# VCPU调度

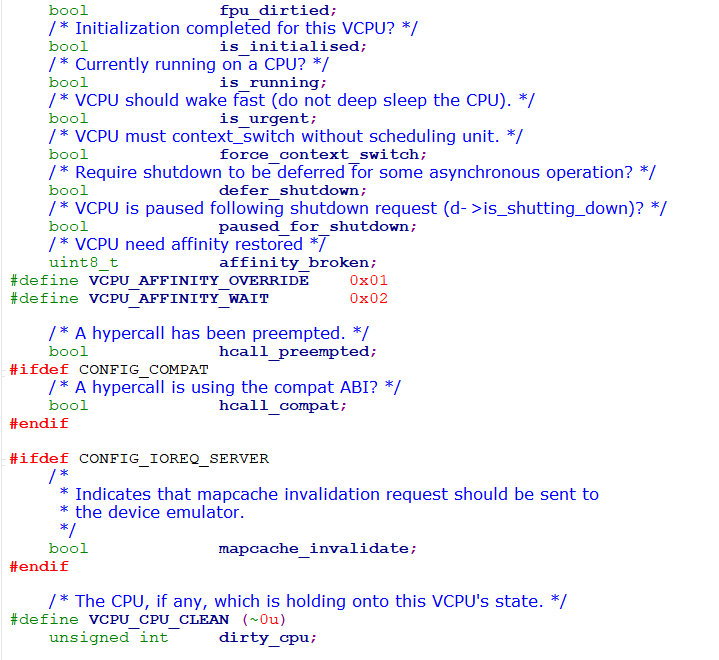
调度的实现要从以下几方面考虑：调度器、调度算法、中断、软中断、时钟（包括虚拟时钟）。

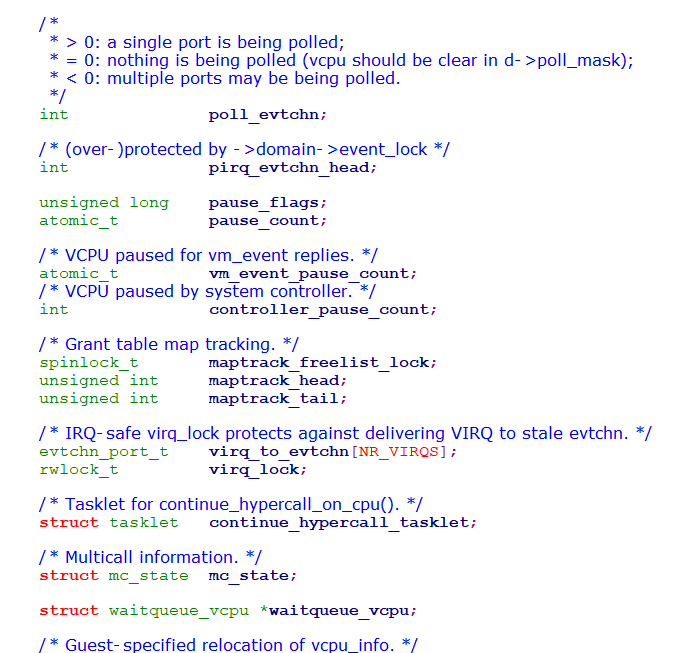
## 两级调度

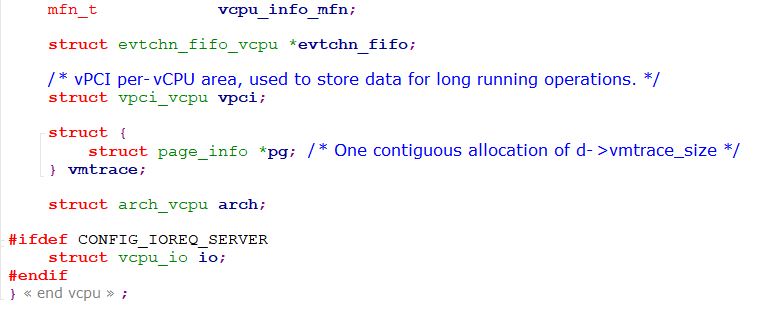
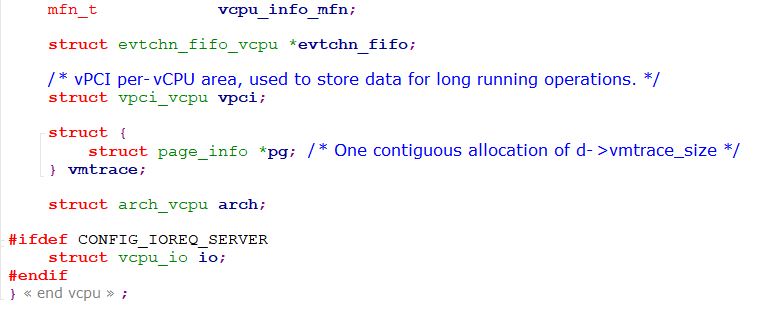
在 Xen 平台上，Guest OS 的调度器和 Xen 自身的调度器共同构成了着两级调度，这里称 Guest OS 对任务的调度为第二级调度，称 Hypervisor 对 VCPU 的调度为第一级调度。两级调度模型抽象了 Guest OS 内部任务在 CPU 上的执行行为，这非常适合用于分析运行于 Xen 上的 RTOS 是否仍然满足实时任务的实时需求。

参与调度的主体主要包括客户 OS 中运 行的任务 Task、客户 OS 的调度器、VCPU 以及 Xen 的调度器。在两级调度框架 下，Guest OS 按照自身的调度算法选取任务使之在 VCPU 上执行，然后 Xen 按照 VCPU 调度算法选取VCPU 使之能在 CPU 上执行，需要注意的是，VCPU 中需要 被装载到 CPU 上的数据不仅仅是任务执行的上下文，还涉及到许多 CPU 控制寄存器的值。Xen 调度的基本单位是 VCPU，所有域的 VCPU 一起竞争有限的 CPU 资源，因此 VCPU 调度的目的是，采用一种算法，从众多就绪的 VCPU 中，挑选出最合 适的 VCPU 并使之占有 CPU 资源，并达到一种所需的效果。例如，为了使所有 VCPU 公平地占有 CPU 资源，可以使用时间片轮转算法来对 VCPU 进行调度。 VCPU的结构如下：









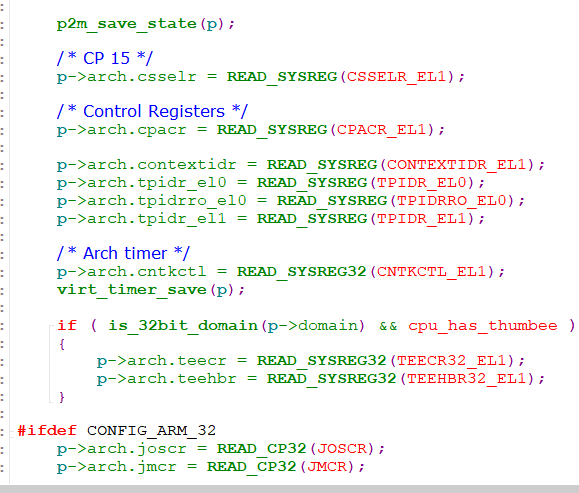
与硬件相关的CPU上下文保存在arch\_vcpu结构体中。从下文中可以看到与进程上下文相比，VCPU的上下文还需要保存mmu、Control Registers、HYP configuration等硬件状态。这些都需要在VCPU切换时恢复到目标CPU上。



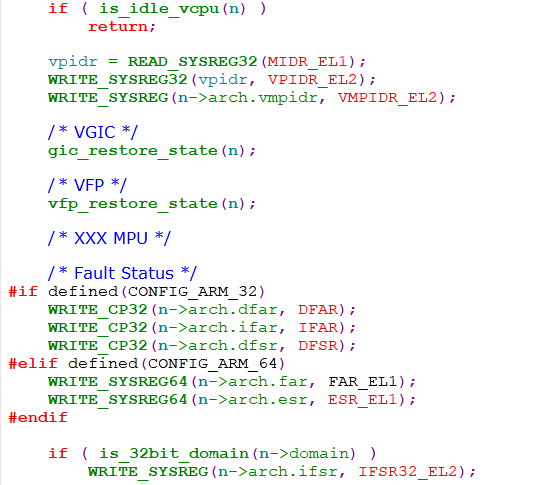


VCPU切换的上下文函数由sched\_context\_switch完成。函数首先停下prev VCPU的timer，启动next VCPU的timer，最终调用\_context\_switch和schedule\_tail。\_\_context\_switch依靠arm64的架构实现，在entry.S中实现，他同进程上下文切换相同，保存和恢复的是cpu\_context。

schedule\_tail函数去完成上下文切换的收尾工作，用于执行任务切换后的一些清理工作和新任务的初始化。这其中就包括进程上下文意外的CPU硬件上下文。如下图CPU硬件上下文的保存：

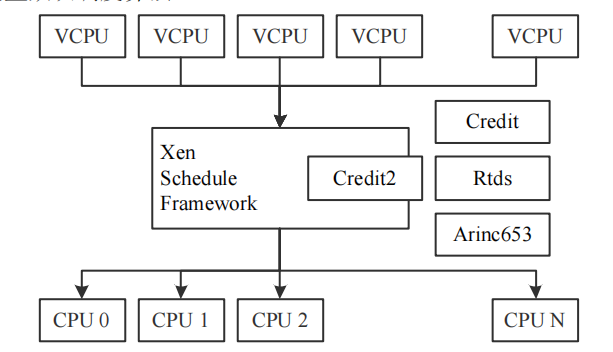


CPU硬件上下文的恢复：



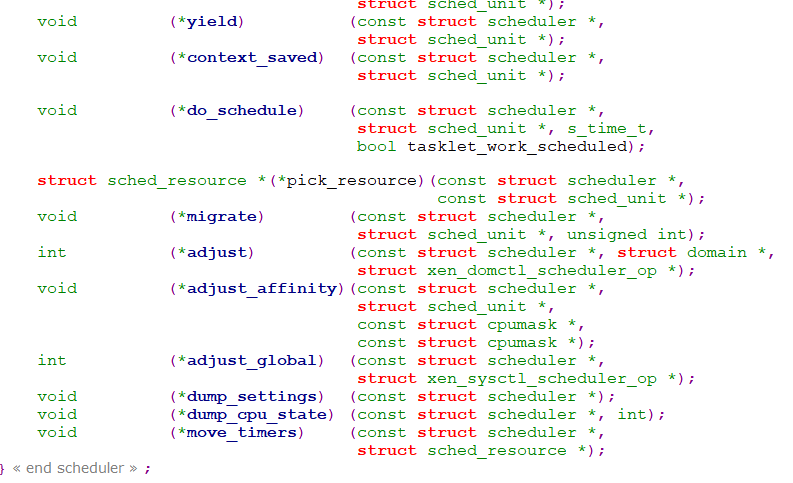
## **调度框架**

Xen 拥有良好的调度框架，方便开发者定制自己的调度算法，调度算法以模块的形式存在，图中列出了 Xen 支持的 4 种调度算法，可以在 Xen 的启 动参数中设置默认调度算法。



Xen 的调度框架包括两组 API，除了使用一组 API 向 Xen 的其它模块提供调 度服务外，它还提供一个调度器对象，使调度框架和调度算法的实现分离，降低 了调度器的实现难度，只需实现调度器对象中的几个关键函数，即可实现一个 Xen 调度器。 在 Xen 调度器对象中，比较重要的函数有 insert\_vcpu，do\_schedule，wake， context\_saved。函数 insert\_vcpu 决定了 VCPU 进入就绪队列的算法；do\_schedule 是核心调度算法，它按照一定的算法来挑选可执行的 VCPU；wake 函数决定了调度算法是否可抢占，对于可抢占算法，当一个 VCPU 被唤醒时，会立即执行抢占算法；context\_saved 函数决定了被换出的 VCPU 是否仍有机会抢占其它正在运行的 VCPU。 Xen 调度由调度软中断触发，而调度软中断则通常由软时钟来触发，当然，在 需要重新调度的时候可以通过触发调度软中断使 CPU 进行重新调度。Xen 将调度 软中断和调度时钟实现在 Xen 调度框架中，使调度器设计者不再需要考虑这些问题。下面是 Xen 调度器对象的定义：





在 Xen 的调度框架中，参与调度的核心数据结构有：域 Domain，VCPU，当 前 CPU 所用的调度器 scheduler，以及 CPU 的调度数据结构 sched\_unit。其中 sched\_unit 是所有调度器都必须依赖的，系统为每个 CPU 提供一个 sched\_unit 结构体，它拥有以下几个参数：调度锁，正在当前 CPU 上执行的 VCPU 指针，调度器私有数据 prv，调度时钟以及紧迫 VCPU 个数。根据调度算法实现的差异，可以实现一把全局调度锁，也可以为每个 CPU 提供一把调度锁。调度时钟s\_time\_t 是一个软件定时器，用于周期性触发调度。 ，Xen 引入了 sched\_unit 结构体来表示一个物理 CPU 的调度单元。每个 sched\_unit 对应一个物理 CPU 上的调度队列，并包含了该调度队列中所有 VCPU 的相关信息，如调度优先级、状态、调度参数等。因此，sched\_unit 和 VCPU 是一一对应的关系，一个 sched\_unit 中包含了一个或多个 VCPU，而一个 VCPU 也对应于一个 sched\_unit。vcpu是一个队列，通过vcpu中的next\_in\_list找到下一个VCPU。 struct domain 中的成员 struct vcpu \*\*vcpu 是一个指向指针的指针，它指向一个 struct vcpu 类型的指针数组，表示该域中的所有 vCPU。每个 vCPU 都可以通过索引访问。例如，要访问域中的第一个 vCPU，可以使用 dom->vcpu[0]。dom0在初始化时创建 dom->vcpu[0]，并依次创建max\_cpu个。并且创建vcpu链表，事vcpu[0]的next\_in\_list指向vcpu[1]，依次类推。在sched\_init\_vcpu将该vcpu加入调度队列。为这个vcpu选择一个物理cpu，并将物理processor绑定到v->processor中。创建一个sched\_unit与vcpu绑定，绑定实现是在（sched\_unit\_add\_vcpu函数中，使unit->vcpu\_list = v，unit->unit\_id = v->vcpu\_id），这属于调度绑定，所以在sched\_alloc\_unit实现。物理CPU的初始化，尤其是VCPU相应的资源在schedule\_cpu\_add中。struct sched\_resource 结构体是 Xen 调度器中的一个重要数据结构，用于跟踪虚拟 CPU 和物理 CPU 的映射关系，管理虚拟 CPU 在物理 CPU 上的运行时间和 CPU 时间片等调度相关的信息。每个物理 CPU 都有一个对应的 struct sched\_resource，表示该物理 CPU 的调度资源。struct sched\_resource 中包含了该物理 CPU 上运行的所有虚拟 CPU 的调度信息，如虚拟 CPU 的优先级、调度状态、剩余时间片等。同时，struct sched\_resource 还保存了物理 CPU 的一些信息，如物理 CPU 的负载、总运行时间等，这些信息用于计算虚拟 CPU 的权重和时间片。

## **Xen 的 Credit2 调度算法**

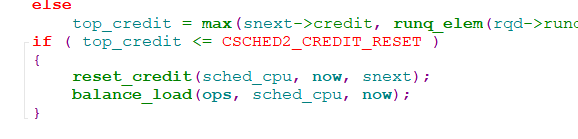
Xen 默认调度算法 Credit2 是Credit 调度算法的升级版，这两种调度算法的设计目的是使各个 VCPU 更为公平地占有 CPU 资源，保持较低的调度开销，拥有全局负载均衡功能，同时也为 Guest OS 提供较好的响应灵敏度。Xen Credit2 调度算法满足了预设目标，可调度性高于 KVM 虚拟机，I/O 响应灵敏度也高于其前身 Credit 调度算法，是一个优秀的 VCPU 调度器。

Credit2 调度算法为每个物理 CPU 维护一个调度队列，使各个 CPU 从自己的调度队列中挑选 VCPU，各个 CPU 的调度并行进行，不会出现多个 CPU 等待一个全局调度锁的情况，提高了 CPU 利用率。此外，Credit2 算法为每个 VCPU 提供一个 Credit 值，用于标识该 VCPU 至少应该运行的时间，VCPU 就绪队列也按照 Credit的多少进行降序排列，每次调度时，CPU 将上次执行的 VCPU 与队首 VCPU 的Credit 值进行对比，选择剩余 Credit 更多的 VCPU 来使用 CPU 资源，这使得各个VCPU 可以更为公平利用 CPU 资源，同时，由于调度时，上一个 VCPU 并不一定让出 CPU 资源，这也减少了上下文切换产生的调度开销。另外，引入 Credit 为限制特定域的 VCPU 对 CPU 的使用时间提供了方法。Credit2 算法中每个 CPU 拥有一个 IDLE CPU，IDLE VCPU 不加入任何就绪队列，仅仅在系统空闲时被调度运行。



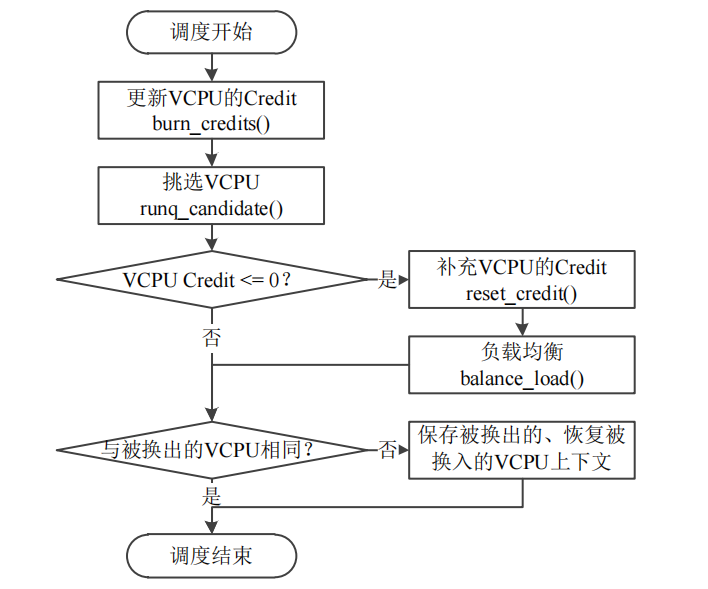
Credit2 调度算法提供了一个负载均衡机制来解决这个问题：在系统运 行过程中，Credit2 调度算法会动态地统计各个 CPU 上负载的大小，调度时，调度算法使用负载均衡算法对系统中各个负载不平衡的 VCPU 就绪队列进行调整。

runq\_insert函数会根据credit值将VCPU插入到运行队列中，他是在wake函数和context\_saved函数中调用。csched2\_unit\_insert函数只是简单的将vcpu插入到运行队列，并不会按照credit值排列，仅仅在vcpu初始化时建立队列调用。VCPU的权重（weight）越高，则其消耗 credit 的速率越慢。最高权重的 unit 每纳秒消耗一个 credit，其他 unit 则按比例消耗。当运行队列中的下一个 unit 的 credit 小于等于零时，所有 unit 的 credit 将被“剪切”到一个较小的值，并且所有 unit 都会获得一个固定的 credit。



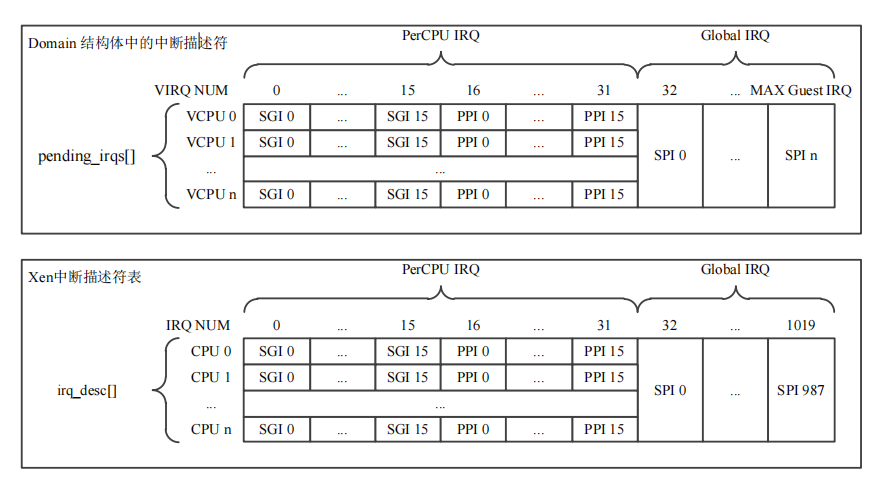
调度算法还为每个域设置了利用率上限。上限是针对域而不是单个单元（unit）的。如果一个域只有一个单元，并且上限为 40%，那么这个单元将使用 40% 的一个物理 CPU。如果一个域有 4 个单元，并且上限为 200%，相当于在 2 个物理 CPU 上的 100% 时间将分配给这些单元。为实现这个上限，使用了以下方法：为每个域设置一个budget，每个域都有一个计时器，定期补充该域的预算。预算是域内单元在每个“周期”内可以使用的时间量。

Credit2 的负载均衡算法可简单描述为：对于低负载的 CPU，该算法从高负载的 VCPU 队列里拉取 VCPU 并添加到自己的就绪队列。图 3-4 的虚线部分展示了这一 过程，CPU3 从负载最重的 CPU1 队列中拉取 VCPU 到自己的就绪队列并调度运行。 Credit2 调度算法的调度流程如上图所示。在该调度流程中，两个判断点作用值得一提，当选出的 VCPU 的 Credit 值小于等于零时，这说明当前 CPU 进入低 负载状态，需要执行负载均衡算法，为其它 CPU 分摊负载。而判断调度选出的 VCPU 是否与上一个 VCPU 相同则可以减少切换上下文带来的开销，此外 Credit2 调度算法也对 VCPU 迁移做了一定的限制，使得 VCPU 不能在物理 CPU 之间频繁迁移，因为频繁的 VCPU 切换会降低缓存命中率，降低整个系统的性能。



## VCPU上中断调度

Xen 负责管理平台上的所有中断，因此 Xen-ARM 的中断描述符表必须涵盖 GIC 支持的所有中断。由于 ARM 的中断包括 CPU 独占中断和共享中断，而 CPU 独占中断又由软中断（SGI）和私有外部中断（PPI）两部分组成。CPU 独占中断是每个CPU核心都必须接收的中断，例如ARM Generic Timer产生的时钟中断，核间中断等等；共享中断是平台硬件设备产生的中断，例如键盘输入中断等等。 为了处理上述两部分中断，Xen 的中断描述符也包括两个部分，它将共享中断是全局变量，而私有中断描述符被申明为 PerCPU 类型，该类型为每个 CPU 维护一份包含 32 个中断的中断描述符表，如下图的 Xen 中断描述符表。为了给 Guest OS 模拟 CPU 独占中断和共享中断，Xen 为每个域也设置了中断描述符表，它的结构和 Xen 的中断描述符类似，区别在于，域的共享中断放在 Domain 结构体的中断描述符表中，而 32 个 CPU 独占中断则放在VCPU 结构体中。注意，域的中断描述符表只是对域中中断的状态进行跟踪和描述，并不用于存放中断处理函数 ISR。下图中展示了 Xen-ARM 和 Domain 对中断描述符的组织形式。当中断由 Xen 直接处理时，Xen 直接从中断描述符中取出中断服务函数，进行中断处理。当中断由 Guest OS 处理时，该中断所对应的中断描述符将被标记为 \_IRQ\_GUEST，表示中断由 Guest OS 进行处理，对于这种中断，Xen 获取并更新对应域中断描述符的信息，然后将中断注入到相应的 Guest OS 中。



基于上述原因，在进行VCPU调度时，需要对VCPU中的中断信息进行维护（尽管guest不会直接使用）。vgic\_irq\_rank 的结构体，它代表一个 32 个中断的块对应的状态。rank->vcpu中保存着每个中断对应的VCPU id。在创建domain时，调用vgic\_rank\_init初始化中断和vcpu的对应关系。这里其实时保存一个中断路由的关系。每次VCPU调度时，判断调度单元中的migrated是否被设置为true。如果被置位，则通过调用arch\_move\_irqs实现。arch\_move\_irqs参数是下一个要被调度的VCPU vnext。函数将遍历要运行的vcpu的pending\_irq，将需要挂起的函数通过irq\_set\_affinity重新绑定要新的CPU上。

数据结构 struct vgic\_cpu，表示虚拟 CPU 的 VGIC 相关状态。其中有两个重要的中断列表。inflight\_irqs 存储已经成功注入到虚拟机中的中断，这些中断已经被虚拟机处理过并被从中断处理器中移除。lr\_pending 存储未能成功注入到虚拟机中的中断，因为此时 LR 寄存器已经用完，需要等待 LR 寄存器的空闲。lr\_pending 是 inflight\_irqs 的子集。每次从Hypervisor返回客户机时从vgic\_cpu数据结构中恢复到硬件LR中。

xen通过vgic\_inject\_irq函数将中断注入到虚拟机中。首先将取出该中断对应的vpu的vgic的pending\_irq。如果在客户机中这个中断是使能状态，则通过gic\_raise\_guest\_irq继续注入中断。在这里会判断当前运行的VCPU是否是要注入中断的VCPU。只有vcpu匹配以及vgic.lr\_pending（lr\_pending是pending\_irq的排队列表）为空时，才会进行实际的物理注入，否则只是将中断加入到lr\_pending的列表中去。待CPU从hypervisor返回到客户机时，才会调用gic\_restore\_pending\_irqs进行实际的注入。

## 虚拟时钟的实现

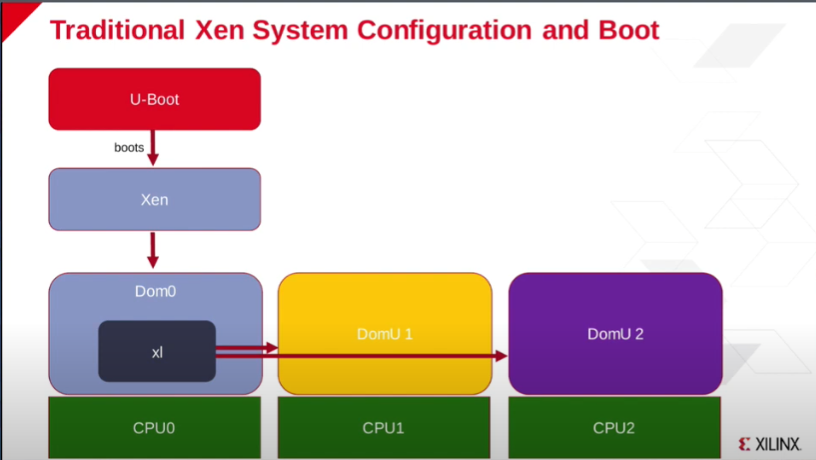
Guest OS 的运行离不开时钟的驱动，此外 Guest OS 也需要实时时钟来获取现 在的时间，为系统内部的软件提供时间。Xen 为 Guest OS 提供了时钟虚拟化支持， Xen 利用 ARM 的 Generic Timer 实现了时钟虚拟化。 ARM Generic Timer 是一个实时时钟，它包含一个可以计时至少四十多年的实 时时钟，以及一个 PL2 物理时钟、一个 PL1 物理时钟以及一个虚拟时钟，这 三个时钟本质上没有任何区别，都是物理上存在的，物理时钟和虚拟时钟的区别 在于它们的计数器之间存在一个可设置的偏移量，默认值为 0。为了方便描述，下文称 PL2 物理时钟为 HYP\_TIMER，称 PL1 物理时钟为 PHY\_TIMER，称虚拟时钟为 VIRT\_TIMER。Xen-ARM 使用 HYP\_TIMER 配合软件定时器为 Xen 的运行提供定时功能，例如 Xen 调度时钟。

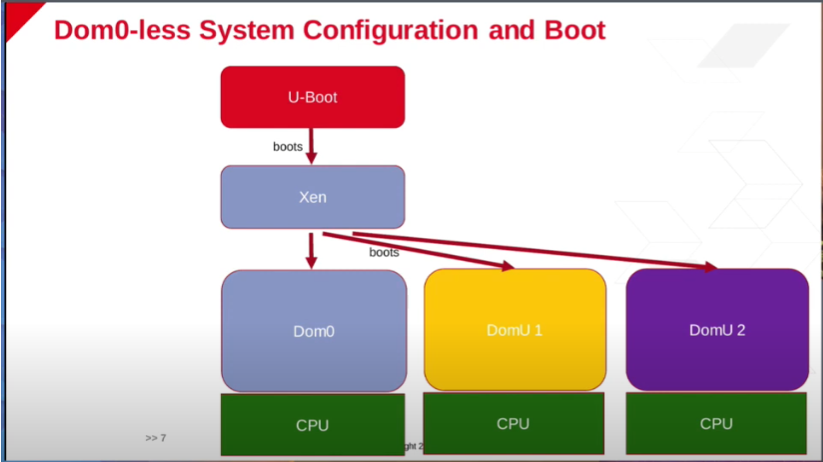
Xen为每个 VCPU 提供两个时钟 phys\_timer 和 virt\_timer，分别于为客户域模拟 PHY\_TIMER 和 VIRT\_TIMER。这两个虚拟时钟仅仅在对应 VCPU被调度运行时进行计数，这保证了 Guest OS 内的时钟只会记录 VCPU 在运行时经过的时间，使客户域内部时间和实际运行时间相一致。虚拟时钟的定义如下，其中 timer 结构体是软件定时器，ctl 和 cval 是虚拟时钟的寄存器。virt\_timer 则截然不同，它仅仅作为一个容器存在。由于 Xen-ARM 不捕捉读 写 VIRT\_TIMER 寄存器的指令，在 Guest OS 中读写 VIRT\_TIMER 并不会导致 CPU 陷入 Hypervisor。为了保证每个 VCPU 使用的 VIRT\_TIMER 与自身的 virt\_timer

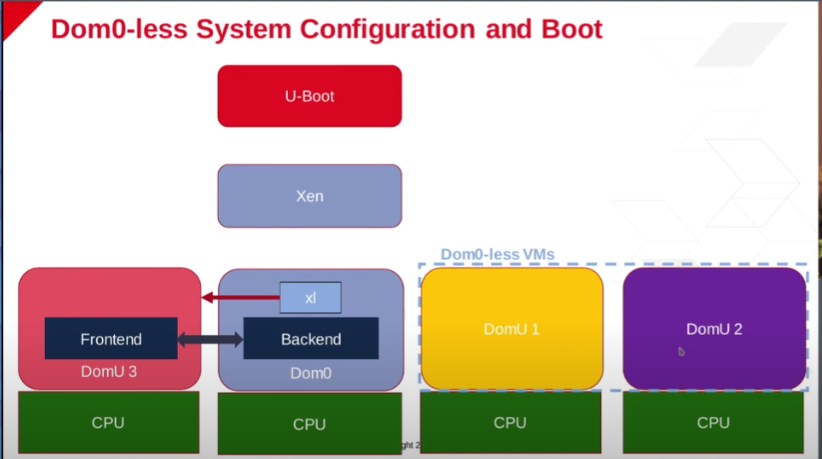
时间一致，当需要切换 VCPU 上下文时，Xen-ARM 将物理 VIRT\_TIMER 的寄存

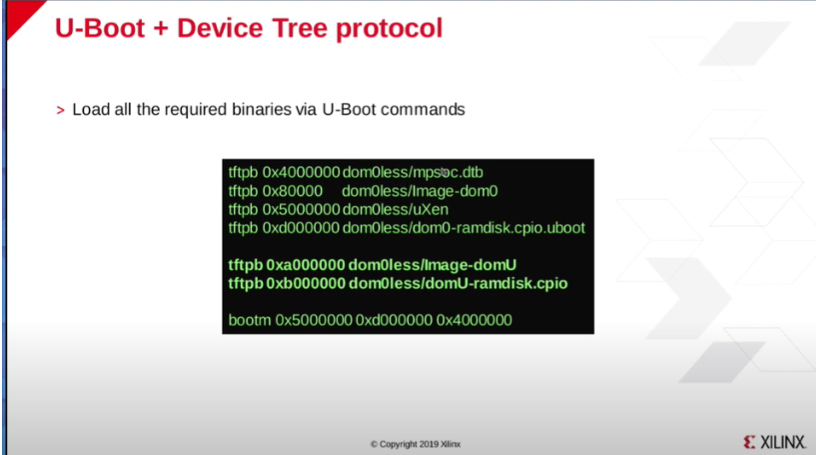
器保存到要换出的 VCPU 的 virt\_timer 中，并将要换入的 VCPU 的 virt\_timer 恢复 到 VIRT\_TIMER 上，如图 3-17 所示。VIRT\_TIMER 实现的高效性使它非常适合用于高频使用环境，例如为 Guest OS 提供调度滴答。

## dom0-less

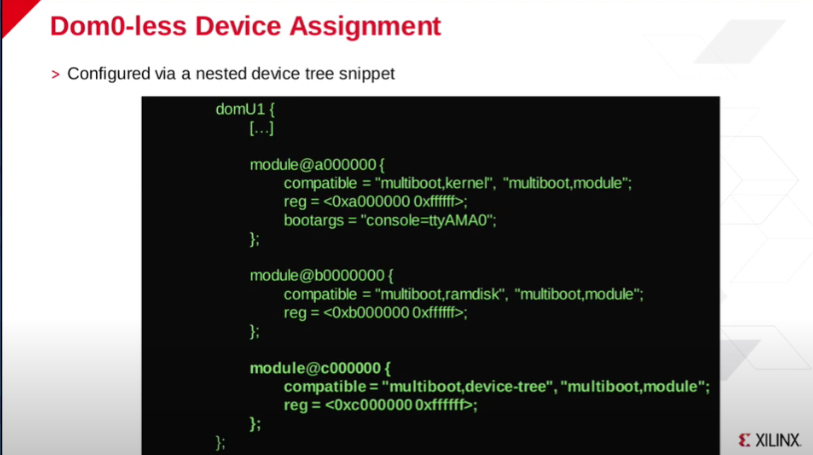


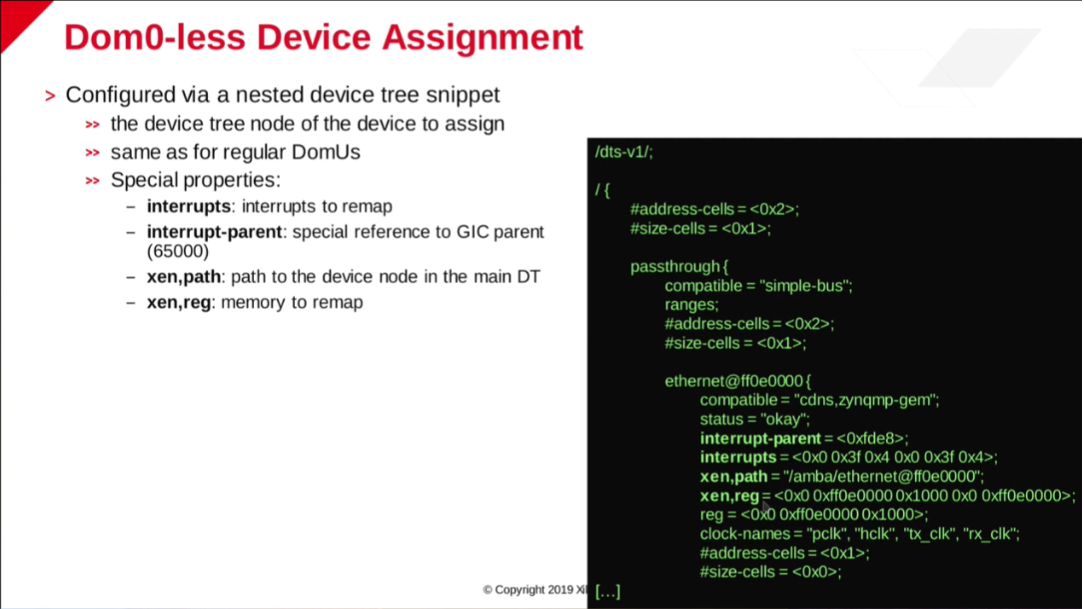


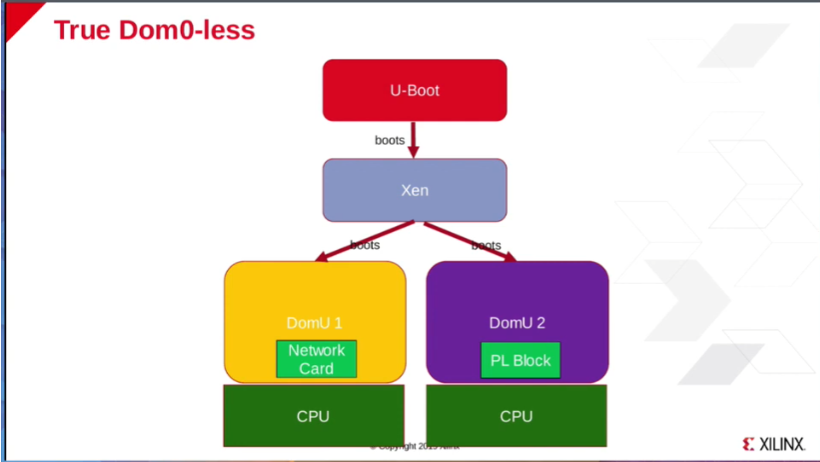


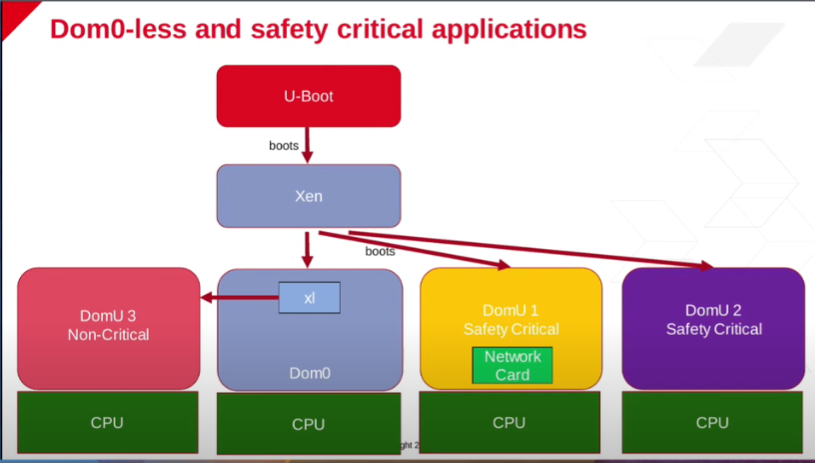


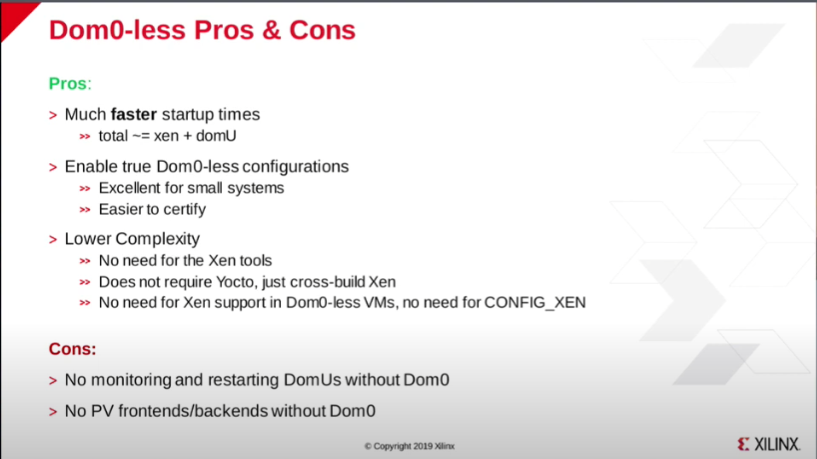


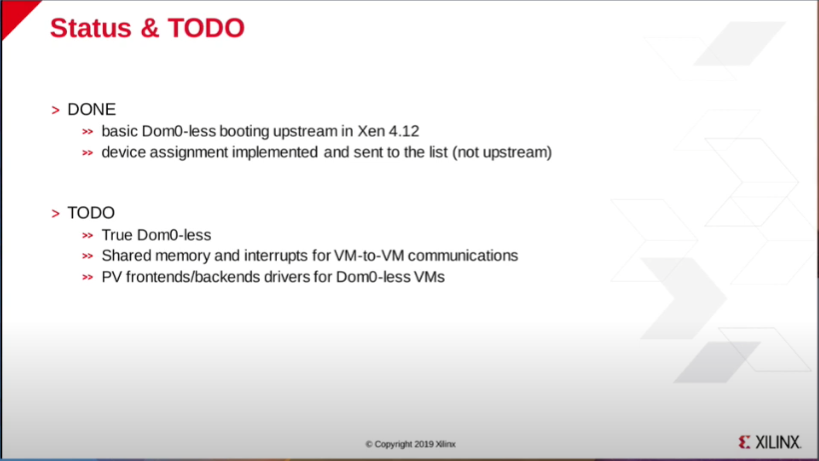












## type2型调度

但站在host的角度看，vcpu只是其创建的一个普通线程，因此它也需要被host os的调度器管理。当为vcpu线程分配的时间片用完以后，其需要让出物理cpu。在其再一次被调度时，其又需要切入guest运行。因此，在vcpu相关的线程切换时，会同时涉及以下两部分上下文：

（1）vcpu所在的普通线程上下文

（2）vcpu guest和host之间切换的上下文

xen基于flask的安全控制，可以换成麒麟的kysec。