[4-28 系统虚拟化 1](#_Toc105266860)

[review 1](#_Toc105266861)

[理论上虚拟化的模型 2](#_Toc105266862)

[系统虚拟化技术 2](#_Toc105266863)

[Intel VT-x 5](#_Toc105266864)

[VT-x执行过程 7](#_Toc105266865)

[Arm虚拟化技术 7](#_Toc105266866)

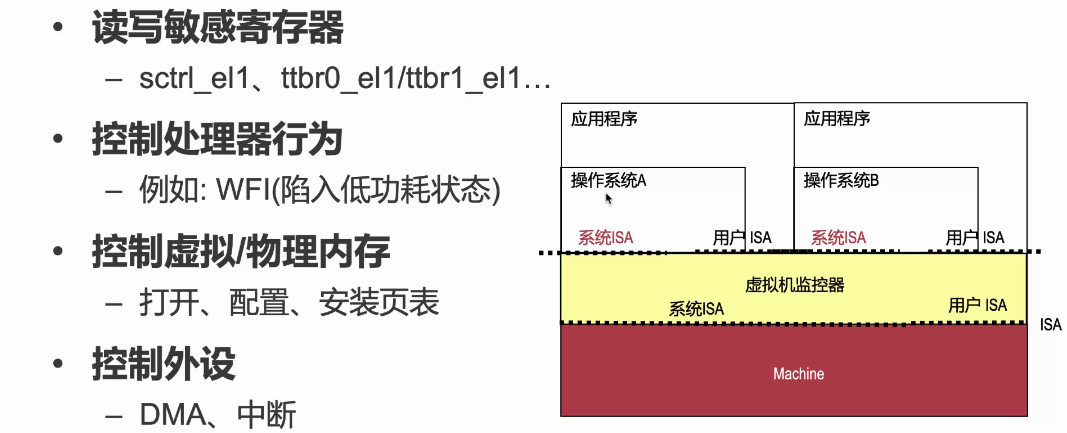
# 4-28 系统虚拟化

**虚控=虚拟机监控器**

上一节课夏老师已经开了个头，系统虚拟化能给我们理解操作系统提供一个很好的视角。进程和OS kernel中间这一层的接口是OS定义的。而如果从系统虚拟化的角度思考，并不是所有的接口都是可以设计出来的，OS是在虚拟化之前就有的。

## review

系统ISA：



ttbr0和ttbr1是和页表相关的寄存器，很显然不能让用户态进程去修改。

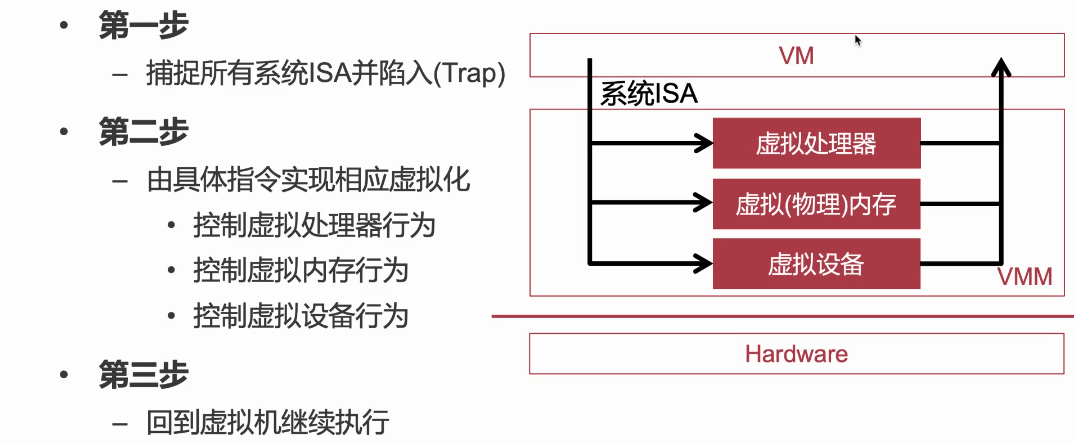
wfi：wait for interrupt

当一个软件在一个核上没有事情做的时候与其while loop空转不如进入低功耗状态，来一个中断硬件再把我唤醒。如果用户态调了这样一个指令，系统就进入低功耗了，恢复回来还是要不少时间的，不应该让虚拟机做这个事情。虚拟机的OS调了WFI这个指令应该如何操作呢？

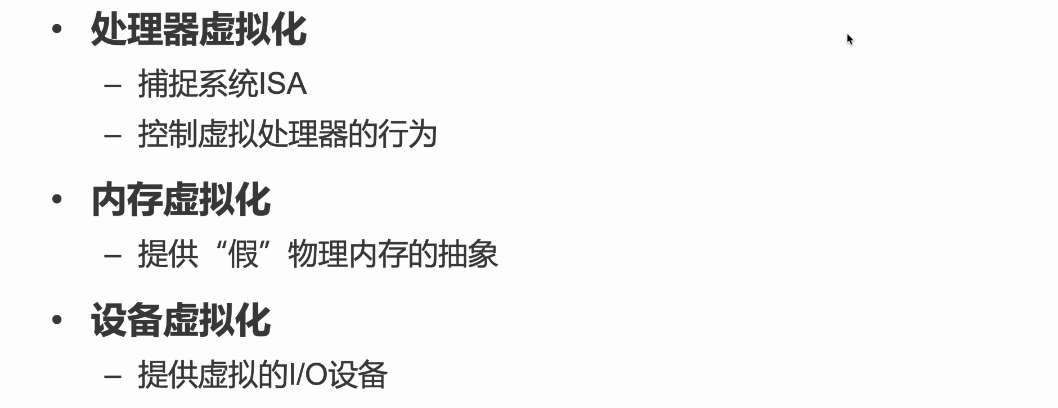
1. 不让它做WFI，你没事情做了不代表host上没事情做了。虚拟机监控器拿到这个指令之后，把时间片交给其他虚拟机。
2. 页表。虚拟机并不是唯一机器上跑的OS了，所以不能直接把物理内存管理权限交给它。这种系统ISA也应该我们给它拿到。之后让虚拟机监控器去操作。
3. 用户态驱动想要控制外设：1. DMA（异步）2.中断——为什么列为系统ISA呢？虚拟机操作磁盘甚至可以读到其他虚拟机的磁盘。场景：虚拟机格式化磁盘，不应该把真正的物理磁盘给格了。其实提供的并不是磁盘，而是FS中的一个文件。

操作系统用用户的ISA不重要，hypervisor让其继续执行。

### 理论上虚拟化的模型



### 系统虚拟化技术





如果这时候发生一个syscall了，这个syscall理论上是有对应的虚拟机中的kernel处理的。

1. 某个进程调svc

2. VMM抓到，把这个syscall转到Kernel里

3. kernel进到exception handle，eret的地址应该由VMM配置好

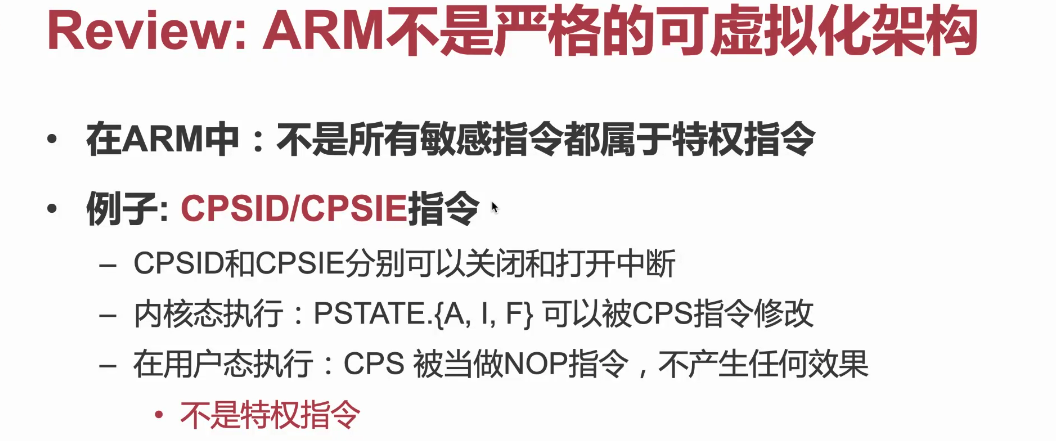
4. kernel处理过程中可能会执行系统ISA，会不断发生虚拟机下陷

5. eret也是系统ISA，依然会下陷。VMM会看到它想回到user thread，把它当做一个IPC转回到之前的进程。

IPC在进程间切换很重要页表做切换。

Q：这样一个架构在arm上能实现吗？

A：不能。上节课讲过，arm不是完全可虚拟化架构。不是所有敏感指令都会造成下陷。



> kernel可以关掉中断的，如果它想进入一个critical section。

如果不下陷，VMM不知道。就无法解决。上节课提到四个方法。



x86和arm都不是完全可虚拟化的架构。

这些都是有历史原因的：arm最初目标场景并不是高精尖场景。主要是IoT设备。虚拟化是在计算能力有富余的情况下才考虑的。

x86是面向PC的，PC对虚拟化要求不高。

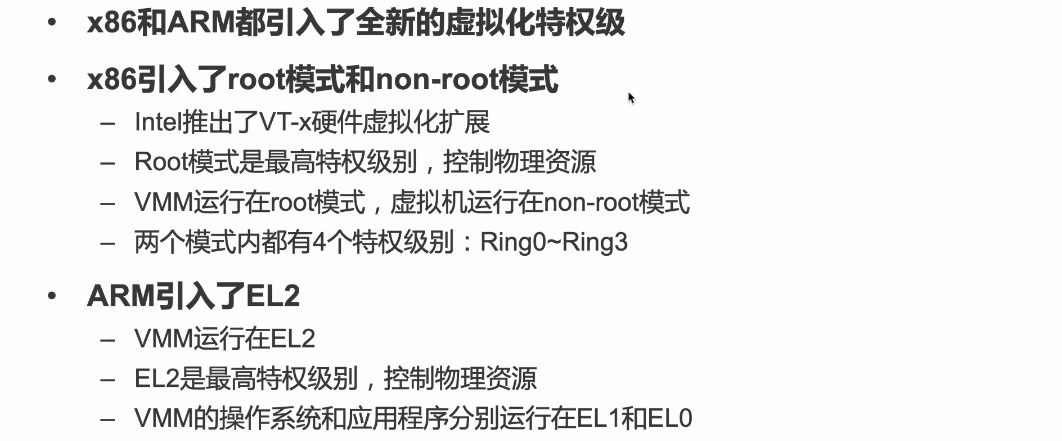
RISC-V在最开始的时候就把ISA设计成完全可虚拟化的。所有特权指令都会下陷。MIT已经基于RISC-V做了这样一个系统。

解释执行：把指令从内存里读过来，decode，再模拟等等——性能会非常差，一条硬件指令变成多条指令。

二进制翻译：vmware前身发在SOSP上的。不是所有指令都一条条执行，用户ISA可以直接执行，只有我关注的，系统ISA我才trap下来。通过动态改写换成另外一条指令。用cache的方法加速翻译过程。OS有很强的局部性的。这就是二进制翻译加速翻译速度很好的一个技术。

第三种方法：系统研究的时候很多假设是可以打破的。最开始虚拟化定义：不知道自己跑在虚拟化环境中。——这真的需要遵守吗？Xen之前就有半虚拟化了，03年Cambridge就搞了这个半虚拟化。虚拟机监控器和OS kernel协同设计。把换页表啊之类的换成syscall。

第四个就是今天着重介绍的：改硬件。



目前大部分用的都是硬件虚拟化。

ARM的设计和X86完全不同。

## Intel VT-x



ring-3用户态——arm el0

ring-0 kernel模式——arm el1

有了硬件虚拟化之后多了一个正交的模式：在root里也能看到四个ring，non-root里面也能看到4个ring，所以一共有8个ring。

如上图：虚拟机kernel跑在non-root的ring 0。虚拟机监控器跑在root ring 0。

不是所有的系统ISA都该下陷：syscall hypervisor应该知道吗？应该直接到虚拟机内核里，不要下陷了。

page fault，除0错啊之类的还要下陷吗？也不用下陷。hypervisor也解决不了这个问题。

在虚拟机监控器看来虚拟机也是进程。

Q：还有个问题，虚拟跑的时候，里面的OS有自己的抽象，比如进程线程，这些东西虚拟机监控器能看到吗？这些东西也要调度的，虚控要参与调度吗？

A：这个其实是不应该的。如果这么实现的话会有什么问题。OS要创建以一个新的进程，fork，假设虚控能看到，会有什么问题？如果虚拟机进程有点问题，比如while loop fork。Q：但是呢如果不这么做如何实现安全调度呢？

A：进程依然OS管的。创建之后进程信息加到guest OS信息里面去。但是真正的调度是虚控搞的——解决方案：参与调度但是看不到VM里面的进程，增加一个新的抽象：VCPU。

所以说虚拟机监控器提供这样一个抽象（VCPU是什么呢？其实内部就是一个个线程。）

在虚拟机启动的时候，虚控告诉VM你有几个VCPU。它就会搞四个队列，每个队列维护线程，执行的时候这四个VCPU都是虚控在启动的时候用4个线程来维护的。VM拿到这个线程的时间片之后就会调度自己的线程。两层这样一个结构。

4个VCPU因为是线程，所以不是同时执行的，这会造成一个很有意思的问题：double scheduling。

传统OS：spin lock想进critical section，另外一个核也想进，拿锁拿不到，就等，因为spin lock比较短。

但是一旦要虚拟化里面：因为不是所有VCPU都是同时执行的。所以critical section就会执行到一般被切出去，另外一个在等这个锁，这一等就可长了。

这就是比较经典的引入虚拟化抽象之后带来的新的问题。

我们还是再讲一些细节。

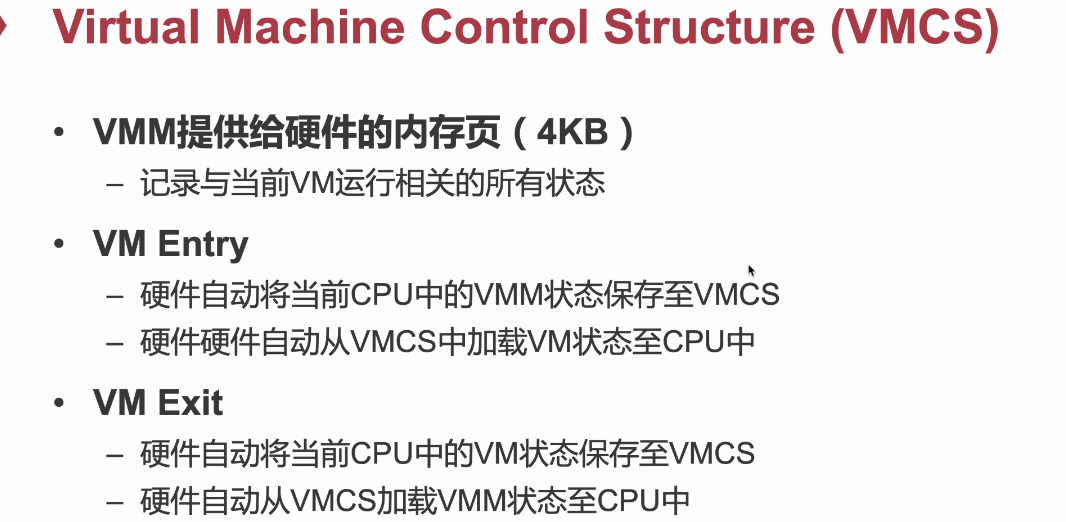
OS切换线程的时候会把通用寄存器从栈里pop出来，对于系统寄存器还能跟之前一样把这些系统寄存器从栈里面pop出来吗？

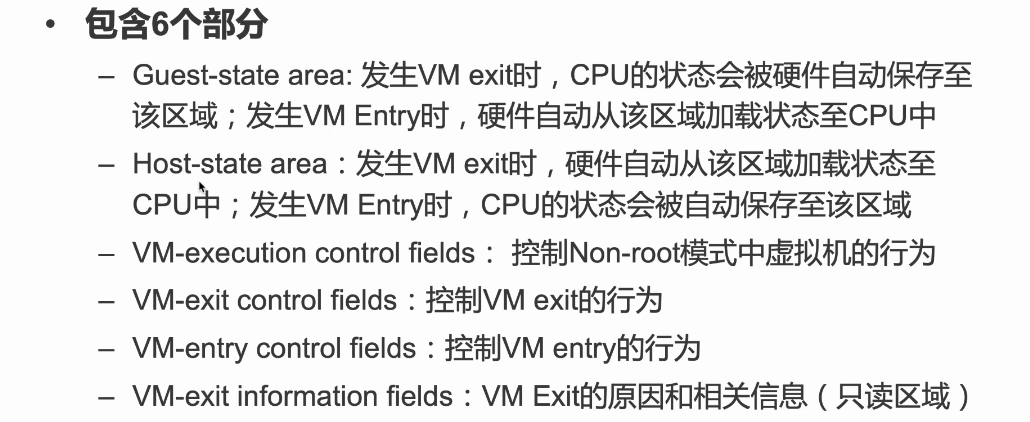
状态恢复上面有什么区别？

其实这个问题是比较tricky的，普通的寄存器是没有问题的，那页表会不会有问题呢？一个核上只能看到一个页表，一部分映射内核空间，一部分映射用户空间。但是引入虚拟化之后，虚拟机OS内核里面也有页表，但是虚控也是一个内核，这就不得不去换这个页表。进虚控用虚控页表，进VM kernel用它的页表有问题吗？这么做是可以做的。真正做起来有问题，在这个切换点这个位置，切换之后要还要能够执行就意味着从栈pop出来的页表里面也有代码页，这个页在虚控页表和vm kernel里面都要映射。要非常小心的context switch。guest能看到自己的页表里居然有不是自己的页就知道自己跑在虚拟化里面了。这就不太好。

纯靠软件做比较难。

Intel提出了VMCS





Host就是虚控，为什么要在里面保存虚控的状态信息呢？因为机器中系统寄存器只有一份，如果在VM entry执行之前cpu之前所有的寄存器都是host的，执行之后全是vm了。这些host的状态就要记录到这块。这一串硬件所做的事情有点像之前的context switch。只不过之前context switch是完全软件做的，intel现在提供了硬件的context switch能力。这是CSIC体系结构的特点：一条指令能做好多事情。所以vm entry和vm exit的开销是比较大的。单次下来单次回去开销很大。所以很多人做避免虚拟机下陷。现在intel不断优化，开销不断减少。

Q： VMCS保存虚拟机上下文状态的特殊的内存也。如果它有4个VCPU，它需要多少个VMCS呢？

A：4个。

还有四个state跟vm行为相关：guest在虚拟跑的时候为了性能应该直接kernel处理掉syscall。这个下陷是可以通过VM-execution-control-fields控制的。

甚至可以精确的配置某个exception是否下陷。

控制vm entry行为：中断是不让虚拟机管理中断的，在虚控看来中断是虚拟中断，不是硬件触发的。因为guest其实看不到device的，如果这个设备是模拟出来的话，所以让虚控插中断。

VM-exit information fields：vm entry是可以通过指令控制的，vm exit是事件驱动的并不是能够精准知道的，最后一个field就记录了最近的vm exit是什么原因。

### VT-x执行过程

（略不全见PPT 14）

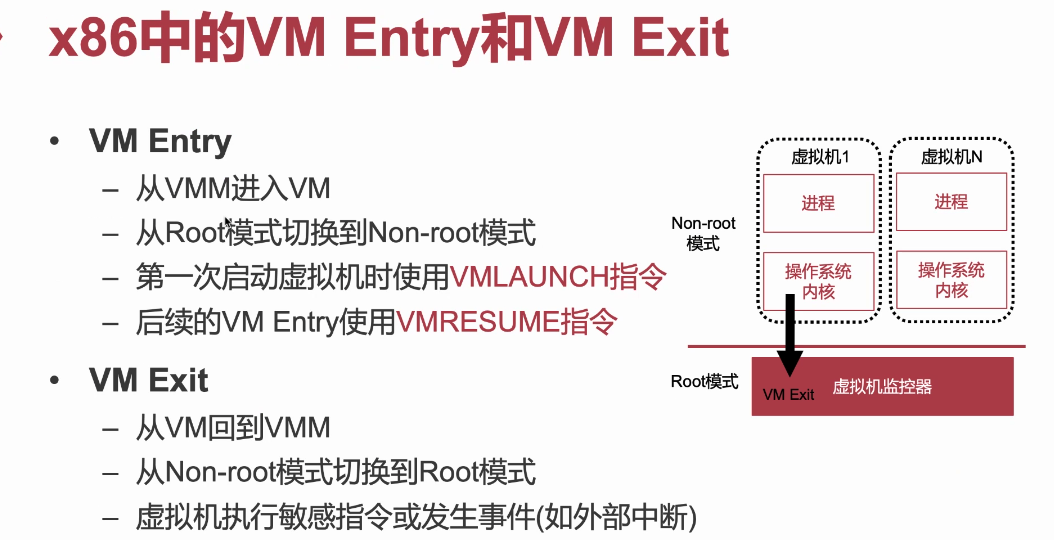
root ring0调VMXON

第一次跑VM的时候调vm launch，跑这个指令之前应该把VMCS准备好

虚拟运行的过程中会vm exit

执行完之后调vmresume

每个vcpu对应相应的VMCS



以上就是intel VT-x，x86的虚拟化机制

## Arm虚拟化技术

x86为业界做了一个很好的策略，但是arm另辟蹊径搞了另外一套硬件虚拟化。



多了一个特权级EL2，专为虚拟机监控器跑的。

这个设计就跟x86不太一样了。更本质的区别：寄存器命名方式完全不同。x86寄存器都通用的。系统寄存器也是同一套。cr0哪个模式下都是一样的，某一个瞬间只能有一个cr0。

arm不同，每个寄存器名字都有一个后缀，elx。就是这些不同，虚控给arm做硬件虚拟化支持的时候遇到了很多障碍。

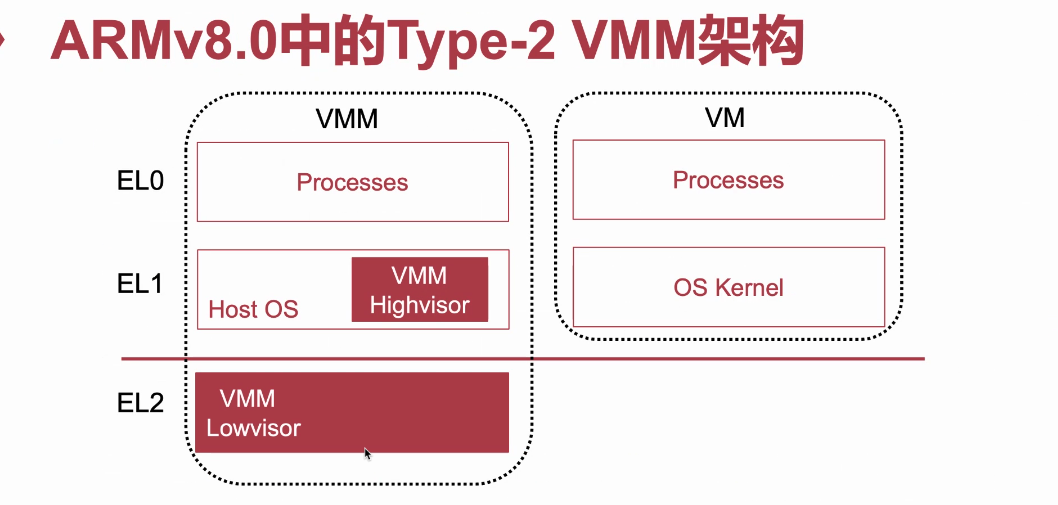
页表格式发生变化了，一开始el2里只有一个ttbr。更奇怪的这个页表和el1里的格式不太一样。

所以之前为arm设计的OS不能直接变成虚控。

虚控有两种类型：type1和type2

如果type1可以，改就改吧。

但是type2，想要把linux变成一个虚控就很难做了。

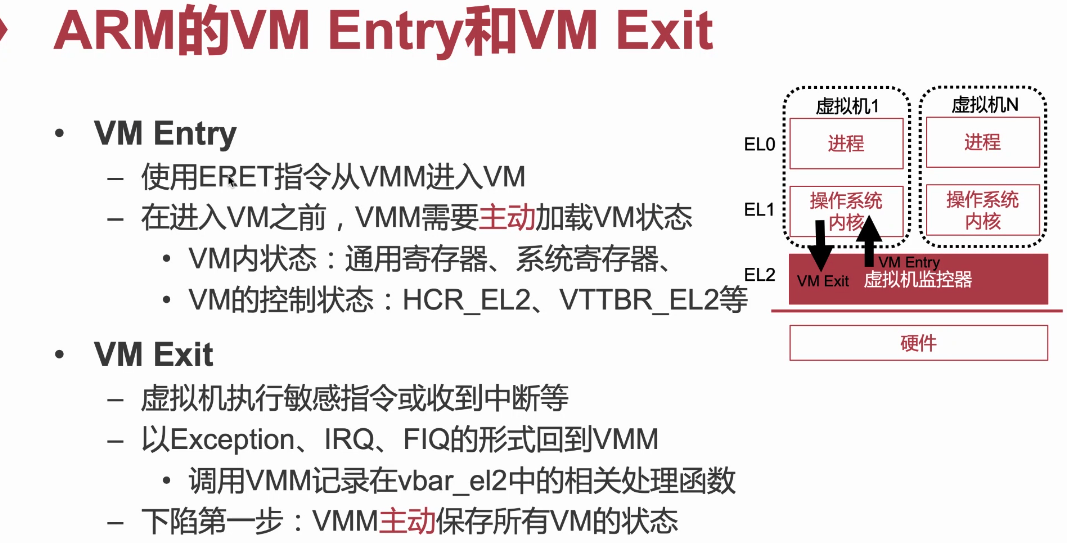


linux本来可以直接加一个模块就可以变成一个虚控。

但是arm就不行。linux社区为了适配专门搞了一个low visor和high visor。分成两部分了。

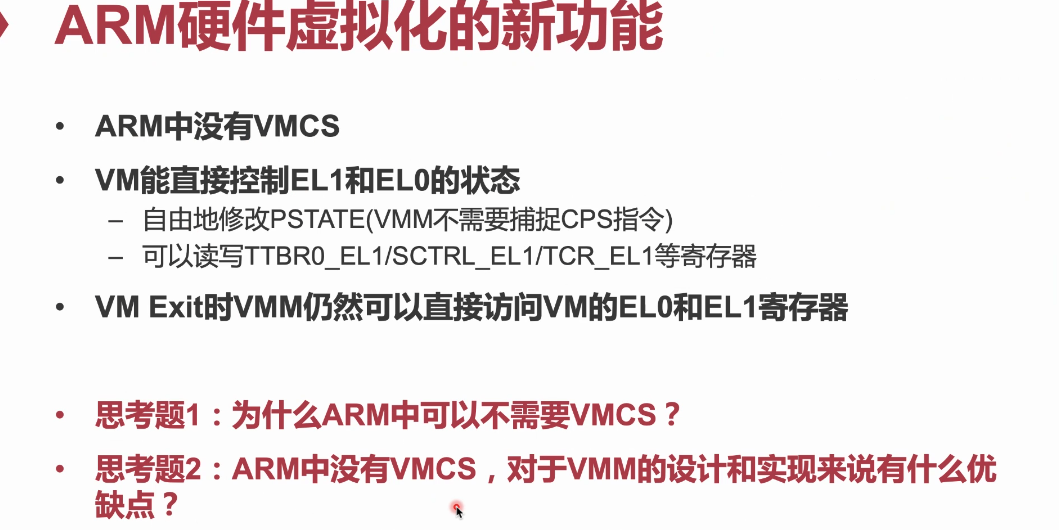
它会带来一些性能上的开销。

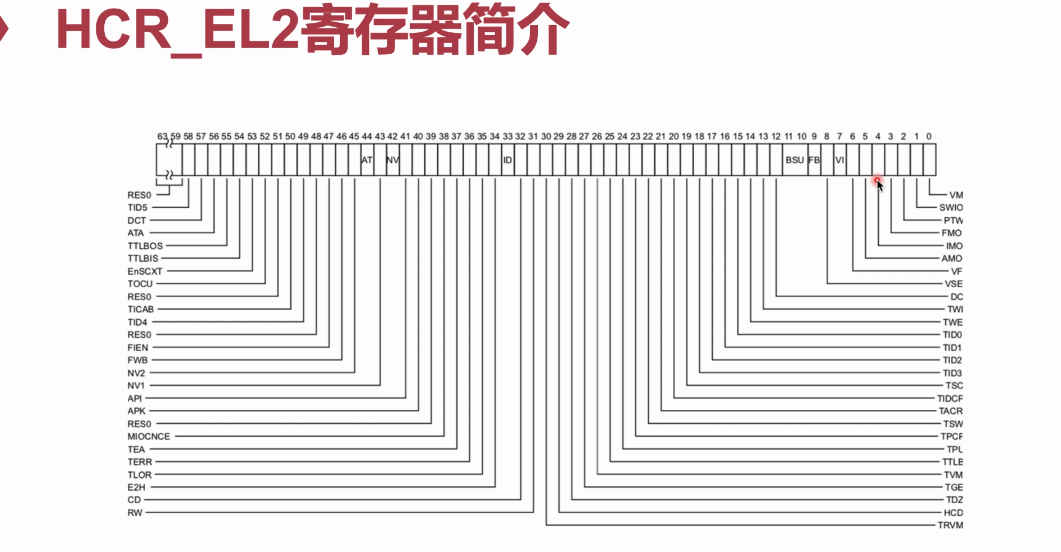
vm entry在arm里也不太一样。



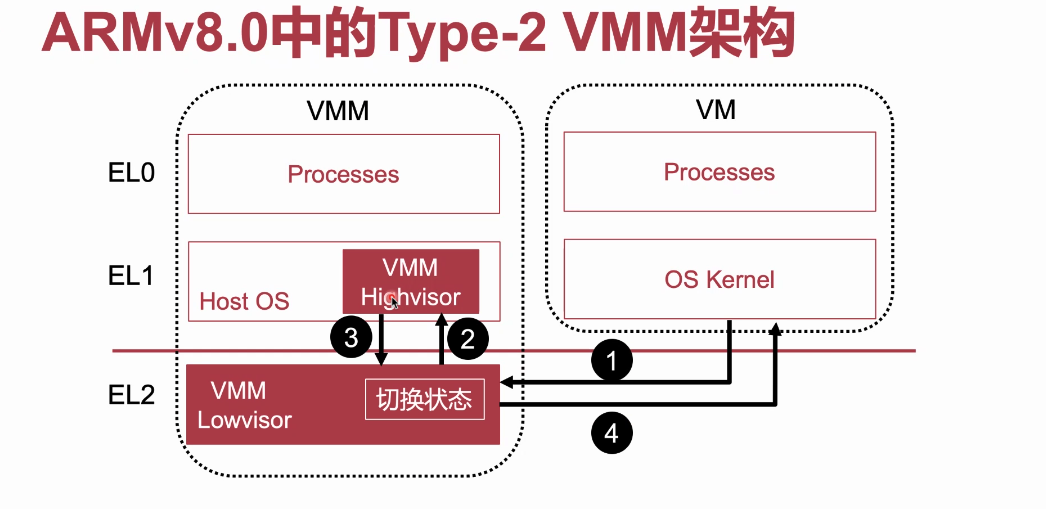
都是eret，arm里没有vmcs。为什么不用vmcs？

arm寄存器命名的特殊性，有好多套寄存器……硬件自动的帮我在eret的瞬间换页表。arm所有的寄存器都是冗余的。所以不需要vmcs。所以不需要保存恢复。





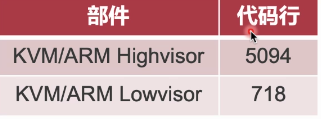
对于type2的虚控不得不分成两部分

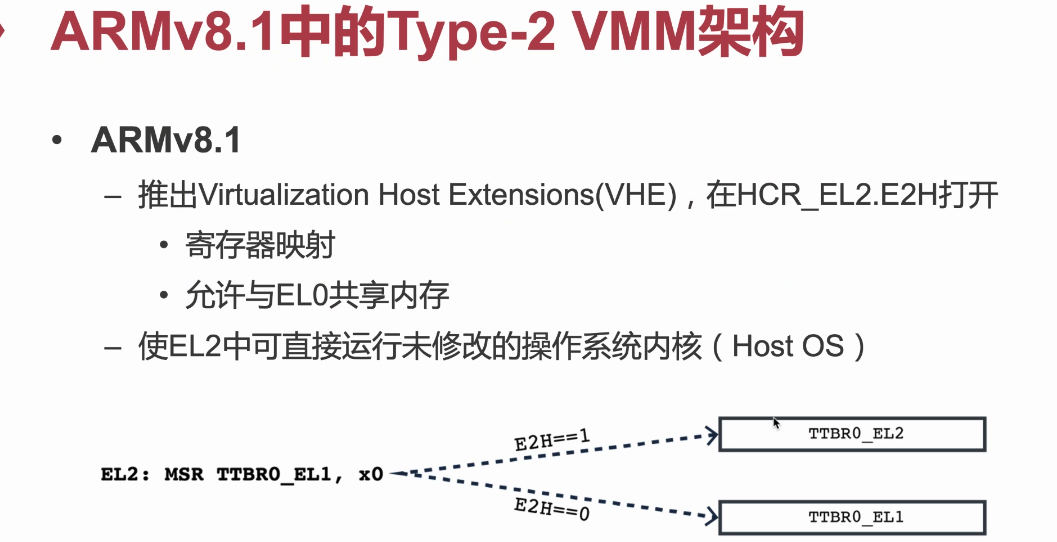


lowvisor其实没法处理vm exit它会交给vmm highvisor去处理。

它会保存vm os kernel的el1的状态（因为host kernel也跑在el1）

这就意味着这个设计本来是增加虚拟机的性能的，但是type2没法享受到这个好处，所以不得不采取分拆的模式，但是一旦分拆了就没法享受冗余的寄存器。vm exit开销并不比x86里的vm exit小，甚至更大。一开始linux用arm虚拟化的时候性能非常不好。

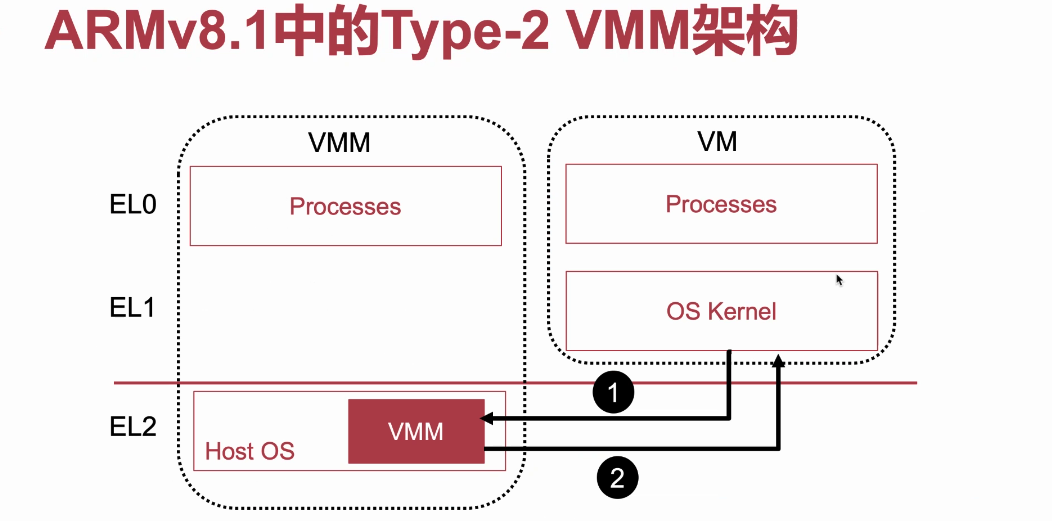




v8.1之后提出新的硬件拓展——VHE专门为type2 hypervisor做的适配

寄存器映射：硬件会自动将你的ttbr0\_el1映射到ttbr0\_el2里去。

然后呢她还允许页表的格式也做了一些修改结果就是：el2可以直接跑未经修改的操作系统内核了。linux可以整个跑在el2。



有了VHE开销大大减少——VHE也是云上一个主流的硬件拓展

故事结束了吗？并没有。

8.1之后这个HCR\_EL2.E2H bit打开了VHE才会生效。为什么要这样呢？在手机里不想用，在手机里还是8.0的设计。为什么这样？分成两部分——安全。linux漏洞太多了。

protective KVM——专注于安全保护。甚至可以假设host os不可信。检查host os配的页表之类的东西。