

虚拟内存

第十九章——TLB& 更小的页表

刘玉峰

Fx_yfliu@163.com 湖

南大学

- 使用分页 作为支持虚拟内存 的核心机制会导致 **高性能 开销**。
- 通过 将 地址空间切割成小的、固定大小的单元（页），页需要大量的映射信息。
- 因为映射信息通常存储在物理内存中，所以分页逻辑上**需要**对 程序生成的每个虚拟 地址进行额外的内存查找。
- 在每次取指令或显式 加载 或存储 之前，到内存中获取转换信息的速度 **非常慢**。

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？

- 为了加快地址转换，我们将 添加 所谓的 **转换后备缓冲区**（**TLB**）。
- **TLB** 是 芯片 **内存管理单元**（**MMU**）的一部分，只是一个流行的虚拟到物理地址转换的硬件缓存。
- 在每个虚拟存储器引用时，硬件首先检查**TLB**以查看是否保持期望的转换;如果是，则执行转换（快速）**而不必咨询页表**。

TLB 基本 算法

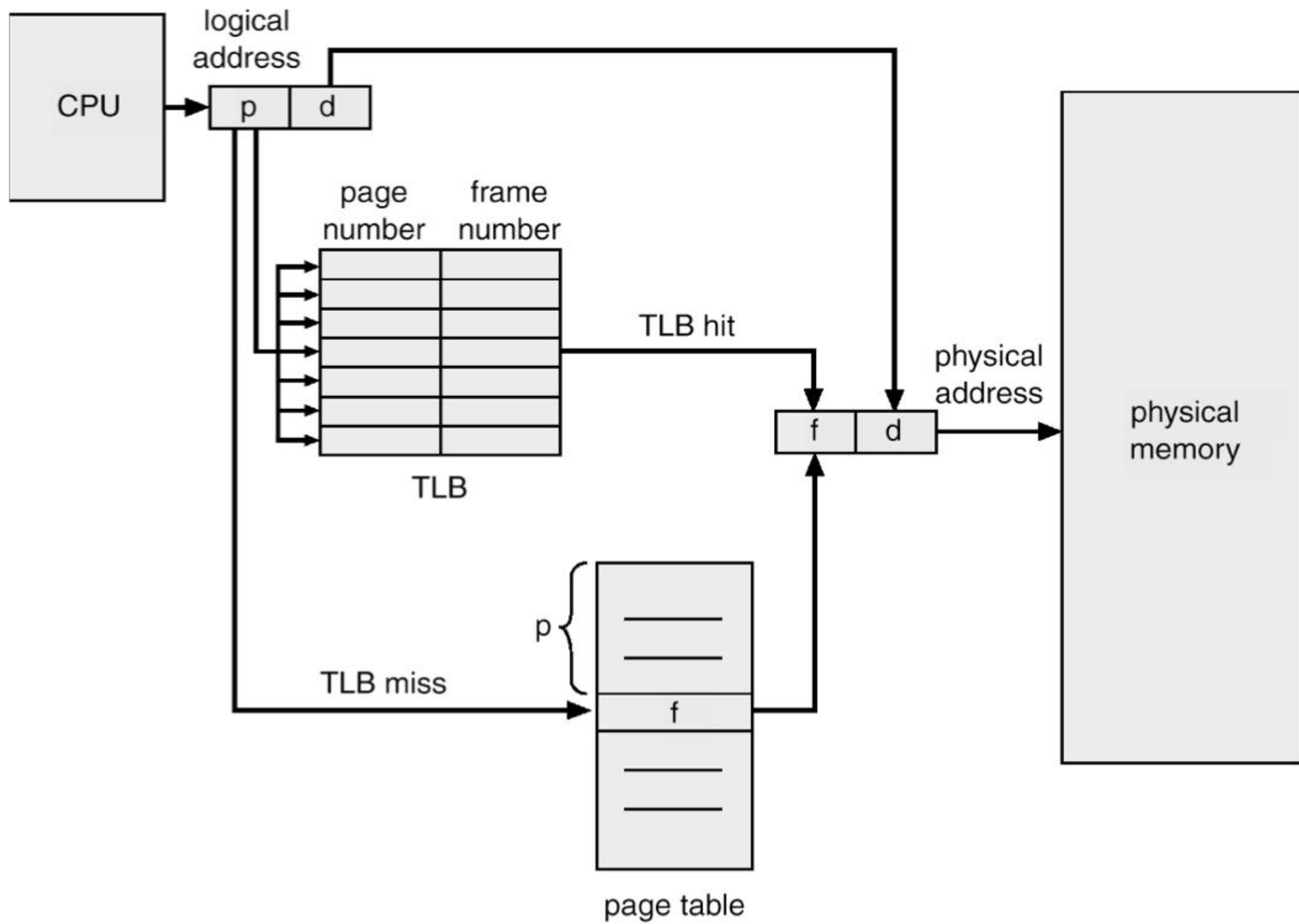
- 下图显示了硬件如何处理 虚拟 地址转换的粗略示意图， 假设使用简单的 **线性页表**（即，页表是 阵列） 和 **硬件管理的 TLB**。

```

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()

```

TLB Control Flow Algorithm



- TLB与所有缓存一样，都是基于以下前提构建的：
通常情况下，在缓存中找到转换。
- 如果是这样的话，几乎不会增加开销，因为 TLB靠近处理核心并且被设计为相当快。
- 当发生未命中时，会产生很高的分页开销；必须访问页表才能找到转换。
- 如果这种情况经常发生，程序可能会明显运行得更慢；**内存访问的成本相当高**，并且 TLB未命中会导致更多的内存访问。
- 因此，我们希望尽可能地**避免TLB 未命中**。

示例：访问数组

- 假定，有一个 8 位的虚地址空间，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）。
- 在系统的 16 个 16 字节的页上数组的第一项（**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）。

	Offset				
	00	04	08	12	16
VPN = 00					
VPN = 01					
VPN = 02					
VPN = 03					
VPN = 04					
VPN = 05					
VPN = 06		a[0]	a[1]	a[2]	
VPN = 07	a[3]	a[4]	a[5]	a[6]	
VPN = 08	a[7]	a[8]	a[9]		
VPN = 09					
VPN = 10					
VPN = 11					
VPN = 12					
VPN = 13					
VPN = 14					
VPN = 15					

- 现在让我们考虑 一个访问 每个 数组 元素的简单循环：

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

- 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 ($\text{VPN}=06$), 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。

	Offset				
	00	04	08	12	16
VPN = 00					
VPN = 01					
VPN = 02					
VPN = 03					
VPN = 04					
VPN = 05					
VPN = 06		a[0]	a[1]	a[2]	
VPN = 07	a[3]	a[4]	a[5]	a[6]	
VPN = 08	a[7]	a[8]	a[9]		
VPN = 09					
VPN = 10					
VPN = 11					
VPN = 12					
VPN = 13					
VPN = 14					
VPN = 15					

- 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。

	Offset				
	00	04	08	12	16
VPN = 00					
VPN = 01					
VPN = 02					
VPN = 03					
VPN = 04					
VPN = 05					
VPN = 06		$a[0]$	$a[1]$	$a[2]$	
VPN = 07	$a[3]$	$a[4]$	$a[5]$	$a[6]$	
VPN = 08	$a[7]$	$a[8]$	$a[9]$		
VPN = 09					
VPN = 10					
VPN = 11					
VPN = 12					
VPN = 13					
VPN = 14					
VPN = 15					

- 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 ($a[4] \cdots a[6]$) 都会命中 TLB，因为它们位于内存中的同一页。

	Offset				
	00	04	08	12	16
VPN = 00					
VPN = 01					
VPN = 02					
VPN = 03					
VPN = 04					
VPN = 05					
VPN = 06		$a[0]$	$a[1]$	$a[2]$	
VPN = 07	$a[3]$	$a[4]$	$a[5]$	$a[6]$	
VPN = 08	$a[7]$	$a[8]$	$a[9]$		
VPN = 09					
VPN = 10					
VPN = 11					
VPN = 12					
VPN = 13					
VPN = 14					
VPN = 15					

- 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。

	Offset				
	00	04	08	12	16
VPN = 00					
VPN = 01					
VPN = 02					
VPN = 03					
VPN = 04					
VPN = 05					
VPN = 06		$a[0]$	$a[1]$	$a[2]$	
VPN = 07	$a[3]$	$a[4]$	$a[5]$	$a[6]$	
VPN = 08	$a[7]$	$a[8]$	$a[9]$		
VPN = 09					
VPN = 10					
VPN = 11					
VPN = 12					
VPN = 13					
VPN = 14					
VPN = 15					

- 让我们总结一下TLB在对数组的十次访问期间的活动：
未命中、命中、命中、未命中、命中、命中、未命中、命中、命中。
因此，我们的TLB命中率（命中数除以访问总数）为 **70%**。
- Even though this is the first time the program accesses the array, the TLB improves performance due to **spatial locality**（空间局部性），and thus only the first access to an element on a page yields a TLB miss.

- 如果 页面大小是原来的**的两倍**（**32**字节，而不是**16**字节），则阵列 访问 将遭受更少的未命中。
- 由于 典型的页面大小更接近**4KB**，因此这些类型的密集的、基于阵列的访问实现了出色的**TLB**性能。

- 如果 程序在这个 循环 完成后不久再次访问数组，我们会看到一个更好的结果：**hit, hit, hit, hit, hit, hit, hit, hit, hit, hit, hit**。
- 在 这种情况下，TLB命中率会很高， 因为 **temporal locality**（时间局部性）。

- 如果缓存（比如 TLB）这么棒，**为什么我们不做更大的缓存**，把我们所有的数据都保存在里面呢？
- 不幸的是，这是我们遇到更基本的法律，如物理学。如果你想要**一个快速的缓存，它必须很小**，因为像光速和其他物理约束这样的问题变得相关。
- 根据定义，任何大型缓存都是缓慢的，因此无法达到目的。
- 因此，我们只能使用小而快的缓存;剩下的问题是**如何最好地使用它们来提高性能**。

谁来处理 TLB 未命中？

- 硬件 **管理** 的**TLB**，如Intel x86 架构。
- 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），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。

谁来处理 TLB 未命中？

- 软件**管理的TLB**，如 Sun 的 SPARKV9。
- 发生 TLB 未命中时，硬件系统会抛出一个异常（`RaiseException(TLB_MISS)`），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。

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。

TLB 内容： 里面 有什么？

- 典型的 TLB 可能有 32、64 或 128 项，并且是所谓的全 关联。
- 基本上，这 只是 意味着任何给定的转换可以在 TLB 中的任何地方，并且硬件 将并行 搜索整个 TLB 以 找到所需的 转换。
- TLB 条目可能 看起来 像 这样：

VPN | PFN | other bits

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。

有效位表明了该虚拟页当前是否存在于物理内存中

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。

TLB 问题： 上下文切换

- The TLB contains virtual-to-physical translations that are **only valid for the currently running process**(即每个进程都有自己的虚拟地址空间).
- 当 从一个进程切换到另一个进程时， 硬件或操作系统（或两者） **必须 小心， 以 确保**即将运行的进程不会**意外地使用**来自一些先前 运行 的进程的转换。

VPN	PFN	valid	prot
10	100	1	rwX
—	—	0	—
10	170	1	rwX
—	—	0	—

- 在上面的TLB中，我们显然遇到了一个问题：VPN 10 转换为 PFN 100（P1）或PFN 170（P2），但硬件无法区分哪个条目是针对哪个进程的。
- 因此，我们需要做更多的工作，以使TLB正确和有效地支持跨多个进程的虚拟化。

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？

- 一种方法是在上下文切换时刷新**TLB**，在运行下一个进程之前清空它。
- 然而，这是有代价的：每次进程运行时，它都必须**招致TLB 未命中**。
- 为了减少该开销，一些系统添加硬件支持以使得能够跨上下文切换共享 **TLB**。具体地，一些硬件系统在**TLB**中提供**地址空间标识符**（**ASID**）字段。

VPN	PFN	valid	prot	ASID
10	100	1	rwX	1
—	—	0	—	—
10	170	1	rwX	2
—	—	0	—	—

- 使用 地址空间标识符，TLB可以同时保存来自不同进程的翻译，而不会产生任何混淆。

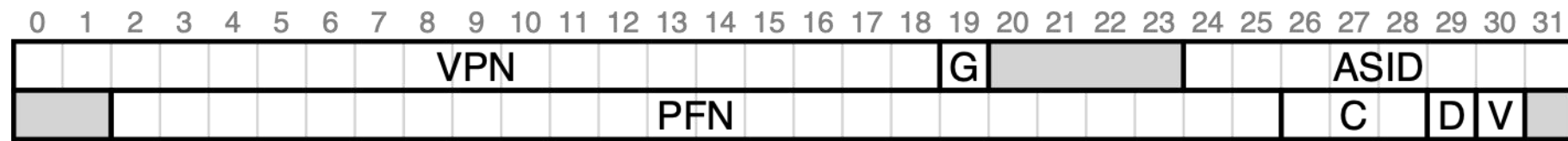
VPN	PFN	valid	prot	ASID
10	101	1	r-x	1
—	—	0	—	—
50	101	1	r-x	2
—	—	0	—	—

- 在该示例中，存在针对具有指向**相同物理 页**的**两个不同VPN**的**两个不同进程**的**两个条目**。
- 例如，当两个进程**共享一个 页面**（例如，代码页）。
- 共享代码页（在二进制文件或共享库中）非常有用，因为它可以减少使用的物理页的数量。

问题： 更换 政策

- 在TLB中 安装新条目 时，我们 必须 换 一个旧的， 换哪 一个？
- 目标 是 最小化 未 **命中率**，从而 提高性能。
- 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。

一个真实的TLB条目



A MIPS TLB Entry

- 此示例来自**MIPS R4000**，这是一个使用软件管理的TLB的现代系统。
- MIPS R4000支持**32位**地址空间和 **4KB** 页面。
- 因此，我们期望在我们的典型虚拟地址中有 **20位VPN** 和 **12位偏移**。
- 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 物理内存的系统。（ 2^{24} 4KB 页）
- 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
- 脏位（dirty），表示该页是否被写入新数据；
- 有效位（valid），告诉硬件该项的地址映射是否有效

有效存取时间

- 如果查找TLB 需要20 纳秒，访问内存需要100 纳秒，那么当页号在TLB 中时一次内存映象访问（mapped-memory access）需要120 纳秒。如果不能够在TLB 中找到页号（这个过程需要20 纳秒），那么我们必须访问内存来查找页表和帧号（100 纳秒），然后才可以访问内存（100 纳秒），总共需要220 纳秒。
- 用下面的公式可以获得有效内存访问时间（effective memory-access time）（设TLB的命中率为80%）：
$$\text{有效内存访问时间} = 0.80 \times 120 + 0.20 \times 220 = 140 \text{ 纳秒}$$
- 在这个例子中，我们比内存访问时间（memory access time）慢了40%（从100 纳秒上升到了140 纳秒）。对于98%的命中率，我们有：
$$\text{有效内存访问时间} = 0.98 \times 120 + 0.02 \times 220 = 122 \text{ 纳秒。}$$
- 这个命中率仅产生了22%的访问时间延迟。

SummaRY

- 使用一个小的片上 TLB 作为地址转换缓存，大多数内存引用将被处理，而不必访问主内存中的页表。
- 在通常情况下，程序的性能几乎就像内存根本没有被虚拟化一样。
- TLB 对于现代系统中的分页使用是 必不可少的。

第二十章——较小的页表

- 我们现在解决分页引入的第二个问题：**页表太大**，因此会消耗太多内存。
- 回想一下，线性页表变得相当大。再次假设**32位**地址空间（ **2^{32}** 字节），具有**4KB**（ **2^{12}** 字节）页和4字节页表条目。
- 因此，一个地址空间大约有一百万个虚拟页；我们的页表大小为**4 MB**。
- 如果有**一百个活动进程**，我们将为页表分配数百MB的内存！

CRUX: HOW TO MAKE PAGE TABLES SMALLER?

Simple array-based page tables (usually called linear page tables) are too big, taking up far too much memory on typical systems. How can we make page tables smaller? What are the key ideas? What inefficiencies arise as a result of these new data structures?

简单的解决方案： 更大的

页面

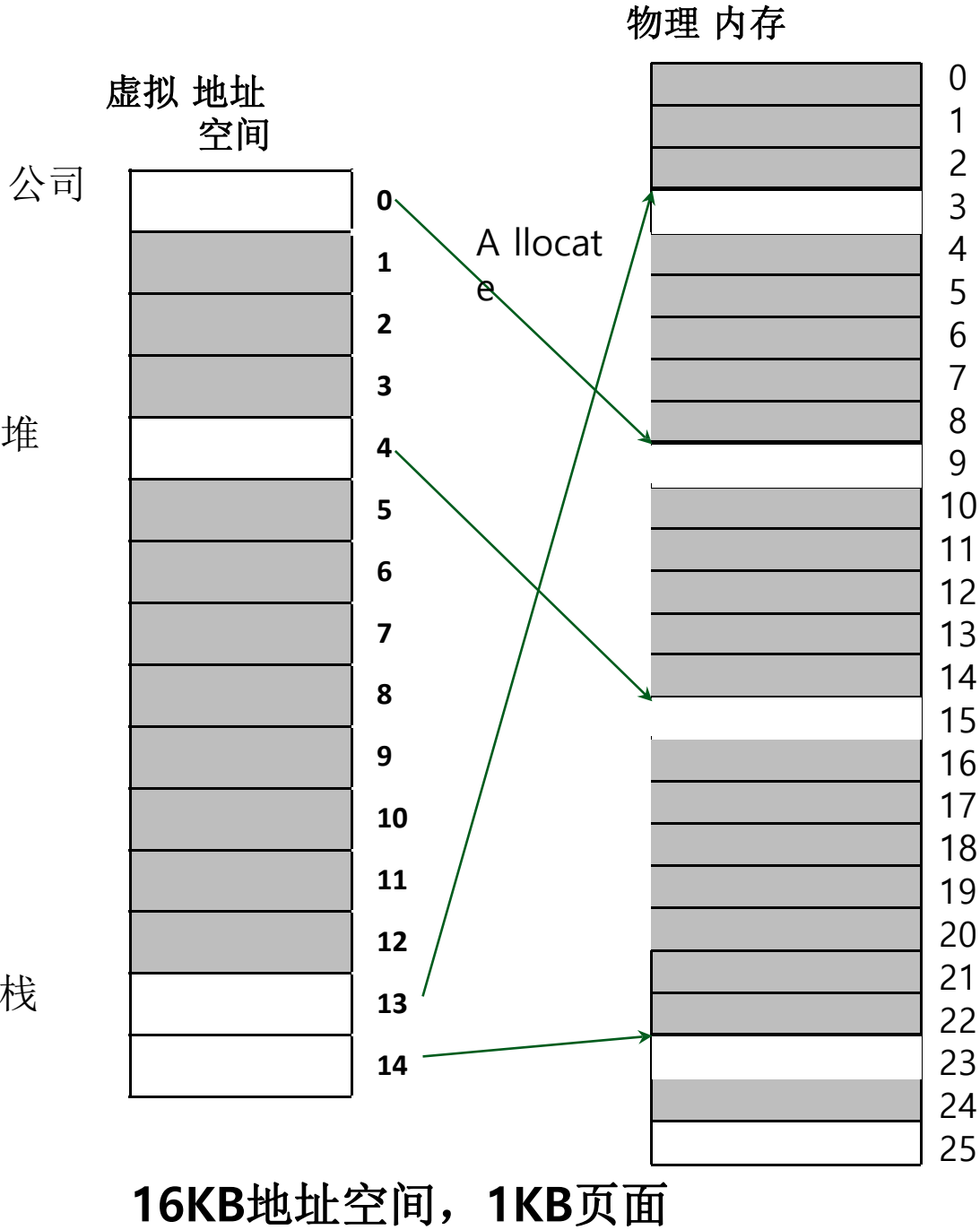
- 我们可以 通过 一种简单的方法来减小页表的大小：
使用更大的页。
- The major problem with this approach, however, is that big pages lead to waste within each page, a problem known as **internal fragmentation(内零头)**.
- 再以 32 位地址空间为例，但这次假设用 16KB 的 页。因此，会有 18 位的 VPN 加上 14 位的偏移量。假设每个页表项（4 字节）的大小相同，现在线性页表中有 2^{18} 个项，因此每个页表的总大小为 1MB，页表缩 到四分之一。

更大的页面

- 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。
- 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。

问题

- 进程的 条目地址空间的单页表。

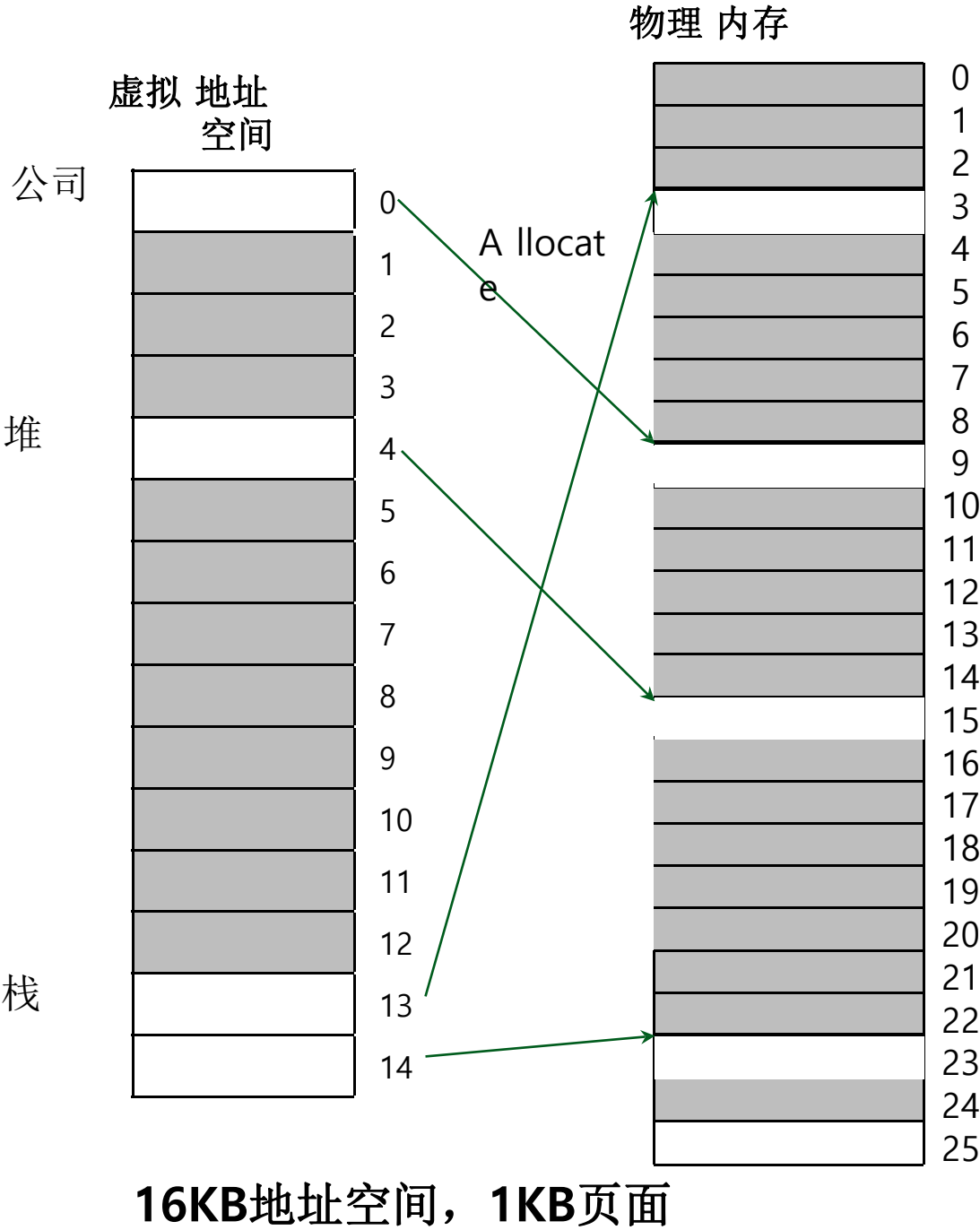


PFN	有效的	保护区	现在	肮脏的
10	1	r-x	1	0
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
15	1	rw-	1	1
...
-	0	-	-	-
3	1	rw-	1	1
23	1	rw-	1	1

16KB地址空间的 页表

问题

- 页表的大部分 **未使用**， 充满 了无效 条目。



PFN	有效的	保护区	现在	肮脏的
10	1	r-x	1	0
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
15	1	rw-	1	1
...
-	0	-	-	-
3	1	rw-	1	1
23	1	rw-	1	1

16KB地址空间的 页表

混合方法：分页和分段

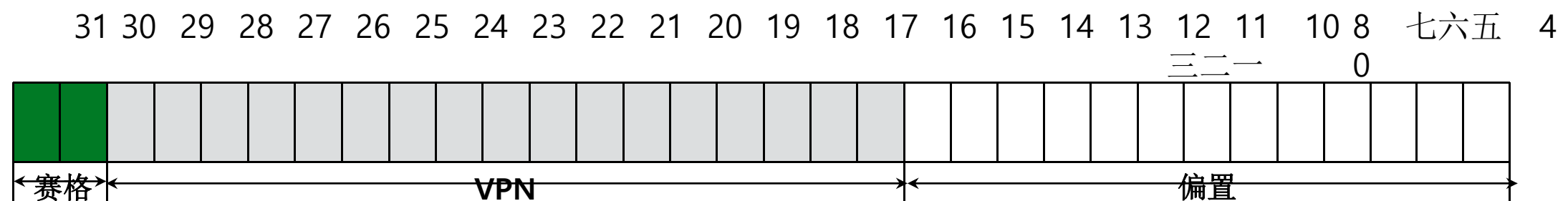
- 每当你生活中的某件事有两种合理但不同的方法时，你应该总是检查两者的结合，看看你是否能获得两个世界的最佳结果。我们把这样的组合称为 **混合体**。
- Multics的创建者（特别是**Jack Dennis**）在构建Multics虚拟内存系统时偶然发现了这样的想法。

混合方法：分页和分段

- 以减少 页表的内存 开销 。
- 使用base不指向段本身，而是保存该段页表的 **物理地址**。
- **边界寄存**器 用于指示 页表的结束 。

混合方法的简单 示例

- 本例中，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，用地址空间的前两位区分。假设 00 是未使用的段，01 是代码段，10 是堆段，11 是栈段。
- 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



具有4KB页面的32位 虚拟 地址空间

分段 值	内容
00	未用 段
01	代码
10	堆
11	栈

混合方法上的TLB 未命中

- 硬件 从页表中获取物理地址。
- 硬件使用段位 **(SN)** 来确定要使用哪个碱基和边界对。
- 然后，硬件获取其中的 **物理地址**，并如下 **将** 其与VPN 组合 以形成页表条目 (**PTE**) 的地址。

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

段页式存储管理

- 段页式存储管理
- 1) 地址空间映射
 - 段表：纪录每一段段内页表地址

段号	段长	页表始址	页表长
2	3072		3

- 段内页表



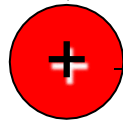
页号	块号
0	2
1	1
2	40

页号	块号
0	10
1	9
2	20

段页式存储管理的地址变换机构

- 地址变换机构及过程

有效地址寄存器



段表始址

段表大小

段表寄存器
(JT内容)

段表

段号	页表始址

页表

页号	块号



块号

块内地址

物理地址寄存器

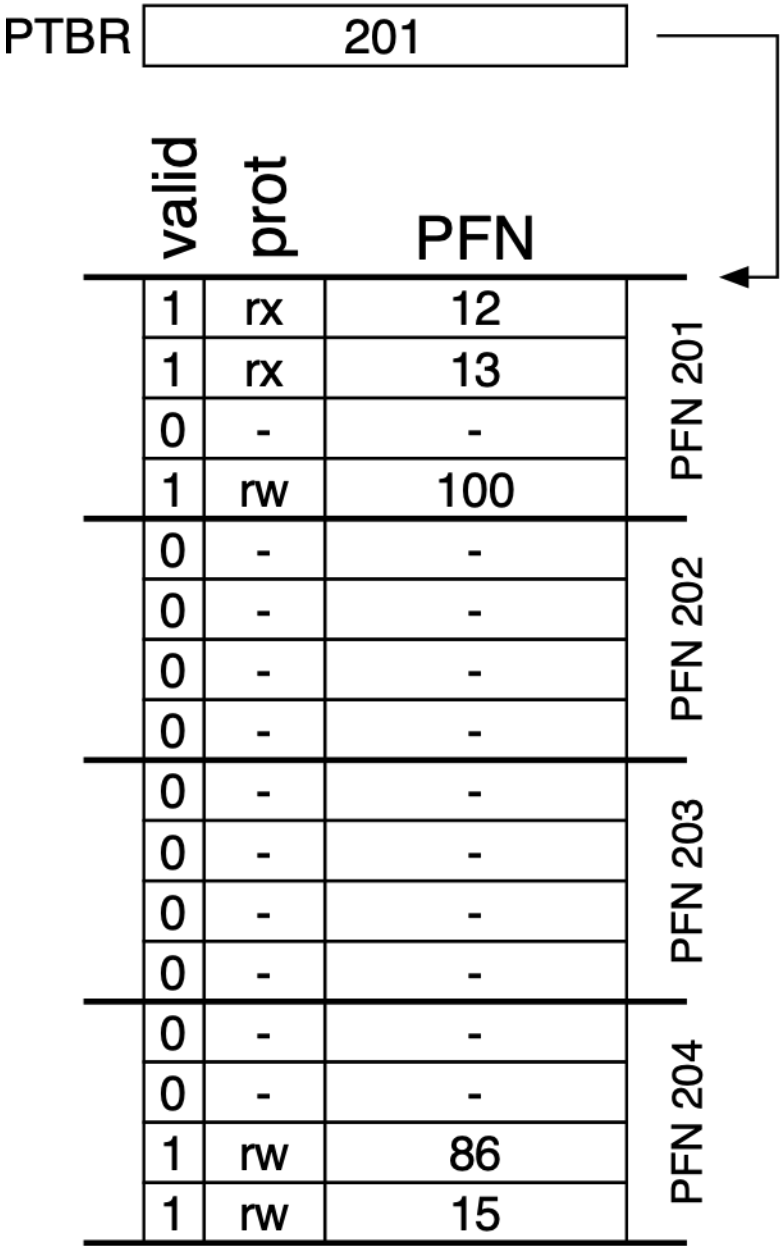
混合 方法的问题

- 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，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。
- 页表的问题：作为线性结构的页表要求分配连续的物理内存

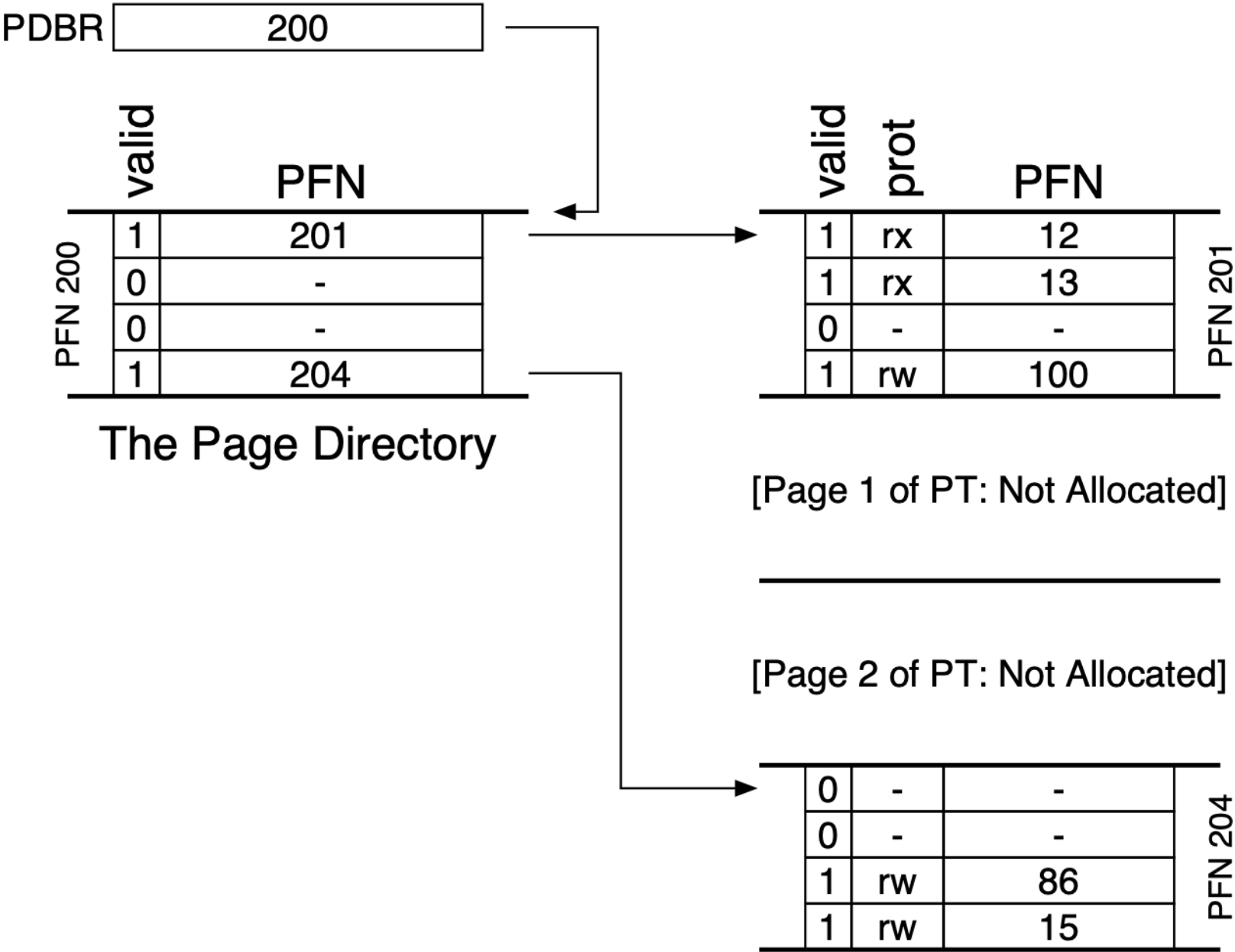
多级页表

- 首先，将页表分割成页大小的单元；然后，如果页表条目（PTE）的整个页无效，则不分配页表的该页。
- 要跟踪页表中的某个页是否有效，请使用一个称为页目录的新结构。
- 页目录可以用来告诉您页表中某个页的位置，也可以用来告诉您页表的整个页不包含有效页。

Linear Page Table



Multi-level Page Table



Linear (Left) And Multi-Level (Right) Page Tables

页目录项（Page Directory Entries，PDE）

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

多级页表的优点

- 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。
- 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

多级页表的问题

- 在 TLB 未命中时，**将需要**从 存储器 进行两次加载以从页表获得正确的 转换信息，**这与线性页表的仅一次 加载**形成对比。
- 另一个 明显的缺点是 **复杂性**（无论是 硬件还是操作系统处理 页表查找）。

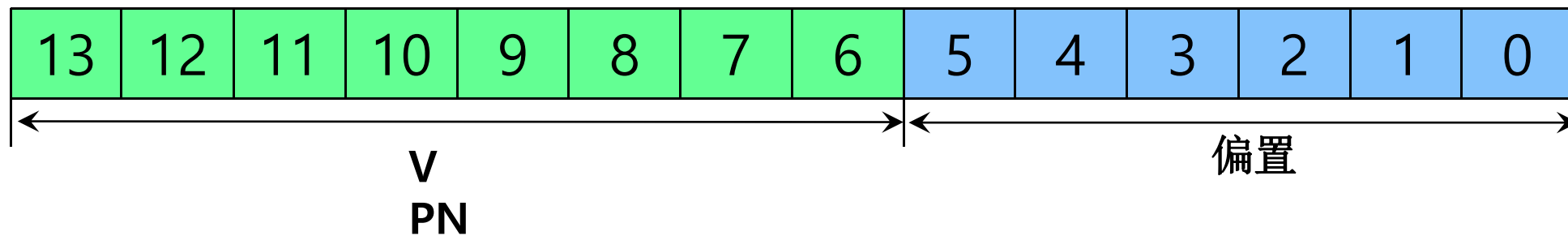
详细的多级示例

- 为了更好地理解多级页表背后的思想，让我们做一个例子



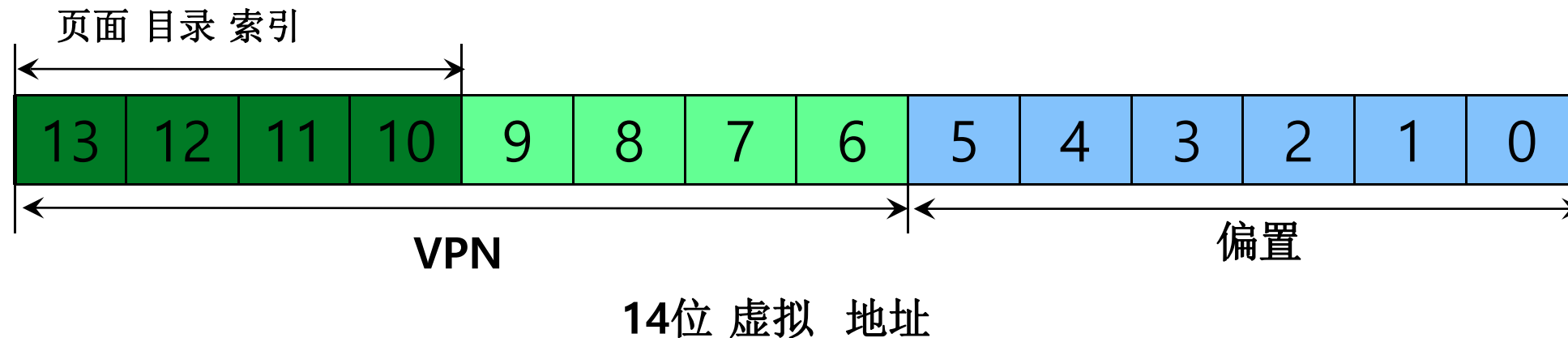
旗	详细内容
地址 空间	简体 中文
页面 大小	64 字节
虚拟 地址	14 位
VPN	8 位
偏置	6 位
页 表 项	$2^8(256)$

16 KB地址空间和64字节页面



详细的多级示例：页目录 Idx

- 页目录 需要页表的每页有一个 条目
 - 它有 16个条目。
- 页面目录条目无效?? 引发异常（访问无效）



$$\text{PDE_A_d_r} = \text{PageDirBase} + (\text{PDIndex} * \text{sizeof}(\text{PDE}))$$

PageDirBase: 页目录基址

页表索引 (Page-Table Index, PTIndex) 页目录项 (Page Directory Entries, PDE)

该公式给出了从页目录表中找到下一级页表入口地址的计算方法

页目录

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

A Page Directory, And Pieces Of Page Table

- 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。页表的这一页包含前 16 个 VPN 的映射。在Page of PT (@PFN:100)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，其中的下一级包含了VPN 254 和 255（栈）的有效映射。
- 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。

页目录

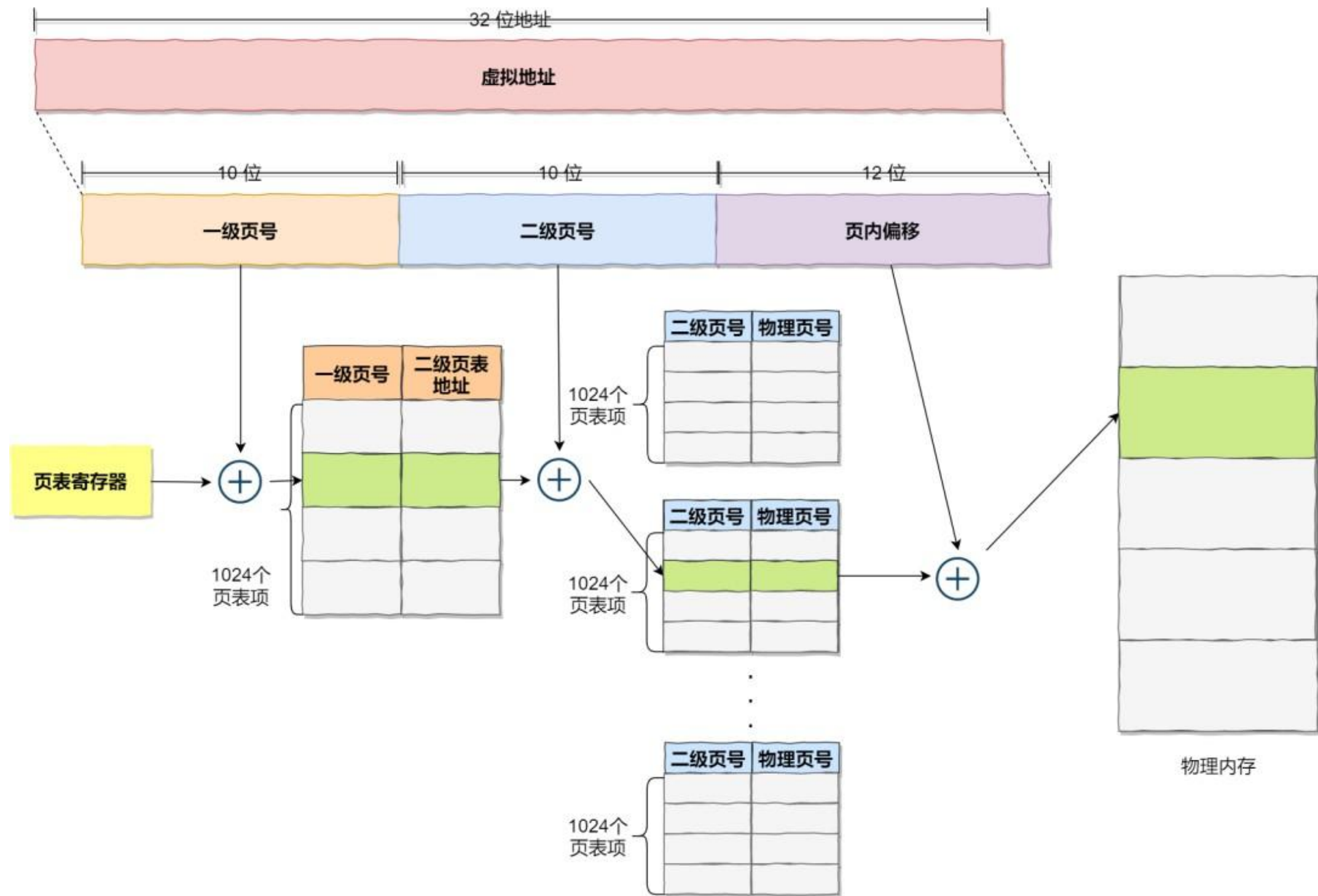
Page Directory		Page of PT (@PFN:100)			Page of PT (@PFN:101)		
PFN	valid?	PFN	valid	prot	PFN	valid	prot
100	1	10	1	r-x	—	0	—
—	0	23	1	r-x	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	80	1	rw-	—	0	—
—	0	59	1	rw-	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—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

页表

A Page Directory, And Pieces Of Page Table

- 0x3F80，或二进制的 11 1111 1000 0000
- 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，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

典型的32位地址和4K页面



为什么多级页表节约空间

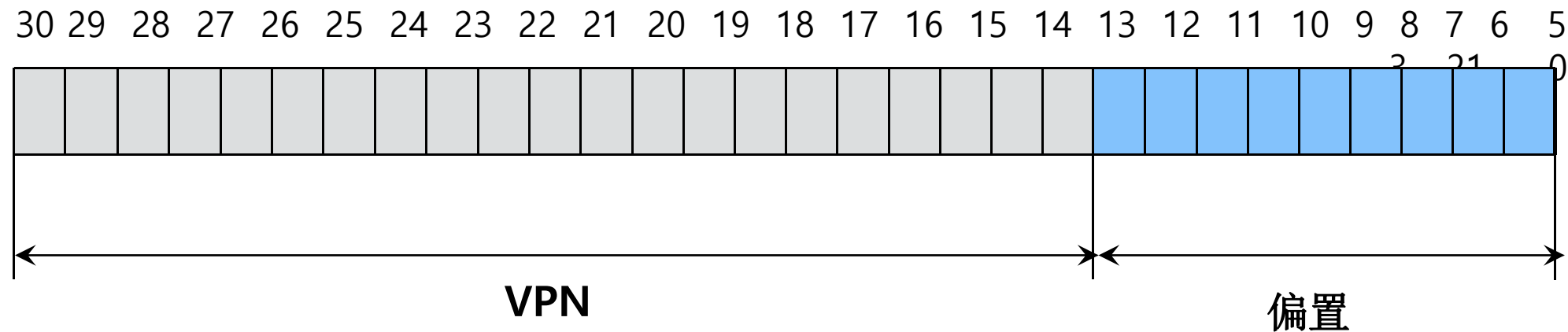
使用了二级分页，一级页表就可以覆盖整个 4GB 虚拟地址空间，但如果某个一级页表的页表项没有被用到，也就不需要创建这个页表项对应的二级页表了，即可以在需要时才创建二级页表。做个简单的计算，假设只有 20% 的一级页表项被用到了，那么页表占用的内存空间就只有 4KB（一级页表） + 20% * 4MB（二级页表）

= 0.804MB，这对比单级页表的 4MB 是一个巨大的节约

我们从页表的性质来看，保存在内存中的页表承担的职责是将虚拟地址翻译成物理地址。假如虚拟地址在页表中找不到对应的页表项，计算机系统就不能工作了。所以页表一定要覆盖全部虚拟地址空间，不分级的页表就需要有 100 多万个页表项来映射，而二级分页则只需要 1024 个页表项（此时一级页表覆盖到了全部虚拟地址空间，二级页表在需要时创建）。我们把二级分页再推广到多级页表，就会发现页表占用的内存空间更少了，这一切都要归功于对局部性原理的充分应用

超过两个级别

- 在某些情况下，更深的树是可能的。

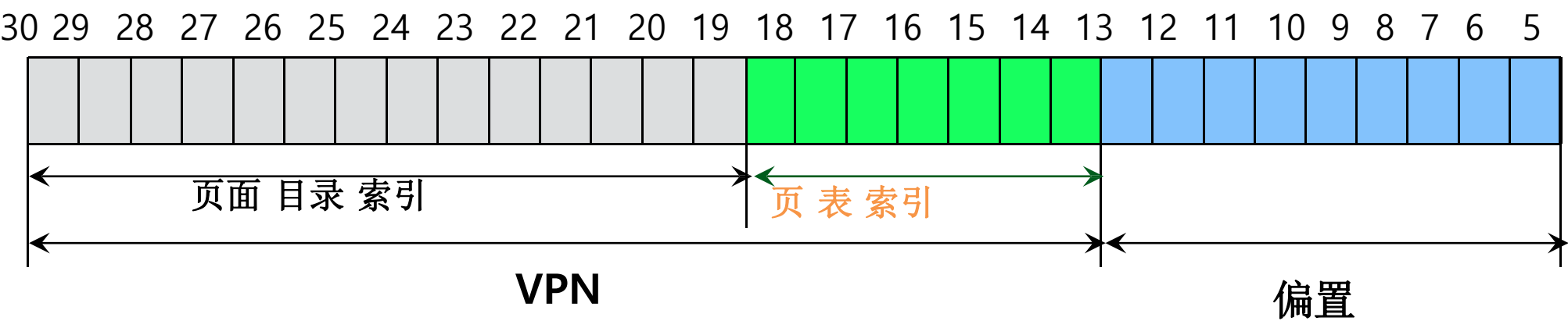


旗	详细内容
虚拟 地址	30 位
页面 大小	512 字节
VPN	21 位
偏置	9 位

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？

两级 以上： 页表 索引

- 在 某些情况下， 更 深的树 是 可能的。



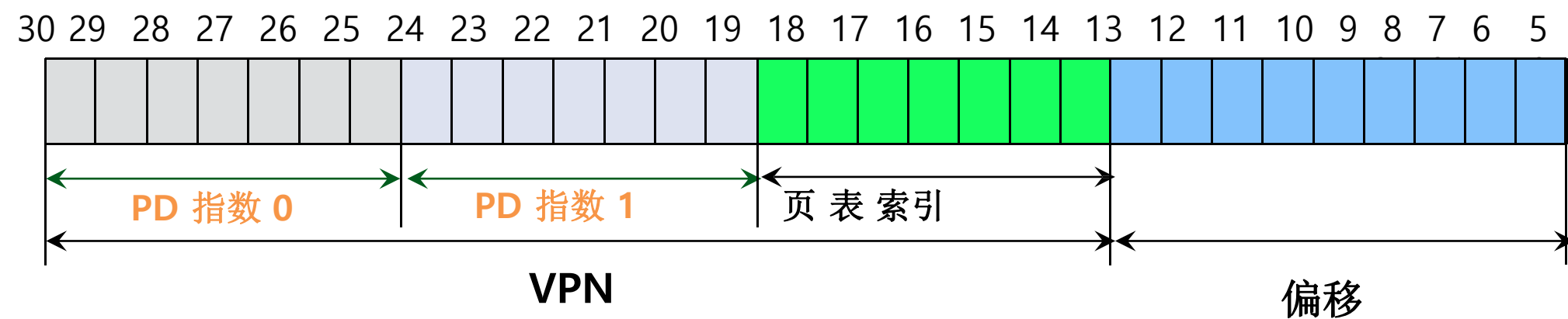
旗	详细内容
虚拟 地址	30 位
页面 大小	512 字节
VPN	21 位
偏置	9 位
每页的页 条目数	128 个PTE

$\log_2 128 = 7$

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 位作为索引。页目录现在有 14 位，显然无法用一页装载下页目录。

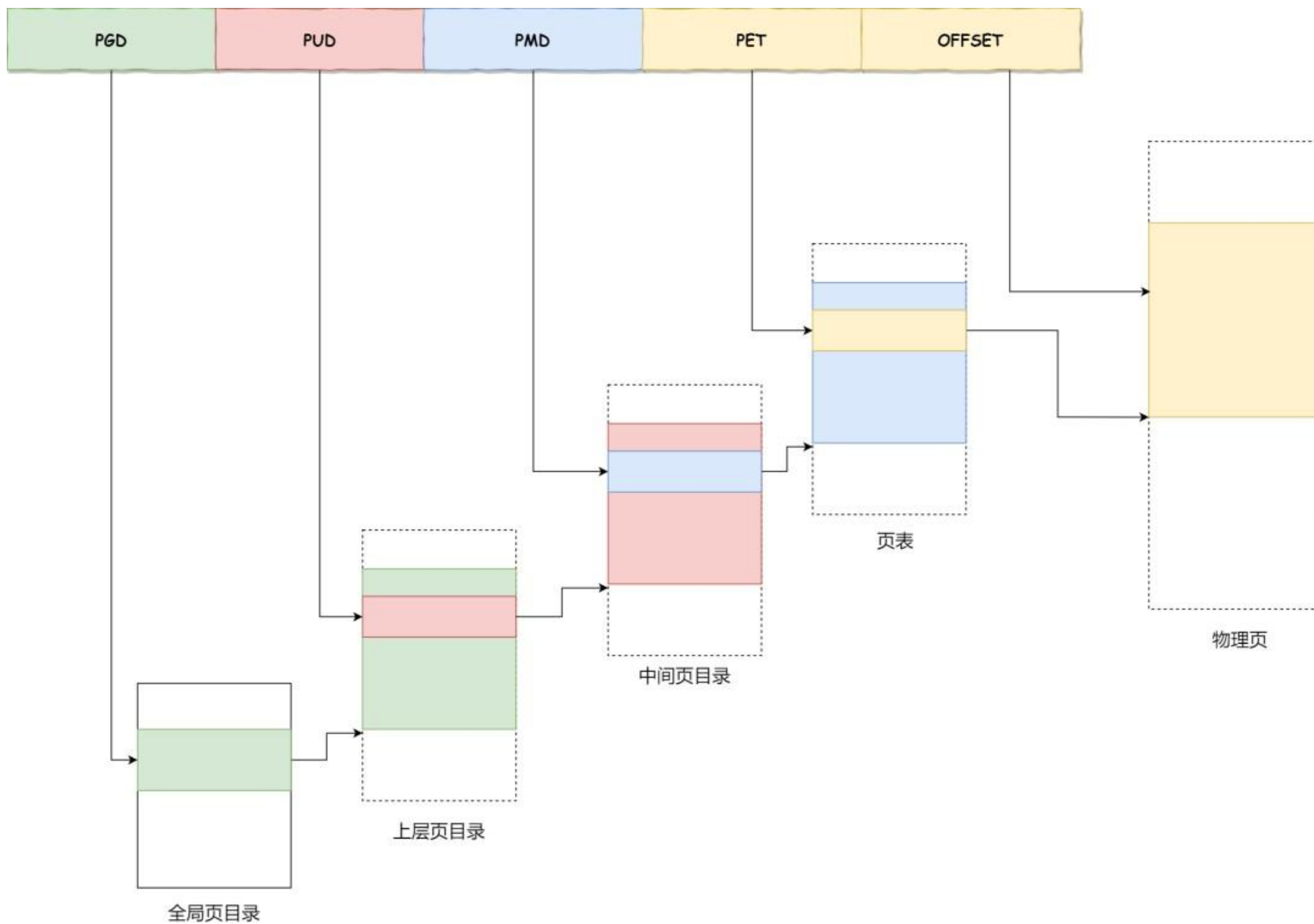
两级以上：页面目录

- 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



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 index 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 index 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

多级页表



某计算机有64位虚地址空间，页大小是2048B.每个页表项长为4B。因为所有页表都必须包含在一页中，故使用多级页表，问一共需要多少级？

$$2048B=2^{11}$$

$64-11=53$ （地址中扣除页内地址位数） 共有 2^{53} 页

一页中可以装 $2048/4=2^9$ 个页表项

$9*6>53$ 至少需要6级页表

已知系统为32位实地址，采用48位虚拟地址，页面大小4KB，页表项大小为8B;每段最大为4GB。若系统采用段页式存储，则每用户最多可以有多少个段，段内采用几级页表

已知系统为32位实地址，采用48位虚拟地址，页面大小4KB，页表项大小为8B;每段最大为4GB。若系统采用段页式存储，则每用户最多可以有多少个段段内采用几级页表

系统采用48位虚拟地址，每段最大为4GB，故段内地址为32位，段号为 $48-32=16$ 位。每个用户最多可以有 2^{16} 个段。段内采用页式地址，与1)中计算同理， $(32-12)/9$ ，取上整为3，故段内应采用3级页表。

多级 页表控制 流程

```
01:    VPN = (VirtualAddress&VPN_MASK) >> SHIFT
02:    (成功, TlbEntry) =TLB_Lookup (VPN)
03:    if (true  =  True )        //TLB 命中
04:        if (CanAccess (TlbEntry.ProtectBits) ==True)
05:            偏移=虚拟地址&OFFSET_MASK
06:            PhysAddr = (TlbEntry.PFN<<SHIFT) | 偏移
07:            寄存器= AccessMemory (PhysAddr)
08:            System . out . println ( ) ;
09:    else//执行完整的多级 查找
```

- ◆ (1 行) 提取虚拟页码 (VPN)
- ◆ (2 行) 检查TLB 是否 持有此VPN 的 transalation
- ◆ (5-8行) 从相关TLB条目中提取页帧 号 , 并形成所需的 物理 地址 和访问存储器

多级 页表控制 流程

```
十一:  否则
十二:      PDIndex = (VPN & PD_MASK) >> PD_SHIFT
十三:      PDEAddr = PDBR + (PDIndex * sizeof(PDE))
十四:      PDE = 访问存储器(PDEAddr)
十五:      if (PDE.Valid == False)
十六:          RaiseException(SEGMENTATION_FAULT)
十七:      else // PDE 有效: 现在 从PT获取PTE
```

- ◆ (11行) 提取页 目录 索引 (PDIndex)
- ◆ (13行) 获取页 目录 项 (PDE)
- ◆ (15-17行) 检查PDE有效标志。如果 有效标志 为 真, 则从页表中获取页表条
目

翻译过程： 记住 TLB

```
十八: PTIndex = (VPN&PT_MASK) >> PT_SHIFT
十九: PTEAddr = (PDE.PFN_SHIFT<<) + (PTIndex *sizeof (PTE) )
二十: PTE =访问存储器 (PTEAddr)
    21:
        if (PTE.Valid ==False)
二十二:     RaiseException (SEGMENTATION_FAULT)
二十三:     else if (CanAccess (PTE.ProtectBits) ==False)
二十四:         System . out. println ( ) ;
    25: 否则
    26:     TLB_Insert (VPN, PTE.PFN, PTE.ProtectBits)
    27:
    28:     RetryInstruction ( )
    29:
    30: 
```

反向页表

- 在页表的世界中，一个更极端的空间节省是使用**反转页表**。
- 在这里，我们没有使用许多页表（系统的每个进程一个），而是保留**一个页表**，该表为系统的每个**物理页**都有一个条目。
- 该条目告诉我们哪个进程正在使用这个页面，以及该进程的哪个虚拟页面映射到这个物理页面。

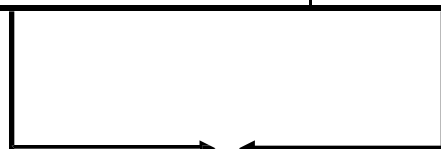
- 找到正确 的 条目现在 是 在这个 数据 结构中搜索 的问题。
- 线性 扫描将是昂贵的 ， 因此 通常在基础结构上 构建哈希表以 加速查找。
- **PowerPC** 就是 这种体系 结构的一个例子

- 倒排页表说明了我们从一开始就说过的：页表只是数据 结构 。
- 你可以用数据结构做很多疯狂的事情，使它们更小或更大，使它们更慢或更快。
- 多级和倒排页表只是可以做的许多事情中的两个例子。

反置页表

逻辑地址寄存器

进程标识符	页号	页内地址
-------	----	------



物理块号	PID/页号
0	
1	
...	
	<i>pid/page</i>
	...

区块编号	块内地址
------	------

物理地址寄存器

反置页表

Accessing Hashed Inverted Page Table

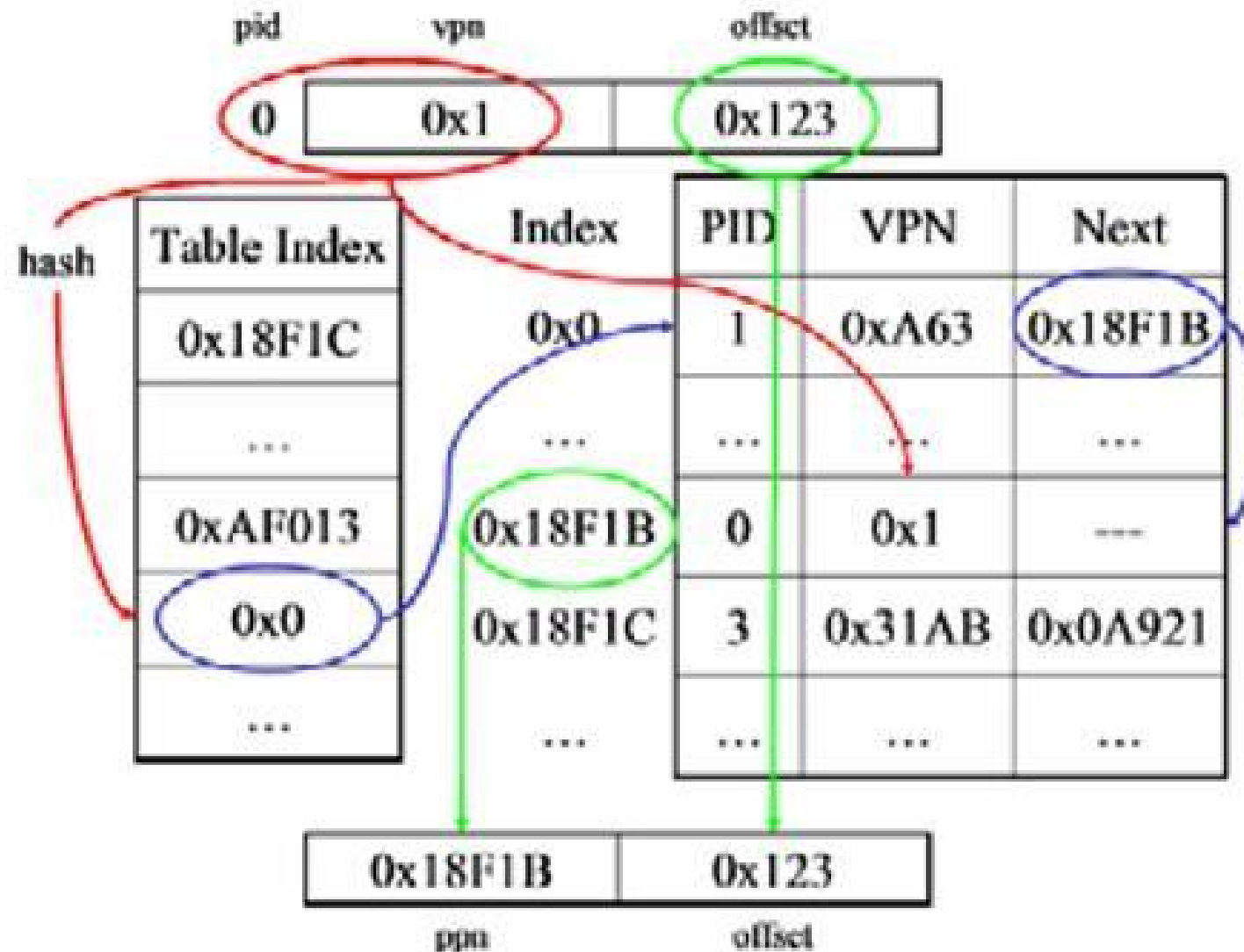


Image from <http://www.cs.berkeley.edu>

将页号`vpn`做一个`hash`计算（使用硬件加速），得到一个页帧号，也就是反向页表的`index`，但是这样可能会存在`hash`冲突，因此在传参的时候需要传入当前进程的`PID`作为标识，以确保找到对应的页帧号，`NEXT`中存放的是由于`hash`冲突导致的相同`hash`值的下一个条目的`index`（页帧号），这样就不需要一个一个进行比对了，如图，0x1经过计算得到0x0，因此去访问反向页表的`index`为0x0这个条目，比对之后发现`PID`不对，因此访问`NEXT`，找到相同`hash`值的下一个条目.....找到之后，`index+offset`即为物理访问地址

SummaR Y

- 在 内存受限的系统中（就像许多较老的系统一样），小的结构是有意义的;
- 在 具有合理内存量 和 主动使用 大量 页面的工作负载的系统中，可以加速TLB 未命中的较大表 可能是正确的 选择。

已知系统为32位实地址，采用48位虚拟地址，页面大小4KB，页表项大小为8B;每段最大为4GB。

- 1 假设系统使用纯页式存储，则要采用多少级页表，页内偏移多少位？
- 2 假设系统采用一级页表，TLB命中率为98%，TLB访问时间为10ns，内存访问时间为100ns，并假设当TLB访问失败后才访问内存，问平均页面访问时间是多少？
- 3 如果是二级页表，页面平均访问时间是多少？
- 4 上题中，如果要满足访问时间 $\leq 120\text{ns}$ ，那么命中率需要至少多少？

1 页面大小为4KB，故页内偏移需要12位来表示。系统虚拟地址一共48位，所以剩下的 $48-12=36$ 位可以用来表示虚页号。每一个页面可以容纳的页表项为： $4KB/8B=2^9$ （也就是可以最多表示到9位长的页号），而虚页号的长度为36位，所以需要的页表级数为： $36/9=4$ 级。

2 当进行页面访问时，首先应该先读取页面对应的页表项，98%的情况可以在TLB中直接得到得到页表项，直接将逻辑地址转化为物理地址，访问内存中的页面。如果TLB未命中，则要通过一次内存访问来读取页表项，所以页面平均访问时间是：

$$98\% \times (10+100) \text{ ns} + 2\% \times (10+100+100) \text{ ns} = 112 \text{ ns}$$

3 二级页表的情况下：如果TLB命中，和（2）的情况一样，如果TLB没有命中，采用二级页表需要访问3次内存，所以页面平均访问时间是： $98\% \times (10+100) \text{ ns} + 2\% \times (10+100+100+100) \text{ ns} = 114 \text{ ns}$

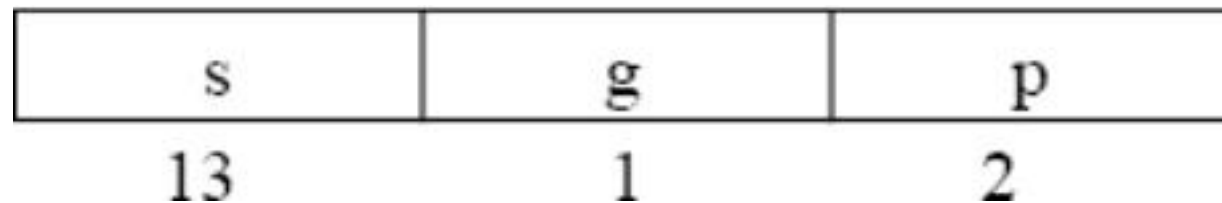
（4）假设TLB的命中率为 p ，应该满足以下式子： $p \times (10+100) \text{ ns} + (1-p) \times (10+100+100+100) \text{ ns} \leq 120 \text{ ns}$

可以解得： $p \geq 95\%$ ，所以如果要满足访问时间 $\leq 120 \text{ ns}$ ，那么命中率至少为95%。

在多级页表的情况下，如果TLB没有命中，则需要从虚拟地址的高位起，每N位（其中N就是类似于（1）中的情况逐级访问各级页表，以第（1）问为例，如果快表未命中，则需要访问5次内存才能得到所需页面。

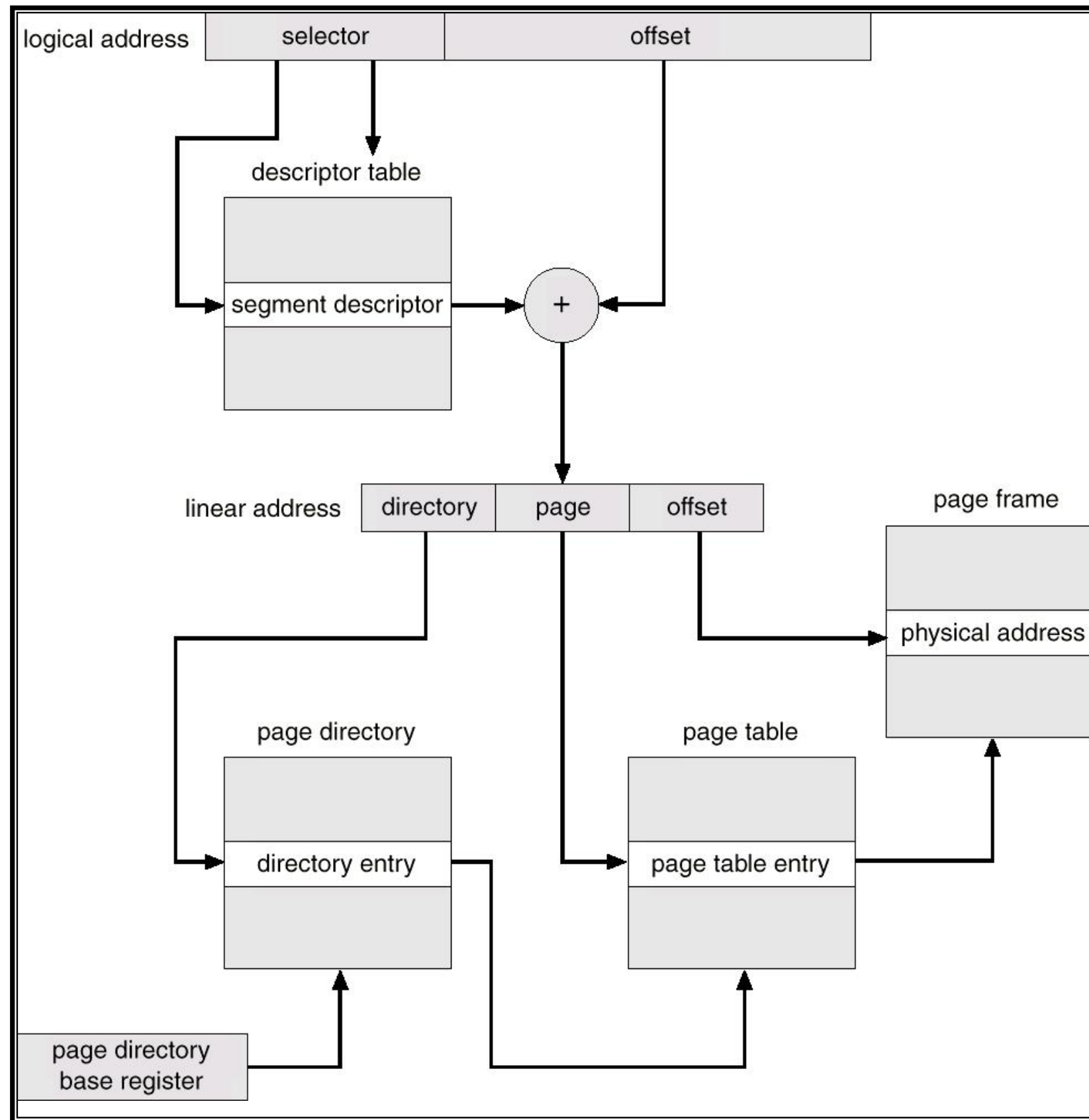
分页分段-英特尔 386

- 逻辑地址是一个二元组（选择符，偏移量），偏移量为32位。
- 选择符（selector）为16位，如下所示：



- 其中，s 指明段号，g 指明这个段是在GDT 中还是在LDT 中，p 用于保护。偏移量为32位，它指明目标在段内的位置。
- 段寄存器指向LDT或GDT中的表项。段的基地址和界限（limit）信息被用来产生一个线性地址（linear address）。首先，用界限值来检查地址是否合法。如果地址不合法就产生异常（memory fault），向操作系统发出中断。如果合法，那么将偏移量与基地址相加得出32位线性地址。
- 然后把这个线性地址按分页的思想转换为物理地址。

英特尔 386 地址转换



结束