고려대학교
KOREA UNIVERSITY

KU-The Future

# Operating Systems (10th Ed., by A. Silberschatz)

## Chapter 9 Main Memory

**Heonchang Yu**

**Distributed and Cloud Computing Lab.**

# Contents

- Background
- Contiguous Memory Allocation
- Paging *memory Allocation*의 대표 사례로 *paging*이 언급된다
- Structure of the Page Table
- Swapping
- Example: Intel 32- and 64-bit Architectures
- Example: ARMv8 Architecture

# Objectives

- Explain the difference between a logical and a physical address and the role of the memory management unit (MMU) in translating addresses.
- Apply first-, best-, and worst-fit strategies for allocating memory contiguously.
- Explain the distinction between internal and external fragmentation.
- Translate logical to physical addresses in a paging system that includes a translation look-aside buffer (TLB).
- Describe hierarchical paging, hashed paging, and inverted page tables. *paging*기법이 다양하다. *9*장에서 모두 다룰 내용들
- Describe address translation for IA-32, x86-64, and ARMv8 architectures.

# Background

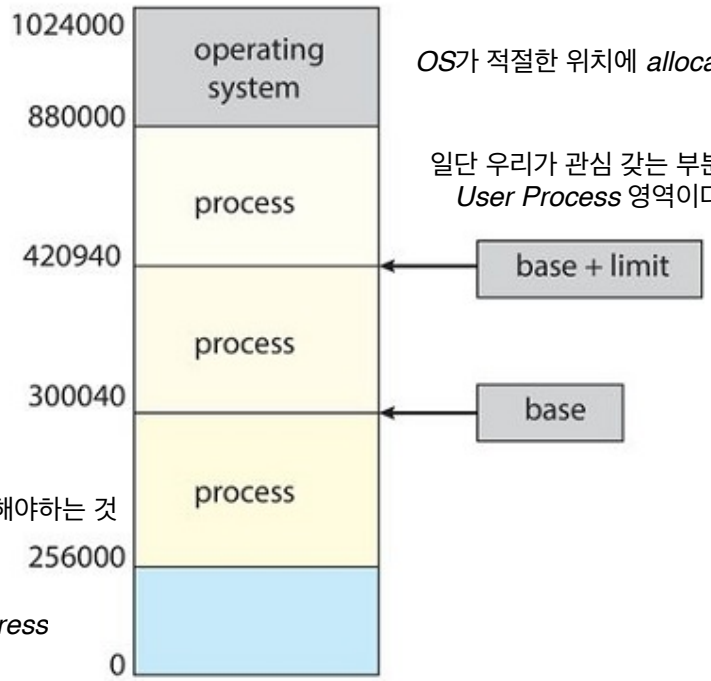
- Basic hardware 멀티프로그래밍이 지원되는 환경이기 때문에 여러개의 프로세스가 상주 가능하다 *(프로세서 –코어 – 갯수와는 관계 없음)*
  - Each process has a separate memory space.
    - ✓ We need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses.
    - ✓ Base register : the smallest legal physical memory address
    - ✓ Limit register : the size of the range 프로세스의 *capacity*

서로다른 프로세스들의 영역을 구분할 수 있을 것이다.

어떤 특정 프로세스가 연속적인 공간*(continuous)*을 할당 받아 저장한다는 것은 유저의 관점

*OS* 입장에서 중요한 것은
*Process i & j*의 영역이 겹치지 않게 해야하는 것

*base + limit = last address*




*OS*가 적절한 위치에 *allocation*

일단 우리가 관심 갖는 부분은
*User Process* 영역이다

**Figure 9.1** A base and a limit register define a logical address space.

# Background

- Basic hardware
  - Protection of memory space
    - ✓ This scheme prevents a user program from modifying the code or data structures of either the operating system or other users.

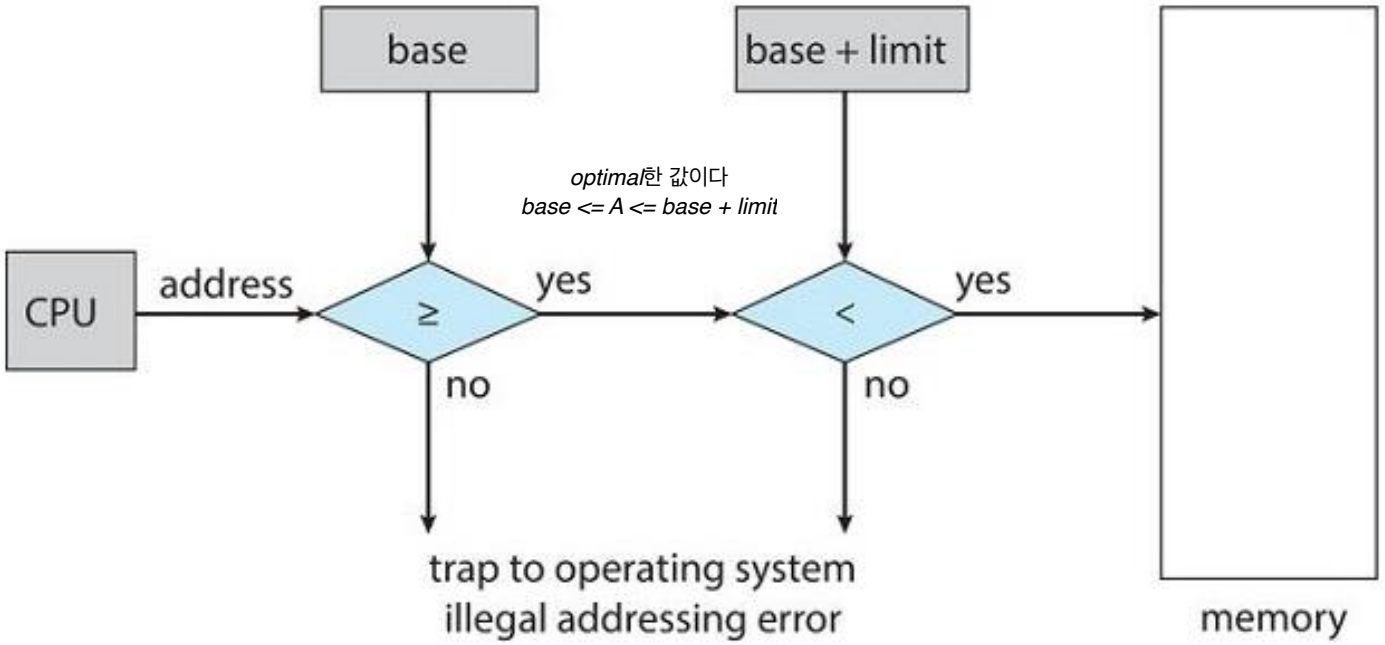


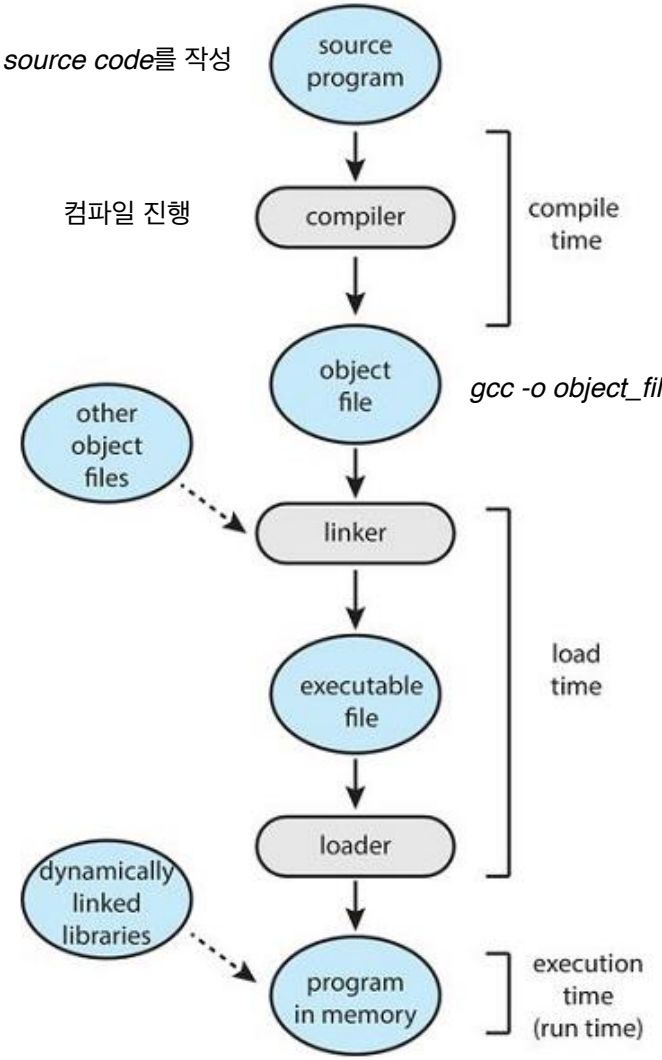

**Figure 9.2** Hardware address protection with base and limit registers.

# Background

- Address Binding
  - To run, the program must be brought into memory and placed within the context of a process.
  - A user program will go through several steps before being executed.
  - Addresses may be represented in different ways during these steps.
    - ✓ Addresses in the source program are generally symbolic. 주소를 명명하는 방법들이 단계마다 다를 수 있다.
    - ✓ A compiler binds these symbolic addresses to relocatable addresses. 컴파일러 -> *relocatable* 주소로 *binding*
    - ✓ The linker or loader binds the relocatable addresses to absolute addresses. *absolute address :* 절대 주소 *relocatable address :* 상대 주소값

# Background

일반 유저가 *source code*를 작성




컴파일 진행

*compiler, linker, state* 값은 물리적으로 인지할 수 있는 주소 값을 인지할 때까지 행위를 지속하는 시스템 소프트웨어

*gcc -o object_file_name source_file*

물리적으로 인지할 수 있는 주소 값을 생성하는 과정
*Ch 9*에서는 결국 물리적 주소로 *mapping*하는 과정을 이해하는 것

**Figure 9.3** Multistep processing of a user program.

# Background

유저 관점의 주소가 어떻게 시스템 관점의 주소로 변경되는가?

- Address Binding
  - The binding of instructions and data to memory addresses can be done at any step along the way.
    - ✓ Compile time: If you know at compile time where the process will reside in memory, then absolute code can be generated.
    - ✓ Load time: If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code.
    - ✓ Execution time: If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.

# Background

- Logical versus Physical Address Space
  - Binding addresses at either compile or load time generates identical logical and physical addresses.
  - The execution-time address-binding scheme results in differing logical and physical addresses.
    - ✓ Logical address (virtual address): An address generated by the CPU
    - ✓ Physical address: An address seen by the memory unit (that is, the one loaded into the memory-address register of the memory)

메인 메모리에 할당되는 것을 어떻게 *physical address*로 매핑을 해줄 것인가?

  - The run-time mapping from virtual to physical addresses is done by a hardware device called the memory-management unit (MMU)
    - ✓ The value in the relocation register (base register) is added to every address generated by a user process at the time the address is sent to memory.
    - ✓ The user program never accesses the real physical addresses.

# Background

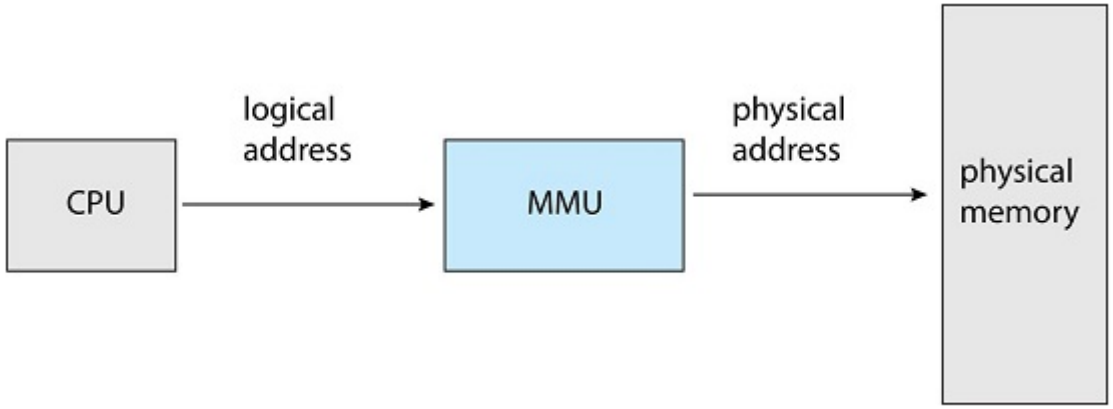



**Figure 9.4** Memory management unit (MMU).

*physical address*를 매핑해주는 것이 *paging* 기법!

- ✓ If the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346.

프로세스의 시작 주소를 갖고 있는 *register*

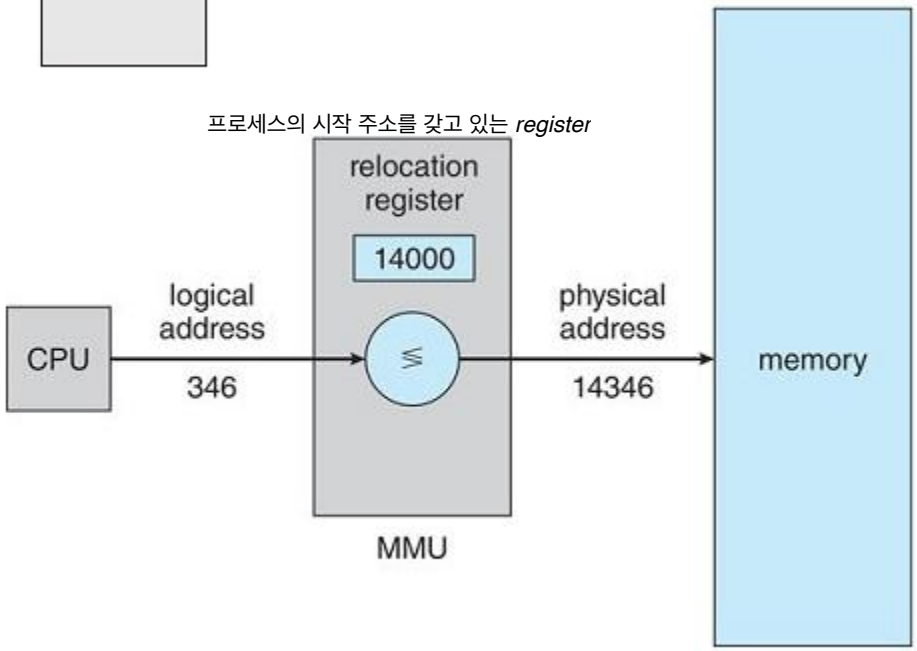



**Figure 9.5** Dynamic relocation using a relocation register.

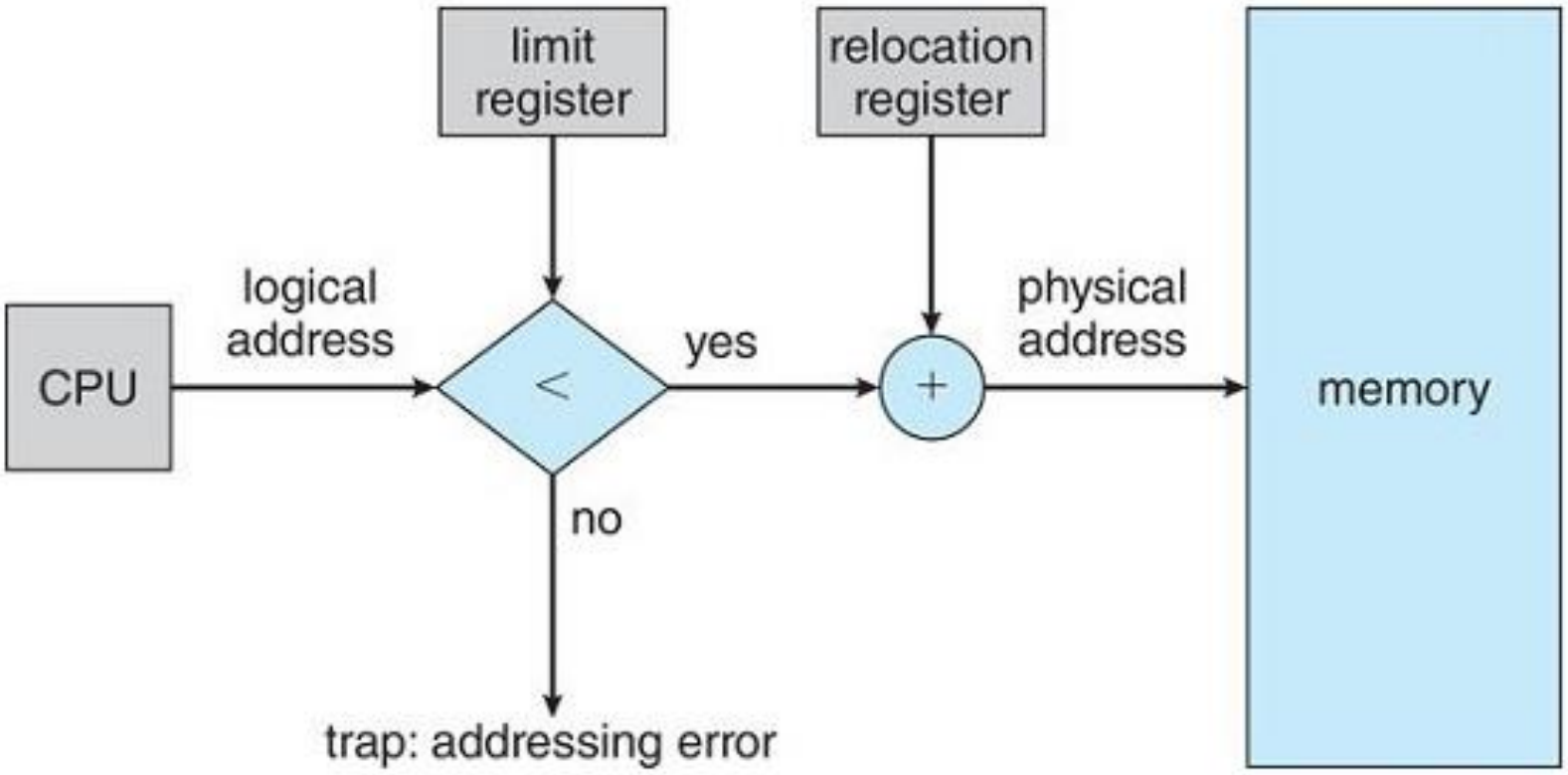
# Continuous Memory Allocation

- The memory is usually divided into two partitions.
  - ✓ One for the resident operating system
  - ✓ One for the user processes
- We usually want several user processes to reside in memory at the same time.
  - ✓ We therefore need to consider how to allocate available memory to the processes that are waiting to be brought into memory.

– Memory Protection

- Relocation register : the value of the smallest physical address
- Limit register : the range of logical addresses
- Each logical address must fall within the range specified by the limit register.
- The MMU maps the logical address dynamically by adding the value in the relocation register.

# Continuous Memory Allocation




*Trapping :* 에러가 발생한다

**Figure 9.6** Hardware support for relocation and limit registers.

# Continuous Memory Allocation

- Memory Allocation
  - Multiple-partition method

*page* 기법이 사용되고 있기 때문에 이 기법은 잘 사용 안된다

*partition*에 프로세스를 할당하고 남은 공간은 다른 프로세스를 위해 사용할 수 없음
결국 프로세스의 비용으로 남게 된다

- ✓ It is to divide memory into several fixed-sized partitions.
- ✓ Each partition may contain exactly one process.
- ✓ The degree of multiprogramming is bound by the number of partitions.

아마 *partition*보다 작은 값이 할당되게 될 것

- ✓ When a partition is free, a process is selected from the input queue and is loaded into the free partition.
- ✓ When the process terminates, the partition becomes available for another process.

# Continuous Memory Allocation

- Memory Allocation
  - In the variable-partition scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied.
    - ✓ All memory is available for user processes and is considered one large block of available memory, a **hole**.

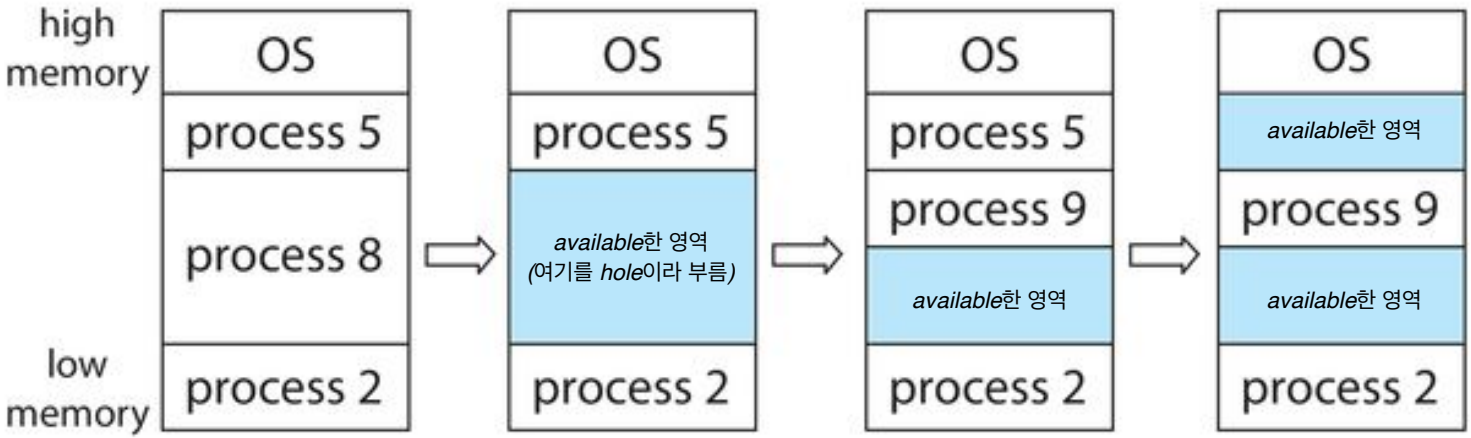



**Figure 9.7** Variable partition.

# Continuous Memory Allocation

- Memory Allocation
  - 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 (충분한 사이즈의 첫번째 *hole*) ; we can stop searching as soon as we find a free hole that is large enough
    - ✓ Best fit : allocate the smallest hole that is big enough ; we must search the entire list, unless the list is ordered by size ; this strategy produces the smallest leftover hole (최적의 *hole*을 찾는 것)
    - ✓ Worst fit : allocate the largest hole ; we must search the entire list, unless it is sorted by size. (가장 큰 *hole*)
  - Both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization.

# Continuous Memory Allocation

– Fragmentation

- Both the first-fit and best-fit strategies for memory allocation suffer from external fragmentation.
- **External fragmentation** exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous.
  - ✓ Storage is fragmented into a large number of small holes.
- Depending on the total amount of memory storage and the average process size, external fragmentation may be a minor or a major problem.
- The general approach to avoiding the fragmentation problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size.
- The memory allocated to a process may be slightly larger than the requested memory – the difference between these two numbers is **internal fragmentation**.

각 파티션은 정확히 하나의 프로세스만 들어가는 것이기 때문이 이 경우에 자투리로 남는 프로세스는 다른 프로세슬 위해 제한 될 수 없어 비용으로 남는다

# Continuous Memory Allocation

- Fragmentation
  - Compaction *memory allocation*을 할 때 *continuous* 하게 할당이 필요하기 때문에 *compaction*이 필요할 수 있음
    - ✓ One solution to the problem of external fragmentation
    - ✓ Is to shuffle the memory contents so as to place all free memory together in one large block
    - ✓ Is possible only if relocation is dynamic and is done at execution time 컴파일을 다시하면 오버헤드가 발생할 수 있음
      *complie or linking* 에서 이뤄지면 그것은 이전 단계임.
      때문에 실행 시에 이뤄진다는 것은 *remapping*되고, 컴퍼일 단계를 스킵하는 것임
    - ✓ When compaction is possible, we must determine its cost.
      - ➢ The simplest compaction algorithm is to move all processes toward one end of memory.
      - ➢ All holes move in the other direction, producing one large hole of available memory.

# Paging

- Paging is a memory-management scheme that permits a process's physical address space to be noncontiguous.
- Paging avoids external fragmentation.

– Basic Method

아키텍처 사이즈로 페이지도 나뉜다

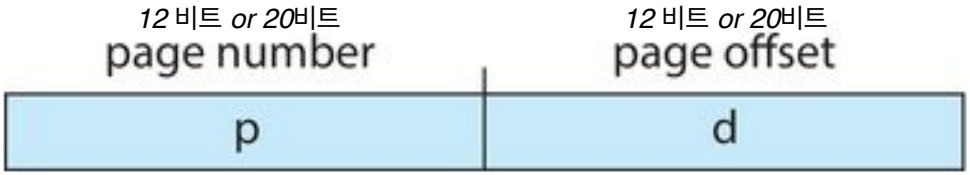
- Breaking physical memory into fixed-sized blocks called **frames** and breaking logical memory into blocks of the same size called **pages**
- Every address generated by the CPU is divided into two parts: a page number (p) and a page offset (d)

*logical memory* 는 유저 입장에서 보는 것

  - ✓ The page number is used as an index into a page table.
  - ✓ The page table contains the base address of each frame in physical memory. This base address of the frame is combined with the page offset to define the physical memory address.

페이지 테이블 내의 특정 위치를 나타내는 것이 *table number*

*logical address*의 주소 체계가 *32 bit or 64 bit*인지 상관 없이 *page number & page offset*으로 나눠짐




*Logical Address* 주소 체계에 따라 구조를 나눠서 *addressing*을 진행한다.

*n*개의 페이지 갯수를 말할 때 *page number*

전체 *32* 비트

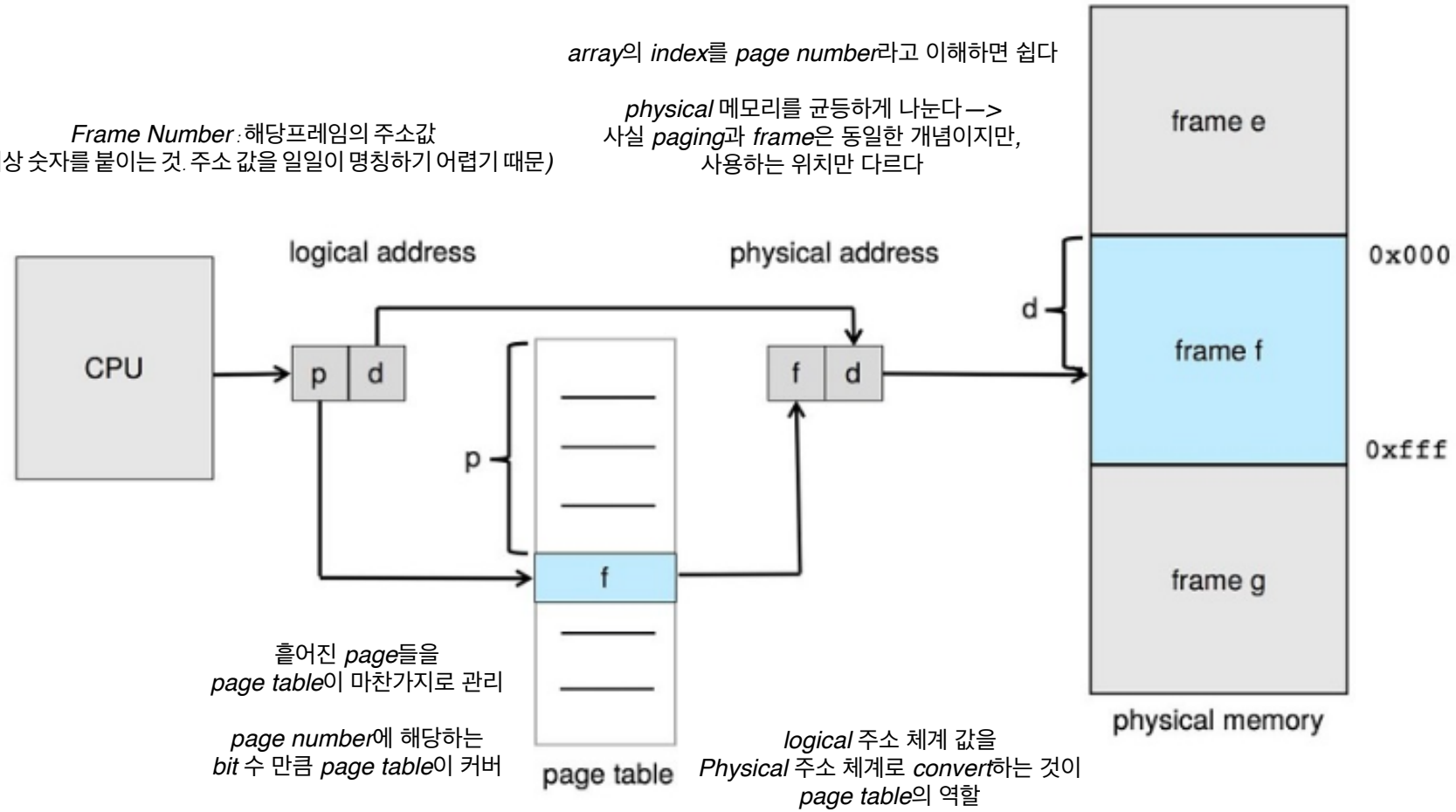
페이지 하나의 사이즈를 나타내는 것이 *page offset*

# Paging

*array*의 *index*를 *page number*라고 이해하면 쉽다

*physical* 메모리를 균등하게 나눈다 —>
사실 *paging*과 *frame*은 동일한 개념이지만,
사용하는 위치만 다르다

*Frame Number* : 해당프레임의 주소값
*(*편의상 숫자를 붙이는 것. 주소 값을 일일이 명칭하기 어렵기 때문*)*




흩어진 *page*들을
*page table*이 마찬가지로 관리

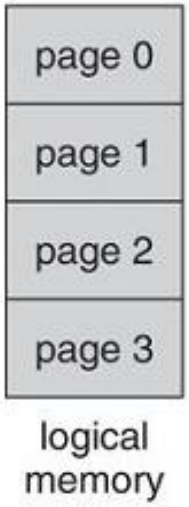
*page number*에 해당하는
*bit* 수 만큼 *page table*이 커버

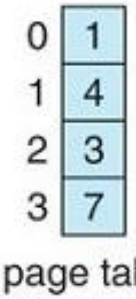
*logical* 주소 체계 값을
*Physical* 주소 체계로 *convert*하는 것이
*page table*의 역할

**Figure 9.8** Paging hardware.

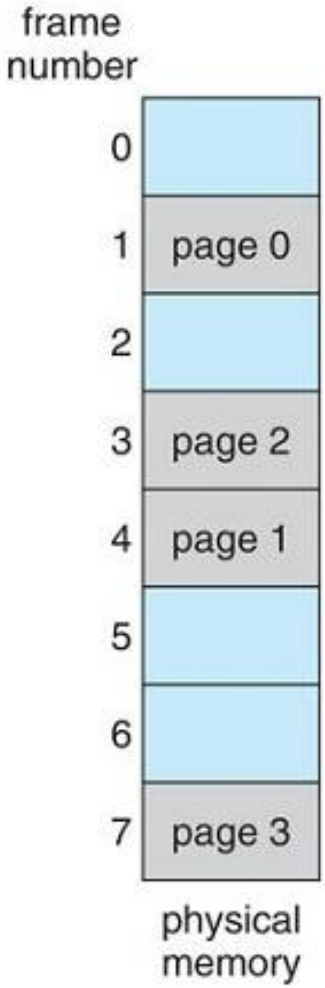
# Paging

*Logical Memory*는 여기서
*Pi*하나고
*4*개의 페이지로 나눠짐




메인 메모리의 어디에 주소가
있는지 확인하는 것이 *page table*

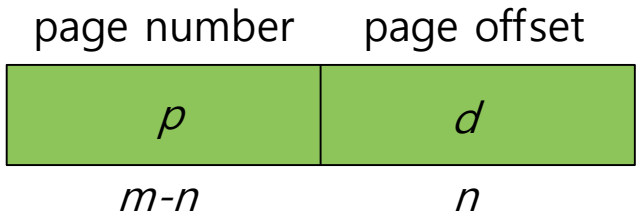



*Frame Number :* 해당프레임의 주소값
*(*편의상 숫자를 붙이는 것.
주소 값을 일일이 명칭하기 어렵기 때문*)*

**Figure 9.9** Paging model of logical and physical memory.

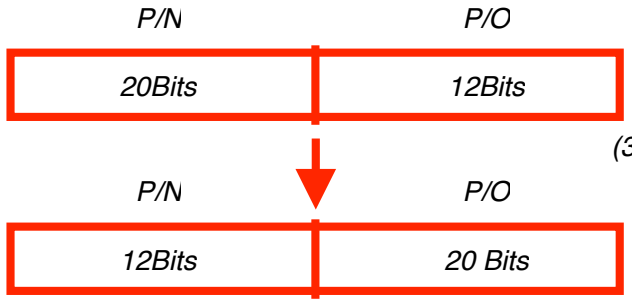
# Paging

- The size of a page is typically a power of 2, varying between 4 KB and 1 GB per page.
- If the size of the logical address space is $2^m$ and a page size is $2^n$ bytes, then the high-order *m-n* bits of a logical address designate the **page number**, and the *n* low-order bits designate the **page offset**.
- *p* is an index into the page table and *d* is the displacement within the page.




0부터 2^m-1까지 block이 됨

전체가 m비트라고 봤을 때
page size가 n




(32비트 환경에서 가정)

Page offset이 커진다는 것은 Page size가 커진다는 것
Page Table은 20비트에서 12 비트로 감소
(table사이즈 감소, 관리되는 page 갯수가 줄어든다)

그러나 Table에서 관리하는 page 갯수가 줄면서
disk access time이 줄어 더 효율적임

(table 용량이 300MB이라 할때,
30MB 10개 와 50MB 6개가 들어오면
50MB 6개가 disk access time을 줄임)

# Paging

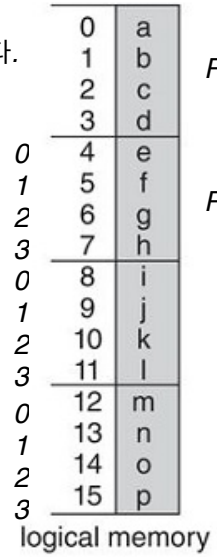
- Figure 9.10 – in the logical address, $n$ = 2 and $m$ = 4. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages).
  - ✓ Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 [=(5x4)+0].
  - ✓ Logical address 3 (page 0, offset 3) maps to physical address 23 [=(5x4)+3].
  - ✓ Logical address 4 (page 1, offset 0) maps to physical address 24 [=(6x4)+0].
  - ✓ Logical address 13 (page 3, offset 1) maps to physical address 9 [=(2x4)+1].

여기 예제들이 아래 페이지의 그림으로 확인이 가능하다
예제 실습 반드시 필요하다

# Paging

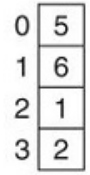
*page offset*은 시작 위치에서 항상 *0*이다.
*page sizing* 되는 게 *4*개

*page offset* 값이 *2*비트이므로
*0,1,2,3* 형태로 사이즈가 넣어짐




*Page Size 1*

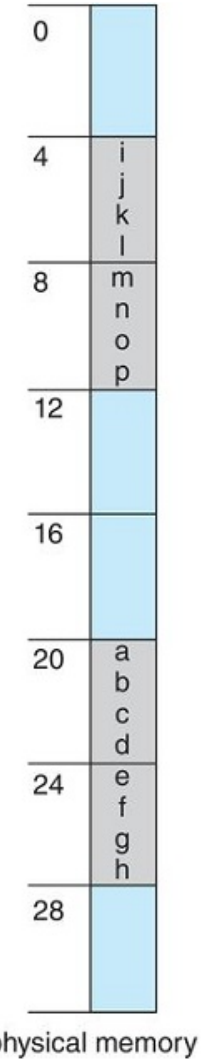
*Page Size 2*




page table

*page*의 매핑은
*page table*을 통해 확인한다

*0 ~ 15 : Logical Address*




*0 ~ 7*개의 주소 공간
*: Frame Number*

*a*의 *logical address*는 *0*
*physical address*는 *20*

총 *8*개의 *Physical Memory* 공간이 있음

*k*의 *logical address*는 *10*
*physical address*는 *6*
*1 * 4 + 2*

**Figure 9.10** Paging example for a 32-byte memory with 4-byte pages.

# Paging

*External Fragmentation*에서 *Paging*은 발생하지 않는다

퍼팩트하게 마지막 페이지까지 딱 사이즈를 맞춰서 사용하는 것은 거의 불가능하다.
어느 정도의 *internal fragementation*은 감수해야한다.

- No external fragmentation, but some internal fragmentation
- If the memory requirements of a process do not happen to coincide with page boundaries, the last frame allocated may not be completely full.

*page number* = 전체 *m* 크기 - *2^11*
페이지 사이즈 = *page offset* (여기서 *2^11*, *11*비트)

*72,766 / 2,048* = 대략 *35*

  - ✓ If page size is 2,048 bytes, a process of 72,766 bytes will need 35 pages plus 1,086 bytes. It will be allocated 36 frames, resulting in internal fragmentation of 2,048-1,086=962 bytes.

*internal fragmentation*, *page*는 *36*개다

- We expect internal fragmentation to average one-half page per process. The consideration suggests that small page sizes are desirable.

*35*개 까지는 꽉 채우고,
*36*번째 페이지에서는 *1,086 Byte*까지 채우고,
*962 Byte*는 *internal fragmentation*이 발생함

*page-table entry* = 페이지 테이블 *index*

- Overhead is involved in each page-table entry, and this overhead is reduced as the size of the pages increases. Also, disk I/O is more efficient when the amount data being transferred is larger.

페이지 사이즈 커진다 => *disk I/O*는 좋아진다.

- Generally, page sizes have grown over time as processes, data sets, and main memory have become larger.

# Paging

- The frame table has one entry for each physical page frame, indicating whether the latter is free or allocated.

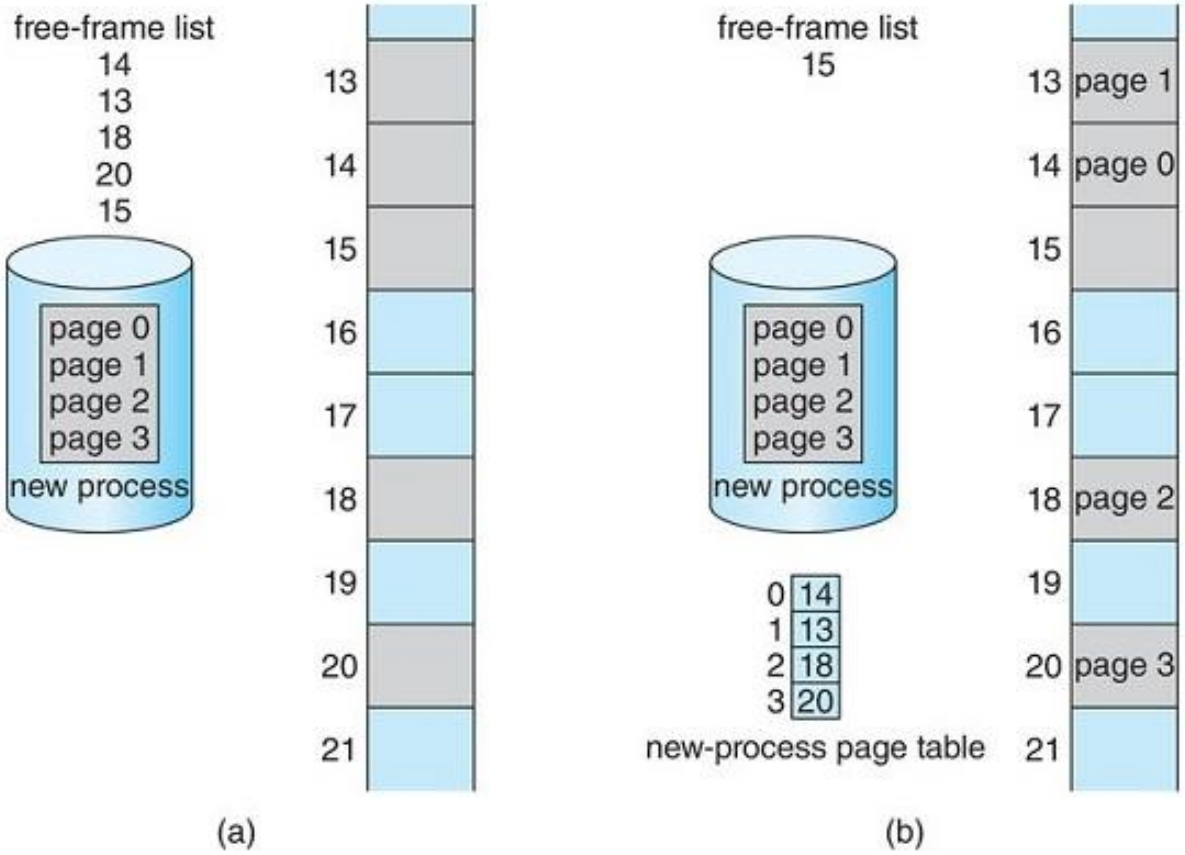



**Figure 9.11** Free frames (a) before allocation and (b) after allocation.

# Paging

*memory access*는 기본인데, *page access* 횟수에 대해 생각해보자

- Hardware Support (HW implementation of the page table)
  - The page table is implemented as a set of dedicated high-speed hardware registers, which makes the page-address translation very efficient.
    - ✓ The use of registers for the page table is satisfactory if the page table is reasonably small. Most contemporary CPUs, however, support much larger page tables.
  - The page table is kept in main memory.
    - ✓ A page-table base register(PTBR) points to the page table.
    - ✓ The problem is the time required to access a user memory location (Two memory accesses are needed to access data.)
      - ❖ If we want to access location *i*, we must first index into the page table, using the value in PTBR offset by the page number for *i*.
      - ❖ It provides us with the frame number, which is combined with the page offset to produce the actual address.

# Paging

- Hardware Support

굳이 *TLB*는 사용할 필요는 없지만,
시간적인 측면에서 소요되는 것을 줄여보자는 취지

  - The standard solution is to use a special, small, fast-lookup hardware cache, called a translation look-aside buffer(TLB).

*TLB*는 결국 일종의 캐시라고 보면 된다.

    - ✓ Each entry in the TLB consists of two parts: a key(or tag) and a value.
    - ✓ The TLB is used with page tables: TLB contains only a few of the page-table entries.

어떤 *entry*가 관리되는지는 논외로 생각하자 *(*여기서 언급할 필요 전혀 없음*)*

      - ❖ If the page number is found, its frame number is immediately available and is used to access memory.
      - ❖ If the page number is not in the TLB (known as a TLB miss), a memory reference to the page table must be made.
    - ✓ If the TLB is already full of entries, an existing entry must be selected for replacement. Replacement policies range from least recently used (LRU) through round-robin to random.
    - ✓ Some TLBs allow certain entries to be wired down, meaning that they cannot be removed from the TLB.

# Paging

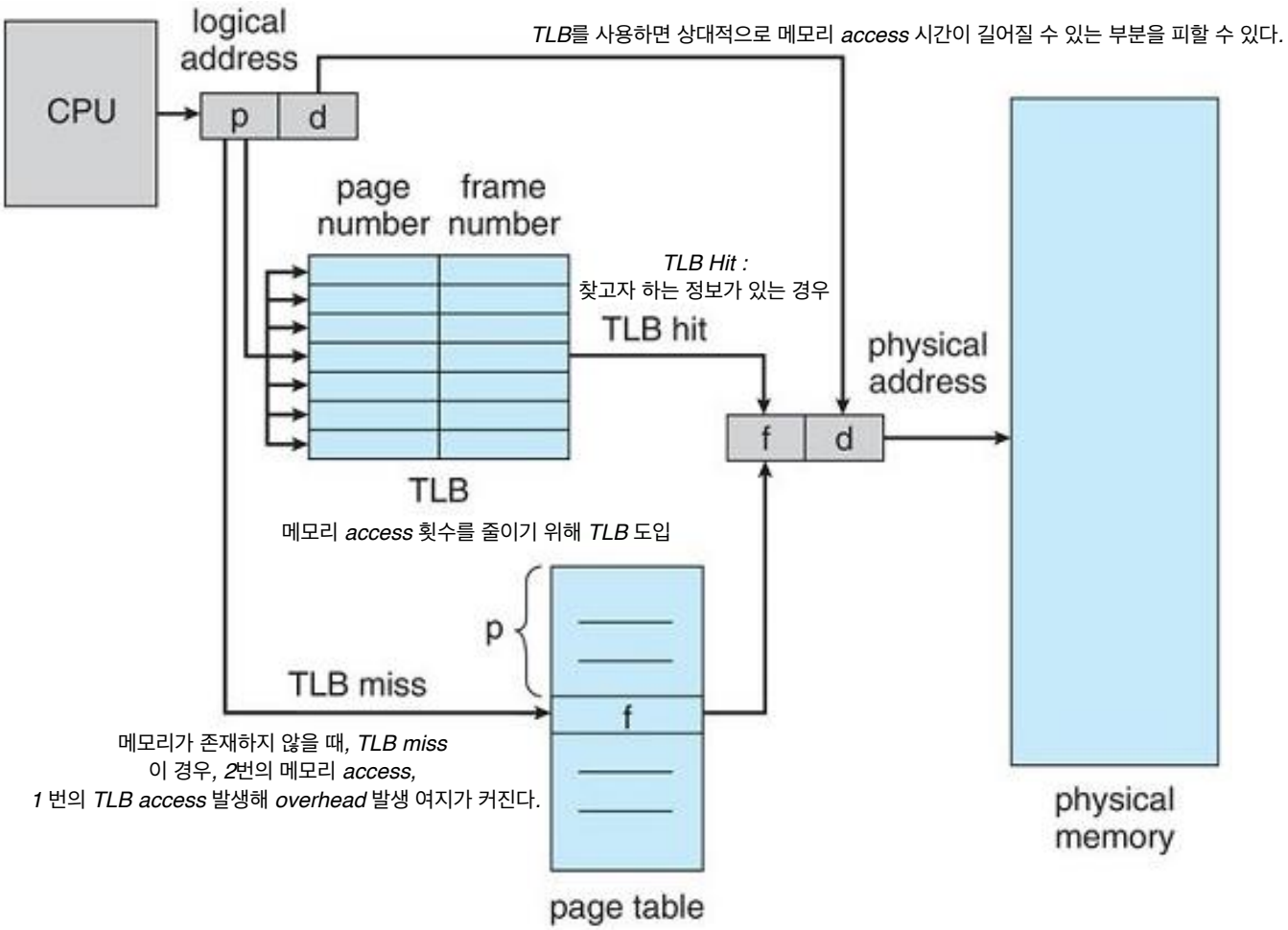



**Figure 9.12** Paging hardware with TLB.

# Paging

- The percentage of times that the page number of interest is found in the TLB is called the **hit ratio**.
  - ✓ If it takes 10ns to access memory, then a mapped-memory access takes 10ns when the page number is in the TLB.
  - ✓ If we fail to find the page number in the TLB, then we must first access memory for the page table and frame number(10ns) and then access the desired byte in memory(10ns), for a total of 20ns. (We are assuming that a page-table lookup takes only one memory access, but it can take more, as we shall see.)
  - ✓ Effective memory-access time (한 번의 *TLB* *(main memory X)*, *Miss*가 발생하면 *TLB + main memory access*)
    - ❖ for 80-percent hit ratio = 0.80 x 10 + 0.20 x 20 = 12ns
    - ❖ for 99-percent hit ratio = 0.99 x 10 + 0.01 x 20 = 10.1ns
  - ✓ CPUs today may provide multiple levels of TLBs. Calculating memory access times in modern CPUs is therefore much more complicated than shown in the example above.
    - ❖ Intel Core i7 CPU has a 128-entry L1 instruction TLB and a 64-entry L1 data TLB.

# Paging

- Protection
  - Memory protection in a paged environment is accomplished by protection bits associated with each frame.
    - ✓ One bit can define a page to be read-write or read-only.
      - ❖ The protection bits can be checked to verify that no writes are being made to a read-only page.
    - ✓ One additional bit is generally attached to each entry in the page table: a valid-invalid bit.
      - ❖ When this bit is set to "valid", the associated page is in the process's logical address space and is thus a legal (or valid) page.
      - ❖ When the bit is set to "invalid", the page is not in the process's logical address space.

# Paging

- Figure 9.13 – Addresses in pages 0,1,2,3,4, and 5 are mapped normally through the page table. But, for 6 and 7, invalid page reference.

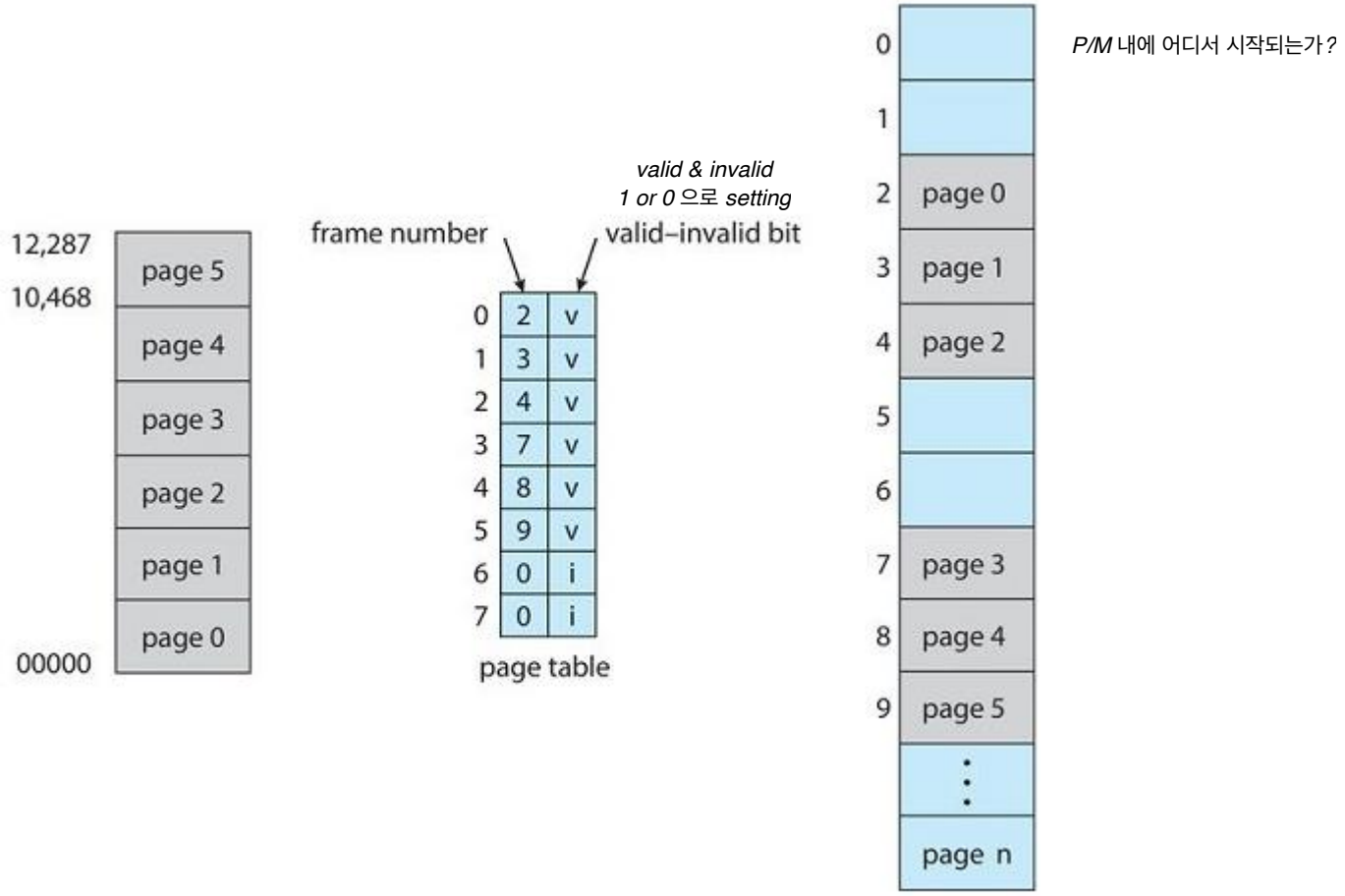



**Figure 9.13** Valid (v) or invalid (i) bit in a page table.

# Paging

- Shared Pages
  - An advantage of paging is the possibility of sharing common code.
  - If the code is reentrant code, it can be shared.
    - ✓ Reentrant code is non-self-modifying code: it never changes during execution.
      쉽게 말해 *read-only* 파일을 의미한다.
      유저가 임의로 쉽게 바꿀 수 없는 것들
    - ✓ Two or more processes can execute the same code at the same time.
      - ❖ Only one copy of the standard C library need be kept in physical memory, and the page table for each user process maps onto the same physical copy of `libc`. .
      - ❖ To support 40 processes, we need only one copy of the library and the total space required is now 2 MB instead of 80 MB – a significant saving!

# Paging

*library*를 프로세스들이 서로 공유하는 상황

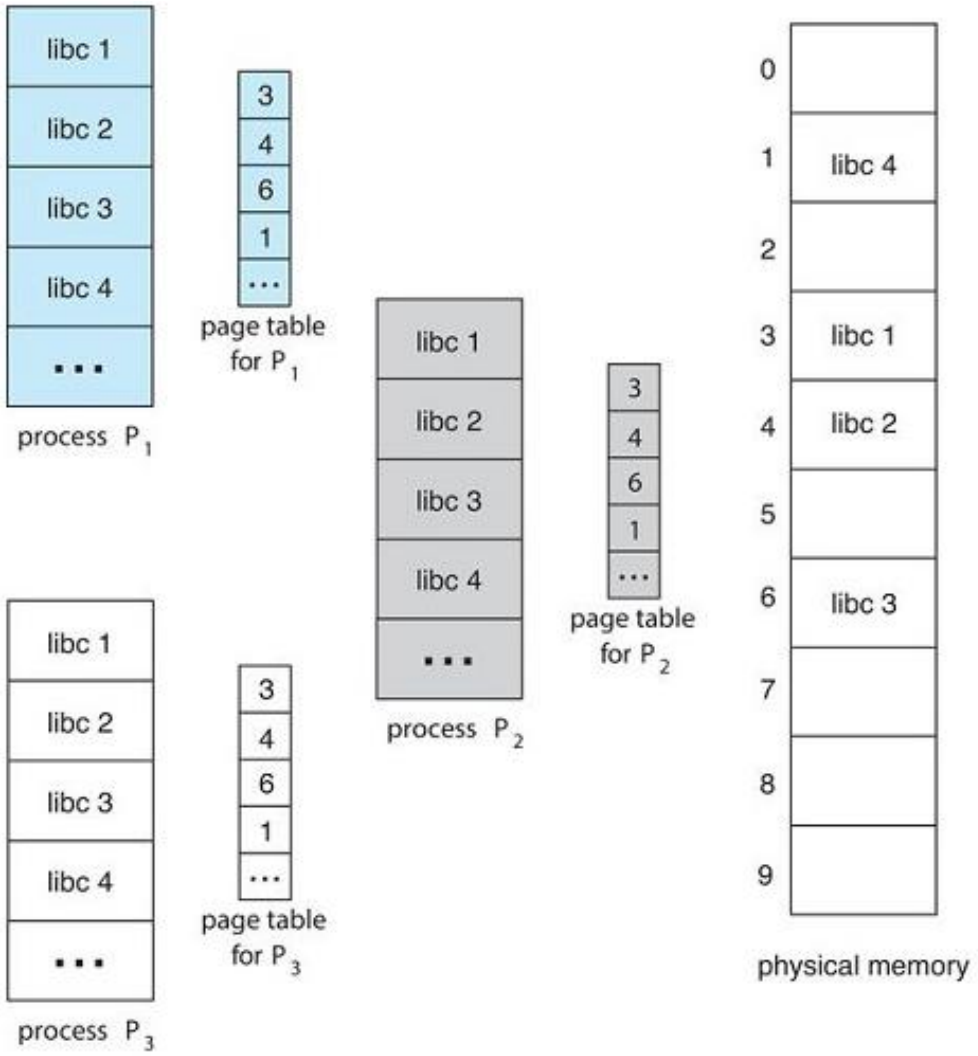



**Figure 9.14** Sharing of standard C library in a paging environment.

# Structure of the Page Table

- Hierarchical Paging
  - The page table itself becomes excessively large. 페이지 테이블이 커진다 --> 감당하기 어려울 것이다.
    - ✓ For example, consider a system with a 32-bit logical address space.
      - ❖ If the page size in such a system is 4 KB ($2^{12}$), then a page table may consist of over 1 million entries ($2^{20} = 2^{32}/2^{12}$).
      - ❖ Assuming that each entry consists of 4 bytes, each process may need up to 4MB of physical address space for the page table alone.
    - ✓ We would not want to allocate the page table contiguously in main memory. One simple solution to this problem is to divide the page table into smaller pieces.

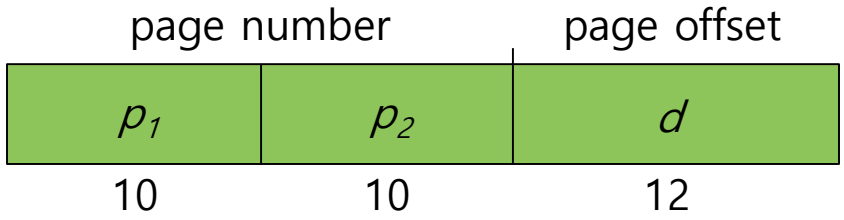
*KB MB GB*
*2^10 2^20 2^30*

# Structure of the Page Table

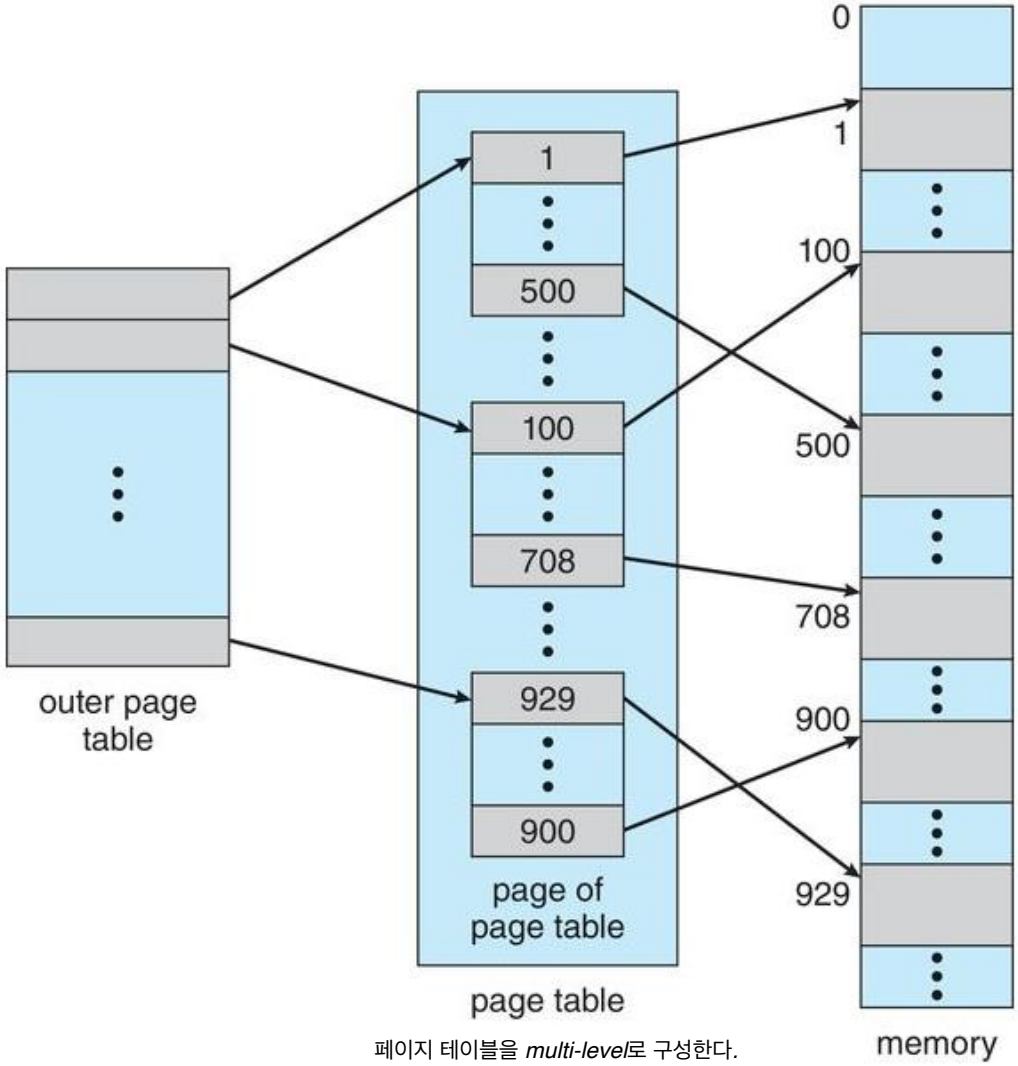
- Hierarchical Paging
  - Two-level paging algorithm
    - ✓ Consider the system with a 32-bit logical address space and a page size of 4KB.
      - ❖ A logical address is divided into a page number consisting of 20 bits and a page offset consisting of 12 bits.
      - ❖ Because we page the page table, the page number is further divided into a 10-bit page number and a 10-bit page offset.
      - ❖ $p_1$ is an index into the outer page table and $p_2$ is the displacement within the page of the inner page table.

*page number* 자체를 또 나누게 된 상황임 => *2^20*이 부담스러운 상황이므로

*p2*는 *index table*이라고 봐도 무방하다



# Structure of the Page Table




페이지 테이블을 *multi-level*로 구성한다.

**Figure 9.15** A two-level page-table scheme.

# Structure of the Page Table

- Figure 9.16
  - ✓ Because address translation works from the outer page table inward, this scheme is also known as a forward-mapped page table.

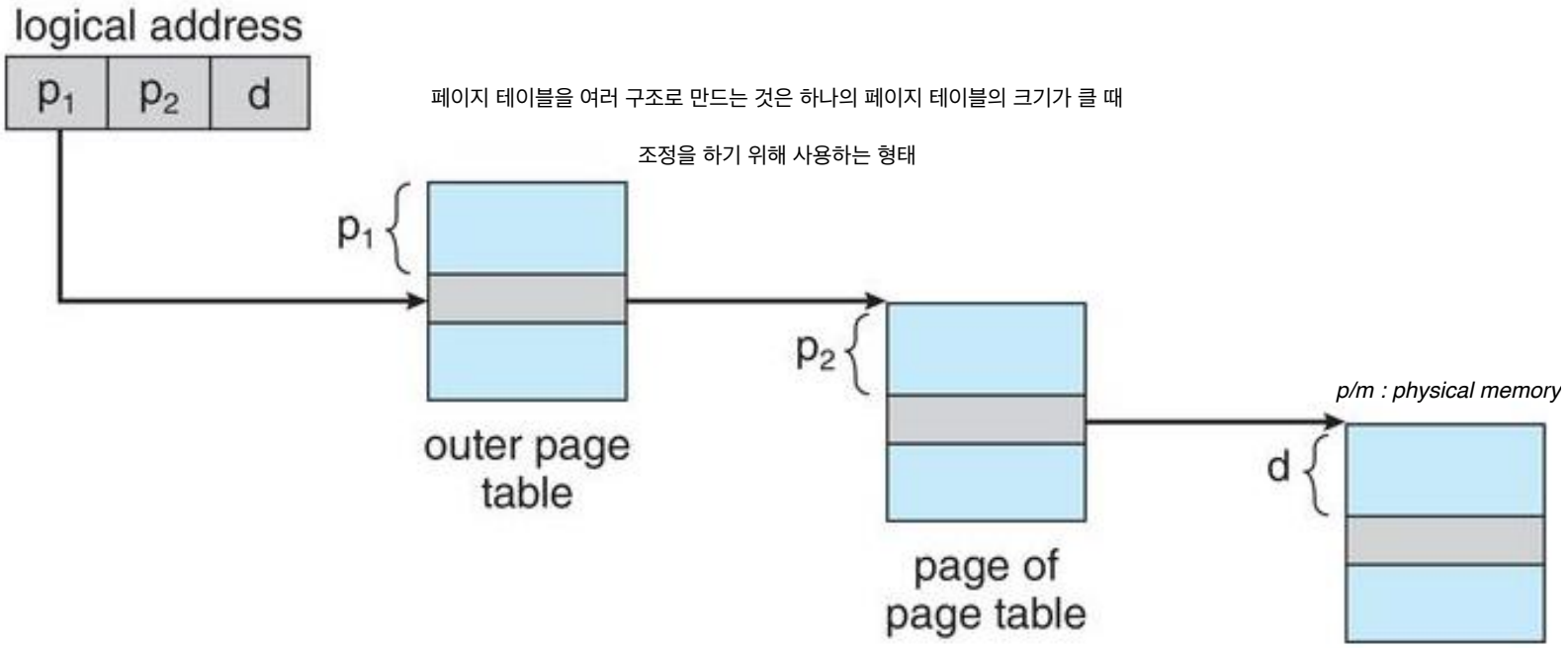
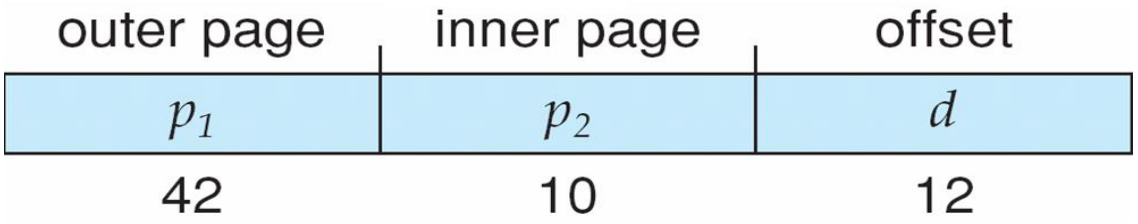



**Figure 9.16** Address translation for a two-level 32-bit paging architecture.

# 64-bit Logical Address Space

- Two-level paging scheme is no longer appropriate
- If page size is 4 KB ($2^{12}$)
  - ✓ The page table consists of up to $2^{52}$ entries
  - ✓ If we use a two-level paging scheme, then the inner page tables can be one page long, or contain $2^{10}$ 4-byte entries.
  - ✓ The addresses look like this:

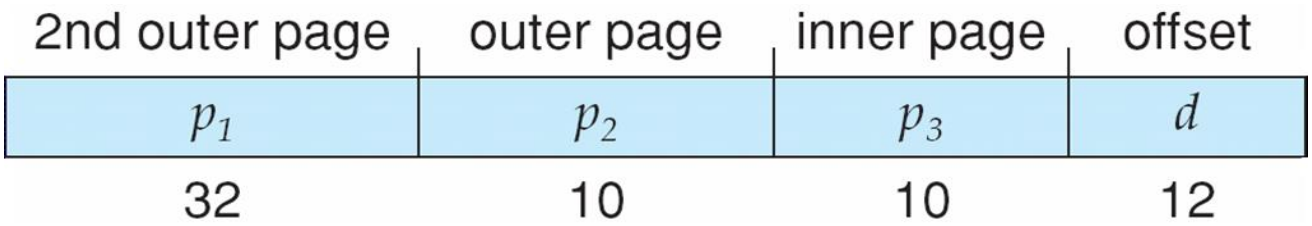



  - ✓ The outer page table has $2^{42}$ entries or $2^{44}$ bytes.

# Three-level Paging Scheme

- We can page the outer page table, giving us a three-level paging scheme
- Suppose that the outer page table is made up of standard-size pages ($2^{10}$ entries, or $2^{12}$ bytes).
- The outer page table is still $2^{34}$ bytes (16 GB) in size.

지금 크기도 너무 크다 ==> *3 level paging scheme*으로 구성한다



# Structure of the Page Table

여기부터는 교수님이 그냥 가볍게 읽어보라고 9장을 끝냄

- Hashed Page Tables
  - One approach for handling address spaces larger than 32 bits is to use a hashed page table.
    - ✓ Each entry in the hash table contains a linked list of elements that hash to the same location.
    - ✓ Each element consists of three fields: (1) the virtual page number, (2) the value of the mapped page frame, and (3) a pointer to the next element in the linked list.
    - ✓ The algorithm works as follows:
      - ❖ The virtual page number in the virtual address is hashed into the hash table.
      - ❖ The virtual page number is compared with field 1 in the first element in the linked list.
      - ❖ If there is a match, the corresponding page frame is used to form the desired physical address.
      - ❖ If there is no match, subsequent entries in the linked list are searched for a matching virtual page number.

# Structure of the Page Table

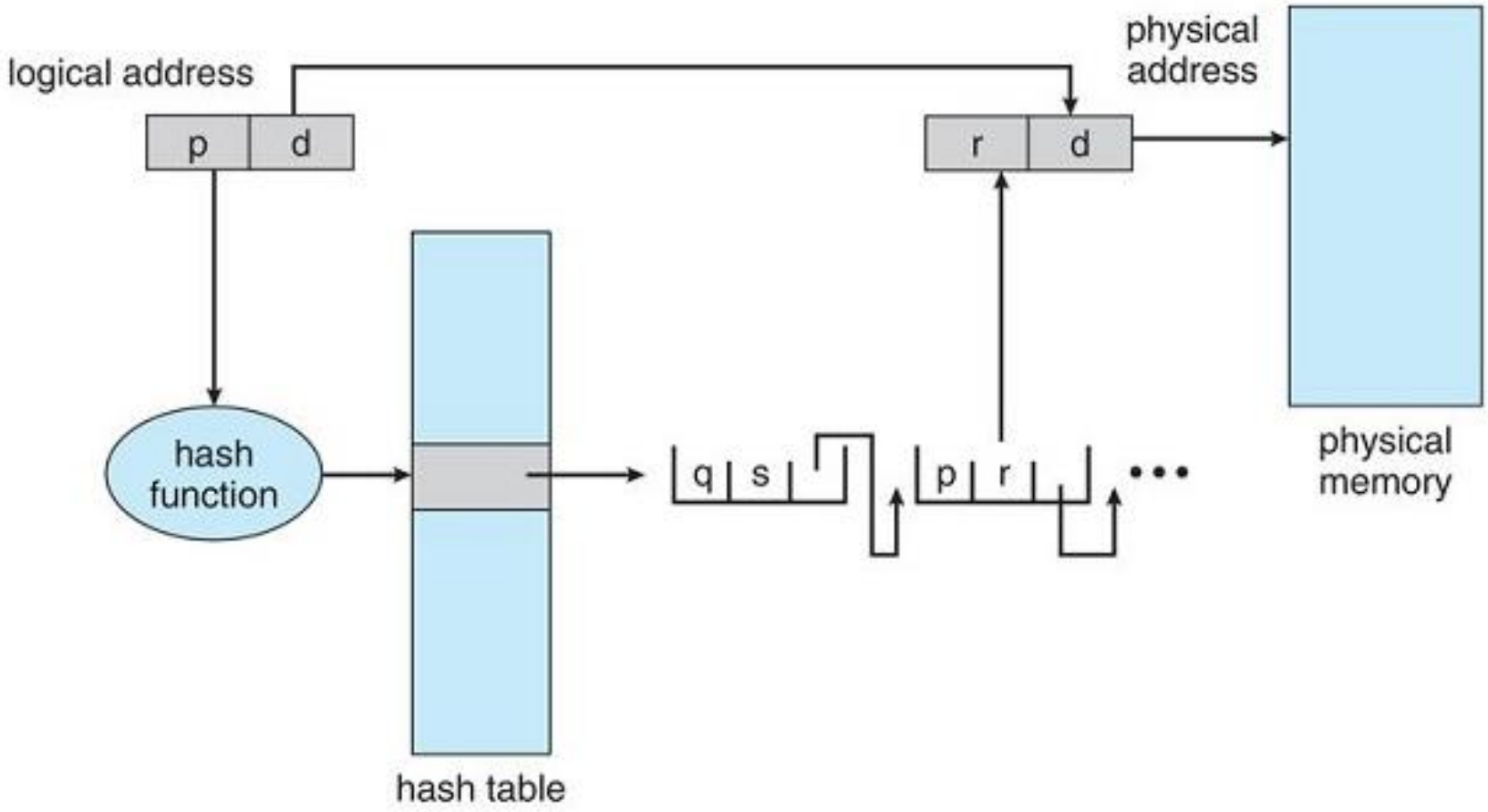



**Figure 9.17** Hashed page table.

# Structure of the Page Table

- Inverted Page Tables 이론적으로 나온 개념이지, 물리적으로 (현실적으로) 사용하기는 어려움
  - Usually, each process has an associated page table. The page table has one entry for each page that the process is using.
  - One of the drawbacks of this method is that each page table may consist of millions of entries.
    - ✓ These tables may consume large amounts of physical memory just to keep track of how other physical memory is being used.
  - To solve this problem, we can use an inverted page table. An inverted page table has one entry for each real page of memory.
    - ✓ Only one page table is in the system, and it has only one entry for each page of physical memory.
    - ✓ Each inverted page-table entry is a pair <process-id, page-number> where the process-id assumes the role of the address-space identifier.
    - ✓ The inverted page table is searched for a match.
      - ❖ If a match is found – say, at entry i – then the physical address <i, offset> is generated.

# Structure of the Page Table

프로세스 구분법
*Pid & Page Number*가 *pair*로 구성되어 특정 페이지 지칭

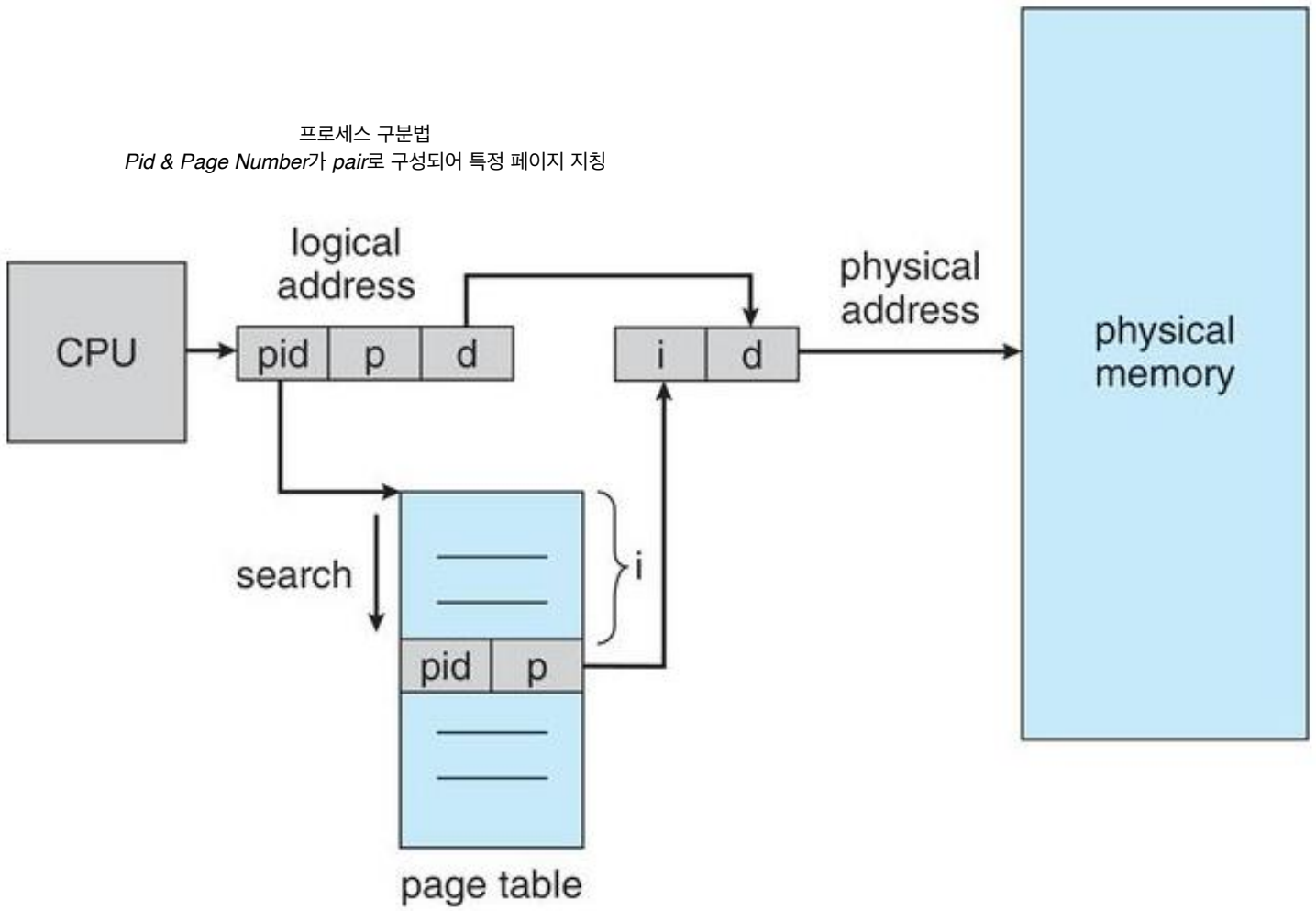



**Figure 9.18** Inverted page table.

# Structure of the Page Table

- Inverted Page Tables
  - Although this scheme decreases the amount of memory needed to store each page table, it increases the amount of time needed to search the table when a page reference occurs.
  - One issue with inverted page tables involves shared memory.
    - ✓ With standard paging, each process has its own page table, which allows multiple virtual addresses to be mapped to the same physical address.
    - ✓ This method cannot be used with inverted page tables.
      - ❖ Because there is only one virtual page entry for every physical page, one physical page cannot have two (or more) shared virtual addresses.

# Swapping

- A process can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution.
- Standard swapping involves moving entire processes between main memory and a backing store.
- The advantage of standard swapping is that it allows physical memory to be oversubscribed, so that the system can accommodate more processes than there is actual physical memory to store them.
    - ✓ Idle or mostly idle processes are good candidates for swapping.
    - ✓ If an inactive process that has been swapped out becomes active once again, it must then be swapped back in.

# Swapping

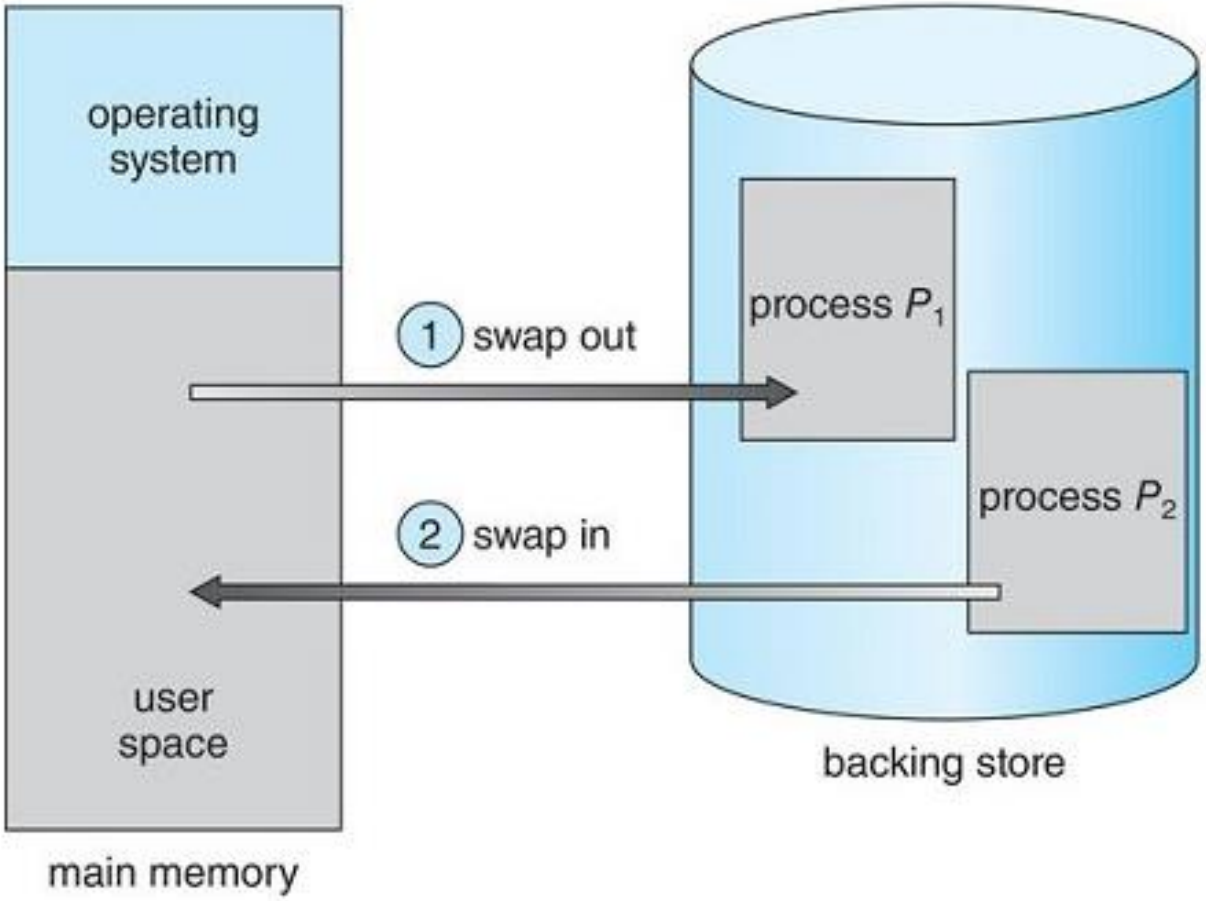



**Figure 9.19** Standard swapping of two processes using a disk as a backing store.

# Swapping

- Most systems, including Linux and Windows, now use a variation of swapping in which pages of a process – rather than an entire process – can be swapped.
- A **page out** operation moves a page from memory to the backing store; the reverse process is known as a **page in**.

# Swapping

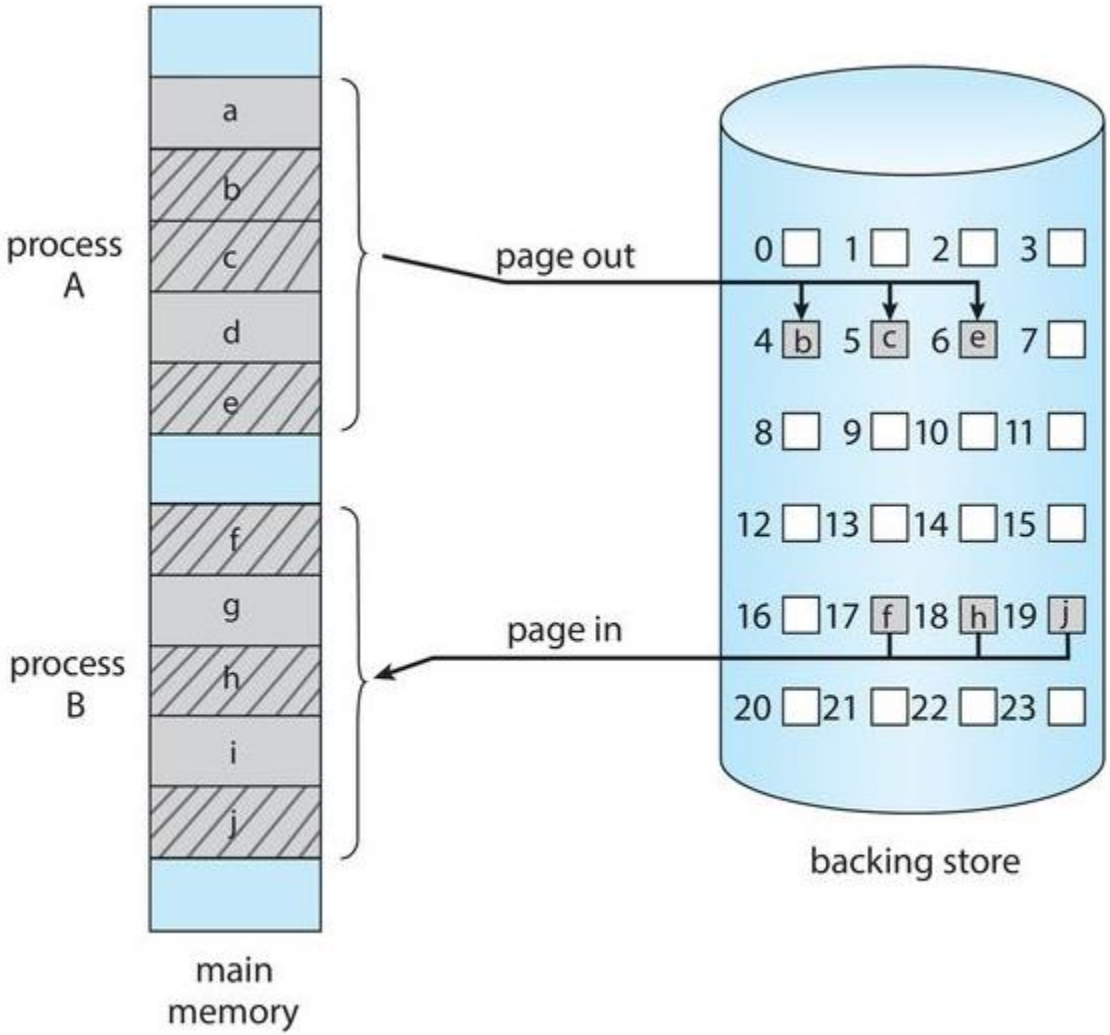



**Figure 9.20** Swapping with paging.

# Swapping on Mobile Systems

모바일 기기는 *swapping*을 사용하지 않는다.

- Mobile systems typically do not support swapping in any form.
  - ✓ Mobile devices generally use flash memory rather than more spacious hard disks for nonvolatile storage.
    - ❖ Limited number of writes that flash memory can tolerate before it becomes unreliable
    - ❖ Poor throughput between main memory and flash memory in these devices
- Instead of using swapping, when free memory falls below a certain threshold,
  - ✓ iOS asks apps to voluntarily relinquish allocated memory.
    - ❖ Read-only data are removed from main memory and later reloaded from flash memory if necessary.
    - ❖ Any apps that fail to free up sufficient memory may be terminated by the OS.
  - ✓ Android may terminate a process if insufficient free memory is available. However, before terminating a process, Android writes its application state to flash memory so that it can be quickly restarted.

# Example : Intel 32 and 64-bit Architectures

- The architecture of Intel chips
  - ✓ The 16-bit Intel 8086 appeared in the late 1970s and was soon followed by another 16-bit chip – the Intel 8088.
  - ✓ Intel produced a series of 32-bit chips – the IA-32 – which included the family of 32-bit Pentium processors.
  - ✓ Intel has produced a series of 64-bit chips based on the x86-64 architecture.

# Example : Intel 32 and 64-bit Architectures

- Memory management in IA-32 systems is divided into two components – segmentation and paging – and works as follows:
  - ✓ The CPU generates logical addresses, which are given to the segmentation unit.
  - ✓ The segmentation unit produces a linear address for each logical address.
  - ✓ The linear address is then given to the paging unit, which in turn generates the physical address in main memory.

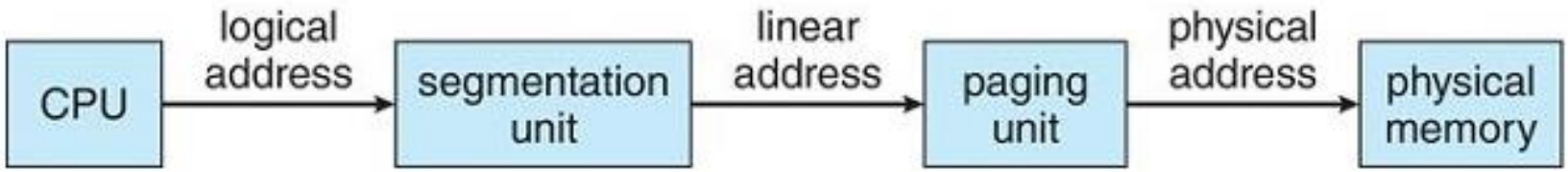
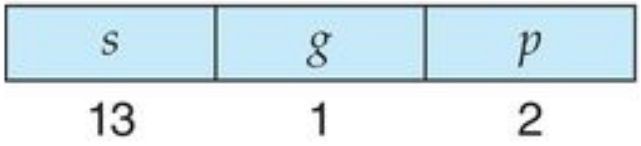



**Figure 9.21** Logical to physical address translation in IA-32.

# Example : Intel 32 and 64-bit Architectures

- IA-32 Segmentation
  - The IA-32 architecture allows a segment to be as large as 4 GB, and the maximum number of segment for process is 16 K.
    - ✓ The logical-address space of a process is divided into two partitions.
      - ❖ The first partition consists of up to 8 K segments that are private to that process. – local descriptor table(LDT)
      - ❖ The second partition consists of up to 8 K segments that are shared among all the processes. – global descriptor table(GDT)
    - ✓ The logical address is a pair (selector, offset), where the selector is a 16-bit number.
      - ❖ s designates the segment number, g indicates whether the segment is in the GDT or LDT, and p deals with protection.



# Example : Intel 32 and 64-bit Architectures

- The linear address on the IA-32 is 32 bits long.
  - ✓ The segment register points to the appropriate entry in the LDT or GDT.
  - ✓ The base and limit information about the segment is used to generate a linear address.
  - ✓ If the address is valid, then the value of the offset is added to the value of the base, resulting in a 32-bit linear address.

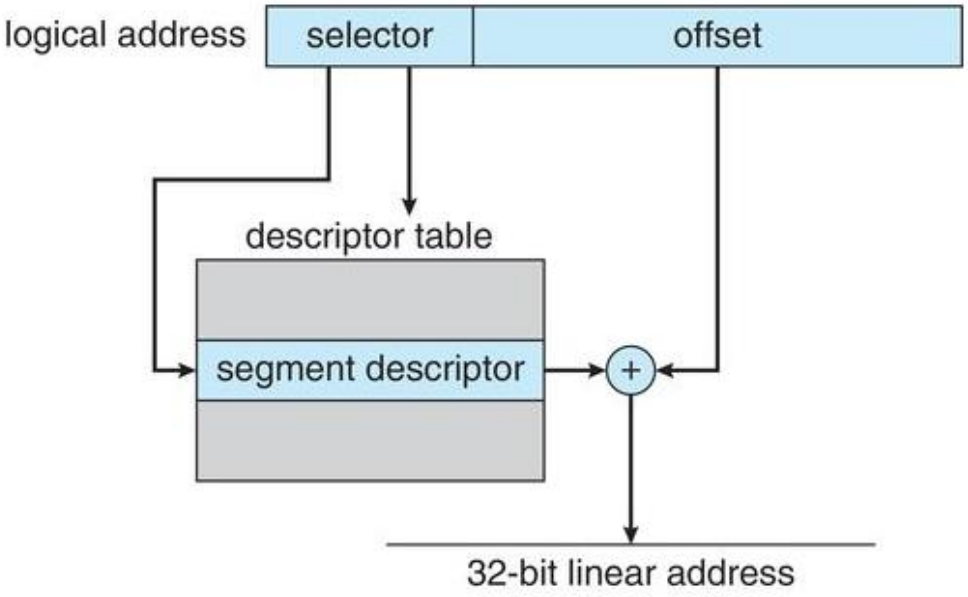
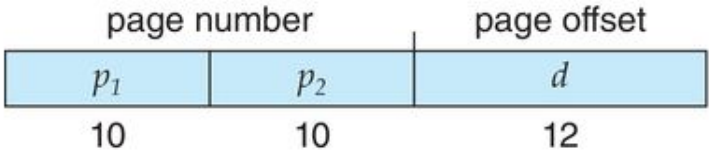



**Figure 9.22** IA-32 segmentation.

# Example : Intel 32 and 64-bit Architectures

– IA-32 Paging 인텔 코어 자체에서 *2 level* 형태로 나눠 작업한다고 생각하면 됨 *(32bits)*

- The IA-32 architecture allows a page size of either 4 KB or 4 MB. For 4 KB pages, IA-32 uses a two-level paging scheme in which the division of the 32-bit linear address is as follows.




✓ The 10 high-order bits reference an entry in the outermost page table, which IA-32 terms the page directory.

✓ The page directory entry points to an inner page table that is indexed by the contents of the innermost 10 bits in the linear address.

✓ The lower-order bits 0-11 refer to the offset in the 4-KB page pointed to in the page table.

✓ One entry in the page directory is the Page_Size flag.

❖ If this flag is set, the page directory points directly to the 4-MB page frame, bypassing the inner page table; and the 22 low-order bits in the linear address refer to the offset in the 4-MB page frame.

# Example : Intel 32 and 64-bit Architectures

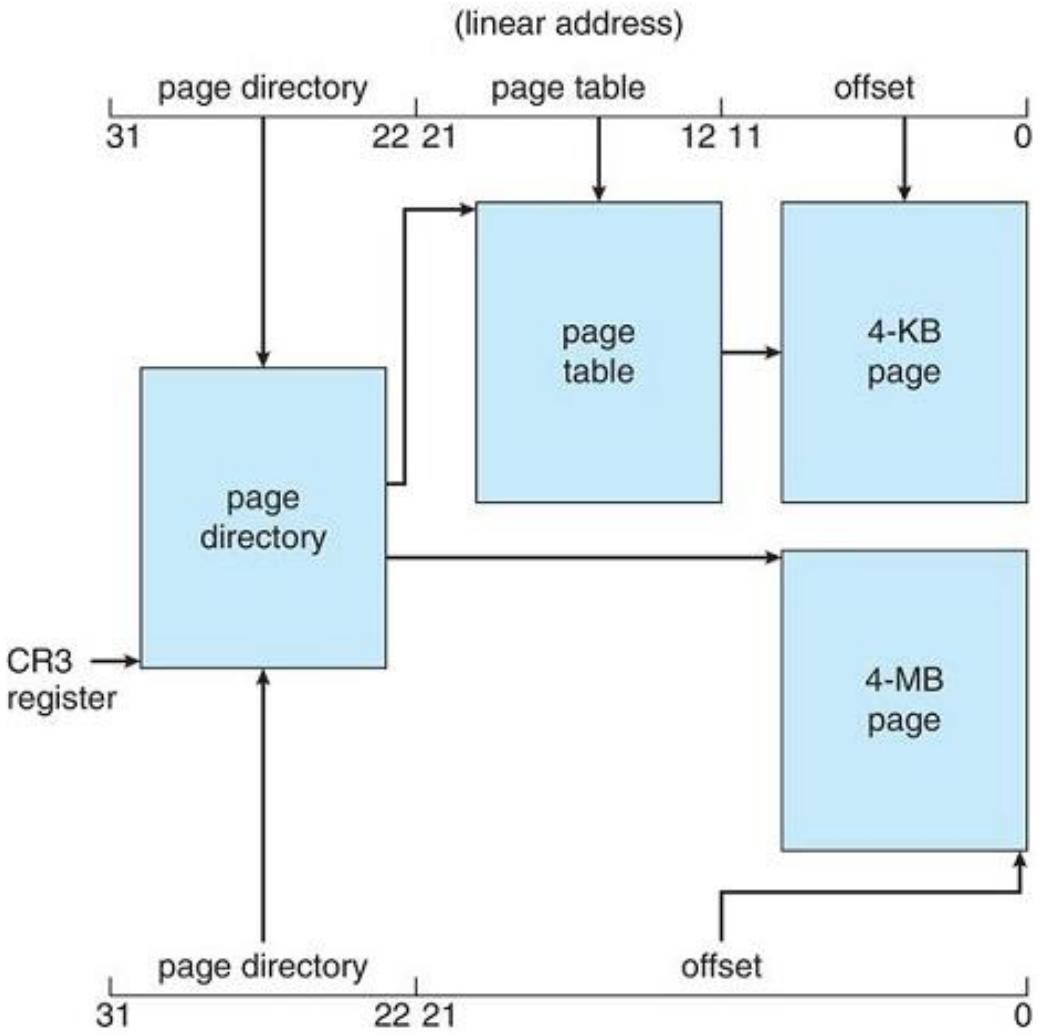



**Figure 9.23** Paging in the IA-32 architecture.

# Example : Intel 32 and 64-bit Architectures

- As software developers began to discover the 4-GB memory limitations of 32-bit architectures, Intel adopted a **page address extension (PAE)**, which allows 32-bit processors to access a physical address space larger than 4GB.
  - ✓ Paging went from a two-level scheme to a three-level scheme.

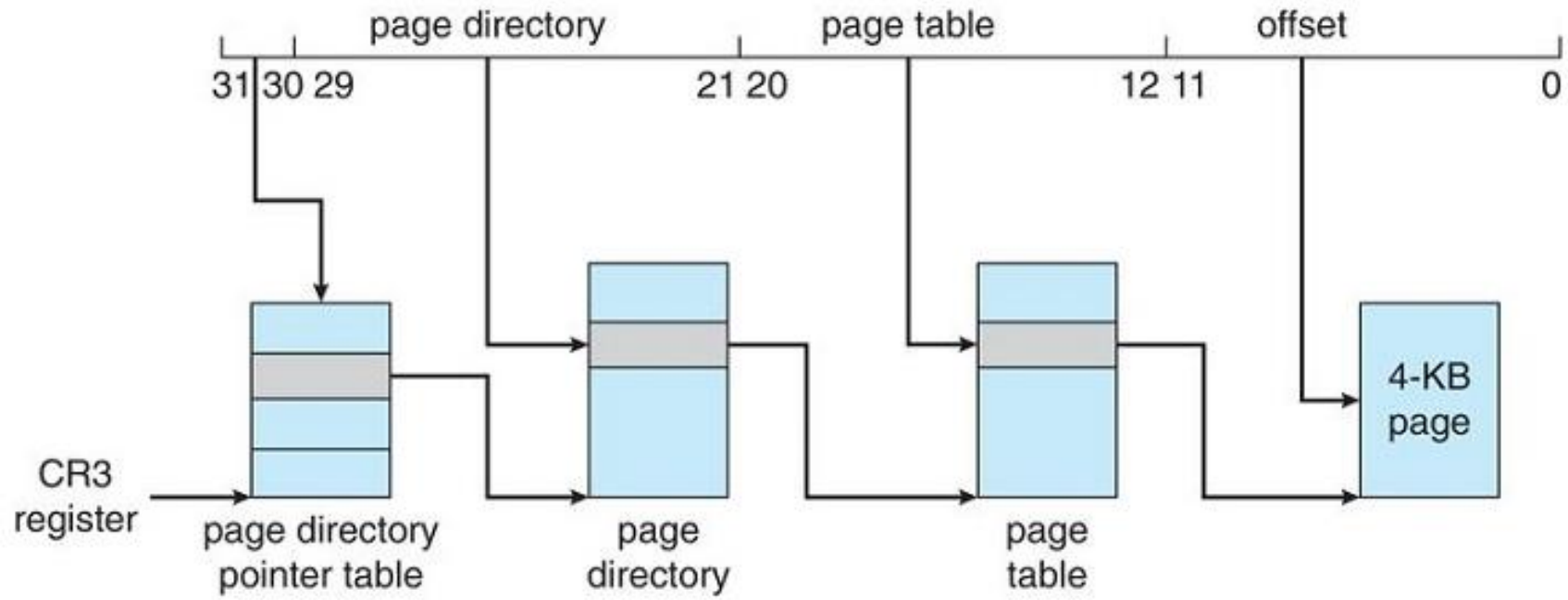



**Figure 9.24** Page address extensions.

# Example : Intel 32 and 64-bit Architectures x86-64

- Intel: IA-64 architecture (later named Itanium)
- AMD: a 64-bit architecture known as x86-64
- Rather than using the commercial names AMD64 and Intel 64, we will use the more general term **x86-64**.
- Provides a 48-bit virtual address
  - ✓ Page sizes of 4 KB, 2 MB, 1 GB
  - ✓ Four levels of paging hierarchy
- Because this addressing scheme can use PAE, virtual addresses are 48 bits in size but support 52-bit physical addresses (4,096 TB).

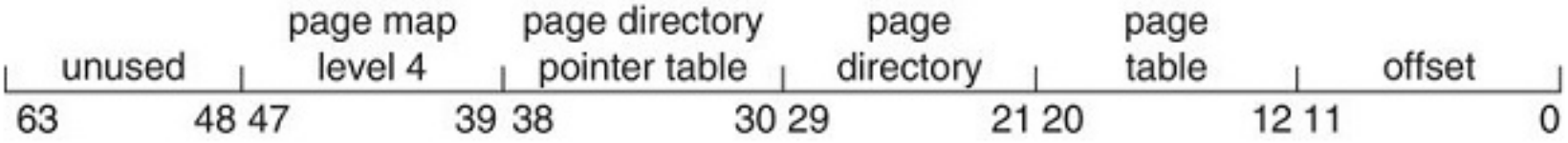



**Figure 9.25** x86-64 linear address.

# Example : ARM Architecture

- Chips for mobile devices such as smartphones and tablet computers often instead run on ARM processors.
  - ✓ iPhone and iPad mobile devices, and most Android-based smartphones
- ARMv8 has three different translation granules: 4 KB, 16 KB, 64 KB.
  - ✓ Each translation granule provides different page sizes, as well as larger sections of contiguous memory, known as regions.

| Translation Granule Size | Page Size | Region Size |
|---|---|---|
| 4 KB | 4 KB | 2 MB, 1 GB |
| 16 KB | 16 KB | 32 MB |
| 64 KB | 64 KB | 512 MB |

# Example : ARM Architecture

- For 4-KB and 16-KB granules, up to four levels of paging may be used, with up to three levels of paging for 64-KB granules.
  - ✓ Although ARMv8 is a 64-bit architecture, only 48 bits are currently used

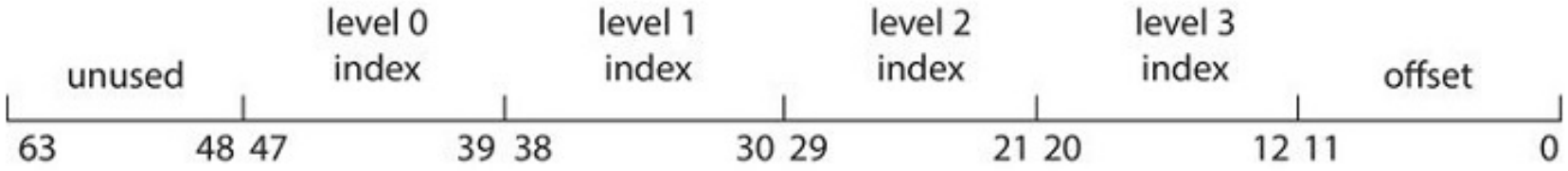



**Figure 9.26** ARM 4-KB translation granule.

- If all four levels are used, the offset (bits 0–11 in Figure 9.26) refers to the offset within a 4-KB page.
  - ✓ The table entries for level 1 and level 2 may refer either to another table or to a 1-GB region (level-1 table) or 2-MB region (level-2 table).
  - ✓ **[Example]** If the level-1 table refers to a 1-GB region rather than a level-2 table, the low-order 30 bits (bits 0–29 in Figure 9.26) are used as an offset into this 1-GB region

# Example : ARM Architecture

- TTBR register (translation table base register) : points to the level 0 table for the current thread

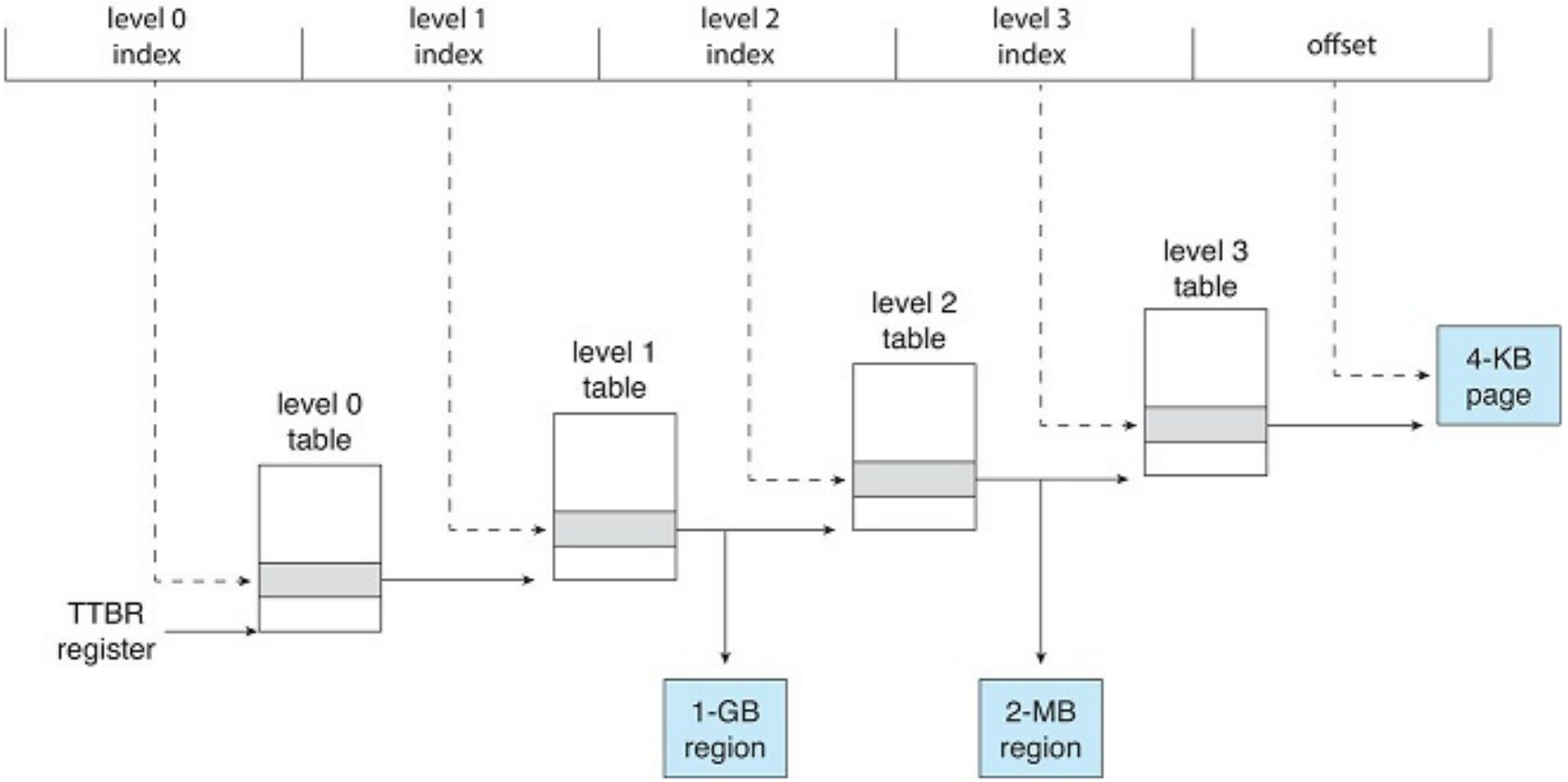



**Figure 9.27** ARM four-level hierarchical paging.