



Memory Management(1)

OS: | 메모리 관리 | 핵심: CPU가 명령어를 수행할 수 있도록 해주는 것

$PC \rightarrow Inst$

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 *Program: | 실행하기| DISK → memory / (Address Space)*
- Main memory and registers are the only storage CPU can access directly
- Register access in one CPU clock cycle (or less) *← one clock*
- Main memory can take many cycles *← 느림*
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

↳ [进程中] Memory Address / Kernel Address | ~~进程~~ 병합

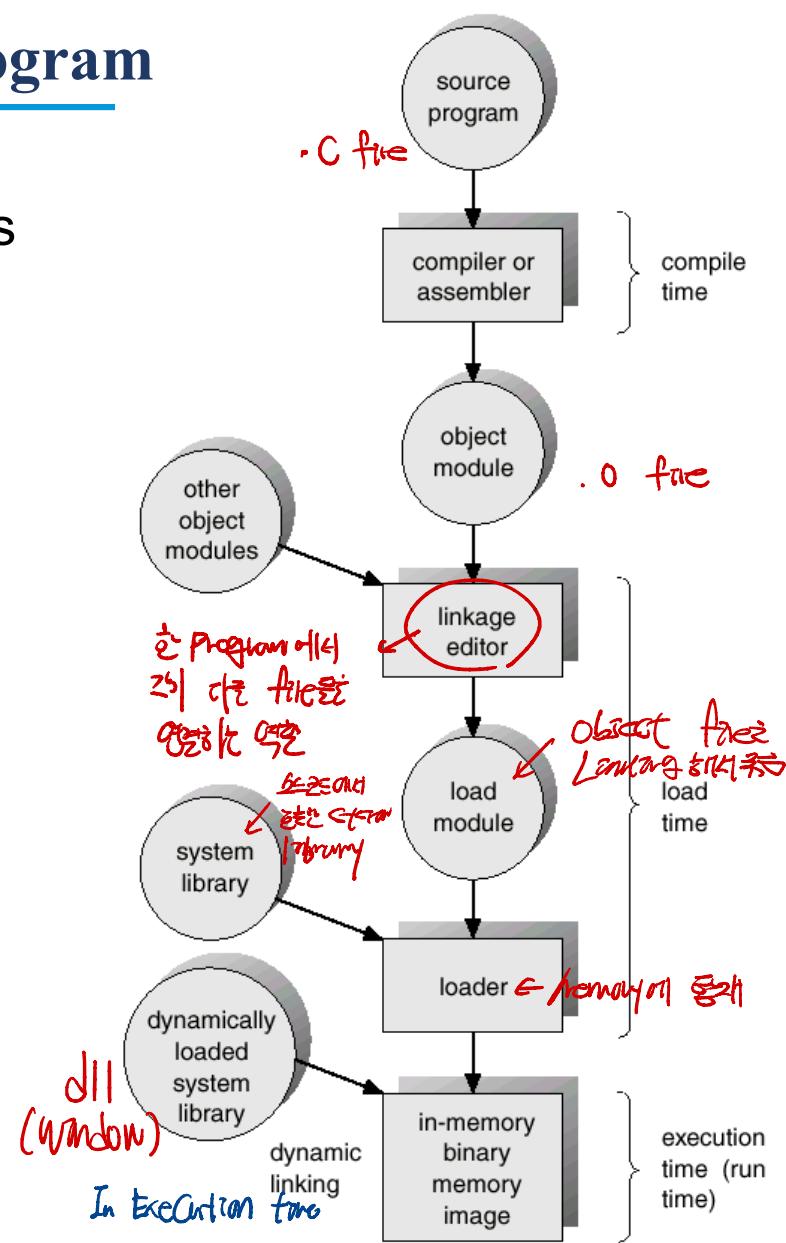
↳ System Call ~~접근~~

→ System Call Kernel Code | ~~접근~~ 가능



Multistep Processing of a User Program

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



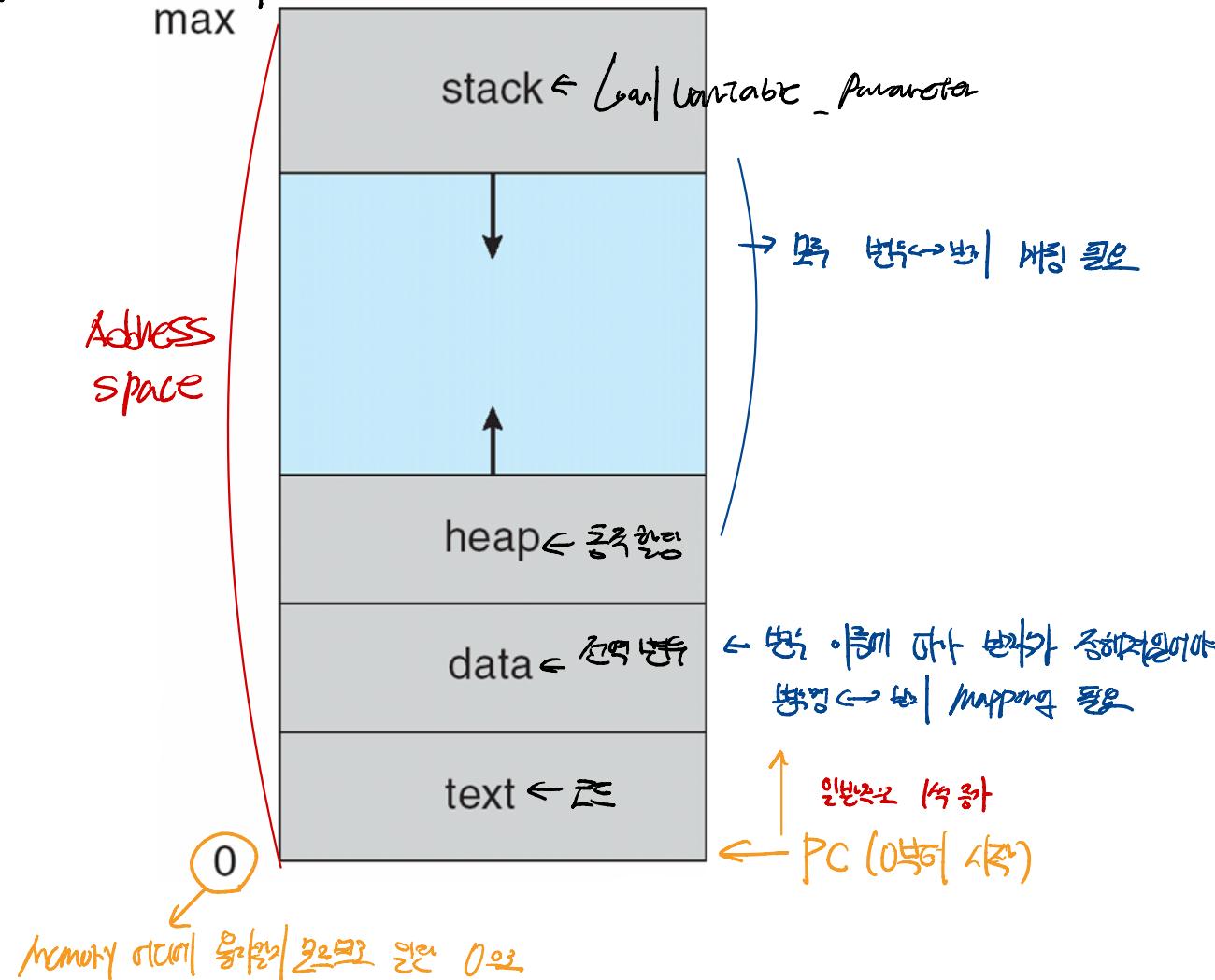


Process in Memory: Address Space

Program In Memory

max

Address
space



* memory → kernel

Binding of Instructions and Data to Memory

Bindung / Bindung의 차이점 / Bindung은 CPU는 복잡성을 줄인다

- **Compile time binding**

← 예전에 사용 X

- 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 (Flexibility ↓)

→ Multitasking의 시기 예전에는 시작하는 위치 → OS가 디바이스 허브 / User 및 Kernel 면접 → Recompiling

↳ Multitasking 시기 Kernel / User 예전에는, User 프로그램 하드웨어 program

- **Load time binding**

- Loader assigns absolute address to each symbol

- Compiler generates relocatable code containing relative addresses

→ Recompiling 하지 않음 ↳ program code가 0번주소 사용하는 위치, 시작점 offset (값)

- **Execution time binding**

- Used when process moves its in-memory location during execution

- Whenever CPU generates address, binding is required (address mapping table)

Mapping table (속대주소 → 절대주소) • | 틈유형

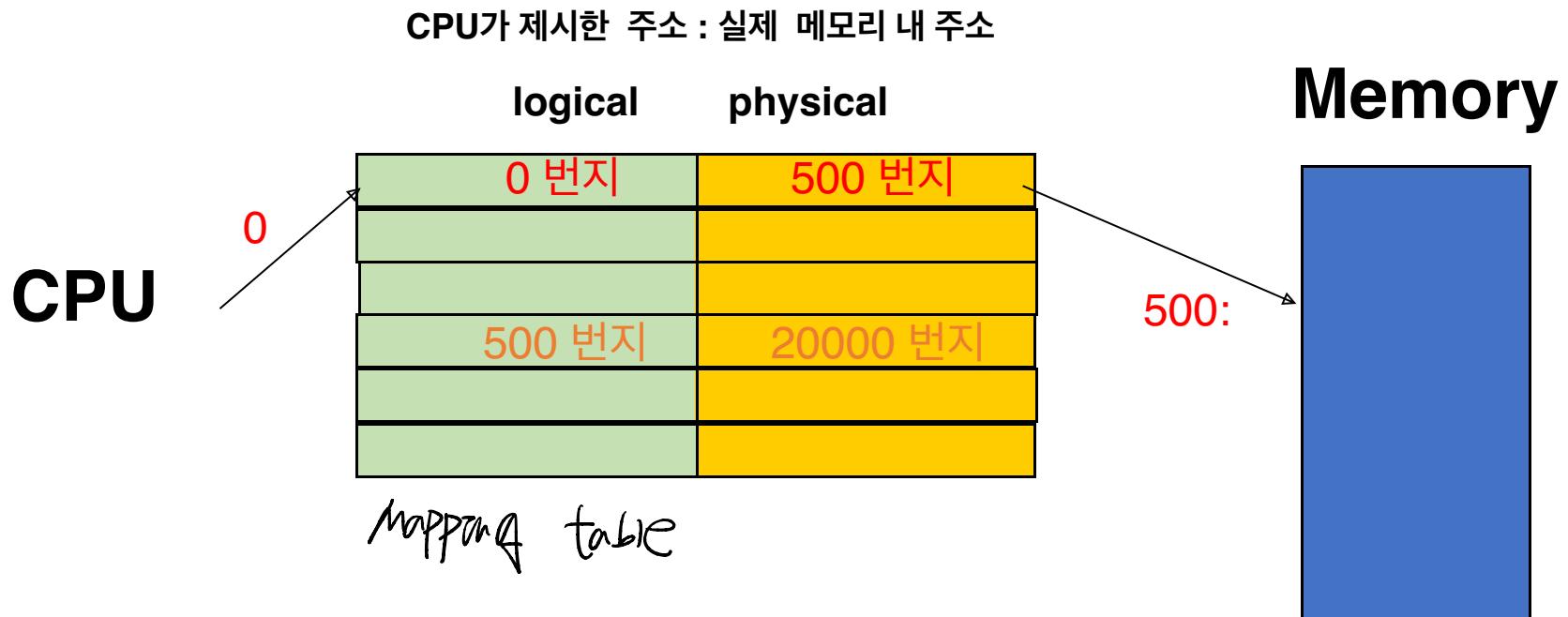
→ Mapping을 지원하는 Unit

- need hardware support (e.g., base and limit registers, **MMU**)

→ 실행 중 Memory에서 이를(기반주소)로부터 (속대주소는 빼기로, Memory 주소 + offset) 계산해 OS가 번호하는 값을



Address Mapping Table





Base and Limit Registers

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

Multitasking에서 각 Process가 동일한 memory에 출입할 때 각각의 Process가 다른 주소,

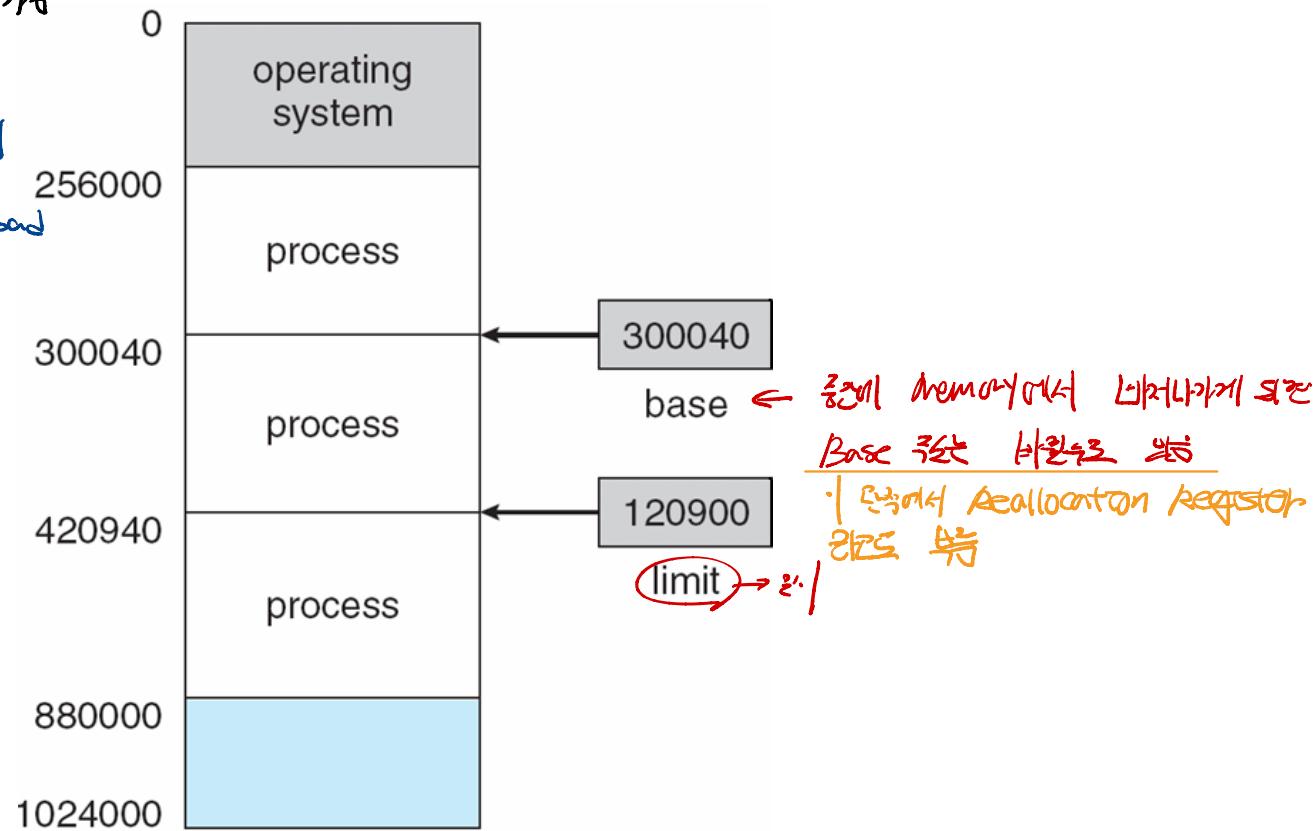
즉 다른 블록을 사용하는 경우

PCB에 저장되는 번역 테이블

Process | 실행할 때, 해당 번역

등록 | CPU의 Registers load

Base, Limit Register





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 *CPU가 인식한 주소체계*

- Also referred to as **virtual address**

- Physical address** *← DRAM: 실제 주소*

- Address seen by the memory unit

왜 봉지?

*CPU가 Physical addresses를 보통해
Logical addresses로 쓰는.
간접주소와 같이
주소를 확장하는.*

*Logical address → physical
address mapping을 OS가 해!*

- Logical and physical addresses are the same** in compile-time and load-time address-binding schemes

Memory에 할당된 때 실제 주소는 Logical = Physical *logical = physical (Compile time binding)*

- Logical (virtual) and physical addresses differ in** execution-time address-binding scheme

실행에 사용

logical = physical (Compile time binding)

Logical ≠ Physical

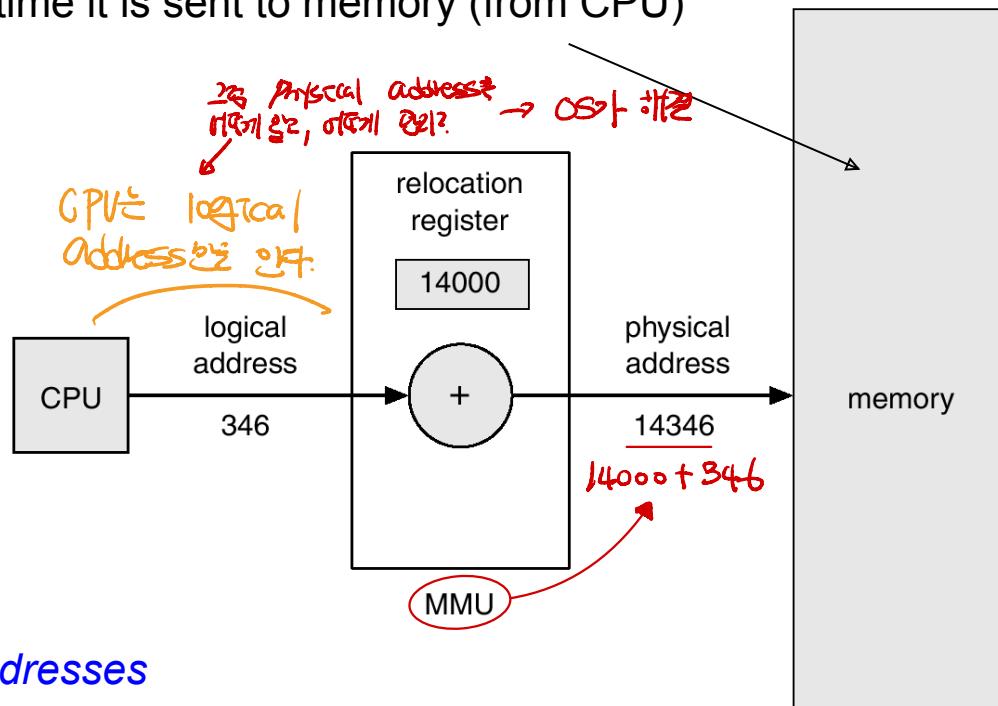
*↑
MMU가 mapping*



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)

Memory-এ দুটির মধ্যে, Base Register (memory এর অবস্থা) হলো Logical address কে পরিষ্কার Physical address করে দেয়।



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



Swapping necessitates dynamic relocation

- **Swapping** ← Mid term schedule

- A process can be **swapped temporarily** out of memory to a **backing store** and then brought back into memory sometime later

- **Backing store**

Suspend 대기된 Process가 메모리에 유통해 memory 사용률이
작은 편이다. ← Execution Bandwidth swapping 기법이 있다.
(Dynamic Allocation 기법)

Swap out된 Process가 swap된 메모리에 유통해 memory 사용률이
작은 편이다. ← Execution Bandwidth swapping 기법이 있다.
(Dynamic Allocation 기법)

- Fast disk large enough to accommodate all memory images for all users

Swap out된 Process가 swap된 메모리에 유통해 memory 사용률이
작은 편이다. ← Execution Bandwidth swapping 기법이 있다.
(Dynamic Allocation 기법)

- Must provide direct access to these memory images

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

설명

1. MMU는 단위 단위 → 빠른 단위 단위 → 처리(처리)하기 편한지?

2. Process가 memory에서 처리(처리) 단위 단위 → ← . | 처리(처리)

File system

→ 파일을 그룹화하기

→ 연속화 X

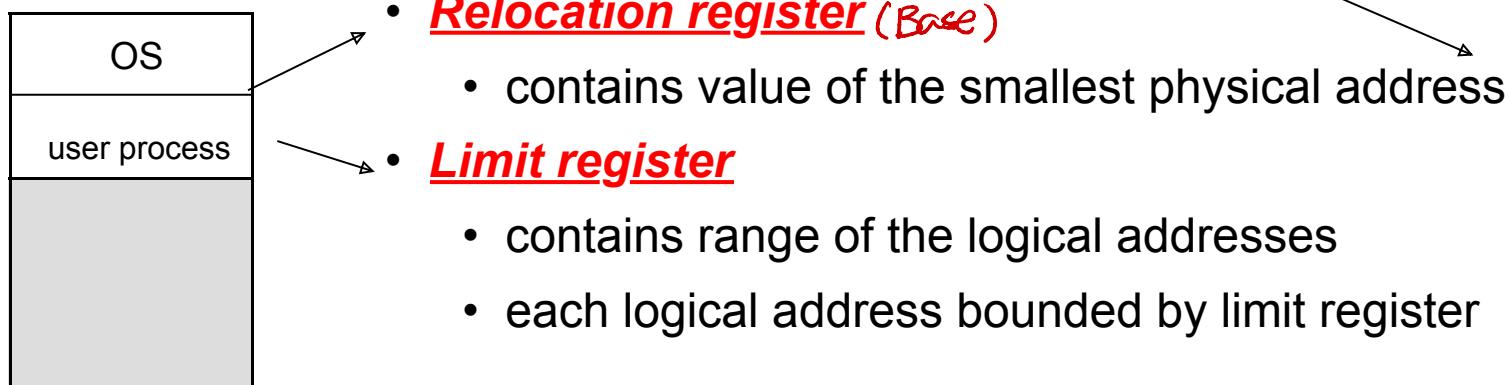
→ Swap partition
File system은 투명화하기



Contiguous Allocation: ~~process DRAM~~ ~~contiguous~~ ~~memory~~

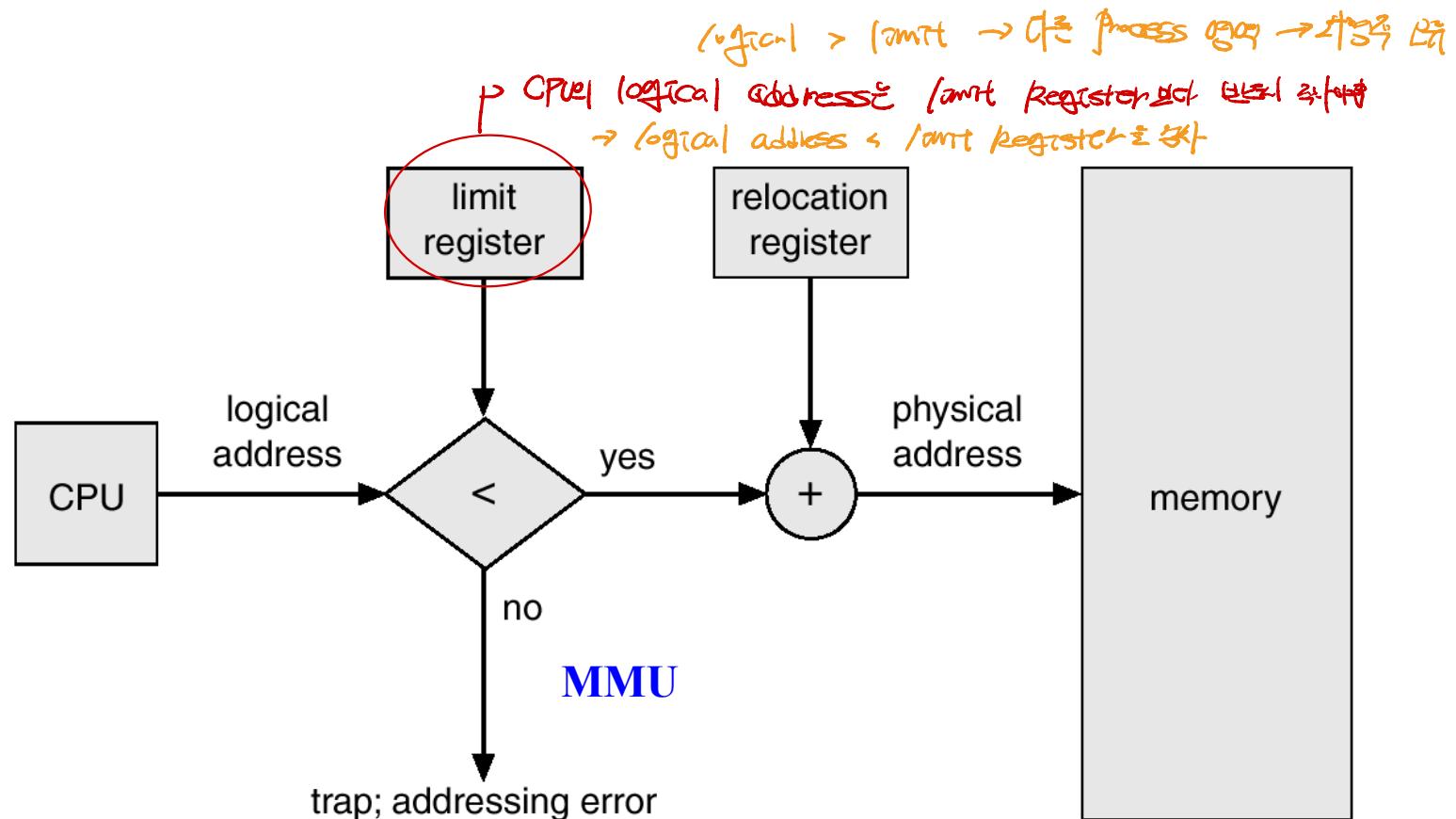
Memory Mapping or KA: ~~process~~ address space memory on ~~fixed~~ id.

- Main memory is usually divided into two partitions:
 - *Resident operating system*, usually held in *low memory* with interrupt vector \leftarrow kernel 주소
 - *User processes* then held in *high memory*
 - To *protect* user processes from each other and OS





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 시스템 메모리 관리 프로그램에 빈 공간 생성 및 관리.

Hole - 크기도 다 다음

& process size

Hole는 OS가 관리해야함

→ 사용할 때마다 할당과 해제

→ 어느 hole이 할당되었는지

hole의 크기는

(hole의 크기)



Dynamic Storage-Allocation Problem

Hole 흑자 / 여과자

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

- **First-fit**

← 첫번째 흑자

- Allocate the **first hole** that is big enough

- **Best-fit**

← . | process size가 가장 작은 크기의 흑자에 할당

- 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** ← 가장 큰 흑자에 할당
- must also search entire list
- Produces the largest leftover hole

进程 시작하는 process에 맞는 흑자에 할당
작은 흑자에 할당되는 process는 실행하기에
실행되지 못할 수 있다.
→ 최악의 옵션(선택지)/최선/최악

Best fit은 흑자에 적합한
작은 흑자로 할당된다
실행되는 흑자에 따라 결과가 달라지며
Worst fit은 가장 큰 흑자로
할당되는 흑자로 할당된다.
→ 어느 방법이 낫고 나쁨은 알수가 없다

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



Fragmentation : 높은 용량 부족

- External fragmentation ← Best fit 방식 / 훈련된 예상과 다른 차이로 발생하는 것
• Total memory space exists to satisfy a request, but it is not contiguous
 - Internal fragmentation ← 훈련된 예상과 실제 사이즈 차이 예 : EX) 1000 size의 1024size 할당
• 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
 - Heavy operation: Basic address translation
→ 사용자 영역과 물리적 영역 간의 Memory Copy 필요
 - Compaction is possible only if relocation is dynamic, and is done at execution time

가정학

action → ~~External state like~~ fragmentation

→ 사용자에게 보여줄 정보를 **Memory**에 COPY 한다.

→ External fragmentation = չօրէ
չօպէ



Paging : External Fragmentation 문제 해결 하는 방법

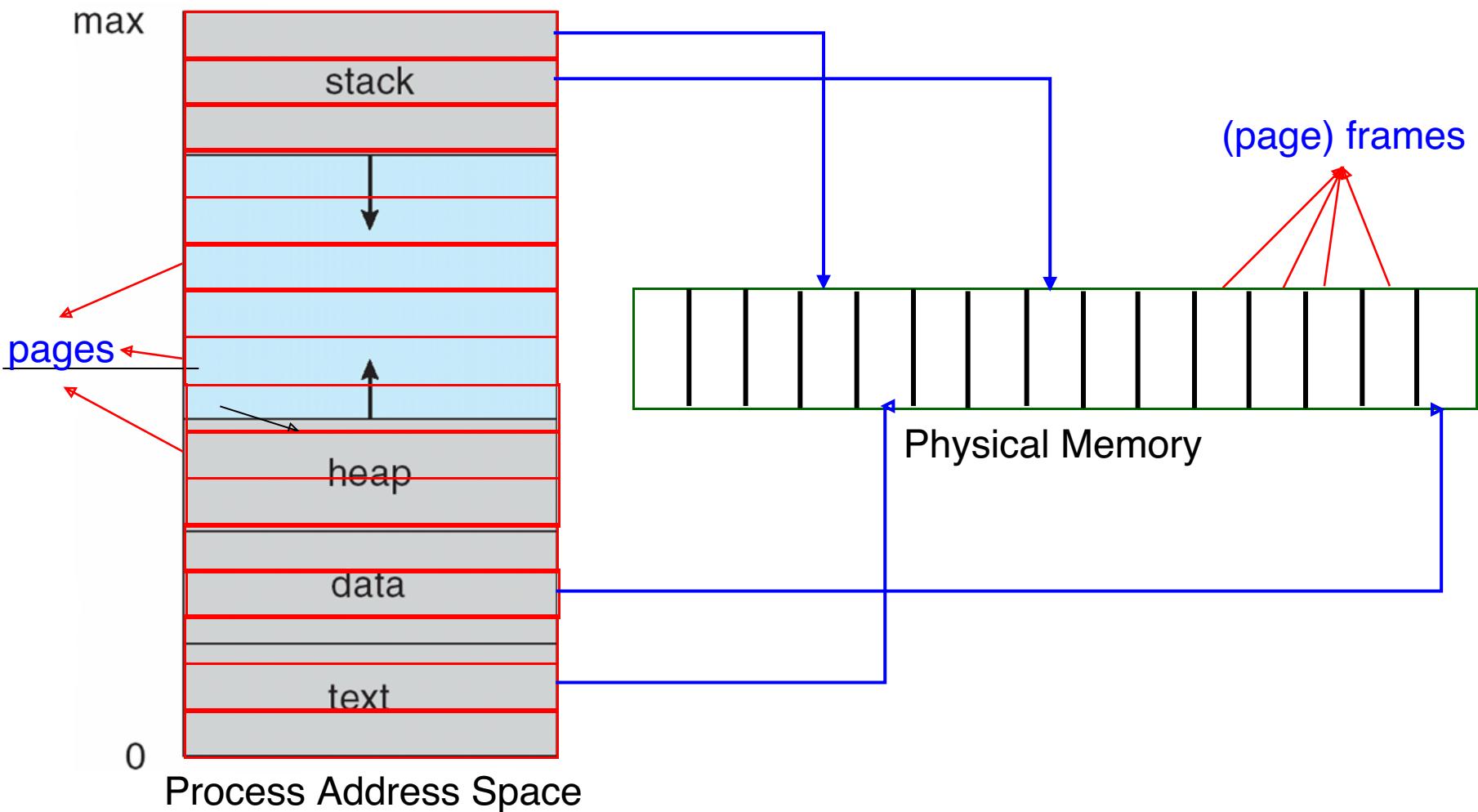
• Paging

- Is a scheme that **permits address space to be noncontiguous**
리소스는 memory 공간이 Address Space를 까지하는다. Memory와 Address Space 모두 같은 Size로 만들고 까지면 → ··· Address Space는 까지기 어렵다
- Basic Method ~~frame~~ page를 ~~fixed~~. The frame.
~~비교적 작은 크기~~
frame size = page size
- Divide physical memory into **fixed-sized blocks** called **frames** (size is power of 2, between 512 bytes and 8 MB) DRAM의 일정한 크기가 2의 배수(512B ~ 4MB)
- Divide logical memory into **blocks of same size** called **pages** 같은 크기가 2의 배수
- Keep track of all free frames
- To run a program of size **n** pages, need to find **n** free frames and load program ↗ Page가 비연속적이며 그에 맞는 Page가 필요하니 어디에 놓을지 찾기
- Set up a **page table** to translate logical to physical addresses
- **Internal fragmentation**, but no **external fragmentation**

즉) Memory에서 모든 공간은 Frame 단위) 대로 쓸 수 있음
→ External Fragmentation 발생

그러나 Internal Fragmentation 발생 ← 아직 Page 단위만 발생 / 성능 차이는 없음
Ex) 4KB를 4KB로 Gt, Program 1KB를 4KB로 Frame(4KB)에 넣는

Paging



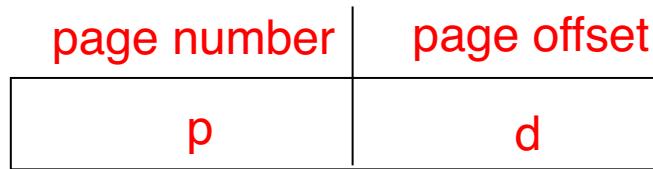


Address Translation Scheme in Paging

- 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

Paging 사용한 frame 번호는 Base Reg에 있는



→ page 2의 주소는

Address Space : 2^m

→ 1 bit

Page size : 2^n

→ 1 bit

2^n 주소 | Page | 2^{m-n} 주소

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

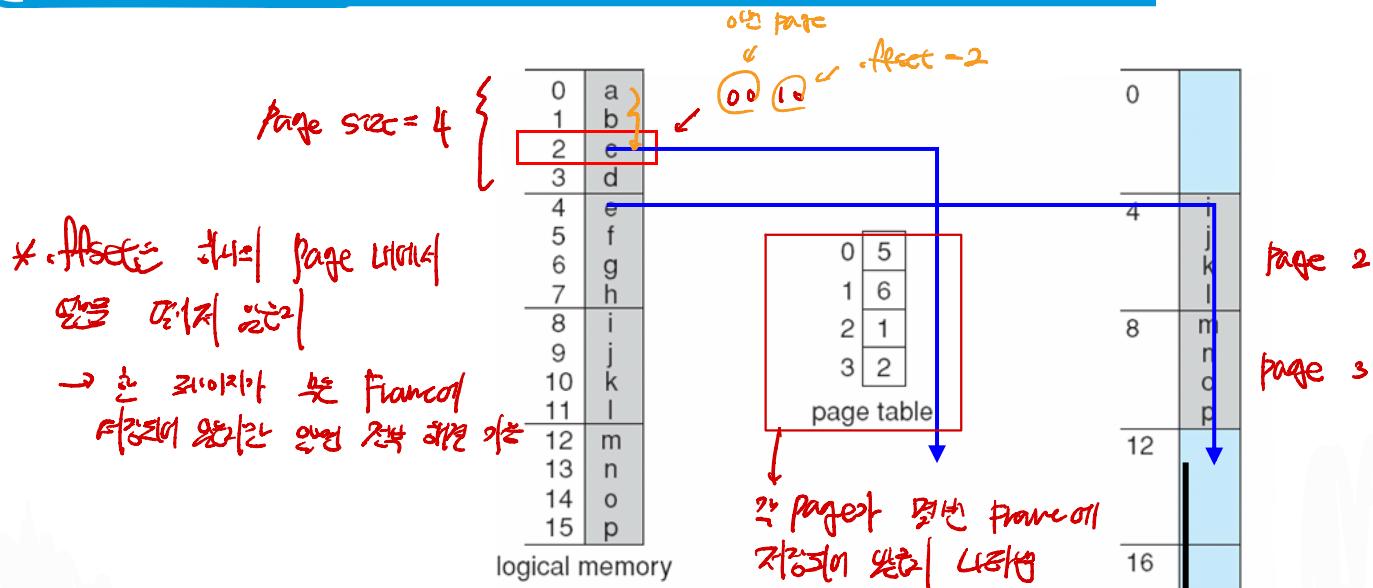
n bits

한 페이지당 n bit

즉 총 n은 2^m 의 (m-n) bit



Paging Example



$$\text{Page Number} = 2/4 = 0$$

$$\text{Offset} = 2 \% 4 = 2$$

Logical address (m): 4 bits ← 16-bit address space

Page number (m-n): 2 bits ← 16/4 → 4 (#page)

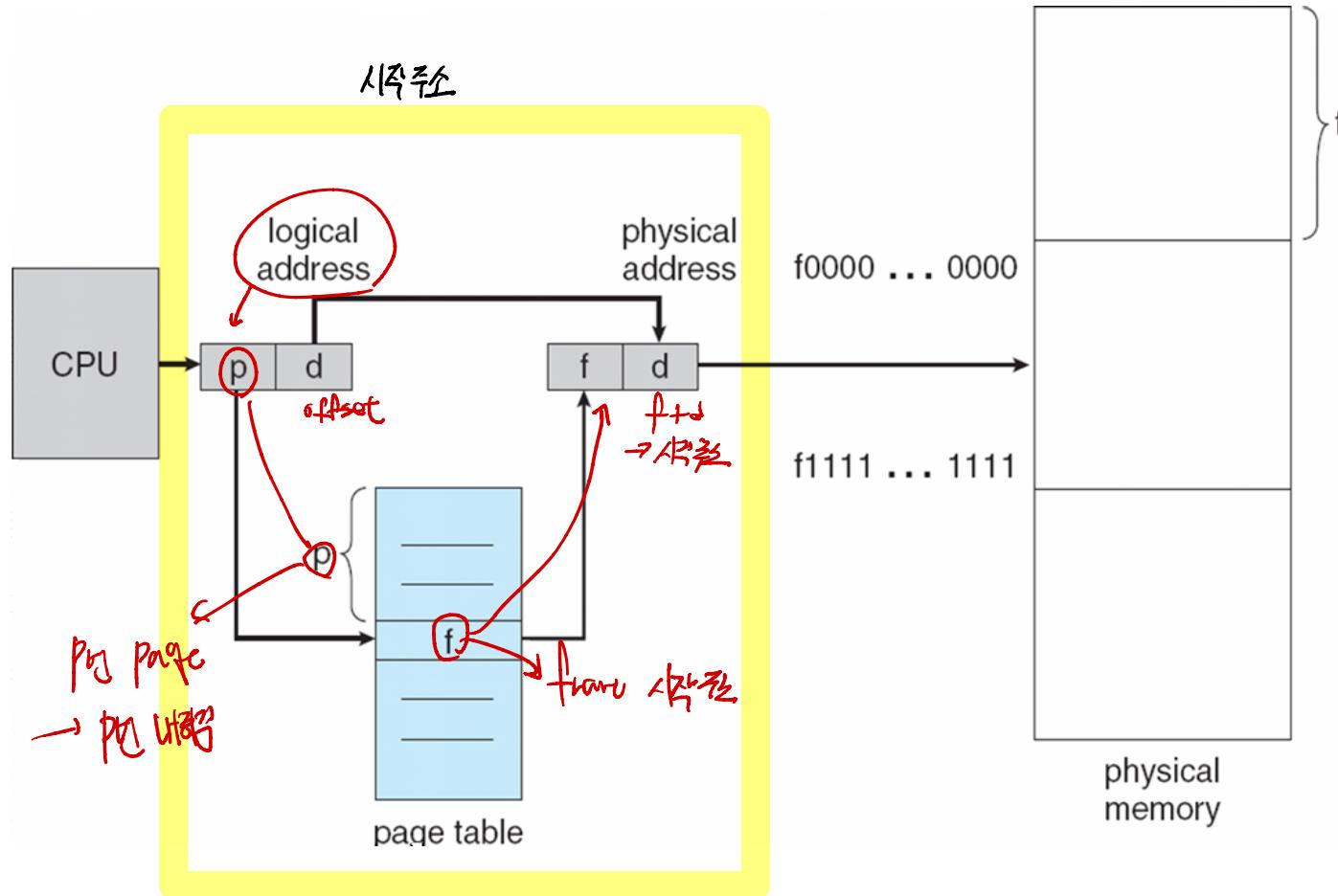
Offset (n): 2 bits

Page size = 4

4-byte page n = 2 bits

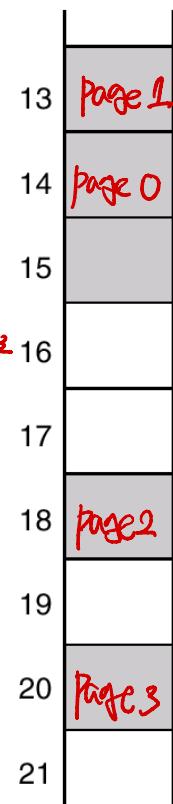
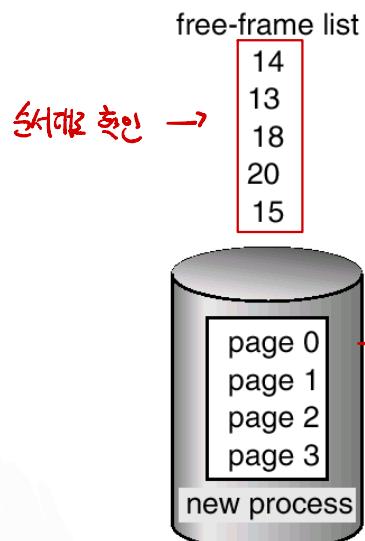


Address Translation Architecture

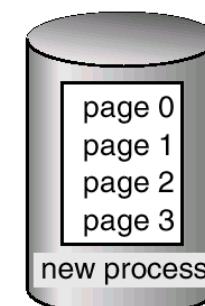


Free Frames

free-frame 놓쳤을까?
→ Virtual memory

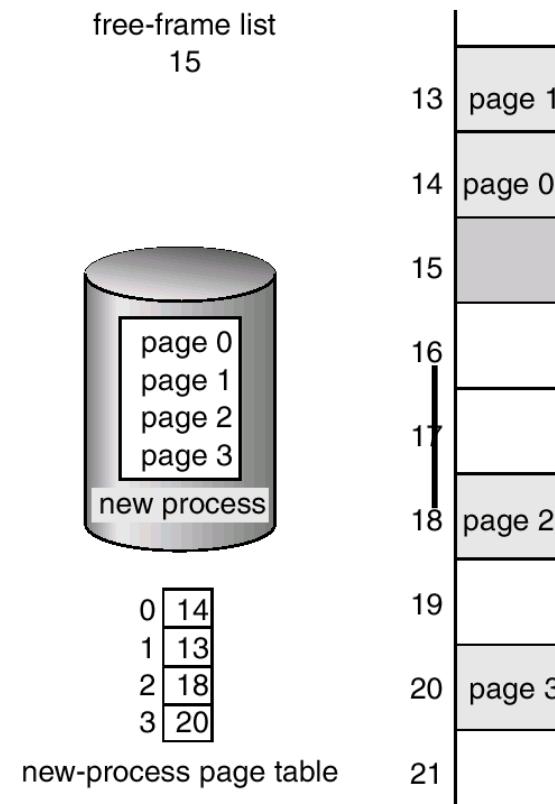


free-frame list
15



0	14
1	13
2	18
3	20

new-process page table



(a)

Before allocation

(b)

After allocation



Implementation of Page Table

- Page table is kept in main memory
Page는 OS가 관리하는 틀
Page-table base register (PTBR) points to the page table
Page-table length register (PTLR) indicates size of the page table
32bit page number
4KB pages
- Problem: Every memory (data/instruction) access requires two memory accesses
예: CPU가 데이터/인струк션을 찾을 때 메모리에 두 번이나 접근
memory access overhead ↑ → 3x slower
1. Page table
2. 실제 Data, Inst
- Solution: A special fast-lookup hardware cache called Translation Look-aside Buffer (TLB) or Associative memory is used
Memory에 있는 Page table의 주소를 찾는 데서 TLB(캐시)를 사용
 - 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
 1. 캐시에 찾기
 2. TLB에 있는 실제 메모리 주소

TLB Size ↑ → Hit ↑↑
어디로? → Locality of reference (한 페이지 → 많은 주소)

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

→ 모든 단위 단위로 + matching 과정을 통해 찾는 money

■ **Associative Memory** (TLB) – parallel search

- Since only part of page table is in TLB

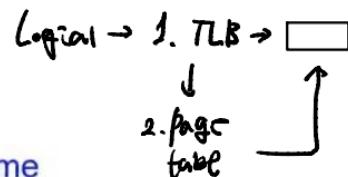
record

■ 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

→ Process of (flush page table) happens

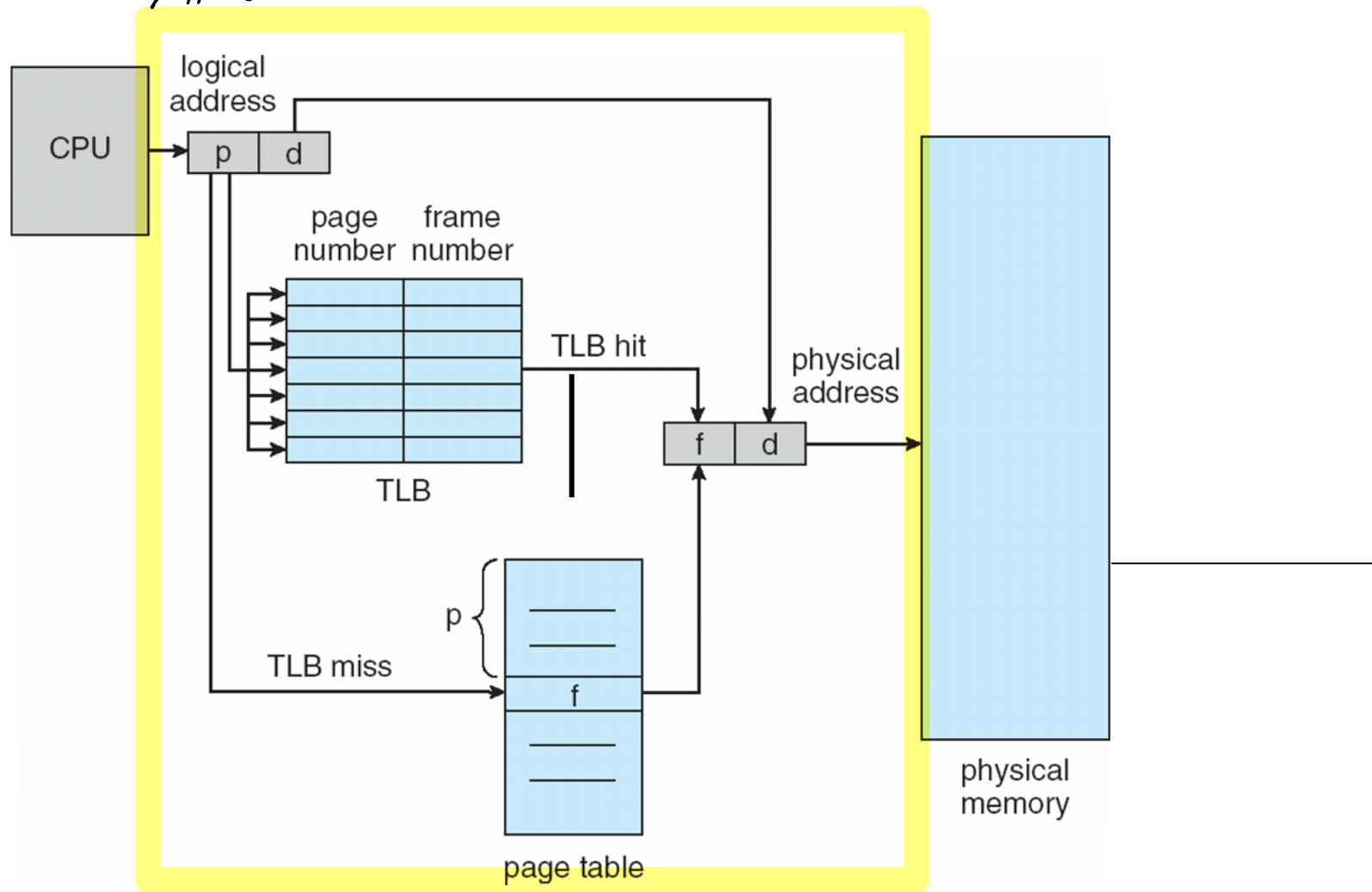
→ After process is done





Paging Hardware with TLB

MMU



TLB 히트

Effective Access Time : CPU가 Memory에 접근하기 위해 First, Direct \rightarrow 2단계

- Associative lookup = α time unit
- memory access time = β $\alpha < \beta$
- Hit ratio = ε (percentage found in the associative memory)
 $(0 \sim 1)$
- Effective Access Time (EAT)

$$\begin{array}{c}
 \text{---} \\
 \text{<hit>} \quad \text{<miss>} \quad \text{---} \\
 \text{EAT} = (\alpha + \beta)\varepsilon + (\alpha + 2\beta)(1 - \varepsilon) \\
 = \alpha + (2 - \varepsilon)\beta
 \end{array}$$

↳ Memory 2단계 (page table 찾기)

$\alpha + (2 - \varepsilon)\beta$ vs β ← Paging은 2단계로 이루어짐

1. $\varepsilon = 1 \rightarrow \alpha + \beta$ vs β
 α 는 정합/맞는 → $\alpha + \beta \approx \beta$ Paging 1단계 때: $\varepsilon = 1$ → Page Hit↑↑

2. $\varepsilon = 0 \rightarrow \alpha + 2\beta$ vs β

$\alpha + 2\beta \gg \beta$

Memory Protection

- Memory protection implemented by associating protection bit (valid, invalid bit) with each frame $1/17 \rightarrow 22$ ok $2021.22.2.55$

Valid → 220K

卷之六

- 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:

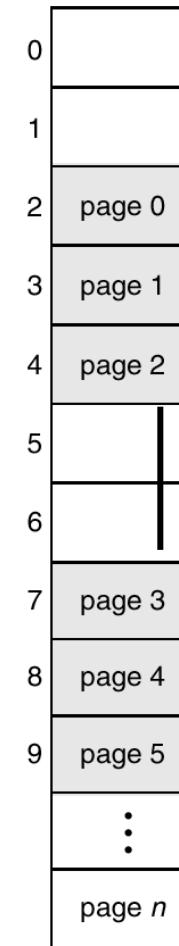
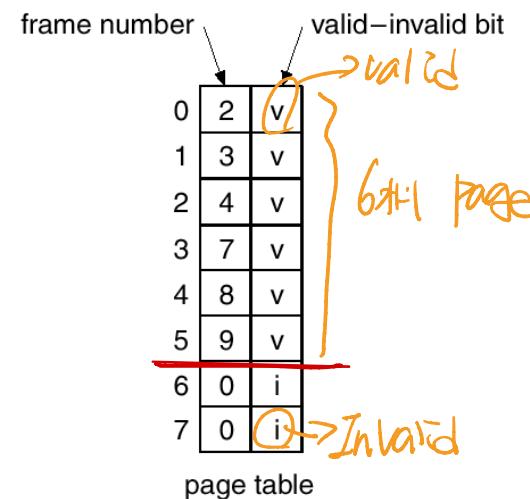
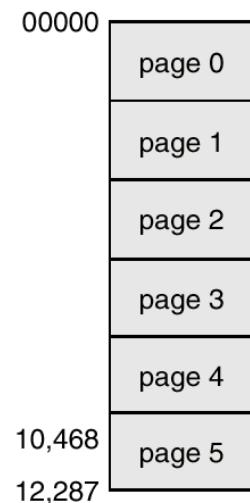
- “*valid*” indicates the page is legal (a valid page) → ~~Valid-Invalid~~ *Valid-Invalid*
 - The associated page is in the process’ logical-address space *if*
 - “*invalid*” indicates otherwise (access not allowed)





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



와 Page table! Address Space 봐여?

- Address Space: $0 \sim MAX$

• ~~정수~~ 가능한 MAX

→ OS: 32비트 (32bit)

→ $0 \sim 2^{32}-1 \leftarrow 4GB$

↳ Stack, Heap이 높은 번역

→ Page table은 높은 번역 만드어야함. ↳ 2³²

→ MAX가 몇개? $MAX \rightarrow$ Stack, Heap 번역 ↳ 2³² 번역

Page tables이 2³² 번역 만들어야 함

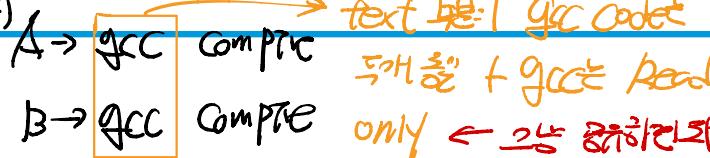
→ 2³² 번역 Stack, Heap이 번역에 따라 다른 경우

→ Page tables 번역 문제인가? Valid/Invalid bit



Shared Pages

Paging : Page 틀이



Program Code 틀 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

Paging : 같은 주소에서 부호화된 아ドレス

→ 128 쪽으로서 offset → 2:2
 예. 같은 파일의 A+320+640
 주소는 .Contanans-1 틀
 Data 틀과 같이 → 640

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

Paging : 각 Section .1

C 틀 가변적 틀

→ 풀 가능



Shared Pages Example

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

