每个程序员都应该了解的“虚拟内存”知识

[*编辑注：本文是Ulrich Drepper的“每个程序员应该了解的内存方面的知识”文章的第三部分；这一部分谈论了虚拟内存，特别是TLB性能。没有阅读第1部分和第2部分的人可能现在就想读一读了。和往常一样，请将排字错误报告之类发送到lwn@lwn.net，而不要发送到这里的评论。*]

4 虚拟内存

处理器的虚拟内存子系统为每个进程实现了虚拟地址空间。这让每个进程认为它在系统中是独立的。虚拟内存的优点列表别的地方描述的非常详细，所以这里就不重复了。本节集中在虚拟内存的实际的实现细节，和相关的成本。

虚拟地址空间是由CPU的内存管理单元(MMU)实现的。OS必须填充页表数据结构，但大多数CPU自己做了剩下的工作。这事实上是一个相当复杂的机制；最好的理解它的方法是引入数据结构来描述虚拟地址空间。

由MMU进行地址翻译的输入地址是虚拟地址。通常对它的值很少有限制 — 假设还有一点的话。虚拟地址在32位系统中是32位的数值，在64位系统中是64位的数值。在一些系统，例如x86和x86-64，使用的地址实际上包含了另一个层次的间接寻址：这些结构使用分段，这些分段只是简单的给每个逻辑地址加上位移。我们可以忽略这一部分的地址产生，它不重要，不是程序员非常关心的内存处理性能方面的东西。{*x86的分段限制是与性能相关的，但那是另一回事了*}

4.1 最简单的地址转换

有趣的地方在于由虚拟地址到物理地址的转换。MMU可以在逐页的基础上重新映射地址。就像地址缓存排列的时候，虚拟地址被分割为不同的部分。这些部分被用来做多个表的索引，而这些表是被用来创建最终物理地址用的。最简单的模型是只有一级表。

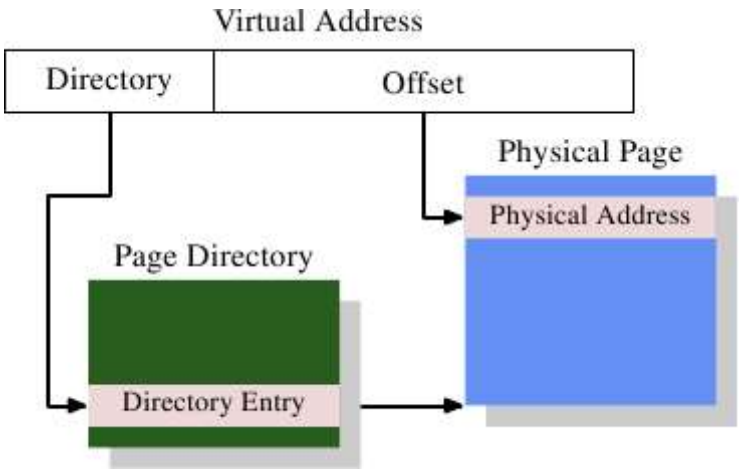

Figure 4.1: 1-Level Address Translation

图 4.1 显示了虚拟地址的不同部分是如何使用的。高字节部分是用来选择一个页目录的条目；那个目录中的每个地址可以被OS分别设置。页目录条目决定了物理内存页的地址；页面中可以有不止一个条目指向同样的物理地址。完整的内存物理地址是由页目录获得的页地址和虚拟地址低字节部分合并起来决定的。页目录条目还包含一些附加的页面信息，如访问权限。

页目录的数据结构存储在内存中。OS必须分配连续的物理内存，并将这个地址范围的基地址存入一个特殊的寄存器。然后虚拟地址的适当的位被用来作为页目录的索引，这个页目录事实上是目录条

目的列表。

作为一个具体的例子，这是 x86机器4MB分页设计。虚拟地址的位移部分是22位大小，足以定位一个4M页内的每一个字节。虚拟地址中剩下的10位指定页目录中1024个条目的一个。每个条目包括一个10位的4M页内的基地址，它与位移结合起来形成了一个完整的32位地址。

4.2 多级页表

4MB的页不是规范，它们会浪费很多内存，因为OS需要执行的许多操作需要内存页的队列。对于4kB的页（32位机器的规范，甚至通常是64位机器的规范），虚拟地址的位移部分只有12位大小。这留下了20位作为页目录的指针。具有2^{20}个条目的表是不实际的。即使每个条目只要4比特，这个表也要4MB大小。由于每个进程可能具有其唯一的页目录，因为这些页目录许多系统中物理内存被绑定起来。

解决办法是用多级页表。然后这些就能表示一个稀疏的大的页目录，目录中一些实际不用的区域不需要分配内存。因此这种表示更紧凑，使它可能为内存中的很多进程使用页表而并不太影响性能。.

今天最复杂的页表结构由四级构成。图4.2显示了这样一个实现的原理图。

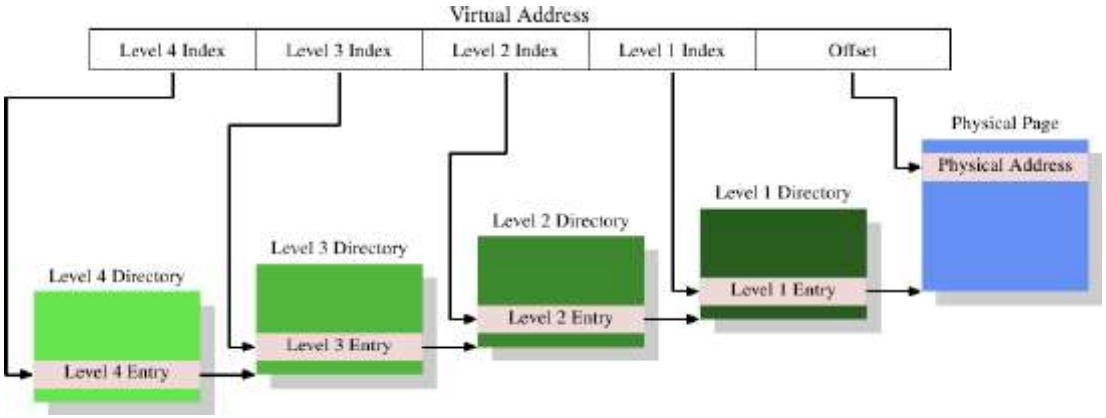

Figure 4.2: 4-Level Address Translation

在这个例子中，虚拟地址被至少分为五个部分。其中四个部分是不同的目录的索引。被引用的第4级目录使用CPU中一个特殊目的的寄存器。第4级到第2级目录的内容是对次低一级目录的引用。如果一个目录条目标识为空，显然就是不需要指向任何低一级的目录。这样页表树就能稀疏和紧凑。正如图4.1，第1级目录的条目是一部分物理地址，加上像访问权限的辅助数据。

为了决定相对于虚拟地址的物理地址，处理器先决定最高级目录的地址。这个地址一般保存在一个寄存器。然后CPU取出虚拟地址中相对于这个目录的索引部分，并用那个索引选择合适的条目。这个条目是下一级目录的地址，它由虚拟地址的下一部分索引。处理器继续直到它到达第1级目录，那里那个目录条目的值就是物理地址的高字节部分。物理地址在加上虚拟地址中的页面位移之后就完整了。这个过程称为页面树遍历。一些处理器（像x86和x86-64）在硬件中执行这个操作，其他的需要OS的协助。

系统中运行的每个进程可能需要自己的页表树。有部分共享树的可能，但是这相当例外。因此如果页表树需要的内存尽可能小的话将对性能与可扩展性有利。理想的情况是将使用的内存紧靠着放在虚拟地址空间；但实际使用的物理地址不影响。一个小程序可能只需要第2，3，4级的一个目录和少许第1级目录就能应付过去。在一个采用4kB页面和每个目录512条目的x86-64机器上，这允许用4级目录对2MB定位（每一级一个）。1GB连续的内存可以被第2到第4级的一个目录和第1级的512个目录定位。

但是，假设所有内存可以被连续分配是太简单了。由于复杂的原因，大多数情况下，一个进程的栈与堆的区域是被分配在地址空间中非常相反的两端。这样使得任一个区域可以根据需要尽可能的增

长。这意味着最有可能需要两个第2级目录和相应的更多的低一级的目录。

但即使这也不常常匹配现在的实际。由于安全的原因，一个可运行的（代码，数据，堆，栈，动态共享对象，aka共享库）不同的部分被映射到随机的地址[未选中的]。随机化延伸到不同部分的相对位置；那意味着一个进程使用的不同的内存范围，遍布于虚拟地址空间。通过对随机的地址位数采用一些限定，范围可以被限制，但在大多数情况下，这当然不会让一个进程只用一到两个第2和第3级目录运行。

如果性能真的远比安全重要，随机化可以被关闭。OS然后通常是在虚拟内存中至少连续的装载所有的动态共享对象(DSO)。

4.3 优化页表访问

页表的所有数据结构都保存在主存中；在那里OS建造和更新这些表。当一个进程创建或者一个页表变化，CPU将被通知。页表被用来解决每个虚拟地址到物理地址的转换，用上面描述的页表遍历方式。更多有关于此：至少每一级有一个目录被用于处理虚拟地址的过程。这需要至多四次内存访问（对一个运行中的进程的单次访问来说），这很慢。有可能像普通数据一样处理这些目录表条目，并将他们缓存在L1d,L2等等，但这仍然非常慢。

从虚拟内存的早期阶段开始，CPU的设计者采用了一种不同的优化。简单的计算显示，只有将目录表条目保存在L1d和更高级的缓存，才会导致可怕的性能问题。每个绝对地址的计算，都需要相对于页表深度的大量的L1d访问。这些访问不能并行，因为它们依赖于前面查询的结果。在一个四级页表的机器上，这种单线性将 至少至少需要12次循环。再加上L1d的非命中的可能性，结果是指令流水线没有什么能隐藏的。额外的L1d访问也消耗了珍贵的缓存带宽。

所以，替代于只是缓存目录表条目，物理页地址的完整的计算结果被缓存了。因为同样的原因，代码和数据缓存也工作起来，这样的地址计算结果的缓存是高效的。由于虚拟地址的页面位移部分在物理页地址的计算中不起任何作用，只有虚拟地址的剩余部分被用作缓存的标签。根据页面大小这意味着成百上千的指令或数据对象共享同一个标签，因此也共享同一个物理地址前缀。

保存计算数值的缓存叫做旁路转换缓存(TLB)。因为它必须非常的快，通常这是一个小的缓存。现代CPU像其它缓存一样，提供了多级TLB缓存；越高级的缓存越大越慢。小号的L1级TLB通常被用来做全相联映像缓存，采用LRU回收策略。最近这种缓存大小变大了，而且在处理器中变得集相联。其结果之一就是，当一个新的条目必须被添加的时候，可能不是最久的条目被回收于替换了。

正如上面提到的，用来访问TLB的标签是虚拟地址的一个部分。如果标签在缓存中有匹配，最终的物理地址将被计算出来，通过将来自虚拟地址的页面位移地址加到缓存值的方式。这是一个非常快的过程；也必须这样，因为每条使用绝对地址的指令都需要物理地址，还有在一些情况下，因为使用物理地址作为关键字的L2查找。如果TLB查询未命中，处理器就必须执行一次页表遍历；这可能代价非常大。

通过软件或硬件预取代码或数据，会在地址位于另一页面时，暗中预取TLB的条目。硬件预取不可能允许这样，因为硬件会初始化非法的页面表遍历。因此程序员不能依赖硬件预取机制来预取TLB条目。它必须使用预取指令明确的完成。就像数据和指令缓存，TLB可以表现为多个等级。正如数据缓存，TLB通常表现为两种形式：指令TLB(ITLB)和数据TLB(DTLB)。高级的TLB像L2TLB通常是统一的，就像其他的缓存情形一样。

4.3.1 使用TLB的注意事项

TLB是以处理器为核心的全局资源。所有运行于处理器的线程与进程使用同一个TLB。由于虚拟到物理地址的转换依赖于安装的是哪一种页表树，如果页表变化了，CPU不能盲目的重复使用缓存的条目。每个进程有一个不同的页表树（不算在同一个进程中的线程），内核与内存管理器VMM(管理

程序)也一样，如果存在的话。也有可能一个进程的地址空间布局发生变化。有两种解决这个问题的办法：

- 当页表树变化时TLB刷新。
- TLB条目的标签附加扩展并唯一标识其涉及的页表树

第一种情况，只要执行一个上下文切换TLB就被刷新。因为大多数OS中，从一个线程/进程到另一个的切换需要执行一些核心代码，TLB刷新被限制进入或离开核心地址空间。在虚拟化的系统中，当内核必须调用内存管理器VMM和返回的时候，这也会发生。如果内核和/或内存管理器没有使用虚拟地址，或者当进程或内核调用系统/内存管理器时，能重复使用同一个虚拟地址，TLB必须被刷新。当离开内核或内存管理器时，处理器继续执行一个不同的进程或内核。

刷新TLB高效但昂贵。例如，当执行一个系统调用，触及的内核代码可能仅限于几千条指令，或许少许新页面（或一个大的页面，像某些结构的Linux的就是这样）。这个工作将替换触及页面的所有TLB条目。对Intel带128ITLB和256DTLB条目的Core2架构，完全的刷新意味着多于100和200条目（分别的）将被不必要的刷新。当系统调用返回同一个进程，所有那些被刷新的TLB条目可能被再次用到，但它们没有了。内核或内存管理器常用的代码也一样。每条进入内核的条目上，TLB必须擦去再装，即使内核与内存管理器的页表通常不会改变。因此理论上说，TLB条目可以被保持一个很长时间。这也解释了为什么现在处理器中的TLB缓存都不大：程序很有可能不会执行时间长到装满所有这些条目。

当然事实逃脱不了CPU的结构。对缓存刷新优化的一个可能的方法是单独的使TLB条目失效。例如，如果内核代码与数据落于一个特定的地址范围，只有落入这个地址范围的页面必须被清除出TLB。这只需要比较标签，因此不是很昂贵。在部分地址空间改变的场合，例如对去除内存页的一次调用，这个方法也是有用的，

更好的解决方法是为TLB访问扩展标签。如果除了虚拟地址的一部分之外，一个唯一的对应每个页表树的标识（如一个进程的地址空间）被添加，TLB将根本不需要完全刷新。内核，内存管理程序，和独立的进程都可以有唯一的标识。这种场景唯一的问题在于，TLB标签可以获得的位数异常有限，但是地址空间的位数却不是。这意味着一些标识的再利用是有必要的。这种情况发生时TLB必须部分刷新（如果可能的话）。所有带有再利用标识的条目必须被刷新，但是希望这是一个非常小的集合。

当多个进程运行在系统中时，这种扩展的TLB标签具有一般优势。如果每个可运行进程对内存的使用（因此TLB条目的使用）做限制，进程最近使用的TLB条目,当其再次列入计划时，有很大机会仍然在TLB。但还有两个额外的优势：

1. 特殊的地址空间，像内核和内存管理器使用的那些，经常仅仅进入一小段时间；之后控制经常返回初始化此次调用的地址空间。没有标签，就有两次TLB刷新操作。有标签，调用地址空间缓存的转换地址将被保存，而且由于内核与内存管理器地址空间根本不会经常改变TLB条目，系统调用之前的地址转换等等可以仍然使用。
2. 当同一个进程的两个线程之间切换时，TLB刷新根本就不需要。虽然没有扩展TLB标签时，进入内核的条目会破坏第一个线程的TLB的条目。

有些处理器在一些时候实现了这些扩展标签。AMD给帕西菲卡（Pacifica）虚拟化扩展引入了一个1位的扩展标签。在虚拟化的上下文中，这个1位的地址空间ID（ASID）被用来从客户域区别出内存管理程序的地址空间。这使得OS能够避免在每次进入内存管理程序的时候（例如为了处理一个页面错误）刷新客户的TLB条目，或者当控制回到客户时刷新内存管理程序的TLB条目。这个架构未来会允许使用更多的位。其它主流处理器很可能会随之适应并支持这个功能。

4.3.2 影响TLB性能

有一些因素会影响TLB性能。第一个是页面的大小。显然页面越大，装进去的指令或数据对象就越多。所以较大的页面大小减少了所需的地址转换总次数，即需要更少的TLB缓存条目。大多数架构

允许使用多个不同的页面尺寸；一些尺寸可以并存使用。例如，x86/x86-64处理器有一个普通的4kB的页面尺寸，但它们也可以分别用4MB和2MB页面。IA-64 和 PowerPC允许如64kB的尺寸作为基本的页面尺寸。

然而，大页面尺寸的使用也随之带来了一些问题。用作大页面的内存范围必须是在物理内存中连续的。如果物理内存管理的单元大小升至虚拟内存页面的大小，浪费的内存数量将会增长。各种内存操作（如加载可执行文件）需要页面边界对齐。这意味着平均每次映射浪费了物理内存中页面大小的一半。这种浪费很容易累加；因此它给物理内存分配的合理单元大小划定了一个上限。

在x86-64结构中增加单元大小到2MB来适应大页面当然是不实际的。这是一个太大的尺寸。但这转而意味着每个大页面必须由许多小一些的页面组成。这些小页面必须在物理内存中连续。以4kB单元页面大小分配2MB连续的物理内存具有挑战性。它需要找到有512个连续页面的空闲区域。在系统运行一段时间并且物理内存开始碎片化以后，这可能极为困难（或者不可能）

因此在Linux中有必要在系统启动的时候，用特别的Huge TLBfs文件系统，预分配这些大页面。一个固定数目的物理页面被保留，以单独用作大的虚拟页面。这使可能不会经常用到的资源捆绑留下来。它也是一个有限的池；增大它一般意味着要重启系统。尽管如此，大页面是进入某些局面的方法，在这些局面中性能具有保险性，资源丰富，而且麻烦的安装不会成为大的妨碍。数据库服务器就是一个例子。

增大最小的虚拟页面大小（正如选择大页面的相反面）也有它的问题。内存映射操作（例如加载应用）必须确认这些页面大小。不可能有更小的映射。对大多数架构来说，一个可执行程序的各个部分位置有一个固定的关系。如果页面大小增加到超过了可执行程序或DSO(Dynamic Shared Object)创建时考虑的大小，加载操作将无法执行。脑海里记得这个限制很重要。图4.3显示了一个ELF二进制的对齐需求是如何决定的。它编码在ELF程序头部。

```
$ eu-readelf -l /bin/ls
Program Headers:
  Type    Offset   VirtAddr           PhysAddr           FileSiz  MemSiz   Flg Align
...
  LOAD    0x000000 0x0000000000400000 0x0000000000400000 0x0132ac 0x0132ac R E 0x200000
  LOAD    0x0132b0 0x00000000006132b0 0x00000000006132b0 0x001a71 0x001a71 RW  0x200000
...
```

Figure 4.3: ELF 程序头表明了对齐需求

在这个例子中，一个x86-64二进制，它的值为0x200000 = 2,097,152 = 2MB，符合处理器支持的最大页面尺寸。

使用较大内存尺寸有第二个影响：页表树的级数减少了。由于虚拟地址相对于页面位移的部分增加了，需要用来在页目录中使用的位，就没有剩下许多了。这意味着当一个TLB未命中时，需要做的工作数量减少了。

超出使用大页面大小，它有可能减少移动数据时需要同时使用的TLB条目数目，减少到数页。这与一些上面我们谈论的缓存使用的优化机制类似。只有现在对齐需求是巨大的。考虑到TLB条目数目如此小，这可能是一个重要的优化。

4.4 虚拟化的影响

OS映像的虚拟化将变得越来越流行；这意味着另一个层次的内存处理被加入了想象。进程（基本的隔间）或者OS容器的虚拟化，因为只涉及一个OS而没有落入此分类。类似Xen或KVM的技术使OS映像能够独立运行 — 有或者没有处理器的协助。这些情形下，有一个单独的软件直接控制物理内存的访问。

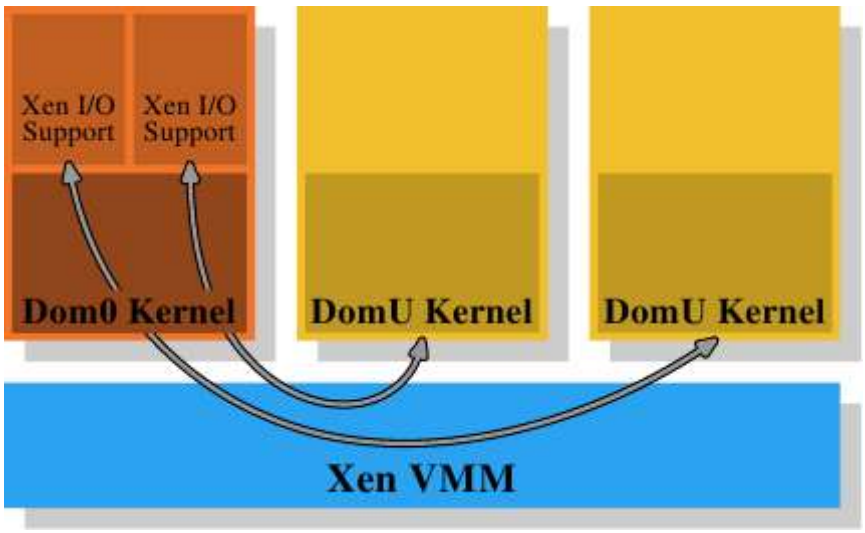

图 4.4: Xen 虚拟化模型

对Xen来说（见图4.4），Xen VMM(Xen内存管理程序)就是那个软件。但是，VMM没有自己实现许多硬件的控制，不像其他早先的系统（包括Xen VMM的第一个版本）的VMM，内存以外的硬件和处理器由享有特权的Dom0域控制。现在，这基本上与没有特权的DomU内核一样，就内存处理方面而言，它们没有什么不同。这里重要的是，VMM自己分发物理内存给Dom0和DomU内核，然后就像他们是直接运行在一个处理器上一样，实现通常的内存处理

为了实现完成虚拟化所需的各个域之间的分隔，Dom0和DomU内核中的内存处理不具有无限制的物理内存访问权限。VMM不是通过分发独立的物理页并让客户OS处理地址的方式来分发内存；这不能提供对错误或欺诈客户域的防范。替代的，VMM为每一个客户域创建它自己的页表树，并且用这些数据结构分发内存。好处是对页表树管理信息的访问能得到控制。如果代码没有合适的特权，它不能做任何事。 在虚拟化的Xen支持中，这种访问控制已被开发，不管使用的是参数的或硬件的（又名全）虚拟化。客户域以意图上与参数的和硬件的虚拟化极为相似的方法，给每个进程创建它们的页表树。每当客户OS修改了VMM调用的页表，VMM就会用客户域中更新的信息去更新自己的影子页表。这些是实际由硬件使用的页表。显然这个过程非常昂贵：每次对页表树的修改都需要VMM的一次调用。而没有虚拟化时内存映射的改变也不便宜，它们现在变得甚至更昂贵。 考虑到从客户OS的变化到VMM以及返回，其本身已经相当昂贵，额外的代价可能真的很大。这就是为什么处理器开始具有避免创建影子页表的额外功能。这样很好不仅是因为速度的问题，而且它减少了VMM消耗的内存。Intel有扩展页表(EPTs)，AMD称之为嵌套页表(NPTs)。基本上两种技术都具有客户OS的页表，来产生虚拟的物理地址。然后通过每个域一个EPT/NPT树的方式，这些地址会被进一步转换为真实的物理地址。这使得可以用几乎非虚拟化情境的速度进行内存处理，因为大多数用来内存处理的VMM条目被移走了。它也减少了VMM使用的内存，因为现在一个域（相对于进程）只有一个页表树需要维护。 额外的地址转换步骤的结果也存储于TLB。那意味着TLB不存储虚拟物理地址，而替代以完整的查询结果。已经解释过AMD的帕西菲卡扩展为了避免TLB刷新而给每个条目引入ASID。ASID的位数在最初版本的处理器扩展中是一位；这正好足够区分VMM和客户OS。Intel有服务同一个目的的虚拟处理器ID(VPIDs)，它们只有更多位。但对每个客户域VPID是固定的，因此它不能标记单独的进程，也不能避免TLB在那个级别刷新。

对虚拟OS，每个地址空间的修改需要的工作量是一个问题。但是还有另一个内在的基于VMM虚拟化的问题：没有什么办法处理两层的内存。但内存处理很难（特别是考虑到像NUMA一样的复杂性，见第5部分）。Xen方法使用一个单独的VMM，这使最佳的（或最好的）处理变得困难，因为所有内存管理实现的复杂性，包括像发现内存范围之类“琐碎的”事情，必须被复制于VMM。OS有完全成熟的与最佳的实现；人们确实想避免复制它们。

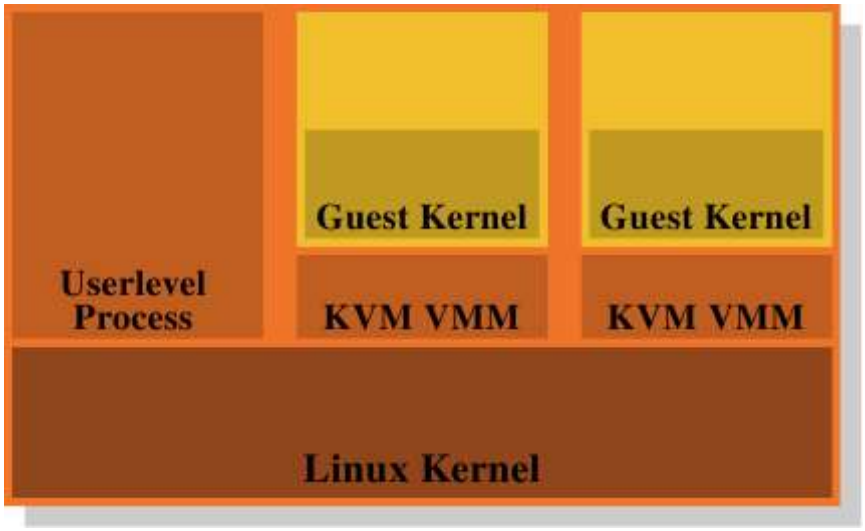

图 4.5: KVM 虚拟化模型

这就是为什么对VMM/Dom0模型的分析是这么有吸引力的一个选择。图4.5显示了KVM的Linux内核扩展如何尝试解决这个问题的。并没有直接运行在硬件之上且管理所有客户的单独的VMM，替代的，一个普通的Linux内核接管了这个功能。这意味着Linux内核中完整且复杂的内存管理功能，被用来管理系统的内存。客户域运行于普通的用户级进程，创建者称其为“客户模式”。虚拟化的功能，参数的或全虚拟化的，被另一个用户级进程KVM VMM控制。这也就是另一个进程用特别的内核实现的KVM设备，去恰巧控制一个客户域。

这个模型相较Xen独立的VMM模型好处在于，即使客户OS使用时，仍然有两个内存处理程序在工作，只需要在Linux内核里有一个实现。不需要像Xen VMM那样从另一段代码复制同样的功能。这带来更少的工作，更少的bug，或许还有更少的两个内存处理程序接触产生的摩擦，因为一个Linux客户的内存处理程序与运行于裸硬件之上的Linux内核的外部内存处理程序，做出了相同的假设。

总的来说，程序员必须清醒认识到，采用虚拟化时，内存操作的代价比没有虚拟化要高很多。任何减少这个工作的优化，将在虚拟化环境付出更多。随着时间的过去，处理器的设计者将通过像EPT和NPT技术越来越减少这个差距，但它永远都不会完全消失。

本文地址：https://www.oschina.net/translate/what-every-programmer-should-know-about-virtual-memory-part3
原文地址：http://lwn.net/Articles/253361/
