[5-5 CPU&内存虚拟化 1](#_Toc105267315)

[CPU虚拟化 1](#_Toc105267316)

[QEMU/KVM 7](#_Toc105267317)

[KVM\_RUN时发生了什么 8](#_Toc105267318)

[case：WFI 9](#_Toc105267319)

[case：IO指令 9](#_Toc105267320)

[内存虚拟化 10](#_Toc105267321)

[客户物理地址 11](#_Toc105267322)

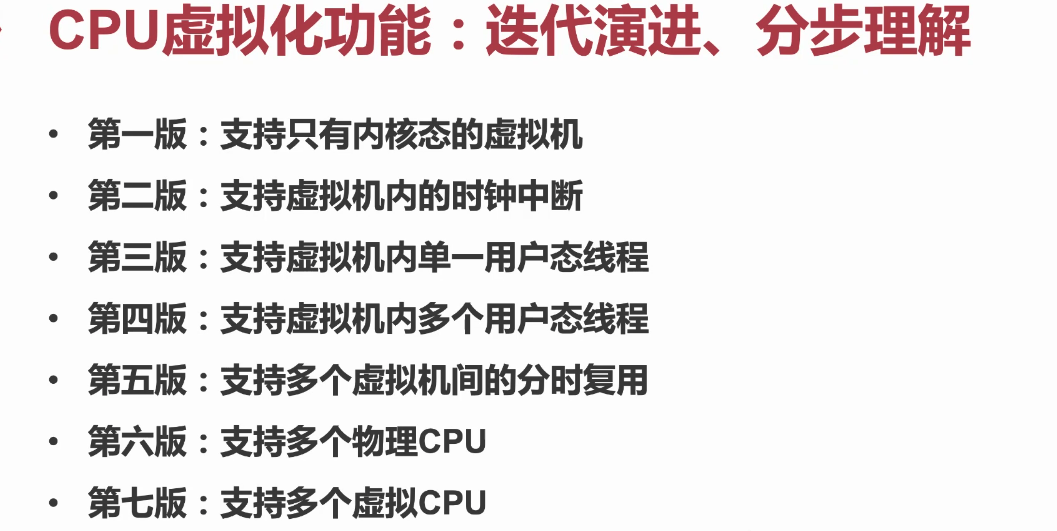
[direct paging 13](#_Toc105267323)

# 5-5 CPU&内存虚拟化

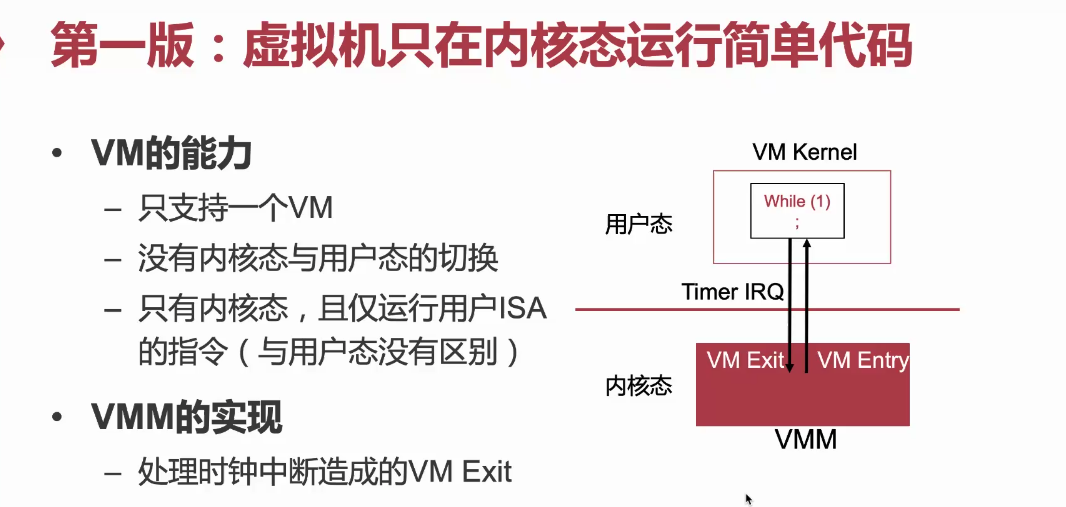
今天我们还是讲系统虚拟化，我们之前学OS的时候，讲到了调度，外设，网络，这里你既然要虚拟一个完整的虚拟机，这些东西肯定都要有。

今天我们聚焦在CPU虚拟化，前两节课我们讲传统硬件不是完全可虚拟化的，我们前两次课讲的东西主要在缺陷弥补上。CPU虚拟化到底是什么呢?之前OS看到的就是真实的物理核，我们经常听到说给虚拟机配几个CPU，这里的CPU到底是什么？

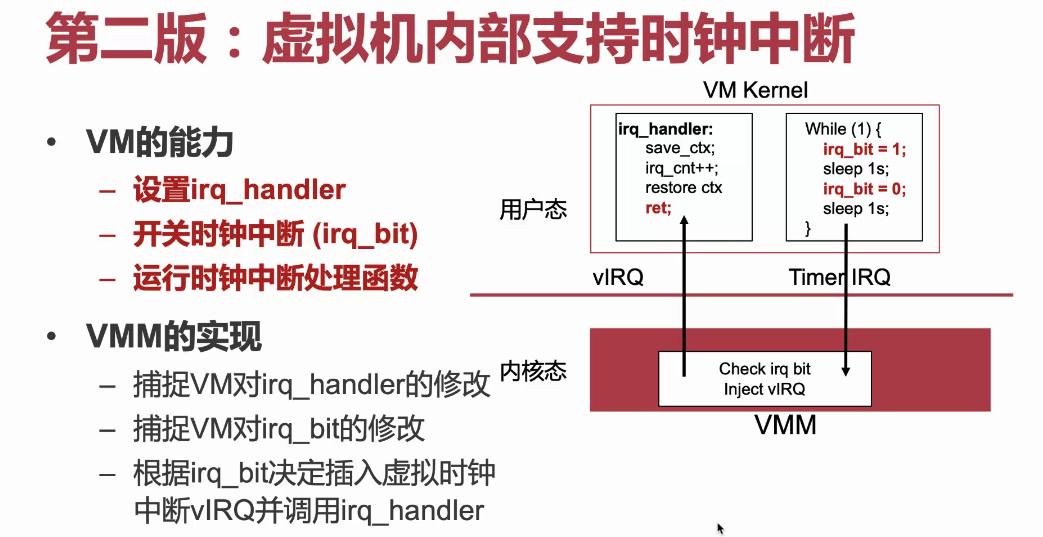
## CPU虚拟化



目前没有讲到IO虚拟化，内存虚拟化，目前虚拟机功能还是阉割的，比如没法配页表，没有IO功能。



物理机上只跑一个虚拟机，只有一个内核态，仅运行用户ISA，不会陷入内核态。这个虚拟机在内核态运行不会造成任何下陷。右图例子，这个内核态做一个while 1的死循环。我们的hyper visor只要把这个循环跑下去就可以了。只要支持时钟中断（怎么保证系统不被这个while 1 loop带崩掉呢？）只要用物理时钟中断回收控制权就可以了。time IRQ把控制权换回VMM，要保存用户态的所有寄存器，第一版非常简单。和没有虚拟化的用户态进程没什么区别。



它要不断的触发配置时钟中断，虚拟机内部有处理时钟中断的需求。CPU上只有一个物理时钟中断，已经被VMM用了，这时候怎么办呢？

借助于VMM，通过软件的方法虚拟出一个时钟中断。怎么让用户态感知不到底下是软件还是硬件呢？

第一点虚拟机能去配置它，当中断触发之后能设置handler，第三点有中断屏蔽功能，做一些敏感操作的时候能不被打断，提供一个中断屏蔽位。

右上角那张图：没有中断触发的时候kernel有一个逻辑，这个while 1循环里做了几件事：

写irq\_bit：irq\_bit 1表示中断打开，能接受中断。

irq\_bit：0表示不想接受任何时钟中断。

它的意图告诉我们在这个1s的区间，它是愿意接受时钟中断的。

虚控应该可以知道，要在这个时间中断给它插进去，为什么叫插呢？因为我们要通过软件的方法插进去。虚控的中断是硬件调irq handler，而VM的时钟中断是通过VMM主动调用虚拟机内核的IRQ handler。

ret就是回到它下陷的位置。

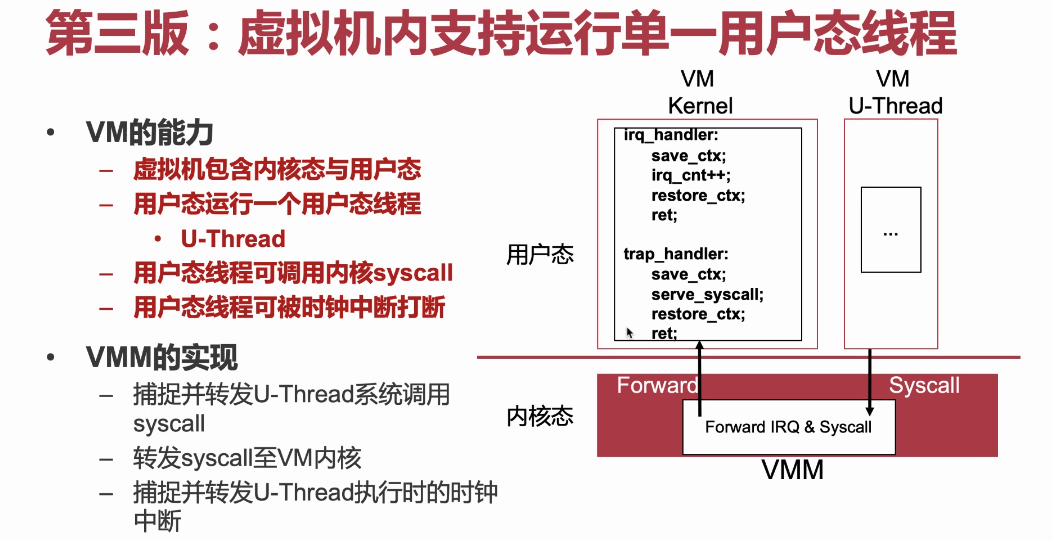
VMM实现：

1. VM设置irq\_handler应该trap下来，然后把这个地址记住。

2.它也应该知道虚拟机什么时候想接受中断，什么时候不愿意接受中断，比如访问irq\_bit的时候下陷到内核态（比如是个特殊的寄存器）。

3.当真的中断触发的时候，能动态插进去。

4.怎么插呢？借助于物理上真实的物理中断，中断来的时候打断VM的执行，进入VMM，然后VMM处理的时候可以看一下虚拟机的时间片到了没有，irq bit设置了没有，如果设了，就调用这个irq handler，返回虚拟机的时候把pc换成irq handler，ret的时候也会下陷，这时候再把控制流转回下陷的位置，VMM保存这个位置，其实guest kernel也是可以知道的，VMM传给kernel的时候可以把这个值传给它。



为什么要时钟中断呢？支持多线程。

此时这个VM：

用户态线程可以调syscall

可被虚拟的时钟中断打断

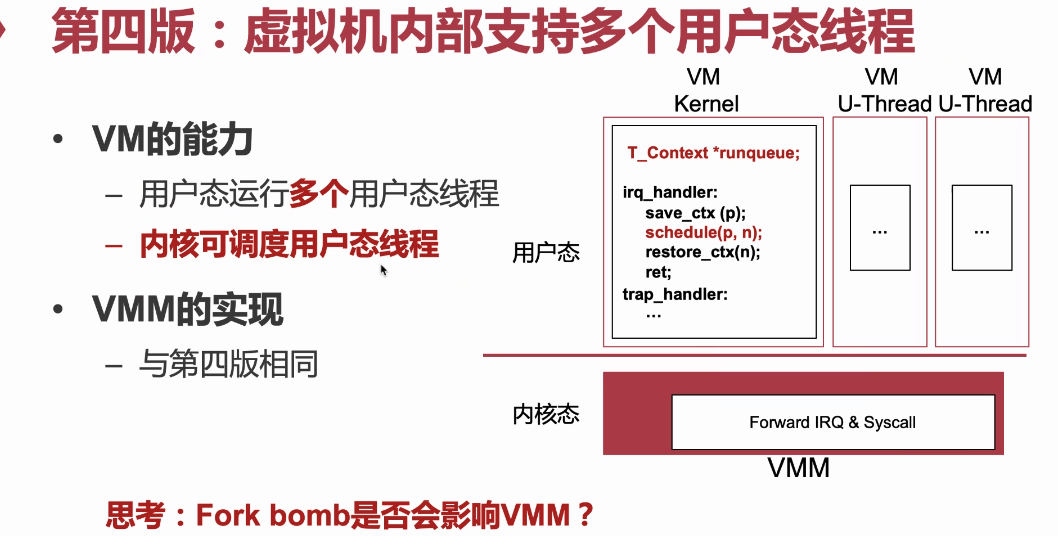
trap\_handler只用来处理syscall

举个例子syscall怎么处理呢？用户态调用svc，因为这是个sys isa，就下陷到VMM，看一下下陷的原因是啥，发现是syscall，不是我能处理的，所以转发到VM kernel。怎么知道转发到哪呢？虚拟机启动的时候会注册自己有哪些trap handler，注册时候VMM应该能捕捉到这个注册的过程，从而记录一下各个syscall的offset。发现是syscall，就把这个ip直接设置到syscall的offset。执行完之后调eret，这个也是sys ISA。也会下陷。再返回用户态线程。

用户态线程调syscall应该替他转发，如果内核调syscall怎么样呢？在我们这版实现里：这个系统出了问题了，不应该给它转发，通过这个区别，这个VMM应该能标记出哪些是内核态，哪些是用户态。怎么做呢？启动的时候可以，调eret的时候进入新的线程标记一下这个新的线程是uthread。

这是第三版的虚拟机，第四版呢？

支持多个用户态线程。



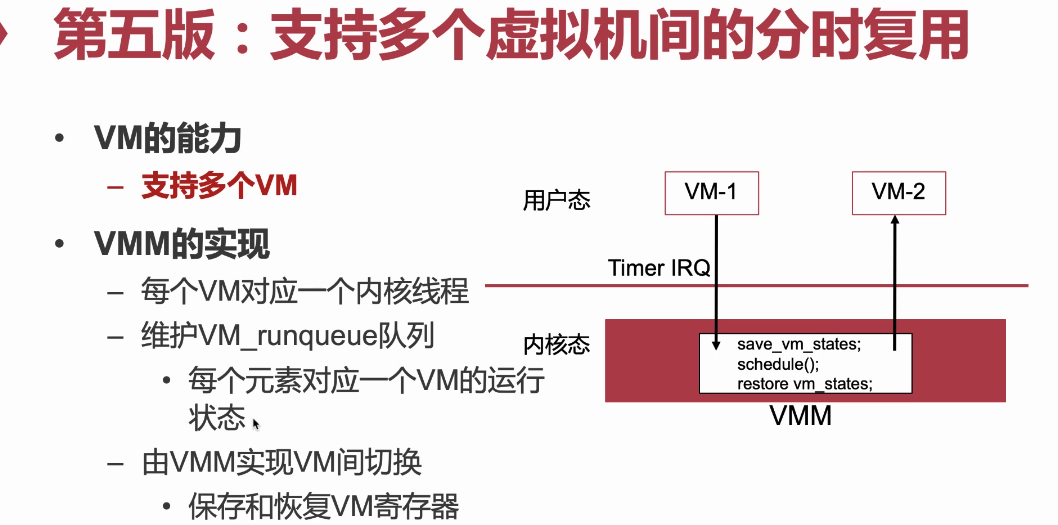
维护一个runqueue，每个元素应该是thread context，保存线程的状态。第一，用户态线程跑着跑来了一个time interrupt，VMM看了一下VM的时间片也到了，该插一个虚拟的时钟中断，然后进kernel的irq handler，VM 的kernel先保存ctx然后进调度器了，这个调度策略自己随便怎么实现，接口第一个参数就是previous thread，第二个就是next thread。从调度器出来，恢复的就是下一个要跑的新的thread 的ctx。然后ret返回到新的用户态线程。（其实是先返回到VMM，然后VMM再返回用户态）

VMM其实不知道你刚才干了什么它只知道你处理完了timer，它就会看你的寄存器里的ip是什么，然后把这个ip设过去。返回用户态。

大家看到为了支持用户态线程，内核态虚控没做什么事情。只不过多了runqueue里面的一个entry。

思考题：Fork bomb是否会影响VMM？

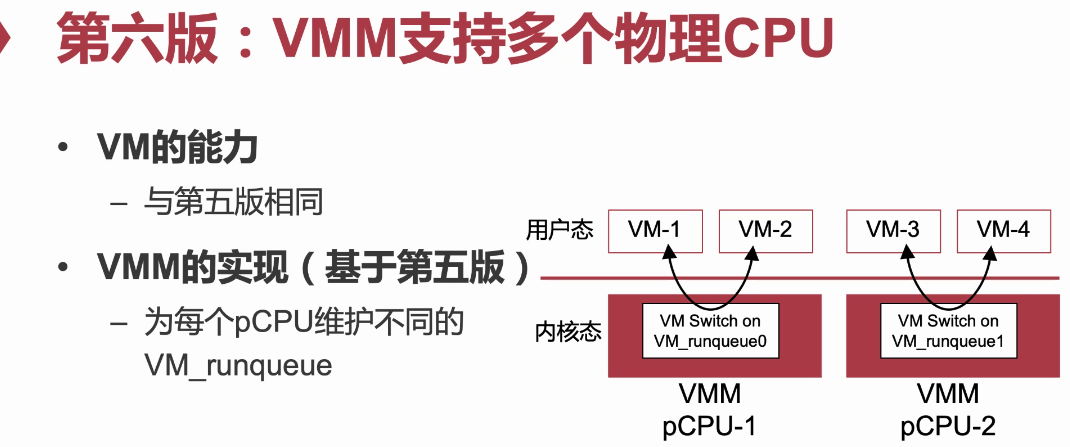
A：不会影响VMM，因为它都不知道上面在干什么，上面有哪些有哪些线程。因为这些线程信息都保存在VM kernel的struct里。它不关心虚拟机内核的实现，它只要在虚拟机的系统isa中做一层trap之类的事情就好了。进一步思考一下：如果runqueue爆了，里面entry次数不断增长，kernel内存就爆了，会不会对VMM有影响呢，答案又是不会的，在后面内存虚拟化的时候会讲到，VMM有办法限制虚拟机内存使用量。



给这个机器上跑多个虚拟机，怎么跑多个虚拟机呢？这个图上只画了两个虚拟机。虚拟机之间的切换是靠物理CPU的时钟中断触发的，下陷到内核态VMM，它会有自己的调度，给虚控增加调度。之前看到加用户态线程加runqueue，vmm也有自己的runqueue，每个entry也是context，不过这个ctx是vm的context，**这个可以理解为内核态的线程。一个vm就是内核态的线程**。什么是线程——一组寄存器和一个stack。所以我们ctx里面为每个内核态线程保存一组寄存器状态和stack，它和虚拟机怎么对应上的呢？进入新的ctx之后，这个ctx为每个虚拟机服务的。

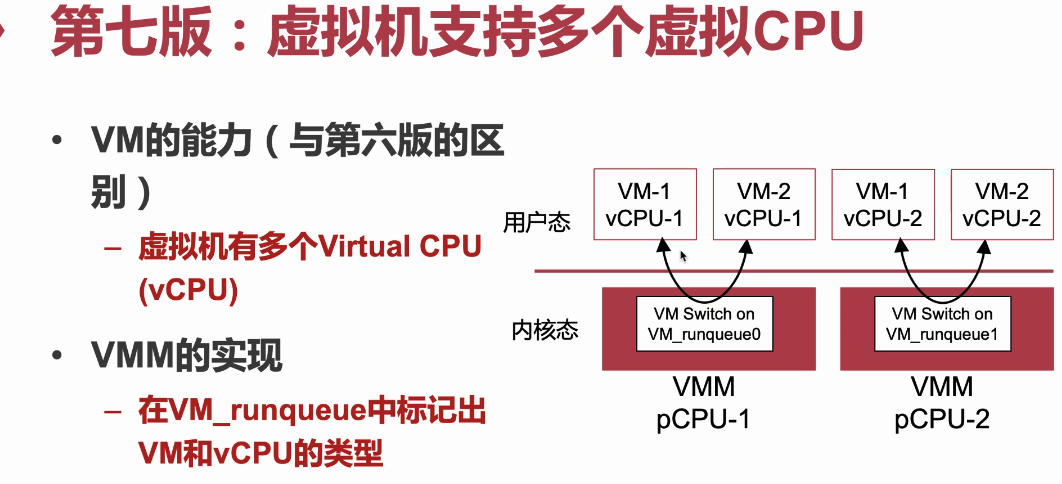
**时钟中断触发——进入内核态VMM，保存下陷的所有的虚拟机寄存器状态——进调度器——下一个要跑的vm的context拿出来，restore到物理寄存器——跳到新的vm执行**

不用关心vm内部的实现，在我看来它只是一组寄存器嘛。



VMM能不能利用多个物理CPU呢？

每个物理核上维护一个对应的runqueue，这个runqueue只为物理核调度服务。当我这个物理核上跑的是vm1和vm2，vm1跑的时候触发一个时钟中断，下陷到内核，内核作相应的调度，然后VMM pCPU切到vm-2，这个过程中pCPU2完全不受影响，vm3仍然可以运行。每个核都是独立的。



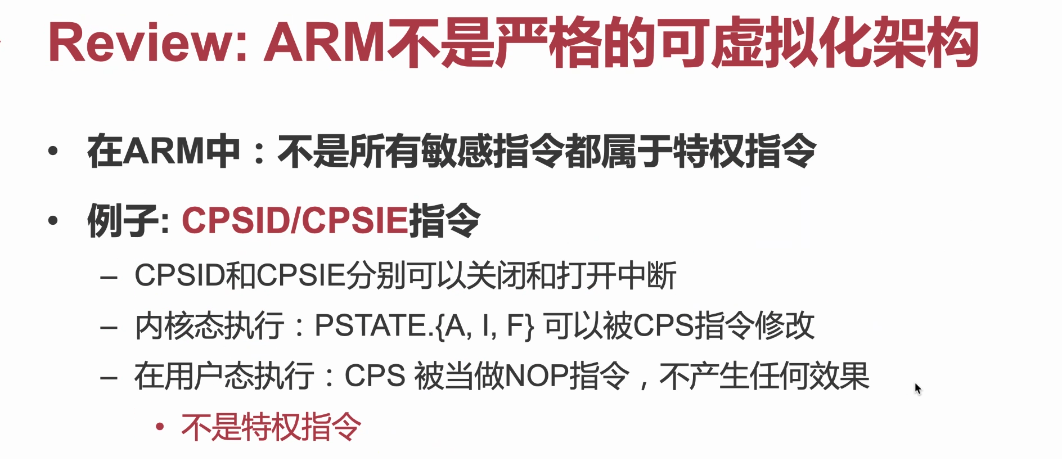
vCPU的概念，怎么去虚拟这个虚拟CPU呢？我们VMM要做的就是每次虚拟启动的时候如果配了两个vCPU，对于VMM来说就是多了两个context而已。调度也是非常相似，vm1有两个vCPU，一个在pCPU1（vCPU1）上一个pCPU2（vCPU2）上，调度也是一样的。

有了VCPU的概念之后，第六版调度的基本单元是虚拟机，这里调度的基本单元是vCPU。对于用户态来说变化巨大，因为它有多个CPU了。VM就可以在每个vcpu上维护一个runqueue，做一些传统操作系统的事情。

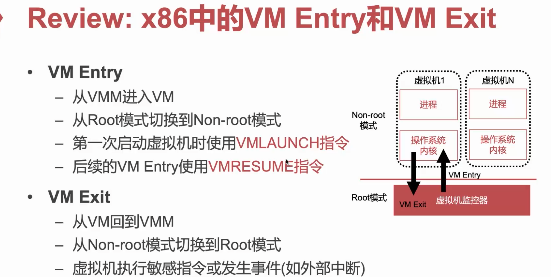
VM虽然看到两个VCPU，但是它不能确定自己真的就跑在两个pCPU上。虚控也不能保证任一时刻一个虚拟机的vCPU都在跑。也是分时复用的。

> 上节课提到一个很有意思的例子：spin lock进critical section的时间很短。

上面讲的这7个版本都是理想化的模型。但并不是所有的sys ISA都是可下陷的。



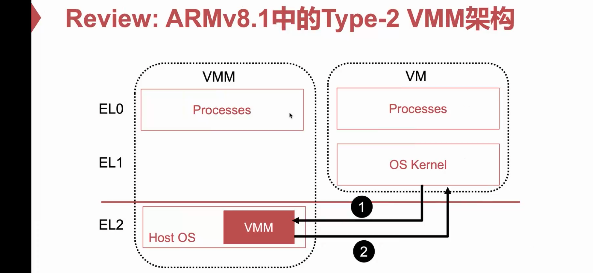
上节课讲了硬件虚拟化，让所有敏感指令都下陷。不光把敏感指令都下陷，其实还做了一些其他事情，比如不该下陷的不下陷，比如syscall不下陷。



主动下陷：syscall

被动下陷：irq（时钟中断）

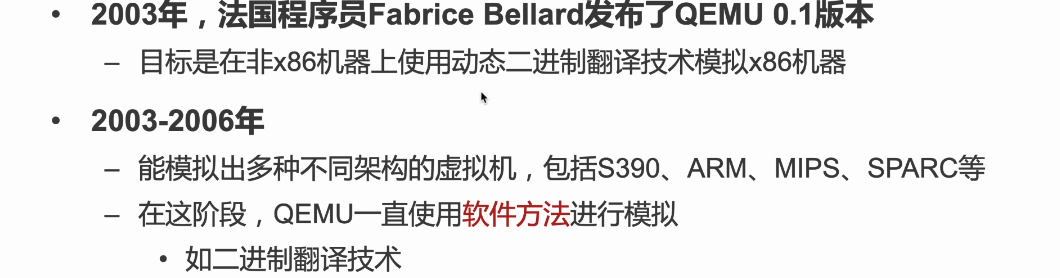
Armv8.0是为type1设计的，比如Xen。



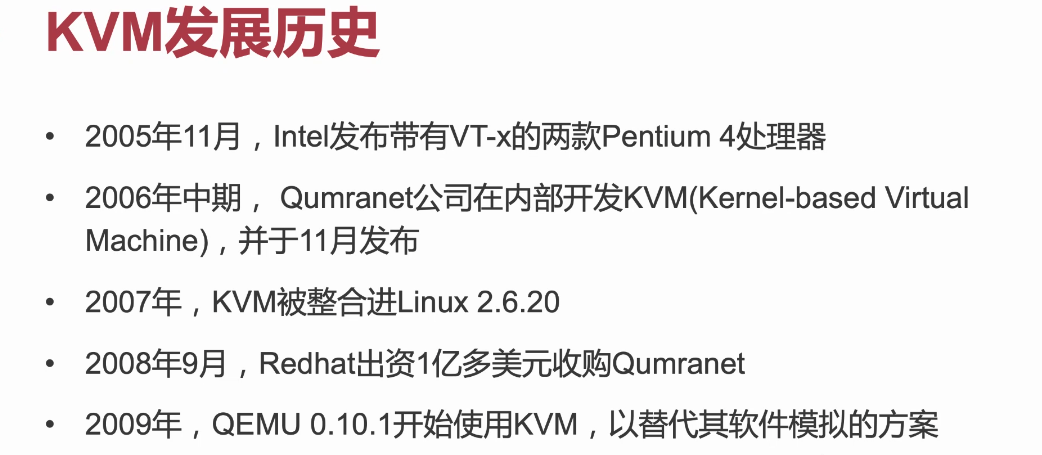
这里有个问题，是回到虚拟机的用户态还是虚拟机内核态呢？不重要，只要把寄存器配置到位，就会回到之前的位置。

## QEMU/KVM

其实是两个完全不一样的东西。



在同一台机器上模拟不同的架构。通过dynamic translation的方法。



kernel based vm 虚拟化功能全部基于kernel实现，这个kernel就是linux kernel。

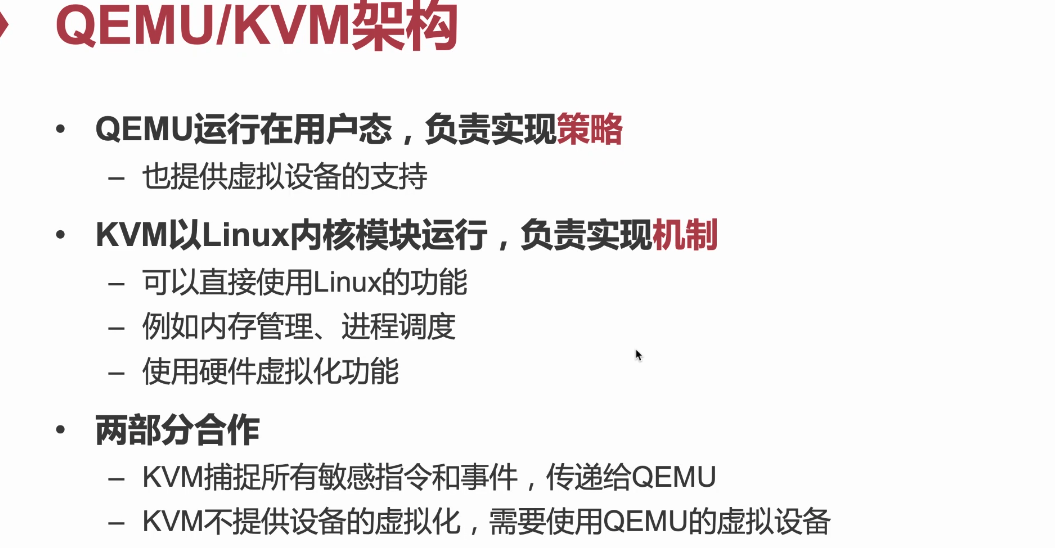
而且完全用硬件虚拟化的机制。

05Intel VT-x硬件出现——06软件出现（kvm linux加了个硬件虚拟化的驱动）—— 07整合进主线

对比一下Xen：99年开始Xen —— 03SSB发布 —— 适配的时候必须要改guest也就是linux

Xen是一个协同设计的方法，要改linux，所以一直没有被整合进主线。

为什么说用kvm替换软件方案呢？qemu主动用硬件虚拟化的能力控制虚拟机，怎么实现虚拟机各项功能仍然是qemu实现的，比如io设备虚拟化，怎么配置内存等。



kvm不仅可以做虚拟化，还可以调kernel里面的其他东西，比如内存分配的功能。

kvm要提供一套接口暴露给用户态，文件接口（everything is file），在/dev/kvm这个特殊的文件里。要和它交互就要用文件接口，open，close，ioctl等文件接口。



创建VM，创建VCPU。

qemu创建VCPU要做什么，就是创建kvm里面的一个struct——context

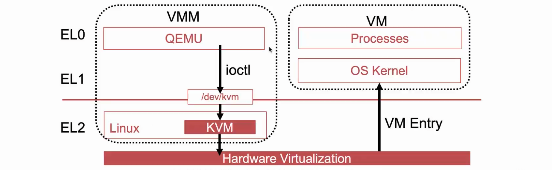
open dev把这个fd作为参数传给ioctl（这里没画出来）

最终环境都设置好之后调kvm\_run，在内核里面调真正的vm entry。

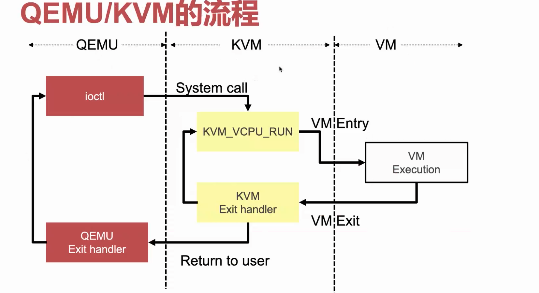
### KVM\_RUN时发生了什么



当虚拟机下陷的时候会发生什么呢？下陷到kvm，kvm里把状态保存好，然后交给用户态的qemu去做相应的事情。ioctl是个block的syscall。执行完kvm run说明vm已经跑完了，遇到了一个vm exit，它不会处理，应该交给qemu处理，根据exit reason qemu做相应的处理，比如做IO啊之类的。



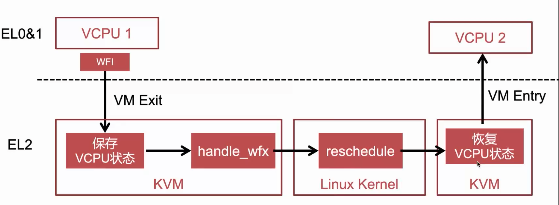
qemu跑在el0，调了ioctl，进入kvm，kvm借助硬件虚拟化机制进入虚拟机等等，然后再回到qemu。



qemu调ioctl进入内核态kvm，kvm里面有一套vm exit handler因为有一些exit是kvm能处理的，比如调度（时间片到了切换，这个kvm可以自己处理不用给qemu知道），遇到不能处理的exit reason，就交给用户态的qemu去处理。qemu怎么处理呢？比如虚拟机要发网络包，就交给qemu，qemu看到要下发网络包，qemu作为一个用户态进程怎么发网络包，它可以调用丰富的linux syscall，比如建一个socket。这些都是借助于linux内核的。

### case：WFI

当虚拟机运行过程中触发一个wfi，主动释放VCPU时间片，就会下线到kvm，kvm保存状态之后进入内部的handler——handle\_wfx，然后进入linux调度器，linux会调度到其他用户态进程或者其他vm等等，假设是vm，就恢复另一个VCPU的状态然后恢复到另一个虚拟机里面去跑。

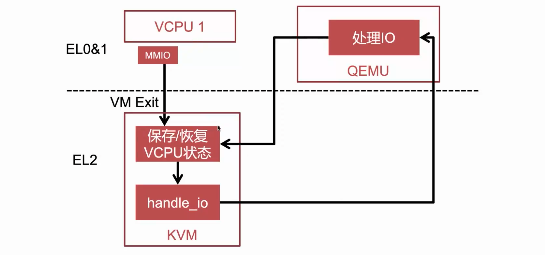


再说一下linux调度的时候需要知道自己要调度的是什么吗？不需要知道，它认为这些都是用户态线程。

### case：IO指令

MMIO

kvm保存当前VCPU状态，发现处理不了，然后转发到qemu，qemu处理完之后调ioctl恢复VCPU。

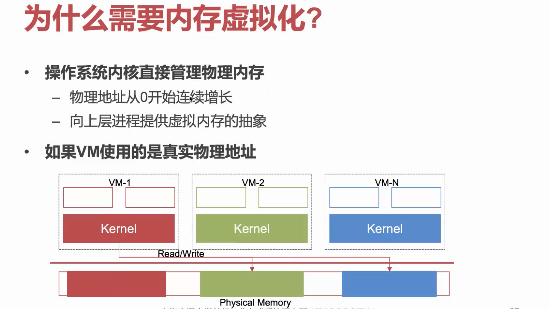


这就是qemu和kvm怎么配合去做的。

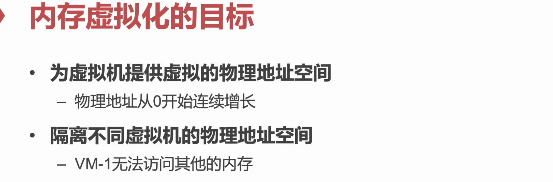
下面开始讲内存虚拟化和IO虚拟化

cpu虚拟化的时候我们刻意忽略了memory怎么办。比如上例中mmio下陷是因为kvm有意的配置了一个页表，这个页表访问这个页的时候是一个缺页。

## 内存虚拟化



不一定是连续的，有洞的（设备），所有的OS无论是在虚拟机还是裸机上跑都觉得管理所有物理内存。假设每个VM都让管所有内存，假设kernel能看到真实的物理地址，那它可以读别人的物理内存了，这显然是有问题的。



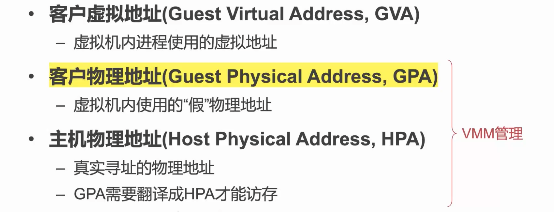
正确性（被改坏了）和安全性（vm不想被其他vm修改）

1. 完整的物理地址空间。（不应该有些不能访问）

2. 不能让它读到其他vm的信息。

怎么做呢？

### 客户物理地址

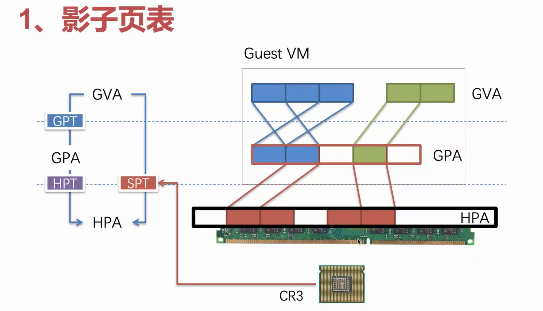


过去的虚存叫做客户虚拟地址，真实用来的寻址的Pa叫做主机物理地址。

GVA和GPA虚拟机内部能看到，自己建立页表。

这里排列组合其实应该有4种情况，还有一种HVA：VMM用的虚拟地址。





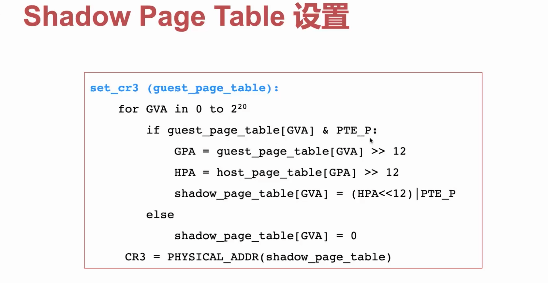
先讲第一个影子页表：两个页表做映射。

GVA到GPA： guest内部页表做映射

GPA到HPA：VMM应该把假的物理地址翻译成真的物理地址

shadow page table是一个很老的技术了。

x86问题：只有一个页表基地址寄存器。硬件只能提供一个页表，硬件不行，软件来凑。这就是影子页表一个思路。现在只有一个页表，这个页表应该翻译的是虚拟机内部的GVA到HPA，为什么不用管GPA呢，因为虚拟机跑起来的时候大部分时候用到的都是虚拟地址。我只要做这一件事，但做这件事不是很容易的，当guest配置完页表之后，它要装页表了，它要写CR3，trap住这个写CR3的操作，然后进行检查：比如先去检查是不是有效的，如果是有效的，右移12位拿到GPA，然后VMM看自己的HPT（page table），然后把这个对应的HPA翻译成对应的GVA，这个页表叫shadow page table。我们把这个present bit置上。然后把这个页表装在CPU上唯一这个基地址寄存器里。



Q1：shadow page table保存在guest还是VMM里面？

A：肯定是VMM里吧，guest能不能读到这个shadow里面的映射呢？不能，为什么叫影子页表，因为guest看不到它，也读不到。如果guest想要修改页表，就会修改自己的guest page table，只修改guest page table没有用，生效的只是CR3，并不是GPT。但它修改影子页表应该让VMM知道它做了修改，然后它这个修改才会生效。怎么让Guest修改GPT的时候让VMM知道呢？

trap and emulate即可。

怎么trap呢？guest配置内存就是访存，它能不能把所有store指令trap住，虚拟机跑不起来了，store太多了。它只要trap对GPT的修改，GPT就是内存吗，只要把GPT对应的内存页的修改trap住，变成read only即可。一句话：在影子页表里把GPT所在的内存页设置为read only。然后改的时候就会到VMM，VMM看一下你想怎么改，然后把这个修改同步到影子页表里。

建立Guest page table的时候是往CR3写trap，如果读CR3要不要下陷，如果不下陷会不会有问题，会读到影子页表，不行，所以也要trap，然后VMM给它返回它自己的GPT的地址。

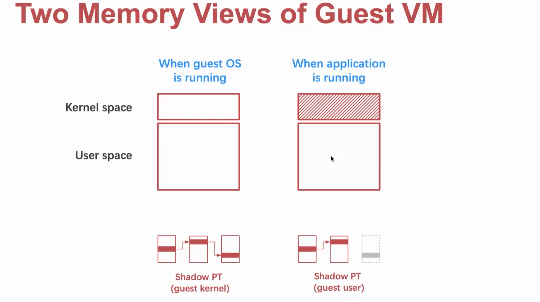
最后一个问题：VMM给它配影子页表的时候要把这些映射变成用户态可以访问的映射。因为kernel跑在用户态，但是uthread也跑在用户态，只有一个页表既要用户态进程访问，也要kernel访问，用户态和内核态隔离没有了。

U bit：用户态内核态都能访问

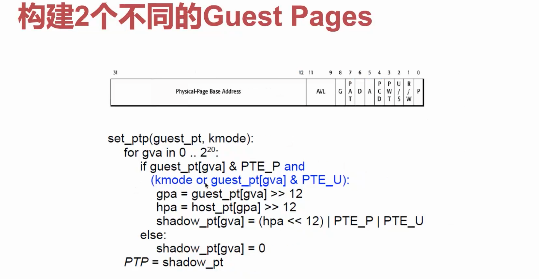
K bit：PTE对应的页只能内核态访问

虚拟机里面应该保证kernel和用户态进程的隔离：怎么解决？

产生两个影子页表，一个专门为内核服务，一个用户态服务。



把kernel对应的映射挖掉。



如果是内核态就把所有映射加上，加的映射GVA是不是ubit的，如果是才在page table里加映射。什么时候决定用哪个页表呢？

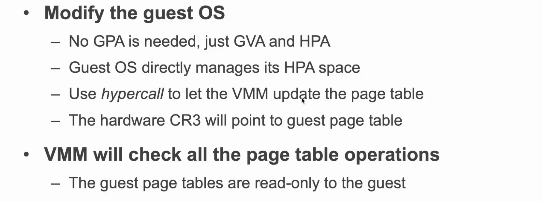
虚拟机内核态切到用户态过程中VMM帮它切换。（因为kernel和用户态切换都要经过VMM，所以VMM可以帮它换）

**shadow page table is awesome软件设计技术，用软件弥补硬件**

为什么我们还要讲呢？加深大家对页表机制的理解。

最后我们讲一下direct paging

### direct paging



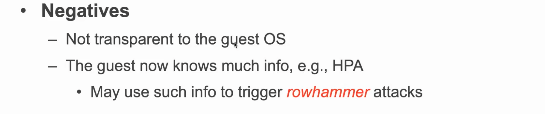
guest OS和VMM协同设计，让你知道你在虚拟机里，配置页表调hyper call就行了。

配置的时候也要检查一下配置的映射是不是有问题。

好处就是容易实现，性能很好。影子页表每一次guest改页表就要下陷，这里direct pageing不用每次改页表都修改，可以batch。

缺点：需要改guest kernel，要对不同的内核做修改，闭源的win就不行。

guest能看到HPA就比较危险，可以做rowhammer攻击。



疯狂写一个物理地址会导致它相邻的地址也作了翻转，如果让VM知道它要访问的物理地址是什么是很危险的。