

CMSC 125: Operating Systems

- ❑ Instructor: **Joseph Anthony C. Hermocilla**
- ❑ Email: jchermocilla@up.edu.ph
- ❑ Web: <https://jachermocilla.org>



Resources

Book: <https://pages.cs.wisc.edu/~remzi/OSTEP/>

Slides Template:

<https://pages.cs.wisc.edu/~remzi/OSTEP/Educators-Slides/Youjip/>



Acknowledgement

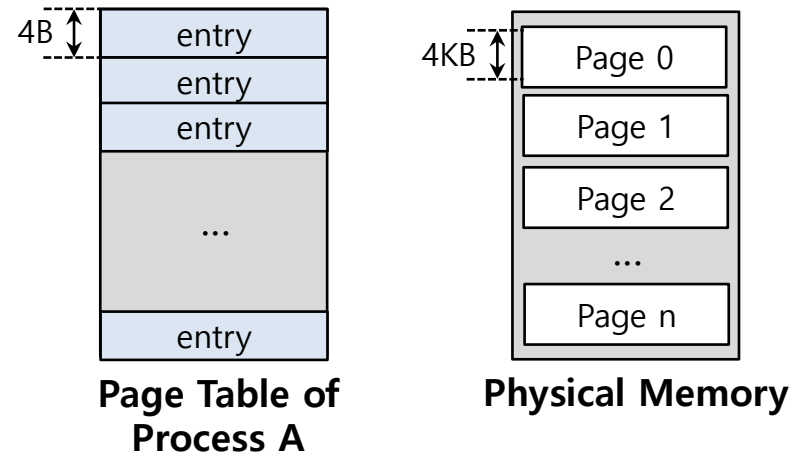
- ▣ This lecture slide set was initially developed for Operating System course in Computer Science Dept. at Hanyang University. This lecture slide set is for OSTEP book written by Remzi and Andrea at University of Wisconsin.

20. Paging: Smaller Tables

Operating System: Three Easy Pieces

Paging: Linear Page Tables

- We usually have one page table for every process in the system
 - ◆ Assume that 32-bit address space with 4KB pages and 4-byte page-table entry

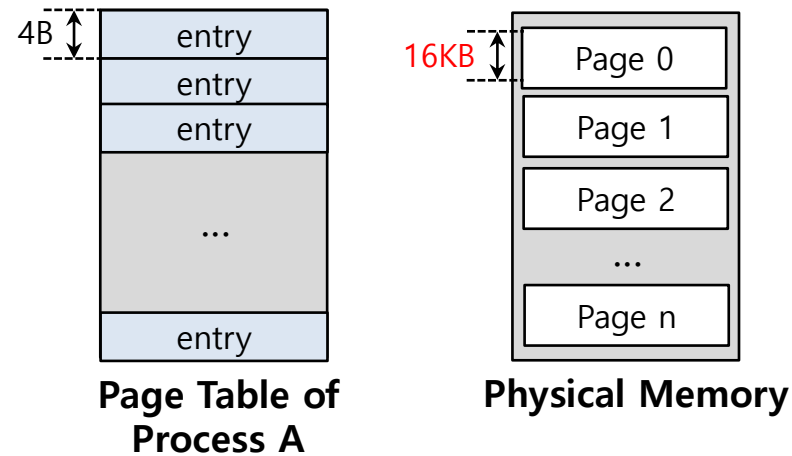


$$\text{Page table size} = \frac{2^{32}}{2^{12}} * 4\text{Byte} = 4\text{MByte}$$

Page table are **too big** and thus consume too much memory.

Paging: Smaller Tables by using Bigger Pages

- Page table are too big and thus consume too much memory
 - ◆ Assume that 32-bit address space with **16KB** pages and 4-byte page-table entry

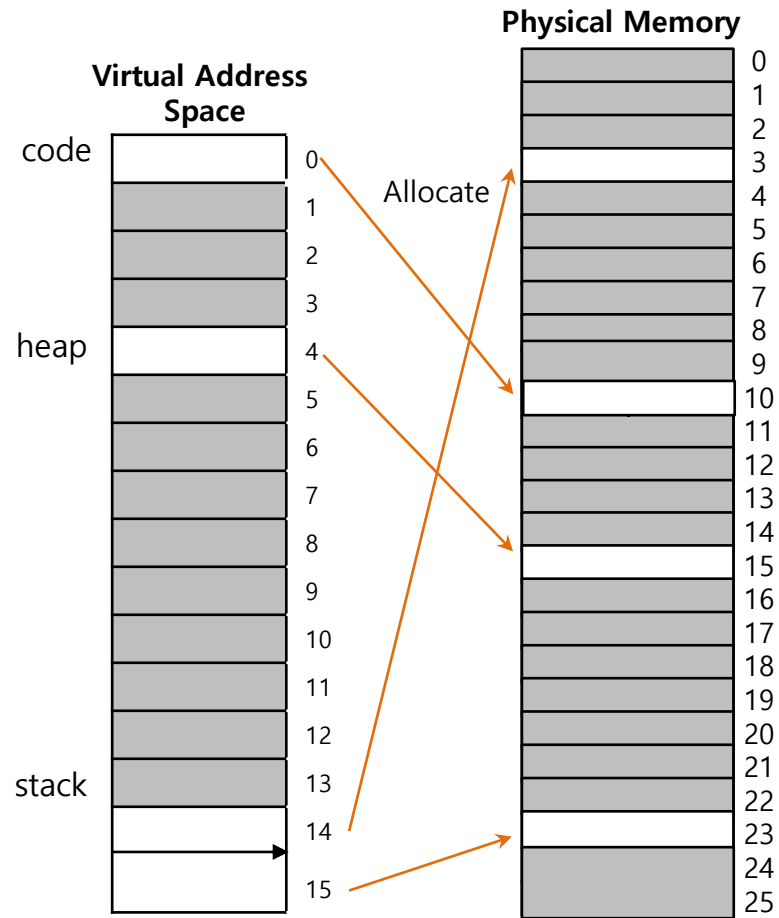


$$\frac{2^{32}}{2^{16}} * 4 = 1MB \text{ per page table}$$

Big pages lead to **internal fragmentation**.

Problem

- Single page table entries for a process' address space



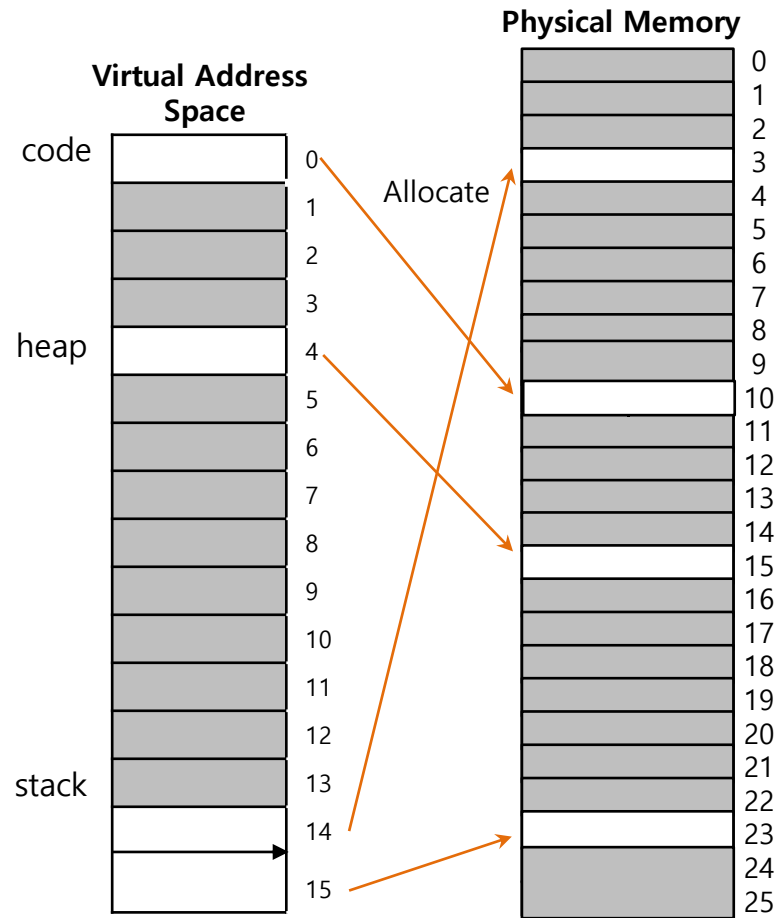
A 16KB Address Space with 1KB Pages

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

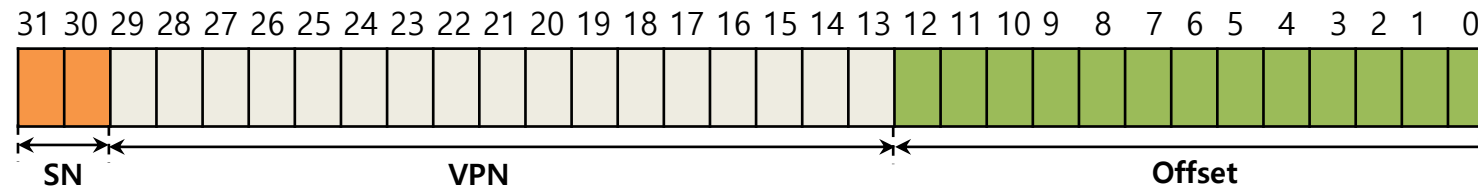
A Page Table For 16KB Address Space

Hybrid Approach: Paging and Segments

- 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

- Each process has **three** page tables associated with it
 - ◆ When process is running, the **base register for each of these segments** contains the **physical address of a linear page table for that segment**



32-bit Virtual address space with 4KB pages

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

TLB miss on Hybrid Approach (Hardware-managed TLB)

- ❑ 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))
```

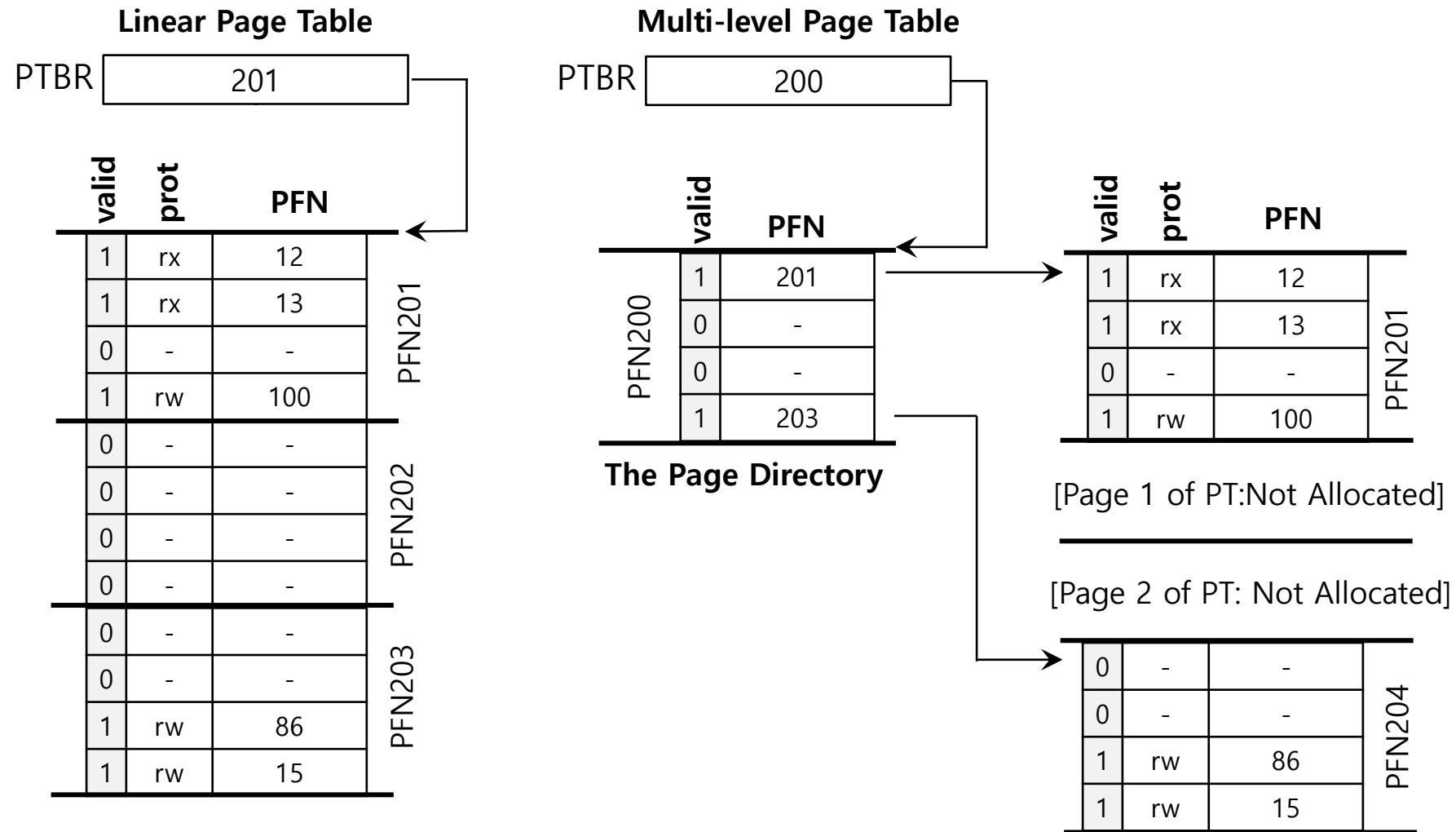
Problems of Hybrid Approach

1. If we have a large but sparsely-used heap, we can still end up with a lot of page table waste
2. External fragmentation can arise again because page tables can be arbitrary in size

Multi-level Page Tables

- Turns the linear page table into something like a tree
 - ◆ Chop up the page table into page-sized units
 - ◆ If an entire page of page-table entries is invalid, don't allocate that page of the page table at all
 - ◆ To track whether a page of the page table is valid, use a new structure, called **page directory**

Multi-level Page Tables: Page directory



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

Multi-level Page Tables: Page directory entries

- The page directory contains one entry per page of the page table
 - ◆ It consists of a number of **page directory entries(PDE)**
- PDE has a valid bit and page frame number(PFN)

Multi-level Page Tables: Advantage & Disadvantage

□ Advantage

1. Only allocates page-table space in proportion to the amount of address space you are using
2. The OS can grab the next free page when it needs to allocate or grow a page table

□ Disadvantage

1. Multi-level page table is a small example of a **time-space trade-off**
2. Complexity

Multi-level Page Tables: Level of indirection

- A multi-level structure can adjust **level of indirection** through use of the page directory
 - ◆ Indirection place page-table pages wherever we would like in physical memory

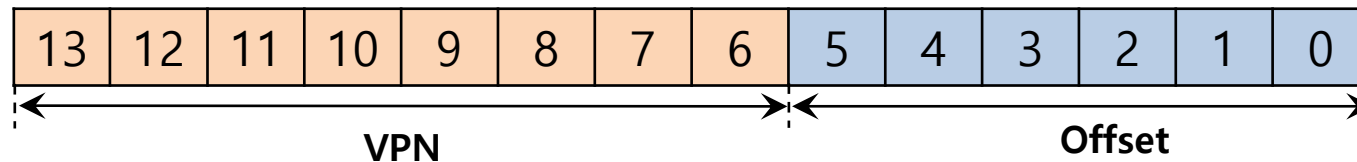
A Detailed Multi-Level Example

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

0000 0000	code
0000 0001	code
...	(free)
	(free)
0000 0100	heap
0000 0101	heap
...	(free)
	(free)
	stack
1111 1111	stack

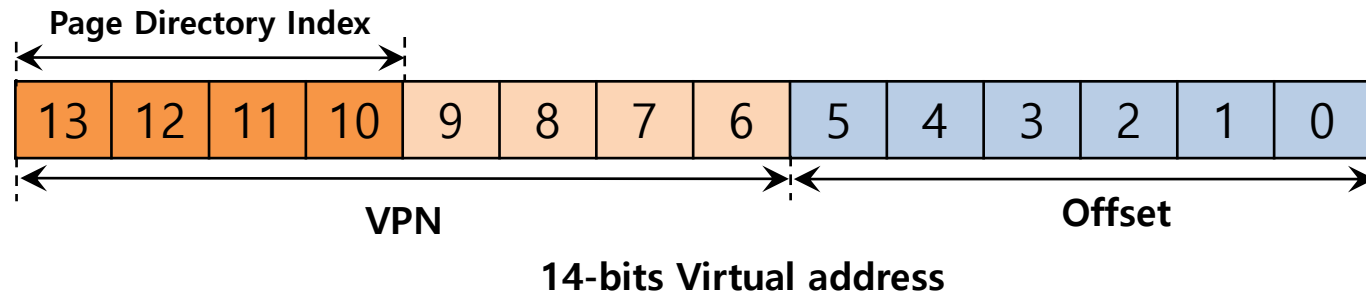
Parameter	Detail
Address space	16 KB
Page size	64 bytes
Virtual address	14 bits
VPN	8 bits
Offset	6 bits
Page table entry	$2^8(256)$

A 16-KB Address Space With 64-byte Pages



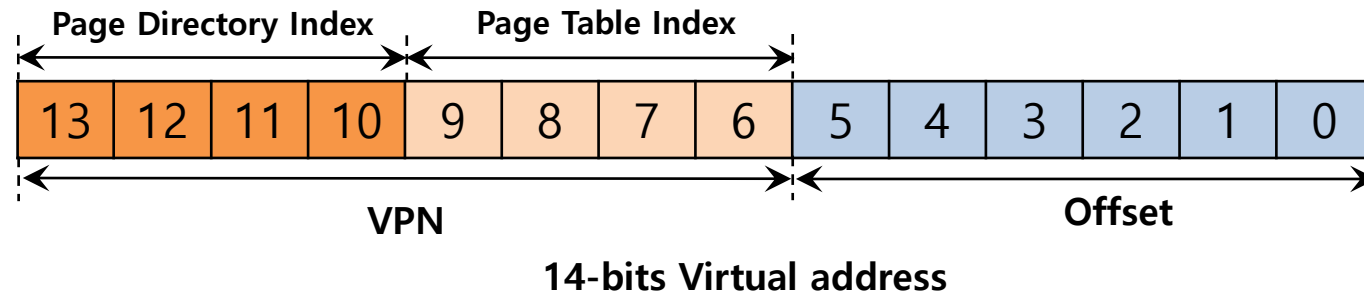
A Detailed Multi-Level Example: Page Directory Idx

- The page directory needs one entry per page of the page table
 - ◆ It has 16 entries
- If the page-directory entry is **invalid** → Raise an exception (The access is invalid)



A Detailed Multi-Level Example: Page Table Idx

- If the PDE is valid, we have more work to do
 - ◆ To fetch the page table entry(PTE) from the page of the page table pointed to by this page-directory entry
- This **page table index(PTIndex)** can then be used to index into the page table itself
- Thus: $\text{PTEAddr} = (\text{PDE.PFN} \ll \text{SHIFT}) + (\text{PTIndex} * \text{sizeof}(\text{PTE}))$



A Detailed Multi-Level Example: Memory Usage

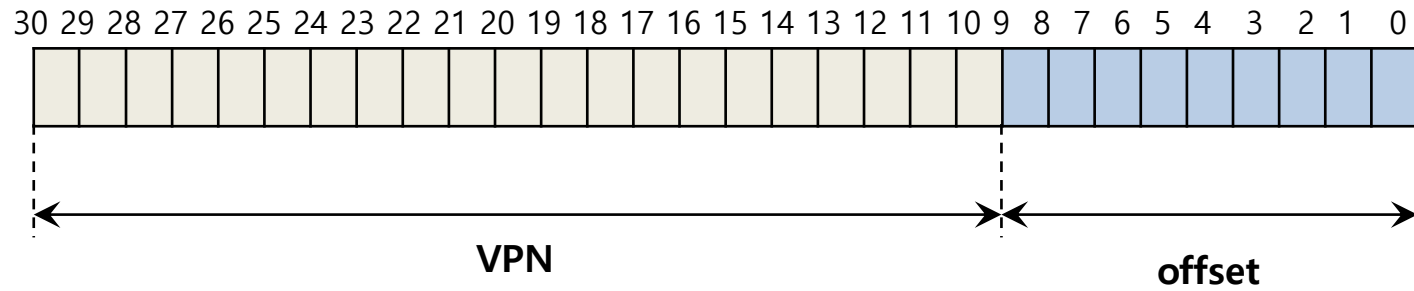
Page Directory		Page of PT (@PFN:100)			Page of PT (@PFN:101)		
PFN	valid?	PFN	valid	prot	PFN	valid	prot
100	1	10	1	r-x	—	0	—
—	0	23	1	r-x	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	80	1	rw-	—	0	—
—	0	59	1	rw-	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—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

A Detailed Multi-Level Example: Translation of virtual address 0x3F80

- ❑ Binary = 11 1111 1000 0000
- ❑ VPN = 11111110
- ❑ Offset = 000000
- ❑ Page Directory Index = 1111, we obtain a valid PFN = 101
- ❑ Index into the page of PT @PFN:101 = 1110, we obtain a valid PFN = 55 = 0x37
- ❑ Add offset after shifting to left 6 bits to obtain the physical address: 00 1101 1100 0000 = 0x0DC0

More than Two Level

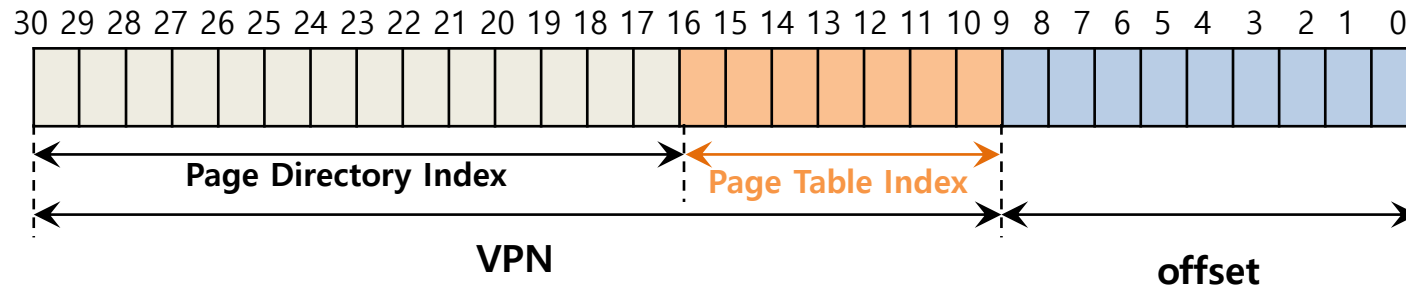
- In some cases, a deeper tree is possible



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

More than Two Level : Page Table Index

- In some cases, a deeper tree is possible

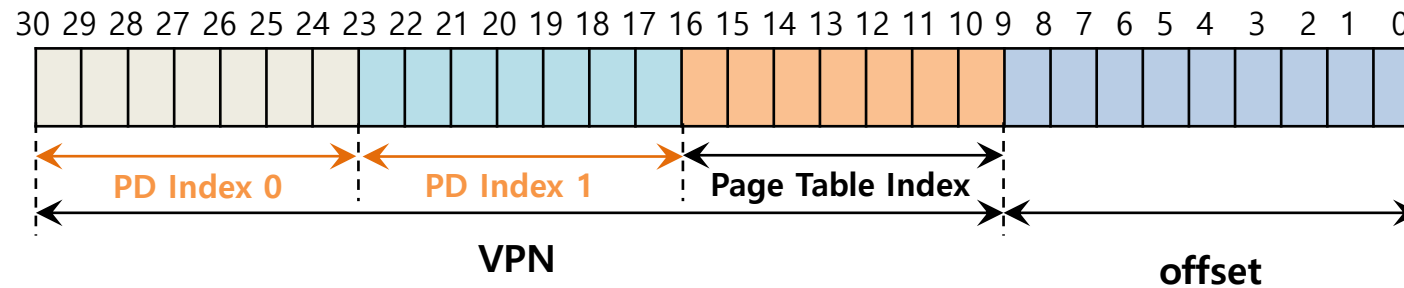


Flag	Detail
Virtual address	30 bit
Page size	512 byte
VPN	21 bit
Offset	9 bit
Page entry per page	128 PTEs

→ $\log_2 128 = 7$

More than Two Level : Page Directory

- If our page directory has 2^{14} entries, it spans not one page but 128
- To remedy this problem, we build a **further level** of the tree, by splitting the page directory itself into multiple pages of the page directory



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 translation 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

- ❑ Keeping 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
- ❑ PowerPC uses inverted page tables