

# Chapter 8: Main Memory

---



# Chapter 8: Memory Management

---

- ❑ Background
- ❑ Swapping
- ❑ Contiguous Memory Allocation
- ❑ Paging
- ❑ Structure of the Page Table
- ❑ Segmentation
- ❑ Example: The Intel Pentium



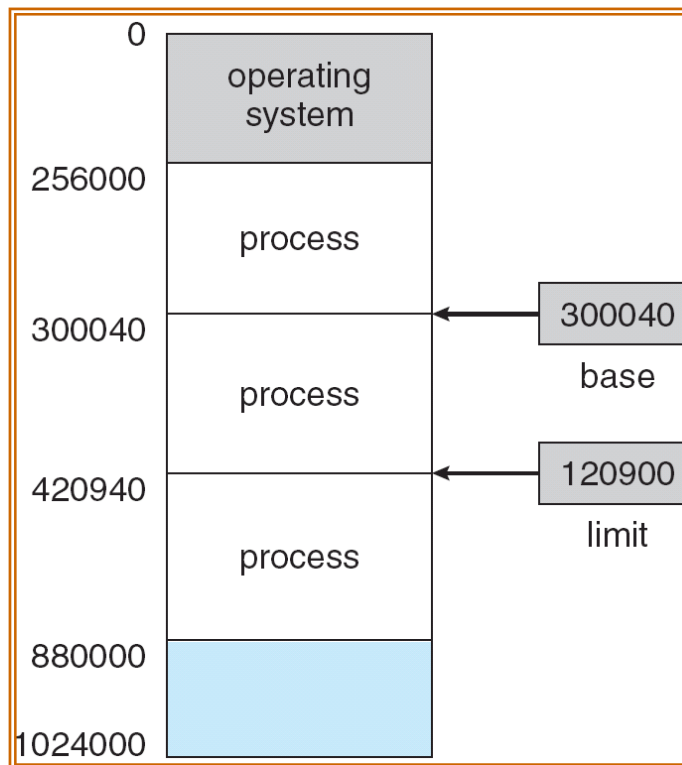
# Background – 기본 하드웨어

- Main memory and registers는 CPU가 접근할 수 있는 유일한 storage
  - 즉, Disk에 저장된 데이터라도 일단 메모리에 먼저 load 되어야 함
  - CPU에 의해 실행되는 모든 명령은 메모리 주소만을 인수로 사용해야함
- Register VS. Main Memory
  - Register : one CPU clock (or less)
  - Main memory : Many Clock Tick Cycle
    - **Cache** : Register와 Main Memory 사이의 속도차이에 의해 stall(지연) 현상 발생을 완화시킴(L1, L2)
- Memory의 보호(protection) 전략 필요



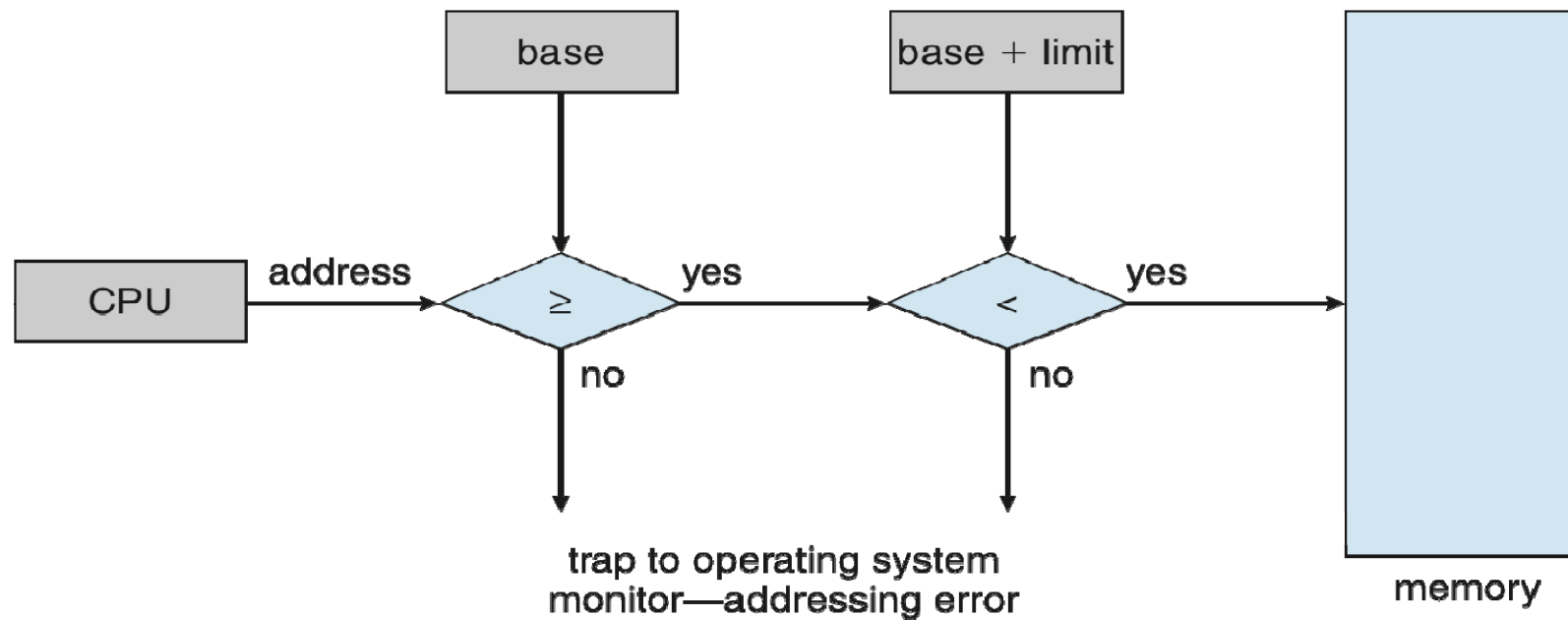
# Background – 기본 하드웨어

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



# Hardware Address Protection

with Base and Limit Registers



# Background – Address Binding

- 프로그램의 생애에서 address가 표현되는 서로 다른 방법들
  - **Symbolic** : Source code addresses usually **symbolic**
    - `int x`
    - `x = x + 1;`
  - **Relocatable** : Compiled code addresses **bind** to relocatable addresses
    - i.e. “14 bytes from beginning of this module”
  - **Absolute** : Linker or loader will bind relocatable addresses to absolute addresses
    - i.e. 74014
  - Each binding maps one address space to another



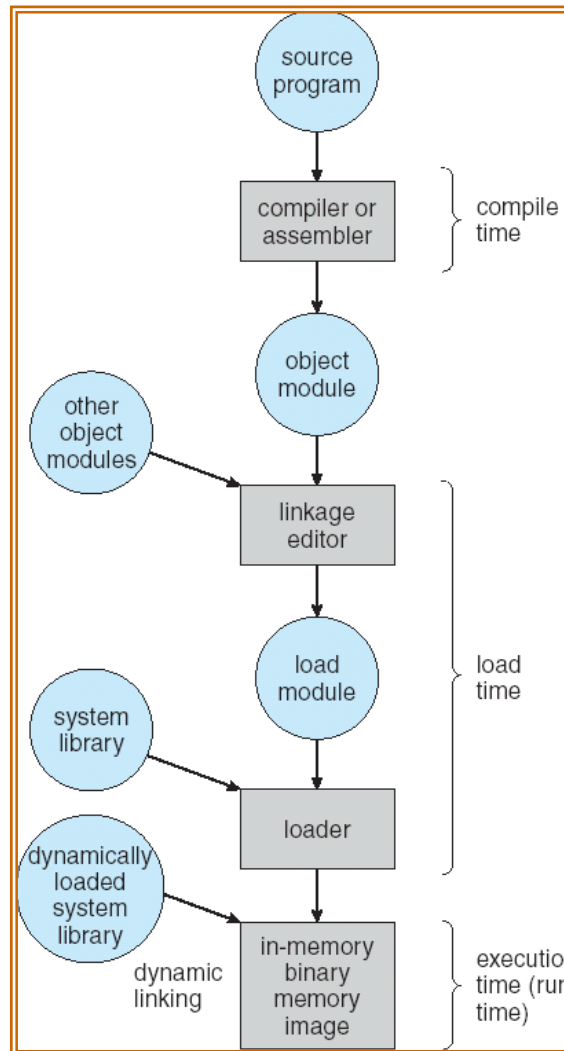
# Background – Address Binding

- 메모리 주소가 바인딩 되는 시점들
  - **컴파일 시점(Compile time):**
    - 메모리 위치가 사전에 알려진다면, 절대 위치 값을 갖는 코드(**absolute code**)가 생성될 수 있음
      - 반드시 그 주소에서 실행되며, 옮길 경우 재 컴파일
  - **적재 시점(Load time):**
    - 실행될 곳의 주소가 메모리가 적재되는 시점에 확정되는 경우로서, 컴파일 시점에 알려지지 않은 위치값을 재위치시키기 위한 코드(**relocatable code**)를 생성
  - **실행 시점(Execution time):**
    - 컴파일할때 생성된주소대로 **CPU**가 실행하고, **CPU**가 메모리로 주소를 보내는 실행시점에 **binding**
      - 별도의 주소변환 하드웨어가 필요
      - \*.dll 파일들



# Background – Address Binding

## 사용자 프로그램의 단계별 처리과정





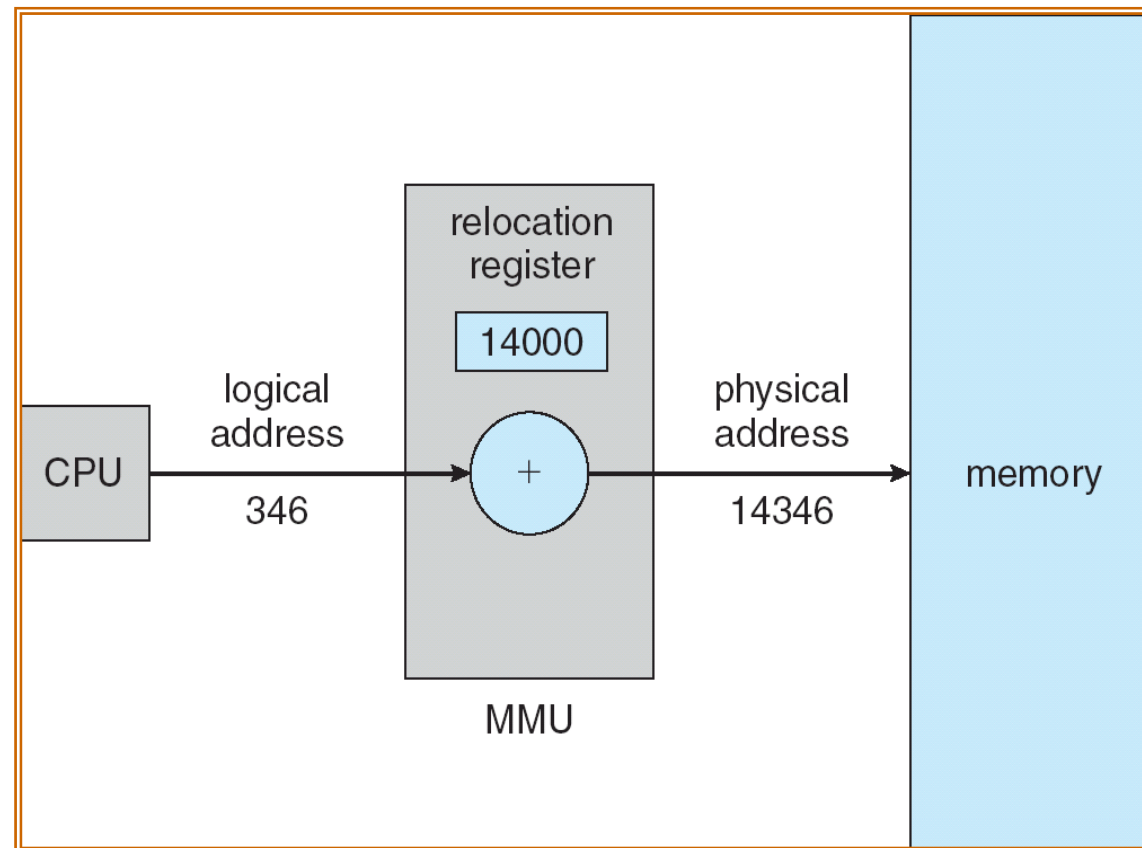
# Background – 주요 개념

- 논리 주소와 물리 주소
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
  
- Memory-Management Unit (MMU)
  - CPU가 메모리에 접근하는 것을 관리하는 컴퓨터 하드웨어 부품
  - 가상 메모리 주소를 실제 메모리 주소로 변환
  - **Relocation Register**
    - 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

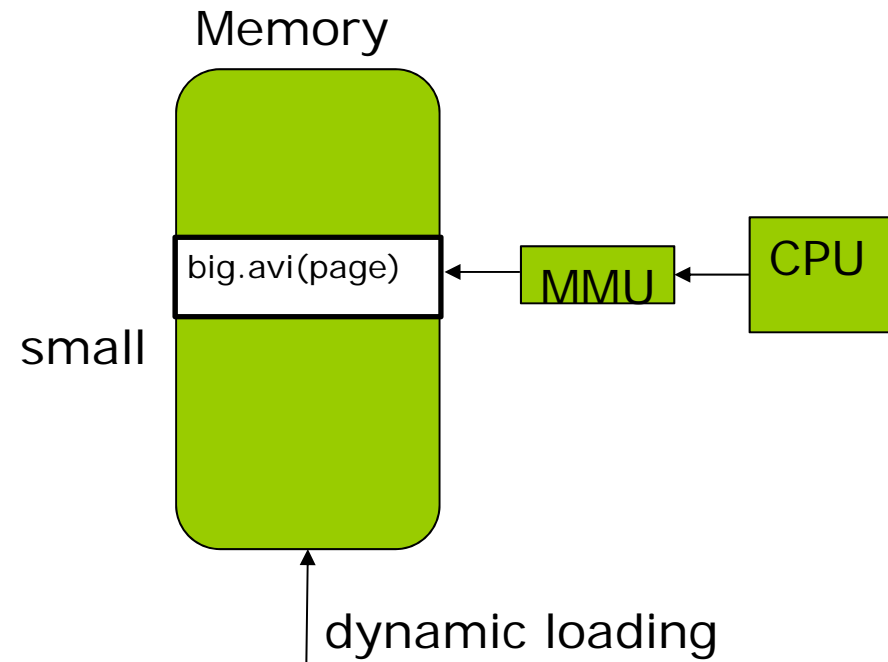
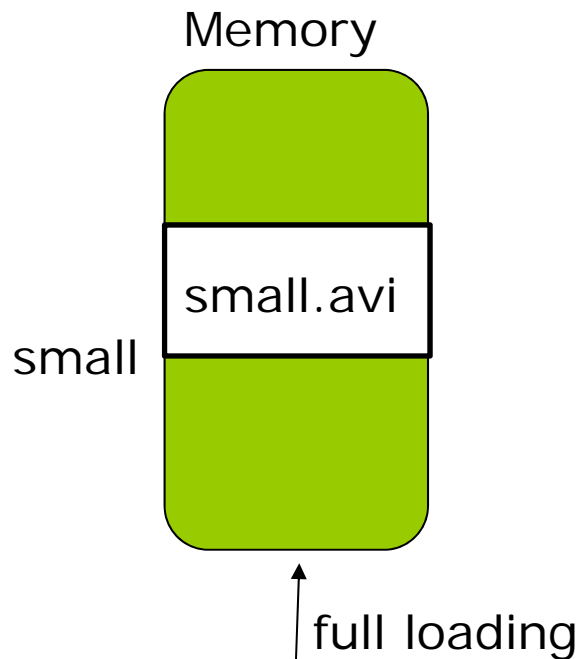


# Background – 논리주소 vs 물리주소

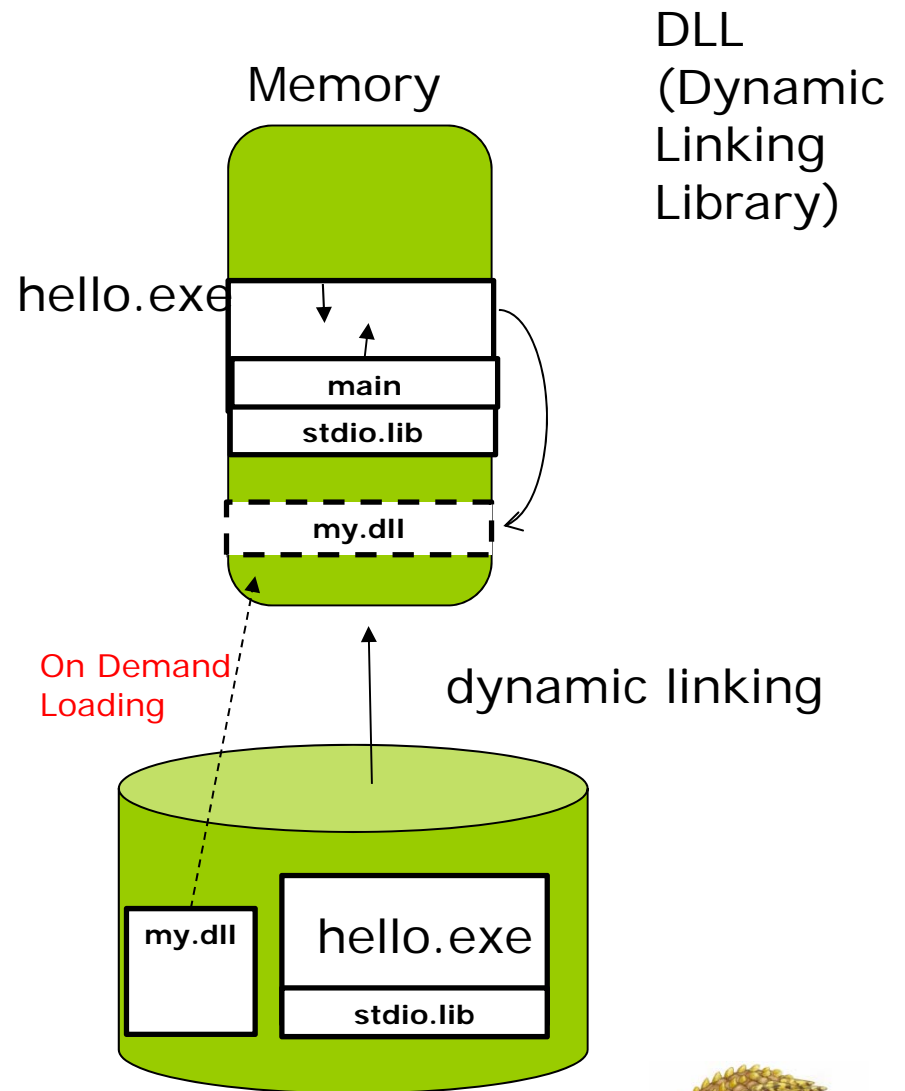
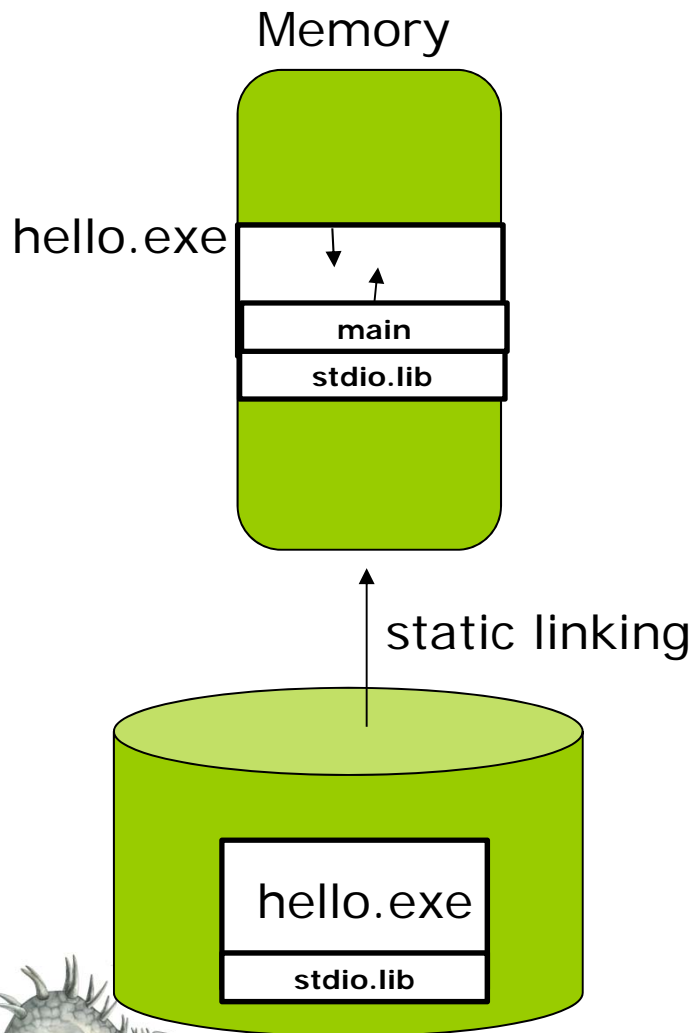
Relocation Register를 이용한 동적 재배치



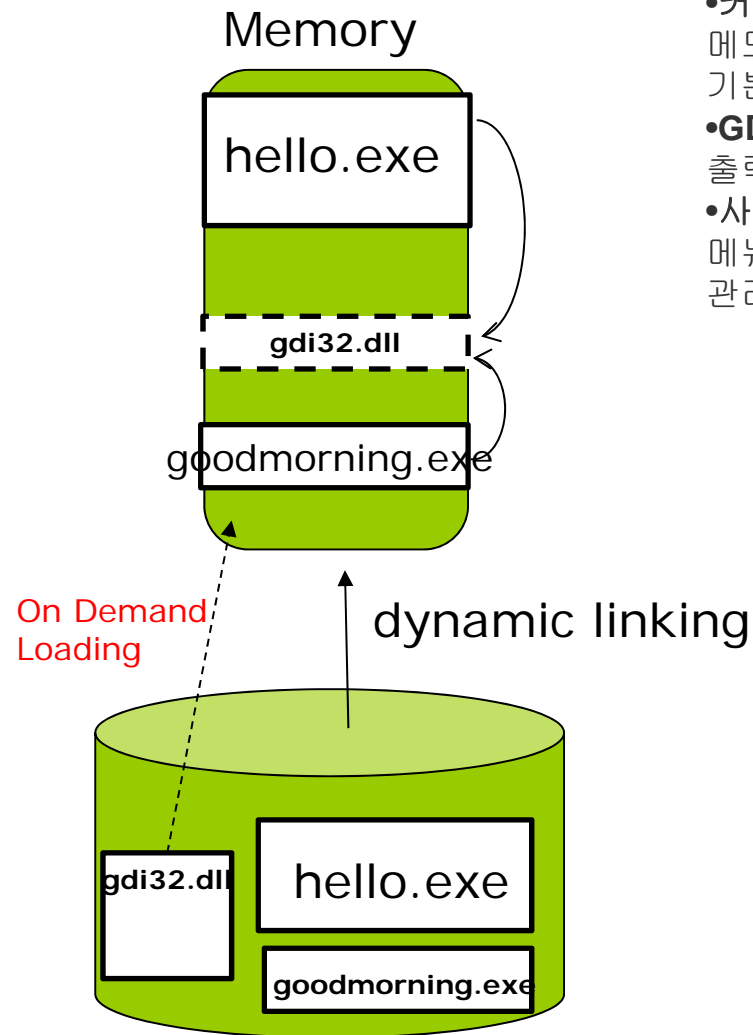
# 동적 적재(Dynamic Loading)



# 동적 링킹(Dynamic Linking)



# 공유 라이브러리(Shared Library)



- 커널 (KERNE32.DLL)** : Windows OS의 핵심으로 메모리관리, 파일 입출력, 프로그램의 로드와 실행 등 OS 기본기능을 수행한다.
- GDI (GDI32.DLL)** : 화면이나 프린터 같은 출력장치의 출력을 관리한다.
- 사용자 인터페이스 (USER32.DLL)** : 윈도우, 다이얼로그 메뉴, 커서, 아이콘 등과 같은 사용자 인터페이스 객체들을 관리한다.



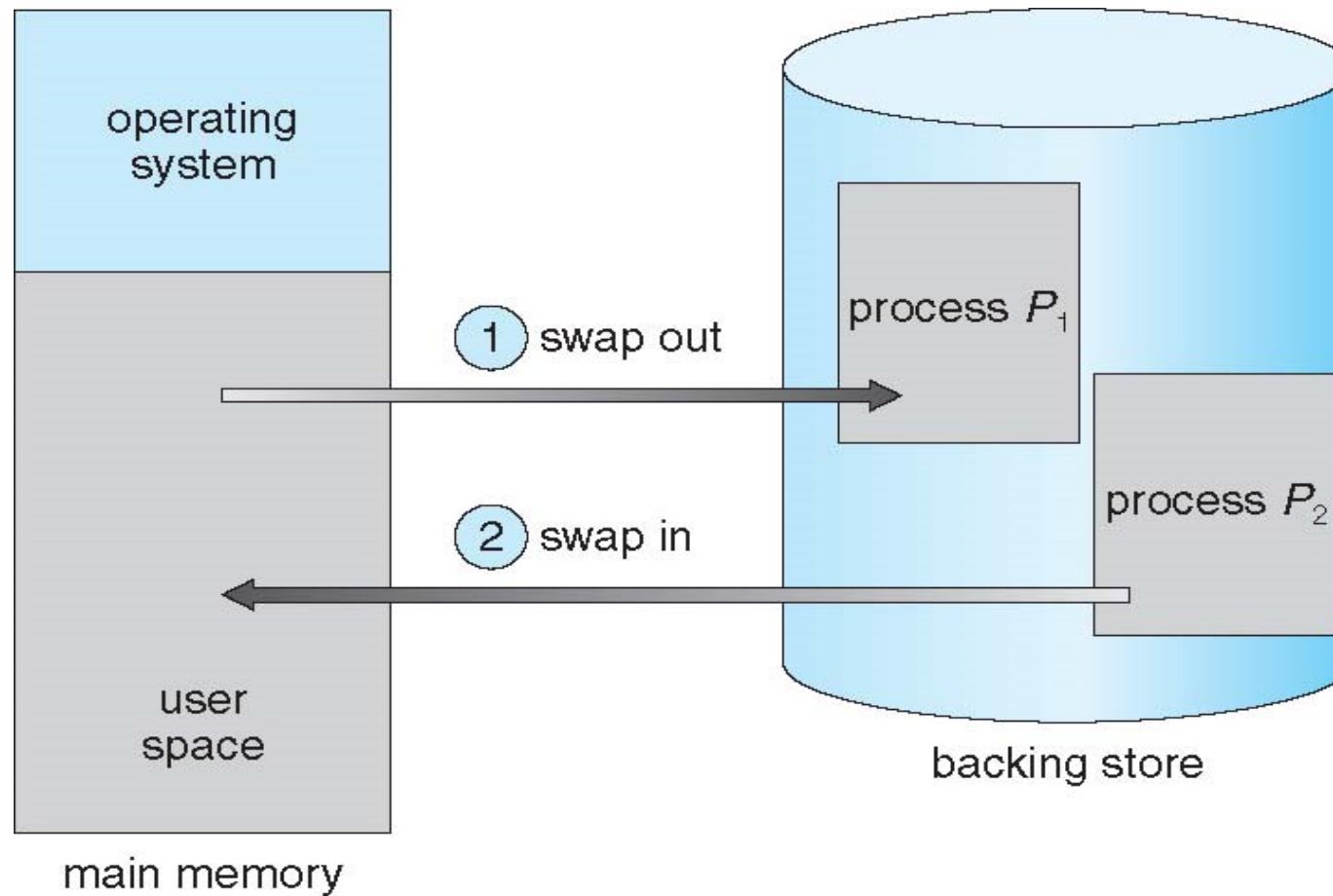
# Swapping

---

- ❑ **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- ❑ Swap out, Swap in
- ❑ **Roll out, roll in** – swapping variant used **for priority-based scheduling algorithms**; lower-priority process is swapped out so higher-priority process can be loaded and executed

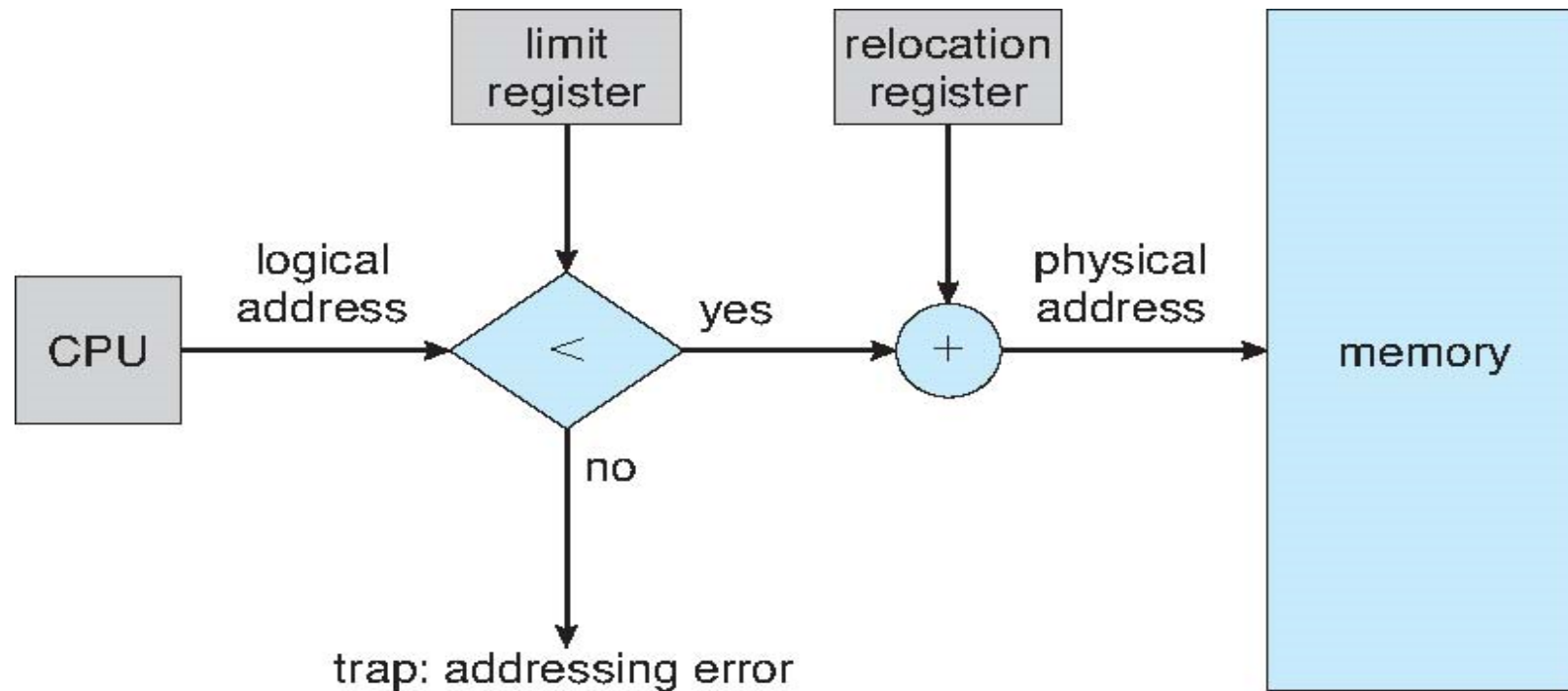


# Swapping



# Relocation and Limit Registers

## □ Limit Register를 이용한 Memory Protection





# 메모리 할당 전략

---

## □ 연속 메모리 할당

- 단일 프로그래밍/연속할당
- 멀티 프로그래밍/고정 분할(Fixed Partition)
- 멀티 프로그래밍/가변분할(Variable Partition)

## □ 비연속 메모리 할당

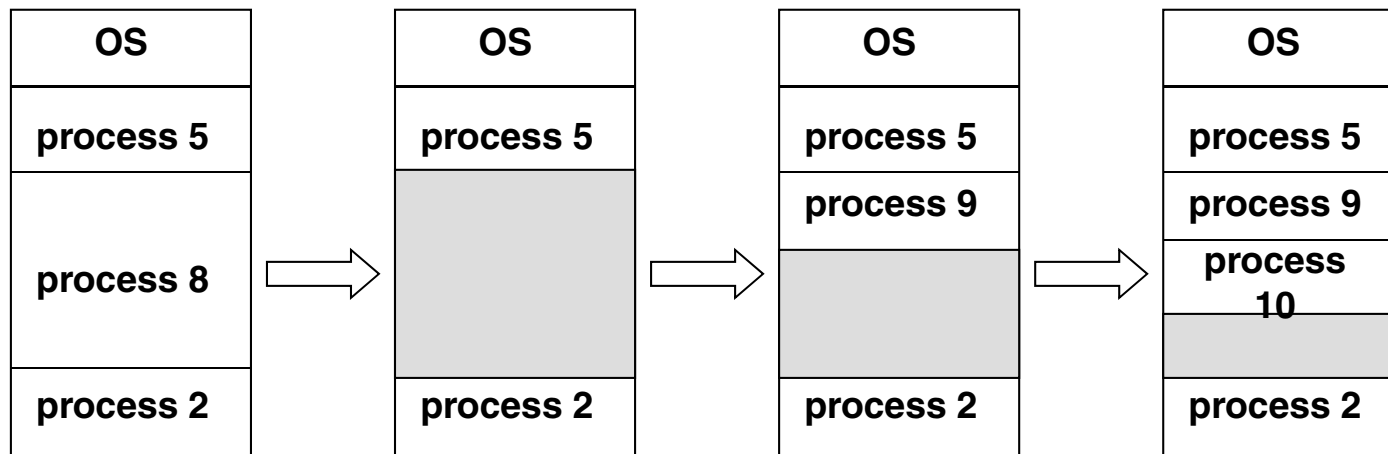
- 페이징(Paging) 기법
- 세그멘테이션(Segmentation) 기법



# Contiguous Allocation (Cont.)-할당방법

## □ 연속할당 전략(가변 분할)

- 각 프로세스가 요청하는 크기에 맞추어 메모리 공간을 할당해 주는 전략
- 프로세스의 종료와 신규 실행에 따라 메모리에 단편화 현상이 발생(Hole)
  - 조각화를 최소화 할수 있는 프로세스 배치 전략이 요구됨



빈공간 리스트와 사용공간 리스트로 관리



# 배치전략 - 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 the smallest 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 speed and storage utilization**



# 연속할당의 문제점 - Fragmentation

- 단편화(Fragmentation)의 정의
  - 프로그램들이 사용하지 못하는 메모리 공간 조각 부분
- 단편화의 종류
  - 외부 단편화(External Fragmentation)
    - 빈 공간 리스트에 속하는 공간중 크기가 너무 작아 어느 누구도 쓸수 없는 공간
  - 내부 단편화(Internal Fragmentation )
    - 프로그램이 요구한 것보다 더 큰 공간을 할당한 경우 낭비되는 부분
      - 예) 512byte block에서 100byte만 사용한 경우



# 연속할당의 문제점

- 외부 단편화 해결 방안
  - 통합(Coalescing)
    - 빈 공간 리스트의 공간들중 연속된 공간이 있는 경우 통합
  - 압축(Compaction)
    - 사용중인 프로세스들을 옮겨 모든 빈공간을 하나로 모음
- 사용자 프로그램의 크기가 커서 적재가 어려울 경우
  - 오버레이(overlay) 기법
  - 비연속 기억장치 할당기법
  - 스와핑(swapping) 기법



# 메모리 관리 - Paging

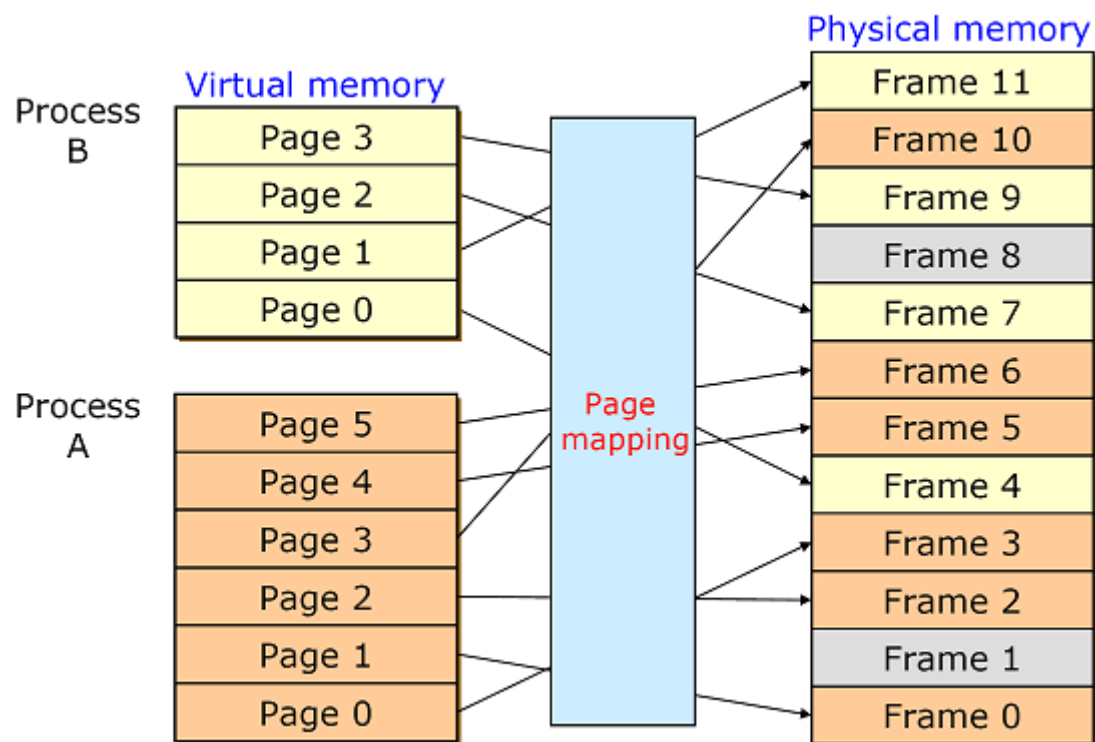
## □ 기본 방법

- Divide physical memory into fixed-sized blocks called **frames**
  - **Frames** : size is power of 2, between 512 bytes and 8,192 bytes
- Divide logical memory into blocks of same size called **pages**
- To run a program of size  **$n$**  pages, need to find  $n$  free frames and load program
- **Page table** 이용 맵핑: logical to physical addresses

## □ Internal fragmentation 발생



# 메모리 관리 - Paging



# Paging - Address Translation Scheme

- 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

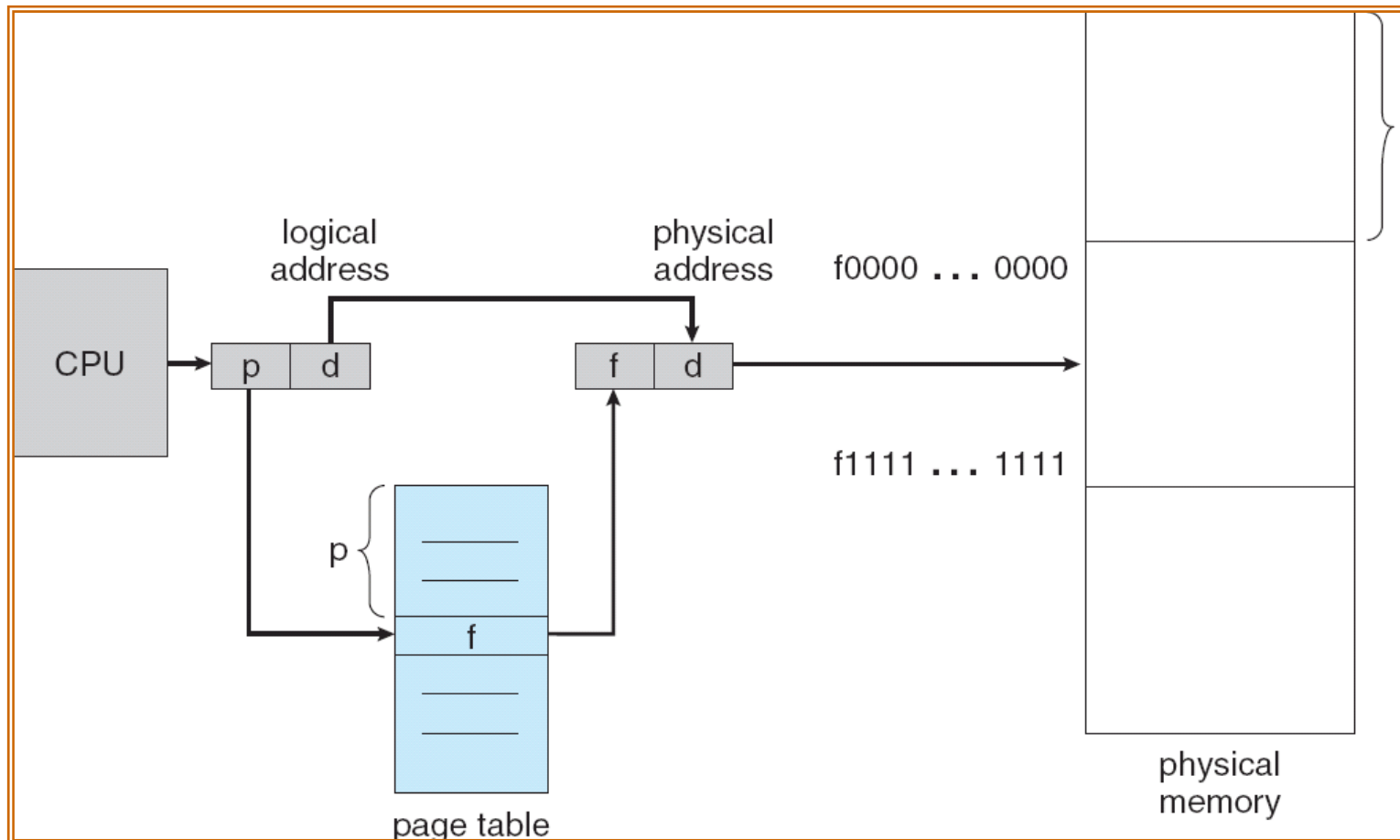
page number	page offset
$p$	$d$
$m - n$	$n$

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

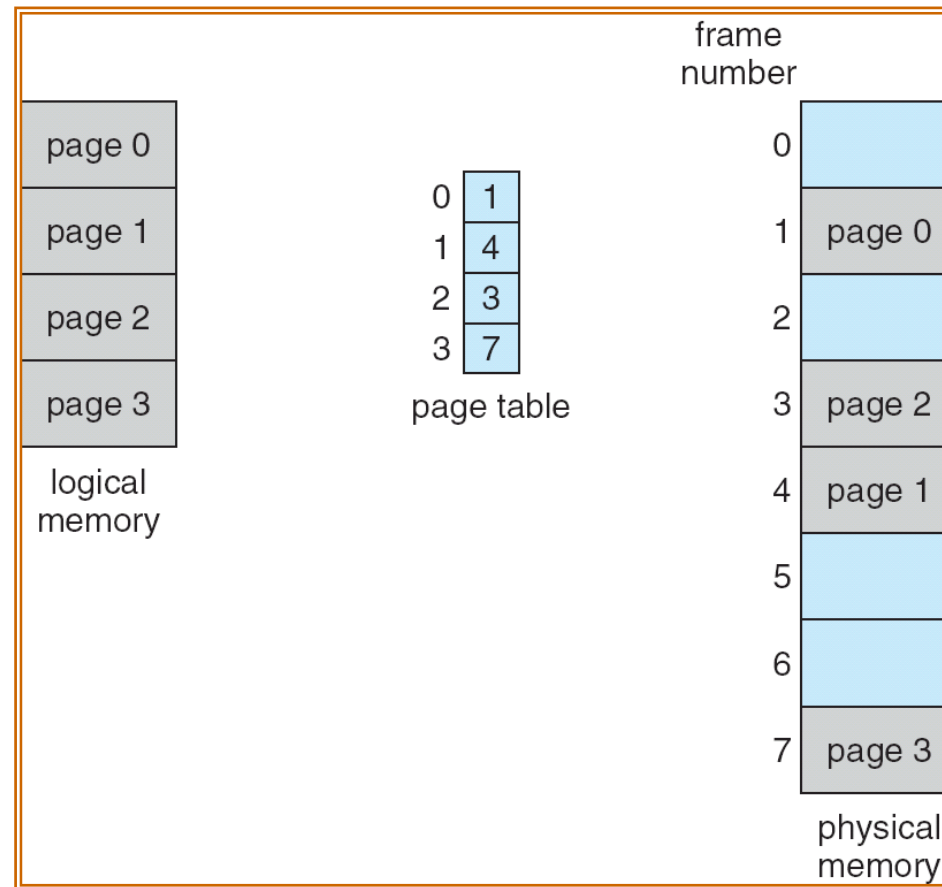




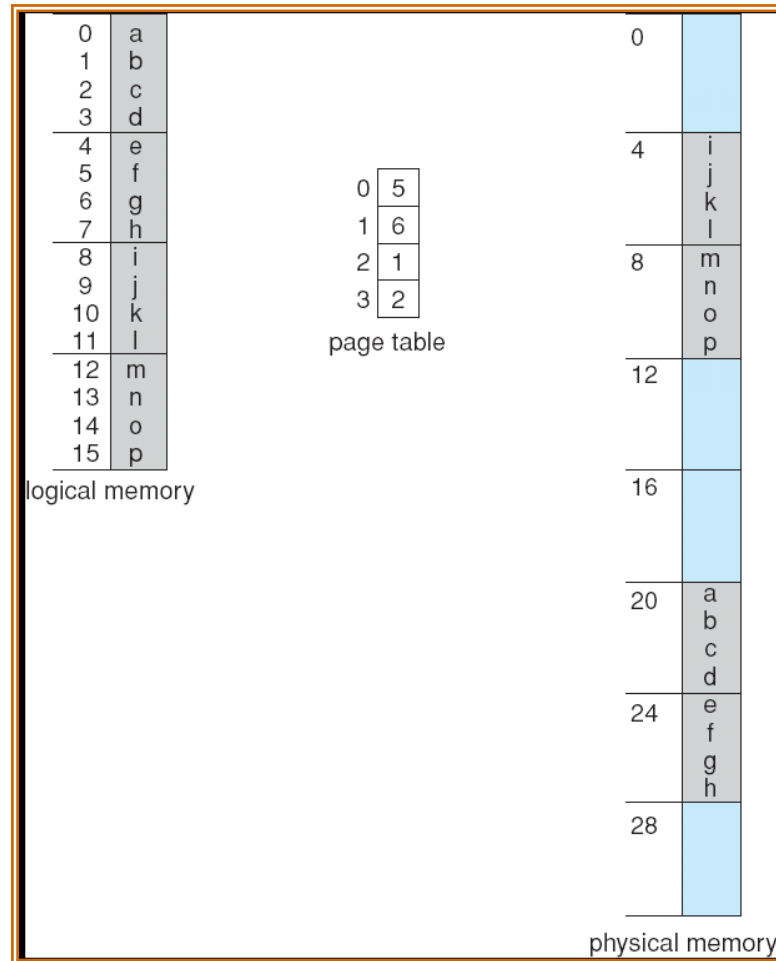
# Paging - Paging Hardware



# Paging - Paging Model of Logical and Physical Memory



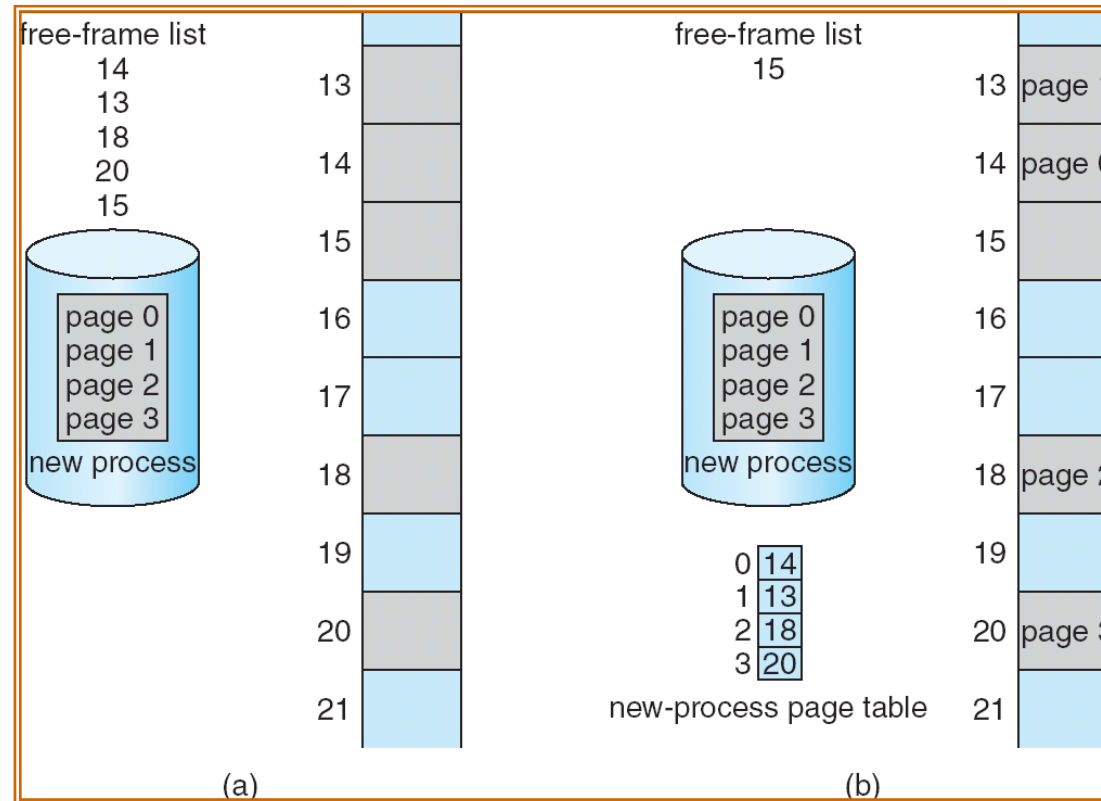
# Paging - Paging Example



**32-byte memory and 4-byte pages**



# Paging - Free Frames



**Before allocation**

**After allocation**



# Paging – 하드웨어 지원

- PTBR과 PRLR을 이용한 Page Table 접근 속도 향상
  - Page-table base register (PTBR) points to the page table
  - Page-table length register (PTLR) indicates size of the page table
    - In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
- TLB를 이용한 하드웨어 가속
  - 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)**
- ASID를 이용한 TLB 속도 향상
  - Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process



# Associative Memory

- Associative memory – parallel search

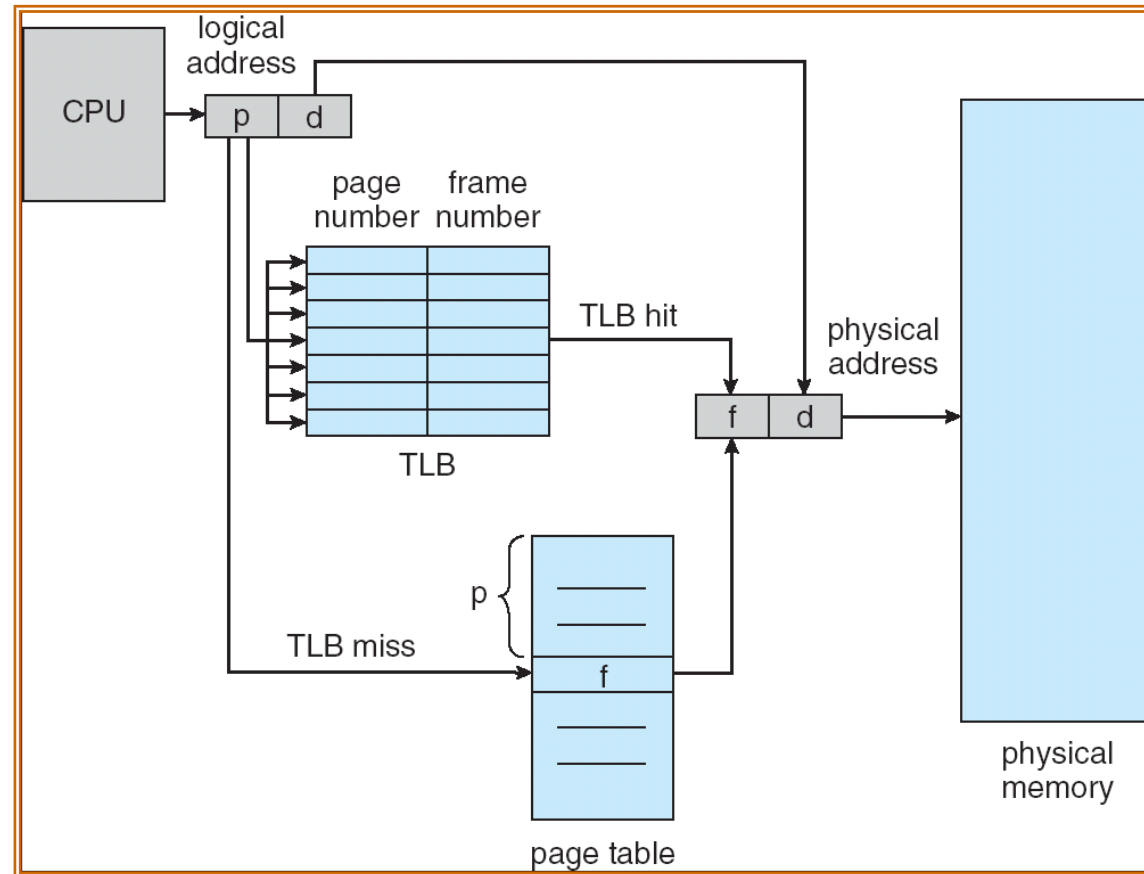
Page #	Frame #

Address translation (p, d)

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



# Paging Hardware With TLB



# TLB를 이용한 Effective Access Time

- Associative Lookup =  $\varepsilon$  time unit
- Assume memory cycle time is 1 microsecond
- **Hit ratio** – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Hit ratio =  $\alpha$
- **Effective Access Time (EAT)**

$$\begin{aligned} \text{EAT} &= (1 + \varepsilon) \alpha + (2 + \varepsilon)(1 - \alpha) \\ &= 2 + \varepsilon - \alpha \end{aligned}$$





# TLB를 이용한 Effective Access Time

## □ Access Time 계산의 예

### ■ 가정

- TLB 탐색시간 : 20nano

- 메모리 접근시간 : 100nano

  - TLB에 있다면 접근시간 : 120 nano

  - TLB에 없다면 접근시간 : 220 nano

### ■ Hit Ratio에 따른 접근시간 예측

- Hit Ratio 80%인 경우

  - Access time =  $0.8 \times 120 + 0.2 \times 220 = 140$  nano

- Hit Ratio 98%인 경우

  - Access time =  $0.98 \times 120 + 0.02 \times 220 = 122$  nano



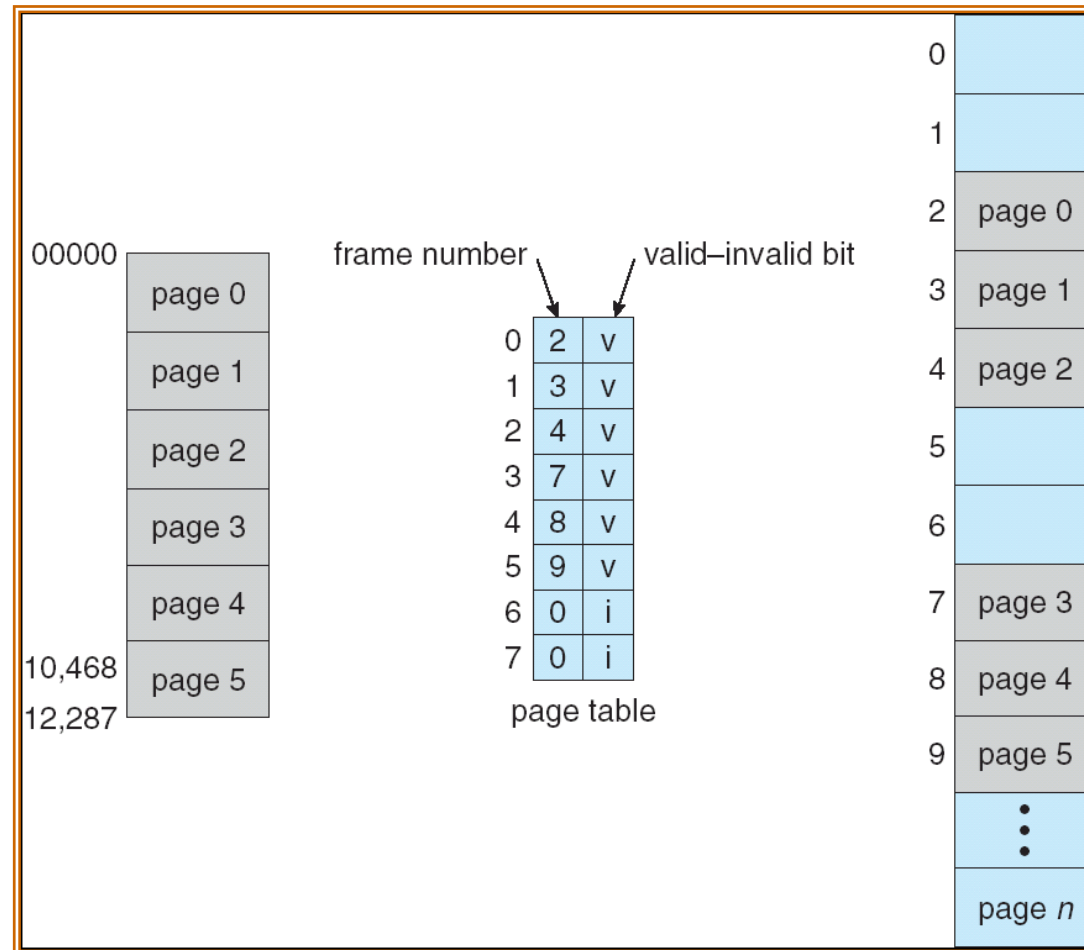
# Memory Protection

---

- ❑ Memory protection implemented by associating protection bit with each frame
- ❑ **Valid-invalid** bit attached to each entry in the page table:
  - “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page
  - “invalid” indicates that the page is not in the process’ logical address space



# Valid (v) or Invalid (i) Bit In A Page Table



# Shared Pages

---

## ❑ Shared code

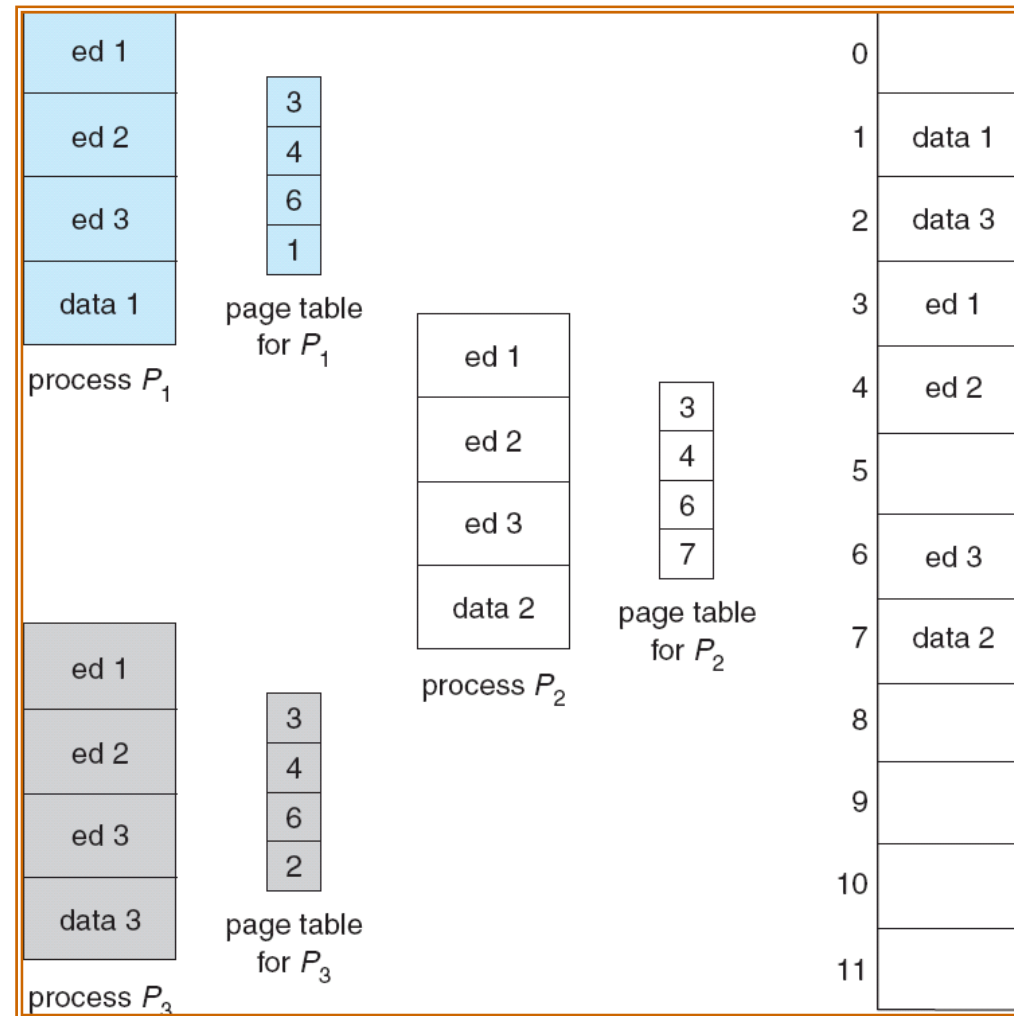
- One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
- Shared code must appear in same location in the logical address space of all processes

## ❑ 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

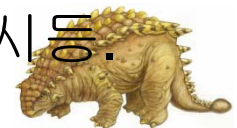


# Shared Pages Example



# Page Table : 개념

- 페이지 테이블 공간에 대한 오버헤드
  - Page Table은 프로세스당 운용
    - 페이지 4KB를 가지는 32비트 어드레스 공간의 페이지 테이블 사이즈는 4MB
    - 단일 공간에 할당될수 없을 가능성이 큼
- 오버헤드를 줄이는 방법
  - 실제 사용되는 주소 공간에 대해서만 맵을 할당함.
  - 전체 공간의 일부로 (Fraction) 유지.
- 어떻게 사용되는 영역에 대한 맵만 유지 할 수 있는가
  - 페이지 테이블을 다이나믹하게 확장 가능하도록 만듦
  - Indirection 레벨을 통해서 : 2레벨, 하이라키컬, 해시등.



# Page Table : 종류

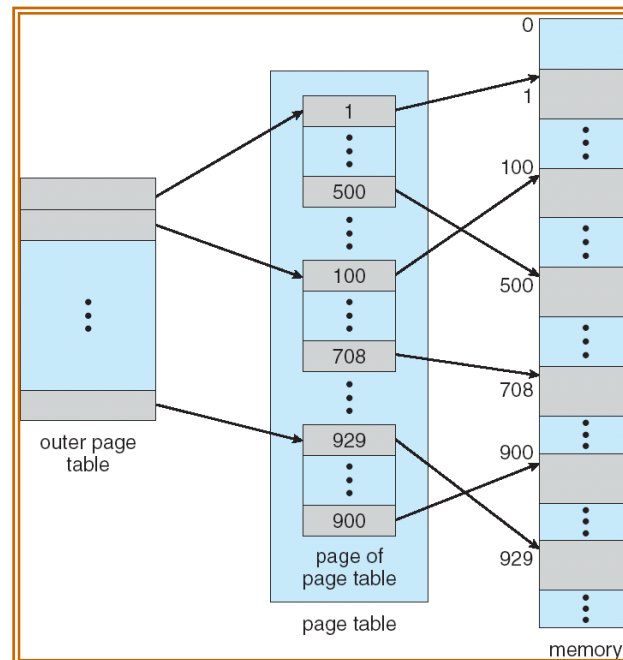
---

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables



# Page Table : Hierarchical Page Tables

- Break up the logical address space into multiple page tables
  - 2단계 페이지 테이블(Two Level Page Scheme)





# Page Table : Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:

page number		page offset
$p_1$	$p_2$	$d$
12	10	10

where  $p_1$  is an index into the **outer page** table, and  $p_2$  is **the displacement within the page** of the outer page table

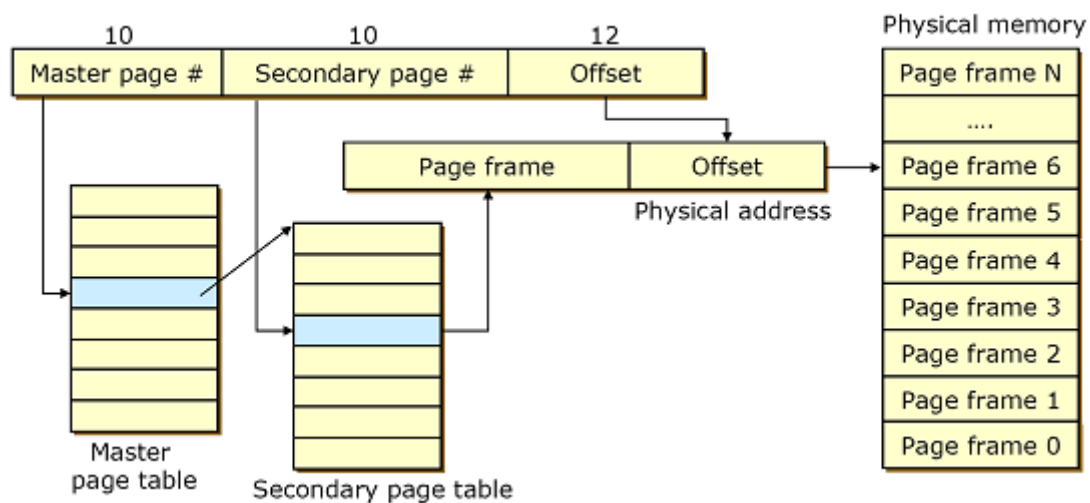
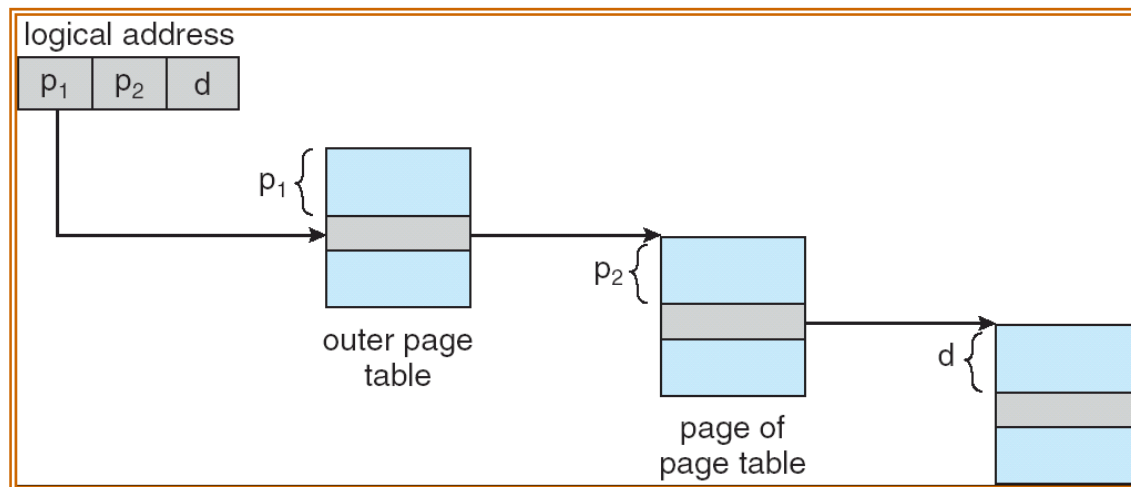


VAX : s(2), page(21), d(9)



# Page Table : Two-Level Paging Example

- 2단계 페이지 테이블에서 Address-Translation Scheme



# Page Table : Three-level Paging Scheme

64bit page table

outer page	inner page	offset
$p_1$	$p_2$	$d$
42	10	12

64bit page table : three level page table

2nd outer page	outer page	inner page	offset
$p_1$	$p_2$	$p_3$	$d$
32	10	10	12



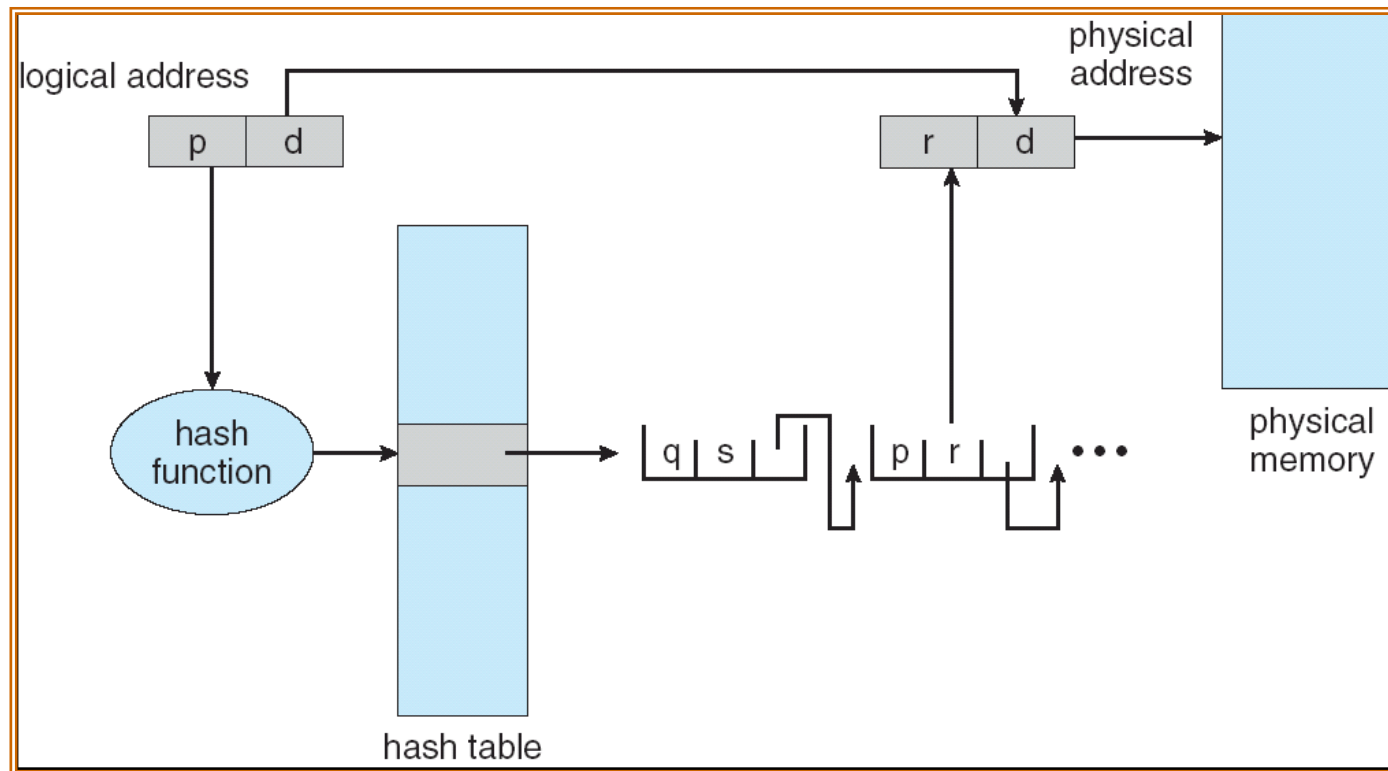
# Page Table : Hashed Page Tables

- 주소공간이 32 bits 보다 클 때
- The **virtual page number** is hashed into a page table. This page table contains a chain of elements hashing to the same location.
- Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.

**Cluster Page Table** : 64bits 시스템용 Page Table  
하나의 해쉬테이블 항목에 여러 개의 프레임 주소  
Sparse 공간에 사용



# Page Table : Hashed Page Table



# Page Table : Inverted Page Table

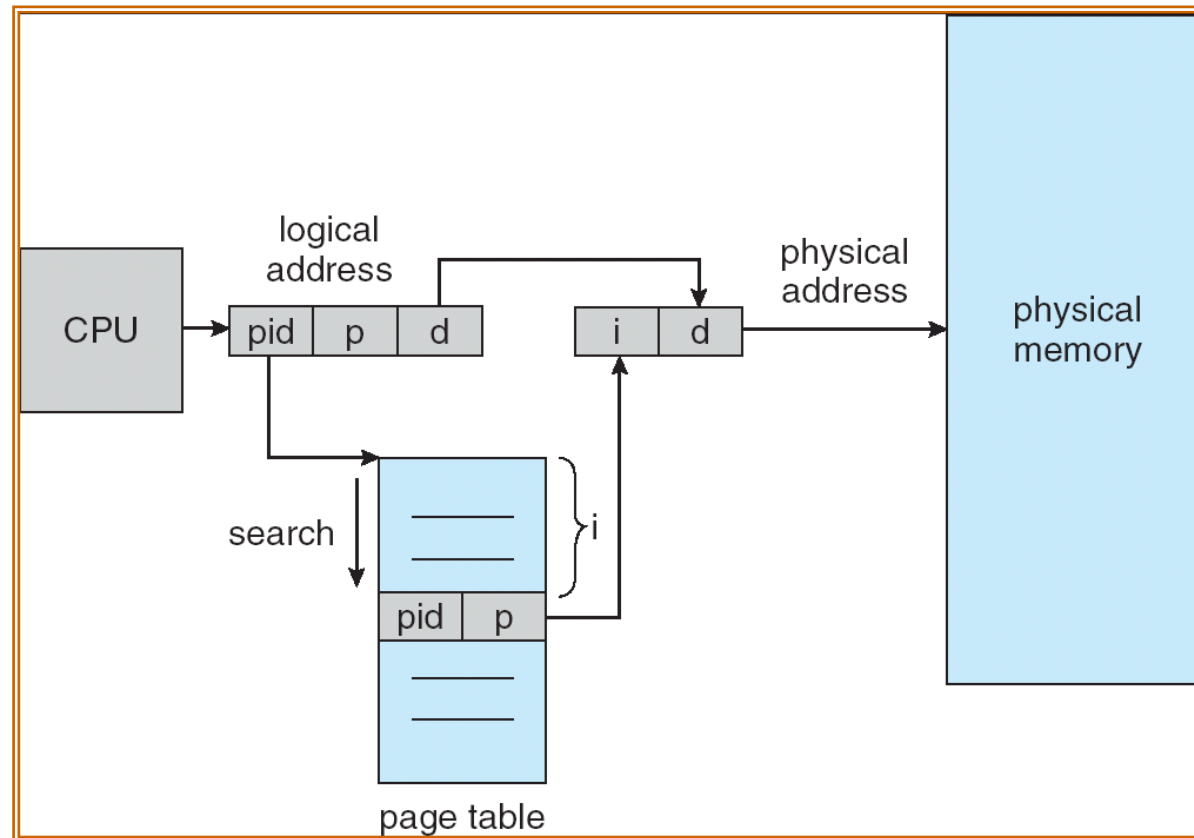
- 실제 메모리의 페이지가 하나의 엔트리가 됨.
  - 실제 메모리 페이지인 하나의 엔트리는 VPN과 프로세스 정보가 저장 되어 있음
    - 일반적으로 VPN이 인덱스가 되고 그 내에 매핑된 PPN이 기술 되는데 이는 VPN이 제공하는 공간 만큼의 엔트리가 필요페이지 테이블이 커짐 이에 반해 IPT는 VPN을 저장하고 인덱스가 PPN이 되므로 실제 물리 메모리 사이즈 만큼의 엔트리만 존재 하므로 페이지 테이블 사이즈가 매우 작음
- 대신 페이지를 찾기 위해 검색 시간이 들어가고 메모리 접근이 많아짐.
  - 이것을 해결 하기 위해서 Hashed Inverted Page Tables을 쓰거나 TLB와 같은 하드웨어 도움을 받음



64bit UltraSPARC, PowerPC



# Inverted Page Table Architecture



# Segmentation

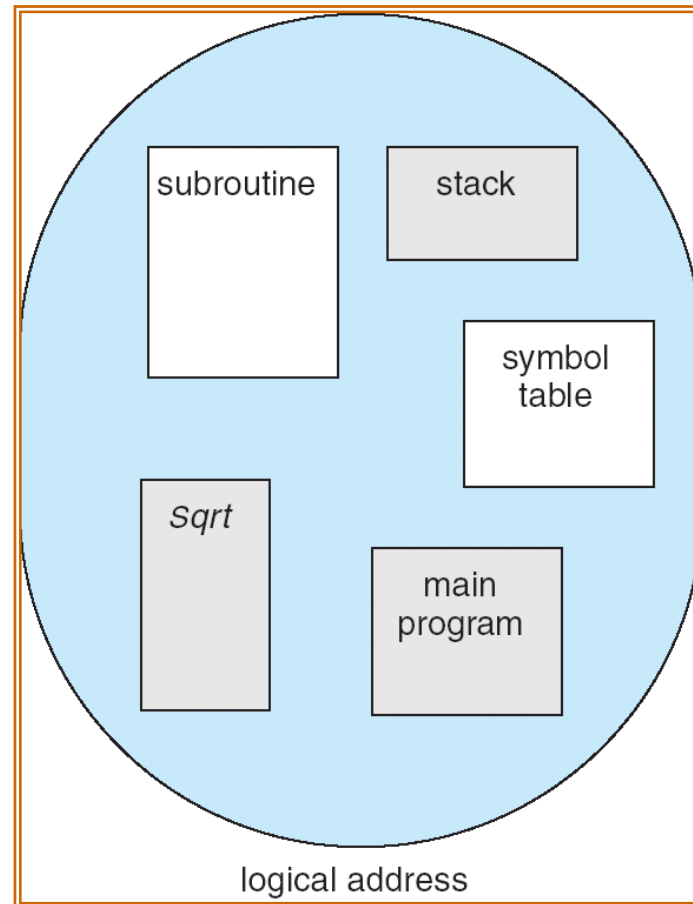
---

- 세그멘테이션 기법의 정의
  - Segment 단위로 메모리에 적재 할당하는 기법
- A program is a collection of segments. A segment is a logical unit such as:
  - main program,
  - procedure,
  - function,
  - method,
  - object,
  - local variables, global variables,
  - common block,
  - stack,
  - symbol table, arrays

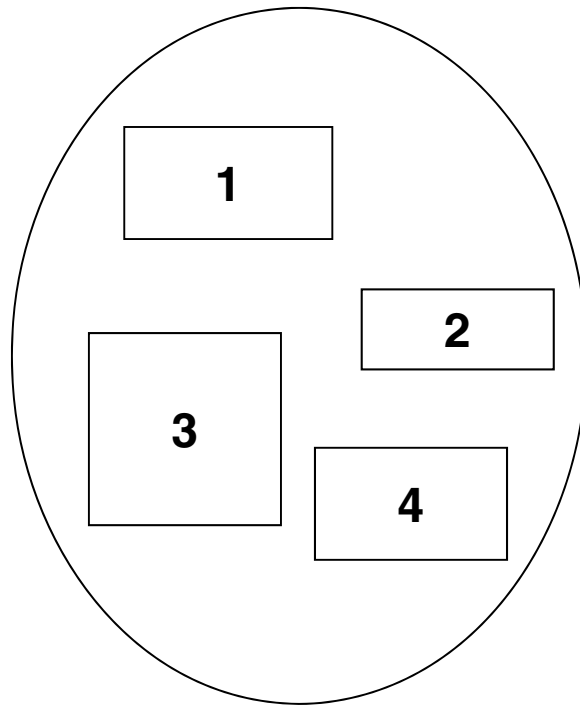




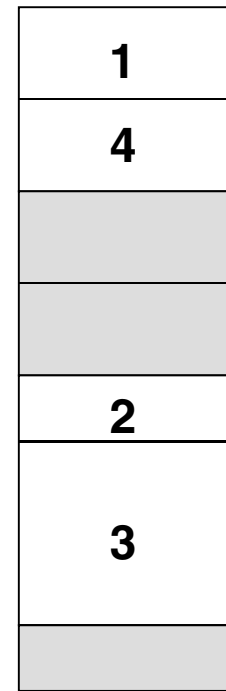
# User's View of a Program



# Logical View of Segmentation



**user space**



**physical memory space**

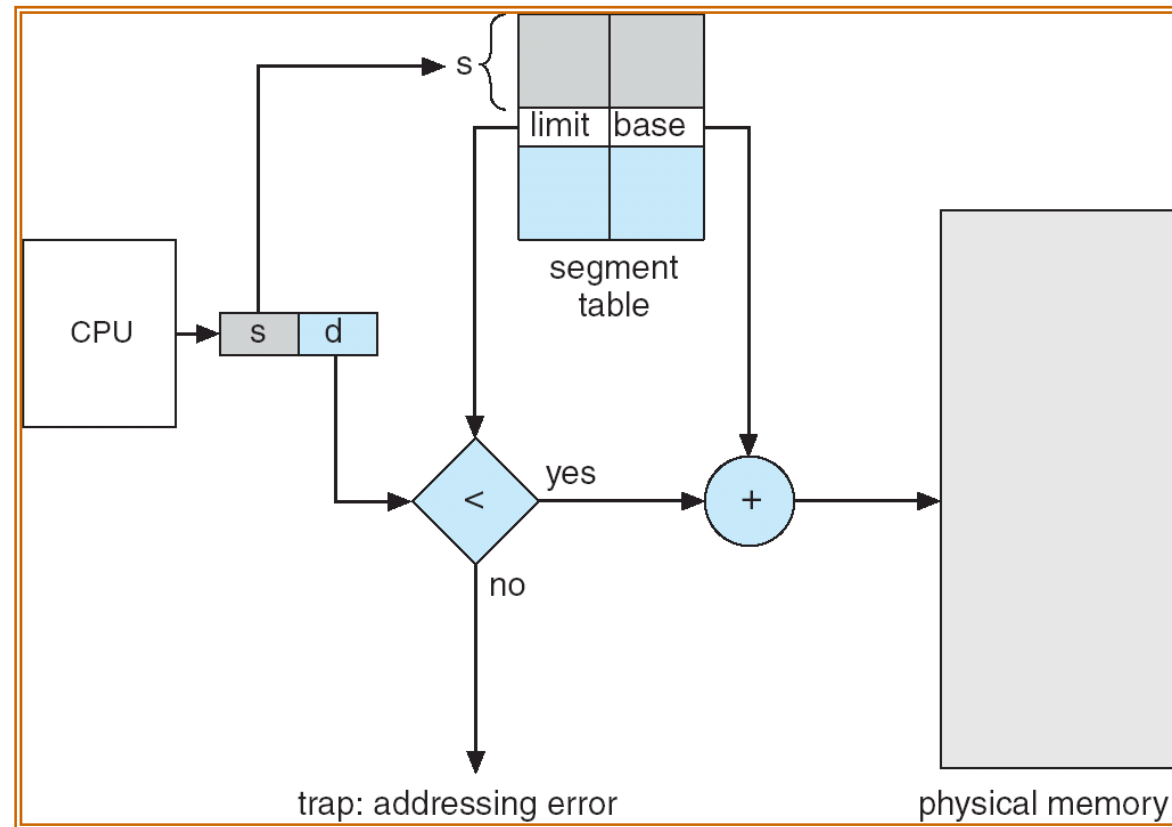


# Segmentation Architecture

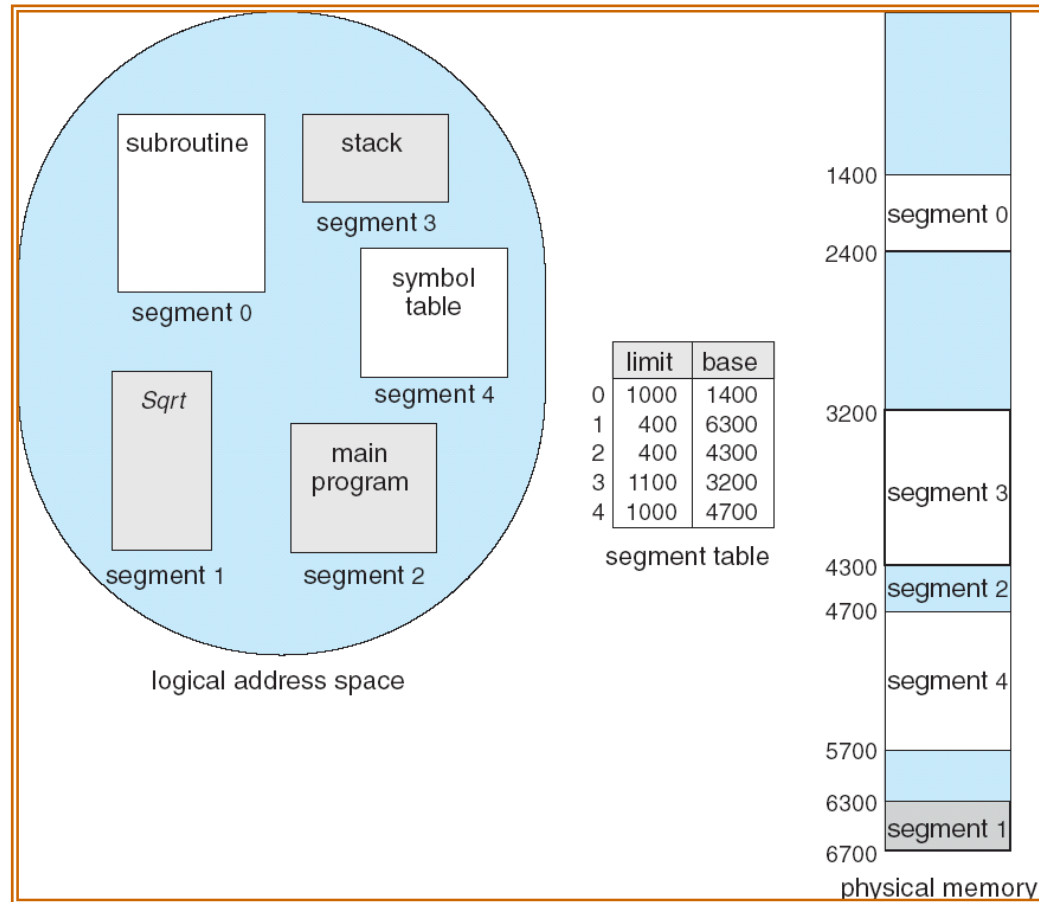
- ❑ Logical address consists of a two tuple:  
    <segment-number, offset>,
- ❑ **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment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



# Segmentation Hardware



# Example of Segmentation



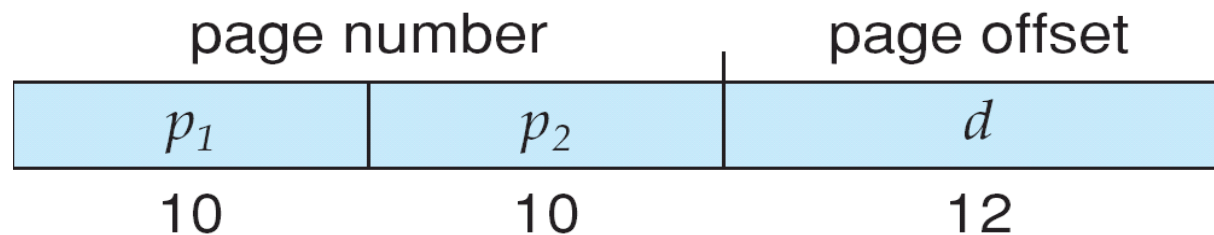
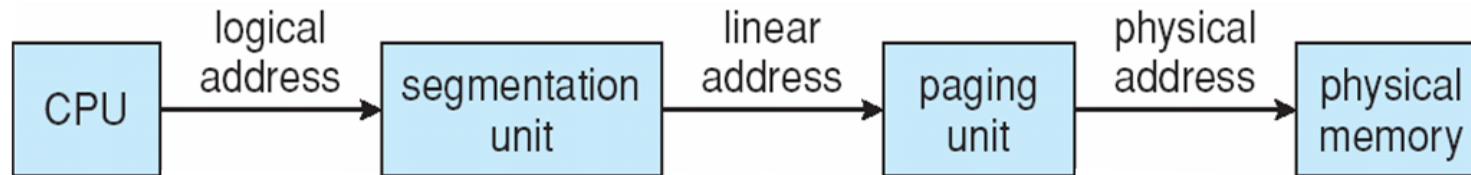
# Example: The Intel Pentium

---

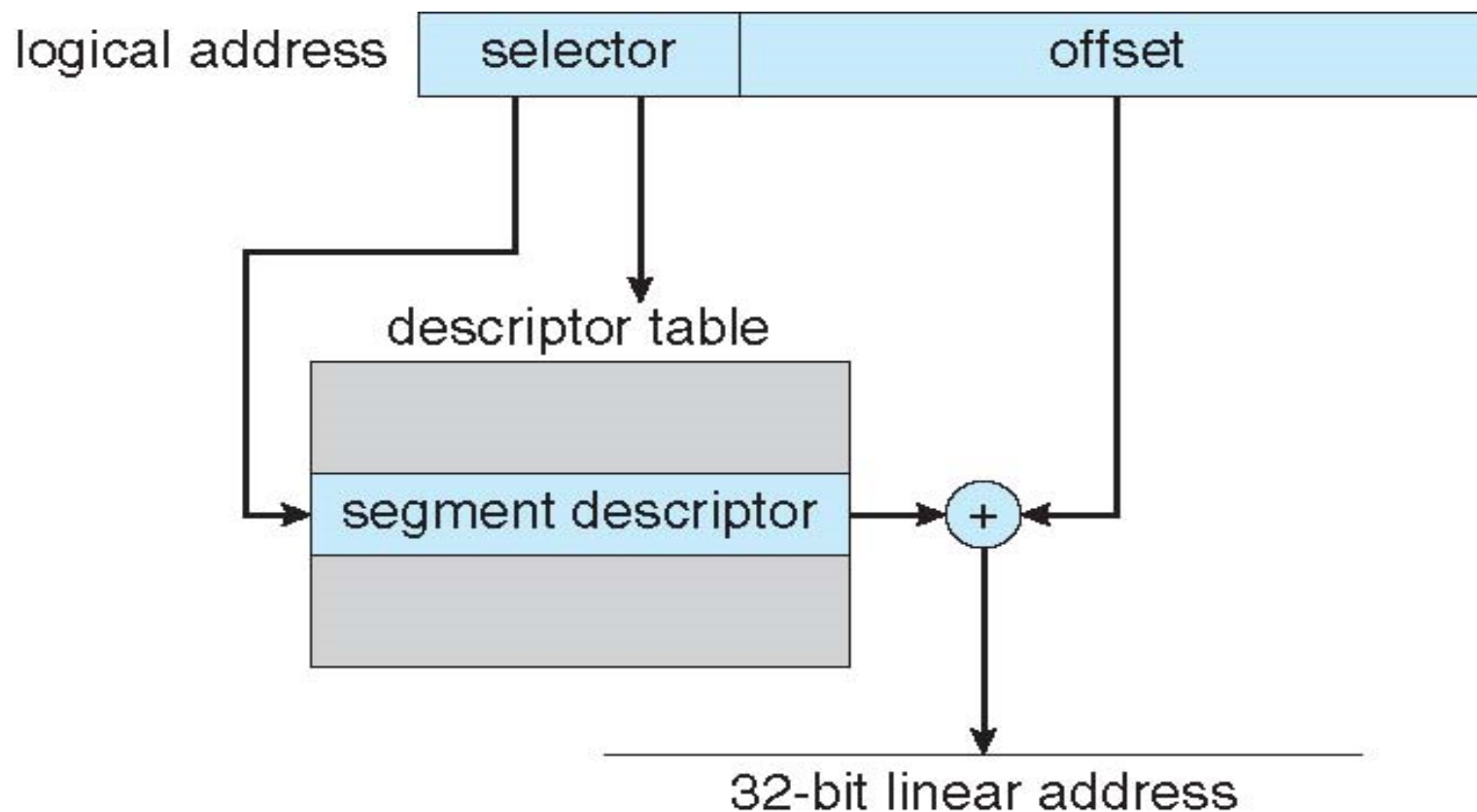
- ❑ Supports both segmentation and segmentation with paging
  - Each segment can be 4 GB
  - Up to 16 K segments per process
  - Divided into two partitions
    - ❑ First partition of up to 8 K segments are private to process (kept in **local descriptor table LDT**)
    - ❑ Second partition of up to 8K segments shared among all processes (kept in **global descriptor table GDT**)
  
- ❑ CPU generates logical address
  - Given to segmentation unit
  - Linear address given to paging unit
    - ❑ Pages sizes can be 4 KB or 4 MB



# Logical to Physical Address Translation in Pentium

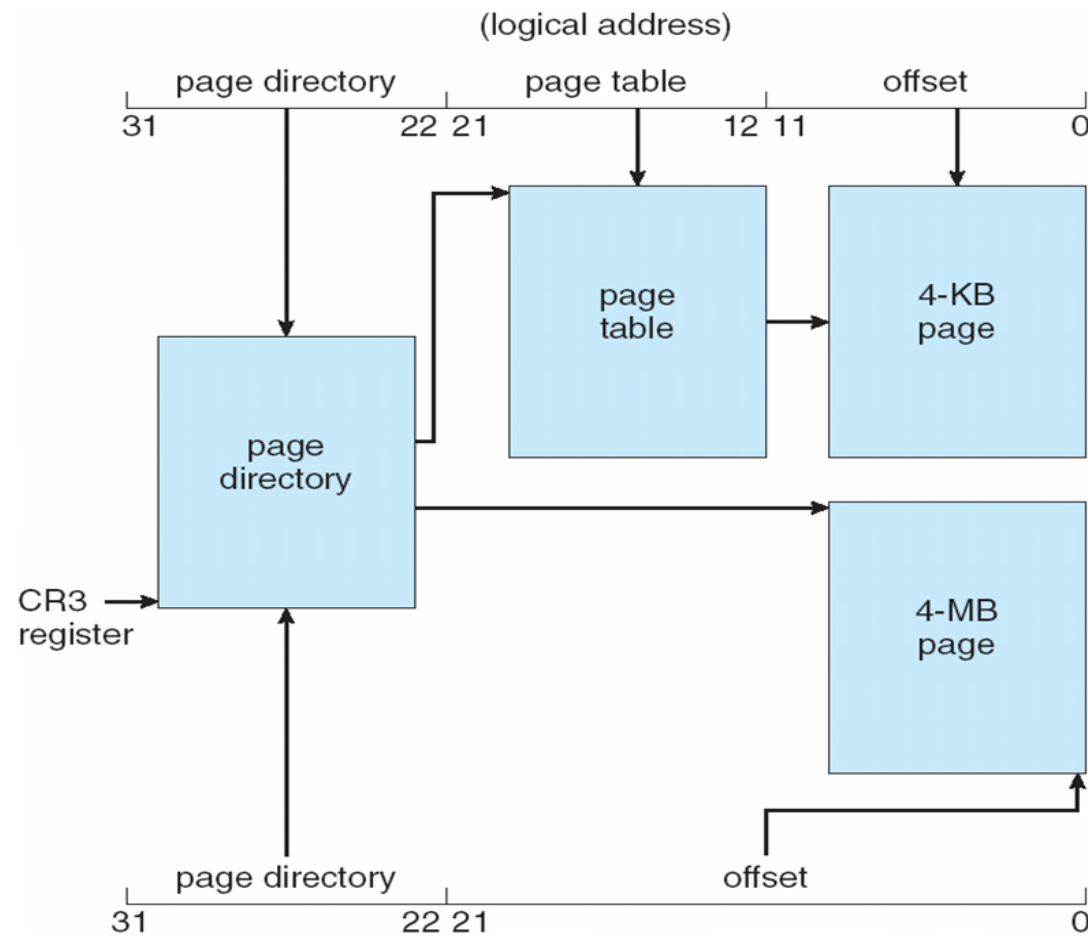


# Intel Pentium Segmentation





# Pentium Paging Architecture



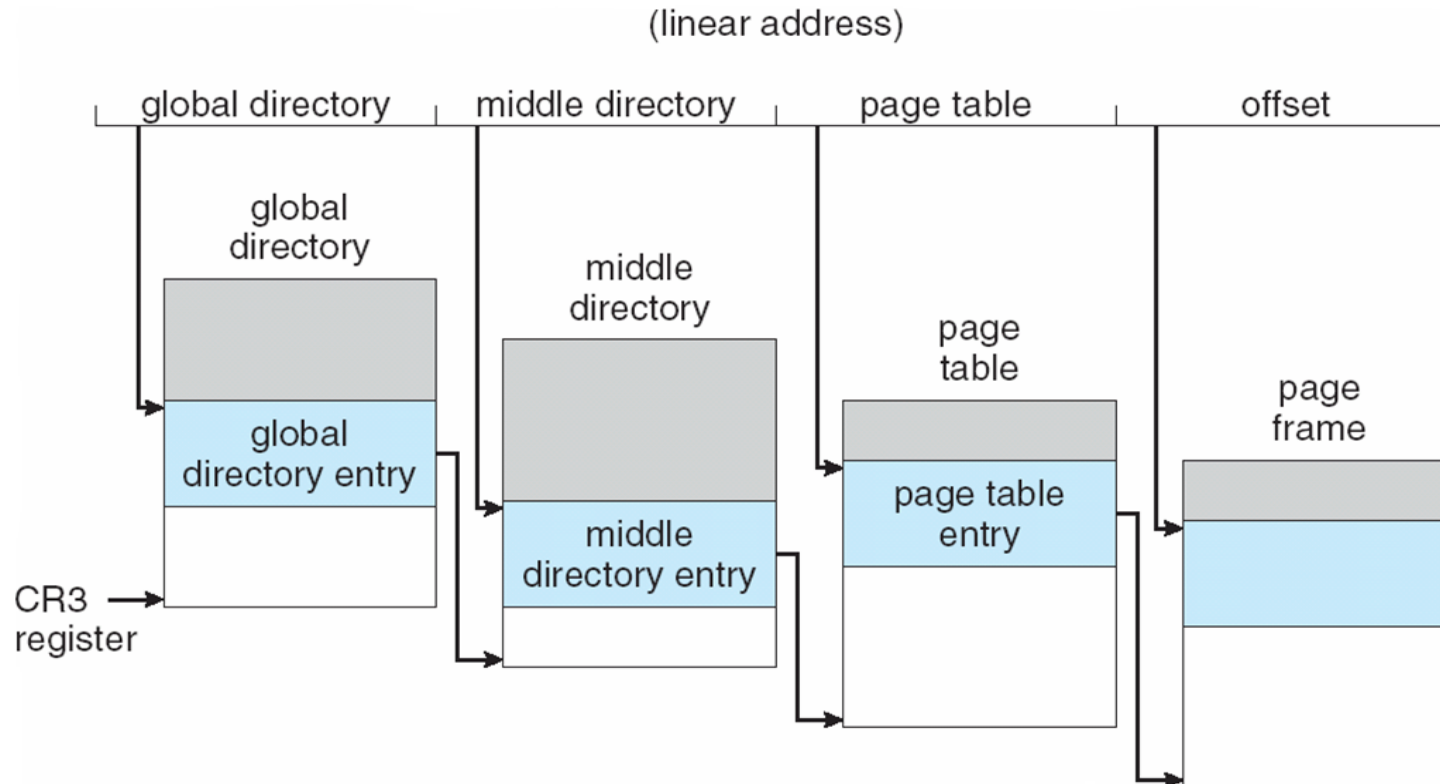
# Linear Address in Linux

- n Linux uses only **6 segments**
  - n kernel code, kernel data, user code, user data, task-state segment (TSS), default LDT segment
- n Linux only uses two of four possible modes – kernel and user
- n Uses a three-level paging strategy that works well for 32-bit and 64-bit systems
- n Linear address broken into four parts:
- n But the Pentium only supports 2-level paging?!

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



# Three-level Paging in Linux



# End of Chapter 8

---

