

分类号 TP311

UDC 004

学校代码 10590

密 级 公开

# 深圳大学硕士学位论文

## KVM 虚拟机的性能研究与改进

奉 华

学 位 类 别 工程硕士专业学位

专 业 名 称 软件工程


学 院（系、所） 计算机与软件学院

指 导 教 师 梁正平

## 深圳大学学位论文原创性声明和使用授权说明

### 原创性声明

本人郑重声明：所呈交的学位论文 KVM 虚拟机的性能研究与改进 是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不含任何其他个人或集体已经发表或撰写过的作品或成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本声明的法律结果由本人承担。


论文作者签名：   
日期：2018 年 11 月 28 日

### 学位论文使用授权说明

（必须装订在印刷本首页）

本学位论文作者完全了解深圳大学关于收集、保存、使用学位论文的规定，即：研究生在校攻读学位期间论文工作的知识产权单位属深圳大学。学校有权保留学位论文并向国家主管部门或其他机构送交论文的电子版和纸质版，允许论文被查阅和借阅。本人授权深圳大学可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（涉密学位论文在解密后适用本授权书）

论文作者签名：   
日期：2018 年 11 月 28 日

导师签名：   
日期：2018 年 11 月 28 日

## 摘 要

随着大规模集成电路技术的迅猛发展, 计算能力将不再成为业务应用的瓶颈, 业务独占物理资源成为了一种浪费。如何利用过剩的计算资源, 同时提供安全、稳定、快速部署、按需分配的计算资源服务, 是当前计算机技术领域研究的主流课题, 虚拟化技术使这些特性成为现实, 而虚拟机性能表现则是用户选择服务的关键指标。

本文以 KVM 虚拟机的性能研究与改进为课题, 首先介绍了 KVM 的起源, 发展背景, 构建最新版本的 KVM 虚拟化平台, 并完成虚拟化环境优化。然后, 从 KVM 虚拟化原理、虚拟机调度、内存使用和 I/O 访问优化等方面深入研究虚拟机的性能优化方案和实现方法。围绕 KVM 虚拟机性能改进, 研究的具体工作如下:

(1) 采用开源 Linux 作为 VMM, 在具体的 Centos7 系统上编译最新内核, 并修改和调整内核中影响 KVM 性能的功能模块, 安装最新版本 I/O 模拟软件 QEMU 和虚拟机管理软件 libvirt, 构建了经过环境优化的 KVM 虚拟化平台。

(2) 研究了 KVM 硬件辅助完全虚拟化 (Intel VT-x 技术) 的实现原理和深入分析 KVM 内核模块的软件驱动原理和运行机制, 在深入理解虚拟机运行模式的基础上, 提出基于处理器角度的 KVM 虚拟机性能优化思路, 设计和实现了一套 VCPU 陷入事件的捕获方案, 针对第三种思路, 验证了改进的可行性。

(3) 深入研究 Linux 进程管理、调度策略、运行优先级对进程占用处理器资源的影响, 提出了 KVM 虚拟机调度优化方案, 并设计和实现了一个进程监控程序, 结合实时监控, 可针对业务特点灵活改变虚拟机的调度策略和运行优先级, 使虚拟机能够按实际需求获得更多占用物理处理器资源的机会, 改进 KVM 虚拟机的整体性能。实验验证, 该优化方案具有可行性。

(4) 从优化内存的使用率和访问效率两方面, 研究优化内存使用的三种主要技术, 根据物理资源环境和业务系统需求提出内存优化方案和适应的场景, 并给出了具体的实现方法和步骤。

(5) 研究以 virtio 为主的半虚拟化设备访问原理和实现过程, 确定半虚拟化是 KVM 虚拟机 I/O 访问性能改进的主要方案。提出了硬件直通模型中 I/O 虚

拟化技术的应用场景和以具体业务相结合实施 I/O 优化的思想。通过实验，给出了 virtio、Intel VT-d 和 SR-IOV 标准下虚拟机 I/O 性能优化的实现方法和步骤。

**关键词：**KVM；Intel VT-x；调度策略；性能优化；Virtio

# Abstract

With the rapid development of large-scale integrated circuit technology, computing power will no longer be the bottleneck of business applications, meanwhile, the exclusive occupation of physical resources becomes a waste. The mainstream topic in current computing research is how to use excess computing resources when providing secure, stable, fast deployment, and on-demand computing resource services. Virtualization technology is the key to realize those features and the virtual machine performance is the key parameter in customer service.

This paper will study the properties and optimizations of KVM virtual machines in the following steps: Firstly, we will present the origination and development background of KVM, with the virtualization environment optimized. Then, from the aspects of KVM virtualization principle, virtual machine scheduling, memory usage and I/O access optimization, the performance optimization scheme and implementation method of virtual machine are comprehensively studied. Around the KVM virtual machine performance improvement, the specific work of the research is as follows:

(1) Using open source Linux as VMM, compiling the latest kernel on a Centos7 system, and modifying/adjusting the function modules in the kernel that affect KVM performance, installing the latest version of I/O simulation software QEMU and virtual machine management software libvirt, and building an environment-optimized KVM virtualization platform.

(2) The realization principle of KVM hardware-assisted full virtualization (Intel VT-x technology) and the in-depth analysis of the software-driven principle and operation mechanism of KVM kernel module are studied. Based on the in-depth understanding of the virtual machine operation mode, a processor-based algorithm is proposed. From the perspective of KVM virtual machine performance optimization, a set of VCPU capture event capture schemes was designed and implemented. For the third approach, the feasibility of the improvement was verified.

(3) In this work, we also perform in-depth study of the impact of Linux process management, scheduling strategy, and operational priority on processor resource consumption, proposed a KVM virtual machine scheduling optimization solution, designed and implemented a process monitoring program combined with real-time monitoring, which enables priority change regarding the properties of computing needs, therefore, the virtual machine can get more opportunities to occupy physical processor resources according to actual needs, and improve the overall performance of the KVM virtual machine. The optimization scheme is verified by experiment.

(4) From the aspects of optimizing memory usage and access efficiency, three approaches for optimizing memory usage were discussed, depending on the adaptation scenarios according to physical resource environment and business system requirements, detailed implementation methods are proposed.

(5) The access principle and implementation process of paravirtualized devices based on virtio are investigated, and the improvements on accessing performance of KVM virtual machine I/O using paravirtualization is discussed and determined. In addition, we propose the application scenario of I/O virtualization technology in hardware pass-through model and the idea of implementing I/O optimization with specific services. Using experiments, the implementation methods and steps of virtual machine I/O performance optimization under virtio, Intel VT-d and SR-IOV standards are presented.

**Key words:** KVM; Intel VT-x; scheduling strategy; performance optimization; Virtio

# 目 录

摘 要.....	I
Abstract.....	III
第 1 章 绪论.....	1
1.1 研究背景.....	1
1.2 KVM 虚拟化的意义 .....	2
1.3 虚拟化发展现状.....	2
1.4 研究内容与论文结构.....	5
1.4.1 研究内容 .....	5
1.4.2 论文结构 .....	6
1.4.3 主要创新点 .....	7
第 2 章 KVM 虚拟平台的构建.....	9
2.1 KVM 虚拟化架构 .....	9
2.2 KVM 性能优化技术 .....	12
2.3 KVM 虚拟平台构建 .....	13
2.3.1 更换 Centos7 内核 .....	13
2.3.2 编译和安装 QEMU.....	15
2.3.3 安装与配置 Libvirt.....	16
2.3.4 创建虚拟机 .....	17
2.4 本章小结.....	20
第 3 章 KVM 虚拟化实现原理与机制.....	21

3.1 Intel VT-x 辅助 CPU 虚拟化实现技术 .....	21
3.2 KVM 虚拟化实现原理 .....	24
3.2.1 KVM 环境初始化.....	25
3.2.2 KVM 面向用户空间的接口.....	29
3.2.3 虚拟机运行模式 .....	32
3.2.4 QEMU 创建虚拟机流程.....	34
3.3 KVM 虚拟机性能优化思路 .....	36
3.4 VCPU 陷入事件的捕获方法与信息统计 .....	37
3.5 本章小结.....	47
第 4 章 KVM 虚拟机调度优化.....	48
4.1 Linux 进程调度策略 .....	48
4.1.1 进程优先级与时间片关系 .....	50
4.1.2 普通进程调度算法 .....	51
4.1.3 实时进程调度算法 .....	52
4.2 KVM 虚拟机调度优化方案 .....	53
4.3 KVM 虚拟机调度优化方案实施 .....	54
4.3.1 实现进程信息实时监控 .....	54
4.3.2 调整调度策略 .....	57
4.3.3 修改运行优先级 .....	58
4.4 本章小结.....	59
第 5 章 虚拟机内存使用优化 .....	60
5.1 内存虚拟化技术原理.....	60



5.2 内存使用性能优化.....	61
5.2.1 KSM 内存使用优化.....	61
5.2.2 Virtio 下的 balloon 气球技术.....	63
5.2.3 大页 (Huge Page) 技术 .....	64
5.3 本章小结.....	66
第 6 章 I/O 性能优化 .....	67
6.1 virtio 半虚拟化技术 .....	67
6.2 设备直接分配.....	71
6.2.1 网卡直接分配 .....	72
6.2.2 硬盘直接分配 .....	73
6.2.3 USB 直接分配.....	73
6.3 网络访问优化技术 (SR-IOV) .....	74
6.4 本章小结.....	75
第 7 章 性能优化测试与结果分析 .....	76
7.1 测试环境.....	76
7.2 KVM 虚拟机调度优化测试 .....	77
7.3 内存使用优化测试.....	78
7.4 I/O 优化测试 .....	79
7.4.1 网络访问性能测试 .....	79
7.4.2 磁盘读写性能测试 .....	79
7.5 VCPU 陷入事件捕获数据与优化分析 .....	80
7.6 本章小结.....	81

第 8 章 总结与展望 .....	82
参 考 文 献 .....	84
致 谢 .....	87

## 第 1 章 绪论

### 1.1 研究背景

随着计算资源应用技术研究与实践不断发展,云计算、公有云、私有云、IT 服务云端化已成为用户使用计算资源的一种高效、廉价、优质的选择,与之相关的虚拟化技术是这些云服务的基础。在各领域中的应用实践使信息化服务模式发生了改变,使数据中心运维能效得到了提升。虚拟化技术可应用的范围很广,可以实施虚拟化的领域包括服务器、网络、存储、桌面甚至某些应用程序等<sup>[1]</sup>。本文介绍的虚拟化主要是基于 KVM 虚拟化方案下的服务器虚拟化。在计算机硬件性能不断提高的同时,研究如何充分利用硬件资源,节能提效,已成为计算机应用的关注点。将一台物理计算机虚拟化,使其能够创建多台虚拟机是从时分复用、资源共享的角度来提高资源利用率的有效方法。服务器做为基本的物理计算节点,在其上通过软件对 CPU、内存、输入输出设备等进行抽象和虚拟,再安装不同的操作系统,使其相互隔离,那么数据的保存、复制和恢复将变得更加容易,其在云计算、数据中心、软件测试、服务器应用部署、桌面系统、自动化管理等多个方面都有广泛的需求和应用前景<sup>[2]</sup>。

在实际应用中,每一个从事机房与数据中心运维管理的技术人员,都需要面对大量新旧不一的服务器和多台 IP San 网络存储的资源分配与监管。节约人力和设备成本,提高资源利用率及运维效率,是当前计算机技术应用的一个主题。直接购买公有云服务可以实现这一目标,为一种转变运维思路的方案。但公有云主要面向整个互联网,提供网络和虚拟化基础设施规模虽然很大,但服务质量不可控,目前我国发展情况表明性价比不高。因此,构建私有云或服务器虚拟化平台是达到资源共享、节能高效目标的必然选择。近年来,虚拟化技术发展迅猛,KVM 作为后起之秀,以 Linux 内核服务的模式为我们提供了一个低成本、高性能、易用稳定的开源虚拟化计算资源的解决方案。在数据中心中合理的应用开源虚拟化技术,实现资源分配与管理,根据实际运行状态有针对性的进行性能优化,完全可以与商业化方案同效,确保提供高质量,高数据安全,高稳定性的信息化服务,使信息化服务和运维变得更加便捷和智能。同时,开源技术有着先天的成本优势,是值得主导和推广的应用技术。基于这些原因,本文拟从实际工作需求出发,对 KVM 虚拟机的运行性能进行研究与改进,寻求一种低成本、简单高效、管理方便的虚拟化解决方案。

## 1.2 KVM 虚拟化的意义

在互联网快速发展的今天,运用开源虚拟化技术保障与推动现代数据中心的建设与维护有非常广阔的发展前景和革新信息服务模式的深远意义。比如方便、快速部署、安全备份、随处使用、按需分配与付费等,这些构想完全可以通过虚拟化技术来实现。虽然服务器虚拟化技术已经比较成熟,得到了广泛应用,但在实际运用中并非所有虚拟化的服务都令用户满意,如何实施性能优化,一直是虚拟化研究的主题,在不断发展改进。因此,研究分析 KVM 虚拟机的性能,通过设计相应的改进方案,解决在实际工作中遇到的性能瓶颈问题,实现开源技术在服务器虚拟化应用中不输于商业方案的价值,对数据中心运维而言,有着重要的作用和意义。

1. 构建 KVM 虚拟化平台,实现资源快速、高效、稳定、安全、透明、按需分配,充分利用开源技术的优势,节省数据中心运维成本。

2. 研究 KVM 在 Intel x86 服务器上的硬件辅助虚拟化(Intel VT-x 技术)的运行原理,结合 Linux 中 KVM 虚拟机进程的控制策略,掌握虚拟化过程中性能损耗环节的特点,有针对的实施处理器调度、内存管理和 I/O 设备访问性能等的优化,提高硬件的有效利用率,改善 KVM 虚拟化资源的整体运行性能,在实际应用中可作为商业虚拟化方案的有效补充。

3. 通过分析虚拟资源与物理资源在运行性能上的区别,运用各种资源调度优化策略,有效推动 KVM 虚拟化技术的改进和发展,进一步在实际应用中缩小在成本、维护灵活性、硬件利用率、高性能等方面的差距,探索一种优于商业化产品的数据中心虚拟化运维解决方案。

## 1.3 虚拟化发展现状

虚拟化技术是一套软硬件相结合,将硬件抽象封装并虚拟成各种计算资源的虚拟化解决方案,可产生的虚拟计算资源包括 CPU、内存、I/O 设备、存储及网络等,提供分布式计算与存储应用基础,是云计算服务发布的关键技术<sup>[2]</sup>。它直接提供云计算底层需要的虚拟服务器、网络 and 存储等资源。越来越多的知名实力 IT 厂商都在研发和发布自己的虚拟化产品或云计算解决方案。这些企业产品中最有代表性的是 Amazon(亚马逊)的弹性计算云,Google(谷歌)的云计算平台,华为云、阿里云、腾讯云、华三云、深信服 acloud 云架构等,都是当前拥有较多

用户，业务范围广泛的主流平台。其中谷歌 GCE 的底层采用 KVM 虚拟化技术，其他云服务厂商也同样选择 KVM、Xen 等开源虚拟方案来开发自己的云操作系统。老牌虚拟化软件厂商 Vmware 也在发展自己的云服务产品，但其服务相对封闭，只支持自有的产品。IBM、思科、微软等厂商也都在加强虚拟化技术的研发和服务提供<sup>[3]</sup>。

政府、教育、银行、企业等的数据中心正在改变传统的运维模式，通过构建私有云或购买云服务，或虚拟化数据中心等方式来获取更高效、灵活、可靠的信息化服务。云计算应用的深入正在不断推动虚拟化技术的更新与发展<sup>[4]</sup>。值得一提的是，大部分厂商包括采用开源虚拟化技术的产品，在提供虚拟化和云服务时都变得不再开源，必须支付高昂的软件授权费用，产品的横向扩展性较弱，体现在增加计算节点时会遇到“困难”，是有条件的扩展。比如增加新的计算节点，需要新的 CPU 颗数授权，软件与其他厂商不兼容，或都只能采购指定厂商的物理设备，使用成本仅是相对于传统数据中心因提高硬件资源的利用率而产生的有限下降。虽然部分厂商在推广期免费使用，但并不适用长期用户，而且多数虚拟化软件捆绑硬件进行销售或做 CPU 授权限制。同时，商业利益的驱动使得虚拟化及云服务厂商考虑要实现的都是大范围内的通用型服务，只能满足用户的基本业务需求，行业适应性较差，而针对具体业务做定制化改变，成本非常大，中小数据中心无法满足这样的服务开支。另外，由于产品化的服务，一定程度损失了维护的灵活性。当服务器安装完虚拟化或云平台软件之后，呈现给管理者的通常是 WEB 管理界面。服务器系统级的维护基本由厂商完成，适合只使用基本云服务如申请、创建、访问、虚拟主机管理等终端用户，他们不处理故障和关心性能优化。对于具有较强的维护力量和计算机专业能力的运维人员而言，更想要一个能监管虚拟计算资源的承载环境，但这类产品执行的是排他性的维护模式，平台的运维和管理侧重于便利而失去了灵活。因此，要在数据中心实现真正的降低成本，节能减排，研究与应用开源虚拟化技术是计算机技术中值得投入的主题。

近 10 年来，虚拟化开始应用在 x86 架构上。在 x86 计算机体系架构中，世界两大 CPU 厂商 Intel 和 AMD 都开发了各自的硬件辅助虚拟化技术，Intel 从 2005 年开始在其处理器产品线中推广应用 Intel Virtualization Technology(Intel VT) 虚拟化技术，AMD 比 Intel 晚几个月发布 AMD Secure Virtual Machine(AMD SVM) 虚拟化技术。虚拟化软件结合硬件辅助技术，以及计算机各组成硬件性能的提升

使虚拟化技术的应用越来越普及。发展成了开源的和商业的虚拟化方案并存的局面，它们相互竞争，都在不停的更新发展。目前应用广泛的主要有 Vmware、Xen、QEMU、VirtualBox、Hyper-V、KVM 等，究竟谁才是未来的虚拟化技术引领者，还未可知<sup>[3]</sup>。

### 1. VMWare 的虚拟化技术

Vmware 是大家非常熟悉的虚拟化软件厂商，其产品采用全完虚拟化技术架构，主要产品是 Vmware vSphere、Vmware vStorage、Vmware ESX、等。产品服务在虚拟化领域几乎全覆盖，如服务器、桌面、存储虚拟化等。其中 VMware Workstation 是工作站桌面虚拟化软件，用于个人和开发人员，应用相当广泛，其他为企业级产品，都是商业化付费服务产品。

### 2. Xen

Xen 是由剑桥大学开发的一款开源虚拟化软件，2005 年从 Xen3.0 开始正式支持 Intel 的 VT 技术和 IA64 架构，使 Xen 虚拟机可以运行完全没有修改的操作系统。2007 年被思杰公司收购，开始提供商业化的桌面和服务器虚拟化产品。值得说明的是 Xen 在 Linux 系统中继续保持开源性，但 Linux 系统中原生的开源虚拟化解方案采用了 KVM<sup>[5]</sup>。

### 3. QEMU 软件

QEMU 是非常有名的开源虚拟机软件，以软件方式实现了完全虚拟化服务，可虚拟处理器、内存、I/O 设备（如网卡、显卡、存储控制器和硬盘等）。QEMU 是纯软件的虚拟化方案，基本没有直接应用其做虚拟化服务的，而是选择与 Xen、KVM 等技术相结合，加入硬件辅助虚拟化支持，从而提供高效可用的虚拟化解方案。

### 4. VirtualBox

VirtualBox 与 KVM 一样是开源虚拟化软件，目前由 Oracle 公司进行开发。支持纯软件完全虚拟化，也支持 Intel VT-x 与 AMD-V 硬件辅助虚拟化。因为各种虚拟化技术都在不断完善和发展，与 KVM 比较，目前在图形显示效果和性能上略优，更适合桌面虚拟化应用。

### 5. KVM 虚拟化技术

KVM 最早由以色列 Qumranet 公司开发，是基于 GPL 授权方式的开源虚拟机软件，目前以模块方式集成在 Linux 内核中，其虚拟机以 Linux 普通进程方式

存在, 与其他应用程序一起, 由 Linux 内核的进程调度策略统一管理。KVM 必须有硬件辅助虚拟化技术的支持, 其核心源代码很少, 只实现了 CPU、内存、和设备直接访问等虚拟化功能, I/O 处理相关的设备模拟是通过 QEMU 实现, 并由 QEMU 启动虚拟机。支持的虚拟机操作系统比较全面, 是 Linux 完全原生的基于 Intel VT 或者 AMD SVM 技术的硬件辅助全虚拟化解决方案<sup>[6]</sup>。

## 1.4 研究内容与论文结构

### 1.4.1 研究内容

目前 Intel x86 架构的服务器在各机构的数据中心日益普及, 硬件虚拟化提高资源利用率和节能环保越来越成为主流。本文通过研究 KVM 在 Intel x86 服务器上的硬件辅助虚拟化 (Intel VT-x 技术) 的运行原理<sup>[7]</sup>, 分析影响虚拟机性能的关键环节, 提出基于处理器角度的性能优化思路, 设计和实现 KVM 机制下引起 VCPU 中断陷入 (VM-Exit) 原因的信息统计方法。研究 Linux 的进程调度策略, 提出虚拟机调度优化方案, 设计并实现进程信息的实时监控程序, 监控虚拟机运行进程的各项资源占用状态, 实现虚拟机进程优先级和调度策略的实时修改方法, 根据各虚拟机的实际需求优化处理器调度策略, 进行个性化设置。研究 KVM 在内存管理、I/O 性能优化领域的最新技术, 在上述监控统计信息的基础上, 实现动态调整虚拟机内存, 灵活应用 I/O 优化技术, 提高虚拟机的整体运行性能。主要的研究内容如下:

(1) 研究 KVM 虚拟化架构, 与具体的 Linux 系统相结合, 延伸 KVM 的 VMM 范畴, 将 Linux 内核扩展成主要为 KVM 服务的 Hypervisor。通过深入理解虚拟化模型和架构实现的相关技术, 进一步总结与性能优化有关的各项虚拟化技术, 确定虚拟机的性能研究与改进的可行方向。并以此为基础构建 KVM 虚拟化平台, 将物理服务器转化为可实际运用在数据中心的虚拟计算资源节点。

(2) 处理器虚拟化是 KVM 技术中最核心的部分, 通过理解 Intel VT-x 技术文档, 分析 Linux 内核中 KVM 关于 CPU VT-x 虚拟化实现的源代码, 研究 KVM 在 Intel x86 服务器上的硬件辅助完全虚拟化 (Intel VT-x 技术) 的实现原理和运行机制。

(3) 结合虚拟机运行模式, 从虚拟机中运行敏感指令使 VCPU 触发异常陷入 (VM-Exit) 的执行路径出发, 分析 VCPU 陷入-操作模式切换-重新进入虚拟机 (VM-Entry) 运行的处理机制和影响虚拟机性能的关键环节, 提出基于处

理器角度的 KVM 虚拟机性能优化思路,设计和实现 KVM 机制下引起 VCPU 中断陷入原因的信息统计方法。

(4) 研究 Linux 的进程调度策略和优先级与 CPU 时间片分配的联系,结合虚拟机承载的业务系统要求,提出了 KVM 虚拟机调度优化方案。通过动态添加 Linux 内核模块(mykvm\_LKM),创建/proc/taskinfo 文件读取进程信息

(task\_struct),使用 C 语言设计并实现进程信息的实时监控程序 getProcessInfo。根据虚拟机进程的相关状态,有针对的实施虚拟机的调度优化,能灵活且实时的修改虚拟机进程运行优先级和调度策略,达到了按实际业务需求来优化处理器调度策略和个性化设置的目标,使虚拟机能够按实际需求获得更多占用物理处理器资源的机会,改进 KVM 虚拟机的整体性能。

(5) 从优化内存的使用率和访问效率两方面,研究优化内存使用的三种主要技术,根据物理资源环境和业务系统需求提出内存优化方案和适应的场景,并给出了具体实现方法和步骤。

(6) 深入分析虚拟机的三种 I/O 访问类型,研究以 virtio 为主的半虚拟化设备访问原理和实现过程,确定半虚拟化是 KVM 虚拟机 I/O 访问性能改进的主要方案。提出了硬件直通模型中 I/O 虚拟化技术的应用场景和以具体业务相结合实施 I/O 优化的思想。通过实验,给出了 virtio、Intel VT-d 和 SR-IOV 标准下虚拟机 I/O 性能优化的实现方法和步骤。

#### 1.4.2 论文结构

本文包含八章,各章内容如下:

第1章 绪论。介绍 KVM 虚拟化发展背景和现状,以及 KVM 虚拟化研究的意义。简要介绍了本文研究的主要内容和论文组织结构。

第2章 KVM 虚拟平台的构建。介绍 Centos7 系统上编译最新 KVM 内核模块、安装最新版本 I/O 模拟软件 QEMU 和虚拟机管理软件 libvirt,构建 KVM 虚拟化平台,并进行环境调优的过程。

第3章 KVM 虚拟化实现原理与机制。研究 KVM 硬件辅助完全虚拟化(Intel VT-x 技术)的实现原理和运行机制,提出基于处理器角度的 KVM 虚拟机性能优化思路,设计和实现了一套 VCPU 陷入事件的捕获方案。

第4章 KVM 虚拟机调度优化。在研究 Linux 进程管理方式基础上介绍了 KVM 虚拟机调度优化方案,并设计和实现该方案,使虚拟机能够获得更多处理



器资源，改进 KVM 虚拟机的整体性能。

第5章 虚拟机内存使用优化。介绍优化内存的使用率和访问效率的主流技术，提出了内存优化方案和适应的场景，并给出了具体实现方法和步骤。

第6章 I/O 性能优化。深入分析虚拟化 I/O 访问原理和实现过程，重点介绍 I/O 半虚拟化的性能改进方案。提出了硬件直通模型中 I/O 虚拟化技术的应用场景和以具体业务相结合实施 I/O 优化的思想。

第7章 性能优化测试与结果分析。通过实验对 KVM 性能优化方案的可行性进行测试，分析相关结果数据。

第8章 总结与展望。对本文进行了总结

### 1.4.3 主要创新点

基于上述研究内容，该论文有以下几点创新：

(1) 在深入理解虚拟机运行模式的基础上，提出了三种基于处理器角度的 KVM 虚拟机性能优化思路，设计和实现了一套 VCPU 陷入事件的捕获方案，针对第三种思路，以虚拟机运行过程中统计上下文切换数量作为分析维度。在该思路下，还可以改变统计的维度（即统计其他切换信息），开展定性定量分析，并以此为基础针对具体虚拟机实施相应的性能优化。

该 VCPU 陷入事件捕获方案的特点是：在第一种优化思路的基础上，不修改异常处理方式，不人为改变系统的切换规律，将每个虚拟机的切换信息记录下来，并传递给用户空间，一方面可以为分析虚拟机性能开销提供具体异常下的切换数据，判断虚拟机的业务类型，从而决定采取的性能改进方案。另一方面，建立了用户程序获取 KVM 模块运行时数据的一种简便方法，为开展虚拟机性能优化研究提供定量和定性分析的数据支持。

(2) 通过研究 Linux 进程管理机制，设计和实现了一个进程监控程序，结合实时监控，可针对业务特点灵活改变虚拟机的调度策略和运行优先级，使虚拟机能够按实际需求获得更多占用物理处理器资源的机会，改进 KVM 虚拟机的整体性能。

(3) 结合内存虚拟化实现原理，对提升 KVM 虚拟机内存访问性能的 KSM、balloon、大内存页技术进行了深入分析，从虚拟机内存访问效率和使用率两方面提出了根据具体业务特点开展优化配置的方案。

(4) 论证并确定了半虚拟化是 KVM 虚拟机 I/O 访问性能改进的主要方案。

提出了硬件直通模型中 I/O 虚拟化技术的应用场景和以具体业务相结合实施 I/O 优化的思想。

## 第 2 章 KVM 虚拟平台的构建

KVM 的全称是基于内核的虚拟机 (Kernel Virtual Machine)<sup>[8]</sup>, 最早由以色列的 Qumranet 公司开发, 在 2008 年被 Red Hat 收购, 并从 Linux 2.6.20 版本开始作为内核模块加入 Linux 操作系统。Linux 内核源代码中就包含了 KVM 模块的源代码, 最新版本的内核中 KVM 版本也是最新的。通过 KVM 构建服务器虚拟平台, 是 Linux 系统上完全原生的基于 x86 平台的硬件辅助的完全虚拟化解决方案。包括 x86\_64 平台在内, KVM 需要 Intel VT 或 AMD SVM 的硬件虚拟化技术支持<sup>[9]</sup>, 本文主要在开启 Intel VT 支持的服务器上开展 KVM 虚拟机的性能研究与改进。

本章介绍 KVM 虚拟化架构, 总结与性能优化有关的各项虚拟化技术, 并在最新的 Centos7 64 位操作系统上构建 KVM 虚拟化平台, 通过修改操作系统管理和调度资源的部分服务设置, 延伸 KVM 的 VMM 范畴, 将 Linux 内核扩展成主要为 KVM 服务的 Hypervisor<sup>[10]</sup>。以将物理服务器转化为可实际运用在数据中心的虚拟计算资源节点为目标, 完成虚拟化基础软硬件环境的配置, 为后续虚拟机的性能研究与改进确定可行方向。

### 2.1 KVM 虚拟化架构

KVM 虚拟化技术已发展超过十年, 各种优化技术在快速改进, 而 KVM 架构和虚拟化模型相对保持稳定, 深入理解其架构是虚拟机性能研究和改进的基础。

首先, KVM 是完全虚拟化解决方案。虚拟环境由硬件、虚拟机监控器 (VMM) 和虚拟机三个部分组成, KVM 内核模块就是 VMM。在虚拟机安装操作系统与应用软件和物理机上完全一样, 客户机操作系统不需要做任何修改就能正常运行, 管理并使用属于自己的各种“物理硬件”。KVM 方案中将 VMM 运行在 x86 处理器指令集的最高特权级 Ring 0 上, 虚拟机 (客户机) 操作系统作为 VMM 中运行的一个程序, 运行在最低特权级 Ring 3 上, 当虚拟机操作系统执行特权指令时, 因需要最高的特权却无法fe足, 由 CPU 控制引发异常 (中断), 使 CPU 退出客户机运行, 进入 VMM 控制模式 (称为陷入到 VMM, 即 VM-Exit)。VMM 捕获到这个异常, 进行处理后再由 VMM 发起, 恢复客户机退出时保留的现场, CPU 重新载入客户机代码继续运行 (称为进入虚拟机, 即 VM-Entry)。在虚拟机的角度, 认为特权指令执行过程和物理平台上完全相同, VMM 的中间处理层

成功的屏蔽了虚拟化对客户机操作系统的影响。KVM 通过这种模式实现了客户机操作系统无需任何改动的完全虚拟化<sup>[11]</sup>。详细的实现原理将在下一章中介绍。

其次，KVM 是硬件辅助虚拟化技术。相对于通过优先级压缩和二进制代码翻译相结合的软件完全虚拟化，硬件辅助虚拟化的效率更高。VMM 处理完敏感指令（为区别物理机操作系统，称虚拟机中的特权指令为敏感指令）后再次进入虚拟机时不需要模拟返回的数据。KVM 实现物理资源的处理器虚拟化、内存虚拟化、I/O 虚拟化方面都有硬件虚拟化技术支持，Intel 平台上硬件虚拟化总称是 Intel VT，在 CPU 虚拟化方面提供 Intel VT-x (Intel Virtualization Technology for x86) 技术，在内存虚拟化方面提供 EPT 扩展页表 (Extended Page Table) 技术，在 I/O 设备虚拟化方面提供了 Intel VT-d (Intel Virtualization Technology for Direct I/O) 技术。AMD 平台也提供了类似的技术<sup>[1]</sup>。

最后，KVM 是宿主型软件实现技术。目前 VMM 架构主要有三种类型，分别是 Hypervisor 模型、宿主 (Hosted) 模型、混合模型<sup>[1]</sup>。KVM 的 VMM 由 `kvm-amd.ko`、`kvm-intel.ko`、`kvm.ko` 三个 Linux 内核模块组成，最初只负责硬件处理器的虚拟化解析，为改进虚拟机性能，后续加入了对内存虚拟化和 I/O 虚拟化的解析<sup>[2]</sup>。

在 KVM 虚拟化方案中，Linux 系统将物理资源和管理、调度交给内核其他模块处理，不由 KVM 模块负责，从而使 Linux 内核变成了 KVM 的宿主机，因此，在虚拟机的管理和调度上会受宿主机的限制。Hypervisor 是系统管理程序的意思，代表硬件和操作系统之间的资源管理软件，虚拟化技术中承担 VMM 的角色，Linux 内核在 KVM 方案的作用就相当于这个 Hypervisor 程序，实际接管了操作系统的资源管理和调度权，但还负责除虚拟化之外的应用服务。如果换一种角度，延伸 KVM 的 VMM 范畴，将 Linux 内核中影响和限制虚拟化的模块关闭，打开提升虚拟化性能的模块服务，使 Linux 内核主要为虚拟化提供服务，原 VMM 转变成功能更强大的 Hypervisor (即提升了管理调度权限的 VMM)。根据上述理解和分析，可得到 KVM 虚拟化架构如图 2-1 所示：

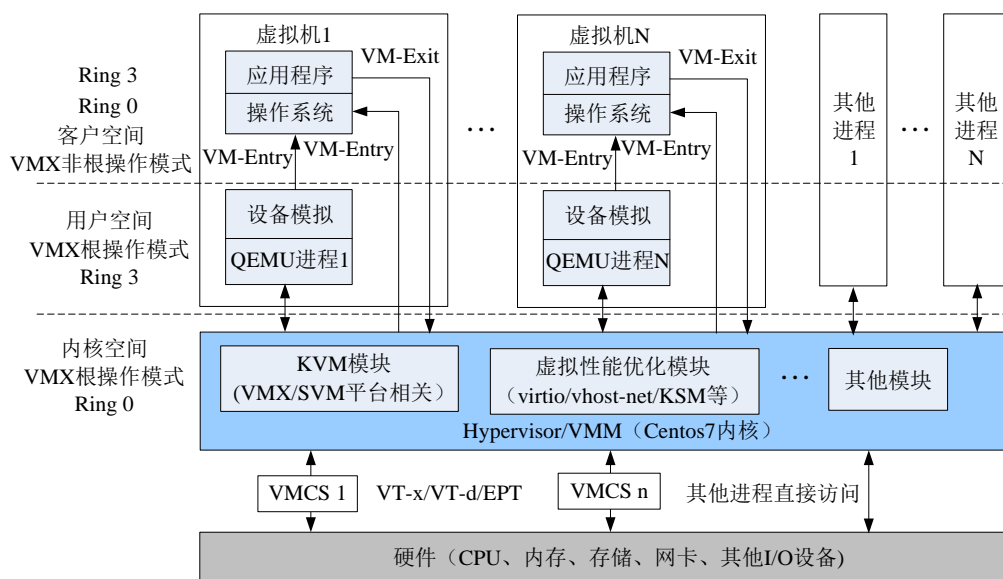


图 2-1 KVM 虚拟化架构

KVM 虚拟化架构是为更好的实现虚拟化机制。由上图可知，目前架构中由 QEMU 负责设备模拟，虚拟出启动客户机所需要的 BIOS 设备，通过内核中 `/dev/kvm` 文件设备与 KVM 模块通信（VMM 中与处理器、内存、DMA 方式 I/O 直接访问有关的核心部分），执行 `ioctl` 系统调用函数<sup>[1]</sup>，转到内核 VMM 中创建虚拟处理器（VCPU）线程，加载客户机操作系统代码执行，实现虚拟机的创建和运行。这个过程称为进入虚拟机（VM-Entry），可以创建多个 VCPU 线程运行客户机代码，因为都由一个 QEMU 进程创建，所以都归属于同一个虚拟机。

完全虚拟化时 QEMU 负责虚拟机的 I/O 设备模拟和访问处理，当虚拟机代码执行了敏感指令，比如需要访问磁盘、网络等 I/O 设备时，VMM 会控制 VCPU 中断执行，转交给 QEMU，这个过程称为退出虚拟机（VM-Exit），QEMU 模拟设备处理后执行 `ioctl` 系统调用，恢复中断现场创建新的 VCPU 线程继续运行客户机代码。这种方式称为“陷入再模拟”。KVM 虚拟机的所有敏感指令都能被 VMM 识别为特权指令，进而能引发陷入，实现处理器指令级别的截获并重定向。因为对真实计算机系统各部分的访问都是由指令承载的，所以 VMM 通过这种机制可以模拟出一个与真实物理机完全一样的环境，就是虚拟机。

半虚拟化(也叫准虚拟化)时与全完虚拟化的运行架构基本一致，只在 QEMU 模拟设备与虚拟机之间直接建立了驱动通道，改进了纯软件对设备驱动指令的原生模拟来转换数据的性能，如 virtio 驱动中采用缓冲机制。

设备直接访问在上述的架构中表现为在引发陷入后（部分 VT-d 技术可不陷

入），由 VMM 在内核及硬件虚拟化支持下实现虚拟机直接访问真实硬件设备，跳过 QEMU 的中间层处理，建立通道后再进入虚拟机（VM-Entry）。

在该架构下 KVM 虚拟机的运行流程如图 2-2 所示：

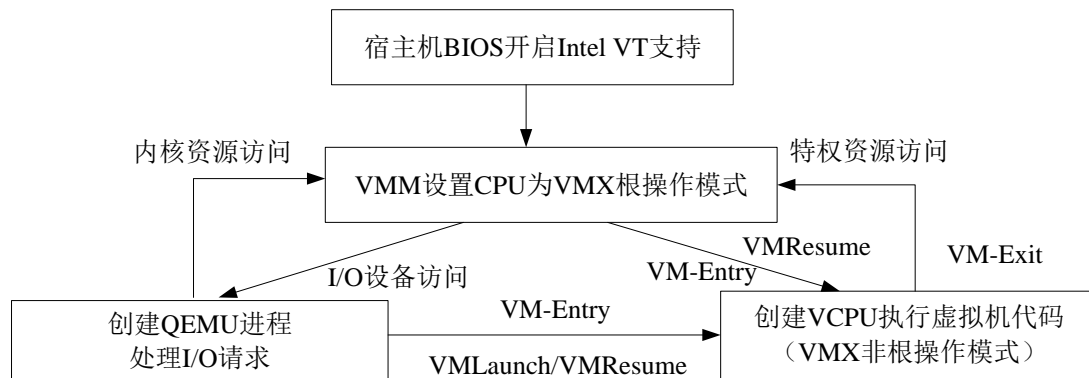


图 2-2 KVM 虚拟机的运行流程

## 2.2 KVM 性能优化技术

虚拟化通过在一套硬件系统上并发运行多个操作系统来增加程序的运行密度，从而提高硬件的实际使用率，但“陷入再模拟”的机制使虚拟机代码执行过程增加了大量的模拟和上下文切换，造成明显的性能下降。要达到接近物理系统的执行性能，需要从 VMM 资源管理与调度、处理器、内存、I/O 设备虚拟化等几个方面进行性能优化。改进时涉及的具体领域、方法和相关技术如下：

（1）Linux 内核资源管理与调度优化。KVM 是基于内核的虚拟化实现，只完成 CPU、内存与硬件相结合的程序实现，作为 VMM，它的功能较弱，必须借助 Linux 内核的其他各项功能模块来构建完整的 VMM 系统。因此，本文首先从该领域开始实施性能改进。

（2）硬件辅助虚拟化支持。这是 KVM 实现的基础，体现在 Intel 处理器 VT 技术和其他芯片组、网卡、存储设备、GPU 的虚拟化支持。其中，深入理解处理器虚拟化在 KVM 中的实现原理和运行机制，利用硬件支持来有针对性的实施优化是提升虚拟机效率的关键。

（3）内存优化技术，包括内存扩展页表（EPT）<sup>[2]</sup>、内存大页（Huge Page）、ballooning 气球技术、内核同页合并 KSM<sup>[12]</sup>技术（Kernel SamePage Merging）。

（4）半虚拟化（准虚拟化）技术，包括 virtio 设备驱动技术，主要有 virtio\_net、virtio\_blk、vhost\_net 等 I/O 设备半虚拟化驱动技术。

（5）设备直接分配技术（VT-d），包括网卡直接分配、硬盘直接分配、

USB 设备直接分配、显卡直接分配等技术<sup>[13]</sup>。

(6) SR-IOV(Single Root I/O Virtualization)单根 I/O 虚拟化技术, 主要用于优化网卡性能<sup>[14]</sup>。

上述技术的应用能不同程度改进 KVM 虚拟机性能, 本文对性能的研究与改进将先从虚拟平台构建的环境优化开始, 再到其他关键优化技术的应用, 在 KVM 平台整体上采取分步优化的策略, 最后结合实际运行的虚拟机应用情况选择性进行优化。

## 2.3 KVM 虚拟平台构建

KVM 性能研究与改进首先应从虚拟平台的构建开始, 优化 Linux 内核模块, 增加对 VMM 的支持, 减少资源管理和调度的限制。本文在最新的 Centos7 64 位操作系统上构建 KVM 虚拟化平台, 通过修改操作系统管理和调度资源的部分服务设置, 延伸 KVM 的 VMM 范畴, 将 Linux 内核扩展成主要为 KVM 服务的 Hypervisor<sup>[11]</sup>。以将物理服务器转化为可实际运用在数据中心的虚拟计算资源节点为目标, 开展 KVM 运行环境的性能研究与改进。虚拟平台构建和优化流程如图 2-3 所示:

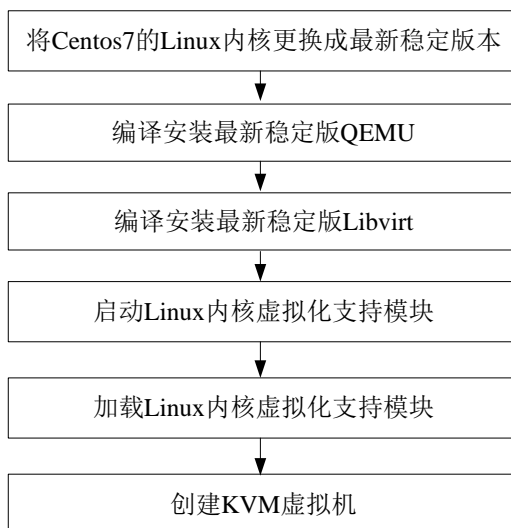


图 2-3 KVM 虚拟平台构建和优化流程

KVM 源代码的最新版本包含在 Linux 内核的最新版源代码中, 编译安装内核后也安装了 KVM 模块, 后续对 KVM 源代码的修改, 只需重编译 KVM 内核模块即可完成改动, 而不用编译整个内核。

### 2.3.1 更换 Centos7 内核

到 Linux 内核官网 (<http://www.kernel.org>) 上下载最新版的内核<sup>[15]</sup>, 解压到

/usr/src/kernels/linux-4.17.8 目录，执行以下命令进行编译和安装：

```
cd /usr/src/kernels #进入工作目录
```

```
tar -xvf linux-4.17.8.tar.xz #将内核源代码解压到工作目录下的 linux-4.17.8 目录中，以此作为源代码根目录。
```

```
make menuconfig #设定编译选项，主要确认有关虚拟化的模块是否选择，其他保持默认状态。该命令后会在当前目录生成.config 文件（为隐藏文件），其中保存的内容是新内核即将编译的功能选项记录。
```

```
make -j 16 #调用 16 个 cpu 进程来执行编译工作，数字越大，速度越快，当然，需要服务器有这个数量的 cpu。该命令相当于执行了内核 kernel 编译（make vmlinux -j 16）、bzImage 编译（make bzImage -j 16）、module 编译（make modules -j 16），并生成相关目标文件到源码目录结构中。在编译过程中可能遇到缺少某些软件包，逐一安装后可顺利完成编译。
```

```
make modules_install #将编译好的 module 安装到相应的目录之中，在默认情况下 module 被安装到/lib/modules/$kernel_version/kernel 目录中。
```

```
make install #将在/boot 目录下生成 vmlinuz 和 initramfs 等内核启动所需的文件。grub 配置文件中也将自动添加最新内核的启动选项。
```

设置默认的启动内核：

```
cat /boot/grub2/grub.cfg |grep menuentry #查看当前系统包含几个内核
```

```
grub2-set-default "CentOS Linux (4.17.8) 7 (Core)" #设置最新内核为默认启动的内核。
```

```
grub2-editenv list #验证是否修改成功，如显示 saved_entry=CentOS Linux (4.17.8) 7 (Core)，则设置成功。
```

最后，重启服务器，用最新的 Linux 内核管理 Centos7，最新版本的 KVM 模块默认被加载到内核服务中。配置内核编译选项界面如图 2-4 所示：



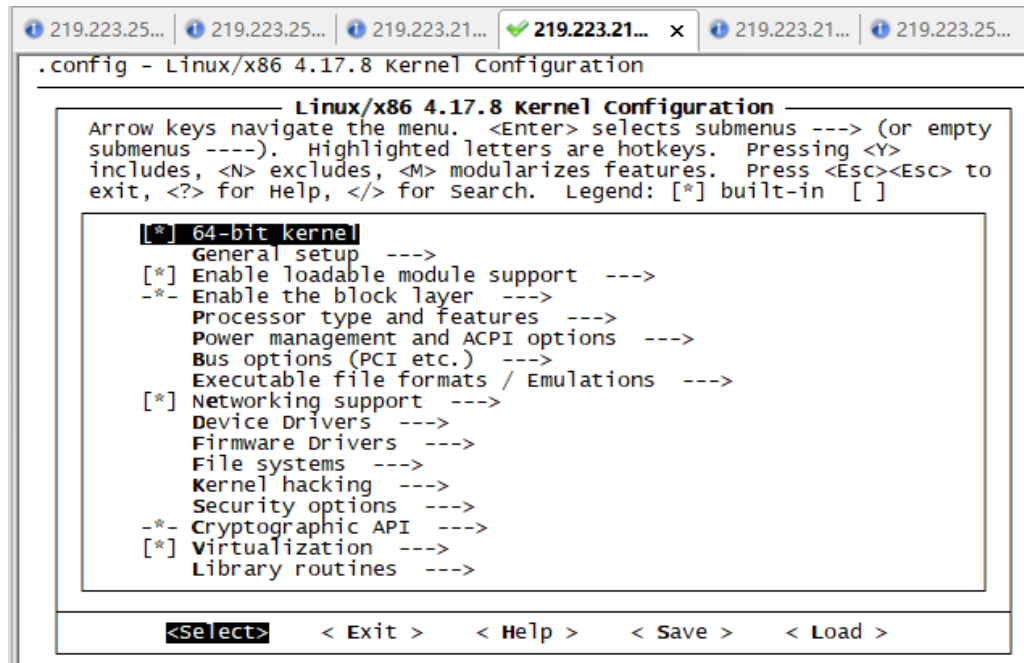


图 2-4 配置 Linux 内核编译选项界面

### 2.3.2 编译和安装 QEMU

KVM 的输入输出都交给了 QEMU 处理，QEMU-KVM<sup>[16]</sup>是专为 KVM 虚拟机加速而产生的一个分支，在发布了三个版本后（最后一个版本为 1.3）与 QEMU 的主版本合并<sup>[3]</sup>，在 Linux 的发行版安装包还以 qemu-kvm 命名，现在最新的 QEMU 版本为 QEMU emulator version 2.12.91，已默认支持 KVM 了<sup>[17]</sup>。到 QEMU 官网（<http://www.qemu.org>）下载最新的源代码<sup>[18]</sup> qemu-3.0.0-rc1.tar.bz2，执行以下命令编译和安装：

```
cp qemu-3.0.0-rc1.tar.bz2 /usr/local/src # 复制源代码到工作目录
cd /usr/local/src # 进入工作目录
tar -xjvf qemu-3.0.0-rc1.tar.bz2 # 解压源代码到 qemu-3.0.0-rc1 目录
cd qemu-3.0.0-rc1 # 进入源代码根目录
yum install libaio-devel libcap-devel libiscsi-devel # 安装可能需要的包
./configure --enable-debug --prefix=/usr/local --target-list=x86_64-softmmu
# 配置 qemu 的安装选项，通过./configure -h 查看帮助，知道--enable-kvm
--enable-vnc 等是默认支持的，所以不需要显示写在 configure 后面。
make -j16 # 编译 qemu
make install | tee make-install-qemu.log # 安装 qemu，将编译生成的可执行
文件，一些库文件、配置文件复制到指定的目录。tee 命令表示安装信息在输出
```

到屏幕的同时再输出到指定的文件中。

安装后将在 `/usr/local/bin` 目录生成 `qemu-system-x86_64` 可执行文件，该文件是启动 QEMU 进程的命令行工具，发行版 rpm 包方式安装的命令名字为 `qemu-kvm`。为了方便程序调用，可在 Centos7 默认的命令执行路径中加入指向该命令的软链接：`ln -s /usr/local/bin/qemu-system-x86_64 /usr/bin/kvm`。这样在任意目录下执行 `kvm` 都将等同执行 `qemu-system-x86_64`，方便启动 QEMU 进程（即 KVM 虚拟机）。

### 2.3.3 安装与配置 Libvirt

Libvirt 和 Linux、KVM、QEMU 一样是一套免费开源的 C 函数库，可为 KVM 虚拟化管理提供 API 调用，即通过程序监控虚拟机的接口实现，是虚拟化管理工具管理虚拟机的底层开发工具包，也可以通过它提供的 shell 程序（`virsh`）直接管理虚拟机。当前最新版本是 `libvirtd (libvirt) 4.5.0`，可到 libvirt <sup>[19]</sup> 官网（<https://libvirt.org/index.html>）下载最新的源代码<sup>[20]</sup>，执行以下命令编译并安装：

```
cp libvirt-4.5.0.tar.xz /usr/local/src
cd /usr/local/src
tar -xvf libvirt-4.5.0.tar.xz # 解压源代码到 libvit-4.5.0 目录
cd libvirt-4.5.0
yum install libxml2* gnutls* pkg-config* libnl* libyajl* yajl-devel xslt*
device-mapper-devel libdevmapper libpciaccess* # 安装可能需要的包
./configure --prefix=/usr/local # 配置 libvirt 的安装选项
make -j16. # 编译 libvirt
make install | tee make-install-libvirt.log # 安装 libvirt，将编译生成的可执行文件，一些库文件、配置文件复制到指定的目录。tee 命令表示安装信息在输出到屏幕的同时再输出到指定的文件中。
```

安装的主要可执行程序（shell 命令）有 `virsh`、`libvirtd`。通过 `service libvirtd start` 或 `systemctl start libvirtd.service` 启动 libvirt 服务。也可执行 `libvirtd -d` 以守护进程的方式在后台运行 libvirt 服务。

一些重要的配置文件及目录说明：

`/usr/local/etc/libvirt/libvirt.conf` 用于配置 libvirt 连接的别名（如远程连接别名），方便 libvirt API 代码调用。

/usr/local/etc/libvirt/libvirtd.conf    libvirt 的守护进程 libvirtd 的配置文件。

/usr/local/etc/libvirt/qemu.conf    libvirt 管理 qemu 进程的配置文件，用于管理 KVM 虚拟机。

/usr/local/include 是 libvirt 头文件存放目录

/usr/local/bin 是 libvirt 相关命令存放目录，如 virsh

/usr/local/sbin 是 libvirtd 可执行文件存放目录

/usr/local/lib,libexec,share 是共享库文件存放目录

用 systemctl enable libvirtd.service 设置开机服务，修改相关启动权限，修改 /usr/local/etc/libvirt/qemu.conf，删除以下三项的#号注释：

```
vnc_listen = "0.0.0.0"
```

```
user = "root"
```

```
group = "root"
```

重启 libvirt 使修改生效。

### 2.3.4 创建虚拟机

上述软件安装完成后，该物理服务器就成功转变成了一台 KVM 虚拟计算节点，并且各部分都使用了最新最优的虚拟化技术，可以开始创建虚拟机提供业务服务。为了增加虚拟机执行性能，在创建之前还需要进一步调整 Linux 内核，关闭或删除与 KVM 无关的模块、进程，加载提升虚拟机性能的模块，这些优化调整如下：

(1) 建立虚拟网卡与物理网卡的桥接。

在物理网卡的配置文件中增加桥接选项 **BRIDGE=br0**，再新建虚拟网卡配置文件 ifcfg-br0。将 IP、子网掩码、网关、DNS 等主要选项内容设置成与物理网卡一致。执行命令如下：

```
vim /etc/sysconfig/network-script/ifcfg-br0
```

```
DEVICE=br0
```

```
TYPE=bridge
```

```
ONBOOT=yes
```

```
BOOTPROTO=static
```

```
IPADDR=219.223.215.198
```

```
PREFIX=25
```

```
GATEWAY=219.223.215.129
```

```
DNS1=219.223.254.2
```

重启网络服务 `service network restart` 完成虚拟网卡桥接设置。

如果内核不能自动桥接成功，可执行以下命令手动桥接：

```
brctl show # 查看桥接网络
```

```
brctl addbr br0 # 加 br0 虚拟网卡接口，重启动网卡时会读 ifcfg-br0 配置
```

```
brctl addif br0 eth0 # 桥接物理网卡 eth0，该名称根据不同服务器可能不同，出错时可通过 brctl delbr br0 删除虚拟网卡 br0，重复上面的操作，直到成功桥接。
```

## （2）加载 virtIO 半虚拟化设备驱动模块。

Centos7 最小安装后，内核默认不加载 virtio 模块，需要手动按顺序加载，执行命令如下：

`modprobe virtio`，`modprobe virtio_ring`，`modprobe virtio_pci` 这三个命令按顺序加载，提供对 virtio API 的基本支持，任何 virtio 前端驱动都需要使用。

```
modprobe virtio_blk # 加载磁盘半虚拟化驱动
```

```
modprobe virtio_net # 加载网络设备半虚拟化驱动
```

```
modprobe vhost_net # 加载网络设备半虚拟化后端驱动，该模块运行架构优于 virtio_net，可跳过 QEMU 层直接在内核中完成与虚拟机进程的网络数据传输，但该驱动不能自动同步内核与虚拟机网络收发速率，两者差距较大时，效率反而下降，需要根据具体业务情况应用。
```

## （3）开启内存大页支持，设置大页数量。

检查 `hugetlbfs` 文件系统是否挂载，如未挂载，则执行以下命令：

```
Mount -t hugetlbfs hugetlbfs /dev/hugepages
```

Centos7 默认的内存页面大小为 4KB，可以分出 2G 内存先设置 1024 个 2M 大小的内存大页面，执行以下命令完成设置：

```
sysctl vm.nr_hugepages=1024
```

```
cat /proc/meminfo # 查看大页分配情况
```

## （4）启动内存同页合并（KSM）服务。

KSM 允许内核在多个进程之间共享完全相同的内存页，可以减少虚拟机实际的内存使用量。Centos7 最小安装默认没有启动，执行以下命令启动：

```
echo 1 >/sys/kernel/mm/ksm/run
```

(5) 开放修改进程调度策略、优先级的权限。

在内核启动参数加上 "cgroup\_disable=cpu,cpuset,memory" 停止 cgroup 对这几项资源的管理,在虚拟机性能优化时可以无限制动态调整进程调度策略和优先级。

经过上述的改进,使 Linux 内核的资源管理与调度以 KVM 虚拟化为中心,形成了可用于生产环境的 VMM 管理软件,完成了 KVM 虚拟平台的构建。

在 KVM 虚拟平台,通过 QEMU 命令或 Libvirt 命令创建和启动虚拟机,执行命令如下:

qemu-img create -f qcow2 win2008-test3.img 100G # 创建 100G 容量的磁盘文件。

```
kvm -enable-kvm -m 3072 -hda win2008-test3.img -smp 2 \
-name guest=win2008-test3 -boot d -vnc :0 --usbdevice tablet
```

或者用 virsh create win2008-test3.xml 来启动虚拟机, win2008-test3.xml 配置如下:

```
<domain type='kvm'>
  <name>win2008-test3</name>
  <memory>3145728</memory>
  <currentMemory>3145728</currentMemory>
  <vcpu>2</vcpu>
  <os>
    <type arch='x86_64' machine='pc'>hvm</type>
    <boot dev='hd' />
  </os>
  <devices>
    <emulator>/usr/bin/kvm</emulator>
    <disk type='file' device='disk'>
      <driver type='qcow2' cache='writeback' />
      <source file='/home/win2008-test3.img' />
      <target dev='vda' bus='virtio' />
```

```
</disk>

<interface type='bridge'>
  <source bridge='br0'/>
  <model type='virtio'/>
</interface>

<graphics type='vnc' port='-1' keymap='en-us'/>
<input type='mouse' bus='ps2' />
<input type='tablet' bus='usb'/>

</devices>

<features><acpi/></features>

</domain>
```

Libvirt 将调用 `/usr/bin/kvm` 来启动 QEMU 进程，完成虚拟机创建和启动。该虚拟机启动后可用 vnc 连接。

## 2.4 本章小结

本章从 KVM 虚拟化架构出发，介绍了完全虚拟化、半虚拟化、硬件辅助虚拟化在 KVM 架构中的应用，对性能优化涉及的领域、方法和相关技术进行了总结。然后，采用开源 Linux 作为 VMM，在具体的 Centos7 系统上编译最新内核，修改和调整内核中影响 KVM 性能的功能模块，安装最新版本 I/O 模拟软件 QEMU 和虚拟机管理软件 libvirt，构建了经过环境优化的 KVM 虚拟化平台<sup>[21]</sup>。

该平台充分体现了节能减排理念，以低成本实现物理资源高利用率，可成为 IT 从业者的测试或工作平台，也可成为中小企业数据中心的计算资源生产平台。完成了 KVM 虚拟化平台环节的性能优化。

### 第 3 章 KVM 虚拟化实现原理与机制

处理器虚拟化是 KVM 中最核心的部分，深入理解其实现原理和运行机制，将有助于形成正确的性能改进思路和实施方向。本章首先对 Intel x86 服务器上的硬件辅助完全虚拟化（Intel VT-x 技术）的实现原理进行研究，再分析 Linux 内核中 KVM 关于 CPU VT-x 虚拟化实现的源代码，研究 KVM 虚拟化实现原理和运行机制。然后，结合虚拟机运行模式，从虚拟机中运行敏感指令使 VCPU 触发异常陷入（VM-Exit）的执行路径出发，分析 VCPU 陷入-操作模式切换-重新进入虚拟机<sup>[22]</sup>（VM-Entry）运行的处理机制和影响虚拟机性能的关键环节，提出基于处理器角度的 KVM 虚拟机性能优化思路。最后，设计和实现了一套 VCPU 陷入事件的捕获方案，对事件信息进行统计，验证了改进思路的可行性。

#### 3.1 Intel VT-x 辅助 CPU 虚拟化实现技术

KVM 虚拟化的中心是处理器虚拟化，它必须与 CPU 厂商的硬件辅助虚拟化技术结合，在虚拟机运行控制中能触发硬件中断，即执行敏感指令后能够通过硬件辅助技术，通知硬件引发异常陷入，并且，在 KVM 进行模拟后再次运行虚拟机不需要指令翻译，而是由硬件直接还原中断现场进入虚拟机运行。Intel VT-x 就对 KVM 提供了这种实现技术。

当程序所执行的指令涉及访问系统硬件资源或调用操作系统内核功能时，该指令属于特权指令<sup>[23]</sup>，如各类系统调用、访问硬件驱动、输入输出等。敏感指令<sup>[2]</sup>是指在虚拟机系统中的特权指令，从虚拟机角度看和物理机系统中有一样的指令系统，为了区别而定义的名称。

虚拟化的原理就是虚拟机执行敏感指令后像物理机执行了特权指令一样，可以引发中断（也称为异常），由操作系统的相应中断处理程序处理，在虚拟机中称为陷入（对应物理机的中断，同样的为区别而命名，实际就是异常或中断），相应的陷入处理程序就由 VMM 提供。此时如果是软件虚拟化方案，则采取模拟敏感指令，通过指令翻译或调用模拟设备驱动程序接口后，在物理 CPU 上执行，而如果是硬件辅助虚拟化方案，则直接在 CPU 上执行。经过处理的数据或结果再次返回虚拟机，两种方案将重复上面的处理模式，软件方案模拟结果后进入虚拟机，硬件辅助切换状态后直接进入虚拟机<sup>[22]</sup>。

从虚拟化原理出发，KVM 在 VT-x 硬件辅助技术下是否可虚拟化，由可虚

拟化标准判断，这个标准是当虚拟机执行任意敏感指令时，系统都能引发陷入而退出虚拟机状态，进入 VMM 中。这是一个充要条件，如果不符合这个标准，则不是可虚拟化的，称存在“虚拟化漏洞”<sup>[1]</sup>。

虚拟化原理架构上比较简单，其中，处理器的虚拟化是关键，Intel VT-x 就是通过硬件中增加虚拟化所需的特殊寄存器实现了这种机制。VT-x 技术规定 CPU 分成两种运行模式，一种是根操作模式，另一种是非根操作模式，统称为 VMX 操作模式<sup>[1]</sup>。宿主机操作系统（内核、VMM 及应用程序）运行时 CPU 处于根操作模式（VMX Root Operation），虚拟机运行时 CPU 处于非根操作模式（VMX Non-Root Operation）。在非根模式下，CPU 执行所有敏感指令时都将触发异常，指令行为都被重新定义，使它们通过“陷入再模拟”的方式来处理。

CPU 非根模式和根模式都有 0 到 3 四个特权级<sup>[1]</sup>，操作系统运行在各自的 0 特权级，应用程序运行在各自的 3 特权级。VT-x 技术中，虚拟机操作系统和其上的应用程序都运行在非根模式下。当虚拟机应用程序执行敏感指令，访问虚拟机视角下的系统资源时，由虚拟机操作系统处理，因为特权级的限制，此时虚拟机操作系统向 VMM 申请处理用户请求，所以必须引起中断，“陷入”到 VMM 中，这个过程就称为 VM-Exit。接下来，CPU 退出虚拟机代码的执行，切换到根模式，进入到宿主机操作系统的 VMM 中，做相应处理后由 VMM 发起，将 CPU 切换回非根模式继续虚拟机代码的执行，称为 VM-Entry。VT-x 技术通过硬件辅助完成上面的处理过程，其 CPU 虚拟化实现原理如图 3-1 所示：

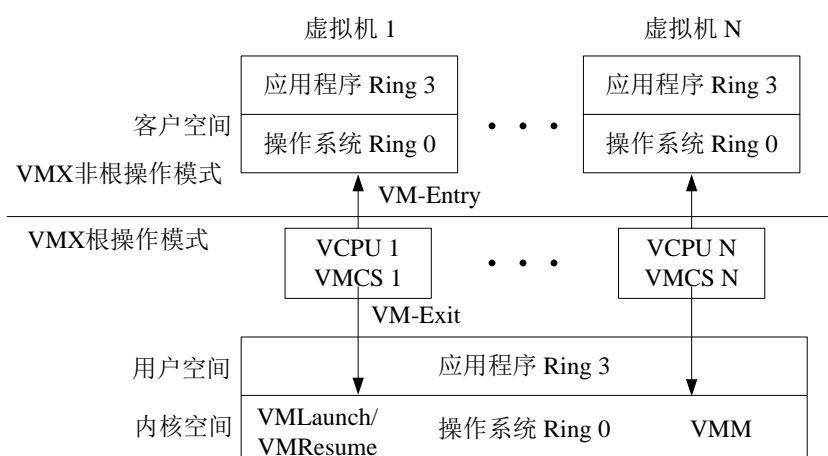


图 3-1 VT-x 技术 CPU 虚拟化实现原理

为实现上述过程，VT-x 从硬件上定义了虚拟机控制结构 VMCS<sup>[2]</sup>，主要保存 CPU 在两种模式下特权寄存器的值，当 VMM 控制虚拟机发生 VM-Exit 和



VM-Entry 时, CPU 通过 VMREAD、VMWRITE、VMCLEAR 等命令来更新相关状态, 与 VMM 配合实现虚拟化的控制。而 VMLaunch 在第一次进入虚拟机时执行, VMResume 在重新进入虚拟机时执行。

VMCS 和 VCPU 是两个描述处理器信息的结构体, VMCS 与硬件紧密联系, 主要存放各类寄存器信息, 与物理 CPU 的寄存器对应, 可通过特殊指令直接读写, VCPU 中包含 VMCS 结构体变量, 从而与物理 CPU 建立联系, 并且定义了与虚拟机相关的各类逻辑寄存器。如 CS 代码段寄存器、DS 数据段寄存器、SS 堆栈寄存器、EFLAGS 标志寄存器、MSR 特殊用途寄存器、CR0-CR8 控制寄存器, 其中 CR3 用于页目录基地址寄存器, 还有 ES、FS、GS 等程序可自由使用的段寄存器。物理 CPU 的寄存器数量有限, 程序运行时通过将 VCPU 中的逻辑寄存器指向物理寄存器的方式在 CPU 上执行。VCPU 还包括 MMU 结构体, 记录了内存相关信息, 除与硬件相关的部分之外, 还包括运行的状态信息、虚拟 LAPIC (Local Advanced Programmable Interrupt Controller)、VCPU 标识等, 可由 VMM 控制的信息。VMM 创建虚拟机, 首先要创建 VCPU, 整个虚拟机的运行就是 VMM 调度不同的 VCPU 运行虚拟机代码的过程。X86\_64 平台和传统的 IA-32 模式下 CPU 指令集能使用的物理寄存器组如表 3-1 所示:

表 3-1 x86\_64 平台及 IA-32 模式寄存器组<sup>[1]</sup>

寄存器	64 位模式			传统的 IA-32 环境		
	名称	数量	大小 (位)	名称	数量	大小 (位)
通用寄存器	RAX、RBX、 RCX、RDX、RBP、 RSI、RDI、RSP、 R8-R15	16	64	EAX、EBX、ECX、 EDX、EBP、ESI、EDI、 ESP	8	32
指令寄存器	RIP	1	64	EIP	1	32
标志寄存器	EFLAGS	1	32	EFLAGS	1	32
流 SIMD 扩展 (SSE) 寄存器	XMM0-XMM15	16	128	XMM0-XMM7	8	128

由此可见, 处理器虚拟化机制的实现过程是 VMM 创建 VCPU 结构体, 装载虚拟机程序运行信息到各虚拟寄存器, 再通过 VMCS 载入物理 CPU 寄存器,

最后运行程序。在这个虚拟运行环境中，每个 VCPU 对应一个 VMCS，再对应一个物理 CPU。如果是物理系统运行环境则直接由操作系统调度程序，直接载入物理 CPU 执行。

不论哪种运行环境，程序执行时都存在异常中断，需要进行上下文环境的切换。在虚拟机中，保存和恢复现场分为两个部分，由硬件读写控制 VCPU 中记录的 VMCS 信息，由 VMM 保存和恢复 VCPU 的其他信息。VCPU 是连接 VMM 与虚拟机，虚拟机连接物理 CPU 的关键信息结构。每次 VCPU 的信息切换都会产生性能开销，VMM 通常采用进程切换的思路，使用 Lazy Save/Restore 的方法进行优化。其基本思想是尽量不对寄存器进行保存与恢复操作，直到必须要这么做时才进行操作。

在处理器虚拟化中上下文切换次数过多将带来较大的性能开销，是虚拟机性能优化的关键。由于业务应用场景不同，虚拟机的这种切换的频率和引发切换的起因也不同，一个计算型虚拟机和一个输入输出型虚拟机，在相同 VCPU 个数下表现的性能差别很大，VT-x 硬件辅助虚拟化只从硬件处理机制上，面向所有虚拟机减少了以前用软件进行切换时的性能开销，是一个里程碑式的性能提升，但在每个具体虚拟机上，VMM 还可以针对具体的虚拟机类型进一步的改进性能。

### 3.2 KVM 虚拟化实现原理

KVM 称为基于内核的虚拟机，在虚拟化技术中充当 VMM 的角色，必须与硬件虚拟化结合。本身只实现了 CPU、内存、设备直接访问的虚拟化。对比标准的 VMM，KVM 的功能不完整，还需要内核的其他模块支持，共同管理和调度整个系统的资源。因此，KVM 虚拟化的 VMM 应该是内核与 KVM 模块的组合。另外，KVM 没有实现输入输出模拟，这部分工作交给了 QEMU 软件。一个 KVM 虚拟机运行在 VMM 环境控制下，由三部分组成：

- (1) I/O 及其他虚拟设备。
- (2) QEMU 创建的 VCPU。
- (3) VCPU 运行线程。

第一部分由 QEMU 模拟和管理，后两个部分由 KVM 管理和控制，VCPU 线程承载虚拟机代码的执行，这一环节的优化是提高资源使用效率的关键。KVM 虚拟机组成结构如图 3-2 所示：

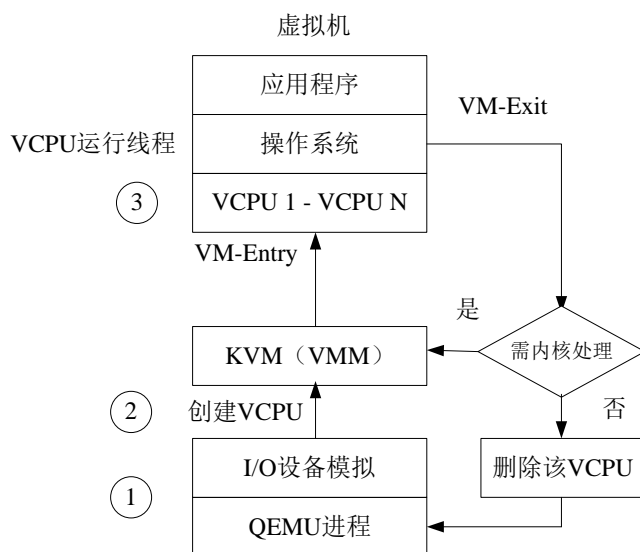


图 3-2 KVM 虚拟机组成结构

KVM 虚拟化实际就是硬件虚拟化技术的驱动程序的应用，实现了进程从用户空间到内核空间再对处理器、内存等硬件设备的访问。在 Intel VT-x 硬件辅助技术支持下，其实现原理包括三个部分，一是 KVM 环境的初始化，二是用户空间的 QEMU 进程如何调用内核空间的 KVM 模块，创建并启动虚拟机，三是 KVM 虚拟机运行模式，即 KVM 如何控制虚拟机的运行。

### 3.2.1 KVM 环境初始化

KVM 与内核一同发布，其源代码版本与内核一致，本文分析的是 v4.17.8 版本。KVM 源代码存放在内核代码树根目录下的 virt 目录和与具体平台相关的 arch/x86/kvm 目录，编译后将生成 kvm.ko、kvm-intel.ko、kvm-amd.ko 三个主要模块，存放在 /lib/modules/4.17.8/kernel/arch/x86/kvm/ 目录下<sup>[24]</sup>。默认 KVM 模块已编译进内核，因此，KVM 可随系统启动自动加载，也可以作为内核普通模块通过以下命令手动加载：

```
modprobe kvm
```

```
modprobe kvm-intel 或者 modprobe kvm-amd
```

加载后将打开 CPU 的虚拟化模式，通过 module\_init() 函数，执行 VMXON 指令，将 CPU 设置为 VMX 根操作模式，再调用 kvm\_main.c 中的 kvm\_init() 函数，初始化 KVM 环境所需的内部数据结构。KVM 环境初始化步骤如图 3-3 所示：

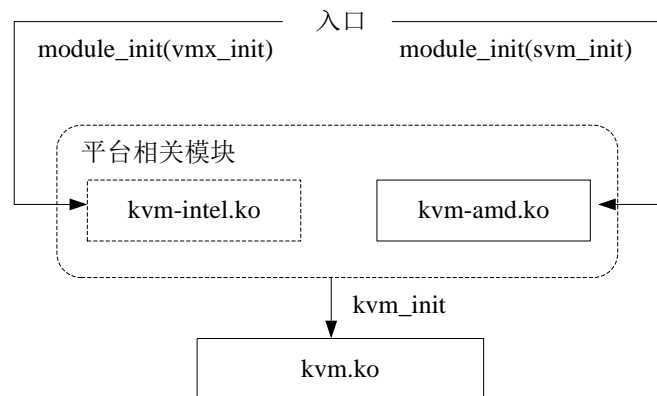


图 3-3 KVM 环境初始化步骤

KVM 环境初始化的过程在 `vmx.c`、`kvm_main.c`、`x86.c` 文件中实现，主要完成以下具体工作：

(1) Intel VT 硬件辅助虚拟化技术加载的是 `kvm-intel.ko` 模块，对应 `vmx.c` 源文件，加载后首先执行该文件中的 KVM 模块初始化函数 `module_init(vmx_init)`，实际对应的是 `vmx_init(void)` 函数。该函数执行 `VMXON`，并调用 `kvm_init()` 函数，其调用代码为：

```
kvm_init(&vmx_x86_ops, sizeof(struct vcpu_vmx),\
        __alignof__(struct vcpu_vmx), THIS_MODULE);
```

第一个参数传入了与硬件相关的 KVM 操作函数结构体变量的地址，通过该变量可调用实际的 KVM 底层硬件操作函数。变量声名为 `struct kvm_x86_ops vmx_x86_ops`，其所有成员都是函数指针，指向 KVM 提供的所有实际函数。

(2) 在 `kvm_init()` 函数中执行了具体的初始化流程，其关键代码如表 3-2 所示：

表 3-2 KVM 初始化函数关键代码

```
int kvm_init()
{
    . . . . .

    //初始化与硬件相关的变量，将 vmx_x86_ops 的地址作为参数传入

    r = kvm_arch_init(opaque);

    r = kvm_arch_hardware_setup(); //调用硬件初始化函数

    //将字符设备、虚拟机文件对像、虚拟 CPU 文件对像的操作函数与 KVM 模块相关联

    kvm_chardev_ops.owner = module;
```

```

    kvm_vm_fops.owner = module;

    kvm_vcpu_fops.owner = module;

    //传 kvm_dev 结构体变量的地址到注册函数，注册 kvm 字符设备，即/dev/kvm 设备

    r = misc_register(&kvm_dev);

    r = kvm_init_debug(); //初始化调试环境

    . . . . .

}

```

上表中 `kvm_dev` 变量定义了一个字符杂项设备，主设备号默认固定为 10，设备名称为 `kvm`，从下面表 3-3 的声明中可知指定 `kvm_chardev_ops` 为操作对像。

表 3-3 KVM 字符设备声明代码

```

static struct miscdevice kvm_dev = {

    KVM_MINOR, //次设备号，主设备号已经定为 10

    "kvm", //设备名称

    &kvm_chardev_ops, //字符设备操作结构体变量指针指向 kvm_chardev_ops 变量，

    //该变量定义了字符设备规定的具体操作函数，如 kvm_dev_ioctl() 函数。

};

```

在 `x86.c` 文件中 `kvm_arch_init()` 和 `kvm_arch_hardware_setup()` 函数又进行了一系列初始化，关键代码如表 3-4、表 3-5 所示：

表 3-4 KVM 全局变量初始化代码

```

int kvm_arch_init(void *opaque)

{
    . . . . .

    struct kvm_x86_ops *ops = opaque; //ops 指向了变量 vmx_x86_ops。

    r = kvm_mmu_module_init(); //kvm 中内存管理单元初始化

    kvm_x86_ops = ops; //全局变量 kvm_x86_ops 指向了 vmx_x86_ops 变量

    kvm_timer_init(); //定时器初始化

    kvm_lapic_init(); //本地高级可编程中断控制器初始化

    . . . . .

}

```

表 3-5 KVM 硬件相关初始化代码

```

int kvm_arch_hardware_setup(void)
{
    . . . . .

    r = kvm_x86_ops->hardware_setup(); //调用硬件初始化函数。

    kvm_init_msr_list(); //初始化特殊寄存器

    . . . . .
}

```

从上面这些函数实现的功能可知，在内核加载完 KVM 模块后，首先执行的就是 KVM 环境的初始化，该过程实施了一系列操作：包括初始化 KVM 与硬件相关的数据结构，比如 VMCS、声名全局变量 `kvm_x86_ops`，使它指向了 KVM 函数指针变量 `vmx_x86_ops`，初始化 MMU、Timer 定时器等。

创建了 `/dev/kvm` 字符设备，提供用户空间访问内核空间的接口。当用户空间 QEMU 创建虚拟机时，将通过映射 `/dev/kvm` 文件，分配虚拟机相关的内存空间，调用 `kvm_x86_ops` 操作对象中定义的 `hardware_setup` 函数执行实际的与硬件相关的结构体变量的初始化工作。最后初始化了 `debugfs` 的调试环境。

(3) 结束 KVM 初始化，等待用户空间的程序访问。KVM 环境的初始化流程如图 3-4 所示：

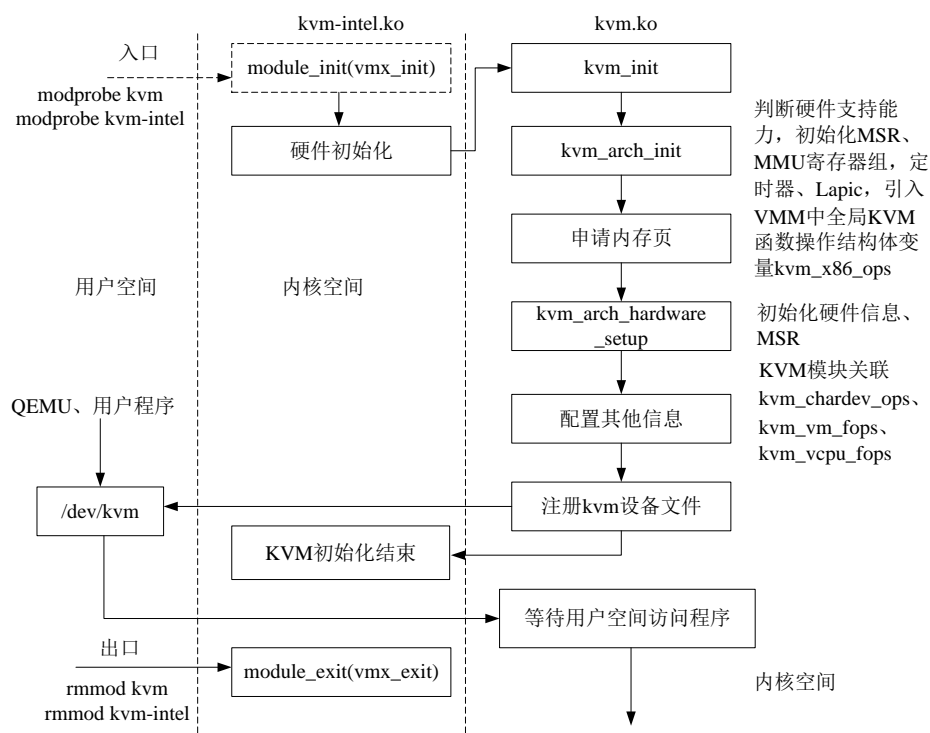


图 3-4 KVM 环境初始化流程

### 3.2.2 KVM 面向用户空间的接口

完成 KVM 环境初始化后, Linux 内核生成了一个杂项字符设备/dev/kvm, 用户空间程序可通过该文件建立与 KVM 模块的联系, 执行 KVM 功能函数来控制虚拟机运行。QEMU 是 Linux 中的用户空间程序, 由它创建、启动虚拟机, 结合 KVM 内核模块, 控制虚拟机的运行和关闭。

在 Linux 中, 用户空间的进程无法直接访问内核空间, CPU 执行指令时都限制在对应的特权级中, 用户程序运行在 Ring 3 特权级, 内核在更高的 Ring 0 级。因此 QEMU 只能通过系统调用来访问内核中 KVM 规定的函数。这些函数是 KVM 面向用户空间的接口。KVM 面向用户空间程序提供接口的方式如图 3-5 所示:

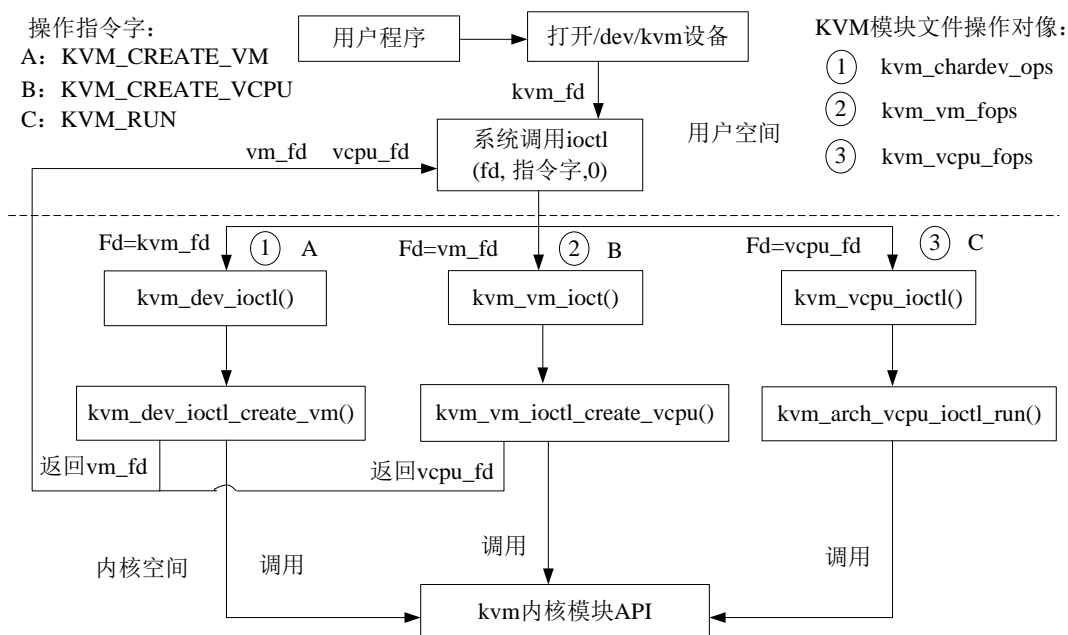


图 3-5 用户空间调用 KVM 接口方式

用户程序通过 `open` 函数打开/dev/kvm 字符设备, 获取 KVM 文件描述符 (`kvm_fd`), 该设备文件注册时关联了名称为 `kvm_chardev_ops` 的文件操作对象, 其中定义了系统调用 `ioctl` 的实际执行函数, 用户程序传入 `kvm_fd`, 操作指令字等参数执行 `ioctl` 调用, 找到实际的实现函数 `kvm_dev_ioctl()`, 在该函数中根据操作指令字又转向具体的功能函数, 每个指令字都对应一个需要 KVM 模块完成的具体功能。以 KVM\_CREATE\_VM 为例, 表示用户程序请求创建一个 KVM 虚拟机文件对象, 将调用内核空间的 `kvm_dev_ioctl_create_vm()` 函数, 进入 KVM

内核模块执行虚拟机文件对像初始化，成功则返回虚拟机文件对像描述符（`vm_fd`）。通过这种映射关系，用户程序实现了从用户空间到内核 KVM 模块功能的调用。该过程实现的关键代码如表 3-6 所示：

表 3-6 用户程序调用 KVM 模块功能函数的关键代码

```
kvm_fd = open("/dev/kvm", O_RDWR | O_CLOEXEC); //获取 kvm 文件描述符。

vm_fd = ioctl(kvm_fd, KVM_CREATE_VM, NULL); //创建 kvm 虚拟机文件描述符。

static struct file_operations kvm_chardev_ops = {

    // 指明用户空间执行系统调用 ioctl 函数时，实际执行的函数名称

    .unlocked_ioctl = kvm_dev_ioctl,

    .compat_ioctl    = kvm_dev_ioctl,

    .llseek          = noop_llseek,

};

//这里定义了 ioctl 函数实际执行的函数

static long kvm_dev_ioctl(struct file *filp, unsigned int ioctl, unsigned long arg)

{
    . . . . .

    switch (ioctl) {

        case KVM_CREATE_VM:

            r = kvm_dev_ioctl_create_vm(arg); //具体调用的 KVM 模块的功能函数（创建虚拟机）

            break;

        . . . . .

    }
}
```

除上表中提到的 `kvm_chardev_ops` 字符设备操作对像外还有虚拟机文件操作对像 `kvm_vm_fops` 和虚拟 CPU 文件操作对像 `kvm_vcpu_fops`，都以相同的映射关系，向用户空间提供访问 KVM 内核模块功能函数的 API 调用。最终，用户空间程序（如 QEMU）可通过这三个对像提供的操作函数实现虚拟机的运行。KVM 虚拟机启动、运行的流程和调用顺序总结如下：

- （1） `int kvm_fd = open("/dev/kvm", O_RDWR | O_CLOEXEC);` // 打开 `/dev/kvm` 设备，获取 `kvm` 文件描述符，建立与 KVM 内核模块的联系。
- （2） `int vm_fd = ioctl(kvm_fd, KVM_CREATE_VM, NULL);` // 创建虚



拟机，获取虚拟机文件描述符。

(3) `int ret = ioctl(vm_fd, KVM_SET_USER_MEMORY_REGION, \&memory_region);` // 为vm\_fd描述的虚拟机分配运行内存，具体内存空间由memory\_region指针确定，其中加载的虚拟机代码如启动程序、操作系统或应用程序等，均由调度程序完成分配。此阶段还可以设置PCI、信号处理、中断设置等运行前的初始化。

(4) `int vcpu_fd = ioctl(vm_fd, KVM_CREATE_VCPU, NULL);` // 为vm\_fd描述的虚拟机创建VCPU结构体变量，获取VCPU文件描述符。

(5) `int ret = ioctl(vcpu_fd, KVM_RUN, NULL);` // 创建VCPU运行线程，将VCPU结构体信息加载到物理CPU，运行虚拟机代码，进入虚拟机非根操作模式。

(6) 捕获虚拟机退出原因，根据KVM模块的异常处理结果进入虚拟机运行循环或传回QEMU进程，根据QEMU的异常处理结果，结束该VCPU线程或重新进入第(5)步在已创建VCPU线程运行循环。

因为/dev/kvm设备关联的文件操作对象中除ioctl外，没有定义write、read等接口函数，用户程序只能使用ioctl系统调用，具体访问KVM的什么功能，由操作指令字来确定。可以将指令字当成访问KVM所提供的用户控件API的说明。ioctl函数是面向用户空间程序的特殊接口，通过向ioctl发送相应的操作指令字实现对KVM的所有操作，完成对虚拟机的所有控制。

操作指令字的定义包含在include/uapi/linux/kvm.h头文件中，值为一个数字。按在不同KVM文件操作对象中使用，可划分为三种类型。第一种在kvm\_chardev\_ops对象中使用，属于系统指令字，用户空间程序通过这些指令字在KVM模块中配置或返回虚拟机相关参数，设置全局参数以及控制虚拟机创建等操作，如指令字KVM\_CREATE\_VM能执行创建虚拟机操作，而指令字KVM\_GET\_API\_VERSION能返回用户空间KVM当前版本信息。第二种在kvm\_vm\_fops对象中使用，属于虚拟机指令字，用户程序通过这些指令字来调用KVM模块内相关功能函数，控制具体的KVM虚拟机执行相关操作，如为虚拟机初始化内存、分配VCPU结构体变量并创建对应VCPU线程等。第三种在kvm\_vcpu\_fops对象中使用，属于VCPU指令字，这类指令字操作VCPU结构体变量，设置其具体参数，多用于上下文切换时的虚拟运行环境的恢复，如中断控制、

读写相关寄存器等。指令字都是通过设备或文件操作对像中 `ioctl` 函数指针指向实际的执行函数传递进去后得以使用的。

### 3.2.3 虚拟机运行模式

用户程序调用接口创建虚拟机、VCPU，这个过程将完成内存分配、虚拟机代码加载、相关寄存器初始化等工作，接下来用户程序可以传入 `KVM_RUN` 指令字，运行指定的 VCPU，虚拟机代码将调度到 VCPU 结构体描述的物理 CPU 上运行。因此，虚拟机的运行模式指的就是 VCPU 的运行模式。KVM 通过控制 VCPU 的运行来实现虚拟化运行机制。在内核 4.17.8 版本中，分析 KVM 模块源代码得到 VCPU 运行的过程是：`ioctl(vcpu_fd, KVM_RUN, NULL)`

→ `kvm_vcpu_ioctl()` → `kvm_arch_vcpu_ioctl_run()`  
 → `vcpu_run()` → `vcpu_enter_guest()` → `kvm_x86_ops->run(vcpu)`  
 → `vmx_vcpu_run()`

上述第一个 `ioctl` 函数由用户空间程序启动，通常是 QEMU 进程。在进入内核 KVM 模块中的执行函数中比较重要的是 `vcpu_run()`、`vcpu_enter_guest()`、`vmx_vcpu_run()`。前两个函数定义在 `x86.c` 源文件中，后一个函数定义在具体平台相关的 `vmx.c` 文件中。其中 `vcpu_run()` 是 VCPU 运行的主循环函数，循环调用 `vcpu_enter_guest()` 函数，后者对应虚拟化实现原理中的进入虚拟机运行（VM-Entry）的过程。当后者返回值小于等于 0 时，退出循环，保存退出现场信息，关闭（结束）该 VCPU 线程，返回到用户空间程序（如 QEMU）中进行处理，否则再次调用函数 `vcpu_enter_guest()`。反映 VCPU 运行过程的关键代码如表 3-7 所示：

表 3-7 KVM 模块虚拟机运行的主循环函数关键代码

```
static int vcpu_run(struct kvm_vcpu *vcpu) {
    int r;

    struct kvm *kvm = vcpu->kvm;

    for (;;) { // 进入虚拟机循环

        if (kvm_vcpu_running(vcpu)) { // 检查是否可运行

            r = vcpu_enter_guest(vcpu); // 设置 VCPU 运行前状态转运行函数，返回 1，
            // 留在内核空间，并再次进入虚拟机，返回 0，退出循环，返回用户空间。

        } else {
```

```

        r = vcpu_block(kvm, vcpu); }

    if (r <= 0)    break; //r 小于等于 0 时退出循环

    . . . . . }

    return r;

}

```

上表中的 `vcpu_enter_guest()` 函数代表进入虚拟机运行，其中再次做一些进入虚拟机运行前的准备工作，然后调用与硬件平台相关的 CPU 运行函数，真正转入到虚拟机中运行。在 `vcpu_enter_guest()` 函数中，当虚拟机运行过程中引发 VCPU 陷入（VM-Exit），退出虚拟机后调用异常处理函数，这是在内核对退出引发的异常的处理，如果处理不了，则返回用户空间处理。其关键代码如表 3-8 所示：

表 3-8 KVM 模块进入虚拟机运行的主函数关键代码

```

static int vcpu_enter_guest(struct kvm_vcpu *vcpu)
{
    . . . . .

    r = kvm_mmu_reload(vcpu); // 内存初始化

    vcpu->mode = IN_GUEST_MODE; // 设置 VCPU 为进入虚拟机模式

    kvm_x86_ops->run(vcpu); // 转到真正的 VCPU 运行函数，进入虚拟机

    // 一个函数执行完了，说明已退出虚拟机的运行，设置 VCPU 为退出虚拟机模式

    vcpu->mode = OUTSIDE_GUEST_MODE;

    // 调用引起退出虚拟机的陷入（异常）处理函数

    r = kvm_x86_ops->handle_exit(vcpu);

    // 返回 <= 0 的值则退出内核回到用户空间，返回 1 则留在内核，等待下一次进入虚拟机运行。

    return r;

}

```

上表中 `kvm_x86_ops` 变量指向 `vmx_x86_ops` 变量，后者的 `run(vcpu)` 函数指向了 `vmx_vcpu_run()` 函数。因此，`vmx_vcpu_run()` 函数是实际进入虚拟机运行其操作系统（代码）的执行函数。该函数中，先存储宿主机当前进程的寄存器信息，再加载虚拟机寄存器，检查需要调用 `vmlaunch` 还是 `vmresume` 指令来进入虚拟

机，之后进入虚拟机模式（VCPU 运行虚拟机代码），当前内核进程在此等待。当发生陷入时，退回并保存虚拟机寄存器（即退出的现场信息），继续执行当前内核进程，读取 VM-Exit 的原因后退出该函数。

通过上述详细分析，得出虚拟机运行模式如图 3-6 所示：

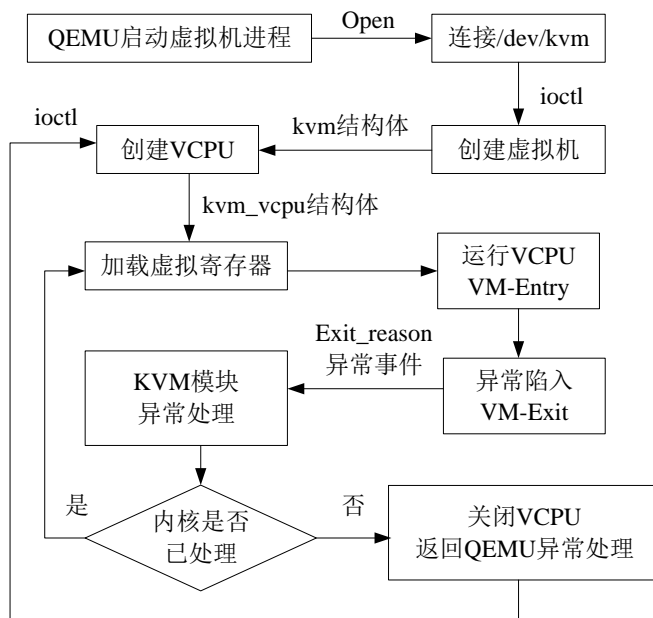


图 3-6 虚拟机运行模式

### 3.2.4 QEMU 创建虚拟机流程

QEMU 作为用户空间程序，在 KVM 虚拟化实现中负责设备模拟、I/O 处理，引导装载虚拟机操作系统，并启动虚拟机。如果不以物理机组成部分角度看待虚拟机组成，QEMU 实际可属于 VMM 在用户空间的延伸，而虚拟机只是它创建的不同线程（VCPU 线程），虚拟机实际包括的只是 VCPU 线程中的客户机操作系统及应用程序<sup>[25]</sup>。

本文从 QEMU 官网（<https://www.qemu.org/>）下载了最新源代码<sup>[18]</sup>，目前已经将 qemu-kvm 合并进新版本中，不再单独发布 QEMU 的 KVM 加速分支版本了。经过 gdb 调试，查看 bt 函数调用栈，可得到 QEMU 启动虚拟机调用 KVM 接口的过程。经过一系列初始化后，最终调用 ioctl() 函数，传入 KVM\_RUN 指令字到接口，实现 VCPU 的创建和运行。具体调用流程如图 3-7 所示：

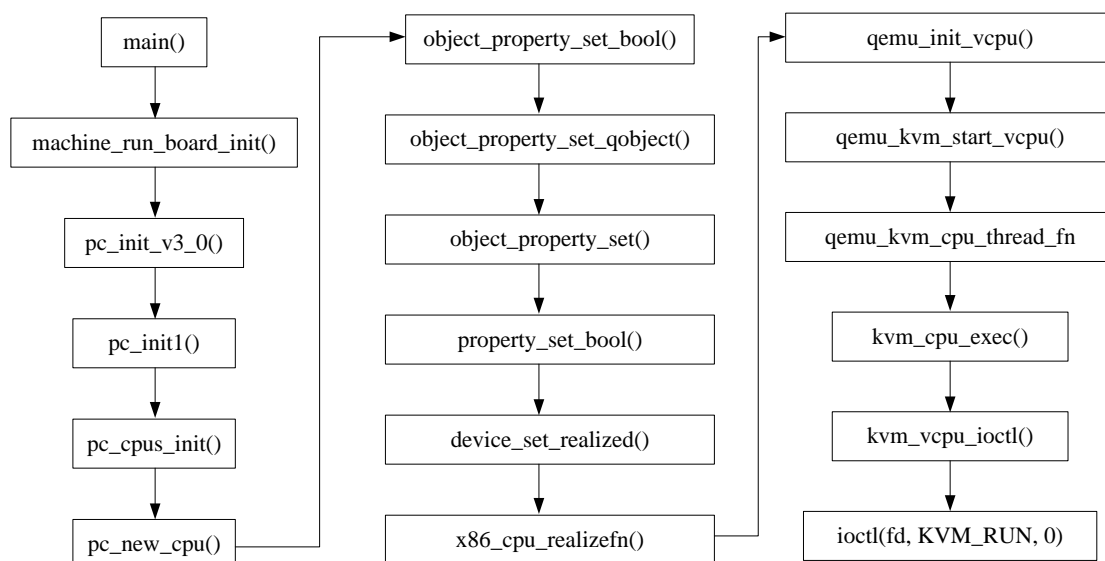


图 3-7 qemu3.0 创建并启动虚拟机流程

当 VCPU 异常陷入（VM-Exit）退出虚拟机时，首先在内核 KVM 模块进行异常处理，处理不了则再返回 QEMU 进程，由其中的 `kvm_cpu_exec()` 函数处理，该函数体现了虚拟机运行模式在用户空间程序 QEMU 中的实现，其关键代码如下表 3-9 所示：

表 3-9 QEMU 进程调用 KVM 模块运行虚拟机主函数关键代码

```

int kvm_cpu_exec(CPUState *cpu) { . . . . .

    do{ run_ret = kvm_vcpu_ioctl(cpu, KVM_RUN, 0); // 创建 VCPU 运行虚拟机 (VM-Entry)

        switch (run->exit_reason) {

            // 分析 VCPU 退出虚拟机的异常，当内核 KVM 模块不能处理时，在这里予以处理

            case KVM_EXIT_IO:

                kvm_handle_io();

                ret = 0; //虚拟机陷入是因 I/O 服务引起时退出当前 VCPU 线程，由 QEMU 模拟处理

                break; . . . . . }

    }while (ret == 0);

    //根据异常处理结果，控制虚拟机关闭该 VCPU 线程，或者继续由该 VCPU 进入虚拟机运行。

    //关闭再创建和继续该 VCPU 运行所产生的上下文切换开销不同，前者大于后者。

    . . . . .

}
  
```

### 3.3 KVM 虚拟机性能优化思路

虚拟化的特点决定了,处理器在运行虚拟机时进行上下文切换的次数必然大于物理机,因此虚拟机的性能理论上必然达不到物理机的水平,但是,当处理器运行速度足够快时,用户对这些性能开销将是无感知的,而多个虚拟机增加了业务处理密度,能明显提高处理器的整体使用率。Intel VT 技术改进了虚拟化中指令的执行方式,由软件翻译模拟提升到直接在 CPU 上执行,即 KVM 模块运行 VCPU 进入虚拟机的切换由硬件完成,极大的改善了虚拟机性能。

通过上述对虚拟机运行模式的研究与分析,可以看到,虚拟化的核心是处理器的“陷入再模拟”机制<sup>[26]</sup>,如果改变这种模式,那么就无法虚拟化。因此,处理器必须在各种环境下进行切换。主要有三种模式切换,包括虚拟机从非根模式陷入到 KVM 内核模块的根模式切换,KVM 模块到用户空间 QEMU 的切换,QEMU 模拟设备 I/O 处理执行系统调用到内核空间的切换。每一次切换都涉及上下文保存和恢复,对执行性能造成一定开销。第一种属于内核到虚拟机之间的切换,由硬件辅助完成,开销较小,对性能影响不大。其他两种由程序完成切换,当切换次数过于频繁时会带来较大的开销。因此,KVM 虚拟机性能优化除环境优化外,还应该从 VCPU 的上下文切换方面提出优化方法和思路。结合之前的研究与分析,本文提出以下几种性能优化思路:

(1) 在 CPU 的三个切换模式的影响中,虚拟机与 KVM 内核模块的切换有硬件支持,其切换由硬件实现,速度很快,对虚拟机性能影响不大。而 KVM 模块到用户空间 QEMU 之间的切换,由软件程序完成,系统开销较大。因此,分析虚拟机异常陷入的类型,修改对 VM-Exit 异常的处理方式,通过控制两个模式的切换频率来达到优化虚拟机性能的目的。一方面将虚拟机的陷入异常直接在 KVM 内核模块中处理,避免切换到用户空间的 QEMU 处理产生较大的开销,另一方面,也可将不常出现的异常交给 QEMU 处理,减轻 KVM 内核模块的处理任务,减少 KVM 虚拟机的系统开销<sup>[27]</sup>。

(2) 针对 CPU 的第三个切换模式,主要是虚拟机因 I/O 操作需求,产生陷入而退出到 KVM 内核模块,因内核 KVM 模块不对 I/O 访问请求引起的异常做实质性处理,所以再退出到用户空间的 QEMU 进程处理<sup>[28]</sup>,即 QEMU 在 I/O 处理时执行系统调用到内核空间的切换。如果 QEMU 与内核进行 I/O 数据交换时不使用单次处理模式,而采取 I/O 缓冲队列方式,在一次切换中完成多个 I/O

处理,那么肯定能显著提高 I/O 处理效率。基于这个思想,可以在 KVM 虚拟机中使用半虚拟化驱动来实现该虚拟机的性能优化目标。目前, virtio 设备驱动框架是一个很好的选择。在主要的 I/O 设备磁盘、网卡访问的驱动程序中, Linux、QEMU 中都有实现相应的接口<sup>[29]</sup>。

(3) 在第一种优化思路的基础上,不修改异常处理方式,不人为改变系统的切换规律,而是对异常事件进行统计,将每个虚拟机的切换信息记录下来,并传递给用户空间的监控程序,由程序或平台管理员分析切换数据,判断虚拟机属于计算密集型、输入/输出密集型、混合型等中的哪一种。再针对具体虚拟机实施相应的性能优化。

以上性能优化思路后两种的实施意义较大,第一种思路修改了 KVM 虚拟机现有异常的处理流程,不但需要改动 KVM 模块,还对 QEMU 的处理范围做了调整,将多个环节的功能修改应用在所有虚拟机上的稳定性无法保证。因此,本文主要使用后两种思路开展对虚拟机的性能研究与改进。

### 3.4 VCPU 陷入事件的捕获方法与信息统计

根据虚拟机运行模式特点,VCPU 的陷入是三种上下文切换模式的起因,成为影响虚拟机执行效率的主要因素。虚拟机随着运行的业务不同,将产生大量的各种类型的 VCPU 陷入事件。每一次陷入必然引起一次或多次上下文切换。造成的性能开销,目前 KVM 和 QEMU 都没有方法对其进行统计,仅实现了最简洁的异常捕获与处理,而虚拟机的所有者也只能通过实际使用来感知道系统性能状态。如果设计一种方法,能统计这些陷入事件的相关信息,比如,三种切换模式已切换的次数,在某一时段发生的频率,以及几种主要陷入事件的次数或频率等,并提供查询的途径,那么,上节中所述的第三种优化思路就具有可实施性。因此,本文将通过修改 KVM 模块中异常处理部分的相关代码,设计对陷入事件基本信息进行捕获和简单统计的方法,研究并验证第三种优化思路的可行性。

在异常信息统计的选择结点上,可以根据虚拟机陷入后异常处理过程特点来考虑。一是在 KVM 内核模块异常处理入口调用统计函数,另一种方法是在 QEMU 模块的异常处理程序中调用统计函数,具体实现需要根据虚拟机实际用途来确定。在本文的实验中,遵循了不轻易修改 KVM 虚拟机运行模式的任何业务逻辑的原则,使统计模块与 KVM 或 QEMU 模块之间的关系满足高内聚低耦合标准。因为,虚拟机陷入内核退出时先由 KVM 内核模块定义好的各类异常处

理程序处理，无法处理的再退出当前 VCPU 的运行，返回 QEMU 程序处理。所以，本文选择直接在 KVM 内核模块进行统计。

首先，设计一个用于统计陷入事件信息的结构体（KvmCpuExceptionList），作为 KVM 模块的全局变量。当虚拟机触发异常后，因为有 VT-x 技术硬件支持，处理器将陷入 Linux 内核完成从 VMX 非根操作模式到 VMX 根操作模式的切换，并把退出原因存于 kvm\_run 结构体变量中，通过 kvm\_vcpu 变量传入 KVM 异常解析处理函数，在其中就退出原因作出相应处理，如虚拟机代码缺页中断、执行了系统调用等。而此时，可加入对应异常事件的统计函数。

然后，在具体的异常处理程序中更新统计信息。本文主要统计了 I/O 和虚拟机主要异常处理发生的切换次数。在进入虚拟机的主循环函数中，根据 KVM 模块的异常处理结果来统计第一、第二种上下文切换的次数。

最后，将统计信息输出到用户空间，或提供用户查询的接口。通过在 KVM 内核模块初始化时创建 proc 文件系统的内存文件(kvminfo)，将 proc 文件操作的相关函数指定为 seq 序列文件类型的函数，重新实现 open 函数<sup>[30]</sup>，并在其中显示统计信息，用户通过 cat /proc/kvminfo 即可查看内核传出的数据。KVM 虚拟机异常信息统计方案流程如图 3-8 所示：

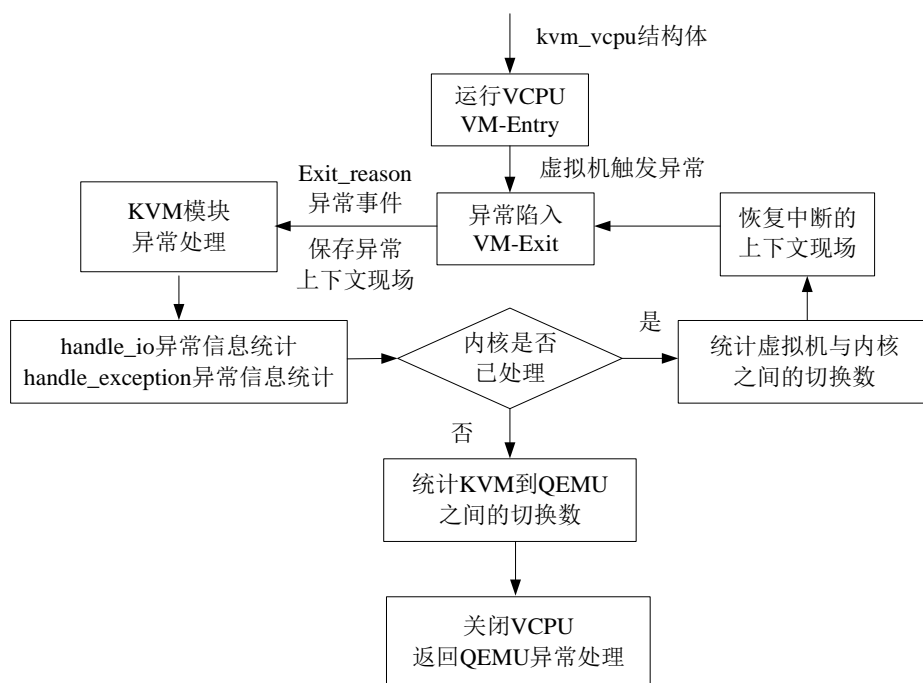


图 3-8 KVM 虚拟机异常信息统计方案流程图

具体实施可以分为三步：



(1) 在 x86.c 源文件中增加程序代码, 内容为统计陷入事件信息的结构体 (KvmCpuExceptionList) 和具体的统计函数。其关键代码如表 3-10 所示:

表 3-10 统计陷入事件信息结构体的关键代码

```
struct KvmCpuExceptionInfo { //虚拟机中 vcpu 陷入事件相关统计信息结构体, 用于统计陷入次数 (即上下文切换次数) . . . . .

    unsigned long handle_io_num; //记录虚拟机申请 I/O 服务引起 VCPU 陷入 (中断) 的数量
    //记录虚拟机发生主要异常引起 VCPU 陷入 (中断) 的数量

    unsigned long handle_exception_num;

    //记录第一种切换的次数, 即虚拟机从非根模式陷入到 KVM 内核模块的根模式切换

    unsigned long kvm_vm_switch_num;

    //记录第二种切换的次数, 即 KVM 模块到用户空间 QEMU 的切换

    unsigned long kvm_qemu_switch_num;

    //标识该虚拟机信息是否可用, 0 表时无效, 即可以装入新的虚拟机信息, 装入后设置为 1。

    int flag;

};

//定义虚拟机 VCPU 例外信息统计结构体

struct KvmCpuExceptionList {

    int total; //当前在统计的虚拟机数量

    // 虚拟机的 VCPU 陷入事件相关信息的统计数组, 限定只统计 100 个虚拟机的数据

    struct KvmCpuExceptionInfo kei[100];

};
```

统计函数根据 stype 参数指定的 VCPU 陷入类型进行数量统计, 其关键代码如表 3-11 所示:

表 3-11 统计陷入事件信息函数的关键代码

```
void handleExceptionInfo(struct kvm_vcpu *vcpu, struct KvmCpuExceptionList * k, \
    char *stype) { . . . . .

    for(i=0; i<n; i++) { //查找统计信息中与该虚拟机进程 id 对应的记录

        if(userspace_pid==k->kei[i].userspace_pid) {

            j=i; //找到该 VCPU 对应的虚拟机 pid 的已有统计信息所在的记录索引号
```

```

        b=true; break;

    }

} . . . . .

if(strcmp(stype,"handle_io")==0) { //记录陷入原因是申请 I/O 服务时的相关数据,

    k->kei[j].handle_io_num++; //即 I/O 申请引发异常产生的上下文切换次数递加 1

}

else if(strcmp(stype,"kvm_vm_switch")==0) {

    //记录第二种切换的次数, 即 KVM 模块到用户空间 QEMU 的切换

    k->kei[j].kvm_vm_switch_num++;

}

else if(strcmp(stype,"kvm_qemu_switch")==0) {

    //记录第一种切换的次数, 即虚拟机从非根模式陷入到 KVM 内核模块的根模式切换

    k->kei[j].kvm_qemu_switch_num++;

} . . . . .

}

EXPORT_SYMBOL_GPL(handleExceptionInfo); //引出该函数在其他模块使用

```

## (2) 确定放置统计函数的具体位置。

根据虚拟机运行模式的规律可知, 在 KVM 内核模块接收到用户程序 QEMU 创建虚拟机的系统调用 `ioctl` 后, 将依次执行函数的流程如下:

```

→ kvm_vcpu_ioctl() → kvm_arch_vcpu_ioctl_run()
→ vcpu_run() → vcpu_enter_guest() → [kvm_x86_ops->run(vcpu)]
→ vmx_vcpu_run()

```

其中, 函数 `vcpu_run()` 将循环调用进入虚拟机函数 `vcpu_enter_guest()`, 每循环一次判断返回值是否小于等于 0, 如果是则退出循环结束该 VCPU 的进入虚拟机函数, 大于 0 时, 再次调用进入虚拟机函数。在 `vcpu_enter_guest()` 函数中做进入虚拟机之前的最后一次准备工作, 然后调用 `kvm_x86_ops` 函数结构体变量的 `run(vcpu)` 变量, 而 `run(vcpu)` 是一个函数指针变量, 实际指向了 `vmx_vcpu_run()` 函数, 而后者是与 Intel VT-x 技术相关的平台执行函数, 会使 CPU 从 VMX 内核根模式切换到 VMX 虚拟机非根模式, 根据 VCPU 结构体变量内容装载 VMCS

结构体变量，然后，调用一段汇编代码执行 **VMLaunch** 或 **VMResume** 进入虚拟机代码运行。

当虚拟机操作系统因为异常触发退出时，在虚拟机操作系统保存相关中断状态，退出虚拟机陷入到内核，再次发生 **CPU VMX** 的模式切换，并继续执行 **vmx\_vcpu\_run()**函数后面的程序，后面主要是记录退出原因。

之后再返回到 **vcpu\_enter\_guest()**函数继续执行后面的程序，其中将调用 **r = kvm\_x86\_ops->handle\_exit(vcpu)**语句，该函数是 **KVM** 内核模块捕获到虚拟机陷入异常后，进行异常处理的主函数入口，在其中将根据退出原因调用预先定义的处理程序，处理程序返回的结果保存在变量 **r** 中，作为函数的返回值，传递回调用函数 **vcpu\_run()**，然后判断是否再次调用进入虚拟机函数。因此，可将统计函数插入到 **vcpu\_enter\_guest()** 函数中异常处理入口函数的下面，根据 **handle\_exit(vcpu)**函数的处理结果，来调用第一、第二种上下文切换模式的统计函数。其核心代码如下表 3-12 所示：

表 3-12 KVM 模块中放入第一、二种切换的统计函数的关键代码

```
static int vcpu_enter_guest(struct kvm_vcpu *vcpu)
{
    . . . . .

    kvm_x86_ops->run(vcpu); // 转到真正的 VCPU 运行函数，进入虚拟机

    //上一个函数执行完了，说明已退出虚拟机的运行，设置 VCPU 为退出虚拟机模式

    vcpu->mode = OUTSIDE_GUEST_MODE;

    r = kvm_x86_ops->handle_exit(vcpu); //调用引起退出虚拟机的陷入（异常）处理函数。根据异常处理后的返回值，决定是否退出循环，<=0 时，退出循环结束该 VCPU 的运行，否则继续进入虚拟机。当<=0 时，对应第二、第三种切换模式，即 KVM 模块到用户空间 QEMU 的切换和 QEMU 模拟设备 I/O 处理执行系统调用到内核空间的切换。当大于 0 时，对应第一种切换模式，虚拟机从非根模式陷入到 KVM 内核模块的根模式切换。

    if (r<=0) { //记录第二种切换模式发生的次数，即 KVM 模块到用户空间 QEMU 的切换次数

        handleExceptionInfo(vcpu, &vcpu_exception_info, "kvm_qemu_switch_num");

    } else { //记录第一种切换模式发生的次数，即虚拟机从非根模式陷入到 KVM 内核模块的根模式切换的次数

        handleExceptionInfo(vcpu, &vcpu_exception_info, "kvm_vm_switch_num");

    }
}
```

```
return r; //返回<=0 的值则退出内核回到用户空间，返回 1 则留在内核，等待下一次进入虚拟机运行。
```

KVM 模块中根据虚拟机陷入退出的具体原因，已预先定义了 50 种常见的陷入异常的处理函数，由数组 `kvm_vmx_exit_handlers[]` 统一组织声明和指定了每种异常处理程序的函数名。每种退出原因用一个从 0-49 的整型值表示，与 KVM 定义的异常类型一一对应，通过关联到异常处理函数数组下标，实现由退出原因值指向对应的异常处理函数数组的下标，再调用该下标对应的异常处理函数。上述 `handle_exit(vcpu)` 函数是一个函数指针，实际指向了 `vmx_handle_exit()` 函数，其核心代码如表 3-13 所示：

表 3-13 KVM 模块处理虚拟机退出异常的主函数关键代码

```
static int vmx_handle_exit(struct kvm_vcpu *vcpu)
{
    . . . . .

    if (exit_reason < kvm_vmx_max_exit_handlers
        && kvm_vmx_exit_handlers[exit_reason])
        //关联调用 exit_reason 对应的异常处理函数。

        return kvm_vmx_exit_handlers[exit_reason](vcpu);

    else { //当退出原因值大于预先定义的 50 种异常处理函数的下标值时

        vcpu_unimpl(vcpu, "vmx: unexpected exit reason 0x%x\n", exit_reason);

        kvm_queue_exception(vcpu, UD_VECTOR);

        return 1; //对未识别的异常，加入不处理队列，返回 1，表示再次进入虚拟机。

    }
}
```

目前，在 Linux 内核 4.17.8 版本中 KVM 模块已预定义的异常处理函数是 50 种，后续还会根据技术发展，继续增加。异常处理函数数组的定义在 `vmx.c` 源文件中。其中，`handle_exception` 是处理虚拟机主要异常的函数入口，`handle_io` 是处理虚拟机因 I/O 访问产生陷入异常的函数入口。因此，在统计某种原因引发陷入的数量时，可将对应的统计函数插入到这些函数中，本文实验主要统计两种

类型下的异常发生数量。其关键代码如表 3-14 所示：

表 3-14 KVM 模块统计主要异常处理和 I/O 异常处理函数关键代码

```
//虚拟机 VCPU 发生陷入异常，退出虚拟机后的主要异常处理函数

static int handle_exception(struct kvm_vcpu *vcpu){ . . . . .

    //统计主要异常处理执行的次数

    handleExceptionInfo(vcpu,&vcpu_exception_info,"handle_exception");

    return 1;

}

static int handle_io(struct kvm_vcpu *vcpu){ . . . . .

    //统计虚拟机申请 I/O 服务引起的 VCPU 陷入切换次数

    handleExceptionInfo(vcpu,&vcpu_exception_info,"handle_io");

    return kvm_fast_pio(vcpu, size, port, in);

}
```

(3) 将统计信息输出到用户空间。通过在 KVM 内核模块初始化时创建 proc 文件系统的内存文件(kvminfo)，采用 seq\_file 文件读取机制，在读取 /proc/kvminfo 文件时，将统计变量的内容格式化输出到标准输出设备，使用户可查看，获取虚拟机的相关切换次数。也可通过程序分析虚拟机某个时段的活跃度。具体实现的关键代码如表 3-15 所示：

### 声名相关变量

表 3-15 向用户空间输出统计信息变量声名的关键代码

```
char kvm_proc_name[]="kvminfo"; //定义生成的 proc 文件名称

static int showKvmInfo(struct seq_file *m, void *v); //格式化输出，显示统计信息

static int kvminfo_proc_open(struct inode *inode, struct file *file); //打开 proc 文件

static const struct file_operations kvminfo_fops = {

//proc 文件操作的相关函数指定为 seq 序列文件类型的函数，对应的是 seq_file 文件操作机制

    .owner      = THIS_MODULE,

    .open       = kvminfo_proc_open, //定义 open 函数实际指向的函数

    .read       = seq_read,

};
```

## 实现声名中定义的函数

表 3-16 实现向用户空间输出统计信息的关键代码

```
static int kvminfo_proc_open(struct inode *inode, struct file *file) {

    //打开 proc 文件类型的/proc/kvminfo 文件，并关联 showKvmInfo() 函数打开文件的同时调用
    showKvmInfo() 函数，显示统计信息

    return single_open(file, showKvmInfo, NULL);
}

//显示统计信息函数
static int showKvmInfo(struct seq_file *m, void *v){

    struct KvmCpuExceptionList *p=&vcpu_exception_info; //指向统计信息队列变量

    struct KvmCpuExceptionInfo *t; //用于指向存放统计信息的结构体变量。

    char title[6][50]={"序号", "PID", "handle_io_num", "handle_exception_num", \
        "kvm_vm_switch_num", \    //表示第一种切换模式的切换数量

        "kvm_qemu_switch_num"}; //表示第二种切换模式的切换数量

    . . . . .

    for(i=0; i<n; i++){

        t=&p->kei[i];

        //取出统计信息队列中第 i 个统计变量，每个统计变量记录一个虚拟机的相关统计信息。

        if(t->flag==1){ //输出第 i 个统计变量，即第 i 个虚拟机的统计信息

            seq_printf(m, "%3d, %6d, %10lu, %10lu, %10lu, %10lu\n", i+1, \

                t->userspace_pid, t->handle_io_num, t->handle_exception_num, \

                t->kvm_vm_switch_num, t->kvm_qemu_switch_num); }

        }

    return 0;

}
```

在 vmx.c 源代码文件的 vmx\_init(void)函数中插入创建 proc 文件系统对像的程序，生成/proc/kvminfo 文件，该代码如下：

```
kvm_info_proc_entry=proc_create(kvm_proc_name,0x666,NULL,
&kvminfo_fops);
```

在 `vmx_exit(void)` 函数中增加移除模块代码，该代码如下：

`remove_proc_entry(kvm_proc_name, NULL);` // 删除 proc 文件

最后，重新编译 KVM 模块代码并加载 `kvm_intel` 等 KVM 模块到内核中。

实验中，在间隔不大的同一时段连续启动十台虚拟机，运行 5 小时后，对其上下文切换数量的统计情况如图 3-9 所示：

```
[root@localhost home]# virsh list
Id      名称                      状态
-----
1       centos7-test-1            running
2       win2003-200G-test-2      running
3       win2008-test3            running
4       win2008-test4            running
5       centos7-test-5           running
6       centos7-test-6           running
7       win2003-test-7           running
8       win2003-test-8           running
9       win2003-test-9           running
10      win2003-test-10          running

[root@localhost home]# netstat -tunpl
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp    0      0 0.0.0.0:5909            0.0.0.0:*               LISTEN      2767/kvm
tcp    0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      1498/sshd
tcp    0      0 0.0.0.0:5900            0.0.0.0:*               LISTEN      2035/kvm
tcp    0      0 0.0.0.0:5901            0.0.0.0:*               LISTEN      2135/kvm
tcp    0      0 0.0.0.0:5902            0.0.0.0:*               LISTEN      2193/kvm
tcp    0      0 0.0.0.0:5903            0.0.0.0:*               LISTEN      2267/kvm
tcp    0      0 0.0.0.0:5904            0.0.0.0:*               LISTEN      2351/kvm
tcp    0      0 0.0.0.0:5905            0.0.0.0:*               LISTEN      2417/kvm
tcp    0      0 0.0.0.0:5906            0.0.0.0:*               LISTEN      2504/kvm
tcp    0      0 0.0.0.0:5907            0.0.0.0:*               LISTEN      2575/kvm
tcp    0      0 0.0.0.0:5908            0.0.0.0:*               LISTEN      2619/kvm
tcp6   0      0 :::22                   :::*                    LISTEN      1498/sshd

[root@localhost home]# cat /proc/kvminfo
序号  PID      io_num   exception_num  kvm_vm_switch_num  kvm_qemu_switch_num
1     2035     172666   0              8132965            174197
2     2135     19738809 0              19523186           19655793
3     2193     1790029  0              59107343           22772082
4     2267     1497176  0              38492507           22050122
5     2351     230926   0              10697798           251194
6     2417     299097   0              10995347           301210
7     2504     14147604 0              11433987           14172128
8     2575     14418019 0              11269126           14435663
9     2619     13750531 0              10910633           13782007
10    2767     13916885 0              10997062           13946096
```

图 3-9 虚拟机陷入异常切换数量统计

根据上图的统计信息，对各虚拟机性能表现横向比较，进行定量分析。首先，比较 `io_num`、`kvm_vm_switch_num` 和 `kvm_qemu_switch_num` 三个指标的数据，得到每台虚拟机当前时间段内总的上下文切换次数，计算虚拟机与 KVM 内核模块之间的切换百分比。该切换属于第一种切换类型，由硬件实现，性能开销较小，因此，其占百分比越大，性能表现就越好，越接近物理机的性能。

然后，计算 I/O 切换的百分比。该切换属于第二种切换类型中的一种，可由软件模拟、半虚拟化、硬件直接分配三种处理方式中的一种来实现，不同的处理技术，产生的性能开销差异很大。因此，其占百分比越大，性能表现的差异就越大，最终的性能表现取决于采用的 I/O 处理技术。但在此处的横向比较中，百分比越大，性能表现就越差。

最后，根据虚拟机执行的任务来直接比较总切换次数和第二种切换类型的上

下文切换次数。当执行相同任务（如文件服务器或 WEB 服务器）时，切换次数越多的，其性能表现越差，反之则性能越好。在此例实验中，可以看到序号 1、5、6 对应三台 centos7 的虚拟机，序号 3 和 4 对应两台 windows2008 的虚拟机，序号 2 以及 7 到 10 对应五台 windows2003 的虚拟机，启动后正常运行 5 小时。相同操作系统的虚拟机的三项指标为同一数量级。不同操作系统的虚拟机的三项指标差异较大。

上述定量分析得出具体数据如表 3-17 所示：

表 3-17 各虚拟机切换数据定量分析情况

序号	PID	io_num	kvm_vm_switch_num	kvm_qemu_switch_num	总切换次数	虚拟机与 KVM 之间切换的百分比	I/O 切换的百分比	虚拟机操作系统
1	2035	172666	8132965	174197	8307162	0.97903	0.0207	centos7
2	2135	19738809	19523186	19655793	39178979	0.49830	0.5038	win2003
3	2193	1790029	59107343	22772082	81879425	0.72188	0.0218	win2008
4	2267	1497176	38492507	22050122	60542629	0.63579	0.0247	win2008
5	2351	230926	10697798	251194	10948992	0.97705	0.0210	centos7
6	2417	299097	10995347	301210	11296557	0.97333	0.0264	centos7
7	2504	14147604	11433987	14172128	25606115	0.44653	0.5525	win2003
8	2575	14418019	11269126	14435663	25704789	0.43840	0.5609	win2003
9	2619	13750531	10910633	13782007	24692640	0.44185	0.5568	win2003
10	2767	13916885	10997062	13946096	24943158	0.44088	0.5579	win2003

从上表中可得出结论：centos7 虚拟机的第一种切换类型占比为 97%，最接近物理服务器的性能。Windows2008 虚拟机占比为 63%，而 windows2003 虚拟机占比为 44%，因此，centos7 64 位虚拟机性能优于 windows2008 64 位虚拟机，后者又优于 windows2003 32 位虚拟机。Windows2003 虚拟机的 I/O 切换百分比超过了 50%，可以看成是输入/输出密集型虚拟机，要获得高性能体验，就必须优化 I/O 访问性能，可根据承载的具体业务，来决定采用 virtio 半虚拟化 I/O 驱



动、设备直接分配、SR-IOV 中的某一种优化技术。

上述分析是第三种性能优化思路的一个实例，它以虚拟机运行过程中统计上下文切换数量作为分析维度。在该思路下，还可以改变统计的维度（即统计其他切换信息），开展定性定量分析，并以此为基础针对具体虚拟机实施相应的性能优化。

综上所述，本章主要实现了一种简单统计虚拟机运行过程中产生上下文切换的次数，并在第一种优化思路的基础上，不修改异常处理方式，不人为改变系统的切换规律，将每个虚拟机的切换信息记录下来，并传递给用户空间，一方面可以为分析虚拟机性能开销提供具体异常下的切换数据，判断虚拟机的业务类型，从而决定采取的性能改进方案。另一方面，建立了用户程序获取 KVM 模块运行时数据的一种简便方法，为开展虚拟机性能优化研究提供定量和定性分析的数据支持。

### 3.5 本章小结

本章首先研究了 Intel VT-x 硬件辅助技术下 CPU 虚拟化的实现原理，深入理解 CPU 的两种 VMX 操作模式以及虚拟化时物理寄存器、VMCS、VCPU 和 VMM 之间的逻辑关系，明确了在处理器虚拟化中上下文状态切换是影响虚拟机性能的关键因素。然后，参照 KVM 源代码，从 KVM 环境初始化、面向用户空间的接口和虚拟机运行模式三个方面，深入分析 KVM 虚拟化的原理和运行机制，理解 KVM 虚拟机的组成结构。在此基础上，提出了三种基于处理器切换角度的 KVM 虚拟机性能优化思路。

最后，针对第三种性能优化思路，通过修改 KVM 模块中异常处理部分的相关代码，设计和实现了一套 VCPU 陷入事件的捕获方案，对事件基本信息进行简单统计，为用户空间程序分析 VCPU 运行状态提供数据基础，通过实验，研究并验证了第三种优化思路的可行性。

## 第 4 章 KVM 虚拟机调度优化

由 KVM 虚拟机运行模式可知, KVM 虚拟机是从用户空间程序 QEMU 发起创建, QEMU 是虚拟机运行的主进程, 通常根据配置中指定的 VCPU 数来创建对应数量的 VCPU 线程, 并控制这些线程进入虚拟机, 运行相应操作系统和应用程序。我们将一个 QEMU 主进程创建的所有 VCPU 线程统称作该虚拟机进程。Linux 内核对虚拟机所采取的调度机制, 就是普通的 Linux 进程调度机制。同时, 虚拟机进程在 Linux 中获得的优先级和调度策略和 QEMU 主进程一致。

Linux 的进程管理将 QEMU 主进程确定为普通进程, 采用普通分时进程调度策略和完全公平调度算法 CFS(Completely Fair Scheduler)。而占更多处理器计算资源更高优先执行权的实时调度策略只应用于 Linux 的某些系统进程。

KVM 虚拟机在普通应用场景下, Linux 内核对虚拟机进程采用默认进程优先级和完全公平调度算法, 这种状态只能保证虚拟机正常运行, 满足访问量较低的虚拟机运行。因为分时占用 CPU 的特点, 决定了要想获得更快响应、更高性能体验, 就必须提高 CPU 的占用时间。本章通过研究 Linux 的进程调度策略, 以及各策略下使用的调度算法, 结合 KVM 虚拟机运行模式和进程特点, 提出了一种调度优化策略, 可根据实际业务来实时修改虚拟机进程的调度策略和运行优先级, 有区别的调整虚拟机占用 CPU 资源的比例, 达到性能优化目的, 提高物理计算资源利用率, 使用虚拟平台中所有虚拟机的整体性能得到改进。

为使上述优化策略具有可实施性, 本章设计实现了一个进程信息的实时监控程序, 监控虚拟机运行进程在 Linux 系统中的各项资源占用状态<sup>[27]</sup>, 实现虚拟机进程优先级和调度策略的实时修改方法, 使虚拟化平台可灵活的根据各虚拟机的实际业务需求, 优化其对处理器资源的占用时间, 进行个性化设置。

### 4.1 Linux 进程调度策略

进程是程序在 CPU 上的一个执行过程, 其功能业务上的性能高低表现与获得的 CPU 运行时间多少密切相关。Linux 是可抢占式的多用户分时操作系统, 各程序按不同时间段在处理器上运行。为了便于程序调度, 系统将 CPU 时间分成固定大小的时间片, 按策略给每个进程分配规定的时间片数量, 通过程序调度, 在指定 CPU 上运行进程, 达到给定时间片数量的时间服务。而单处理器在任何给定的时刻都只能运行一个进程, 在多处理器的多核环境下, 每个 CPU 逻辑核

和单处理器的调度规律是一致的。如何分配进程的 CPU 运行时间以及何时运行该进程，是由 Linux 系统拟定的一套规则决定，称为进程调度策略。它受多个因素影响，如进程类型、优先级等，负责分配处理器时间和调度进程执行，其策略的具体实现，决定了系统面向用户使用时性能表现的整体印象<sup>[31]</sup>。

Linux 内核支持高运行优先级进程可抢占正在运行的低优先级进程，如果新来进程是可运行状态，在每次时钟节拍时，调度程序将检测各就绪进程的动态优先级，如果其值高于当前运行进程<sup>[32]</sup>，则当前进程被该可运行进程抢占。另外，进程的时间片用完时也将被抢占。

基于上述这些特点，在内核 4.17.8 版本中，Linux 进程调度策略按照不同的进程类型所需要的调度效果，主要实现了四种策略，每个进程使用 CPU 的规则都被设置成其中的一种。进程的调度策略取值范围在内核源文件 `include/uapi/linux/sched.h` 中定义。

四种调度策略管理了普通、实时和批处理三种进程类型的调度，每种策略都有详细的调度规则和实现算法。具体规定如下：

(1) **SCHED\_NORMAL**（普通策略）。该策略定义为普通进程调度策略，取值为 0，用于普通非实时进程的调度，是 Linux 默认的调度策略<sup>[27]</sup>。在内核 2.6.23 版本后，都采用完全公平调度算法(CFS)对普通进程进行调度管理。该算法仅适用普通进程，将进程按优先级分类，每个优先级对应一颗红黑树，所有普通进程按 CFS 约定插入它对应的优先级的红黑树中，使用静态和动态优先级相结合，静态优先级决定分配 CPU 时间的权重比例，会根据已运行时间调整动态优先级，CFS 调度器按动态优先级由高到低选出进程，而在同一优先级又有多个进程需要调度时，将通过红黑树来选出进程，通常选择树中最左边叶子结点记录的进程执行<sup>[27]</sup>。

(2) **SCHED\_FIFO**（先进先出策略）。该策略用于先进先出的实时进程，取值为 1，使用静态优先级。该策略下的进程主要比较优先级来决定是否占用 CPU，在调度时，优先于普通策略调度的进程。不使用时间片，采用先入先出调度算法，可以一直运行下去，只有比它高优先级的实时进程就绪、自我阻塞、显式让出处理器或运行结束时才能被其他进程抢占。

(3) **SCHED\_RR**（时间片轮询策略）。该策略属于实时调度策略<sup>[27]</sup>，取值为 2，与 **SCHED\_FIFO** 相似类，采用静态优先级，调度器不为其计算动态优

优先级。它保证了在同一优先级的实时进程按时间片轮流执行，优先级高的实时进程总是能抢占低优先级的进程。

(4) **SCHED\_BATCH** (批调度策略)。该策略主要应用于批处理类型的普通非实时的进程调度，取值为 3，在 Linux 系统调度中应用较少，一般用于任务型，对响应时间不做要求的非交互式的进程<sup>[27]</sup>。

#### 4.1.1 进程优先级与时间片关系

Linux 进程调度策略都与进程优先级和时间片有一定联系，每种策略的调度算法在应用优先级和时间片时又不相同，正确理解优先级、时间片与调度算法的关系，有助于 KVM 虚拟机进程调度的优化。为后面设计 KVM 虚拟机进程优化方案时采用何种优先级和相应调度策略提供依据。Linux 内核在每个进程描述符中定义了三个优先级变量，分别对应静态优先级(static\_prio, 等于 nice 值加 120)、动态优先级(prio)和实时优先级(rt\_priority)，取整型值，定义在 include/linux/sched.h 文件中。

Linux 采用两种不同的优先级范围，一种用于普通非实时进程，用 nice 表示进程的静态优先级，取值范围是从-20（最高优先级）到 19（最低优先级），默认值为 0。应用上规定，越大的 nice 值优先级越低，其作用在普通进程调度策略中对应进程权重，用来计算调度器实体结构中进程的虚拟运行时间(vruntime)。在实时进程中则用于确定轮询调度策略下的基本时间片的长度，即该类进程的时间片与静态优先级相关<sup>[33]</sup>。

第二种优先级用于实时进程，所取值是从 0 到 99(即从最低到最高优先级)。应用上规定，越大的优先级值其实时优先级越高，这与静态优先级表示正好相反。另外，在 Linux 的静态优先级的另一种表示方法中共用了这个取值空间，用 100（最高优先级）到 139(最低优先级)对应 nice 的-20 到 19 取值，并存储在 static\_prio 变量中。因此，静态优先级的值等于 nice 值加上 120。

在实时进程中的时间片轮询调度策略下，由静态优先级决定实时进程的基本时间片。此时，进程的优先级与所获得的时间片两者的关系可用表 4-1 中的公式确定：

表 4-1 静态优先级与时间片大小的关系公式<sup>[34]</sup>

基本时间片	=	(140-静态优先级) x 20	若静态优先级<120	(4-1)
(单位为 ms)		(140-静态优先级) x 5	若静态优先级>=120	(4-2)

从上面可看出静态优先级越高（其值越小），基本时间片越长。普通进程在 CFS 调度算法下虽然没有直接与时间片有上述联系，但其结果是，高静态优先级的进程能获得比低静态优先级进程更长的 CPU 时间片。同时，静态优先级是影响动态优先级的一个因素。不论是哪种进程调度算法都是先从优先级高的运行队列中选择进程运行，CFS 调度算法是比较进程的动态优先级，而先进先出与轮询调度算法是比较进程的实时优先级。

动态优先级是决定普通进程被 CFS 调度先后顺序的指标，它不能由程序自动设置，初始是等于静态优先级，根据进程运行状态会有变化，这种变化与静态优先级有一定关系，总的来说，当平均睡眠时间越大时在原有静态优先级基础上提高 0 到 5 个数值，反之，如果进程一直运行，则动态优先级会降低 0 到 5 个数值。

#### 4.1.2 普通进程调度算法

Linux 中普通进程调度策略采用的是 CFS 算法，该算法的思想是从高到低遍历动态优先级对应的调度器类，在同一优先级调度器类中选择具有最小虚拟运行时间（vruntime）的进程运行。完全公平调度就体现在 CPU 运行时间按权重比例分配和 vruntime 的计算方法中<sup>[35]</sup>。

一个普通进程在一个调度周期中的 CPU 运行时间的公式如下：

$$\text{进程的 CPU 运行时间} = \text{调度周期} \times \text{进程权重} / \text{所有进程权重之和} \quad (4-3)$$

上式中调度周期在 CFS 调度器中是固定的，默认值为 6ms，值保存在 kernel/sched/fair.c 文件中的 sysctl\_sched\_latency 变量中。进程权重由 40 个 nice 表示的静态优先级一一对应，其值定义在 /kernel/sched/core.c 文件中的 sched\_prio\_to\_weight[40] 数组中。

由进程权重数组取值可知，默认的 nice 值 0 对应权重为 1024。nice 值越小，权重越大，进程可能分配到的 CPU 时间越多。而进程的虚拟运行时间计算公式如下：

$$\begin{aligned} \text{vruntime} &= \text{实际运行时间} \times \text{NICE\_0\_LOAD} / \text{进程权重} \\ &= \text{实际运行时间} \times 1024 / \text{进程权重} \end{aligned} \quad (4-4)$$

NICE\_0\_LOAD = 1024，表示 nice 值为 0 的进程权重，将公式 4-3 代入 4-4 可得：

$$\begin{aligned}
\text{vruntime} &= \text{进程在一个调度周期内的实际运行时间} \times 1024 / \text{进程权重} \\
&= (\text{调度周期} \times \text{进程权重} / \text{所有进程总权重}) \times 1024 / \text{进程权重} \\
&= \text{调度周期} \times 1024 / \text{所有进程总权重} \quad (4-5)
\end{aligned}$$

由公式 4-3、4-4、4-5 可知，所有进程的 `vruntime` 值在一个调度周期内增长幅度一样，体现了可能获得调度的公平性。同时，进程权重越大，实际 CPU 运行时间相同时，`vruntime` 增长的越慢，其值就越小。因此，进程权重越大越容易成为最左叶子结点。而 CFS 调度类在红黑树中总是选择最左叶子结点（`vruntime` 值最小）的进程运行，由此可知，静态优先级越高，其取值越低，进程权重越大，进程越容易被调度器选中运行。

具体的说，CFS 调度算法不再将静态优先级即 `nice` 值直接对应时间片分给进程，而是对应一个权重，不区分优先级但却按权重比例分配 CPU 时间片。执行调度时，从高到低以动态优先级对应的运行队列为标准，找对应红黑树中最左叶子结点对应的进程来运行。`nice` 值越小，静态优先级越大，代表的进程权重越大，表示一个调度周期内获得的 CPU 时间越多。`nice` 值相同，静态优先级相同，进程权重相同，通常得到的运行时间也相同，当有阻塞而使实际运行时间较少时，`vruntime` 值必然比其他同优先级进程要小，从而转为可运行状态后最先得到调度。另外，休眠时间长也将提高动态优先级<sup>[33]</sup>。

因此，`nice` 取值是决定普通进程使用 CPU 时间多少以及时间片轮转实时进程的基本时间片大小的关键值。

#### 4.1.3 实时进程调度算法

该算法比较简单，主要考虑实时优先级，按优先级由高到低遍历调度器类，从中找出优先级最高的进程运行。当进程是属于 `SCHED_FIFO` 调度策略时，不计算时间片，一直运行，直到阻塞、被高优先级的进程抢占或运行结束。当进程是属于 `SCHED_RR` 调度策略时，按时间片运行，每个时钟节拍都递减时间片，用完后等待再循环运行，直到阻塞、被高优先级的进程抢占或运行结束。

在 Linux 系统中，实时进程的优先级都高于普通进程，能得到更多、更及时的 CPU 响应。一般情况下，性能体验更好。为了获得满意的响应体验，用户空间程序可以通过系统调用来改变进程的调度策略和对应优先级。

## 4.2 KVM 虚拟机调度优化方案

KVM 虚拟机进程由 QEMU 主进程创建, 继承主进程的调度策略和相应优先级, Linux 将 QEMU 设置成 `SCHED_NORMAL` 普通进程调度策略, 静态优先级为 120, 对应的 `nice` 值为 0。可以看到虚拟机进程被 Linux 视为与其他应用程序同等的普通进程。没有 CPU 计算资源使用的偏向<sup>[36]</sup>。根据 Linux 进程调度特点, 当更高优先级的进程进入可运行队列时, 将会抢占 CPU 的使用权。也就是说, KVM 虚拟机在运行时不但要和其它应用程序竞争处理器资源, 还要为一部分可有可无但具有更高优先级的系统进程或应用程序让出被调度的机会。显然, 在虚拟化为主的计算结点上这是不合理的。需要通过改变 KVM 虚拟机的调度策略和运行优先级来进行优化以改进虚拟机的性能表现, 使 Linux 系统中的 KVM 虚拟机运行时能获得比其他应用程序进程更多的占用处理器资源的机会<sup>[27]</sup>。

如何设置调度策略和优先级, 既不浪费物理计算资源又能保证虚拟机的使用有较好的性能体验, 这需要与虚拟机上运行的业务系统相联系。本章结合上述分析, 提出 KVM 虚拟机的调度优化方案如下:

(1) 普通业务虚拟机、测试用虚拟机、访问量较少的 WEB 服务器、日志服务器等采用默认调度策略和优先级。同时, 根据用户具体需求, 提供动态调整的途径, 调度策略不变, 提高虚拟机进程的静态优先级, 可将 `nice` 值降低范围在 0 到 -10 之间, 以在普通进程中优先获得更多的 CPU 运行时间。

(2) 核心业务服务器、文件服务器、FTP 服务器、访问量较大的 WEB 服务器等采用时间片轮转的实时调度策略, 实时优先级为 90, `nice` 取值范围 0 到 -10 之间, 使这些虚拟机比普通进程和普通虚拟机优先得到调度, 虚拟机业务运行流畅, 具有很高的实时响应表现。

(3) 网络监控服务器、单位 OA 服务器、实时性要求高的业务服务器等采用时间片轮转的实时调度策略, 实时优先级为 99 (最高), `nice` 取值范围 -10 到 -20 之间。在物理计算资源负载未达到饱和时, 此类虚拟机将获得接近物理服务器的性能, 用户体验不出业务承载在虚拟服务器上。

(4) 要求最高性能体验的虚拟机, 采用 `SCHED_FIFO` 调度策略, 实时优先级设置为 99 (最高)。这主要针对有特殊性能要求的业务服务器, 既要求有物理机的性能, 又同时需要快速部署、易迁移、按需分配等功能。这类虚拟机在平台中不宜过多运行, 需要控制好部署密度, 属于计算资源消耗较多的进程。

### 4.3 KVM 虚拟机调度优化方案实施

KVM 虚拟机调度优化方案的实现主要包括三个方面：监控虚拟机进程状态、调整调度策略、修改运行优先级。首先，设计实现一个进程信息的实时监控程序，监控虚拟机运行进程的各项资源占用状态。然后，对虚拟机进程优先级和调度策略进行实时修改。最后，根据各虚拟机的业务需求，个性化调整处理器调度策略和运行优先级。实现进程的实时监控是为了掌握 KVM 虚拟运行平台中计算资源的整体开销情况，是上述调度优化方案实施的基础。

#### 4.3.1 实现进程信息实时监控

默认创建的 KVM 虚拟机对应 Linux 系统中的普通进程，其运行状态、相关配置和计算资源的占用情况可以从虚拟机在内核的进程描述符（task\_struct）中获取。通过增加内核模块 mykvm\_LKM，创建一个/proc 内存文件系统对像，重定义该文件的 open 函数，在 open 函数中读取系统的所有进程相关信息，使用用户空间程序可以在/proc/taskinfo 内存文件中获取需要的进程信息<sup>[37-44]</sup>。再经过程序处理，每间隔一小段时间，读取一次，通过程序格式化输出到终端屏幕上，实现进程信息的实时动态监控。重要的进程信息主要有：进程动态优先级、静态优先级、实时优先级、调度策略、当前内存使用量及百分比、一个间隔时间内 CPU 使用率、CPU 占用时间等。通过这些信息，可以实时查看到虚拟机的计算资源占用情况。进程信息获取流程如图 4-1 所示：

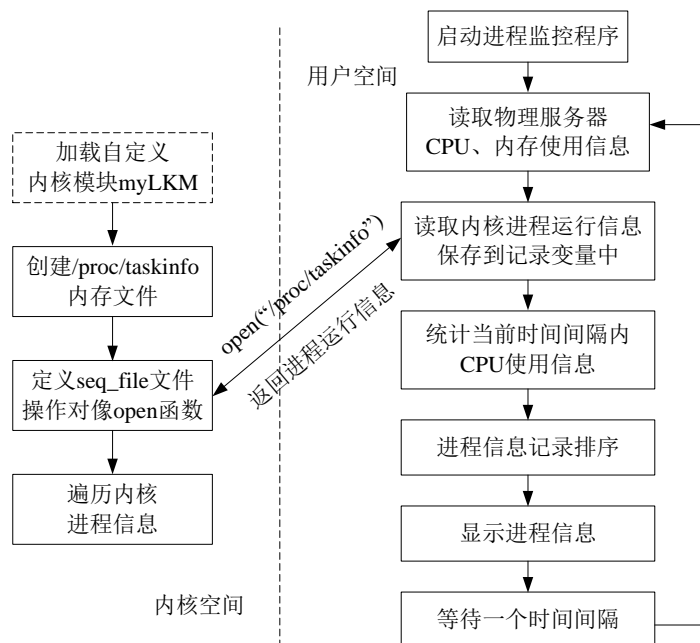


图 4-1 进程信息获取流程



该监控程序命名为 `getProcessInfo`，实现了 Linux 系统进程信息动态显示，主要包括进程调度策略、动态优先级、静态优先级、`nice` 值、实时优先级、系统时间频率、CPU 使用率、内存使用率等。启动步骤为：检查并加载 `mykvm_LKM` 模块->运行监控程序->根据进程运行情况执行 KVM 虚拟机调度优化。检查及加载模块如图 4-2 所示：

```
[root@localhost myLKM]# lsmod |grep kvm
kvm_intel          233472    22
kvm                720896    1  kvm_intel
mykvm_LKM         16384     0
irqbypass         16384    11  kvm
```

图 4-2 检查模块加载情况

如未加载，可执行：

```
cp mykvm_LKM.ko /lib/modules/4.17.8/kernel/arch/x86/myLKM
```

```
modprobe mykvm_LKM
```

监控程序运行界面如图 4-3、图 4-4 所示：

219.223.215.198 219.223.215.198 (1) 219.223.215.198 (2) x 219.223.254.19 219.223.254.221

深圳大学-2016级软件工程-getProcessInfo,服务器开机时间: 2018年8月31日 13:58:18 星期三

Tasks:312 total, 1 running, 0 blocked

cpu_name		当前使用率		平均使用率		最大使用率(peak)							
%cpu		0.29%(8.72%)		0.31%(8.29%)		0.42%(9.96%)							
单位(MB) mem:24087.543 total, 10468.348 free, 13461.855 available, 2.074 文件/磁盘													
巨页数量: 0 total, 0 free, 用户内存使用率(mem_user%): 44.112791%, 系统内存使用率(mem_													
序号	PID	user	状态	pri	ni	poli	rt	sCPU	时间	CPU	sCPU%	内存	mem%
1	1656	root	S	120	0	Normal	0	CPU 0:	38 ms	0	3.721841%	3103.71M	12.89%
2	29299	root	R	120	0	Normal	0	CPU22:	21 ms	1	2.056807%	5.18M	0.02% 0.
3	1949	root	S	120	0	Normal	0	CPU 5:	5 ms	0	0.489716%	3035.27M	12.60%
4	21385	root	I	120	0	Normal	0	CPU23:	3 ms	0	0.293830%	0.00M	0.00% 0.
5	21333	root	I	120	0	Normal	0	CPU 5:	2 ms	0	0.195886%	0.00M	0.00% 0.
6	1687	root	S	120	0	Normal	0	CPU 5:	1 ms	0	0.097943%	0.00M	0.00% 0.
7	1	root	S	120	0	Normal	0	CPU22:	0 ms	0	0.000000%	6.80M	0.03% 0.
8	2	root	S	120	0	Normal	0	CPU22:	0 ms	0	0.000000%	0.00M	0.00% 0.
9	3	root	I	100	-20	Normal	0	CPU 0:	0 ms	0	0.000000%	0.00M	0.00% 0.
10	5	root	I	100	-20	Normal	0	CPU 0:	0 ms	0	0.000000%	0.00M	0.00% 0.
11	8	root	I	100	-20	Normal	0	CPU 0:	0 ms	0	0.000000%	0.00M	0.00% 0.
12	9	root	S	120	0	Normal	0	CPU 0:	0 ms	0	0.000000%	0.00M	0.00% 0.
13	10	root	I	120	0	Normal	0	CPU10:	0 ms	0	0.000000%	0.00M	0.00% 0.
14	11	root	I	120	0	Normal	0	CPU 4:	0 ms	0	0.000000%	0.00M	0.00% 0.
15	12	root	S	0	0	FIFO	99	CPU 0:	0 ms	0	0.000000%	0.00M	0.00% 0.
16	13	root	S	0	0	FIFO	99	CPU 0:	0 ms	0	0.000000%	0.00M	0.00% 0.
17	14	root	S	120	0	Normal	0	CPU 0:	0 ms	0	0.000000%	0.00M	0.00% 0.
18	15	root	S	120	0	Normal	0	CPU 1:	0 ms	0	0.000000%	0.00M	0.00% 0.

图 4-3 进程信息监控程序局部运行界面

219.223.215.198 | 219.223.215.198 (1) | 219.223.215.198 (2) x | 219.223.254.19 | 219.223.254.221

深圳大学-2016级软件工程-getProcessInfo, 服务器开机时间: 2018年8月31日 13:58:18 星期五, 当前时间: 2018年9月2日 12:57:3 星期天

Tasks: 132 total, 1 running, 0 blocked

cpu\_name 当前使用率 平均使用率 最大使用率(peak) 空闲率(idle)

xcpu 0.34%(7.45%) 0.34%(7.64%) 0.42%(14.87%) 99.66%(92.55%)

单位(MB) mem:24087.543 total, 10470.254 free, 13463.797 available, 2.074 文件/磁盘缓冲(buff), 4457.117 高速缓冲(cached)

内存数据: 0 total, 0 free, 用户内存使用率(mem.user%): 44.104723%, 系统内存使用率(mem.sys%), 56.532495%

序号	PID	user	状态	pri	ni	poli	rt	SCPU	时间	CPU	SCPU%	内存	mem%	TCPU%	TCPU%	nvcs	nvcs	slice	进程名
1	1656	root	S	120	0	Normal	0	CPU 1:	38 ms	0	3.678606%	3103.71M	12.89%	0.1230202680%	0.0001568879%	144309334	273154	100	kvm
2	29299	root	R	120	0	Normal	0	CPU 0:	31 ms	1	3.003876%	5.18M	0.02%	0.0000805817%	0.0000034897%	160	1505	100	getProcessInfo
3	21385	root	I	120	0	Normal	0	CPU23:	5 ms	0	0.484027%	0.00M	0.00%	0.0016396373%	0.0000686835%	7741191	9	100	kworker/23:0
4	29120	root	I	120	0	Normal	0	CPU 3:	3 ms	0	0.290416%	0.00M	0.00%	0.0020256544%	0.0000848914%	8393378	13	100	kworker/3:0
5	1949	root	S	120	0	Normal	0	CPU 3:	3 ms	0	0.290416%	3035.27M	12.60%	0.0193551129%	0.0008114039%	9284607	51663	100	kvm
6	29298	root	I	120	0	Normal	0	CPU 0:	1 ms	0	0.096899%	0.00M	0.00%	0.0000003136%	0.0000000136%	17974	0	100	kworker/0:1
7	25885	root	I	120	0	Normal	0	CPU15:	1 ms	0	0.096805%	0.00M	0.00%	0.0018514527%	0.0000773810%	8140570	21	100	kworker/15:0
8	1699	root	I	120	0	Normal	0	CPU13:	1 ms	0	0.096805%	0.00M	0.00%	0.0019245744%	0.0000806765%	8265306	25	100	kworker/13:2
9	1954	root	S	120	0	Normal	0	CPU20:	1 ms	0	0.096805%	0.00M	0.00%	0.0024592095%	0.0001030948%	4406794	13211	100	vhost-1949
10	1	root	S	120	0	Normal	0	CPU22:	0 ms	0	0.000000%	6.80M	0.03%	0.0000553789%	0.0000023075%	5452	343	100	systemd
11	2	root	S	120	0	Normal	0	CPU22:	0 ms	0	0.000000%	0.00M	0.00%	0.0000003136%	0.0000000131%	1608	1	100	ktreadd
12	3	root	I	100	-20	Normal	0	CPU 0:	0 ms	0	0.000000%	0.00M	0.00%	0.0000000000%	0.0000000000%	2	0	100	rcu_gp
13	5	root	I	100	-20	Normal	0	CPU 0:	0 ms	0	0.000000%	0.00M	0.00%	0.0000000000%	0.0000000000%	13	2	100	kworker/0:0H
14	8	root	I	100	-20	Normal	0	CPU 0:	0 ms	0	0.000000%	0.00M	0.00%	0.0000000000%	0.0000000000%	2	1	100	mm_percpu_wq
15	9	root	S	120	0	Normal	0	CPU 0:	0 ms	0	0.000000%	0.00M	0.00%	0.0000002464%	0.0000000103%	309734	11	100	ksoftirqd/0
16	10	root	I	120	0	Normal	0	CPU 4:	0 ms	0	0.000000%	0.00M	0.00%	0.0000309602%	0.0000012900%	2512190	329	100	rcu_sched
17	11	root	I	120	0	Normal	0	CPU 4:	0 ms	0	0.000000%	0.00M	0.00%	0.0000000000%	0.0000000000%	1666	0	100	rcu_bh
18	12	root	S	0	0	FIFO	99	CPU 0:	0 ms	0	0.000000%	0.00M	0.00%	0.0000004032%	0.0000000168%	2324	0	100	migration/0
19	13	root	S	0	0	FIFO	99	CPU 0:	0 ms	0	0.000000%	0.00M	0.00%	0.0000000448%	0.0000000019%	42284	1	100	watchdog/0
20	14	root	S	120	0	Normal	0	CPU 0:	0 ms	0	0.000000%	0.00M	0.00%	0.0000000448%	0.0000000019%	42	0	100	cpuhp/0
21	15	root	S	120	0	Normal	0	CPU 1:	0 ms	0	0.000000%	0.00M	0.00%	0.0000000000%	0.0000000000%	43	1	100	cpuhp/1
22	16	root	S	0	0	FIFO	99	CPU 1:	0 ms	0	0.000000%	0.00M	0.00%	0.0000000000%	0.0000000000%	42283	0	100	watchdog/1
23	17	root	S	0	0	FIFO	99	CPU 1:	0 ms	0	0.000000%	0.00M	0.00%	0.0000010081%	0.0000000420%	5813	0	100	migration/1
24	18	root	S	120	0	Normal	0	CPU 1:	0 ms	0	0.000000%	0.00M	0.00%	0.0000004256%	0.0000000177%	29887	0	100	ksoftirqd/1
25	20	root	I	100	-20	Normal	0	CPU 1:	0 ms	0	0.000000%	0.00M	0.00%	0.0000000000%	0.0000000000%	14	0	100	kworker/1:0H
26	22	root	S	120	0	Normal	0	CPU 2:	0 ms	0	0.000000%	0.00M	0.00%	0.0000000000%	0.0000000000%	42	0	100	cpuhp/2
27	23	root	S	0	0	FIFO	99	CPU 2:	0 ms	0	0.000000%	0.00M	0.00%	0.0000000000%	0.0000000000%	42284	0	100	watchdog/2
28	24	root	S	0	0	FIFO	99	CPU 2:	0 ms	0	0.000000%	0.00M	0.00%	0.0000599938%	0.0000024997%	9808	0	100	migration/2
29	25	root	S	120	0	Normal	0	CPU 2:	0 ms	0	0.000000%	0.00M	0.00%	0.0000036516%	0.0000001522%	85388	0	100	ksoftirqd/2
30	27	root	I	100	-20	Normal	0	CPU 2:	0 ms	0	0.000000%	0.00M	0.00%	0.0000000000%	0.0000000000%	14	0	100	kworker/2:0H
31	28	root	S	120	0	Normal	0	CPU 3:	0 ms	0	0.000000%	0.00M	0.00%	0.0000000000%	0.0000000000%	42	0	100	cpuhp/3
32	29	root	S	0	0	FIFO	99	CPU 3:	0 ms	0	0.000000%	0.00M	0.00%	0.0000000448%	0.0000000019%	42284	0	100	watchdog/3
33	30	root	S	0	0	FIFO	99	CPU 3:	0 ms	0	0.000000%	0.00M	0.00%	0.0000012321%	0.0000000513%	6020	0	100	migration/3
34	31	root	S	120	0	Normal	0	CPU 3:	0 ms	0	0.000000%	0.00M	0.00%	0.0000002688%	0.0000000112%	9363	0	100	ksoftirqd/3
35	33	root	I	100	-20	Normal	0	CPU 3:	0 ms	0	0.000000%	0.00M	0.00%	0.0000000000%	0.0000000000%	14	0	100	kworker/3:0H

已循环检测 161 次, 每次间隔 1 秒, jiffies=4463792097, jiffies64=4463792097, system HZ=1000! 【KVM虚拟机的性能研究与改进】

【服务器启动到现在的时间: 51天15小时56分钟32秒】

图 4-4 进程信息监控程序运行界面

在源文件 `getProcessInfo.c` 中实现实时监控进程信息的关键代码如下:

```
int main(int argc, char *argv[])
```

```
{ . . . . .
```

//创建一个线程与以下循环同时执行, 并等待键盘输入 `q` 后控制退出循环

```
i=pthread_create(&thread,NULL,waitKeyInput,(void *)&circleShow);
```

```
do{ //循环统计进程、cpu、内存信息
```

```
readCpuInfo();//读物理服务器 cpu 使用信息
```

```
readProcessInfo();//读所有进程运行信息
```

```
readMemInfo();//读物理服务器内存使用信息
```

`cpuInfoStatisticsFromProcess();` // 在读取进程信息基础上, 基于 `/proc/taskinfo` 文件统计当前时间间隔内的 `cpu` 使用信息(计算一个间隔时间内 `cpu` 的使用率, 即由各进程信息统计出 `cpu` 使用率)

`sortProcessInfo();` // 进程信息按指定指标排序, 指标可为按 `cpu` 使用率, 内存使用率。默认按 `cpu` 使用率排序 (按 `c` 键), 通过按 `m` 键切换成按内存使用率排序

```
//以下显示各项统计信息
```

```
circleNumber++; //记录循环次数
```

```
move(0, 0); //屏幕输出起始坐标
```

```
showCpu_MemInfo();//显示 cpu、内存当前的运行信息
```

```

        showProcessInfo();//显示进程当前运行信息
        sleep(sleepJg);//等待指定的时间间隔
        swapProcessPoint();//交换当前和上一次的进程信息队列指针，当前成为上一次，上一次成为当前，下一轮读取记录时成为下一轮间隔的当前队列指针，在完成两次信息采样的同时可避免赋值开销。

        swapCpuInfoPoint();//交换当前和上一次时 cpu 的信息变量指针，完成两次信息采样的同时避免赋值开销。
    }
    while(circleShow);
    . . . . .
}

```

监控程序可按 j 键切换间隔 1 秒、2 秒、3 秒进行实时监控，默认以进程在一个时间间隔内的 CPU 使用率高到低排序，可按 m 键以进程内存使用率高到低排序，按 1 显示每个逻辑 CPU 核使用率，按 2-9 显示 20-90 条进程信息，按 n 将以每屏 100 条记录显示，按 q 退出监控循环。

#### 4.3.2 调整调度策略

在监控进程运行状态基础上，可以实时查看 KVM 虚拟机进程采用的调度策略，在生产系统平台上，可按 4.2 节拟定的虚拟机调度优化方案，结合监控情况，动态调整虚拟机对应进程的调度策略。具体是通过 sched\_setscheduler() 系统调用进行设置，函数原型如下：

```
int sched_setscheduler(pid_t pid,int policy,const struct sched_param *sp)[34];
```

实验中，根据业务需要，实时设置虚拟机的调度策略为 SCHED\_FIFO，实时优先级为 99。以 win2003-200G-test-2 对应的虚拟机进程为例，将进程号、调度策略值 1 和优先级 99 作为参数传递给程序 setSchedulerFromPid 即可。修改过程如图 4-5 所示，监控程序显示修改后的进程运行状态如图 4-6 所示：

```
[root@localhost myLKM]# virsh vncdisplay win2003-200G-test-2
:2

[root@localhost myLKM]# netstat -tunpl
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:22             0.0.0.0:*               LISTEN      1484/sshd
tcp        0      0 0.0.0.0:5900           0.0.0.0:*               LISTEN      1949/kvm
tcp        0      0 0.0.0.0:5901           0.0.0.0:*               LISTEN      1592/kvm
tcp        0      0 0.0.0.0:5902           0.0.0.0:*               LISTEN      1656/kvm
tcp6       0      0 :::22                  :::*                     LISTEN      1484/sshd
[root@localhost myLKM]# ./setSchedulerFromPid 1656 1 99
调度策略取值:
#define SCHED_NORMAL 0
#define SCHED_FIFO 1
#define SCHED_RR 2
#define SCHED_BATCH 3
进程实时优先级范围: 0 (低优先级) -99 (高优先级)
调度策略设置成普通策略时, 运行优先级对应实时优先级, 只能设置成0
程序 ./setSchedulerFromPid 输入的进程pid= 1656, 将设置该进程的调度策略 policy = 1, 设置运行优先级prio= 99
设置结果: 进程 pid = 1656, 调度策略policy= 1, 运行优先级prio = 99
```

图 4-5 程序修改进程调度策略过程

深圳大学-2016级软件工程-getProcessInfo,服务器开机时间: 2016-08-10 10:10:10  
Tasks:310 total, 1 running, 0 blocked

cpu_name	当前使用率	平均使用率
%cpu	0.29%(7.35%)	0.31%(7.58%)

单位(MB) mem:24087.543 total, 10466.480 free, 13462.898 used  
总页数: 0 total, 0 free, 用户内存使用率(mem\_user%): 44.1%

序号	PID	user	状态	pri	mi	pol	rt	SCPU	时间	CPU
1	1656	root	S	0	0	FIFO	99	CPU 3:	30 ms	0
2	29299	root	R	120	0	Normal	0	CPU10:	21 ms	1
3	1949	root	S	120	0	Normal	0	CPU 5:	8 ms	0
4	21333	root	I	120	0	Normal	0	CPU 5:	4 ms	0
5	29120	root	I	120	0	Normal	0	CPU 3:	4 ms	0
6	1898	root	I	120	0	Normal	0	CPU11:	1 ms	0

图 4-6 监控程序中进程调度策略状态

### 4.3.3 修改运行优先级

与调整调度策略过程相同, 在生产系统平台上, 可按 4.2 节拟定的虚拟机调度优化方案, 结合监控情况动态调整虚拟机对应进程的优先级, 使虚拟机获得比其他进程更多的处理器资源。具体是通过 `setpriority()` 系统调用进行设置, 函数原型如下:

```
int setpriority(int which,int who,int prio) [34];
```

该函数主要用于普通进程 `nice` 值的修改, 实时进程优先级的修改方法是通过上节介绍的调整调度策略函数修改。

实验中, 根据业务需要实时设置虚拟机的静态优先级, 将默认 `nice` 值 0 修改为最高值-20, 仍然以 win2003-200G-test-2 对应的虚拟机进程为例, 将进程号、`nice` 值-20 作为参数传递给程序 `setPriorityFromPid` 即可。修改过程如图 4-7 所示, 监控程序显示修改后的进程运行状态如图 4-8 所示:

```
[root@localhost myLKM]# ./setPriorityFromPid 1656 -20
进程静态优先级范围: 100 (高优先级) -139 (低优先级)
对应nice值范围: -20 (高优先级) -19 (低优先级)
程序 ./setPriorityFromPid 输入的进程pid = 1656, 将设置进程静态优先级prio = -20
设置结果: 进程pid = 1656, 静态优先级nice= -20
```

图 4-7 程序修改进程优先级过程

```

深圳大学-2016级软件工程-getProcessInfo,服务器开机时间: 2
Tasks:311 total, 1 running, 0 blocked
cpu_name 当前使用率 平均使用率
%cpu 0.38%(11.74%) 0.37%(10.96%)
单位(MB) mem:24087.543 total, 10464.414 free, 13461.137
巨页数量: 0 total, 0 free,用户内存使用率(mem_user%): 44.
序号 PID user 状态 pri ni pol rt sCPU 时间 CPU
1 1656 root S 0 -20 FIFO 99 CPU 3: 50 ms 0
2 29299 root R 120 0 Normal 0 CPU 4: 22 ms 1
3 1949 root S 120 0 Normal 0 CPU 7: 8 ms 0
4 29120 root I 120 0 Normal 0 CPU 3: 5 ms 0
5 32664 root I 120 0 Normal 0 CPU 7: 2 ms 0
6 1699 root I 120 0 Normal 0 CPU13: 2 ms 0

```

图 4-8 监控程序中进程优先级状态

在监控程序基础上，用户可以实时调整 KVM 虚拟机的调度策略和运行优先级。经过反复测试与实验，在 KVM 虚拟化平台上实施上述虚拟机调度优化，即根据业务要求，对虚拟机进程状态进行灵活的个性化的修改，再从反馈的性能表现来重新调整，最终能够使物理资源利用率和虚拟机性能得到真正的改进。

#### 4.4 本章小结

本章从 KVM 虚拟机在 Linux 系统中的 CPU 资源占用率出发，深入研究 Linux 进程管理的核心内容，理解进程的调度策略、运行优先级对进程占用处理器资源的影响。在掌握了进程优先级与时间片关系、普通进程采用的完全公平调度算法和实时进程调度算法基础上，提出了 KVM 虚拟机调度优化方案。该方案可根据实际业务来实时修改虚拟机进程的调度策略和运行优先级，使虚拟机能够按需求获得比普通进程更多占用物理处理器资源的机会，同时又可以有区别的调整虚拟机之间占用 CPU 资源的比例，达到性能优化目的。

为使该方案具有可实施性，本章设计并实现了一个进程信息的实时监控程序，监控虚拟机运行进程的各项资源占用状态。在监控的基础上，实现了虚拟机进程优先级和调度策略的实时修改方法。经过反复测试与实验，确定能够提高物理计算资源利用率，使虚拟平台中所有虚拟机的整体性能得到改进，验证了该优化方案的可实施性。

## 第 5 章 虚拟机内存使用优化

影响虚拟机性能的因素中最主要的是处理器、内存和 I/O 访问，前面已对处理器的虚拟化原理及有关的调度优化进行了介绍。本章从提高虚拟机内存实际使用率出发，理解内存虚拟化技术原理，分析各种内存调优方法，提出内存使用的性能优化方案。

### 5.1 内存虚拟化技术原理

在虚拟化环境下，虚拟机使用内存，需要经过两次地址转换才能实现虚拟机内存地址到物理机内存地址的转换<sup>[45]</sup>。两次转换分别是虚拟机逻辑地址（简称为 GVA）到虚拟机物理地址（简称为 GPA）的转换和虚拟机物理地址（GPA）到宿主机物理地址（简称为 HPA）的转换<sup>[45]</sup>。即 GVA-→GPA-→HPA 的两次转换规律。其中，GVA 到 GPA 的转换是由虚拟机操作系统实现，直接通过 VMCS 结构体变量中的虚拟机页表寄存器 CR3 指向的页表来指定。而 GPA 到 HPA 的转换则由 KVM 内核模块通过将物理内存分配给虚拟机来实现。KVM 中的 `kvm_run` 数据结构会记录这个映射关系，每个 KVM 虚拟机会动态更新 GPA 到 HPA 的映射表<sup>[1]</sup>。

通过内存虚拟化。使每个虚拟机拥有自己视角的内存系统，并且可以随机分布在物理内存空间的各个区域，实现了虚拟机之间、虚拟机与宿主机之间内存的隔离，避免了虚拟机内部的程序执行影响到其他虚拟机或宿主机的程序运行，为实现完全虚拟化提供了程序安全运行的环境<sup>[1]</sup>。

虚拟化技术最早由软件实现从虚拟机虚拟逻辑地址（GVA）到宿主机物理地址（HPA）之间的映射，软件实现原理采用的是影子页表（Shadow Page Tables）的映射机制<sup>[45]</sup>。在影子页表中，当虚拟机程序访问虚拟内存时可从映射关系表中找到对应的宿主机物理内存地址，直接完成 GVA 到 HPA 的转换，从而避免了上述的两次地址转换。但由于影子页表的实现过程非常复杂，内存开销比较大，性能表现较差。目前只在软件实现的完全虚拟化系统中使用，而在 KVM 虚拟化解决方案中，已通过 Intel CPU 提供的硬件 EPT<sup>[1]</sup> (Extended Page Tables, 扩展页表) 技术代替了影子页面技术，从硬件上实现了 GVA-→GPA-→HPA 的两次地址转换<sup>[46]</sup>。KVM 方案的这个优化极大的提高了内存虚拟化的性能<sup>[2]</sup>。在 AMD 平台，同样对 KVM 提供类似的技术支持，叫做 NPT<sup>[1]</sup>。

前面提到，KVM 内核模块可以直接通过分配物理内存给虚拟机的方式，完成 GPA 到 HPA 的转换。这个转换通常由用户空间程序执行 `ioctl` 系统调用，传入 `KVM_SET_USER_MEMORY_REGION` 指令字完成，虚拟机中的物理地址通过该函数映射到宿主机的物理内存中，KVM 模块执行 EPT 技术对应的调用就能完成地址转换过程的硬件实现。

## 5.2 内存使用性能优化

KVM 虚拟机内存使用的性能优化可分为两个方面，一是优化内存的使用率，分配了多少内存，分配的内存是否在使用，如果没有使用能否释放，回收另做他用，如果在用，是否可以共享，由此提高内存的使用效率，间接提升内存访问的性能。二是直接优化内存的访问效率，对给定的内存，在访问时能否降低缺页率，减少 VCPU 因缺页而陷入的次数，从而提高内存访问性能。

KVM 虚拟化平台对应上述两方面都分别有比较成熟的优化技术。在内存使用率方面，主要是 KSM（Kernel SamePage Merging）内核同页合并技术、Virtio 下的 balloon 气球技术。在内存访问效率方面，主要是大页（Huge Page）技术。本章通过掌握这些技术原理，提出了应用到具体平台的优化方案。执行时，尽可能减少各种优化技术的负面影响，提高优化效率<sup>[47]</sup>。

### 5.2.1 KSM 内存使用优化

KSM 允许 Linux 系统中的多个进程之间共享完全相同的内存页。启动 KSM 服务后，内核 `ksmd` 守护程序会定期检查正在运行的所有进程程序，对这些进程所占用的内存进行比较，如果有使用相同的内存页或者区域，就把这些使用相同内存页的进程的内存页合并，形成一个共享的内存页，并把这一个内存页面标记成“写时复制”，该页内存为只读模式。这样就起到节省内存提高内存使用率的作用。系统运行过程中，如果有进程需要写入它共享的内存页时，Linux 系统会先把该内存页复制一份，该进程使用新复制的这页内存，不再关联原来共享的内存页了。而其他进程仍然可以共享原来的内存页<sup>[2]</sup>。

KSM 设计的初衷就是针对 KVM 的虚拟化场景。在 KVM 虚拟化环境中，如果物理服务器的性能足以支持 KSM 守护进程的消耗，那么 KSM 能够显著提高内存的速度和使用效率，具体可以从以下两个方面来理解：

- （1）经过 `ksmd` 对各内存页的比较，将相同的内存页进行合并，同构模式

的虚拟机必然有大量相同内容的内存页，这样就把虚拟机的总内存使用量降低了。所以在分配给一个虚拟机 3G 内存时，运行时看到的占用内存量却只有 500M。一方面，因为合并使得相同内存页减少了，保存到 CPU 的缓存中的其他内存页就可以增加一些，从而提高程序高速缓存的命中率。另一方面，减去的原来存相同内容的内存页可以用来缓存其它磁盘数据，使得磁盘数据的缓存命中率也提高了。因此，优化了内存的使用效率，改进了虚拟机运行性能。

(2) 由 KSM 的服务原理可知，能够节省内存，可将虚拟机实际需要的内存容量扩展到大于物理内存，所有虚拟机仍然可以正常且流畅的运行，这就形成了内存过载现象。因此，对于使用相同共享内存的虚拟机，在物理内存量不变的情况下，可以创建更多的虚拟机，提高了虚拟机的运行密度，也就提高了物理计算资源的利用率。

如果 KVM 虚拟平台中的虚拟机都是不同操作系统，非同构模式下的虚拟机使用相同的内存页会少很多，KSM 的优化效率也会相应地减弱。

启动 KSM 服务的方法在第 2 章创建虚拟机时已说明。在 `/sys/kernel/mm/ksm/` 目录中可以查看 KSM 的运行情况的相关统计文件，这些文件主要如表 5-1 所示：

表 5-1 KSM 运行情况统计文件

<code>pages_shared</code>	合并的页面数
<code>pages_sharing</code>	正在共享单个页面的虚拟页面数
<code>pages_unshared</code>	作为共享候选者但当前未共享的页数
<code>pages_volatile</code>	作为共享候选者但频繁更改的页数，KSM 服务不会合并这些页面
<code>full_scan</code>	为重复内容扫描 KSM 的次数
<code>merge_across_nodes</code>	是否允许在 NUMA 节点中执行合并
<code>pages_to_scan</code>	一次扫描的页数，该数字越大越降低系统性能
<code>sleep_miliseconds</code>	扫描之间的时间间隔

根据上述原理，KVM 虚拟机应使用以下方案来应用 KSM 技术优化内存访问性能：

在生产环境中打开 KSM 服务是为了内存超用，如果虚拟机上承载的业务系统访问量大，就必须监控 CPU 和内存使用情况，当 CPU 开销超过 70% 或内存超用过太时，因为要预留一部分内存用于写时复制而新增的内存页，所以，此时应



关闭该服务，否则可观察运行。

在测试环境和桌面虚拟化环境中，就尽可能打开 KSM 服务，节省内存用量。

### 5.2.2 Virtio 下的 balloon 气球技术

Virtio 半虚拟化驱动中有一个用于内存使用优化技术，称为 balloon（气球）技术，它实现了不关闭虚拟机的情况下动态调整内存用量的功能。

Balloon 气球技术的原理是通过在虚拟机和宿主机之间转移可用的内存量来实现。为了区分可用内存的转移状态，规定哪些内存虚拟机能用，哪些内存宿主机能用，实现互斥使用，而引入气球（Balloon）的概念。具体的实现原理是：首先，控制虚拟机的内存使用，在每个虚拟机中设置一个内存气球，使之能够提取虚拟机中的内存放置到气球中，虚拟机不能使用气球中的内存，宿主机可以使用气球中的内存。balloon 技术根据 VMM 中对宿主机和虚拟机内存的使用情况和两者提出的内存使用要求来控制气球中内存的大小。然后，当宿主机内存不足时，会通知 VMM，要求 balloon 技术从虚拟机回收初始分配的部分内存，放到气球中，内存气球充气膨胀，宿主机可以使用气球中的内存解决不足问题。最后，当虚拟机需要使用更多内存时，也会通知 VMM，要求 balloon 技术压缩本机的气球，缩小气球后释放出内存供该虚拟机使用。由此可知，虚拟机通过 balloon 技术实现了动态调整虚拟机所分得的内存，采取过剩才膨胀，需要时压缩，改善了虚拟机和宿主机两者的内存利用率。

Balloon 在节约内存和灵活分配内存方面有明显的优势，但也还有一些不完善的地方，具体使用时要视情况而定：

（1）针对虚拟机内存总用量要求高但平时维持业务运转内存使用率较低的情况，可应用 balloon 技术，优化该虚拟机的内存使用。在这种情况下，管理员初始分配较高的内存给虚拟机，确保完全满足业务需求。然后，不再人工调整内存。当业务系统访问量大或程序频繁运行需要大量内存时，初始分配的高内存能满足其性能要求，用户对虚拟机的访问不会有卡顿。当内存使用峰值过后，在相当长时间内，内存使用率较低，此时，balloon 技术调整气球大小，实现灵活分配内存的目的。使宿主机内存充裕，从而提高 VMM 下整个虚拟化平台的内存分配效率。

（2）对虚拟机内存使用量要求较大，但物理内存不足时，应该使用 balloon 技术，能提高虚拟机和宿主机的性能体验。在这种情况下，管理员可只分配基本

的内存量给虚拟机，确保其业务能正常稳定运行，同时，在业务繁忙需要更多内存支持时，可通知 VMM 调用 balloon 模块，向宿主机申请内存。以动态调节所有虚拟机的内存用量来满足本虚拟机业务处理所需的内存，保证虚拟机性能稳定。

(3) 在非主要业务承载的虚拟机上不开启 balloon 功能，避免频繁调整内存引起内存碎片大量增加，减少内存不足引起进程执行出错的概率。有效使用 balloon 技术才能真正提高虚拟机访问性能，确保用户的较好体验。

根据上述分析可知，Balloon 技术主要用两种操作，一是膨胀，将虚拟机的内存拿掉分给宿主机，二是压缩，将宿主机的内存还给虚拟机。实施虚拟机内存使用优化时，具体可按下面的步骤执行：

(1) 加载 virtio 驱动：modprobe virtio\_balloon

(2) 在虚拟机的 xml 配置文件中增加以下配置：

```
<memballoon model='virtio'>
  <alias name='balloon0' />
</memballoon>
```

(3) 虚拟机内存气球配置

//查看 centos7-test-1 虚拟机当前内存大小

```
virsh qemu-monitor-command centos7-test-1 --hmp --cmd info balloon
```

//限制上述虚拟机内存大小为 2GB

```
virsh qemu-monitor-command centos7-test-1 --hmp --cmd balloon 2048
```

重复第 (3) 步，应用 balloon 气球技术。

### 5.2.3 大页 (Huge Page) 技术

x86( 包括 x86-32 和 x86-64)架构下 CPU 默认使用的内存页面大小为 4k 字节<sup>[45]</sup>,同时,该架构也支持 CPU 使用更大的内存页,如:可设置内存页大小为 2MB,把这种大于 4k 的内存页称为大页 (huge page)。目前, Linux 的发行版都已支持 huge page。

在 KVM 中使用大内存页的目的是为了提高整个平台的内存访问效率,不同于之前研究的 KSM 和 balloon 技术,它们主要是提高内存的使用率,满足更多的虚拟机对内存用量的要求,即内存过载使用是它们应用的亮点。而设置一定数量的内存大页,可以从提高数据访问效率角度来提升虚拟机的性能表现。比如,在系统中设置了一定数量的 2M 内存页,按原来的 4k 一页计算,一个 2M 页面

可代替 512 个 4k 页面，可大量减少内存页面数量，也减少页表数量，使实际可用的物理内存得到增加。同时，程序一次全部加载到一个大页中，减少换入换出的次数。在常用进程执行指令时，转换检测缓冲区（TLB 快表）因为大页容纳了更多的数据，而提高访存的命中率。最终，使虚拟机性能得到较大提高<sup>[48]</sup>。

由上述 huge page 技术的原理可知，使用大页技术是 KVM 虚拟机的性能优化在内存访问上的一种应用。综合考虑该技术的优缺点，可以按以下方案使用大页技术：

（1）如果虚拟机访问量或业务压力较大时，应用大页技术，提高虚拟机访问速度。

（2）如果宿主机内存较小时，不建议使用大页，一般可设置物理总内存 10% 的容量做大页。1GB 甚至更大的内存页，视具体业务确定应用规则。

大页的具体应用，在第 2 章创建虚拟机中已有说明，本章补充 libvirt 管理虚拟机时手工配置大页的方法，在要使用大页的虚拟机 XML 配置文件中增加以下内容：

```
<memoryBacking>
  <hugepages/>
</memoryBacking>
```

上面的配置说明，虚拟机自动使用大页，数量由 VMM 自动确定，也可以在配置中指定使用的大页数量：

```
<memoryBacking>
  <hugepages>
    <page size="100" unit="M" nodeset="0"/>
  </hugepages>
  <source type="file"/>
  <access mode="private"/>
  <allocation mode="immediate"/>
</memoryBacking>
```

通过上述三个内存调优技术的应用，KVM 虚拟机的性能根据业务特点都将有不同程度的改进。

### 5.3 本章小结

影子页表是软件内存虚拟化的实现，而硬件扩展页表 EPT 技术是 KVM 方案中硬件辅助的内存虚拟化的实现。本章在深入理解内存虚拟化技术原理的基础上，从优化内存的使用率和访问效率两方面，研究了 KSM 内核同页合并技术、Virtio 下的 balloon 气球技术、大页（Huge Page）技术。根据物理资源环境和业务系统需求提出内存优化方案和适应的场景，并介绍了具体实现的方法和步骤。经过实验，这些技术都能一定程度上改进 KVM 虚拟机的性能。

## 第 6 章 I/O 性能优化

计算机的输入输出处理是与用户交互最多的一个部分，KVM 虚拟机执行性能的表现，很多时候都体现在 I/O 任务的处理上。KVM 虚拟化的 I/O 服务都由 QEMU 处理，因此，完全虚拟化中 QEMU 处理 I/O 任务的性能就决定了虚拟机的 I/O 性能。

QEMU 本身是一个集 I/O 设备、CPU、内存等各类硬件模拟功能于一体的支持完全虚拟化的模拟软件，通过翻译模拟设备发出的二进制指令到物理平台执行的纯软件方式实现虚拟化<sup>[22]</sup>。加入 KVM 支持后将 CPU、内存等模拟功能让给了硬件，由 KVM 模块负责了。虚拟机的 I/O 虚拟依然是默认的软件虚拟化方式。虽然保证了虚拟机不经过任何修改就能运行在 KVM 环境中，但原生的 KVM 虚拟机在磁盘、网络等 I/O 设备的访问性能非常低，仅满足于实验或测试场景的应用。虚拟机要获得较好的 I/O 性能就必须抛弃 QEMU 提供的完全虚拟化下的模拟设备驱动。目前 KVM 虚拟机 I/O 性能优化的解决方案是主推 virtio 设备驱动标准下的半虚拟化 I/O 驱动，为了获得更高性能可以采用 Intel VT-d 提供的设备直接访问技术或单根 I/O 虚拟化（SR-IOV）标准<sup>[25]</sup>。

本章先介绍 virtio 半虚拟化驱动的 I/O 性能优化技术的架构和特点，结合其实现原理，以磁盘和网卡设备为例，提出了通过 virtio 半虚拟化技术改进 KVM 虚拟机 I/O 访问性能的具体步骤。然后，介绍硬件支持下的 KVM 虚拟化 I/O 优化技术，包括设备直接访问和 SR-IOV 技术的应用。

### 6.1 virtio 半虚拟化技术

Virtio 已经是一个稳定成熟的驱动技术，KVM 选择它作为 I/O 半虚拟化的标准，在 Linux 内核 2.6 以后的版本都支持 virtio 驱动<sup>[14]</sup>。其实现性能优化的思路正是第 3 章虚拟机优化思路的第二个思路：I/O 任务被 KVM 传送给 QEMU 处理，属于 VCPU 的第三个切换模式，即 QEMU 在 I/O 处理时执行系统调用到内核空间的切换。virtio 半虚拟化通过采取 I/O 缓冲队列在一次切换中完成多个 I/O 处理，比完全虚拟化时减少了切换次数，也就是减少 VCPU 的 VM-Exit 次数，从而使虚拟机的 I/O 服务达到接近本地物理设备的性能表现。

Virtio 技术是一套设备半虚拟化组件的软件驱动实现<sup>[49]</sup>，包括块设备（virtio-blk）、网络服务（virtio-net）、PCI 总线（virtio-pci）、内存管理（virtio-balloon）、

环形缓冲虚拟队列（virtio-ring）、控制台（virtio-console）等组件，以模块的方式加载到 Linux 内核中，在 KVM 中的设备层半虚拟化架构如图 6-1 所示：

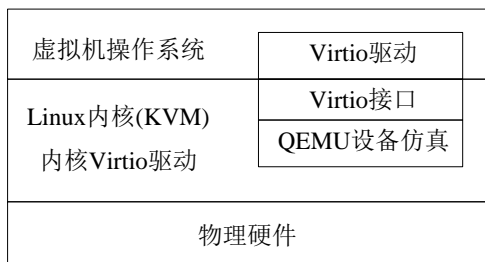


图 6-1 KVM 的 virtio 设备层半虚拟化架构

Virtio 实现设备半虚拟化访问的流程如图 6-2 所示：

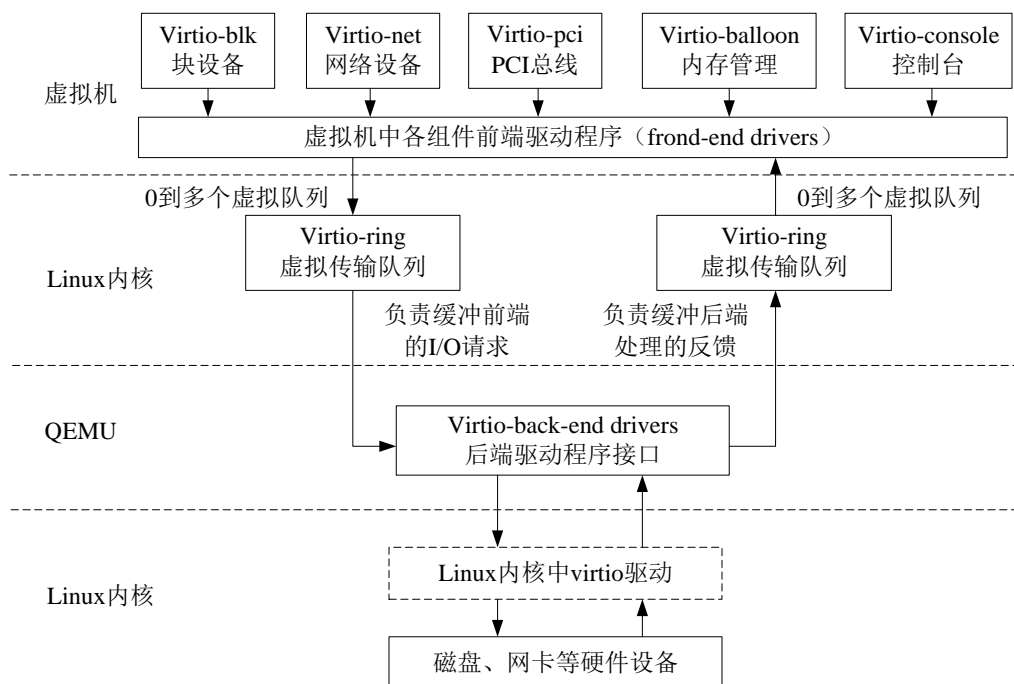


图 6-2 virtio 设备半虚拟化访问流程

上图可以看到，virtio 处理虚拟机的 I/O 请求流程是：前端驱动接收虚拟机 I/O 请求，由前端驱动与内核的 virtio-ring 环形虚拟传输队列建立联系，实际由 KVM 模块在内核创建缓冲区，完成这个衔接。可以缓冲保存前端驱动发来的多次 I/O 请求，附加到 QEMU 进程中的后端处理程序，利用内核 virtio 驱动调用宿主机中真实设备驱动，执行 I/O 操作，批量处理这些请求。最后 QEMU 进程将处理结果反馈回环形虚拟传输队列，虚拟机前端驱动从队列中取得处理结果。

在 KVM 虚拟化方案中，一方面虚拟机是 QEMU 的线程，也是一个 Linux 用户态中的进程，通过共享内存的方式，虚拟机与 QEMU 在 virtio 半虚拟化环境下交换数据，效率非常高。另一方面，virtio 环形缓冲虚拟队列和后端处理都

以批处理方式执行，有效减少了 VCPU 的上下文切换和数据复制次数。所以虚拟机能够获得很好的 I/O 性能。

Virtio 技术的应用需要虚拟机、宿主机和 QEMU 都支持才能实施<sup>[50]</sup>。目前 Linux 的内核中都集成了 virtio 模块，加载后即可使用其驱动，QEMU 从早期版本开始已支持 virtio 功能，虚拟机中需要确定安装了 virtio 驱动。应用 virtio 半虚拟化技术改进虚拟机 I/O 访问性能的具体步骤和相关实验如下：

(1) 宿主机 Linux 内核中加载 virtio 组件模块(virtio、virtio\_ring、virtio\_pci 是必须加载的模块，virtio\_net、virtio\_blk、virtio\_balloon 是可选模块)。

```
modprobe virtio
modprobe virtio_ring
modprobe virtio_pci
modprobe virtio_net
modprobe virtio_blk
modprobe virtio_balloon
```

(2) 虚拟机中安装相应 virtio 设备的驱动程序<sup>[47]</sup>。

虚拟机是 Linux 系统时，只需按上述方法加载对应 virtio 设备的内核模块即可。虚拟机是 windows 系统时，可到网站下载对应的 virtio 设备驱动，网址为：<https://docs.fedoraproject.org/en-US/quick-docs/creating-windows-virtual-machines-using-virtio-drivers/index.html>。最新版本为 virtio-win-0.1.160.iso，稳定版本为 virtio-win-0.1.141.iso。包括了 window10 等最新系统的 32 位和 64 位的驱动程序。将该 iso 文件加载到虚拟机的光盘中，安装对应驱动程序即可<sup>[49]</sup>。

(3) 在 KVM 虚拟化平台中的虚拟机上启用 virtio 半虚拟化服务。以配置磁盘和网卡为例，先关闭虚拟机，修改 xml 配置文件，加载测试磁盘，挂载驱动程序 iso 文件，开启虚拟机并在系统中安装 virtio 磁盘和网卡驱动程序。用到的命令和配置如下：

新增测试磁盘：qemu-img create -f qcow2 test.img 5G

修改 win2008-test3.xml 配置文件：加载新建的磁盘镜像文件 test.img，修改 disk 元素中的属性 dev='vdc'，bus='virtio'，增加 cache='writeback'。修改网卡的 type 属性 type='virtio'。

```
<disk type='file' device='disk'>
```

```
<driver type='qcow2' />
<source file='/home/win2008-test3.img'/>
<target dev='hda' />
</disk>
<disk type='file' device='cdrom'>
  <source file='/home/virtio-win-0.1.160.iso'/>
  <target dev='hdb'/>
  <readonly/>
</disk>
<disk type='file' device='disk'>
  <driver type='qcow2' cache='writeback'/>
  <source file='/home/test.img'/>
  <target dev='vdc' bus='virtio'/>
</disk>
<interface type='bridge'>
  <source bridge='br0'/>
  <model type='virtio'/>
</interface>
```

修改并保存配置文件后启动虚拟机，系统将识别到新的磁盘和网卡类型，安装好 virtio 磁盘及网卡驱动程序，再次关闭虚拟机，删除测试磁盘，虚拟机磁盘镜像对应的 xml 修改成测试磁盘一样的属性值：

```
<disk type='file' device='disk'>
  <driver type='qcow2' cache='writeback'/>
  <source file='/home/win2008-test3.img'/>
  <target dev='vda' bus='virtio'/>
</disk>
```

再次开机，完成 windows 下 virtio 半虚拟化设备驱动的启用，虚拟机操作系统的磁盘和网卡在设备管理器中将更新为 virtio 半虚拟化设备。此时，虚拟机知道了自己处于虚拟化环境之中。

在网卡 virtio 半虚拟化中，Linux 内核还提供一个 vhost-net 后端驱动模块。



虚拟机通过网桥模式使用宿主机网卡时，如果启用 `vhost-net` 服务，那么虚拟机的所有网络请求传递给 `virtio` 后端处理程序时，由原来在用户空间的 `QEMU` 进程执行，改为在内核空间的 `vhost-net` 进程来处理，能达到更高的网络吞吐量和更少的网络延迟，使网卡收发效率更高，但如果客户机的处理能力跟不上内核 `vhost-net` 进程的处理速度时，网络性能将不升反降。因此，在实际应用时需要根据具体业务做改进。可在虚拟机配置文件中修改相关设置来启动和关闭 `vhost-net` 服务，默认是开启，具体修改操作如下：

```
<interface type='bridge'>
    <source bridge='br0'>
    <model type='virtio'>
    <driver name='vhost'> #开启服务，关闭时改为<driver name='qemu'>
</interface>
```

磁盘和网卡是虚拟机 I/O 性能优化的主要对象，直接反应虚拟机的性能体验，因此，在 KVM 虚拟平台用于生产环境时，`virtio` 半虚拟化是必须实施的 I/O 性能改进策略。

## 6.2 设备直接分配

KVM 虚拟机的 I/O 访问可分成三种类型，一是纯软件模拟的设备访问，由 `QEMU` 直接处理，属于完全虚拟化类型。二是 `virtio` 为主的半虚拟化设备访问，由 `QEMU` 代理或内核使用 `virtio` 后端处理方式加速访问，属于半虚拟化类型<sup>[14]</sup>。三是采用硬件芯片支持的虚拟化方案，由支持 `Intel VT-d` 和 `SR-IOV` 标准的硬件处理设备访问，属于 KVM 下的硬件虚拟化类型。三种类型的性能表现从理论上判断是越来越好。但 KVM 虚拟机的性能优化还要结合平台兼容、可移植等因素，所以，硬件支持的 I/O 虚拟化应成为 KVM 虚拟机性能改进的有效补充，在应用时可视具体情况而定。

`Intel VT-d` 提供了硬件 I/O 性能优化的支持，该技术可将物理设备直接分配给 KVM 虚拟机使用，主要应用在 PCI 总线设备如网卡、磁盘和 USB 设备上<sup>[13]</sup>。首先，确定物理服务器硬件是否支持 `Intel VT-d`，如支持，通常可在 BIOS 中打开该支持。`Linux` 内核默认已打开支持选项。然后，通过 `QEMU` 命令行或 `libvirt` 的 `xml` 配置，分配设备给虚拟机。最后，在宿主机上隐藏硬件设备<sup>[51]</sup>，防止其他程序再占用。

### 6.2.1 网卡直接分配

直接分配技术用得最多的是网卡直接分配，一台 HP580G7 服务器有 4 个千兆/万兆自适应电口，都使用 PCI 总线提供网络服务，这种场景下，可能将多余的网卡直接分配给 KVM 虚拟机以提供最佳的网络 I/O 性能。实现步骤和配置方法如下：

(1) 在 Linux 中使用 `lspci` 命令查看物理机网卡设备信息，`virsh nodedev-list -tree` 查看更详细信息，本文实验服务器的网卡信息如图 6-3 所示：

```
02:00.0 Ethernet controller: Emulex Corporation OneConnect 10Gb NIC
02:00.1 Ethernet controller: Emulex Corporation OneConnect 10Gb NIC

+- pci_0000_00_03_0
|
| +- pci_0000_02_00_0
| | +- net_enp2s0f0_78_e3_b5_02_75_88
| |
| +- pci_0000_02_00_1
| | +- net_enp2s0f1_78_e3_b5_02_75_8c
| |
| +- pci_0000_02_00_2
```

图 6-3 物理服务器 PCI 网卡设备信息

(2) 通过 `virsh nodedev-dumpxml pci_0000_02_00_0` 命令得到 libvirt 管理下该设备可被虚拟机识别的 xml 配置信息，配置如图 6-4 所示：

```
[root@localhost home]# virsh nodedev-dumpxml pci_0000_02_00_0
<device>
  <name>pci_0000_02_00_0</name>
  <path>/sys/devices/pci0000:00/0000:00:03.0/0000:02:00.0</path>
  <parent>pci_0000_00_03_0</parent>
  <driver>
    <name>be2net</name>
  </driver>
  <capability type='pci'>
    <domain>0</domain>
    <bus>2</bus>
    <slot>0</slot>
    <function>0</function>
    <product id='0x0710'>OneConnect 10Gb NIC (be3)</product>
    <vendor id='0x19a2'>Emulex Corporation</vendor>
    <capability type='virt_functions' maxCount='8' />
    <pci-express>
      <link validity='cap' port='0' speed='5' width='8' />
      <link validity='sta' speed='5' width='8' />
    </pci-express>
  </capability>
</device>
```

图 6-4 libvirt 下 PCI 网卡 xml 配置

(3) 在虚拟机对应的 xml 配置信息中加入直接分配该 PCI 网卡设备的配置，具体配置项如下：

```
<hostdev mode='subsystem' type='pci' managed='yes'>
  <source>
    <address domain='0x0000' bus='0x02' slot='0x00' function='0x0' />
```

```
</source>
```

```
</hostdev>
```

上述 domain、bus、slot、function 的值可从图 6-5 中对应的选项中取得。然后使用 libvirt 命令 `virsh create win2008-test3.xml` 启动虚拟机即可。如果是通过 QEMU 命令行启动虚拟机，那么需要在启动命令中加入以下参数：

```
-device pci-assign,host=02:00.0,id=be2net -net none
```

因为设备直接分配是独占该设备<sup>[52]</sup>，占用的虚拟机性能虽然可以提高，但无助于整个虚拟平台 I/O 性能的优化。在使用时除非虚拟机有特殊要求需要这么做，否则，从物理资源利用率和虚拟化可移植值方面考虑，不推荐此种方式改进 KVM 虚拟机性能。

### 6.2.2 硬盘直接分配

硬盘的访问性能也是优化的重点，同样在 virtio-blk 半虚拟化驱动不能满足特殊虚拟机的应用要求时可以采用直接分配方案。实现方法和步骤与网卡分配一样。同样的理由，相比网卡的直接分配，在 KVM 虚拟机性能改进方案中，更不推荐硬盘直接分配方案。

### 6.2.3 USB 直接分配

USB 在物理服务器上也是以 PCI 总线提供服务，因此也能使用上述方法实现直接分配。USB 设备以独占方式访问，不需迁移，即插即用。所以，直接分配是 KVM 虚拟化中使用 USB 的最佳方案。本章以另一种方式，实现 USB 的直接分配，方法和步骤如下：

- (1) 查看物理服务器上插入的 USB 设备信息，如图 6-5 所示：

```
root@ubuntu-utlz-2:~# lsusb
Bus 005 Device 002: ID 04b4:4a59 Cypress Semiconductor Corp.
Bus 006 Device 002: ID 03f0:7029 Hewlett-Packard
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 003 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 004 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 005 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 006 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
```

图 6-5 物理服务器中 USB 设备信息

- (2) 用 QEMU 命令行启动虚拟机时加入以下参数：

```
-usbdevice host:04b4:4a59
```

或者使用 libvirt 启动虚拟机，在虚拟机配置文件中加入以下标签：

```

<hostdev mode='subsystem' type='usb'>
  <source startupPolicy='optional'>
    <vendor id='0x04b4'>
      <product id='0x4a59'>
    </source>
  </hostdev>

```

这种方式不同于将整个 PCI 的 USB Host Controller 分配给虚拟机,而是将一个单独的 USB 设备进行分配。在实验中,分配了达实智能的 C3V2006 门禁管理软件的 USB 加密狗给业务虚拟机。分配情况如图 6-6 所示:

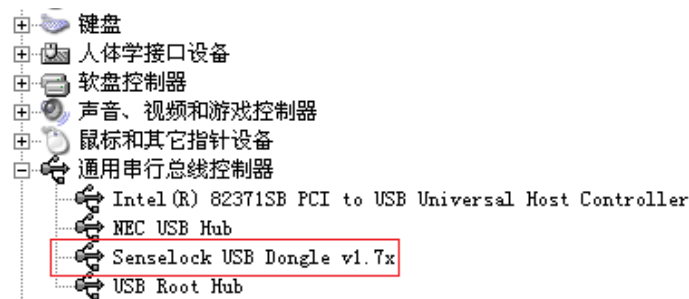


图 6-6 USB 直接分配给虚拟机界面

### 6.3 网络访问优化技术 (SR-IOV)

KVM 虚拟机的 I/O 硬件虚拟化除 Intel VT-d 技术外,还有 SR-IOV 技术应用得较多,是单根输入输出虚拟化的简称,虚拟化原理是将一个 PCI-E 设备共享给虚拟机使用。该技术主要用在网卡的性能优化上,需要网卡硬件支持<sup>[53]</sup>。

SR-IOV 标准定义了 PCI-E 设备的两种功能<sup>[51]</sup>,分别是 Physical Function(PF, 物理功能)和 Virtual Function(VF, 虚拟功能)。PF 管理 PCI-E 设备和相应的 VF 设备,本身也可以当做一个 PCI-E 设备使用,一个设备只能有一个 PF。VF 将 PCI-E 设备虚拟成多个只有数据传输功能的部件,一个 VF 设备就成了一个只带数据传送功能的 PCI-E 设备,一个设备可以虚拟出多个 VF。

设备启用 SR-IOV 功能后,多个 VF 可以通过直接分配方式,给 KVM 虚拟机独占使用,使虚拟机绕过 QEMU(虚拟化层)和系统内核,直接访问 VF 虚拟的硬件设备。由此可知,VF 才是真正的硬件虚拟化,能减少硬件设备数量提高设备使用率,体现了虚拟化的节能减排思想。而直接分配只能看成支持虚拟化环境

访问。

以网卡的 SR-IOV 实现为例，通常一个千兆网卡可以虚拟出 8 个 VF，万兆网卡可以虚拟出 64 个 VF，目前大多数服务器的网卡都支持 SR-IOV 技术。实现方法和步骤如下：

(1) 内核中加载网卡驱动模块并指定每个 PCI 网卡虚拟的 VF 数量

`modprobe igb max_vfs=7` #对应千兆网卡

`modprobe ixgbe max_vfs=20` #对应万兆网卡

(2) `lspci` 查看是否出现 VF 网卡配置。

(3) 按设备直接分配方法使用虚拟出来的 VF 网卡。

本章从改进 KVM 虚拟机 I/O 性能出发，介绍了 virtio 半虚拟化技术和硬件虚拟化技术的实际应用，验证了优化 I/O 性能的具体方法。

## 6.4 本章小结

QEMU 本身是一个集 I/O 设备、CPU、内存等各类硬件模拟功能于一体的支持完全虚拟化的模拟软件。在 KVM 虚拟化方案中仅使用了 QEMU 的 I/O 模拟设备处理 I/O 请求。本章首先分析软件完全虚拟化 I/O 服务与半虚拟化驱动在性能上的差距，明确了虚拟机要获得较好的 I/O 性能就必须抛弃 QEMU 提供的完全虚拟化下的模拟设备驱动。提出目前 KVM 虚拟机 I/O 性能优化的解决方案是以 virtio 设备驱动标准下的半虚拟化 I/O 驱动为主。

然后，研究 virtio 半虚拟化的设备访问原理和实现过程。介绍 Intel VT-d 和 SR-IOV 标准下的硬件支持 I/O 虚拟化技术。分析 I/O 性能表现，提出了 I/O 虚拟化技术的应用场景和以具体业务相结合实施 I/O 优化的思想。

最后，通过实验，给出了 virtio、Intel VT-d 和 SR-IOV 标准下虚拟机 I/O 性能优化的实现方法和步骤。

## 第 7 章 性能优化测试与结果分析

本章将通过实验对性能优化方案的可行性进行测试，分析结果数据。所有实验在进程调度、内存使用、I/O 处理三个方面同时优化的情况下进行，通过 `super pi` 来测试 KVM 虚拟机调度优化方案，`LMbench3` 内存检测工具测试优化后宿主服务器与虚拟机中内存带宽和访问延迟的指标数据，`scp/pscp` 测试网络 I/O 性能，`cp` 脚本测试磁盘 I/O 性能。最后，结合第三种优化思路分析 KVM 虚拟机运行时宿主机 CPU 的上下文切换数据，比较优化效果。

### 7.1 测试环境

实验测试主要涉及 2 台 KVM 宿主服务器，4 台虚拟机，centos7 和 windows2008 为 64 位系统，KVM 版本与内核版本一致，QEMU 版本是 2.12.91，libvirt 版本是 4.5.0，具体配置情况如表 7-1 所示：

表 7-1 实验设备配置表

设备名称	配置	操作系统/内核 版本	Ip 地址	测试环境
Dell R710 服务器	E5520/2.27GHz/2x8 核/10G 网卡/16G 内存/1T 硬盘	Centos7 64 位 /4.17.10	219.223.254.2 21	KVM/QEMU/libvirt
HP 580G7 服务器	E7-4830/2.13GHz/2x16 核/10G 网卡/32G 内存/500G 硬盘	Centos7 64 位 /4.19.0-rc3	219.223.254.2 22	KVM/QEMU/libvirt
centos7-test-1 虚拟机	QEMU VCPU1.5.3/2.26GHz/4 核/虚拟网卡 3G 内存/100G 硬盘	Centos7 64 位 /3.10.0	10.6.255.201	配置相关优化， 如 virtio
centos7-test-2 虚拟机	QEMU VCPU1.5.3/2.26GHz/4 核/虚拟网卡 3G 内存/100G 硬盘	Centos7 64 位 /3.10.0	10.6.255.202	未配置优化
Win2008-test-5 虚拟机	QEMU VCPU1.5.3/2.26GHz/4 核/虚拟网卡 3G 内存/200G 硬盘	Windows 2008 64 位	10.6.255.205	配置相关优化， 如 virtio
Centos7-ftp 虚拟机	QEMU VCPU1.5.3/2.13GHz/4 核/虚拟网卡 4G 内存/200G 硬盘	Centos7 64 位 /3.10.0	10.6.255.195	配置相关优化， 如 virtio

上表中虚拟机 centos7-test-1 和 centos7-test-2 配置完全相同，前者配置相关优化，后者未配置优化，用于模拟同一台虚拟机在优化前后两种状态下的测试效果。

## 7.2 KVM 虚拟机调度优化测试

首先，在虚拟机 Win2008-test-5 上执行 super pi 测试程序，运行时设置圆周率到小数点后 2 的 21 次方（2M）个数据位，该程序为 CPU 密集型测试。然后，采用进程调度优化方案，在宿主服务器 Dell R710 上将虚拟机 win2008-test-5 对应的进程由普通调度策略调整为 FIFO 实时调度策略，优先级设置成 99，再次执行 super pi 测试程序。通过在同一台虚拟机不同调度策略和优先级下的三次测试，取平均值相比较来评估调度优化的效果。计算所花费的平均时间越少说明性能越好。具体测试截图如图 7-1、图 7-2、图 7-3 所示：

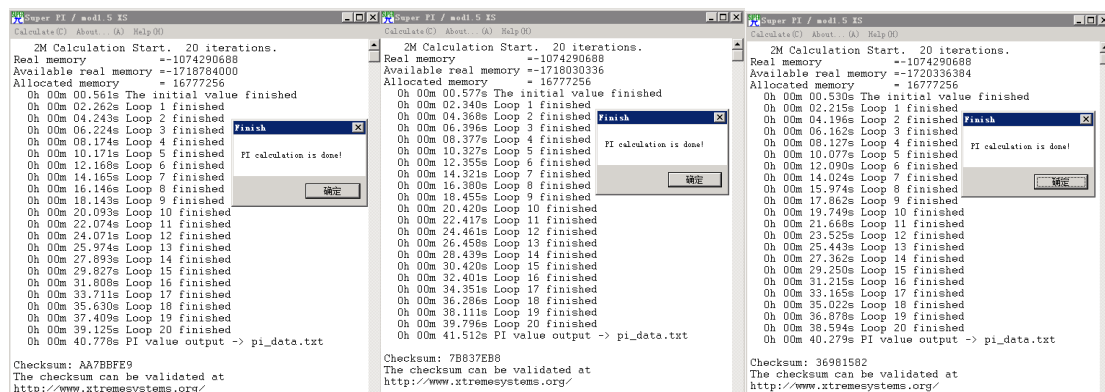


图 7-1 虚拟机 win2008-test-5 为普通进程及默认静态优先级的测试用时

```
[root@localhost myLKM]# ./setschedulerFromPid 11554 1 99
调度策略取值:
#define SCHED_NORMAL      0
#define SCHED_FIFO        1
#define SCHED_RR          2
#define SCHED_BATCH       3
进程实时优先级范围: 0 (低优先级) ~ 99 (高优先级)
调度策略设置成普通策略时, 运行优先级对应实时优先级, 只能设置成0
程序 ./setschedulerFromPid 输入的进程pid= 11554, 将设置该进程的调度策略 policy = 1, 设置运行优先级prio= 99
设置结果: 进程 pid = 11554, 调度策略policy= 1, 运行优先级prio = 99
```

图 7-2 调整为 FIFO 实时调度策略及高优先级截图

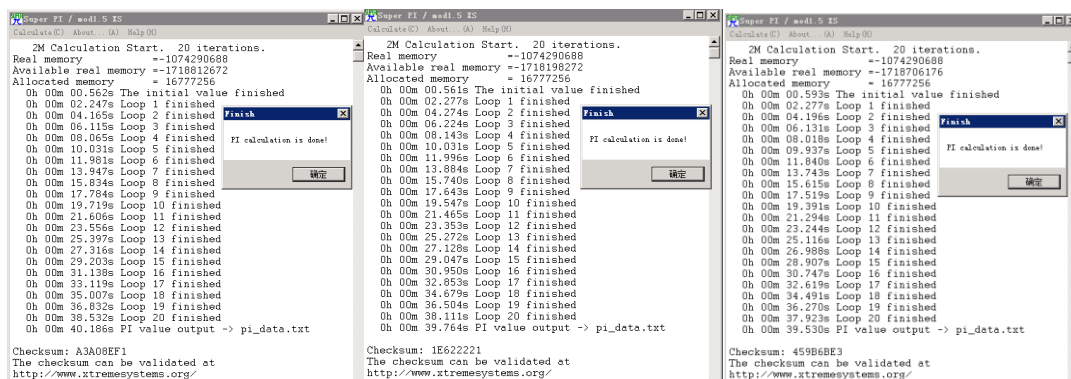


图 7-3 虚拟机 win2008-test-5 为 FIFO 实时进程及 99 动态优先级的测试用时

由以上测试数据可计算虚拟机在两种调度策略下的平均计算用时，具体的 super pi 测试数据对比如表 7-2 所示：

表 7-2 super pi 测试数据对比表(单位秒)

进程类型	第一次计算用时	第二次计算用时	第三次计算用时	平均用时
CFS 普通进程	40.778	41.512	40.279	40.85633
FIFO 实时进程	40.186	39.764	39.53	39.82667

显然，虚拟机进程修改成 FIFO 实时进程后，相同计算基准下的计算用时要少于普通进程的计算用时。当宿主服务器负载较大时，这种差距将更大。实验证明，合理使用虚拟机调度优化方案，能够提高物理计算资源利用率，并改进虚拟机的运行性能，该优化方案具有可行性。

### 7.3 内存使用优化测试

Lmbench 是一款内存性能测试工具，实验选择 HP 580G7 宿主服务器和其上运行的虚拟机 centos7-ftp 做对比，设置相同的测试基准，在虚拟机内存使用优化环境下测试 1G 内存在使用带宽 (bandwidth) 和延时 (latency) 两个方面的表现。测试结果如图 7-4 所示：

```
*Local* Communication bandwidths in MB/s - bigger is better
-----
Host      OS      Pipe AF      TCP  File  Mmap  Bcopy  Bcopy  Mem  Mem
          UNIX  reread reread (libc) (hand) read write
-----
localhost Linux 3.10.0-      1860.6 1717.2 3360 2595.
localhost Linux 4.19.0-      1703.4 1631.0 3021 2423.

Memory latencies in nanoseconds - smaller is better
(WARNING - may not be correct, check graphs)
-----
Host      OS      Mhz  L1 $  L2 $  Main mem  Rand mem  Guesses
-----
localhost Linux 3.10.0- 2131 1.8150 4.7480 58.4      185.8
localhost Linux 4.19.0- 2131 1.7760 4.5740 87.1      153.4
make[1]: 离开目录"/root/lmbench3/results"
[root@localhost lmbench3]#
```

图 7-4 宿主机与虚拟机内存使用性能对比测试

上图中 linux 内核版本 3.10.0 对应的是虚拟机 centos7-ftp，内核版本 4.19.0 对应的是 HP 580G7 宿主服务器。可以看出使用带宽方面虚拟机略优于宿主机，这是因为还有其他虚拟机在使用内存，造成一些影响，而在访问延时上，宿主机少于虚拟机。综合比较，两者在内存使用性能上很接近，优化方案实施后，虚拟机内存使用性能可以接近或达到物理内存的效率。



## 7.4 I/O 优化测试

### 7.4.1 网络访问性能测试

实验主要测试 virtio 半虚拟化优化方案下网络 I/O 的性能表现。在 linux 系统中使用 scp，windows 系统中使用 pscp 命令，远程复制传送一个 7GB 大小的文件到虚拟机和宿主机上，通过从相同源到不同主机的方式比较网络 I/O 访问性能。实验中将待复制的源文件放在 HP 580G7 服务器上，其他主机从源服务器复制数据，传输速率如图 7-5 所示：

```
[root@localhost ~]# scp centos7-test-5.img root@219.223.254.221:/root
root@219.223.254.221's password:
centos7-test-5.img                                100% 7097MB  89.0MB/s   01:19
[root@localhost ~]# scp centos7-test-5.img root@10.6.255.201:/root
root@10.6.255.201's password:
centos7-test-5.img                                100% 7097MB  49.9MB/s   02:22
[root@localhost ~]# scp centos7-test-5.img root@10.6.255.202:/root
root@10.6.255.202's password:
centos7-test-5.img                                100% 7097MB  26.6MB/s   04:26
```

图 7-5 网络 I/O 访问性能测试数据

从测试数据可知，virtio 半虚拟化技术应用后虚拟机的网络 I/O 性能较接近物理服务器（虚拟机配置仅物理服务器的四分之一），可满足 WEB 或一般文件服务器的网络访问需求。未优化的虚拟机网络访问性能只有物理服务器的三分之一，可用于测试业务。而设备直接分配优化方案则可满足更高的性能要求，达到物理服务器性能。

### 7.4.2 磁盘读写性能测试

实验主要测试 virtio 半虚拟化优化方案下磁盘 I/O 访问的性能表现。设计一个基于 cp 命令的脚本来测试磁盘 I/O 性能，脚本名为 cptest.sh，在测试主机上运行该脚本，使主机在本地重复 6 次复制一个 7GB 大小的文件，通过比较复制速率来评估磁盘读写性能。分别在 Dell R710 服务器、虚拟机 centos7-test-1 和 centos7-test-2 上进行了实验，具体测试数据如图 7-6 所示：

```
[root@localhost ~]# ./cptest.sh
cptest 1
0 directories 1 files 6.93GB detected.
0 directories 1 files 6.93GB copied, 2m 9s, 55.02MB/s.
cptest 2
0 directories 1 files 6.93GB detected.
0 directories 1 files 6.93GB copied, 2m 18s, 51.43MB/s.
cptest 3
0 directories 1 files 6.93GB detected.
0 directories 1 files 6.93GB copied, 2m 14s, 52.96MB/s.
cptest 4
0 directories 1 files 6.93GB detected.
0 directories 1 files 6.93GB copied, 2m 8s, 55.45MB/s.
cptest 5
0 directories 1 files 6.93GB detected.
0 directories 1 files 6.93GB copied, 2m 7s, 55.88MB/s.
cptest 6
0 directories 1 files 6.93GB detected.
0 directories 1 files 6.93GB copied, 2m 6s, 56.33MB/s.
[root@localhost ~]#

[root@localhost home]# ./cptest.sh
cptest 1
0 directories 1 files 6.93GB detected.
0 directories 1 files 6.93GB copied, 3m 1s, 39.21MB/s.
cptest 2
0 directories 1 files 6.93GB detected.
0 directories 1 files 6.93GB copied, 2m 58s, 39.87MB/s.
cptest 3
0 directories 1 files 6.93GB detected.
0 directories 1 files 6.93GB copied, 2m 55s, 40.55MB/s.
cptest 4
0 directories 1 files 6.93GB detected.
0 directories 1 files 6.93GB copied, 2m 53s, 41.02MB/s.
cptest 5
0 directories 1 files 6.93GB detected.
0 directories 1 files 6.93GB copied, 2m 55s, 40.55MB/s.
cptest 6
0 directories 1 files 6.93GB detected.
0 directories 1 files 6.93GB copied, 2m 55s, 40.55MB/s.
[root@localhost home]#

[root@localhost home]# ./cptest.sh
cptest 1
0 directories 1 files 6.93GB detected.
0 directories 1 files 6.93GB copied, 4m 12s, 28.16MB/s.
cptest 2
0 directories 1 files 6.93GB detected.
0 directories 1 files 6.93GB copied, 4m 34s, 25.90MB/s.
cptest 3
0 directories 1 files 6.93GB detected.
0 directories 1 files 6.93GB copied, 4m 6s, 28.85MB/s.
cptest 4
0 directories 1 files 6.93GB detected.
0 directories 1 files 6.93GB copied, 4m 5s, 28.97MB/s.
cptest 5
0 directories 1 files 6.93GB detected.
0 directories 1 files 6.93GB copied, 4m 3s, 29.21MB/s.
cptest 6
0 directories 1 files 6.93GB detected.
0 directories 1 files 6.93GB copied, 4m 33s, 26.00MB/s.
[root@localhost home]#
```

图 7-6 磁盘 I/O 访问性能测试数据

根据测试数据得到磁盘读写性能对比如表 7-3 所示：

表 7-3 cp 命令磁盘读写性能测试数据对比表（单位 MB/s）

设备名称	第一次 速率	第二次 速率	第三次 速率	第四次 速率	第五次 速率	第六次 速率	平均速率
Dell R710 宿主机	55.02	51.43	52.96	55.45	55.88	56.33	54.51167
centos7-test-1 虚拟机	39.21	39.87	40.55	41.02	40.55	40.55	40.29167
centos7-test-2 虚拟机	28.16	25.9	28.85	28.97	29.21	26	27.84833

可以看到, virtio 半虚拟化技术应用后虚拟机具有接近物理服务器的磁盘 I/O 访问性能, 可满足多数业务系统的磁盘读写应用需求。未优化的虚拟机磁盘 I/O 性能只有物理服务器的一半, 可用于测试业务。而设备直接分配优化方案则可满足更高的性能要求, 达到物理服务器性能。

## 7.5 VCPU 陷入事件捕获数据与优化分析

结合第三种优化思路, 捕获 VCPU 陷入事件触发时统计的数据, 主要是 KVM 虚拟机运行过程中宿主机 CPU 的上下文切换数据, 通过比较 CPU 三种模式切换的次数来分析实施优化方案的效果。实验中, 以虚拟机 centos7-test-1 和 centos7-test-2 为例, 前者进行性能优化, 后者不做优化, 相同配置的两台虚拟机, 运行 1 天后, 其上下文切换数量的统计情况如图 7-7 所示:

```
[root@localhost ~]# ps -ef |grep qemu |grep centos7-test-1
root      15749      1  3 11月21 ?        00:34:18 /usr/bin/kvm -name centos7-test-1

[root@localhost ~]# ps -ef |grep qemu |grep centos7-test-2
root      15795      1  3 11月21 ?        00:35:00 /usr/bin/kvm -name centos7-test-2

[root@localhost ~]# cat /proc/kvminfo
序号      PID      io_num      exception_num      kvm_vm_switch_num      kvm_qemu_switch_num
1          15749      2336976      0                  30511935                217136
2          15795      15146219    0                  34481937                22130431
```

图 7-7 统计 VCPU 陷入异常的上下文切换数量

参考第 3 章对切换数据进行定量分析可得具体数据如表 7-4 所示:

表 7-4 两台虚拟机切换数据定量分析情况

PID	io_num	kvm_vm_swit tch_num	kvm_qemu_s witch_num	总切换次数	虚拟机与 KVM 之 间切换的百分比	I/O 切换的 百分比
15749	2336976	30511935	217136	30729071	0.992933857	0.076051
15795	15146219	34481937	22130431	56612368	0.609088406	0.267543

由第 3 章介绍可知，第一种切换类型占比越大性能开销越小，第二、第三种切换类型占比越小性能开销也越小，上表中分别对应虚拟机到 KVM 之间切换百分比、I/O 切换百分比。因此，已优化虚拟机 centos7-test-1 两项占比分别为 99% 和 7%，非常接近物理服务器的性能，未做优化的虚拟机 centos7-test-2 两项占比分别为 60% 和 26%，大约只有物理服务器一半的性能表现。

## 7.6 本章小结

本章根据前几章 KVM 虚拟机性能研究进行了实验，通过优化前后的测试数据来对比虚拟机同时在 CPU 进程调度、内存使用、I/O 处理三个方面性能改进的效果，验证三种性能优化思路下的各优化设计方案的可行性。

## 第 8 章 总结与展望

现代社会科技发展日新月异，每种技术的起源、背景、基础、特点和前景将决定技术的发展速度和方向。在信息化领域，计算资源的虚拟化已成为当前计算机世界发展的一项主流技术。本文以开源的 KVM 虚拟化解方案为研究对象，介绍了 KVM 的起源，发展背景，深入分析 KVM 虚拟化的硬件支持技术，以及与之对应的软件驱动原理与运行机制，从 KVM 虚拟机的性能研究与改进角度，开展具体研究和实验，得出以下结论：

（1）采用开源 Linux 作为 VMM，在具体的 Centos7 系统上编译最新内核，并修改和调整内核中影响 KVM 性能的功能模块，安装最新版本 I/O 模拟软件 QEMU 和虚拟机管理软件 libvirt，构建了经过环境优化的 KVM 虚拟化平台。该平台充分体现了节能减排理念，以低成本实现物理资源高利用率，可成为 IT 从业者的测试或工作平台，也可成为中小企业数据中心的计算资源生产平台。完成了 KVM 虚拟化平台环节的性能优化。

（2）研究了 Intel x86 服务器上的硬件辅助完全虚拟化（Intel VT-x 技术）的实现原理和分析 KVM 内核模块的软件驱动原理和运行机制，在深入理解虚拟机运行模式的基础上，提出基于处理器角度的 KVM 虚拟机性能优化思路，设计和实现了一套 VCPU 陷入事件的捕获方案，通过实验，对事件信息进行统计，针对第三种思路，验证了改进的可行性。

（3）在深入研究 Linux 进程管理、调度策略、运行优先级对进程占用处理器资源的原理和执行方式基础上，提出了 KVM 虚拟机调度优化方案，并设计和实现了一个进程监控程序，结合实时监控，可针对业务特点灵活改变虚拟机的调度策略和运行优先级，使虚拟机能够按实际需求获得更多占用物理处理器资源的机会，改进 KVM 虚拟机的整体性能。实验验证，该优化方案具有可行性。

（4）从优化内存的使用率和访问效率两方面，研究优化内存使用的三种主要技术，根据物理资源环境和业务系统需求提出内存优化方案和适应的场景，并给出了具体的实现方法和步骤。

（5）深入分析虚拟机的三种 I/O 访问类型，研究以 virtio 为主的半虚拟化设备访问原理和实现过程，确定半虚拟化是 KVM 虚拟机 I/O 访问性能改进的主要方案。提出了硬件直通模型中 I/O 虚拟化技术的应用场景和以具体业务相结合实施 I/O 优化的思想。通过实验，给出了 virtio、Intel VT-d 和 SR-IOV 标准下虚

拟机 I/O 性能优化的实现方法和步骤。

KVM 与 Linux 结合已过 10 年，随着技术不断发展，KVM 虚拟机将越来越稳定，兼容性和性能优化会进一步完善。具有开源、架构简单、支持硬件虚拟化、完全虚拟化、半虚拟化等特点的 KVM 虚拟化解决方案必然成为企业和各类 IT 技术人员可以信赖的虚拟化平台。有了稳定、高效的基础技术，再与桌面虚拟化、分布式文件系统相结合，随着 KVM 虚拟机性能的不断改进，将很可能成为未来云计算 IaaS、SaaS、PaaS 层依赖的主流虚拟化技术，有着广阔的发展和应用前景。

KVM 在当前虚拟化技术中属于较后发展的技术，限于自身所学水平，专业能力和经验不足，本文难免出现一些错漏，希望老师和同学批评指正，我将认真学习和改进。

## 参 考 文 献

- [1] 广小明. 虚拟化技术原理与实现[M]. 北京:电子工业出版社, 2012.
- [2] 任永杰, 单海涛. KVM 虚拟化技术实战与原理解析[M]. 北京:机械工业出版社, 2013.
- [3] 何坤源. Linux KVM 虚拟化架构实战指南[M]. 北京:人民邮电出版社, 2015.
- [4] Chao-Tung Yang, et al. Virtual machine management system based on the power saving algorithm in cloud[J]. Journal of Network and Computer Applications, 2017, 80:165 - 180.
- [5] Varun Kumar Manik, Deepak Arora. Performance Comparison of Commercial VMM:ESXI, XEN, HYPER-V&KVM:2016 3rd International Conference on Computing for Sustainable Global Development, New Delhi, March 16-18, 2016[C]. India:IEEE, 2016.
- [6] Uchit Gandhi, et al. Distributed Virtualization Manager for KVM Based Cluster[J]. Procedia Computer Science, 2016, 79:182-189.
- [7] 徐杏芳. 浅析处理器虚拟优化技术及 KVM 实现[J]. 福建电脑, 2012, 28(5):63-65.
- [8] 黄煜, 罗省贤. KVM 虚拟化技术中处理器隔离的实现[J]. 计算机系统应用, 2012, 1:179-182.
- [9] Luca Abeni, et al. On the performance of KVM-based virtual routers[J]. Computer Communications, 2015, 70:40-53.
- [10] Dordevic B, Macek N, Timcenko V. Performance Issues in Cloud Computing:KVM Hypervisor's Cache Modes Evaluation[J]. Acta Polytechnica Hungarica, 2015, 12(4):147-165.
- [11] 百度百科. 虚拟化技术[EB/OL]. [2017-12-25] [https://baike.baidu.com/item/虚拟化技术/276750?fr=aladdin#4\\_8](https://baike.baidu.com/item/虚拟化技术/276750?fr=aladdin#4_8).
- [12] dylloveyou. KVM 总结 -KVM 性能优化之内存优化[EB/OL]. [2018-8-25] <https://blog.csdn.net/dylloveyou/article/details/71338378>.
- [13] 车翔, 王华军. QEMU-KVM 虚拟 PCI 设备优化方法[J]. 电脑与电信, 2011, 11:57-58.
- [14] Bujor Alexandru-Catalin, Dobre Razvan. 2013 RoEduNet International Conference 12th Edition: Networking in Education and Research, KVM IO profiling[C]. Iasi Romania:IEEE, 2013.
- [15] Linux Kernel. The Linux Kernel Archives [EB/OL]. [2018-8-25] <https://www.kernel.org/>.
- [16] QEMU, KVM, XEN&LIBVIRT. Installation von QEMU und KVM unter Linux [EB/OL]. [2018-8-25] [http://qemu-buch.de/de/index.php?title=QEMU-KVM-Buch/\\_QEMU%2BKVM\\_unter\\_Linux](http://qemu-buch.de/de/index.php?title=QEMU-KVM-Buch/_QEMU%2BKVM_unter_Linux).
- [17] QEMU Docs. QEMU user manual [EB/OL]. [2018-8-25] <https://www.qemu.org/documentation/>.
- [18] OenHan. KVM 源代码分析 1: 基本工作原理[EB/OL]. (2013-12-22) [2018-8-25] <http://oenhan.com/kvm-src-1>.
- [19] Libvirt Docs. Libvirt XML Format[EB/OL]. [2017-12-25] <https://libvirt.org/format.html>.
- [20] KVM Site. Kernel Virtual Machine [EB/OL]. [2018-8-25] <http://www.linux-kvm.org/page/Documents>.
- [21] Edwar Ali, et al. Optimizing Server Resource by Using Virtualization

- Technology[J].Procedia Computer Science, 2015, 59:320-325.
- [22] 彭晓平, 张雪坚, 黄波. 基于 KVM 的虚拟化技术研究[J]. 中国新通信, 2017, 19(20):77-80.
- [23] 房胜, 李旭健, 黄玲, 李哲. 操作系统实践[M]. 北京:清华大学出版社, 2015.
- [24] bootlin. Linux 内核源代码在线文档 [EB/OL]. [2018-8-25] <https://elixir.bootlin.com/linux/latest/source>.
- [25] 肖力, 汪爱伟, 杨俊俊, 赵德禄. 深度实践 KVM[M]. 北京:机械工业出版社, 2017.
- [26] 邢静宇. KVM 虚拟化技术基础与实践[M]. 西安:西安电子科技大学出版社, 2015.
- [27] 黄煜. KVM 虚拟机 CPU 虚拟化的研究与调度策略的优化[D]. 成都:成都理工大学, 2012.
- [28] linuxheik. 虚拟化原理之 KVM [EB/OL]. [2018-8-25] <https://blog.csdn.net/linuxheik/article/details/52121010>.
- [29] 宋翔. 多核虚拟环境的性能及可伸缩性研究[D]. 上海:复旦大学计算机科学技术学院, 2014.
- [30] 吴国伟, 李张, 任广臣. Linux 内核分析及高级编程[M]. 北京:电子工业出版社, 2008.
- [31] 吕斯特, 陈奋. 虚拟化技术指南[M]. 北京:机械工业出版社, 2011.
- [32] 黄秋兰, 李莎, 程耀东, 陈刚. 高能物理计算环境中 KVM 虚拟机的性能优化与应用[J]. 计算机科学, 2015, 1(42):67-70.
- [33] Robert Love. Linux 内核设计与实现(第三版)[M]. 北京:机械工业出版社, 2017.
- [34] Daniel p. bovet, Marco Cesati. 深入理解 Linux 内核(第三版)[M]. 北京:中国电力出版社, 2018.
- [35] 蒋迪. KVM 私有云架构设计与实践[M]. 北京:上海交通大学出版社, 2017.
- [36] Du Jiaqing, Sehrawat Nipun, Zwaenepoel Willy. Performance profiling of virtual machines[J]. ACM SIGPLAN Notices, 2011, 46(7):3-14.
- [37] 杨宗德, 吕光宏, 刘雍. Linux 高级程序设计(第三版)[M]. 北京:人民邮电出版社, 2012.
- [38] 陈锐, 华庆一, 耿国华, 张永新, 常言说. 最新 C/C++函数与算法速查速用大辞典[M]. 北京:中国铁道出版社, 2015.
- [39] 邱铁, 陈晨, 周玉. Linux 内核 API 完全参考手册[M]. 北京:机械工业出版社, 2016.
- [40] 郑阿奇, 孙承龙. Linux 内核精析[M]. 北京:电子工业出版社, 2017.
- [41] Neil Matthew, Richard Stones. Linux 程序设计(第三版)[M]. 北京:人民邮电出版社, 2008.
- [42] 杜永文, 李金玉, 兰丽. Linux 开发及使用详解[M]. 兰州:兰州大学出版社, 2016.
- [43] Jonatban Corbet, Alessandro Rubini, Greg Kroab-Hartman. Linux 设备驱动程序(第三版)[M]. 北京:中国电力出版社, 2018.
- [44] Robert Love. Linux 系统编程(第二版)[M]. 北京:人民邮电出版社, 2018.
- [45] 执假以为真. 《KVM 虚拟化技术实战和原理解析》读书笔记(三) [EB/OL]. [2018-8-25] <https://blog.csdn.net/nirendao/article/details/52988799>
- [46] 李绍, 罗省贤. 一种基于内存虚拟化技术的 EPT 机制优化方法[J]. 电脑与电信, 2011, 2:52-54.
- [47] 杜炜. KVM 客户机主动共享的内存超量使用策略研究[D]. 杭州:杭州电子科技大学, 2013.
- [48] 朱爽. 基于 KVM 虚拟化技术的研究与实验评估[D]. 沈阳:东北大学硕士论文, 2014.
- [49] fedora Docs. Creating Windows virtual machines using virtIO drivers [EB/OL]. [2018-8-25] <https://docs.fedoraproject.org/en-US/quick-docs/creating-windows-virtual-machin>

es-using-virtio-drivers/index.html.

[50] 车翔. QEMU\_KVM 设备虚拟化研究与改进[D]. 成都:成都理工大学, 2012.

[51] 彭春洪, 刘丹. 一种基于 KVM 虚拟机的隐藏进程检测算法[J]. 小型微型计算机系统, 2016, 37(02):231-235.

[52] 杨洪波. 高性能网络虚拟化技术研究[D]. 上海:上海交通大学电子信息与电气工程学院, 2012.

[53] 孟秉能, 时佃兴. 用 KVM 虚拟化网络服务[J]. 网络安全和信息化, 2016, 9:92-96.



## 致 谢

时间转瞬即逝，三年的研究生学习即将结束。在论文定稿，靠背沉思之时，周末和晚上学习的场景历历在目，诸多感慨纷至沓来，却无法诉诸言语。回顾这三年的学习，似乎又回到了在学校的生活，让人回忆往昔，感慨时间流逝。在此，向帮助、陪伴和关心我的所有人先说一声感谢。

本论文完成的前后，梁正平老师的指导和帮助发挥了很大的作用。开题时在办公室帮我理清思路，启发我确定论文的方向和研究的内容。中期检查时，要求严格，肯定研究内容的同时，细心指出论文的难点、要点和不足，给了我继续研究的动力。提交初稿后，又帮我一页页审核圈点，提出针对性极强的修改意见。正是老师的严格要求，使我更深入的理解了专业知识，培养了严谨的学习态度，提升了论文质量。

我首先要向梁老师表达尊敬之心，是他对我论文写作的悉心指导，才有论文成稿的从容。还有三年来教导过我课程的老师，是他们孜孜不倦的传授知识，给了我应用计算机技术的自信。在这里，我真心向老师们说一声感谢。

其次，十分感谢深圳市科技图书馆，在那里我借阅了大量专业书籍，学到了很多专业知识，为完成论文提供很大助力。感谢我的家人，正是因为他们的理解和支持，我才有更多的时间，全身心的投入到学习、阅读，以及实验上，让毕业论文最终顺利地完成。

最后，向在百忙之中评审本文的专家和老师表示真挚的感谢！