的环境下, 虚拟机有诸多的设备, 包括 VMM 模拟的虚拟设备和直接分配给客户机的物理设备, 这些设备都需要发送中断给 VCPU, 以便得到处理。因此, VMM 需要提供中断虚拟化的支持。

在介绍中断虚拟化之前, 再来回顾一下物理平台上的中断架构。通过第 2 章 2.4 节的介绍, 我们知道外部中断的流程如图 5-9 所示。首先, I/O 设备通过中断控制器(I/O APIC 或者 PIC)发出中断请求, 中断请求经由 PCI 总线发送到系统总线上, 最后目标 CPU 的 Local APIC 部件接收中断, CPU 开始处理中断。

在虚拟机的环境中, VMM 也需要为客户机操作系统展现一个与物理中断架构类似的虚拟中断架构。图 5-10 展现了虚拟机的中断架构。和物理平台一样, 每个 VCPU 都对应一个虚拟 Local APIC 用于接收中断。虚拟平台也包含了虚拟 I/O APIC 或者虚拟 PIC 用于发送中断。和 VCPU 一样, 虚拟 Local APIC、虚拟 I/O APIC 和虚拟 PIC 都是 VMM 维护的软件实体。当虚拟设备需要发送中断时, 虚拟设备会调用虚拟 I/O APIC 的接口发送中断。虚拟 I/O APIC 根据中断请求, 挑选出相应的虚拟 Local APIC, 调用其接口发出中断请求。虚拟 Local APIC 进一步利用 VT-x 的事件注入机制将中断注入到相应的 VCPU(见 5.3.5 节)。

由此可以看出, 中断虚拟化的主要任务是实现图 5-10 描述的虚拟机中断架构, 具体来说包括虚拟 PIC、虚拟 I/O APIC 和虚拟 Local APIC, 并且实现中断的生成、采集和注入的整个过程。

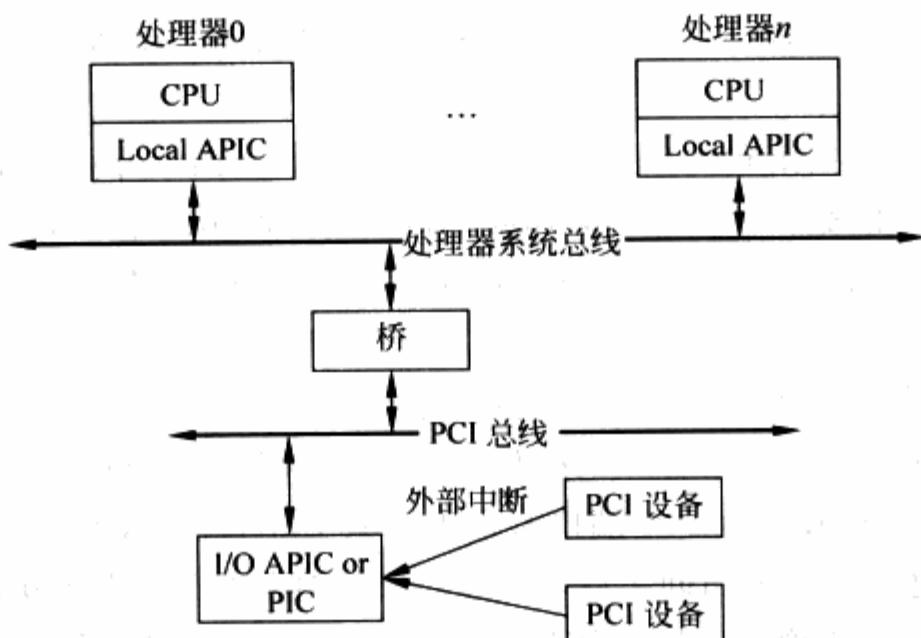


图 5-9 物理平台的中断架构

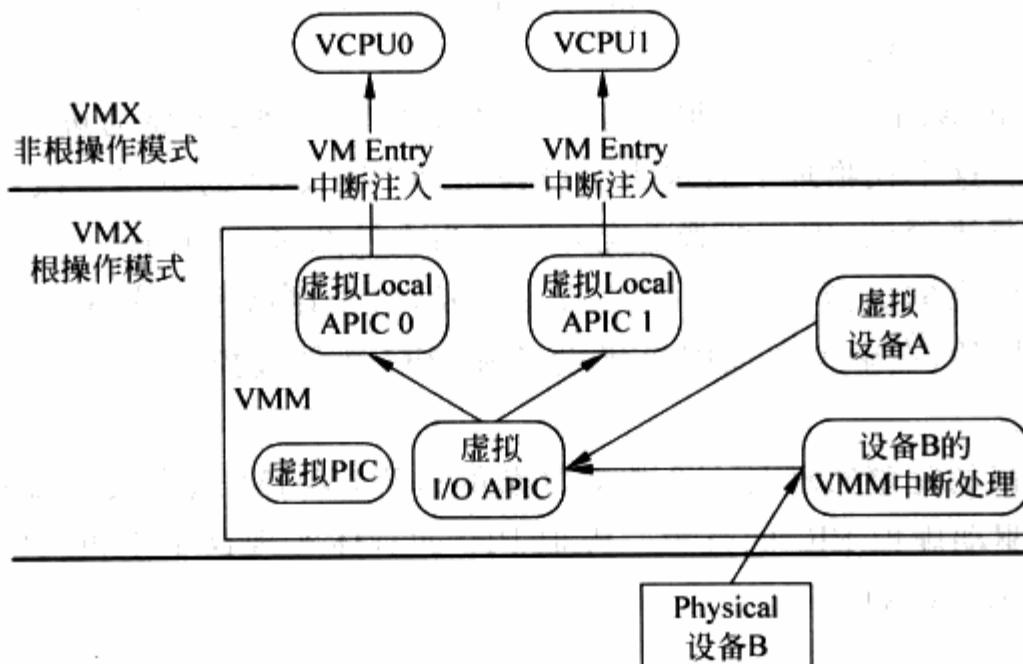


图 5-10 虚拟机中断架构

此外，PCI/PCIe 设备还支持另外一种中断方式 MSI，MSI 可以允许设备直接发送中断到 Local APIC，不需要通过中断控制器(I/O APIC)。MSI 虚拟化的原理和本节描述的原理是类似的，因此，MSI 虚拟化此处不再赘述。读者感兴趣的话，可以进一步参考 PCI/PCIe 规范中关于 MSI 的描述，以及 IA32 手册^[16]9.11 节 MESSAGE SIGNALLED INTERRUPTS 的相关内容。

5.4.2 虚拟 PIC

通过第 2 章的学习，了解了物理 PIC 的功能。PIC 本质上是芯片组的一个设备，参考文献[20]描述了 PIC 完整的规范。因此，虚拟 PIC 的实现，实际上和第 5 章中的设备模拟是

一样的,即根据 PIC 硬件规范,在软件上模拟出虚拟 PIC,为虚拟机提供和物理 PIC 一样的接口。

虚拟 PIC 首先要虚拟出和物理 PIC 一样的软件接口。PIC 为软件提供了如下接口用于操作 PIC。4 个初始化命令字(Initialization Command Words): ICW1~4,用于初始化操作;3 个操作命令字(Operation Command Words): OCW1~3,用于操作 PIC。

在 IA32 平台上,PIC 的 ICW1~4 和 OCW1~3 都是通过 I/O 端口访问的。因此,在 VT-x 的帮助下,VMM 很容易就可以实现这些接口的虚拟化。具体而言,这些接口是通过 I/O 端口 0x20/0x21 以及 0xA0/0xA1 来访问,因此,VMM 可以设置 VMCS 的 I/O bitmap 中的相应位,使得客户机在访问这些端口时发生 VM-Exit,便于 VMM 截获。

VMM 在截获这些接口的访问之后,下一步就是按照 PIC 硬件规范对这些接口的定义,实现相应的逻辑。举例来说,接口 OCW1 的功能是用于操作 IMR 寄存器,控制指定中断是否被屏蔽。因此,VMM 会分析客户机的 OCW1 命令,判断出是对哪个中断进行屏蔽或者解除屏蔽,VMM 继而在内部逻辑中记录指定中断是否被屏蔽。如果指定中断被屏蔽了,相应的虚拟中断就不会被提交。

此外,虚拟 PIC 除了为客户机提供正确的虚拟接口以外,还要为虚拟设备提供接口用于发送中断请求。这个在物理上表现为 I/O 设备和 PIC 之间的电气连线,在虚拟环境中由于设备和 PIC 都是虚拟的,因而两者的交互表现为直接的函数调用。

虚拟 PIC 最终会向虚拟 Local APIC 提交中断,这个在物理上表现为 PIC 和 CPU 之间的电气连线。同样的,在虚拟环境中由于设备和 PIC 都是虚拟的,因而两者的交互表现为直接的函数调用。

虚拟 PIC 接口的完整实现是一个相对复杂的过程,VMM 通常会为虚拟 PIC 维护一个内部的状态机来驱动虚拟 PIC 的行为。虚拟 PIC 的具体实现这里不再赘述,读者有兴趣可以参考 Xen 或者 KVM 的实现。

5.4.3 虚拟 I/O APIC

正如第 2 章的 2.4 节指出的那样,PIC 只适用于单 CPU 系统,对于多 CPU,必须通过 I/O APIC 来发送中断。因此,对于多 CPU 虚拟平台,必须实现虚拟 I/O APIC。

和虚拟 PIC 的实现类似,虚拟 I/O APIC 在 VMM 中也是一个虚拟设备。VMM 根据其硬件规范来实现虚拟设备。Intel ICH 系列 Datasheet 详细定义了 I/O APIC 的软件接口。

和 PIC 类似,虚拟 I/O APIC 也会根据硬件规范实现相应的接口内部逻辑,也会为虚拟设备提供接口用于发送中断请求。虚拟 I/O APIC 最后也是通过调用虚拟 Local APIC 的接口来提交中断。

操作系统通过 MMIO 的方式访问 I/O APIC,因此,VMM 的实现和虚拟 PIC 有所不同。VMM 会将虚拟 PIC 的 MMIO 地址对应的页表项置为“该页不存在”,因此,当客户机访问对应的 MMIO 寄存器时,就会发生原因为 Page Fault 的 VM-Exit。这样,VMM 就能

截获客户机对虚拟 I/O APIC 的访问,进而正确地虚拟化。

虚拟 I/O APIC 的具体实现这里不再赘述,读者可以参考 Xen 或者 KVM 的实现。

5.4.4 虚拟 Local APIC

Local APIC 是 CPU 上一个内部部件,负责接受中断。此外,还提供了产生中断的功能,例如 Local APIC Timer Interrupt 和处理器间中断 IPI。Intel 手册 [16] 的第 9 章 ADVANCED PROGRAMMABLE INTERRUPT CONTROLLER (APIC) 详细定义了 Local APIC 的规范。

和虚拟 PIC、虚拟 I/O APIC 一样,虚拟 Local APIC 在 VMM 中被实现为一个模拟设备。和 I/O APIC 一样,Local APIC 提供给软件的接口是 MMIO 寄存器,因此,VMM 也通过 Page Fault 来截获客户机对 Local APIC 的访问,进一步实现内部逻辑。

虚拟 Local APIC 的最主要功能是向 VCPU 注入中断。在 VT-x 的帮助下,虚拟 Local APIC 可以借助 VM-Entry 事件注入机制简单地实现这个功能。当然,在具体的实现中,还有几个情况需要额外处理,这个在后面的内容中会进一步描述。

5.4.5 中断采集

上面介绍了虚拟机中断架构的各个组件,下面介绍这些组件如何一起工作为客户机产生虚拟中断。这个过程包括两个部分:中断的采集和中断的注入。

中断的采集是指如何将虚拟机的设备中断请求送入虚拟中断控制器。在虚拟机环境里,客户机中断有两种可能的来源。

(1) 来自于软件模拟的虚拟设备,例如一个模拟出来的串口,可以产生一个虚拟中断。

(2) 来自于直接分配给客户机的物理设备的中断,例如一块物理网卡,可以产生一个真正的物理中断。

采集这两种中断的方法大不相同,前者比较简单,后者则相对复杂。

对于虚拟设备而言,它们是一个软件模块。当虚拟设备需要发出中断请求时,可以通过虚拟中断控制器提供的接口函数发出中断请求,例如使用虚拟 PIC 或者虚拟 I/O APIC 提供的接口。

采集直接分配给客户机的物理设备发出的中断请求要复杂得多。一个物理设备被直接分配给一个客户机,意味着当设备发生中断时,该物理中断的处理函数位于客户操作系统中。而在虚拟化环境中,物理中断控制器由 VMM 控制,且中断发生时 CPU 的 IDT 表通常不是客户机的 IDT 表,因此,物理中断需要首先由 VMM 的中断处理函数接收,再注入给客户机。

下面通过一个例子概要地介绍物理中断采集过程。

(1) 物理设备发生中断,假定设备的 IRQ 号为 14,对应的中断向量号为 0x41。

(2) CPU 收到中断,执行标准的中断处理流程,例如应答 PIC、过中断门中断自动屏蔽等。最后,CPU 跳转到 IDT 表中 0x41 表项所指定的处理函数。注意,该处理函数是 VMM

提供的,其目的是将物理中断注入给客户机。

(3) VMM 的中断处理函数对中断进行检查,发现该中断是分配给客户机的设备产生的,因此,VMM 调用虚拟中断控制器的接口函数,将中断发送给虚拟 Local APIC。之后,虚拟 Local APIC 就会在适当的时机将该中断注入给客户机,由客户操作系统的处理函数处理。

(4) 在将中断事件通知客户机以后,VMM 会进行后续处理,例如开中断等。

在上述过程中,有两点信息是需要在创建客户机的过程中提供的。

(5) 设备的分配信息。在第(3)步中,VMM 必须了解,中断 0x41 所对应的设备是否被分配给了某个客户机以及是哪个客户机。通常,这是在创建客户机的时候由用户决定,用户通过管理工具通知 VMM 相关的绑定信息。

(6) 设备在客户机平台上的管脚信息。在第(3)步中,VMM 在调用虚拟中断控制器的接口函数时,需要提供 IRQ 号,即管脚号。需要注意的是,虚拟中断控制器的输入管脚与物理中断控制器的输入管脚并不一定相同。物理的输入管脚是由物理平台决定,而虚拟中断控制器的输入管脚是由 VMM 所提供的虚拟平台决定,通常在创建客户机的时候就已经确定了。VMM 负责在两者之间做转换。

5.4.6 中断注入

中断注入负责将虚拟中断控制器采集到的中断请求按照其优先级,逐一注入客户机虚拟处理器。这里有两个问题需要解决,首先是如何取得需要注入的最高优先级中断的相关信息,其次是如何才能将一个中断注入客户机 VCPU。

对于第一个问题,虚拟中断控制器会负责将中断按照优先级排序,VMM 只需要调用虚拟中断控制器提供的接口函数,就可以获得当前最高优先级中断的信息。

对于第二个问题,虚拟 Local APIC 提供了将中断注入客户机 VCPU 最基本的功能。VMM 可以调用虚拟 Local APIC 的接口来实现中断注入。在这里,VMM 的虚拟中断注入逻辑需要考虑下面的几个问题。

(1) 如果目标 VCPU 正在物理 CPU 上运行,如何注入中断?如前所述,只能在 VM-Entry 的时候将中断注入客户机,因此,为了保证中断的及时注入,需要强迫 VCPU 发生 VM-Exit,这时就可以在 VM-Entry 返回客户机的时候注入中断。常用的使客户机发生 VM-Exit 的方法是向 VCPU 所在的物理 CPU 发送 IPI 中断。

(2) 如果目标 VCPU 目前无法中断,例如 VCPU 目前正处于关中断的状态(客户机的 EFLAGS.IF 为 0),如何注入中断?Intel 的 VT-x 技术对这种情况提供了一个解决机制,即使用前面章节描述的中断窗口(Interrupt Windows)。该机制通过设置 VMCS 的一个特定字段,告诉物理 CPU,其当前运行的客户机 VCPU 有一个中断需要注入。一旦客户机 VCPU 开始可以接受中断,例如进入开中断状态,物理 CPU 会主动触发 VM-Exit,从客户机陷入到 VMM 中,虚拟中断注入模块就可以注入等待的中断了。

(3) 什么时候来触发中断注入?通常的方法是,当中断采集逻辑调用虚拟中断控制器

接口请求发出中断后,虚拟中断控制器会根据内部的状态,例如虚拟 IMR/ISR 寄存器的值,来决定是否需要注入中断给客户机。其判断过程和物理中断控制器判断是否提交中断给 CPU 一样。

图 5-11 给出了中断注入逻辑的基本过程。

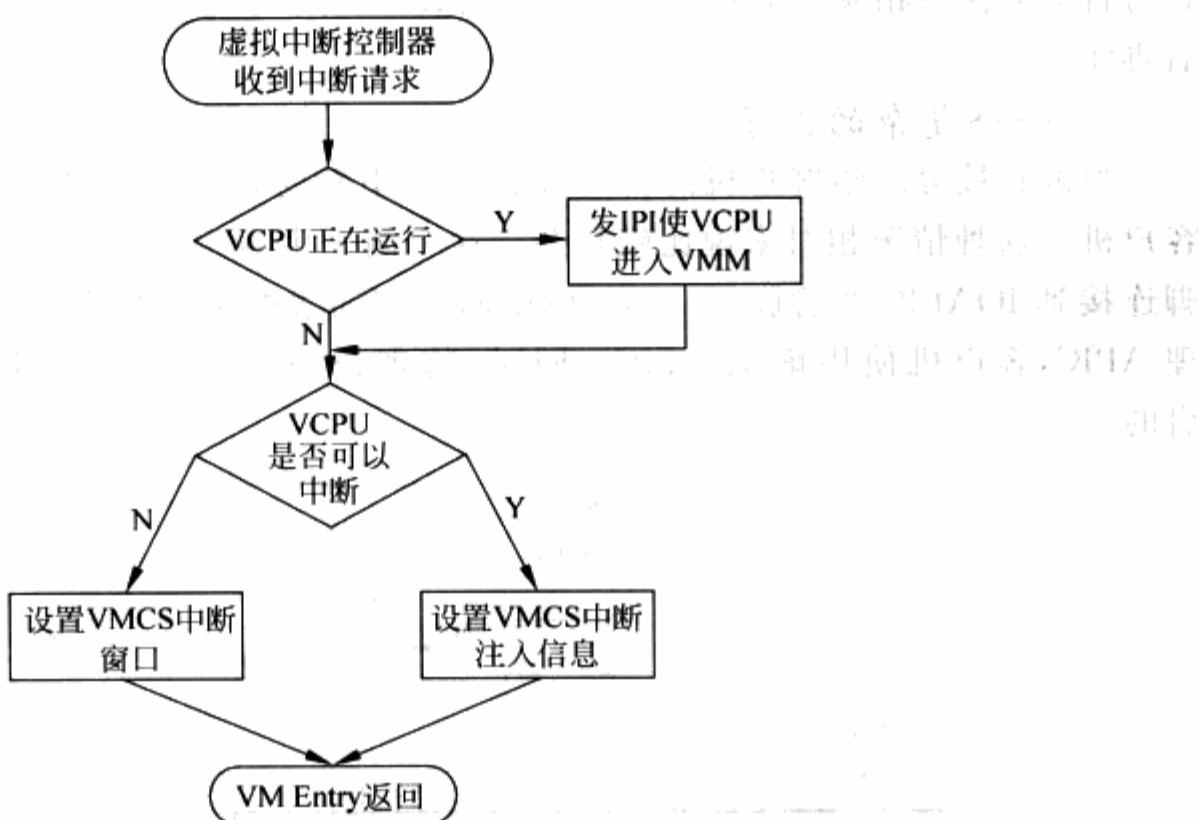


图 5-11 中断注入的过程

5.4.7 案例分析

最后分别举两个例子论述虚拟中断从产生到注入的全过程。

1. 一个简单的例子

在 2.7 节提到,物理平台上的可编程中断时钟 PIT 会定期产生时钟中断。VMM 模拟的虚拟 PIT 也一样,可以定时为客户机产生虚拟时钟中断。本节就以虚拟 PIT 为例,说明虚拟可编程时钟中断是如何被注入客户机的。

(1) 虚拟 PIT 产生中断请求时,调用虚拟中断控制器提供的接口通知虚拟 PIC/IOAPIC 自己有一个中断需要处理。

(2) 虚拟 PIC/IOAPIC 记录下这个中断请求,并检查内部寄存器,如 IMR、IRR 和 ISR 等,以决定是否需要将中断注入给客户机。如果不是,将这个中断请求保存在虚拟 PIC/IOAPIC 的内部逻辑中。当虚拟 PIC 内部状态改变以后(通常是客户机写 PIC/IOAPIC 的寄存器时),虚拟 PIC/IOAPIC 会检查内部是否有等待处理的中断请求并重复这个过程。

(3) 如果此时需要注入中断,调用中断注入逻辑。

(4) VMM 检查客户机 VCPU 是否正在运行,如果是,则发一个 IPI 强制其进入 VMM

上下文。

(5) 在客户机 VCPU 再运行前, VMM 会检查发现该 VCPU 有中断需要注入, 接着 VMM 会检查当前客户机 VCPU 是否能够被注入中断。如果能, 使用虚拟中断控制器提供的接口, 获取最高优先级中断的信息, 设置好 VMCS 中的相应字段, 使得当 VCPU 投入运行时自动去执行相应矢量号的中断处理函数。否则, 设置中断窗口, 等待下次 VM-Exit 之后再注入。

2. 一个复杂的例子

当被直接分配给客户机的物理设备发生中断时, VMM 需要将该中断注入给对应的客户机。这种情况相对来说比较复杂。这个例子中假定物理设备产生中断, 设备中断管脚连接到 IOAPIC 的管脚 0x12, 对应中断重定向表的矢量号为 0x41, 并假定系统使用物理 APIC, 客户机使用虚拟 APIC。同时, 物理设备中断发生时, 物理 CPU 的中断是开启的。

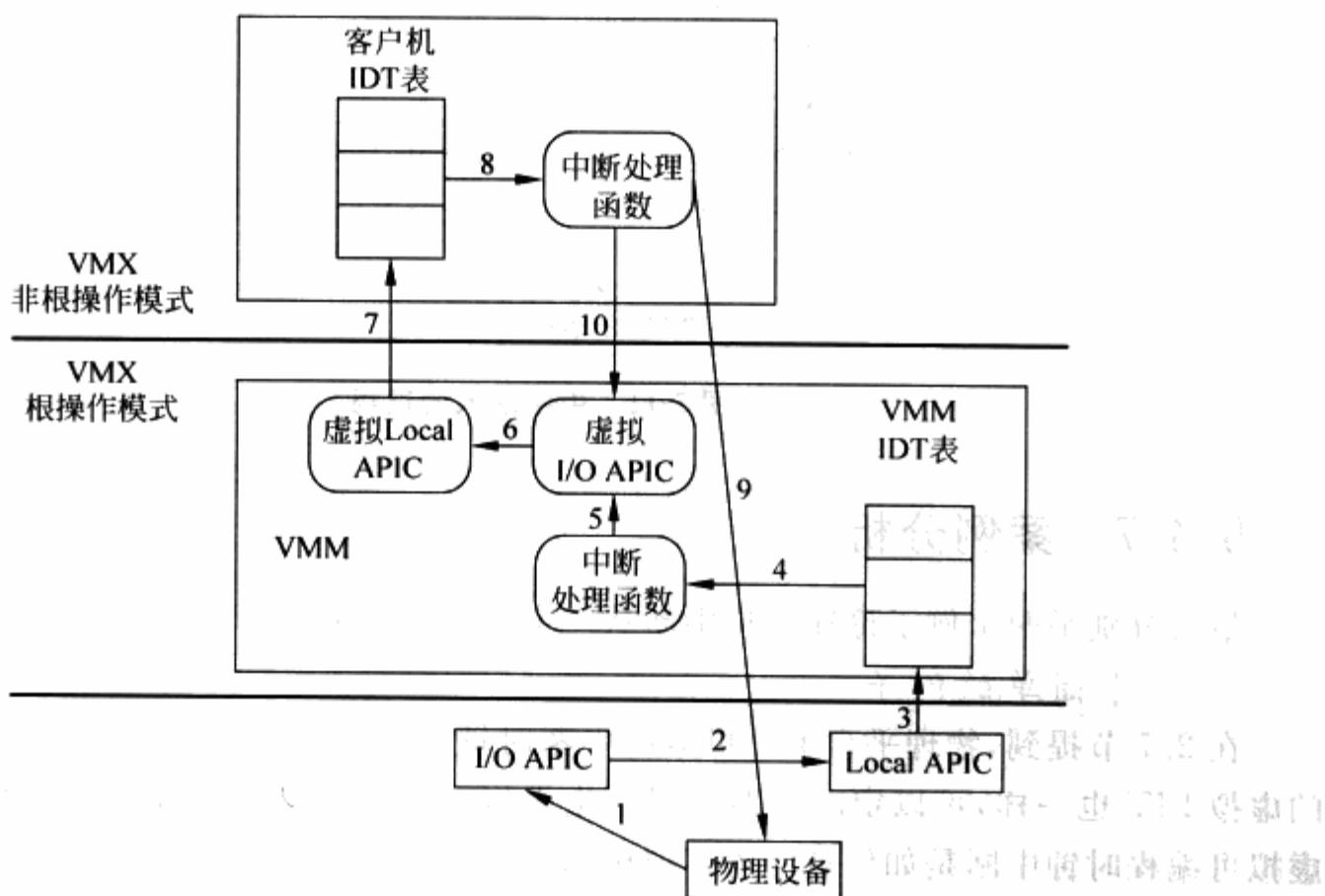


图 5-12 例子：直接分配设备的中断虚拟化

图 5-12 给出了该例子的流程。

- (1) 物理设备发生中断, 将中断发送给物理 I/O APIC 的管脚 0x12。
- (2) 物理 I/O APIC 收到后, 将管脚 0x12 转化为中断向量 0x41 发送给 Local APIC。
- (3) Local APIC 将中断 0x41 注入到 CPU, CPU 跳转到 IDT 表中 0x41 表项所指定的处理函数。同时, Local APIC 的 ISR 寄存器对应 0x41 的位被置上, 后面的等于或低于 0x41 的中断被屏蔽。

(4) VMM 的相关中断处理函数被执行。

(5) 中断处理函数检查发现这个中断是属于客户机的设备产生的中断,故调用虚拟中断控制器的接口函数。在将中断事件注入客户机以后,VMM 通过置一物理 IOAPIC 第 0x12 个 RTE 的屏蔽位,屏蔽后续的物理中断。VMM 向物理 Local APIC 写入 EOI,以清掉 Local APIC 的 ISR 寄存器 0x41 位,从而其他的中断也可以被接受。

VMM 中断处理函数对物理 APIC 的操作到此结束,下面是虚拟 APIC 工作的流程。

(1) 虚拟的 IOAPIC 中断控制器调用虚拟的 Local APIC 的接口函数,并将虚拟 IOAPIC 中相应的矢量号传入。虚拟 Local APIC 的 ISR 相关 bit 位被置上。

(2) 虚拟的 Local APIC 通过中断注入逻辑模块将中断注入到客户机。关于中断注入逻辑模块,参见前面的介绍。

(3) 客户机执行相关中断处理函数。

(4) 客户机中断处理函数处理物理设备的中断。

(5) 客户机向虚拟 Local APIC 写入 EOI,EOI 操作被 VMM 截获。虚拟 Local APIC 的 ISR 位被清掉,同时,虚拟 Local APIC 通知 VMM 客户已经完成对中断 0x21 的处理,VMM 清除物理 IOAPIC 第 0x12 个 RTE 的屏蔽位。

5.5 内存虚拟化

5.5.1 概述

5.2 节已经介绍了如何通过软件实现内存虚拟化,本节介绍硬件辅助的内存虚拟化。

内存虚拟化的主要任务是实现地址空间的虚拟化,内存虚拟化通过两次地址转换来支持地址空间的虚拟化,即客户机虚拟地址 GVA→客户机物理地址 GPA→宿主机物理地址 HPA 的转换。其中,GVA→GPA 的转换是由客户机软件决定的,通常是客户机操作系统通过 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 决定的,VMM 在将物理内存分配给客户机时就确定了 GPA→HPA 的转换,VMM 通常会用内部数据结构来记录这个映射关系。

传统的 IA32 架构只支持一次地址转换,即通过 CR3 指定的页表来实现“虚拟地址”→“物理地址”的转换。这和内存虚拟化所要求的两次地址转换产生了矛盾。可以通过将两次转换合并为一次转换来解决这个问题,即 VMM 根据 GVA→GPA→HPA 的映射关系,计算出 GVA→HPA 的映射关系,并将其写入“影子页表”。类似于“影子页表”这样的软件方法尽管能够解决问题,但是缺点也很明显。首先是实现非常复杂,例如需要考虑各种各样页表同步情况等,这样导致开发、调试和维护都比较困难。读者有兴趣可以参考一下 Xen/KVM 中影子页表的实现。此外,“影子页表”的内存开销也很大,因为需要为每个客户机进程对应的页表都维护一个“影子页表”。

为了解决这个问题,VT-x 提供了 Extended Page Table(EPT)技术,直接在硬件上支持 GVA→GPA→HPA 的两次地址转换,大大降低了内存虚拟化的难度,也进一步提高了内存虚拟化的性能。

此外,为了进一步提高 TLB 的使用效率,VT-x 还引入了 Virtual Processor ID(VPID)功能,进一步增加了内存虚拟化的性能。

5.5.2 EPT

1. EPT 原理

图 5-13 描述了 EPT 的基本原理。在原有的 CR3 页表地址映射的基础上,EPT 引入了 EPT 页表来实现另一次映射。这样,GVA→GPA→HPA 两次地址转换都由 CPU 硬件自动完成。

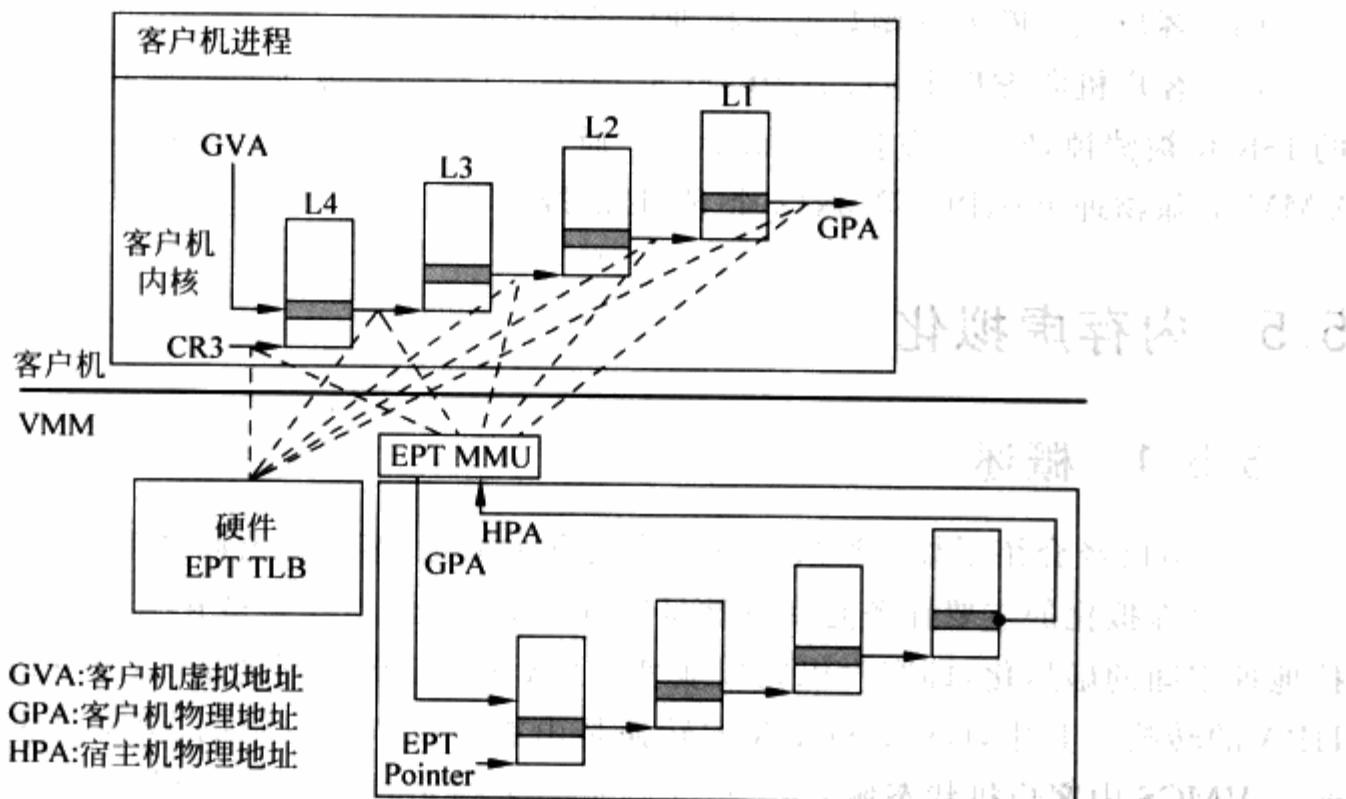


图 5-13 EPT 原理图

这里假设客户机页表和 EPT 页表都是 4 级页表,CPU 完成一次地址转换的基本过程如下。

CPU 首先会查找 Guest CR3 指向的 L4 页表。由于 Guest CR3 给出的是 GPA,因此 CPU 需要通过 EPT 页表来实现 Guest CR3 GPA→HPA 的转换。CPU 首先会查看硬件的 EPT TLB,如果没有对应的转换,CPU 会进一步查找 EPT 页表,如果还没有,CPU 则抛出 EPT Violation 异常由 VMM 来处理。

获得 L4 页表地址后,CPU 根据 GVA 和 L4 页表项的内容,来获取 L3 页表项的 GPA。如果 L4 页表中 GVA 对应的表项显示为“缺页”,那么 CPU 产生 Page Fault,直接交由

Guest Kernel 处理。注意,这里不会产生 VM-Exit。获得 L3 页表项的 GPA 后,CPU 同样要通过查询 EPT 页表来实现 L3 GPA→HPA 的转换,过程和上面一样。

同样的,CPU 会依次查找 L2、L1 页表,最后获得 GVA 对应的 GPA,然后通过查询 EPT 页表获得 HPA。从上面的过程可以看出,CPU 需要 5 次查询 EPT 页表,每次查询都需要 4 次内存访问,因此最坏情况下总共需要 20 次内存访问。EPT 硬件通过增大 EPT TLB 来尽量减少内存访问。

2. EPT 的硬件支持

为了支持 EPT,VT-x 规范在 VMCS 的“VM-Execution 控制域”中提供了 Enable EPT 字段。如果在 VM-Entry 的时候该位被置上,EPT 功能就会被启用,CPU 会使用 EPT 功能进行两次转换。

EPT 页表的基地址是由 VMCS“VM-Execution 控制域”的 Extended page table pointer 字段来指定的,它包含了 EPT 页表的宿主机物理地址。

EPT 是一个多级页表,每级页表的表项格式是相同的,如表 5-8 所示。

表 5-8 EPT 页表的表项格式

字段名称	描述
ADDR	下一级页表的物理地址。如果已经是最后一级页表,那么就是 GPA 对应页的物理地址
SP	超级页(super page):所指向的页是大小超过 4KB 的超级页。CPU 在遇到 SP=1 时,就会停止继续往下查询。对于最后一级页表,这一位可以供软件使用
X	可执行。X=1 表示该页是可执行的
R	可读。R=1 表示该页是可读的
W	可写。W=1 表示该页是可写的

EPT 页表转换过程和 CR3 页表转换是类似的。图 5-14 展现了 CPU 使用 EPT 页表进行地址转换的过程。

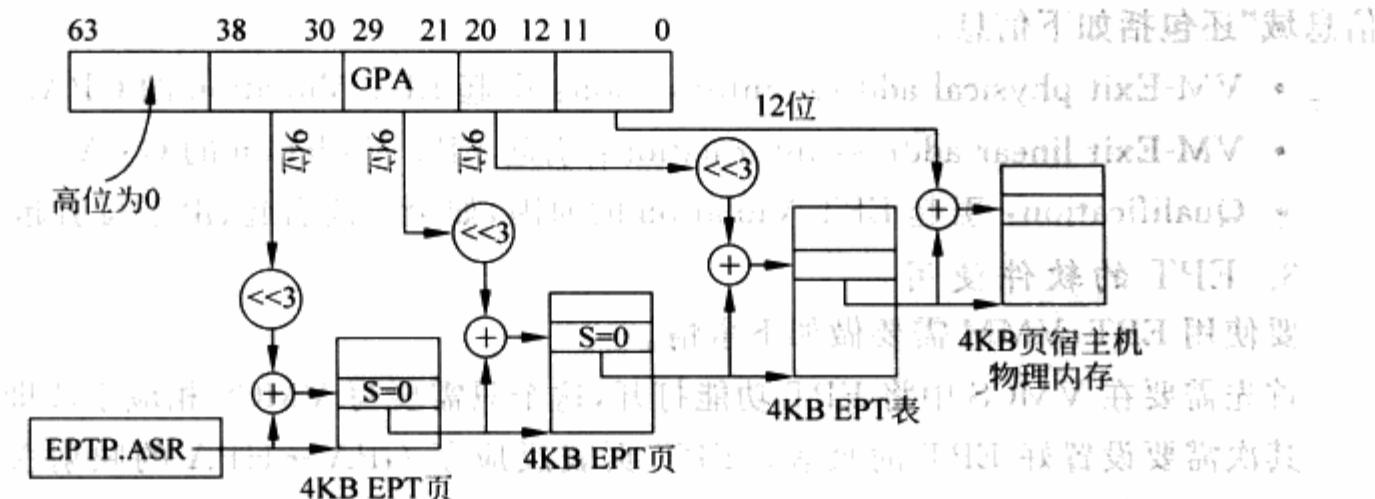


图 5-14 EPT 页表转换

EPT 通过 EPT 页表中的 SP 字段支持大小为 2MB 或者 1GB 的超级页。图 5-15 给出了 2MB 超级页的地址转换过程。和图 5-14 的不同点在于,当 CPU 发现 SP 字段为 1 时,就停止继续向下遍历页表,而是直接转换了。

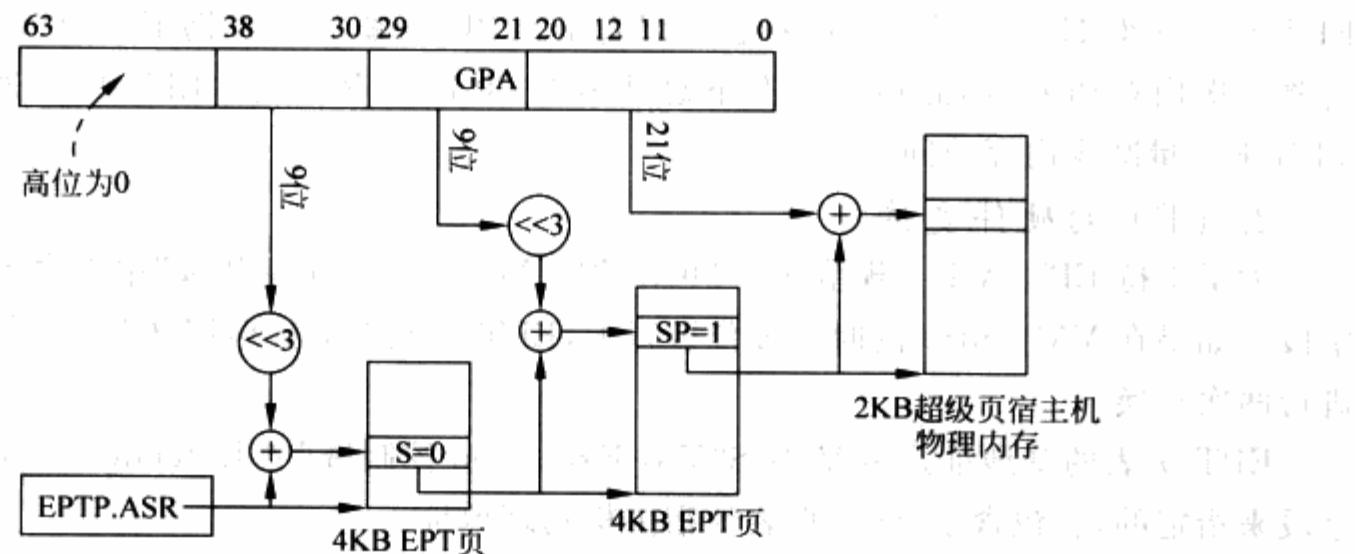


图 5-15 EPT 页表转换: 超级页

EPT 同样会使用 TLB 缓冲来加速页表的查找过程。因此,VT-x 还提供了一条新的指令 INVEPT,可以使 EPT 的 TLB 项失效。这样,当 EPT 页表有更新时,CPU 可以执行 INVEPT 使旧的 TLB 失效,使 CPU 使用新的 EPT 表项。

和 CR3 页表会导致 Page Fault 一样,使用 EPT 之后,如果 CPU 在遍历 EPT 页表进行 GPA→HPA 转换时,也会发生异常。

- (1) GPA 的地址位数大于 GAW。
- (2) 客户机试图读一个不可读的页($R=0$)。
- (3) 客户机试图写一个不可写的页($W=0$)。
- (4) 客户机试图执行一个不可执行的页($X=0$)。

发生异常时,CPU 会产生 VM-Exit,退出原因为 EPT Violation。VMCS 的“VM-Exit 信息域”还包括如下信息。

- VM-Exit physical-address information: 引起 EPT Violation 的 GPA。
- VM-Exit linear-address information: 引起 EPT Violation 的 GVA。
- Qualification: 引起 EPT Violation 的原因,如由于读引起、由于写引起等。

3. EPT 的软件使用

要使用 EPT,VMM 需要做如下事情。

首先需要在 VMCS 中将 EPT 功能打开,这个只需要写 VMCS 相应字段即可。

其次需要设置好 EPT 的页表。EPT 页表反应了 GPA→HPA 的映射关系。由于是 VMM 负责给虚拟机分配物理内存,因此,VMM 拥有足够的信息来建立 EPT 页表。此外,如果 VMM 给虚拟机分配的物理内存足够连续的话,VMM 可以在 EPT 页表中尽量使用超

级页,这样有利于提高 TLB 的性能。

当 CPU 开始使用 EPT 时,VMM 还需要处理 EPT Violation。通常来说,EPT Violation 的来源有如下几种。

- (1) 客户机访问 MMIO 地址。这种情况下,VMM 需要将请求转给 I/O 虚拟化模块。
- (2) EPT 页表的动态创建。有些 VMM 采用懒惰方法,一开始 EPT 页表为空,当第一次使用发生 EPT Violation 时再建立映射。

由此可以看出,EPT 相对于传统的“影子页表”方法,其实现大大地简化了。而且,由于客户机内部的 Page Fault 不用发生 VM-Exit,也大大减少了 VM-Exit 的个数,提高了性能。此外,EPT 只需要维护一张 EPT 页表,不像“影子页表”那样需要为每个客户机进程的页表维护一张影子页表,也减少了内存的开销。

5.5.3 VPID

TLB 是页表项的缓存,对地址转换的效率至关重要。TLB 需要和对应的页表一起工作才有效。因此,当页表发生切换时,TLB 原有的内容也就失效了,CPU 需要使用 INVLPG 指令使其所有项失效,这样才不会影响之后页表的工作。例如,我们知道进程切换时,需要切换进程地址空间(通过切换页表的起始物理地址 CR3),使前一个进程的 TLB 项全部失效。

类似地,在每次 VM-Entry 和 VM-Exit 时,CPU 会强制 TLB 内容全部失效,以避免 VMM 以及不同虚拟机虚拟处理器之间 TLB 项的混用,因为硬件无法区分一个 TLB 项是属于 VMM 还是某一特定的虚拟机虚拟处理器。

VPID 是一种硬件级的对 TLB 资源管理的优化。通过在硬件上为每个 TLB 项增加一个标志,来标识不同的虚拟处理器地址空间,从而区分开 VMM 以及不同虚拟机的不同虚拟处理器的 TLB。换而言之,硬件具备了区分不同的 TLB 项属于不同虚拟处理器地址空间(对应于不同的虚拟处理器)的能力。这样,硬件可以避免在每次 VM-Entry 和 VM-Exit 时,使全部 TLB 失效,提高了 VM 切换的效率。并且,由于这些继续存在的 TLB 项,硬件也避免了 VM 切换之后的一些不必要的页表遍历,减少了内存访问,提高了 VMM 以及虚拟机的运行速度。

VT-x 通过在 VMCS 中增加两个域来支持 VPID。第一个是 VMCS 中的 Enable VPID 域,当该域被置上时,VT-x 硬件会启用 VPID 功能。第二个是 VMCS 中的 VPID 域,用于标识该 VMCS 对应的 TLB。VMM 本身也需要一个 VPID,VT-x 规定虚拟处理器标志 0 被指定用于 VMM 自身,其他虚拟机虚拟处理器不得使用。

因此,在软件上使用 VPID 非常简单,主要做两件事情。首先是为 VMCS 分配一个 VPID,这个 VPID 只要是非 0 的,且和其他 VMCS 的 VPID 不同就可以了;其次是在 VMCS 中将 Enable VPID 置上,剩下的事情硬件会自动处理。

5.6 I/O 虚拟化的硬件支持

5.6.1 概述

在软件虚拟化章节已经阐述过如何用软件的方式实现 I/O 虚拟化, 目前流行的“设备模拟”和“类虚拟化”都有各自的优点, 以及与生俱来的缺点。前者通用性强, 但性能不理想; 后者性能不错, 却又缺乏通用性。为此, 英特尔公司发布了 VT-d 技术 (Intel (R) Virtualization Technology for Directed I/O), 以帮助虚拟软件开发者实现通用性强、性能高的新型 I/O 虚拟化技术。

在介绍 VT-d 技术前, 先量化一下评价 I/O 虚拟技术的两个指标——性能和通用性。性能不必说, 越接近无虚拟机环境下的 I/O 性能越好; 通用性主要是和全虚拟化挂钩的, 使用的 I/O 虚拟化技术对客户操作系统越透明, 则通用性越强。通过 VT-d 技术, 可以很好地实现这两个指标, 无须像“设备模拟”和“类虚拟化”两种技术一样, 为了提高某个指标而使另一个指标打折。

如何实现这两个指标呢? 对于高性能, 最直接的方法就是让客户机直接使用真实的硬件设备, 这样客户机的 I/O 操作路径几乎和无虚拟机环境下的 I/O 路径相同, 获得高性能是理所当然的; 对于通用性, 就要用全虚拟化的方法, 让客户机操作系统能够使用自带的驱动程序发现设备、操作设备。下面先来看看实现这些目标所面临的挑战。

客户机直接操作设备面临如下两个问题。

- (1) 如何让客户机直接访问到设备真实的 I/O 地址空间(包括端口 I/O 和 MMIO)。
- (2) 如何让设备的 DMA 操作直接访问到客户机的内存空间? 要知道, 设备可不管系统中运行的是虚拟机还是真实操作系统, 它只管用驱动提供给它的物理地址做 DMA。

通用性面临的问题和(1)是类似的, 要有一种方法把设备的 I/O 地址空间告诉给客户操作系统, 并能让驱动通过这些地址访问到设备真实的 I/O 地址空间。VT-x 技术已经能够解决第一个问题, 可以允许客户机直接访问物理的 I/O 空间。Intel 的 VT-d 技术则让第二个问题的解决成为可能, 它提供了 DMA 重映射技术, 以帮助 VMM 的实现者达到目标。

VT-d 技术通过在北桥(MCH)引入 DMA 重映射硬件, 以提供设备重映射和设备直接分配的功能。在启用 VT-d 的平台上, 设备所有的 DMA 传输都会被 DMA 重映射硬件截获。根据设备对应的 I/O 页表, 硬件可以对 DMA 中的地址进行转换, 使设备只能访问到规定的内存。使用 VT-d 后, 设备访问内存的架构如图 5-16 所示。

图 5-16(a)中是没有 VT-d 的平台, 此时设备的 DMA 可以访问整个物理内存。图 5-16(b)是启用 VT-d 的情况, 此时, 设备只能访问指定的物理内存。相信读者有似曾相识的感觉, 是的, 这和使用页表将进程的线性地址空间映射到指定物理内存区域的思想一样, 只不过对

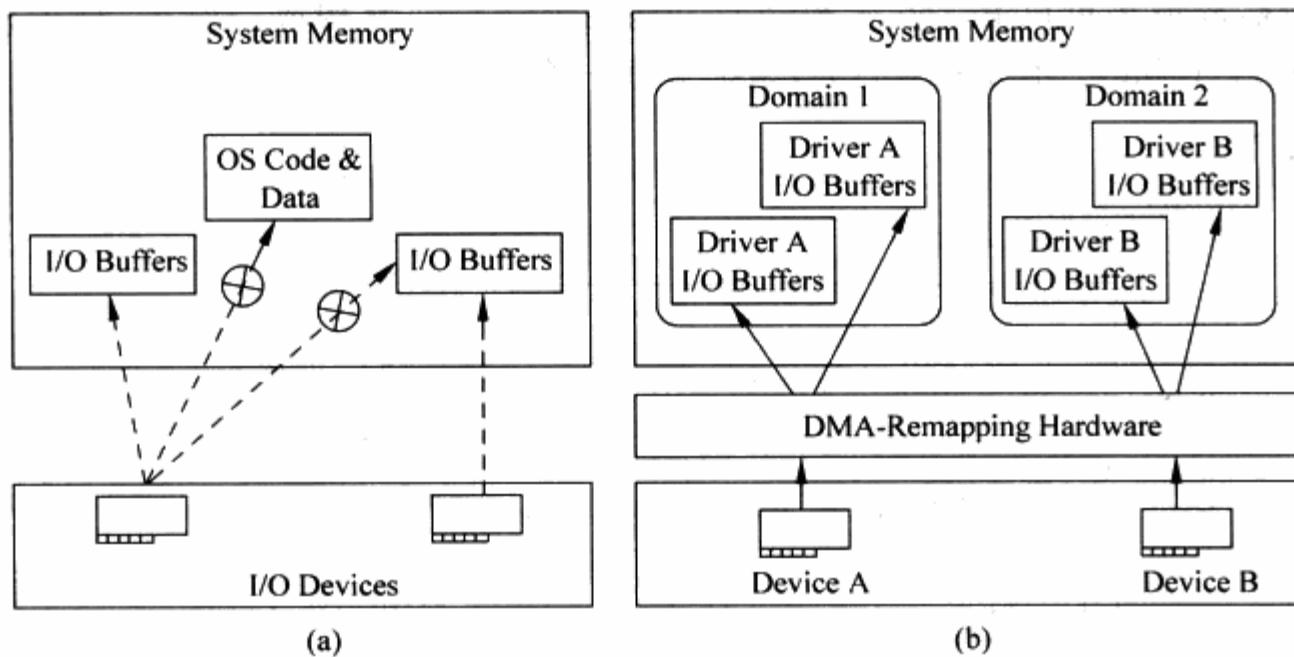


图 5-16 使用 VT-d 后访问内存架构

象换成了设备。下面介绍 VT-d 中核心的 DMA 重映射技术以及如何探测 DMA 重映射硬件,设备分配的内容在 6.7 节中介绍。VT-d 技术较为复杂,限于篇幅,只能对主要技术进行介绍,完整的论述请参考“Intel(R) Virtualization Technology for Directed I/O Architecture Specification”。

5.6.2 VT-d 技术

在上一节提到的诸多难点中,最主要的问题是 DMA 问题。设备对系统中运行的软件是一无所知的,在进行 DMA 时,设备唯一做的是向(从)驱动程序告知的“物理地址”复制(读取)数据。在内存虚拟化相关章节已经知道,虚拟机环境下客户机使用的是 GPA,则客户机的驱动直接操作设备时也是用 GPA。而设备进行 DMA,需要用 MPA,如何在 DMA 时将 GPA 转换成 MPA 就成了关键问题。要知道,通常无法通过软件的方法截获设备的 DMA 操作,VT-d 的技术提供的 DMA 重映射就是为解决这个问题而提出的。

1. DMA 重映射(DMA Remapping)

先来回忆一下第 2 章中提到的 PCI 总线结构,通过 BDF 可以索引到任何一条总线上的任何一个设备。同样,DMA 的总线传输中包含一个 BDF 以标识该 DMA 传输是由哪个设备发起的。在 VT-d 技术中,标识 DMA 操作发起者的结构称为源标识符(Source Identifier)。对于 PCI 总线,VT-d 使用 BDF 作为源标识符,在下面的内容中提到 BDF 均代表源标识符。其格式在第 2 章已经给出,读者也可以参考“Intel(R) Virtualization Technology for Directed I/O Architecture Specification”的 3.2.1 节获取详细内容。

除了 BDF 外,VT-d 还提供了两种数据结构来描述 PCI 架构,分别是根条目(Root Entry)和上下文条目(Context Entry)。

(1) 根条目: 用于描述 PCI 总线,每条总线对应一个根条目。由于 PCI 架构支持最多

256 条总线,故最多可以有 256 个根条目。这些根条目一起构成一张表,称为根条目表(Root Entry Table)。有了根条目表,系统中每一条总线都会被描述到。图 5-17 是根条目的结构。

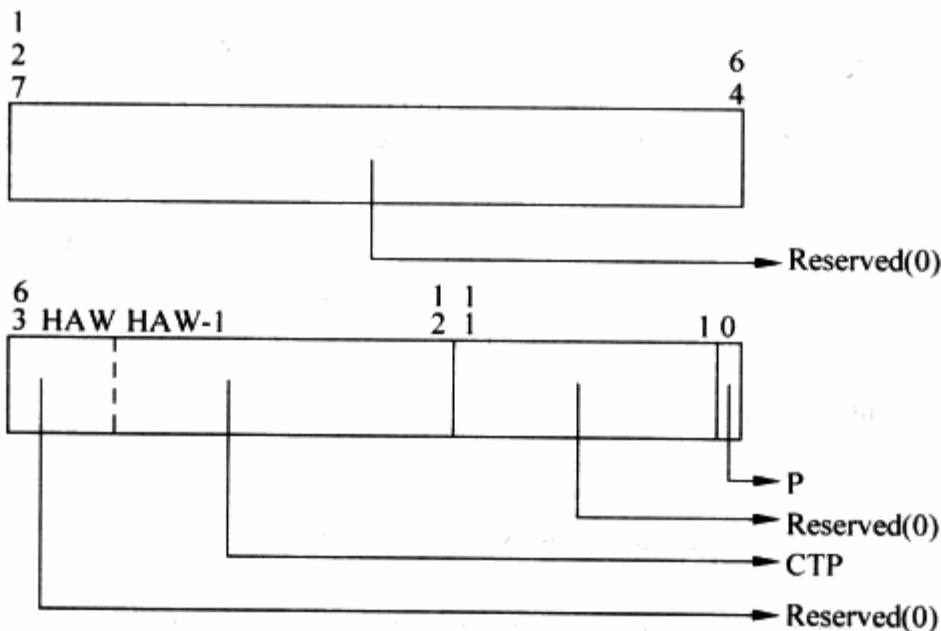


图 5-17 根条目结构

主要字段如下。

- P: 存在位。为 0 时条目无效,来自该条目所代表总线的所有 DMA 传输被屏蔽。为 1 时,该条目有效。
- CTP(Context Table Pointer,上下文表指针): 指向上下文条目表。

(2) 上下文条目: 用于描述某个具体的 PCI 设备,这里的 PCI 设备是指逻辑设备(见第 2 章关于 BDF 中 function 字段的阐述)。一条 PCI 总线上最多有 256 个设备,故有 256 个上下文条目,它们一起组成上下文条目表(Context Entry Table)。通过上下文条目表,可描述某条 PCI 总线上的所有设备。图 5-18 是上下文条目的结构。

主要字段如下。

- P: 存在位。为 0 时条目无效,来自该条目所代表设备的所有 DMA 传输被屏蔽。为 1 时,表示该条目有效。
- T: 类型,表示 ASR 字段所指数据结构的类型。目前,VT-d 技术中该字段为 0,表示多级页表。
- ASR(Address Space Root,地址空间根): 实际是一个指针,指向 T 字段所代表的数据结构,目前该字段指向一个 I/O 页表(见后面的内容)。
- DID(Domain ID,域标识符): VT-d 技术中 Domain 的具体含义请参考“Intel(R) Virtualization Technology for Directed I/O Architecture Specification”的 3.1 节的定义。在此,读者可以理解为本文中的客户机,DID 可以看作用于唯一标识该客户机的标识符,例如 Guest ID。

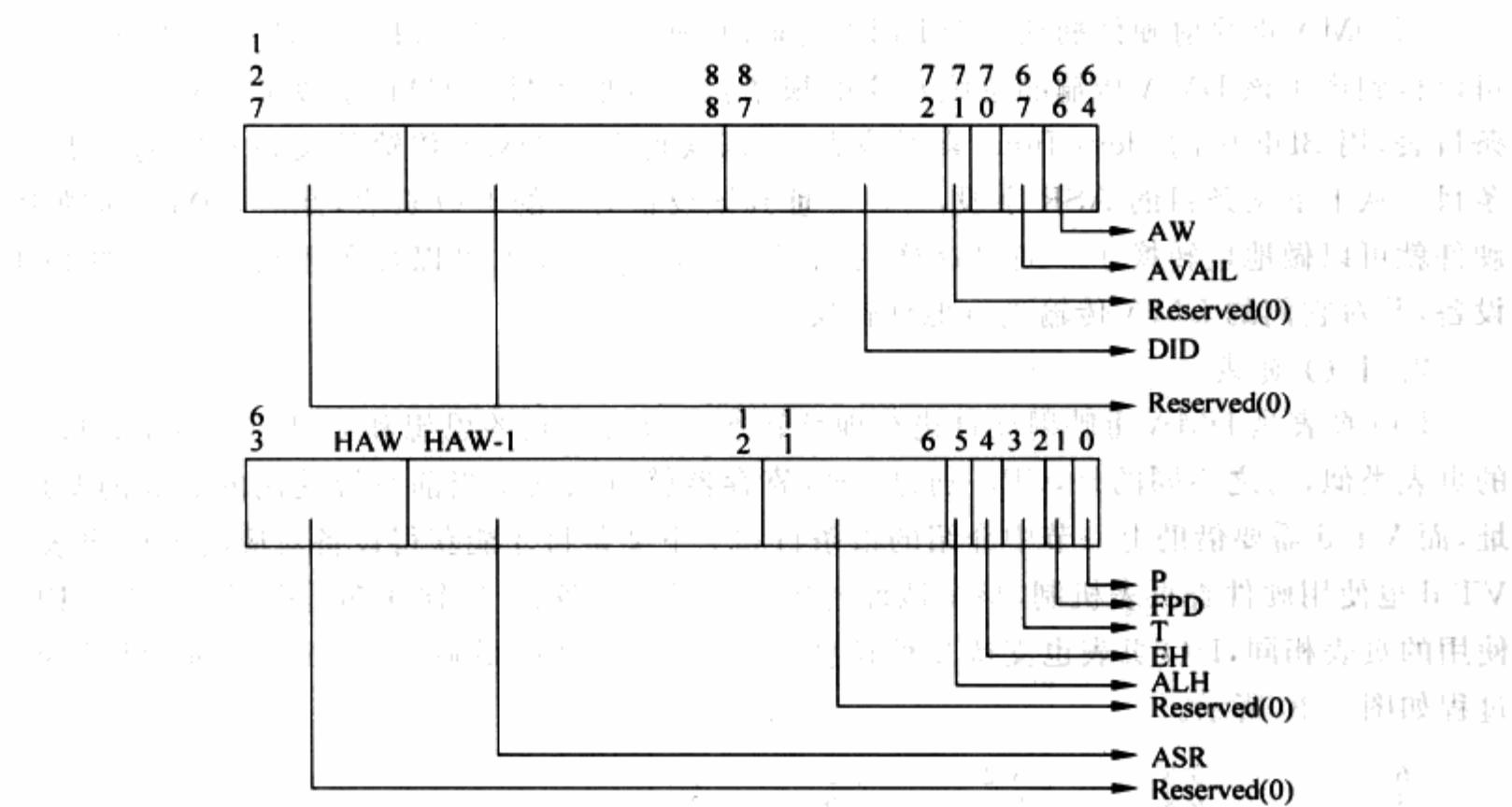


图 5-18 上下文条目结构

其余字段的具体解释请参见“Intel(R) Virtualization Technology for Directed I/O Architecture Specification”的 3.2.3 节。

根条目表和上下文条目表一起构成了图 5-19 所示的两级结构。

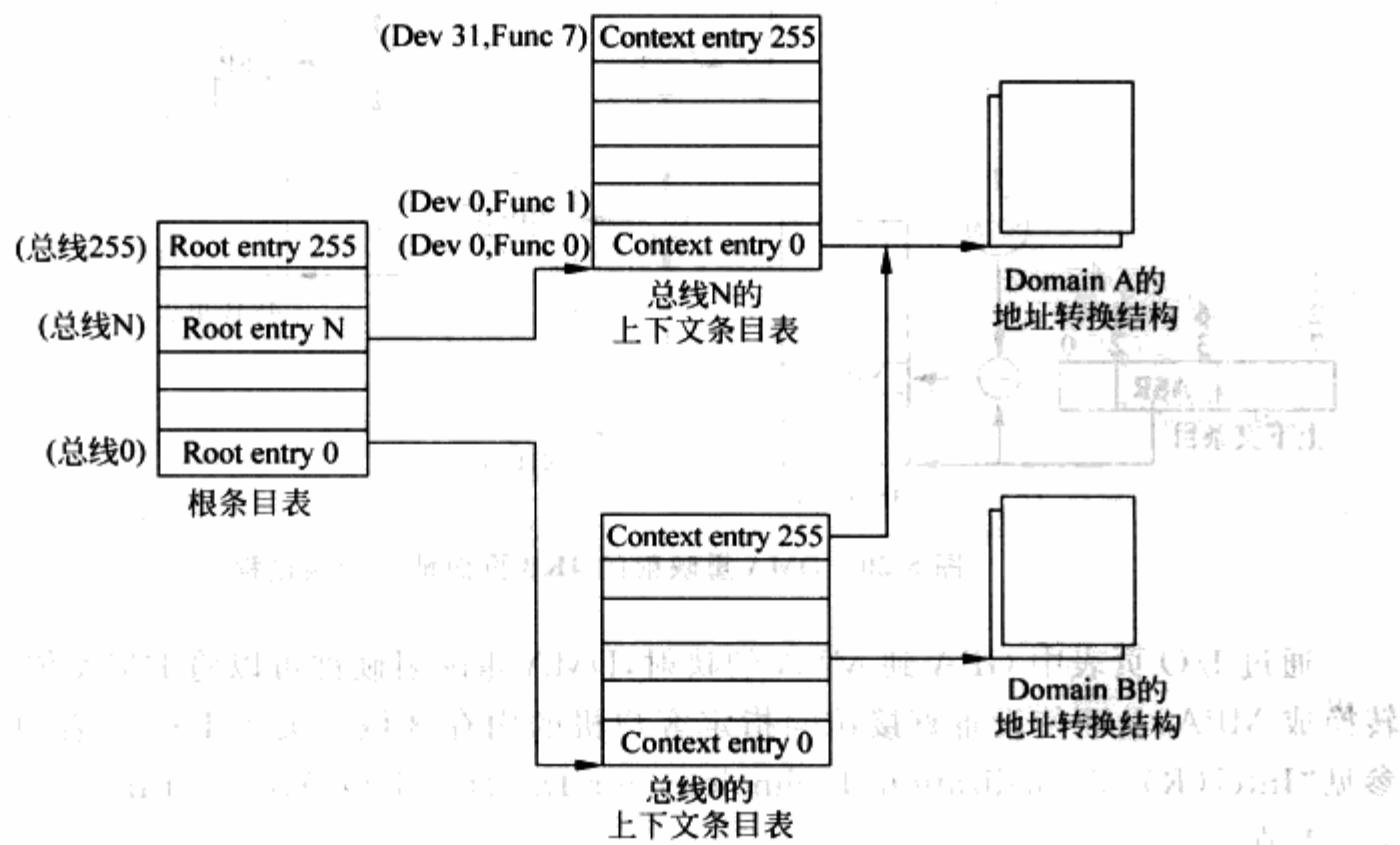


图 5-19 根条目表和上下文条目表构成的两级结构

当 DMA 重映射硬件捕获一个 DMA 传输时,通过其中 BDF 的 bus 字段索引根条目表,可以得到产生该 DMA 传输的总线对应的根条目。由根条目的 CTP 字段可以获得上下文条目表,用 BDF 中的{dev: func}索引该表,可以得到发起 DMA 传输的设备对应的上下文条目。从上下文条目的 ASR 字段,可以寻址到该设备对应的 I/O 页表,此时,DMA 重映射硬件就可以做地址转换了。通过这样的两级结构,VT-d 技术可以覆盖平台上所有的 PCI 设备,并对它们的 DMA 传输进行地址转换。

2. I/O 页表

I/O 页表是 DMA 重映射硬件进行地址转换的核心。它的思想和 CPU 中 paging 机制的页表类似,与之不同的是,CPU 通过 CR3 寄存器就可以获得当前系统使用的页表的地址,而 VT-d 需要借助上一节中介绍的根条目和上下文条目才能获得设备对应的 I/O 页表。VT-d 也使用硬件查页表机制,整个地址转换过程对于设备、上层软件都是透明的。与 CPU 使用的页表相同,I/O 页表也支持几种粒度的页面大小,其中最典型的 4KB 页面地址转换过程如图 5-20 所示。

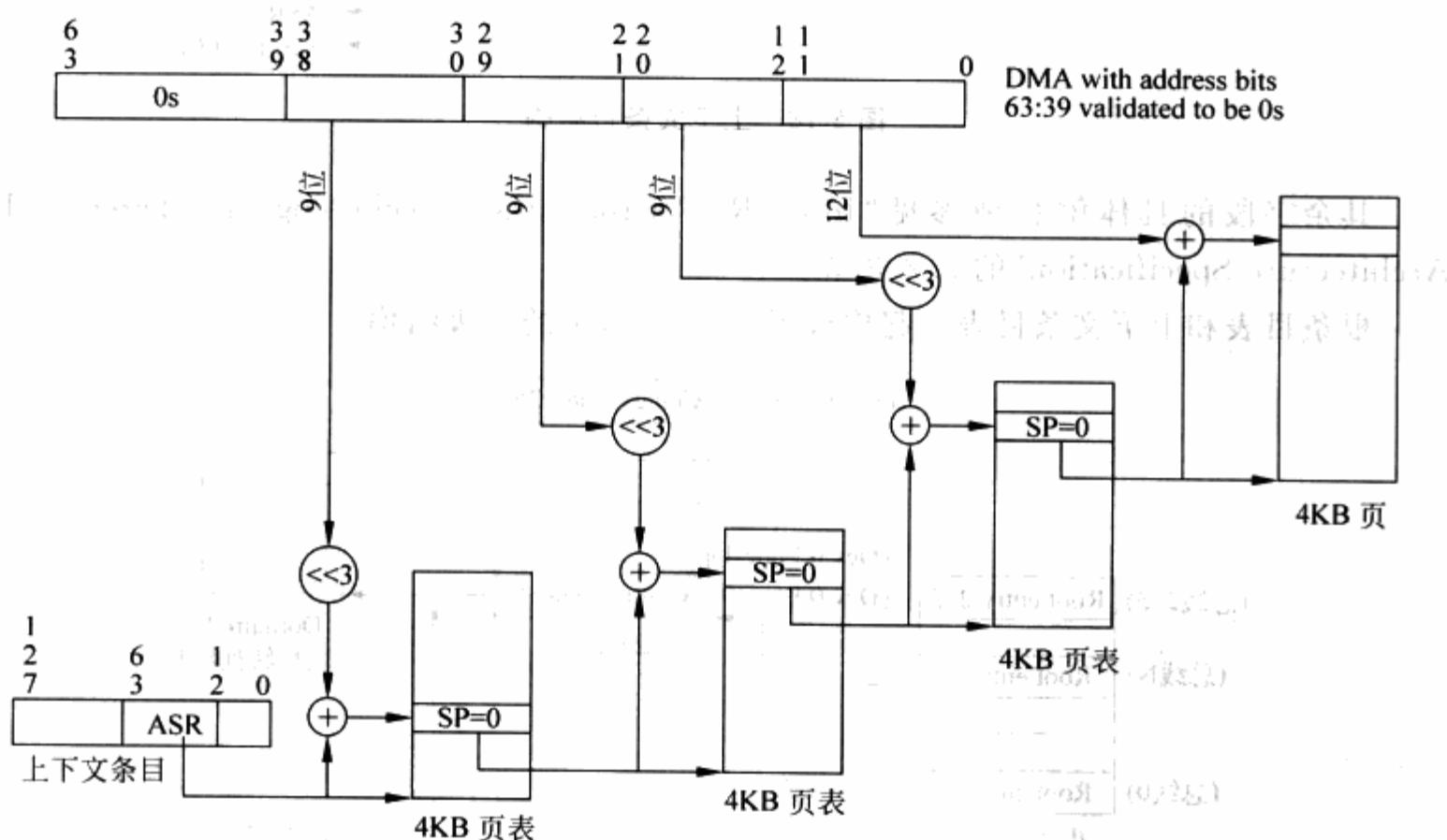


图 5-20 DMA 重映射的 4KB 页面地址转换过程

通过 I/O 页表中 GPA 到 MPA 的映射,DMA 重映射硬件可以将 DMA 传输中的 GPA 转换成 MPA,从而使设备直接访问指定客户机的内存区域。关于 I/O 页表的详细内容请参见“Intel(R) Virtualization Technology for Directed I/O Architecture Specification”的 3.3.1 节。

3. VT-d 硬件缓存

VT-d 硬件使用了大量的缓存以提高效率。其中,和地址转换相关的缓存称为 IOTLB,它和 CPU 中的 TLB 功能一样。此外,对于上下文条目,VT-d 硬件提供了上下文条目表。当软件修改了 I/O 页表、上下文条目表之后,要负责对这些缓存进行刷新。

VT-d 对两种缓存分别提供三种粒度的刷新操作。

(1) 全局刷新(Global Invalidiation): 整个 IOTLB 或上下文条目表中所有条目无效。

(2) 客户机粒度刷新(Domain-Selective Invalidation): IOTLB 中或上下文条目表中和指定客户机相关的地址条目或上下文条目无效。

(3) 局部刷新: 对于 IOTLB,称为 Domain vPage-Selective Invalidation,指定客户机某一地址范围内的页面映射条目无效。对于上下文条目表,称为 Device Selective Invalidiation,和某个指定设备相关的上下文条目无效。

硬件可以实现上述三种刷新操作的一种或多种。对于系统软件来说,它并不知道自己发起的刷新操作被硬件使用哪一种粒度的刷新操作完成。具体内容请参考“Intel(R) Virtualization Technology for Directed I/O Architecture Specification”的 3.2.3.1 和 3.3.1.3 节。

4. VT-d 硬件的探测

和所有的硬件一样,在使用 DMA 重映射硬件之前需要对它进行探测。VT-d 通过 BIOS 的 ACPI 表向上层软件汇报平台 DMA 重映射硬件的情况,硬件由三个主要数据结构描述。

- DMAR(DMA Remapping Reporting): 该结构汇报平台 VT-d 相关硬件的整体情况,可以看作一个总表。其主要字段如表 5-9 所示。

表 5-9 DMAR 结构

字段名	描述
Length	以字节数表示 DMAR 表占用的内存大小
HAW	该平台支持的 DMA 操作可寻址的最大物理地址空间
DMA Remapping Structures	指向下一级硬件描述数据结构,包括 DHRD 和 RMRR 两种

本文中,只介绍 DMA Remapping Structures 字段为 DHRD 的情况。

- DHRD(DMA Remapping Hardware Unit Definition): 用于描述 DMA 重映射硬件,一个 DHRD 结构对应一个 DMA 重映射硬件。典型的实现是平台只有一个 DMA 重映射硬件并管辖所有设备,但 VT-d 技术也支持一个平台多个 DMA 重映射硬件。DHRD 的主要字段如表 5-10 所示。关于 INCLUDE_ALL 和非 INCLUDE_ALL 模式,可以换一种方式理解。对于前者,表示该 DHRD 管辖所有设备(平台只有一个 DMA 重映射硬件的情况);对于后者,该 DHRD 只管辖 Device Scope 字段描述的设备。

表 5-10 DHRD 结构

字段名	描述
Flag	指明该 DMA 重映射硬件截获哪些设备的 DMA 传输,它有两种模式。 (1) INCLUDE_ALL 模式: 在此模式下,截获所有未被其他 DMA 重映射硬件截获的 DMA 传输;对于平台只有一个 DMA 重映射硬件的情况下,该模式截获所有设备的 DMA 传输。 (2) 非 INCLUDE_ALL 模式: 在此模式下,只截获来自 Device Scope 字段所描述设备的 DMA 传输
Register Base Address	指向 DMA 重映射硬件寄存器的基地址,系统软件通过读写这些寄存器操作 DMA 重映射硬件
Device Scope	设备域,数组结构,成员为 DSS。当 DMA 重映射硬件工作在非 INCLUDE_ALL 模式时,该数组所包含设备的 DMA 传输被当前 DMA 重映射硬件截获

- DSS(Device Scope Structure): 描述 DHRD 所管辖的设备。DHRD 的 Device Scope 指向的数组中的每个元素以 DSS 结构表示。该结构可以代表两种类型的设备,一种是 PCI 终端设备,一种是 PCI 桥设备。该结构有三个重要字段,如表 5-11 所示。

表 5-11 DSS 结构

字段名	描述
Type	指明该 DSS 描述的设备类型,1 代表 PCI 终端设备,2 代表 PCI 桥设备
Starting Bus Number	当 DSS 描述的设备位于某特定 PCI 桥下时,该 PCI 桥下第一条总线的总线号
PCI Path	当 Type 字段为 PCI 桥时,指明该桥所属设备的路径

三种数据结构构成了图 5-21 所示的层次。

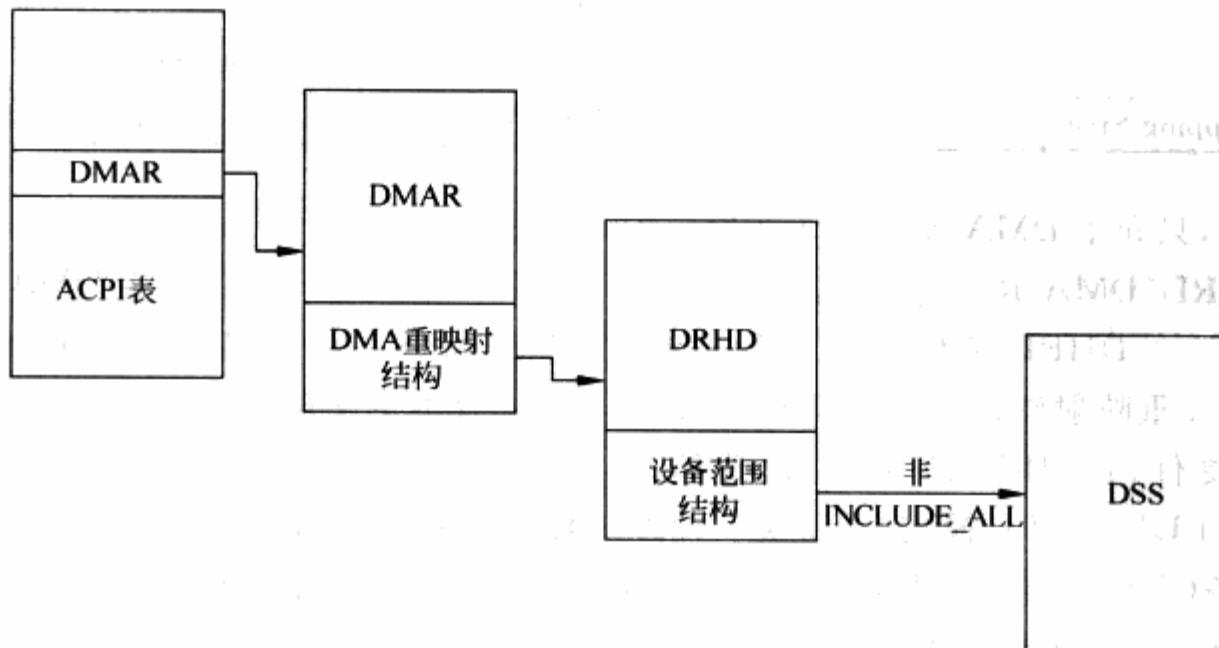


图 5-21 DMAR、DRHD 和 DSS 的层次

其中,第一级是 ACPI 表,从中获得 DMAR,然后依据前面描述的各个结构的各字段,可以解析出平台每个 DMA 重映射硬件的所有信息,例如该硬件的寄存器地址、该硬件管辖的设备等。

本节只介绍了探测 VT-d 硬件用到的主要数据结构以及它们的主要字段,详细信息请参考“Intel(R) Virtualization Technology for Directed I/O Architecture Specification”的第 5 章。

5.7 I/O 虚拟化的实现

5.7.1 概述

在上一节,介绍了英特尔公司的 VT-d 技术为 I/O 虚拟化带来了怎样的便利。本节中,会介绍如何将 VT-d 技术运用到 VMM 的 I/O 虚拟化实现中,并会对上节提及的设备直接分配技术进行介绍。

5.7.2 设备直接分配

在“设备模拟”和“类虚拟化”两种 I/O 虚拟化技术中,所有客户机都共享平台硬件设备。考虑这样一种情况,当 VMM 运行在一台拥有 10 块网卡的服务器上时,前两种技术完全可能只使用一块网卡来满足所有客户机的网络 I/O 需求,这必然导致了低性能和资源的浪费。设备直接分配技术很好地解决了这个问题。可以把某个设备直接分配给一个客户机,让客户机的 I/O 访问直接访问到设备的 I/O 地址空间,从宏观上看,这个客户机直接操作了平台的硬件设备。还记得第 2 章 I/O 架构中的内容吗? I/O 只有如下三个方面。

- (1) 驱动程序通过 I/O 地址空间操作设备,即设备直接分配技术解决的问题。
- (2) 设备通过 DMA 读取/复制数据。
- (3) 已经由 VT-d 的 DMA 重映射技术解决了,(3)中断。在 2.4 节已经介绍过。

设备直接分配的一个问题是如何阻止来自未分配到该设备的客户机的 I/O 访问。例如,系统中有三个客户机(0、1、2),其中客户机 1 分配到了网卡 A,则要阻止客户机 0、客户机 2 对网卡 A 的访问,一个最直接的方法是隐藏,让客户机 0、客户机 1 认为网卡 A 根本就不存在。实际上,无论是“设备模拟”还是“类虚拟化”技术,平台的硬件对于客户机都是透明的,所谓隐藏主要是针对运行“设备模拟器”的客户机/宿主机,或拥有“类虚拟化”后端驱动的客户机/宿主机而言的。隐藏的方式视具体的情况而定,例如可以在拥有硬件设备的客户机/宿主机加载驱动程序前,先给要分配出去的设备加载一个伪驱动作为占位符。由于没有真正的驱动程序,该设备就不会被访问到。

设备直接分配的另一个问题是如何让客户机的 I/O 操作直接访问到设备真实的 I/O 地址空间,该问题在下一节介绍。至此,读者应该理解可以通过设备直接分配将某一设备直

接分配给某个客户机，并让客户机直接操作该设备。

5.7.3 设备 I/O 地址空间的访问

在本节中，只讨论 PCI 设备在设备直接分配的情况下，客户机如何直接访问设备的真实 I/O 地址空间。

在第 2 章已经介绍了，PCI 设备的 I/O 地址空间通过 PCI BAR (Base Address Register) 报告给操作系统。为此，有两种选择供设备直接分配技术使用。

(1) 将设备的真实 PCI BAR 报告给客户机，并通过 VMCS 的 I/O bitmap 和 EPT 使客户机的端口 I/O 和 MMIO 都不引起 VM-Exit，则客户操作系统的驱动程序可以直接访问设备的 I/O 地址空间。

(2) 建立转换表，报告虚拟的 PCI BAR 给客户机，当客户机访问到虚拟的 I/O 地址空间时，VMM 负责截获操作，并通过转换表把 I/O 请求转发到设备的 I/O 地址空间。

两种方法中，方法(1)是高效的(不引起 VM-Exit)和简单的(直接报告真实的 PCI BAR 给客户机)，但在实际运用中会存在一些问题。通常，VMM 产品会使用多种 I/O 虚拟化技术，客户机的 I/O 请求，可能一部分由“设备模拟”技术满足，一部分由设备直接分配技术满足。例如一个客户机，它的显卡是由“设备模拟器”模拟的，但网卡又是操作的真实设备。在第 2 章 2.6 节提到过，设备的 PCI BAR 通常是 BIOS 配置并由操作系统直接使用的。那在上述情况中，由“设备模拟器”提供的设备的 PCI BAR 由虚拟 BIOS 配置，而真实设备的 PCI BAR 由平台的 BIOS 配置，两者之间就可能产生冲突。当这种情况发生时，在操作系统看来就是资源冲突，很可能停用其中一个设备而满足另一个设备。此外，操作系统是有权利修改设备的 PCI BAR 的，但应该阻止客户机直接修改真实设备的 PCI BAR，这是为了防止真实设备之间的 PCI BAR 冲突，以及在客户机销毁时把设备分配给其他客户机使用。由于这些原因，在实现设备直接分配技术时，通常采用的是方法(2)，即建立转换表。根据 I/O 地址空间的划分，转换表分为 Port I/O 转换表和 MMIO 转换表。

对于端口 I/O，在介绍 VMCS 时已经提到过，可以通过 I/O bitmap 来控制客户机访问某个端口是否引起 VM-Exit。这样，完全可以使用“设备模拟器”的虚拟 BIOS(或是其他手段，取决于 VMM 使用的 I/O 虚拟技术)为分配给客户机的真实设备生成虚拟的 PCI BAR，将它报告给客户操作系统，并修改 I/O bitmap 使客户机在访问这些 I/O 端口时产生 VM-Exit。同时，VMM 维护一张虚拟 PCI BAR 到真实 PCI BAR 的映射表。当客户机通过虚拟的 PCI BAR 发起 I/O 操作时，会因为 VM-Exit 陷入到 VMM 中，VMM 即可以通过转换表获得真实设备的 I/O 端口，帮客户机将请求转发给真实硬件。

对于 MMIO，其访问方式和内存访问无异。完全可以使用内存虚拟技术来解决这个问题。在虚拟 BIOS 产生虚拟 PCI BAR 之后，只需将虚拟的 MMIO 地址空间映射到设备真实的 MMIO 地址空间上，当客户机通过虚拟的 MMIO 地址空间访问设备时，内存虚拟机制会处理一切。举个例子，如果当前 VMM 使用 EPT，则客户机在第一次访问虚拟 MMIO 地

址空间时会陷入到 VMM。此时,可以修改 EPT 页表建立起虚拟 MMIO 地址空间到设备真实 MMIO 地址空间的映射,则在以后的访问中,客户机对该虚拟 MMIO 地址空间的访问就不会再陷入到 VMM。

除了解决客户机直接访问设备 I/O 地址空间的问题外,转换表还可以满足客户机修改设备 PCI BAR 的情况。此时,只需要修改虚拟的 PCI BAR 并维护修改后的值到真实 PCI BAR 的映射即可。至于如何截获修改 PCI BAR 的操作,请参照 2.6 节中提到的两个 I/O 端口 0xCF8~0xCFF。只需要截获客户机对这两个端口的操作即可。

5.7.4 设备发现

前面的内容解决了客户机直接访问真实设备的问题,但如何让客户机中的操作系统发现真实的设备呢?

在上一节也提到,VMM 通常会同时使用多种 I/O 虚拟化技术,其中一项必然会虚拟 PCI 总线(一般来说,这是“设备模拟器”的工作),所以只需将真实设备“挂接”到这条虚拟的 PCI 总线上,客户操作系统枚举 PCI 设备时必然会出现分配它。从第 2 章关于 PCI 设备的介绍中可以知道,PCI 设备暴露给操作系统的接口是 PCI 配置空间,一个很自然的想法是将真实设备的 PCI 配置空间暴露给客户操作系统。前面也提到,对于 PCI 配置空间中的 PCI BAR 通常使用的是虚拟 BIOS 生成的。那么更进一步,可以为设备生成整个虚拟的 PCI 配置空间。

为了让客户操作系统正确地识别分配给它的设备,这个虚拟的 PCI 配置空间中,表示设备标识的前 16 个字节(见 2.6 节关于 PCI)需要使用真实的信息,这是没有关系的,这些信息不会被客户操作系统修改,也不会引起冲突。将生成的 PCI 配置空间以一个虚拟设备的形式挂接在虚拟 PCI 总线上,当客户操作系统枚举总线时即可发现该设备并加载正确的驱动程序。

5.7.5 配置 DMA 重映射数据结构

在上一节介绍 VT-d 技术时,已经介绍 DMA 重映射进行地址转换的过程。对于 VMM 的实现者来说,使用该技术的关键是为所分配的设备正确设置根条目和上下文条目,以及建立 I/O 页表。每个客户机都有一张 I/O 页表,通常在客户机创建初期根据客户机的内存大小、VT-d 硬件支持的页表级数、页大小创建。下面用一个例子说明如何配置设备对应的根条目、上下文条目。

例如,假设 Guest ID 为 1,I/O 页表已经创建好位于地址 A,根条目表和上下文条目表已经在 VMM 加载初期创建好,要分配设备的 BDF 为{00: 03: 00}。

- (1) 找到设备对应的 DHRD 结构。
- (2) 获得该结构的根条目表,通过 BDF 的 bus 字段获得设备对应根条目(以下称 Root Entry0)。

(3) 通过 Root Entry0 的 CTP 字段获得上下文条目表,用 BDF 的 dev: func 字段索引该表,获得设备对应的上下文条目。如果该上下文条目不存在,分配一个并将地址填入 Root Entry0 的 CTP 字段。

(4) 将 I/O 页表的地址 A 填入上下文条目的 ASR 字段,在 DID 字段中填入 Guest ID 1,在 p 字段中填入 1。

(5) 刷新上下文条目的缓存。

上述步骤中,只介绍了主要字段的配置,其余字段也要根据格式正确配置。在刷新操作之后,该设备的 DMA 请求就会被 DMA 重映射硬件截获并进行地址转换。

5.7.6 设备中断虚拟化

DMA 最后一个步骤往往是设备用中断报告驱动程序操作完成。在设备直接分配给客户机,以及 DMA 重映射到客户机内存的情况下,设备的中断也需要注入给客户机。此部分内容已经在 5.4 节中介绍过了,在此不再累述。

5.7.7 案例分析: 网卡的直接分配在 Xen 里面的实现

上面的内容介绍了设备直接分配的种种技术,下面以网卡为例,介绍如何应用这些技术将一个物理网卡直接分配给客户机。为了方便起见,这里以 Xen3.2 为例,当然,其他 VMM 原理也是类似的。

首先,VMM 需要知道要直接分配的设备的标识,这里通常使用 PCI 设备的 BDF 号(Bus/Device/Function)来标识设备。设备标识是由用户指定的,因为是用户决定哪些设备被直接分配。用户可以通过 lspci 等工具获得 PCI 设备的 BDF 号,然后通过指定的方式告诉 VMM。这里假设网卡的 BDF 是 03: 00. 1。

知道设备的标识后,下一步就是隐藏该设备。如上所述,隐藏设备的方法有很多,Xen 的方法是在 GRUB 启动选项中指定设备的 BDF,然后在启动过程中不去使用这些指定的设备。以网卡为例,用户可以增加 Dom 0 的启动选项 pciback.hide = (03: 00. 0),这样,Dom 0 就不会为网卡(03: 00. 0)加载驱动,该网卡在系统启动之后就处在未使用状态。

设备被隐藏之后就可以被直接分配给客户机。同样的,用户需要在客户机的配置文件中指定该设备的 BDF 号,告诉 VMM 要将该设备直接分配。在 Xen 里面,用户可以在 HVM 的配置文件中增加 pci = ['03: 00. 1']。

Xen 获得直接分配的请求之后,首先需要将该设备呈现给客户机,这是通过在设备模型中虚拟一个 PCI 网卡来实现的。这个虚拟 PCI 网卡可以看作是该物理网卡在客户机中的代言人,客户机操作系统对该虚拟 PCI 网卡的操作请求都会被转交到物理网卡上来,物理网卡处理完后的结果又会通过虚拟 PCI 网卡返回给客户机操作系统。

Xen 负责虚拟 PCI 网卡和物理网卡之间的交互。具体来说,Xen 主要需要做如下的工作。首先,由于虚拟 PCI 网卡的 BDF 和物理网卡的 BDF 不一定相同,因此 Xen 负责进行两

者的转换。其次，虚拟 PCI 网卡的 I/O 空间(MMIO 和 I/O Port)和物理网卡的不一定相同，因此 Xen 也需要为两者建立映射。再次，虚拟网卡的中断和物理网卡也是不一样的，因此 Xen 负责进行中断的转发。最后，Xen 需要为物理网卡设置 VT-d 页表，确保客户机操作系统发出 DMA 请求时，硬件能够正确地处理 DMA 地址。这些工作的具体实现前面章节已经详细阐述过了，这里不再赘述。

5.7.8 进阶

在一个客户机被销毁后，它所拥有的真实硬件设备应该能够被重新利用，分配给其他客户机使用，这就要求设备能够进行再分配。通过前面的内容介绍，读者应该很快能够想到再分配的方法，只需要重新为设备生成新的 PCI 配置空间，建立映射，并挂接到新客户机的虚拟 PCI 总线上，并修改设备对应的上下文条目，就可以将设备分配给新的客户机使用了。其中涉及的步骤前面已经介绍，在此不再累述。

5.8 时间虚拟化

5.8.1 操作系统的时间概念

时间管理是操作系统的一个重要模块。在第 2 章已经介绍了操作系统使用时钟的两种方式，并指出时钟的虚拟化关键是正确的模拟时钟中断。在开始本节内容之前，先来介绍两个操作系统的时间概念。

- 绝对时间(Wall Time)：又称墙上时间。即操作系统启动后到目前为止的总运行时间，它是个单调递增的值。
- 相对时间：指两个时间之间的间隔。例如，两次时钟中断的间隔、两次使用 RDTSC 指令读取 TSC 间的间隔。

正如第 2 章介绍的，硬件定时器如 RTC、PIT 或者 HPET 等都能够以某种频率触发时钟中断，触发的频率可以由软件编程控制。通常，操作系统会将频率设定为一个给定的值(例如 10ms)，从而可以知道两次时钟中断之间的时间差(在本例中是 10ms)。同时，硬件定时器也会提供计数器(Counter)的功能，操作系统可以知道两次读取计数器之间的时间差，从而得到相对时间概念。

操作系统在启动的时候会读取 CMOS 的实时时钟，或通过 NPT 协议，得到系统启动时的绝对时间。同时，系统通过维护相对时间，可以知道系统总共运行的时间，从而操作系统可以得到任意时刻点的绝对时间，如下面的公式所示：

$$\text{当前绝对时间} = \text{系统启动时的时间} + \text{系统启动后运行的时间}$$

图 5-22 描述了操作系统的时钟概念。系统在时间 t_0 的时候启动，当实际时间到达 t_1 的时候，系统内部维护的运行时间为 $t_1 - t_0$ ，而系统内部的绝对时间为 $t_0 + (t_1 - t_0)$ ，从而

保证了内部时间与实际时间的一致性。每次时钟中断发生时(t_1, t_2, t_3 等),操作系统都会更新内部的时间概念。而在 $t_1 \sim t_2$ 时间内,操作系统可以通过读时间设备的计数器得到相对时间。

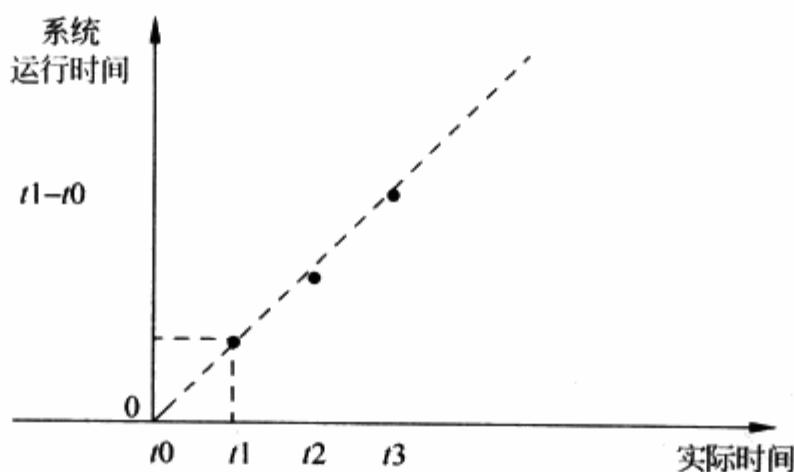


图 5-22 操作系统的时间概念

5.8.2 客户机的时间概念

在硬件辅助的虚拟环境下,客户机操作系统仍然需要维护正确的时间概念,包括相对时间和绝对时间。这意味着 VMM 需要为客户机提供系统硬件时钟设备的仿真,包括 PIT、HPET 和 TSC 等。下面将讨论 VMM 如何虚拟化这些时间设备。

在虚拟环境下,由于客户机是和其他的客户机以及 VMM 共享物理平台,客户机只能得到部分的处理器时间(即使当前只有一个客户机,VMM 本身运行也需要占用 CPU 时间)。在这种情况下,“正确的时间概念”是随着应用的不同有着不同的含义,如何维护正确的时间概念存在着诸多的问题。

图 5-23 和图 5-24 给定了不同的客户机时间概念的实现。假定客户机在 t_1 时刻处于运行状态, t_2 的时候被调度进入睡眠状态, t_3 的时候重新被调度执行,直到 t_5 的时候被再次调度出去。如果客户机内的某一个程序(操作系统内核或者应用程序)希望得到 t_4 和 t_1 之间的相对时间,那么返回值应该是多少呢?

显然,对于不同的情况,返回值应该是不相同的,考虑下面两种应用。

(1) 进程记账:主要用于统计某一个进程的执行时间。假定客户机内的某个进程在 t_1 被调度执行,在 t_4 的时候被调度出去(请注意不要与客户机本身的调度混淆)。显然,进程记账程序希望得到的时间是进程真正运行的时间,也就是 $(t_4 - t_3) + (t_2 - t_1)$ 。在这种情况下,客户机的时间和实际时间的关系如图 5-23 所示。

(2) 网络速度检测程序:网络速度检测程序通过向远端服务器发送数据包,远端服务器收到数据包以后,会发送一个应答包回来,通过计算发送数据包和收到应答包的时间,就可以大概了解网络速度。假定程序在 t_1 时向远程的服务器发送数据包,并在 t_4 的时刻得到服务器的反馈。显然,网络速度检测程序希望得到的是真正的时间,也就是 $t_4 - t_1$ 。在这

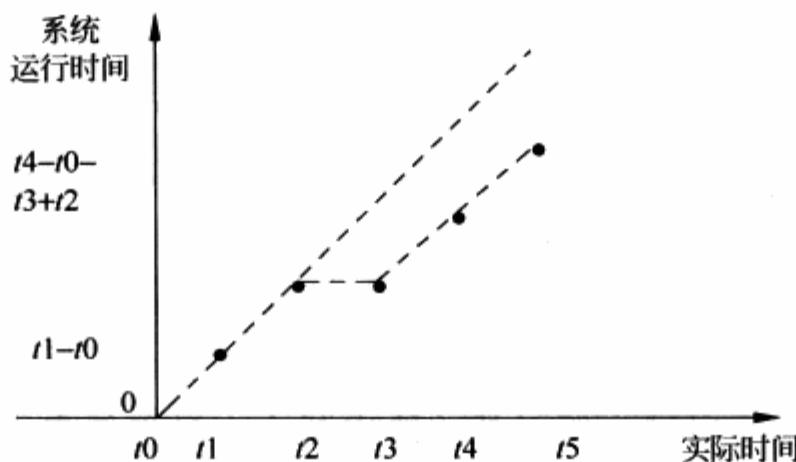


图 5-23 客户机的时间概念(客户机时间 != 实际时间)

种情况下，客户机的时间和实际时间的关系如图 5-24 所示。

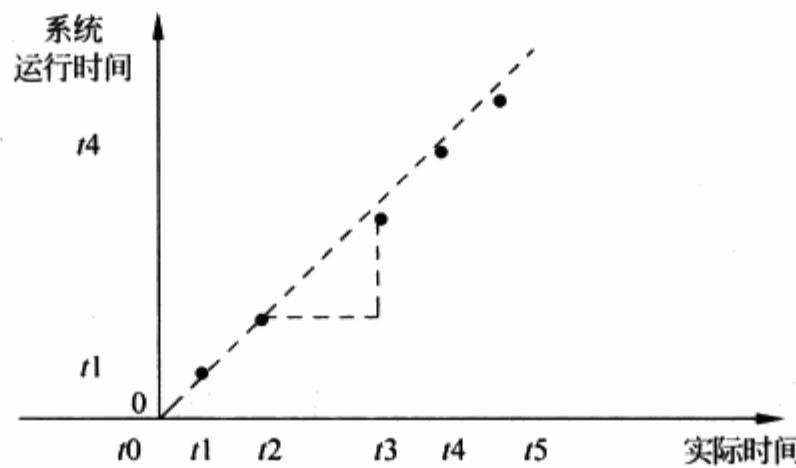


图 5-24 客户机时间概念(客户机时间 == 实际时间)

在实际需求中，大多数的应用都是像网络速度检测这样的应用，因此，通常时间虚拟化的策略都是给客户机呈现与实际时间相同的时间概念。后面的讨论均是基于客户机时间与实际时间相等的情况。对于需要客户机时间与实际时间不同的情况，会在最后进行简单的介绍。

如前所述，操作系统通过系统中的时钟设备（包括 PIT、HPET 和 TSC 等）得到自己的绝对时间或相对时间，因此首先介绍时钟设备虚拟化的实现方法，然后再讨论如何给客户机提供与实际时间相等的时间。

5.8.3 时钟设备仿真

x86 系统中的时间设备包括 PIT、HPET、ACPI PM Timer 和 TSC 等。本节介绍客户机不会被调度出去的情况，PIT 设备如何虚拟化。然后会简单介绍 HPET、ACPI Timer 等其他时间设备的虚拟化。客户机被调度出去的情况在下一节介绍。

第 2 章介绍了 PIT 的基本概念。其功能主要是为操作系统提供定时的时钟中断和时钟计数器。操作系统通过对 PIT 设备的 I/O 端口读写，设定期钟中断的触发频率，设置和读取时钟计数器。

为了实现时间设备的虚拟化,VMM 必须要提供软件定时器机制,使得程序可以设定在某个未来的时间执行一段代码(参见第 2 章关于操作系统定时器的说明),同时还需要提供接口,使得程序可以了解当前的实际时间。

假定客户机操作系统设定 PIT 时钟中断频率为 10ms,VMM 截获这一设定(如何截获的方法请参考前面对 I/O 设备虚拟化的讨论),并通知 PIT 设备模型。PIT 设备模型会向 VMM 注册一个间隔为 10ms 的软件定时器,并提供回调函数,这个回调函数的功能就是向客户机注入一个时钟中断。如果客户机不被调度出去,每隔 10ms,VMM 都会调用这个回调函数,向客户机注入一个时钟中断。具体的中断注入过程请参考前面中断虚拟化相关章节。

当客户机读取 PIT 的 Counter 寄存器时,PIT 设备模型通过 VMM 了解当前的实际时间,并减去 PIT 的时间计数器被初始化时的实际时间,以得到这之间所流逝的时间,经过 PIT 频率的转换后返回给客户机。

对于 HPET 和 ACPI PM Timer,其基本的实现方法相同,不同点在于,客户机读取 PIT 设备是通过 IO 实现,而对于 HPET 和 ACPI PM Timer 是通过 MMIO 截获实现的。同时,各个时间设备的中断号并不相同。

由于操作系统可以依赖于多个时钟设备实现内部时间的维护,因此,当 VMM 提供多个时钟设备的仿真时,需要保证各个设备模型之间的时间一致性。

5.8.4 实现客户机时间概念的一种方法

下面讨论当客户机被调度出去的情况下,如何通过设备仿真实现客户机的时间概念,以使得客户机时间等于实际的时间。如图 5-25 所示,假定客户机在 t_2 的时候被调度出去,并在 t_5 的时候被调度进来。在这个过程中,根据客户机操作系统对虚拟 PIT 的设置,在 t_3 和 t_4 的时候需要插入时钟中断,以使得客户机操作系统能够维持内部的时间计数。然而,在 t_3 和 t_4 的时候客户机并没有运行,因此,VMM 没有机会把中断注入给客户机。

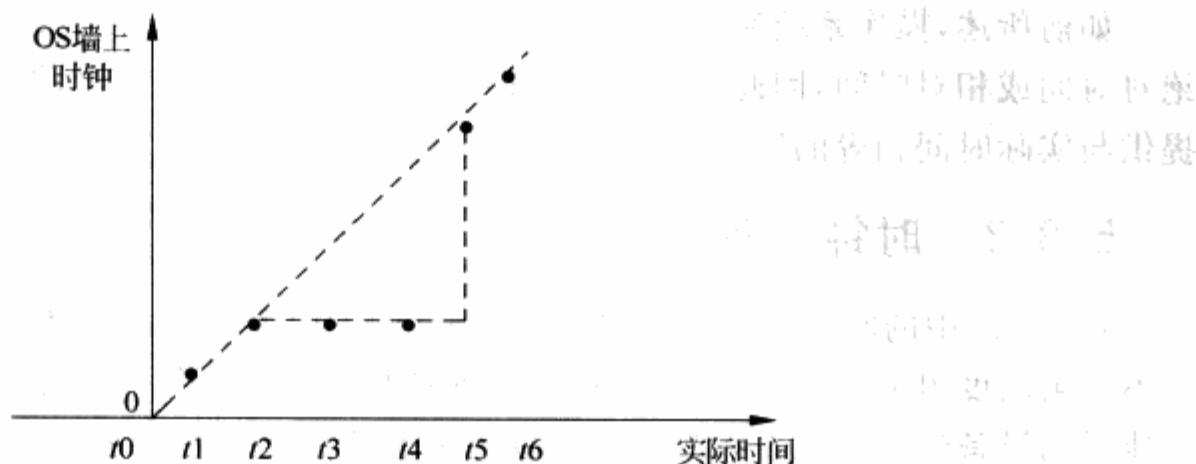


图 5-25 客户机被调度情况下的时钟实现

通常的做法是,当客户机在 t_5 时刻被调度回来以后,VMM 连续把 t_3, t_4 时刻丢掉的两个时钟中断连续地注入给客户机。“连续”的意思就是,当客户机处理完 t_3 的时钟中断后,立刻把 t_4 时刻的时钟中断注入给客户机,在客户操作系统看来就在一个时钟中断处理完后另一个时钟中断紧接着就发生了。由于这个过程中没有其他的程序被运行,因此,当应用程序或者内核中需要时间服务的程序运行的时候, t_3 和 t_4 丢失的时钟中断都已经补偿给客户机了。这样,VMM 保证了客户机内部的时间与实际时间一致。当然,如果在操作系统的时钟中断处理函数中,有需要时钟服务的代码运行,那么这些代码仍然会得到不正确的时间。但是,操作系统为了保证中断的快速反应,通常并不会出现这种情况。

图 5-26 给出了实现这一过程的微观示意。当客户机在 t_5 时刻被调度运行时,VMM 立刻注入 t_3' 时刻的时钟中断给客户机,使得客户机的时钟概念跳变到了 t_3' ,客户机在 t_5' 的时候执行完 t_3' 中断的中断处理函数后,VMM 立刻注入 t_4' 时刻的时钟中断,使得客户机的时钟跳变到了 t_4' 。由于客户机的时钟处理函数执行速度都很快,因此 $t_5' - t_5$ 远远小于 $t_6 - t_5$ 。

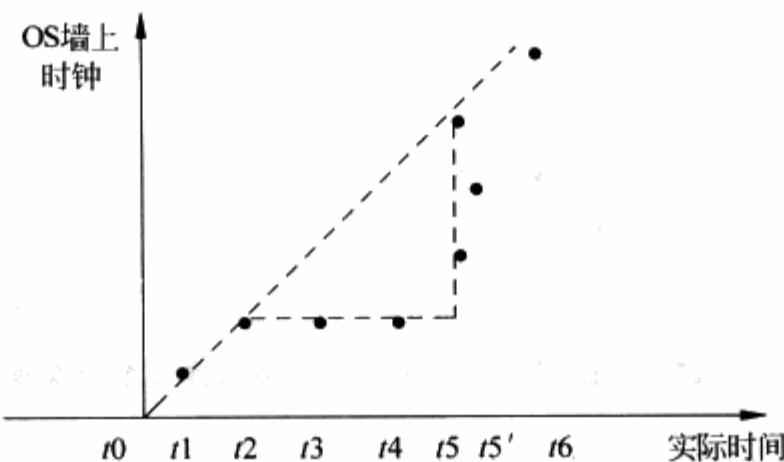


图 5-26 客户被调度出去的情况下中断注入的微观示意图

下面考虑这种情况下,设备模型中计数器的实现方法。在图 5-26 中,在 t_5' 后,由于所有的中断都已经被注入到客户机内,客户机的时钟已经调整到和实际时间一致,设备模型中计数器的值与实际时间也是一致的。而在 t_5 到 t_5' 的过程中,由于客户机的时钟概念仍然停留在 t_3 的时刻,因此计数器的返回值也应该在 t_3 和 t_4 之间。由于 t_5 到 t_5' 的时间非常短,因此,具体在 t_3 和 t_4 之间的那个点依赖于具体的实现。

5.8.5 实现客户机时间概念的另一种方法

上面描述了实现时间虚拟化的一种常见的方法。但是,这种实现方法也存在着一些问题。例如,由于需要把客户机被调度出去时的时钟中断补偿回客户机,因此它对系统的性能会有一定的影响,特别是在运行的客户机数量比较多的情况下,时钟中断的补偿会消耗很多的时间。其次,在 SMP 的情况下,各个 VCPU 之间 tsc 的同步存在着一些问题。

正如前面提到的,时钟虚拟化的主要目的是保证客户机内部的时间概念的正确性,因此,可以针对操作系统特定的时间概念维护机制,修改 VMM 时间虚拟化的方法。本节给出一个其他的实现方法。

在硬件平台上,虽然时钟中断丢失的情况非常少见,但还是存在这种可能性,通常的例子是因为操作系统关闭中断的时间过长。然而,有些操作系统已经考虑了时钟中断丢失的情况,它们在收到时钟中断后,会读取时钟设备中的计数器,并根据计数器的值进行修正。如图 5-27 所示,假定操作系统没有收到 t_3 时刻的时钟中断,当 t_4 时刻时钟中断注入的时候(注意,这是在物理机器上,因此 t_3 时刻的中断并没有机会补偿给操作系统),中断处理函数会读取时钟设备中的计数器,发现和上一次时钟中断的发生已经超过了 10ms,因此操作系统会认为自己错过了时钟中断,并根据时钟设备中的计数器来修改自己的时钟概念。

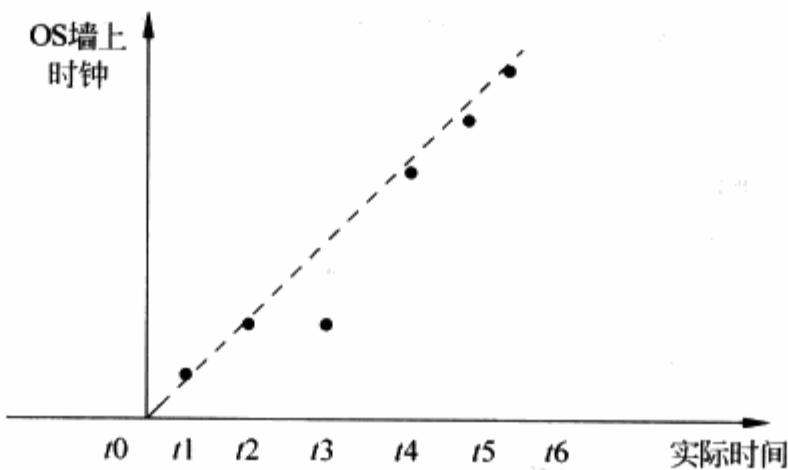


图 5-27 考虑时钟中断丢失情况下的操作系统实现

针对这种操作系统,时钟虚拟化不再需要将错过的中断补偿给客户机。仍然以图 5-25 为例子进行说明。在 t_5 时刻客户机被重新调度运行的时候,VMM 并不需要将 t_3/t_4 时刻的时钟中断都注入客户机,而只需要注入一次。当客户机的时钟中断处理函数读取时钟设备的计数器时,直接返回实际时间。客户机操作系统通过检查计数器返回值,就可以知道错过了 t_4/t_3 的时钟中断,并把自己的时间概念更新到 t_5 。

5.8.6 如何满足客户机时间不等于实际时间的需求

前面讨论了如何通过设备仿真,使得客户机内部的时间概念与实际的时间相等。然而,正如在前面所说,有些应用需要客户机时间等于客户机实际运行的时间,例如进程记账软件。对于这种需求,一种做法是在客户机内部引入 PV 的时间模块,使得客户机被调度出去的时间(也就是图 5-25 中的 $t_2 \sim t_5$)被计算在 PV 的时间模块,而不是当前进程上。更详细的资料请参考 VMware 的相关文献^[21,22]。

5.9 思考题

- (1) Intel 处理器引入哪种新的运行模式来支持虚拟机？这个运行模式有哪些特性？
- (2) 对于旧的用 2 级页表并运行在 4 级 EPT 的 32 位客户机操作系统，要完成指令 mov [%esp], %eax 需要多少次物理内存访问？如果是用影子页表，又需要多少次物理内存访问？

类虚拟化技术

除了利用第 4 章的陷入与模拟技术外,还有一个方法能够在存在虚拟化漏洞的体系结构上实现系统虚拟化。在计算机硬件不能改变的情况下,VMM 提供给虚拟机的硬件抽象是可以修改的。这里所说的硬件抽象,是指硬件平台掩去了内部的具体实现,暴露给软件的抽象接口。完全虚拟化的系统需要给虚拟机提供与硬件平台完全一致的硬件抽象,这样操作系统以及上层软件才能够不做修改地在虚拟机中运行。类虚拟化(par-Virtualization)技术的主要思想就是通过修改暴露给虚拟机的硬件抽象以及上层操作系统,使得操作系统与 VMM 配合工作,避开虚拟化漏洞,从而实现系统虚拟化。修改后的操作系统能够意识到虚拟环境的存在。

虚拟硬件抽象能够被修改成各种形式,因而对应的虚拟机也各种各样。例如,虚拟硬件抽象可以不包括虚拟内存的支持,这样,在虚拟机中只能运行实模式的系统和应用软件。每个类虚拟化的系统都可以有各自的抽象层设计。与完全虚拟化的系统不同,为了实现不同的目的,两个类虚拟化系统的设计可以有很大的区别。

当然,虚拟硬件抽象与实际硬件的差别越大,客户机操作系统和应用程序需要做的修改也越大。为了尽可能地保持上层操作系统和应用软件的兼容性,Xen 系统提供了一种实现方式。在 x86、安腾等体系结构上,只需要对客户机操作系统作少量的改动,无须对应用程序作任何改动,常用的操作系统如 Linux、NetBSD 和 Windows 等就能够运行在 Xen 系统上。

本章后面的内容将以 Xen 为主线介绍类虚拟化的主要原理。需要说明的一点是,本章内容不是介绍 Xen 虚拟化系统本身,而是以 Xen 为例来介绍类虚拟化的原理。6.1 节对类虚拟化技术从宏观上作一个概述。在 6.2 节,将具体介绍 Xen 的虚拟硬件抽象。Xen 的内部机制将在 6.3 节中展开介绍。在 6.4 节把前面小节的技术合在一起,介绍 Xen 的虚拟机是怎样运行的整个流程。

6.1 概述

6.1.1 类虚拟化的由来

类虚拟化技术通过暴露给客户机操作系统一个修改过的硬件抽象，修改部分操作系统的代码，使得操作系统与 VMM 配合，实现系统虚拟化。

“类虚拟化”这个名词的翻译比较有争议。与完全虚拟化对应，Para-Virtualization 在一些文章中被译为“部分虚拟化”或“半虚拟化”。由于 para 前缀在英文中有“类似”和“辅助”的意思，因此也有文章将其译作“泛虚拟化”或“协同虚拟化”。这些翻译都反映了 para 词缀具有多重含义。最直接的类虚拟化技术需要修改现有的操作系统，因而打破了传统的虚拟化三个要求中的第一条，即同质性。类虚拟化的虚拟层暴露给上层操作系统的不是一个现实存在的硬件抽象层，而是一个更改过的形式。因而，上层操作系统需要经过修改才能运行于这个新的虚拟硬件抽象上。从这个意义上讲，类虚拟化不是传统意义上的虚拟化。“泛虚拟化”中带有超越传统的更广义的虚拟化的意思，由于这种翻译偏抽象晦涩，本书中将不采用。另一方面，作为一种新的虚拟化形式，类虚拟化技术也同样能支持在一个物理机器上运行多个虚拟机。并且，通过客户机操作系统与底层的 VMM 配合工作，类虚拟化技术将虚拟化的开销进一步降低。从这个意义上讲，“协同虚拟化”表达了上下协作工作的一层含义。这些翻译都有其优劣，本书经过权衡，在后面章节将以“类虚拟化”来指代这种独特的虚拟化技术。

类虚拟化技术代表了一个方向，即改变现有的为真机设计的操作系统，使操作系统意识到虚拟环境的存在，更好地配合 VMM，是更适合运行在虚拟环境中的操作系统。这种意识也被称为受启发或点化的(Enlightened)。

类虚拟化的优势大致有如下三点。

(1) 类虚拟化系统能够为降低虚拟化技术带来的性能开销作最大限度的优化，其主要途径包括消减冗余代码、减少地址空间切换和跨特权级切换、减少内存复制等。操作系统与 VMM 都会带有 CPU 调度、内存管理以及外设驱动等代码，其功能逻辑很多是重复的。在完全虚拟化的系统中，一个磁盘读写请求要完整地经过虚拟机操作系统中的磁盘驱动和 VMM 的真实磁盘驱动两层代码逻辑。在类虚拟化系统中，操作系统里冗余的设备驱动就可以被替换成精简的调用服务的钩子(Stub)，从而缩短一个系统服务的代码路径。

(2) 类虚拟化系统在一定程度上消除了虚拟层和上层操作系统的语义鸿沟(Semantic Gap)，使得整个系统的管理更为方便有效。在这里，语义鸿沟指客户机操作系统内部的运行状态不能够被 VMM 所获得，从而无法最优地调配资源。例如，操作系统需要使用的内存总量是动态变化的，传统上一个虚拟机启动时会被分配到固定大小的内存，VMM 并不知道一个虚拟机在某一时刻具体需要多少内存。而如果这一语义鸿沟能够被打破的话，

VMM 能够在多个或繁忙或空闲的虚拟机之间平衡内存的使用,提高内存使用的效率。

(3) 类虚拟化技术还能够用于其他探索和应用。由于能够修改操作系统,类虚拟化方法能够尝试在不同抽象高度提供硬件抽象,甚至提供语义更强大的硬件抽象接口,来优化性能,提供新的功能。

当然,类虚拟化的一个很大的问题是需要对操作系统进行修改。第一,类虚拟化增加了操作系统开发与调试的工作量。每一种操作系统都需要移植后才能运行在类虚拟化 VMM 上。对于同一个操作系统的不同版本,也需要对类虚拟化的支持代码进行相应的修改或维护。第二,对于非开源的操作系统,其修改和维护会有较大的障碍。另外,对于已经停止开发维护的经典操作系统,在其中加入类虚拟化的支持也比较困难。

对于第一个问题,一个缓解的方法是将类虚拟化的支持代码划分出来,封装成一个方法集合,操作系统中的其他代码通过接口来调用这些方法。这样,在操作系统版本衍化的过程中,类虚拟化的支持代码和其他部分的代码能够各自开发,从而最大程度地减少维护工作。

在 x86 等平台的硬件虚拟化辅助技术日趋成熟的情况下,系统虚拟化的实现可以大大地简化,但类虚拟化技术依然有其存在的意义。首先,硬件的支持无法解决所有虚拟化的问题,例如语义鸿沟和冗余代码是不能通过硬件支持来解决的。其次,类虚拟化为虚拟化系统的深度优化提供了空间。

6.1.2 类虚拟化的系统实现

类虚拟化技术修改了硬件抽象,操作系统也需要作相应的修改。然而,如何修改硬件抽象和操作系统并没有一个统一的标准。事实上,不同的类虚拟化的系统为了实现不同的目的,对于硬件抽象层和操作系统的修改都是不同的。最著名的 Denali 系统和 Xen 实现的方式就有很大的差异。Denali 系统的设计目的是在一台物理机器上运行 1000 台或以上的虚拟机,这些虚拟机之间差异很小,能够很大程度上共享资源,并且大多数虚拟机不会同时运行。Xen 的设计目的是在一个 x86 平台上高效运行 100 台以下的虚拟机,并且为了兼容性,客户机操作系统向应用程序提供的应用程序二进制接口 (Application Binary Interface, ABI) 不变。这样,应用程序不需要修改即能运行在虚拟机操作系统上。Xen 支持的类虚拟化客户机操作系统包括 Linux、NetBSD 等,其对应的移植版本叫做 XenLinux、XenNetBSD。下面分别简要介绍一下 Denali 和 Xen 系统。

1. Denali

Denali 系统在 2002 年由美国华盛顿大学的 Andrew Witaker 等设计实现,旨在将不受信任的应用程序隔离在单独的执行环境中运行的内核。其论文发表在操作系统研究领域最高级的两大会议之一的操作系统设计与实现大会(OSDI)上。

Denali 的虚拟机是轻量级的,其允许 1000 台以上的虚拟机在同一硬件平台上同时运行,以达到高可扩展性(Scalability)。为此,Denali 不像传统虚拟化技术中的 VMM 一样精确、完整地模拟整个真实硬件体系结构,而是为客户机呈现了一个简化了的虚拟硬件抽象。

在 x86 体系结构上运行的操作系统,需要经过移植才能在 Denali 的虚拟硬件体系结构上作为客户机操作系统运行。

Denali 为客户提供虚拟指令集是 x86 指令集的一个子集。大多数客户机的指令可以直接在物理 CPU 上运行。与第 4 章提到的用扫描和修补技术或 VMM 模拟执行敏感指令和特权指令不同的是,Denali 不支持模拟执行这些指令。经过修改,操作系统避免了直接使用这些未被 Denali 支持的指令。另一方面,VMM 为客户提供操作系统提供了基于虚拟机调度而非阻塞式的 idle 指令,提高了 CPU 资源利用的效率。

由于 Denali 的轻量级虚拟机运行的是小型的应用程序,Denali 中的每个虚拟机只有一个内存地址空间。客户机和 Denali 内核共享这个地址空间,这样避免了虚拟机内应用程序间切换时的 TLB 刷新和虚拟机与 VMM 间切换时的 TLB 刷新。

由于 Denali 上同时运行了许多个虚拟机,当属于某个虚拟机的中断发生时,这个虚拟机很有可能不再运行。这时,Denali 并不立即唤醒这个虚拟机以处理中断,而是将属于这个虚拟机的待处理的中断批量集中起来。等到这个虚拟机下次被轮循调度到时,只需要经过一次 VMM 到虚拟机的切换,客户机就能批量处理所有分发给它的中断。

2. Xen

Xen 类虚拟化系统最早是在 2003 年由英国剑桥大学的系统实验室开发的,其论文发表在操作系统研究领域最高级的两大会议之一的操作系统原理大会(SOSP)上。由于 Xen 系统是开源的,其流传程度十分广泛,对于后来的虚拟化技术的研究有很大的影响。

图 6-1 是 Xen 系统的整体结构图。虚线框围住的部分表示虚拟化层的范围,它是由一个 VMM 和一个特权虚拟机组成。

在 Xen 的术语中,每一个系统就是一个域(Domain)。图 6-1 中特权虚拟机被称为 Domain0(或 Dom0),而其他虚拟机被称为 DomainU(或 DomU)。

由于是类虚拟化,Dom0 和 DomU 的操作系统都是需要移植方能够运行的。

类虚拟化系统的实现形式是可以多种多样的,要进一步介绍类虚拟化技术,需要借助于特定的系统来阐述。目前,x86 体系结构中,Xen 是类虚拟化的代表性系统之一。下面的介绍将主要围绕 Xen 和 XenLinux 展开,但需要强调的是,Xen 只是类虚拟化可能的设计和实现的一种。

6.1.3 类虚拟化接口的标准化

x86 架构上支持类虚拟化 Linux 的系统包括 Xen、lguest、KVM、VMware ESX Server 和微软的 Hyper-V 等。由于每个虚拟化系统对于虚拟硬件抽象的接口各有不同,Linux 中定义了一些标准的类虚拟化接口来覆盖和统一各个类虚拟化解决方案的接口,使得 Linux 操作系统能够在启动时选择使用哪种接口来运行。

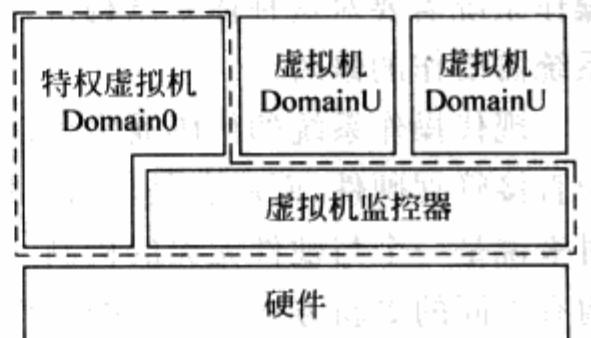


图 6-1 Xen 整体结构

VMI(Virtual Machine Interface)是 VMware 公司在 2006 年提出的一种操作系统与 VMM 连接的方法。VMI 的方案中,在操作系统启动时,它会加载一个模块,或称为 ROM。之后,操作系统会调用这个模块中的方法与 VMM 交互。在启动时,操作系统能够根据下层所运行的 VMM 来加载不同的 ROM。这样,同一个操作系统镜像就能够在不同的 VMM 上运行,而不需要重新编译。

由于 VMI ROM 是编译好的二进制模块,它有时不是由内核开发人员编写的。因而,内核开发人员将一部分 ROM 的功能加入了 Linux 内核中,并用了一个统一的接口 Paravirt_ops (pv_ops)来调用。在目前的 x86 平台 Linux 内核,有 Xen、KVM 和 lguest 三个系统虚拟化方案加入了内核开发树。

微软的 Hyper-V 系统中有一层接口转换层,能够兼容 Xen 的类虚拟化接口。

Linux 在启动时能够选择加载真实硬件或者虚拟硬件的 VMI ROM,也可以从各种类虚拟化 Paravirt_ops 集合中选择一个集合。如果直接运行在硬件上,那 Linux 可以选择真实硬件的 ROM 以及 Paravirt_ops 集合。

6.2 类虚拟化体系结构

从操作系统的角度来看,类虚拟化的硬件抽象是一种与 x86 架构有所不同的体系结构。操作系统需要对这种体系结构进行移植。本节将检视一下类虚拟化的硬件抽象,以及操作系统需要作的修改。

现代操作系统的代码通常可以分成硬件相关和硬件无关两种,前者更靠近底层硬件,用于直接管理硬件,而后者是比较高层的软件逻辑。拿内存管理模块来说,如何调配内存,相对来说是一个与硬件无关的机制,而更新页表映射的操作则与具体硬件相关,不同的体系结构有不同的更新方式。高层的硬件无关代码在不同的体系结构间是可以通用的,因而在不同的体系结构间移植一个操作系统,需要改变的通常只是底层与硬件相关的代码。

为了使得操作系统更具有可移植性,现代操作系统通常将与硬件关系紧密的代码集中起来,封装成为一个硬件抽象层。有了硬件抽象层的封装后,操作系统中与硬件相关的代码就不会散落在其他代码中。

XenLinux 是 Linux 修改了硬件抽象层代码以及加入了一些特殊的设备驱动得到的。其修改的部分主要包括指令集、外部中断、内存空间及内存管理方式、I/O 设备驱动、时钟等。下面将分别加以解释。需要强调的是,本节的虚拟硬件抽象是指提供给一个虚拟机的,在 Xen 系统的构成中,Dom0 处于比较特殊的位置,它能够直接管理一部分 I/O 设备,所以 Xen 提供给 Dom0 的硬件抽象有所不同,在这里不作讨论。

6.2.1 指令集

类虚拟化虚拟机能够使用的指令是实际 CPU 指令的一个子集,它不包括特权指令和

敏感指令。对于敏感指令,更准确地说,只是这些指令对于敏感信息的操作不被支持。Xen 提供了与这些指令功能相对应的函数,客户机操作系统能够以类似于系统调用的同步通信方式来调用这些函数。与系统调用相对应,这种从操作系统到 VMM 的调用通常被称为超调用(Hypercall)。

为了防止客户机操作系统执行特权指令,通常的做法是降低客户机操作系统的特权级。在 Xen 中,VMM 运行在特权级 0,而客户机操作系统运行在特权级 1。如果客户机操作系统使用的指令超出了 VMM 支持的指令集,由于特权级保护机制,这些指令的执行会发生陷入。对于敏感指令,操作系统虽然能够执行这些指令,但由于这些指令没有 VMM 的支持,因而可能返回真实硬件的一些执行结果,这样,操作系统就可能无法正确处理。值得注意的是,虽然这些敏感指令能够执行,但由于敏感指令不包括那些写指令,故其不会影响到整个系统的安全性。换句话说,即使操作系统不遵照 Xen 的规范来使用敏感指令,这个违法的行为只会影响到那个虚拟机本身,而不会影响到系统其他部分的安全与隔离性。

下面以两个 x86 构架的具体指令来看一下特权指令是如何由超调用所替代的。HLT 指令用于在没有工作时停止 CPU 运行,进入省电模式,直到被下一个外部中断唤醒。类虚拟化操作系统不会使用这条指令,而是发起一个超调用来告知 VMM 没有需要运行的工作了。在超调用服务函数中,VMM 要做的是将客户机的当前 VCPU 移出等待队列,使得它不再被调度到硬件 CPU 上运行。由于 VMM 上可能同时在运行多个虚拟机,因此一个虚拟机或一个 VCPU 闲置时,其他虚拟机很可能有正在等候的任务。只有在系统中所有虚拟机都闲置时,VMM 才会发起 HLT 指令,让 CPU 进入节能状态。

再如,LGDT 指令用于给 GDTR 寄存器装载 GDT 表。Xen 允许客户机操作系统装载自定义的 GDT 表,客户机操作系统发起一个超调用来装载 GDT 表。在 Xen 中的超调用服务函数会验证作为 GDT 表装载的页的类型是 GDT 数据页,并验证这些 GDT 描述符项的完整性。Xen 同时在 VCPU 数据结构中保存指向这些 GDT 页的指针,当虚拟机切换时,Xen 会装载被调度执行的客户机的 GDT 表到硬件寄存器中。

6.2.2 外部中断

在虚拟化环境下,一个虚拟机将不会直接收到来自硬件的外部中断,而只会收到由 VMM 注入的虚拟中断^①。

当 VMM 向虚拟机注入一个虚拟中断时,一个虚拟机可能正在 CPU 上运行,也可能在调度队列中处于等待状态。如果是后一种情况,只有在虚拟机下一次被调度运行时才能处理这个中断事件。因而,需要一种异步的通信机制将中断传递给客户机操作系统。

根据来源分类,一个虚拟机可能收到的虚拟外部中断有 4 种:来自物理外部中断、来自 VMM、来自同一个虚拟机其他 VCPU 以及来自其他虚拟机。

^① 直接由硬件向特定虚拟机发送中断的技术目前尚处于研发中。

最容易理解的虚拟中断是来自物理外部中断的事件,称作物理 IRQ(pIRQ)。例如,一个来自磁盘 A 的 IRQ,经过 VMM 的注入,在虚拟机中就收到一个 pIRQ。这一类虚拟中断来自真实硬件设备,这也就暗示说这个虚拟机拥有对磁盘 A 的设备驱动,磁盘 A 被直接分配给了这个虚拟机。

来自 VMM 的中断事件称为虚拟 IRQ(vIRQ)。vIRQ 不是从硬件收到的 IRQ,而是由 VMM 生成的。例如虚拟时间中断,每个虚拟机可以注册不同时间间隔的虚拟时间中断,VMM 每个指定的时间间隔为虚拟机注入一个时间中断事件。

如果一个虚拟机有超过一个 VCPU,多个 VCPU 之间就可以通过虚拟 IPI 来同步。虚拟 IPI 可以是从一个 VCPU 到其他 VCPU 的中断群发机制,也可以是一个 VCPU 到另一个 VCPU 的点对点中断事件。

一个虚拟机还能够收到来自一个虚拟机的虚拟中断,称为虚拟机间中断(Inter-Domain Interrupt,IDI)^①。一个虚拟机注册一个 IDI 中断,再通过某种通信方式将 IDI 号码告知另一个虚拟机,这样后者就能够通过这个 IDI 号向前者发送虚拟中断事件了。在 Xen 系统中,虚拟机间中断常用于 DomU 与 Dom0 交互,完成 I/O 请求,这部分内容将在下面的 I/O 子系统部分介绍。

6.2.3 物理内存空间

在虚拟环境中,虚拟机所拥有的物理内存是非连续的。多个虚拟机共享宿主机的物理内存,因此各个虚拟机所拥有的物理页在物理内存中可能是交错分布的,并且可能会动态地发生变化,例如 VMM 有换页机制或者虚拟机迁移等。

如前面章节所介绍的,在虚拟环境中,除了虚拟地址和物理地址外,还有一层地址映射,即实际在 CPU 中使用的地址,在 Xen 的术语中被称为机器地址。

传统的操作系统不能理解非连续的动态变化的内存空间和三层的地址映射。但类虚拟化操作系统可以通过暴露给操作系统更多的信息来使之理解其所处的虚拟环境。Xen 将物理地址到机器地址的翻译表(Physical-to-Machine 表,P2M 表)暴露给操作系统,这样操作系统就能够直接使用机器地址来更新页表映射了。

另一个物理内存相关的问题是操作系统如何侦测可用物理内存。在非虚拟化环境中,操作系统可以通过调用 BIOS 的服务来获得一张内存分布表。例如,PC 兼容机的 BIOS 能够通过中断请求获得一张 E820 表,其中包含了物理内存空间中 ROM 和 RAM 等的分布。在类虚拟化的虚拟机启动时,操作系统不能获取系统真实的 E820 表,而是从 VMM 中获得属于本虚拟机的内存空间信息。

^① 特权虚拟机还能够注册接收物理 IRQ,从而能够直接驱动外部设备。这种类型的 IRQ 在这里不加以讨论。

6.2.4 虚拟内存空间

x86 架构的另一个问题是 TLB 是由硬件管理的，软件无法很好地管理 TLB。如果 VMM 与虚拟机处于不同的虚拟内存空间，这样在从 VMM 到虚拟机或者从虚拟机到 VMM 都需要进行地址空间切换，而这样的切换都会刷新整个 TLB，从而带来很大的性能损失。另外，VMM 与虚拟机驻留不同虚拟内存空间还有一个问题，即虚拟机中至少需要有一段代码来完成地址空间切换。

对于这个问题，Xen 采用的方法是将 VMM 和虚拟机置于一个虚拟地址空间，这样就解决了 TLB 刷新的问题。为了使 Xen VMM 能够放入 4GB 的虚拟地址空间，Xen 将操作系统可用的 4GB 的虚地址空间压缩了，将最靠近 4GB 顶端的 64MB 虚地址划出来给 Xen VMM 使用。

6.2.5 内存管理

由于 VMM 和客户操作系统共用同一虚拟地址空间，必须有一种机制来保证 VMM 所占据那部分地址空间不被客户机操作系统所访问。通过设定段描述符中的相关标记位，可以限定访问该段的特权级。在 Xen 中，VMM 运行在特权级 0，而客户操作系统运行在特权级 1。因此，通过对段描述符的适当修改，可以限定只有运行在特权级 0 上才能访问 VMM 所在的虚拟地址空间，从而实现对 VMM 地址空间的保护。

在 Xen 的实现中，VMM 和客户机操作系统内核使用不同的 GDT 和 LDT。内核段被设成特权级 1 可以访问，段的界限被设成 VMM 空间的起始地址以下。在客户操作系统启动时，VMM 会向客户机操作系统提供默认的 GDT，该 GDT 并不在分配给客户机操作系统可写内存范围之内。如果客户操作系统想使用默认 GDT 所能提供的特权级 1 和特权级 3 以外的段，就可以在 GDT 表中分配新的 GDT 以及 LDT，并且向 VMM 进行注册。

x86 架构的分页机制可以被用来作虚拟机之间的内存隔离，只要控制了页表的更新，就能控制整个物理内存各虚拟机之间的分配。在 VMM 中，需要以某种方式记录每个虚拟机所分配到的物理内存页。一个虚拟机不能使用不属于自己的内存页，即不能在页表中建立非法的映射。

对于客户机操作系统，页表页都是只读的，任何对于页表页的修改都会陷入 VMM。这保证了客户操作系统不能随意修改自己的页表，从而保证了一个虚拟机不能随意访问其他虚拟机的内存，起到了虚拟机之间的内存隔离作用。

客户机操作系统可以通过发起超调用来更新页表项。VMM 会对更新的请求进行验证，任何试图映射所辖内存以外物理页的更新都会被禁止。客户机操作系统不能创建到页表页的可写映射。相对普通的内存访问，超调用是开销比较大的操作，所以一个可能的优化是将多个页表项的更新请求缓存起来，在达到一定数目后用一次超调用进行集中更新。

另外，对于单个页表项的更新，VMM 也可以通过陷入与模拟的方式完成。简单地说，

客户机操作系统直接修改只读的页表页,从而触发一个页保护错误。VMM 在页错误处理函数中模拟执行发生页错误的指令,先验证其更新内容的合法性,通过后 VMM 代为完成页表项的更新。这种方式相对于上一种方式实现难度更大,性能也比较差,不能支持批量的更新请求,但能够减少对于操作系统的修改,不需要操作系统显式地发起超调用来修改页表。

6.2.6 I/O 子系统

在完全虚拟化中,I/O 驱动包括客户机操作系统的硬件驱动、设备模拟层,以及 VMM 中真实的硬件驱动。类虚拟化系统不需要如此冗余的三层架构,I/O 交互能够以高于 I/O 指令级别的抽象来实现。

一个类虚拟化的 I/O 子系统至少需要由三个部分组成:一个 I/O 请求发起机制、一个异步的反馈机制和高效的数据交换机制。

在 Xen 中,客户机操作系统不使用任何现有的硬件设备驱动,而是使用一种前端端(Frontend/Backend)交互的设备驱动来发送 I/O 请求和接收 I/O 反馈。处于客户机操作系统内的一端称为前端设备驱动,处于 Dom0 的一端称为后端设备驱动。Xen 的类虚拟化设备驱动区分设备型号,而使每个设备类型使用一种设备驱动,称为类设备驱动(Classed Device Driver)。例如,最常见的磁盘和网卡,这些设备的驱动是通用型的,即虚拟机能够配备网卡设备,但并不是某个具体型号的网卡,如 RTL8139 百兆以太网卡或者 Intel 千兆网卡。

Xen 的 I/O 请求发起机制是用了一个称为环形缓冲区(Ring Buffer)的机制实现的。环形缓冲区是一个虚拟机与 VMM 间共享的页,用于存放 I/O 请求和 I/O 响应的描述符。需要注意的是,这个缓冲区并不存放实际读写的 I/O 数据,而仅仅是 I/O 请求的描述信息,因而一个页能够存放许多个请求。

环形缓冲区是前端驱动通过一个超调用初始化的。为了发送一个 I/O 请求,前端驱动通过超调用将请求描述符写入环形缓冲。与之相应,当一个 I/O 响应到达时,前端驱动通过超调用从环形缓冲中读取 I/O 响应,进行一些清理工作,通知客户机操作系统 I/O 的完成。

前端驱动与后端驱动间通过事件通道机制进行异步通信。前端驱动初始化事件通道,后端驱动将它与这个事件通道绑定。当发送一个 I/O 请求时,前端驱动通过事件通道通知后端 I/O 请求的到达;当 I/O 响应到达前端时,前端驱动会收到一个事件来告知 I/O 响应已到达,进而从环形缓冲中获取 I/O 响应的描述符。

前端驱动通过授权表机制(Grant Table,在 6.3.6 节具体介绍)将 I/O 数据共享给后端,以实现数据传输。前端驱动分配共享页保存 I/O 数据,通过授权表允许后端映射和访问这个 I/O 数据页。通过页共享,后端驱动被允许通过 DMA 直接读写属于虚拟机的 I/O 数据页,这样就节省了 I/O 数据复制所带来的额外的性能开销。

6.2.7 时间与时钟服务

类虚拟化系统的虚拟时间根据不同的时间服务精度的需求,其设计可以是各种各样的。

Xen 通过时间虚拟化保证客户机中各个任务仍能像在非虚拟化环境中一样得到公平的调度,获得相同比例的 CPU 执行时间资源。即客户机中每个进程所分配到的运行时间资源的比例,不受下层 Xen 调度各个客户机行为的影响(当然,虚拟化所带来的额外性能开销会影响各个进程执行的绝对时间)。

为此,Xen 维护了三种时间:实际时间(Real Time)、虚拟时间(Virtual Time)和墙钟时间(Wall Clock Time)。实际时间以纳秒为单位,从计算机启动即开始计时。虚拟时间是每个客户机实际占用 CPU 资源执行所消耗的时间,当客户机不在 CPU 上运行时,虚拟时间的流逝也相应地暂停,直到客户机下次被调度到继续运行。墙钟时间是为每个客户机单独维护的逻辑上消耗的时间,它与真实的时间流逝同步。当一个客户机不在 CPU 上运行时,这个客户机的挂钟时间依然在流逝,并始终与现实世界时间保持同步。

计算虚拟时间的方法显而易见。当客户机被 VMM 调度运行时,它接收来自 VMM 的时间中断,相应地更新虚拟时间。客户机参考虚拟时间来调度各个进程。客户机无须关心真实世界的时间流逝,而只需关心客户机内部虚拟世界的时间流逝。这样,保证了每个进程得到固定长短的虚拟时间,也就保证了每个进程得到固定比例的真实 CPU 执行时间。

为了计算挂钟时间,VMM 从 CPU 的 TSC 计数器(Time-Stamp Counter)换算出从开机起经过的时间。将这个值与初始系统时间相加,即可得到当前系统挂钟时间。TSC 计数器包含在 x86 芯片中,自开机起随每个 CPU 指令时间片单调增加。根据它和 CPU 主频的相对关系,可以换算出自开机器所经过的时间。当每次客户机被调度运行时,VMM 会用以上方法计算一次这个客户机的墙钟时间,保存在与客户机的共享页中,供客户机读取,作为客户机的系统时间。由于挂钟时间计入了客户机不在运行时流逝的时间,因此虚拟化环境下这个客户机的系统时间和非虚拟环境下的系统时间一样,是和真实世界的时间保持同步流逝的客户机系统时间。

Xen 还为客户机操作系统提供闹钟服务。客户机操作系统能够配置一对闹钟,一个是按实际时间计时,一个是按虚拟时间。客户机操作系统还需要自己维护内部的闹钟队列。

6.3 Xen 的原理与实现

6.3.1 超调用

超调用是从客户机操作系统到 VMM 的系统调用。与系统调用类似,Xen 启用 130 号中断向量端口(十六进制的 82H)作为超调用的中断号。这一个中断向量的 DPL 被设置为 1,类型为中断门。这样,超调用能够由处于特权级 1 的客户机操作系统发起,而不能从用户

态发起。

超调用页在虚拟机启动前被初始化，在虚拟机内核启动时，这个页被映射在客户机操作系统的一个固定的虚拟地址上。一个超调用页被划分成 128 块，每块长度为 32 字节。在初始化时，每一块都会写入一段发起超调用的汇编代码。第 i 块的汇编代码会将编号 i 作为参数来发起超调用。在需要发起超调用时，客户机操作系统将超调用页中每块汇编片段当作一个函数，需要调用第 i 号超调用就调用第 i 块代码片段。

6.3.2 虚拟机与 Xen 的信息共享

在客户机操作系统启动时，Xen 会提供一个启动信息页（start_info）给操作系统，提供启动时需要的多种参数。

在运行时，客户机操作系统与 Xen 共享一个信息页，称为共享信息页（shared_info）。

1. 启动信息页

启动信息页中包括的主要信息有内存页的总数、共享信息页的地址、Xenstore 的地址和事件通道号、控制台信息页的地址和事件通道号、页表基址、P2M 表地址等。

上述这些信息在启动时是必需的。由于这些信息体积比较大，并且在启动时还没有初始化 IDT 表前就会被用到，所以不能通过超调用来获取。

启动信息页在虚拟机启动时由 Xen 填写，被虚拟机读取，在启动过程结束后不再使用。

2. 共享信息页

出于效率的考虑，Xen 将一些常用的共享信息放在共享信息页上，方便与客户机操作的通信。在共享信息页上的主要信息包括 VCPU 相关信息、事件通道相关数据结构、时间以及一些体系结构相关的信息。

在运行时，共享信息页能够被 Xen 和客户机操作系统修改。其中，一些字段由 Xen 更新，例如当前时间等；一些数据由操作系统更新，例如事件通道的事情屏蔽位（Event Mask Bits）等；一些字段由操作系统和 Xen 一起更新，如事件通道的未决事件位（Event Pending Bits）等。

6.3.3 内存管理

在虚拟环境下，VMM 负责对所有的物理页进行管理，负责为各个客户机分配和回收内存，同时保证对于分页和段相关硬件的安全使用。除了 VMM 自己保留的内存以外，其他物理内存都可以以页为粒度进行分配。当前，大部分操作系统都假设所占据的内存是连续的，但是在虚拟环境下，这一假设就不成立了，VMM 需要向客户机操作系统提供一种连续物理内存的假象。

1. 伪物理内存

现代操作系统都支持保护模式，在该模式下，每个应用程序都有独立的地址空间。在应用程序看来，它们对整个内存享有独占权，而不必关心实际分配给它们的是哪些物理内存。

在虚拟环境下,VMM 需要为客户提供操作系统做类似的事情。大部分操作系统默认看到的是一块连续的,从地址 0 开始的物理内存。为了实现这类操作系统的虚拟化,就需要提出伪物理内存的概念,以区别于实际的物理内存,即机器内存。机器内存是指物理主机上所安装的所有内存,包括被 VMM 和虚拟域所使用的以及尚未分配的。可以认为,机器内存是由一系列连续的,从地址 0 开始的 4KB 大小的物理页组成的。对 VMM 和客户机操作系统而言,机器页号是相同的。伪物理内存是相对于每个物理域的抽象,它允许客户机操作系统把自己所分配到的内存看作是一系列从地址 0 开始的连续的物理页,而不必担心实际的物理页号是不是连续的问题。

要实现这一抽象,VMM 需要维护一张全局的可读的 Machine-to-Physical 映射表,记录从机器页到伪物理页的映射。同时,需要向每个虚拟域提供一张 Physical-to-Machine 的映射表,来完成相反的映射。很明显的是,Machine-to-Physical 映射表的大小与物理机上实际安装的 RAM 相关,而 Physical-to-Machine 映射表的大小则与分配给相应域的内存大小有关。客户机操作系统中的体系结构相关代码,就可以利用这两张表实现伪物理内存的抽象。概括来讲,只有客户机操作系统的一部分(如页表的管理)需要知道机器地址和伪物理地址的区别。在大部分情况下,操作系统内核并不需要知道物理页的信息,实际需要时则可以通过查询相关映射表来获取信息。

2. 物理页信息的维护

VMM 对于物理页信息的维护主要包括引用计数、所有者(虚拟域号)和页类型。对每个物理页的所有者和用途进行跟踪,使得 VMM 能够对不同客户机进行安全隔离。

在虚拟环境下,物理页按用途可以分为可读写页、页表页、页目录页和描述符表页。这些页的类型是互斥的。也就是说,一张物理页不可能同时为两种类型。对于每种类型,VMM 都维护了一个独立的引用计数,在对页类型进行修改时,首先要求当前页类型的引用计数为 0。

3. 页分配及回收

通过内存自伸缩调节驱动,客户机操作系统可以放弃或者申请额外的内存。在内存使用量较小时,行为良好的客户机会把大量闲置内存返还给 VMM。如果客户机发现自己使用内存过量,它可能尝试把缓冲区数据换出,并且释放一些内存;反之,如果客户机发现还可以再申请一些内存时,它可能尝试扩大自己的块设备缓冲区。所有的这些操作都必须通过显式调用超调用来完成。

6.3.4 页表虚拟化

VMM 往往支持两种模式的内存虚拟化:直接模式和影子模式。这里主要介绍直接模式下的页表虚拟化。

1. 地址翻译

在直接模式下,客户机操作系统中的页表记录的是从虚拟地址到机器地址的映射(页表

项存放的是机器页号),并被直接装载到 MMU 中运行。所以,该模式下的地址翻译与非虚拟环境下没有大的差别。但是,为了保证安全性和隔离性,VMM 必须保证客户机操作系统只能映射给它分配的那些页,并且保证客户机操作系统不能创建到页表页的可写映射(否则前一条规则就被破坏了)。

2. 页表更新

VMM 为每个物理页维护了类型信息。描述表、页目录和页表都有对应的页类型。在客户机操作系统的页表映射中,这几种类型的页必须被映射成只读的。这保证了客户机操作系统不能随意修改自己的页表,从而保证了一个虚拟域不能随意访问其他域的内存。在不能直接修改页表的情况下,客户机操作系统可以通过调用相关超调用来更新页表。在进行实际更新之前,VMM 会对更新的请求进行验证,任何试图映射所分配内存以外物理页的更新都会被禁止。进行验证时,VMM 所维护的页类型和引用计数就起作用了。例如,客户机操作系统不能创建到页表页的可写映射,因为这要求相关页既为页表页,又同时为可读写页,违反了页类型的互斥原则。相对普通的内存访问,超调用是开销比较大的操作,所以在一次超调用中可以进行多次页表的更新。客户机操作系统可以更新普通页表,也可以更新 Machine-to-Physical 映射表。当然,这些更新都必须首先通过 VMM 的验证。

3. 缺页异常的处理机制

客户操作启动时,通过超调用的方式向 VMM 注册一系列陷阱处理函数,这其中就包括缺页异常的处理函数。同时,VMM 允许客户机操作系统装载自己的 IDT。发生缺页异常时,VMM 会产生一个陷阱,并且调用自己的缺页异常处理函数。和正常情况下处理缺页异常一样,VMM 首先根据 CR2 寄存器内容确定导致发生缺页异常的地址。因为缺页异常处理函数本身也会访问内存,可能导致额外的缺页异常。这种情况一旦发生,就会导致 CR2 寄存器的内容被覆盖,所以 VMM 首先需要把 CR2 寄存器的内容复制到一个安全的地方。缺页异常的地址被确定之后,VMM 进一步确定该地址的范围,如果发现是在 VMM 的地址空间范围之内,就做相应的处理;如果该地址在客户机操作系统的地址范围之内,VMM 就在做完相关准备工作之后把缺页异常传递给客户机操作系统。VMM 做的准备工作主要是在客户机操作系统的栈上(1-特权级的栈)创建发生缺页异常的栈帧,客户机操作系统发生缺页异常时的现场也被做相应恢复。客户操作通过调用自己的缺页异常处理函数来完成对缺页异常的处理。页表的更新就是在此次通过超调用的方式完成的。

可写页表(Writable Page Table)提供一种自动批量更新页表的方式。在发生缺页异常而需要对页表进行更新时,VMM 首先对该页表页的内容做一份备份,再清空上一层页表指向该页表项的存在位,然后把该页表页标记为可读写页,最后再把该缺页异常传递给客户机操作系统处理。由于要修改的页表页已经被修改为可读写页,客户机操作系统可以直接对其进行修改,不需要调用超调用。当下一次地址翻译需要用到该页表页时,VMM 首先根据之前保存的页表备份确定客户机操作系统对该页表页所作的所有修改,并进行验证。在所有的更新都验证通过后,VMM 重新把该页表页挂载到上一层页表上去。为了找到对应该

页表页的上一层页表的表项, VMM 需要对所有的页表页维护一个反向影射。

4. 模拟对客户机页表的修改

在 1.4.4 节中已经提到, VMM 写保护客户机的页表页。因此, 客户机操作系统无法直接修改其页表页, 客户机操作系统可以通过显式调用超调用来更新页表。并且为了降低每次超调用带来的性能开销, 一个超调用可以以批处理的方式处理多个修改页表页的操作。

然而, 在应用程序进程的创建和销毁等情况下, 页表页被频繁地修改。除了批处理的超调用之外, VMM 还提供了模拟对客户机页表的修改的方式, 来高效地处理大批量的修改客户机页表页的操作。

由于 VMM 写保护了客户机的页表页, 当客户机操作系统试图以修改普通数据页的指令来写入页表页时, 发生缺少写页表权限的缺页异常。VMM 截获这个异常, 代替客户机操作系统进行修改页表的操作。这样既能通过写保护截获对客户机页表的修改, 防止客户机操作系统随意修改客户机页表, 又在逻辑上模拟了修改页表页的操作。

具体地说, 若客户机操作系统试图直接写入被 VMM 写保护的页表页而发生缺页异常, VMM 首先根据 EIP 得到引起异常的修改页表页的二进制指令的地址。VMM 解析这条二进制指令, 通过与预先存储的指令类型表比对, 解析出指令的类型、操作数的类型和操作数的值, 从而分析出需要写入的新的页表项的值。同时, CR2 里保存了发生缺页异常的地址, 就是被修改的页表项的地址。

有了需要被修改的页表项的地址和新的页表项的值, VMM 就能代替客户机操作系统, 修改被写保护的页表项。根据页表项修改前后的值, VMM 施行和通过显式调用超调用修改页表页一样的相关验证, 保证宿主机物理页类型的正确性、不同虚拟机间宿主机物理内存的隔离性等。通过了相关验证后, VMM 建立映射客户机页表页的临时的可写映射, 通过这个映射将新的页表项的值写入, 然后取消这个临时映射。至此, 完成了模拟客户机操作系统修改其页表页的过程。

6.3.5 事件通道

在 Xen 中, 事件是包括中断在内的异步消息触发的抽象。Xen 中的中断主要包括硬件中断和虚拟中断。前者是物理硬件触发的中断, 例如网络包的到达。后者是 VMM 为客户提供创造的, 模拟的客户机虚拟硬件设备的中断。例如, 对应于客户机特定虚拟时间的时间中断。

在 VMM 和每个客户机之间, 有一张双方都可以访问的页用于数据共享。VMM 在这张共享页中维护了一个事件向量。此向量的每一位都对应一个事件。当一个事件发生时, VMM 就将向量中对应这一事件的位置上。基于事件向量, Xen 实现了事件通道(Event Channel)这一抽象, 作为 VMM 通知客户机某一事件发生的通道。VMM 可以为一个事件分配和绑定一个特定的事件通道, 在事件发生时通过这一通道通知通道另一端的客户机。

中断处理是 Xen 事件通道机制的一种应用。硬件的 IDT 表中转载的中断处理函数通

常是 Xen 的特殊处理函数。当中断发生时,硬件根据 IDT 表调用相应的由 Xen 注册的中断处理函数。Xen 的中断处理函数通过事件通道将中断事件通知给相应的客户机的中断函数处理。对每一个需要处理这个中断的客户机,Xen 的中断处理函数创建一个事件通道,并将这个事件通道在事件向量中对应的未决位(Pending)置上,从而实现了异步通知客户机这个中断的发生。如果需要处理中断的客户机正在执行,Xen 立即打断它的正常控制流以处理中断;否则,Xen 为客户提供一次调度,客户机被调度执行时,Xen 通过 upcall 进入客户机的控制流,调用客户机相应的中断处理函数正确地处理这个中断。在这里,客户机的中断处理函数通常是客户机启动时通过一个超调用向 Xen 注册的。事件通道的端点通过端口号唯一标示。

除了处理中断,Xen 还利用事件通道进行跨客户机通信。所不同的是,这时触发事件的不是 Xen,而是其中一个客户机;并且这种事件通道是双向的。通道任何一端的客户机都可以通过它通知另一端的客户机事件的发生。

6.3.6 授权表

客户机之间的大数据块传输通过共享内存的方式实现。VMM 为客户提供了一套共享内存的接口。一个客户机可以申明它所拥有的某些内存页可以被其他客户机共享;另一个客户机可以将这些内存页映射到自己的地址空间中。这样,两个客户机都能够访问这些共享的内存页,从而实现数据交换。

每个虚拟机都有自己的授权表,来控制其他虚拟机访问这个虚拟机所拥有的共享页的权限。授权表的每个表项定义了对当前虚拟机的某一内存页,其他的某个客户机具有哪些访问权限(读、写)。通过 VMM 提供的接口,客户机操作系统初始化它的授权表,在客户机操作系统的内核空间中;客户机操作系统修改授权表,就能定义其他客户机对自己所拥有的内存页的访问权限。

在客户机操作系统修改授权表以创建新的访问权限时,会产生新的授权引用(Grant Reference)。授权引用是用来索引授权表的整数下标。其他的客户机可以通过授权引用指定要访问的当前授权表所在的客户机中的物理页。若客户机通过 VMM 提供的接口试图映射授权引用所指向的其他客户机拥有的某个物理页到自己的地址空间中时,VMM 通过授权引用在拥有共享页的客户机的授权表中找到相应的表项。根据这个表项确认拥有共享页的客户机允许将这个页映射给申请共享的客户机。只有通过了权限检查,这个共享页才能被顺利地映射到申请共享的客户机的地址空间中。

6.3.7 I/O 系统

1. 设备驱动划分

设备驱动用于操作系统内核与硬件设备之间的交互。正如前文提到的,在虚拟化环境中,VMM 为客户提供虚拟出来的虚拟设备,作为硬件资源的抽象。在完全软件

虚拟化技术中,VMM 用纯软件方式来模拟虚拟设备的行为。操作系统使用本地环境下的设备驱动程序,无须经过修改,就能在虚拟环境下驱动这些虚拟设备。VMM 的设备模型负责接收来自客户机操作系统的请求,调用真正的设备驱动与真实的硬件设备交互,完成 I/O 操作。

模拟设备的方式虽然可以直接使用本地环境下的设备驱动程序,但是不能够满足效率上的需求。客户机操作系统的驱动程序不能直接与硬件设备交互。在类虚拟化技术中,通过允许修改客户机操作系统的驱动程序,使得设备驱动可以通过 VMM 提供的接口直接与硬件设备交互。

Xen 使用了前端模式实现类虚拟化中的外设虚拟化。Xen 的特权虚拟机 Dom0 负责与真实硬件设备的直接交互,它运行了为 Dom0 扮演代理的后端驱动和直接与真实硬件设备交互的原生设备驱动;后端驱动接收来自 DomU 的请求,调用原生设备驱动访问硬件处理这个请求。每个非特权虚拟机 DomU 中运行的是为 DomU 创建虚拟硬件设备的前端驱动;通过事件通道和环缓冲,它将客户机操作系统的设备访问请求传递给后端驱动,并处理来自后端驱动的反馈。前端驱动和后端驱动之间是一一对应的关系。共享同一个真实硬件的多个 DomU 会拥有独立的前端驱动、后端驱动实例对,维护各自的状态信息。前端驱动和后端驱动间通过事件通道进行异步消息传递;通过基于授权机制的共享内存传递数据;通过环缓冲发出和接收 I/O 请求和反馈的描述符。前端驱动和后端驱动本身由客户机操作系统实现,运行在 DomU 和 Dom0 的客户机操作系统中,所以并不是 VMM 的一部分,只是利用 VMM 提供的机制进行相互间的通信。通常,由前端负责分配用于大块数据传输的共享页并通过授权表允许 Dom0 访问;并在共享页中初始化请求/反馈描述符队列(即下文中的环形队列)和其他私有数据结构;创建事件通道与后端连接。

在默认情况下,只有 Dom0 直接与硬件交互。然而,也可以用专门的外设虚拟机(Driver Domain)驱动某些特定的设备。使用外设虚拟机的好处有两个:容错性和兼容性。驱动程序是操作系统中出错率最高的部分,一旦它们出错,会导致整个操作系统崩溃。将驱动程序放到一个特殊的外设虚拟机中,就可以避免驱动程序出错时操作系统崩溃的情况,只需要重新启动外设虚拟机就能使对应的硬件重新恢复工作。这样就提高了容错性。在兼容性方面,在外设虚拟机中运行的特定设备的本地设备驱动所依赖的客户机操作系统不必与 Dom0 一致,可以为了这种设备按需安装和配置特定的操作系统。不同 DomU 的客户机操作系统内的前端驱动实现可以对应同一种后端驱动。

2. 控制流与数据流

正如前面章节中提到的,在 Xen 中,非特权虚拟机和特权虚拟机之间通过事件通道机制来通知 I/O 请求或反馈的到达。

I/O 请求和反馈本身以描述符的方式,保存在使用生产—消费者模型存取的环形缓冲中。每一对前端驱动和后端驱动之间共享一个消费者生产者队列实例。如图 6-2 所示,非特权虚拟机的前端驱动调用 Xen 提供的接口将 I/O 请求描述符保存在队列中,更新指向请

求队列尾部的请求生产者(Request Producer)指针；与之对应的特权虚拟机或外设虚拟机中的后端驱动读取第一个 I/O 请求，更新指向请求队列头部的请求消费者(Request Consumer)指针。后端驱动调用本地驱动处理这个请求。当 I/O 请求处理完成后，后端驱动将 I/O 反馈写入到队列中对应的原先 I/O 请求的位置，更新指向 I/O 反馈队列头部的反馈生产者(Response Producer)指针。队列中的每个元素实际上是 I/O 请求描述符和反馈描述符的联合，因此可以保存两种描述符中的任意一种。当客户机操作系统在执行时，其中的前端驱动读取消费者生产者队列中的第一个 I/O 反馈，处理它，更新指向 I/O 反馈队列尾部的反馈消费者(Response Consumer)指针。生产消费者队列由前端驱动初始化，通过授权机制共享给后端驱动。I/O 请求、反馈描述符的具体结构与具体的设备驱动种类相关。

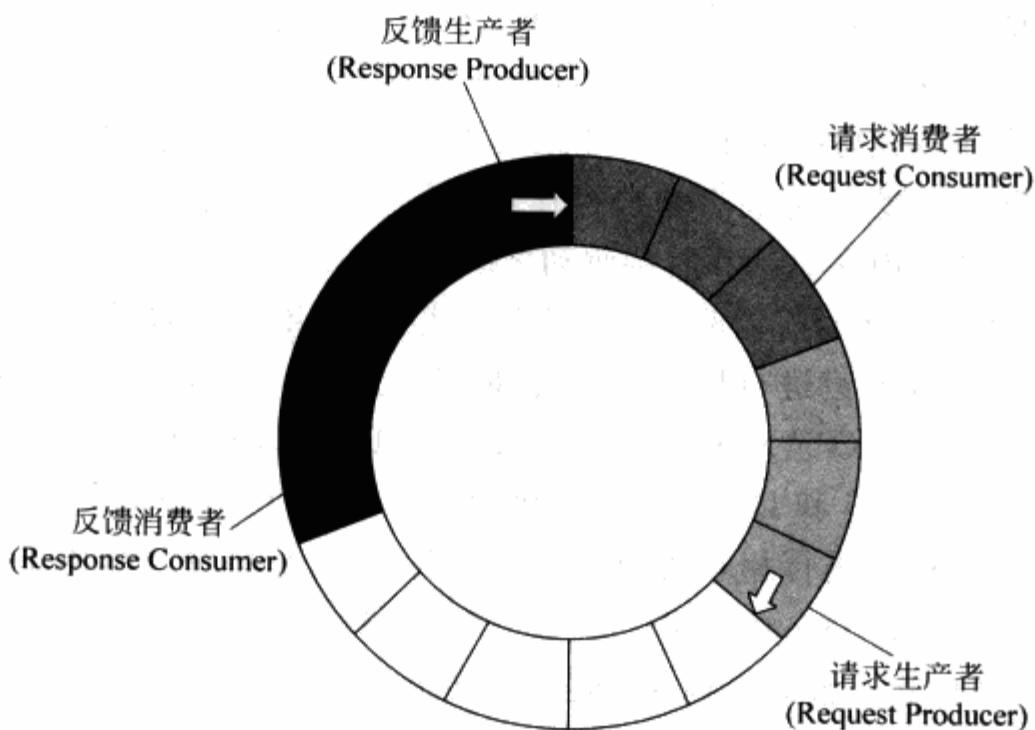


图 6-2 输入输出描述符环

3. 零拷贝技术

在高性能计算中，需要将通信延迟减至最低。通信延迟，除了网络传输延迟外，主要是软件消息处理机制造成的延迟。在传统的操作系统中，消息处理机制包括多次数据复制和变换，造成了时间性能上的开销。例如，在接收网络包时，系统需要将从网卡读取的网络包复制到内核缓存中，再将网络包从内核缓存复制到用户态应用程序的缓存中。这两次复制带来了一定的性能损失。

为了避免不必要的复制，零拷贝技术(Zero-Copy)应运而生。零拷贝技术的设计目的，就是为了避免 I/O 过程中发生任何的数据复制，使得用户空间的数据和外部设备之间能进行直接的数据交互，从而提升性能。

零拷贝技术的实现依赖于 DMA 技术和内存映射技术。下面以网络收包为例加以说明，发包的情况是与之类似的。首先，零拷贝技术运用 DMA 来实现把网络数据包从网卡直接复制到内核缓存中。由于使用了 DMA 技术，这一步完全不需要 CPU 的参与，因此有效

地节省了 CPU 资源。接着,包含网络包数据的内核缓存被重新映射到了用户空间中。这样,用户应用程序就可以直接访问保存在内核缓存中的网络包数据。这不仅避免了从内核空间向用户空间复制数据,也节省了用户应用程序通过系统调用获取内核缓存数据所带来的运行级别切换的开销。

在虚拟化技术中应用零拷贝技术,面临着新的难题。除了虚拟网卡与客户机内核空间、客户机内核空间与用户空间之间的数据复制需要优化之外,虚拟网卡与物理网卡之间存在的网络包的复制也应该避免。通过使用授权表机制,需要保存网络包数据的虚拟网卡缓存所对应的客户机物理页,通过 VMM 修改虚实地址映射表的方式,被直接映射到保存物理网卡所接收到的网络包的宿主机物理页上。这样,就避免了从物理网卡缓存的宿主机物理页复制网络包数据到虚拟网卡缓存的宿主机物理页。客户机可以直接、透明地通过虚拟网卡缓存的客户机物理页映射,访问保存了网络包数据的物理网卡缓存的宿主机物理页。

6.3.8 实例分析: 块设备虚拟化

原生的虚拟磁盘驱动为上层的操作系统提供了抽象的虚拟设备接口,读写一个磁盘块;将读写请求翻译成符合硬件手册规定的操作下层硬件设备的对应请求,发送给磁盘。Xen 的块设备虚拟化(Virtual Block Device, VBD)实现了基于前端驱动模型的虚拟磁盘。前端驱动为客户机操作系统提供虚拟磁盘设备抽象,后端驱动调用原生磁盘驱动与磁盘交互,相互间利用事件通道、共享内存进行异步通信和数据传输。

在初始化时,前端分配共享页通过授权表供后端共享、初始化基于共享页的环形缓冲和私有数据结构、创建事件通道;与之相对地,后端映射共享页,将自己与前端的事件通道绑定。

在这里,概述 VBD 处理一个块设备请求的流程。当接收到请求时,前端驱动将请求描述符加入到和 Dom0 共享环形缓冲中;通过事件通道通知后端驱动请求的到达;将 I/O 读写数据写入到共享页中,并通过授权表机制允许 Dom0 来读取这个页。

在这里,请求描述符中包含了操作类型(读或写)、段的数量、读写句柄、ID、第一个数据块相对于磁盘的块号以及一个段数组。段数组中的每个元素都包含了指向 I/O 数据页的授权引用,以及这个 I/O 数据段在数据页中的起始和结束位置。值得注意的是,这里逻辑上由连续块组成的段,在磁盘上的实际排列仍然可能是不连续的。物理磁盘的原生驱动本身提供了这一层包含逻辑上连续块的段抽象。

当后端驱动被调度执行(或者正在执行,被 VMM 打断后),从环形缓冲中读取请求描述符;将包含 I/O 数据的共享页在 DomU 的授权下映射到自己的地址空间中。Dom0 通过授权机制直接访问 DomU 的数据页,节省了额外的数据拷贝的开销,使得 Dom0 可以直接通过 DMA 读写数据页中的 I/O 数据到磁盘中,从而提高了性能。

根据请求描述符和 I/O 数据,后端请求生成一个原生磁盘访问请求,调用原生磁盘驱动访问磁盘。在这里,后端可能合并来自前端的多个访问相邻数据块的请求。原生磁盘驱

动处理完 I/O 请求后,后端驱动生成反馈描述符,保存在环形缓冲中,并通过事件通道通知前端驱动反馈的到达。

前端驱动阻塞式等待,直到收到反馈,完成一些收尾工作,通知客户机操作系统 I/O 完成。为了提高磁盘的吞吐量,VBD 允许乱序处理来自不同虚拟机的 I/O 请求。因此,反馈抵达客户机操作系统的顺序也可能是乱序的。这时,客户机操作系统通过标示请求—反馈配对的唯一标示符(ID)来查找与这个反馈对应的请求。

6.4 XenLinux 的运行

DomainU 虚拟机的创建启动都是由 Dom0 负责。由 Dom0 代替 Xen VMM 进行这项工作可以减少 Xen VMM 的实现复杂度,增加整个系统的健壮性。

那么,首先就需要 Dom0 能够通过一套适当的机制把 DomU 引导起来。尤其是如何能够使 Dom0 读到 DomU 的内核镜像以及 ramdisk 文件,使得在新的虚拟机被创建后,控制流能够顺利地转移到该虚拟机内核入口点。

从 Dom0 文件系统对 DomU 文件系统的访问通常会有很大的困难,而在如今 Xen 的实现中已经能够支持启动加载程序来代劳完成这些工作,它的作用和工作流程就和普通 Linux 启动加载程序工作的机制类似。启动加载程序运行在用户空间,在目前的 Xen 视线中这部分组件通过一个 python 的脚本来完成,它把启动一个新的虚拟机需要的内核以及 ramdisk 文件从它的文件系统中复制出来并存放于 Dom0 的一个临时文件夹中。完成这一步骤后,由虚拟机构建器(Domain Builder)来完成新虚拟机的创建。在 Xen 的实现中,启动加载程序同时会配有一个用来模拟 GRUB 系统的脚本工具,它就如同 Linux 中的 GROB 菜单。进入这个模拟 GRUB 菜单之后,用户可以看到一个或多个启动选项,当用户选择了其中一个条目后,GRUB 系统会找到相应的内核等文件并复制出来交给虚拟机构建器。

对每个虚拟机内存的初始化分配都是在该机创建的时候完成的,各个不同的 DomU 之间的内存分配都是静态完成的,而不是动态调整,这一措施很好地保证了不同虚拟机之间内存空间的隔离,有效地防止了一些非法的跨域的内存访问。而一个虚拟机对内存大小的需求可能会变大,在这个时刻该域可以从 Xen VMM 处请求更多的内存空间。当然,每个虚拟机都有它所能要求的内存空间的最大值,这个值在初始化的时候就会被设定。

至于磁盘设备的分配,对于真实的设备只有 Dom0 才有权访问,Xen 也提供了一套有效的分离设备驱动的机制来保证其他非特权虚拟机对设备的访问。因此,所有的 DomU 将在初始化的时候根据其配置文件由 Dom0 来分配管理属于它的那些虚拟块设备。

同样,根据启动一个新的虚拟机的配置文件,Dom0 也管理分配 DomU 对网络资源的访问。每个分配给 DomU 的虚拟网络接口都会在 Dom0 有对应的网络防火墙路由器(Firewall-Router),一个虚拟的网络接口就模拟了一个真实的网络适配器。

6.5 思考题

- (1) 在 Xen 系统中,假设平台上只运行了一个使用类虚拟化技术的 DomU 客户机。在 DomU 运行 CPU 密集型的应用程序时,其可能的性能损失来自于哪里?
- (2) Xen 的 I/O 虚拟化技术中,I/O 环相对于 I/O 指令提高了一个抽象层。I/O 环相对于 I/O 指令级的模拟优势在哪里?是否在更高的抽象层上能够找到效率更好的 I/O 虚拟化方法?
- (3) 在 x86 平台上加入了硬件对虚拟化的支持后,Xen 或其他的类虚拟化技术是否还有存在的必要?
- (4) Paravirt_ops 和 VMI 对于 Linux 的开发具有什么帮助与阻碍?
- (5) 如果要开发一个通用操作系统专门用于虚拟化环境,现有的类虚拟化能够做哪些深度优化?操作系统与 VMM 的功能分界线可以做哪些调整?