学位论文原创性声明

本人郑重声明：所呈交的论文是本人在导师的指导下独立进行研究所取得的研究成果。除了文中特别加以标注引用的内容外，本论文不包括任何其他个人或集体已经发表或撰写的成果作品。本人完全意识到本声明的法律后果由本人承担。

作者签名：　　年　月　日

学位论文版权使用授权书

本学位论文作者完全了解学校有关保障、使用学位论文的规定，同意学校保留并向有关学位论文管理部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权省级优秀学士论文评选机构将本学位论文的全部或部分内容编入有关数据进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

本学位论文属于 1、保密□，在　　年解密后适用本授权书

2、不保密□。

（请在以上相应方框内打“√”）

作者签名：　　年　月　日

导师签名：　　年　月　日

摘　要

近年来云计算技术发展迅速，而虚拟化技术作为云计算技术的基础技术在其中起到了关键作用。通过虚拟化技术，多个虚拟机运行在一个主机上，按需分割硬件资源，供多个用户使用，提高了硬件资源的利用率。在虚拟化技术当中，因为 I/O 设备的多样性，I/O 设备的虚拟化是较为复杂的一项，根据不同的虚拟化方式又有着不同的优势和劣势。目前 I/O 设备虚拟化主要有三种方式：设备接口完全虚拟化、前后端虚拟化、硬件辅助虚拟化。硬件辅助虚拟化性能最高，但是硬件设备无法被共享使用，设备接口完全虚拟化方式性能太低，前后端虚拟化方式性能较高，但需要在客户机中加载特定驱动程序。采用较为高效的设备虚拟化方式对密码卡进行虚拟化可以为客户机提供一个高效的、安全的加解密环境，从而提高客户机中数据的安全性，这在云计算技术当中有较高的利用价值。

为了保证设备的共享性，同时保证虚拟密码卡高效性，文中将采用前后端方式对密码卡进行虚拟化。为了实现多个虚拟机共享使用多个密码卡的系统，针对多虚拟机对应多密码卡设备的问题，基于 HRRN（Highest Response Ratio Next）进程调度算法，提出了一个可修改等待时间对优先级的影响度的调度算法。云计算技术中经常因为负载均衡、设备故障等原因对虚拟机进行动态迁移，虚拟机动态迁移的同时需要将对应设备也进行动态迁移，因此，为了使得虚拟密码卡系统能够应用到云计算技术中，也提供了虚拟密码卡动态迁移的方案，同时为了保证虚拟密码卡动态迁移的安全性，也提出了相应的安全策略。

最后，基于 QEMU-KVM（Quick Emulator Keyboard Video Mouse）的 virtio 机制实现了密码卡虚拟化系统，并对系统进行了功能和性能方面的测试和分析。主要测试方面有：在客户机中的虚拟密码卡的加解密功能、性能测试、调度算法测试和虚拟密码卡动态迁移功能测试。实验结果表明密码卡虚拟化系统能够高效的完成加解密功能，并且支持动态迁移功能，同时还支持调度算法的调整让系统更加适应硬件环境。

关键词：I/O 设备虚拟化；虚拟密码卡；I/O 设备动态迁移；调度算法

Abstract

In recent years, cloud computing technology develops rapidly, and virtualization technology as the basic technology of cloud computing technology plays a key role in it. Through virtualization technology, multiple virtual machines run on one host OS, and hardware resources are divided as needed for multiple users, which improves the utilization of hardware resources. In the virtualization technology, because of the diversity of I/O devices, the virtualization of I/O devices is a more complex one. According to different virtualization methods, there are different advantages and disadvantages. At present, there are three main ways of I/O device virtualization: full device emulation, para-virtualization, and direct I/O. The performance of direct I/O is the highest, but the hardware devices cannot be shared in this way. The performance of full device emulation is too low. And The performance of para-virtualization is high, but specific drivers need to be loaded in the guest OS. Using a efficient device virtualization method to virtualize the encryption card can provide an efficient and secure encryption and decryption environment for the guest OS, and improve the security of data in the guest OS, which has a high utilization value in cloud computing technology.

In order to ensure the sharing of encryption card and efficiency of virtual encryption card, the para-virtualization mode will be used. In order to realize the sharing of multiple encryption cards among multiple virtual machines, this paper proposes a scheduling algorithm which can modify the influence of waiting time on priority based on HRRN(Highest Response Ratio Next) process scheduling algorithm for the problems of multi virtual machines corresponding to multi password card devices. In cloud computing technology, virtual machines are often dynamically migrated due to load balancing or equipment failure. While dynamic migration of virtual machines, corresponding devices need to be dynamically migrated too.

Therefore, in order to make the virtual encryption card system be applied to cloud computing technology, it also provides a dynamic migration scheme of virtual encryption card, In order to ensure the security of dynamic migration of virtual encryption card, the corresponding security strategy is also proposed.

Finally, based on the virtio mechanism of QEMU-KVM(Quick Emulator Keyboard Video Mouse), the virtual system of encryption card proposed in this paper is implemented. The function and performance of the system will be tested and analyzed at the end of the paper. Card function, performance test, scheduling algorithm test and dynamic migration function test of multiple virtual encryption cards are the main testing aspects. Experimental results show that the encryption card virtualization system can efficiently complete the encryption and decryption function. This system supports the dynamic migration function and also the adjustment of the scheduling algorithm to make the system more suitable for the hardware environment.

Key Words： I/O device virtualization, Virtual encryption cards, I/O device dynamic migration, scheduling algorithm

目　录

1 绪论

1.1 课题背景与研究意义

从 2006 年亚马逊 CEO 贝索斯第一次提出关于云存储和云计算的概念到如今，云计算已经经过十余年的发展，并且逐渐成熟了起来。现在计算机硬件发展很快，一个或少数几个用户是很可能无法完全利用一个服务器的资源，所以云服务提供商在对外提供服务时，都会先通过虚拟机将服务器的资源进行分割，根据用户所需提供服务，让多个用户通过虚拟化技术同时使用一个服务器，并且能够互不干扰。

云计算技术使用虽然方便了很多用户，但是也给用户带来了很多安全[1,2]问题，用户的软件和数据都是在他人的硬件资源上的，数据的完整性和安全性与传统的运行环境相比，安全性有所降低。为了数据的安全，对数据进行加密是最普遍的手段，如果仅仅是使用软件进行加解密，安全性不高，因为密钥和数据都会呈现在内存中，如果虚拟机被入侵，数据和密钥很有可能就会被窃取，同时，软件层次的加密操作效率不高。硬件密码卡可以提供高速独占式的加解密服务[3]，对机密性要求很高的组织会使用硬件密码卡来完成加解密运算，当需要进行大规模的加解密运算时，硬件密码卡可以提供软件所不能达到的高性能。因此，为了方便用户的加解密，云计算提供商要提供硬件密码卡，并将其虚拟化到虚拟机里面，为用户所使用的。

为了让虚拟化的设备效率更高，目前多采用 Passthrough I/O[4,5]技术对 I/O 设备进行硬件辅助虚拟化，然而这种将一个硬件设备直接分配给虚拟机的方式会导致一个设备会被一个虚拟机独占使用，导致硬件资源无法在虚拟机之间共享。随后，出现了 SR-IOV（Single Root I/O Virtualization）设备虚拟化技术，SR-IOV 设备有一个或多个 PF（Physical Function），一个 PF 就相当于一个普通的硬件设备，具有标准的 PCIe（Peripheral Component Interconnect Express）设备的功能，一个 PF 对应多个 VF（Virtual Function），每个 VF 都是一个轻量级的 PCIe 设备，利用 Passthrough I/O 技术为虚拟机分配一个 VF 来使用，这样就在一定程度上缓解了硬件设备不够虚拟机使用的问题。但是 VF 的数量还是有限的，当虚拟机的数

目超过 VF 的数目时，依然会出现虚拟机没有可用的硬件设备的问题。还有一种半虚拟化方式，通过前后端通信机制，让虚拟机可以使用硬件密码卡，并且多个虚拟机可以共享同一个硬件密码卡，虽然效率比 Passthrough I/O 技术要低，但是好在可以使硬件设备达到共享的要求。

当服务器群中负载严重不均衡时会影响用户的使用体验，并且硬件资源利用率也会很低，云计算服务中有时为了负载均衡等原因，会将一个服务器的虚拟机迁移到另一个服务器上，给负载过重的服务器减轻负担，为闲置的服务器分配任务，此时虚拟机所拥有的设备也必须完成迁移才能不影响用户的使用，因此设计虚拟化密码卡方案时也要考虑动态迁移的问题。

多密码卡虚拟化以及动态迁移的研究与实现，对需要密码服务的云计算而言有重要的意义和应用价值，高效并且可共享的密码卡虚拟化技术能够在保证用户较好的利用硬件的同时，又能解决虚拟机无法分配到硬件设备的问题。虚拟机迁移功能是云计算中非常重要的功能，解决密码卡的动态迁移问题为整个方案应用在云计算中提供了可能性。

1.2 国内外研究现状

目前国内外也有不少关于设备虚拟化和设备安全动态迁移的研究和方案，但是针对密码卡甚至是多密码卡的虚拟化和动态迁移的研究却很少，与虚拟密码卡最为接近的就是 vTPM（Virtual Trusted Platform Module）。TPM（Trusted Platform Module）可以为用户提供机器是否安全的证明，硬件里有存储密钥和加解密功能，可以将重要的数据进行密封加密，开机时自检完整性等等功能。要设计出密码卡的虚拟化[6]和动态迁移方案[7]，可以借鉴 TPM 设备的虚拟化和迁移的研究。

TPM 中有重要的三个密钥：EK（Endorsement Key），TPM 制作时就产生的密钥，标志 TPM 的身份信息，且用来生成 AIK（Attestation Identity Key）；AIK，用来做身份认证；SRK（Storage Root Key），用来保护要存储到 TPM 外部的密钥。三个密钥都是不可迁移的，然而为了保证虚拟机迁移后，功能不会缺失，这些 vTPM 的 vEK（Virtual Endorsement Key）、vAIK（Virtual Attestation Identity Key）、vSRK（Virtual Storage Root Key）都需要重新生成，极大的降低了虚拟机迁移的效率。文献[8]中为了能够尽量的减少迁移后的密钥再生成，提出新的 vTPM

的密钥层次，加入了两个不可迁移密钥 gSRK（Global Storage Root Key）和 SK，在保证 vTPM 和 TPM 的绑定关系的同时，还使得 vSRK 和 vAIK 可以迁移。 为了迁移的安全性，虚拟机不能迁移到恶意的主机上，恶意的虚拟机也不能迁移到目的主机上，因此迁移前源主机和目的主机需要相互认证，认证的同时交换会话密钥，vTPM 的迁移也要加密进行。

文献[9,10]中提出，在文献[1]中提到的密钥层次不符合 vTPM 密钥使用规范，物理硬件 TPM 应该为每个 vTPM 生成一个 EK，但是在文献[1]中不再使用 EK 来生成 vAIK，消除了 EK 的作用，vAIK 由 TPM 的 AIK 决定，但是 AIK 会被用来做远程证明，有被破解的风险，因此 vAIK 也就有了被伪造的风险。在文献[2]中提出了新的密钥层次结构，添加了一个全局的 gSRK 和一个全局的 gEK（Global Endorsement Key），分别将 TPM 的 SRK 和 EK 关联到 vTPM 的 vSRK 和 vEK。gSRK：一个不可迁移的非对称加密密钥，是 TPM 中 SRK 的直接后代，用于保护 vTPM 的 vSRK，因此，vSRK 不再是 SRK 的直接后代，可以迁移。gEK：用于关联和保护 vEK，使得 vEK 可迁移，文献中设计的迁移协议也有身份验证这一保证安全性的阶段。

文献[11]中提出了单密码卡设备虚拟化及迁移的方案，针对的是 Xen 这种 Hypervisor 模型，通过前后端的 I/O 设备虚拟化模型来模拟密码卡，文中还引入了一个 vedInstance 数据结构来支持密码卡的动态迁移，vedInstance 是虚拟化密码卡的实例，完成密码卡的功能，并存储着密码卡的状态。为了安全，vedInstance 中不存放密钥本身，而是存放密钥号，密钥本身需要主机通过密钥号向密钥服务器申请获得，因此，整个系统中又需要一个密钥管理服务器。此外，vedInstance 中还有与后端的共享内存 shareMemory，完成加解密任务的线程 edThread，处理迁移任务的线程 migThead 等等。文中还提出，密码卡的迁移要有四个阶段：保存加解密现场、导出加解密现场、导入加密现场、重启加解密，保存加密现场时密码卡就要停止加密了，虚拟机在开始迁移后不会立即停止运行，这时还可能在使用密码卡，所以密码卡现场的保存应该在虚拟机停机后再进行，在虚拟机重启前也要先加载密码卡的现场并重启密码卡，如此，可以尽量减少密码卡的停滞时间。但是在文中并未考虑多虚拟机对应多密码卡的情况进行方案设计。

文献[12,13]中主要提出了一种基于 BLP（Bell-La Padula）模型设计的一个新的

虚拟密码卡隐私保护模型 vED-PPM(Virtual Encryption Device-Privacy Preserving Model)，在这个模型中有：vED-Manager、vED-Instance、real-ED 三部分组成，vED-Manager 用于管理和创建 vED-Instance，引入 vED-Instance 主要是为了更好的管理密码卡，里面像文献[11]中的一样保存了密码卡的信息，方便迁移，real-ED 是硬件密码卡。在虚拟机中的应用安全级不同，导致数据的安全级也不同，但是这些数据都会通过统一的计算服务，当密码卡加解密不同安全级的数据时，虚拟机不能局限于特定的静态安全级别，数据的流动必须满足不上读不下写的规则，文中重点提出了一种虚拟安全级，使得主体可以拥有多个安全级的访问权限，不同于 BLP 的静态安全级，一个虚拟安全级实质上是一个或多个静态安全级的离散映射或集合，为密码卡安全级的动态调整提供了基础，密码卡空闲时安全等级较低，而在使用时安全等级很高，不能被随意的占用或剥夺，密码卡通过动态调整自己的安全等级来保护用户的数据。

文献[14]通过软件形式模拟 SR-IOV 设备[15]，SR-IOV 规范没有规定 PF 和 VF 通信方式，但是 PF 和 VF 之间需要为配置、管理信息和事件发送进行通信，所以文中给出了 PF 和 VF 的通信方式的软件形式的实现方案，利用了邮箱/门铃的机制实现的，这个机制与前后端的通信机制十分相似，邮箱即是共享缓冲区，门铃即是 PF 和 VF 相互通知的通道。对于 SR-IOV 设备，VF 的数量总是有限的，虚拟机的数量很有可能大于 VF 的数量，此时就不能为每一个虚拟机分配到一个 VF 来使用，所以文中还提出了一种 VF 的动态分配算法，按照一定的优先级判定策略为每一个虚拟机对 VF 的使用申请进行排队，采用可剥夺方式的分配方法为排在前面的申请分配 VF。

文献[16]中提到了一种密码卡虚拟化的方式：没有使用前后端的机制，客户机和宿主机利用 virtio 通信通道进行信息交互，进而使得客户机可以使用宿主机中的密码卡驱动。

文献[17,18]中针对虚拟机迁移中的预拷贝算法做了优化，提出了判断脏页是否是高脏页的算法，对高脏页进行延迟拷贝，直到它变成干净页或是普通脏页，减少内存拷贝时的数据量，提高了虚拟机的迁移速度。

1.3 主要内容

主要内容是设计基于 QEMU-KVM（Quick Emulator Keyboard Video Mouse）的多密码卡虚拟化系统，支持多虚拟机对多密码卡的调度，以及虚拟密码卡的迁移功能。首先学习使用 QEMU 启动和管理虚拟机，以及 QEMU-KVM 的前后端机制 virtio，分析了多虚拟机对应多密码卡系统的设计需求，由于没有硬件密码卡，所以只能先使用 Linux 下的字符设备驱动模块来模拟密码卡功能，但这并不影响系统的设计和实现，软件模拟的密码卡和硬件密码卡的区别仅仅在于性能和接口。然后设计出一个比较完整的系统并实现，接着使用一定的策略对虚拟化密码卡的性能进行优化。由于云计算中虚拟机的迁移功能非常重要，所以为了让虚拟化密码卡的方案能够应用到云计算中，对密码卡做了可迁移的功能支持。最终，对设计和实现的系统进行测试，并分析实验结果。

1.3.1 研究难点

（1）学习 QEMU-KVM 的 virtio 机制，通过 virtio 机制，以前后端的方式实现密码卡的虚拟化，保证功能的正确性。前后端方式实现的虚拟化密码卡与硬件辅助虚拟化方式实现的虚拟化密码卡的性能相比较差，所以为了让其性能更加接近原生系统中使用密码卡的性能，对前后端方式实现的密码卡进行性能优化。

（2）为了能让文中设计的密码卡虚拟化系统应用到云计算中，做多密码卡可迁移的功能，要考虑密码卡状态的保存和恢复问题，保证迁移过程数据不丢失，保证迁移过程中的加解密任务不失败。

（3）密码卡中的数据是必须要保密，不可泄露并不可被篡改，尤其是密钥。因此密码卡的状态信息在传送时要考虑其机密性，不能让这些秘密信息在不安全的通信信道上传送，并且在目的端也要验证其完整性，同时还要考虑重放攻击等威胁。

（4）由于多虚拟机对多密码卡的系统中，虚拟机和密码卡的映射关系是动态的，并且，虽然源主机和目的主机要求拥有同样的密码卡设备，但是在源主机中虚拟机对应的密码卡可能被其它主机占用，也很可能没有空闲密码卡可用，所以虚拟机迁移过后也需要重新申请密码卡使用权。

（5）找到一个合适的调度算法，为每个虚拟机的密码卡使用申请排队，并为优先级最高的申请分配密码卡使用权，保证系统的公平性和性能。

1.3.2 工作意义

主要是完成并优化了多密码卡的虚拟化功能，同时实现了较为安全的虚拟密码卡动态迁移功能。整体工作的主要意义有以下几点。

（1）使用前后端的 I/O 设备模拟方式实现了多虚拟机对多密码卡的虚拟化系统，提供了较为合理的调度算法来保证虚拟机对密码卡的使用，为 SR-IOV 设备的 VF 数量不够用时情况提供了合理的解决方案。

（2）在原本密码卡虚拟化的基础之上，优化了虚拟密码卡的性能，让其更加接近原生系统中密码卡的性能，尽量缩小与硬件辅助 I/O 设备虚拟化方式的差距。

（3）为了让多密码卡虚拟化系统能够应用在云计算技术中，为虚拟化密码卡提供了迁移功能，保证了迁移后不破坏加解密任务，同时在一定程度上保证了迁移过程的安全性。

1.4 文章结构

在概况整篇文章的核心内容的摘要之后，按照一定的逻辑顺序，主要分为六个章节，下面对这六个章节分别进行概况。

第一章是绪论部分，对 I/O 设备虚拟化领域的相关研究进行总结，分析现在在密码卡虚拟化方面的研究现状的不足，提出自己的主要研究内容、解决思路和技术方案，然后描述了工作的难点和主要贡献，最终描述整篇文章的大致结构。

第二章是相关技术和理论部分，这一部分将会较为详细的概况出研究时所需要的相关理论和技术，对这些理论技术进行分析，为后面的系统设计提供了理论基础。

第三章是基于 QEMU-KVM 的虚拟密码卡设计，在本章将详细的介绍整体的工作内容。首先分析虚拟密码卡的设计需求，然后提出虚拟密码卡的设计方案，设计出较为合理的调度算法，提出了密码卡虚拟化的性能优化策略，最终设计多密码卡安全动态迁移的方案。

第四章是基于 QEMU-KVM 的虚拟密码卡实现，在本章将对基于 QEMU-KVM 的 virtio 机制实现的多密码卡虚拟化的重点部分：调度算法、迁移功能、性能优化等进行较为详细的描述。

第五章是实验结果与分析，将对实现的虚拟密码卡进行功能和性能的测试，分析实验结果，判断方案的可行性和价值。

第六章总结和展望，将分析研究工作中的不足之处，提出未来需要为之努力的各个方向。

2 相关技术与理论

虚拟化技术自上世纪 60 年代发展以来，很多技术已经相当成熟，可以很好的投入实际的使用，CPU 和内存的硬件辅助虚拟化让两者几乎达到了与原生系统差不多的性能，然而 I/O 设备的虚拟化技术还有很多需要改善的地方，I/O 操作在整个系统运行期间所有操作中占绝大部分，所以 I/O 设备虚拟化的性能也将极大的影响了整个系统的虚拟化性能，并且由于 I/O 设备的多样性，虚拟 I/O 设备的动态迁移方案的设计也会根据不同设备而不同，较为复杂。研究的主要的工作就是密码卡这一 I/O 设备的虚拟化，在本章将介绍设计虚拟密码卡时所需要的相关技术。

2.1 虚拟化技术

2.1.1 系统虚拟化

系统虚拟化是指将一台物理计算机系统虚拟化为一台或者多台虚拟计算机系统[19]。每个虚拟计算机系统（简称为虚拟机）都拥有自己的虚拟硬件（如 CPU、内存和设备等），来提供一个独立的虚拟机执行环境。在计算机硬件发展迅速的今天，虚拟化技术有效的解决了硬件资源不能被充分利用的问题，极大的提高了云计算效率和可用性。

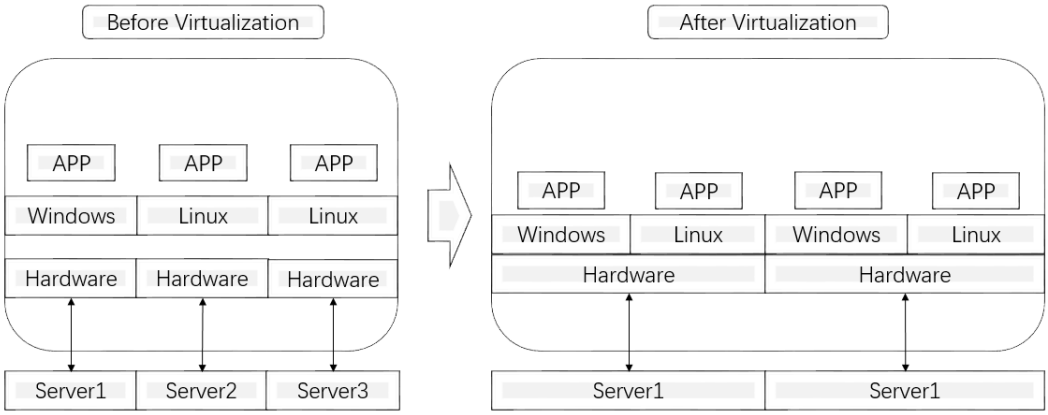

图 2-1 引入虚拟化层前后

图 2-1 所描述的是引入虚拟化层前后，两类计算模型的对比，左边的是传统计算模型，每个服务器上只能在同一时间运行一个系统，只能在这一个系统中运行这个系统所支持的应用程序，而右边的是加入虚拟化层之后的计算模型，在这种情况下，虚拟化层将物理资源虚拟化出多个拷贝，使得一个服务器中可以在同一时间运行多个操作系统，并且操作系统的类型也可以不同，这些操作系统之间可以互相联系，也可以互不干扰，达到了一机多用、提高物理资源利用率的目的。

虚拟机通常由虚拟机监视器 VMM(Virtual Machine Monitor)来管理，根据 VMM 所提供的虚拟平台类型可以将 VMM 分为两类：第一类 VMM 虚拟的是现实存在的平台，并且在客户机操作系统来看，虚拟的平台和现实的平台是一样的，客户机操作系统并不知道自己运行在一个虚拟平台上，这样的平台可以运行现有的操作系统，无需对操作系统进行任何修改，这类方式称为完全虚拟化。第二类 VMM 虚拟的平台是现实中不存在的，而是经过 VMM 重新定义的，此时必须对原有的操作系统进行修改才可以运行在这样的平台之上，客户机操作系统知道自己运行在虚拟平台上，这类方式称为类虚拟化。按照 VMM 的实现结构又可以将主流的虚拟化技术分为两类：Hypervisor 模型和宿主模型。Hypervisor 模型下 VMM 可以被看做是一个完备的操作系统，同时具有虚拟化功能，所有的物理资源都归 VMM 管理，同时 VMM 负责虚拟机的创建和管理，Xen 就是这种类型的 VMM。宿主模型下，在硬件之上运行的是常规的操作系统，硬件资源仍然由它来管理，VMM 是在这个系统之上一个应用程序，通过调用操作系统的服务来获取资源并为虚拟机分配，实现虚拟化功能，QEMU-KVM、VMware 和 VirtualBox 就属于这一类型。硬件资源的虚拟化主要是 CPU、内存和设备三部分，CPU 和内存的虚拟化经过多年的发展，通过 Intel VT-x 技术已经可以达到高性能的效果，并且规范和简化了 VMM 的设计。

2.1.2 I/O 设备虚拟化

系统在运行期间，I/O 交互操作占很大的比例，因此 I/O 设备模拟的性能也会对虚拟机的整体性能造成很大的影响。现如今主要的 I/O 设备模拟方式分为：设备接口完全模拟、前后端模拟、硬件辅助模拟，不同的 I/O 设备模拟方式有着不同的优势和劣势。

（1）设备接口完全模拟

如图 2-2 所示，在该模型下，客户机操作系统中运行的设备驱动可以是原本的未经修改的设备驱动程序，由 VMM 中的代码来模拟设备接口。当客户机发出 I/O 请求后，VMM 会通过 I/O 操作捕获代码让其陷入到 VMM，经过一定处理后放入到 I/O 共享页面，通知 I/O 设备模拟代码来获取 I/O 请求，此时模拟程序会根据情况调用真实的物理设备的驱动程序来完成 I/O 请求，或者完全通过代码逻辑来完成设备 I/O 请求，最终将处理结果放回 I/O 共享页，I/O 操作捕获代码会将其结果送回客户机。

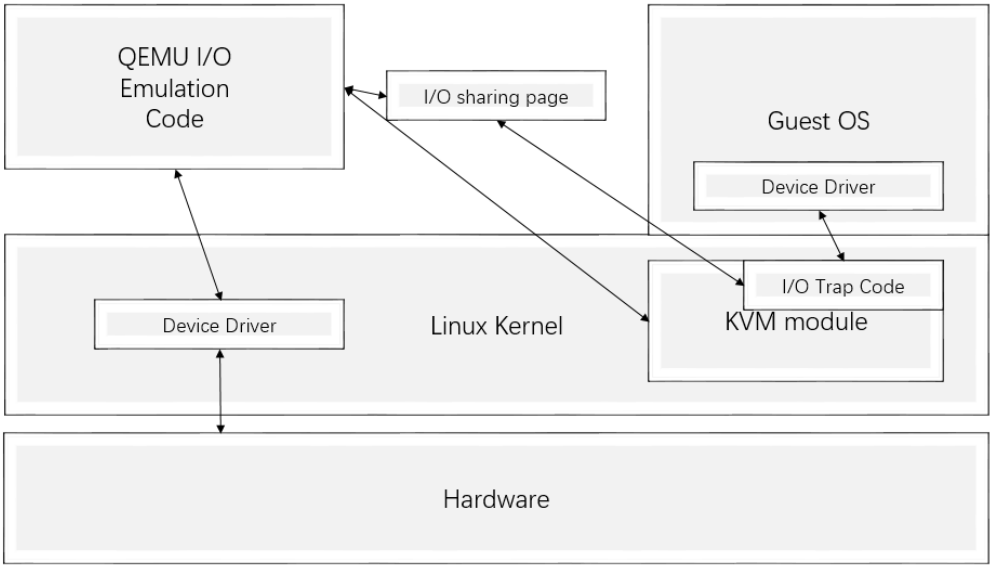

图 2-2 设备接口完全模拟

由于客户机操作系统处于的 CPU 安全等级不能进行 I/O 操作，所以在整个 I/O 操作过程中会存在很多的客户机和 VMM 之间的环境切换，CPU 安全等级也随之进行多次切换，另外还有大量的数据拷贝，这些导致了这类模拟方式的性能不高。但是此类方式很灵活，客户机可以用原本的驱动程序，宿主机上即便没有相应的物理设备，也可以通过 VMM 中的代码来模拟设备功能，从而完成设备模拟，不同的虚拟机也可以共享同一个 I/O 设备。

（2）前后端模拟

如图 2-3 所示，在该模型下，客户机中加载了一个定制的前端驱动程序，VMM 中加载着一个特定的后端驱动程序来调用真实物理设备驱动程序，两者通过共享内存来交互，完成客户机的 I/O 请求。当客户机发起 I/O 请求时，调用前

端驱动程序，将 I/O 请求发送到环形缓冲区，通过事件通道通知后端驱动，后端驱动接到通知后，从缓冲区中取出 I/O 请求，然后转发给真实的物理设备驱动来完成 I/O 操作，最后将返回的结果送到缓冲区，通知前端获取处理结果。

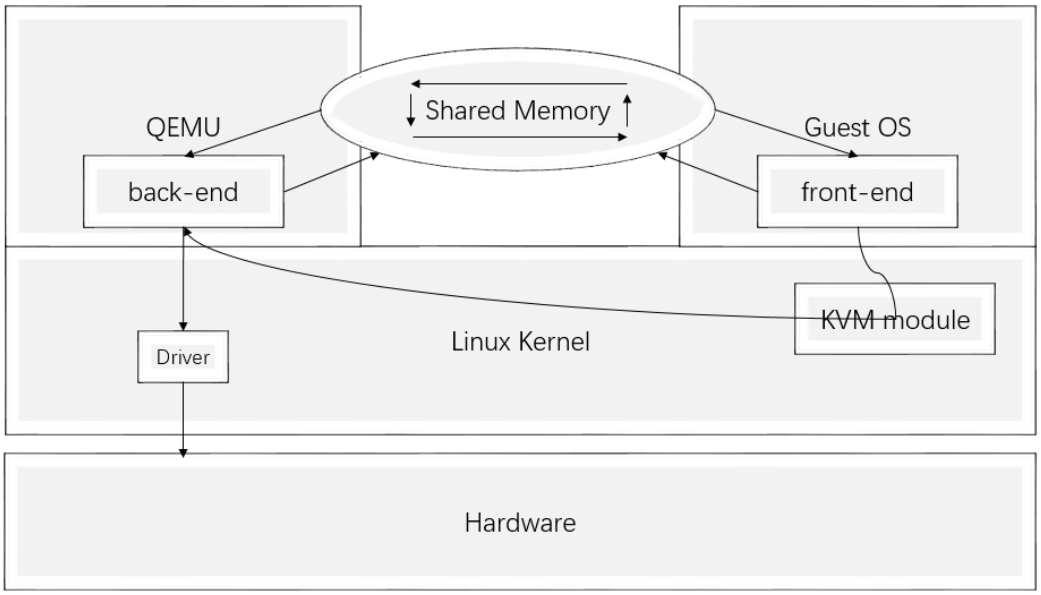

图 2-3 前后端模拟

与设备接口完全模拟的方式相比，I/O 设备请求的处理交由物理设备请求，后端只是起到一个转发请求的作用，并且客户机和 VMM 通过事件通道和共享缓冲区就完成了 I/O 请求和处理结果的数据传送，减少了 VMM 和客户机之间的环境切换次数，极大的提高了 I/O 设备模拟性能。但是，这种方式需要在客户机中加载一个特定的前端驱动程序，失去了一定的灵活性，并且 I/O 操作过程中仍有一些额外的数据拷贝。

（3）硬件辅助模拟

如图 2-4 所示，在该模型下，VMM 的设备管理器会为每一个虚拟机分配一个由它单独使用的物理设备，客户机内加载原本的驱动程序，可以直接使用物理设备，地址的问题由 DMA 重映射模块处理，在这种方式下，虚拟机可以像原生系统一样使用物理设备，性能几乎一模一样。

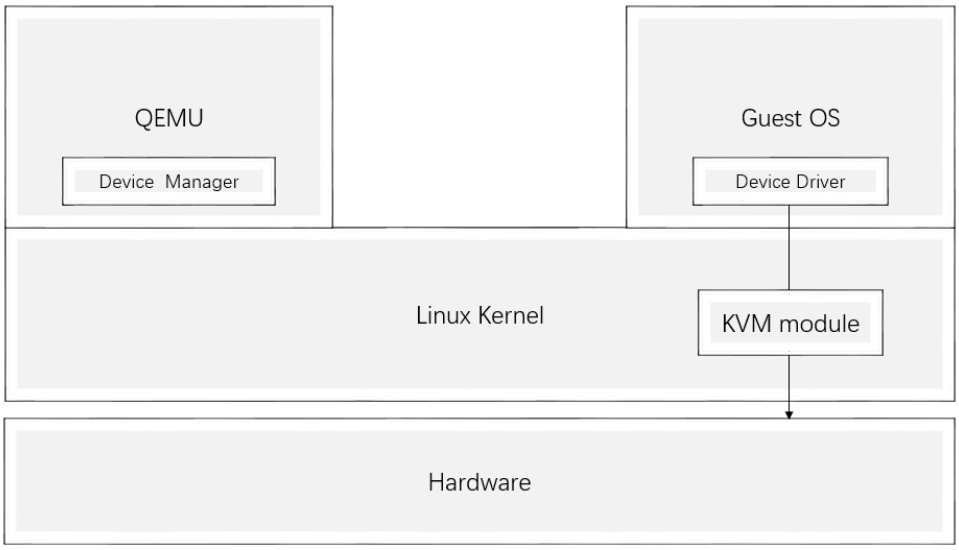

图 2-4 硬件辅助模拟

这类设备模拟需要硬件的支持，比如 Intel 的 VT-d 技术（支持 Passthrough I/O 技术和 DMA（Direct Memory Access）重映射技术）。虽然这类方式性能极高，但是牺牲了设备的共享性，一个设备一旦被分配给一个虚拟机就只能被这个虚拟机使用，如果虚拟机的数量超过了物理设备的数量，总会出现虚拟机没有设备可使用的情况。为此，出现了 SR-IOV 设备，在保证设备模拟性能的同时，在一定程度上解决了设备共享问题。SR-IOV 设备可以具有一个或多个物理功能 PF，PF 可以被看做是一个标准的传统 PCIe 设备，每个 PF 可以创建多个虚拟功能 VF，与 PF 相比是一种轻量级的 PCIe 设备。SR-IOV 也同样使用了 Passthrough I/O 技术，将每个 VF 分配给不同的虚拟机来使用，虚拟机可以将 VF 当做一个物理设备来使用，完成 I/O 请求处理的任务，如此，就实现了一个 SR-IOV 设备被多个虚拟机共享使用的目的，但是 VF 的数量终究是有限的，SR-IOV 设备也只是在一定程度上缓解了硬件辅助虚拟化中设备不够用的问题。

2.1.3 virtio 技术

为了达到物理设备能够共享的目的，使用了前后端模拟的方式来虚拟化密码卡，由于是基于QEMU-KVM[20]实现的密码卡虚拟化，所以就使用了QEMU-KVM中专门为前后端设备模拟提供的 virtio 机制，接下来对 virtio 机制进行详细的描述。

virtio 是一个半虚拟化框架，旨在为所有的虚拟化平台提供一个可扩展、高效便捷的 I/O 虚拟化实现框架，解决了每个虚拟化平台都要为自己设计一个特定的虚拟设备模型的问题。

图 2-5 virtio 框架

如图 2-5 所示，virtio 大致可以分为四层，最上面一层是运行在客户机操作系统中的前端驱动程序，因为设备类型很多，所以也衍生出了很多类型驱动程序模块，最后一层是运行在 QEMU 等虚拟化平台的后端驱动程序，中间的 virtio 层和 virtio-ring 层用于实现前端和后端的通信机制。在 virtio 层和 virtio-ring 层中有几个重要的组成部分：virtqueue、virtio-ring。

（1）virtqueue

virtqueue 实现了前端驱动与后端驱动的接口，主要有：virtqueue_add_buf、virtqueue_kick、virtqueue_get_buf 三个接口，其中 virtqueue_add_buf 主要提供向缓冲区中添加数据的功能，virtqueue_kick 主要是前后端在向缓冲区添加了数据之后提醒另一端的接口，virtqueue_get_buf 提供在接收到提醒后从缓冲区取出数据的功能。

（2）virtio-ring

virtio-ring 主要包括：Available Ring、Used Ring、Descriptor Table。Available Ring 结构会告诉前端还有多少可以向里面添加数据的缓冲区，并且供前端添加 I/O 请求，后端将从 Available Ring 中取得 I/O 请求，后端处理完 I/O 请求后，将处理结果放到 Used Ring 中，并通知客户机中的前端模块，前端将从 Used Ring 中获得处理结果。Descriptor Table 中每一个描述符都表示一个缓冲区 buffer，Available Ring 和 Used Ring 也是通过描述符来获得用来存取数据的缓冲区的地址和长度的。

2.2 虚拟机动态迁移技术

虚拟机迁移分为动态迁移和离线迁移，离线迁移是先关闭虚拟机，再完成存储数据的迁移，然后在目的主机开启虚拟机，这种迁移会导致虚拟机很长时间无法运行，在云计算服务中对于负载均衡和服务效果意义不大，而动态迁移指的是在虚拟机运行过程中将其迁移到其它主机上，期间虚拟机只会停机很少时间，用户几乎察觉不到虚拟机停机现象。

如图 2-6 所示，虚拟机的存储数据可以通过 NFS 等网络共享的方式由多个主机共享，免去硬盘存储数据的迁移，动态迁移的主要内容包括内存、CPU、I/O 设备等状态信息。其中，CPU 信息只要将虚拟机停机时的状态信息拷贝过去就可以，最复杂和主要的还是内存迁移和 I/O 设备的迁移。

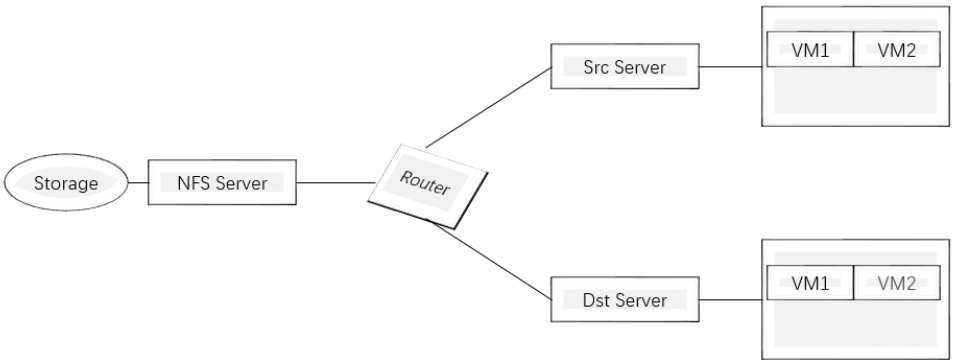

图 2-6 virtio 框架

（1）内存迁移

内存拷贝方式有很多，其中最常用的是预拷贝方法，这也是 QEMU 的默认动态迁移方式。接下来介绍预拷贝算法的过程。

1）源主机首次会将所有的内存页发送到目的主机，发送期间虚拟机仍在运行，内存中的页会更替，所以在这期间要记录脏页。

2）之后每次迭代都会将上次记录的脏页同步到目的主机，并记录本次发送过程中的脏页。

3）经过多轮迭代，脏页的数量将越来越少，最终等到脏页占的比例达到一定标准时，暂停虚拟机。

4）暂停虚拟机后就开启停机拷贝阶段，此阶段将虚拟机中剩下的所有脏页

全部拷贝到目的主机上，此时目的主机上就有了源主机上的运行环境和状态数据，虚拟机恢复在目的主机上恢复运行。

（2）I/O 设备迁移

因为I/O设备的多样性，I/O设备的动态迁移与内存动态迁移相比更加复杂，动态迁移时需要考虑迁移前后虚拟设备和真实物理设备的映射关系问题、状态信息一致性的问题和安全性问题。针对多密码卡设备，加解密状态信息的安全迁移、密钥的安全传递和迁移后虚拟密码卡与真实密码卡的映射关系等问题需要重点考虑。

2.3 进程调度算法

就像 CPU 数量有限，但是运行的进程很多这种情况一样，密码卡数量有限，但是虚拟机数量较多，此时就需要一个合适的调度算法保证每个虚拟机都能用到密码卡设备，并且算法要相对公平和实用，算法可以借鉴操作系统中常用的几种进程调度算法[21,22]。

（1）FCFS（First Come First Serve）算法

所谓 FCFS[23]就是先来先服务，每个进程根据进入内存的时间先后排队，每当 CPU 上的进程结束或阻塞，就将队首的进程调度到 CPU 上运行。这种模式下公平，但是当多个进程所需处理时间差较大时，短任务完成所需实际时间就会远远的超过理论所需。

（2）SPN（Short Job First）算法

所谓 SPN 即是短任务优先，每次选择新的进程到 CPU 上运行时都是从等待队列中选择所需处理时间最短的进程，在这种算法下，可以保证短任务能够快速的处理完成，但是如果短任务特别多，那么其它进程就会迟迟得不到处理，较为不公平。

（3）HRRN（Highest Response Ratio Next）[24,25]算法

这个算法是将前两种算法的整合，设每个申请等待的时间为t，完成任务所需时间为T，计算响应比$p = (t + T)/T$。高响应比任务优先处理，此时，当两个任务等待时间间隔不大时，小任务将优先被处理，大任务便会多等待一些时间，随着等待时间的增加，响应比随之增大，随后可能会排在小任务前面，优先处理。

针对特定的应用场景，可以对这个算法进行适当的修改，从而让算法能够更加适合系统。

调度算法中最终要的两个评估指标是周转时间和等待时间，一个作业从申请到完成的过程有三个重要时间：到达时间、完成时间、运行时间。周转时间=完成时间-到达时间；等待时间=周转时间-运行时间。为了评估周转时间对整个系统不同大小作业的影响，还提出了带权周转时间和平均周转时间：带权周转时间=周转时间/运行时间；平均周转时间=总周转时间/作业数目。

2.4 本章小结

本章主要介绍了虚拟化技术中的 I/O 设备虚拟化技术、QEMU-KVM 的 virtio 机制、动态迁移技术和一些进程调度算法，支持动态迁移的虚拟密码卡系统需要借鉴这些技术来完成。

3 虚拟密码卡设计

本章首先对应用场景进行描述，体现研究的意义和价值，然后针对应用场景进行需求分析，接着给出虚拟密码卡的设计方案，并对虚拟密码卡的性能进行优化，然后设计一个调度算法用来支持多虚拟机对应多密码卡的调度，最终设计一个支持虚拟密码卡动态迁移的方案。

3.1 应用场景

硬件密码卡可以快速的处理加解密任务，并且对密钥等秘密信息的保护力度更高，因此，如果虚拟机需要大量的处理大量加解密任务时，如果能够有虚拟密码卡设备进行支持，加解密任务完成的速度与软件方式实现的加解密要快的多，也安全的多。

一般云计算环境下，一个服务器上将运行多个虚拟机，这些虚拟机很可能同时使用虚拟密码卡，如果采用硬件辅助虚拟化的方式虚拟化密码卡，那么密码卡硬件个数必须大于等于虚拟机个数，而服务器的硬件资源可能不支持这么多的硬件密码卡，所以要对多个硬件密码卡通过前后端模拟方式进行虚拟化，以便这些虚拟机进行共享使用。云计算中有时会因为机器故障、负载均衡等原因将虚拟机进行迁移，此时为了不影响用户对虚拟机的使用，虚拟密码卡的动态迁移功能也将在这里起作用，总之，设计支持动态迁移的虚拟密码卡在云计算中会有很大的应用价值和意义。

3.2 需求分析

由于硬件密码卡资源无法获得，使用 Linux 系统下的字符设备驱动模块来设计一个硬件密码卡，并且其中有支持动态迁移的接口，与真实硬件密码卡的区别仅仅在于性能和接口的不同。接下来对设计支持动态迁移的多虚拟化密码卡进行需求分析。

（1）高效的密码卡虚拟化方案。使用虚拟密码卡来完成加解密任务主要还是为了速度的考虑，并且增强安全性，所以使用前后端这种较为快速的 I/O 设备

模拟方案，为了使其更接近硬件辅助模拟方式的虚拟密码卡的性能，要对虚拟密码卡的性能进行优化。

（2）合适的调度算法。使用前后端方式模拟密码卡一个很重要的目的就是密码卡硬件的共享，与硬件辅助虚拟化不同，虚拟机和密码卡之间的映射关系不固定，当虚拟机申请使用密码卡时会为它临时分配一个密码卡。虚拟机的个数很可能大于硬件密码卡的个数，所以当所有的密码卡都在使用中时，后续申请会进入等待状态，这里就需要一个适用的调度算法为这些等待的申请进行排队，让优先权更高的申请先被处理。

（3）虚拟密码卡动态迁移功能。为了让密码卡虚拟化方案适用于云计算技术中，作为 I/O 设备的虚拟密码卡也需要支持动态迁移功能，否则虚拟机迁移时会破坏正在进行的加解密任务。

（4）虚拟密码卡动态迁移时数据的完整性和保密性。密码卡中的状态信息有高隐秘性的要求，所以要考虑动态迁移时安全问题，保证这些状态信息在传送时不会被窃听，也不会被篡改。

3.3 虚拟密码卡的设计与性能优化

3.3.1 虚拟密码卡设计

对于密码卡的虚拟化，采用了前后端模拟的方式，如图 3-1 所示，客户机操作系统中运行着一个前端驱动程序，VMM 中运行着后端驱动程序，前端程序将为客户机中的应用提供密码卡接口，后端程序主要负责接受前端申请，使用真实的硬件密码卡处理申请，并将处理结果返回给前端。接下来描述一下虚拟密码卡的工作流程。

（1）客户机中的应用程序通过前端提供的接口处理加解密任务。当应用程序申请加解密时，前端会向共享缓冲区写入申请数据，并通过通知通道告诉后端程序。

（2）后端会从共享缓冲区中获取申请数据，然后调用硬件密码卡的接口处理申请，获得处理结果。

（3）后端将处理结果放回共享缓冲区，通过通知通道告诉前端，前端将会

从缓冲区中获得处理结果，完成一次加解密任务。

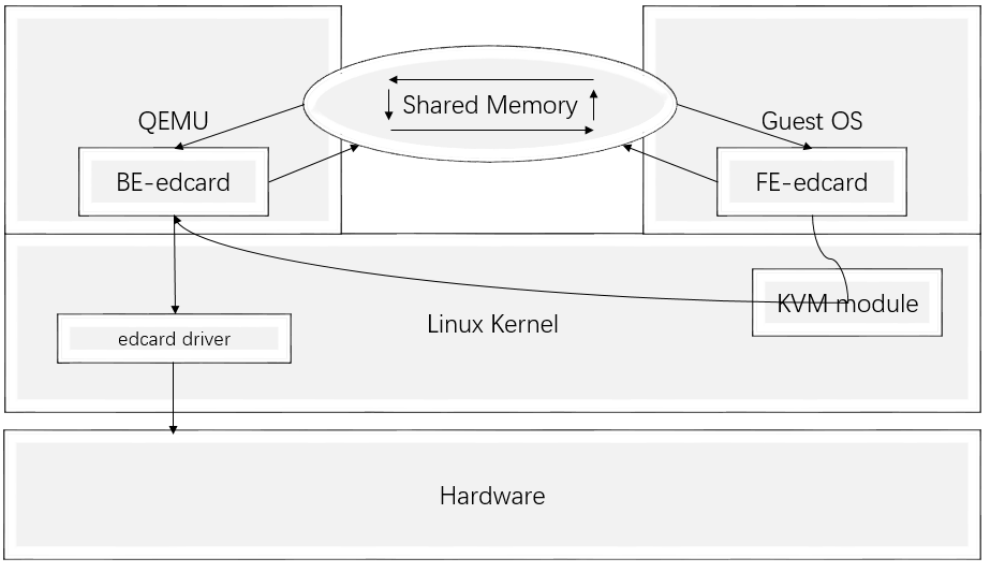

图 3-1 虚拟密码卡设计

3.3.2 虚拟密码卡性能优化方案

在这种方案下，客户机中使用密码卡与在宿主机中使用密码卡的区别在于：多了一些数据的拷贝和程序之间的通知消耗，每一次的加解密申请都将会多一次前后端的通信时间消耗。无论是硬件密码卡，还是使用 Linux 字符设备驱动模块模拟的密码卡，一次加解密所能处理的数据量都是有限的，所以针对大文件加解密时应用程序会将其分为多次加解密申请，然而，与原生系统中的密码卡使用相比，如果在客户机中加解密大文件，就会多出很多前后端通信的代价，会很大程度上影响用户的使用体验。

因此，要想减少虚拟密码卡的性能损耗，就是要尽量减少前后端通信次数。所以这里提出一个优化方案：在客户机中前端驱动所提供的一次最大处理数据量要比硬件密码卡的大，此时，客户机中应用程序一次申请的待处理数据，后端需要与密码卡驱动程序多次交互才能完成，对于同样的数据量，本来需要多次前后端通信的消耗才能完成加解密，现在只需要一次前后端通信就能完成。

虽然这样修改前端后，前端驱动所提供的接口将会与密码卡本身提供的接口有很大不同，但是在前后端 I/O 设备模拟方式下，这个问题并没有很大影响，因为前后端 I/O 设备模拟方式下，客户机本来就知道自己运行在虚拟环境下，并且

对于虚拟化设备只能加载定制的前端驱动，而不是原本的设备驱动，接口有所不同是必然的。

3.4 虚拟密码卡调度算法设计

为了实现多虚拟机对多密码卡的共享，设计出了如图 3-2 所示的系统架构。当多个虚拟机同时申请密码卡的使用权时，在后端程序中就需要一个合适的调度算法来为这些申请排序，并从队首开始分配密码卡，后端程序中也需要记录多硬件密码卡的使用状况，保证不同虚拟机不会同时获得同一个密码卡的使用权。

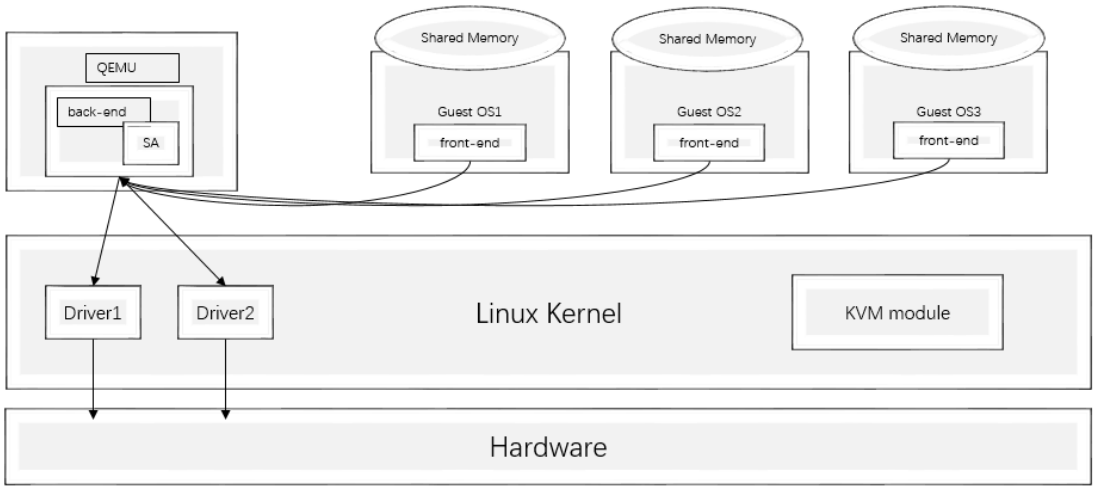

图 3-2 虚拟密码卡设计

当申请密码卡的虚拟机的个数少于密码卡个数时，不会有申请处于等待的情况，虚拟机可以立即获得一个密码卡的使用权，而当同时申请密码卡的虚拟机个数大于密码卡个数时，将会有申请处于等待状态，当之前的虚拟机释放了密码卡使用权后，后续申请才能得到处理，接下来介绍具体工作流程。

（1）同一时间多个虚拟机的前端驱动将应用程序的加解密申请放到共享缓冲区，由后端取出，如果有空闲的密码卡，后端会为先到的申请直接分配，而对于后面的申请要根据一定的优先级策略计算优先级，并根据优先级高低进行降序排序处理。

（2）由于密码卡一次性处理的数据量是有限的，一次的加解密任务可能涉及多次前后端通信，即多次 I/O 请求，但是大部分加解密的工作模式对当前数据块的处理需要前面数据块处理的结果作为输入，所以当仅仅是完成了一次 I/O 请

求，而没有完成一次加解密任务时，虚拟机是不能释放密码卡使用权的，直到剩下的 I/O 请求全部完成才可以，为了数据安全，释放密码卡使用权时要清除里面的数据痕迹。

（3）等到某个虚拟机的加解密任务完成后，它将释放密码卡的使用权，等待队列中的首个 I/O 请求所对应的虚拟机将获得这个密码卡的使用权，并开始处理加解密任务。

调度算法中最核心的就是优先级判定策略，首先，为了公平性，I/O 申请的时间顺序必须是优先级判定的一大重要因素，先到的申请应该先处理，然而考虑到加解密操作本身特别耗时的因素，如果只是按照申请的先后顺序来对申请进行排队，一旦较小的加解密任务比很大的加解密任务晚到了一点，就会导致本来只需要很短时间就能完成的加解密任务要等待很长时间才能完成，这对某些经常进行小数据加解密的用户非常不友好，如果让小的任务先被处理，即便是大任务多等待了一些时间，这些多出的时间消耗相比于自己任务本身完成所需时间几乎可以忽略，此时整个系统的用户的平均使用体验将会提高，因此，优先级大小的判定不仅要依据申请等待时间，还要考虑处理数据的多少。接下来介绍具体的优先级判定策略。

（1）客户机加解密任务的第一次 I/O 申请中加入这次任务的数据总量大小$size$和申请时间t_1。

（2）每当有密码卡的使用权被释放后，获得当前为申请队列中的每个申请计算等待总时间$t = t_2 - t_1$和数据总量$size$得出此申请的当前优先级，然后将所有的申请重新排队，取出优先级最高的申请，为其分配密码卡使用权。

（3）由于申请的等待时间t和数据总量$size$是两个不同的计量单位，以这两个不同的计量单位共同决定优先级，需要将他们关联起来，而它们之间的唯一关联就是密码卡的加解密速度，假设为s，那么任务处理时间$T = size/s$。所以，可以借鉴 HRRN 算法，使用公式$proprity = (t + T)/T$来得到优先级，公式中既考虑了等待时间，又考虑了数据量，等待时间越长优先级越高，数据量越小优先级越高。

（4）如果两个请求的优先级一样高，则按照具体的请求时间，将先到来的请求排在前面了，为它先分配密码卡。

通过上面所述的调度算法和优先级判定策略，可以让整个系统的用户拥有较好的使用体验。当然，优先级的判定策略可以根据特殊的需求做改变，通过实验数据得知$proprity = (t + T)/T$对T的变化较为敏感，比如：$proprity1 = (90 + 50)/50 = 2.80$， $proprity2 = (100 + 60)/60 = 2.66$，即虽然后者的申请先到，却因为处理的数据量比前者大了一点点，使得后者的优先级较低，这个决策或许对系统不是最好的，因此需要考虑降低所需服务时间T对优先级的影响力度。

这里对公式做了修改：$proprity = \frac{t}{T+n} + 1$，添加了$n$作为弱化因子，此时$T$对优先级的影响将会变小，$proprity1 = \frac{90}{50+50} + 1 = 1.900$， $proprity2 = \frac{100}{60+50} + 1 = 1.909$，这种情况下，后者将会先被处理，接下来进行理论分析：假设任务 1 和任务 2 的等待时间分别是t_1和t_2，服务所需时间分别是T_1和T_2，优先级分别为p_1和p_2，其中$t_1 > t_2$，$T_1 > T_2$，$t_1 * T_2 - t_2 * T_1 < 0$且$p_1 < p_2$，添加了弱化因子$n$，目的是让$y = \frac{t_1}{T_1+n} + 1 - (\frac{t_2}{T_2+n} + 1)$的符号更倾向于$z = t_1 - t_2$的符号，即这里更希望$(t_1 - t_2) * n + t_1 * T_2 - t_2 * T_1 > 0$，令$a = (t_1 - t_2) * n$，$b = t_1 * T_2 - t_2 * T_1$，$b$是添加弱化因子前的部分，a是添加弱化因子后的新增的部分，很明显当$n > 0$时，$a > 0$，且n越大，a越大，当 n 达到一定值时$p_1 > p_2$；当$t_1 < t_2$，更期望$y < 0$，此时$a < 0$，确实有利于$y < 0$。综上所述，添加了弱化因子后，任务所需时间T对优先级的影响相对变小了，n的值可以根据需要来设置，n越大，优先级与时间的相关性越大。

3.5 虚拟密码卡动态迁移设计

为了能够让所设计的虚拟密码卡能够运用到云计算技术当中，这里为虚拟密码卡添加动态迁移功能。首先便是要考虑其功能的完整性，虚拟机的动态迁移可能发生在任何时候，当发生动态迁移时，虚拟机没有在使用密码卡，也就没有虚拟密码卡的状态信息需要转移，这种情况非常简单，但是如果虚拟机正在使用密码卡设备，为了不中断加解密任务，需要将密码卡中完成加解密任务所需的所有状态信息全部转移到目的主机的密码卡之上，密码卡中的状态信息包括密钥、IV等等。如图 3-3 所示，是虚拟密码卡迁移的步骤，暂停并从密码卡中导出状态信息，以及加载状态信息到密码卡中并重启密码卡两个步骤需要硬件密码卡的功能

支持。

状态信息的迁移其实就是数据的发送，因为需要保证这些信息的机密性和完整性，并且还要防止重放攻击等恶意行为，信道必须是安全的。接下来介绍状态信息在安全信道转发的过程。

（1）不能将安全的虚拟机迁移到恶意的目的主机上，也不能将恶意的虚拟机迁移到完好的目的主机上，所以源主机和目的主机首先要相互认证，证书需要第三可信方颁发。

（2）认证的过程中要确认会话密钥，用于通信加解密，状态信息在加密后并附加对状态信息的签名信息，然后再加上一个时间戳来防止重放攻击，最后再将这些数据全部发送到目的主机。

（3）目的主机接收到数据解密，验证状态信息的完整性和时间戳，如果状态信息没有被篡改，并且信息并非是重放的，目的主机就成功获得了状态信息。

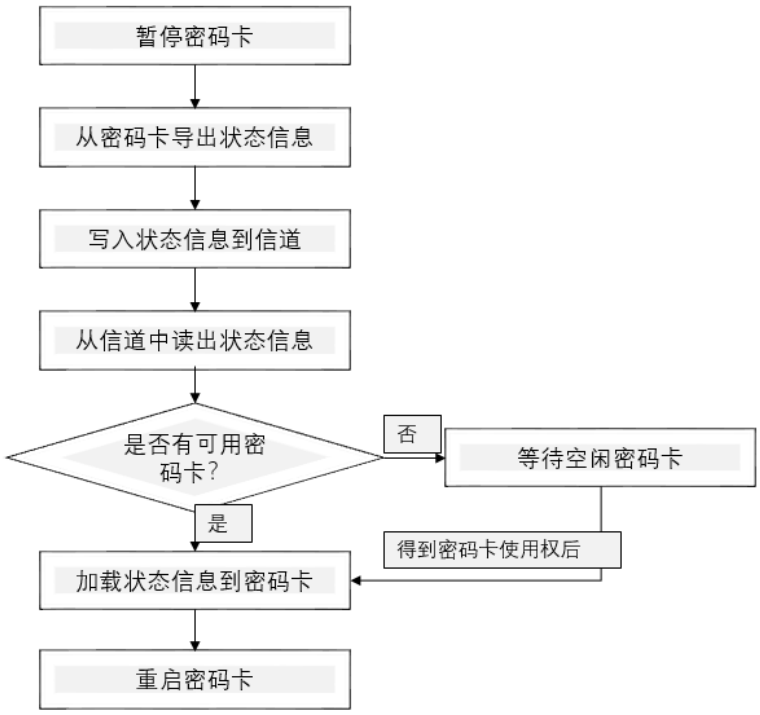

图 3-3 密码卡迁移步骤

由于系统中是多密码卡，那么密码卡迁移过程中会遇到这种情况：目的主机中可能已经有虚拟机正在运行，并且占用了密码卡设备，此时迁移过去的虚拟机将没有密码卡使用，所以这里也要有合适的调度算法来为迁移过来的虚拟机分配密码卡的使用权。这里不会再采用 3.4 章节描述的调度策略，由于虚拟机在源主

机中已经通过之前的调度算法获得了密码卡的使用权，如果此时再让虚拟机因为数据量的问题多等待一些时间会影响用户的使用体验，因此，这里的调度将仅仅考虑申请的先后顺序，并且这里的申请会优先与本地虚拟机的申请，以此达到尽快为迁移过来的虚拟机分配密码卡的使用权的目的。

3.6 本章小结

本章首先对虚拟密码卡的应用场景和设计需求分析做了描述，然后介绍了虚拟密码卡的设计方案，并对虚拟密码卡的性能进行了优化。调度算法借鉴了HRRN，同时提出了一种比 HRRN 更加灵活的一种调度算法，供使用者根据需要调整等待时间对优先级的影响力度。最后为了能够让虚拟密码卡方案在云计算技术中得到应用，提出了一个较为安全的虚拟密码卡动态迁移方案。

4 基于 QEMU-KVM 的虚拟密码卡实现

上一章设计的方案将会通过修改 QEMU-KVM 中的代码来实现，本章主要是对添加的代码中的重要模块和数据结构做些描述。

4.1 虚拟密码卡的实现与性能优化

QEMU-KVM 中提供了 virtio 机制来帮助开发者通过前后端方式添加虚拟设备，这里选择将密码卡作为块设备添加到虚拟机中。接下来对后端程序和前端程序分别做描述。

4.1.1 后端程序实现

表 4-1 VirtIOPCIEReq 数据结构

```
typedef struct VirtIOPCIEReq {
    VirtQueueElement elem;
    int64_t sector_num;
    VirtIOPCIE *dev;
    VirtQueue *vq;
    struct virtio_pcie_inhdr *in;
    struct virtio_pcie_outhdr out;
    QEMUIOVector qiov;
    size_t in_len;
    struct VirtIOPCIEReq *next;
    struct VirtIOPCIEReq *mr_next;
    char *buff;
} VirtIOPCIEReq;
```

首先是 VirtIOPCIEReq 数据结构，如表 4-1 所示，这个数据结构将会存放从前端获得的 I/O 请求，其中重要的成员有：dev，在后端程序中存放 virtio 设备信息的结构，对设备的读写操作都将以它为目标对象；vq，后端就是从 VirtQueue

结构中获得请求、具体数据等信息的；elem，后端从 VirtQueue 中获得的具体数据和相关信息会存放在这个成员中；buff，用于暂时存放后端与宿主机中的设备驱动交互时的数据。

如表 4-2 所示，对后端程序中重要的函数进行简单的介绍。后端通过这些函数实现与前端的通信，同时与宿主机中的设备进行交互，将前端发送来的请求交给设备，完成虚拟机想要虚拟设备完成的任务，然后将设备的处理结果再返回给前端。

表 4-2 后端程序中重要函数功能

virtio_pcie_device_realize(2)	对 virtio 和 VirtQueue 等进行初始化。
virtio_pcie_handle_vq(2)	等待前端通知，前端通知后，从 VirtQueue 中循环获得并处理申请，直到将所有目前所有的申请处理完，重新回到等待前端通知的状态。
virtio_pcie_get_request(2)	从 VirtQueue 中获得 I/O 申请。
virtio_pcie_handle_request(2)	处理一个 I/O 申请，此函数中将从 VirtQueue 中取出要处理的数据，完成与设备驱动程序的交互，再将处理结果放回到缓冲区，最后通知前端程序。
virtio_pcie_class_init(2)	对 VirtioDeviceClass 对象进行初始化操作，为其中的函数指针成员赋值。
virtio_register_types(void)	根据对设备的设定信息，注册块设备。
virtio_pcie_handle_write(2)	在函数 virtio_pcie_handle_request(2)中被调用，这里面是后端程序与硬件设备程序的具体交互代码。

4.1.2 前端程序实现

如表 4-3 所示，frontendcard 是一个字符设备的数据结构，它和其它 Linux 的自定义字符设备驱动程序一样，有 cdev 成员存放设备信息，有 io_buffer 成员存放读写数据。

表 4-3 frontendcard 数据结构

```
struct frontendcard
{
    struct cdev cdev;
    char * io_buffer;//缓存应用层传递下来的缓冲区
    int flag ;
    wait_queue_head_t front_wait_on_interrupt;
    void * priv;
};
```

如表 4-4 所示，virtio_blk 数据结构中的一些成员比较重要：frontendcard 成员是字符设备中的数据结构，即 blk 设备中存在一个字符设备；vdev 成员中存储了 blk 设备的基本设备信息；vq 成员是 blk 设备中的一个 VirtQueue，在实现的密码卡前后端驱动程序中只需要一个这样的队列。

表 4-4 virtio_blk 数据结构

```
struct virtio_blk
{
    struct frontendcard frontendcard;
    struct virtio_device *vdev;
    struct virtqueue *vq;
    wait_queue_head_t queue_wait;
    mempool_t *pool;
    struct work_struct config_work;
    struct mutex config_lock;
    bool config_enable;
    unsigned int sg_elems;
    int index;
    int flag;
    struct scatterlist sg[/*sg_elems*/];
};
```

前端驱动程序是一个 virtio_blk 设备驱动程序，而 virtio_blk 设备驱动程序中包含一个字符设备驱动程序，所以其对外提供的接口和其它 Linux 下字符设备驱动程序提供的 read、write 接口一样，接口的实现方法也很相似。virtio_blk 驱动程序与普通的 Linux 字符设备驱动程序相比，只是多了几个函数和处理步骤，如表 4-5 所示，是前端程序中重要的函数功能介绍。

表 4-5 前端程序重要功能函数

frontcard_open(2)	字符设备的 open 操作。
frontcard_close(2)	字符设备的 close 操作。
frontcard_read(4)	字符设备的 read 操作
frontcard_write(4)	字符设备的 write 操作，与一般的字符设备不同，这里面要额外的创建一个前端程序对后端程序的 I/O 申请。
setup_chardev(1)	字符设备驱动模块的注册代码部分。
destroy_chardev(1)	字符设备驱动模块的销毁代码部分。
virtblk_done(1)	用来处理后端返回结果的函数，从缓冲区中用 virtqueue_get_buf 函数获取数据
process_req(2)	构建一个 I/O 申请，并且将它添加到 VirtQueue 中，然后使用 kick 函数通知后端程序。
virtblk_probe(2)	此函数进行初始化 virtio_blk 结构、调用 setup_chardev 和 init_vq 等初始化函数。
virtblk_remove(1)	对前端程序进行销毁处理。
init_vq(1)	初始化 VirtQueue，并指定前端用来处理后端返回结果的函数。

4.1.3 性能优化

性能优化的具体实现，在前后端通信中的数据中多附加一些信息，并且每次传送的数据足够设备驱动程序处理 100 次，这样就可以减少前后端的通信次数，达到高性能的要求。密码卡使用的是 Linux 的字符设备驱动程序，对缓冲区大小

设置有要求，这里设置的 7168 个字节，即每次与密码卡设备交互能加解密 7168 个字节，那么后端程序中每次会从前端获取 716800 个字节。

如图 4-1 所示，应用程序要按照一定的数据结构往字符设备中写数据，只有这样才能正确的完成加解密任务。加密模式，比如：ECB，CBC 等，由于设计的模拟密码卡中只有两种，所以这里只需要 1bit 的内存空间；加密/解密，0 表示加密，1 表示解密；需要与密码卡交互的次数，这个是待处理数据字节数/7168 得到的；后端程序一般与密码卡驱动有三种交互方式：获得密码卡使用权并导入密钥、加解密、释放密码卡使用权；任务数据总量，用来确定优先级的因素之一。

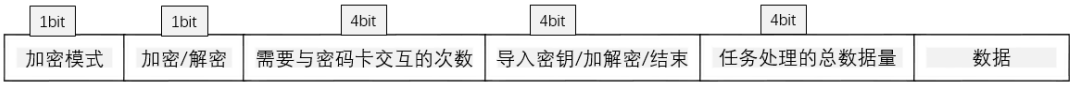

图 4-1 io-buffer 内部结构

当使用硬件密码卡时，可能密码卡一次性交互能够处理的数据很大，超过后端能够使用的缓冲区，此时采用上述实现方法不太可行。此时可以根据同样的优化思想，采用另一种实现方案：多次 write 并往 VirtQueue 中提交多次 I/O 申请后再调用 kick 函数通知后端程序，而不是前端每次使用 virtqueue_add_buf 后都调用 kick 函数。让后端接收到一次通知后处理多个 I/O 申请，这样同样可以减少很多通知过程的时间消耗，也可以达到性能优化的目的，但同时也需要应用程序和前端驱动的在功能和异常处理方面做一些配合。

4.2 虚拟密码卡调度算法实现

如表 4-6 所示，是一个存放多个密码卡设备的信息的数据结构，为了让多个虚拟机的申请能够合理的被调度，这里使用了信号量集的方式来同步 QEMU-KVM 的进程，首先设置一个密码卡的信号量集 card_val，初始值是空闲密码卡的个数，通过信号量集的 P 操作，当还有空闲密码卡时，进程会被放行，找到空闲密码卡并使用，当没有空闲密码卡时，进程会被阻塞，直到有其它进程进行了 V 操作来释放密码卡资源。每个密码卡也都有自己独立的缓冲区，所以对密码卡的使用进行同步时，也需要设置一个缓冲区的信号量 buf_val 来对相应的缓冲区的读写进行同步。从而保证不同的虚拟机同一时间不会操作同一个密码卡，也不会在有空闲密码卡时让虚拟机等待。

表 4-6 多密码卡数据结构

```
typedef struct _multi_card
{
    int cardlock_id;
    int buflock_id;
    int buf_id;
    int hDevice;//密码卡的句柄
    int flag_card;//判断密码卡资源有没有进行资源回收
    int flag_buf;//判断共享缓冲区的锁有没有释放
    union semun card_val;//记录密码卡个数
    union semun buf_val;//记录缓冲区个数
}multi_card;
```

因为仅使用信号量集的操作只能让虚拟机的申请按照时间顺序获得密码卡的使用权，为了实现对数据量的考虑，这里申请了一个共享缓冲区，每当进程被阻塞时将会往缓冲区中填入如图 4-7 所示的数据结构，包括申请时间和数据量等信息。每当有密码卡资源被释放时，将会先从共享缓冲区中取出这些数据，将所有的申请的优先级重新计算，并得到最高优先级的申请，唤醒与其相关的进程。为了防止共享缓冲区中的数据因多个进程同时读写而产生错误，再设置一个信号量集让不同进程对共享缓冲区的操作达到互斥的目的。

表 4-7 申请信息数据结构

```
typedef struct _req_blk
{
    int pid;
    unsigned int start;//申请时间
    unsigned int size;//数据量
    double proprity;//优先级
}req_blk;
```

表 4-8、表 4-9、以及表 4-10 是 virtio_pcie_handle_write 函数中的调度算法实现的几个重要部分，里面省略了变量声明、初始化、异常处理等代码，只显示了

部分重要代码，并且做了较为详细的注释。

表 4-8 初始化部分

```
static void virtio_pcie_handle_write(VirtIOPCIEReq *req, MultiReqBuff *mrb){
    //只显示关键代码行
    ......
    per_blk[0]=(req->buff)[0];
    per_blk[1]=(req->buff)[1];
    int num=(int)(req->buff)[2];          //需要对密码卡请求的次数
    int eof_flag=(int)(req->buff)[3]; //0:首次请求，1:中间请求，2:最后请求
    unsigned int total_size;
    memcpy(&total_size,req->buff+4,4);//请求加解密的数据总量
    pv_id=semget(pv_key,1,(IPC_CREAT|IPC_EXCL)|0666);
    semctl(pv_id,0,SETVAL,resources_edcard);//初始为空闲密码卡个数
    pv_shared_id=semget(pv_shared_key,1,(IPC_CREAT|IPC_EXCL)|0666);
    semctl(pv_shared_id,0,SETVAL,shared_buf);//共享缓冲区能同时被一个进程访问
    ......
}
```

在表 4-8 的初始化部分，会初始化声明并初始化一些重要的变量，比如：前端发送过来的请求本身的字符串，以及对这个字符串解析后的信息，还有进程同步需要的 PV 操作的初始化，申请一些共享缓冲区用于进程之间的通信等等。其中 req->buff 最重要，它是前端传来的申请，可以从中获取一些关键信息，比如：加密/解密、加密模式、本次请求的操作类型、任务总数据量等等，req->buff 中的数据结构与图 4-1 所描述的一样。使用 semget 申请两个 PV 操作需要的 id，然后使用 semctl 函数用这两个 id 申请两个信号量集，分别用于管理多进程对多密码卡的访问和申请等待队列的共享缓冲区的访问。

表 4-9 所示的代码是紧接于表 4-8 中的代码的。由于虚拟机需要加密的数据量很大，远远超过前后端之间的缓冲区上限，为了完成一次加解密，需要前后端进行多次交互，而在第一次交互时，虚拟机还没有获取某个密码卡的使用权限，所以需要寻找空闲密码卡，并申请使用权限。

表 4-9 寻找空闲密码卡部分

```
static void virtio_pcie_handle_write(VirtIOPCIEReq *req, MultiReqBuff *mrb){
    ......//进行一些初始化操作
    if(是否寻找空闲密码卡){
        //申请一个共享缓冲区，存放密码卡使用权的申请队列
        buf_id=shmget(shared_key,1604,(IPC_CREAT|IPC_EXCL)|0666);
        req_init(req_queue[wait_num]);//构造一个申请，并添加到队列中
        while(没有找到空闲密码卡){
            P(pv_id,0);//如果密码卡不够用则等待
            for(i=0;i<wait_num;i++){//有密码卡空闲，计算各个申请的优先级。
                req_queue[i].proprity=(now_time-req_queue[i].start)/(nd_time+50.0)+1;
            }
            ......//找到最高优先级的申请
            if(自身优先级最高){
                ......//接管空闲密码卡使用权，删除自身代表的申请
            }
            else{
                V(pv_id,0);//最高优先级不是自己，释放占有的资源
            }
        }
        ioctl(hDevice,KEY_IMPORT,req->buff+offset_data);//导入密钥
    }
    ......
}
```

在表 4-9 中，req->buff 中将会有标志来表示这次的数据是不是第一批待加解密数据，如果是，则伴随有密钥。代码首先会通过 P 操作将申请加入到等待队列。等到有空闲密码卡时程序继续运行，计算各个申请的优先级（其中 50.0 是弱化因子，可以根据自己的硬件环境和需求来做相应的调整）。如果自身优先级最高，则获得密码卡使用权，通过 ioctl 与密码卡设备交互并导入密钥，否则需要通过

V 操作来释放自己占用的资源，重新排队等待密码卡。

当 req->buff 中的数据不是第一批加解密数据时，虚拟机应该已经占有了某个密码卡的使用权，所以无需再向宿主机申请密码卡的使用权，可以直接进行加解密操作。

表 4-10 与真实密码卡交互部分

```
static void virtio_pcie_handle_write(VirtIOPCIEReq *req, MultiReqBuff *mrb){
    ......//一系列的初始化操作
    ..... //申请密码卡的使用权限
    if(是加解密申请){//与密码卡驱动程序进行交互，将待处理数据全部处理
        int offset=8,j;
        for(j=0;j<num;j++){
            if(j==num-1){
                per_len=(length-offset_data)%per_len==0?7168:(length-
offset_data)%per_len;
            }
            memcpy(per_blk+2,req->buff+offset,per_len);
            int writeLen = write(hDevice, per_blk, per_len+2);
            int readLen = read(hDevice, buffer+offset-offset_data, 7168);
            offset+=per_len;
        }
        memcpy(req->buff, buffer, length-offset_data);
    }
    ......
}
```

如表 4-10 所示，在已经有了密码卡的使用权后，系统将会要求后端与真实密码卡驱动程序交互，完成数据的加解密操作（write 是设备的写接口，设备会对写入的数据进行加解密，read 接口将会读取设备加解密后的结果）。因为一次前后端交互中所需要处理的数据超过了密码卡驱动程序一次所能处理数据量，所以这里通过一个循环对所有数据进行处理。

表 4-11 释放密码卡使用权部分

```
static void virtio_pcie_handle_write(VirtIOPCIEReq *req, MultiReqBuff *mrb){
    ......//申请密码卡权限并加解密
    if(是否释放密码卡资源){
        close(hDevice);
        hDevice=-1;
        V(pv_id,0);
        flag_card=1;
    }
    virtio_pcie_rw_complete(req,0);
    return;
}
```

如表 4-11 所示，在完成加解密任务后，需要释放密码卡使用权，并且通知其它等待的进程去获取密码卡使用权。使用 QEMU-KVM 管理虚拟机时偶尔也会出现异常情况，这可能导致虚拟机进程崩溃，此时若不释放密码卡资源，可能导致密码卡资源缺失，因此添加了一些标志信息（如 flag_card）用来帮助异常处理，在 QEMU 的异常处理函数 qemu_kill_report 中添加针对信号量集和共享缓冲区的清理操作，提高系统的异常处理能力。如果在释放密码卡权限时崩溃了，flag_card 将等于 0，在异常处理函数中将会对密码卡进行权限释放操作。在释放了密码卡资源后，后端将会通过 virtio_pcie_rw_complete 通知前端自己已经完成了它的申请。

4.3 虚拟密码卡动态迁移实现

VirtioDeviceClass 为开发者提供了添加 blk 设备的接口，里面有众多函数指针，如表 4-12 所示，realize 和 unrealize 两个函数指针将分别指向设备初始化函数和设备资源释放函数；save 和 load 函数指针将指向保存和加载设备信息的函数，save 函数应该在动态迁移前保存一些必要的状态信息，load 函数需要在完成迁移后恢复之前保存的信息，在迁移过程中，状态信息存放在 QEMUFile*指向的文件中。

表 4-12 virtio 设备类

```
typedef struct VirtioDeviceClass {
    DeviceClass parent;
    DeviceRealize realize;
    DeviceUnrealize unrealize;
    void (*reset)(VirtIODevice *vdev);
    ......
    void (*save)(VirtIODevice *vdev, QEMUFile *f);
    int (*load)(VirtIODevice *vdev, QEMUFile *f, int version_id);
    const VMStateDescription *vmsd;
} VirtioDeviceClass;
```

如表 4-13 所示，在此函数中会进行一些设备初始化的操作，并且将 save 函数指针指向密码卡状态信息保存函数 virtio_pcie_save_device，load 函数指针指向密码卡状态信息加载函数。QEMUFile*指针指向的文件会随着 QEMU 的虚拟机迁移迁移到目的主机，然后目的主机就可以将密码卡状态信息加载到密码卡中继续加解密任务。

表 4-13 密码卡设备类初始化函数

```
static void virtio_pcie_class_init(ObjectClass *klass, void *data){
    DeviceClass *dc = DEVICE_CLASS(klass);
    VirtioDeviceClass *vdc = VIRTIO_DEVICE_CLASS(klass);
    ......
    vdc->realize = virtio_pcie_device_realize;
    vdc->unrealize = virtio_pcie_device_unrealize;
    ......
    vdc->reset = virtio_pcie_reset;
    vdc->save = virtio_pcie_save_device;
    vdc->load = virtio_pcie_load_device;
    return ;
}
```

如表 4-14 所示，是动态迁移时在源主机上运行的函数 virtio_pcie_save_device，首先检查虚拟机有没有使用密码卡，如果没有则不需要迁移任何信息，如果有则将信息导出到 QEMUFile（状态信息主要是密钥已经当前设备中已经处理完成的数据的结果），并重置密码卡中的信息，最后还要根据之前提到的标志变量决定是否对信号量集和共享内存做一些释放处理，以便其它未迁移的虚拟机能够正常的使用空闲下来的密码卡。

表 4-14 virtio_pcie_save_device 函数

```
static void virtio_pcie_save_device(VirtIODevice *vdev, QEMUFile *f){
    unsigned char buff[33];
    if(hDevice>0)
    {
        buff[0]=1;
        ioctl(hDevice, MIGRATE_STATE_SAVE, buff+1);//获取当前加密卡状态
        ioctl(hDevice, RESET);//将加密卡 reset
        close(hDevice);
    }
    else{
        buff[0]=0;
    }
    //释放同步时使用的锁或共享内存
    ......
    qemu_put_buffer(f, buff, 33);//保存加密卡状态
}
```

如表 4-15 所示，是动态迁移时在目的主机上运行的函数，如果虚拟机本来正在使用密码卡，则需要为它申请密码卡使用权。因为虚拟机是迁移过来的，为了不对用户的使用体验造成不良影响和公平性，这里的密码卡使用权的申请的优先级将高于正常情况下的申请（只考虑申请时间的因素，不考虑处理数据量的大小）。得到密码卡使用权后将 QEMUFile 中的信息加载到密码卡中，然后继续加解密任务。

表 4-15 virtio_pcie_load_device 函数

```
static int virtio_pcie_load_device(VirtIODevice *vdev, QEMUFile *f, int version_id){
    unsigned char buff[32];
    qemu_get_buffer(f, buff, 33);//获取状态
    if(需要获取密码卡使用权){
        //通过信号量集与其它虚拟机竞争密码卡资源，并且只考虑申请时间因素
        key_t pv_key=ftok("/tmp/",21);
        pv_id=semget(pv_key,1,(IPC_CREAT|IPC_EXCL)|0666);
        while(还没有找到空闲密码卡)){
            P(pv_id,0);//等待有空闲的密码卡
            for(i=0;i<EDCARD_NUM;i++){
                hDevice = open(edcard_name[i], O_RDWR);
                int err=flock(hDevice,LOCK_EX|LOCK_NB);
                if(err<0){
                    close(hDevice);
                    hDevice=-1;
                }else{//找到了空闲密码卡
                    break;
                }
            }
        }
        ioctl(hDevice, MIGRATE_STATE_LOAD, buff+1);//将状态加载到新的加密卡
    }
}
```

4.4 本章小结

本章主要对虚拟密码卡如何借助 virtio 机制实现做了详细描述，并对优化部分、调度算法、迁移功能的实现中的重要数据结构和关键函数功能做了较为详细的描述和解释。

5 虚拟密码卡系统的测试与分析

本章主要分为：密码卡功能测试、性能优化效果测试、调度算法测试、迁移功能测试。通过这些测试结果对系统的应用价值提供实验数据的支撑。

5.1 环境配置信息

VMware 运行在 12 核的 Intel i7-10750 的 CPU 上，内存是 16G，QEMU-KVM 运行在 VMware 启动的 Ubuntu16.04-64 位系统下，为每一个 QEMU-KVM 虚拟机分配 2G 内存和 2 个处理核心，启动 kvm 加速模块，客户机系统是 OpenSUSE15.3。

5.2 虚拟密码卡加解密功能测试

如图 5-1 所示，在客户机中运行加解密测试程序，对原文件 plain 进行加密，然后再解密，最后使用 diff 命令查询两个文件的区别，结果两个文件完全一致，说明加解密功能上没有错误。

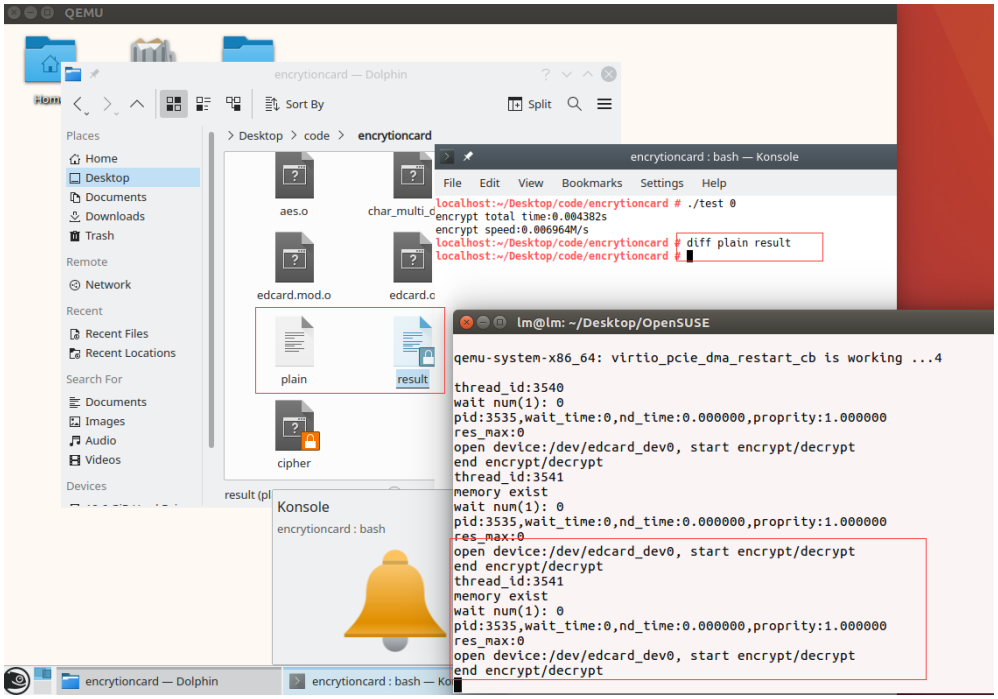

图 5-1 密码卡加解密功能测试

5.3 性能优化测试

分别在宿主机中和客户机中运行密码卡测试程序，对比加解密速度，得到性能损耗的比例，并分别在优化策略实施前后进行相同测试，对比测试结果，分析优化后的效果。

如图 5-2 所示，是在优化后对不同数据量进行加解密的性能测试，在这里，单次前后端通信的数据量需要后端与密码卡设备驱动进行 100 次交互完成加解密，对比发现，虚拟密码卡的加解密速度稳定在实际密码卡的 95%以上，图中折线有浮动是因为最后一次可能只剩下少量数据没有加密，但还是需要一次前后端的通信才能完成，与每次前后端通信都完成足够的数据加解密相比，性能比肯定会稍低一些，但浮动并不大，性能较为稳定。

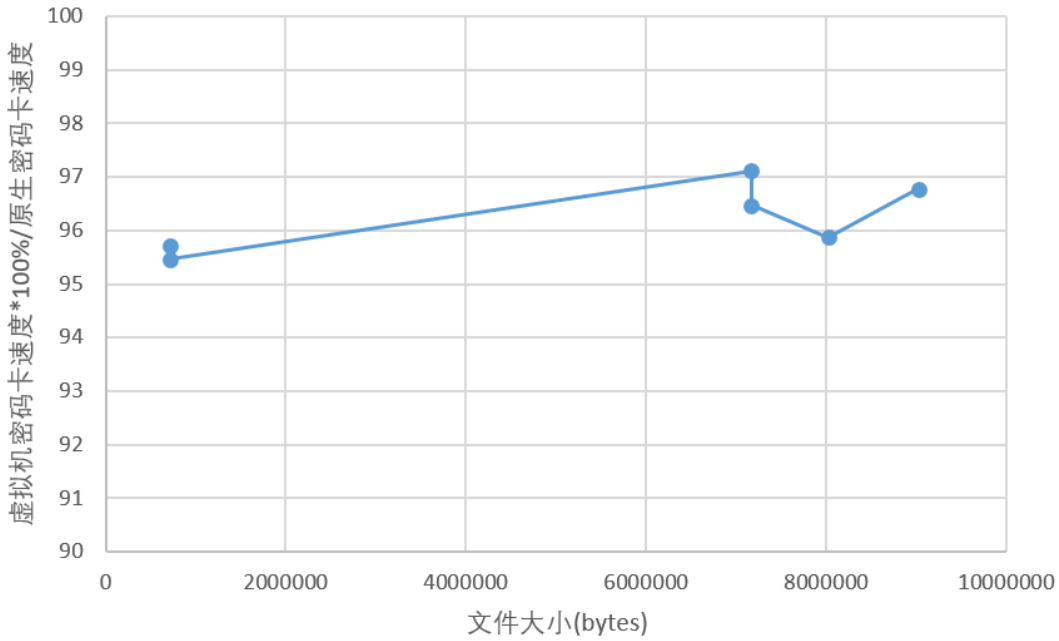

图 5-2 优化后性能测试

如图 5-3 所示，在优化前，前端为应用程序提供的接口和硬件密码卡一样，每次前后端通信的数据量只需要后端与密码卡驱动交互一次就能完成，虚拟密码卡的性能只有实际密码卡的 65%，随着每次前后端通信数据量的增多，虚拟密码卡的性能逐渐提高，最后趋近于 96%左右。由以上的测试结果来看，优化策略的效果较为明显，有很大的实用价值。

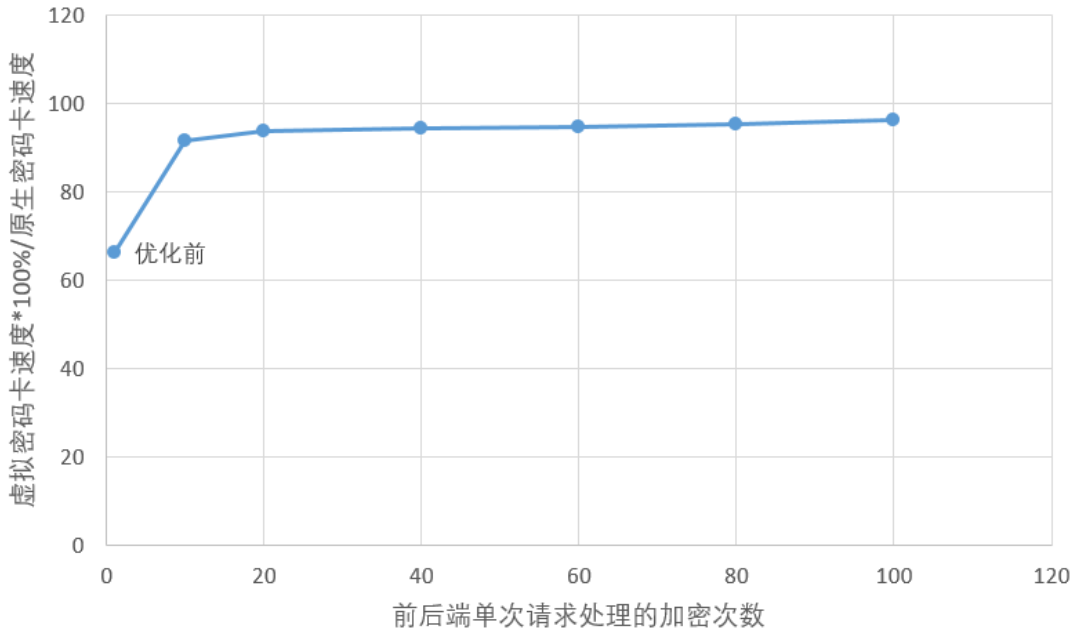

图 5-3 优化前后对比

5.4 调度算法测试

调度算法中的两个重要评估指标带权周转时间和平均周转时间分别体现不同作业的体验和整个系统的性能，本节主要通过这两个指标来评估调度算法对不同作业的影响和整个系统的影响。

在测试系统中，客户机中的虚拟密码卡加解密速度是 40Mb/s，如表 5-2 所示，在只考虑申请时间先后顺序时，虽然公平，但是小数据加解密的申请到达时间较晚，就会排在大数据量加解密任务的后面，从而导致小任务的周转时间较大，这种方式对小数据量的加解密很不友好，反观表 5-1 中的数据，借鉴了 HRRN 算法中的响应比计算公式计算优先级后，虽然 VM2 的加解密任务会排在 VM3 后处理，但是由于本身加解密就需要不少时间，多等待一小段时间对带权周转时间影响不大，与此同时极大的降低了 VM3 的带权周转时间。表格 5-1 中的平均周转时间：$Average\ Turnaround\ time = \frac{25+49+10}{3} s = 28s$，表格 5-2 中的平均周转时间：$Average\ Turnaround\ time = \frac{25+46+40}{3} s = 37s$，由此可见，借鉴 HRRN 算法后，系统的整体性能也有提升。

数据加解密是个很耗时的任务，在实际应用中会出现各种数据量大小的加解密需求，很容易出现表中所述的情况，所以为了提高系统整体性能和小任务的体

验，将等待时间和数据量大小都考虑到较为合适。

表 5-1 借鉴 HRRN 算法的测试结果

虚拟机	数据	到达时间	竞争时优先级	运行时间	周转时间	等待时间	带权周转时间
VM1	1000M	0s		25s	25s	0s	1
VM2	1200M	9s	1.5	30s	49s	19s	1.63
VM3	120M	18s	3.3	3s	10s	7s	3.33

表 5-2 先来先服务策略的测试结果

虚拟机	数据	到达时间	运行时间	周转时间	等待时间	带权周转时间
VM1	1000M	0s	25s	25s	0s	1
VM2	1200M	9s	30s	46s	16s	1.53
VM3	120M	18s	3s	40s	37s	13.3

有时系统为了更加公平，更加重视申请到达的时间顺序，即便可能在某些情况下降低了系统的整体性能，也会希望先处理等待更长时间的任务，但同时又不想摒弃对数据量大小的考虑，此时可以采用第三章所描述的添加了弱化因子的公式，对表 5-3 和表 5-4 中所述场景进行测试，VM2 和 VM3 的数据量相差 400M，服务所需时间相差 10s，VM2 比 VM3 的申请早到 9s，系统为了公平性，更希望 VM2 的申请先得到处理，然而由于原本的 HRRN 算法的优先级计算公式中任务处理所需时间对优先级影响较大，导致了 VM3 的申请先得到处理，在添加了弱化因子后，在同样的情形下，通过对n的设置，可以调整等待时间对优先级的影响力度，如表 5-4 所示，当n设置为 50 时，VM2 的申请先得到了处理。n的设置根据需求可以改变。根据表中显示，调整之后减少了 VM2 的带权周转时间，理所当然的，VM3 的带权周转时间会增加。表格 5-3 中的平均周转时间：$Average\ \ Turnaround\ \ time = \frac{25+64+30}{3} s = 39.67s$，表格 5-4 中的平均周转时间：$Average\ \ Turnaround\ \ time = \frac{25+44+60}{3} s = 43s$。由此得出结论，添加了弱化因子后，可以通过调整弱化因子来调整等待时间对优先级的影响度。弱化因子的选择要根据密码卡的处理速度，以及用户的使用需求来共同决定。如果需要让申请的时间对申请的优先级影响更大，需要加大弱化因子的值，值的大小取决于物理密码卡的处理速度。

表 5-3 借鉴 HRRN 算法的测试结果

虚拟机	数据	到达时间	竞争时优先级	运行时间	周转时间	等待时间	带权周转时间
VM1	1000M	0s		25s	25s	0s	1
VM2	1200M	11s	1.467	30s	64s	34s	2.13
VM3	800M	15s	1.500	20s	30s	10s	1.5

表 5-4 添加弱化因子后的测试结果

虚拟机	数据	n	到达时间	竞争时优先级	运行时间	周转时间	等待时间	带权周转时间
VM1	1000M	50	0s		25s	25s	0s	1
VM2	1200M	50	11s	1.175	30s	44s	14s	1.467
VM3	800M	50	15s	1.143	20s	60s	40s	3

5.5 迁移功能测试

测试场景：源主机中运行着一个正在使用密码卡的虚拟机，目的主机中也有一个正在使用密码卡的虚拟机，此时将源主机中虚拟机迁移到目的主机中，如图 5-4 所示，虚拟机在进行迭代拷贝时，加解密任务仍在运行，当某时刻还没完成一组数据的加解密时，虚拟机开始了停机拷贝。

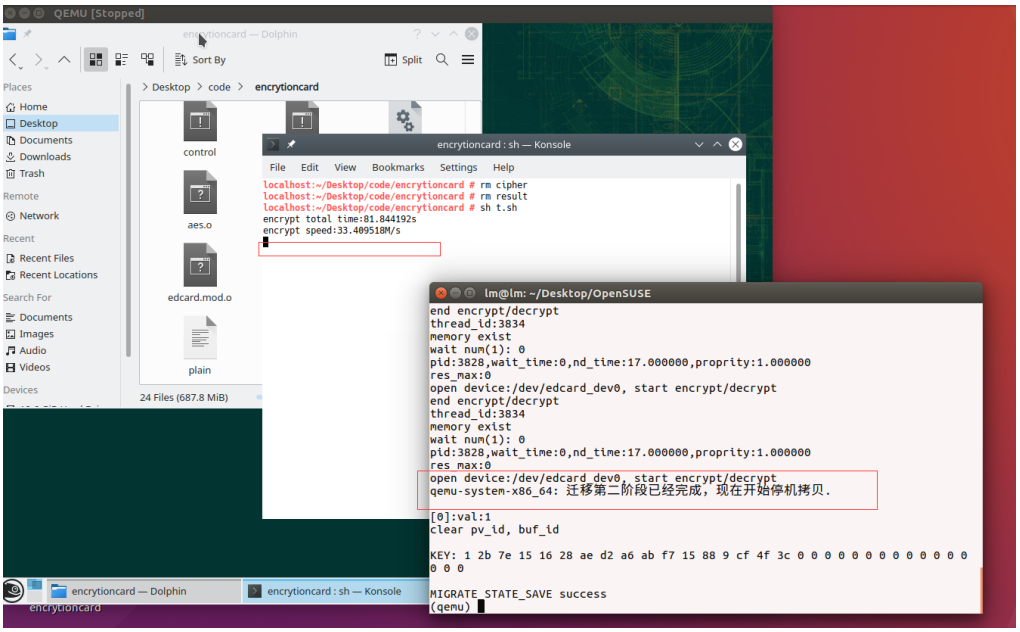

图 5-4 源主机

如图 5-5 所示，停机拷贝完成后，在目的主机中重启虚拟机，并加载密码卡状态信息，继续加解密。最后将解密结果和原文件进行对比，发现 plain 文件和 result 文件没有区别，加解密任务正确完成，迁移没有破坏虚拟密码卡功能。

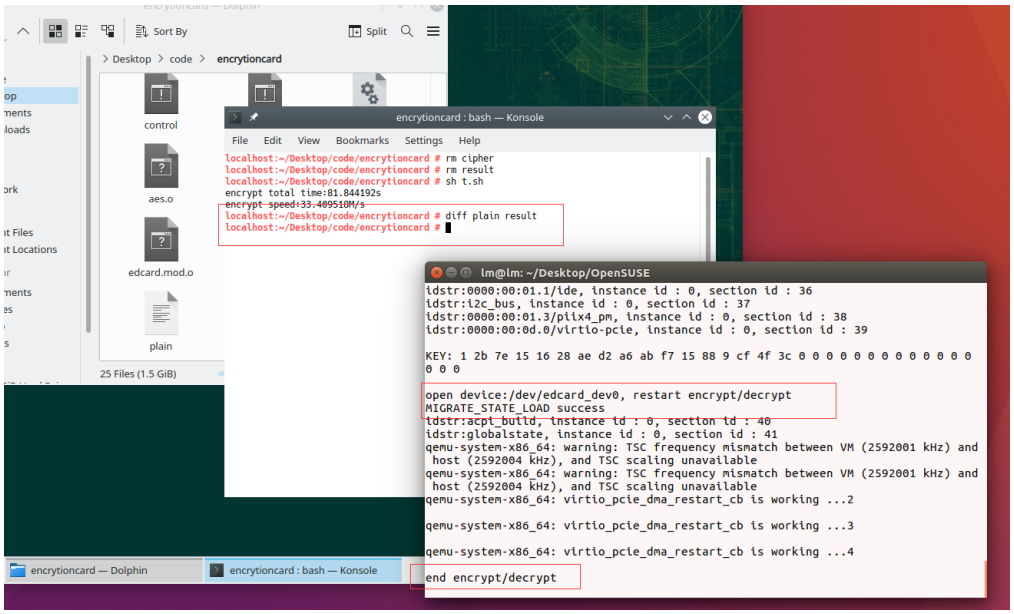

图 5-5 目的主机

5.6 本章小结

本章主要是对文中提到的各种功能的实现的测试，通过测试截图来证明测试结果的真实性，通过这些测试结果来证明方案的可行性和利用价值。

6 总结和展望

6.1 研究总结

首先调研分析了与虚拟密码卡设备相关的研究，确定了自己的研究内容，然后对相关技术理论做了较为详细的介绍。接着通过对虚拟密码卡的应用场景的介绍和分析，体现了虚拟密码卡的应用价值。然后分析了实现虚拟密码卡在云计算技术中应用的需求，根据这些需求提出了可行的虚拟密码卡设计方案，主要工作有以下几点。

（1）给出了基于 virtio 机制的虚拟密码卡实现方案，然后对虚拟密码卡的性能进行优化，减少因为前后端通信造成的性能损耗。

（2）根据多虚拟机对应多密码卡的场景需求，借鉴进程调度算法，设计了一个较为合理的调度算法为这些虚拟机分配密码卡使用权。

（3）为了让虚拟密码卡能够在云计算中使用，提供了虚拟密码卡动态迁移功能，并对迁移的安全问题做了分析并提出解决方案，配合虚拟机动态迁移，保证云计算中用户对虚拟机的正常使用。

（4）对设计方案的具体实现做较为详细的描述，展示出重要的数据结构和部分重要的函数功能。

（5）最终对实现的系统做功能测试和性能测试，通过测试数据来体现方案的可行性和价值。

6.2 课题展望

在高效率虚拟密码卡、虚拟密码卡调度算法、以及虚拟密码卡动态迁移方面的工作虽然有所效果，但是还有很多不足可以在今后更加努力的去改善。

（1）文中采用 virtio 机制去实现了虚拟密码卡，这种实现方式的虚拟密码卡性能与硬件辅助虚拟化的虚拟密码卡性能有不小的差距，但前者又具有设备可共享这一个特点。如何去更好的平衡性能与共享性，在今后的工作中需要更加深入的思考和更多的实验。并且，由于硬件条件缺失，实验时使用的是软件模拟的密

码卡，虽然对设计方案没什么影响，但是对具体代码实现和实验测试有些影响，如果以后条件允许，应该将方案实现到更加真实的环境来测试。

（2）密码卡中的信息的机密性要求很高，虽然文中提出了认证和安全通道等安全策略，但这也只是在一定程度上提高了密码卡信息的安全性，如何让虚拟密码卡迁移时更加安全，今后需要更加深入的思考。

致谢

时光飞逝，转眼间四年过去了，大四生活即将随着毕业设计的结束而结束，在这四年的时间学习和历练中，我学会了很多，与同学和老师的交流开阔了我的眼界，虽然也有争吵和不和，但是终究还是较为顺利，体会到了多人合作完成一个跨期较长的项目的成就感，感受着周围每个人都在不断奋进，提高自己的氛围，感受到了学生时光的美好。

首先我要对这四年里辅导我学习，督促我奋进的老师们表示由衷的感谢，是他们的辛苦教导和丰富的知识让我能在这短短几年中有了较为明显的提高，我也终不再是四年前只会做题的书呆子；我还要感谢在这四年中和我一块儿学习的同学们，我们可能因为一些共同的爱好一起讨论，一起学习，可能因为共同的体育爱好经常一块儿锻炼等等，这些让我的大学生活不再孤单，反而变得极为丰富，这些都将成为我今后美好的、难忘的回忆；我还要感谢我的家人和朋友，虽然家人和朋友们可能不怎么了解我在学校的状况和学习压力，但是在我失落时总是家人和朋友们的话语给了我再次前进的动力，没有他们不会有今天的我。在这里我要特别感谢我的指导教师付才教授，大三暑期实习和大四的毕业设计都是他指导的，他让我了解和熟悉了课题的研究过程，让我更加了解自己的专业知识，在实验室学习的这段时间我提高了很多。

大学生活即将结束，新的旅程即将开始，过往变成美好的记忆，怀揣着这份美好走向前方，未来必然有坎坷，但必然也有美好，在这里祝愿大家前程似锦。

参考文献

[1] Krautheim F J. Private virtual infrastructure for cloud computing[J]. HotCloud, 2009, 9: 1-5.

[2] Chen D, Zhao H. Data Security and Privacy Protection Issues in Cloud Computing[C]. International Conference on Computer Science and Electronics Engineering, 2012, 1:647-651.

[3] 武越, 刘向东, 段翼真. 桌面虚拟化安全访问控制架构的设计与实现[J]. 计算机工程与设计, 2014, 35(5):1572-1577.

[4] Zhai E, Cummings G D, Dong Y. Live Migration with Pass-through Device for Linux VM[C]. The 2008 Ottawa Linux Symposium, 2008, 8(1):261-268.

[5] Pan Z, Dong Y, Chen Y, et al. CompSC: Live migration with pass-through devices[C]. Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments, 2012:109-120.

[6] 苏振宇. 密码卡虚拟化技术研究与实现[J]. 集成技术, 2019, 8(3):31-41.

[7] 孔德举. 虚拟化密码卡的动态迁移访问控制研究[D]. 华中科技大学, 2020:23-42.

[8] Danev B, Masti R J, Karame G O, et al. Enabling secure VM-vTPM migration in private clouds[C]. Proceedings of the 27th Annual Computer Security Applications Conference, 2011:187-196.

[9] Liang X, Jiang R, Kong H. Secure and reliable VM-vTPM migration in private cloud[C]. In 2013 2nd International Symposium on Instrumentation and Measurement, 2013:510-514.

[10] Xin W, Zhang X, Chen L. An improved vTPM migration protocol based trusted channel[C]. 2012 International Conference on Systems and Informatics, 2012:870–875.

[11] 刘涛. 支持动态迁移的加密卡设备虚拟化[D]. 华中科技大学, 2016:6-54.

[12] 张嘉夫. 基于密码卡虚拟化的虚拟桌面隐私保护研究[D]. 华中科技大学, 2017:8-36.

[13] Xu D, Fu C, Li G, et al. Virtualization of the Encryption Card for Trust Access in Cloud Computing[J]. IEEE Access, 2017, 5:20652–20667.

[14] 李超. SR-IOV 虚拟化技术的研究与优化[D]. 国防科技大学, 2010:8-44.

[15] 马龙宇. 基于 SR-IOV 虚拟化技术高速密码卡的设计与实现[D]. 上海交通大学, 2016:22-45.

[16] 侍言. 网络加密服务的虚拟化研究与实现[D]. 华中科技大学, 2020:10-20.

[17] 李刘威. 虚拟机动态迁移算法的优化与实现[D]. 华中科技大学, 2019:14-34.

[18] Cheng G, Jin H, Zou D, et al. Building dynamic and transparent integrity measurement and protection for virtualized platform in cloud computing[J]. Concurrency and Computation: Practice and Experience, 2010, 22(13):1893-1910.

[19] 英特尔开源软件技术中心复旦大学并行处理研究所. 系统虚拟化原理与实现[M]. 清华大学出版社, 2009:74-150.

[20] 任永杰, 单海涛. KVM 虚拟化技术实战与原理解析[M]. 机械工业出版社, 2013:45-103.

[21] Nie B, Du J, Xu G, et al. A New Operating System Scheduling Algorithm[C], International Conference on Electronic Commerce, Web Application, and Communication, 2011:92-96.

[22] Shidali G A, Junaidu S B, Abdullahi S E. A new hybrid process scheduling algorithm(pre-emptive modified highest response ratio next)[J]. Computer Science and Engineering, 2015, 5(1):1–7.

[23] Latip R, Idris Z. Highest Response Ratio Next (HRRN) vs First Come First Served (FCFS) Scheduling Algorithm in Grid Enviroment[C], International Conference on Software Engineering and Computer Systems, 2011: 688–693.

[24] Varma P S. Design of Modified HRRN Scheduling Algorithm for priority systems Using Hybrid Priority scheme[J], Journal of Telematics and Informatics, 2013, 1(1):14-19.

[25] Seethalakshmi V, Govindasamy V, Akila V. To improvise the mapreduce performance using hrrn algorithm[J]. International Journal of Pure and Applied Mathematics, 2018, 119(18):2057–2063.