Memory Management(1)

OS의 메모리관리 핵심: CPU가 명령어를 수행할 수 있도록 해주는 것 PC → Inst

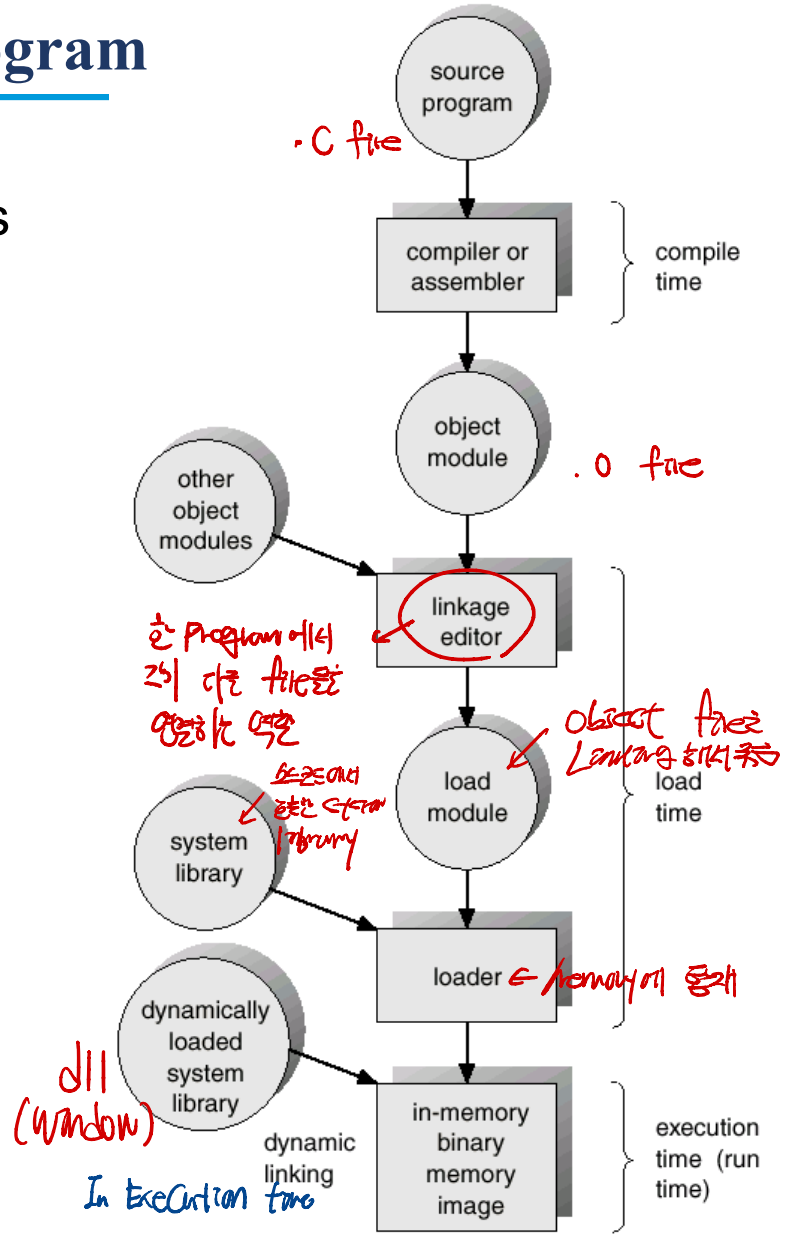
Dept. of Computer Science
Hanyang University

Background

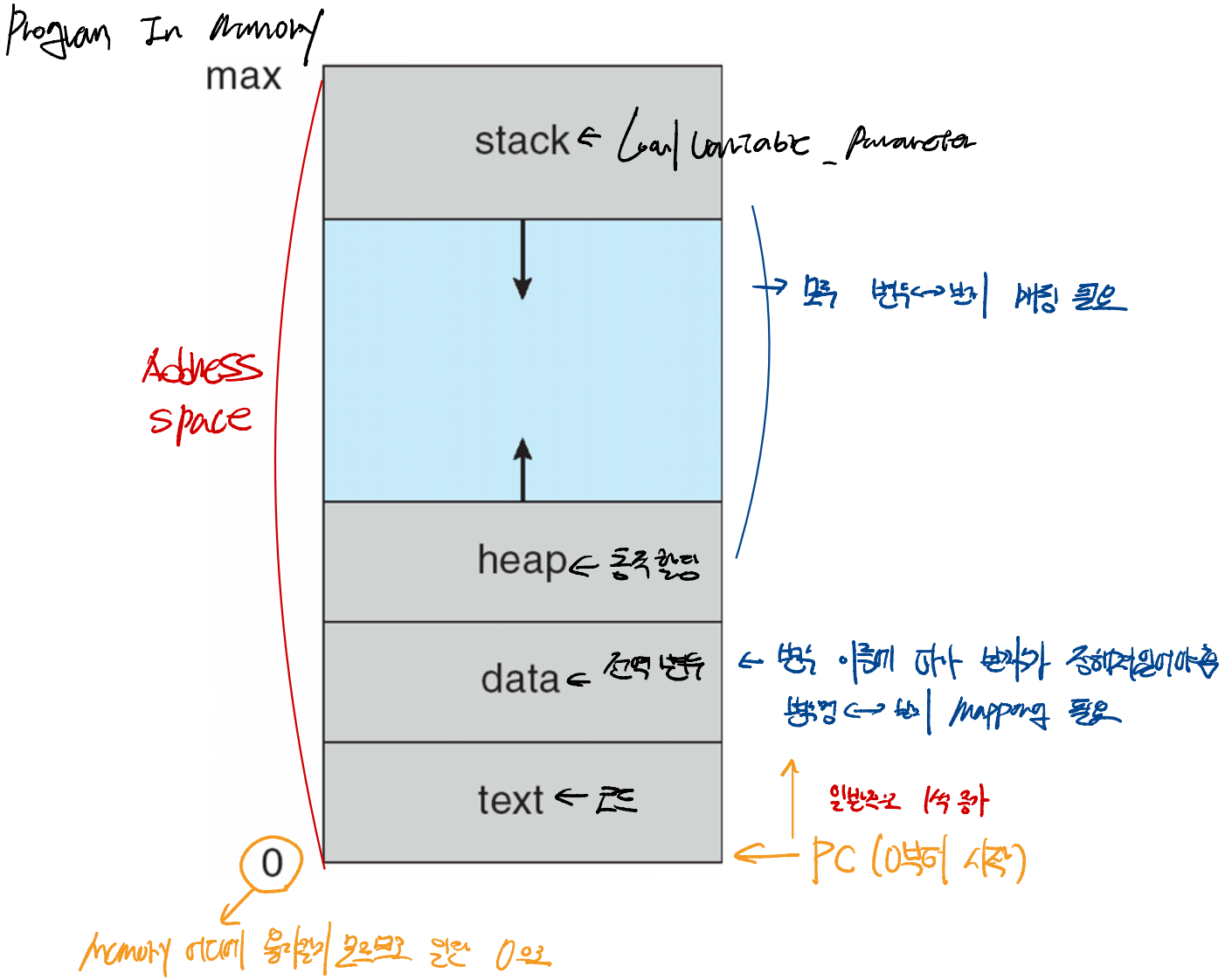
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are the only storage CPU can access directly
- Register access in one CPU clock cycle (or less)
- Main memory can take many cycles
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

Multistep Processing of a User Program

User programs go through several steps before being run on memory.



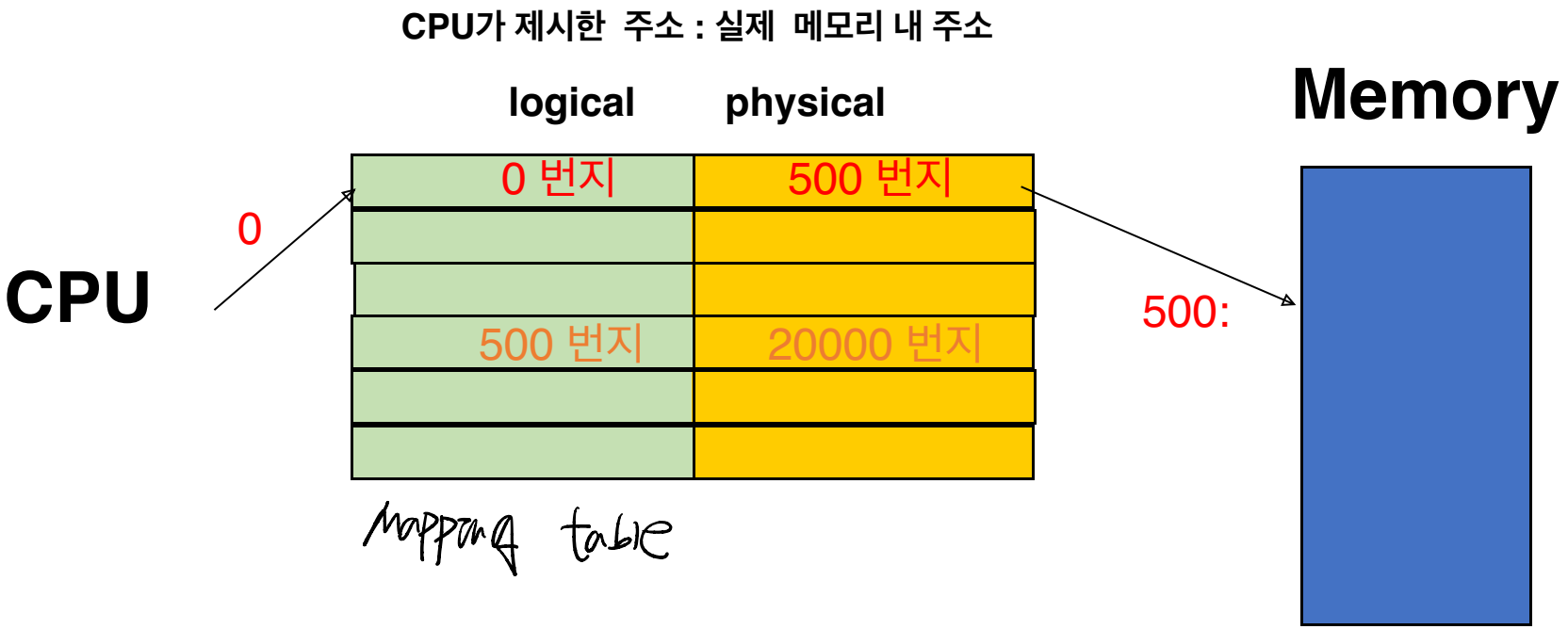
Process in Memory: Address Space



Binding of Instructions and Data to Memory

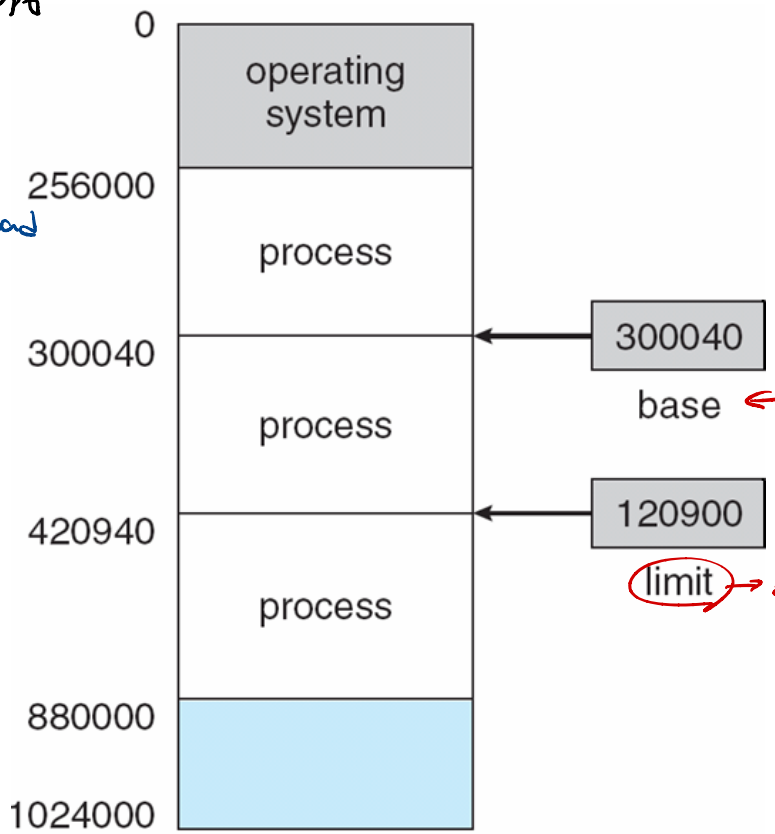
- **Compile time binding**
  - Absolute address of each symbol must be known at this time
  - *absolute code* containing absolute address is generated
  - *must recompile* code if starting location changes
- **Load time binding**
  - Loader assigns absolute address to each symbol
  - Compiler generates *relocatable code* containing relative addresses
- **Execution time binding**
  - Used when process moves its in-memory location during execution
  - Whenever CPU generates address, binding is required (address mapping table)
  - *need hardware support* (e.g., *base* and *limit registers*, MMU)

Address Mapping Table



Base and Limit Registers

- A pair of **base** and **limit** registers define the physical address space



Logical vs. Physical Address Space

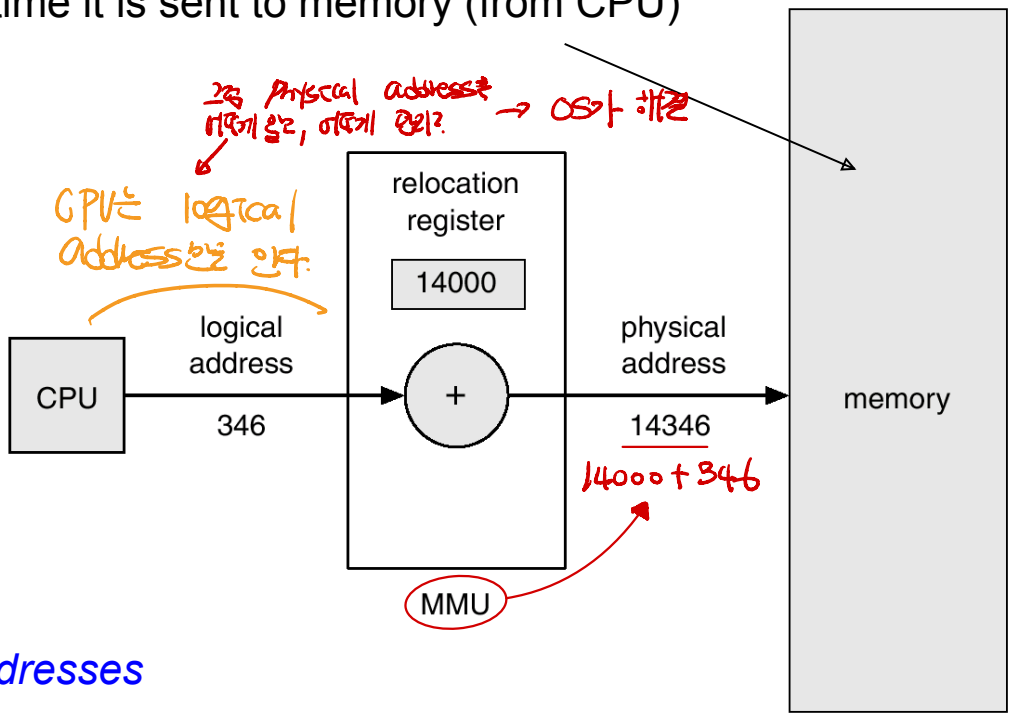
- The concept of a logical address space that is bound to a separate physical address space is central to proper memory management
  - Logical address (virtual address)
    - Generated by the CPU
    - Also referred to as virtual address
  - Physical address
    - Address seen by the memory unit

왜 분리?
CPU는 Physical address를 모른채 Logical address에 접근.
Logical address → Physical address mapping을 OS가 관리

- Logical and physical addresses are the same in compile-time and load-time address-binding schemes
- Logical (virtual) and physical addresses differ in execution-time address-binding scheme

Memory-Management Unit (MMU)

- ***MMU (Memory-Management Unit)***
  - A *Hardware device* that maps *virtual address* to *physical address*
- In MMU scheme
  - the value in the *relocation register* is added to every address generated by a user process at the time it is sent to memory (from CPU)



- The user program
  - deals with *logical addresses*
  - *never sees* the *real physical addresses*

Swapping necessitates dynamic relocation

- **Swapping** ← mid term scheduler
  - A process can be *swapped* <u>temporarily</u> out of memory to a *backing store* and then brought back into memory sometime later

Suspend 해제된 process가 다시 memory에 올라올 때 memory 상에서의 위치는 달라진다. ← Execution Binding은 swapping을 고려할 수 있어야 (Dynamic Allocation이기 때문)

- <u>**Backing store**</u>
  - Fast disk large enough to accommodate all memory images for all users
  - Must provide direct access to these memory images

Swap out된 Process가 저장된 공간 (DISK)

일반적으로 process address space ↑ → 위 DISK에 저장/읽는 속도가 빨라야 함 ↓ 빠른 I/O 필요, 최적화된 메커니즘 → 연속적으로 쓰고/읽는 I/O

↳ process의 address space가 커서 혹여 address space가 작으면 오히려 이 방법이 비효율적

- Major part of swap time is transfer time
  - Total transfer time is proportional to the amount of memory swapped

궁금한 점

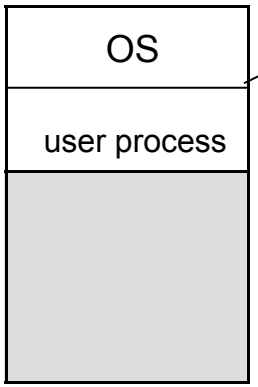
1. MMU는 관련 정보 → 빠른 연산을 위해 하드웨어까지 필요?
2. Process가 Memory에서 쫓겨나는 상황은 얼마나 일어나는가 ← 페이지

File system
→ 빈공간을 고려 안 해도
→ 연속적 X
→ Swap partition은 File system과 분리되어 있음

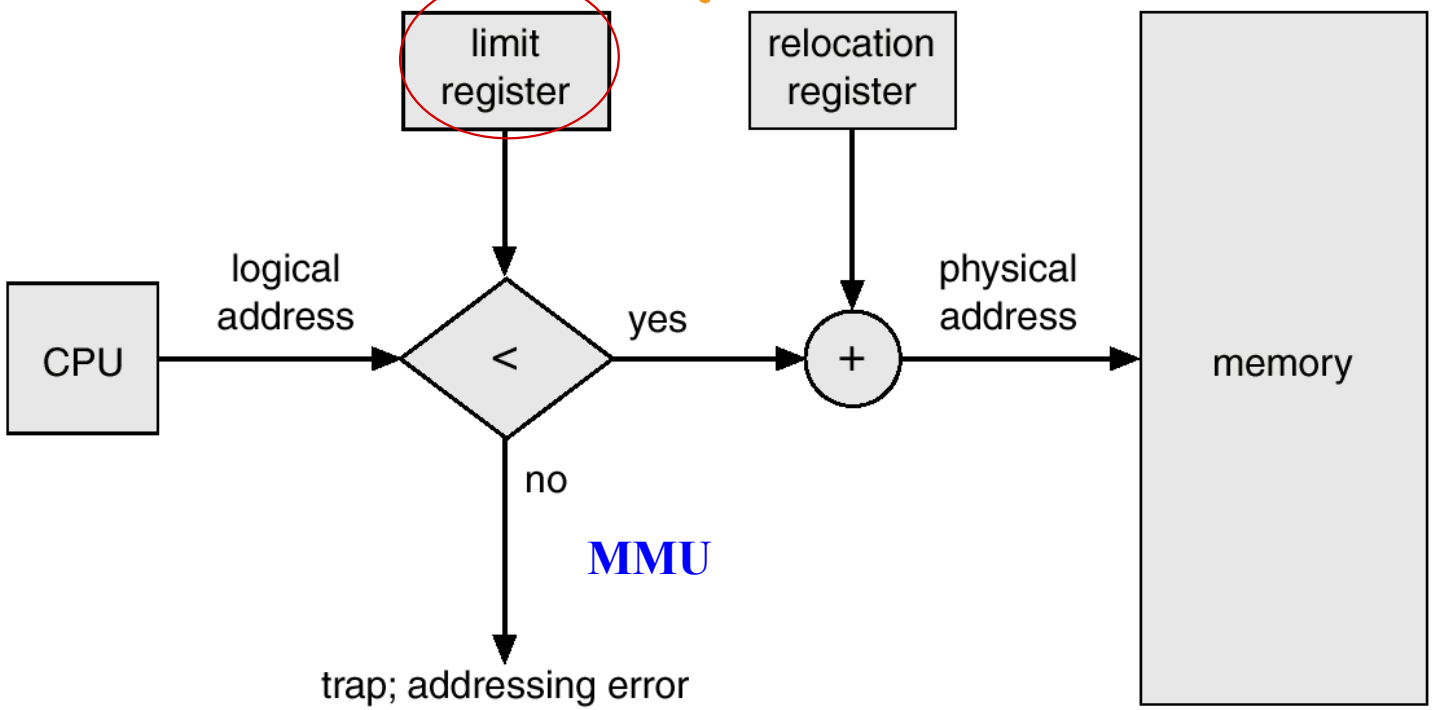
Contiguous Allocation : process를 DRAM의 연속된 공간에 할당하는 방법

MMU의 Mapping규칙의 예: process의 address space가 memory에서 연속적.

- Main memory is usually divided into two partitions:
  - *Resident operating system*, usually held in *low memory* with interrupt vector ← kernel 영역
  - *User processes* then held in *high memory*
  - To *protect* user processes from each other and OS
    - ***Relocation register*** (Base)
      - contains value of the smallest physical address
    - ***Limit register***
      - contains range of the logical addresses
      - each logical address bounded by limit register



Hardware Support for Relocation and Limit Registers



Contiguous Allocation (Cont.)

- 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
  - Operating system maintains information about:
    a) allocated spaces b) free spaces (hole)

Swapping 때문에 Memory 중간중간에 빈 공간이 생길 수 있다.

Hole - 크기도 다 다름
∝ process size

Hole을 OS가 관리해야함
→ 새로운 process를 Hole에 할당하기 때문
어느 Hole이 할당되는지
Hole에 대한 정보
Hole의 크기

Dynamic Storage-Allocation Problem

How to satisfy a request of size n from a list of free holes

- ***First-fit***
  - Allocate the *first hole* that is big enough
- ***Best-fit***
  - Allocate the *smallest hole* that is big enough
  - must search entire list, unless ordered by size
  - Produces many small leftover hole
- ***Worst-fit***
  - Allocate the *largest hole*
  - must also search entire list
  - Produces the largest leftover hole

First-fit and best-fit better than worst-fit in terms of storage utilization

Fragmentation

- External fragmentation
  - Total memory space exists to satisfy a request, but it is not contiguous
- Internal fragmentation
  - Allocated memory may be slightly larger than requested memory; this size difference is internal fragmentation : memory internal to a partition, but not being used
- Reduce external fragmentation by *compaction*
  - 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

Paging : External fragmentation을 발생하지 않도록 하는 방법

- **Paging**
  - is a scheme that *permits address space* to be *noncontiguous*

    레고처럼 memory 공간에 Address space를 끼워넣음. Memory와 Address Space 모두 같은 Size로 분할
    끼워넣음 → 이 경우 Address space가 연속적이지 않음

- Basic Method

  frame에 page를 넣음. 대신 frame이 비연속적이어도 된다.

  frame size = page size

  - Divide physical memory into *fixed-sized blocks* called *frames* (size is power of 2, between 512 bytes and 8 MB)

    FRAME은 일정한 size 지정

  - Divide logical memory into *blocks of same size* called *pages*

    2의 거듭 (보통 4KB)

    같은 크기로 나눔

  - Keep track of all free frames
  - To run a program of size *n* pages, need to find *n* free frames and load program

    → page가 비연속적이기 때문에 각 page가 메모리 어디에 있는지 관리

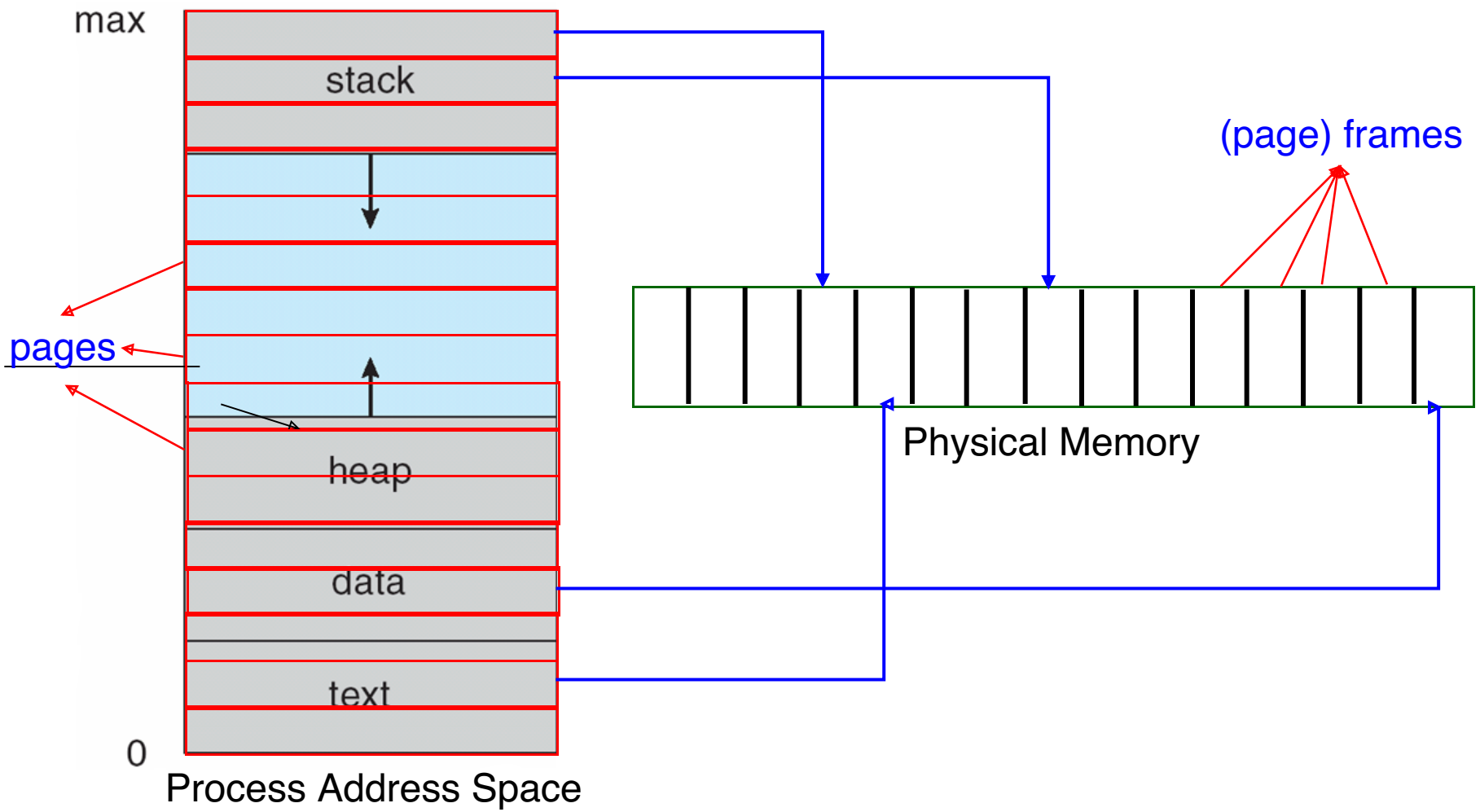
  - Set up a page table to translate logical to physical addresses
  - Internal fragmentation, but no external fragmentation

장점) Memory에서 남는 공간은 Frame 단위 → External fragmentation 발생X) 메모리 손실 개선

그러나 Internal fragmentation은 발생 ← 마지막 page에서만 발생 / 심각하지는 않음

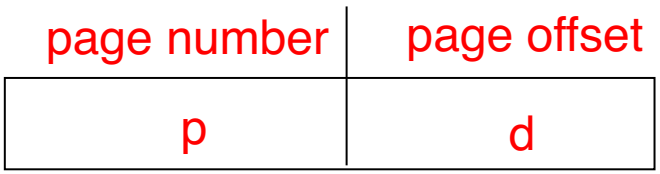
Ex) 4KB로 나눴을 때, Program이 1KB가 남으면 Frame(4KB)에 저장

Paging


max
stack
pages
heap
data
text
0
Process Address Space
(page) frames
Physical Memory

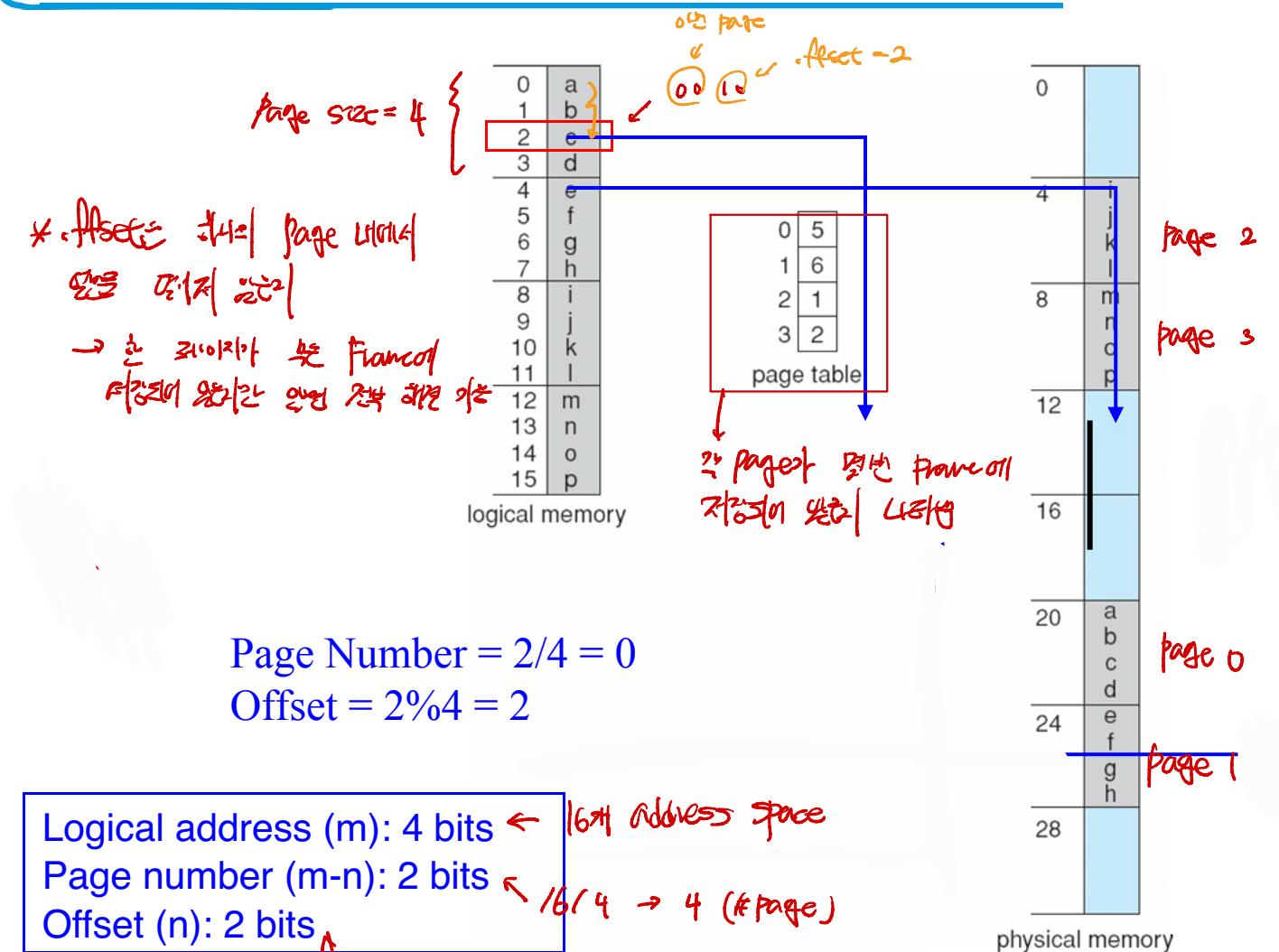
Address Translation Scheme in Paging

- Address generated by CPU is divided into:
  - **Page number (*p*)** – used as an index into a *page table* which contains base address of each page in physical memory
  - **Page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit



- For given logical address space 2*m* and *page size* 2n

Paging Example

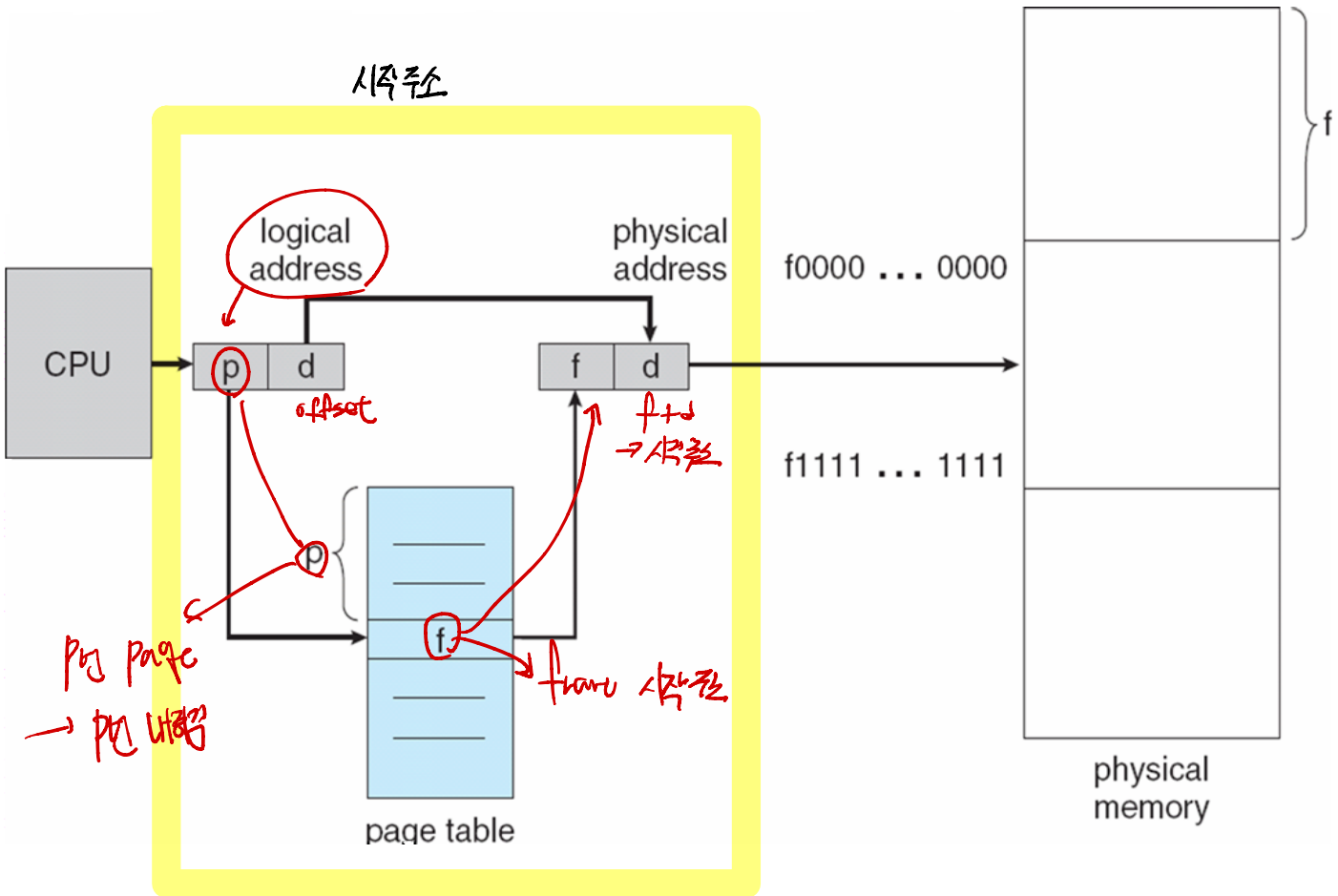


Page Number = 2/4 = 0
Offset = 2%4 = 2

Logical address (m): 4 bits
Page number (m-n): 2 bits
Offset (n): 2 bits

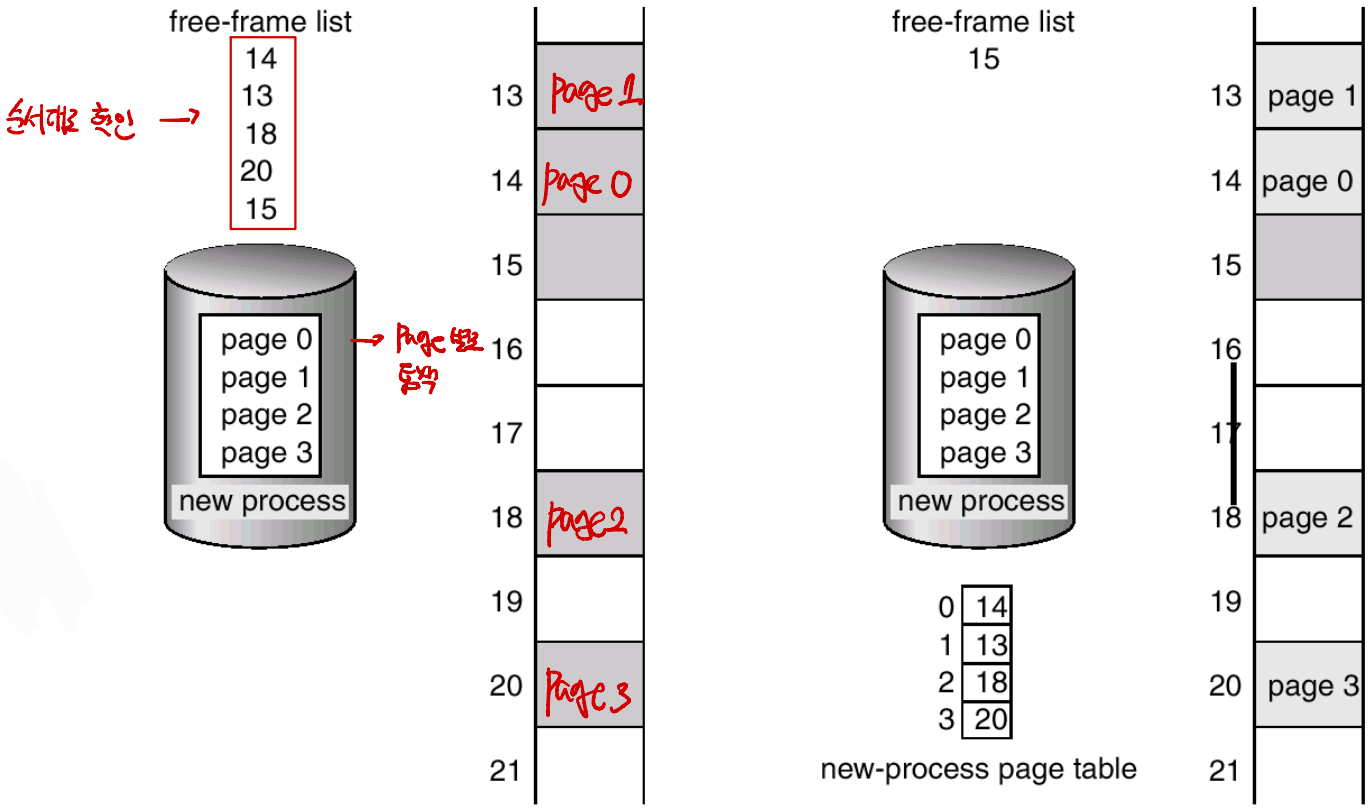
4-byte page n = 2 bits

Address Translation Architecture



Free Frames

free-frame 관리 방법?
→ Virtual memory



(a) Before allocation

(b) After allocation

Implementation of Page Table

- Page table is *kept in main memory*
  - *Page-table base register (PTBR)* points to the *page table*
  - *Page-table length register (PTLR)* indicates *size of the page table*
- Problem: *Every memory (data/instruction) access requires two memory accesses*
  - One for the *page table* and one for the *data/instruction*
- Solution: A special fast-lookup hardware cache called *Translation Look-aside Buffer (TLB)* or *Associative memory* is used
  - In general, TLB is flushed (remove old entries) at context switching time
  - Some TLBs store address-space identifiers (ASIDs) in each TLB entry, to avoid frequent TLB flushing
    - Uniquely identifies each process to provide address-space protection for that process

Associative Memory (TLB)

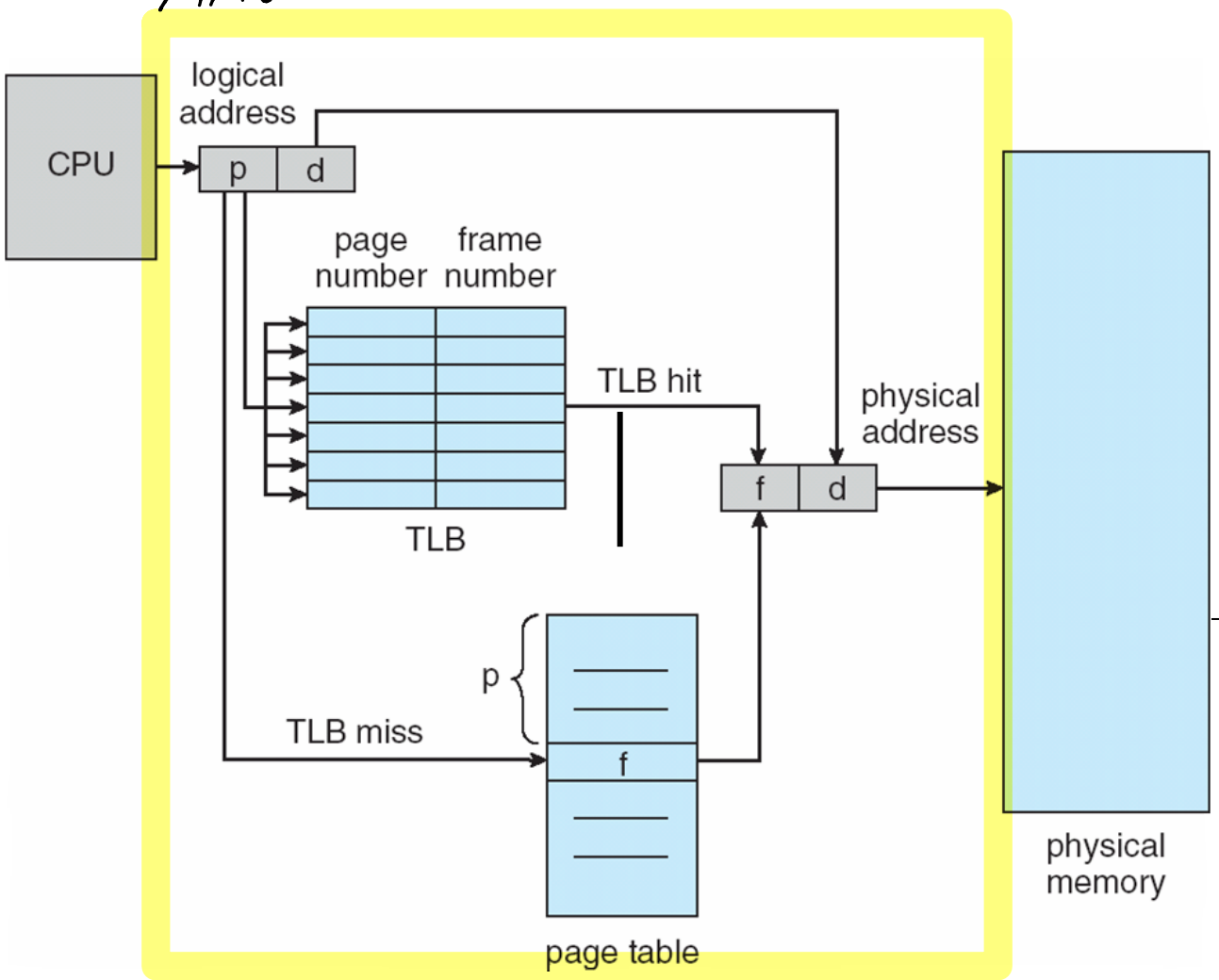
- Two types of memory
  1. General memory: (ex: DRAM)
     - give address -- return record
  2. Associative memory: (ex: Phone book)
     - give field of record -- return record
     - Very slow without parallel search (or indexing)
     - High cost for implementation

Page #	Frame #

- ***Associative Memory*** (TLB) – parallel search
  - Since only part of page table is in TLB **record**
- Address translation (p, d)
  - Try portion of page table in associative memory first
  - If p is in associative register, get frame # out
  - Otherwise, get frame # from page table in main memory
  - TLB is flushed (remove old entries) at context switching time

Paging Hardware with TLB

MMU



Effective Access Time

- Associative lookup = α time unit
- memory access time = β
- Hit ratio = ε (percentage found in the associative memory)
- Effective Access Time (EAT)

<hit> <miss>

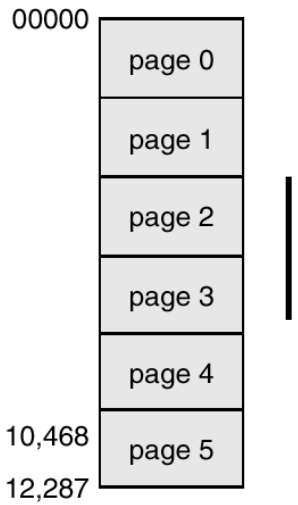
$$EAT = (\alpha + \beta)\varepsilon + (\alpha + 2\beta)(1 - \varepsilon)$$
$$= \alpha + (2 - \varepsilon)\beta$$

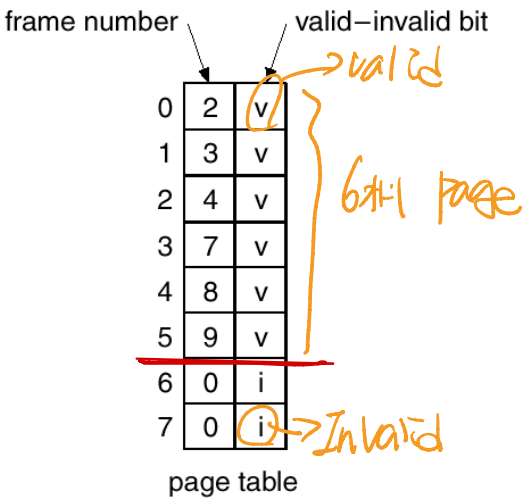
Memory Protection

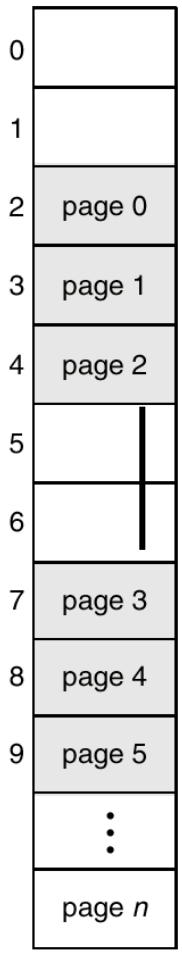
- Memory protection implemented by associating protection bit (valid-invalid bit) with each frame
  - PTLR can be used to test the validity of the address instead of the valid-invalid bit
  - However, we do not use PTLR for protection in modern operating systems due to some reasons that will be discussed in Virtual Memory
- *Valid-invalid bit* - each entry in the page table:
  - *"valid"* indicates the page is legal (a valid page)
    - The associated page is in the process' logical-address space
  - *"invalid"* indicates otherwise (access not allowed)

Valid (v) or Invalid (i) Bit in a Page Table

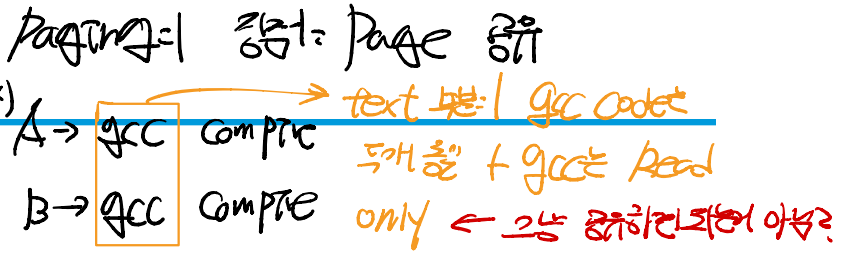
- Page size = 2 KB
- Uses only addresses 0 to 10468
- 6 (=10469/2048) pages are allocated to the process
- Only 6 entries are used in the page table
- PTLR can be used to test the validity of the address instead of the valid-invalid bit







Shared Pages



Program code 공유 가능

ex) gcc, VI, EMAX ...

- *Shared code*
  - One copy of read-only *(re-entrant)* code shared among processes (eg, text editors, compilers, window systems)
  - Shared code must appear in same location in the logical address space of all processes
    - To enable self-reference in the shared code

Paging시 어느 곳에서 부르든 어떤 → 시작점에서 offset거리로 갔. 공유하려면 시작점가동일 해야함. Contexts시 별개 Data 부분이 존재 → 문제

- *Private code and data*
  - Each process keeps a *separate copy* of the data
  - The pages for the private code and data can appear anywhere in the logical address space

Paging은 각 Section이 다른 위치에 존재 → 문제 없음

Shared Pages Example

Editor is shared (Editor consists of 3 pages -- ed1, ed2, ed3)