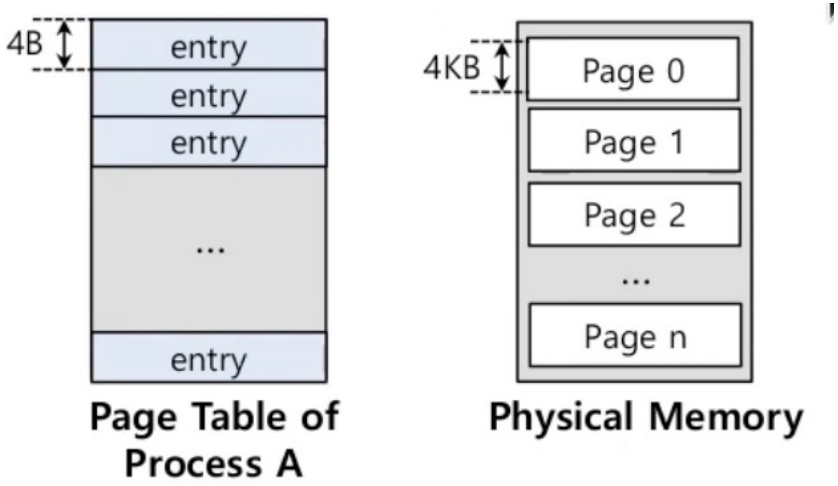
分页：较小的表

邰颖
南京大学
智能科学与技术学院

分页：线性页表

- 我们通常为系统中的每个进程设置一个页表。
  - 假设32位地址空间，页面大小为4KB，每个页表条目占用4字节。

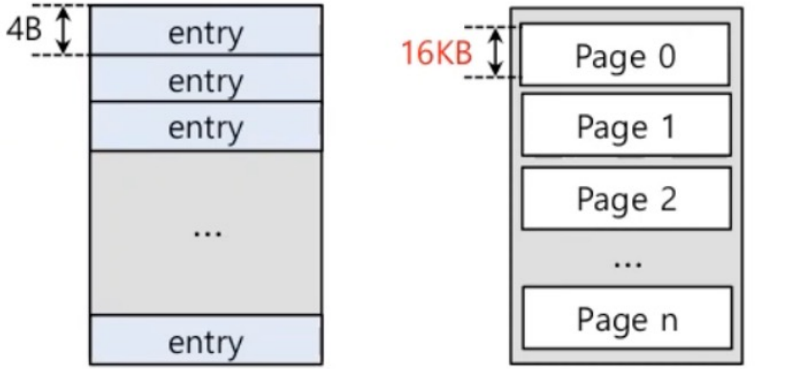


页表过大，消耗了过多的内存。

$$\text{Page table size} = \frac{2^{32}}{2^{12}} * 4Byte = 4MByte$$

简单的解决方案：使用更大页面减少页表大小

- 页表太大，导致消耗过多内存
  - 假设32位地址空间，页面大小为16KB，每个页表条目占4字节。

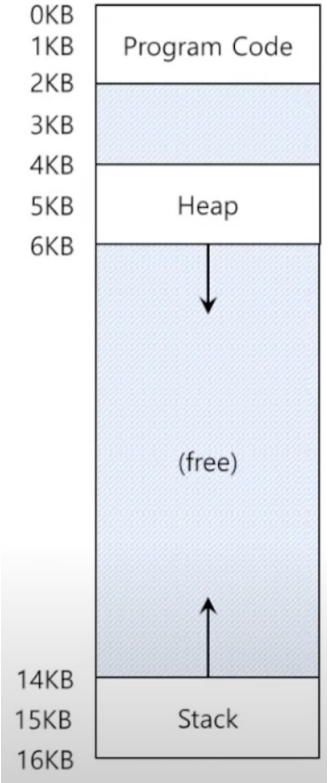


大页面导致内部碎片化。

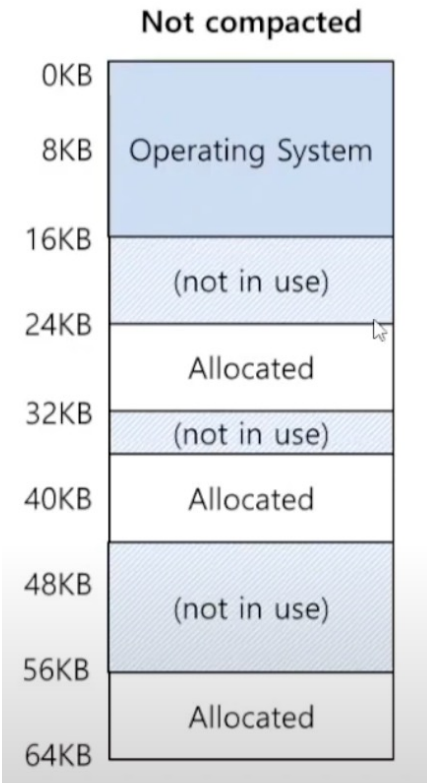
$\frac{2^{32}}{2^{16}} * 4 = 1MB$ per page table

回顾：内部碎片 & 外部碎片

内部碎片

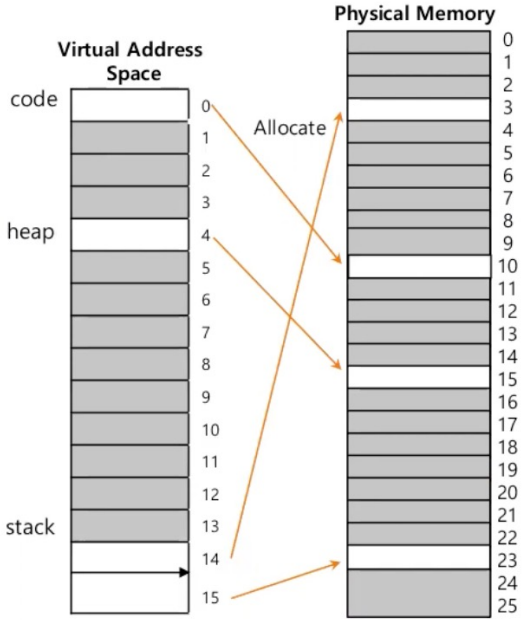


外部碎片



问题分析

- 一个进程地址空间的单一页表项



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

虚拟地址空间：

•假设使用16KB的地址空间，每个页面为1KB。

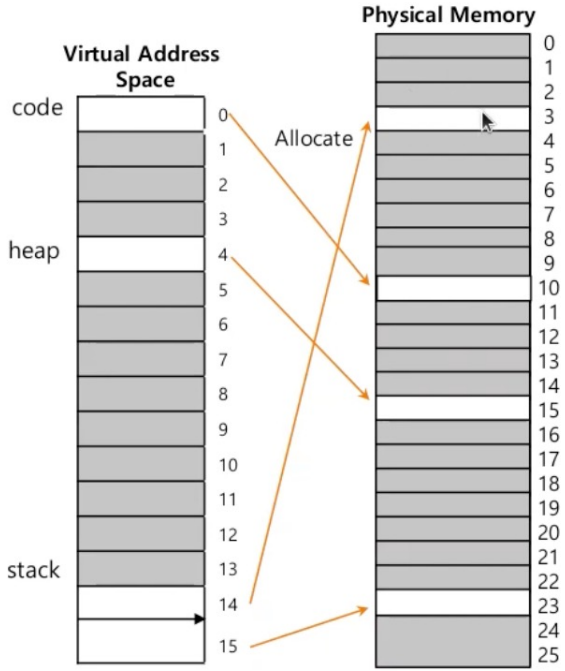
•分别有代码段、堆段和栈段。

物理内存：

•映射虚拟地址到物理内存中的页帧。

问题分析（续）

- 大部分的页表是未使用的，充满了无效条目。



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

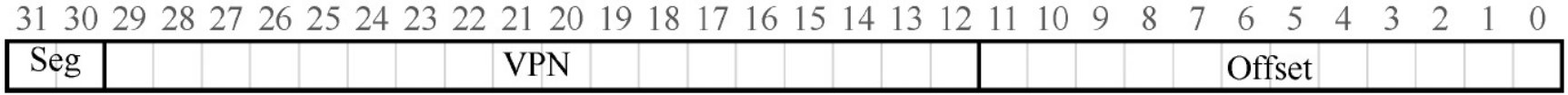
A Page Table For 16KB Address Space

混合方法：分页和分段

- 为了减少页表的内存开销
  - 给每个段一个页表，而不是整个虚拟地址空间一个页表
  - 使用基址不指向段本身，而是存储该段页表的物理地址。
  - **界限寄存器**用于指示页表的结束位置。

混合方法的简单示例

- 每个进程关联有三个页表
  - 当进程运行时，**每个段的基址寄存器**包含**该段线性页表的物理地址**。

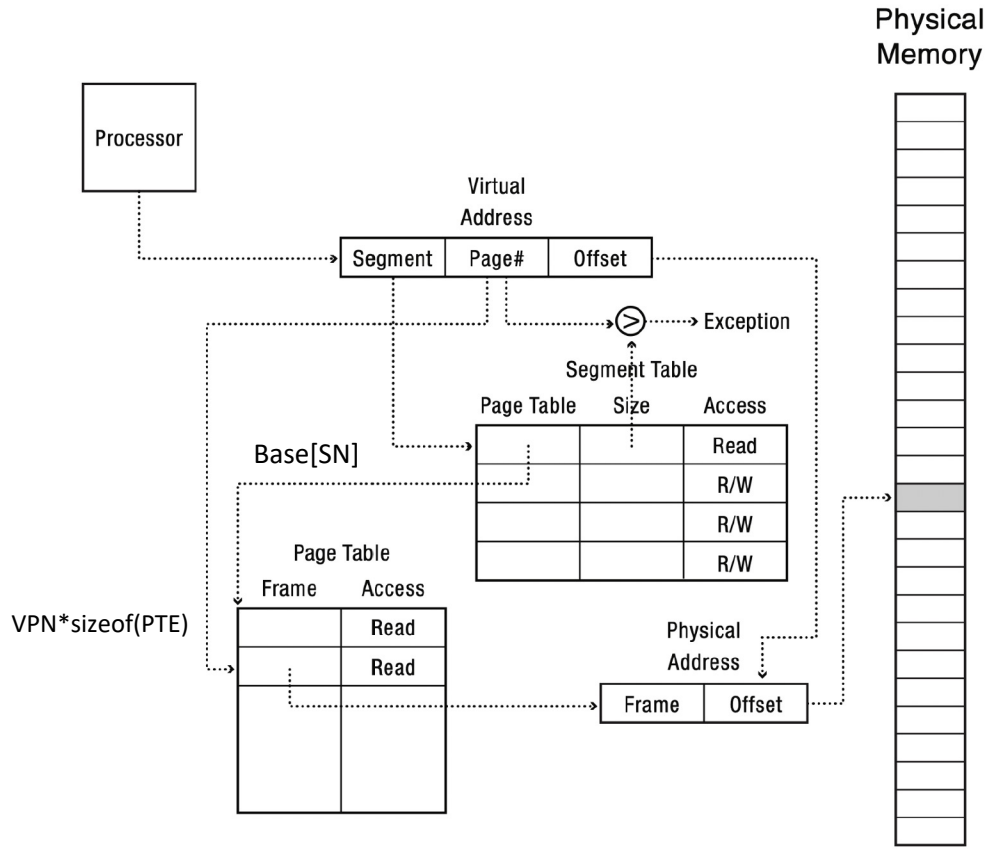


SN

32-bit Virtual address space with 4KB pages

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

混合方法：某一个段的地址转换流程



- **段表查找**：通过段表，首先获取虚拟地址中的段信息，确认是否超出界限，若超出则抛出异常。若有效，获取该段页表的起始物理地址

- **页表查找**：根据虚拟地址中的虚拟页号，根据该段页表映射到物理页帧号

- **物理地址**：最终通过该段页表获得的物理页帧号和虚拟地址中获取的偏移量组成物理地址，访问物理内存

分段、分页、混合方法的区别

- **分段：维护单个段表**
- 分成多个逻辑段（代码、堆、栈）
- 段表记录每个段的基址、界限

- **分页：维护单个页表**
- 将地址空间分割成固定大小的页
- 页表记录虚拟页号到物理页帧号的映射

- **分页+分段：维护单个段表和多个页表**
- 每个逻辑段分配一个页表
- 段表项中的基址记录该段页表的物理地址，界限寄存器指示该段页表的结束位置

混合方法下的TLB未命中（硬件管理的TLB）

- 硬件使用**段号位（SN）来确定使用哪个基址（base）和界限（bounds）寄存器**对
- 然后硬件获取其中的**物理地址**，并将其与虚拟页号（VPN）**组合，以计算**页表项（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))
```

混合方法：为什么能减少页表内存开销？

- 一些假设：**32位**虚拟地址空间：4GB (2^32，32位)；页面大小：4KB (2^12)
  - 进程使用内存：512MB（代码段：256MB，数据段：128MB，堆栈段：128MB）

场景1：纯分页模式

页表项总数计算：

•每个页对应1个页表项

•**总页数** = 虚拟地址空间 / 页面大小 = 4GB / 4KB = **1,048,576项** (即1M项)

•即使进程只使用512MB，页表仍需覆盖整个32位虚拟地址空间（所有可能的虚拟地址都需要页表项映射，无论物理内存是否实际分配）

混合方法：为什么能减少页表内存开销？（续）

场景2：分段+分页混合模式

- 将32位虚拟地址空间划分为3个独立段：代码段：0-256MB，数据段：256MB-384MB，堆栈段：384MB-512MB
- **每个段独立维护页表**，互不重叠

分步计算：

1.代码段页表项：

1. 段大小：256MB
2. 所需页数 = 256MB / 4KB = (256×1024KB) / 4KB = **65,536项** (即64K项)

2.数据段页表项：

1. 段大小：128MB
2. 所需页数 = 128MB / 4KB = (128×1024KB) / 4KB = **32,768项** (即32K项)

3.堆栈段页表项：

1. 段大小：128MB
2. 所需页数 = 128MB / 4KB = **32,768项** (即32K项)

总页表项 = 64K + 32K + 32K = **131,072项** (即128K项)

混合方法：为什么能减少页表内存开销？（续）

- **核心思想**：分段将虚拟地址空间“剪裁”成多个子空间（段），每个段有独立页表，**避免为未使用的虚拟地址预留页表项**。

- 对比节省量
  - •纯分页：1,048,576项
  - •分段分页：128,000项
  - •节省比例 = (1M - 128K)/1M = **87.5%**

```
[纯分页页表]
0x00000000 ~ 0xFFFFFFFF（全覆盖4GB）
├── 代码段 256MB（实际使用）
├── 数据段 128MB（实际使用）
├── 堆栈段 128MB（实际使用）
└── 剩余3.5GB空白区域（仍需保留页表项）

[分段分页页表]
代码段页表：0x00000000 ~ 0x0FFFFFFF（仅覆盖256MB）
数据段页表：0x10000000 ~ 0x17FFFFFF（仅覆盖128MB）
堆栈段页表：0x18000000 ~ 0x1FFFFFFF（仅覆盖128MB）
空白区域无需维护页表
```

混合方法的问题（Problems of Hybrid Approach）

- 如果我们拥有一个**大但使用稀疏的堆**（heap），仍然可能导致大量的**页表浪费**（page table waste）。
  - 在某些系统中，为了管理一个大的堆段，可能需要分配一个**大页表**来存储**整个堆段的页映射**。
  - 即使**只有少量页面真正被使用**，也可能需要维护一个**完整的页表结构**，导致页表本身占用大量内存。

- **外部碎片（External fragmentation）可能再次出现，因为页表的大小可以是任意的（arbitrary）。**

多级页表（Multi-level Page Tables）

- 将线性页表转变为类似树状结构。
  - 首先，将页表划分为按页大小单位的块。
  - 其次，如果页表中整页的页表项无效，则完全不分配该页表的内存。
  - 为了追踪页表中页的有效性，使用一个新的结构：称为**页目录（page directory）**。

多级页表：页目录（一个两级页表例子）

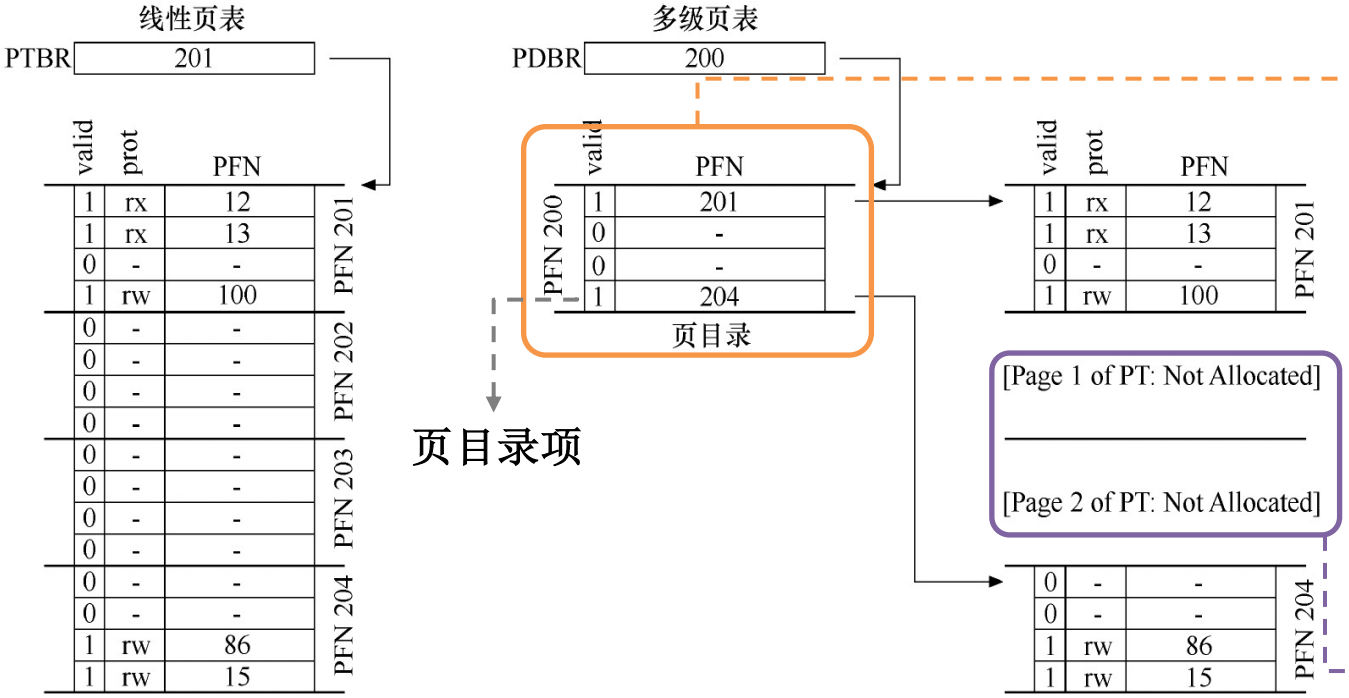


图 20.2　线性（左）和多级（右）页表

- 左边：线性页表
  - **PTBR**：页表基址寄存器
- 右边：多级页表
  - **页目录**：指向下一级页表的物理页帧号
- 如果页表条目无效，则不分配该页表页。

用页目录来记录页表的哪些页被分配

多级页表：页目录项（Page directory entries, PDE）

•页目录每个条目对应一个页表页。

- 页目录由多个**页目录项（PDE）**组成。

•**PDE**（页目录项）包含：有效位（valid bit）和物理页帧号（PFN）

•如果PDE 项是有效的，意味着该项指向的页表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。

多级页表：优点与缺点

优点：

1.只根据实际使用的地址空间按比例分配页表空间。

2.当操作系统需要分配或扩展页表时，可以直接使用下一个空闲页。

缺点：

1.多级页表是一个典型的时间-空间权衡（time-space trade-off）的例子。

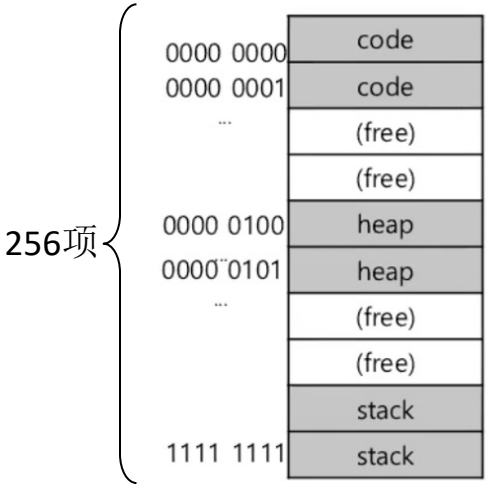
2.实现复杂性较高。

多级页表：间接寻址级别

- 多级结构可以通过页目录的使用调整**间接寻址的级别**。
  - **间接寻址**允许我们将页表页放置在物理内存中的任何位置。

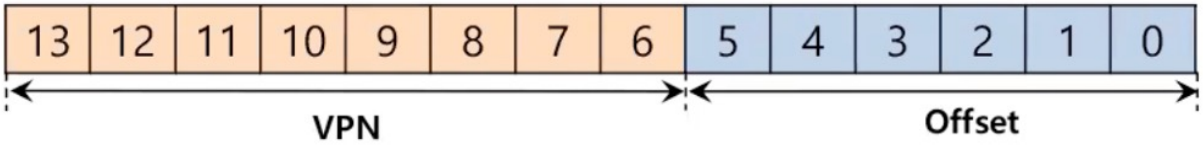
详细的多级页表示例

- 为了更好地理解多级页表背后的概念，我们来看一个示例



参数	详情
地址空间	16 KB
页大小	64 字节
虚拟地址	14 位
虚拟页号 (VPN)	8 位
偏移量 (Offset)	6 位
页表条目	2^8（256）个条目

16 KB 地址空间，64 字节的页



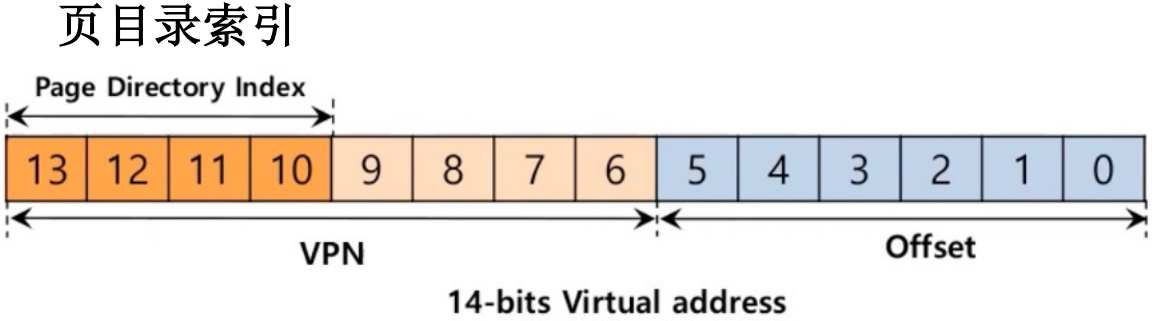
详细的多级页表示例：页目录索引（PDIndex）

•页目录中每个页表页需要一个条目。

- 页目录有16个条目

$$PDEAddr = PageDirBase + (PDIndex \times sizeof(PDE))$$

•如果页目录条目是**无效的**，则会引发异常（访问无效）。



详细的多级页表示例：页表索引（PTIndex）

•如果页目录项（PDE）有效，我们还需要继续进行下一步操作：
  - 从页目录项所指向的页表页中获取对应的页表项（PTE）

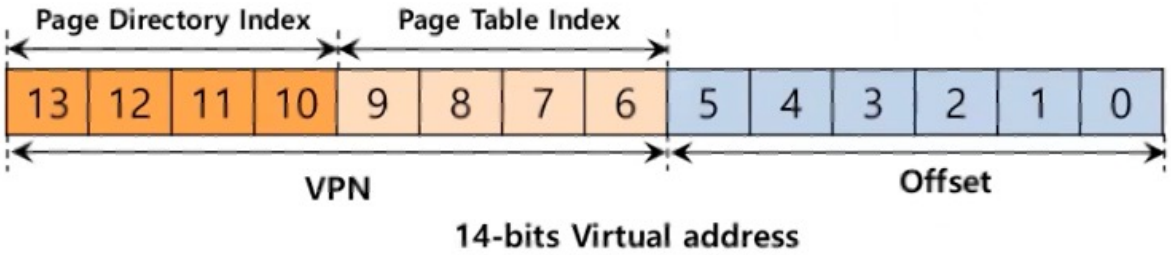
•这个**页表索引**（Page Table Index, PTIndex）可以用来访问页表中的对应条目。

•因此，页表项地址的计算公式为：

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

下一级页表的物理页帧号

页表索引



详细的多级页表示例：内存使用

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	—	55	1	rw-
101	1	—	0	—	45	1	rw-

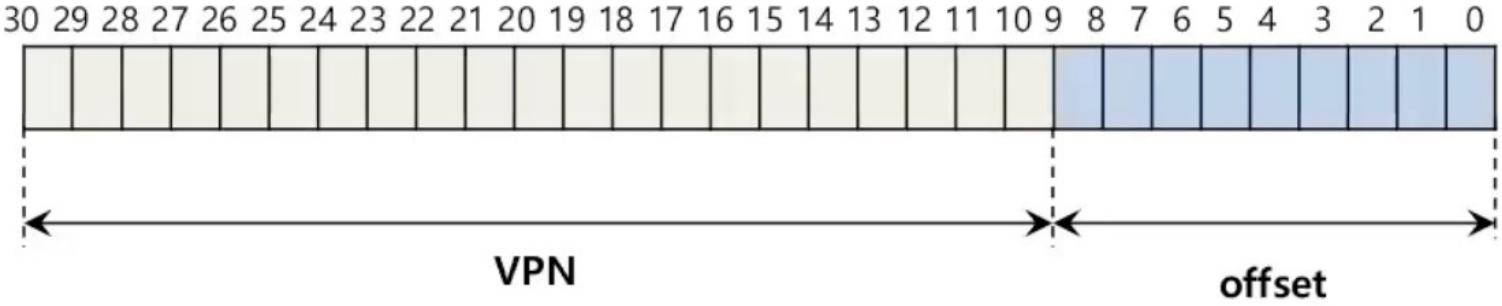
- 页目录中记录了两个有效区域（100 & 101）
- 页表的一个有效页在PFN 100 中。该页包含地址空间的前 16 个VPN 的映射。
- 页表的另一个有效页在PFN 101 中。该页包含地址空间的最后16 个VPN 的映射。

详细的多级页表示例：转换虚拟地址 0x3F80

- 二进制表示（Binary）= 11 1111 1000 0000
- 虚拟页号（VPN）= 11111110
- 页内偏移（Offset）= 000000
- 页目录索引（Page Directory Index）= 1111，查得有效的页框号 PFN = 101
- 页表索引（在页表页 PFN:101 中）= 1110，查得有效的页框号 PFN = 55 = 0x37
- 将 PFN 左移 6 位后加上偏移，得到物理地址：
  - 00 1101 1100 0000 = 0x0DC0

超过两级的页表结构

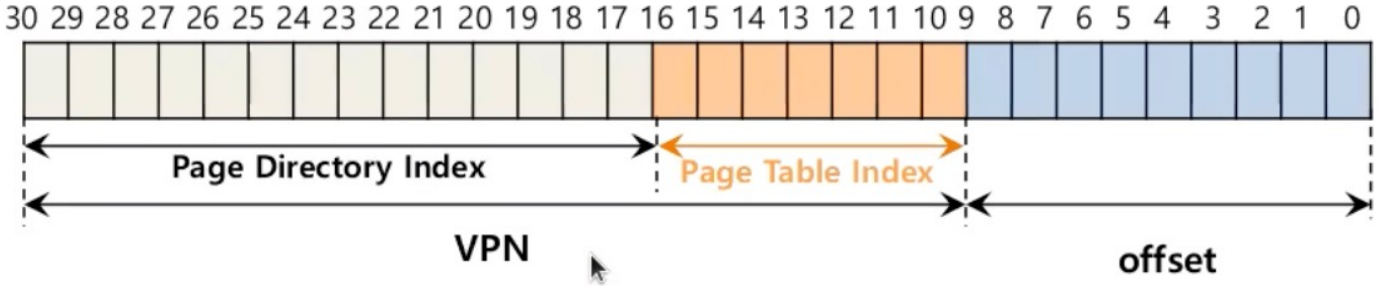
•在某些情况下，可能需要更深层次的页表树结构（即三级或更多级页表）。



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

超过两级的页表结构：页表索引

•在某些情况下，可能需要更深层次的页表树结构（即三级或更多级页表）。



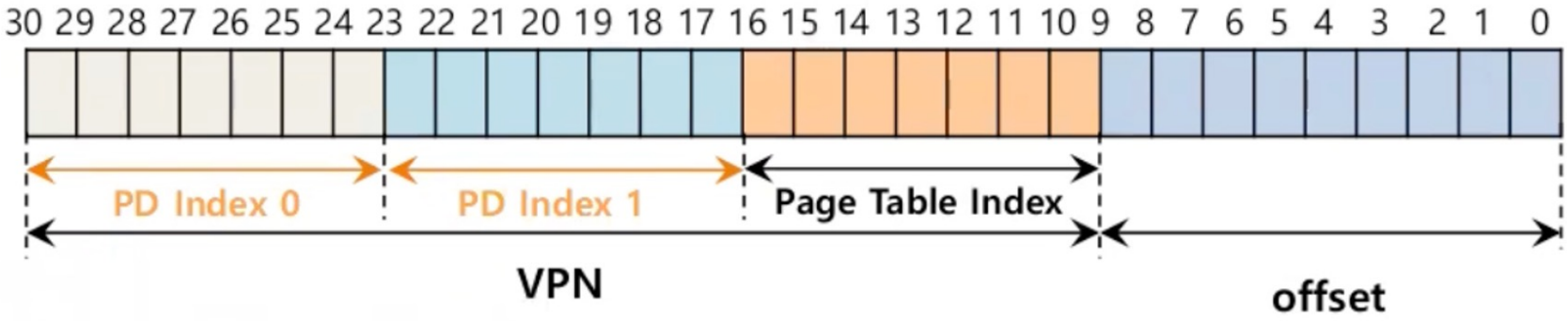
Flag	Detail
Virtual address	30 bit
Page size	512 byte
VPN	21 bit
Offset	9 bit
Page entry per page	128 PTEs

$\log_2 128 = 7$

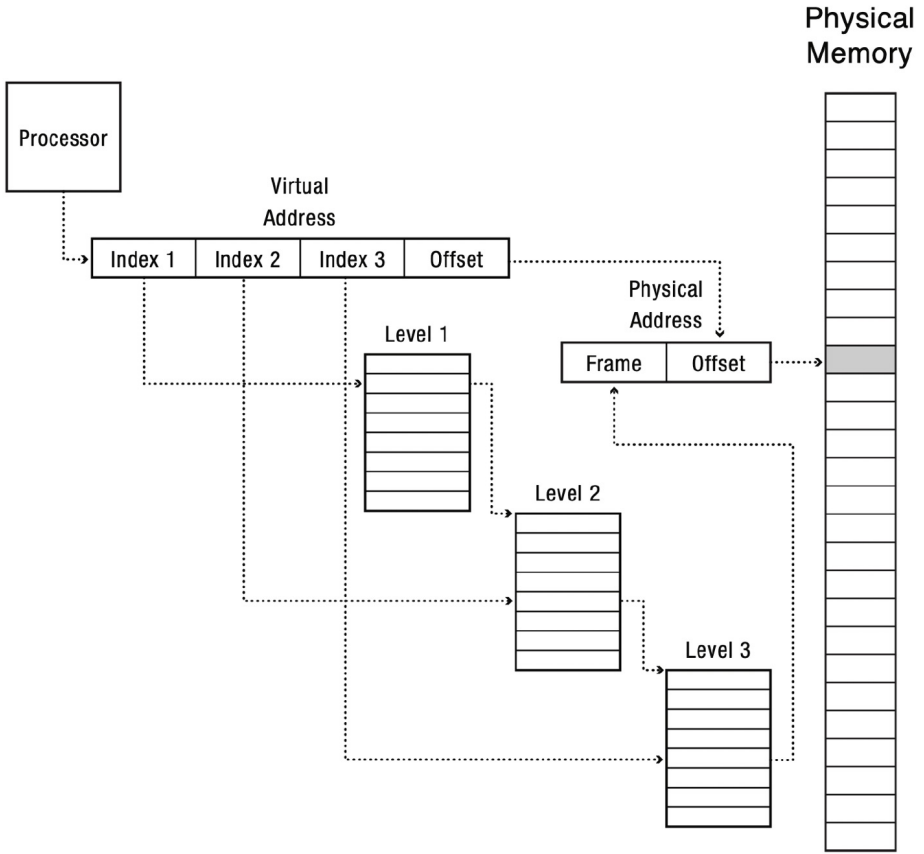
超过两级的页表结构：页目录

•如果我们的页目录包含 2^{14} 个条目，它将不再只占用一页，而是会跨越 128 页。

•为了解决这个问题，我们引入**更深一层的页表树结构**，通过将页目录本身进一步拆分为多个页目录页（即分层的页目录）。



多级页表流程示意图



第一级页目录基址存在PDBR

Index1: 指向第一级页目录索引
Level1: 指向第二级页目录的起始地址

Index2: 指向第二级页目录索引
Level2: 指向页表起始地址

Index3: 指向页表索引
Level3: 指向物理页帧号PFN

多级页表控制流：TLB命中

```
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行**） 提取虚拟页号（VPN）
- （**2行**） 检查TLB是否保存了该VPN的映射
- （**5-8行**） 从相关的TLB项中提取物理页帧号（PFN），并通过合成虚拟地址计算出物理地址，最终访问内存

多级页表控制流：TLB未命中，执行多级查找

```
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
```

第12行：提取页目录索引（Page Directory Index，PDIndex）
第13行：计算页目录项地址（PDEAddr）
第14行：从内存中访问页目录项（PDE）
第15-17行：检查页目录项是否有效。如果无效，抛出段错误；否则继续从页表中获取页表项（PTE）

地址转换过程：页目录项有效，插入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()
```

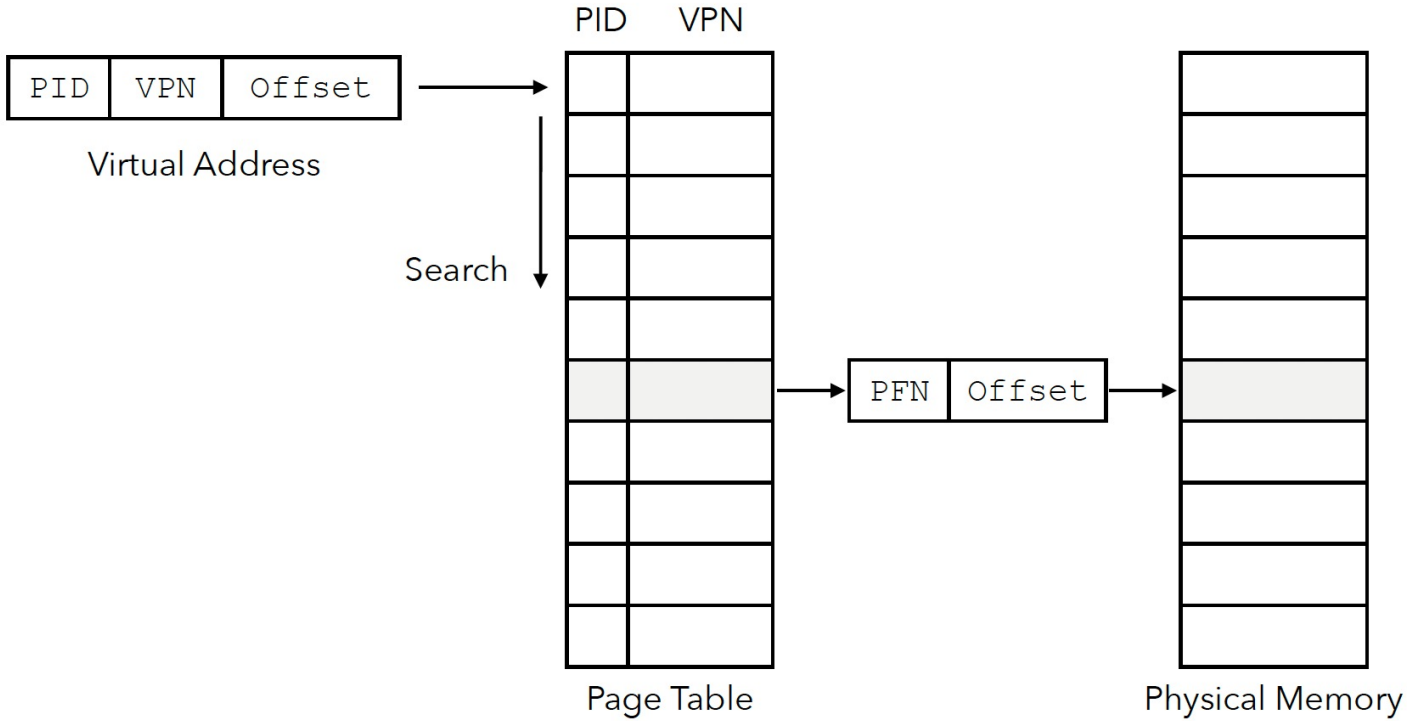
倒排页表（Inverted Page Tables）

- 使用一个**单一页表**，该页表中的每一项对应系统中的**一个物理页帧**；
- 每个页表项告诉我们：**哪个进程正在使用该物理页**，以及**该进程中的哪个虚拟页映射到了这个物理页**；
- PowerPC[1] 架构使用倒排页表。

[1] https://baike.baidu.com/item/PowerPC/7381773

倒排页表 (Inverted Page Tables)

- 与其每个进程维护一个页表 (占很多空间)，还不如让整个系统中就只有一个页表 (每一个 PTE 对应一个物理页)



小结

- 我们介绍了如何构建较小的页表
  - 分页+分段混合方法
  - 多级页表（两级，以及超过两级）
  - 倒排页表