Virtual Memory

TLBs & Smaller Page Tables

Liu yufeng

Fx_yfliu@163.com

Hunan University

- Using paging as the core mechanism to support virtual memory can **lead to high performance overheads**.

- By chopping the address space into small, fixed-sized units (i.e., pages), paging requires a large amount of mapping information.

- Because that mapping information is generally stored in physical memory, paging logically requires an extra memory lookup for each virtual address generated by the program.

- Going to memory for translation information before every instruction fetch or explicit load or store is **prohibitively slow**.

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？

- To speed address translation, we are going to add what is called a **translation-lookaside buffer**, or **TLB**.

- A TLB is part of the chip’s **memory-management unit** (**MMU**), and is simply a hardware cache of popular virtual-to-physical address translations.

- Upon each virtual memory reference, the hardware first checks the TLB to see if the desired translation is held; if so, the translation is performed (quickly) **without having to consult the page table**.

TLB Basic Algorithm

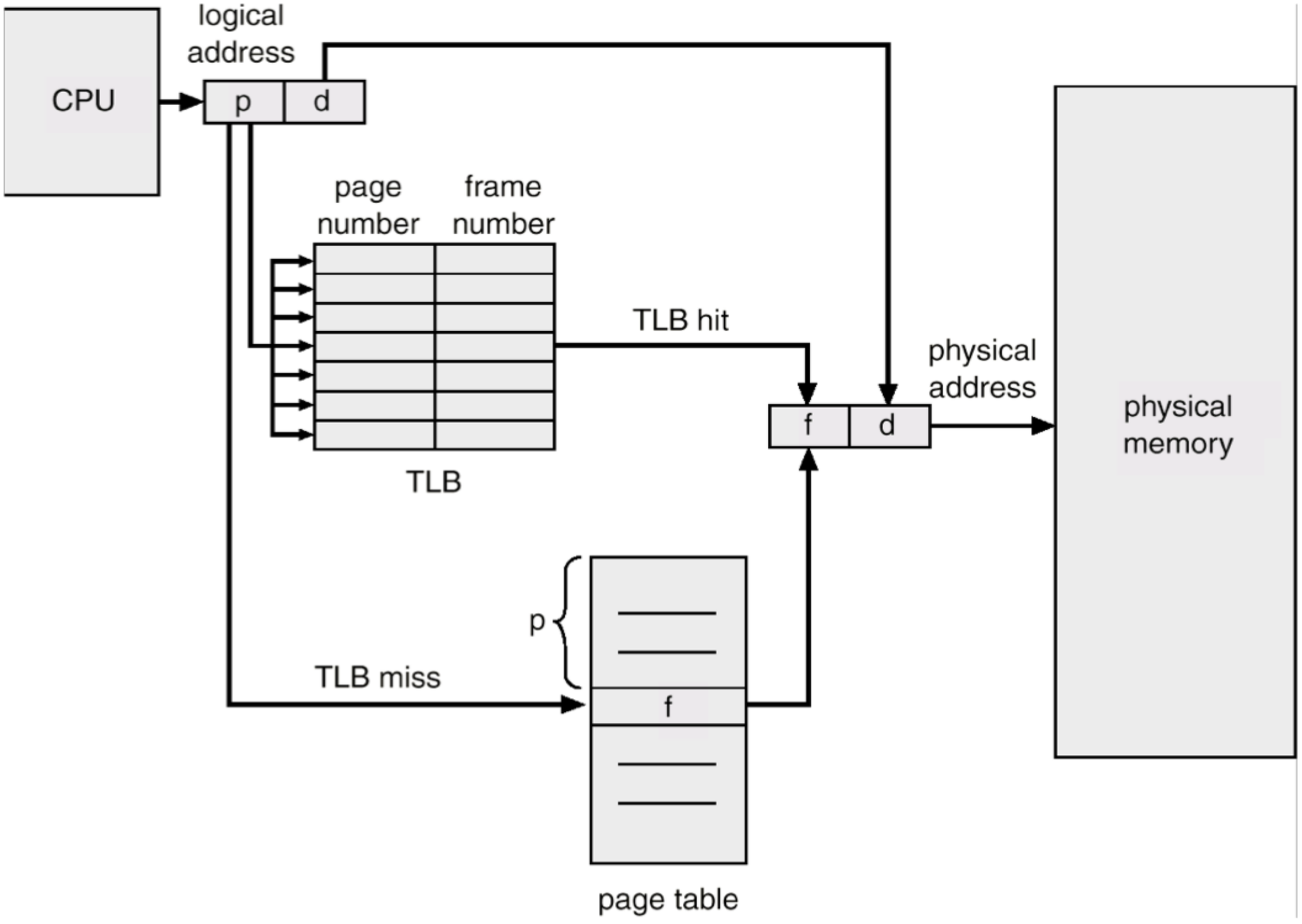
- The following Figure shows a rough sketch of how hardware might handle a virtual address translation, assuming a simple **linear page table** (i.e., the page table is an array) and a **hardware-managed TLB**.

page-table entry (PTE)

PFN(页帧号)

```
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


CPU
logical address
p
d
page number
frame number
TLB hit
TLB
f
d
physical address
physical memory
p
TLB miss
f
page table

- The TLB, like all caches, is built on the premise that in the **common case**, translations are found in the cache.

- If so, little overhead is added, as the TLB is near the processing core and is designed to be quite fast.

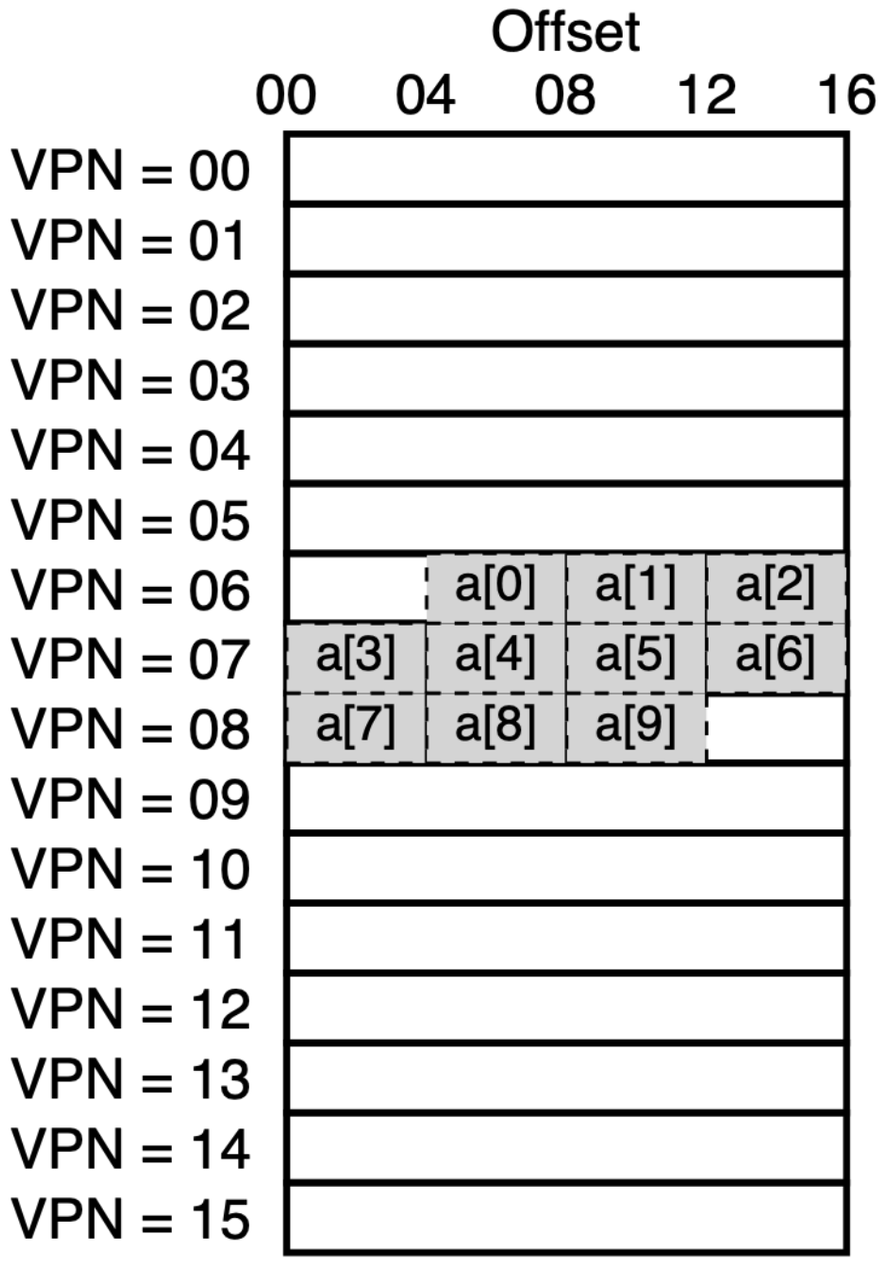
- When a miss occurs, the high cost of paging is incurred; the page table must be accessed to find the translation.

- If this happens often, the program will likely run noticeably more slowly; **memory accesses are quite costly**, and TLB misses lead to more memory accesses.

- Thus, it is our hope to **avoid TLB misses as much as we can**.

Example: Accessing An Array

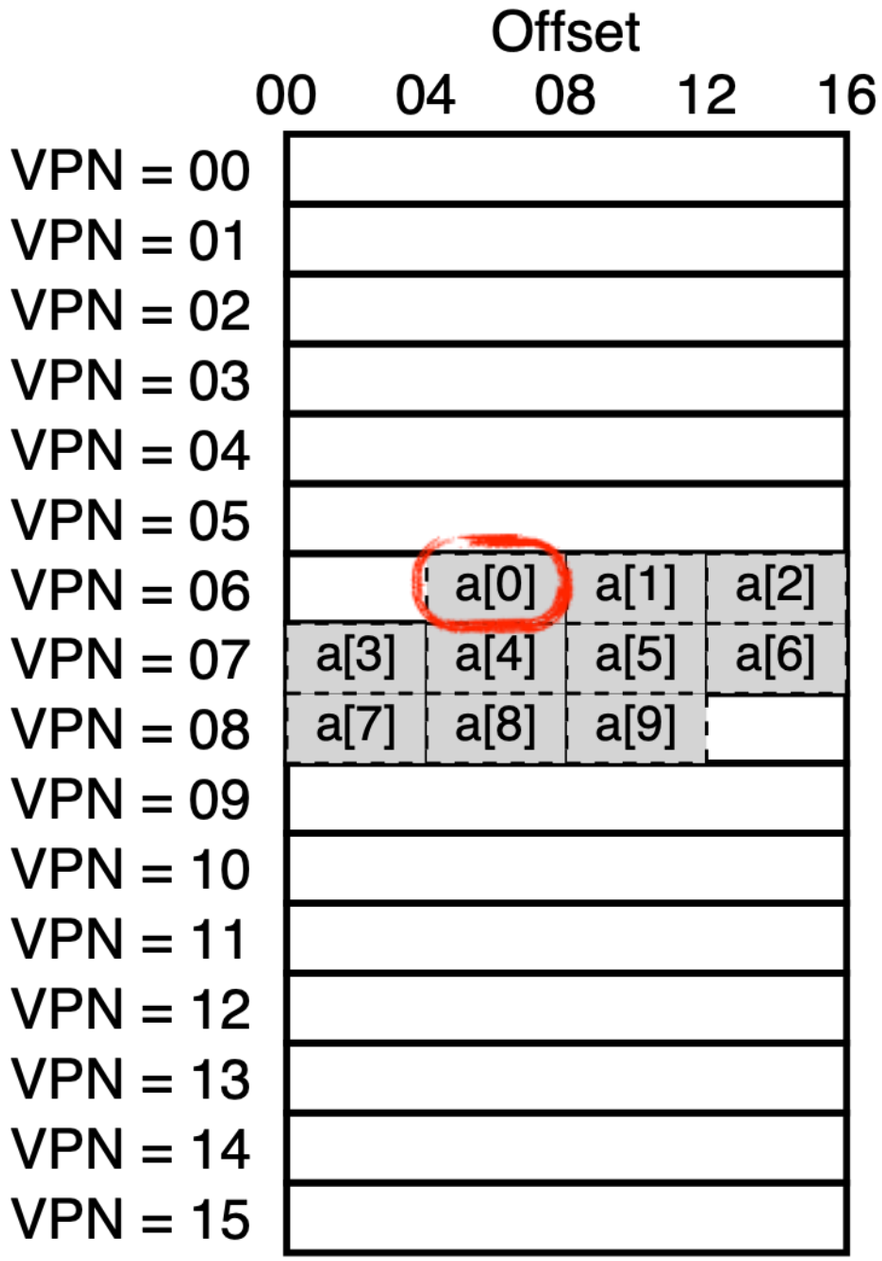
- 假定，有一个 8 位的虚地址空间，页大小为 16B。 我们可以把虚地址划分为 4 位的 VPN（有 16 个虚拟内存 页）和 4 位的偏移量（每个页中有 16 个字节）。
- 在系统的 16 个 16 字节 的页上数组的第一项（a[0]）开始于（VPN=06，offset=04），只有 3 个 4 字节整型数存放在该页。数组在下 一页（VPN=07）继续，其中有接下来 4 项（a[3] ··· a[6]）。 10 个元素的数组的最后 3 项（a[7] ··· a[9]）位于地址空间 的下一页（VPN=08）。



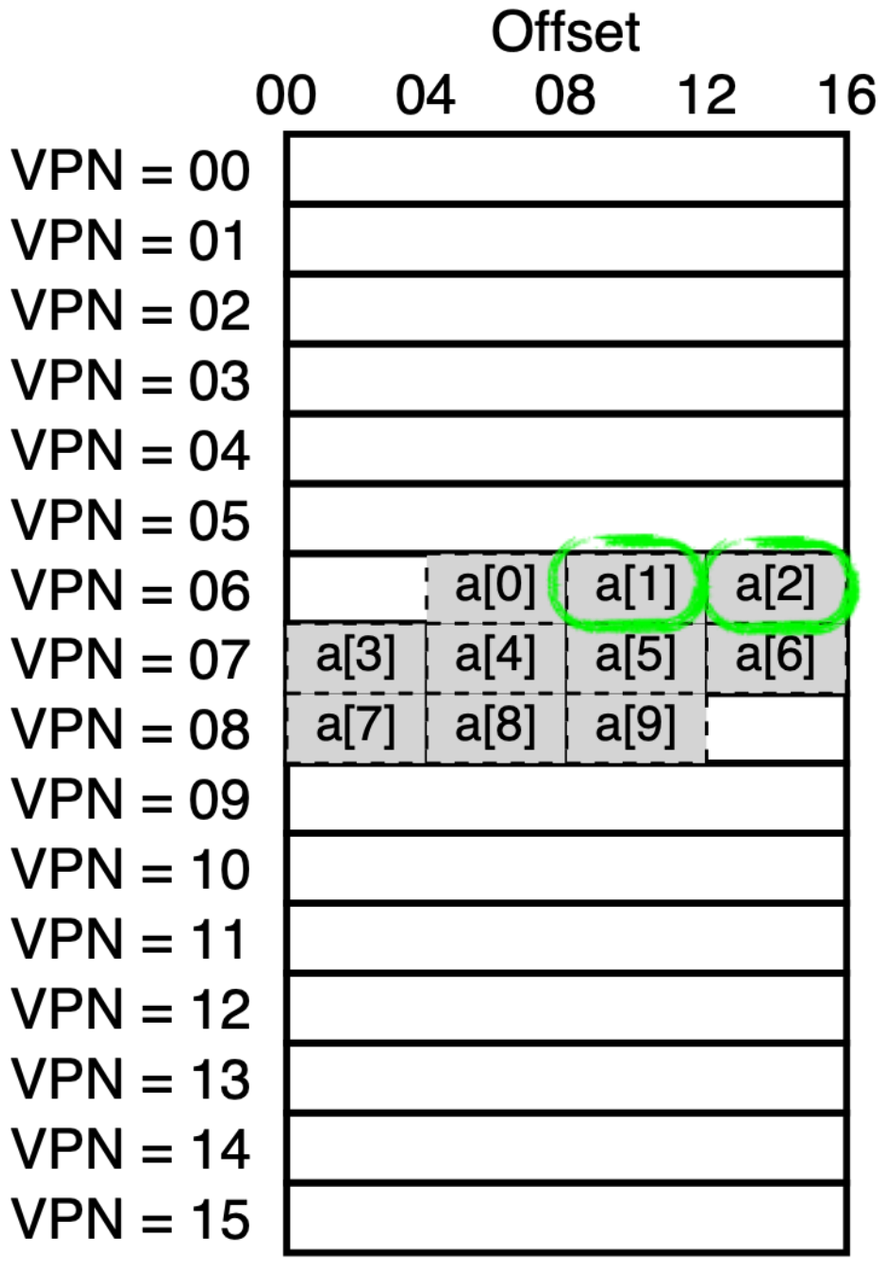
- Now let’s consider a simple loop that accesses each array element:

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

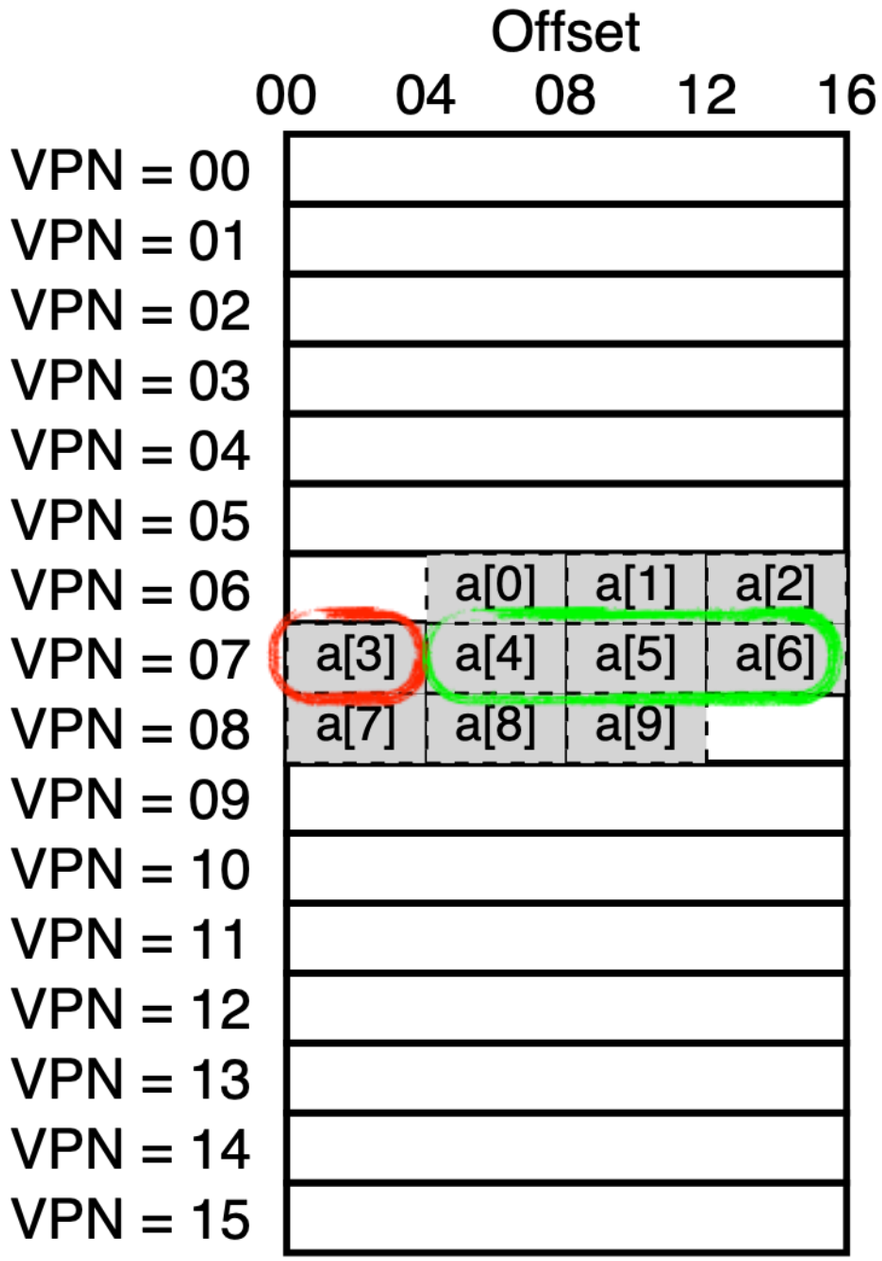
- 当访问第一个数组元素（a[0]）时，CPU 会看到载入虚存地址 100。硬件从中提取 VPN（VPN=06），然后用它来检查 TLB，寻找有效的转换映射。假设这里是程序第一次访 问该数组，结果是 TLB 未命中。



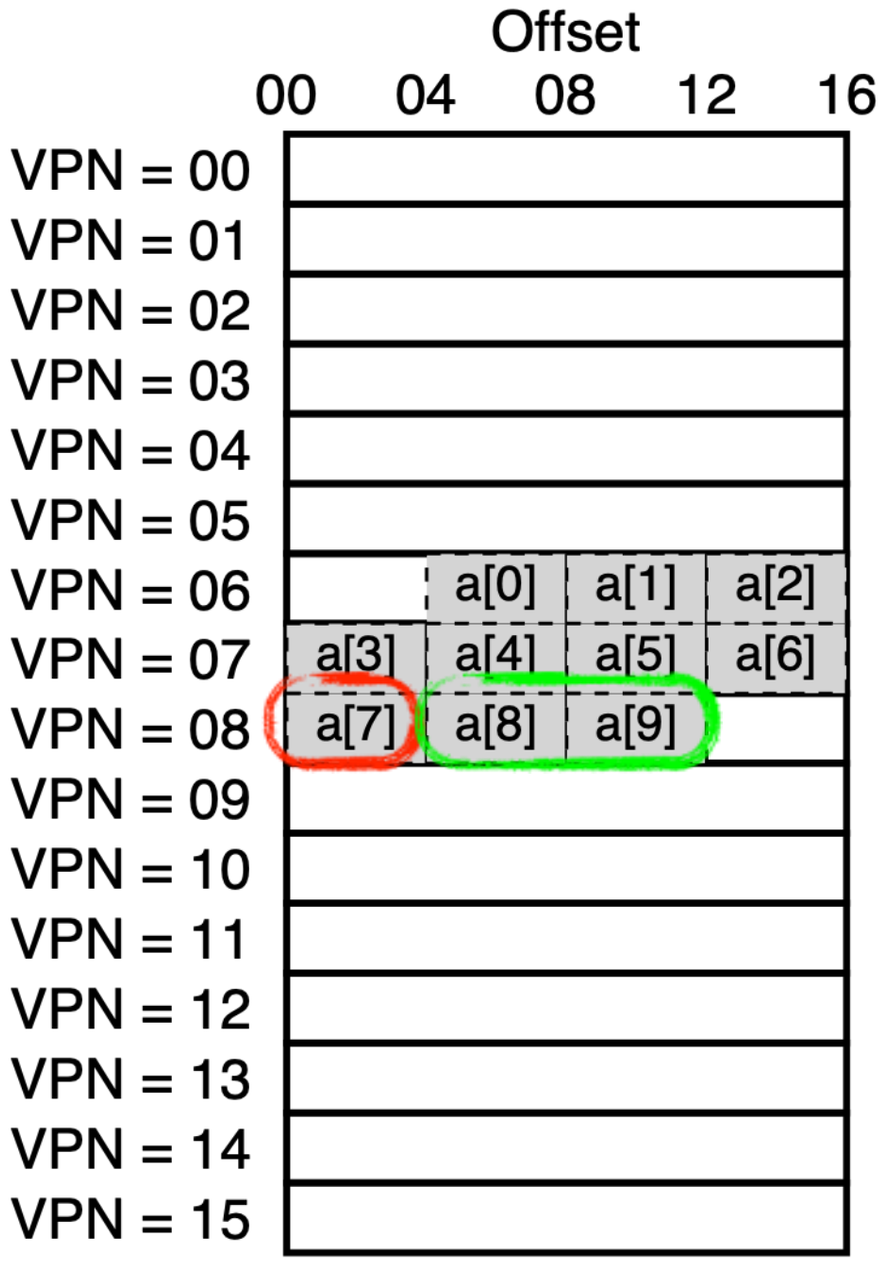
- 接下来访问 a[1]，这里有好消息：TLB 命中！因为数组的第二个元素在第一个元素之后，它们在同一页。因为我们之前访问数组的第一个元素时，已经访问了这一页，所以 TLB 中缓存了该页的转换映射。因此成功命中。访问 a[2] 同样成功（再次命中），因为它和 a[0]、 a[1] 位于同一页。



- 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。



- Let us summarize TLB activity during our ten accesses to the array: **miss, hit, hit, miss, hit, hit, hit, miss, hit, hit**. Thus, our TLB **hit rate**, which is the number of hits divided by the total number of accesses, is **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.

- If the page size had simply been **twice as big** (**32** bytes, not **16**), the array access would suffer even fewer misses.

- As typical page sizes are more like **4KB**, these types of dense, array-based accesses achieve excellent TLB performance.

- If the program, soon after this loop completes, accesses the array again, we’ d see an even better result: **hit, hit, hit, hit, hit, hit, hit, hit, hit, hit**.

- In this case, the TLB hit rate would be high because of **temporal locality**（时间局部性）.

- If caches (like the TLB) are so great, **why don’t we just make bigger caches** and keep all of our data in them?

- Unfortunately, this is where we run into more **fundamental laws** like those of **physics**. If you want **a fast cache, it has to be small**, as issues like the speed-of-light and other physical constraints become relevant.

- Any large cache by definition is slow, and thus defeats the purpose.

- Thus, we are stuck with small, fast caches; the question that remains is **how to best use them to improve performance**.

Who Handles The TLB Miss?

- The **hardware-managed TLB** like the Intel x86 architecture.

- 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.

Who Handles The TLB Miss?

- The **software-managed TLB** like the Sun’ s SPARK V9.

- 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 Contents: What's In There?

- A typical TLB might have 32, 64, or 128 entries and be what is called **fully associative**.

- Basically, this just means that any given translation can be anywhere in the TLB, and that the hardware will search the entire TLB in parallel to find the desired translation.

- A TLB entry might look like this:

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 Issue: Context Switches

- 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)**.

- When switching from one process to another, the hardware or OS (or both) **must be careful to ensure** that the about-to-be-run process does not accidentally use translations from some previously run process.

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

- In the TLB above, we clearly have a problem: VPN 10 translates to either PFN 100 (P1) or PFN 170 (P2), but the hardware can’t distinguish which entry is meant for which process.

- Thus, we need to do some more work in order for the TLB to correctly and efficiently support virtualization across multiple processes.

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？

- One approach is to **flush** the TLB on context switches, emptying it before running the next process.

- However, there is a cost: each time a process runs, it must **incur TLB misses**.

- To reduce this overhead, some systems add hardware support to **enable sharing of the TLB across context switches**. In particular, some hardware systems provide an **address space identifier** (**ASID**) field in the TLB.

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

- With address-space identifiers, the TLB can hold translations from different processes at the same time without any confusion.

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

- In this example, there are two entries for two different processes with two different VPNs that point to **the same physical page**.

- This situation might arise, for example, when two processes **share a page** (a code page, for example).

- Sharing of code pages (in binaries, or shared libraries) is useful as it reduces the number of physical pages in use.

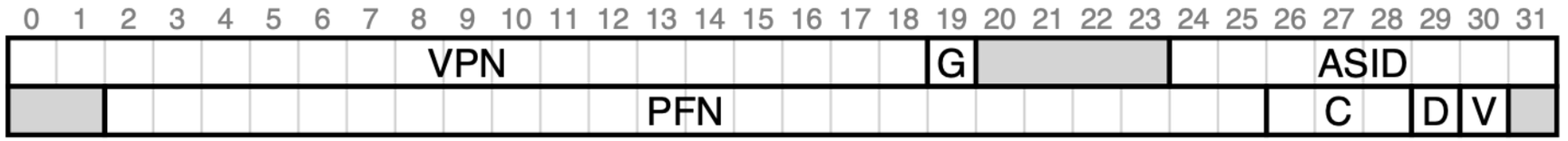
Issue: Replacement Policy

- When installing a new entry in the TLB, we have to **replace** an old one, and which one to replace?

- The goal is to minimize the **miss rate** and thus improve performance.

- 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。

A Real TLB Entry



A MIPS TLB Entry

- This example is from the **MIPS R4000**, a modern system that uses software-managed TLBs.
- The MIPS R4000 supports a **32-bit** address space with **4KB** pages.
- Thus, we would expect a 20-bit VPN and 12-bit offset in our typical virtual address.
- 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 pages)
- 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

Effective Access Time

- 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），那么我们必须要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%):
- 有效内存访问时间 = 0.80 × 120 + 0.20 × 220 = 140 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:
- 有效内存访问时间= 0.98 × 120 + 0.02 × 220 = 122 纳秒。
- 这个命中率仅产生了22%的访问时间延迟。

Summary

- With a small on-chip TLB as an address-translation cache, most memory references will be handled **without having to access the page table** in main memory.

- In the common case, the performance of the program will be almost as if memory isn’t being virtualized at all.

- TLB is **essential to the use of paging** in modern systems.

Smaller Page Tables

- We now tackle the second problem that paging introduces: **page tables are too big** and thus consume too much memory.

- Recall that linear page tables get pretty big. Assume again a 32-bit address space (2^{32} bytes), with 4KB (2^{12} byte) pages and a 4-byte page-table entry.

- An address space thus has roughly one million virtual pages in it; our page table is **4MB** in size.

- With **a hundred active processes**, we will be allocating hundreds of megabytes of memory just for page tables!

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?

Simple Solution: Bigger Pages

- We could reduce the size of the page table in one simple way: **use bigger pages**.

- 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)**.

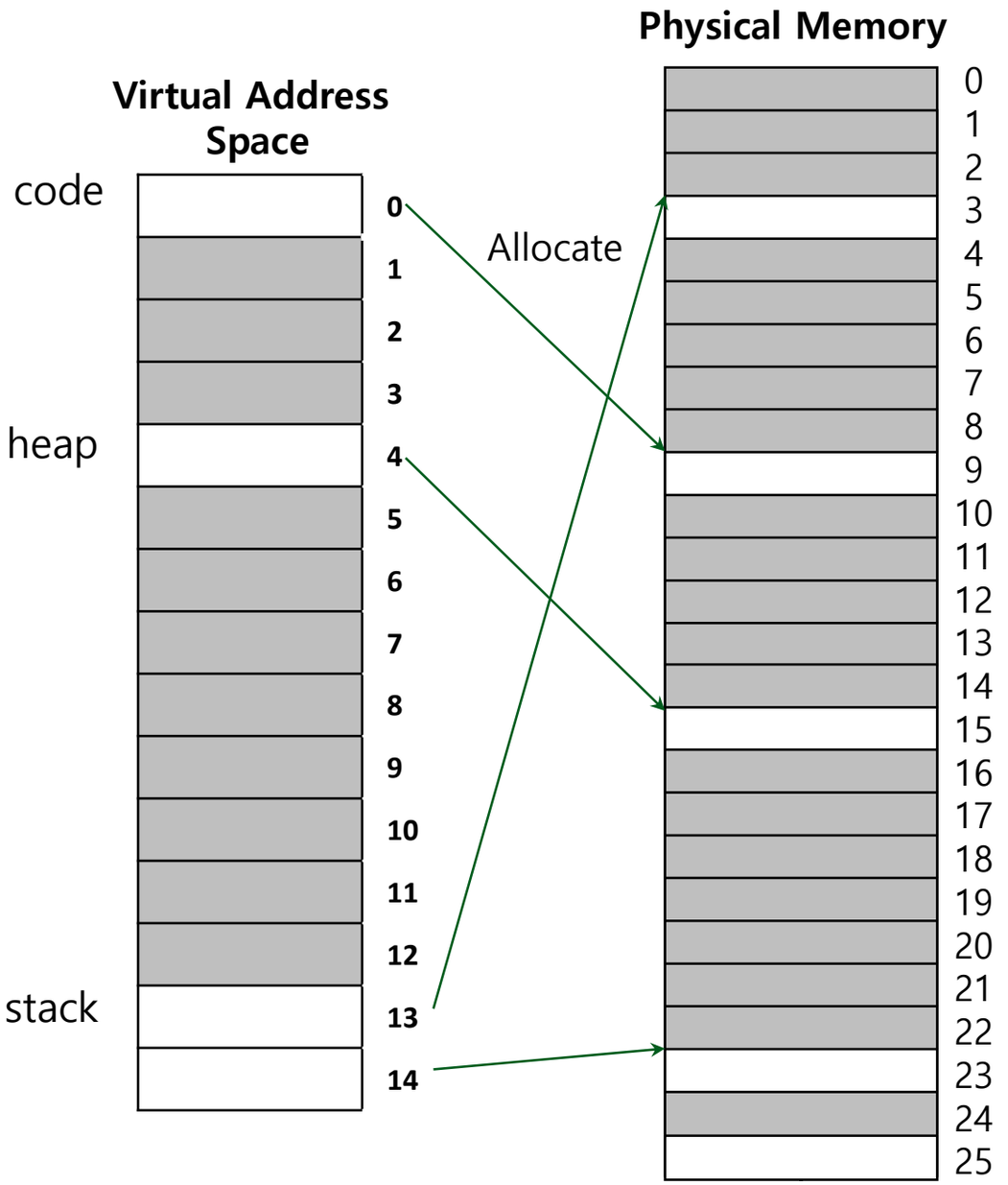
- 再以 32 位地址空间为例，但这 次假设用 16KB 的页。因此，会有 18 位的 VPN 加上 14 位的偏移量。假设每个页表项（4 字节）的大小相同，现在线性页表中有 2^{18} 个项，因此每个页表的总大小为 1MB，页表缩 到四分之一.

Bigger Pages

- 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。

Problem

- Single page table for the entries address space of process.



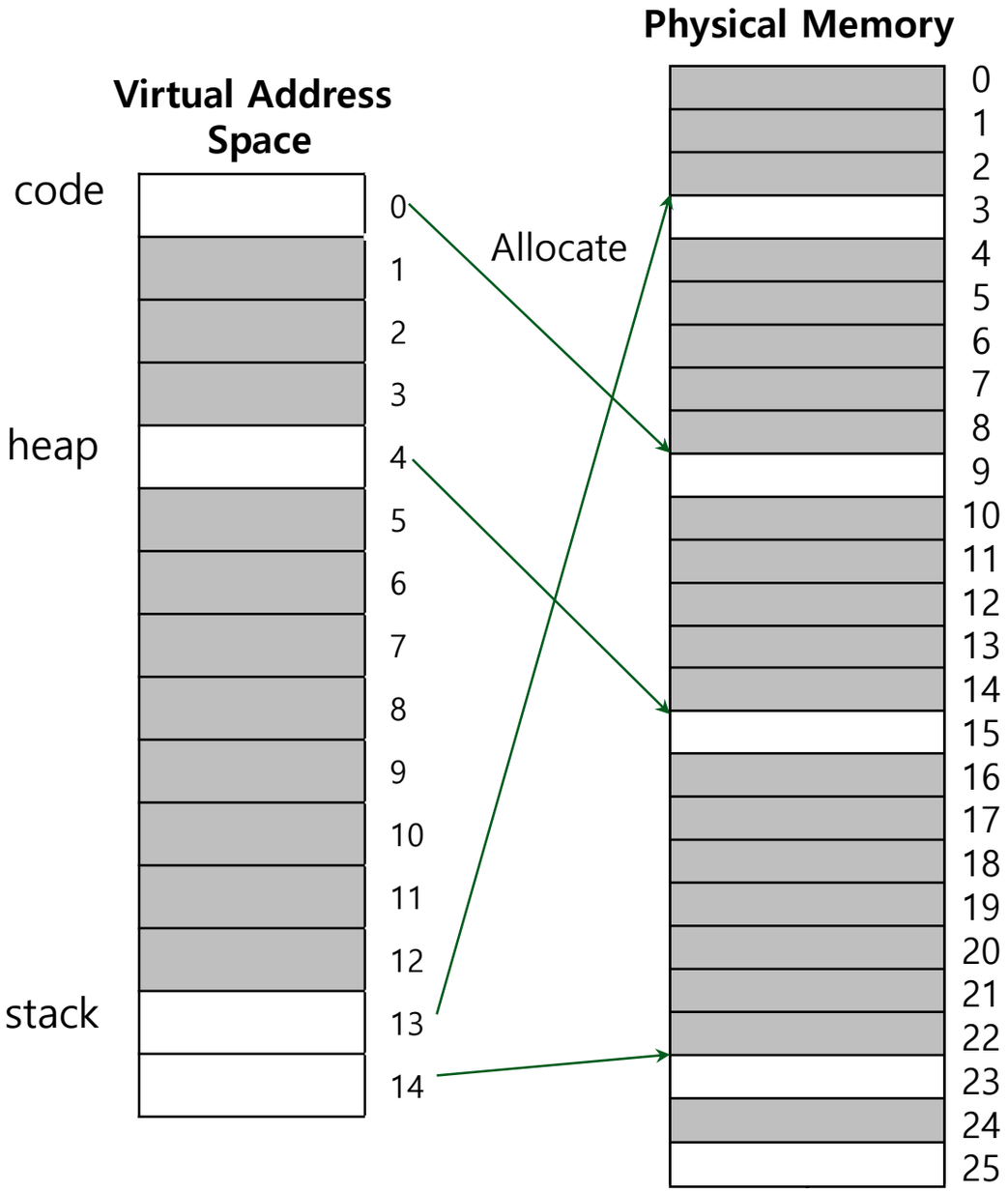
A 16KB Address Space with 1KB Pages

PFN	valid	prot	present	dirty
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

A Page Table For 16KB Address Space

Problem

- Most of the page table is **unused**, full of invalid entries.



A 16KB Address Space with 1KB Pages

PFN	valid	prot	present	dirty
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

A Page Table For 16KB Address Space

Hybrid Approach: Paging and Segments

- Whenever you have two reasonable but different approaches to something in life, you should always examine the combination of the two to see if you can obtain the best of both worlds. We call such a combination a **hybrid**.

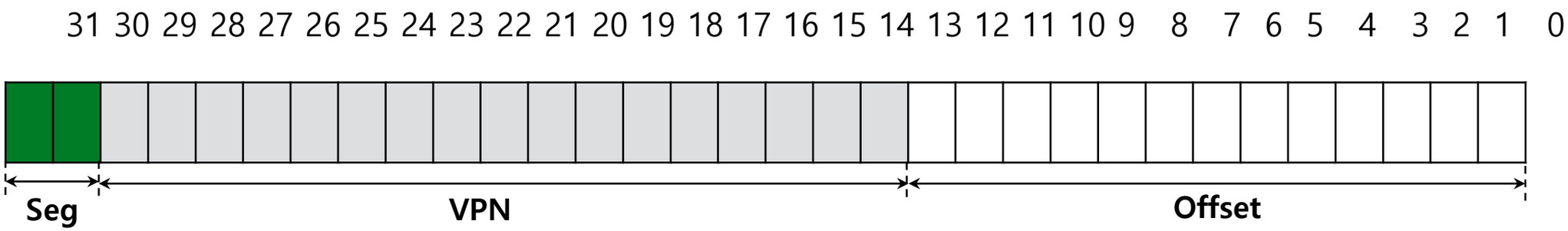
- The creators of **Multics** (in particular **Jack Dennis**) chanced upon such an idea in the construction of the Multics virtual memory system.

Hybrid Approach: Paging and Segments

- In order to reduce the memory overhead of page tables.
  - Using base not to point to the segment itself but rather to hold the physical address of the page table of that segment.
  - The bounds register is used to indicate the end of the page table.

Simple Example of Hybrid Approach

- 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



32-bit Virtual address space with 4KB pages

Seg value	Content
00	unused segment
01	code
10	heap
11	stack

TLB miss on Hybrid Approach

- The hardware get to **physical address** from **page table**.
  - The hardware uses the segment bits**(SN) to determine which base and bounds pair to use**.
- The hardware then takes the physical address therein and combines it with the VPN as follows to form the address of the page table entry(PTE) .

```
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

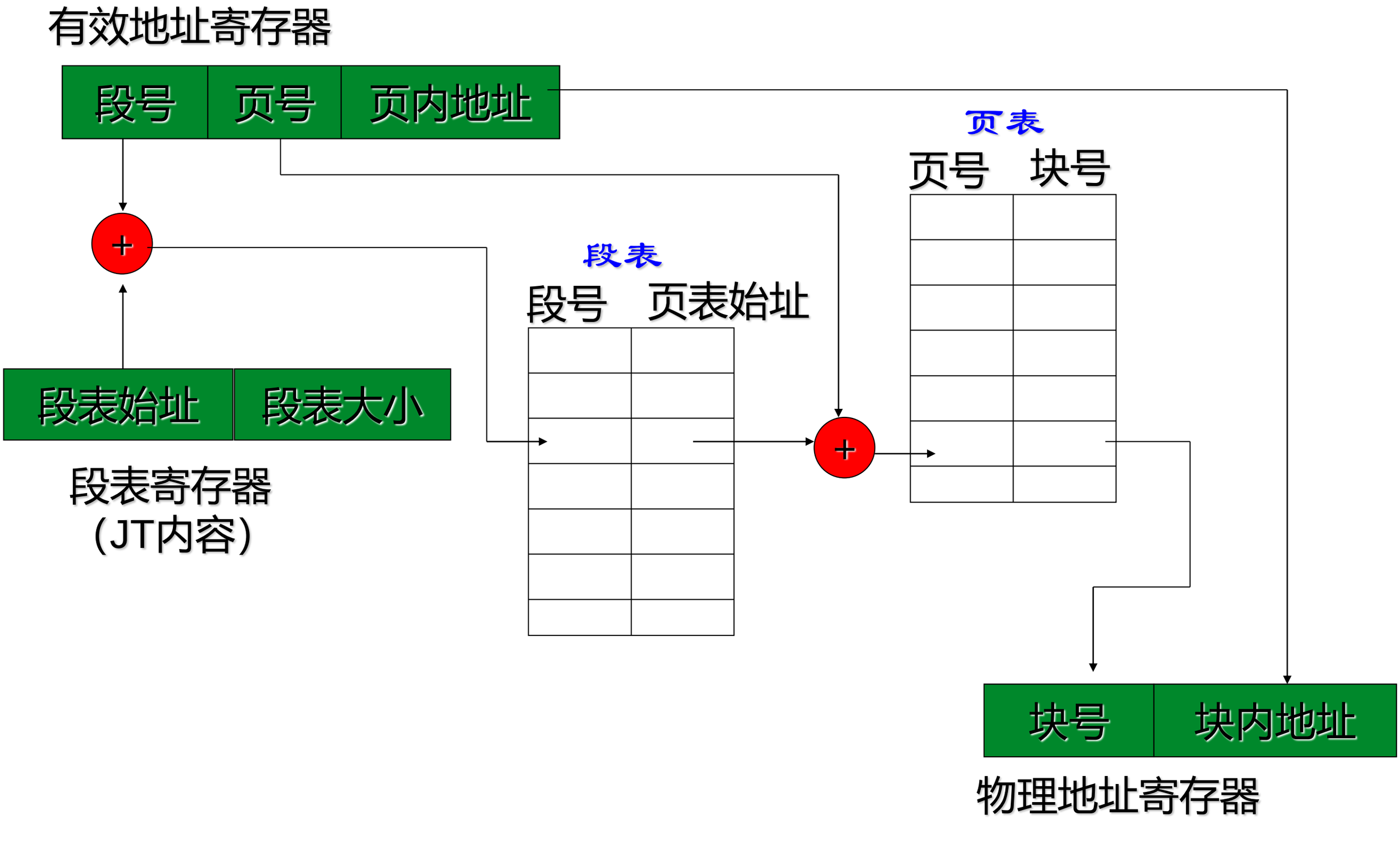
- 段内页表

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



Problems with Hybrid Approach

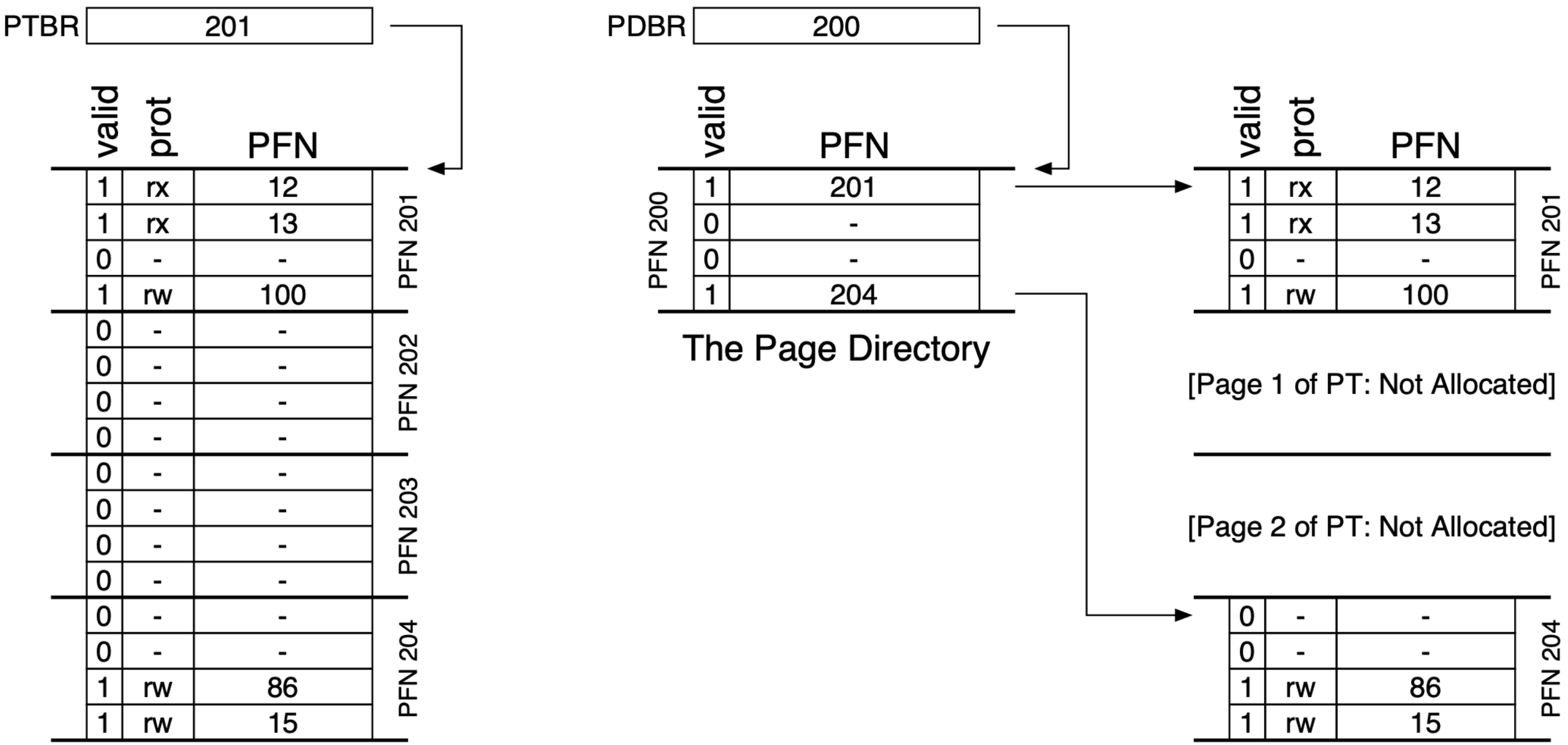
- 首先，它仍然要求使用分段。正如 我们讨论的那样，分段并不像我们需要的那样灵活，因为它假定地址空间有一定的使用模 式。
  - 例如，如果有一个大而稀疏的堆，仍然可能导致大量的页表浪费。
- 其次，导致外部碎片再次出现。尽管大部分内存是以页面大小单位管理的，但页表现在可以是任意大小（是PTE 的倍数）。因此，在内存中为它们寻找自由空间更为复杂。
- 页表的问题：作为线性结构的页表要求分配连续的物理内存

Multi-level Page Tables

- **First**, chop up the page table into page-sized units; **then**, if an entire page of page-table entries (PTEs) is invalid, don’t allocate that page of the page table.

- To track whether **a page of the page table** is valid, use a new structure, called the **page directory**.

- The page directory **either** can be used to tell you where a page of the page table is, **or** that the entire page of the page table contains no valid pages.



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

Advantages of Multi-level Page Tables

- 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

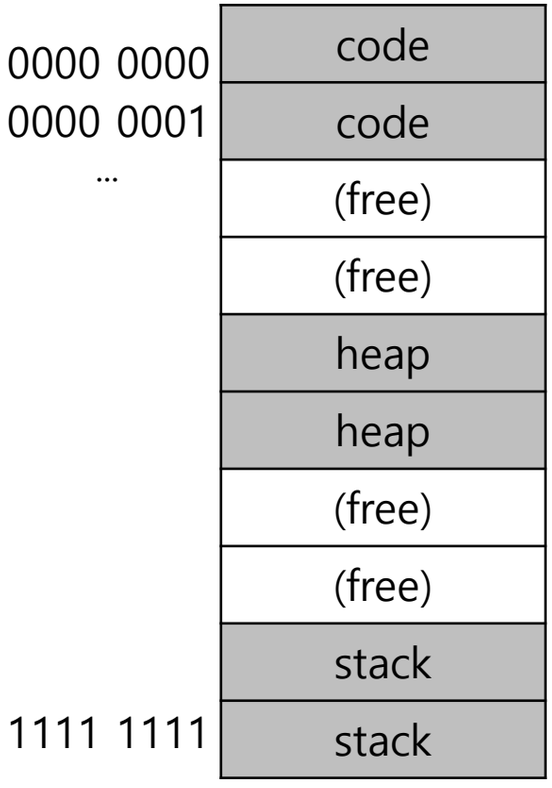
Issues with Multi-level Page Tables

- On a TLB miss, **two loads** from memory will be required to get the right translation information from the page table, **in contrast to** just **one load** with a linear page table.

- Another obvious negative is **complexity** (whether it is the hardware or OS handling the page-table lookup).

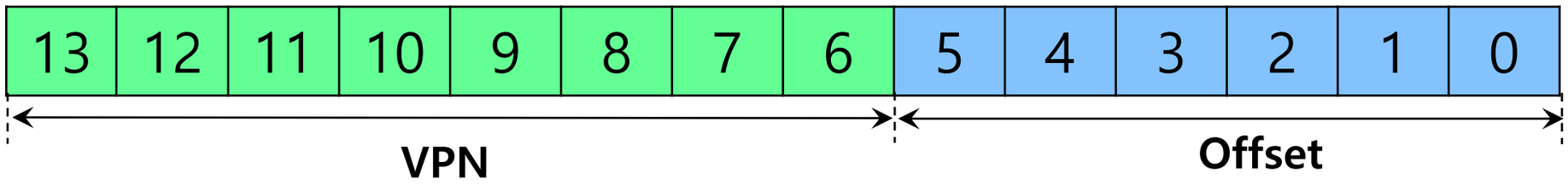
A Detailed Multi-Level Example

- To understand the idea behind multi-level page tables better, let's do an example.



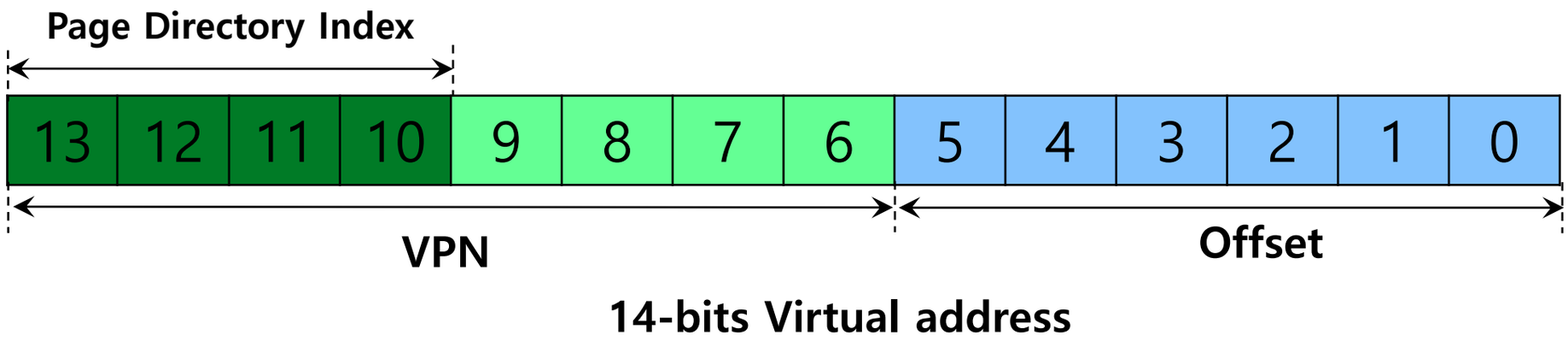
Flag	Detail
Address space	16 KB
Page size	64 byte
Virtual address	14 bit
VPN	8 bit
Offset	6 bit
Page table entry	$2^8(256)$

A 16-KB Address Space With 64-byte Pages



A Detailed Multi-Level Example: Page Directory Idx

- The page directory needs one entry per page of the page table
  - it has 16 entries.
- The page-directory entry is invalid → Raise an exception (The access is invalid)



PDEAddr = PageDirBase + (PDIndex * sizeof(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

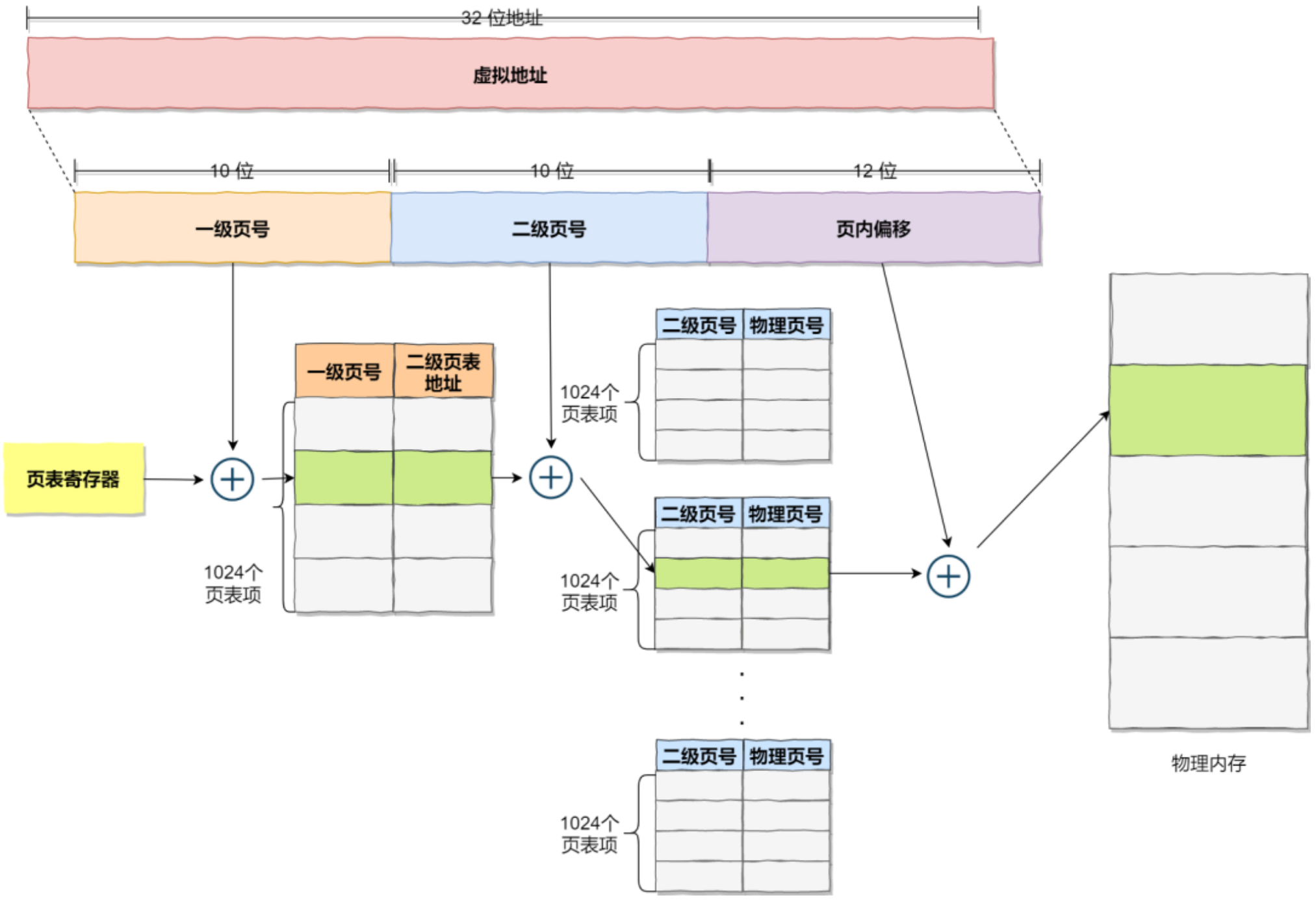
Page Directory		Page of PT (@PFN:100)			Page of PT (@PFN:101)		
PFN	valid?	PFN	valid	prot	PFN	valid	prot
100	1	10	1	r-x	—	0	—
—	0	23	1	r-x	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	80	1	rw-	—	0	—
—	0	59	1	rw-	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—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



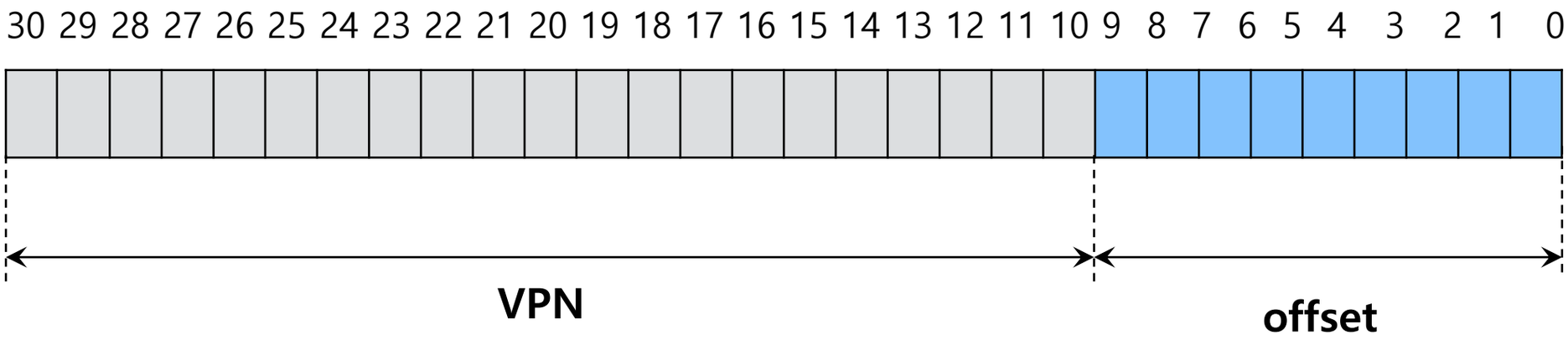
为什么多级页表节约空间

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

More than Two Level

- In some cases, a deeper tree is possible.

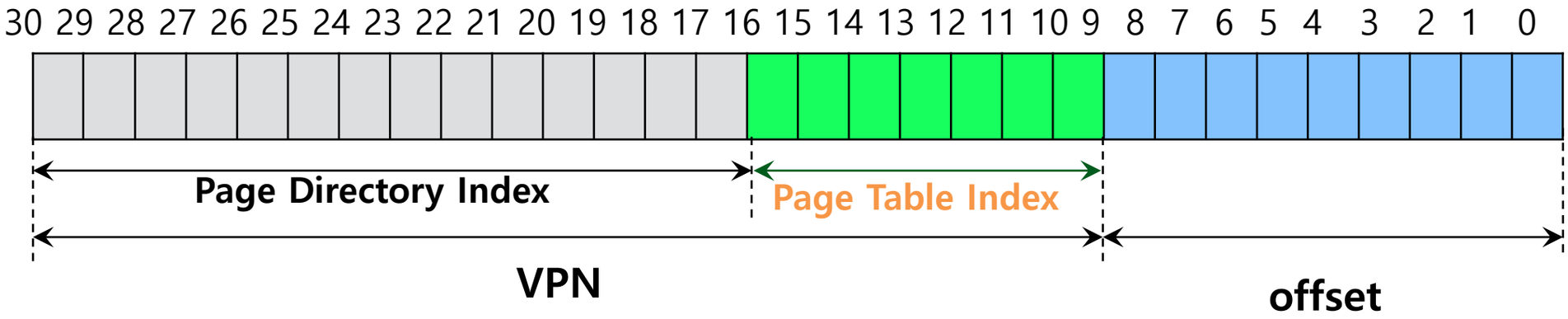


Flag	Detail
Virtual address	30 bit
Page size	512 byte
VPN	21 bit
Offset	9 bit

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?

More than Two Level : Page Table Index

- In some cases, a deeper tree is possible.



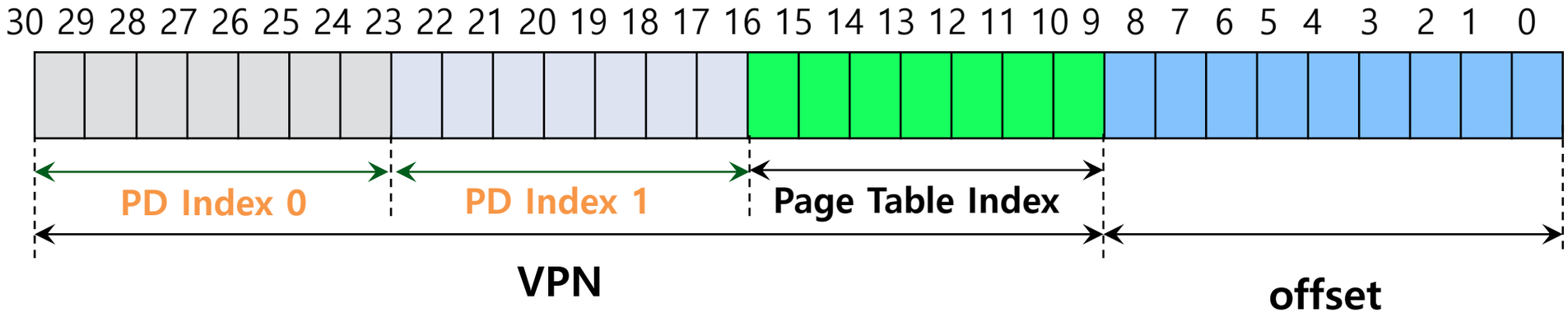
Flag	Detail
Virtual address	30 bit
Page size	512 byte
VPN	21 bit
Offset	9 bit
Page entry per page	128 PTEs

$\log_2 128 = 7$

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。

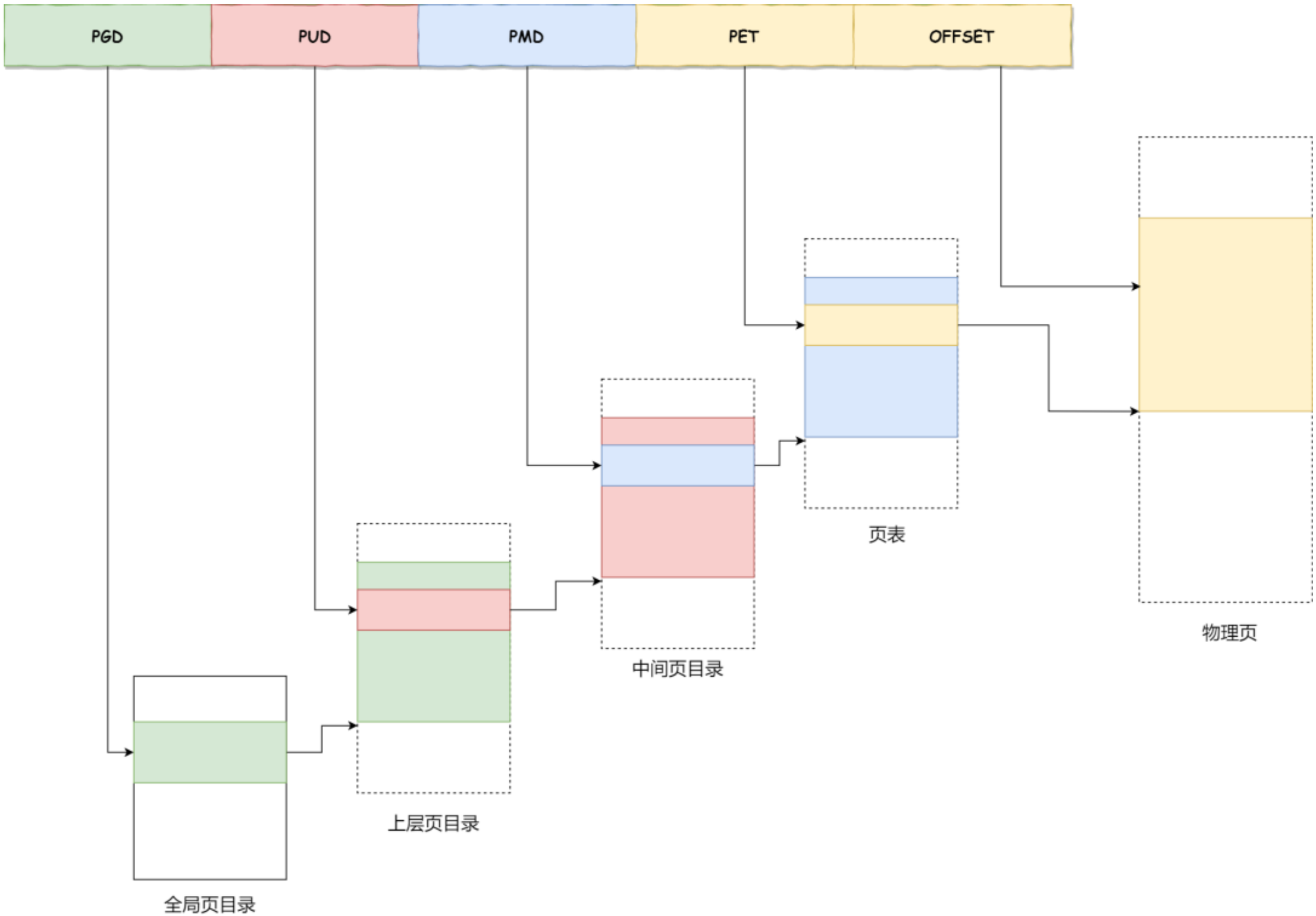
More than Two Level : Page Directory

- 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。

Multi-level Page Table Control Flow

```
01:   VPN = (VirtualAddress & VPN_MASK) >> SHIFT
02:   (Success,TlbEntry) = TLB_Lookup(VPN)
03:   if(Success == True)      //TLB Hit
04:     if(CanAccess(TlbEntry.ProtectBits) == True)
05:         Offset = VirtualAddress & OFFSET_MASK
06:         PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
07:         Register = AccessMemory(PhysAddr)
08:     else RaiseException(PROTECTION_FAULT);
09:   else // perform the full multi-level lookup
```

- (1 lines) extract the virtual page number(VPN)
- (2 lines) check if the TLB holds the transalation for this VPN
- (5-8 lines) extract the page frame number from the relevant TLB entry, and form the desired physical address and access memory

Multi-level Page Table Control Flow

```
11:     else
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 // PDE is Valid: now fetch PTE from PT
```

- (11 lines) extract the Page Directory Index(PDIndex)
- (13 lines) get Page Directory Entry(PDE)
- (15-17 lines) Check PDE valid flag. If valid flag is true, fetch Page Table entry from Page Table

The Translation Process: Remember the TLB

```
18:    PTIndex = (VPN & PT_MASK) >> PT_SHIFT
19:    PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
20:    PTE = AccessMemory(PTEAddr)
21:    if(PTE.Valid == False)
22:          RaiseException(SEGMENTATION_FAULT)
23:    else if(CanAccess(PTE.ProtectBits) == False)
24:          RaiseException(PROTECTION_FAULT);
25:    else
26:          TLB_Insert(VPN, PTE.PFN , PTE.ProtectBits)
27:          RetryInstruction()
```

Inverted Page Tables

- An even more extreme space savings in the world of page tables is found with **inverted page tables**.
- Here, instead of having many page tables (one per process of the system), we keep **a single page table** that has an entry for each **physical page** of the system.
- The entry tells us which process is using this page, and which virtual page of that process maps to this physical page.

- Finding the correct entry is now a matter of searching through this data structure.

- A linear scan would be expensive, and thus a **hash table** is often built over the base structure to speed lookups.

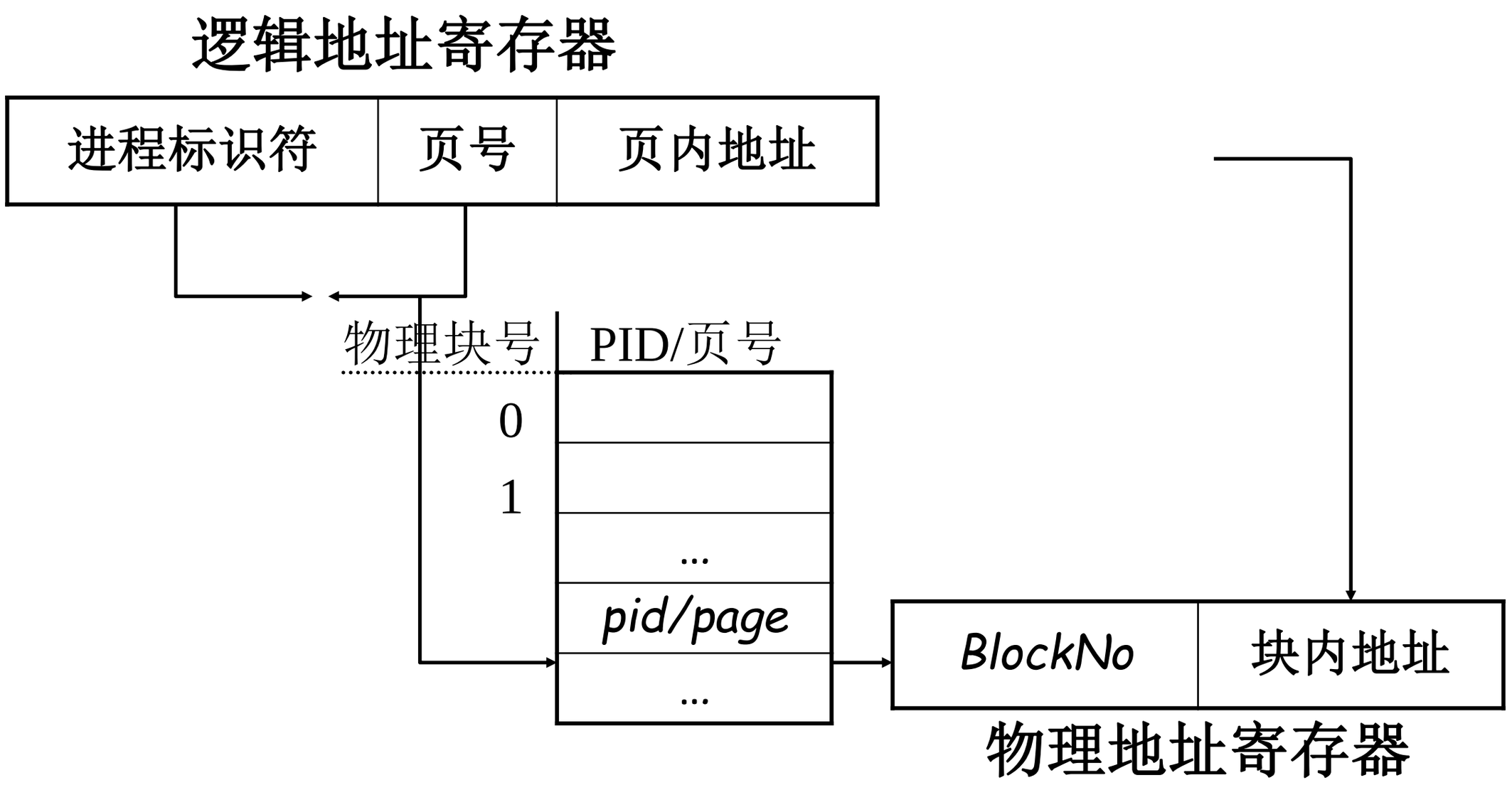
- The **PowerPC** is one example of such an architecture

- Inverted page tables illustrate what we’ ve said from the beginning: **page tables are just data structures**.

- You can do lots of **crazy things** with data structures, making them smaller or bigger, making them slower or faster.

- Multi-level and inverted page tables are just two examples of the many things one could do.

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
...
pid/page
...
BlockNo
块内地址
物理地址寄存器

反置页表

Accessing Hashed Inverted Page Table

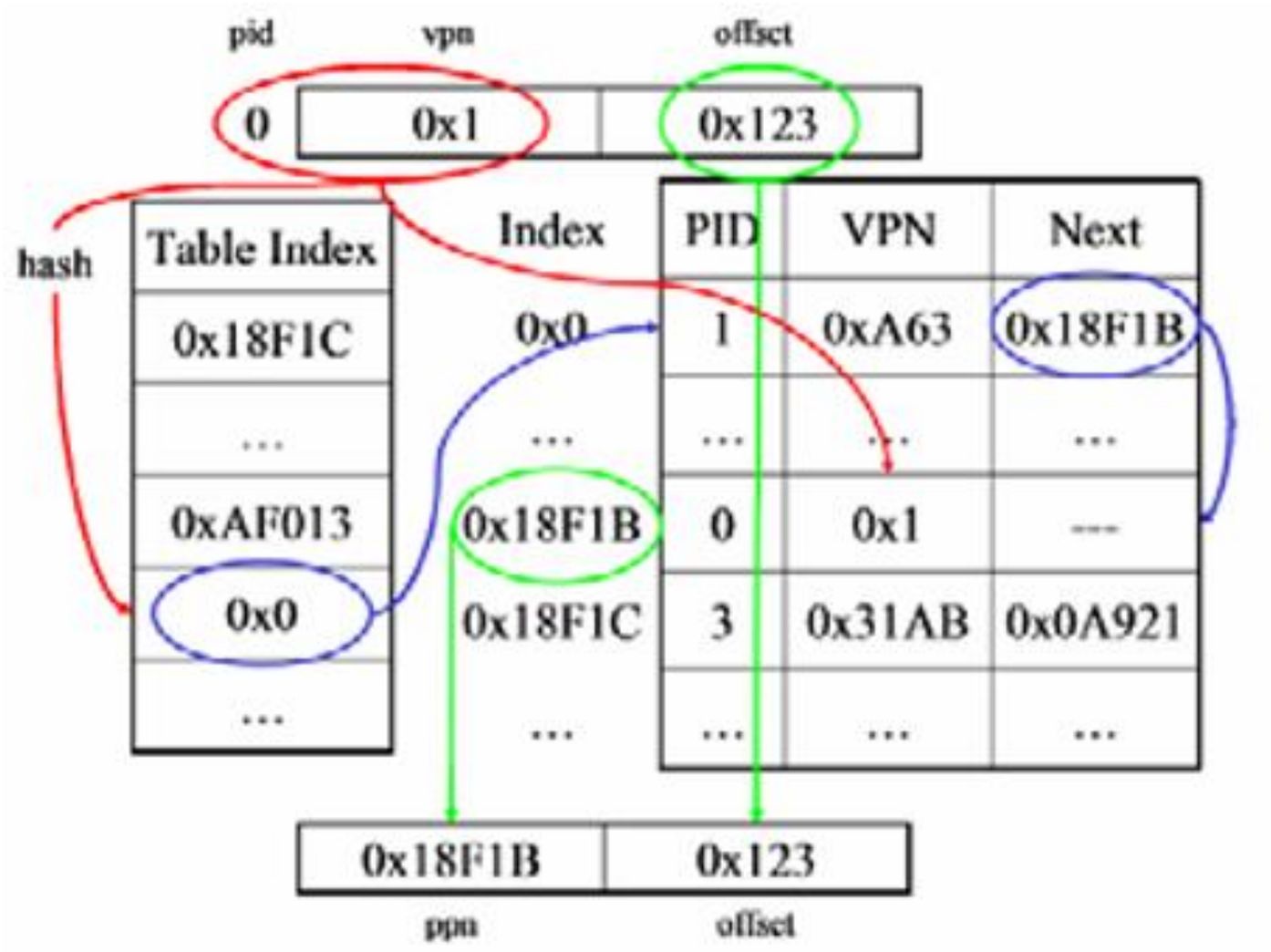


Image from http://www.cs.berkeley.edu

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

Summary

- In a memory-constrained system (like many older systems), small structures make sense;
- In a system with a reasonable amount of memory and with workloads that actively use a large number of pages, a bigger table that speeds up TLB misses might be the right choice.

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<=120ns，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%×（10+100）ns+2%×（10+100+100）ns=112ns

（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%×（10+100）ns+2%×（10+100+100+100）ns=114ns

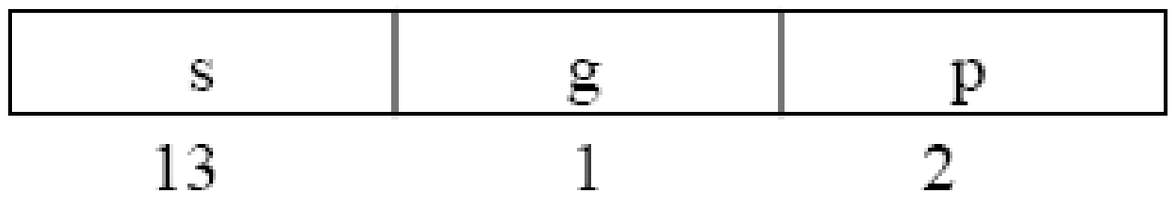
（4）假设TLB的命中率为p，应该满足以下式子：p×（10+100）ns+（1-p）×（10+100+100+100）ns<=120ns

可以解得：p>=95%，所以如果要满足访问时间<=120ns，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。

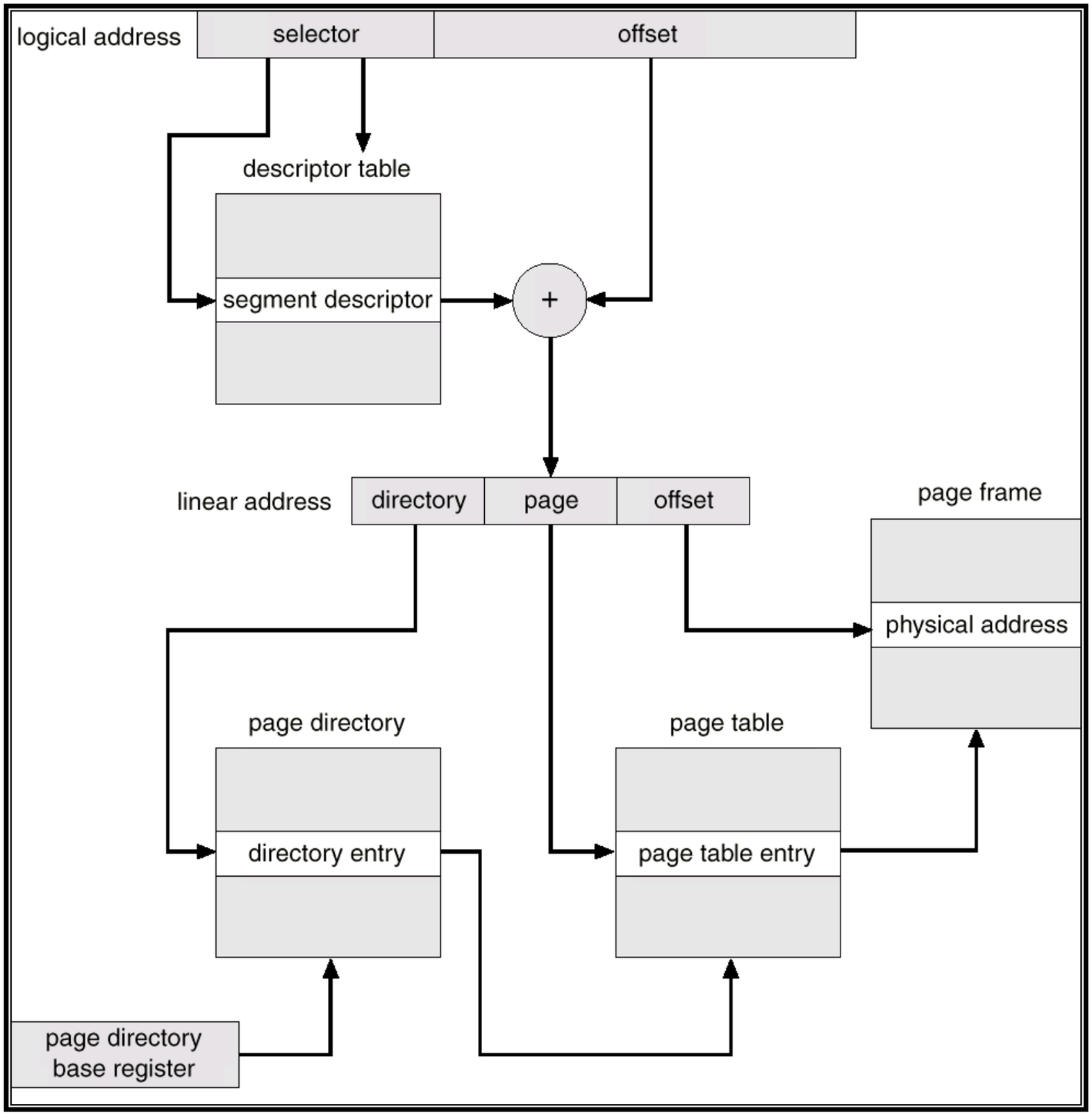
Segmentation with Paging – Intel 386

- 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。

Intel 30386 Address Translation



End