

# 9. Main Memory

ECE30021/ITP30002 Operating Systems

# Agenda

---



- Background
- Swapping
- Contiguous memory allocation
- Paging
- Structure of page table
- Segmentation
- Example: Intel Pentium

# Background

## ■ Importance of memory management

- In multitasking system, several processes may exist in memory
- Memory is central of modern computer system

## ■ Memory structure

- A large array of bytes each with its own **address**.
- Memory unit is interested only in the sequence of memory addresses.

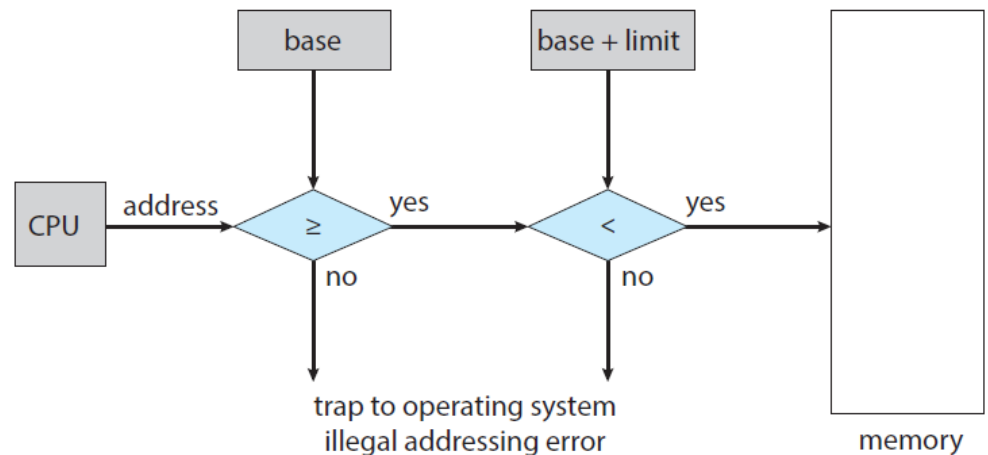
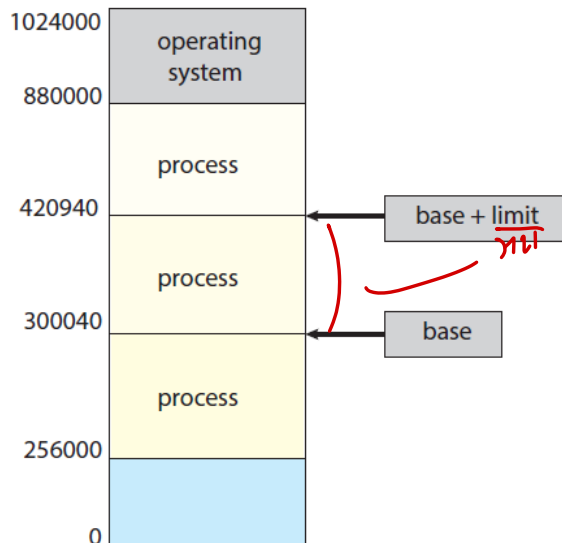
**Main memory**

0				...
4				...
8				...
12				...
16				...
...	...			

# Basic Hardware

## ■ Protection from other processes

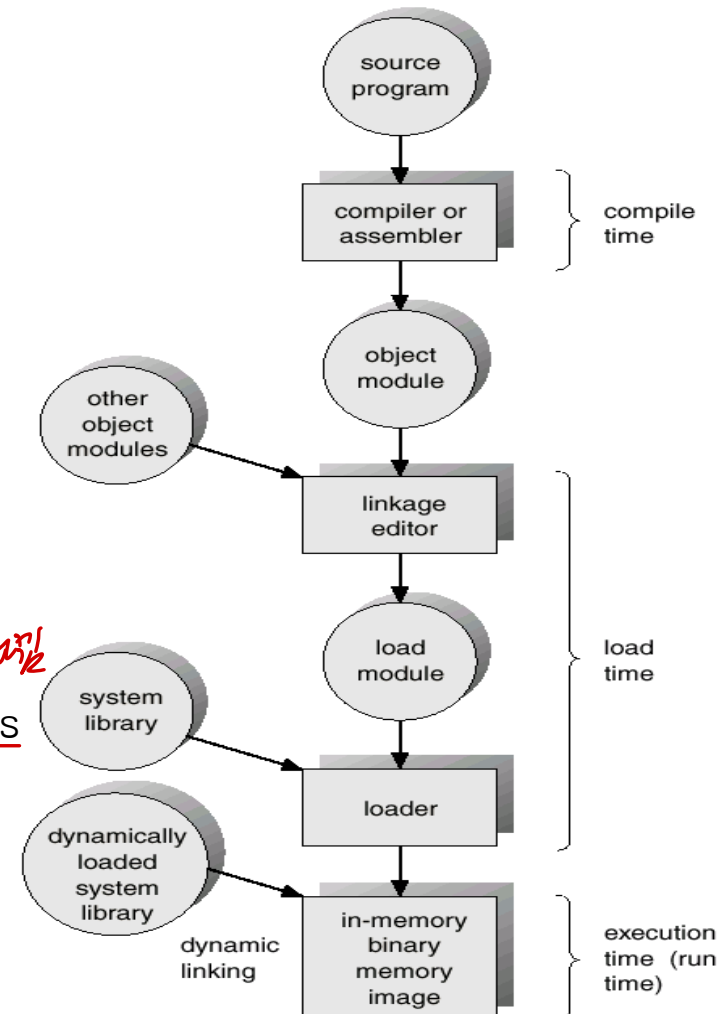
- Each process has a separate memory space.
- Can be implemented by base register and limit register.
- CPU H/W compares every address generated.
- Base/limit registers can be loaded by privileged instruction.



하드웨어적으로 이걸이런식으로 논리적으로 짤

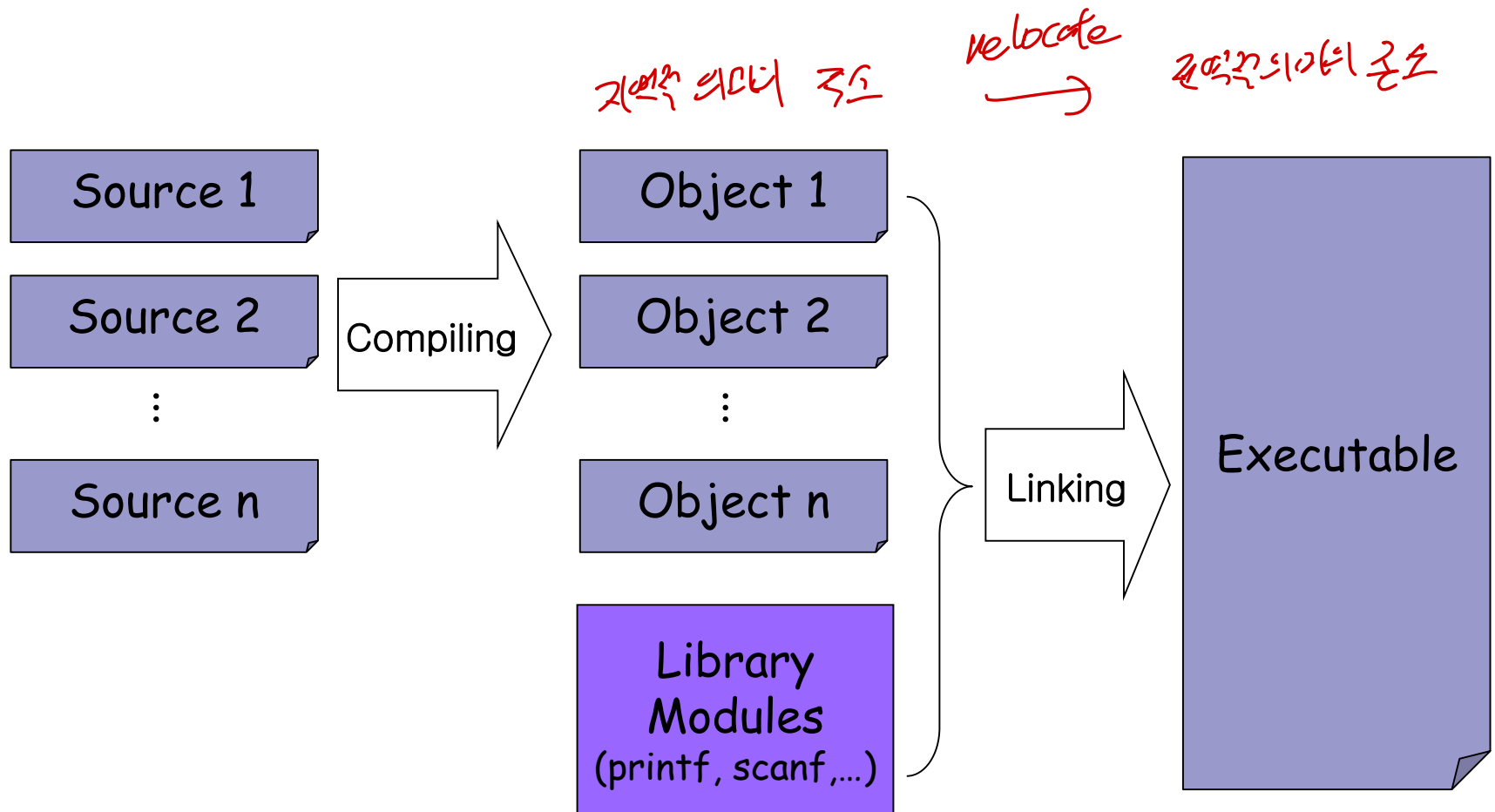
# Address Binding

- Most systems allow a user process to reside in any part of the physical memory.
- Representation of address *실제 주소*
  - Source program: symbolic address  
Ex) *count* *변수 이름, 함수 이름*
  - Compiler
    - Symbol → relocatable address  
Ex) 14 bytes from beginning of the module
  - Linkage editor or loader *로딩하는 소프트웨어*
    - Relocatable address → absolute address  
Ex) 0x74014
- **Address binding**: mapping one address space to another



# Building and Running Programs

- Compile ➔ linking ➔ loading ➔ execution



# Address Binding

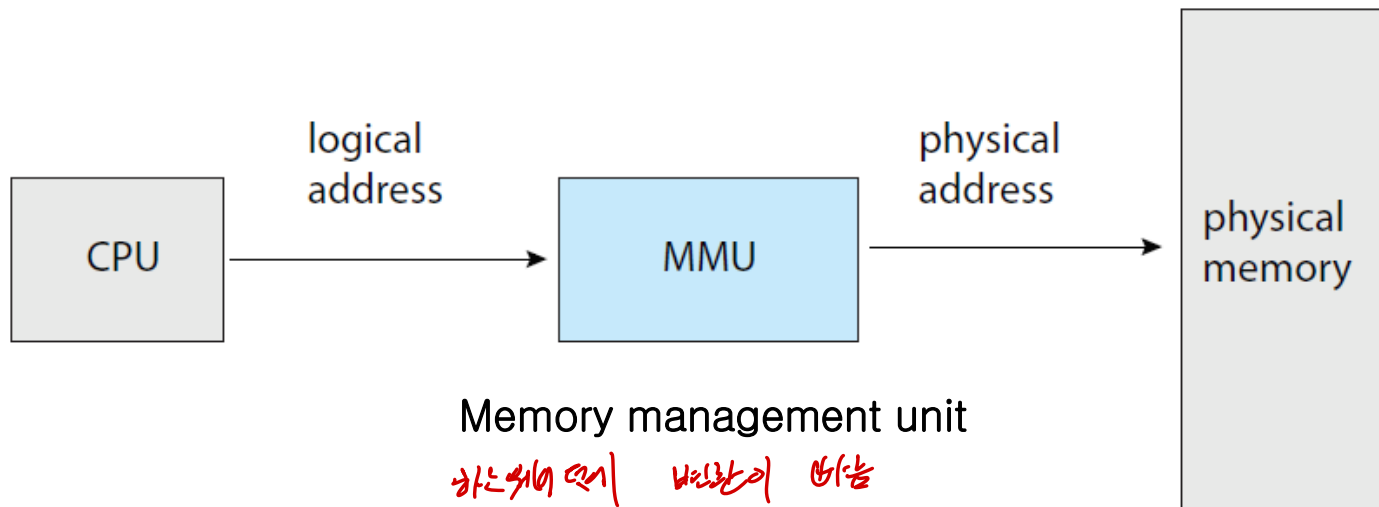


## ■ Address binding time

- Compile time: absolute code
  - MS-DOS .com file
  - If starting location changes, the program should be recompiled
- Load time: relocatable code
  - If starting location changes, the program should be reloaded.
- Execution time
  - Ex) Systems that support logical addressing
    - Used by most general-purpose OS's
    - Requires H/W support for address mapping

# Logical vs. Physical Address Space

- **Logical address**: address generated by CPU in common cases
- **Physical address**: address seen by memory unit



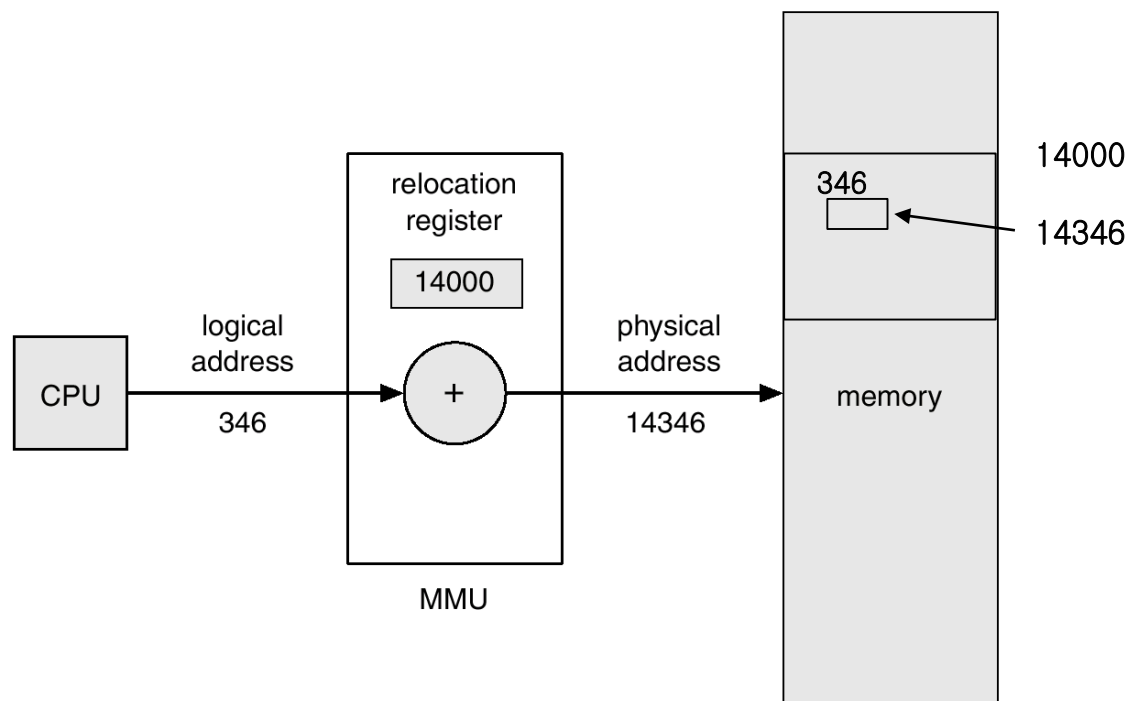


# Logical vs. Physical Address Space

- Logical address vs. physical address
  - In compile-time and load-time address binding
    - Logical address = physical address
  - In execution-time address binding
    - Logical (virtual) address  $\neq$  physical address
- Address spaces
  - **Logical address space**: set of all logical addresses generated by a program
  - **Physical address space**: set of all physical addresses corresponding to these logical addresses
- Run-time address mapping is done by **memory-management unit** (MMU)

# Logical vs. Physical Address Space

- A simple method of address mapping using relocation register (base register)
  - User program never sees physical addresses



# Dynamic Linking and Shared Library

- **Static linking**: all object modules are combined into the binary program image
- **Dynamic linking**: linking is postponed until execution time
  - Ex) System libraries
    - Stub: a small piece of code that indicates ...
      - How to locate the appropriate library routine
      - How to load the library if it is not ready
    - After a library routine is dynamically linked, that routine is executed directly
      - stub replaces itself with address of the routine and execute it.
- **Advantages of dynamic linking**
  - Library code can be shared among processes.
    - Multiple versions of a library can coexist
  - Library update doesn't require re-linking.
- Dynamic linking requires help from OS.

# Agenda

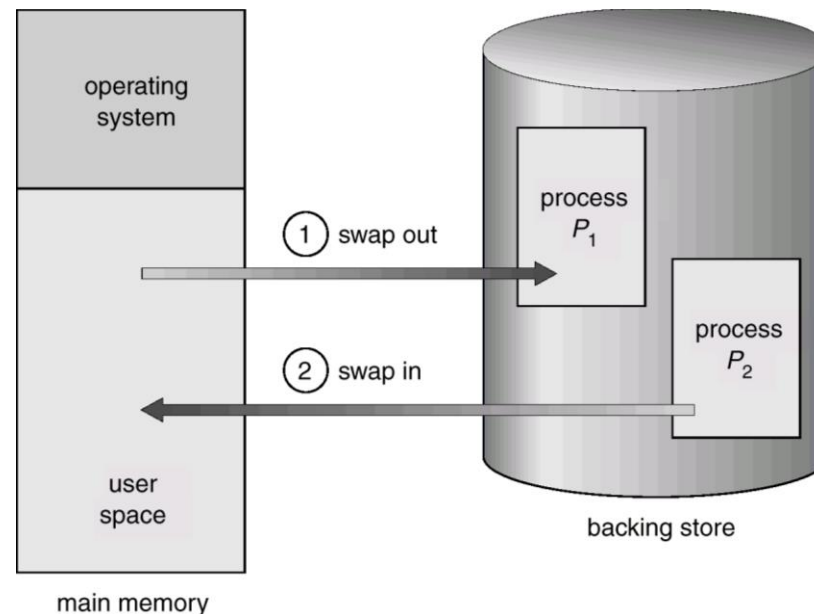
---



- Background
- Swapping
- Contiguous memory allocation
- Paging
- Structure of page table
- Segmentation
- Example: Intel Pentium

# Swapping

- When a process starts, it must be in memory for execution.
- However, a process can be swapped to a backing store and back into memory to continue execution.



# Swapping

---



- A process swapped-out will be swapped back into the same memory space it occupied previously
  - Exception: execution-time binding
- Backing store: fast disk (usually, separated from file system)
  - Should be large enough to accommodate copies of all memory images for all users
  - Must provide direct access to memory images
- Dispatching
  - Firstly see if the process selected by scheduler is in memory.
  - If not, it swaps out a process and swaps in the desired process

# Swapping

메모리 또는 디스크 공간 부족 (overflow)

## ■ Context-switch time in swapping system

Ex) size of a process: 100 MB

disk transfer rate: 50 MB/s

average latency: 8 ms SSD는 읽고 쓰기 속도만 높음

Then, swapping time:  $(100\text{MB} / 50\text{MB/s} + 8\text{ms}) * 2 = 4,016 \text{ ms}$

\* Swapping time is affected by amount of memory swapped

□ We would need to swap only what is actually used.

## ■ Time slice should be long relative to swap time

## ■ Many systems use modified versions of swapping

- Swapping is enabled only when memory usage exceeds some threshold.

# Agenda

---



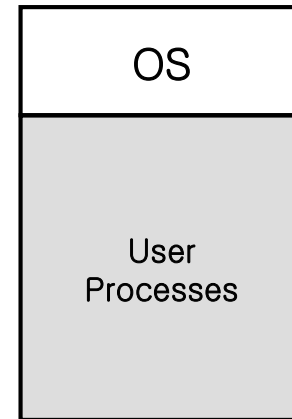
- Background
- Swapping
- Contiguous memory allocation
- Paging
- Structure of page table
- Segmentation
- Example: Intel Pentium



# Contiguous Memory Allocation

- Memory is usually divided into two partitions

- Resident OS
  - High (or low) memory with interrupt vector
- User processes
  - The other part of memory

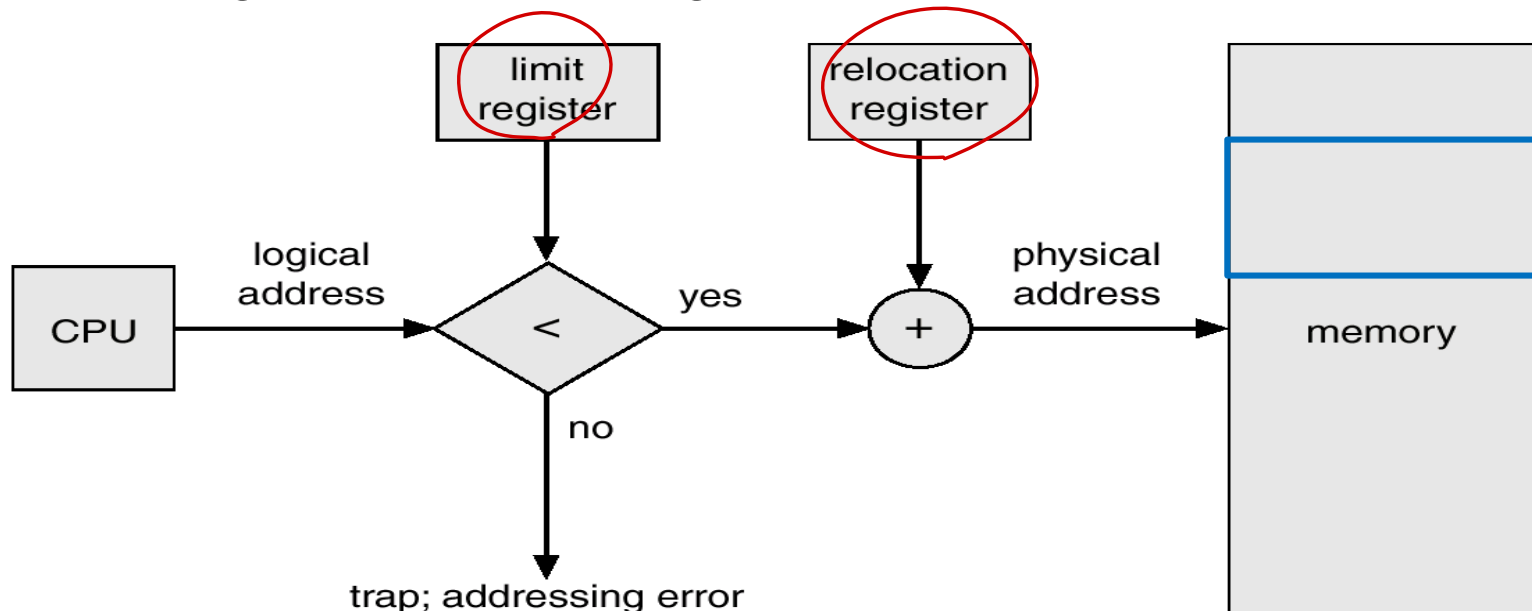


- Several user processes can reside in memory at the same time

→ How to allocate memory to processes waiting in input queue?

# Memory Mapping and Protection

- Memory mapping and protection are provided by **relocation register** and **limit register**
  - Add memory access, MMU maps the logical address dynamically.
  - Relocation and limit registers are loaded by dispatcher during context switching.



# Memory Allocation

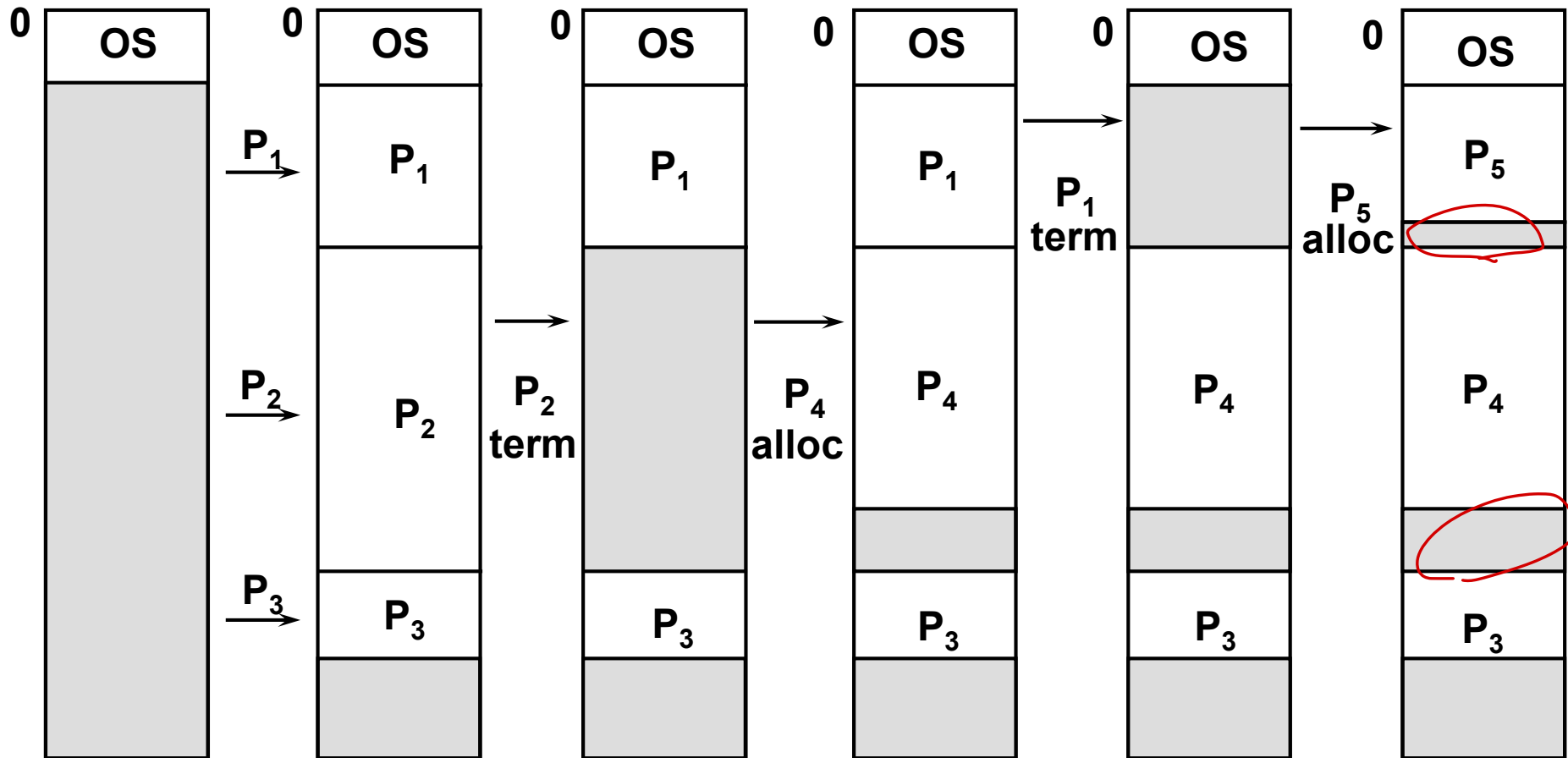


- Variable partition scheme (MVT)
  - Initially all available memory forms a large block of free memory (hole)
  - If a process arrives and needs memory, search a hole large enough for this process.
  - If a process terminates, it releases its memory
    - Adjacent holes can be merged
- \* MVT: Multiprogramming with a Variable number of Tasks

# Memory Allocation

## ■ Variable-partition scheme

fragmentation (파편화) 발생



# Fragmentation



- **External fragmentation:** free memory space broken into little pieces
  - Sometimes, it's impossible to allocate large continuous memory although total size of free memory larger than the required memory
  - 50-percent rule (analysis of first fit)
    - Given  $N$  allocated blocks, another  $0.5N$  blocks will be lost to fragmentation. ( $1/3$  of memory may be unusable)
  - Fragmentation of main memory also affects backing store

# Fragmentation



- Memory allocated to a process may slightly larger than requested memory.

Ex) allocating 18462 bytes from hole whose size is 18464 bytes

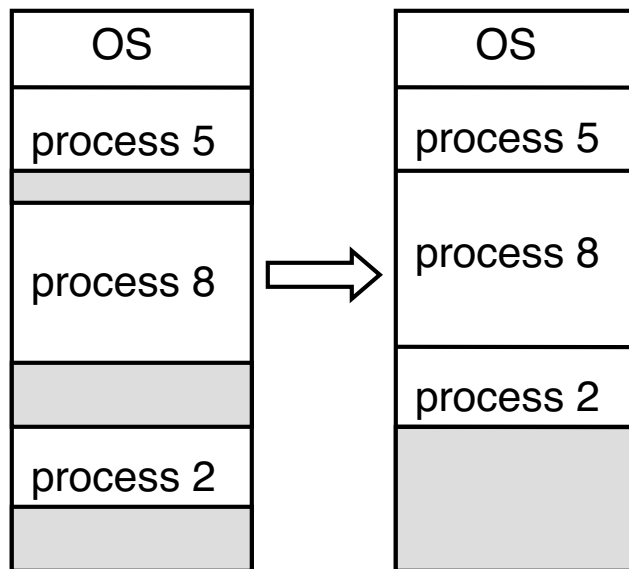
- Reduce overhead to keep small holes.
- Generally, physical memory can be broken into fixed-size blocks.

- **Internal fragmentation**

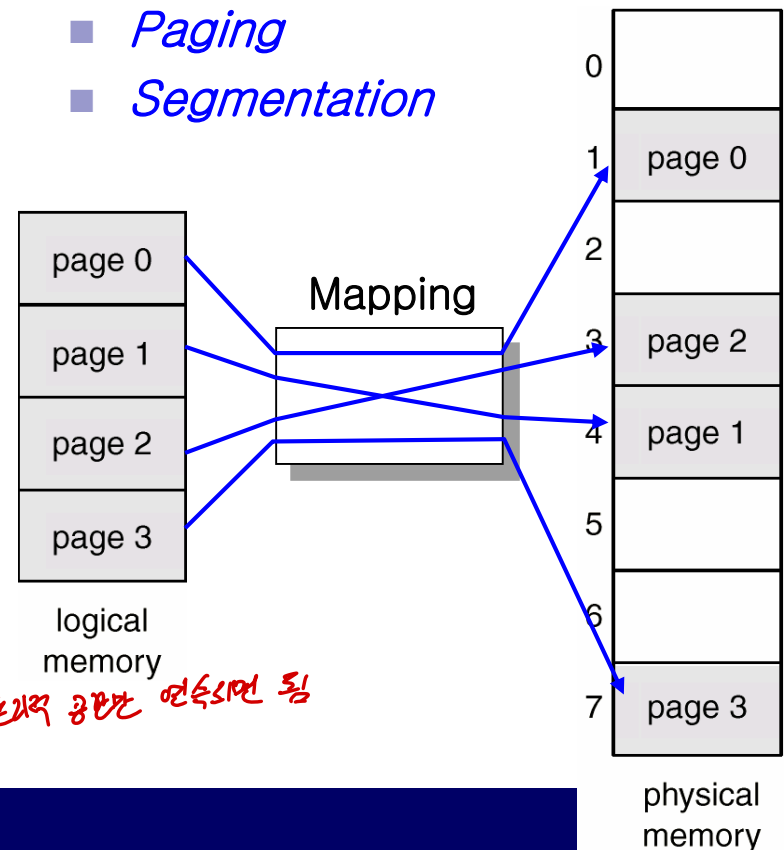
- Difference between the amounts of allocated memory and the requested memory.

# Fragmentation

- Remedy of external fragmentation: compaction
  - Possible only if relocation is dynamic and is done at execution time
  - Expensive



- Alternative remedy: permit logical address space of the processes to be noncontiguous
  - *Paging*
  - *Segmentation*



# Agenda

---

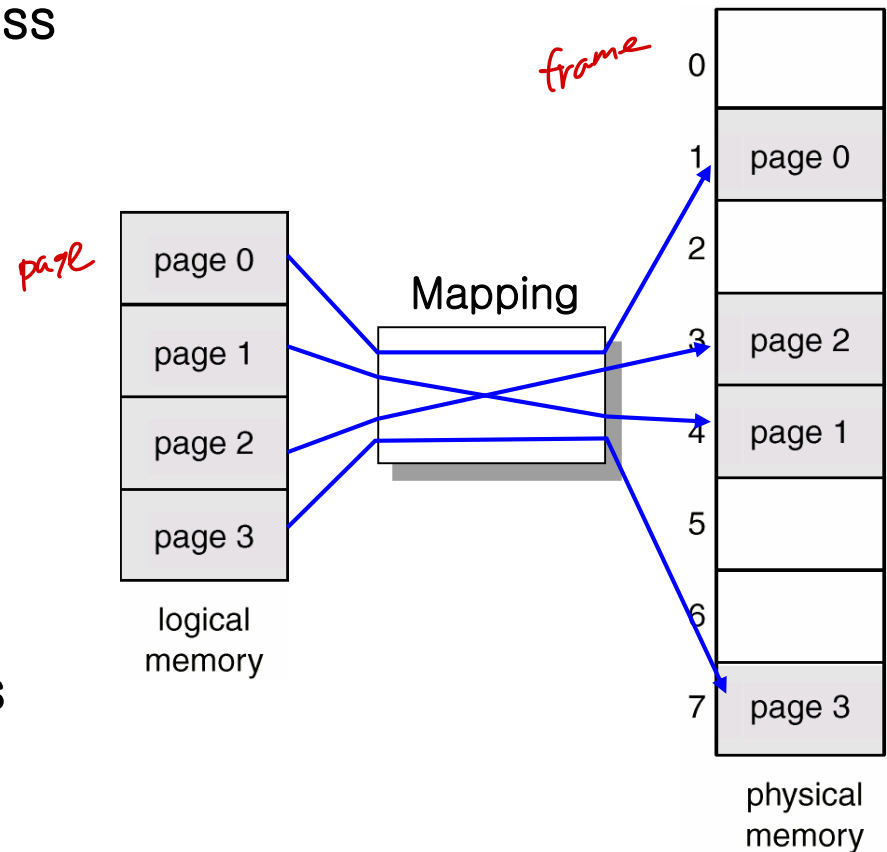


- Background
- Swapping
- Contiguous memory allocation
- Paging
- Structure of page table
- Segmentation
- Example: Intel Pentium



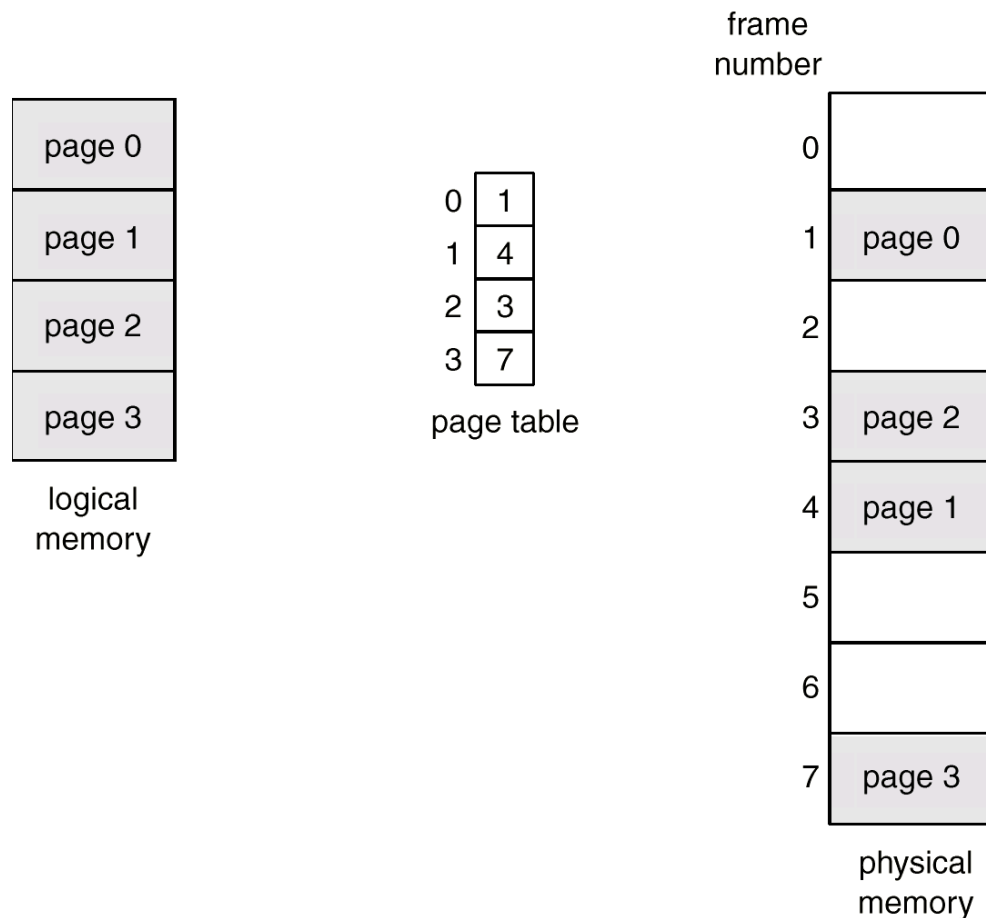
# Paging

- A memory-management scheme that permits the physical address space of a process to be noncontiguous.
- Physical memory: consists of **frames** (fixed-size blocks)
- Logical memory: consists of **pages** (fixed-size blocks)  
*content를 관리하는 단위*
- Pages are mapped with frames through **page table**



# Paging

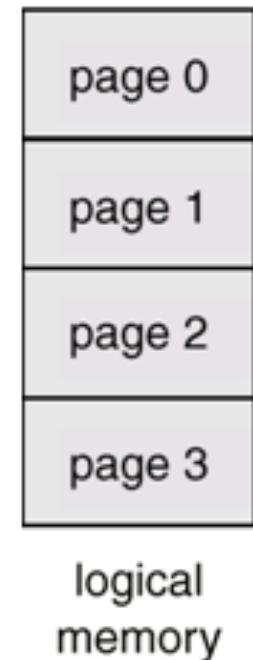
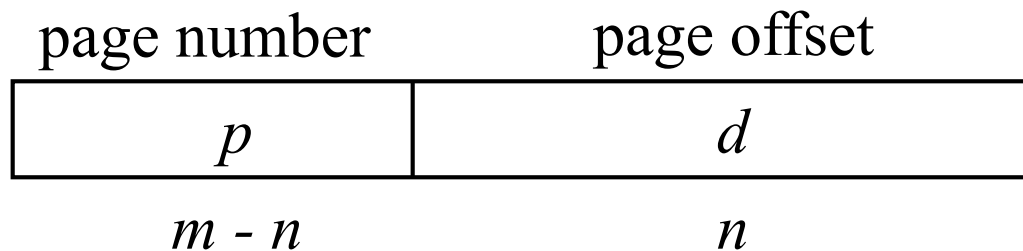
- Paging model of logical and physical memory



# Paging

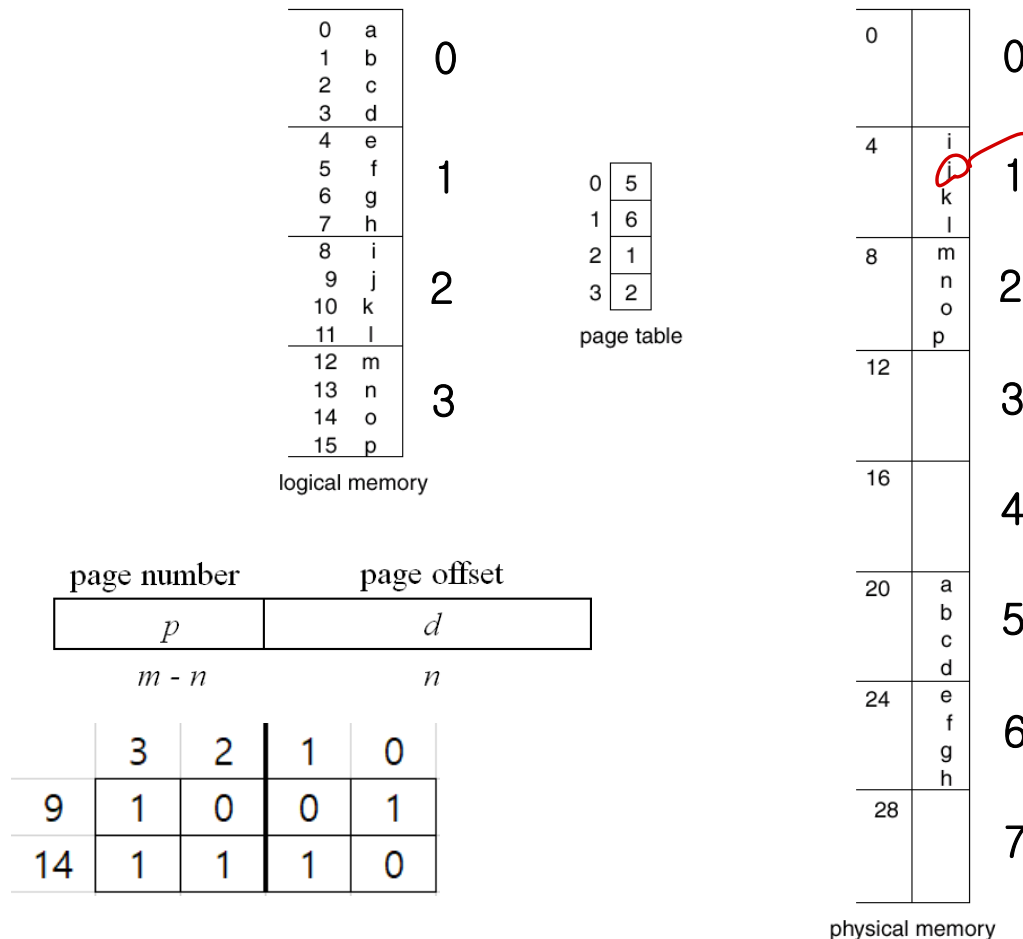
- Logical address consists of a *page number* and a *page offset*
  - Size of logical address space:  $2^m$
  - Page size:  $2^n$

	page number		page offset		
	10000	1000	100	10	1
35821	3	5	8	2	1



# Paging

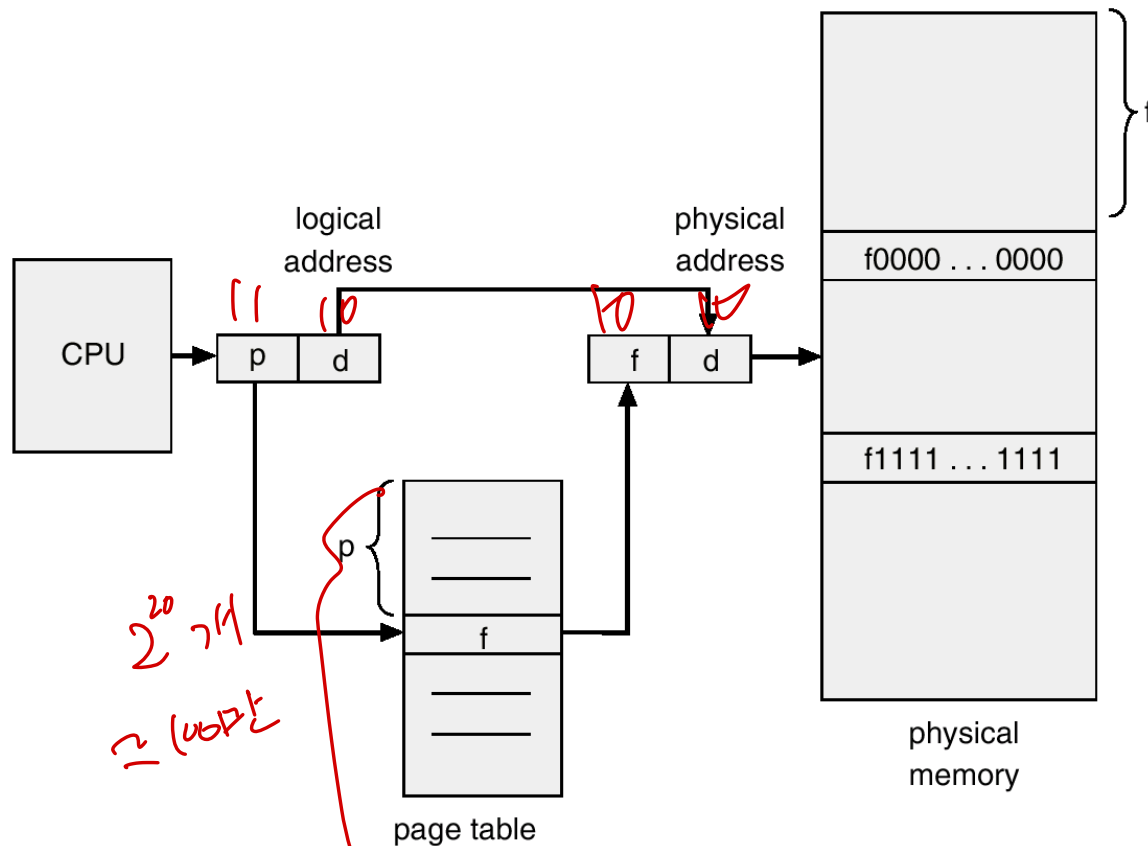
## ■ An example of paging



5 = 0/01  
frame offset

# Paging

## ■ Paging hardware



# Paging

## ■ Overhead of paging

- Page table
- Internal fragmentation
  - In average, half page per process

Note! In paging, no external fragmentation

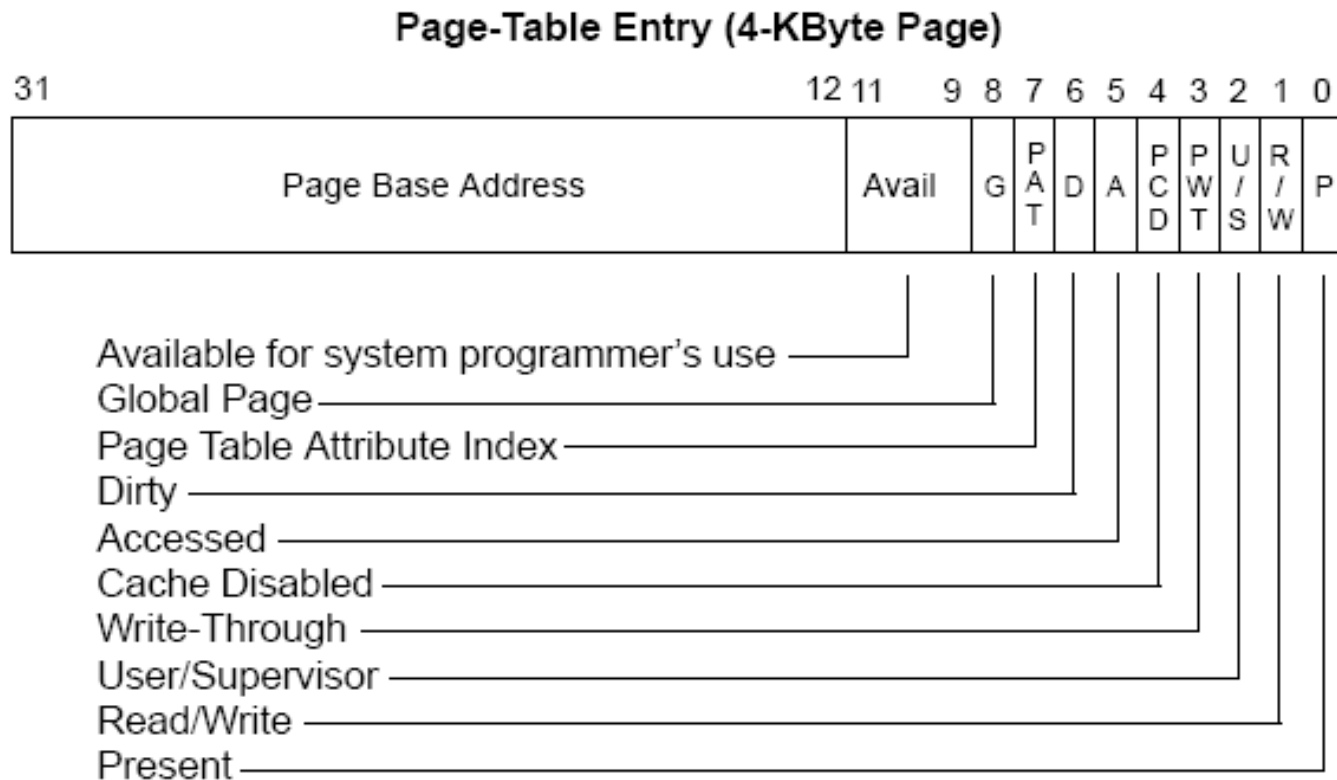
## ■ In many systems ...

- Page size: 4 KB ~ 4 MB
- Page table entry: 4 bytes (32 bits)

페이지 크기: 4KB ~ 4MB  
페이지 테이블 엔트리: 4바이트 (32비트)

# X86 Page-Table Entry

## ■ Format of x86 page table entry



# Frame Table

---

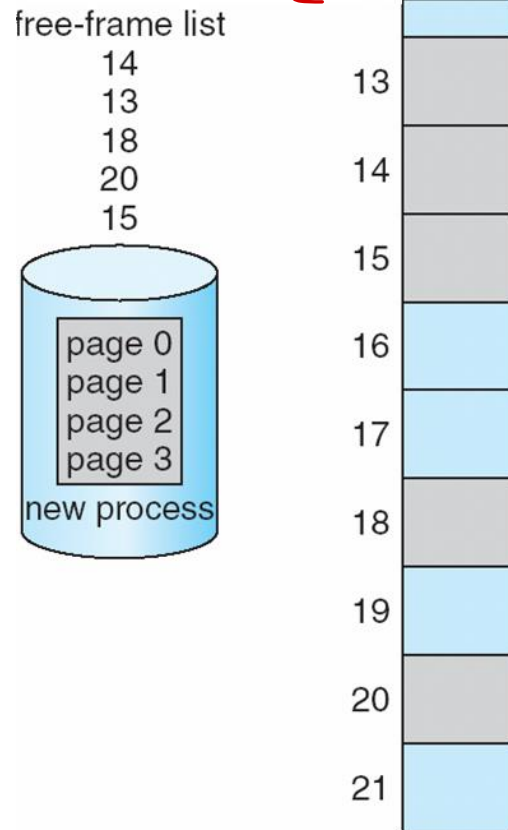


- OS keeps information about frames in a **frame table**
  - # of total frames
  - Which frames are allocated
  - Which frames are available
  - ETC.
- OS also maintains **a list of free frames**



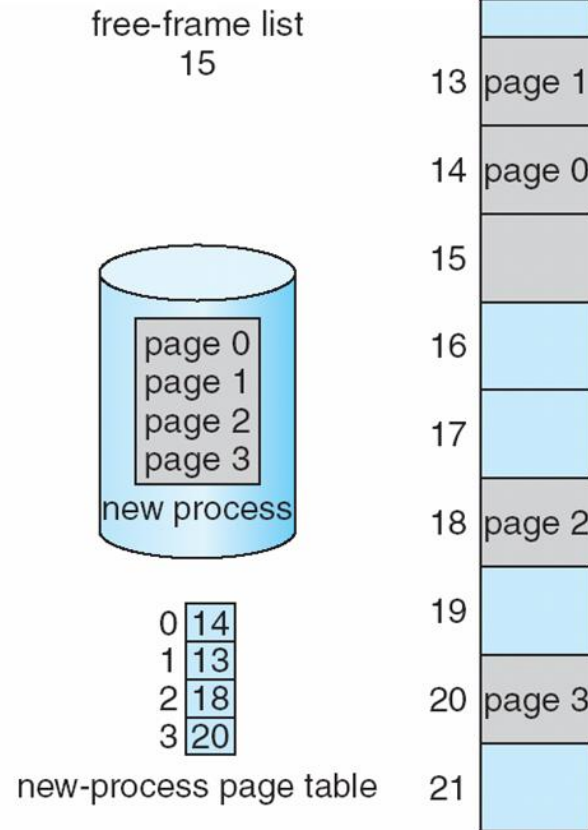
# Free Frames

OS가 OS(2) memory 할당하고  
가이드라인



(a)

**Before allocation**



(b)

**After allocation**

# Hardware Support



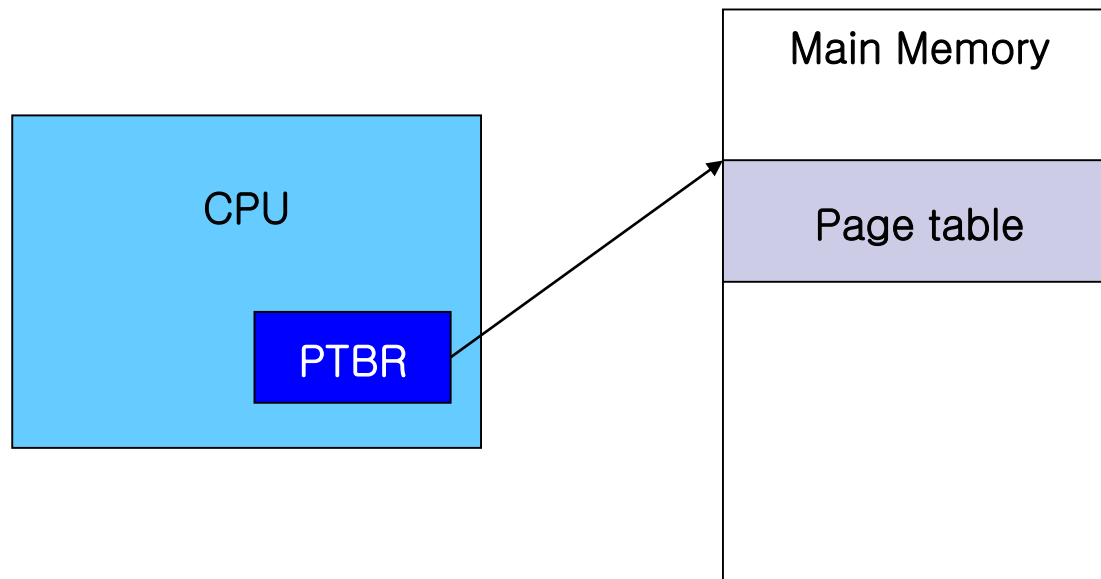
- Most OS's allocate a page table to each process.
  - Pointer to the page table is stored with register values (in PCB).
  - In context-switching, dispatcher also loads the correct H/W page table.
  
- H/W implementation of page table
  - A set of dedicated registers
    - Efficiency is very important.
  - Ex) DEC PDP-11: page table (8 entries) is kept in fast registers  
→ not feasible if page table is large

# Hardware Support

## ■ Page-table base register (PTBR)

- Page table is stored in main memory
- PTBR points the page table (eg. CR3 register of x86)

cf. Remaining problem: overhead to access page table in main memory (requires two accesses to access a byte.)



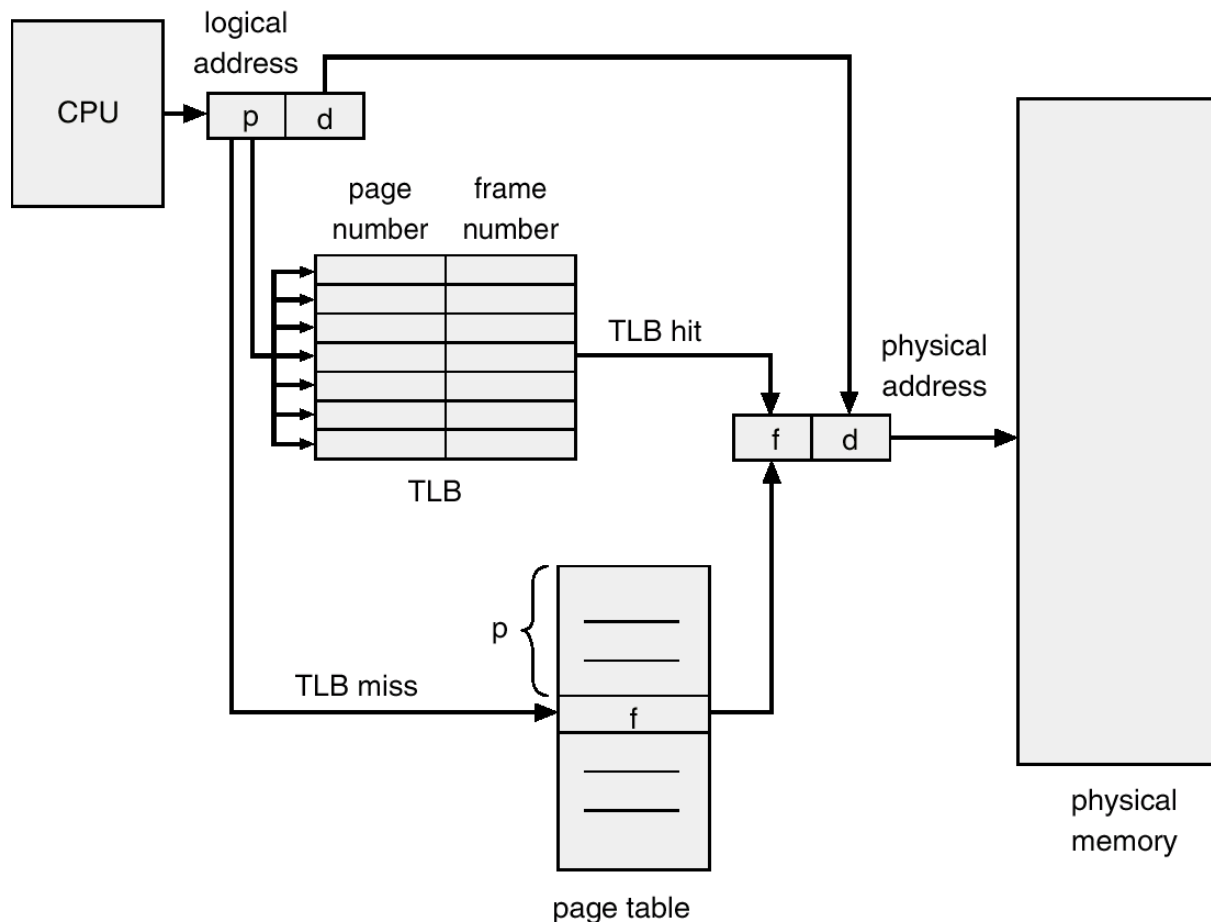
# Hardware Support



- **Translation look-aside buffer (TLB):** small fast look up H/W cache.
  - Associative, high-speed memory, consists of pairs (key, value)
    - If page number of logical address is in TLB, delay is less than 10% of unmapped memory access
    - Otherwise (TLB miss), access page table in main memory
      - Insert (page number, frame number) into TLB
      - If TLB is full, OS select one for replacement
  - Some TLB's store address-space identifiers (ASID) in each entry (ex: MIPS)
    - ASID identifies process
    - Processors w/o ASID (ex: x86) flush entire TLB on process switch

# Hardware Support

## ■ Paging using TLB



# Associative Memory



## ■ Associative memory – parallel search

Page #	Frame #

## ■ Address translation (p, d)

- If p is in associative register, get frame # out
- Otherwise, get frame # from page table in memory

# Protection



- Each frame has a **protection bits**
  - Read-write / read only
  - Attempt to write to a read-only page causes H/W trap
    - Extension: read-only, read-write, execute-only, ...
      - Combination of them
- Each entry of page table has **valid-invalid bit**
  - OS does not allows to access the page if it is invalid
- Rarely does a process use all of its address range
  - Page table can be wasting memory
    - some system provides **page-table length register (PTLR)**

# Valid-Invalid Bit

## Valid-invalid bit in a page table

00000	page 0
	page 1
	page 2
	page 3
	page 4
10,468	page 5
12,287	

frame number		valid-invalid bit
0	2	v
1	3	v
2	4	v
3	7	v
4	8	v
5	9	v
6	0	i
7	0	i

page table

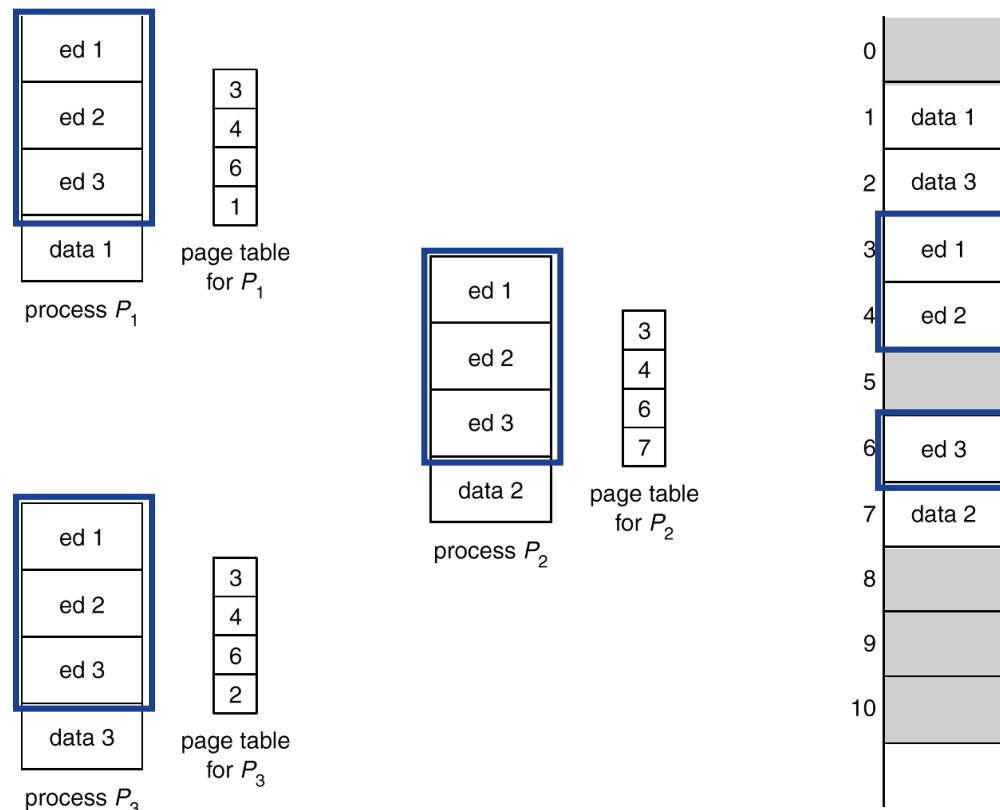
0	
1	
2	page 0
3	page 1
4	page 2
5	
6	
7	page 3
8	page 4
9	page 5
	⋮
	page $n$



# Shared Pages

- Paging provides possibility to share common code

Ex) Process  $P_1$ ,  $P_2$ ,  $P_3$  runs the same text editor with different data (editor should be **non-self-modifying code**)



# Agenda

---



- Background
- Swapping
- Contiguous memory allocation
- Paging
- Structure of page table
- Segmentation
- Example: Intel Pentium

# Hierarchical Paging

- Motivation: If a system supports large logical address space, page table itself becomes significant overhead

Ex) 32-bit address space, page size is 4 KB( $2^{12}$ )

size of page table =  $2^{(32-12)} > 1$  Million

- Difficult to allocate **contiguous** memory → *복합성 문제*

*한 번의 포인터링*

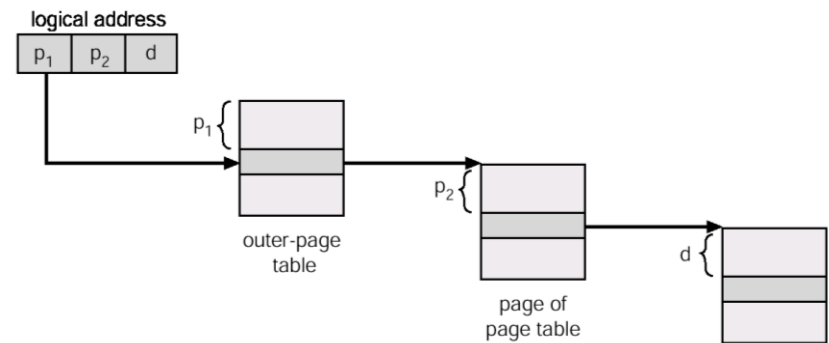
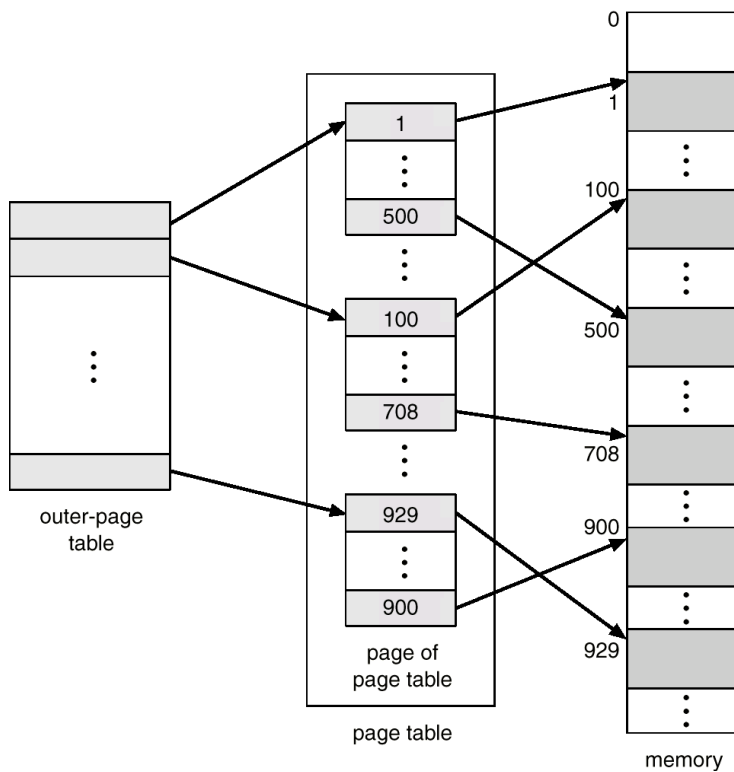
- Hierarchical paging

- Page table itself is also paged

- Structure of logical address

page number		page offset
$p_i$	$p_2$	$d$
10	10	12

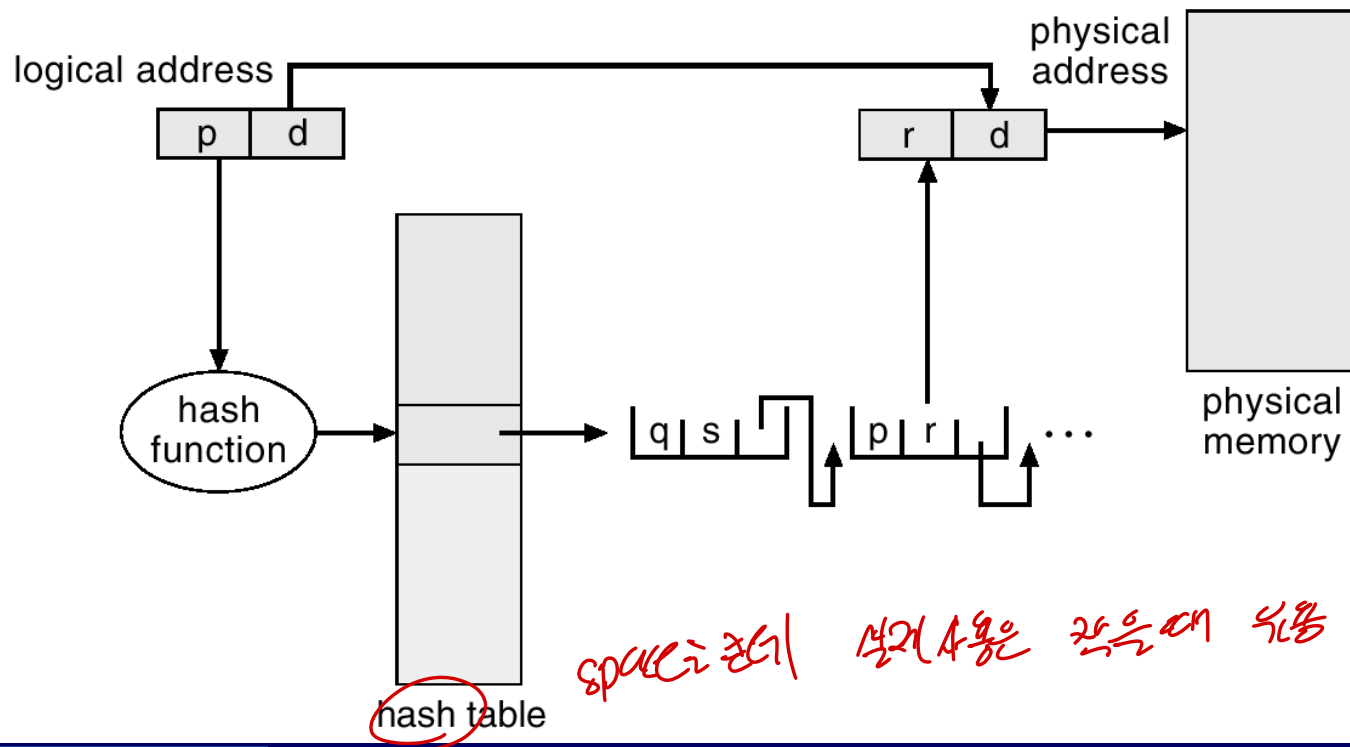
# Hierarchical Paging



# Hashed Page Table

## ■ Use hashing to for page table

- Common approach for large address spaces ( $> 32\text{bits}$ )
- Each location of hash table consists of linked list of page table entries



# Agenda

---

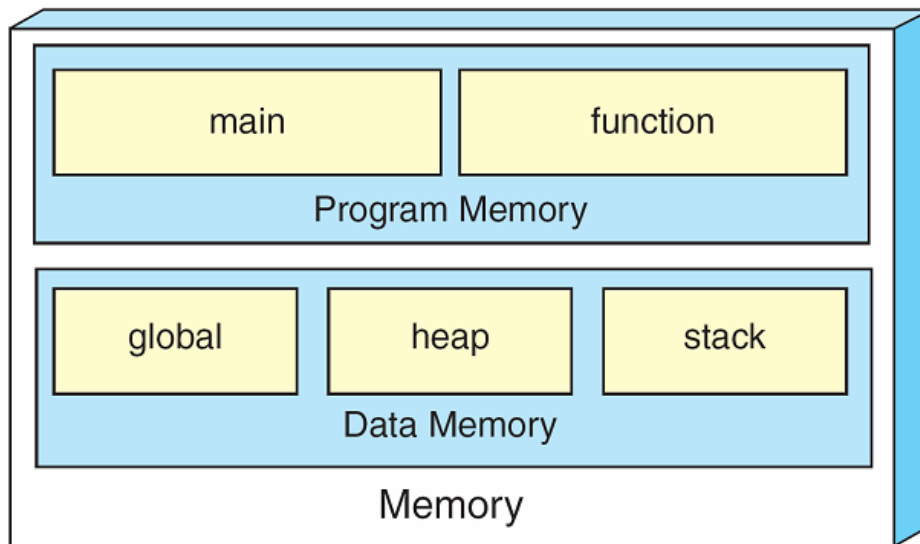


- Background
- Swapping
- Contiguous memory allocation
- Paging
- Structure of page table
- Segmentation
- Example: Intel Pentium

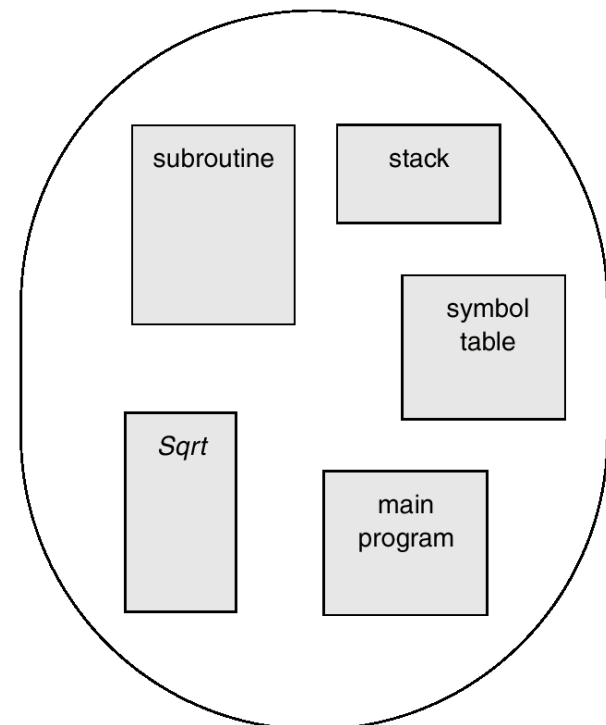
# Segmentation

- Memory from user's view: collection of variable-size segments, with no necessary ordering

Ex) stack, math library, main program, ...



Conceptual view of memory



logical address space

# Segmentation

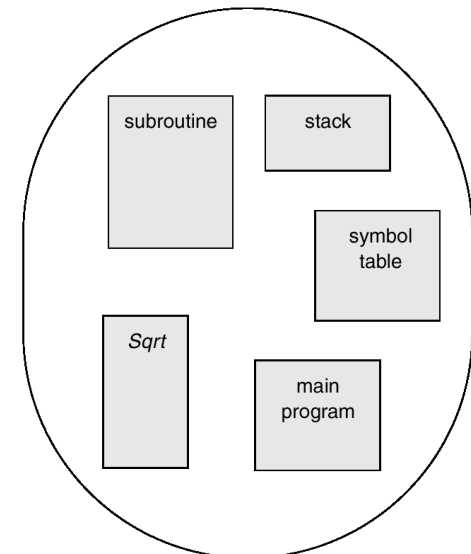
- **Segmentation**: memory-management scheme that supports user view of memory

- Logical address space is a collection of segments
- Each segment has a **name (number)** and **length**

**Logical address** = <segment-number, offset>

Ex) C compiler might create segments such as ...

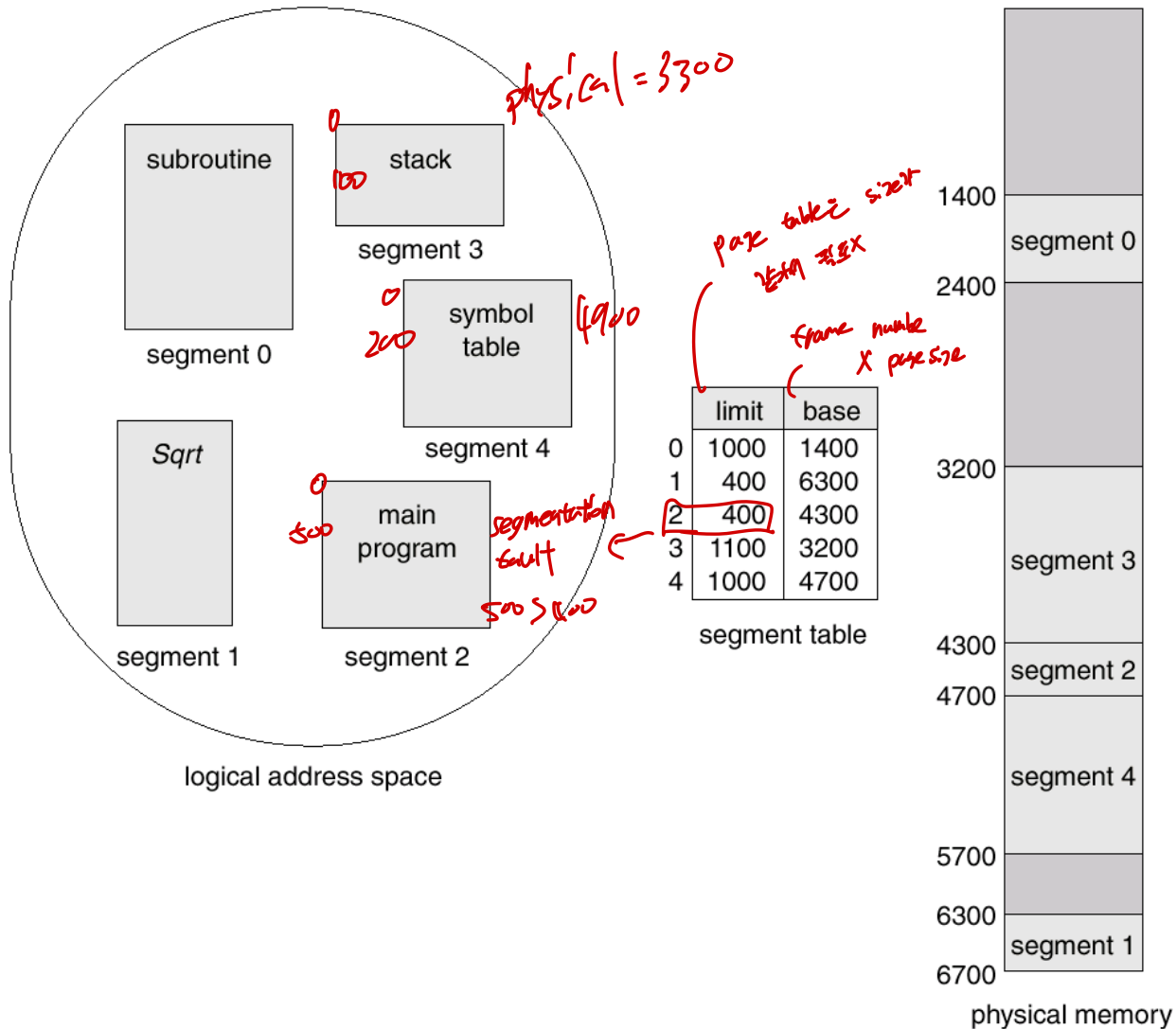
- Code
- Global variables, heap, stacks
- Standard C library



logical address space

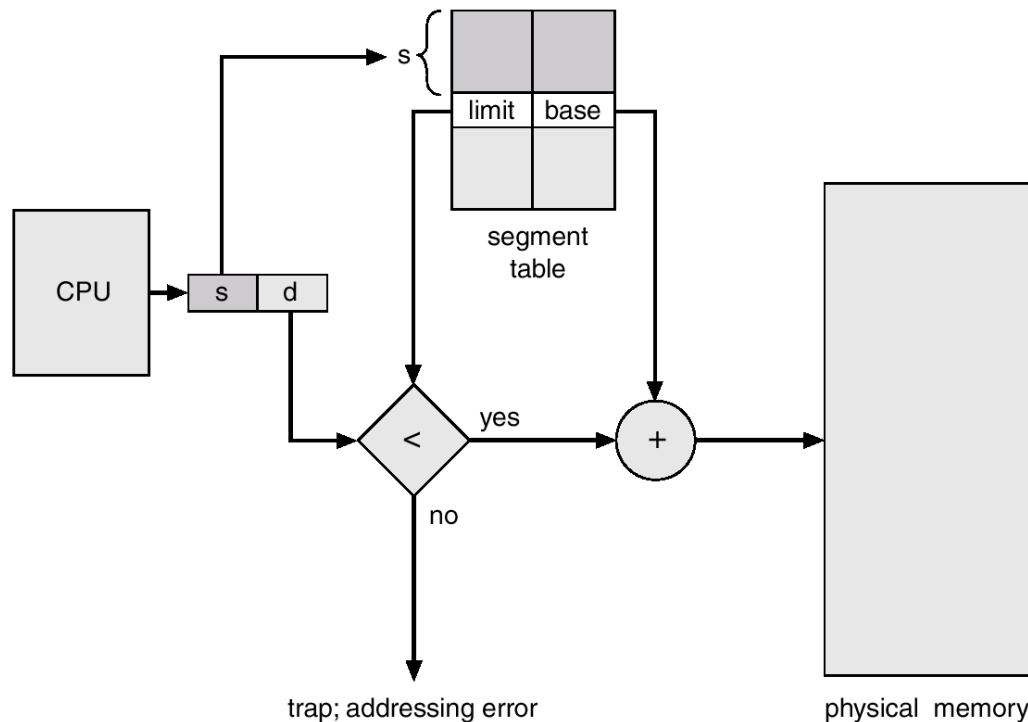


# Example of Segmentation



# Hardware for Segmentation

- 2D address should be mapped 1D sequence of bytes
- Segment table
  - Consists of *segment base* and *segment limit*

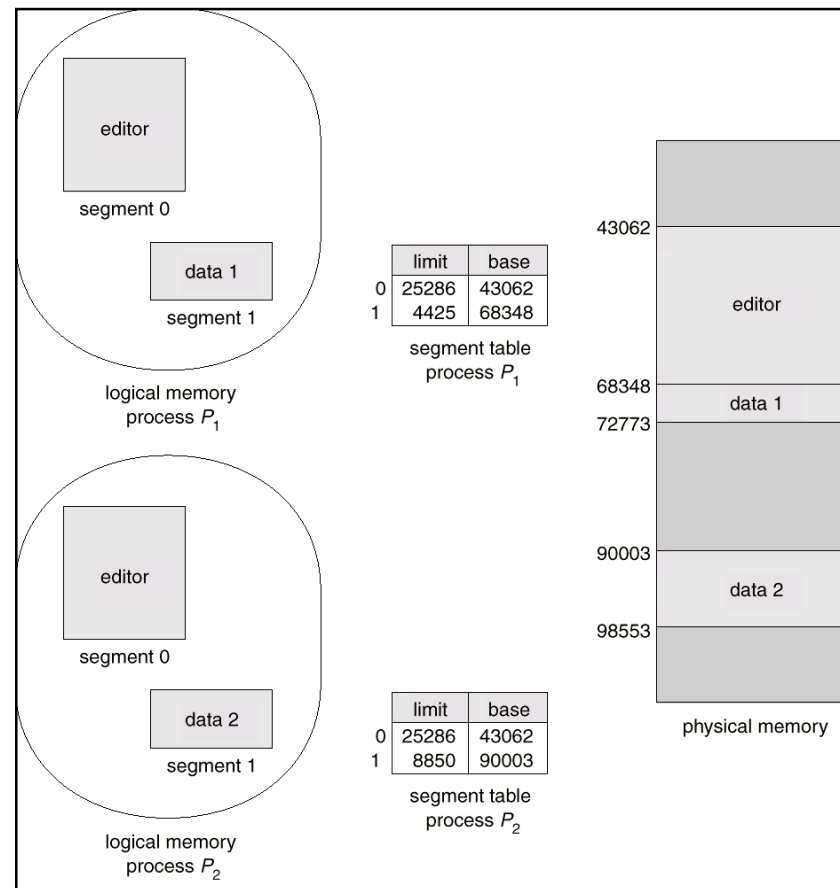


# Segmentation Architecture

- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program;  
segment number  $s$  is legal if  $s < \text{STLR}$

# Segmentation

- Protection and sharing is also possible for segmentation



# Agenda

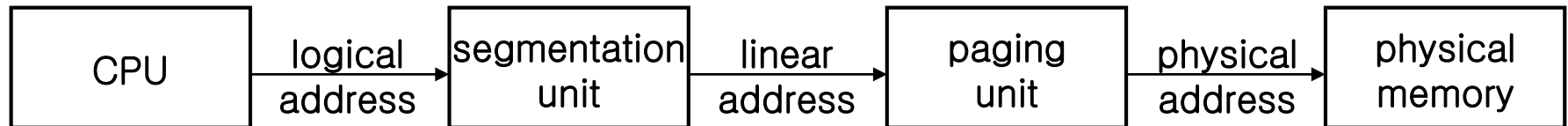
---



- Background
- Swapping
- Contiguous memory allocation
- Paging
- Structure of page table
- Segmentation
- Example: Intel Pentium

# Example: Intel Pentium

- Some architectures supports both paging and segmentation
  - Intel x86: pure segmentation, segmentation with paging



# Pentium Segmentation

- Maximum size of a segment: 4 GB
- Maximum # of segments: 16 K (=16,384)
  - 1<sup>st</sup> partition for private segments (8 K)
    - kept in local descriptor table (LDT) *byte address, limit value*
  - 2<sup>nd</sup> partition 8 K for shared segments (8 K)
    - kept in global descriptor table (GDT)
- Pentium has ...
  - Six segment registers (CS, SS, DS, ES, FS, GS)
  - Six 8-byte **microprogram registers** to hold descriptors from LDT or GDT
    - Cache for LDT/GDT entry

# Pentium Segmentation

## ■ Segment descriptor

- 64 bit information that describes attribute of a segment

<Structure of segment descriptors>

데이터 세그먼트 디스크립터



코드 세그먼트 디스크립터



시스템 세그먼트 디스크립터

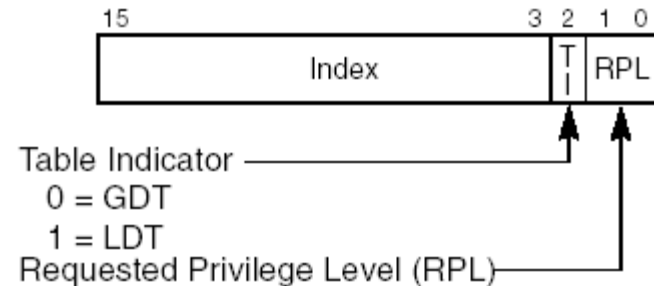




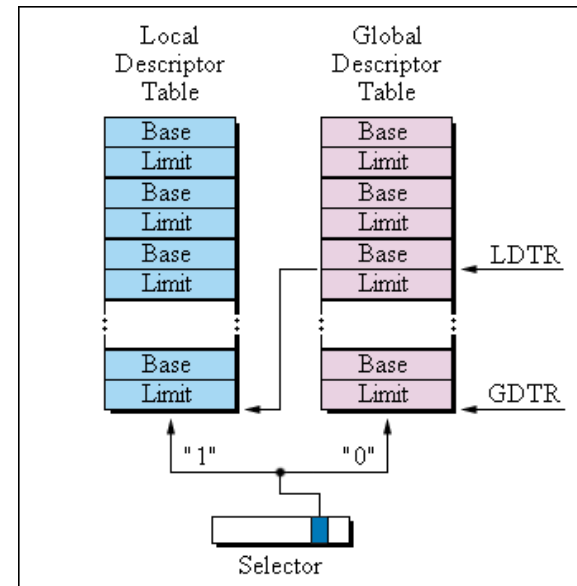
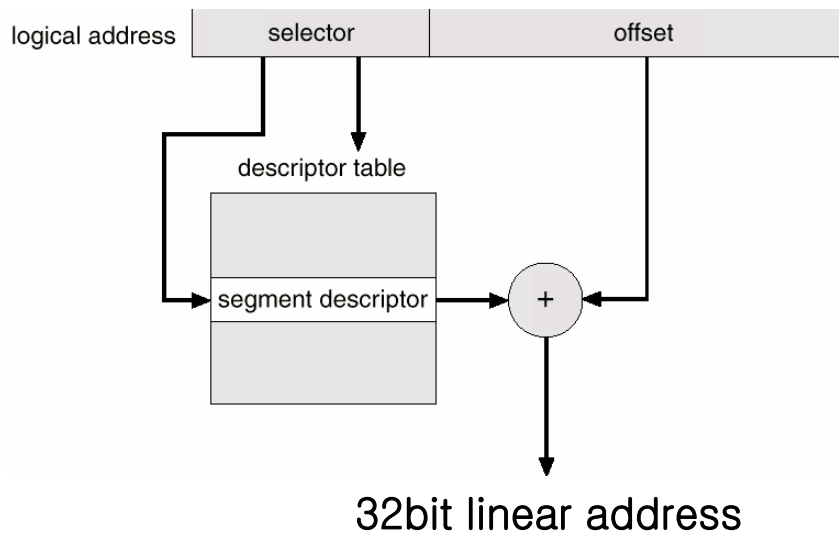
# Pentium Segmentation

## ■ Logical address: pair of (selector, offset)

- Selector: 16 bit number
  - Segment number (13 bits=8 k)
  - GDT or LDT (1 bit)
  - Protection (2 bits)



## ■ Offset: 32-bit number



# Pentium Paging

- Pentium supports a page size of 4 KB or 4 MB
  - Page directory entry includes **page size flag**
  - 4 KB page: two-level paging (page size flag = 0)
    - PSE bit of CR4 register

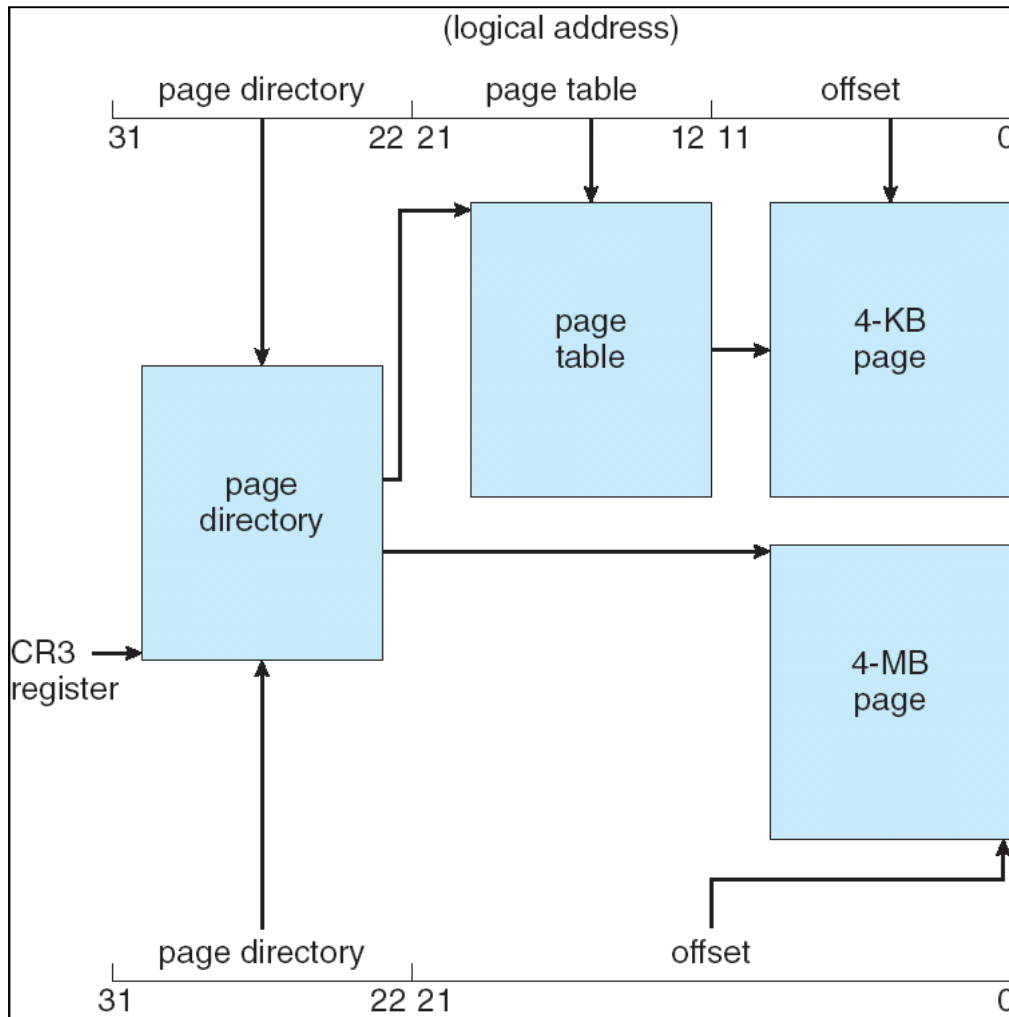
page number		page offset
$p_1$ (10 bits)	$p_2$ (10 bits)	$d$ (12 bits)

- 4 MB page (page size flag = 1)

page number	page offset
$p_1$ (10 bits)	$d$ (22 bits)

$$2^{22} = 4M$$

# Pentium Paging



# Linux on Pentium Systems



- Linux does not rely on Pentium segmentation but uses it minimally
- Linux on Pentium uses six segments
  - Kernel code / data
  - User code / data
  - Task-state segment (TSS)
  - Default LDT segment
    - Shared for all processes, but usually not used
- Segments for user code and user data are shared by all processes
  - All processes share the same logical address space
  - All segment descriptors are stored in GDT
    - If necessary, a process can create LDT and use it

# Linux on Pentium Systems

- Linux uses three-level paging model

- Linear address in Linux

global directory	middle directory	page table	offset
---------------------	---------------------	---------------	--------

→ works well for both 32 bit and 64-bit architectures

- On Pentium, size of middle directory is zero bit