8. 메모리 관리 Memory Management

메모리 관리 기법:

연속 메모리 할당	한 process 전체를 **연속적인** 주소 공간에 적재함.
페이징 Paging	• 하나의 process를 **물리적 단위 (Page)**로 분할하고, • 각 page를 임의 위치에 적재함.
세그멘테이션 Segmentation	• 하나의 process를 **논리적 단위 (Segment)**로 분할하고, • 각 segment 임의 위치에 적재함.

1. 개요

❑ 주기억장치 관리 Memory management

- 프로그램이 실행되려면 주기억장치에 적재되어야 함.

  (NOTE) CPU는 main memory와 CPU 내의 register만을 직접 접근함.
  (NOTE) Memory hierarchy: Registers—Cache memory—Main memory—보조기억장치.

- 가능한 한 많은 프로세스를 적재하는 효율적인 메모리 할당 방법이 요구됨.

  - 너무 적은 수의 프로세스는 CPU를 유휴(*idle*) 상태로 만들 수 있음.
  - 너무 많은 수의 프로세스는 빈번한 *Swapping*을 야기할 수 있음.

- 메모리 관리는 운영체제(*Memory Manager*)에 의해 수행되며,
  이때 hardware의 지원이 필요함. – MMU(Memory Management Unit)

❑ Swapping

- **Swap-in, Swap-out**

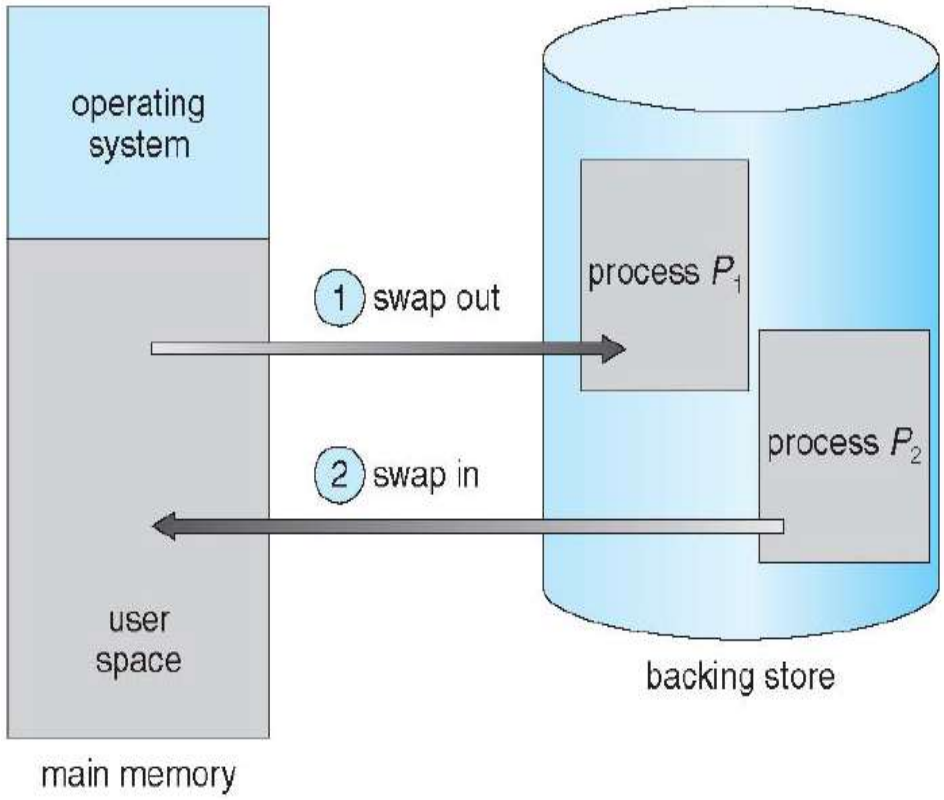
  A process can be
  ① swapped out of memory to *Backing Store*
  ② brought back into *memory*
  for continued execution.

- *victim process*의 선정 방법

- 실행시간 주소 바인딩이 요구됨.

- 비싼 context switching 비용이 요구됨.

  변형 swapping (예) Unix, Windows
  메모리 할당이 임계치(threshold)를
  초과할 동안 만 swapping 수행함.

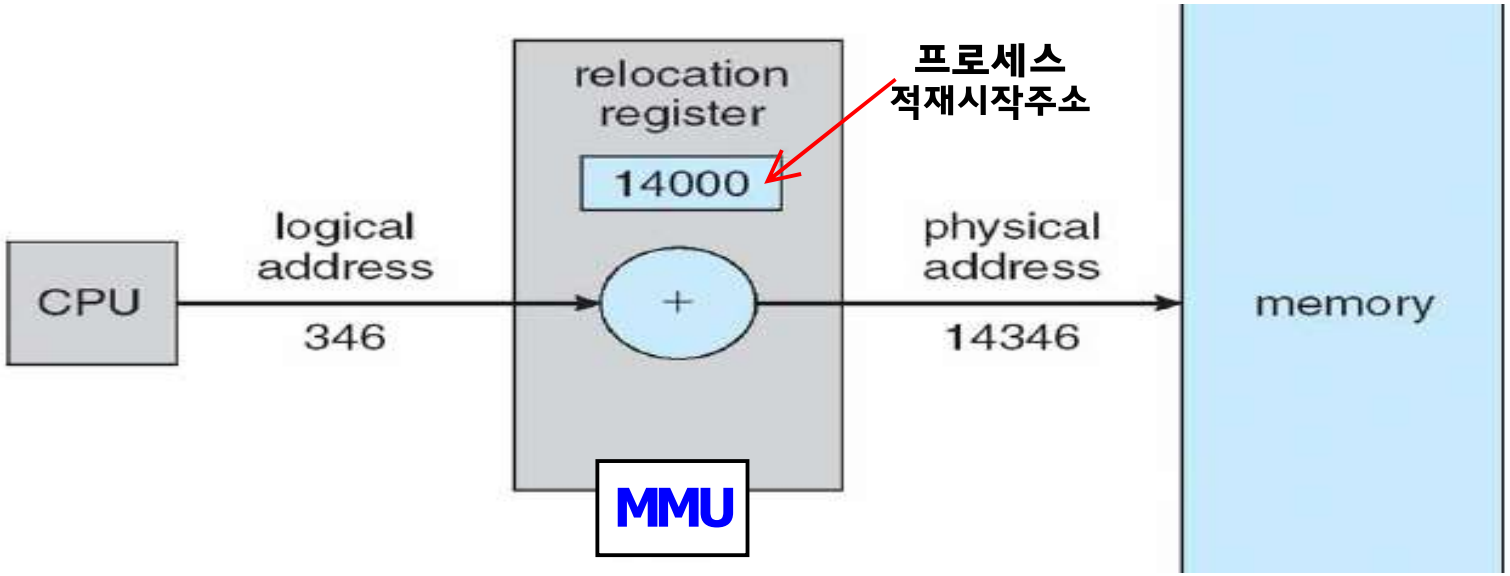


❑ 기억장치관리 요구 사항

① 재배치Relocation를 지원해야 함

- 프로그램은 (최초 적재, swapping 時) **임의** 위치에 적재 가능해야 하며,
  재배치 가능한 적재 모듈 ***Relocatable load module*** with **Relative Addresses**.

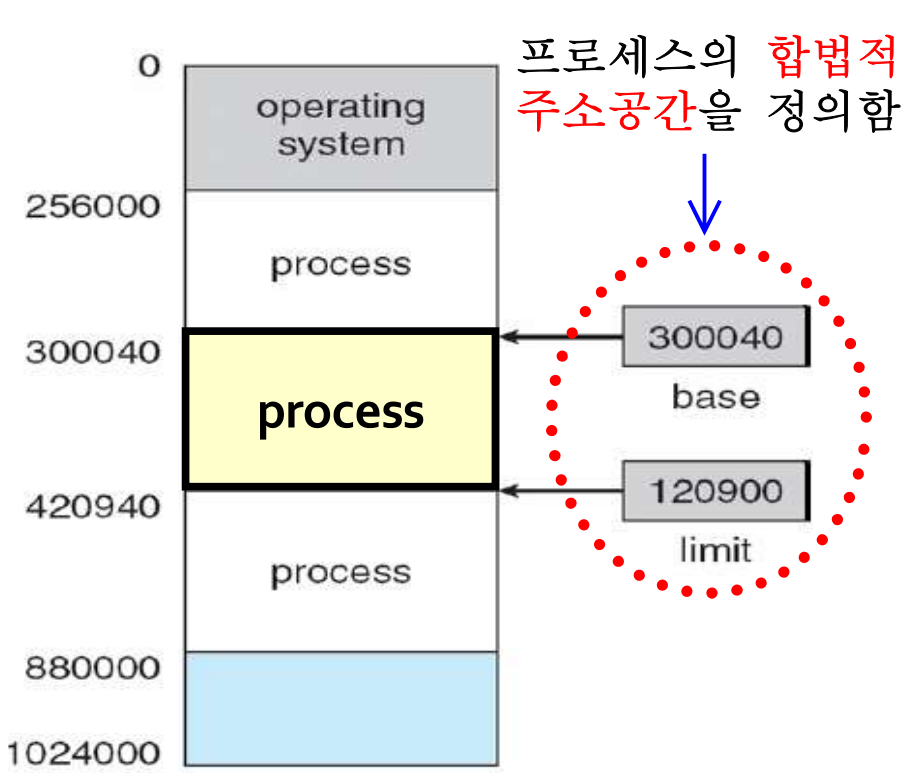
- 동적 주소 바인딩이 요구됨. Address Binding: 논리주소 → 물리주소.
  - By **Relocating loader** at initial load-time.
  - By **Memory Management Unit(MMU)** at execution-time.

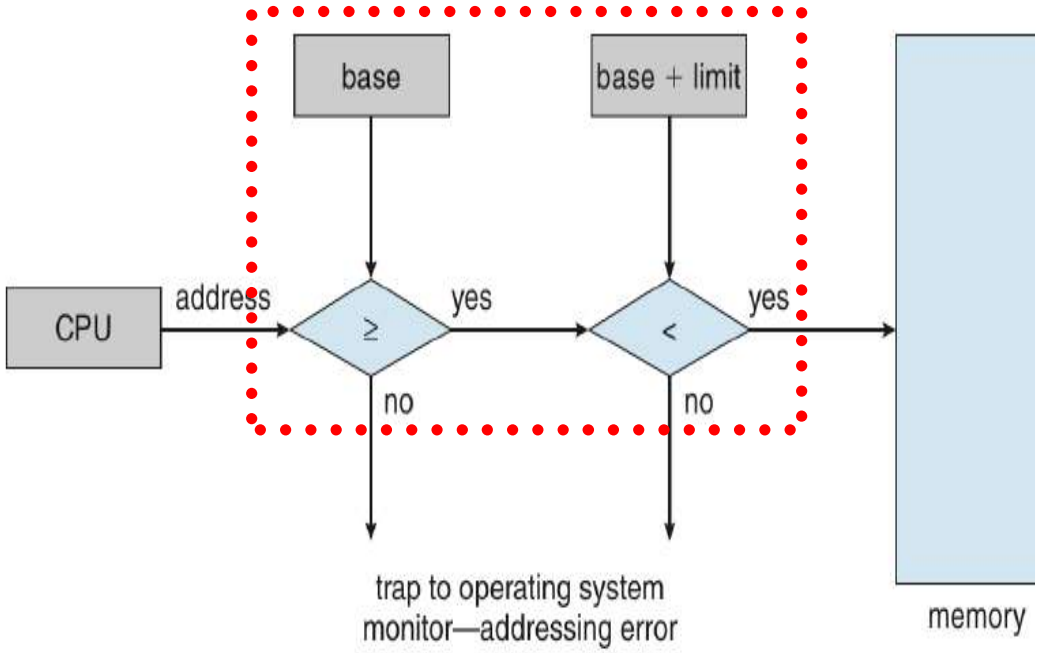


<**Dynamic relocation** using Relocation Register>

② 보호Protection를 지원해야 함

Process는 오직 **자신의 주소공간** *Address Space*만 접근해야 함. ✶ process들은 기본적으로 상호 독립적.	주소검사는 실행 中 **H/W**에 의해 수행됨. ✶ 프로그램은 실행 중 재배치 가능하므로 컴파일 時 불법주소 탐지는 불가능.





Hardware Address Protection with Base and Limit Registers

③ **공유Sharing를 지원해야 함**

- 공유 자원(shared data, shared library) 접근이 가능해야 함.

④ **프로그램의 논리적 구조를 지원해야 함**

- 하나의 프로그램은 여러 개의 **모듈*module***로 구성됨.
  (예) code module, library routine, data module

- 운영체제(와 하드웨어)는 **프로그램 모듈**을 지원해야 함:
  모듈 유형에 따라 다른 보호 기법, 공유 등의 지원.

- **Segmentation** 기법이 프로그램 모듈을 잘 지원함.

⑤ **기억장치의 물리적 구조를 지원해야 함**

- **Memory Hierarchy:** 주기억장치—캐시메모리—보조기억장치.

- **주기억장치와 보조기억장치 間 정보**(process, page, segment) **이동을 관리**해야 함.
  (메모리 관리의 핵심)

2. 연속 메모리 할당 Contiguous Memory Allocation

하나의 프로세스를 연속적인 주소 공간에 적재하는 메모리 관리 기법.

논리 주소

프로그램 시작 주소-보통 0	변위

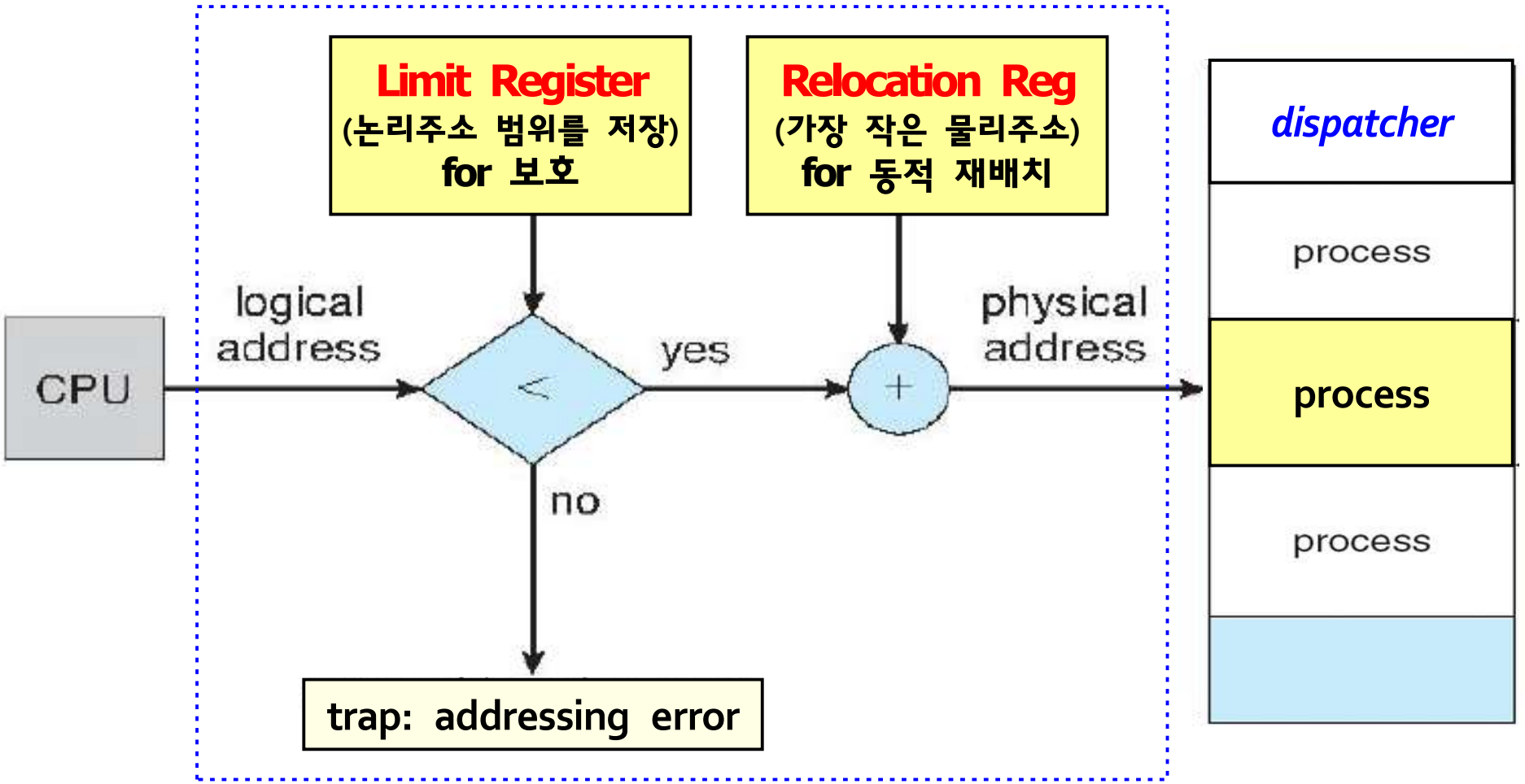
※ 논리주소) 프로그램 내의 주소

물리 주소

적재 모듈 시작 주소	변위

※ 물리주소) 주기억장치 주소

❑ 메모리 보호/ 재배치 by **Memory Management Unit** (MMU)



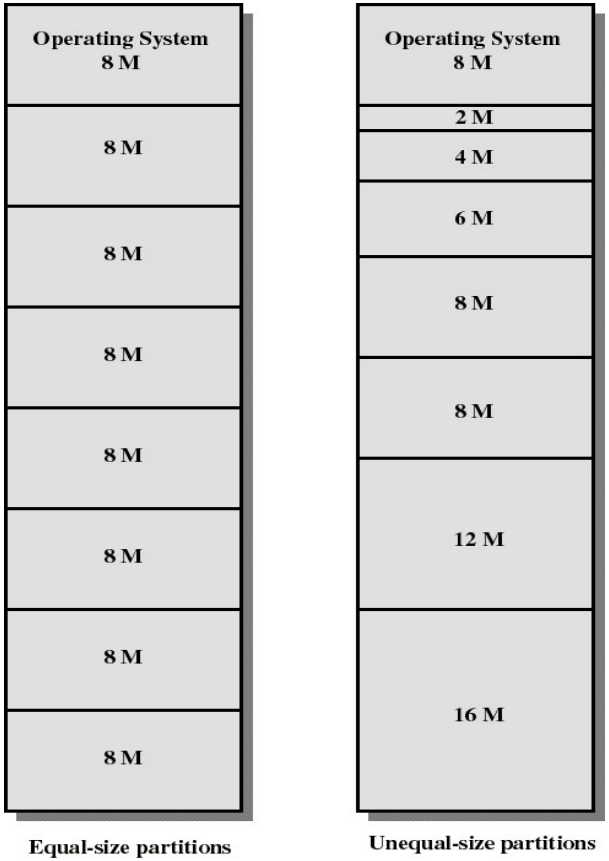
(note) *CPU Scheduling* 時 ***Dispatcher***는 context switching을 수행하며,
이때 두 register의 값을 설정함.

❑ 연속 메모리 할당 기법: 고정, 가변 분할 기법.

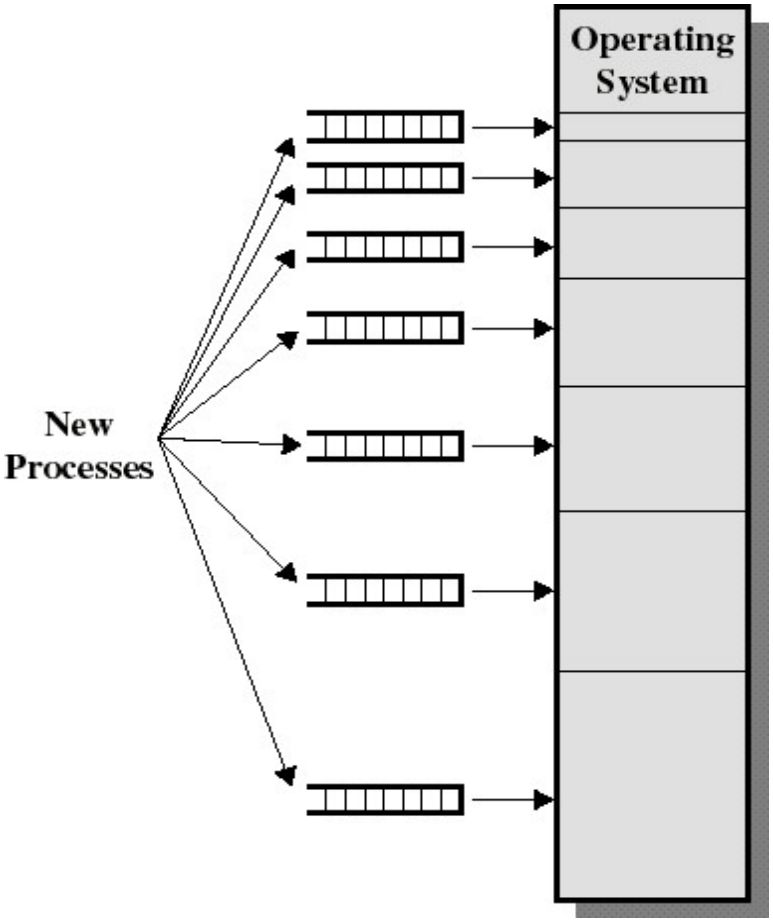
(1) 고정 분할 기법 Fixed Partitioning:

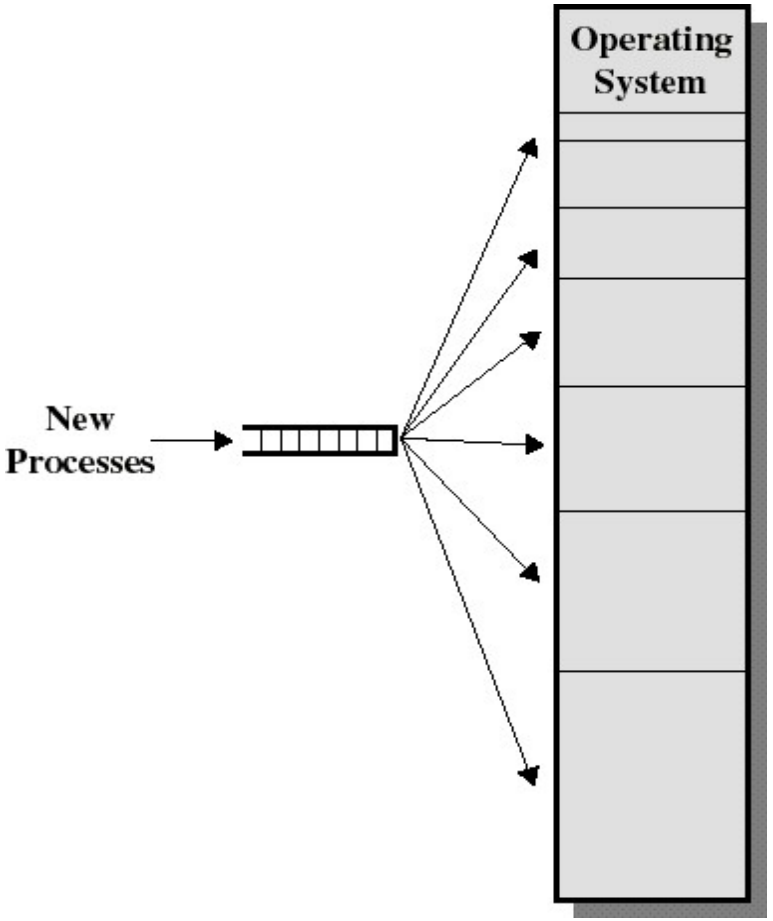
메모리를 여러 개의 **Partition**으로 나눔.
각 **partition**에 하나의 **Process**를 저장함.

- **Equal-size partitioning**
  - **First available** partition 선택.
- **Unequal-size partitioning**
  - **Best-fit** partition 선택.
- Partition의 개수
  = Multiprogramming Degree
- **내부 단편화** 발생함. ***Internal Fragmentation***



Unequal-size Fixed Partitioning





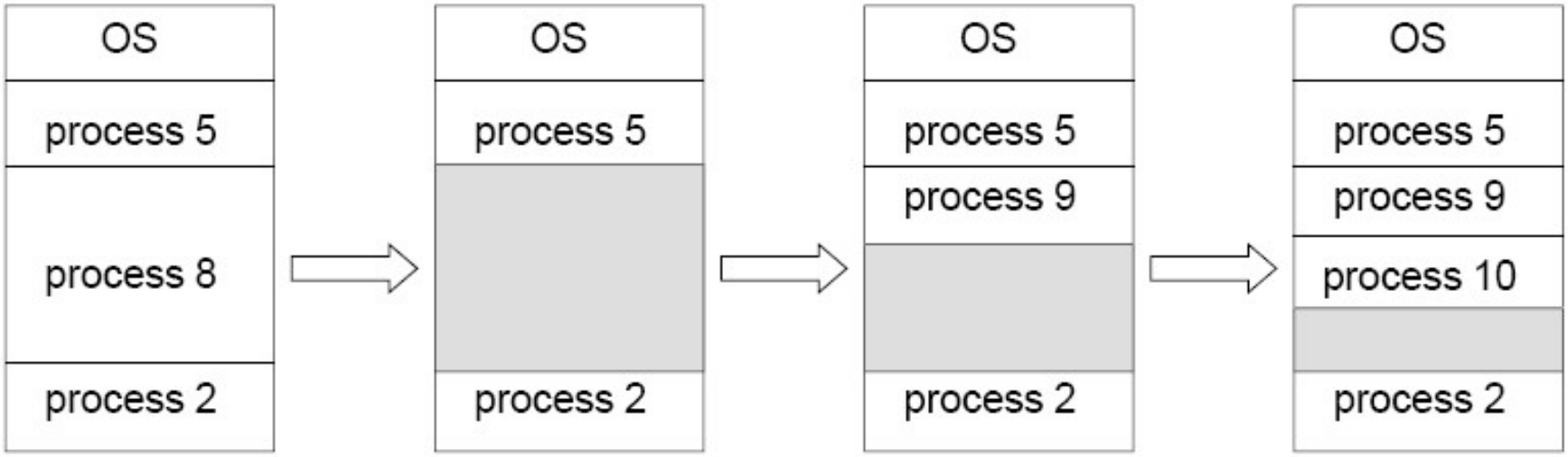
internal fragmentation 최소화

empty queue 존재 가능

multiprogramming degree 높임

(2) 가변 분할 기법 Variable (or Dynamic) Partitioning:

- 시작 時, 모든 메모리가 가용함 (하나의 큰 **Partition**).
- **요구된 만큼** 메모리를 **분할**하여 할당함 - 동적으로 가변 분할.
- **partition** 반환 時, 인접한 **Free Partition**들을 합침(*Merging*).

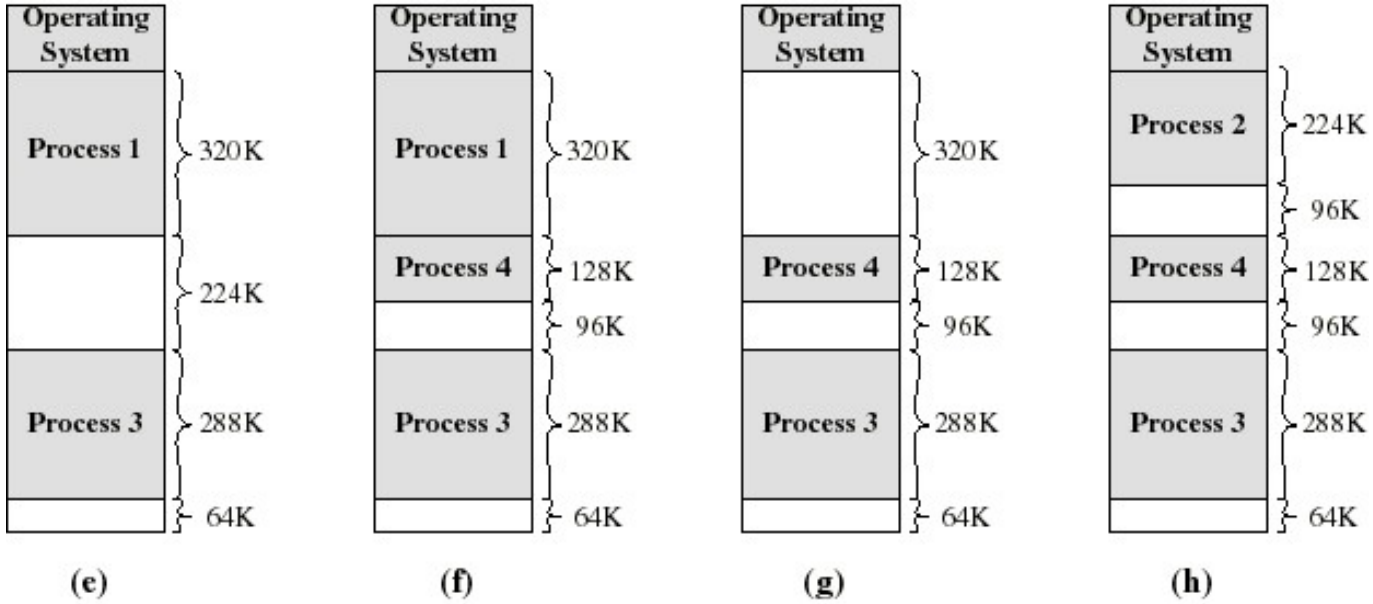


- 할당 기법: First, Best, Worst fit.

  (note) Worst fit 경우 할당 後 충분히 큰 공간이 남음.

- **외부 단편화**External fragmentation와 **압축**Compaction

  - 할당된 **partition** 사이에 요구를 만족할 수 없는 작은 **fragment**들이 생김.



  - 메모리를 **압축**하여 큰 공간을 확보함.

    1. 이웃하는 빈 공간 합치기(통합) - **Coalescing**
    2. 떨어져있는 빈 공간을 하나로 통합 - **Compaction**

    (note) run-time relocation 가능해야 압축할 수 있음.

  - (또 다른 해결책) **Paging, Segmentation.**

3. Paging

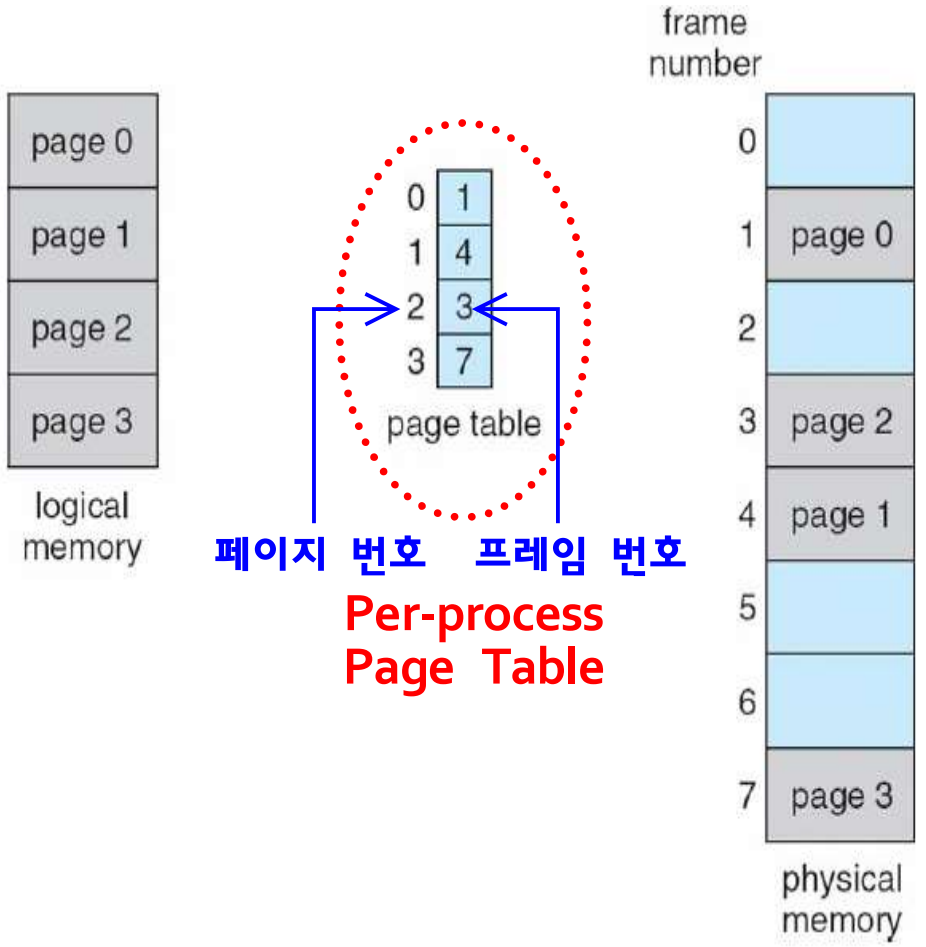
- 하나의 Process를 **물리적 단위(Page)**로 분할하고,
- 각 Page를 임의 위치에 적재함.

- 물리 메모리를 동일 크기로 분할함.
  (각 분할을 ***Page Frame, Frame***이라 함)

  논리 메모리(process)를 page frame과 동일한 크기로 분할함 (***page***라 함):
  논리주소 = (페이지번호, 변위)

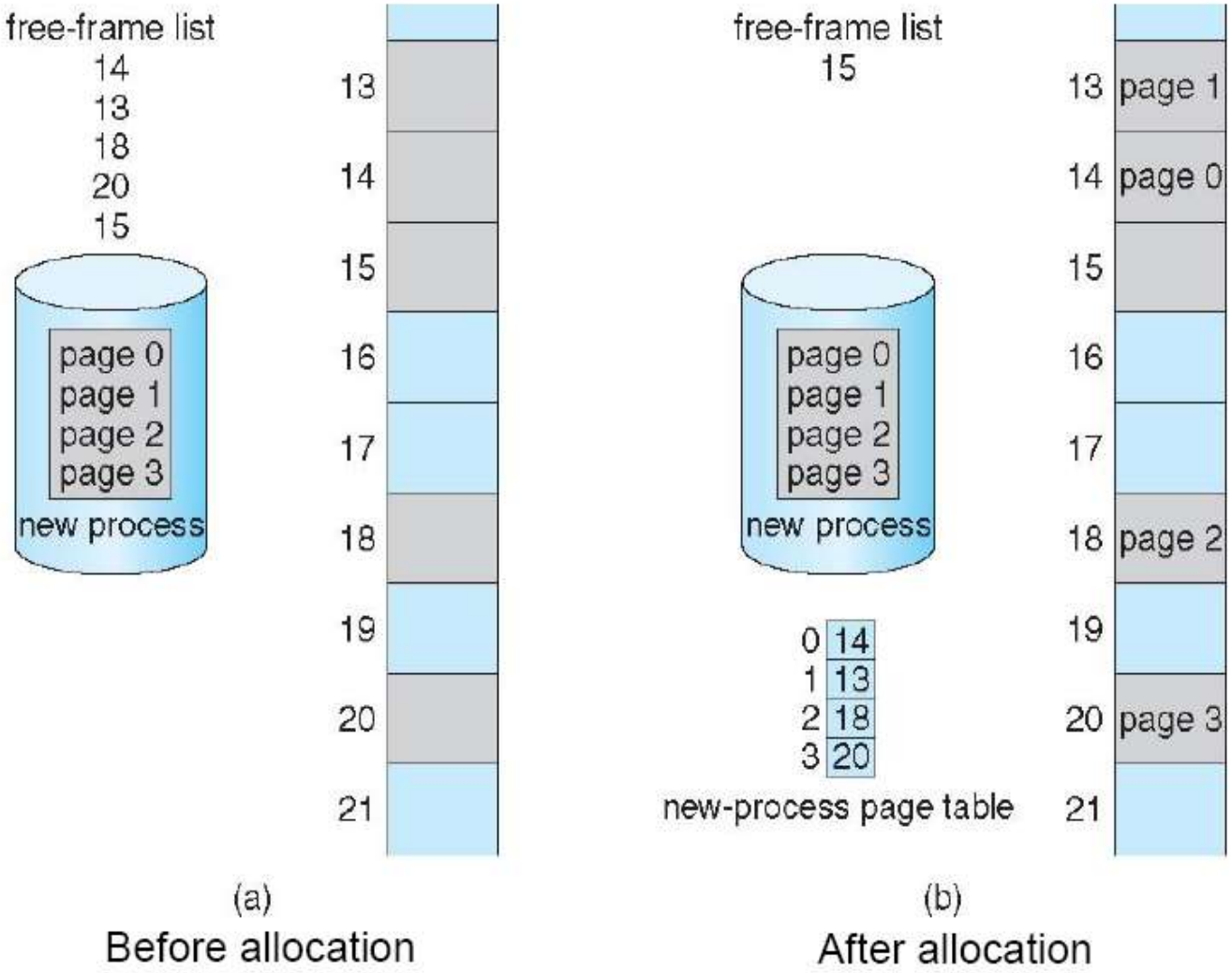
- 각 page는 임의 frame에 적재 가능함.

- 각 프로세스의 page 적재 정보는 자신의 ***Page Table***에 기록됨:
  (페이지번호, 프레임번호)

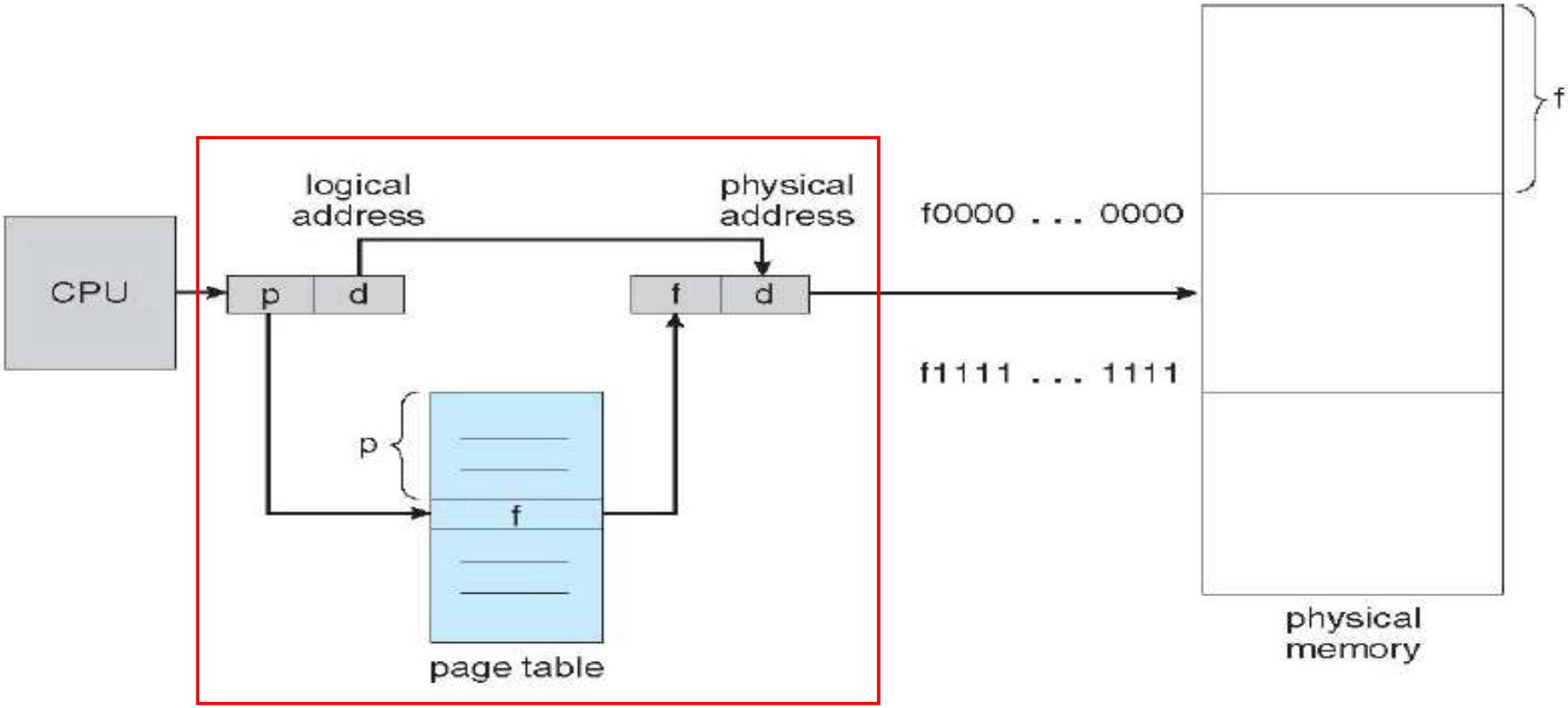


❑ Page Frame 할당 정보

- **Frame Table: { (프레임번호, 가용여부, process id, 페이지번호) }**

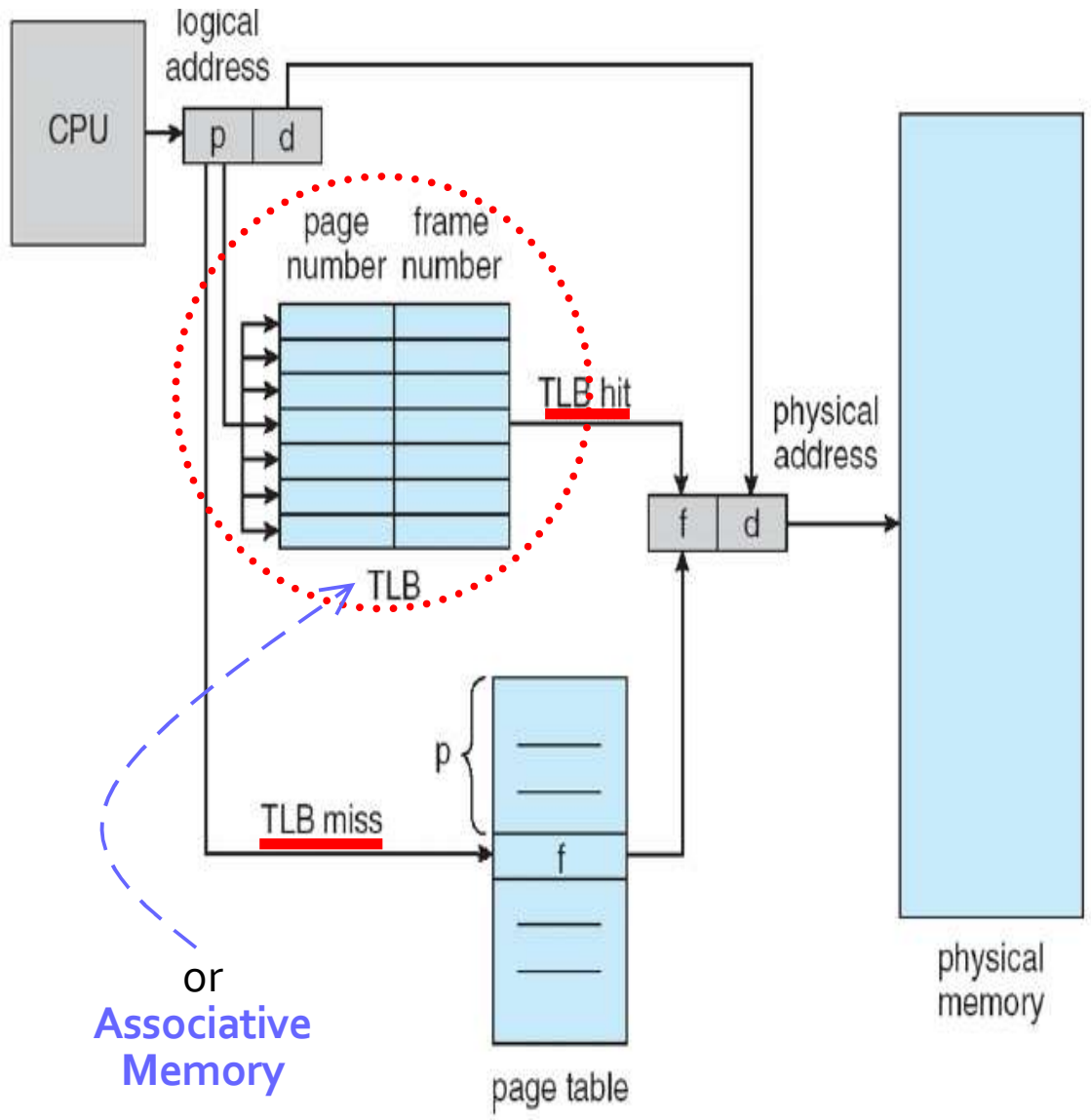


❑ Run-time relocation by **Paging Hardware**



(note) Page Table을 Memory에 저장하는 경우, 하나의 data를 접근하기 위해 memory를 두 번 접근해야 함: (page table 접근, actual data 접근).

❑ Translation Look-aside Buffer(TLB)를 이용한 Paging Hardware



- **TLB Miss 時:**

① 주기억장치 **page table**에서 검색

② **TLB**에 적재

※ victim entry 선정 정책

- **Hit Ratio** (적중률)

  접근하려는 페이지 번호가 **TLB**에 있을 비율.

- **Effective memory access time**

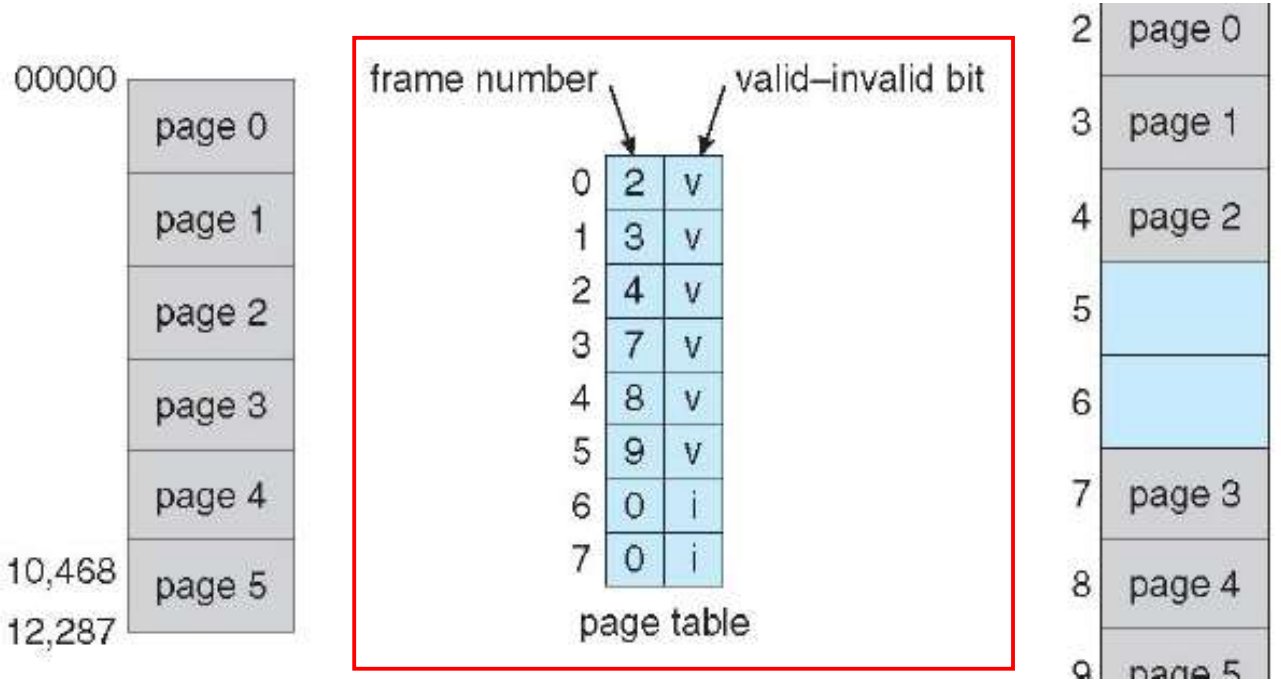
  = 적중률 X TLB검색시간

  + (1-적중률) X 메모리접근시간

❑ 보호 Protection

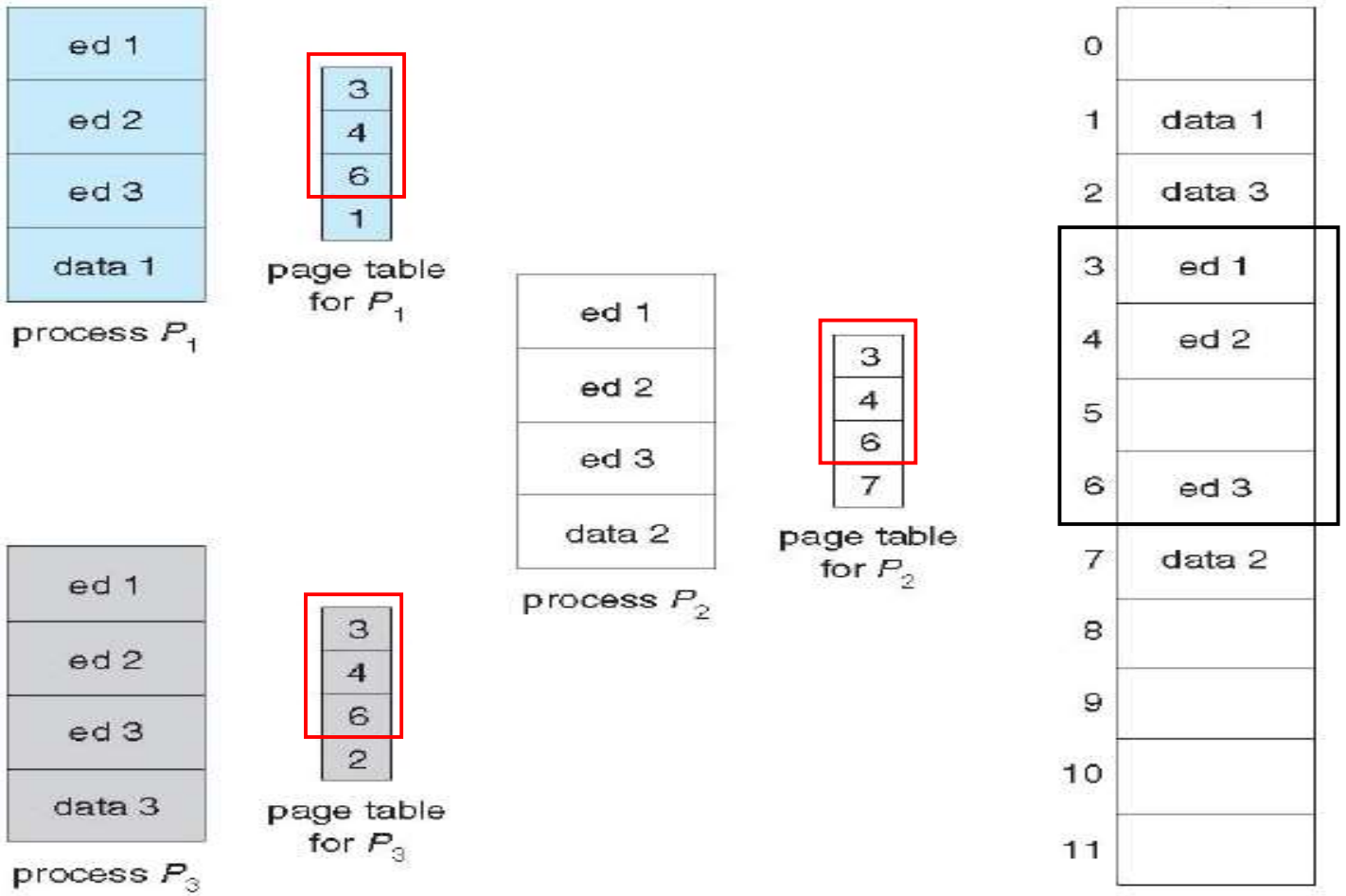
- page table에 **보호비트***protection bit*를 두어 read-write, read-only를 표시함.

- page table에 **유효비트***valid-invalid bit*를 두어 page 합법성 여부를 표시함.
  - *invalid*) 프로세스의 논리주소 공간에 포함되지 않는 페이지(번호).
  - 유효비트 대신 **페이지테이블 길이 레지스터***Page Table Length Register*(**PTLR**) 사용 가능.
    (예) 이 process는 6개의 page를 가지므로 PTLR = 6.
  - 위반 여부는 H/W에 의해 탐지되고 ***hardware trap***이 발생됨.



❑ 공유 페이지 Shared Page

- **재진입 가능 코드**) 실행 동안 절대 변하지 않는 **code.** (예) 편집기, 컴파일러, ...
- 재진입 코드*Reentrant Code*는 여러 **process**에 의해 공유 가능함.
- **frame**의 공유: **1 frame ⇔ n(<1) logical addresses.**

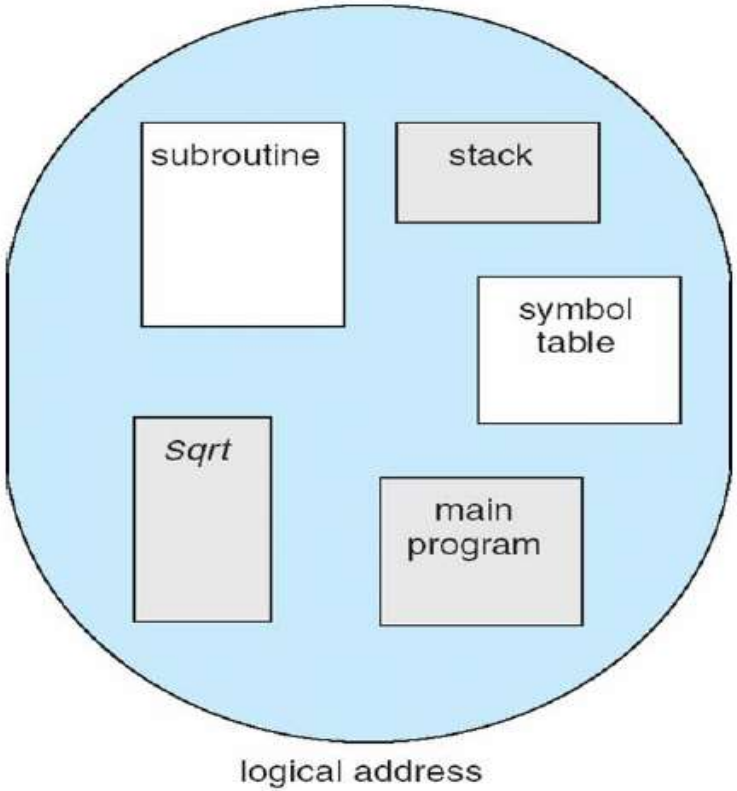


4. Segmentation

- 하나의 process를 **논리적 단위(Segment)**로 분할하고,
- 각 segment 임의 위치에 적재함.

❑ **Segment** – 프로그램의 논리적 구성 단위 (예) function, class, global variables, ...

User's view of a program

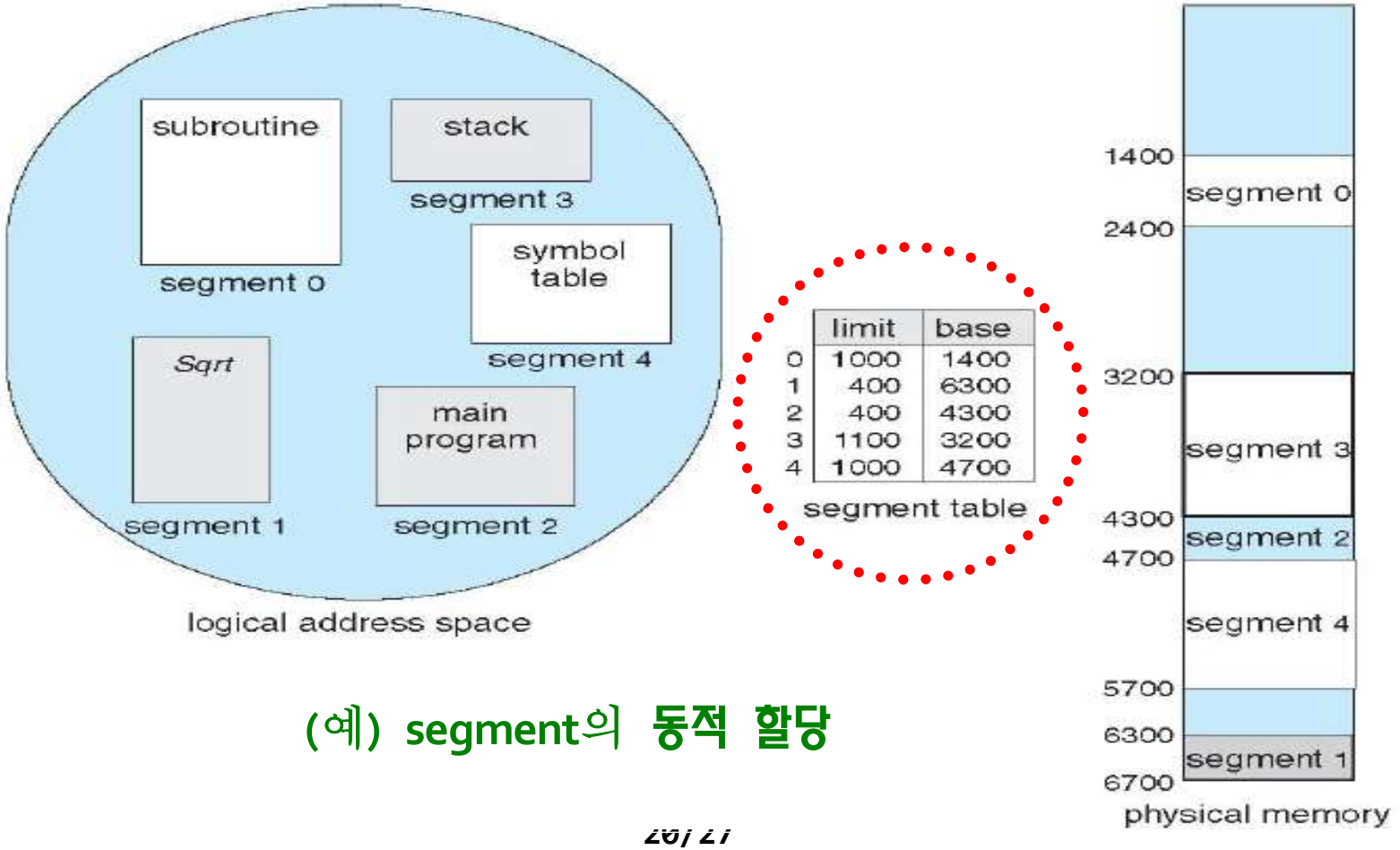


C 컴파일러 생성 세그멘트
• **Code** • **Global variables** • **Heap** for dynamic allocation • **Stacks** per thread • **Standard C library**

Segmentation) 논리 주소 공간을 *Segment* 단위로 분할하고 적재함.

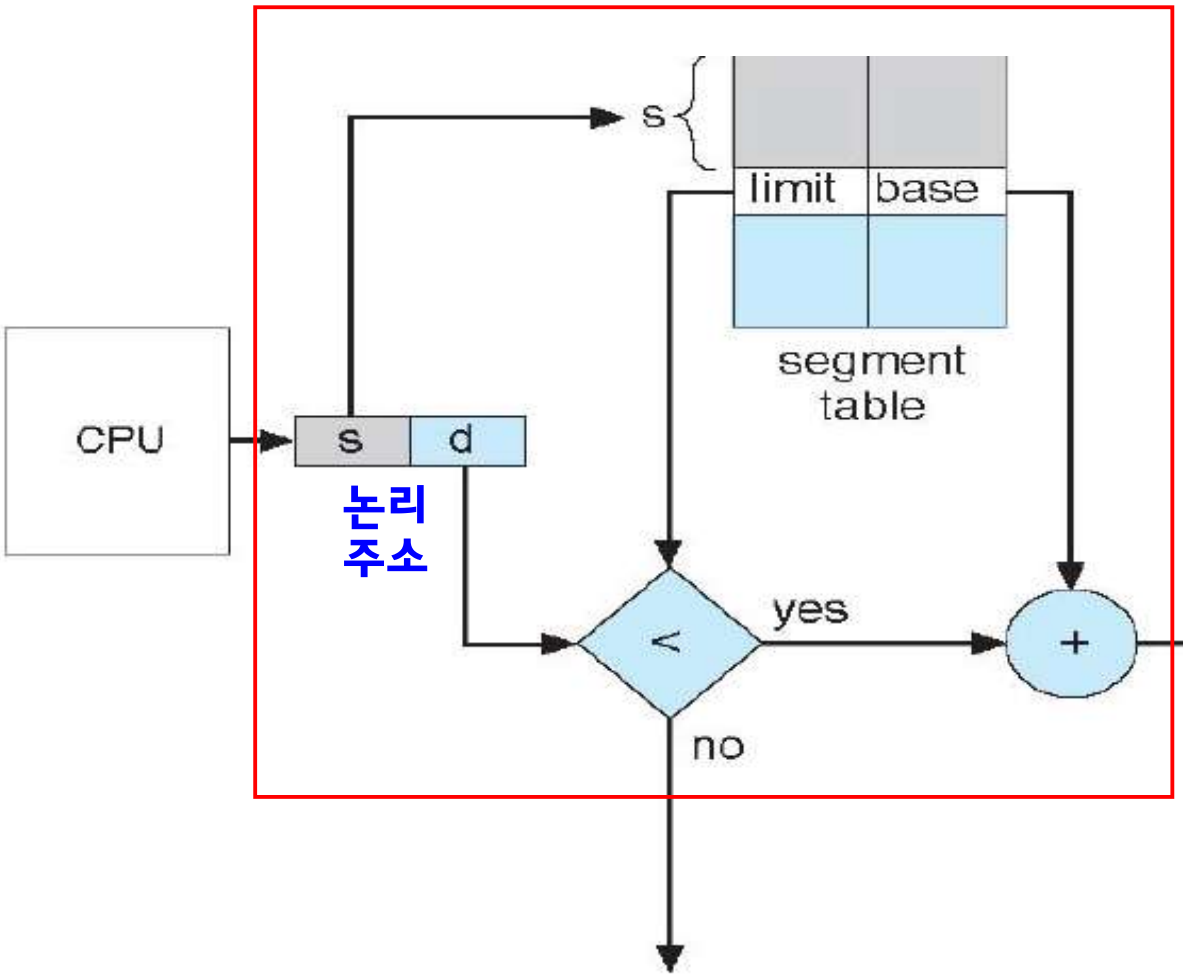
논리 주소	세그멘트번호 s	변위 d

- segment 유형에 따라 서로 다른 code sharing, protection이 가능해짐.
- segment 길이는 가변적이므로 동적 할당 기법 사용함.



(예) segment의 동적 할당

❑ Run-time relocation by **Segmentation H/W**



- **Segment Table** per process.
- Table entry:
  - **세그멘트번호** s
  - **크기** limit (유효 주소)
  - **적재시작주소** base (재배치)
  - **보호비트** (r, w, execute).

5. Page Table의 구조

- **page table** 크기 = 논리주소공간의 페이지 개수 × 프레임번호 크기(*given*)
- 현대 컴퓨터가 지원하는 논리주소공간의 크기: 2^{32}~2^{64} bytes (4GB ~ 16EB)
- **page table**이 너무 큰 경우 **page table**을 연속 할당하기 어려움.

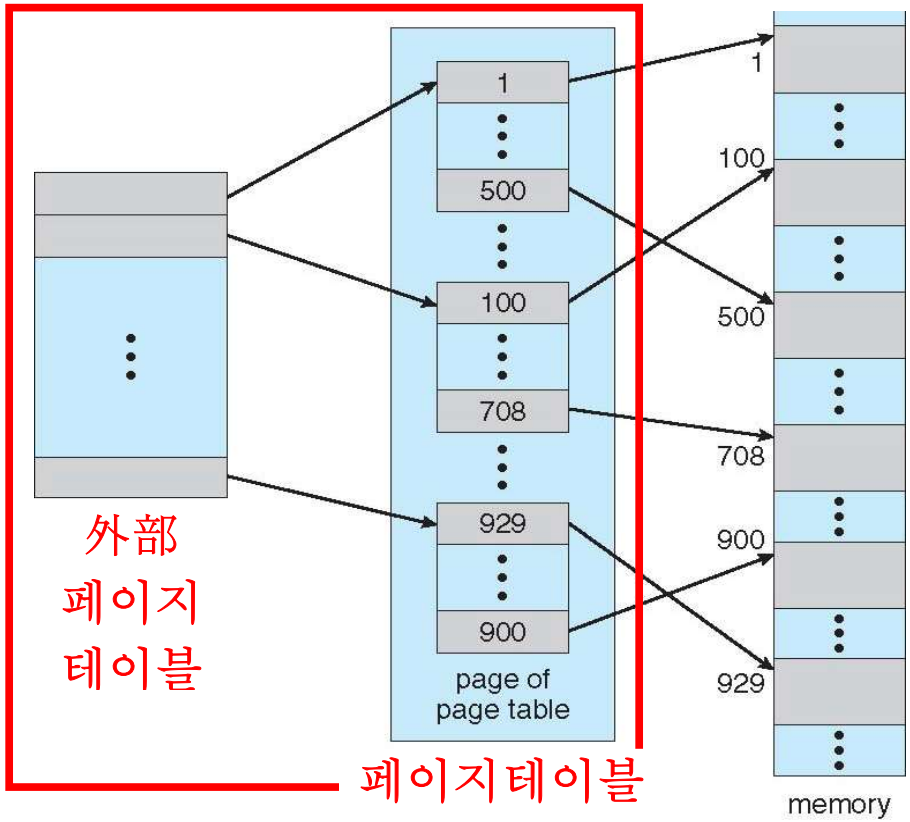
  (예) 64-bit 논리주소, 4KB (2^{12} bytes) page
  - 페이지 테이블의 항목 수 = page 개수 = $2^{64}/2^{12}$ = 2^{52} = 4×Peta 개
  - 페이지 테이블 크기 = 2^{52} × 프레임번호 크기
- 개선된 방법: 계층적, 해시, 역 페이지 테이블

Kilo(2^{10}), Mega(2^{20}), Giga(2^{30}), **Tera(2^{40})**, Peta(2^{50}), **Exa(2^{60})**, Zetta(2^{70}), Yotta(2^{80})

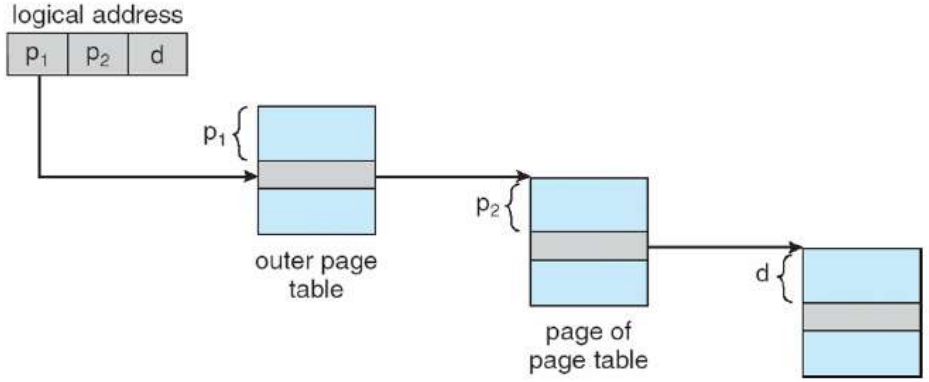
(1) Hierarchical Paging - 페이지테이블을 다시 Paging 함

- 페이지테이블 항목 수 = 1M(개)
- 페이지테이블을 1KB 크기의 page로 나눔

2-단계 페이지테이블



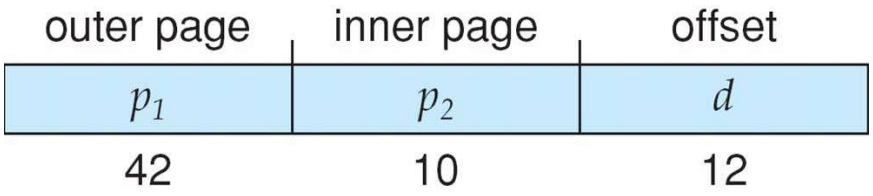
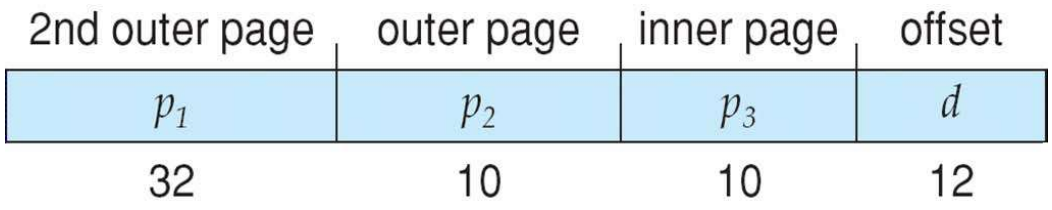
- 주소 변환



- 논리주소:

외부페이지번호	내부페이지번호	변위
p1	p2	d
10	10	12

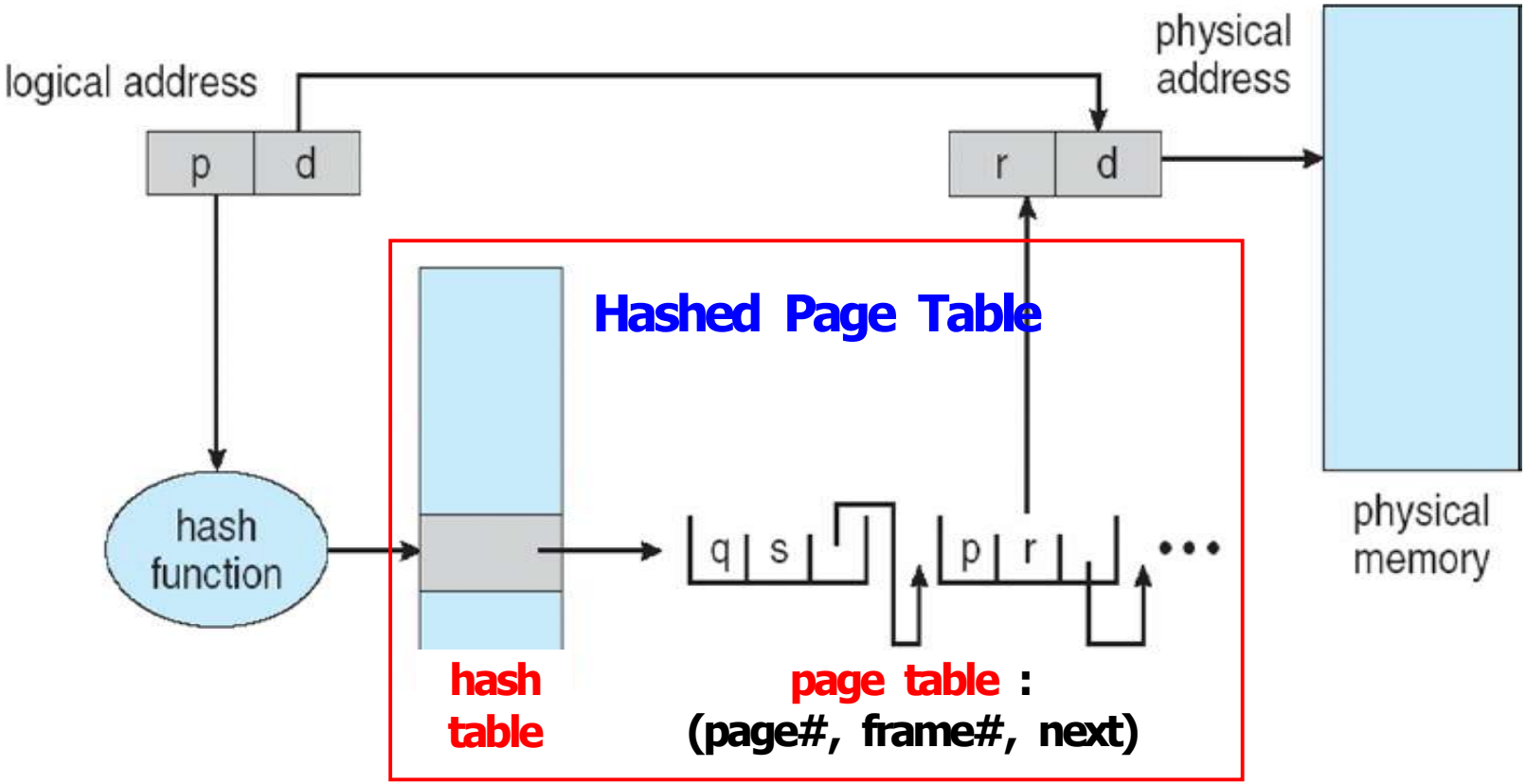
- **64-bit machine (논리주소 크기가 64 bits), 페이지 크기가 4KB(2^{12} bytes)인 경우**

	페이지 테이블의 항목 수 (개)	비고
1-단계 페이징	1-단계 페이지테이블 항목 수 = $2^{64}/2^{12}$ = 2^{52}	4 peta 개
2-단계 페이징	1-단계 페이지테이블을 4KB(2^{10} 4-byte entries)로 나눔. 2-단계 페이지테이블 항목 수 = $2^{52}/2^{10}$ = 2^{42} 	2-단계 페이지 테이블 항목 수: 4 tera 개
3-단계 페이징	2-단계 페이지테이블을 4KB(2^{10} 4-byte entries)로 나눔. 3-단계 페이지테이블 항목 수 = $2^{42}/2^{10}$ = 2^{32} 	3-단계 페이지 테이블 항목 수: 4 giga 개

(note) 64-bit machine의 경우 Hierarchical Paging은 비현실적.

(2) Hashed Page Table:

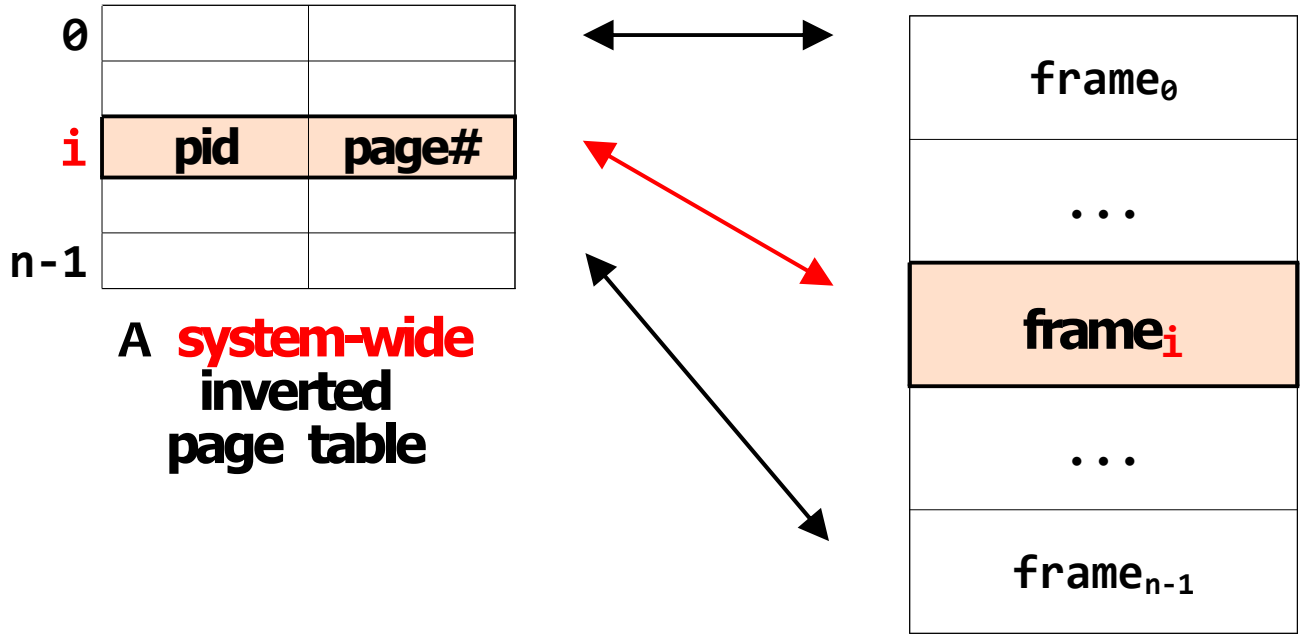
해시테이블[해시함수(page#)]이 가리키는 **list**에서 매치되는 **page#** 검색
⇨ **frame#**



- 주소 공간이 32 비트보다 큰 경우 많이 사용됨.

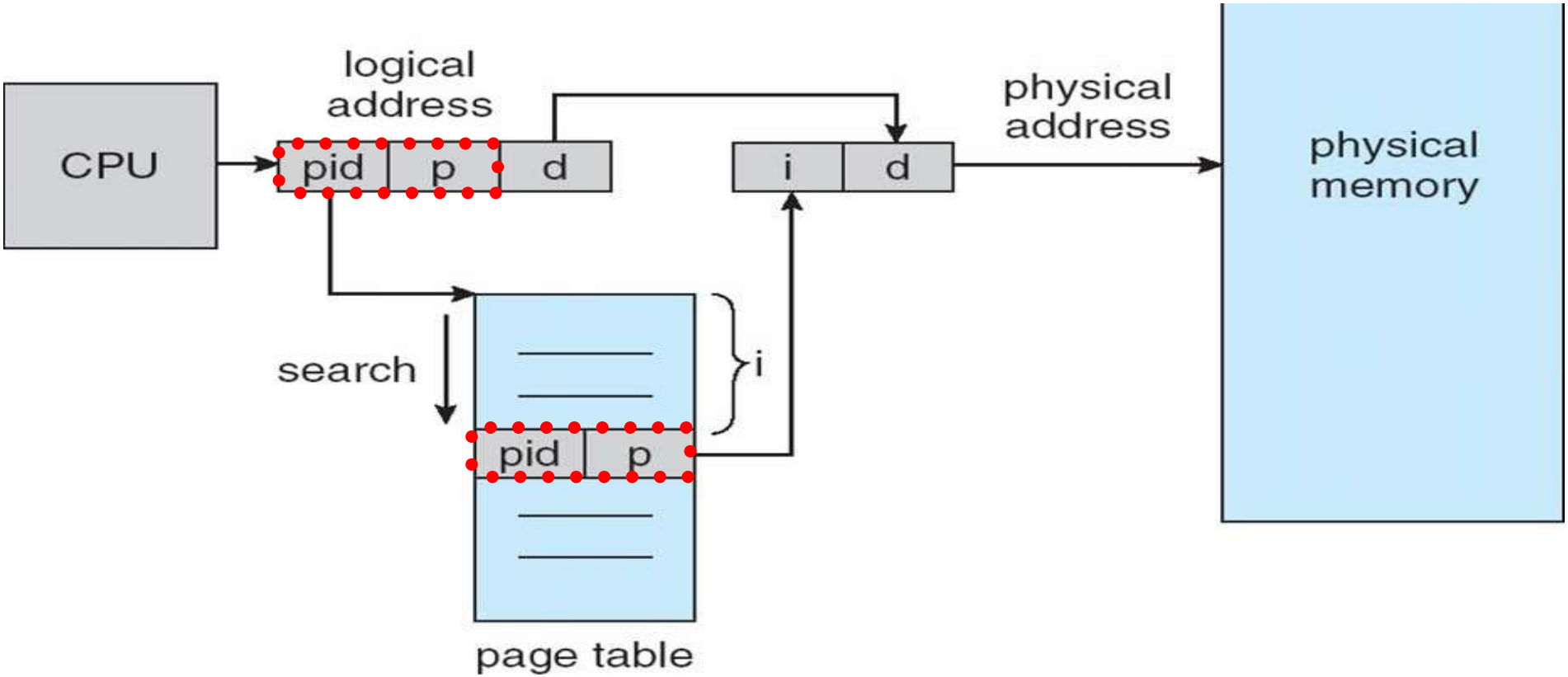
(3) Inverted Page Table

- A single system-wide page table
- One entry per page frame: **(process_id, page#) → frame#**



※ Inverted page table 크기(∝ frame 개수) << system 내 모든 page table의 크기

- **주소매핑 : 논리주소 × InvertedPageTable → i ($frame_i$)**



※ 역 페이지 테이블 search time⬆ ⇨ hashing 또는 TLB 사용하면 ⬇

※ (1 frame ⇔ 1 logical address<pid, p, d>) ⇨ 메모리 공유가 어려움

(note) frame의 공유: 1 frame ⇔ n(<1) logical addresses. (Paging의 공유 참고)