《openEuler内核编程》

课程讲稿

第十一章 第2讲

KVM

软件所制

第十一章 第2讲 KVM

**学时：**2学时

**教学目的：** 了解KVM的运行机制，包括KVM对内存、CPU的模拟，对IO设备的模拟，与QEMU的交互协作过程。

**课程时间线：**

虚拟化方案 P3~P11

KVM架构 P12~P20

内存虚拟化 P21~P32

IO虚拟化 P33~P53

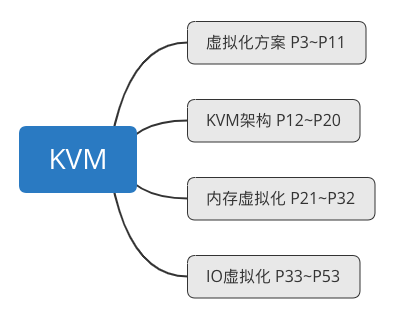
25min

45 min

65min

90 min

**知识框图：**



**课外参考读物：**

1. <https://zhuanlan.zhihu.com/p/19796979>
2. https://www.oschina.net/news/111410/linux-market-growth
3. https://lkml.org/lkml/2020/5/17/314
4. https://www.oschina.net/news/100042/linux-september-2018-stats

**PPT讲稿：**



硬件架构中，操作系统的代码将会认为自身完全占据整个计算机的资源，在内核管理硬件资源的过程中，也需要直接访问硬件和操作内存，因此正常情况下，内核代码需要运行在Ring 0态，以足够高的权限进行资源管理并为用户态程序提供服务。因此，在实现虚拟机的虚拟化时，需要有对应方案来支持Guest OS能够像运行在裸机上一样，进行对硬件的访问、对特权指令的正确执行。对于这一虚拟机的核心需求，根据实现策略的不同，可以分为全虚拟化、半虚拟化。

其中，全虚拟化又可以从软件角度和带硬件辅助机制的角度进行区分。全虚拟化的一大特点是对客户机系统的代码不进行任何修改，在裸机上能够执行的OS代码都应当能够在全虚拟化提供的虚拟化环境中执行。因此，当客户机执行特权指令时，需要有对应机制来满足特权指令的功能。在软件层面，基于二进制翻译的方法主要通过捕获Ring 0态的异常，来提供特权指令的模拟，Guest OS每执行一条特权指令，在Host OS看都都是一次异常陷入、处理、返回，可想而知有较大的性能损耗。另一方面，基于硬件辅助的方法主要体现为CPU硬件实现的虚拟化支持，提供基于硬件机制的指令集支持。借助CPU的虚拟化拓展，提升对各特权指令进行模拟的效率和可靠性。

基于二进制翻译实现的全虚拟化将Guest OS置于Ring 1态运行，Ring 1态的Guest OS调用Ring 0态的特权指令时，将发生异常，进入Ring 0态，异常被Ring 0态的VMM捕获并进行翻译、模拟，然后返回到Ring 1态继续执行，整个过程对Ring 1态的Guest OS透明。

在基于CPU支持的全虚拟化实现中，面向x86指令集的Intel和AMD分别推出了针对自家CPU的硬件拓展。从使用者角度来看，Intel-VT与AMD-V的使用基本相同，此处我们仅介绍Intel-VT相关的内容。Intel-VT在开启虚拟化支持时，将CPU的运行模式分为root模式与non-root模式，两种模式分别支持原先的4个运行等级。根据硬件手册的介绍，正常情况下，VMM运行在root模式下的Ring 0态，Guest OS运行在non-root模式下。

在VMM与Guest OS的相互切换上，Intel-VT提供了VM Entry和VM Exit两个指令集接口，VM Entry相关指令由root模式下的VMM调用，触发VM Entry时，CPU负责到non-root模式，并将已设定好的Guest OS上下文信息恢复、执行Guest OS。VM Exit则是Guest OS在执行过程中触及某些需要在root模式下才能正确执行的特权指令，CPU将自动中止Guest OS，保存上下文，切换到root模式，由VMM判断、模拟对应的特权指令。

从使用者角度，Intel-VT除了包含额外的虚拟机相关指令和寄存器，规范中也规定了软件需要满足的控制结构体布局，即VMCS。VMCS是VMM与CPU交互的基础，VMM负责维护、修改VMCS的内容，CPU负责在需要的时候载入VMCS中的内容到上下文、保存上下文信息到指定内存位置（以VMCS内存布局）。

Intel-VT的一大优势在于避免了逐条指令的二进制翻译，借助硬件机制降低了编程的难度，同时提升了虚拟机的执行效率。

半虚拟化允许对客户机内核进行修改，用Hypercall替换无法虚拟的指令（如特权指令）。Guest OS在运行时通过Hypercall与Hypervisor通信。这种做法的典型实例是Xen，由于对Guest OS源码进行了修改，与Hypervisor可以联动进行更多优化，半虚拟化的性能损耗较低；但也由于修改Guest OS源码，Guest OS开发者需要将Hypercall对应的虚拟硬件情况作为一种架构，长期进行代码维护，这种虚拟化技术的兼容性较差，维护成本较高。

KVM全称为Kernel Virtual Machine，是Linux内核对虚拟化（VMM）的官方支持，实现形式为内核的可加载模块。KVM的实现以内核中已有的虚拟内存、进程管理等部分为基础，极大地降低了实现复杂性。同时，从设计上而言，KVM只提供CPU&内存的初始化，并通过特殊的设备文件/dev/kvm来给出在用户态实现模拟I/O接口的途径。客户机OS运行时，宿主机OS上的用户程序（以QEMU为例）正处于进行系统调用的状态，进行虚拟机控制或I/O接口模拟。

当KVM被加载并执行时，原系统的Linux内核承担起VMM的角色，并将Guest OS作为进程管理起来。

从这张图可以看出，当KVM模块正常运行之后，运行在Host用户空间的QEMU通过系统调用可以与内核空间的KVM进行交互，具体而言是通过/dev/kvm设备文件相关的系统调用，QEMU能够控制各虚拟机的运行并为虚拟机提供I/O接口的虚拟化实现，而KVM模块只负责为虚拟机提供虚拟内存支持并让虚拟机参与CPU调度，对于I/O接口的虚拟化，仅发挥转发作用。

KVM被实现为一个Linux内核模块，当KVM模块被加载时，首先初始化内核中与（虚拟机）虚拟化相关的数据结构，然后尝试对硬件进行虚拟化模式的设置，包括：检测当前CPU支持的虚拟化拓展、打开CR4中的虚拟化开关、执行VMXON指令来将宿主系统置于虚拟化模式的根模式；最后，在VFS层创建特殊的设备文件/dev/kvm，等待宿主OS用户空间的程序（一般为QEMU），通过设备文件/dev/kvm发送下一步运行指令。

QEMU与KVM的通信通过Linux提供的系统调用规范实现，借助了拓展性极强的IOCTL系统调用，所有针对特殊设备文件/dev/kvm的IOCTL系统调用，在进入内核之后都将被转发到KVM的处理代码流程中，来做出对应的响应。同时，也由于/dev/kvm的特殊性，持有对应文件描述符的进程无法被fork，此处主要是考虑到直接复制一个正在运行的虚拟机需要的代价太大。

从左侧的运行流程图可以看出QEMU、KVM、Guest OS三者的关系。从QEMU角度来看，启动一个VM、为VM提供IO设备虚拟化、结束一个VM等，都是在连续的系统调用下完成，当系统调用返回用户态，QEMU执行，Guest OS的执行阻塞；当QEMU进行系统调用，进入内核态，QEMU所在进程阻塞，Guest OS参与调度。

当QEMU准备启动一个Guest OS，不仅需要通过/dev/kvm相关的系统调用在内核中进行Guest OS信息的设置，也需要对驱动及其相关工具进行设置，为之后的IO设备虚拟化做准备，比如对显示器的显卡驱动或framebuffer进行设置，在屏幕上分配一块区域用于模拟虚拟机的显示器。

在KVM看来，一个虚拟机就是一个QEMU进程，虚拟机中的CPU称为vCPU，由一个线程模拟。Guest OS中的线程如果要加载到真实CPU上执行，首先需要经过Guest OS线程调度器的调度，被载入到某一个vCPU上，然后需要该vCPU对应的真实线程被KVM调度器调度，vCPU被加载到真实CPU上执行，从而让Guest OS创建的线程在真实CPU核心上执行。

经过前面阶段的学习，我们对实际操作系统中的虚拟内存管理有了初步的了解，知道需要依靠MMU和TLB等硬件机制来帮助操作系统管理虚拟内存。在全虚拟化的实现中，运行在虚拟环境中的Guest OS同样需要一个虚拟的MMU支持，并且该MMU的虚拟化实现应当是对Guest OS而言完全透明的。

目前，为Guest OS提供内存虚拟化支持的实现主要有两类，一类是通过软件方法实现的影子页表技术，另一类是通过硬件辅助实现的，即使用Intel的EPT技术或者AMD的NPT技术。KVM采用了硬件辅助的内存虚拟化形式。

在继续介绍之前，我们先对之后将会涉及到的几个地址概念进行一下统一，包括：客户机的虚拟地址（GVA），客户机的物理地址（GPA），宿主机的虚拟地址（HVA），宿主机的物理地址（HPA）。HPA也就是真实物理地址，最终用于在内存上寻址。

“影子页表”顾名思义，为VM中的每一张页表都维护一个影子。具体设计方案是，VMM捕获Guest OS执行CR3写指令时发生的异常。在Guest OS准备加载新的页表时，由于异常陷入，执行权被VMM掌控，实际加载到CR3中的是影子页表；同时，为影子页表中的所有内容添加写保护，GuestOS在试图修改页表内容时将触发写保护异常，执行权再次被移交给VMM，进行实际的页表修改，即影子页表的修改。

每一个VM中的进程在逻辑上都对应着一张页表，实际在影子页表的实现方法中，每一个VM中的进程对应着两套页表，一套是设置为只读、页表内容为GPA；另一套被称为影子页表，对VM不可见，页表内容为HPA，这也是实际被载入到CR3的页表。在这里，我们将第一套页表称为VM页表，第二套页表称为其影子页表。影子页表将尽可能与VM页表的内容保持同步，但出于节省宿主机物理内存的角度，也可以仅保持最小代价的同步，在VM页表修改后不立即分配物理内存并映射到影子页表，减少物理内存占用。

当VM的内核试图修改页表内容时，由于存在写保护，将会触发异常，控制权移交给VMM，VMM可以分配真正的物理页帧，修改VM页表的内容和影子页表的内容，然后返回到VM内核继续执行。在VM内核看来，本次修改没有任何异常发生，但实际执行load cr3的结果已经不是预期的结果了。

在Guest OS的内核初始化阶段，“尚未启用”页表，直接访问物理地址或逻辑地址，由于虚拟环境的需要，此时硬件上的MMU应当处于开启状态，将VM中的物理地址（GPA）映射到VMM管辖的实际物理地址（HPA）。当VM“载入”VM页表的基地址时，VMM将载入的页表切换为影子页表，同时还需要在VMM中维护一张映射表，记录VM页表基地址与对应影子页表基地址的对应关系，即从GPA到HPA的映射。

影子页表提供了从GVA到HPA的直接转换，不需要借助额外的硬件机制，仅需要CPU支持原有的MMU即可。但由于为每一个VM进程都维护了一张额外的影子页表，需要的额外内存开销较大。同时，使用纯软件模拟的方法，在VM内核运行过程中将多次触发页表的写保护异常，不利于提升虚拟机的效率。在开发过程中，影子页表的机制对于开发人员提出更高的要求，debug难度比较大。

与影子页表的软件设计方法不同，Intel提供了硬件辅助的VM内存虚拟化，EPT技术。EPT本身的设计与MMU中的页表机制相似，结构几乎完全一致，在x64情况下同样支持多级，EPT中的映射将GPA映射到HPA。当访问不存在的页时，将处罚VM Exit，Guest OS终止，由VMM处理缺页异常。

开启EPT之后，在VM中访问的每一个GPA都将被CPU经过EPT页表重定向为一个HPA，VMM通过设置EPT相关的寄存器，能够设置VM的物理地址空间所对应的实际物理内存范围，进行页式管理，在物理页帧管理方面，让VM与一般的用户进程没有太大差别。

如图，这是Intel硬件手册中对于EPT查询过程的说明。当VM运行时，GVA经过MMU转换为GPA之后，并不会直接访问物理内存，而是再经过EPT的映射，才能到达物理内存。映射过程与MMU转换的过程基本类似，EPT的基地址由VMM设置在EPTP寄存器中。

从前面的介绍可以看出，EPT机制为VM内存虚拟化提供了可靠的硬件辅助，VM内核修改Guest OS页表时无需多次触发写保护，效率提高了。另一方面，一个VM只需要设置一个EPT表，整体内存资源占用大幅减少。对于开发人员来说，只需要正确设置EPT表就能将VM的内存与Host的内存完全隔离，大大降低了开发难度。

QEMU是一款模拟器，既能够在纯软件情况下模拟出用户程序/内核的运行情况，在硬件架构兼容的情况下，也能与KVM结合使用，借助KVM来创建、管理虚拟机。在纯软件情况下，QEMU能够进行复杂的指令翻译，运行目标架构有别于当前硬件的软件，如面向不同指令集编译的内核。

QEMU具有两种模式，用户模式用于模拟用户程序的执行，系统模式则可以模拟不同硬件架构的操作系统执行情况。

在用户模式下，QEMU支持模拟运行面向Linux/BSD的用户态程序，具体做法是对程序中的Syscall指令进行转换、翻译。在用户模式下，QEMU支持POSIX SIGNAL处理以及clone系统调用。

在系统模式下，QEMU能够模拟运行整个系统，包括内核及其用户态程序。纯粹的软件模拟性能较差，如果硬件架构允许，QEMU一般配合KVM/Xen等Hypervisor使用，例如，加载KVM模块之后，QEMU通过系统调用提供IO设备的虚拟化，从而为用户态程序模拟出完整的系统运行情况，并提高了效率。

在QEMU和KVM的使用中，对于IO设备的虚拟化实现，大概有三种不同的实现。包括全虚拟化I/O设备实现，完全由用户态来模拟硬件行为，然后将结果返回到VM中；除此之外还有准虚拟化I/O设备，最常见的即Virtio，通过对VM内核的驱动实现提出一定要求，改进了虚拟IO设备的性能，并成为Linux中的设备标准框架。也可以借助前面提到的VM虚拟内存管理，为VM直接分配PCI设备，提升效率。直接分配IO设备的做法包括单虚机单设备的PCI设备分配，以及多虚机单设备的SR-IOV技术。

I/O设备的全虚拟化将虚拟I/O设备的代价全部放在Host系统的用户态管理程序上，例如QEMU。Guest OS的驱动对于底层是否为真实硬件并不清楚。VMM通过捕获、处理Guest OS执行驱动时的异常，来实现I/O设备的虚拟化。当VMM有实现I/O设备虚拟化的需求时，KVM从QEMU进行系统调用的状态返回，执行权交给QEMU，QEMU在用户态可以进行纯软件的模拟，也可以借助其他已经实现的硬件驱动来模拟相关功能。

全虚拟化实现中，整个I/O设备的虚拟化过程完全由VMM隔开，Guest OS与Host Manager（QEMU）彼此之间不知道对方的状态。

这种实现相当灵活，可以模拟的设备种类非常多，不限于已有设备，也可以直接定义虚拟设备。但由于单次IO操作所需要的调用路径长，上下文切换次数多，同时在用户态/内核态、VMM/Guest OS之间的数据复制也多，效率较低。

出于对全虚拟化性能劣势的考虑，准虚拟化的概念进一步发展，提出了Linux上的一套标准设备框架——virtio。Virtio提供了一种标准的设备框架，能够用于在Guest和Host之间进行虚拟设备层面的交互。不同于全虚拟化的I/O设备，基于virtio标准实现的I/O设备虚拟化，需要Guest OS在内核中提供对应virtio接口的前端驱动，而Host OS中的QEMU则负责实现对应于virtio接口的后端模拟逻辑。由于双方都遵从virtio接口规范，因此可以通过共享内存等策略，来避免频繁地通过VMM进行异常捕获，而让Guest OS与QEMU的I/O模块直接通信，大大提高了效率。

实际运行过程如图，可以看到Guest和Host（QEMU）中都存在virtio的控制器，而vring则是一个共享的环形队列，整个I/O虚拟化的实现不再重度依赖VMM的异常捕获，减少了内存拷贝次数和上下文切换次数。

与全虚拟化相比，KVM在准虚拟化情况下的运行负担更轻，进行异常捕获和数据拷贝、传递的代码逻辑能够从内核中转移到用户程序、Guest OS驱动，减轻了内核的复杂度，也减少了数据传递导致KVM受到攻击或崩溃的可能性。

具体的实现中，KVM通过中断的方式来通知QEMU获取数据，并通过类似手段通知Guest从队列中取数据。

基于virtio框架规范，Linux内核中目前实现由五类前端驱动程序，所涉及硬件包括常见的磁盘、网络设备、PCI设备等。由于Linux内核本身在持续维护且被广泛使用，从而使得virtio这种同时修改前端和后端的设计思路具有一定活力，并且，由于基于Virtio实现的驱动程序进入了内核的主分支，类似Xen那样维护一个额外版本的开销无形之间被抵消了，更促进了virtio框架的普及和使用。

准虚拟化相比全虚拟化而言，提升了性能，但是在性能上仍然存在缺陷，与在物理机上的性能相比还有较大差距。如果VM能够直接操作物理设备，那么在性能上能否做到接近宿主机性能呢？在安全性方面是否会造成影响？由虚拟机直接操作的物理设备，在使用率上和设备数量方面是否会有一定限制呢？

PCI嫁接技术能够在VMM的允许下，将一个物理的设备（如图中的网卡）分配给一个VM。KVM在分配过程中参与程度较高，但在VM与物理网卡交互时可以做到基本不打断，不需要进行异常捕获也不需要别的程序来进行虚拟的网卡模拟，让Guest OS在可控的情况下能够像运行在物理机器上一样，使用物理网卡，独占整个设备。图中展示了一种PCI嫁接技术的应用场景，Guest OS在KVM的允许下直接操作NIC网卡，进行网络信息的收发。

PCI嫁接技术带来的最直观的优势，就是在I/O设备的模拟中，将性能损失几乎降低到了贴近理论极限。在某一些特定场景下，Guest OS操作物理设备（比如网卡）的过程可以做到长期不被VMM打断，性能上与不进行虚拟化时持平；另一方面，使用PCI直通技术能够减少VM/VMM、KVM/QEMU上下文切换，减少VMM管理压力的同时，也降低了VMM的实现复杂性。但缺点与优点同样明显，将一个设备绑定到一个虚拟机，较长时间内该设备无法被其他虚拟机共享，设备数量将会限制能够使用该类设备的虚拟机的数量。另外，单个设备在单个虚拟机中多数时候都满载运行的可能性不大，对物理资源造成了不可忽视的浪费。

PCI嫁接技术的性能很优异，但只能将一个设备分配给一个虚机。为了实现一个设备可以被多个虚机共享，同时又能够直接分配的目的，SR-IOV规范诞生了。SR-IOV定义了一个标准化的机制用以原生地支持实现多个客户机共享一个设备。不过，目前 SR-IOV （单根 I/O 虚拟化）最广泛地应用还是网卡上。

支持单根I/O虚拟化的设备能够提供两种功能，物理功能是完整的PCIe设备，像普通的PCI设备一样能够被发现、管理与配置；虚拟功能则提供简单的PCIe功能，只能处理I/O。每一个VF都是从PF中分离出来，每个物理硬件都有VF数量的限制，可以将一个PF虚拟为有限个VF用于分配。

Hypervisor 能将一个或者多个 VF 分配给一个虚机。在某一时刻，一个 VF 只能被分配给一个虚机。一个虚机可以拥有多个 VF。在虚机的操作系统看来，一个 VF 网卡看起来和一个普通网卡没有区别。SR-IOV 驱动是在内核中实现的。

单根I/O虚拟化的使用相比其他技术有更多限制，不仅需要KVM在软件层面提供支持，还要求CPU本身带有像VT-x这样的硬件拓展，同时对设备本身正确实现了SR-IOV规范也是必须的。

在最后我们对I/O设备的虚拟化进行一个简单的总结，从全虚拟化到准虚拟化，从PCI Pass-Through到SR-IOV的单根I/O虚拟化，可以看到，虚拟化是一个面向实际的技术点，性能和资源利用是其中很重要的评价指标。

KVM是目前实现非常成熟，应用非常广泛的VMM，以Linux模块的形式实现，同时明智地将I/O设备虚拟化等复杂逻辑剥离，保持了简洁高效。内存虚拟化是虚拟机安全和性能的主要突破点，在保证虚拟机正确运行、无法攻击宿主机的同时，尽可能加快虚拟机中的内存访问速度，是重要的设计思路。I/O设备的虚拟化则更体现了虚拟机使用中的不同点，对于一个完整运行、提供服务的计算机系统，有没有可能仅通过虚拟化手段，让该系统的部分运行于虚拟环境，部分运行于真实环境呢？