里，操作系统将地址空间的虚拟页 0 放在物理页帧 3，虚拟页 1 放在物理页帧 7，虚拟页 2 放在物理页帧 5，虚拟页 3 放在物理页帧 2。页帧 1、4、6 目前是空闲的。

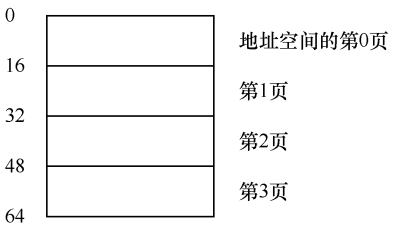


图 18.1 一个简单的 64 字节地址空间

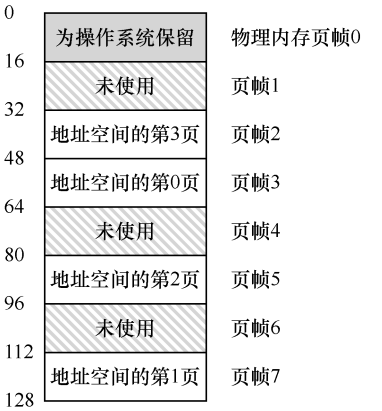


图 18.2 64 字节的地址空间在 128 字节的物理内存中

为了记录地址空间的每个虚拟页放在物理内存中的位置，操作系统通常为每个进程保存一个数据结构，称为页表（page table）。页表的主要作用是为地址空间的每个虚拟页面保存地址转换（address translation），从而让我们知道每个页在物理内存中的位置。对于我们的简单示例（见图 18.2），页表因此具有以下 4 个条目：（虚拟页 0→物理帧 3）、（VP 1→PF 7）、（VP 2→PF 5）和（VP 3→PF 2）。

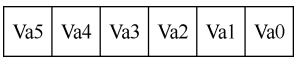
重要的是要记住，这个页表是一个每进程的数据结构（我们讨论的大多数页表结构都是每进程的数据结构，我们将接触的一个例外是倒排页表，inverted page table）。如果在上面的示例中运行另一个进程，操作系统将不得不为它管理不同的页表，因为它的虚拟页显然映射到不同的物理页面（除了共享之外）。

现在，我们了解了足够的信息，可以完成一个地址转换的例子。设想拥有这个小地址空间（64 字节）的进程正在访问内存：

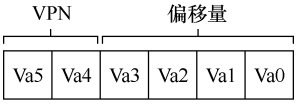
```
movl <virtual address>, %eax
```

具体来说，注意从地址<virtual address>到寄存器 eax 的数据显式加载（因此忽略之前肯定会发生的指令获取）。

为了转换（translate）该过程生成的虚拟地址，我们必须首先将它分成两个组件：虚拟页面号（virtual page number，VPN）和页内的偏移量（offset）。对于这个例子，因为进程的虚拟地址空间是 64 字节，我们的虚拟地址总共需要 6 位（$2^6 = 64$）。因此，虚拟地址可以表示如下：



在该图中，Va5 是虚拟地址的最高位，Va0 是最低位。因为我们知道页的大小（16 字节），所以可以进一步划分虚拟地址，如下所示：



页面大小为 16 字节，位于 64 字节的地址空间。因此我们需要能够选择 4 个页，地址的前 2 位就是做这件事的。因此，我们有一个 2 位的虚拟页号（VPN）。其余的位告诉我们，感兴趣该页的哪个字节，在这个例子中是 4 位，我们称之为偏移量。

当进程生成虚拟地址时，操作系统和硬件必须协作，将它转换为有意义的物理地址。例如，让我们假设上面的加载是虚拟地址 21：

```
movl 21, %eax
```

将“21”变成二进制形式，是“010101”，因此我们可以检查这个虚拟地址，看看它是如何分解成虚拟页号（VPN）和偏移量的：

因此，虚拟地址“21”在虚拟页“01”（或 1）的第 5 个（“0101”）字节处。通过虚拟页号，我们现在可以检索页表，找到虚拟页 1 所在的物理页面。在上面的页表中，物理帧号（PFN）（有时也称为物理页号，physical page number 或 PPN）是 7（二进制 111）。因此，我们可以通过用 PFN 替换 VPN 来转换此虚拟地址，然后将载入发送给物理内存（见图 18.3）。

请注意，偏移量保持不变（即未翻译），因为偏移量只是告诉我们页面中的哪个字节是我们想要的。我们的最终物理地址是 1110101（十进制 117），正是我们希望加载指令（见图 18.2）获取数据的地方。

有了这个基本概念，我们现在可以询问（希望也可以回答）关于分页的一些基本问题。例如，这些页表在哪里存储？页表的典型内容是什么，表有多大？分页是否会使系统变（得很）慢？这些问题和其他迷人的问题（至少部分）在下文中回答。请继续阅读！

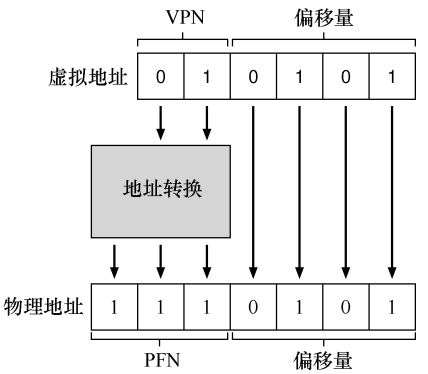


图 18.3 地址转换过程

18.2 页表存在哪里

页表可以变得非常大，比我们之前讨论过的小段表或基址/界限对要大得多。例如，想象一个典型的 32 位地址空间，带有 4KB 的页。这个虚拟地址分成 20 位的 VPN 和 12 位的偏移量（回想一下，1KB 的页面大小需要 10 位，只需增加两位即可达到 4KB）。

一个 20 位的 VPN 意味着，操作系统必须为每个进程管理 2^{20} 个地址转换（大约一百万）。假设每个页表格条目（PTE）需要 4 个字节，来保存物理地址转换和任何其他有用的东西，每个页表就需要巨大的 4MB 内存！这非常大。现在想象一下有 100 个进程在运行：这意味着操作系统会需要 400MB 内存，只是为了所有这些地址转换！即使是现在，机器拥有千兆字节的内存，将它的一大块仅用于地址转换，这似乎有点疯狂，不是吗？我们甚至不敢想 64 位地址空间的页表有多大。那太可怕了，也许把你吓坏了。

由于页表如此之大，我们没有在 MMU 中利用任何特殊的片上硬件，来存储当前正在运行的进程的页表，而是将每个进程的页表存储在内存中。现在让我们假设页表存在于操作系统管理的物理内存中，稍后我们会看到，很多操作系统内存本身都可以虚拟化，因此页表可以存储在操作系统的虚拟内存中（甚至可以交换到磁盘上），但是现在这太令人困惑了，所以我们会忽略它。图 18.4 展示了操作系统内存中的页表，看到其中的一小组地址转换了吗？

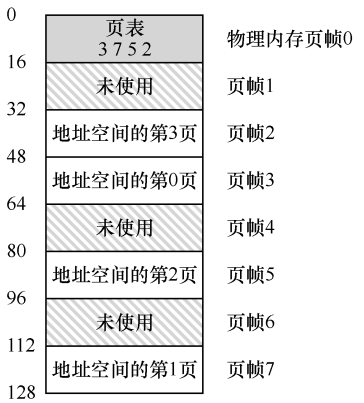


图 18.4 例子：内核物理内存中的页表

18.3 列表中究竟有什么

让我们来谈谈页表的组织。页表就是一种数据结构，用于将虚拟地址（或者实际上，是虚拟页号）映射到物理地址（物理帧号）。因此，任何数据结构都可以采用。最简单的形式称为线性页表（linear page table），就是一个数组。操作系统通过虚拟页号（VPN）检索该数组，并在该索引处查找页表项（PTE），以便找到期望的物理帧号（PFN）。现在，我们将假设采用这个简单的线性结构。在后面的章节中，我们将利用更高级的数据结构来帮助解决一些分页问题。

至于每个 PTE 的内容，我们在其中有许多不同的位，值得有所了解。有效位（valid bit）通常用于指示特定地址转换是否有效。例如，当一个程序开始运行时，它的代码和堆在其地址空间的一端，栈在另一端。所有未使用的中间空间都将被标记为无效（invalid），如果进程尝试访问这种内存，就会陷入操作系统，可能会导致该进程终止。因此，有效位对于支持稀疏地址空间至关重要。通过简单地将地址空间中所有未使用的页面标记为无效，我们不再需要为这些页面分配物理帧，从而节省大量内存。

我们还可能有保护位（protection bit），表明页是否可以读取、写入或执行。同样，以这些位不允许的方式访问页，会陷入操作系统。

还有其他一些重要的部分，但现在我们不会过多讨论。存在位（present bit）表示该页是在物理存储器还是在磁盘上（即它已被换出，swapped out）。当我们研究如何将部分地址空间交换（swap）到磁盘，从而支持大于物理内存的地址空间时，我们将进一步理解这一机制。交换允许操作系统将很少使用的页面移到磁盘，从而释放物理内存。脏位（dirty bit）也很常见，表明页面被带入内存后是否被修改过。

参考位（reference bit，也被称为访问位，accessed bit）有时用于追踪页是否被访问，也用于确定哪些页很受欢迎，因此应该保留在内存中。这些知识在页面替换（page replacement）时非常重要，我们将在随后的章节中详细研究这一主题。

图 18.5 显示了来自 x86 架构的示例页表项[I09]。它包含一个存在位（P），确定是否允许写入该页面的读/写位（R/W） 确定用户模式进程是否可以访问该页面的用户/超级用户位（U/S），有几位（PWT、PCD、PAT 和 G）确定硬件缓存如何为这些页面工作，一个访问位（A）和一个脏位（D），最后是页帧号（PFN）本身。

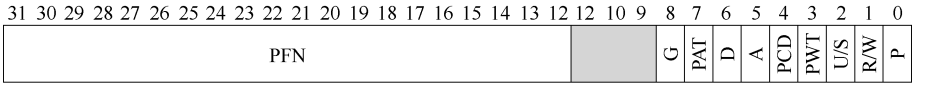


图 18.5 一个 x86 页表项（PTE）

阅读英特尔架构手册[I09]，以获取有关 x86 分页支持的更多详细信息。然而，要事先警告，阅读这些手册时，尽管非常有用（对于在操作系统中编写代码以使用这些页表的用户而言，这些手册当然是必需的），但起初可能很具挑战性。需要一点耐心和强烈的愿望。

18.4 分页：也很慢

内存中的页表，我们已经知道它们可能太大了。事实证明，它们也会让速度变慢。以简单的指令为例：

```
movl 21, %eax
```

同样，我们只看对地址 21 的显式引用，而不关心指令获取。在这个例子中，我们假定硬件为我们执行地址转换。要获取所需数据，系统必须首先将虚拟地址（21）转换为正确的物理地址（117）。因此，在从地址 117 获取数据之前，系统必须首先从进程的页表中提取适当的页表项，执行转换，然后从物理内存中加载数据。

为此，硬件必须知道当前正在运行的进程的页表的位置。现在让我们假设一个页表基址寄存器（page-table base register）包含页表的起始位置的物理地址。为了找到想要的 PTE 的位置，硬件将执行以下功能：

```
VPN     = (VirtualAddress & VPN_MASK) >> SHIFT
PTEAddr = PageTableBaseRegister + (VPN * sizeof(PTE))
```

在我们的例子中，VPN MASK 将被设置为 0x30（十六进制 30，或二进制 110000），它从完整的虚拟地址中挑选出 VPN 位；SHIFT 设置为 4（偏移量的位数），这样我们就可以将 VPN 位向右移动以形成正确的整数虚拟页码。例如，使用虚拟地址 21（010101），掩码将此值转换为 010000，移位将它变成 01，或虚拟页 1，正是我们期望的值。然后，我们使用该值作为页表基址寄存器指向的 PTE 数组的索引。

一旦知道了这个物理地址，硬件就可以从内存中获取 PTE，提取 PFN，并将它与来自虚拟地址的偏移量连接起来，形成所需的物理地址。具体来说，你可以想象 PFN 被 SHIFT 左移，然后与偏移量进行逻辑或运算，以形成最终地址，如图 18.6 所示。

```
offset   = VirtualAddress & OFFSET_MASK
PhysAddr = (PFN << SHIFT) | offset
1    // Extract the VPN from the virtual address
2    VPN = (VirtualAddress & VPN_MASK) >> SHIFT
3
4    // Form the address of the page-table entry (PTE)
5    PTEAddr = PTBR + (VPN * sizeof(PTE))
6
7    // Fetch the PTE
```

```
8    PTE = AccessMemory(PTEAddr)
9
10   // Check if process can access the page
11   if (PTE.Valid == False)
12       RaiseException(SEGMENTATION_FAULT)
13   else if (CanAccess(PTE.ProtectBits) == False)
14       RaiseException(PROTECTION_FAULT)
15   else
16       // Access is OK: form physical address and fetch it
17       offset   = VirtualAddress & OFFSET_MASK
18       PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
19       Register = AccessMemory(PhysAddr)
```

图 18.6 利用分页访问内存

最后，硬件可以从内存中获取所需的数据并将其放入寄存器 eax。程序现在已成功从内存中加载了一个值！

总之，我们现在描述了在每个内存引用上发生的情况的初始协议。基本方法如图 18.6 所示。对于每个内存引用（无论是取指令还是显式加载或存储），分页都需要我们执行一个额外的内存引用，以便首先从页表中获取地址转换。工作量很大！额外的内存引用开销很大，在这种情况下，可能会使进程减慢两倍或更多。

现在你应该可以看到，有两个必须解决的实际问题。如果不仔细设计硬件和软件，页表会导致系统运行速度过慢，并占用太多内存。虽然看起来是内存虚拟化需求的一个很好的解决方案，但这两个关键问题必须先克服。

18.5 内存追踪

在结束之前，我们现在通过一个简单的内存访问示例，来演示使用分页时产生的所有内存访问。我们感兴趣的代码片段（用 C 写的，名为 array.c）是这样的：

```
int array[1000];
...
for (i = 0; i < 1000; i++)
    array[i] = 0;
```

我们编译 array.c 并使用以下命令运行它：

补充：数据结构——页表

现代操作系统的内存管理子系统中最重要的数据结构之一就是页表（page table）。通常，页表存储虚拟—物理地址转换（virtual-to-physical address translation），从而让系统知道地址空间的每个页实际驻留在物理内存中的哪个位置。由于每个地址空间都需要这种转换，因此一般来说，系统中每个进程都有一个页表。页表的确切结构要么由硬件（旧系统）确定，要么由 OS（现代系统）更灵活地管理。

```
prompt> gcc -o array array.c -Wall -O
prompt> ./array
```

当然，为了真正理解这个代码片段（它只是初始化一个数组）进程怎样的内存访问，我们必须知道（或假设）一些东西。首先，我们必须反汇编结果二进制文件（在 Linux 上使用 objdump 或在 Mac 上使用 otool），查看使用什么汇编指令来初始化循环中的数组。以下是生成的汇编代码：

```
0x1024 movl $0x0,(%edi,%eax,4)
0x1028 incl %eax
0x102c cmpl $0x03e8,%eax
0x1030 jne  0x1024
```

如果懂一点 x86，代码实际上很容易理解[①]。第一条指令将零值（显示为$0x0）移动到数组位置的虚拟内存地址，这个地址是通过取%edi 的内容并将其加上%eax 乘以 4 来计算的。因此，%edi 保存数组的基址，而%eax 保存数组索引（i）。我们乘以 4，因为数组是一个整型数组，每个元素的大小为 4 个字节。

第二条指令增加保存在%eax 中的数组索引，第三条指令将该寄存器的内容与十六进制值 0x03e8 或十进制数 1000 进行比较。如果比较结果显示两个值不相等（这就是 jne 指令测试），第四条指令跳回到循环的顶部。

为了理解这个指令序列（在虚拟层和物理层）所访问的内存，我们必须假设虚拟内存中代码片段和数组的位置，以及页表的内容和位置。

对于这个例子，我们假设一个大小为 64KB 的虚拟地址空间（不切实际地小）。我们还假定页面大小为 1KB。

我们现在需要知道页表的内容，以及它在物理内存中的位置。假设有一个线性（基于数组）的页表，它位于物理地址 1KB（1024）。

至于其内容，我们只需要关心为这个例子映射的几个虚拟页面。首先，存在代码所在的虚拟页面。由于页大小为 1KB，虚拟地址 1024 驻留在虚拟地址空间的第二页（VPN = 1，因为 VPN = 0 是第一页）。假设这个虚拟页映射到物理帧 4（VPN 1→PFN 4）。

接下来是数组本身。它的大小是 4000 字节（1000 整数），我们假设它驻留在虚拟地址 40000 到 44000（不包括最后一个字节）。它的虚拟页的十进制范围是 VPN = 39……VPN = 42。因此，我们需要这些页的映射。针对这个例子，让我们假设以下虚拟到物理的映射：

$$(\text{VPN } 39 \rightarrow \text{PFN } 7), (\text{VPN } 40 \rightarrow \text{PFN } 8), (\text{VPN } 41 \rightarrow \text{PFN } 9), (\text{VPN } 42 \rightarrow \text{PFN } 10)$$

我们现在准备好跟踪程序的内存引用了。当它运行时，每个获取指将产生两个内存引用：一个访问页表以查找指令所在的物理框架，另一个访问指令本身将其提取到 CPU 进行处理。另外，在 mov 指令的形式中，有一个显式的内存引用，这会首先增加另一个页表访问（将数组虚拟地址转换为正确的物理地址），然后时数组访问本身。

图 18.7 展示了前 5 次循环迭代的整个过程。最下面的图显示了 *y* 轴上的指令内存引用（黑色虚拟地址和右边的实际物理地址）。中间的图以深灰色展示了数组访问（同样，虚拟在左侧，物理在右侧）；最后，最上面的图展示了浅灰色的页表内存访问（只有物理的，因为本例中的页表位于物理内存中）。整个追踪的 *x* 轴显示循环的前 5 个迭代中内存访问。每个循环有 10 次内存访问，其中包括 4 次取指令，一次显式更新内存，以及 5 次页表访问，

① 我们在这里隐瞒了一点事实，假设每条指令的大小都是 4 字节，实际上，x86 指令是可变大小的。

为这 4 次获取和一次显式更新进行地址转换。

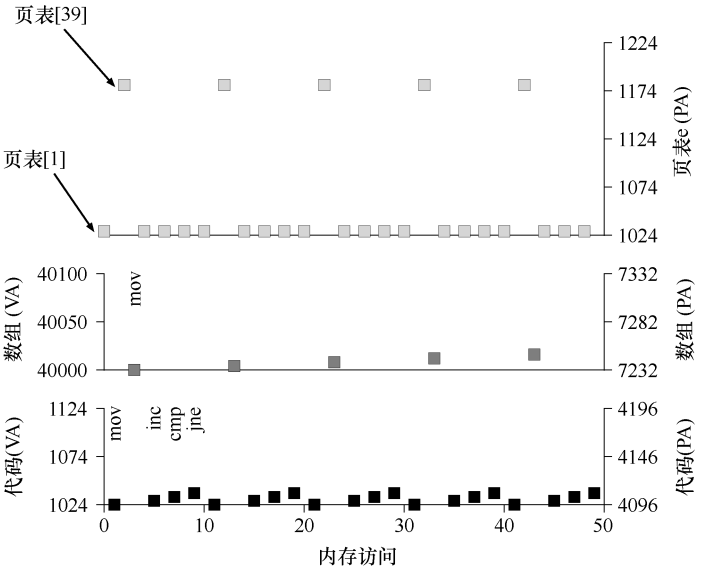


图 18.7　虚拟（和物理）内存追踪

看看你是否可以理解在这个可视化中出现的模式。特别是，随着循环继续，超过前 5 次迭代，会发生什么变化？哪些新的内存位置将被访问？你能弄明白吗？

这只是最简单的例子（只有几行 C 代码），但你可能已经能够感觉到理解实际应用程序的实际内存行为的复杂性。别担心：它肯定会变得更糟，因为我们即将引入的机制只会使这个已经很复杂的机器更复杂。

18.6　小结

我们已经引入了分页（paging）的概念，作为虚拟内存挑战的解决方案。与以前的方法（如分段）相比，分页有许多优点。首先，它不会导致外部碎片，因为分页（按设计）将内存划分为固定大小的单元。其次，它非常灵活，支持稀疏虚拟地址空间。

然而，实现分页支持而不小心考虑，会导致较慢的机器（有许多额外的内存访问来访问页表）和内存浪费（内存被页表塞满而不是有用的应用程序数据）。因此，我们不得不努力想出一个分页系统，它不仅可以工作，而且工作得很好。幸运的是，接下来的两章将告诉我们如何去做。

参考资料

[KE+62]“One-level Storage System”

T. Kilburn, and D.B.G. Edwards and M.J. Lanigan and F.H. Sumner IRE Trans. EC-11, 2 (1962), pp. 223-235

(Reprinted in Bell and Newell,“Computer Structures: Readings and Examples”McGraw-Hill, New York, 1971).
Atlas 开创了将内存划分为固定大小页面的想法，在许多方面，都是我们在现代计算机系统中看到的内存管理思想的早期形式。

[I09]“Intel 64 and IA-32 Architectures Software Developer’s Manuals”Intel, 2009
Available.
具体来说，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》。

[L78]“The Manchester Mark I and atlas: a historical perspective”
S. H. Lavington
Communications of the ACM archive Volume 21, Issue 1 (January 1978), pp. 4-12 Special issue on computer architecture
本文是一些重要计算机系统发展历史的回顾。我们在美国有时会忘记，这些新想法中的许多来自其他国家。

作业

在这个作业中，你将使用一个简单的程序（名为 paging-linear-translate.py），来看看你是否理解了简单的虚拟—物理地址转换如何与线性页表一起工作。详情请参阅 README 文件。

问题

1．在做地址转换之前，让我们用模拟器来研究线性页表在给定不同参数的情况下如何改变大小。在不同参数变化时，计算线性页表的大小。一些建议输入如下，通过使用-v 标志，你可以看到填充了多少个页表项。

首先，要理解线性页表大小如何随着地址空间的增长而变化：

```
paging-linear-translate.py -P 1k -a 1m -p 512m -v -n 0
paging-linear-translate.py -P 1k -a 2m -p 512m -v -n 0
paging-linear-translate.py -P 1k -a 4m -p 512m -v -n 0
```

然后，理解线性页面大小如何随页大小的增长而变化：

```
paging-linear-translate.py -P 1k -a 1m -p 512m -v -n 0
paging-linear-translate.py -P 2k -a 1m -p 512m -v -n 0
paging-linear-translate.py -P 4k -a 1m -p 512m -v -n 0
```

在运行这些命令之前，请试着想想预期的趋势。页表大小如何随地址空间的增长而改变？随着页大小的增长呢？为什么一般来说，我们不应该使用很大的页呢？

2．现在让我们做一些地址转换。从一些小例子开始，使用-u 标志更改分配给地址空间的页数。例如：

```
paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 0
paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 25
paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 50
paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 75
paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 100
```

如果增加每个地址空间中的页的百分比，会发生什么？

3．现在让我们尝试一些不同的随机种子，以及一些不同的（有时相当疯狂的）地址空间参数：

```
paging-linear-translate.py -P 8  -a 32   -p 1024 -v -s 1
paging-linear-translate.py -P 8k -a 32k  -p 1m   -v -s 2
paging-linear-translate.py -P 1m -a 256m -p 512m -v -s 3
```

哪些参数组合是不现实的？为什么？

4．利用该程序尝试其他一些问题。你能找到让程序无法工作的限制吗？例如，如果地址空间大小大于物理内存，会发生什么情况？

第 19 章　分页：快速地址转换（TLB）

使用分页作为核心机制来实现虚拟内存，可能会带来较高的性能开销。因为要使用分页，就要将内存地址空间切分成大量固定大小的单元（页），并且需要记录这些单元的地址映射信息。因为这些映射信息一般存储在物理内存中，所以在转换虚拟地址时，分页逻辑上需要一次额外的内存访问。每次指令获取、显式加载或保存，都要额外读一次内存以得到转换信息，这慢得无法接受。因此我们面临如下问题：

关键问题：如何加速地址转换

如何才能加速虚拟地址转换，尽量避免额外的内存访问？需要什么样的硬件支持？操作系统该如何支持？

想让某些东西更快，操作系统通常需要一些帮助。帮助常常来自操作系统的老朋友：硬件。我们要增加所谓的（由于历史原因[CP78]）地址转换旁路缓冲存储器（translation-lookaside buffer，TLB[CG68,C95]），它就是频繁发生的虚拟到物理地址转换的硬件缓存（cache）。因此，更好的名称应该是地址转换缓存（address-translation cache）。对每次内存访问，硬件先检查 TLB，看看其中是否有期望的转换映射，如果有，就完成转换（很快），不用访问页表（其中有全部的转换映射）。TLB 带来了巨大的性能提升，实际上，因此它使得虚拟内存成为可能[C95]。

19.1　TLB 的基本算法

图 19.1 展示了一个大体框架，说明硬件如何处理虚拟地址转换，假定使用简单的线性页表（linear page table，即页表是一个数组）和硬件管理的 TLB（hardware-managed TLB，即硬件承担许多页表访问的责任，下面会有更多解释）。

```
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            AccessMemory(PhysAddr)
8        else
9            RaiseException(PROTECTION_FAULT)
10   else     // TLB Miss
11       PTEAddr = PTBR + (VPN * sizeof(PTE))
12       PTE = AccessMemory(PTEAddr)
```

```
13      if (PTE.Valid == False)
14          RaiseException(SEGMENTATION_FAULT)
15      else if (CanAccess(PTE.ProtectBits) == False)
16          RaiseException(PROTECTION_FAULT)
17      else
18          TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
19          RetryInstruction()
```

图 19.1 TLB 控制流算法

硬件算法的大体流程如下：首先从虚拟地址中提取页号（VPN）（见图 19.1 第 1 行），然后检查 TLB 是否有该 VPN 的转换映射（第 2 行）。如果有，我们有了 TLB 命中（TLB hit），这意味着 TLB 有该页的转换映射。成功！接下来我们就可以从相关的 TLB 项中取出页帧号（PFN），与原来虚拟地址中的偏移量组合形成期望的物理地址（PA），并访问内存（第 5～7 行），假定保护检查没有失败（第 4 行）。

如果 CPU 没有在 TLB 中找到转换映射（TLB 未命中），我们有一些工作要做。在本例中，硬件访问页表来寻找转换映射（第 11～12 行），并用该转换映射更新 TLB（第 18 行），假设该虚拟地址有效，而且我们有相关的访问权限（第 13、15 行）。上述系列操作开销较大，主要是因为访问页表需要额外的内存引用（第 12 行）。最后，当 TLB 更新成功后，系统会重新尝试该指令，这时 TLB 中有了这个转换映射，内存引用得到很快处理。

TLB 和其他缓存相似，前提是在一般情况下，转换映射会在缓存中（即命中）。如果是这样，只增加了很少的开销，因为 TLB 处理器核心附近，设计的访问速度很快。如果 TLB 未命中，就会带来很大的分页开销。必须访问页表来查找转换映射，导致一次额外的内存引用（或者更多，如果页表更复杂）。如果这经常发生，程序的运行就会显著变慢。相对于大多数 CPU 指令，内存访问开销很大，TLB 未命中导致更多内存访问。因此，我们希望尽可能避免 TLB 未命中。

19.2 示例：访问数组

为了弄清楚 TLB 的操作，我们来看一个简单虚拟地址追踪，看看 TLB 如何提高它的性能。在本例中，假设有一个由 10 个 4 字节整型数组成的数组，起始虚地址是 100。进一步假定，有一个 8 位的小虚地址空间，页大小为 16B。我们可以把虚地址划分为 4 位的 VPN（有 16 个虚拟内存页）和 4 位的偏移量（每个页中有 16 个字节）。

图 19.2 展示了该数组的布局，在系统的 16 个 16 字节的页上。如你所见，数组的第一项（a[0]）开始于（VPN=06，offset=04），只有 3 个 4 字节整型数存放在该页。数组在下一页（VPN=07）继续，其中有接下来 4 项（a[3] … a[6]）。10 个元素的数组的最后 3 项（a[7] … a[9]）位于地址空间的下一页（VPN=08）。

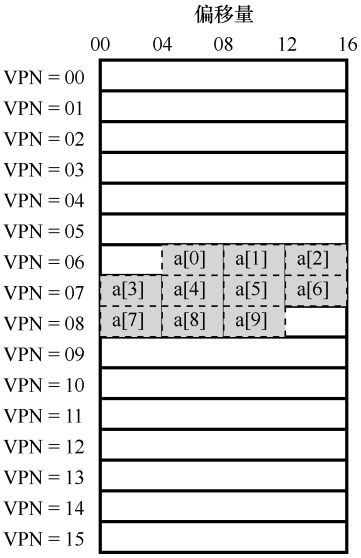


图 19.2 示例：小地址空间中的一个数组

现在考虑一个简单的循环，访问数组中的每个元素，类似下面的 C 程序：

```
int sum = 0;
for (i = 0; i < 10; i++) {
    sum += a[i];
}
```

简单起见，我们假装循环产生的内存访问只是针对数组（忽略变量 *i* 和 *sum*，以及指令本身）。当访问第一个数组元素（a[0]）时，CPU 会看到载入虚存地址 100。硬件从中提取 VPN（VPN=06），然后用它来检查 TLB，寻找有效的转换映射。假设这里是程序第一次访问该数组，结果是 TLB 未命中。

接下来访问 a[1]，这里有好消息：TLB 命中！因为数组的第二个元素在第一个元素之后，它们在同一页。因为我们之前访问数组的第一个元素时，已经访问了这一页，所以 TLB 中缓存了该页的转换映射。因此成功命中。访问 a[2]同样成功（再次命中），因为它和 a[0]、a[1]位于同一页。

遗憾的是，当程序访问 a[3]时，会导致 TLB 未命中。但同样，接下来几项（a[4] … a[6]）都会命中 TLB，因为它们位于内存中的同一页。

最后，访问 a[7]会导致最后一次 TLB 未命中。系统会再次查找页表，弄清楚这个虚拟页在物理内存中的位置，并相应地更新 TLB。最后两次访问（a[8]、a[9]）受益于这次 TLB 更新，当硬件在 TLB 中查找它们的转换映射时，两次都命中。

我们来总结一下这 10 次数组访问操作中 TLB 的行为表现：未命中、命中、命中、未命中、命中、命中、命中、未命中、命中、命中。命中的次数除以总的访问次数，得到 TLB 命中率（hit rate）为 70%。尽管这不是很高（实际上，我们希望命中率接近 100%），但也不是零，是零我们就要奇怪了。即使这是程序首次访问该数组，但得益于空间局部性（spatial locality），TLB 还是提高了性能。数组的元素被紧密存放在几页中（即它们在空间中紧密相邻），因此只有对页中第一个元素的访问才会导致 TLB 未命中。

也要注意页大小对本例结果的影响。如果页大小变大一倍（32 字节，而不是 16），数组访问遇到的未命中更少。典型页的大小一般为 4KB，这种情况下，密集的、基于数组的访问会实现极好的 TLB 性能，每页的访问只会遇到一次未命中。

关于 TLB 性能还有最后一点：如果在这次循环后不久，该程序再次访问该数组，我们会看到更好的结果，假设 TLB 足够大，能缓存所需的转换映射：命中、命中、命中、命中、命中、命中、命中、命中、命中、命中。在这种情况下，由于时间局部性（temporal locality），即在短时间内对内存项再次引用，所以 TLB 的命中率会很高。类似其他缓存，TLB 的成功依赖于空间和时间局部性。如果某个程序表现出这样的局部性（许多程序是这样），TLB 的命中率可能很高。

提示：尽可能利用缓存

缓存是计算机系统中最基本的性能改进技术之一，一次又一次地用于让“常见的情况更快”[HP06]。硬件缓存背后的思想是利用指令和数据引用的局部性（locality）。通常有两种局部性：时间局部性（temporal locality）和空间局部性（spatial locality）。时间局部性是指，最近访问过的指令或数据项可能很快会再次访问。想想循环中的循环变量或指令，它们被多次反复访问。空间局部性是指，当程序访问

内存地址 x 时，可能很快会访问邻近 x 的内存。想想遍历某种数组，访问一个接一个的元素。当然，这些性质取决于程序的特点，并不是绝对的定律，而更像是一种经验法则。

硬件缓存，无论是指令、数据还是地址转换（如 TLB），都利用了局部性，在小而快的芯片内存储器中保存一份内存副本。处理器可以先检查缓存中是否存在就近的副本，而不是必须访问（缓慢的）内存来满足请求。如果存在，处理器就可以很快地访问它（例如在几个 CPU 时钟内），避免花很多时间来访问内存（好多纳秒）。

你可能会疑惑：既然像 TLB 这样的缓存这么好，为什么不做更大的缓存，装下所有的数据？可惜的是，这里我们遇到了更基本的定律，就像物理定律那样。如果想要快速地缓存，它就必须小，因为光速和其他物理限制会起作用。大的缓存注定慢，因此无法实现目的。所以，我们只能用小而快的缓存。剩下的问题就是如何利用好缓存来提升性能。

19.3 谁来处理 TLB 未命中

有一个问题我们必须回答：谁来处理 TLB 未命中？可能有两个答案：硬件或软件（操作系统）。以前的硬件有复杂的指令集（有时称为复杂指令集计算机，Complex-Instruction Set Computer，CISC），造硬件的人不太相信那些搞操作系统的人。因此，硬件全权处理 TLB 未命中。为了做到这一点，硬件必须知道页表在内存中的确切位置（通过页表基址寄存器，page-table base register，在图 19.1 的第 11 行使用），以及页表的确切格式。发生未命中时，硬件会“遍历”页表，找到正确的页表项，取出想要的转换映射，用它更新 TLB，并重试该指令。这种“旧”体系结构有硬件管理的 TLB，一个例子是 x86 架构，它采用固定的多级页表（multi-level page table，详见第 20 章），当前页表由 CR3 寄存器指出[I09]。

更现代的体系结构（例如，MIPS R10k[H93]、Sun 公司的 SPARC v9[WG00]，都是精简指令集计算机，Reduced-Instruction Set Computer，RISC），有所谓的软件管理 TLB（software-managed TLB）。发生 TLB 未命中时，硬件系统会抛出一个异常（见图 19.3 第 11 行），这会暂停当前的指令流，将特权级提升至内核模式，跳转至陷阱处理程序（trap handler）。接下来你可能已经猜到了，这个陷阱处理程序是操作系统的一段代码，用于处理 TLB 未命中。这段代码在运行时，会查找页表中的转换映射，然后用特别的“特权”指令更新 TLB，并从陷阱返回。此时，硬件会重试该指令（导致 TLB 命中）。

```
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       RaiseException(TLB_MISS)
```

图 19.3 TLB 控制流算法（操作系统处理）

接下来讨论几个重要的细节。首先，这里的从陷阱返回指令稍稍不同于之前提到的服务于系统调用的从陷阱返回。在后一种情况下，从陷阱返回应该继续执行陷入操作系统之后那条指令，就像从函数调用返回后，会继续执行此次调用之后的语句。在前一种情况下，在从 TLB 未命中的陷阱返回后，硬件必须从导致陷阱的指令继续执行。这次重试因此导致该指令再次执行，但这次会命中 TLB。因此，根据陷阱或异常的原因，系统在陷入内核时必须保存不同的程序计数器，以便将来能够正确地继续执行。

第二，在运行 TLB 未命中处理代码时，操作系统需要格外小心避免引起 TLB 未命中的无限递归。有很多解决方案，例如，可以把 TLB 未命中陷阱处理程序直接放到物理内存中 [它们没有映射过（unmapped），不用经过地址转换]。或者在 TLB 中保留一些项，记录永久有效的地址转换，并将其中一些永久地址转换槽块留给处理代码本身，这些被监听的（wired）地址转换总是会命中 TLB。

软件管理的方法，主要优势是灵活性：操作系统可以用任意数据结构来实现页表，不需要改变硬件。另一个优势是简单性。从 TLB 控制流中可以看出（见图 19.3 的第 11 行，对比图 19.1 的第 11～19 行），硬件不需要对未命中做太多工作，它抛出异常，操作系统的未命中处理程序会负责剩下的工作。

补充：RISC 与 CISC

在 20 世纪 80 年代，计算机体系结构领域曾发生过一场激烈的讨论。一方是 CISC 阵营，即复杂指令集计算（Complex Instruction Set Computing），另一方是 RISC，即精简指令集计算（Reduced Instruction Set Computing）[PS81]。RISC 阵营以 Berkeley 的 David Patterson 和 Stanford 的 John Hennessy 为代表（他们写了一些非常著名的书[HP06]），尽管后来 John Cocke 凭借他在 RISC 上的早期工作 [CM00]获得了图灵奖。

CISC 指令集倾向于拥有许多指令，每条指令比较强大。例如，你可能看到一个字符串拷贝，它接受两个指针和一个长度，将一些字节从源拷贝到目标。CISC 背后的思想是，指令应该是高级原语，这让汇编语言本身更易于使用，代码更紧凑。

RISC 指令集恰恰相反。RISC 背后的关键观点是，指令集实际上是编译器的最终目标，所有编译器实际上需要少量简单的原语，可以用于生成高性能的代码。因此，RISC 倡导者们主张，尽可能从硬件中拿掉不必要的东西（尤其是微代码），让剩下的东西简单、统一、快速。

早期的 RISC 芯片产生了巨大的影响，因为它们明显更快[BC91]。人们写了很多论文，一些相关的公司相继成立（例如 MIPS 和 Sun 公司）。但随着时间的推移，像 Intel 这样的 CISC 芯片制造商采纳了许多 RISC 芯片的优点，例如添加了早期流水线阶段，将复杂的指令转换为一些微指令，于是它们可以像 RISC 的方式运行。这些创新，加上每个芯片中晶体管数量的增长，让 CISC 保持了竞争力。争论最后平息了，现在两种类型的处理器都可以跑得很快。

19.4 TLB 的内容

我们来详细看一下硬件 TLB 中的内容。典型的 TLB 有 32 项、64 项或 128 项，并且是全相联的（fully associative）。基本上，这就意味着一条地址映射可能存在 TLB 中的任意位置，硬件会并行地查找 TLB，找到期望的转换映射。一条 TLB 项内容可能像下面这样：

VPN | PFN | 其他位

注意，VPN 和 PFN 同时存在于 TLB 中，因为一条地址映射可能出现在任意位置（用硬件的术语，TLB 被称为全相联的（fully-associative）缓存）。硬件并行地查找这些项，看看是否有匹配。

补充：TLB 的有效位!=页表的有效位

常见的错误是混淆 TLB 的有效位和页表的有效位。在页表中，如果一个页表项（PTE）被标记为无效，就意味着该页并没有被进程申请使用，正常运行的程序不应该访问该地址。当程序试图访问这样的页时，就会陷入操作系统，操作系统会杀掉该进程。

TLB 的有效位不同，只是指出 TLB 项是不是有效的地址映射。例如，系统启动时，所有的 TLB 项通常被初始化为无效状态，因为还没有地址转换映射被缓存在这里。一旦启用虚拟内存，当程序开始运行，访问自己的虚拟地址，TLB 就会慢慢地被填满，因此有效的项很快会充满 TLB。

TLB 有效位在系统上下文切换时起到了很重要的作用，后面我们会进一步讨论。通过将所有 TLB 项设置为无效，系统可以确保将要运行的进程不会错误地使用前一个进程的虚拟到物理地址转换映射。

更有趣的是“其他位”。例如，TLB 通常有一个有效（valid）位，用来标识该项是不是有效地转换映射。通常还有一些保护（protection）位，用来标识该页是否有访问权限。例如，代码页被标识为可读和可执行，而堆的页被标识为可读和可写。还有其他一些位，包括地址空间标识符（address-space identifier）、脏位（dirty bit）等。下面会介绍更多信息。

19.5 上下文切换时对 TLB 的处理

有了 TLB，在进程间切换时（因此有地址空间切换），会面临一些新问题。具体来说，TLB 中包含的虚拟到物理的地址映射只对当前进程有效，对其他进程是没有意义的。所以在发生进程切换时，硬件或操作系统（或二者）必须注意确保即将运行的进程不要误读了之前进程的地址映射。

为了更好地理解这种情况，我们来看一个例子。当一个进程（P1）正在运行时，假设 TLB 缓存了对它有效的地址映射，即来自 P1 的页表。对这个例子，假设 P1 的 10 号虚拟页映射到了 100 号物理帧。

在这个例子中，假设还有一个进程（P2），操作系统不久后决定进行一次上下文切换，运行 P2。这里假定 P2 的 10 号虚拟页映射到 170 号物理帧。如果这两个进程的地址映射都在 TLB 中，TLB 的内容如表 19.1 所示。

表 19.1 TLB 的内容

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

在上面的 TLB 中，很明显有一个问题：VPN 10 被转换成了 PFN 100（P1）和 PFN 170