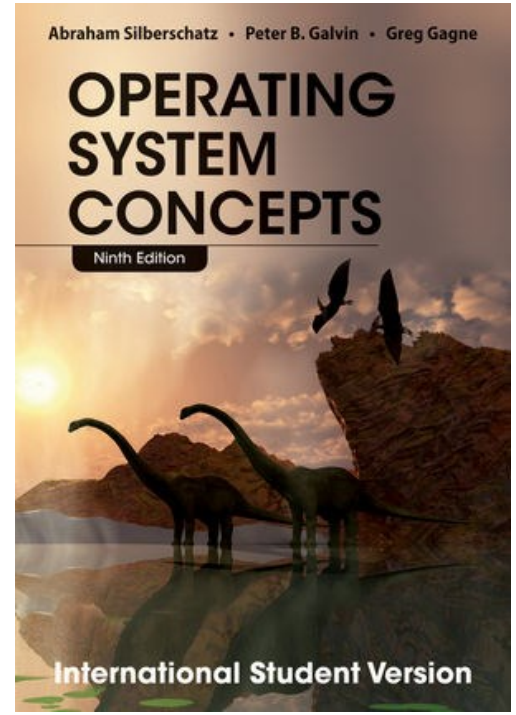
Chapter 9: Memory-Management Strategies

Dept. of Software, Gachon Univ.
Joon Yoo



Chapter 9: Memory Management Strategies

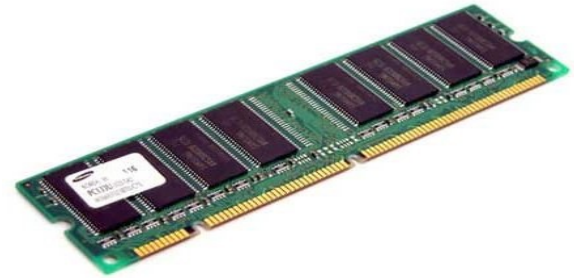
- **Introduction**
- Virtual Memory
- Contiguous Memory Allocation
- Paging

Objectives

- To provide a detailed description of various ways of organizing memory hardware
- To learning the basic concept of virtual memory
- To discuss various memory-management techniques, including paging

Recall: Memory (Ch. 1)

- Address
  - relative position in memory
- Contents
  - the data stored in a memory cell
- The size of each memory cell is **one BYTE**



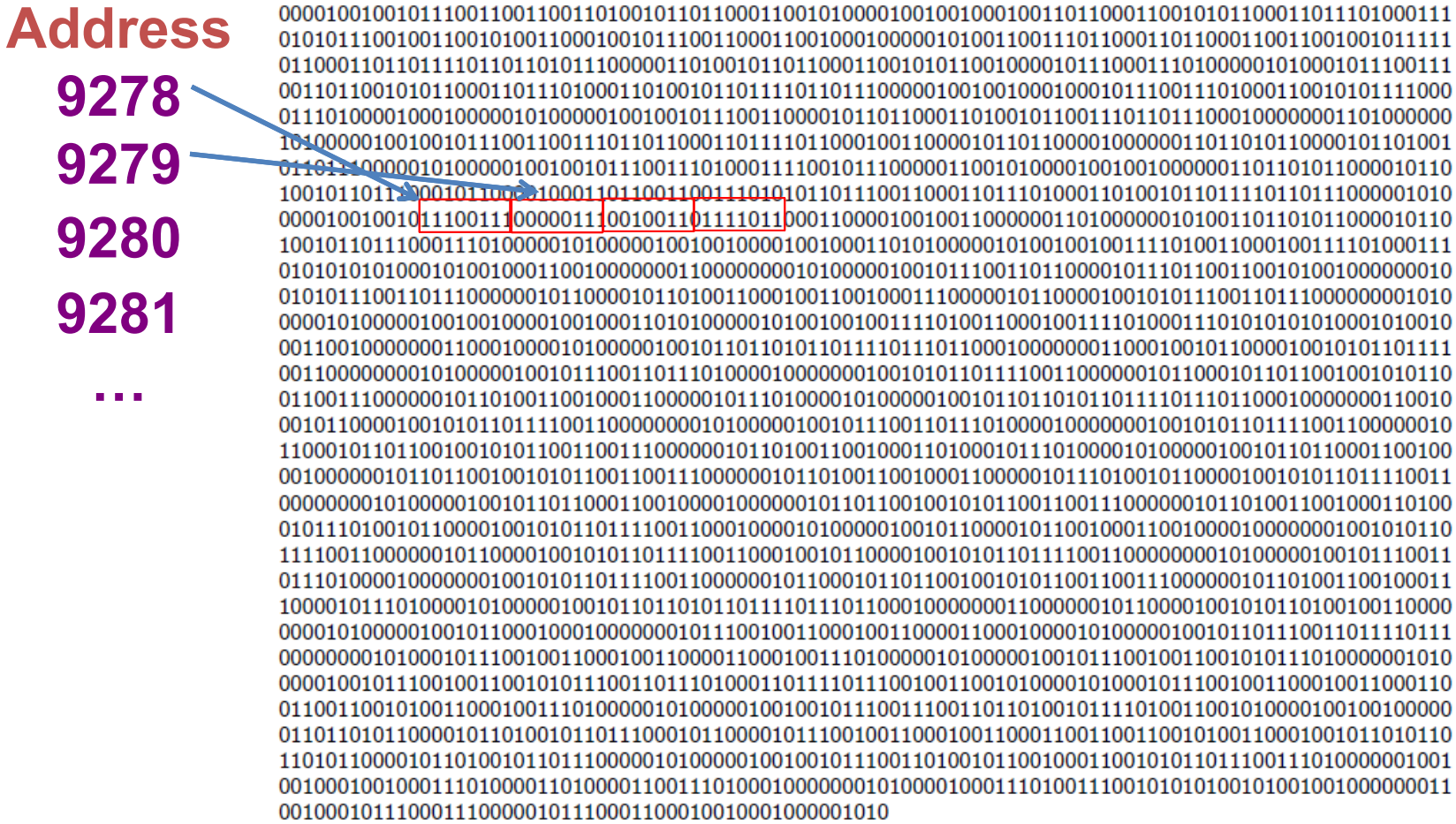
1000 Memory Cells in Main Memory

Memory

Address	Contents
0	–27.2
1	354
2	0.005
3	–26
4	H
.	.
.	.
.	.
998	X
999	75.62

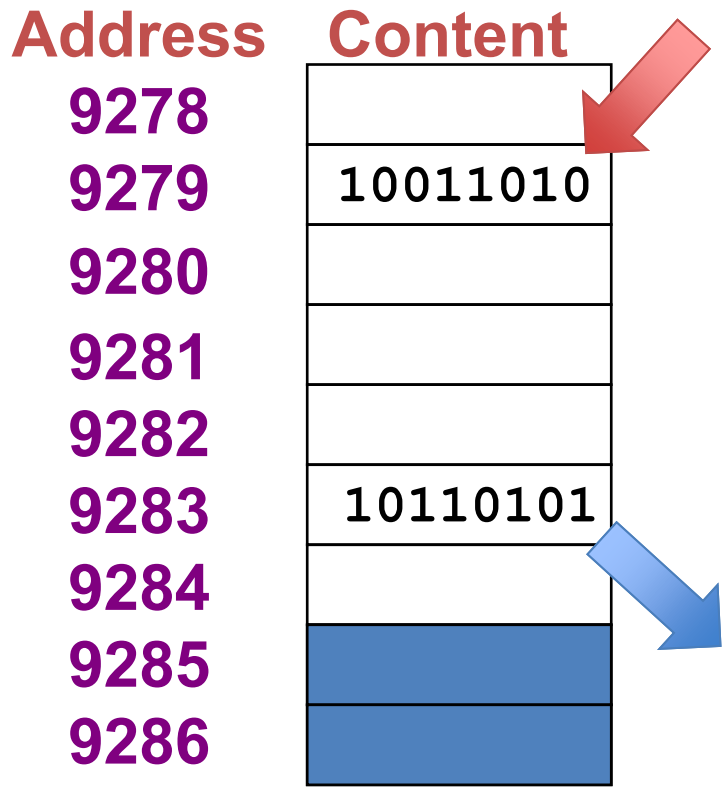
Recall: Memory (Ch. 1)

Basics: Actually, memory looks like …



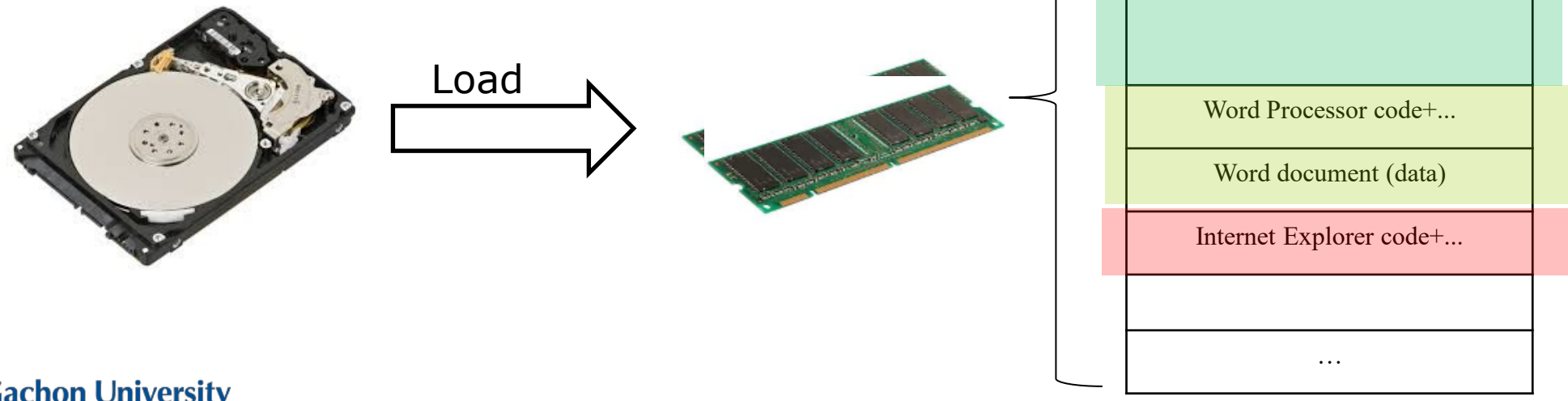
Recall: Memory (Ch. 1)

- **Basics: Storage and Retrieval of Information in Memory**
  - Data storage (Write)
    - Setting the individual bits of a memory cell to 0 or 1, destroying its previous contents
  - Data retrieval (Read)
    - Copying the contents of a particular memory cell to another storage area



Recall: DRAM (main memory)

- DRAM (Dynamic random access memory)
  - **Programs** are usually stored in the **hard disk**
    - When programs are executed (= process), it is loaded on the **main memory** – faster than disk
  - Similarly, when **file** is **open**, it is loaded on the main memory



DRAM: main memory

- Why load program/data from disk to memory?
  - DRAM access time ≈ 50ns
  - Disk access time ≈ 5,000,000ns
  - If we use disks to run programs it will be **100,000x** slower
    - SSD access time ≈ 100,000ns, so 2000x slower than DRAM

SRAM: Cache Memory

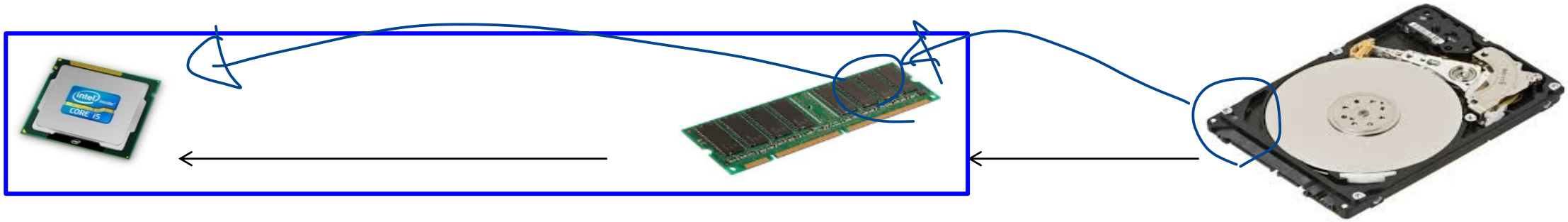
- But DRAM is still much slower than CPU
  - It takes about **100~400 CPU cycles** to access DRAM
  - Use a small but very fast memory = **Cache Memory**!!
    - SRAM access time: only few ns

  - Computer hardware system takes care of cache
  - OS view: A single memory. (does not know cache)

Introduction

- Memory is ***central*** to the operation of a computer system

- A typical **CPU instruction-execution cycle**
  - 1. The CPU **fetches** an **instruction** from **memory**
  - 2. The instruction is then decoded and may cause operands (data) to be fetched from memory
  - 3. the result may be **stored back in memory**



Summary IMPORTANT

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are the only storage that CPU can access directly
  - CPU cannot directly access disk, network, printers, ... – ***must*** be copied from/to memory first
- Speed?
  - Register access in one CPU clock (or less)
  - Main memory can take many cycles, causing a **stall**
    - Note: Faster **cache memory** sits between main memory and CPU registers. **OS view of cache is same as main memory!!**
    - Thus we will not be dealing much with cache here (you will be learning more on cache in Computer Architecture)

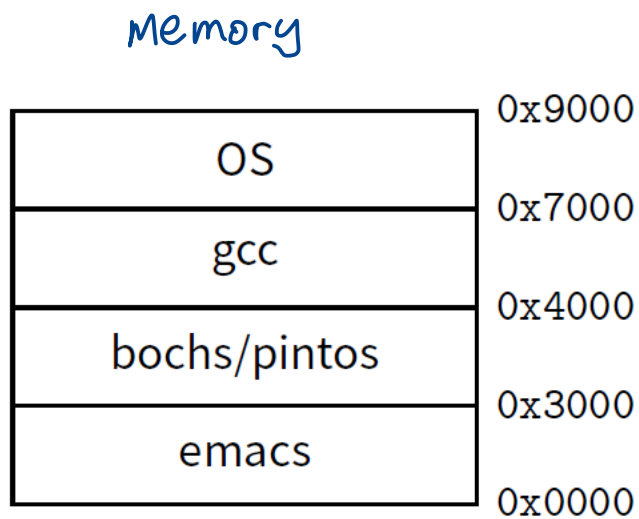
Chapter 9: Memory Management Strategies

- Introduction
- **Virtual Memory**
- Contiguous Memory Allocation
- Paging

Sharing/Protecting the Memory

- Kernel, System/user Processes, Data share the main memory
  - When does gcc have to know it will run at 0x4000? → **Binding** 어디로 가야 하는가? 메모리의 어디로?
  - If pintos has an error and writes to address 0x7100? → **Protection** → 허용하지 않도록 한다. Protect!
  - What happens if pintos needs to expand? → **Memory Allocation/Paging**
  - If emacs needs more memory than is on the machine? → **Demand Paging (Ch. 10)**

물리적 공간이 더 작아도 실행시킬수 있음



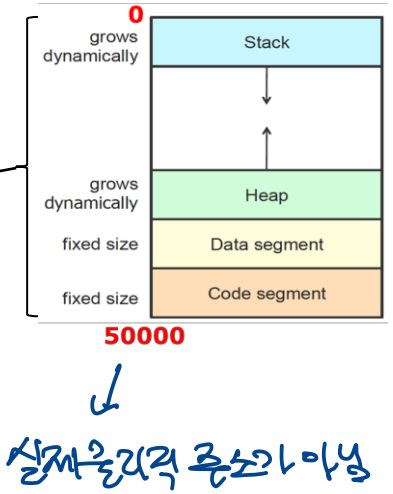
"가상 메모리"를 사용한다!!!

- Answer: Each process uses a separate **Logical (or Virtual) address** instead of actual physical memory address!

IMPORTANT

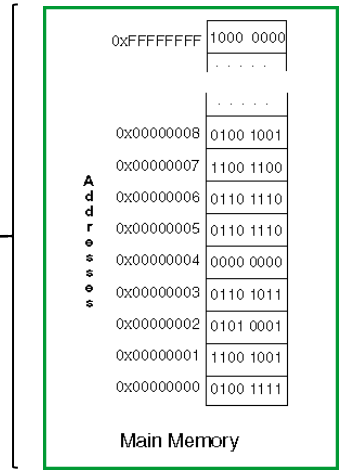
Virtual vs. Physical Address Space (Ch. 9.1.3)

- **Logical address** (= **Virtual address**)
  - **Each user process** has individual virtual address (An address generated by **CPU**)
  - Always starts from **address 0**
  - **Logical Address space**
    - **Set** of all logical addresses generated by a process



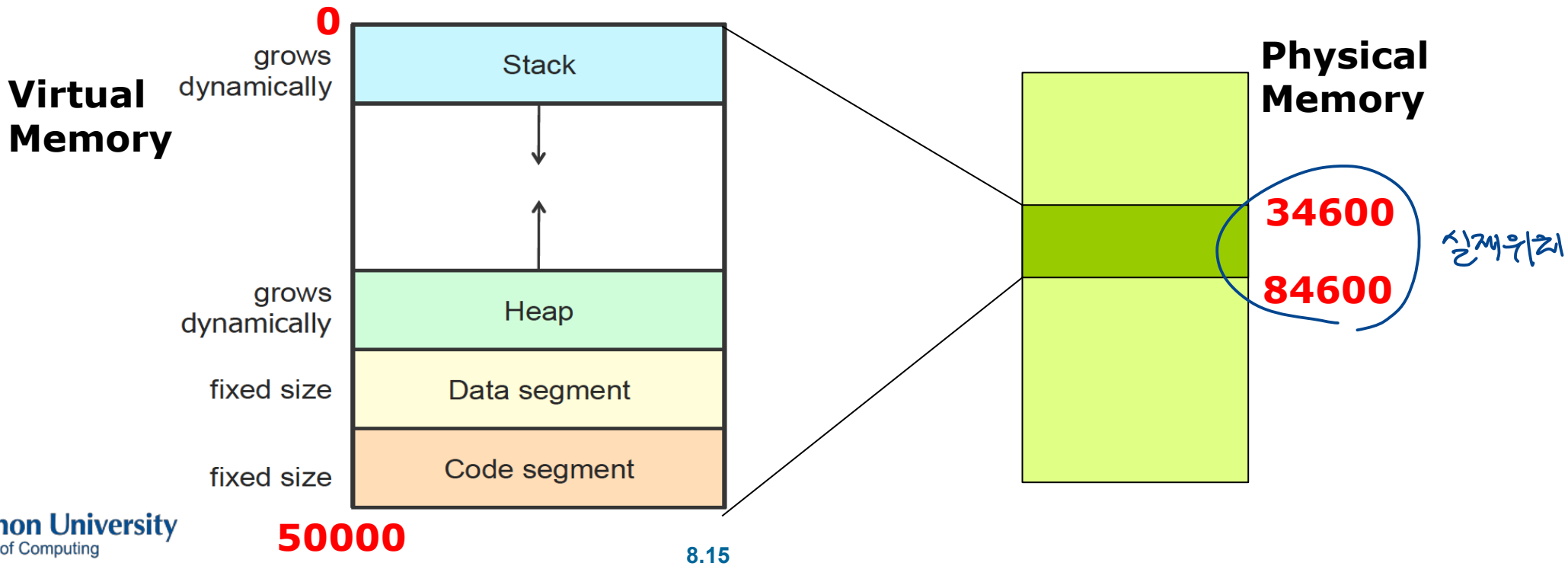
실제물리적 주소가 아님

- **Physical address** 진짜 주소값. 물리적인 위치를 의미함.
  - Address seen by the **physical memory unit**
  - **Physical Address space**
    - Set of all physical addresses



Address Space

- A **logical (virtual) address** is a memory address that a **process** uses to access its own memory
  - Logical address ≠ actual physical RAM address
  - When a process accesses a virtual address, the virtual address must be **translated** into a physical address = **Address binding**

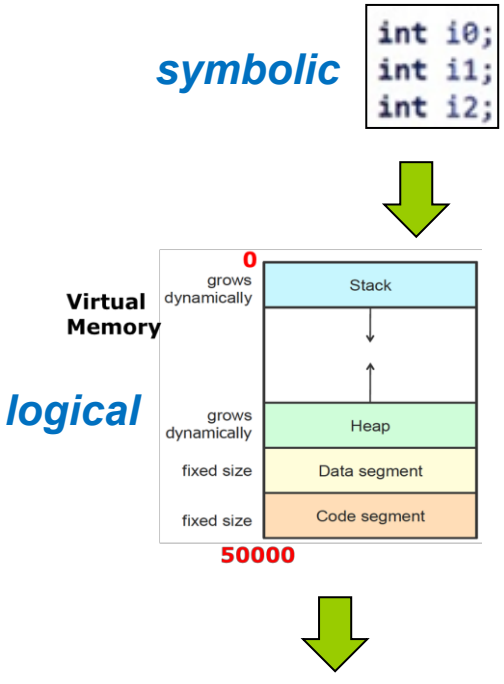


Address Binding (Ch. 9.1.2)

- **Address binding: mapping** between ***logical*** and ***physical*** addresses:

가상메모리 — 실제메모리 위치 할당하는 것

- When should we do address binding?
  - **Address binding** can happen at three different stages:
    - Compile time binding
    - Load time binding
    - Run time binding

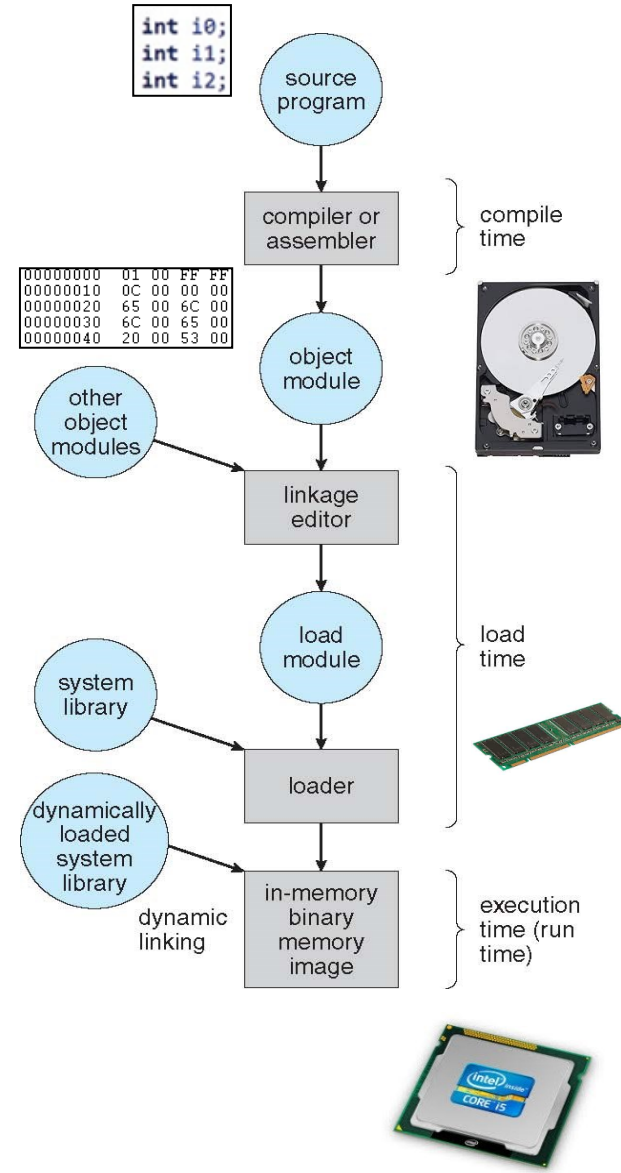


Address Binding

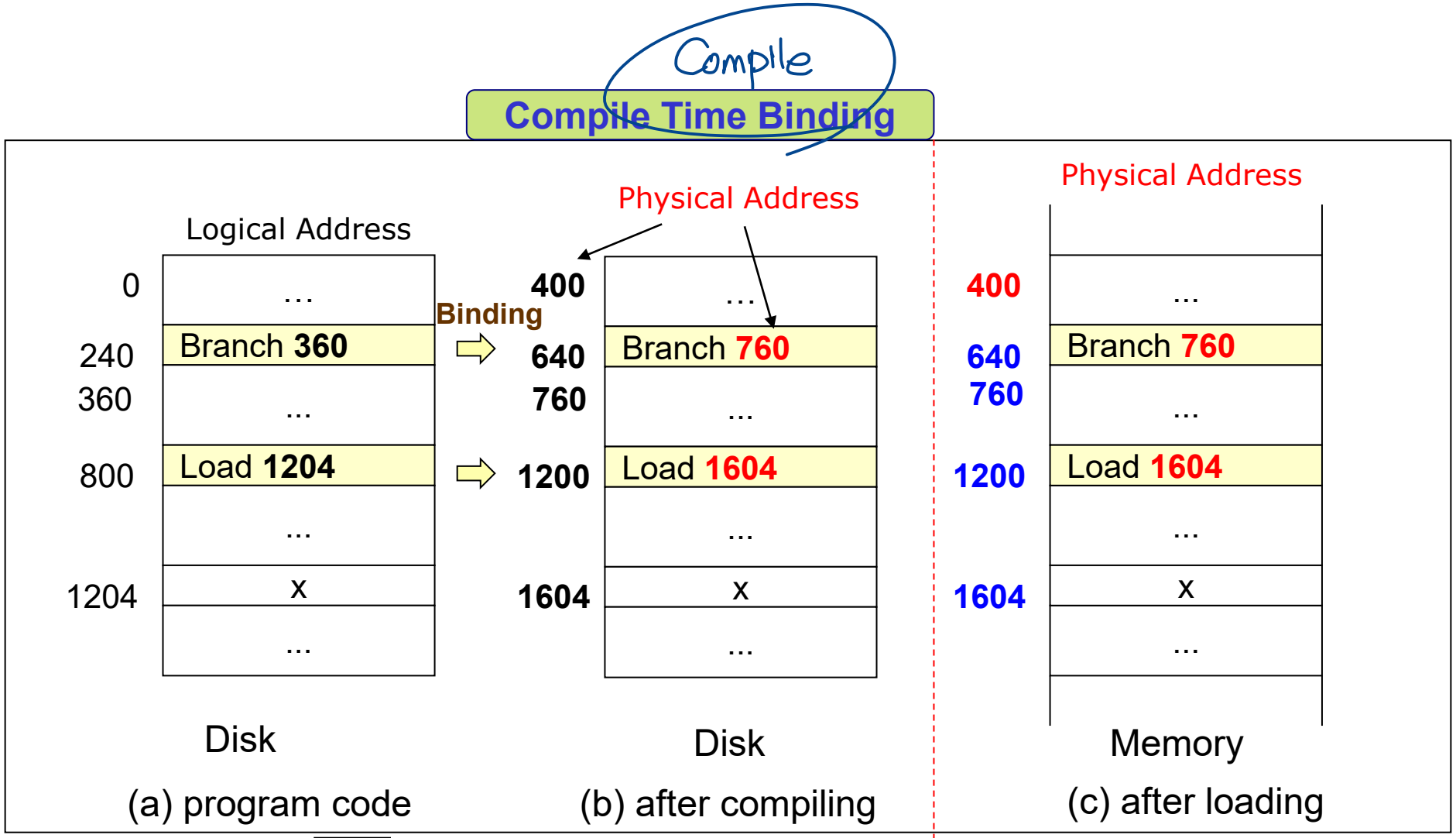
- **Compile time**
  - **Compiler:** If memory location known at compile time, **absolute code** can be generated
  - If starting physical address changes must **recompile** code

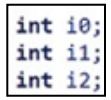
컴파일 할때 physical address에 저장

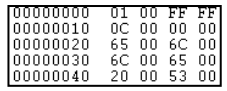
원하는 위치가 바뀌는 경우 recompile 해야 함



Example: Compile Time Binding

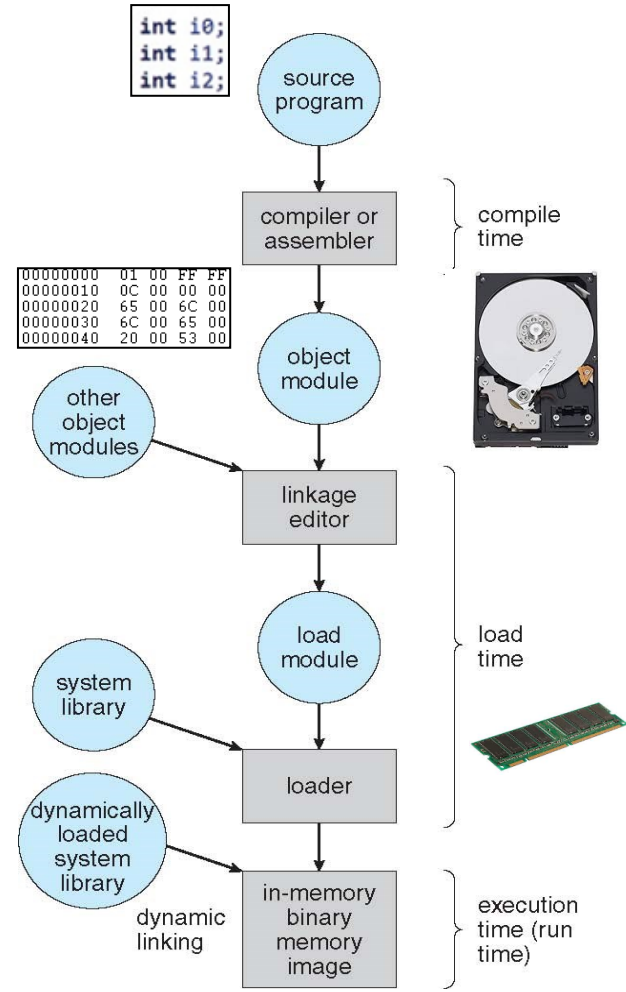




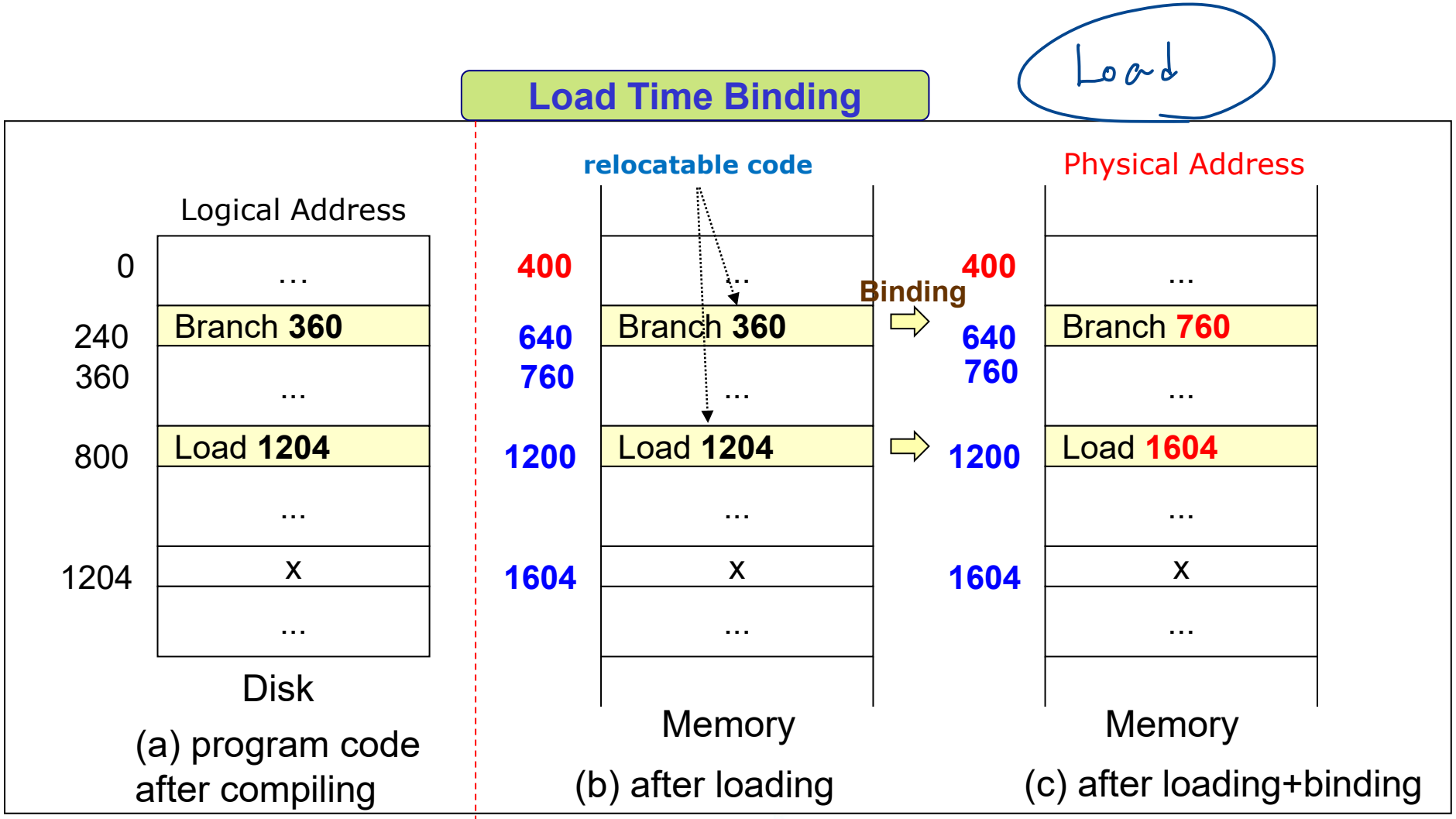


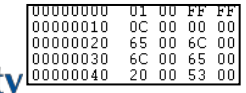
Address Binding

- **Load time**
  - **Loader:** If memory location is not known at compile time, compiler must generate **relocatable code**
    - can change code during load time
  - Address binding is delayed until **load time**
  - If starting address changes, need to **reload** user code



Example: Load Time Binding



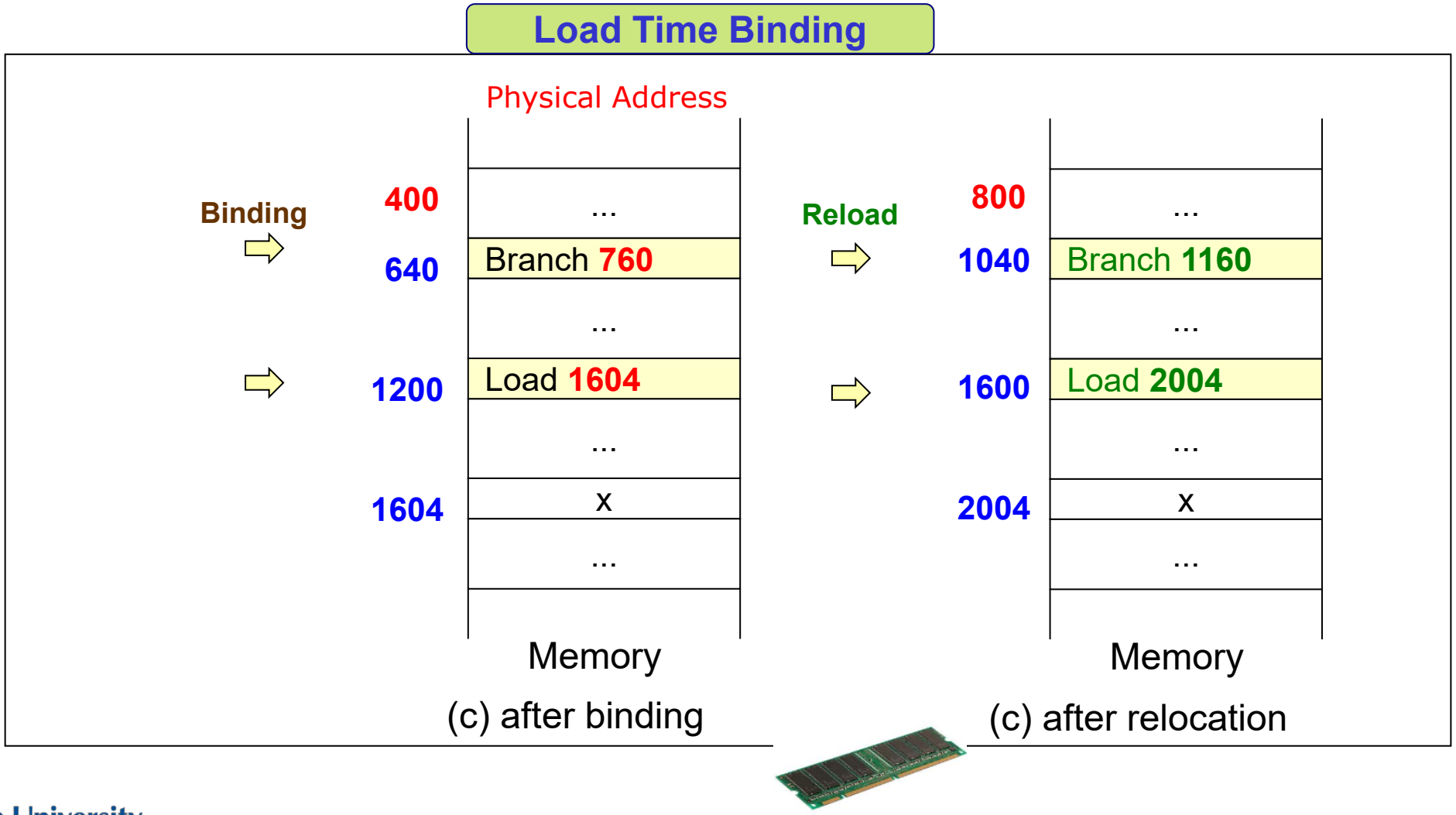


Address Binding

- Q: What if starting address changes to 800?
  - Compile time binding: need to recompile code
  - Load time binding: need to reload

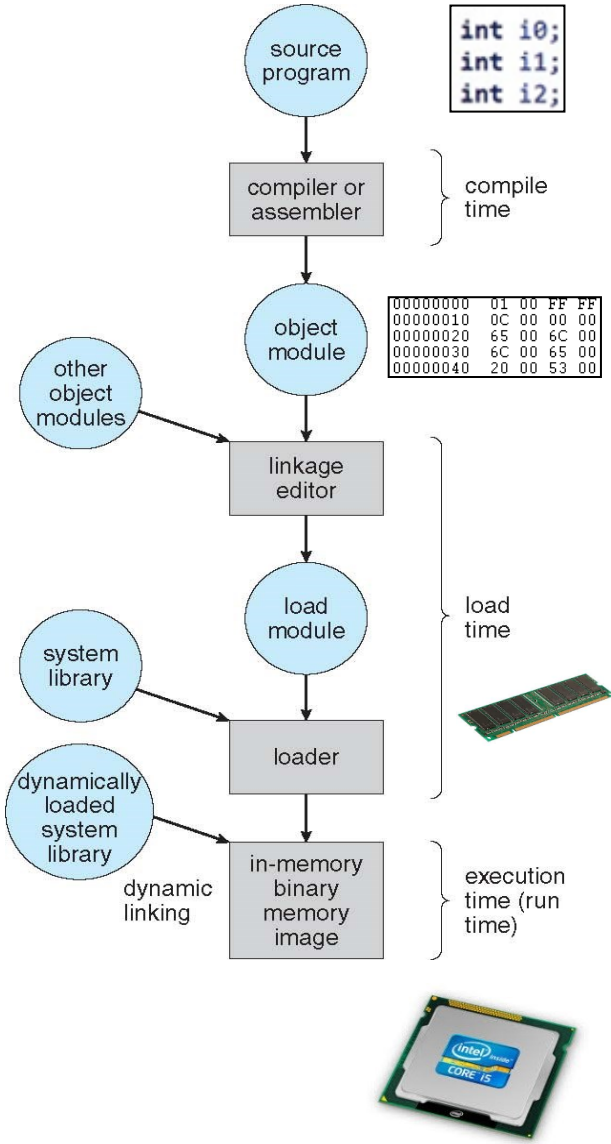
Example: Load Time Binding

Q: What if starting address changes from 400 to 800? Need to reload

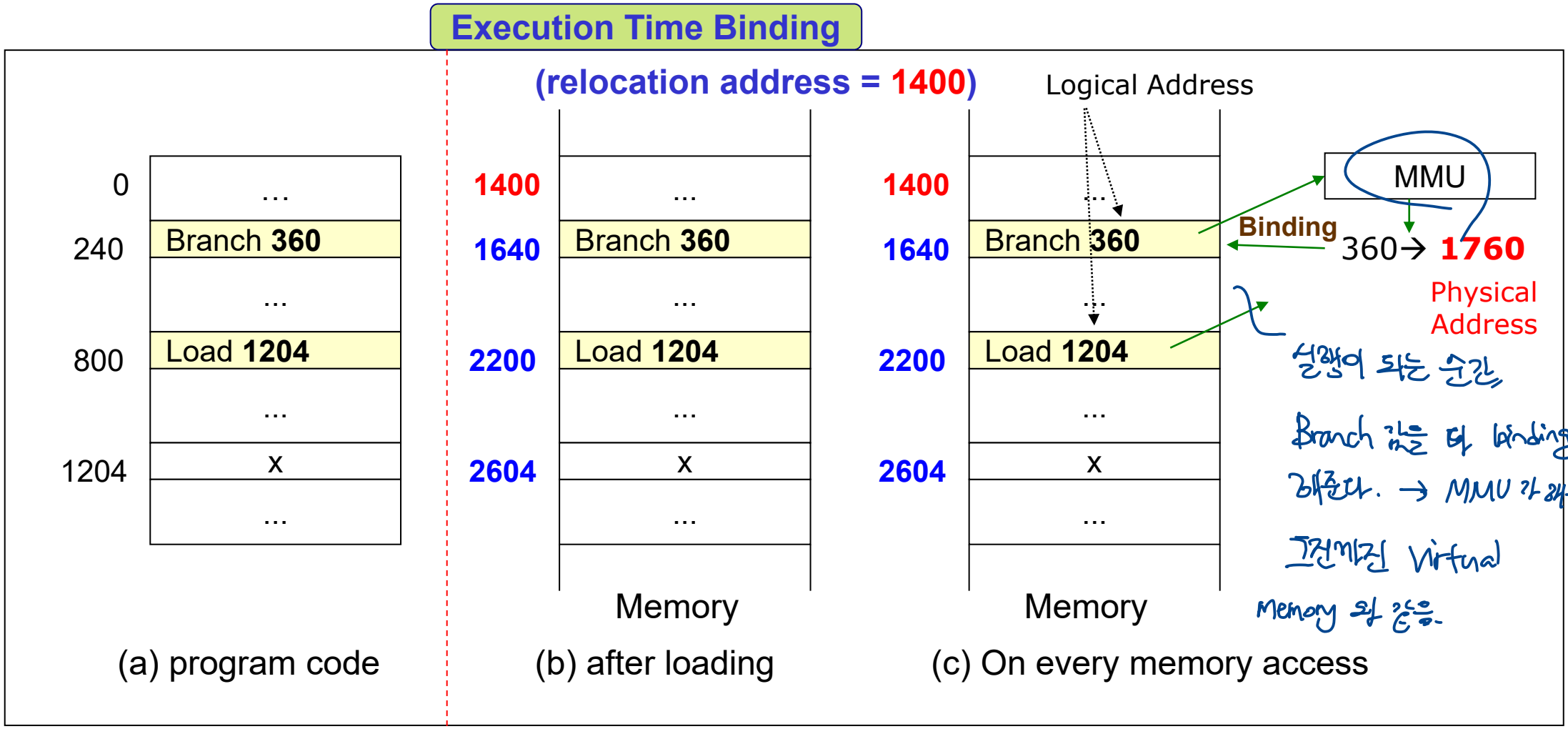


Run Time (Execution Time) Binding

- **Run time (Execution Time)**
  - process memory location can be moved during execution
  - Address binding delayed until **run time**
  - Need hardware support for address maps (e.g., MMU such as base and limit registers)
  - Most OS's uses this method

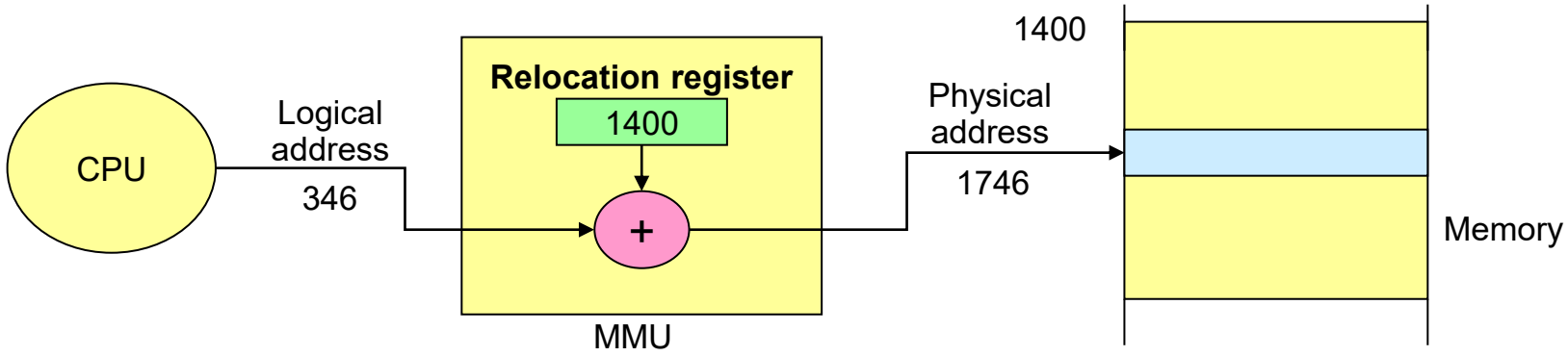


Example: Run Time binding



MMU (Memory Management Unit)

- **MMU** (Memory Management Unit)
  - Hardware supported unit that maps from logical(virtual) address to physical address in Run-time binding

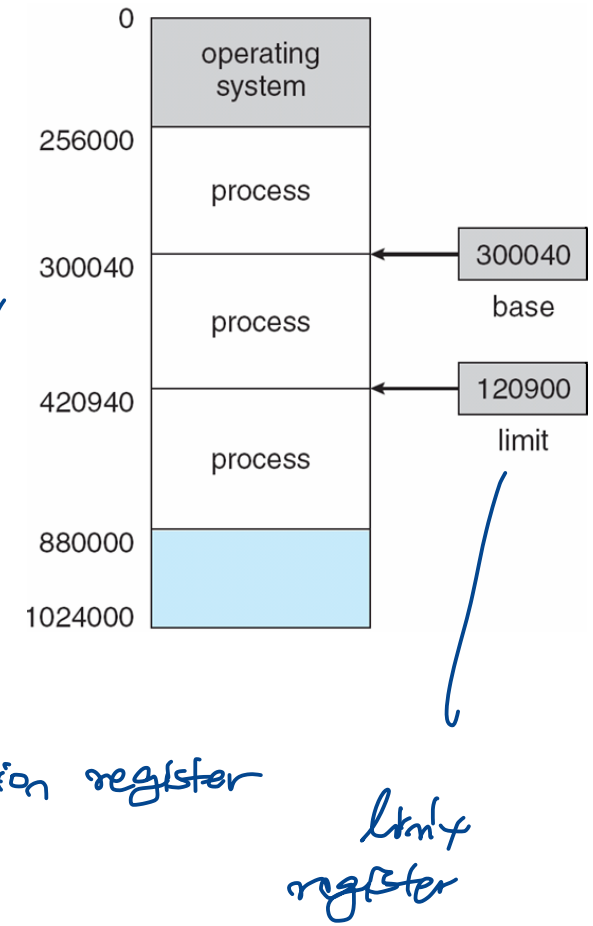


주석 변경!!
Runtime 시에 ..

Q: What if relocation address changes to 1800?

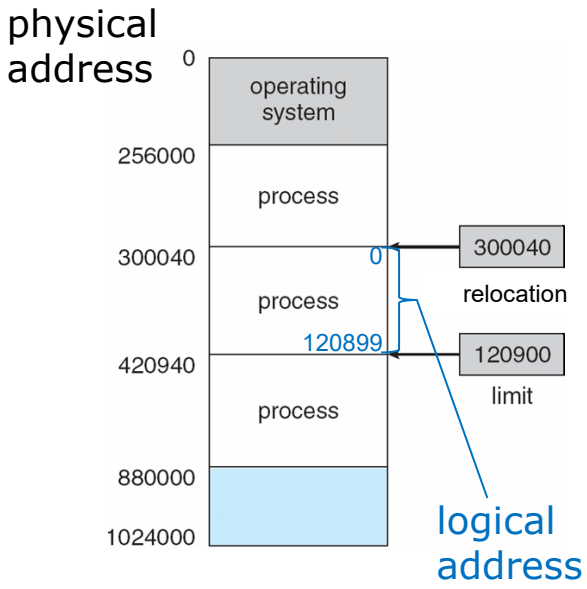
Relocation and Limit Registers (Ch. 9.1.1)

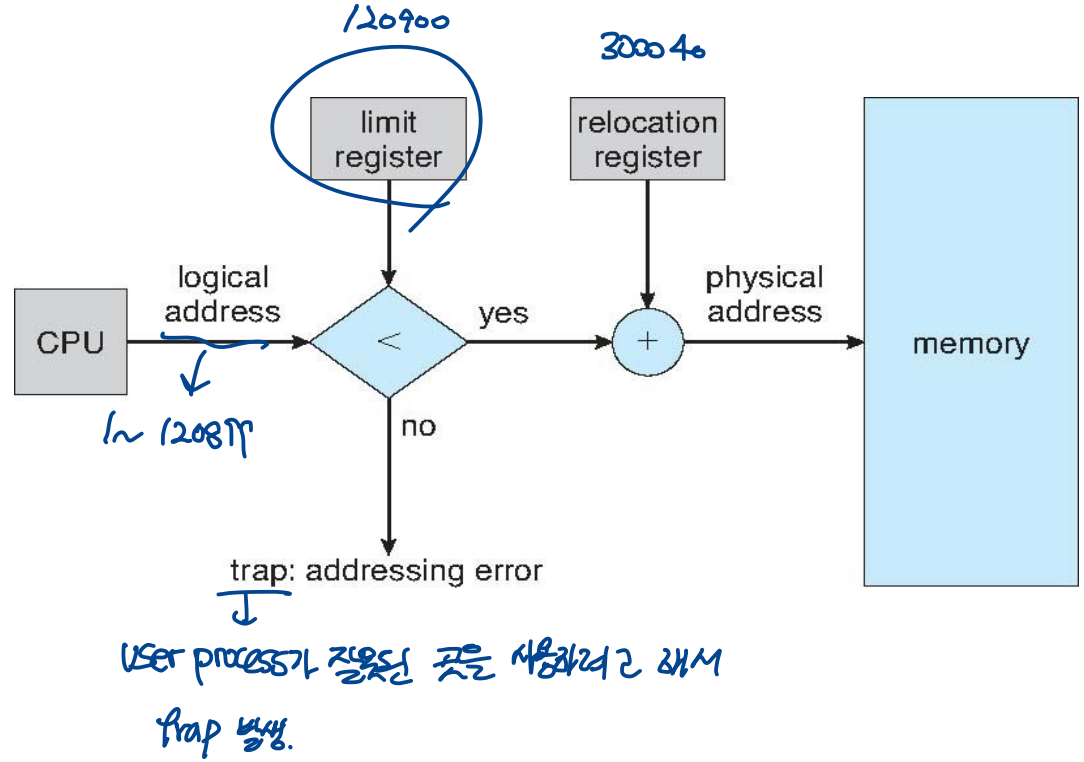
- A pair of **relocation** (= **base**) **register** and **limit register** define the **logical address space** of each process
- Each process has separate memory space
  - User process only sees **logical address**
  - User process cannot see **physical address**
  - User process cannot access other process's memory space = provides **protection**



Memory Protection

- **Memory Protection: Relocation-register scheme**
  - A relocation register and a limit register define a logical address space
  - Address protection with relocation and limit registers





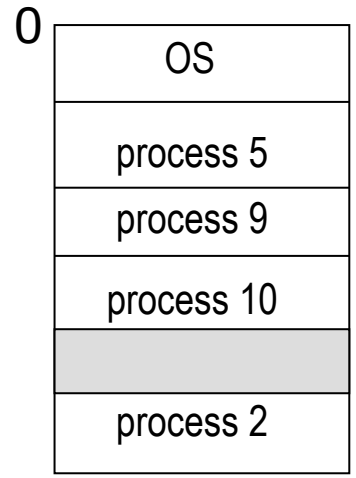
Chapter 9: Memory Management Strategies

- Introduction
- Virtual Memory
- **Contiguous Memory Allocation**
- Paging

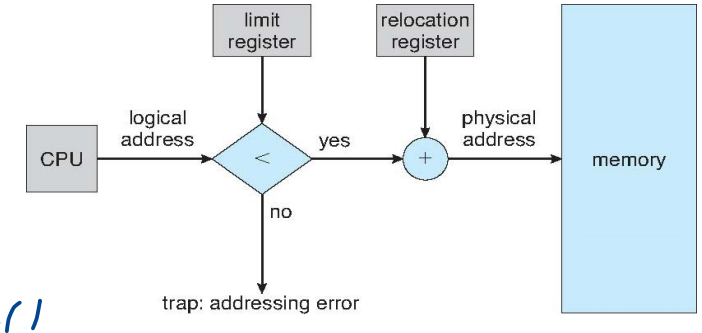
Memory Allocation

- **Memory Allocation**
  - Allocate memory to OS and user processes in an efficient way possible

- Main memory usually into two partitions:
  - Resident **operating system**
    - usually held in low memory with interrupt vector
  - User **processes**
    - held in high memory



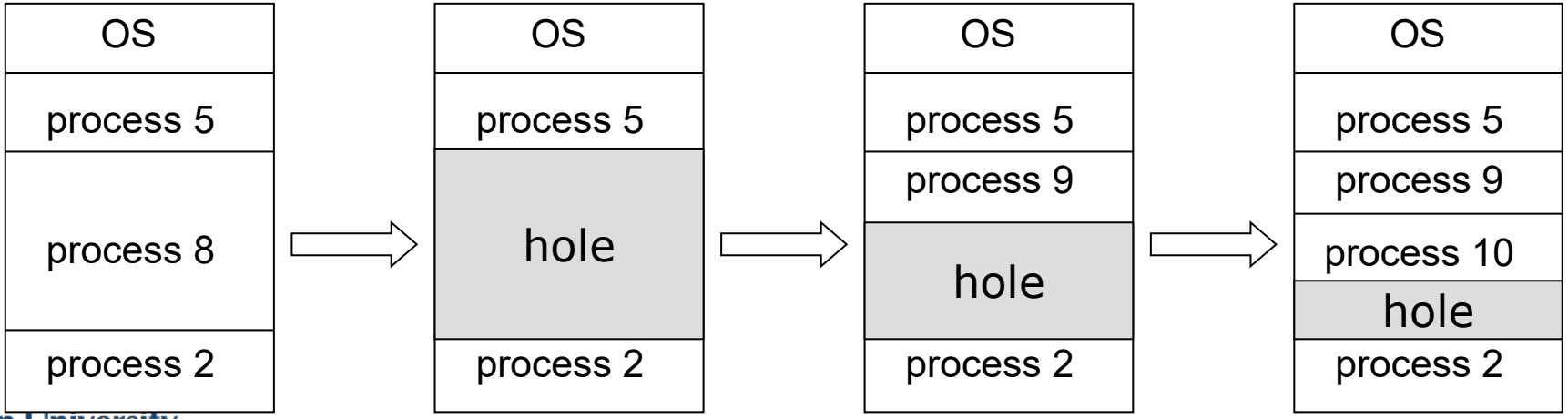
- **Contiguous Memory Allocation**
  - Allocate available memory to processes
  - Each process is contained in a single section of **contiguous** memory



연속적인 메모리 공간을 할당한다!!

Memory Allocation: Contiguous Allocation

- Multiple-partition allocation
  - Degree of multiprogramming limited by number of partitions
  - **Variable-partition** sizes for efficiency (sized to a given process' needs)
  - **Hole** – block of available memory; holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Process exiting frees its partition, adjacent free partitions combined
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (hole)



Contiguous Allocation: Example

- Example

Memory allocation and partition scenario

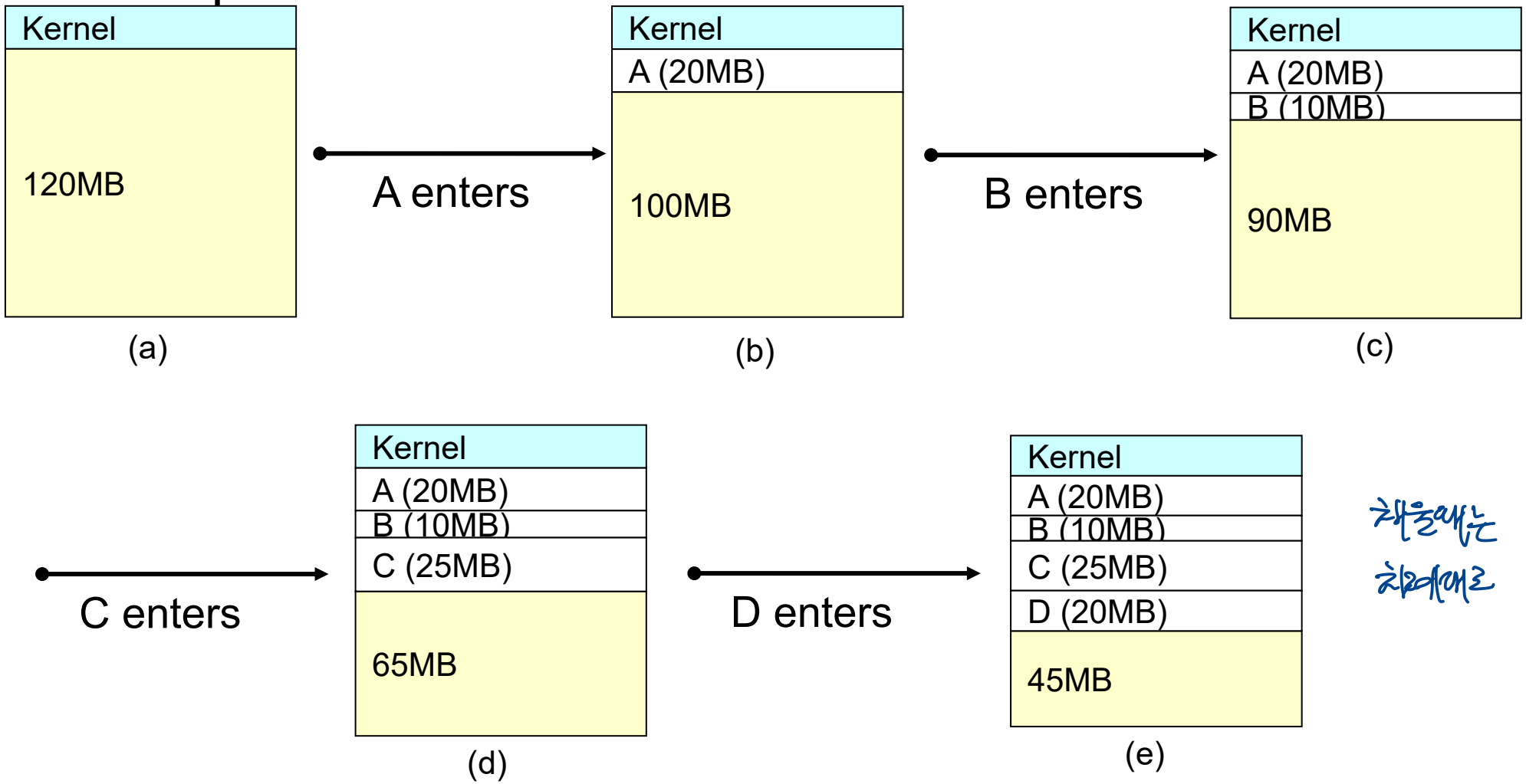
◆Assumption

- Memory space: 120 MB

- (a) : Initial state
- (b) : After loading process A(20MB)
- (c) : After loading process B(10MB)
- (d) : After loading process C(25MB)
- (e) : After loading process D(20MB)
- (f) : After process B releases memory
- (g) : After loading process E(15MB)
- (h) : After process D releases memory

Contiguous Allocation: Example

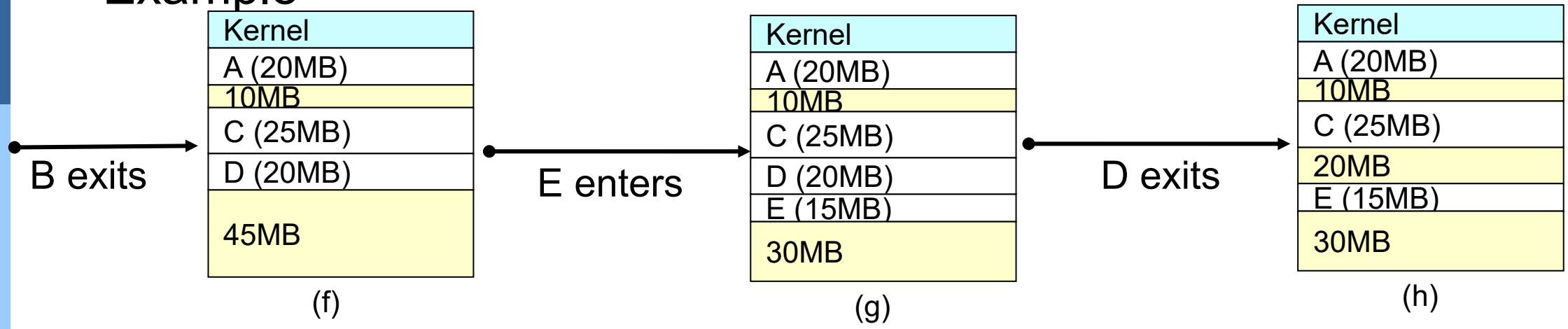
- Example



채울때는
차례대로

Contiguous Allocation: Example

- Example

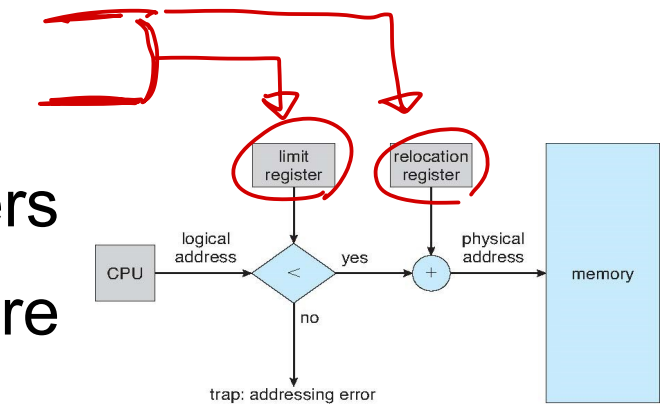


- ***Hole*** **– block of available memory; holes of various size are scattered throughout memory**

빠질때 구멍이 송송...

Contiguous Allocation Pros/Cons

- Pros
  - Cheap in terms of hardware: only two registers
  - Cheap in terms of cycles: do add and compare in parallel

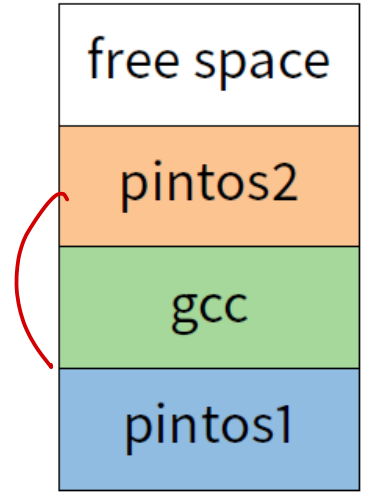


빠르고, 연산 비용이 저렴하다.

logical address

→ 각 process의 위치

- Cons
  - Growing a process is expensive or impossible
  - No way to share code or data (E.g., two processes both running pintos)
  - External Fragmentation (next two slides)



연결성이 끊긴다.

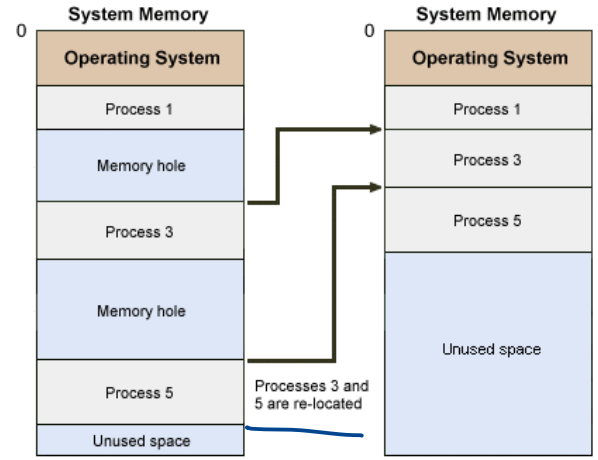
External Fragmentation

F (40MB)?

Kernel
A (20MB)
10MB
C (25MB)
20MB
E (15MB)
30MB

- **External Fragmentation**
  - Total memory space exists to satisfy a request, but it is not contiguous → hole의 합은 충분한데 hole각각은 부족할때

- **Solution1: Compaction** <hole 합치기>
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
    - **NOTE:** We cannot preempt memory space if logical memory is determined in load time or compile time



External Fragmentation

- **Solution2: *noncontiguous allocation***
  - Permit the logical address space of the processes to be Allowing a process to be allocated physical memory wherever such memory is available
  - e.g., **Paging**

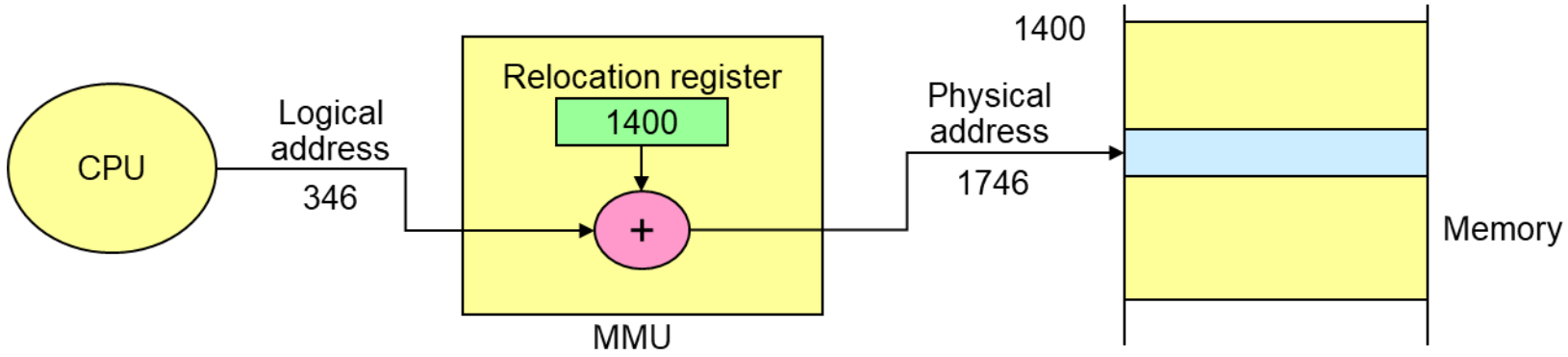
메모리를 쪼개서 할당한다.

Chapter 9: Memory Management Strategies

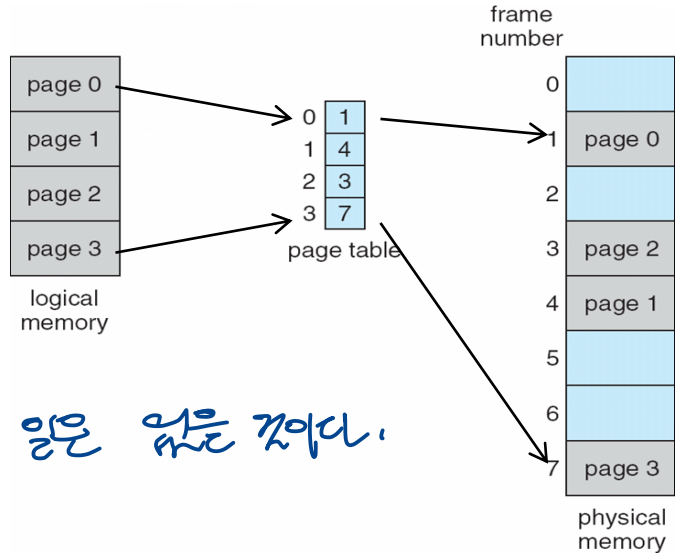
- Introduction
- Virtual Memory
- Contiguous Memory Allocation
- **Paging** 

Contiguous vs. Non-contiguous Memory Allocation

- Contiguous 차곡차곡 . . .



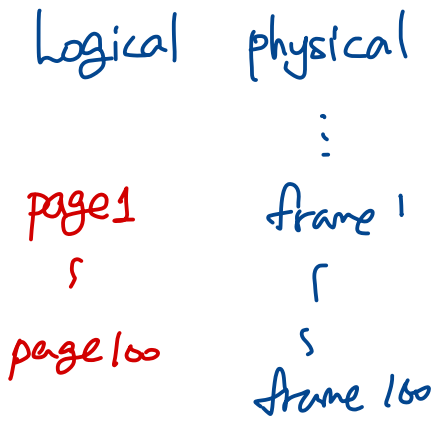
- Non-contiguous
  - **No** external fragmentation
    - no need for compaction



공간이 있는데 넣지 못하는 일은 없는 것이다.

Paging

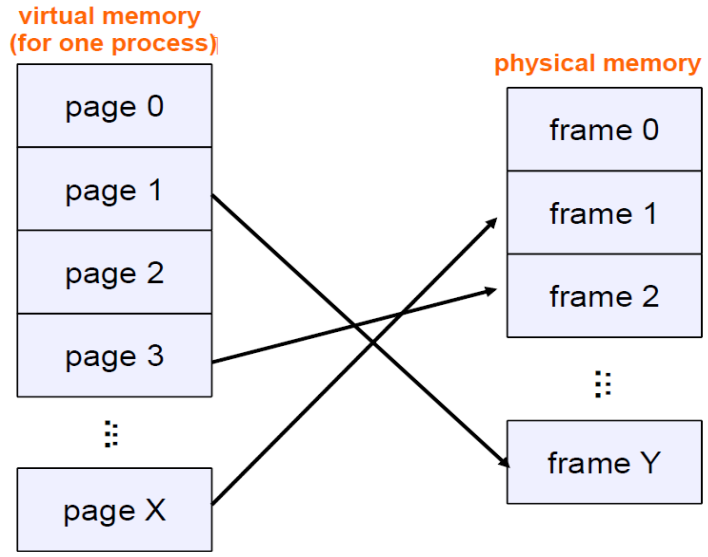
- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames
- To run a program of size ***N*** pages, need to find ***N*** free frames and load program
- Set up a **page table** to translate logical to physical addresses
- Backing store likewise split into pages
- Still have Internal fragmentation



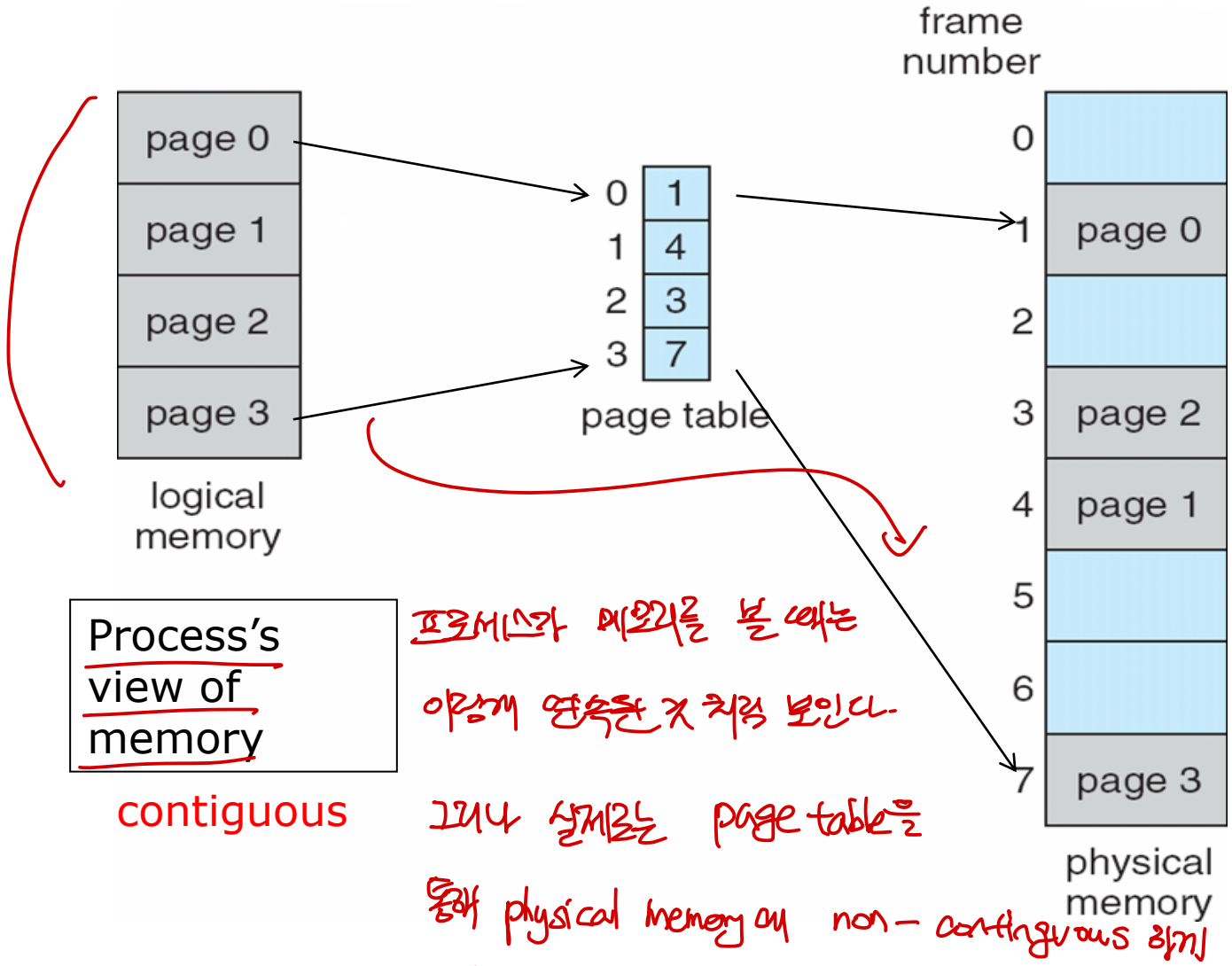
Paging : Basic Method

- **Paging**
  - **physical memory** → *fixed-sized* physical **frames**
  - **logical memory** → *fixed-sized* virtual **pages**
  - size of one frame = size of one page (e.g., 4kB) 대략 4kB정도
  - Set up a per-process **page table** to translate logical to physical addresses

프로세스 마다 page table을 가져야 한다.



Paging Example



Process's view of memory

contiguous

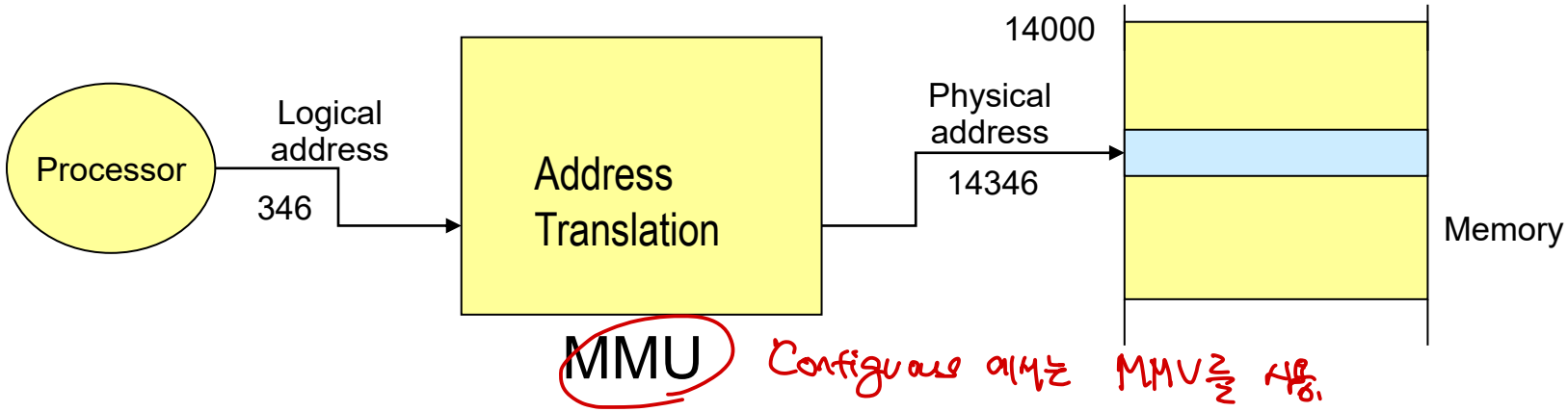
프로세스가 메모리를 볼 때는 아직까지 연속된 것처럼 보인다.

그러나 실제로는 page table을 통해 physical memory에 non-contiguous 하게 저장된다.

Noncontiguous

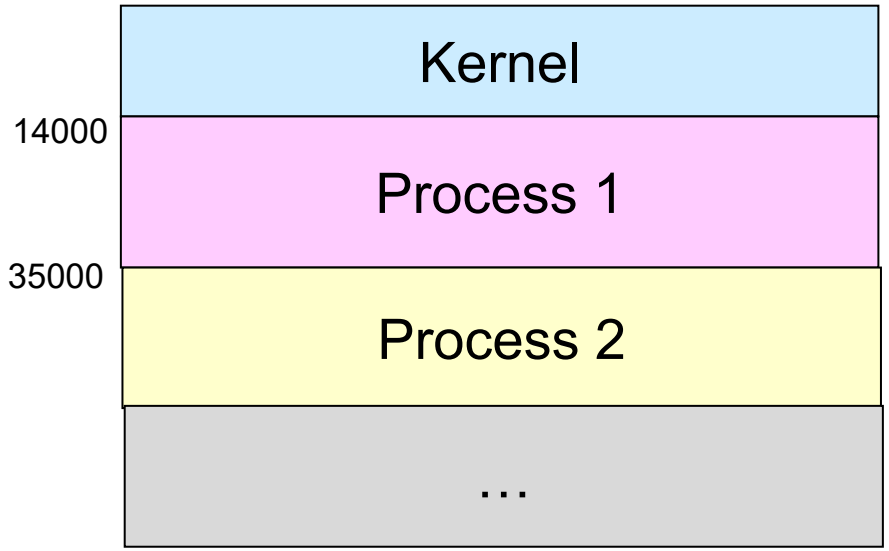
Actual physical memory

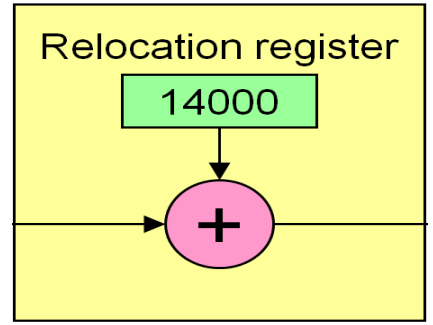
Contiguous vs. Paging



Contiguous 에서는 MMU를 사용.

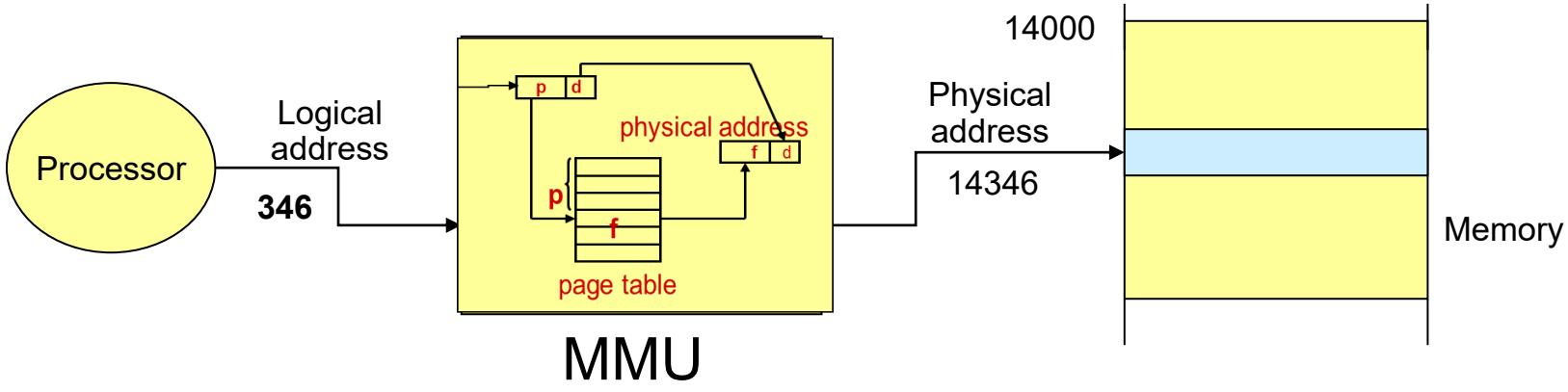
non-인경우 page table을 이용해 구현함.

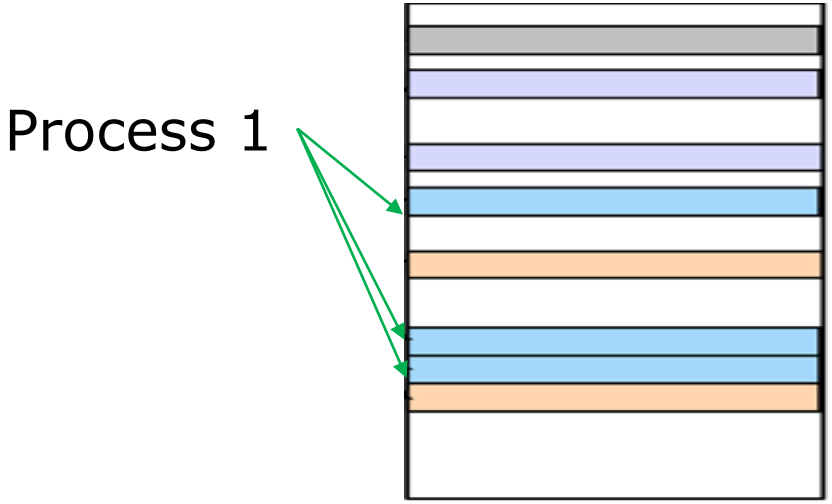


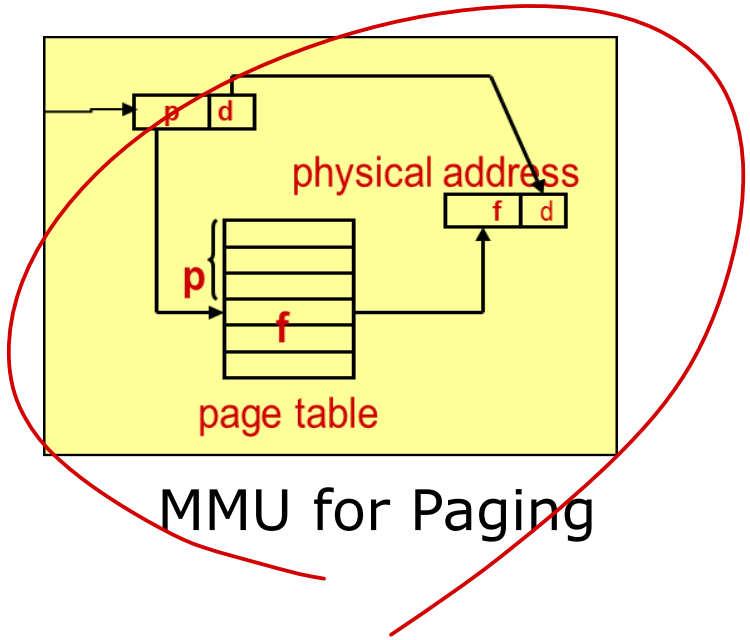


MMU for Contiguous Allocation

Contiguous vs. Paging





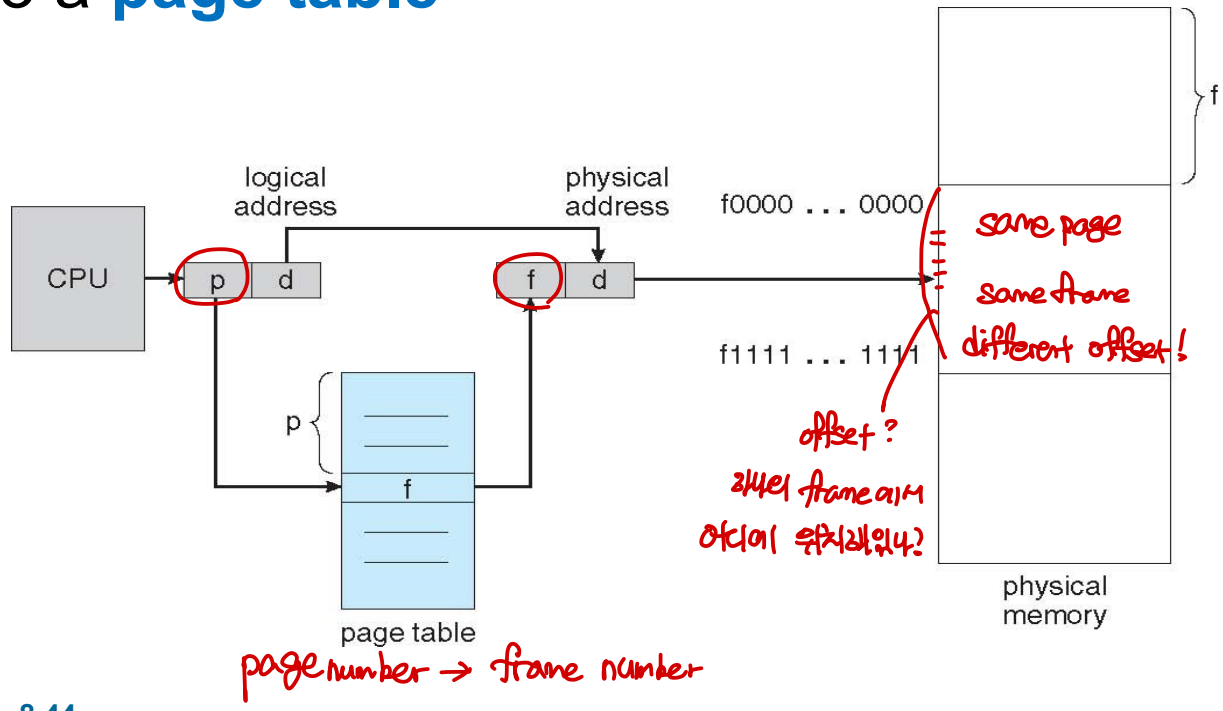


Page Number and Offset

- When a process is executed, its pages are loaded into any available memory frames
- Every address generated by CPU is divided into two parts: **page number (p)** and **page offset (d)**
  - **page number (p)**: index into a **page table**

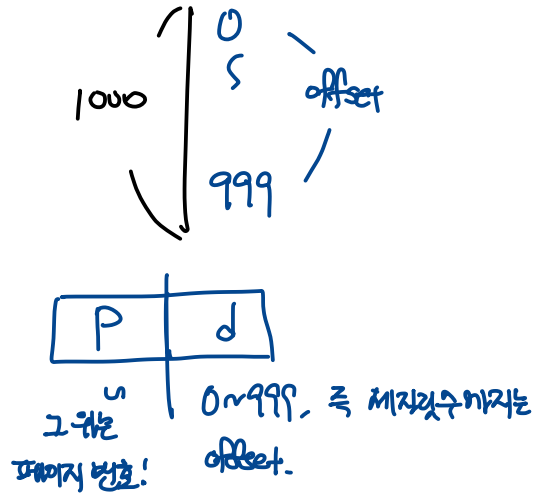
    page table: contains page-frame mapping info.

    page offset (d): page number and page offset combines to define the physical memory address



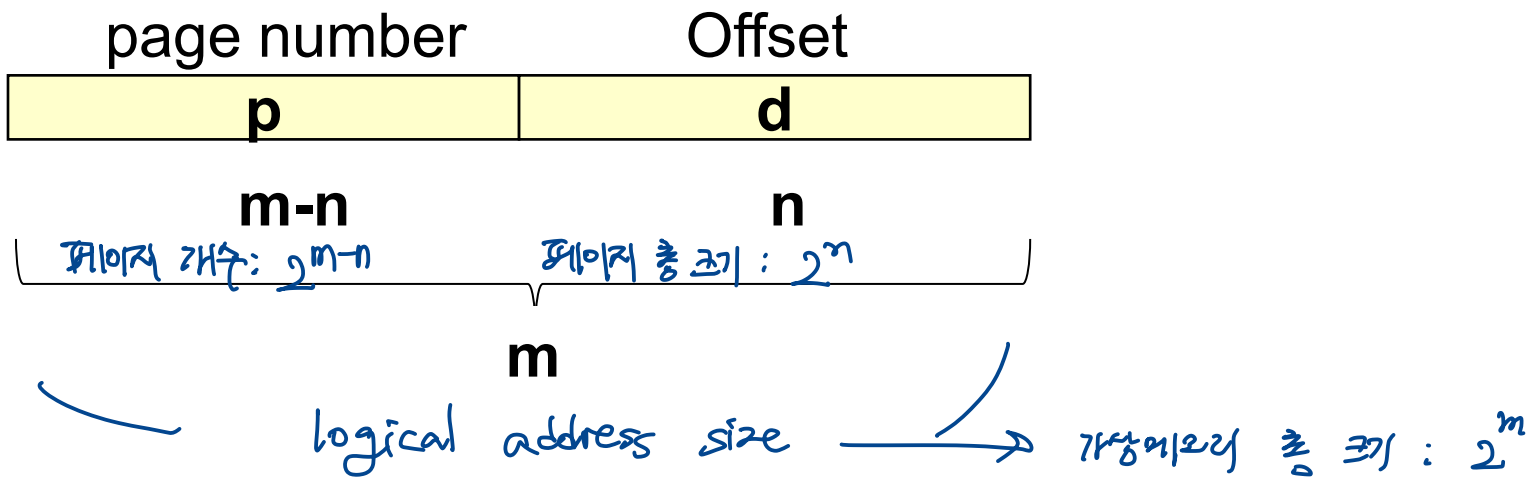
Paging Examples (From 2017 Quiz)

- Assuming a 1000B page size, what are the page numbers and offsets for the following address references (provided as decimal numbers)?
  - (1) 42095: page number 42 offset 95
  - (2) 215201: page number 215 offset 201



Page Size

- Fixed sized chunk: **page size** (=frame size) defined by hardware
  - vary between 512B (=2^9B) and 1GB (=2^{30}B) per page – depending on computer architecture → 우리는 4KB 정도로 사용할 것.
- If the size of **logical address space** is 2^m and **page size** is 2^n
  - **page number(p)**: high-order m-n bits of a logical address: **index** into a *page table*
  - **page offset(d)**: low-order n bits of a logical address



Paging Examples (From 2017 Quiz)

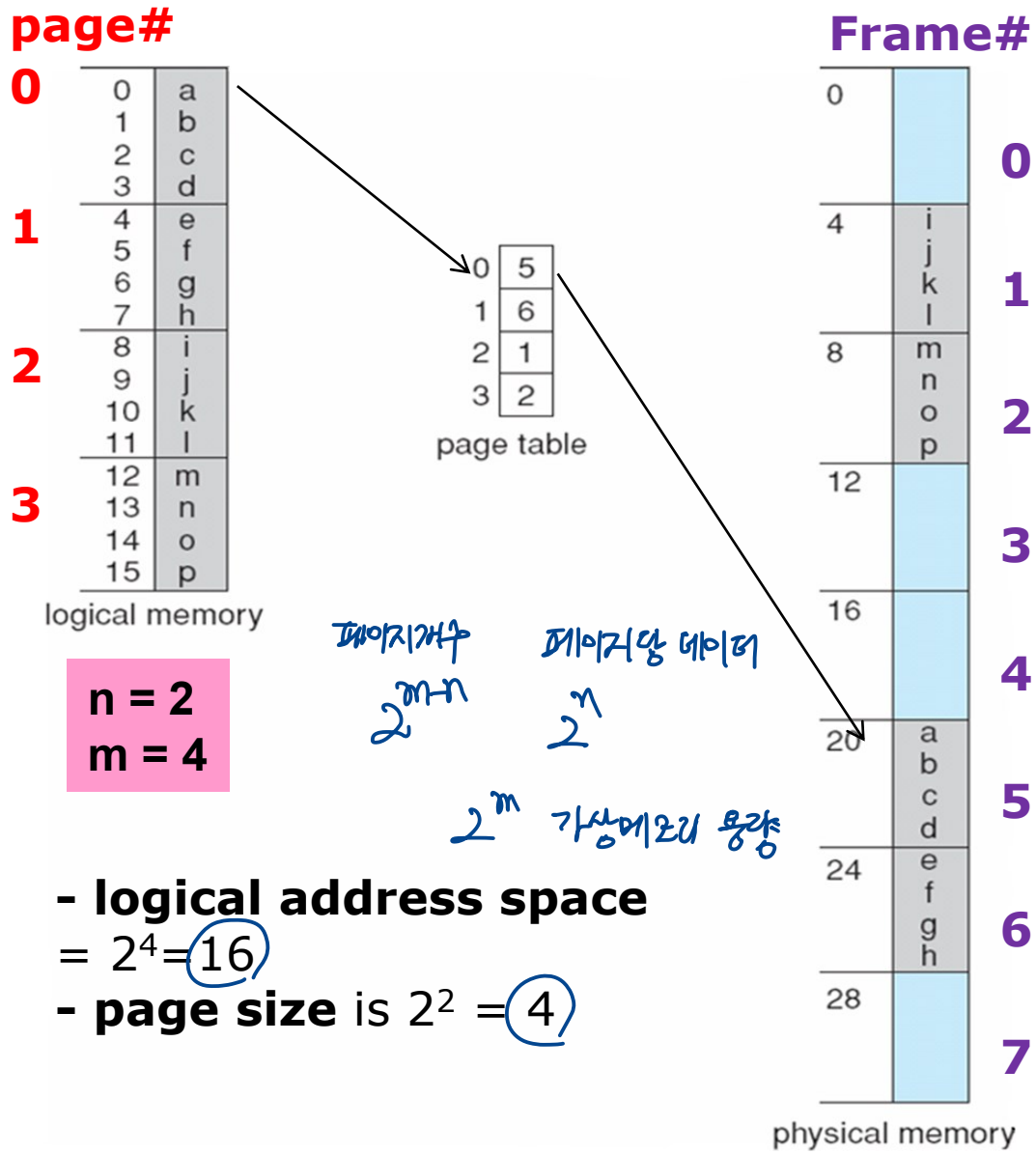
- Consider a logical address space of 128 (=2^7) pages with a 4kB (=2^{12}B) page size, mapped onto a physical memory of 64 (=2^6) frames.
  - (a) How many bits are required in the logical address?
  - 128 pages 2^7 4kB page: 2^{12} → 19bits

    p d

    2^7 2^{12}

    2^{19}
  - (b) How many bits are required in the physical address?
  - 64 frames: 2^6 4k-B frame: 2^{12} → 18bits

Paging Example



n = 2
m = 4

페이지개수 2^{m-n}

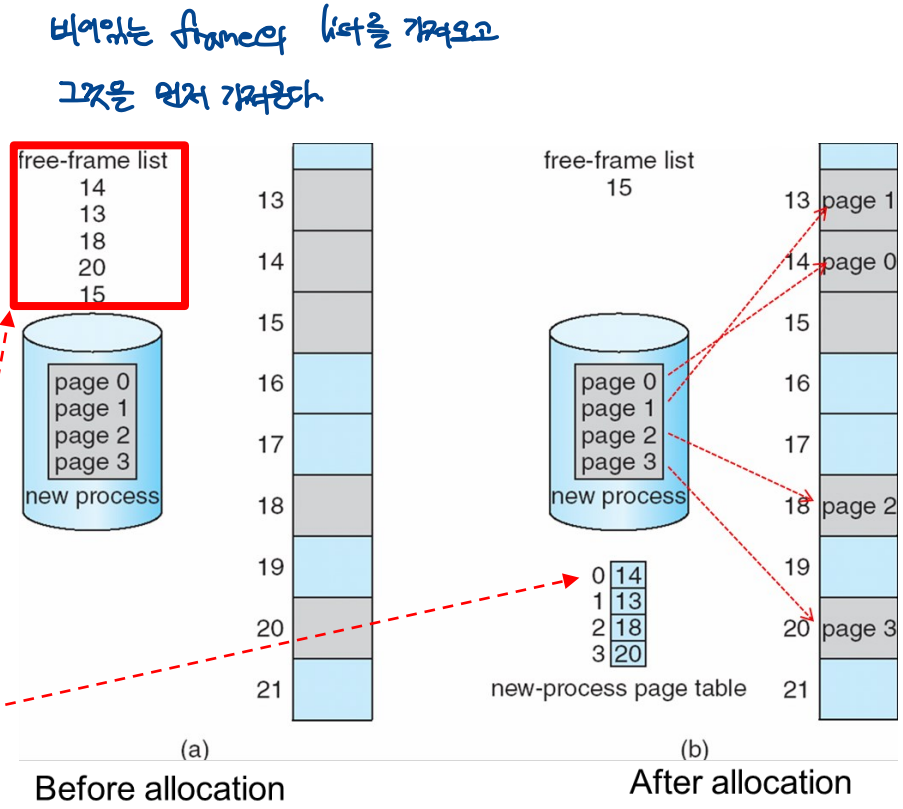
페이지당 메모리 2^{n}

2^{m} 가상메모리 용량

- **logical address space**
= 2^4=16
- **page size** is 2^2 =4

Free-frame management

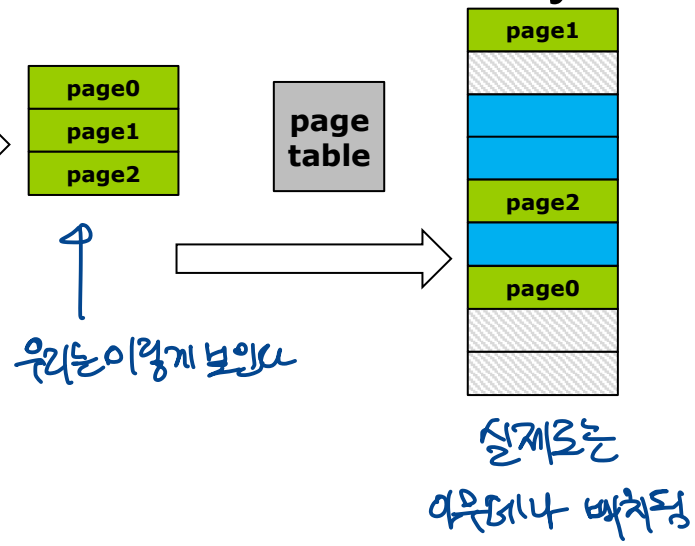
- To execute a process whose size requires **n** pages, at least **n** free frames are required – How do we find the **free frames**?
  - The first page of the process is loaded into the first frame listed on **free-frame list**
  - and the allocated frame number is put into **page table**
  - The next page is loaded into another frame …



A new process with 4 pages

Why Paging? (1)

- Clear separation between the programmer's view of memory (logical memory) and physical memory
  - Programmer: One contiguous single space
  - Actual physical memory: scattered throughout the physical memory
  - Needs ***page table*** for address translation



1. Provides **Transparency**
   - Programmer only sees *logical address*, never sees actual *physical address* – *physical address* is transparent to programmer
   - This mapping is *hidden* from programmer and controlled by OS

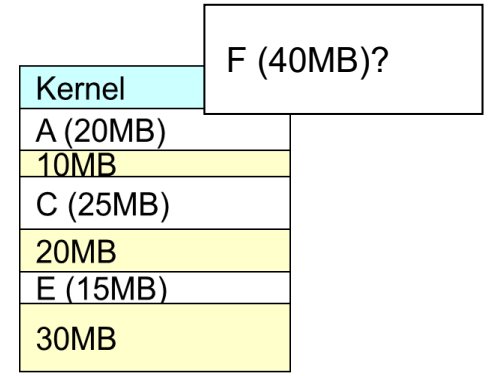
Why Paging? (2)

2. No external fragmentation

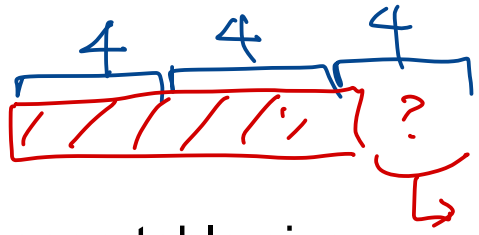
- Noncontiguous memory allocation - ANY free frame can be allocated to a process that needs it.
- However, we may have **internal fragmentation**
  - e.g.) if a page size is 2048 bytes, a process of 72766 bytes would need 35 pages plus 1086 bytes
  - In worst case, a process would need n pages plus 1 byte, and it would be allocated n+1 frames – internal fragmentation in last frame

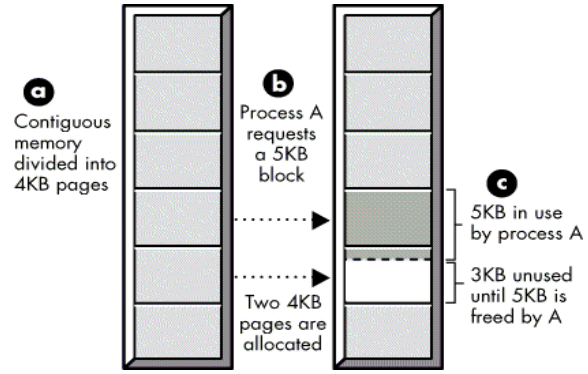
Internal Fragmentation

- **External Fragmentation**
  - Contiguous allocation: Total memory space exists to satisfy a request, but it is not contiguous



- **Internal Fragmentation**
  - Memory is allocated in fixed-sized **pages**
  - Allocated **pages** may be larger than requested memory;
    - page size 4kB,
    - process needs 9kB – allocated 3 pages (12kB)
  - this size difference is memory internal to a partition, but not being used





Internal Fragmentation

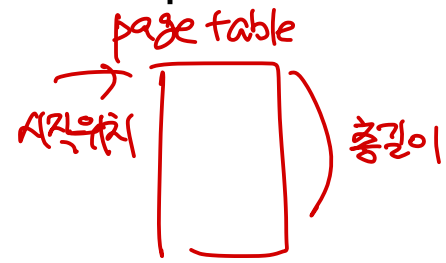
- How do we determine the Page Size?
  - Consider **tradeoff**: Internal fragmentation, page table size, disk I/O efficiency
  - Practically 4 – 8KB in size

Page 크기가 작으면 좋다.

그러나 항상 좋을수는 없다!

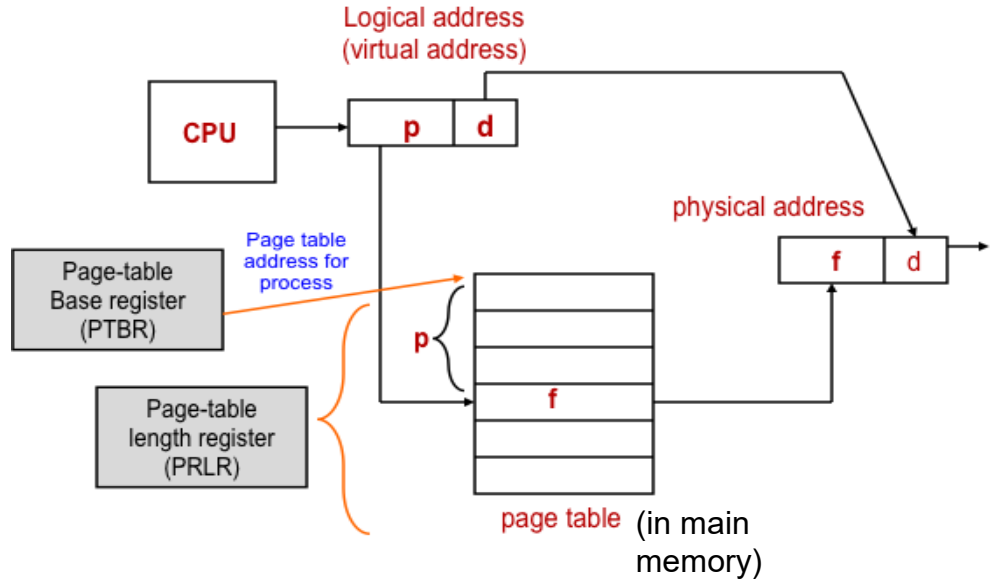
Implementation of Page Table

- There is a **page table** for **each process**!!
  - e.g., process P1's page table, P2's page table, ...
- **Option 1**: Keep in registers and make if fast
  - Implement page table as **registers**: translation very fast but expensive – need to keep it small (e.g., 256 entries)
  - But page table can be very large! (e.g., 1 million entries)
- **Option 2:** Keep in memory with register help
  - **Page table** is kept in **main memory**
  - Use **CPU registers** to store location and size of page table – hardware support

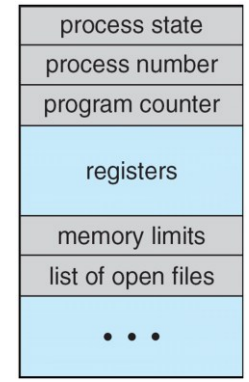


Implementation of Page Table

- **Page table** is kept in **main memory** with **(register) hardware support**:
  - ***Page-table base register*** **(PTBR)**
    - points to the page table for its process
  - ***Page-table length register*** **(PTLR)**
    - size of the page table (for the process)



- When context switch happens,
  - Need to change the page table (why? read the first line above)
    - changing page tables requires changing the **PTBR** and **PTLR**
    - so where should **PTBR**, **PTLR** values be stored when context switching happens?

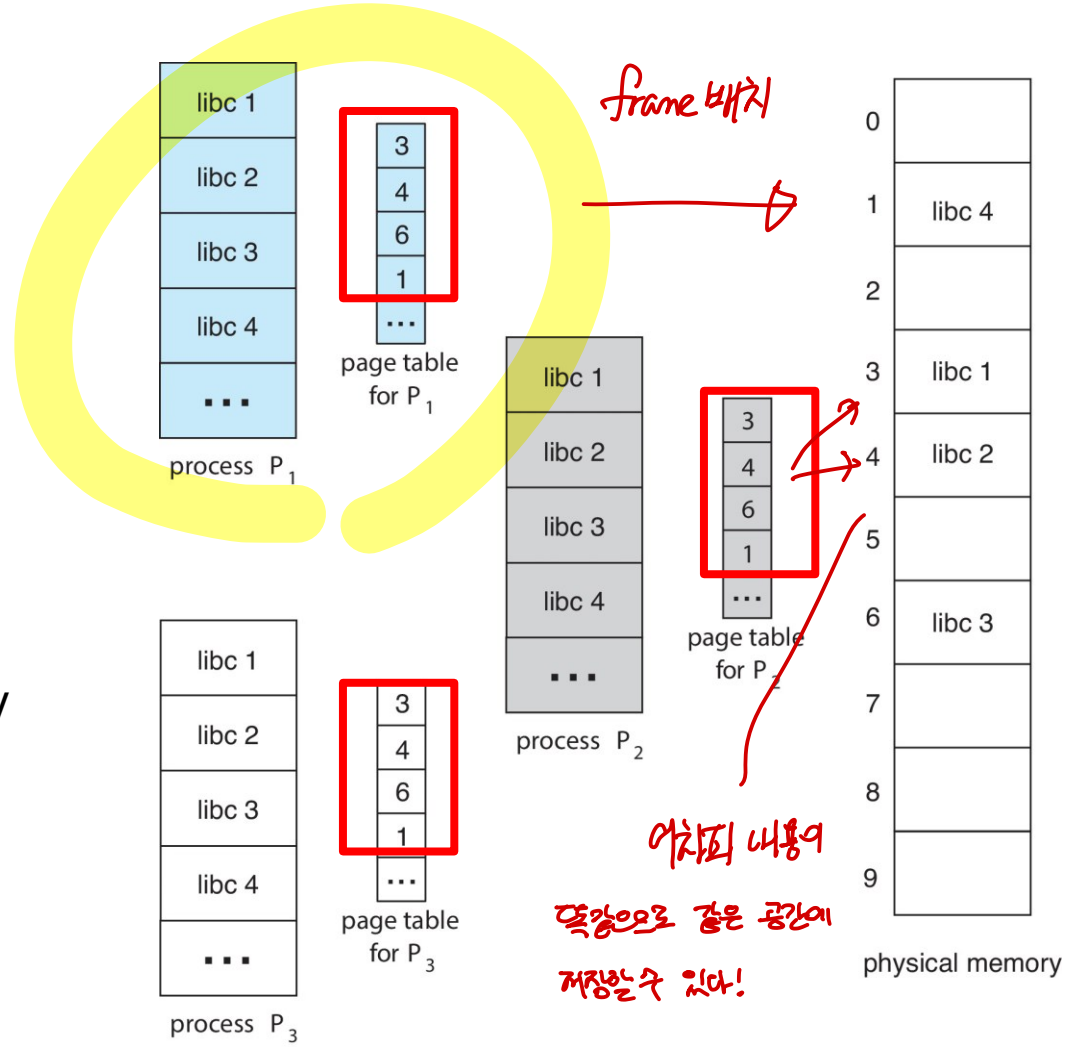


Shared Pages

- **Shared code**
  - One copy of read-only code shared among processes (i.e., libraries, text editors, compilers, database systems)
  - Similar to multiple threads sharing the same process space
  - Also useful for interprocess communication if sharing of read-write pages is allowed
- **Private code and data**
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space

Shared Page (Shared Memory)

- Shared code
  - One copy of library (libc) shared among processes
    - Example: 40 users run libc code (2MB)
      - need total 80MB physical memory
    - Share the code!
      - need only 2MB physical memory



Why Paging? (3)

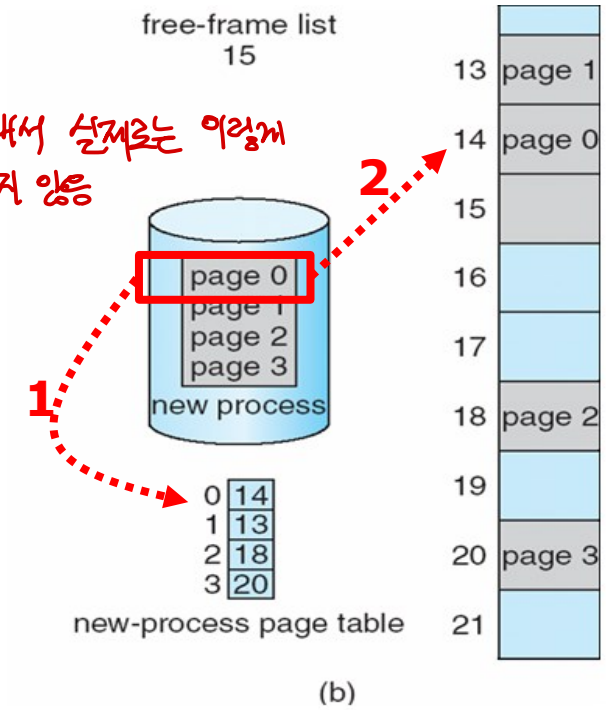
- Provides **Protection** to memory
  - Programmer can never access memory outside the page table.
  - Programmer can never access memory outside the page table length defined in **page-table length register (PTLR)**
- Provides **Shared memory**
  - Use shared pages

Page Table in Main Memory

- **Page table** is kept in **main memory**: Problems?
  - Every data/instruction memory access *actually* requires **two memory accesses**
    - 1. page table 2. data/instruction
  - Example
    - Process wants to read memory at logical address 010 (page 0, offset 10)

    1. First read page table to look for physical frame number of page 0 (=14)

    2. Read frame number 14



Page Table in Main Memory

- Isn't it slow to have to go to memory twice every time?
  - It takes about 100~400 CPU cycles to access DRAM
  - Yes, it would be… so, real MMUs don't

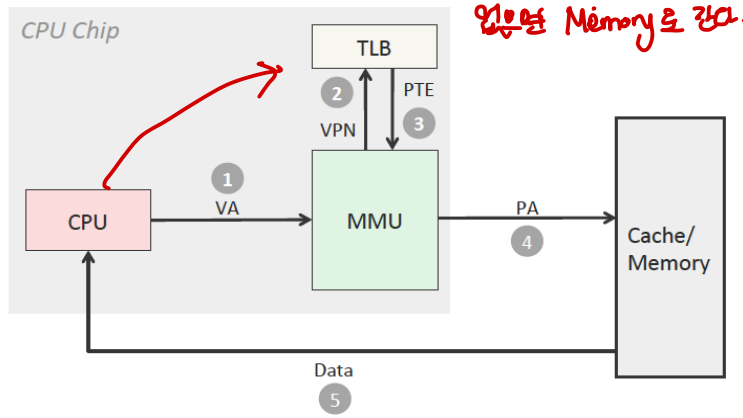
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**
  - **TLB** is like a cache to the page table
    - e.g., cache memory is a cache to the main memory (RAM)
  - Store recently referenced page table entries in the TLB

cache 같은 역할!

TLB (Translation Look-aside Buffer)

- **Small, dedicated, super-fast hardware cache in MMU**
  - High speed memory technology, approaching register speeds
    - Implement using the dedicated registers or cache memories
    - Parallel search
  - Each TLB entry contains a page number and the corresponding page-table entry
    - Search key = logical address, result = frame number

- Memory references that can be translated using TLB entries are much faster to resolve
  - high speed, but Expensive hardware

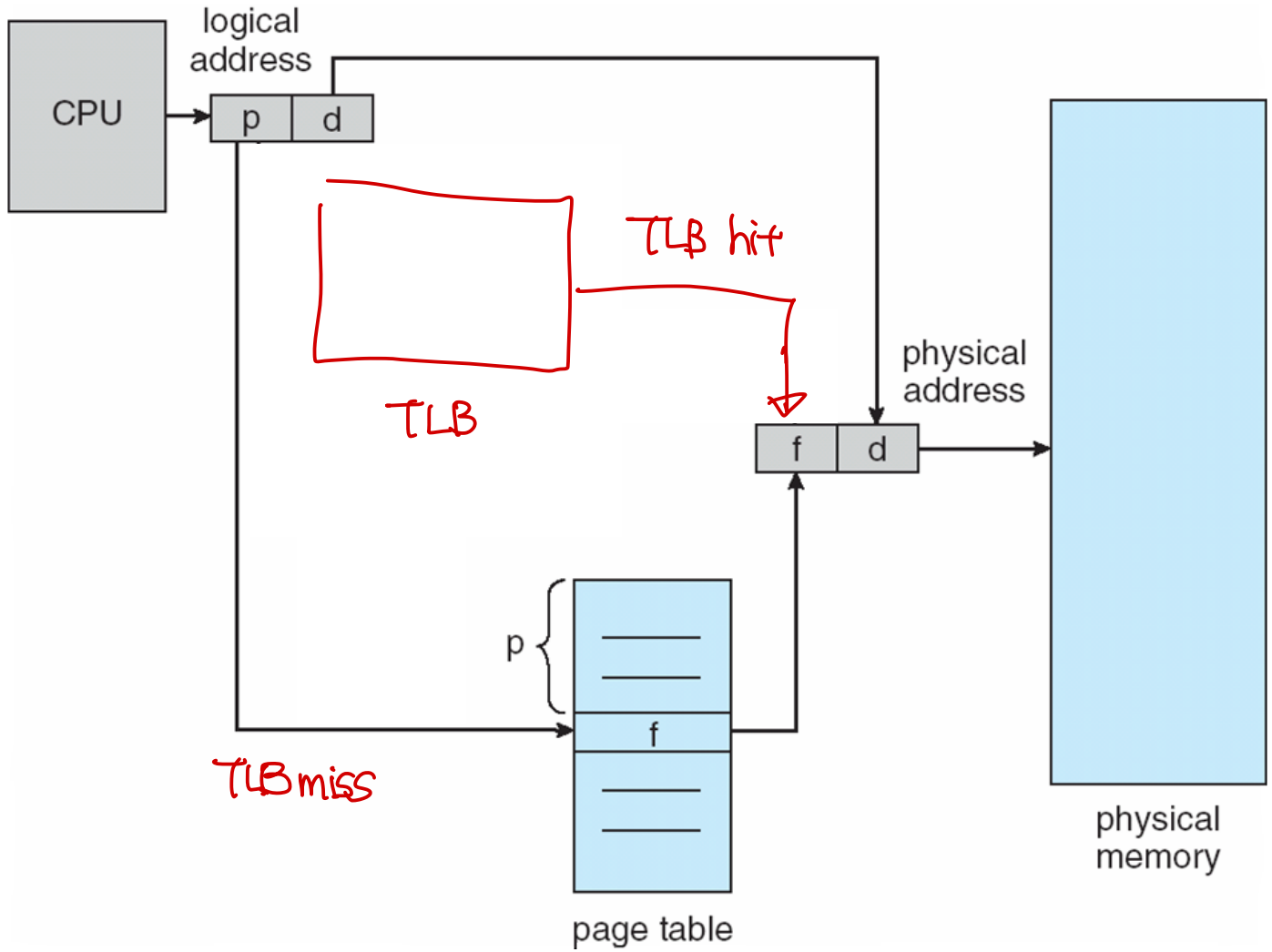


TLB

- **Recently-referenced** Page Table Entries (PTE's) are stored in the TLB
  - only small part of the page table is kept in TLB, typically 32 ~ 1,024 entries

- **TLB hit**: memory reference is in TLB
- **TLB miss**: memory reference is not in TLB
  - A TLB miss incurs an additional memory access to the page table
  - In this case, the page number/frame number is *added* to TLB for next page reference

TLB


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

Effective Access Time (EAT)

- **Hit ratio**
  - percentage of times that a page number is found in **TLB**
  - e.g., 80% hit ratio: find the page in TLB 80% of the time
- **Assume**
  - memory access time = 100 ns, TLB access time = 1ns
  - Hit ratio = α

만약 TLB를 쓰지 않을경우 고정 200ns 발생

- **Effective Access Time (EAT)**

$$EAT = \alpha \times (1 + 100)ns + (1 - \alpha) \times (1 + 100 + 100)ns = 201 - 100\alpha$$

TLB: $\alpha \times (1 + 100)ns$ — data/instruction

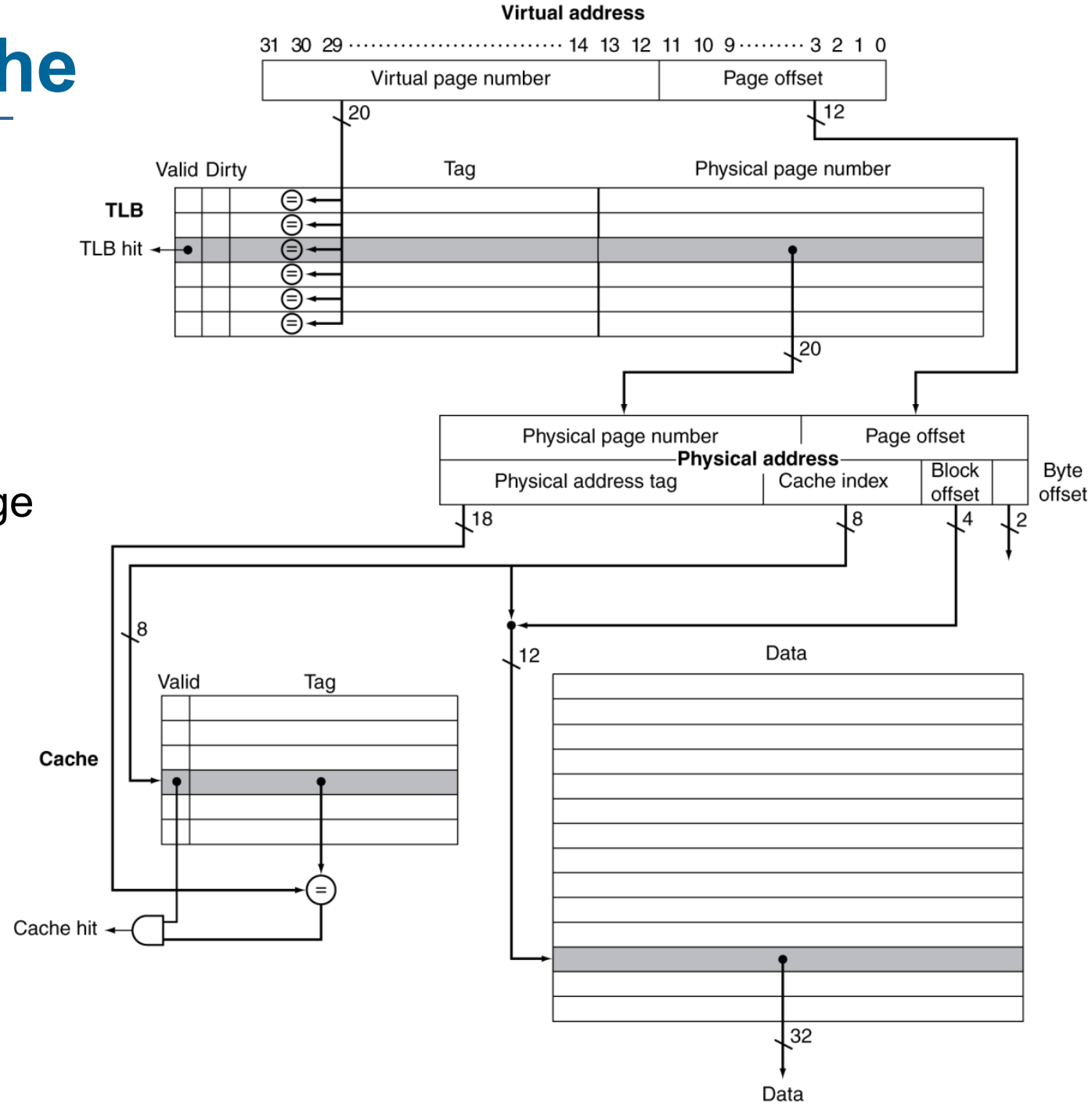
TLB: $(1 - \alpha) \times (1 + 100 + 100)ns$ — page table, data/instruction

if $\alpha = 0.8$ then EAT = 121ns, if $\alpha = 0.99$ then EAT = 102ns

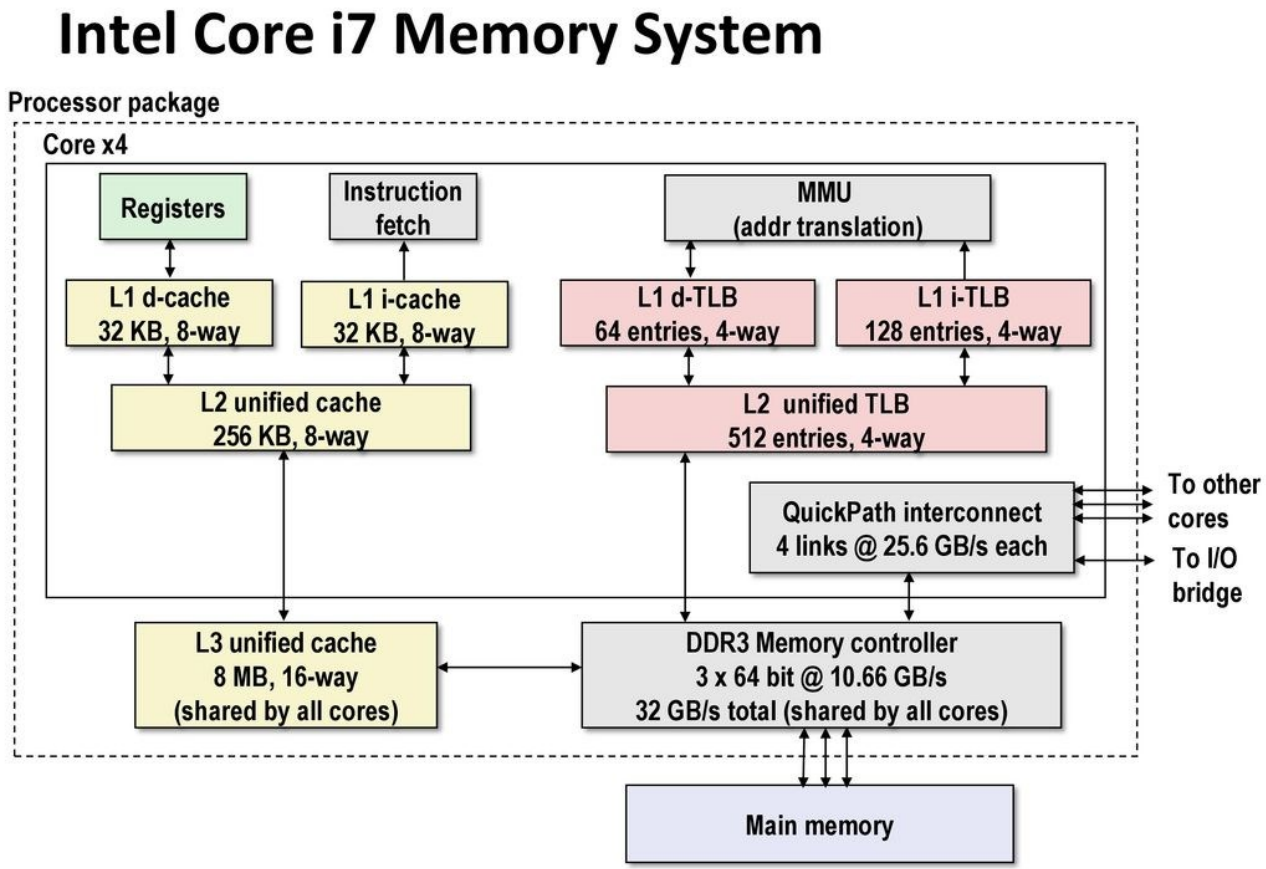
TLB with cache

Memory access order

- **Logical (virtual) address**
- Check **TLB**
  - **TLB hit**: get address from TLB
  - **TLB miss**: check page table in memory
- Acquire **physical address**
- Check **cache**
  - **cache hit**: get data from cache
  - **cache miss**: get data from main memory
- Acquire data



Example: Intel Core i7 Memory System



https://slideplayer.com/slide/13277282/

Chapter 8: Memory Management Strategies

- Introduction
- Virtual Memory
- Contiguous Memory Allocation
- Paging
- **Structure of the Page Table**
  - Hierarchical Paging (Two-level page table)
  - Inverted Page Tables

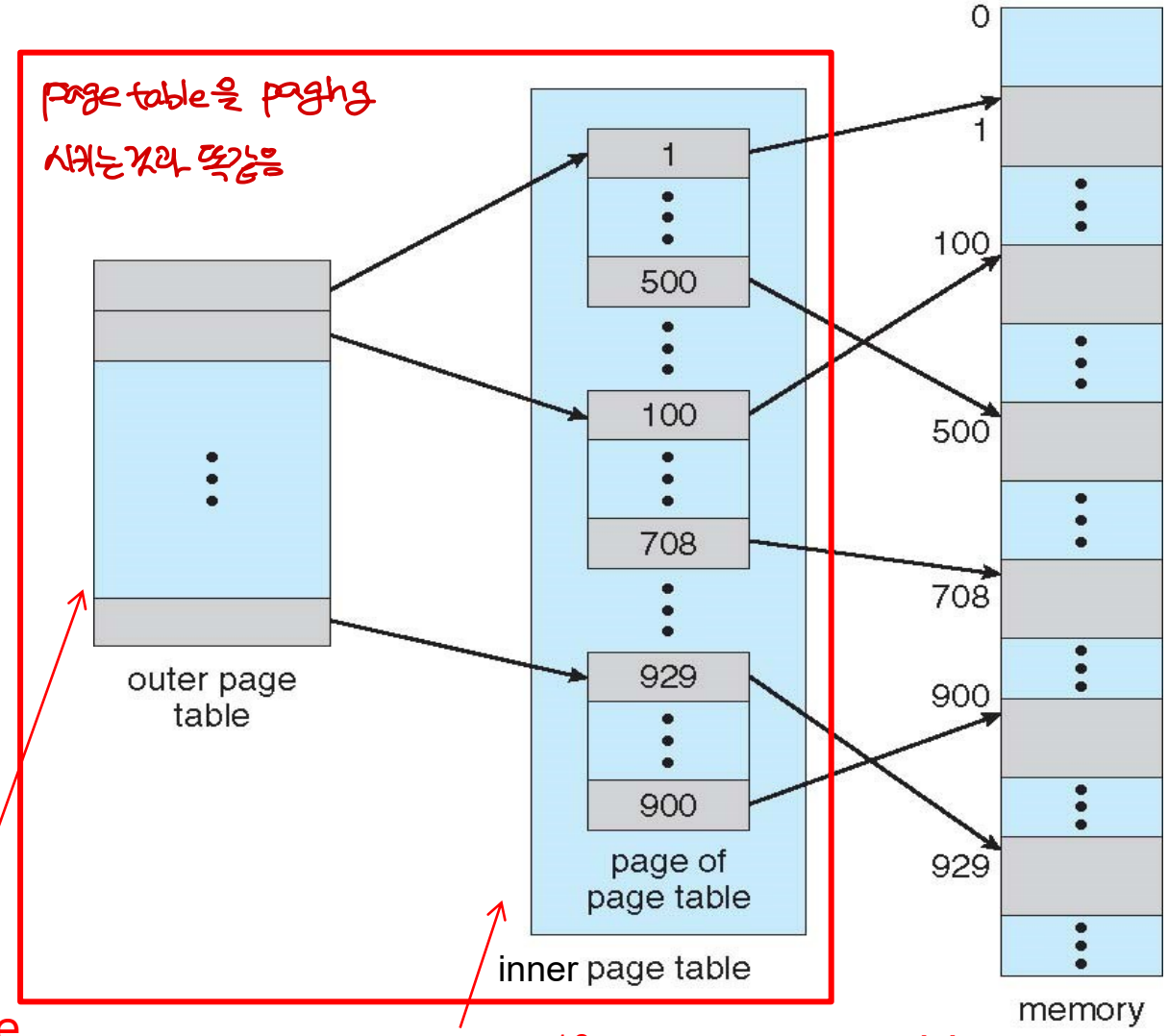
How to Store Page Table

- Consider a system with 32-bit logical address space
  - 32-bit logical address space: 4GB (=2^{32}B)
  - If page size: 4KB (=2^{12}B)
  - Each page table needs up to 4GB/4KB = $2^{20} \approx 10^6$ = 1 million (2^{20}) 4B page entries
    → 4MB ***contiguous*** **physical memory space** per each page table
    - Note: each process needs its own page table!

- Solution? **Hierarchical paging**
  - Divide the page tables into smaller pieces
  - Multi-level paging, in which the page table is also paged

Two-Level Paging (32-bit address, 4kB page)

- Break up the logical address space into multiple page tables (logically)
- Inner page tables are allocated only if needed

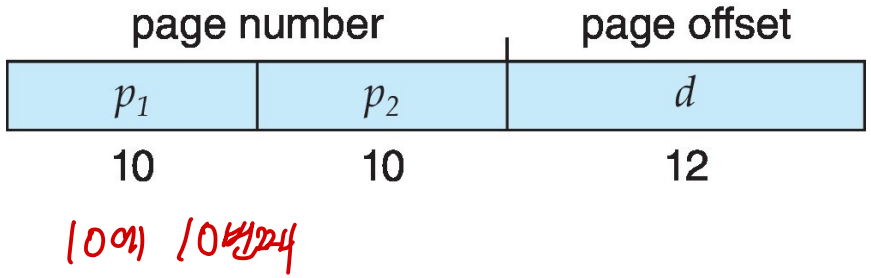


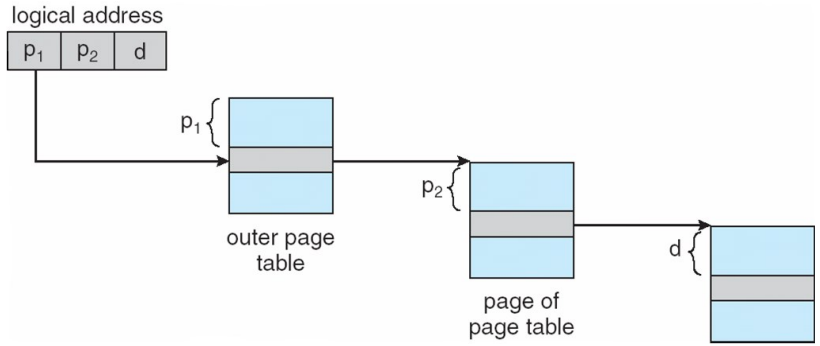
1 outer page table
(with 2^{10} entries = 4kB)

$0 \sim 2^{10}$ inner page tables
(with 2^{10} entries = 4kB)

Two-Level Paging (32-bit address, 4kB page)

- Since **the page table is paged**, the page number is further divided into:
  - a 10-bit outer-page number, a 10-bit inner-page number, a 12-bit page offset.

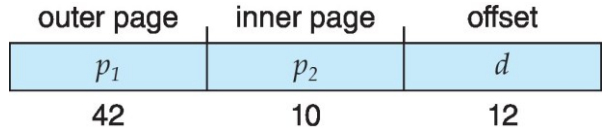




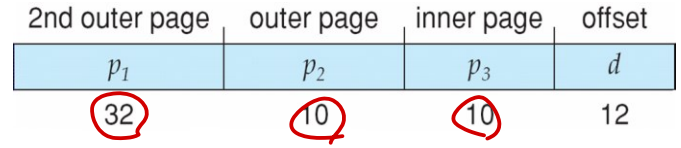
- where p_1 is an index into the **outer page table** (2^{10} = 1,024 entries: 4KB), and p_2 is the displacement within the page of the **inner page table** (2^{10} = 1,024 entries: 4KB).

Three- Four-Level Paging (64-bit address)

- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look like



  - Outer page table has 2^{42} entries or 2^{44} bytes ($2^{42} entires \times 4B/entry = 2^{44}B$)
  - One solution is to add a 2nd outer page table



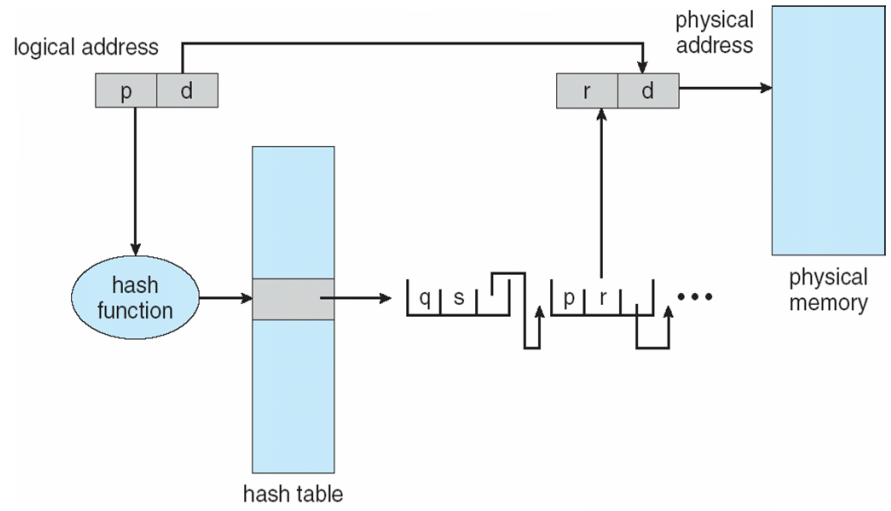
  - But in this example the 2nd outer page table is still 2^{34} bytes in size
    - And possibly **4 memory access** to get to one physical memory location!
    - Hierarchical paging is not appropriate for 64-bit architectures

64bit에서는 … 사용 불가능

Hashed Page Tables

- Common in address spaces larger than 32 bits
- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location (to handle collisions)
- Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted

해쉬 테이블



Inverted Page Table

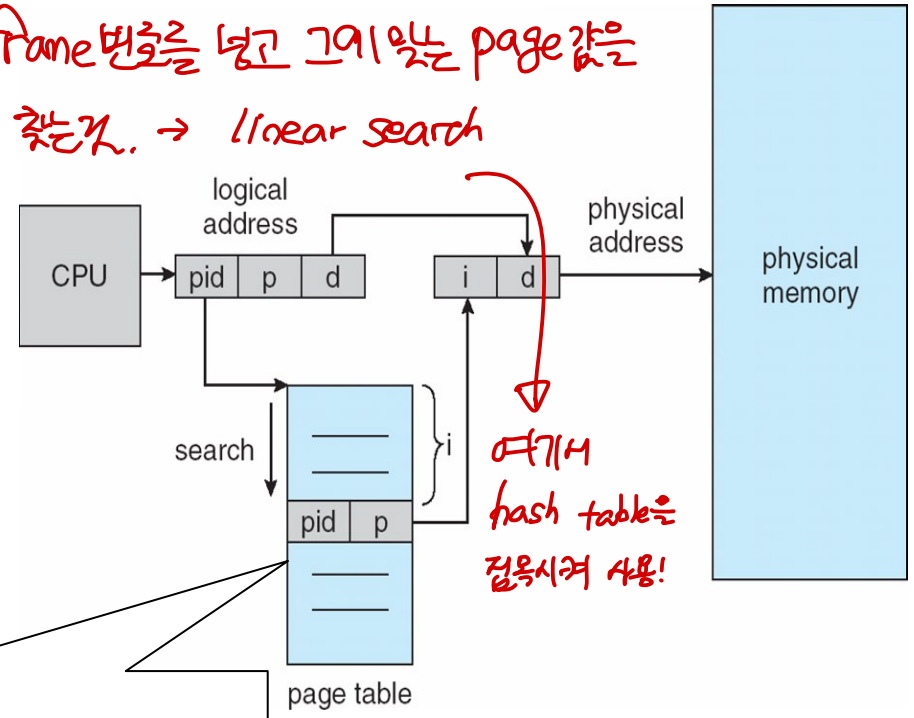
- Usually, each process has a page table
  - Translate logical address (page) into physical address (frame)
  - Each page table can be very large! (as seen in last few slides)

역경

- **Inverted page table**
  - Only one page table in OS!
  - One entry for each real frame of memory
  - Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page

- Decreases memory needed to store each page table
  - Total table size = 1M entries (4MB size)

- Any disadvantages?



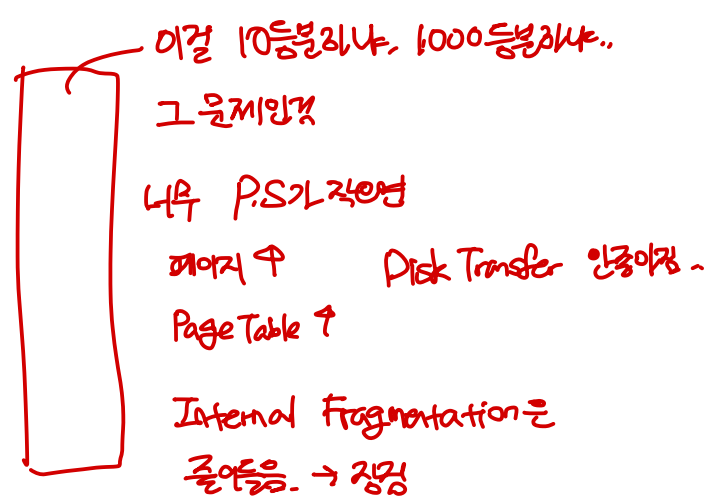
PT: page# → frame#
inverted PT: frame# → pid, page#

Inverted Page Table

- Advantage: Decreases amount of memory needed
- Disadvantages:
  - Increases time needed to search
    - Inverted page table is sorted by physical address, but the lookups occur on virtual addresses: If there are N entries, may need up to N searches!
    - Solution: Use **hash table** to limit the search page-table entries
  - Difficult to implement shared memory
    - One mapping of a virtual address to the shared physical address
      - One physical page cannot have two (or more) shared virtual addresses

Other Considerations: Page Size (Ch. 10.9)

- Sometimes OS designers have a choice
  - Especially if running on custom-built CPU
- Smaller Page size gives:
  - More pages
  - Larger Page Table
  - Inefficient Disk Transfer
    - More I/O overhead
    - More page faults
  - Less Internal Fragmentation
- Trend
  - Larger page sizes



Obtaining the Page Size on Linux Systems

On a Linux system, the page size varies according to the architecture, and there are several ways of obtaining the page size. One approach is to use the system call `getpagesize()`. Another strategy is to enter the following command on the command line:

```
getconf PAGESIZE
```

Each of these techniques returns the page size as a number of bytes

Summary

- Various memory management schemes
  - Contiguous memory allocation
  - Noncontiguous memory allocation – paging
- H/W support for memory protection
  - Base and limit registers
  - TLB – associative mapping table
- Fragmentation problems
  - Internal and external fragmentation → 자리양아요~
- Paging system
- More examples of Intel 32/64-bit and ARMv8 Paging architecture in your textbook

https://www.istockphoto.com/kr/%EC%9D%BC%EB%9F%AC%EC%8A%A4%ED%8A%B8/qa