成空行）。若构建 2.6+ 内核时定义了 CONFIG_PREEMPT 配置，回旋锁定就获得了一个副作用，可以在获取和释放回旋锁之间禁止抢占。

好了，基本上就这些了。“几乎所有”是个危险的用词。只是偶尔情况下，某段数据操作本质上是 SMP 安全的（最明显的例子就是在寻址各 CPU 特有数据时）所以并不需要回旋锁，但的确需要防止抢占，因为同一 CPU 上其它内核代码可能在访问同一数据。加入内核抢占后保持 2.6 内核的稳定仍然需要相当多耐心的代码整理和更多耐心的调试。

14.3 系统调用时发生什么

系统调用“仅仅是”内核实现的供用户程序用的例程。有些系统调用只是返回内核知道而外面不知道的信息（比如当天的精确时间）。

但有两个原因说明为什么情况要比上面的复杂。第一个与安全性有关——人们认为内核在面对有错的或者恶意的应用程序代码时应当仍然是稳健的。

第二个原因与稳定性有关。Linux 内核应当能够运行专门为它生成的程序，但是也应当能够运行过去为同样的或者兼容的体系结构生成的应用程序。系统调用一旦定义，要删除就得经过许多的争论、工作和等待才可以。

我们先回到安全性。既然系统调用运行于核心态，核心态的入口必须受到控制。对于 MIPS 体系结构，这是通过 **syscall** 指令由软件触发的异常来实现的，进入内核异常处理程序时，CPU 的 **Cause(ExcCode)** 寄存器域有一个特殊的代码（8 即 “sys”）。最底层的异常处理程序将会根据该域的值决定下一步干什么，并切换到内核的系统调用处理程序。

那只是单个入口点：应用程序设置一个数字参数选择几百个系统调用功能中的哪一个。系统调用值依赖于体系结构：MIPS 上对某个功能的系统调用号可能不同于 x86 上对同一功能的系统调用号。

系统调用参数尽可能在寄存器中传递，有利于避免用户空间和内核空间之间不必要的数据拷贝。在 32 位 MIPS 系统中：

- 系统调用号放进 **v0**。
- 参数传递按照“o32”ABI 要求。大多数时间着等于说，最多可有四个参数在寄存器 **a0–a3** 中传递；但确实有角落的情形比较特，这些都放在 11.2.1 节讲。[8]

  虽然内核采用类似 o32 的方式定义系统调用中的参数传递，那并不是说用户空间的程序必须要用 o32。良好的编程实践作法要求用户空间对操作系统的系统调用总是通过 C 运行库或者等价的机制传递，这样系统调用的接

[8] 参数传递的最黑暗/偏僻的角落里，许多都不影响内核，因为并不处理浮点值，也不通过值传递参数和返回结果。

口只需要维护一处。库可以从任何 ABI 转换成类似 o32 的系统调用标准，在 64 位 MIPS Linux 上运行 32 位软件时就是这样做的。

- 系统调用的返回值通常位于 **v0**。但是 `pipe(2)` 系统调用返回一个由两个 32 位文件描述符构成的数组，用了 **v0** 和 **v1** ——也许有一天会有另外一个系统调用完成同样的功能。[9]

  一切有可能失败的系统调用在 **a3** 中返回一个状态（0 表示良好，非零表示出错）。

- 为了和调用约定保持一致，系统调用保留了 o32 要求在函数调用时保存的那些寄存器的值。

保证内核对系统调用实现的安全性涉及到适度剂量的偏执。数组下标、指针、缓冲区长度都必须进行检查。并非允许所有的应用程序可以为所欲为，系统调用的代码在必要要时（大多数时候，以超级用户 root 运行的程序可以做任何事情）要检查调用者的“权能(capabilities)。”

运行系统调用的内核代码处于进程环境——内核代码最宽松的环境。系统调用代码可以执行任何在发生 I/O 事件时要求线程睡眠的事情，或者访问需要交换进内存的虚拟地址。

进行数据传输时，你要依赖于应用程序提供的指针，而该指针可能是垃圾值。如果我们用了无效指针，就会引发一个异常，有可能导致内核崩溃——那是不能接受的。

所以从用户空间拷贝输入数据或者向用户空间拷贝输出数据可以通过专门用于该目的的函数 `copy_to_user()/copy_from_user()` 安全完成。如果用户真的传递了一个坏地址，将会导致内核异常。内核维护一个执行危险任务时可信任的函数[10] 列表：`copy_to_user()` 和 `copy_from_user()` 位于表中。当异常处理程序看到异常重新开始地址位于这些函数之一时，就返回到一个精心构造的错误处理程序。应用程序就会被发送一个粗鲁严厉的信号，但是内核安然无恙。

从系统调用返回是通过 **syscall** 异常处理程序结尾的 **eret** 指令返回的。重要的一点是切换回用户态和返回用户指令空间要同时执行。

14.4 MIPS/Linux 系统的地址转换

在我们开始了解怎样实现之前，先对整个工作有个整体轮廓。

图 14.1 是一个 32 位 MIPS/Linux 系统上的线程的存储器映射示意图。[11] 这个映射必须与硬件映射相一致，所以用户可访问的存储区必然位于下半部。

记住下面几点有用：

[9] 我听到内核维护人员异口同声地表示“我们誓死反对（除非踏过我们的尸体）。”

[10] 其实是一个指令地址范围的列表。

[11] 64 位 MIPS/Linux 系统的映射基于同样的原理，但是由于硬件映射更为复杂——参见图 2.2——因而相对麻烦一点。

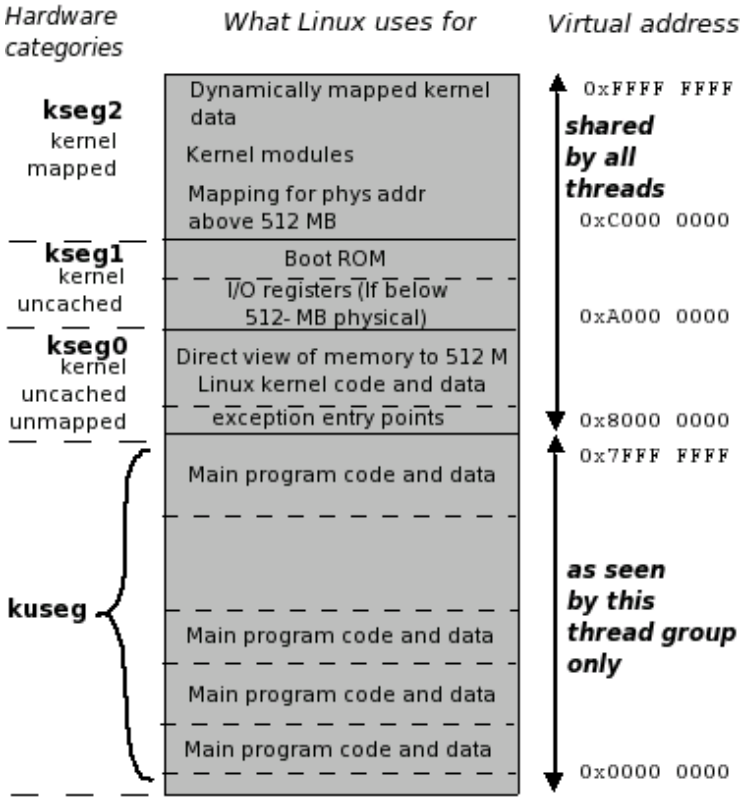


图 14.1: Linux 线程存储器映射

- 内核从什么地方开始运行：MIPS Linux 内核的代码构建为在 kseg0 区运行；虚拟地址从 `0x8000 0000` 向上。这个范围的地址仅仅是一个到物理内存低 512 MB 的窗口，无需 TLB 管理。

- 异常入口点：目前为止的大多数 MIPS CPU 中，这都由硬件布线固化到 kseg0 底部附近。最新的 CPU 可以提供 **EBase** 寄存器，对异常入口重新定位（参见第 3.3.8 节），主要是让多个共享内存的 CPU 能用不同的异常处理程序而不用费力去做特殊的存储器译码。在 Linux 内核中，就算有多个 CPU 也都运行同一个异常处理代码，所以这个特性在 Linux 中不大可能用到。

- 用户程序从什么地方开始运行：MIPS Linux 应用程序（运行于低特权级的“用户态”）虚拟地址从 `0` 到 `0x7FFF FFFF`。该区的地址在用户态可以访问，要经过 TLB 地址转换。

  应用程序的主程序构建时自接近零的地址开始运行。不会真为零——从虚拟零地址开始的一两页不做地址映射，这样企图使用空指针就会被当作内存管理错误捕获。应用程序的库函数部分，在加载或者更晚的时候递增加

载到用户空间。这样做可以是因为库函数构建为位置无关类型（参见第 16 章），可以根据实际被加载的地址空间自动调整。

- 用户堆和栈：应用程序的栈初始设置到用户可以访问的空间（约 2 G 虚拟空间）的顶部并且向下增长。操作系统监测到对已分配的最低栈空间附近未映射的存储器访问时，会自动映射更多的页以满足栈的增长。

  同时，新的共享库或者直接用 `malloc()` 分配的用户数据及其后代从用户空间底部向上增长。只要这些空间的总和不超过 2 GB，什么事都没有：除了最大型的服务器以外，这个限制基本不成问题。

- 512 MB 以内的存储器：可以通过 kseg0 经过高速缓存访问或者通过 kseg1 不用高速缓存访问。历史上，Linux 内核假定自己可以直接访问机器的全部物理内存。对于用 512 MB 或者更少物理内存范围的小 MIPS 系统，这是对的；在这种情况下，全部内存都可以在 kseg0（用高速缓存）和 kseg1（不用高速缓存）区访问。

- 512 MB 以上的“高位存储器”：现在 512 MB 即使对于嵌入式系统也已经不够了。Linux 有一个独立于硬件体系结构的“高位存储器” 概念——要用特殊的、依赖于硬件体系结构的方式处理的物理存储器，对于 32 位 Linux/MIPS 系统，512 MB 以上的物理内存就是高位存储器。当我们要访问时，需要创建适当的地址转换数据项并且即时复制到 TLB。

早期的 MIPS CPU 追求在 Unix 工作站和服务器中应用，所以 MIPS 存储器管理硬件被设计成能够为 BSD Unix 提供存储器管理的最小硬件。BSD Unix 系统是第一个在 DEC VAX 小型机上提供真正的分页虚拟存储的 Unix 操作系统。VAX 在很多方面成为了此后统治计算机界的 32 位分页转换虚拟存储体系结构的典范；也许 MIPS 里有一些 VAX 存储器管理组织的余音并不奇怪。但是这是一个 RISC，MIPS 的硬件做的要少得多。特别是，VAX（或者 x86）用微代码解决的很多问题在 MIPS 系统中都留给了软件去做。

在这一章，我们从 MIPS 硬件开始的地方开始，先考虑一个基本的 Unix 之类的操作系统及其虚拟存储系统的需求；但是这次我们的操作系统是 Linux。

我们将看到 MIPS 的硬件本质上是对这种需求的一种合理回应。对于真正细节，请参阅第 6 章。

14.4.1 问什么要做存储器地址转换

存储器地址转换硬件（为了通用，我们称其为 MMU，即*存储器管理单元 memory management unit*）服务几个不同的目标：[12]

[12] 考虑到 MMU 贡献的分量，让人有点不可思议的是在没有 MMU 的情况下可以构建一个大体可以工作的最小 Linux(uCLinux) ——但那是另外一回事了。

- 隐藏和保护：用户特权级的程序只能访问程序地址位于 kuseg 存储区的（低程序地址）的数据。这样一个程序只能到达操作系统允许的内存区。

  此外，可以单独指定每个页面为可写或写保护；操作系统甚至可以停止一个意外覆盖自己代码的程序。

- 给程序分配连续的存储空间：有了 MMU，操作系统可以从物理上分散的页面构造连续的程序空间，允许我们从一个简单的固定大小页的缓冲池中分配存储器。

- 扩展地址范围：有些 CPU 不能直接访问它们全部的物理存储器范围。MIPS32 CPU 尽管是真正的 32 位体系结构，对地址映射的安排使得不做映射的地址空间窗口 kseg0 和 kseg1（不依赖 MMU 表作地址转换）为物理存储器的低 512 MB。如果你需要更大范围的存储器映射，就必须经过 MMU。

- 使存储器映射适应你的程序：有了 MMU，你的程序可以使用适合自己的地址。在大的操作系统里，可能同一个程序有许多拷贝同时运行，让这些拷贝都用同样的程序地址要容易得多。

- 按需调页：程序运行时就好象需要的所有内存资源都已经分配了一样，但是操作系统实际上只在真正用到时才给出。访问未分配的地址区将产生一个异常，然后由操作系统处理；操作系统加载适当的数据到存储器让程序继续运行。

  理论上（有时在计算机的教科书上）说，按需调页有用是因为这样允许你运行一个物理内存放不下的大程序。如果你有很多空闲时间的话，这话确实不错，但现实中如果一个程序真的需要用超过实际内存的空间，就会不断把自己的数据移出内存，以至于运行得非常非常慢。

  但是按需调页仍然十分有用，因为大程序充满了大量的至少在本次执行中不会运行的生僻代码。也许你的程序内置有对你今天用不到的数据格式的支持；也许程序有很多的错误处理代码，但是这些错误极少发生。在按需调页的系统中，没有用到的程序块从不需要读进内存。这样启动速度也快了，顺便讨好了缺乏耐心的用户。

- 重定位：程序入口点和预先声明的数据的地址在程序编译/构建时是固定的——当然这对于，用于 Linux 的全部共享库和大多数应用程序要用到的位置无关代码来说就不对了。MMU 允许程序在物理内存的任意地址都能运行。

Linux 存储器管理程序的工作的实质就是给每个程序都提供自己的存储空间。

好了，Linux 拥有真正单独的概念：线程是调度单元，而地址空间（存储器映射）是保护单元。一个线程组中的许多线程可以共享一个地址空间。存储器转换系统感兴趣的显然是地址空间而不是线程。Linux 的实现是所有的线程都平等，所有的线程都拥有内存管理的数据结构。运行于同一个地址空间的线程，实际上共享这些数据结构的大部分。

但是就现在而言，如果我们只考虑每个线程一个地址空间的情形就比较简单，我们可以用老的概念“进程”同时指二者。

如果存储器管理工作不出错，每个进程的命运独立于其它进程（操作系统也保护自己）：一个进程可能崩溃或行为不端但不会影响整个系统。对于大学各系的运行学生程序的计算机，这显然是一个很有用的性质，但是即使最严格的商业环境也需要在支持经过测试检验过的软件同时，也支持试验或者原型软件。

MMU 不光是为了大型、完全的虚拟存储系统；即使小的嵌入式系统也能从重定位和更高效的内存分配中受益。任何想要在其中不同时间运行不同程序的系统都会发现，如果可以把程序的地址概念映射到随便一个可用的物理地址空间，一切都要容易得多。

多任务和不同任务地址空间的分离过去只用于大型计算机，后来迁移到个人计算机和小型服务器上，现在在消费类电子设备中的小的计算机上页日渐普及了。

然而，很少的非 Linux 嵌入式操作系统使用独立的地址空间。这可能不是因为该特性没有用处，而是由于嵌入式 CPU 及其可用的操作系统缺乏一致的特性。也可能因为一旦给系统加上独立的地址空间，就太接近于重新发明 Linux 而失去意义!

这对 MIPS 体系结构来说是一个意外的好事。1986 年为了简化让工作站 CPU 而不得不进行的简约设计对于 21 世纪初期的嵌入式系统是个巨大的财富。即使小的应用程序，被快速膨胀的代码所困扰，需要用尽所有已知的技巧管理软件复杂性；以 MIPS 为先驱的基于软件的灵活的方法及有可能提供需要的任何东西。几年前，很难说服定位于嵌入式市场的 CPU 厂商加上 MMU；现在（2006 年）Linux 已经无处不在了。

14.4.2 基本的进程布局和保护

从操作系统的角度来看，低端存储空间(kuseg) 是个“沙盒”，在其中用户程序可以随便玩。如果由于程序失控丢失了自己全部的数据，这不关别人的事。

从应用程序的角度来看，该区可以自由使用来构建任意复杂的私有数据结构以完成工作。

在用户区内程序的沙盒里面，操作系统按需要（随着栈向下增长而隐含）即时提供更多的栈空间。也会提供一个系统调用获得更多的可用空间，从预先声明的数据的最高地址开始向上增长——系统人员称之为堆。堆可供 `malloc()` 之类向程序提供额外内存块的库函数使用。

堆和栈的空间供应的单位小到对于系统存储器适度节俭，大到能避免过多的系统调用或者异常。但是在每个系统调用或异常发生时，操作系统都会得到一个机会来控制应用程序的内存消耗。操作系统可以实施限制以保证应用程序不会因为占用内存太多而威胁其它关键的活动。

Linux 线程在操作系统核心保持自己的身份，当线程在内核里运行于进程环境（比如系统调用）时，其实只是调用一个精心编写的例程。

操作系统自己的代码和数据当然不能让用户空间程序访问。在一些系统上，这是通过把系统放到一个完全独立的地址空间做到的；在 MIPS 上，操作系统共享同一个地址空间，当 CPU 运行在用户程序特权级时，对这些地址的访问是非法的，会触发一个异常。

注意到虽然每个进程的用户空间映射到自己私有的真实存储，但特权空间是共享的。所有进程看到的操作系统代码和数据都位于同一个地址——操作系统内核是一个多线程但里头只有单一地址空间的系统——但是每个进程的用户空间地址访问自己单独的空间。可以信任运行于进程环境的内核例程会安全合作，但是根本不必信任应用程序。

我们提到栈被初始化到允许的最高用户地址：那样做可以支持使用大量存储器的程序。其结果导致使用的地址跨度很大（中间有一个巨大的空洞），这是这种地址映射的一个特征，在进行转换时必须要考虑。

Linux 把应用程序的代码映射为对本程序只读，这样代码可以被不同地址空间的线程安全共享——毕竟，许多进程运行同一个程序的情况很常见。

许多系统不仅共享整个应用程序，而且共享通过库调用访问的应用程序块（共享库）。Linux 用共享库做到这点，我们后面再接着讲那个故事。

14.4.3 进程地址到真实存储器的映射

支持这个模型需要哪种机制？

我们想要能够运行同一个程序的多个拷贝，各个程序拷贝最好使用不同的数据拷贝。这样在程序执行过程中，当程序加载时应用程序地址根据操作系统固定的方案映射到物理地址。

当然让操作系统在我们从一个进程环境切换到另一个的时候急急忙忙到处去拼凑地址转换信息是可能的，但这样会非常低效。取而代之的是，我们给每个活动线程的存储器映射一个 8 位的数，称为地址空间 *ID* 或 ASID。当硬件发现一个地址转换项的时候，寻找一个匹配当前 ASID 及地址的转换项，[13] 这样硬件可以容纳不同空间的转换而不会搞混。现在当调度不同地址空间的线程时软件要做的全部就是加载一个新的 ASID 到 **EntryHi(ASID)**。

映射机制也允许操作系统区分用户空间的不同部分。应用程序空间的有些部分（通常是代码部分）可以映射为只读；有些部分留下不做映射，在访问时自陷，这意味着失去控制的程序可能会早点被停止。

[13] 实际上，我们可以看到有些转换项可以与 ASID 无关，即“全局”的。

进程地址空间的内核部分由所有进程共享。内核构建时就是为了在特定的地址运行，所以基本上不需要灵活的映射机制，可以利用 kseg0 区，把存储器映射资源留给应用程序使用。有些动态生成的内核区域是从映射的存储器（例如，用于容纳设备驱动程序的内核可加载模块等等都是映射的）方便地构建起来的。

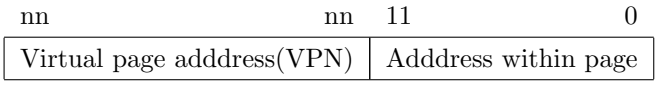
14.4.4 分页映射

为了地址映射尝试过许多古怪的方案。直到 1980 年代中期前后，工业界还在想肯定有比固定大小的分页更好的解决方案。毕竟固定大小的页没有考虑应用程序的行为或需求，而且没有哪种自顶向下的分析会发明这个方法。

但是如果程序想要什么（当时想要的存储块大小），硬件就给什么，可用存储器很快就变成尺寸尴尬的碎片。当我们最后和外界接触时，他们的独轮车的轮子是圆的，而他们计算机可能用固定大小的页。

这样所有实际的系统都以页 ——固定大小的存储块——为单位映射内存。页的大小总是 2 的幂。太小的页可能需要太多的管理（大的转换表、一个程序需要太多的转换）；太大的页可能效率不搞，因为很多小数据也要占一个整页。最后 4 K 大小的页成为 Linux 上占据绝对优势的折衷结果。[14]

采用 4 KB 大小的页，程序/虚拟地址可以这样简单划分：



页内地址位不需要做转换，所以存储器管理硬件只需要把地址的高位（传统上称为虚拟页号，即 VPN(virtual page number)）转换成物理地址的高位（物理帧号，即 PFN(physical frame number) ——没人记得当初为什么没叫做 PPN）。

14.4.5 我们真正想要什么

映射机制必须允许程序使用自己进程/地址空间的具体地址，并把它高效地转换成为真正的物理地址来访问存储器。

一种好办法是弄一个表（即*页表*）为整个虚拟地址空间的每一页都保留一项，该项包含正确的物理地址。这显然是个相当大的数据结构，要保存到主存储器里面。但是有两个大问题。

第一个就是我们现在每次 load/store 需要访问两次存储器，这对显然是个性能杀手。你可能预见到了这个问题的答案：我们可以用一个高速缓冲存储器来存储页地址转换数据项，只有在未命中这个高速缓存时才查找驻留内存的表。既然每个项代表 4 KB 存储空间，我们可以只用一个适度小的高速缓存就可以得到一个满意的高命中率。（在这种方案发明的时候，存储器高速缓存还不多见，

[14]MIPS 硬件对于更大的页也没问题，16 KB 是下一个支持的页大小，对于许多现代的系统来说可能是一个更好的折衷。但是 4 KB 的页极为普及，其影响延伸到许多程序的构建方式；所以我们仍然基本上都用 4 KB。

有时也叫做“旁视缓冲器(lookaside buffer)，”所以存储器转换高速缓存就成了转换旁视缓冲器，即 TLB；这个缩写名称一直流传了下来。）

第二个问题是页表的大小；对于 32 位应用程序，地址空间分为 4 KB 的页，有一百万个数据表项，至少要用 4 MB 空间。我们真得想办法让表小一点，否则就没有剩下运行程序的空间了。

我们把解决这个问题的讨论推后，只是提一下很少真有程序用到 32 位寻址的 4GB 空间。多数中等的程序在它们自己的程序地址空间内部有巨大的空洞，如果我们能够发明一种方案，避免存储那些对应于空洞的“什么也没有”的转换表项，那么事情可能要好多了。

我们现在已经快要到 DEC 为其 VAX 小型机设计的存储器转换系统了，它对随后的体系结构有着极大的影响。概要如图 14.2 所示。

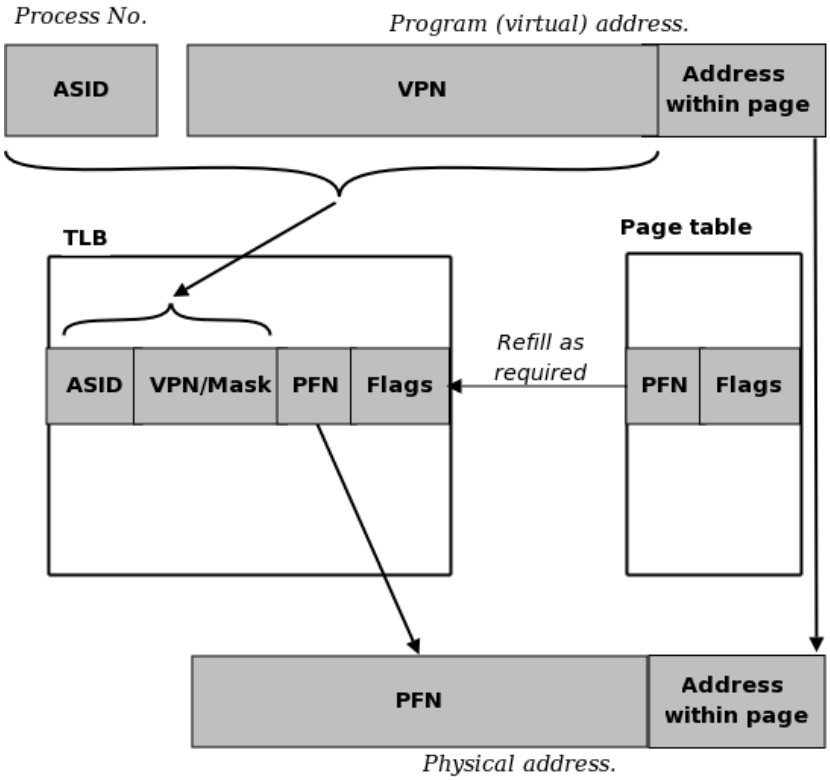


图 14.2: 想要的存储器地址转换系统

硬件工作的步骤如下：

- 虚拟地址分为两部分，最低有效位部分（通常是低 12 位）不做转换直接传递——这样转换总是以页（通常 4 K）为单位进行。
- 较高有效位，即 VPN，与当前线程的 ASID 拼接在一起形成一个唯一的页地址。

- 我们在 TLB（转换高速缓存器）看是否有该页的转换项。如果有，就给出高位的物理地址位，我们就得到了要用的地址。

  TLB 是个专用的存储器，可以用各种有用的方法匹配地址。可能有一个全局标志位告诉它忽略某些项的 ASID，这些 TLB 项可以用来映射某个范围的虚拟地址为每个线程所用。

  类似的，VPN 可以和导致部分 VPN 地址不参与匹配的一些屏蔽位存在一起，允许 TLB 项映射更大范围的虚拟地址。

  这两个特性在 MIPS MMU 中都有（但在一些很老的 MIPS CPU 中没有可变大小的页）。

- 通常有些额外的（标志）位与 PFN 放在一起用来控制访问权限——最明显的就是，只允许读不允许写。我们将在下一节讨论 MIPS 体系结构的标志。

- 如果 TLB 没有匹配的项，系统必须（用驻留主存的页表信息）找出或者创建适当的页表项，加载进 TLB 然后再次运行转换过程。

  在 VAX 小型机里，这个过程由微代码控制，在程序员看来完全是自动的。如果你在存储器中构建了正确格式的页表并让硬件指向它，所有的存储器转换就能工作。

14.4.6 MIPS 设计的起源

MIPS 设计者想要找出一种方法用尽量少的硬件提供和 VAX 同样的功能。微代码控制的 TLB 填充是不可接受的，所以他们走出了大胆的一步，把这部分工作交给软件完成。

那就意味着除了有个寄存器存储当前 ASID 之外，MMU 硬件只是一个简单的高速、固定大小的转换表而已。系统软件可以把（通常也的确用）MMU 硬件看作驻留内存的整个页表的数据的一个高速缓存，所以把硬件表称为 TLB 是有意义的。但是 TLB 硬件中没有作为高速缓存的硬件，除了一点：当给出一个无法转换的地址时，TLB 触发一个特殊的异常（TLB 重填）来调用软件例程。为了帮助软件提高效率，TLB 设计、相关的控制寄存器以及重填异常的细节都要精心设计。

MIPS TLB 总是在片上实现的。即使对于高速缓存引用，存储器地址转换步骤也是必须的，所以基本上处于机器的关键路径上。这意味着 TLB 容量必然小，尤其在早期，所以必须靠设计的巧妙来弥补其容量的不足。

基本上就是一个完全的*相联存储器*。相联存储器的每一项都由一个关键字域和数据域构成；你给出关键字，硬件从该关键字匹配的项返回数据。相联存储器确实非常好，但是其硬件成本很高。MIPS TLB 有 32 到 64 个数据项；这种容量的存储在半导体设计中不会太复杂。

当代所有的 CPU 都用这样一种 TLB，其中每个数据项都加倍，可以映射两个连续的 VPN 到两个互相独立的物理地址。这样一对数据项使得 TLB 只需增加少量逻辑就能让映射的存储器空间加倍，而不需要考虑大幅度的修改 TLB 的管理。

你将会看到，TLB 被说成是全相联的；这强调了所有的关键字都真正并行地与输入值进行比较。[15]

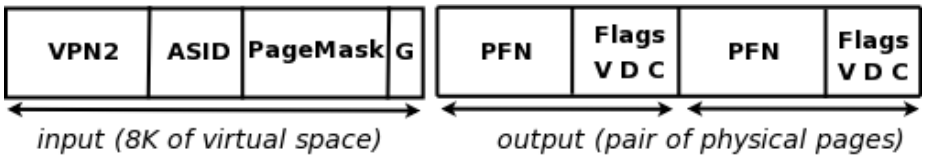


图 14.3: TLB 项的各个域

图 14.3 给出了 TLB 数据项的示意图。就现在而言，我们假定页大小为 4 KB。TLB 的关键字——输入值——由三个域组成：

- VPN2: 页号就是虚拟地址的高序位——当你去掉页内地址 12 位后剩下的部分。VPN2 中的“2”强调每个虚拟项有双倍的输出域而映射 8 KB 地址。虚拟地址的位 12 选择一对物理地址项中的第一个或第二个。

- PageMask: 控制虚拟地址的多少部分和 VPN 相比较，多少部分传递到物理地址；减少匹配位可以映射更大的存储区。“1” 位表示忽略相应的地址位。有些 MIPS CPU 可以设置成最多每项可以映射 16 MB。被忽略位中的最高有效位用来选择奇偶项。

- ASID: 标记该转换属于特定的地址空间，所以只能在给出地址的线程的 **EntryHi(ASID)** 设置等于该值时才参与匹配。

  G 位如果为 1 则关闭 ASID 匹配，使得该转换项适用于所有的地址空间（所以这部分地址映射在所有地址空间之间共享）。ASID 有 8 位：熟悉操作系统的读者可能会意识到即使 256 作为大型 Unix 系统上同时活动的进程数的上限也还是太小了。但是，只要给这个环境下的“活动”赋予特殊的解释“可能在 TLB 中有地址转换项，” 256 还算是一个合理的上限。操作系统软件必要时不得不回收 ASID，这将涉及到对从活动状态降级的进程的地址转换项在 TLB 中进行大清洗。这可是件吃力不讨好的活，但是吃力不讨好的活操作系统不得不干的多了；256 项应该足以保证 TLB 清洗的次数不会多到造成性能问题。

  从编程的角度讲，最简单的是 **G** 位保存在内核的页表中和输出域放在一起。但是当你做地址转换时，**G** 位又属于输入。在 MIPS32/64 CPU 里，两

[15]把通用的 32 项成对 TLB 说成是一个 32 路组相联、每组两个数据项的高速缓存，虽然显得有点学究气，但确实没错。

个输出端值进行“与”操作后生成要用的值，但是要得到有意义的结果，你必须保证两半的 **G** 位设置一样。

TLB 的输出端给出物理页号和少量但是够用的标志位：

- 物理页帧号(PFN): 这是切除低位（如果代表 4 KB 的页就是低 12 位）后的物理地址。
- 写控制位(D): 设为 1 允许对该页写入。“D”来源于该位被称为“dirty” 位；原因请参阅下节内容。
- 有效位(V): 若该位为零则该项不可用。这看上去好像没有什么意义。既然不能用，为什么要把这种记录加载进 TLB 呢？这有两个原因。第一个是因为每一项转换一对虚拟地址，可能只要其中一个。另一个是重填 TLB 的软件为了优化速度，而不想检查特例。如果程序在可以使用驻留内存的页表所指向的页之前还需要进一步的处理，可以保留驻留内存的页表项并标记为无效。在 TLB 重填之后，这会导致另一种自陷来调用特殊的处理，而不需要每次软件重填的时候都作检测。
- 高速缓存控制位(C): 这个 3 位的域的首要目的是区分可高速缓存的（cachable）和不作高速缓存的（uncached）区域。

  但是那还剩下六个取值，用于两个有点互不相容的目的：在共享存储器的多处理器系统中，不同的值用来给出存储器是否共享的提示（在硬件要努力保证整个机器上各个高速缓存的数据一致性时要用到）。在“嵌入式”CPU 中，不同的值选择不同的局部高速缓存管理策略：比如透写还是回写等等。具体请参阅你的 CPU 手册。

地址转换现在简单了，我们可以把上面的描述详细展开如下：

- CPU 生成一个程序地址：这可能是取指或读写操作——而且其程序地址并不位于 MIPS 地址空间特殊的非映射区。

  低 13 位是分开的，剩下的 VPN2 和当前 ASID（**EntryHi(ASID)**）合在一起，在 TLB 中进行查找。查找匹配还受到各个 TLB 项的 **PageMask** 和 **G** 域的影响。
- TLB 匹配关键字：如果没有相匹配的关键字，就触发一个 TLB 重填异常。但是如果有匹配，就选择该项。虚拟地址的位 12 用来选择要用哪半面的物理地址。

  来自 TLB 的 PFN 粘到程序地址的低位上构成一个完整的物理地址。
- 地址有效吗？要察看 V 和 D 位。如果不有效，或者 D 位是零而试图存储，CPU 就会触发异常。与所有地址转换自陷一样，**BadVAddr** 寄存器

会被填上相应的程序地址；与所有的 TLB 异常一样，TLB 的 **EntryHi** 寄存器会预加载相应地址的 VPN。

便利寄存器 **Context**（64 位 CPU 则为 **XContext**）的 **BadVPN2** 域将会（部分）预加载我们在 TLB 异常重填期间未能转换的虚拟地址的若干位。但是规范对于这些地址域在别的异常中的行为没有明确说明。在其它异常中坚持只用 **BadVAddr** 可能是个好办法。

- 是否高速缓存？如果 C 位置位，CPU 在高速缓存中查找物理位置数据的拷贝；如果没有找到，就从存储器中取出数据，并在高速缓存中留下一个拷贝。当 C 位清零时，CPU 既不查找也不填充高速缓存。

当然了，TLB 中数据项的个数允许你只转换相对少量的程序地址——几百 KB。这对于大多数系统来说远远不够。TLB 几乎总是用作比自身容量大得多的地址转换数据表的一个由软件维护的高速缓存。

当程序地址在 TLB 中查找失败时，就发生一次 *TLB 重填自陷*。[16] 系统软件要完成如下的工作：

- 先找出是否存在正确的地址转换；如果没有，该自陷将调用处理地址错误的软件。
- 如果有正确的地址转换，系统将构造一个实现该转换的 TLB 项。
- 如果 TLB 已经满了（实际运行的系统中几乎永远是满的），由软件选择一项可以丢弃的。
- 软件将新的地址转换项写入 TLB。

参见第 14.4.8 节以了解在 Linux 中的具体做法。

14.4.7 跟踪被修改的页（模拟“Dirty”位）

给应用程序提供存储器页使用的操作系统常常想要跟踪自上次操作系统（从磁盘或者网络）取得或者保存该页的拷贝以来该页是否作了修改。未修改的页（即“净(clean)”页）可以直接丢弃，因为再次要用时可以很容易从文件系统恢复。

用操作系统的行话来说，修改的页叫做“脏(dirty)”页，操作系统必须仔细处理，一直到应用程序结束或者脏页因为保存到后备存储器而“净化。” 为了帮助这个处理，CISC CPU 通用的做法是在驻留内存的页表中维护一个位，用以表示发生了对该页的写操作。MIPS CPU 即使在 TLB 数据中也不支持这个特性。

[16] 这究竟应当称作是“”TLB 失效(miss)”（即刚发生的）还是“TLB 重填(refill)”（即我们将要做的）？恐怕我们可能在 MIPS 文档中二者都用。

页表中的 D 位（可从 **EntryHi** 寄存器找到）是写允许位，当然是用来标记只读页的。

所以采用了如下的技巧：

- 当可写的页首次加载进内存时，将其页表项的 D 位清零（表示只读）。
- 当试图对该页写入时，就会发生自陷；系统软件将认出这是一个合法的写操作但利用这个事件在驻留内存页表中设置一个“modified”位——因为该位处于 **EntryLo(D)** 的位置，使得将来的写操作可以不发生异常而正常进行。
- 你也要设置 TLB 项中的 D 位，这样写操作可以进行（但是鉴于 TLB 项的替换是随机不可预测的，把这作为记住被修改状态的方法是没有用的）。

14.4.8 内核对 TLB 重填异常的服务过程

MIPS 的 TLB 重填异常总是有自己独自的入口点（至少在 CPU 尚未处于异常模式时如此；在 Linux 中异常模式中发生重填异常是一个致命错，因此不予考虑）。

当调用异常例程被时，硬件已把 **EntryHi(VPN2)** 设置为刚才未能转换的地址的页号：**EntryHi** 设置成恰好是创建映射相应地址的新 TLB 项所需要的格式。

硬件也要设置一些其它地址相关的域，但是这些我们一律都不用。

特别的，我们不用第 6.2.4 节介绍的用 **Context** 找出相关的页表项的方便易用的“MIPS 标准”做法。MIPS 标准做法要求在 kseg2 构建一个（名义上很长的）线性页表（实际上不会真的占用太多空间，因为 kseg2 区的地址要进行映射，表中大范围的空洞不会映射到真实的存储器）。但是这样就要求每个线程组有各自不同的内核映射，Linux 不想要这种结果。

代替的做法是，Linux 的 TLB 重填页表组织成一个三级页表（分别称为“全局级(global)”、“中间级(middle)”和“PTE”）。但是巧妙的使用 C 语言的宏可以不用改动代码而让中间级完全不出现——两级结构对于 32 位 MIPS 足够了。[17] 这样你可以沿着连接找到你想要的任何 TLB 项的数据——如果该数据存在的话，如图 14.4 所示。

这种结构对于核心存储器使用相对节省：一个 50 MB 虚拟地址空间的大点的 Linux 程序大约有 12 K 4 KB 大小的页，每个需要四字节的 PTE：这需要 48 KB 或者 12 个页。这是一个保守的估计，因为地址空间内部有空洞，但是合理大小的线程组的映射需要占用 15 页左右。内核（kseg2）映射占用的要多些，但是因为对所有存储器映射都通用，所以内核映射只有一组 PTE。

[17] 但是对于 64 位 MIPS 上扩展的虚拟存储器空间，所有三级都是必须的。

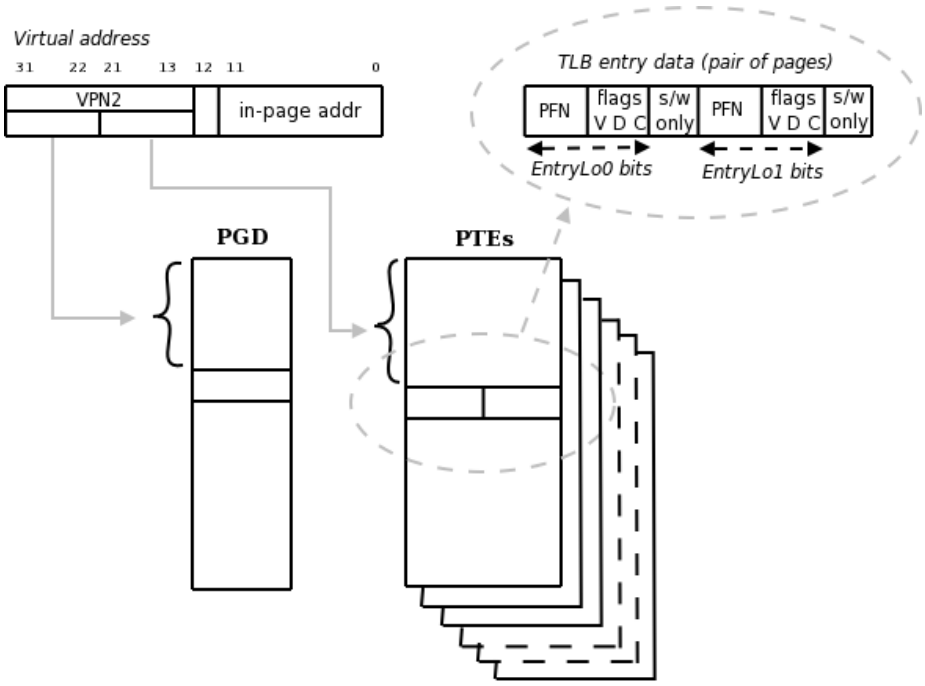


图 14.4: Linux 两级页表（32 位 MIPS 设计）

大多数 32 位 CPU 物理地址空间也限制为 32 位：此时，在 **EntryLo0–1** 寄存器中有六个高位未用。Linux 回收这些位用来记录关于页表项的一些软件状态。

这的确意味着对这 32 位的 TLB 重填处理程序需要做两次索引计算和三次读存。那当然比原始的 MIPS 方案（其中索引计算由 **Context** 寄存器完成的，唯一的读存就是从最后的页表读取）的少量指令序列要长，但是也还不错。

你可能注意到这里的叙述只是针对某类特定的 MIPS CPU。每个 Linux 系统的内核都不同吗？当然不是：但是在当前的 Linux/MIPS 的内核里，某些关键例程——包括 TLB 未命中异常处理程序——是由表驱动的内核软件在启动过程中生成的二进制代码，针对具体 CPU 作了裁减。

作为例子，下面给出为一个基于 32 位 MIPS 24-K 核的系统自动生成的 TLB 未命中处理程序。

```
tlb_refill:
    # (1) get base of PGD into k1
    lui     k1, %hi(pgd_current)

    # get miss virtual address (VA)
    mfc0    k0, c0_badbaddr # (2) moved up to save a cycle
    lw      k1, %lo(pgd_current) (k1)
```

```
srl     k0, k0, 24
sll     k0, k0, 2       # (3) shift and mask VA for PGD index
addu    k1, k1, k0      # got a pointer to the correct PGD entry

# get VA again, this time from Context register
mfc0    k0, c0_context  # (4) moved up to save a cycle
lw      k1, 0(k1)       # OK, read the PTE pointer

srl     k0, k0, 1       # (5) Context register designed for 2x64-bit,
                        # entry, ours are half that size
andi    k0, k0, 0xff8   # (6) mask out higher VPE entry

# load the TLB entries
lw      k0, 0(k1)       # (7) will lost a cycle here
lw      k1, 4(k1)

srl     k0, k0, 0x6     # (8) shift-out software-only bits
mtc0    k0, c0_entrylo0
srl     k1, k1, 0x6     # same for other half
mtc0    k1, c0_entrylo1

ehb                     # (9) wait for CP0 values to be ready
tlbwr                   # (10) write into TLB (somewhere)
eret                    # (11) back to user
```

TLB 重填代码的说明

(1) 在异常处理程序中，软件约定允许我们无偿使用两个寄存器（**k0–1**）。其它所有寄存器都还保留着应用程序的数据，我们不能碰。

(2,4,5) 该代码因为一个序列和另一个交替而搞得比较难读。这样做是为了效率：24-K CPU 读存需要基址寄存器提前一个周期准备好，所以如果你在上一条指令计算基址就会导致一个周期的停顿。在 (2) 和 (4) 处我们找到了一个没有依赖的指令进行交替，但在 (5) 处就没有。

(3) 从图 14.4 可见，我们用虚拟地址的高位和未命中的转换地址（位于 **BadVAddr**）作为 PGD 的索引。因为 PGD 在每一项中都有指针（四个字节），我们需要把索引值左移二位得到字节偏移量。你不能简单的右移 22 位，因为你可能生成一个没有字对齐的指针，这对 MIPS 的字加载操作是非法的。

(4) 参见 (2,4,5)。

(5) 只有两个寄存器可以用，我们不能保存最初的虚拟地址。所以现在再次获取其值，但是这次是从 **Context** 寄存器读取，其中有我们需要的“VPN2” 域的低位。对于 64 位 CPU ——需要保持两个 64 位的项来填充 **EntryLo0–1** 对——结果刚刚好，VPN2 数上移 4 位构成一个由 16 字节项构成的表的索引。但是 MIPS32 上的 Linux/MIPS 的 PTE 表中每项是 8 个字节，所以我们还要右移一位。

(6) **Context** 的高位不是想要的，所以我们将其屏蔽出去。即使在页号之上的位也并非总是为零：SMP Linux 系统用 **Context(PTEBase)** 域，仅由普通的读写位构成，其中来存储一个 CPU ID。

(7) 参见 (2,4,5)。

(8) 在这类拥有 32 位物理地址空间的系统上，**EntryLo** 寄存器只有 26 个有意义的位。Linux 把其余 6 个用作软件标志，我们通过移位去掉了这几位。

(9) 这条执行遇险防护指令保证后续的指令延迟至对 CP0 的 **EntryLo0–1** 寄存器的写入操作生效之后。在许多 CPU 上，从 **mtc0** 指令向 CP0 寄存器的写入擦作在流水线的后期才生效。

(10) **tlwr** 将一个完整的 TLB 项写入 TLB 的一个“随机”位置。事实上，它从 **Random** 获取索引，该寄存器在各个 TLB 位置之间连续循环计数，但是 TLB 未命中发生的时间足够随机，实际效果很好。

注意 TLB 的内容构成了四个 CP0 寄存器：我们刚刚加载了 **EntryLo0–1**，**EntryHi** 由 TLB 未命中异常硬件自动设置，**PageMask** 由 Linux/MIPS 内核维护为对应于 4 KB 页的常数。

(11) **eret** 把我们带回到遭受 TLB 未命中的指令，这回做得应该好一点。注意 **eret** 还是一个 CP0 遇险防护指令，所以直到 **tlbwr** 完成之后才允许对返回用户的指令取指。

14.4.9 TLB 的维护保养注意事项

MIPS TLB 只是一些地址转换数据项的集合。每一项完全是由于重填处理程序的行动才进入 TLB。当然大多数项也以同样方式被覆盖。但是有时候内核想要改变某个页表项，这时作废 TLB 中的拷贝就很重要了。

对于单个的项，我们可以用虚拟地址+原始项的 ASID 查找 TLB，执行 **tlbp**: 如果那里有匹配项，我们可以用一个无效的转换覆盖该页的转换项。

然而，有时候你可能要大规模的动作。ASID 机制允许 TLB 容纳 256 个不同的存储器映射项，但是 Linux 系统在启动和关机之间通常运行的进程数要多于 256 个。所以有时候要清除一大批地址转换，因为要回收 ASID 用于新进程，这样的老的地址转换就完全错了。

没有简洁的方法能做到这点。你只得依次遍历所有的 TLB 项（索引值在 **Index** 寄存器里面），把每一项读到常规寄存器中，检查 **EntryHi(ASID)** 与的 ASID 值，作废与被回收的 ASID 值相匹配的每一项。

14.4.10 存储器地址转换和 64 位指针

当 MIPS 体系结构发明的时候，32 位 CPU 已经出现了一段时间了，当时最大的程序的数据已经快增长到 100 MB ——地址空间只剩下 6 位左右。[18] 因此有足够的理由要精心使用 32 位空间，不要因为昂贵的分段而减少空间；这就是为什么应用程序（以用户特权运行的）为自己保留 31 位寻址空间。

当 MIPS III 指令集于 1991 年引进 64 位寄存器时，一度领先于整个业界，正如我们在 2.7 节讨论的，MIPS 可能比 32 位地址真正显出不足要早四到六年。寄存器宽度的加倍只要给地址空间增加几个位就可以保证未来够用了；更重要的是得考虑操作系统数据结构潜在的爆炸性增长而不是有效地利用全部地址空间。

由于基本的 64 位存储器映射对实际的地址空间的限制在一段时间内还不会达到；理论上允许映射的用户和其它空间在不用重新组织的情况下就能一直增长到 61 位。但是到目前为止，40 位的用户虚拟空间已经很充足了。大多数其它的 64 位 Linux 系统采用 8 KB 的页，但是 8 KB 的页在 MIPS 中很麻烦。

MIPS TLB 的单个转换项可以映射要么 4 KB 要么 16 KB 大小的页，但不能映射 8 KB 的页。这样 64 位 MIPS 内核用 4 KB 的页，16 KB 的页作为在勇敢的未来一个选项也不错。

如果你回头看一下图 14.4 再想象一下在 PGD 和 PTE 之间的一组中间（PMD）表，你就很轻松地在三级页表中解析 40 位虚拟地址。我们把细节留给那些愿意阅读源码的热心读者。

[18] 历史上，应用程序对存储器空间的需求看上去每年增长大约 $\frac{3}{4}$ 位。