Paging: Smaller Tables

Prof. Yongtae Kim

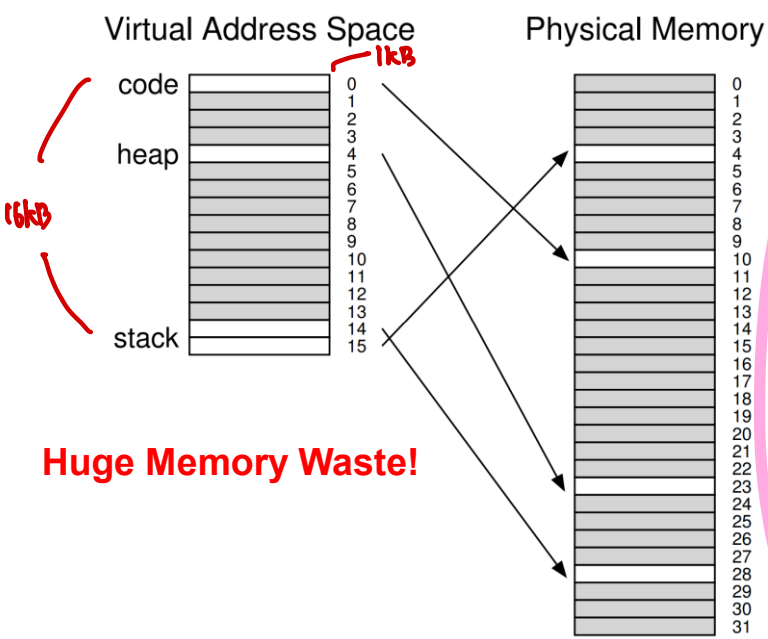
Computer Science and Engineering
Kyungpook National University

Page Table in Memory

- **Page tables are too big and thus consume too much memory**
  - A linear page table with 32-bit address space (2^{32}), 4KB pages (2^{12}), and a 4-byte page table entry requires $2^{32} \div 2^{12} \times 4B$ = 4MB in size
  - Every process have one page table (# of process ↑ → memory usage ↑)
- **One simple solution to reduce the size is to use bigger pages**
  - Using larger pages can reduce the size of the page table
  - 16KB pages (2^{14}) for 32-bit address space ➔ $2^{32} \div 2^{14} \times 4B$ = 1MB in size
  - 1MB pages (2^{20}) for the same space ➔ $2^{32} \div 2^{20} \times 4B$ = 16KB in size
- **Unfortunately, bigger pages leads to waste within each page**
  - This problem is known as internal fragmentation
  - Thus, most systems use relatively small page sizes in the common case: 4KB (as in x86) or 8KB (as in SPARCv9)
- **Are there other approaches to reduce the page table size?**
  1) Hybrid approach: paging + segmentation
  2) Multi-level page tables
  3) Inverted page tables

Observation: Many Invalid Page Table Entries

- **The key observation is many invalid page table entries**
  - e.g.) 16KB address space with 1KB pages (see figures):
    only four page are in use (4 of 16) and most of the page table is unused
    What if 32-bit address space? ➔ !!
  - Instead of having a single page table for entire address space of each process, why not have one per logical segment? → segmentation 을 섞어서 써보자!
    e.g.) three page tables; each for code, heap, and stack of the address space



Huge Memory Waste!

	PFN	valid	prot	present	dirty
0	10	1	r-x	1	0
	-	0	—	-	-
	-	0	—	-	-
	-	0	—	-	-
4	23	1	rw-	1	1
	-	0	—	-	-
	-	0	—	-	-
	-	0	—	-	-
	-	0	—	-	-
	-	0	—	-	-
	-	0	—	-	-
	-	0	—	-	-
	-	0	—	-	-
	-	0	—	-	-
14	28	1	rw-	1	1
15	4	1	rw-	1	1

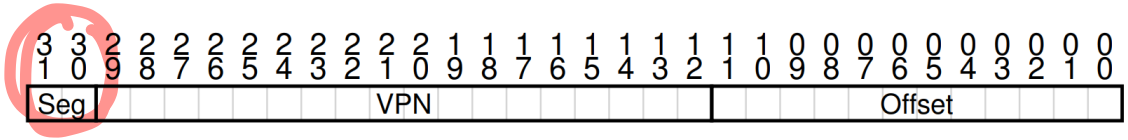
Many Invalid PTEs!

→ page table은 많은 공간을 유지하고 있다.

Hybrid Approach: Paging and Segments

page table size

- **Hybrid approach uses base and bound (limit) registers in MMU**
  - Base register: the physical address of the segment's page table
  - Bound register: the end of the page table (i.e. how many valid pages it has)
  - e.g.) 32-bit address space with 4KB page and four segments:
  - 2 MSBs indicate which segment is; 00: unused, 01: code, 10, heap 11: stack



  - Now, each process has three page tables associated with it
  - On context switch, three base and bound pairs (code, heap, stack segments) must be changed to reflect the location of newly-running process's page tables

- **On TLB miss, the hardware uses segment bits (SN) for base and bounds pairs**

  (SN에 해당하는 page table의 base 주소 + VPN x sizeof PTE) → 이 식에도 limit 조건 추가.

  - Then, look up the PTE by calculating the PTE address with SN and VPN

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

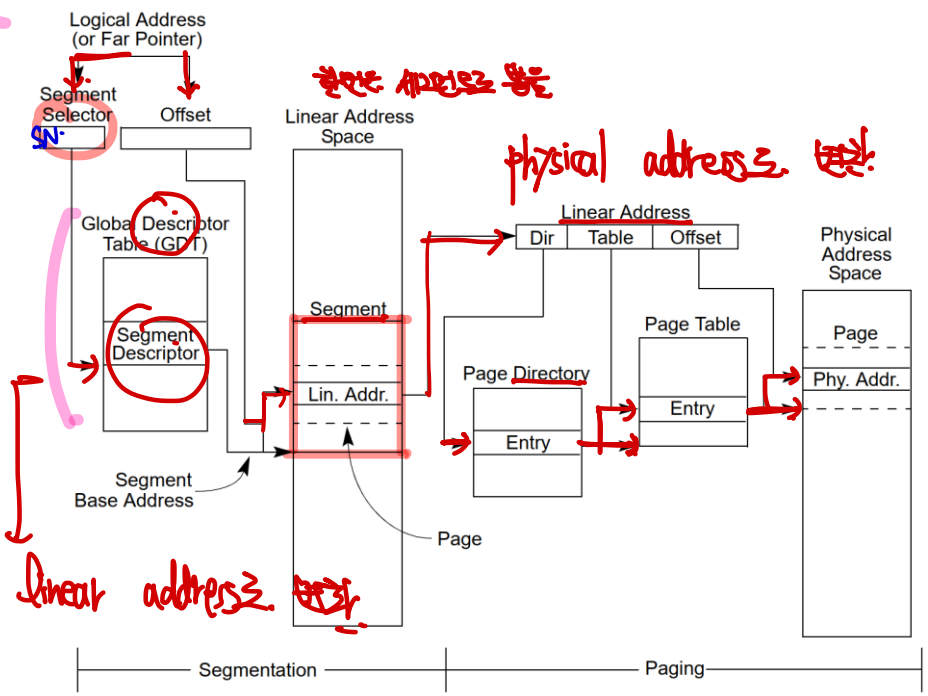
계산하는게 는었는.

Example: Intel Architecture Address Translation

- **This approach is not without problems**
  - Still requiring segmentation (not flexible, sparsely-used heap → page table waste)
  - Causing external fragmentation to arise again (variable sized page tables)

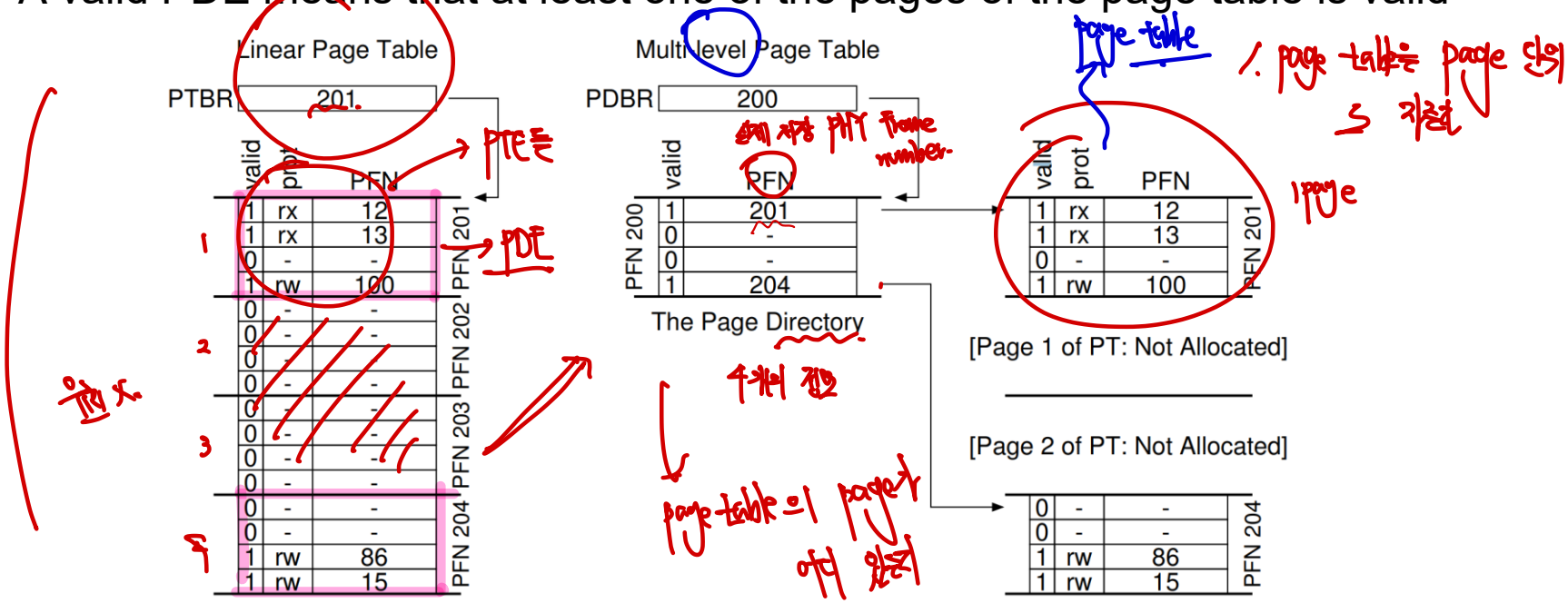
페이지 사이즈의 정수배가 되어야 한다.

- **IA-32 architecture address translation (segmentation & paging)**
  - The CPU core issues logical address (segment selector and offset)
  - The logical address is translated into the linear address by segmentation
  - The linear address is translated into the physical address by paging
  - In IA-32, segmentation must be used but the use of paging is optional
  - In Intel 64, segmentation is generally disabled (IA-32e paging)



Multi-level Page Tables: Concepts

- **Multi-level page table turns the linear page table into tree-style**
  - To do this, **1)** chop up the page table into page-sized units, **2)** if an entire page of PTEs is invalid, don't allocate that page of the table at all
  - To track whether a page is valid, use a new structure called page directory
- **Page directory contains one entry per page of the page table**
  - It consists of page directory entries (PDEs) that has a valid bit and a PFN
  - A valid PDE means that at least one of the pages of the page table is valid



Multi-level Page Tables: Pros and Cons

- **Multi-level page table have some obvious advantages**
  - It only allocates page-table space in proportion to the amount of address space that you are using; generally compact and supports sparse address spaces
  - If carefully constructed, each portion of the page table fits neatly within a page, making it easier to manage memory

    4KB의 관리가 쉬워짐
  - Note that linear page table requires a contiguous memory space (e.g. 4MB) while multi-level table allow each page table to be allocated non-contiguously

    linear page table은 연속적으로 할당
    되어야 했다면, 아주 고통이 있었을 것.
- **This multi-level paging approach also has a cost.**
  - On TLB miss, two loads from memory will be required to get the translation: 1) page directory and 2) PTE itself

    PDE에서 PFN → 그 PFN을 통해 PTE 접근
  - Time-space trade-off: Linear → size↑, $time_{access}$↓, Multi-level → size↓, $time_{access}$↑

    ↓

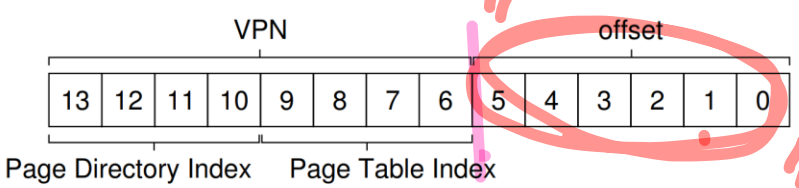
    TLB 때문에 더 중.
  - When TLB hit, the performance is obviously identical
  - Handling the page table lookup is more complicated

A Detailed Multi-Level Example (1)

- **Considering 16KB address space with 64-byte pages**
  - 14-bit virtual address space (2^{14}=16KB); 8-bit VPN and 6-bit offset (2^6=64)
  - Linear page table: 2^8 (256) entries; 0, 1 for code, 4, 5 for heap, 254, 255 for stack, rests are unused, 4-byte PTE → 256×4B=1KB in size
  - To build a two-level page table:
    1) Breaking the table into page-sized units; 1KB÷64B=16 pages and each page can hold 16 PTEs (4B×16=64B)
    2) Indexing into the page directory; the directory needs one entry per page of the page table (16 entries → 4 MSBs of VPN, called page-directory index; **PDIndex**)

       PDEAddr = PageDirBase + (PDIndex * sizeof(PDE))
    3) Indexing into the page table; 16 PTEs → next 4 bits of **PDIndex**, called page-table index; **PTIndex**)

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



0000 0000	code
0000 0001	code
0000 0010	(free)
0000 0011	(free)
0000 0100	heap
0000 0101	heap
0000 0110	(free)
0000 0111	(free)
.....	... all free ...
1111 1100	(free)
1111 1101	(free)
1111 1110	stack
1111 1111	stack

A Detailed Multi-Level Example (2)

- **Multi-level page table with actual values**
  - Two valid page directory entries (PDEs); 0th, 15th entry
  - @PFN:100; VPNs 0, 1 → code segment
    VPNs 4, 5 → heap segment
  - @PFN:101; VPNs 14, 15 → stack segment
  - Size comparison:
    Linear page table requires 16 pages in total
    Multi-level one requires 3 pages (1 PD + 2 valid PTs)
    ➔ Savings for larger space can be much greater
  - Address translation
    `0x3F80 (11_1111_1000_0000)`
    4 MSBs (1111) to index the PD: PFN 101
    Next 4 bits (1110) to index the PT: PFN 55
    Rest 8 bits (000000) for offset
    `PhysAddr=(PTE.PFN<<SHIFT)+offset`
    `= 00_1101_1100_0000 (0x0DC0)`

Page Directory

	PFN	valid?
0	100	1
1	—	0
2	—	0
3	—	0
4	—	0
5	—	0
6	—	0
7	—	0
8	—	0
9	—	0
10	—	0
11	—	0
12	—	0
13	—	0
14	—	0
15	101	1

Invalid PDEs

Page of PT (@PFN:100)

	PFN	valid	prot
0	10	1	r-x
1	23	1	r-x
2	—	0	—
3	—	0	—
4	80	1	rw-
5	59	1	rw-
6	—	0	—
7	—	0	—
8	—	0	—
9	—	0	—
10	—	0	—
11	—	0	—
12	—	0	—
13	—	0	—
14	—	0	—
15	—	0	—

1 page: 64B

0000 0000	code
0000 0001	code
0000 0010	(free)
0000 0011	(free)
0000 0100	heap
0000 0101	heap
0000 0110	(free)
0000 0111	(free)
.....	... all free ...
1111 1100	(free)
1111 1101	(free)
1111 1110	stack
1111 1111	stack

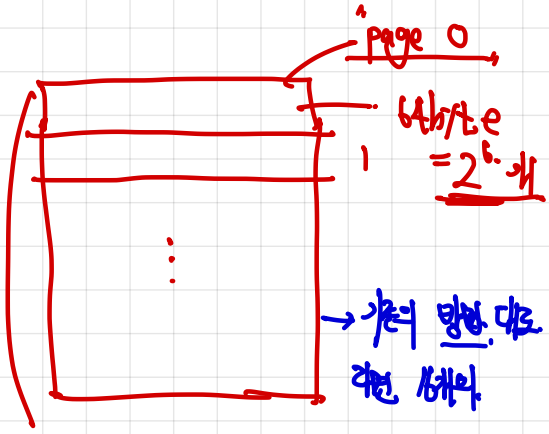
Page of PT (@PFN:101)

	PFN	valid	prot
0	—	0	—
1	—	0	—
2	—	0	—
3	—	0	—
4	—	0	—
5	—	0	—
6	—	0	—
7	—	0	—
8	—	0	—
9	—	0	—
10	—	0	—
11	—	0	—
12	—	0	—
13	—	0	—
14	55	1	rw-
15	45	1	rw-

1 page: 64B

"가로"

2^8개

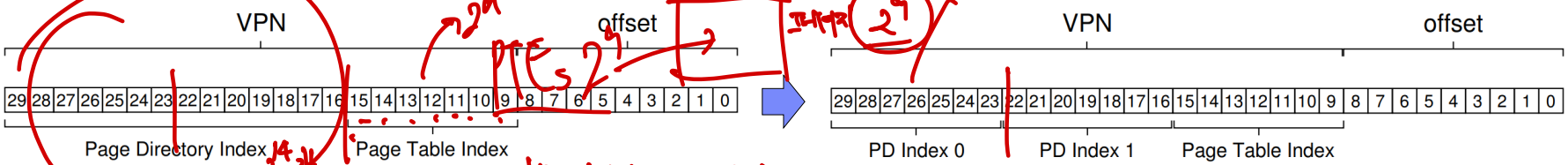


"16kB - address space"

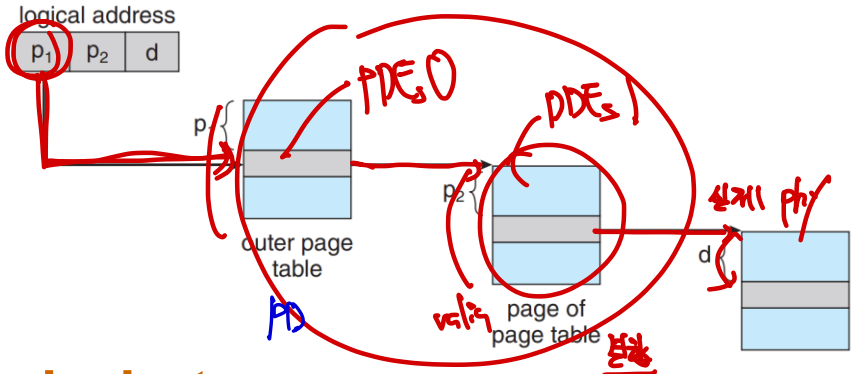
"$2^8 \times 2^6 = 16kB$"

More than Two Levels

- **Considering a deeper multi-level table to see the usefulness**
  - 30-bit address space with a 512-byte page (21-bit VPN & 9-bit offset)
  - For PTE size of 4B, each page can hold 128 PTEs (512÷4B)
  - Indexing PTEs & PDEs requires 7 (2^7=128) & 14 (30-7-9) bits, respectively
  - Page directory has 2^{14} entries, which does not fit one page (needs 128 pages)

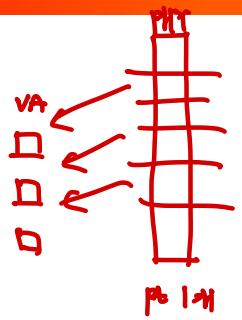


- **To remedy this, splitting the PD itself into multiple pages**
  - PT index: 7 bits, PD index0: 7 bits (upper), PD index1: 7 bits (lower)
  - Translation sequence:
    1) fetch PDE0 from upper directory
    2) fetch PDE1 from lower directory if PDE0 is valid
    3) fetch PTE if PDE1 is valid



- **A deeper table requires more works but can save more memory**

Translation Process: Remember the TLB



- **Control flow of address translation in multi-level paging**
  - TLB hit: directly memory access
  - TLB miss: hardware needs to perform the full multi-level lookup (see codes)
- **Inverted Page Tables**
  - Instead of having per-process page table, we keep a single page table that has an entry for each physical page of the system
  - Finding the correct entry is a matter of searching through this data structure

page table 이

하나의 page table.

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True)   // TLB Hit
  if (CanAccess(TlbEntry.ProtectBits) == True)
    Offset   = VirtualAddress & OFFSET_MASK
    PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
    Register = AccessMemory(PhysAddr)
  else
    RaiseException(PROTECTION_FAULT)
else                   // TLB Miss
  // first, get page directory entry
  PDIndex = (VPN & PD_MASK) >> PD_SHIFT
  PDEAddr = PDBR + (PDIndex * sizeof(PDE))
  PDE     = AccessMemory(PDEAddr)
  if (PDE.Valid == False)
    RaiseException(SEGMENTATION_FAULT)
  else
    // PDE is valid: now fetch PTE from page table
    PTIndex = (VPN & PT_MASK) >> PT_SHIFT
    PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
    PTE     = AccessMemory(PTEAddr)
    if (PTE.Valid == False)
      RaiseException(SEGMENTATION_FAULT)
    else if (CanAccess(PTE.ProtectBits) == False)
      RaiseException(PROTECTION_FAULT)
    else
      TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
      RetryInstruction()
```

→ VPN이 있다면

search-time ↑

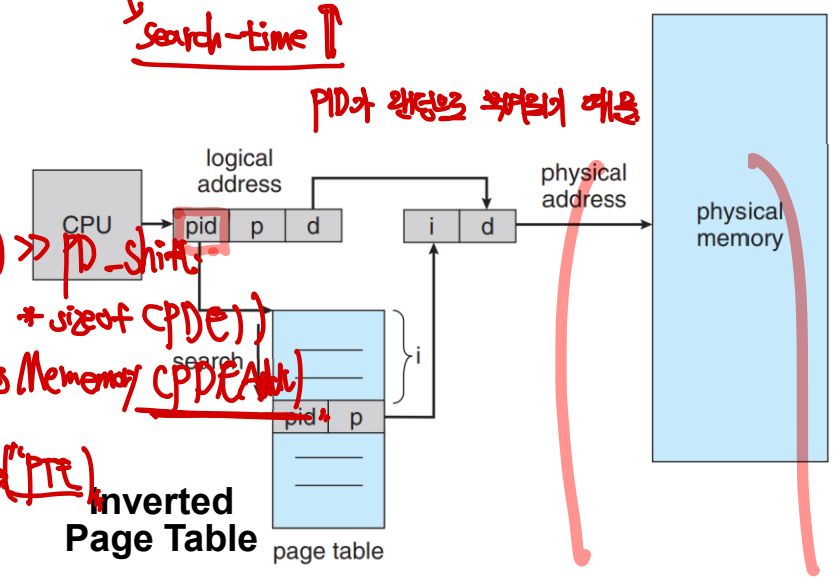
PID가 랜덤으로 확률의 예문

(VPN & PD_MASK) >> PD_SHIFT

PDBR + (PDIndex * sizeof(PDE))

PDE = AccessMemory(PDEAddr)

(PTE)



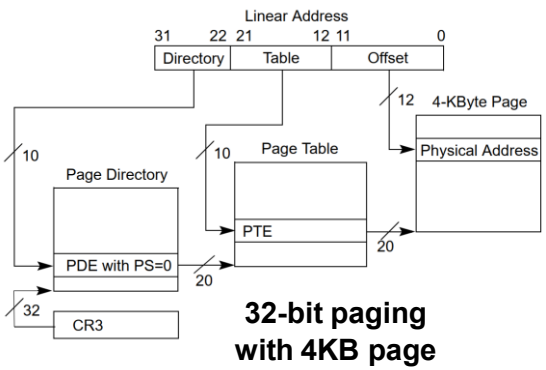
Example: Paging at Intel Architecture

- **Intel processor architecture offers three paging modes**

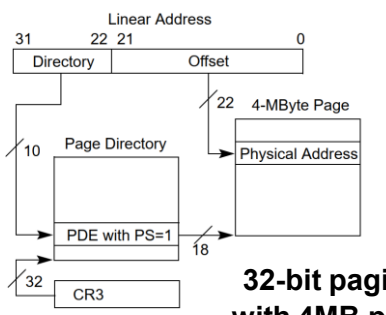
Mode	Linear Space	Physical Space	Page Size
32-bit	32-bit address (4G)	32-, 40-bit address (4G, 1T)	4K(2^{12}), 4M(2^{22})
PAE	32-bit address (4G)	52-bit address (4P)	4K(2^{12}), 2M(2^{21})
IA-32e	48-bit address (256T)	52-bit address (4P)	4K(2^{12}), 2M(2^{21}), 1G(2^{30})

32bit

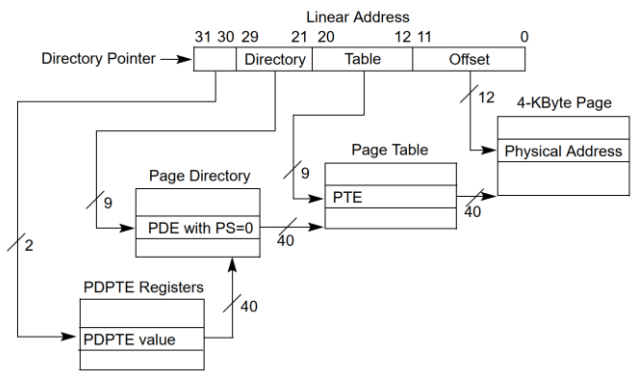
64bit



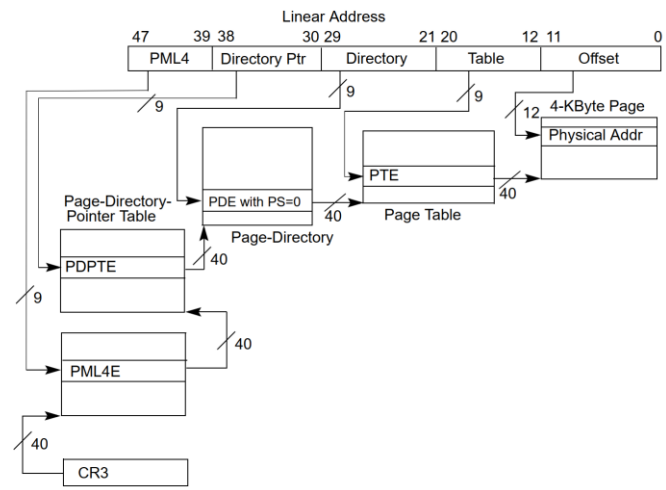
32-bit paging with 4KB page



32-bit paging with 4MB page



PAE paging with 4KB page



IA-32e paging with 4KB page (Intel 64 architecture)

Summary

- **Since a page table may include many invalid PTEs, multi-level page table can be used to save the memory**
  - It is generally compact and supports sparse address spaces
  - The address is divided into PD index, PT index, and offset
  - PD index can be further split into multiple pieces → PDIndex의 크기가 page의 크기와 같아야 메모리 관리가 수월하기 때문.