（P2），但硬件分不清哪个项属于哪个进程。所以我们还需要做一些工作，让 TLB 正确而高效地支持跨多进程的虚拟化。因此，关键问题是：

关键问题：进程切换时如何管理 TLB 的内容

如果发生进程间上下文切换，上一个进程在 TLB 中的地址映射对于即将运行的进程是无意义的。硬件或操作系统应该做些什么来解决这个问题呢？

这个问题有一些可能的解决方案。一种方法是在上下文切换时，简单地清空（flush）TLB，这样在新进程运行前 TLB 就变成了空的。如果是软件管理 TLB 的系统，可以在发生上下文切换时，通过一条显式（特权）指令来完成。如果是硬件管理 TLB，则可以在页表基址寄存器内容发生变化时清空 TLB（注意，在上下文切换时，操作系统必须改变页表基址寄存器（PTBR）的值）。不论哪种情况，清空操作都是把全部有效位（valid）置为 0，本质上清空了 TLB。

上下文切换的时候清空 TLB，这是一个可行的解决方案，进程不会再读到错误的地址映射。但是，有一定开销：每次进程运行，当它访问数据和代码页时，都会触发 TLB 未命中。如果操作系统频繁地切换进程，这种开销会很高。

为了减少这种开销，一些系统增加了硬件支持，实现跨上下文切换的 TLB 共享。比如有的系统在 TLB 中添加了一个地址空间标识符（Address Space Identifier，ASID）。可以把 ASID 看作是进程标识符（Process Identifier，PID），但通常比 PID 位数少（PID 一般 32 位，ASID 一般是 8 位）。

如果仍以上面的 TLB 为例，加上 ASID，很清楚不同进程可以共享 TLB 了：只要 ASID 字段来区分原来无法区分的地址映射。表 19.2 展示了添加 ASID 字段后的 TLB。

表 19.2 添加 ASID 字段后的 TLB

VPN	PFN	有效位	保护位	ASID
10	100	1	rwx	1
—	—	0	—	—
10	170	1	rwx	2
—	—	0	—	—

因此，有了地址空间标识符，TLB 可以同时缓存不同进程的地址空间映射，没有任何冲突。当然，硬件也需要知道当前是哪个进程正在运行，以便进行地址转换，因此操作系统在上下文切换时，必须将某个特权寄存器设置为当前进程的 ASID。

补充一下，你可能想到了另一种情况，TLB 中某两项非常相似。在表 19.3 中，属于两个不同进程的两项，将两个不同的 VPN 指向了相同的物理页。

表 19.3 包含相似两项的 TLB

VPN	PFN	有效位	保护位	ASID
10	101	1	r-x	1
—	—	0	—	—
50	101	1	r-x	2
—	—	0	—	—

如果两个进程共享同一物理页（例如代码段的页），就可能出现这种情况。在上面的例子中，进程 P1 和进程 P2 共享 101 号物理页，但是 P1 将自己的 10 号虚拟页映射到该物理页，而 P2 将自己的 50 号虚拟页映射到该物理页。共享代码页（以二进制或共享库的方式）是有用的，因为它减少了物理页的使用，从而减少了内存开销。

19.6 TLB 替换策略

TLB 和其他缓存一样，还有一个问题要考虑，即缓存替换（cache replacement）。具体来说，向 TLB 中插入新项时，会替换（replace）一个旧项，这样问题就来了：应该替换那一个？

关键问题：如何设计 TLB 替换策略

在向 TLB 添加新项时，应该替换哪个旧项？目标当然是减小 TLB 未命中率（或提高命中率），从而改进性能。

在讨论页换出到磁盘的问题时，我们将详细研究这样的策略。这里我们先简单指出几个典型的策略。一种常见的策略是替换最近最少使用（least-recently-used，LRU）的项。LRU 尝试利用内存引用流中的局部性，假定最近没有用过的项，可能是好的换出候选项。另一种典型策略就是随机（random）策略，即随机选择一项换出去。这种策略很简单，并且可以避免一种极端情况。例如，一个程序循环访问 n+1 个页，但 TLB 大小只能存放 n 个页。这时之前看似“合理”的 LRU 策略就会表现得不可理喻，因为每次访问内存都会触发 TLB 未命中，而随机策略在这种情况下就好很多。

19.7 实际系统的 TLB 表项

最后，我们简单看一下真实的 TLB。这个例子来自 MIPS R4000[H93]，它是一种现代的系统，采用软件管理 TLB。图 19.4 展示了稍微简化的 MIPS TLB 项。

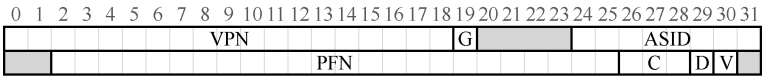


图 19.4 MIPS 的 TLB 项

MIPS R4000 支持 32 位的地址空间，页大小为 4KB。所以在典型的虚拟地址中，预期会看到 20 位的 VPN 和 12 位的偏移量。但是，你可以在 TLB 中看到，只有 19 位的 VPN。事实上，用户地址只占地址空间的一半（剩下的留给内核），所以只需要 19 位的 VPN。VPN 转换成最大 24 位的物理帧号（PFN），因此可以支持最多有 64GB 物理内存（2^{24} 个 4KB 内存页）的系统。

MIPS TLB 还有一些有趣的标识位。比如全局位（Global，G），用来指示这个页是不是所有进程全局共享的。因此，如果全局位置为 1，就会忽略 ASID。我们也看到了 8 位的 ASID，

操作系统用它来区分进程空间（像上面介绍的一样）。这里有一个问题：如果正在运行的进程数超过 256（2^8）个怎么办？最后，我们看到 3 个一致性位（Coherence，C），决定硬件如何缓存该页（其中一位超出了本书的范围）；脏位（dirty），表示该页是否被写入新数据（后面会介绍用法）；有效位（valid），告诉硬件该项的地址映射是否有效。还有没在图 19.4 中展示的页掩码（page mask）字段，用来支持不同的页大小。后面会介绍，为什么更大的页可能有用。最后，64 位中有一些未使用（图 19.4 中灰色部分）。

MIPS 的 TLB 通常有 32 项或 64 项，大多数提供给用户进程使用，也有一小部分留给操作系统使用。操作系统可以设置一个被监听的寄存器，告诉硬件需要为自己预留多少 TLB 槽。这些保留的转换映射，被操作系统用于关键时候它要使用的代码和数据，在这些时候，TLB 未命中可能会导致问题（例如，在 TLB 未命中处理程序中）。

由于 MIPS 的 TLB 是软件管理的，所以系统需要提供一些更新 TLB 的指令。MIPS 提供了 4 个这样的指令：TLBP，用来查找指定的转换映射是否在 TLB 中；TLBR，用来将 TLB 中的内容读取到指定寄存器中；TLBWI，用来替换指定的 TLB 项；TLBWR，用来随机替换一个 TLB 项。操作系统可以用这些指令管理 TLB 的内容。当然这些指令是特权指令，这很关键。如果用户程序可以任意修改 TLB 的内容，你可以想象一下会发生什么可怕的事情。

提示：RAM 不总是 RAM（Culler 定律）

随机存取存储器（Random-Access Memory，RAM）暗示你访问 RAM 的任意部分都一样快。虽然一般这样想 RAM 没错，但因为 TLB 这样的硬件/操作系统功能，访问某些内存页的开销较大，尤其是没有被 TLB 缓存的页。因此，最好记住这个实现的窍门：RAM 不总是 RAM。有时候随机访问地址空间，尤其是 TLB 没有缓存的页，可能导致严重的性能损失。因为我的一位导师 David Culler 过去常常指出 TLB 是许多性能问题的源头，所以我们以他来命名这个定律：Culler 定律（Culler's Law）。

19.8 小结

我们了解了硬件如何让地址转换更快的方法。通过增加一个小的、芯片内的 TLB 作为地址转换的缓存，大多数内存引用就不用访问内存中的页表了。因此，在大多数情况下，程序的性能就像内存没有虚拟化一样，这是操作系统的杰出成就，当然对现代操作系统中的分页非常必要。

但是，TLB 也不能满足所有的程序需求。具体来说，如果一个程序短时间内访问的页数超过了 TLB 中的页数，就会产生大量的 TLB 未命中，运行速度就会变慢。这种现象被称为超出 TLB 覆盖范围（TLB coverage），这对某些程序可能是相当严重的问题。解决这个问题的一种方案是支持更大的页，把关键数据结构放在程序地址空间的某些区域，这些区域被映射到更大的页，使 TLB 的有效覆盖率增加。对更大页的支持通常被数据库管理系统（Database Management System，DBMS）这样的程序利用，它们的数据结构比较大，而且是随机访问。

另一个 TLB 问题值得一提：访问 TLB 很容易成为 CPU 流水线的瓶颈，尤其是有所谓

的物理地址索引缓存（physically-indexed cache）。有了这种缓存，地址转换必须发生在访问该缓存之前，这会让操作变慢。为了解决这个潜在的问题，人们研究了各种巧妙的方法，用虚拟地址直接访问缓存，从而在缓存命中时避免昂贵的地址转换步骤。像这种虚拟地址索引缓存（virtually-indexed cache）解决了一些性能问题，但也为硬件设计带来了新问题。更多细节请参考 Wiggins 的调查[W03]。

参考资料

[BC91]“Performance from Architecture: Comparing a RISC and a CISC with Similar Hardware Organization”
D. Bhandarkar and Douglas W. Clark
Communications of the ACM, September 1991
关于 RISC 和 CISC 的一篇很好的、公平的比较性的文章。本质上，在类似的硬件上，RISC 的性能是 CISC 的 3 倍。

[CM00]“The evolution of RISC technology at IBM”John Cocke and V. Markstein
IBM Journal of Research and Development, 44:1/2
IBM 801 的概念和工作总结，许多人认为它是第一款真正的 RISC 微处理器。

[C95]“The Core of the Black Canyon Computer Corporation”John Couleur
IEEE Annals of History of Computing, 17:4, 1995
在这个引人入胜的计算历史讲义中，Couleur 谈到了他在 1964 年为通用电气公司工作时如何发明了 TLB，以及与麻省理工学院的 MAC 项目人员之间偶然而幸运的合作。

[CG68]“Shared-access Data Processing System”John F. Couleur and Edward L. Glaser
Patent 3412382, November 1968
包含用关联存储器存储地址转换的想法的专利。据 Couleur 说，这个想法产生于 1964 年。

[CP78]“The architecture of the IBM System/370”
R.P. Case and A. Padegs
Communications of the ACM. 21:1, 73-96, January 1978
也许是第一篇使用术语“地址转换旁路缓冲存储器（translation lookaside buffer）”的文章。 这个名字来源于缓存的历史名称，即旁路缓冲存储器（lookaside buffer），在曼彻斯特大学开发 Atlas 系统的人这样叫它。地址转换缓存因此成为地址转换旁路缓冲存储器。尽管术语“旁路缓冲存储器”不再流行，但 TLB 似乎仍在持续使用，其原因不明。

[H93]“MIPS R4000 Microprocessor User’s Manual”. Joe Heinrich, Prentice-Hall, June 1993

[HP06]“Computer Architecture: A Quantitative Approach” John Hennessy and David Patterson
Morgan-Kaufmann, 2006
一本关于计算机架构的好书。我们对经典的第 1 版特别有感情。

[I09]“Intel 64 and IA-32 Architectures Software Developer’s Manuals”Intel, 2009
Available.
尤其要注意《卷 3A：系统编程指南第 1 部分》和《卷 3B：系统编程指南第 2 部分》。

[PS81]“RISC-I: A Reduced Instruction Set VLSI Computer”
D.A．Patterson and C.H. Sequin ISCA ’81, Minneapolis, May 1981
这篇文章介绍了 RISC 这个术语，开启了为性能而简化计算机芯片的研究狂潮。

[SB92]“CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking”
Rafael H. Saavedra-Barrera
EECS Department, University of California, Berkeley Technical Report No. UCB/CSD-92-684, February 1992
一篇卓越的论文，探讨将应用的执行时间分解为组成部分，知道每个部分的成本，从而预测应用的执行时间。也许这项工作最有趣的部分是衡量缓存层次结构细节的工具（在第 5 章中介绍）。一定要看看其中的精彩图表。

[W03]“A Survey on the Interaction Between Caching, Translation and Protection”Adam Wiggins
University of New South Wales TR UNSW-CSE-TR-0321, August, 2003
关于 TLB 如何与 CPU 管道的其他部分（即硬件缓存）进行交互的一次很好的调查。

[WG00]“The SPARC Architecture Manual: Version 9”David L. Weaver and Tom Germond, September 2000
SPARC International, San Jose, California

作业（测量）

本次作业要测算一下 TLB 的容量和访问 TLB 的开销。这个想法参考了 Saavedra-Barrera 的工作[SB92]，他用设计了一个简单而漂亮的用户级程序，来测算缓存层级结构的方方面面。更多细节请阅读他的论文。

基本原理就是访问一个跨多个内存页的大尺寸数据结构（例如数组），然后统计访问时间。例如，假设一个机器的 TLB 大小为 4（这很小，但对这个讨论有用）。如果写一个程序访问 4 个或更少的页，每次访问都会命中 TLB，因此相对较快。但是，如果在一个循环里反复访问 5 个或者更多的页，每次访问的开销就会突然跃升，因为发生 TLB 未命中。

循环遍历数组一次的基本代码应该像这样：

```
int jump = PAGESIZE / sizeof(int);
for (i = 0; i < NUMPAGES * jump; i += jump) {
    a[i] += 1;
}
```

在这个循环中，数组 a 中每页的一个整数被更新，直到 NUMPAGES 指定的页数。通过对这个循环反复执行计时（比如，在外层循环中执行几亿次这个循环，或者运行几秒钟所需的次数），就可以计算出平均每次访问所用的时间。随着 NUMPAGES 的增加，寻找开销

的跃升，可以大致确定第一级 TLB 的大小，确定是否存在第二级 TLB（如果存在，确定它的大小），总体上很好地理解 TLB 命中和未命中对于性能的影响。

图 19.5 是一张示意图。

从图 19.5 中可以看出，如果只访问少数页（8 或更少），平均访问时间大约是 5ns。如果访问 16 页或更多，每次访问时间突然跃升到 20ns。最后一次开销跃升发生在 1024 页时，这时每次访问大约要 70ns。通过这些数据，我们可以总结出这是一个二级的 TLB，第一级较小（大约能存放 8～16 项），第二级较大，但较慢（大约能存放 512 项）。第一级 TLB 的命中和完全未命中的总体差距非常大，大约有 14 倍。TLB 的性能很重要！

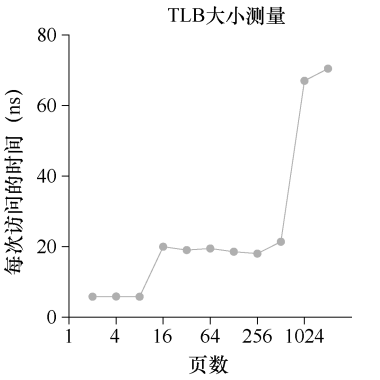


图 19.5　发现 TLB 大小和未命中开销

问题

1．为了计时，可能需要一个计时器，例如 gettimeofday()提供的。这种计时器的精度如何？操作要花多少时间，才能让你对它精确计时？（这有助于确定需要循环多少次，反复访问内存页，才能对它成功计时。）

2．写一个程序，命名为 tlb.c，大体测算一下每个页的平均访问时间。程序的输入参数有：页的数目和尝试的次数。

3．用你喜欢的脚本语言（csh、Python 等）写一段脚本来运行这个程序，当访问页面从 1 增长到几千，也许每次迭代都乘 2。在不同的机器上运行这段脚本，同时收集相应数据。需要试多少次才能获得可信的测量结果？

4．接下来，将结果绘图，类似于上图。可以用 ploticus 这样的好工具画图。可视化使数据更容易理解，你认为是什么原因？

5．要注意编译器优化带来的影响。编译器做各种聪明的事情，包括优化掉循环，如果循环中增加的变量后续没有使用。如何确保编译器不优化掉你写的 TLB 大小测算程序的主循环？

6．还有一个需要注意的地方，今天的计算机系统大多有多个 CPU，每个 CPU 当然有自己的 TLB 结构。为了得到准确的测量数据，我们需要只在一个 CPU 上运行程序，避免调度器把进程从一个 CPU 调度到另一个去运行。如何做到？（提示：在 Google 上搜索“pinning a thread”相关的信息）如果没有这样做，代码从一个 CPU 移到了另一个，会发生什么情况？

7．另一个可能发生的问题与初始化有关。如果在访问数组 a 之前没有初始化，第一次访问将非常耗时，由于初始访问开销，比如要求置 0。这会影响你的代码及其计时吗？如何抵消这些潜在的开销？

第 20 章　分页：较小的表

我们现在来解决分页引入的第二个问题：页表太大，因此消耗的内存太多。让我们从线性页表开始。你可能会记得[①]，线性页表变得相当大。假设一个 32 位地址空间（232 字节），4KB（212 字节）的页和一个 4 字节的页表项。一个地址空间中大约有一百万个虚拟页面（232/212）。乘以页表项的大小，你会发现页表大小为 4MB。回想一下：通常系统中的每个进程都有一个页表！有一百个活动进程（在现代系统中并不罕见），就要为页表分配数百兆的内存！因此，要寻找一些技术来减轻这种沉重的负担。有很多方法，所以我们开始吧。但先看我们的关键问题：

关键问题：如何让页表更小？

简单的基于数组的页表（通常称为线性页表）太大，在典型系统上占用太多内存。如何让页表更小？关键的思路是什么？由于这些新的数据结构，会出现什么效率影响？

20.1　简单的解决方案：更大的页

可以用一种简单的方法减小页表大小：使用更大的页。再以 32 位地址空间为例，但这次假设用 16KB 的页。因此，会有 18 位的 VPN 加上 14 位的偏移量。假设每个页表项（4 字节）的大小相同，现在线性页表中有 218 个项，因此每个页表的总大小为 1MB，页表缩到四分之一。

补充：多种页大小

另外请注意，许多体系结构（例如 MIPS、SPARC、x86-64）现在都支持多种页大小。通常使用一个小的（4KB 或 8KB）页大小。但是，如果一个“聪明的”应用程序请求它，则可以为地址空间的特定部分使用一个大型页（例如，大小为 4MB），从而让这些应用程序可以将常用的（大型的）数据结构放入这样的空间，同时只占用一个 TLB 项。这种类型的大页在数据库管理系统和其他高端商业应用程序中很常见。然而，多种页面大小的主要原因并不是为了节省页表空间。这是为了减少 TLB 的压力，让程序能够访问更多的地址空间而不会遭受太多的 TLB 未命中之苦。然而，正如研究人员已经说明[N+02]一样，采用多种页大小，使操作系统虚拟内存管理程序显得更复杂，因此，有时只需向应用程序暴露一个新接口，让它们直接请求大内存页，这样最容易。

① 或者实际上，你可能记不起来了。分页这件事正在失控，不是吗？虽然这样说，但在进入解决方案之前，一定要确保你理解了正在解决的问题。事实上，如果你理解了问题，通常可以自己推导出解决方案。在这里，问题应该很清楚：简单的线性（基于数组的）页表太大了。

然而，这种方法的主要问题在于，大内存页会导致每页内的浪费，这被称为内部碎片（internal fragmentation）问题（因为浪费在分配单元内部）。因此，结果是应用程序会分配页，但只用每页的一小部分，而内存很快就会充满这些过大的页。因此，大多数系统在常见的情况下使用相对较小的页大小：4KB（如 x86）或 8KB（如 SPARCv9）。问题不会如此简单地解决。

20.2 混合方法：分页和分段

在生活中，每当有两种合理但不同的方法时，你应该总是研究两者的结合，看看能否两全其美。我们称这种组合为杂合（hybrid）。例如，为什么只吃巧克力或简单的花生酱，而不是将两者结合起来，就像可爱的花生酱巧克力杯[M28]？

多年前，Multics 的创造者（特别是 Jack Dennis）在构建 Multics 虚拟内存系统时，偶然发现了这样的想法[M07]。具体来说，Dennis 想到将分页和分段相结合，以减少页表的内存开销。更仔细地看看典型的线性页表，就可以理解为什么这可能有用。假设我们有一个地址空间，其中堆和栈的使用部分很小。例如，我们使用一个 16KB 的小地址空间和 1KB 的页（见图 20.1）。该地址空间的页表如表 20.1 所示。

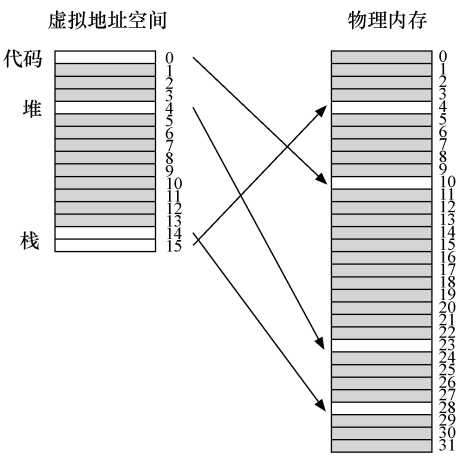


图 20.1 1KB 的页和 16KB 的地址空间

这个例子假定单个代码页（VPN 0）映射到物理页 10，单个堆页（VPN 4）映射到物理页 23，以及地址空间另一端的两个栈页（VPN 14 和 15）被分别映射到物理页 28 和 4。从图 20.1 中可以看到，大部分页表都没有使用，充满了无效的（invalid）项。真是太浪费了！这是一个微小的 16KB 地址空间。想象一下 32 位地址空间的页表和所有潜在的浪费空间！真的，不要想象这样的事情，太可怕了。

表 20.1 16KB 地址空间的页表

PFN	valid	prot	present	dirty
10	1	r-x	1	0
—	0	—	—	—
—	0	—	—	—
—	0	—	—	—
23	1	rw-	1	1
—	0	—	—	—
—	0	—	—	—
—	0	—	—	—
—	0	—	—	—
—	0	—	—	—

续表

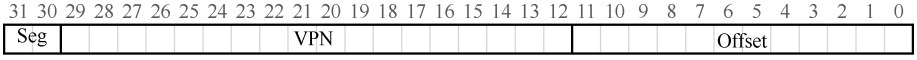
PFN	valid	prot	present	dirty
—	0	—	—	—
—	0	—	—	—
—	0	—	—	—
—	0	—	—	—
28	1	rw-	1	1
4	1	rw-	1	1

因此，我们的杂合方法不是为进程的整个地址空间提供单个页表，而是为每个逻辑分段提供一个。在这个例子中，我们可能有 3 个页表，地址空间的代码、堆和栈部分各有一个。

现在，回忆在分段中，有一个基址（base）寄存器，告诉我们每个段在物理内存中的位置，还有一个界限（bound）或限制（limit）寄存器，告诉我们该段的大小。在杂合方案中，我们仍然在 MMU 中拥有这些结构。在这里，我们使用基址不是指向段本身，而是保存该段的页表的物理地址。界限寄存器用于指示页表的结尾（即它有多少有效页）。

我们通过一个简单的例子来澄清。假设 32 位虚拟地址空间包含 4KB 页面，并且地址空间分为 4 个段。在这个例子中，我们只使用 3 个段：一个用于代码，另一个用于堆，还有一个用于栈。

要确定地址引用哪个段，我们会用地址空间的前两位。假设 00 是未使用的段，01 是代码段，10 是堆段，11 是栈段。因此，虚拟地址如下所示：



在硬件中，假设有 3 个基本/界限对，代码、堆和栈各一个。当进程正在运行时，每个段的基址寄存器都包含该段的线性页表的物理地址。因此，系统中的每个进程现在都有 3 个与其关联的页表。在上下文切换时，必须更改这些寄存器，以反映新运行进程的页表的位置。

在 TLB 未命中时（假设硬件管理的 TLB，即硬件负责处理 TLB 未命中），硬件使用分段位（SN）来确定要用哪个基址和界限对。然后硬件将其中的物理地址与 VPN 结合起来，形成页表项（PTE）的地址：

```
SN           = (VirtualAddress & SEG_MASK) >> SN_SHIFT
VPN          = (VirtualAddress & VPN_MASK) >> VPN_SHIFT
AddressOfPTE = Base[SN] + (VPN * sizeof(PTE))
```

这段代码应该看起来很熟悉，它与我们之前在线性页表中看到的几乎完全相同。当然，唯一的区别是使用 3 个段基址寄存器中的一个，而不是单个页表基址寄存器。

杂合方案的关键区别在于，每个分段都有界限寄存器，每个界限寄存器保存了段中最大有效页的值。例如，如果代码段使用它的前 3 个页（0、1 和 2），则代码段页表将只有 3 个项分配给它，并且界限寄存器将被设置为 3。内存访问超出段的末尾将产生一个异常，并可能导致进程终止。以这种方式，与线性页表相比，杂合方法实现了显著的内存节省。栈和堆之间未分配的页不再占用页表中的空间（仅将其标记为无效）。

提示：使用杂合

当你有两个看似相反的好主意时，你应该总是看到你是否可以将它们组合成一个能够实现两全其美的杂合体（hybrid）。例如，杂交玉米物种已知比任何天然存在的物种更强壮。当然，并非所有的杂合都是好主意，请参阅 Zeedonk（或 Zonkey），它是斑马和驴的杂交。如果你不相信这样的生物存在，就查一下，你会大吃一惊。

但是，你可能会注意到，这种方法并非没有问题。首先，它仍然要求使用分段。正如我们讨论的那样，分段并不像我们需要的那样灵活，因为它假定地址空间有一定的使用模式。例如，如果有一个大而稀疏的堆，仍然可能导致大量的页表浪费。其次，这种杂合导致外部碎片再次出现。尽管大部分内存是以页面大小单位管理的，但页表现在可以是任意大小（是 PTE 的倍数）。因此，在内存中为它们寻找自由空间更为复杂。出于这些原因，人们继续寻找更好的方式来实现更小的页表。

20.3 多级页表

另一种方法并不依赖于分段，但也试图解决相同的问题：如何去掉页表中的所有无效区域，而不是将它们全部保留在内存中？我们将这种方法称为多级页表（multi-level page table），因为它将线性页表变成了类似树的东西。这种方法非常有效，许多现代系统都用它（例如 x86 [BOH10]）。我们现在详细描述这种方法。

多级页表的基本思想很简单。首先，将页表分成页大小的单元。然后，如果整页的页表项（PTE）无效，就完全不分配该页的页表。为了追踪页表的页是否有效（以及如果有效，它在内存中的位置），使用了名为页目录（page directory）的新结构。页目录因此可以告诉你页表的页在哪里，或者页表的整个页不包含有效页。

图 20.2 展示了一个例子。图的左边是经典的线性页表。即使地址空间的大部分中间区域无效，我们仍然需要为这些区域分配页表空间（即页表的中间两页）。右侧是一个多级页表。页目录仅将页表的两页标记为有效（第一个和最后一个）；因此，页表的这两页就驻留在内存中。因此，你可以形象地看到多级页表的工作方式：它只是让线性页表的一部分消失（释放这些帧用于其他用途），并用页目录来记录页表的哪些页被分配。

在一个简单的两级页表中，页目录为每页页表包含了一项。它由多个页目录项（Page Directory Entries，PDE）组成。PDE（至少）拥有有效位（valid bit）和页帧号（page frame number，PFN），类似于 PTE。但是，正如上面所暗示的，这个有效位的含义稍有不同：如果 PDE 项是有效的，则意味着该项指向的页表（通过 PFN）中至少有一页是有效的，即在该 PDE 所指向的页中，至少一个 PTE，其有效位被设置为 1。如果 PDE 项无效（即等于零），则 PDE 的其余部分没有定义。

与我们至今为止看到的方法相比，多级页表有一些明显的优势。首先，也许最明显的是，多级页表分配的页表空间，与你正在使用的地址空间内存量成比例。因此它通常很紧凑，并且支持稀疏的地址空间。

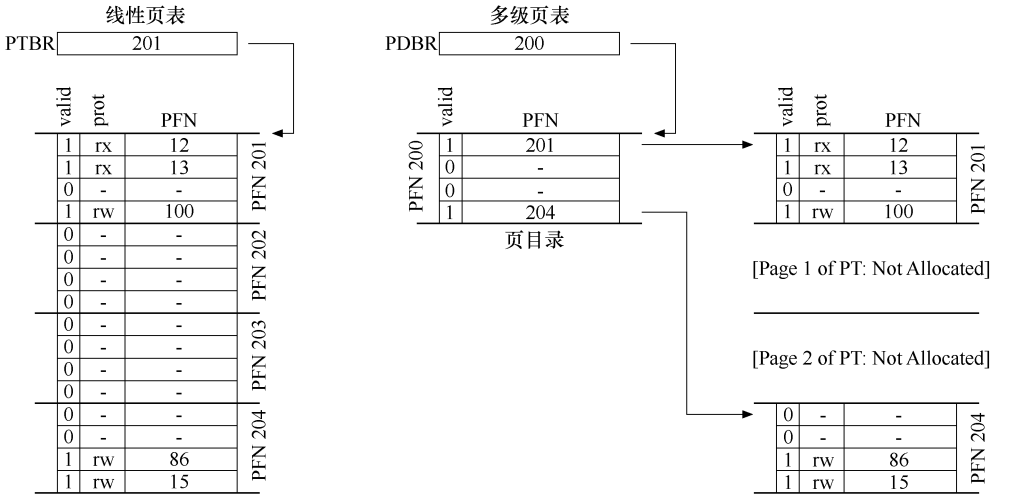


图 20.2 线性（左）和多级（右）页表

其次，如果仔细构建，页表的每个部分都可以整齐地放入一页中，从而更容易管理内存。操作系统可以在需要分配或增长页表时简单地获取下一个空闲页。将它与一个简单的（非分页）线性页表相比[①]，后者仅是按 VPN 索引的 PTE 数组。用这样的结构，整个线性页表必须连续驻留在物理内存中。对于一个大的页表（比如 4MB），找到如此大量的、未使用的连续空闲物理内存，可能是一个相当大的挑战。有了多级结构，我们增加了一个间接层（level of indirection），使用了页目录，它指向页表的各个部分。这种间接方式，让我们能够将页表页放在物理内存的任何地方。

提示：理解时空折中

在构建数据结构时，应始终考虑时间和空间的折中（time-space trade-off）。通常，如果你希望更快地访问特定的数据结构，就必须为该结构付出空间的代价。

应该指出，多级页表是有成本的。在 TLB 未命中时，需要从内存加载两次，才能从页表中获取正确的地址转换信息（一次用于页目录，另一次用于 PTE 本身），而用线性页表只需要一次加载。因此，多级表是一个时间—空间折中（time-space trade-off）的小例子。我们想要更小的表（并得到了），但不是没代价。尽管在常见情况下（TLB 命中），性能显然是相同的，但 TLB 未命中时，则会因较小的表而导致较高的成本。

另一个明显的缺点是复杂性。无论是硬件还是操作系统来处理页表查找（在 TLB 未命中时），这样做无疑都比简单的线性页表查找更复杂。通常我们愿意增加复杂性以提高性能或降低管理费用。在多级表的情况下，为了节省宝贵的内存，我们使页表查找更加复杂。

详细的多级示例

为了更好地理解多级页表背后的想法，我们来看一个例子。设想一个大小为 16KB 的小地址空间，其中包含 64 个字节的页。因此，我们有一个 14 位的虚拟地址空间，VPN 有 8

① 我们在这里做了一些假设，所有的页表全部驻留在物理内存中（即它们没有交换到磁盘）。 我们很快就会放松这个假设。

位，偏移量有 6 位。即使只有一小部分地址空间正在使用，线性页表也会有 2^8（256）个项。图 20.3 展示了这种地址空间的一个例子。

提示：对复杂性表示怀疑

系统设计者应该谨慎对待让系统增加复杂性。好的系统构建者所做的就是：实现最小复杂性的系统，来完成手上的任务。例如，如果磁盘空间非常大，则不应该设计一个尽可能少使用字节的文件系统。同样，如果处理器速度很快，建议在操作系统中编写一个干净、易于理解的模块，而不是 CPU 优化的、手写汇编的代码。注意过早优化的代码或其他形式的不必要的复杂性。这些方法会让系统难以理解、维护和调试。正如 Antoine de Saint-Exupery 的名言：“完美非无可增，乃不可减。”他没有写的是：“谈论完美易，真正实现难。

在这个例子中，虚拟页 0 和 1 用于代码，虚拟页 4 和 5 用于堆，虚拟页 254 和 255 用于栈。地址空间的其余页未被使用。

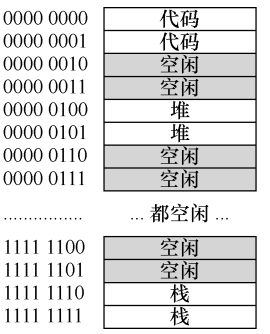
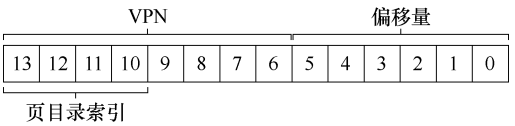


图 20.3 16KB 的地址空间和 64 字节的页

要为这个地址空间构建一个两级页表，我们从完整的线性页表开始，将它分解成页大小的单元。回想一下我们的完整页表（在这个例子中）有 256 个项；假设每个 PTE 的大小是 4 个字节。因此，我们的页大小为 1KB（256×4 字节）。鉴于我们有 64 字节的页，1KB 页表可以分为 16 个 64 字节的页，每个页可以容纳 16 个 PTE。

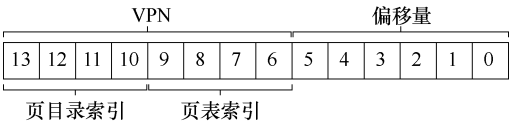
我们现在需要了解：如何获取 VPN，并用它来首先索引到页目录中，然后再索引到页表的页中。请记住，每个都是一组项。因此，我们需要弄清楚，如何为每个 VPN 构建索引。

我们首先索引到页目录。这个例子中的页表很小：256 个项，分布在 16 个页上。页目录需要为页表的每页提供一个项。因此，它有 16 个项。结果，我们需要 4 位 VPN 来索引目录。我们使用 VPN 的前 4 位，如下所示：



一旦从 VPN 中提取了页目录索引（简称 PDIndex），我们就可以通过简单的计算来找到页目录项（PDE）的地址：PDEAddr = PageDirBase +（PDIndex×sizeof（PDE））。这就得到了页目录，现在我们来看它，在地址转换上取得进一步进展。

如果页目录项标记为无效，则我们知道访问无效，从而引发异常。但是，如果 PDE 有效，我们还有更多工作要做。具体来说，我们现在必须从页目录项指向的页表的页中获取页表项（PTE）。要找到这个 PTE，我们必须使用 VPN 的剩余位索引到页表的部分：



这个页表索引（Page-Table Index，PTIndex）可以用来索引页表本身，给出 PTE 的地址：

```
PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
```

请注意，从页目录项获得的页帧号（PFN）必须左移到位，然后再与页表索引组合，才能形成 PTE 的地址。

为了确定这一切是否合理，我们现在代入一个包含一些实际值的多级页表，并转换一个虚拟地址。让我们从这个例子的页目录开始（见表 20.2 的左侧）。

在该表中，可以看到每个页目录项（PDE）都描述了有关地址空间页表的一些内容。在这个例子中，地址空间里有两个有效区域（在开始和结束处），以及一些无效的映射。

在物理页 100（页表的第 0 页的物理帧号）中，我们有 1 页，包含 16 个页表项，记录了地址空间中的前 16 个 VPN。请参见表 20.2（中间部分）了解这部分页表的内容。

表 20.2 页目录和页表

Page Directory		Page of PT (@PFN:100)			Page of PT (@PFN:101)		
PFN	valid	PFN	valid	prot	PFN	valid	prot
100	1	10	1	r-x	—	0	—
—	0	23	1	r-x	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	80	1	rw-	—	0	—
—	0	59	1	rw-	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	55	1	rw-
101	1	—	0	—	45	1	rw-

页表的这一页包含前 16 个 VPN 的映射。在我们的例子中，VPN 0 和 1 是有效的（代码段），4 和 5（堆）也是。因此，该表有每个页的映射信息。其余项标记为无效。

页表的另一个有效页在 PFN 101 中。该页包含地址空间的最后 16 个 VPN 的映射。具体见表 20.2（右侧）。

在这个例子中，VPN 254 和 255（栈）包含有效的映射。希望从这个例子中可以看出，多级索引结构可以节省多少空间。在这个例子中，我们不是为一个线性页表分配完整的 16 页，而是分配 3 页：一个用于页目录，两个用于页表的具有有效映射的块。大型（32 位或 64 位）地址空间的节省显然要大得多。

最后，让我们用这些信息来进行地址转换。这里是一个地址，指向 VPN 254 的第 0 个

字节：0x3F80，或二进制的 11 1111 1000 0000。

回想一下，我们将使用 VPN 的前 4 位来索引页目录。因此，1111 会从上面的页目录中选择最后一个（第 15 个，如果你从第 0 个开始）。这就指向了位于地址 101 的页表的有效页。然后，我们使用 VPN 的下 4 位（1110）来索引页表的那一页并找到所需的 PTE。1110 是页面中的倒数第二（第 14 个）条，并告诉我们虚拟地址空间的页 254 映射到物理页 55。通过连接 PFN = 55（或十六进制 0x37）和 offset = 000000，可以形成我们想要的物理地址，并向内存系统发出请求：PhysAddr =（PTE.PFN << SHIFT）+ offset = 00 1101 1100 0000 = 0x0DC0。

你现在应该知道如何构建两级页表，利用指向页表页的页目录。但遗憾的是，我们的工作还没有完成。我们现在要讨论，有时两个页级别是不够的！

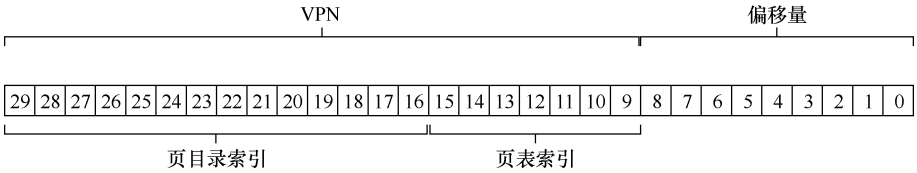
超过两级

在至今为止的例子中，我们假定多级页表只有两个级别：一个页目录和几页页表。在某些情况下，更深的树是可能的（并且确实需要）。

让我们举一个简单的例子，用它来说明为什么更深层次的多级页表可能有用。在这个例子中，假设我们有一个 30 位的虚拟地址空间和一个小的（512 字节）页。因此我们的虚拟地址有一个 21 位的虚拟页号和一个 9 位偏移量。

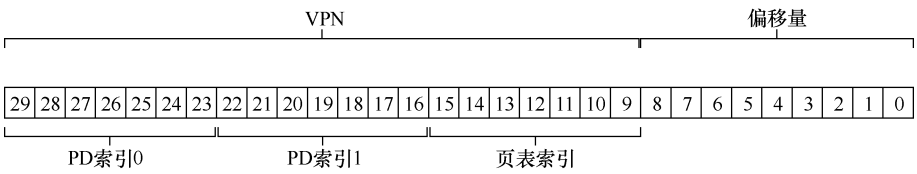
请记住我们构建多级页表的目标：使页表的每一部分都能放入一个页。到目前为止，我们只考虑了页表本身。但是，如果页目录太大，该怎么办？

要确定多级表中需要多少级别才能使页表的所有部分都能放入一页，首先要确定多少页表项可以放入一页。鉴于页大小为 512 字节，并且假设 PTE 大小为 4 字节，你应该看到，可以在单个页上放入 128 个 PTE。当我们索引页表时，我们可以得出结论，我们需要 VPN 的最低有效位 7 位（$\log_2 128$）作为索引：



在上面你还可能注意到，多少位留给了（大）页目录：14。如果我们的页目录有 2^{14} 个项，那么它不是一个页，而是 128 个，因此我们让多级页表的每一个部分放入一页目标失败了。

为了解决这个问题，我们为树再加一层，将页目录本身拆成多个页，然后在其上添加另一个页目录，指向页目录的页。我们可以按如下方式分割虚拟地址：



现在，当索引上层页目录时，我们使用虚拟地址的最高几位（图中的 PD 索引 0）。该索引用于从顶级页目录中获取页目录项。如果有效，则通过组合来自顶级 PDE 的物理帧号和 VPN 的下一部分（PD 索引 1）来查阅页目录的第二级。最后，如果有效，则可以通过使

用与第二级 PDE 的地址组合的页表索引来形成 PTE 地址。这会有很多工作。所有这些只是为了在多级页表中查找某些东西。

地址转换过程：记住 TLB

为了总结使用两级页表的地址转换的整个过程，我们再次以算法形式展示控制流（见图 20.4）。该图显示了每个内存引用在硬件中发生的情况（假设硬件管理的 TLB）。

从图中可以看到，在任何复杂的多级页表访问发生之前，硬件首先检查 TLB。在命中时，物理地址直接形成，而不像之前一样访问页表。只有在 TLB 未命中时，硬件才需要执行完整的多级查找。在这条路径上，可以看到传统的两级页表的成本：两次额外的内存访问来查找有效的转换映射。

```
1    VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2    (Success, TlbEntry) = TLB_Lookup(VPN)
3    if (Success == True)    // TLB Hit
4        if (CanAccess(TlbEntry.ProtectBits) == True)
5            Offset   = VirtualAddress & OFFSET_MASK
6            PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7            Register = AccessMemory(PhysAddr)
8        else
9            RaiseException(PROTECTION_FAULT)
10   else                    // TLB Miss
11       // first, get page directory entry
12       PDIndex = (VPN & PD_MASK) >> PD_SHIFT
13       PDEAddr = PDBR + (PDIndex * sizeof(PDE))
14       PDE     = AccessMemory(PDEAddr)
15       if (PDE.Valid == False)
16           RaiseException(SEGMENTATION_FAULT)
17       else
18           // PDE is valid: now fetch PTE from page table
19           PTIndex = (VPN & PT_MASK) >> PT_SHIFT
20           PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
21           PTE     = AccessMemory(PTEAddr)
22           if (PTE.Valid == False)
23               RaiseException(SEGMENTATION_FAULT)
24           else if (CanAccess(PTE.ProtectBits) == False)
25               RaiseException(PROTECTION_FAULT)
26           else
27               TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
28               RetryInstruction()
```

图 20.4 多级页表控制流

20.4 反向页表

在反向页表（inverted page table）中，可以看到页表世界中更极端的空间节省。在这里，我们保留了一个页表，其中的项代表系统的每个物理页，而不是有许多页表（系统的每个进