Chapter 8: Main Memory

2020.6

Howon Kim

- 정보보호 및 지능형 IoT연구실 – http://infosec.pusan.ac.kr
- 부산대 지능형융합보안대학원 – http://aisec.pusan.ac.kr

Chapter 8: Memory Management

- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table

Objectives

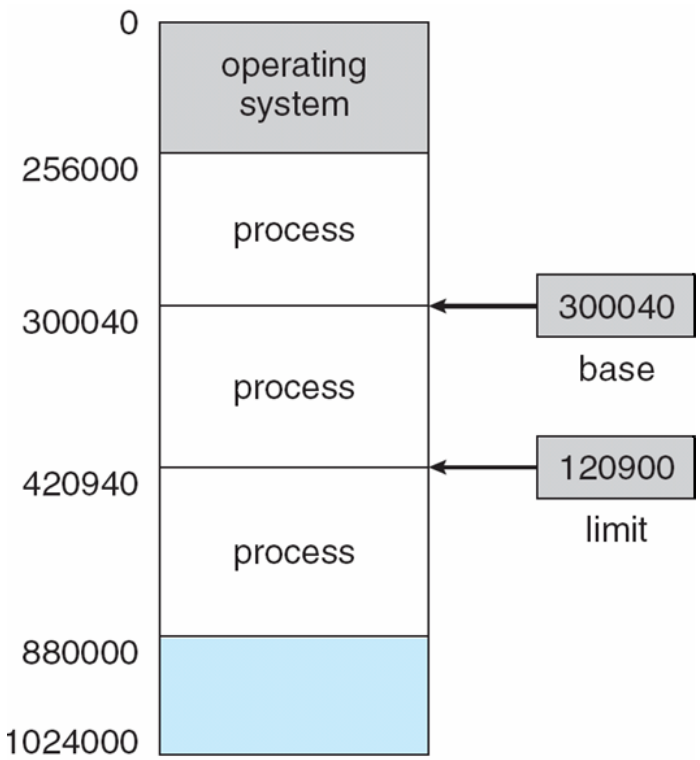
- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation

Background

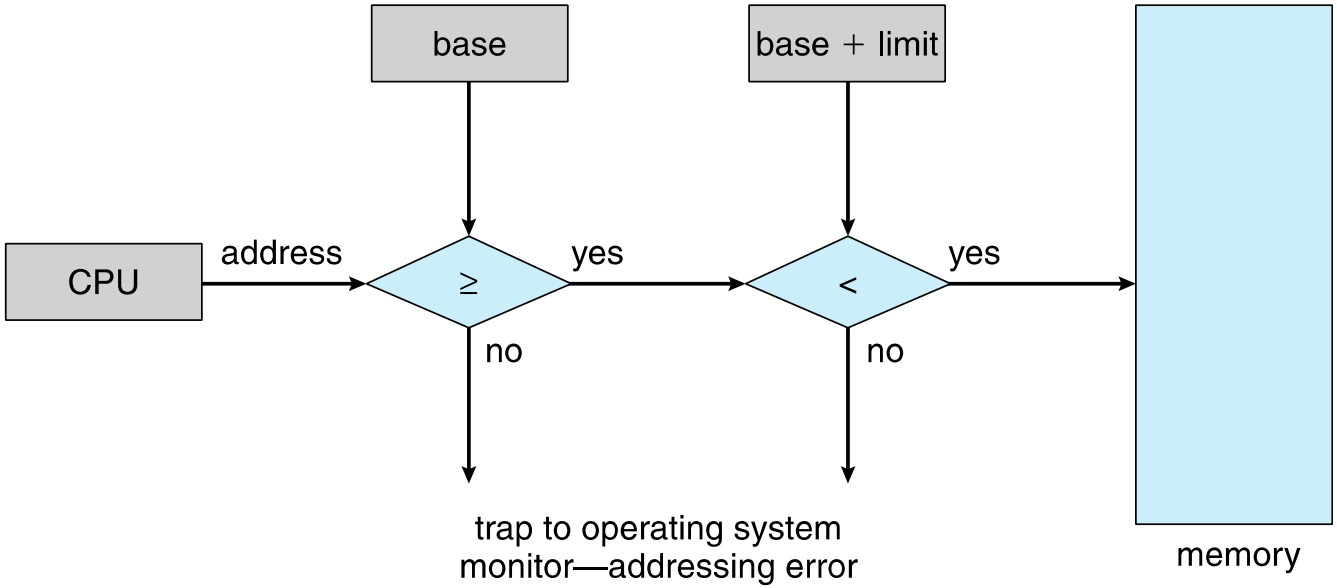
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage that CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- **Register access in one CPU clock (or less)**
- **Main memory can take many cycles, causing a stall**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

Base and Limit Registers

- A pair of **base** and **limit registers** define the logical address space
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



Hardware Address Protection



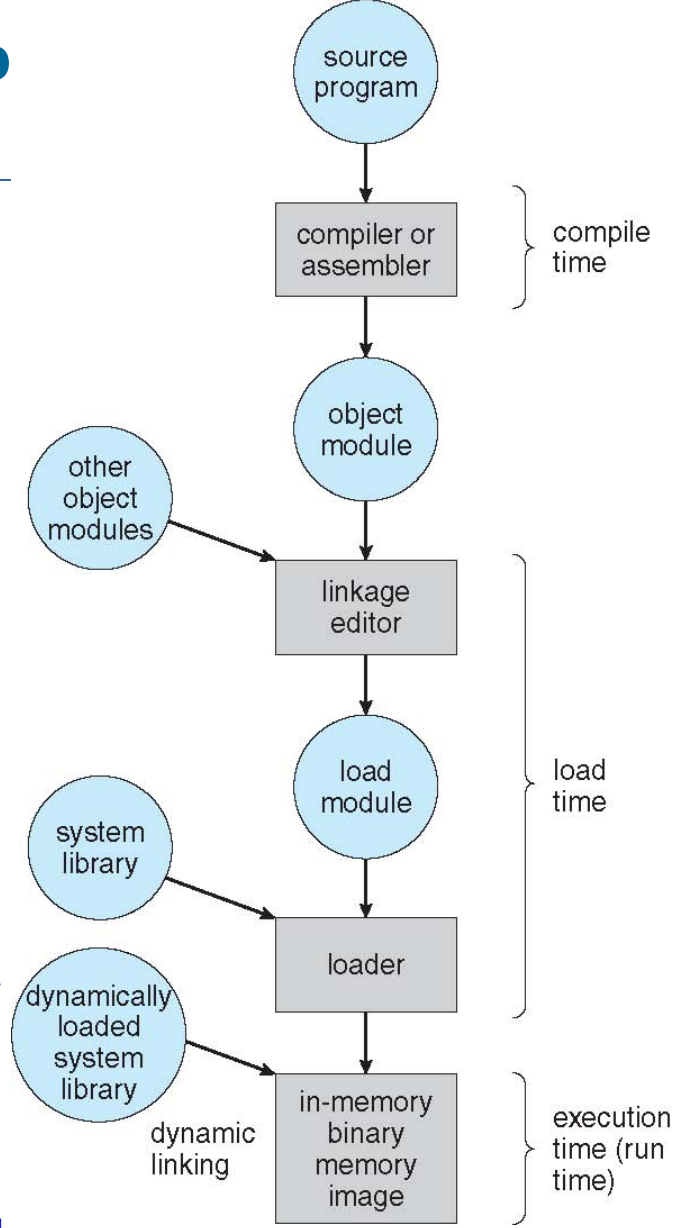
Address Binding

- Programs on disk, ready to be brought into memory to execute form an **input queue**. (하드디스크상의 program이 memory에 loading 되기 전에 input queue에서 대기함. Scheduling 대기)
- Inconvenient to have first user process physical address always at 0000
  - 오히려 모든 user process가 physical 시작 주소를 0000 으로 하면 곤란함
  - 여러 user process가 동일한 메모리 주소를 가지는 셈

- Further, addresses represented in different ways at different stages of a program's life
  - Source code addresses usually symbolic. 소스코드에서 주소는 당연히 숫자가 아닌 symbol 형태로 표시됨(예: 변수 counter)
  - Compiled code addresses **bind** to relocatable addresses. 컴파일러에서는 이 symbol 형태의 주소를 재배치 가능 주소(예: 이 모듈의 첫번째 바이트로부터 열네번째 바이트 주소 등)로 바인딩 시킴
    - i.e. "14 bytes from beginning of this module"
  - Linker or loader will bind relocatable addresses to absolute addresses
    - 링커와 로더는 이 재배치 가능 주소를 절대 주소로 바인딩함
    - i.e. 74014

Binding of Instructions and Data to Memory

- **Address binding** of instructions & data to memory addresses can happen at different stages
  - **Compile time address binding**:
    - If memory location known a priori, absolute code can be generated; must recompile code if starting location changes.
    - 컴파일 시점에 해당 프로세스가 들어갈 특정 주소 정보를 아는 경우. 컴파일된 코드는 특정 주소부터 시작되도록 할 수는 있음
    - 예: MS-DOS의 .COM 양식 프로그램
  - **Link time binding**: dynamic vs. static libraries
  - **Load time binding**:
    - Must generate relocatable code if memory location is not known at compile time
    - 컴파일 시점에 프로세스가 메모리의 어느 곳(주소)에 loading 되는지를 모르면, 컴파일러는 재배치 가능코드 만듦
    - 실제 symbol 및 진짜 주소와의 바인딩은 프로그램이 주 메모리에 실제 로딩 될 때 이뤄짐. (재배치 가능코드이므로 기준 주소값을 정하면 됨)
  - **Execution time binding**:
    - Binding delayed until “run time” if the process can be moved during its execution from one memory segment to another
    - 만약 프로세스가 실행되는 중간에 메모리 내의 한 세그먼트에서 다른 세그먼트로 옮겨질 수 있는 경우, 이를 “바인딩이 실행시간에 이뤄진다고 함“
    - 이처럼 실행시점에서 바인딩이 이뤄지려면 특별한 하드웨어가 있어야 함



< Multistep Processing of a User Program >

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** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit

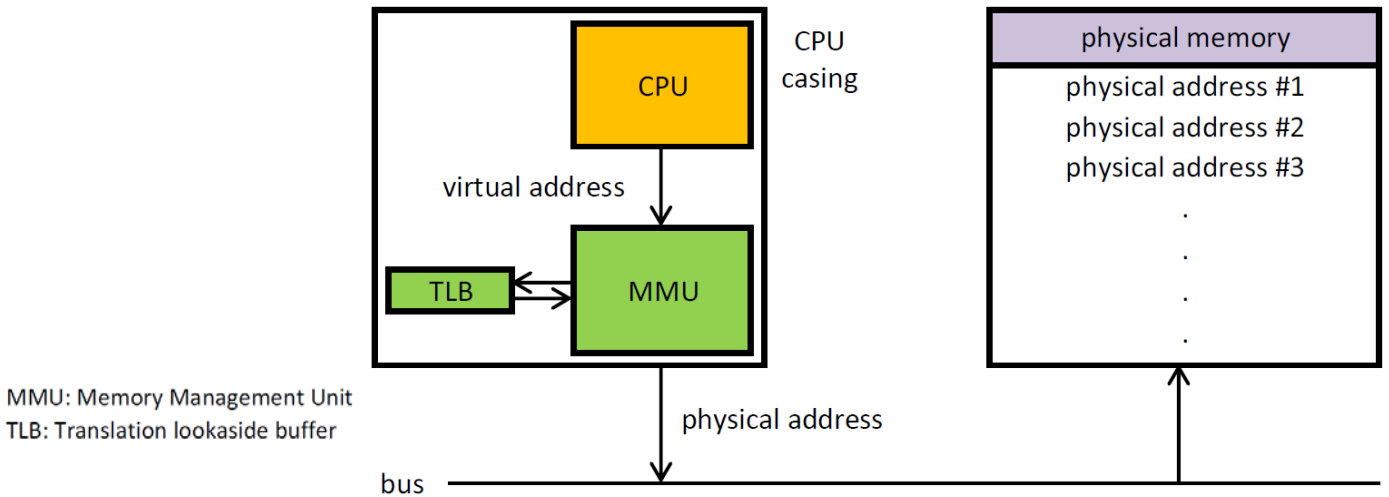
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes;
- logical (virtual) and physical addresses differ in execution-time address-binding scheme

Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address at run time
- To start, consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - Base register now called **relocation register**
  - MS-DOS on Intel 80x86 used 4 relocation registers

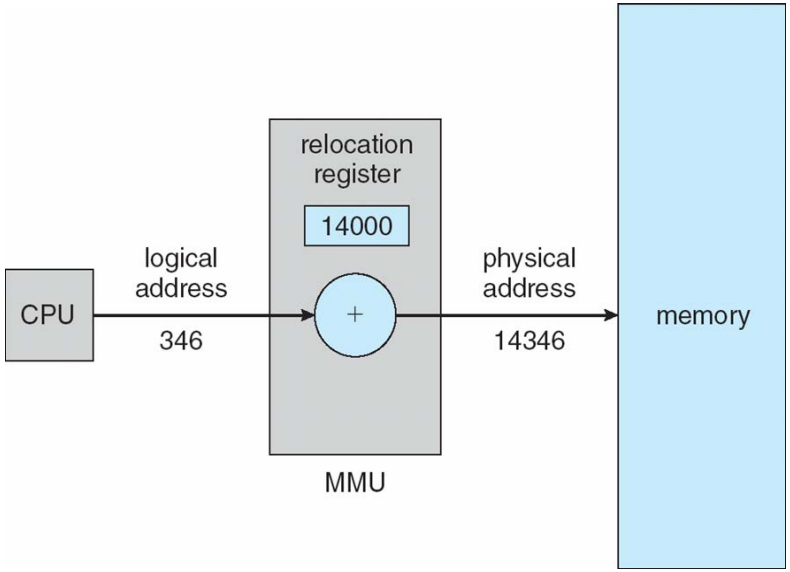
user process 가상주소 + relocation register value(address) → Physical address

- The user program deals with *logical* addresses; it never sees the *real* physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses



Dynamic relocation using a relocation register

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- **All routines kept on disk in relocatable load format**
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required
  - Implemented through program design
  - OS can help by providing libraries to implement dynamic loading



Dynamic Linking

- **Static linking** – system libraries and program code combined by the loader into the binary program image

- **Dynamic linking** –linking postponed until execution time
  - Dynamic linking에서는 실행시 “시스템 library”를 동적으로 loading 함
  - Dynamic linking에서는 “시스템 library”를 linking 시키기 위한 stub 존재
  - Stub replaces itself with the address of the routine, and executes the routine

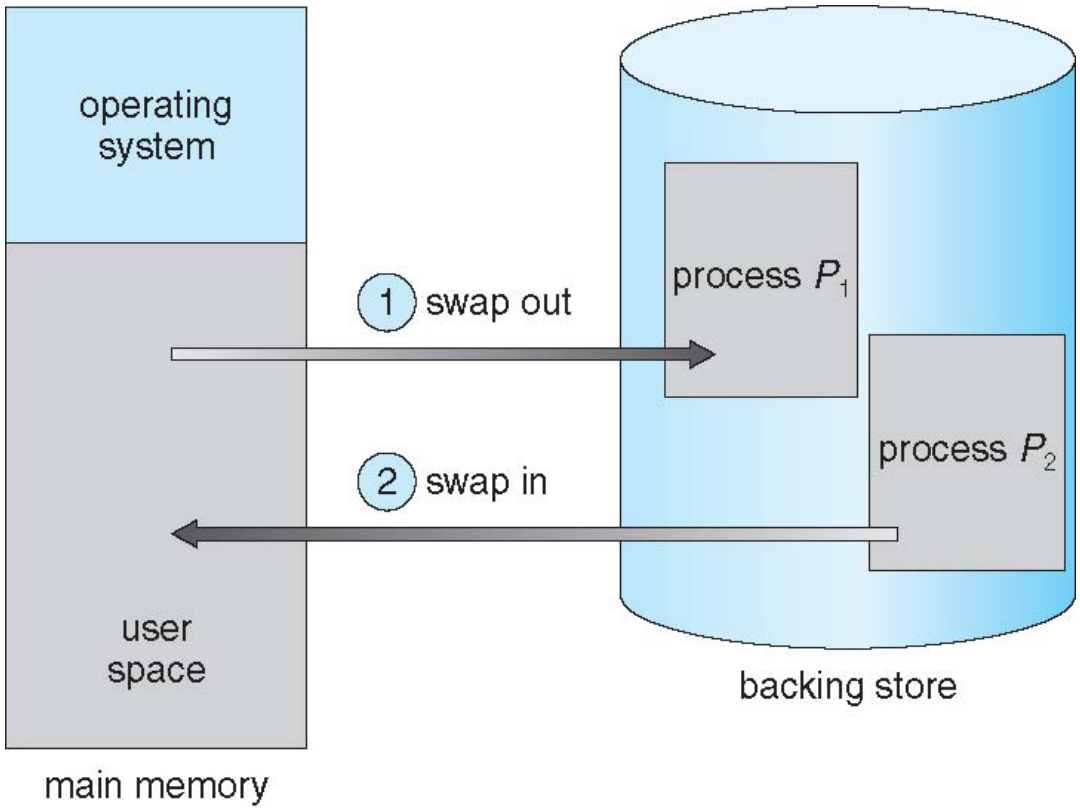
Swapping

- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **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
  - The current program or program segment is stored (rolled out) on disk, and another program is brought into (rolled in) that memory space
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

Swapping (Cont.)

- **Modified versions of swapping are found on many systems** (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold
  - 즉, 최근 여러 OS에서는 수정된 swapping 기법을 사용함. 즉, 기본적으로 swapping을 사용하지 않다가, 메모리 사용량(메모리 할당량)이 어떤 기준 이상이면 swapping을 사용함. → 향후, 다시 메모리 사용량이 어떤 기준 이하가 되면 swapping 을 사용하지 않음

Schematic View of Swapping



Context Switch Time including Swapping

- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process. CPU에서 필요로 하는 next process가 메모리에 없으면 storage로부터 swap in 됨
- Context switch time can then be very high
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
  - Swap out time of 2sec !
  - Plus swap in of same sized process
  - Total context switch swapping component time of 4000ms (4 seconds)
- Context switch time can be reduced if the size of memory swapped is reduced – by knowing how much memory really being used
  - System calls to inform OS of memory use via `request_memory()` and `release_memory()`
  - 즉, 사용자는 메모리 요구 사항의 변화가 있을 때, 이와 같은 시스템 호출을 사용하여, OS에게 메모리 요구 사항의 변화를 알려줌

Context Switch Time and Swapping (Cont.)

- Swapping에는 **여러 제약 사항이 존재함**
  - 어떤 프로세스를 swap out 하기 위해선 해당 프로세스는 완전히 휴지 상태에 있어야 함
  - **만약 Pending I/O 상태라면?** – can't swap out as I/O would occur to wrong process
  - Or always transfer I/O to kernel space(buffer), and then to I/O device
    - Known as **double buffering**, adds overhead
    - 즉, I/O는 OS의 kernel space와 이뤄지고 나중에 kernel space에서 I/O혹은 해당 process와 통신이 이뤄짐 → 오버헤드 유발
- Standard swapping not used in modern operating systems
  - But modified version common
    - Swap only when free memory extremely low

Swapping on Mobile Systems

- Swapping is not typically supported
  - Flash memory based
    - Small amount of space
    - Limited number of write cycles (flash memory 쓰기 허용 횟수 제한)
    - Poor throughput between flash memory and CPU on mobile platform
- Instead use other methods if free memory is low
  - iOS ***asks*** apps to voluntarily relinquish allocated memory
    - Read-only data thrown out and reloaded from flash if needed
    - Failure to free can result in termination
  - Android terminates apps if low free memory, but first writes **application state** to flash for fast restart
  - Both OSes support paging as discussed below

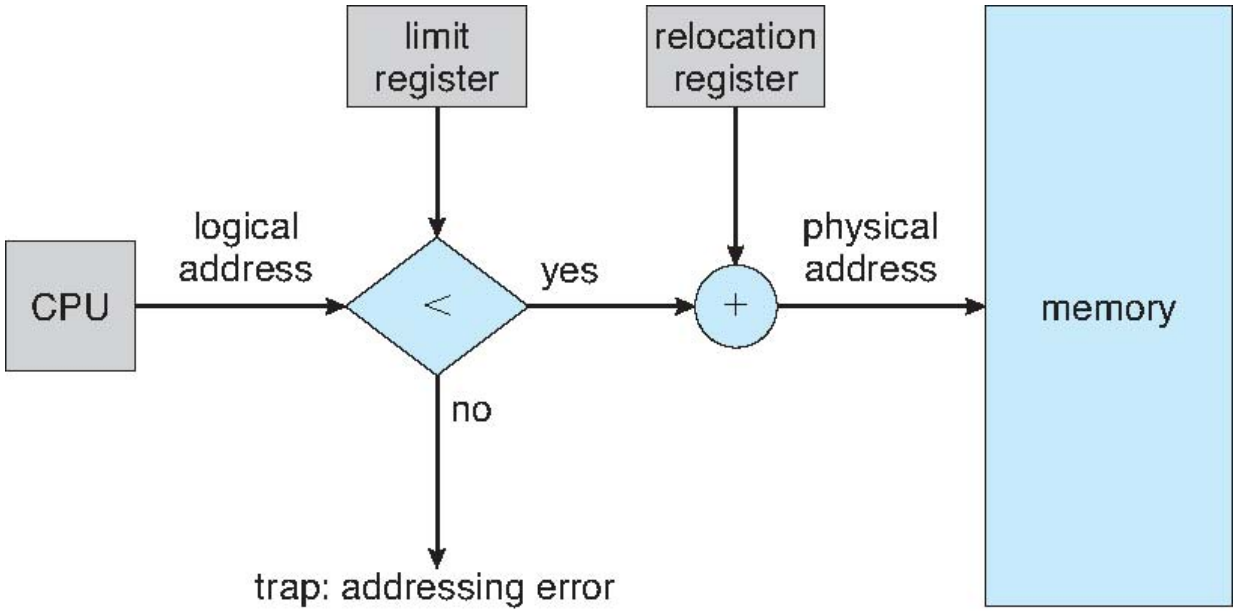
Contiguous Allocation

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
  - If sufficient contiguous memory is found, the process is allocated memory to start its execution.
  - Otherwise, it is added to a queue of waiting processes until sufficient free contiguous memory is available
    - **Contiguous allocation(연속 메모리 할당) :** 각 프로세스는 다음 프로세스를 포함하는 메모리 영역과 연속된 하나의 메모리 영역을 차지함
  - The contiguous memory allocation scheme can be implemented in operating systems with the help of two registers, known as the base and limit registers.
- Main memory usually into two **partitions**:
  - Resident operating system, usually held in low memory address space near interrupt vector(주로 0번지에 위치함)
  - User processes then held in high memory address space
  - Each process contained in single contiguous section of memory

Contiguous Allocation (Cont.)

- Relocation registers (재배치 레지스터) used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*
  - Can then allow actions such as kernel code being **transient** and kernel changing size
    - Relocation register를 사용하면 OS(kernel)는 실행 중이라도 그 크기 변경될 수 있음
    - OS가 지원하던 어떤 주변장치를 향후 더 이상 사용할 필요없다면 OS는 해당 주변장치 관련 코드를 메모리에 유지할 필요 없음 (이를 transient OS code라고 함)

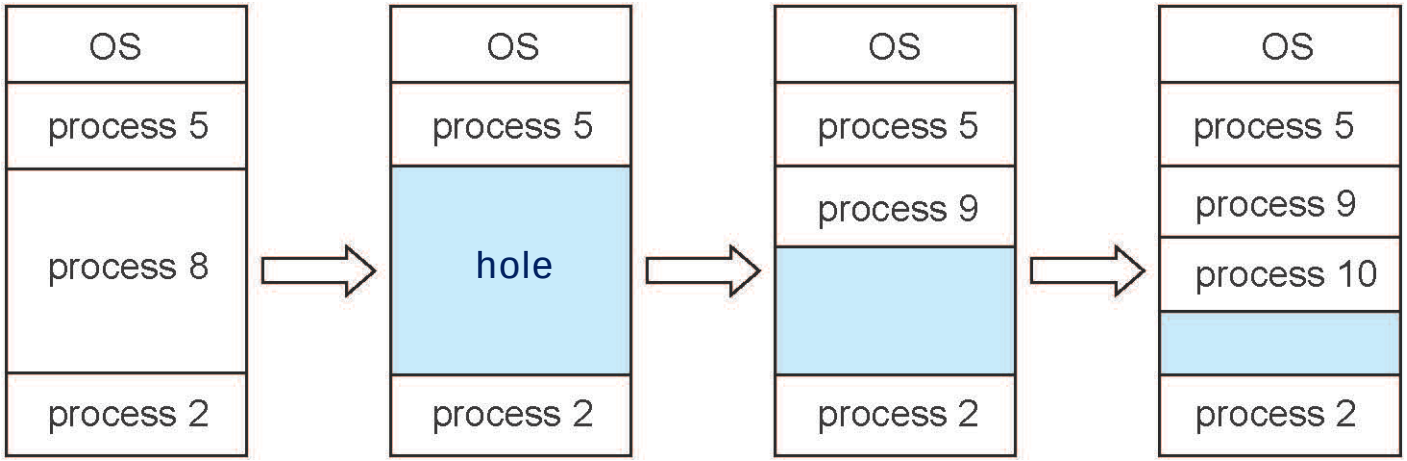
Hardware Support for Relocation and Limit Registers



어떤 프로세스는 자신이
소유하지 않은 메모리 공간
접근할 수 없게 됨

Multiple-partition allocation

- Multiple-partition allocation
  - Degree of multiprogramming limited by number of partitions
  - **Variable-partition sizes** for efficiency (sized to a given process' needs)
  - **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
  - Process exiting frees its partition, adjacent free partitions combined
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (hole)



Dynamic Storage-Allocation Problem

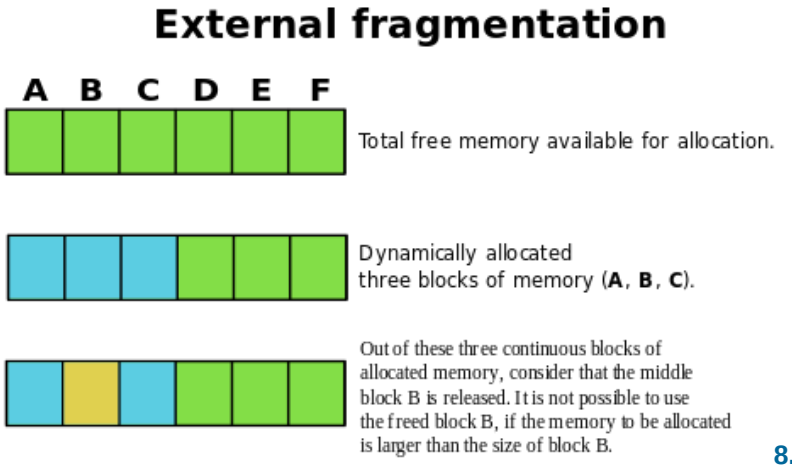
How to satisfy a request of size ***n*** from a list of free holes?

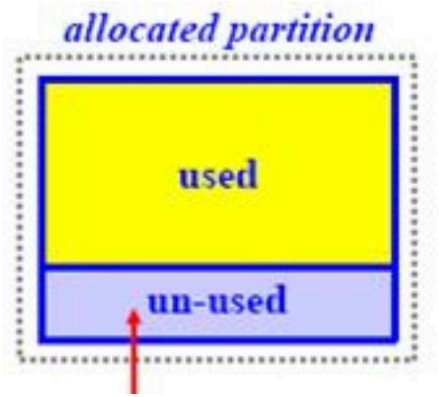
- **First-fit**: Allocate the ***first*** hole that is big enough
  - 즉, 첫번째 사용 가능한 가용 공간을 할당함. 검색 대상 시작부터 찾거나 혹은 지난번 검색이 끝났던 곳에서 시작될 수 있음
- **Best-fit**: Allocate the ***smallest*** hole that is big enough; must search entire list, unless ordered by size
  - 사용 가능한 공간 중에서 (프로세스 크기를 만족시키면서) 가장 작은 것 공간을 선택함
- **Worst-fit**: Allocate the ***largest*** hole; must also search entire list
  - 가장 큰 가용 공간을 선택함

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

Fragmentation

- **Fragment :** 메모리 공간 중, 일부 사용하지 못하는 메모리 공간
- **External Fragmentation**
  - 프로세스가 메모리에 적재/ 제거되는 일이 반복되면서 작은 메모리 조각들이 발생함
  - 이들 조각화 된 공간을 모두 합치면 필요한 메모리 공간에 충분한 공간이 되지만, 실제는 분산화된 작은 조각들이므로 이를 사용하지 못하게 됨
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
  - 필요한 메모리 공간보다 더 많이 메모리가 할당됨
  - 이때 un-used 되는 메모리 공간이 생기는 것을 "내부 단편화"라고 함
- **First fit** analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation
  - 1/3 may be unusable -> **50-percent rule**





Internal fragmentation

Fragmentation (Cont.)

- Reduce external fragmentation by **compaction (압축)**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers

Segmentation

- Memory-management scheme that supports user view of memory
- 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

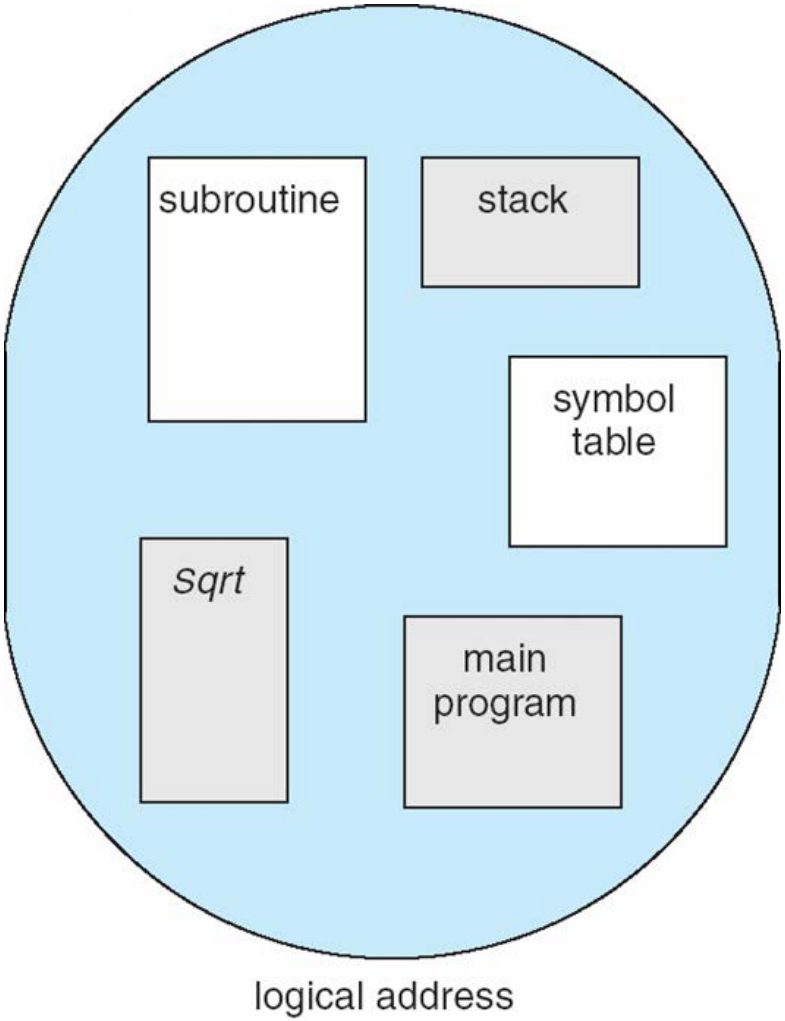
    common block

    stack

    symbol table

    arrays

User's View of a Program

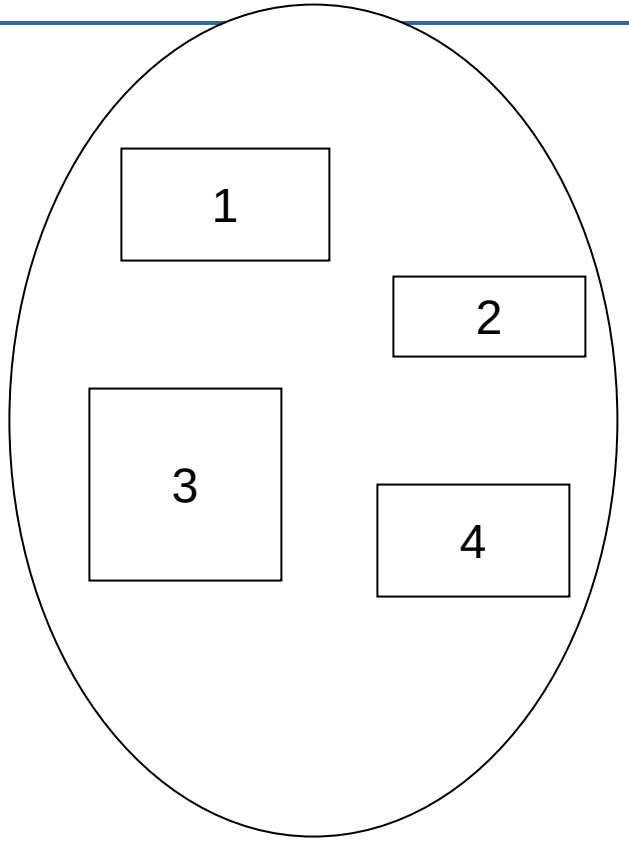


Logical View of Segmentation

- Segmentation은 프로그래머가 생각하는 것을 지원하는 메모리 관리 기법
- 프로그래머가 생각하는 논리 구조 공간은 segment들의 집합으로 이뤄짐
- 각 segment는 특정 이름과 길이 가짐
- Program에서 사용하는 address는 segment 이름과 segment 안에서의 offset값 모두 명시함
- 컴파일러가 프로그램을 반영하여 자동으로 segment 생성함

논리 주소 =
<segment-number, offset>

- C Compiler는 다음과 같은 segment 생성
  - Code, 전역변수, 메모리 할당을 위한 heap, Stack, 표준 C library 영역 등



user space

physical memory spac

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**

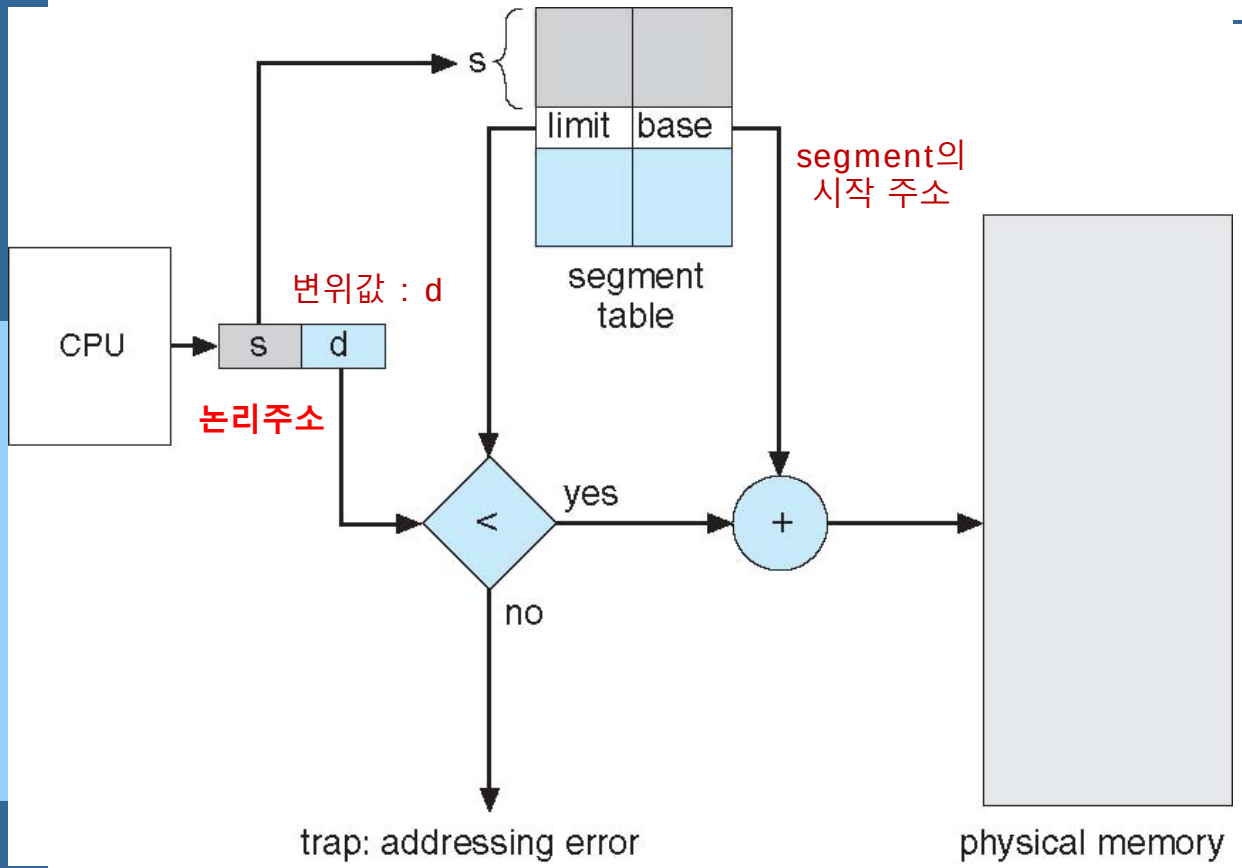
Segmentation Architecture (Cont.)

- Protection
  - With each entry in segment table associate:
    - validation bit = 0 ⇒ illegal segment
    - read/write/execute privileges

valid/invalid

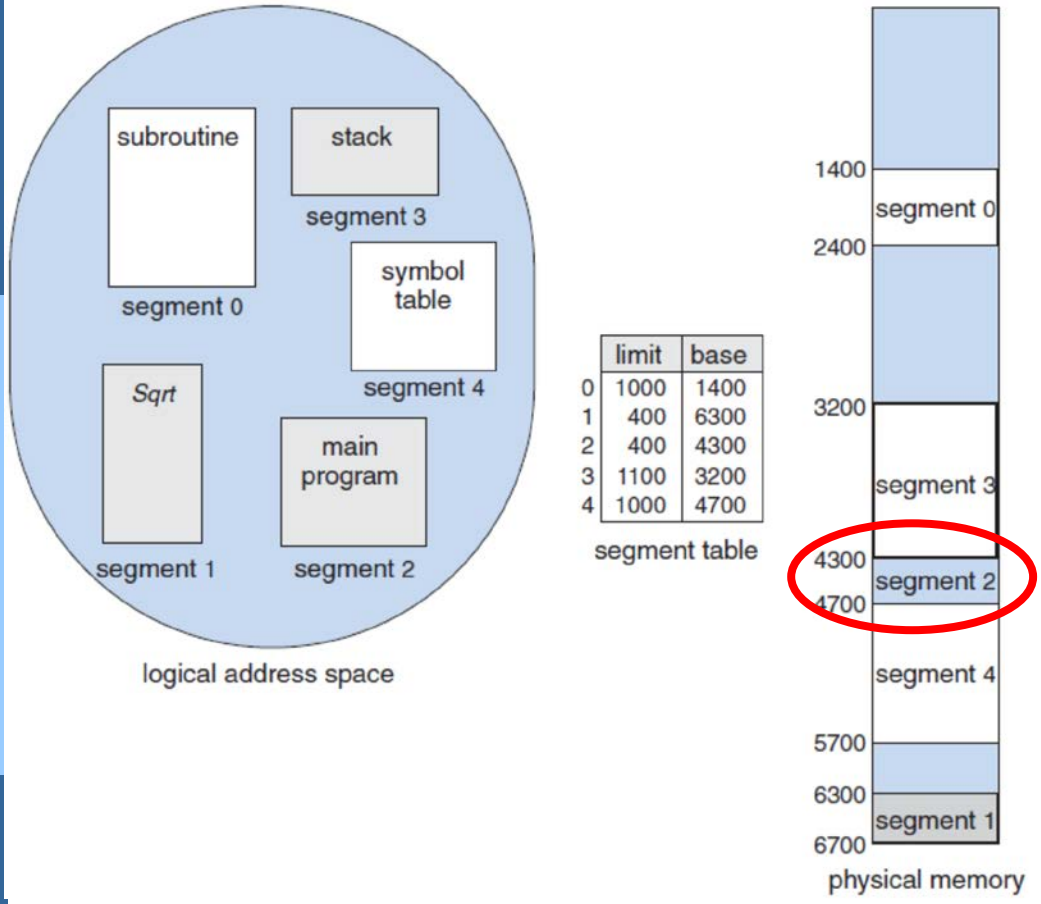
- **Protection bits** associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem

Segmentation Hardware



- **Segment base:** segment의 시작 주소
- **Segment limit:** segment의 길이
- **논리주소는** segment 번호 s와 segment 내의 변위(offset) d로 구성
  s: segment table 색인역할

Example of the Segmentation



- 총 5개의 segment가 있음: segment 0 ~ 4
- 각 segment는 physical memory에 저장되어 있음
- Sement table은 memory내 실제 시작 주소(base)와 segment의 끝(limit) 정보를 제공함
- ex) segment 2: 4300 번지에서 시작하고 길이(limit)은 400 Byte임

Difference b/w Segmentation & Paging

Segmentation	Paging
• Program is divided into variable size segments	• Program is divided into fixed size pages
• User(or compiler) is responsible for dividing the program into segments	• Division into pages is performed by the OS
• Segmentation is slower than paging	• Paging is faster than segmentation
• Segmentation is visible to the user	• Paging is invisible to the user
• Segmentation eliminates internal fragmentation	• Paging suffers from internal fragmentation
• Segmentation suffers from external fragmentation	• There is no external fragmentation
• Processor uses page number, offset to calculate absolute address	• Processor uses segment number, offset to calculate absolute address
• OS maintains a list of free holes in main memory	• OS must maintain a free frame list

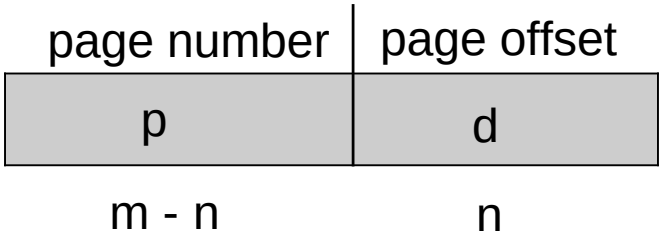
reference: Advanced Microprocessors by A.P.Godse, CH15

Paging

- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames
- **To run a program of size *N* pages, need to find *N* free frames and load program**
- Set up a **page table** to translate logical to physical addresses
- Backing store likewise split into pages
- Still have Internal fragmentation

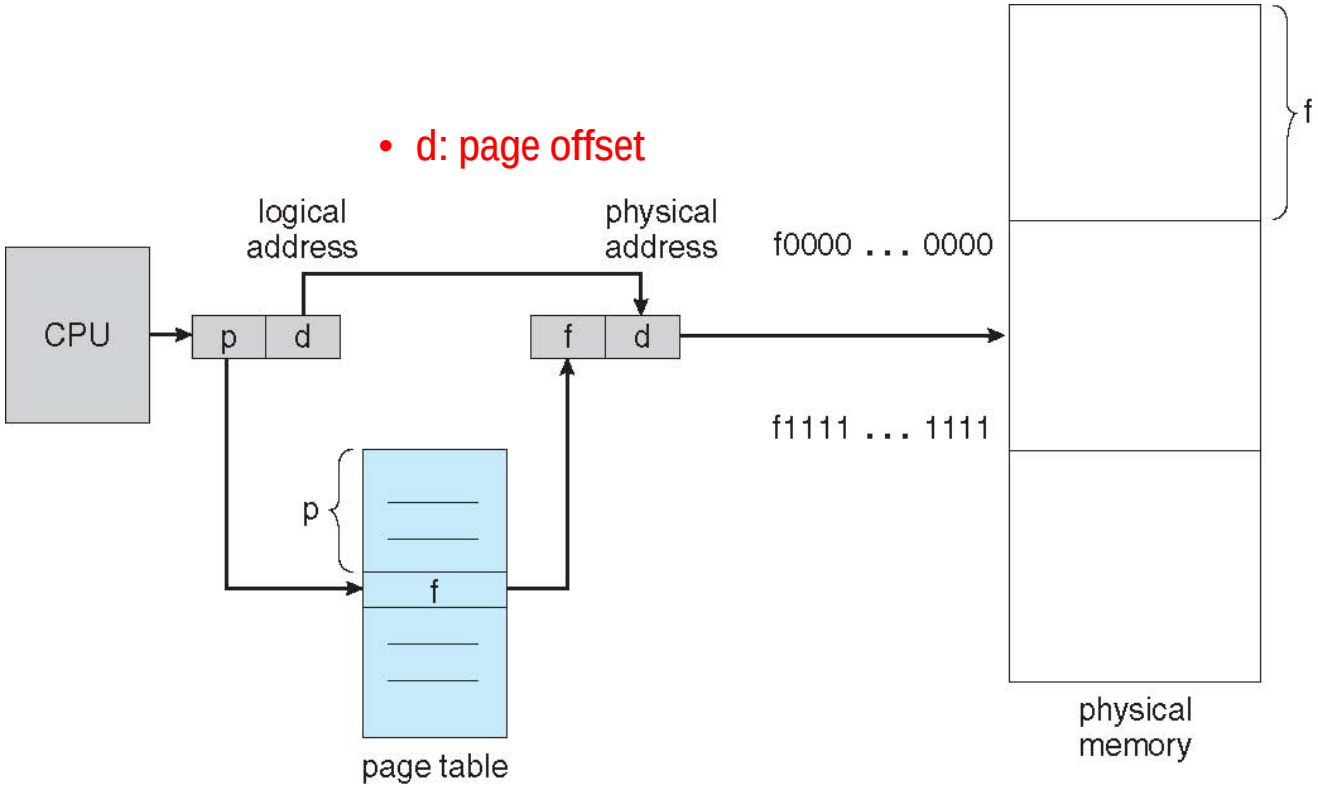
Address Translation Scheme

- 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



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

Paging Hardware

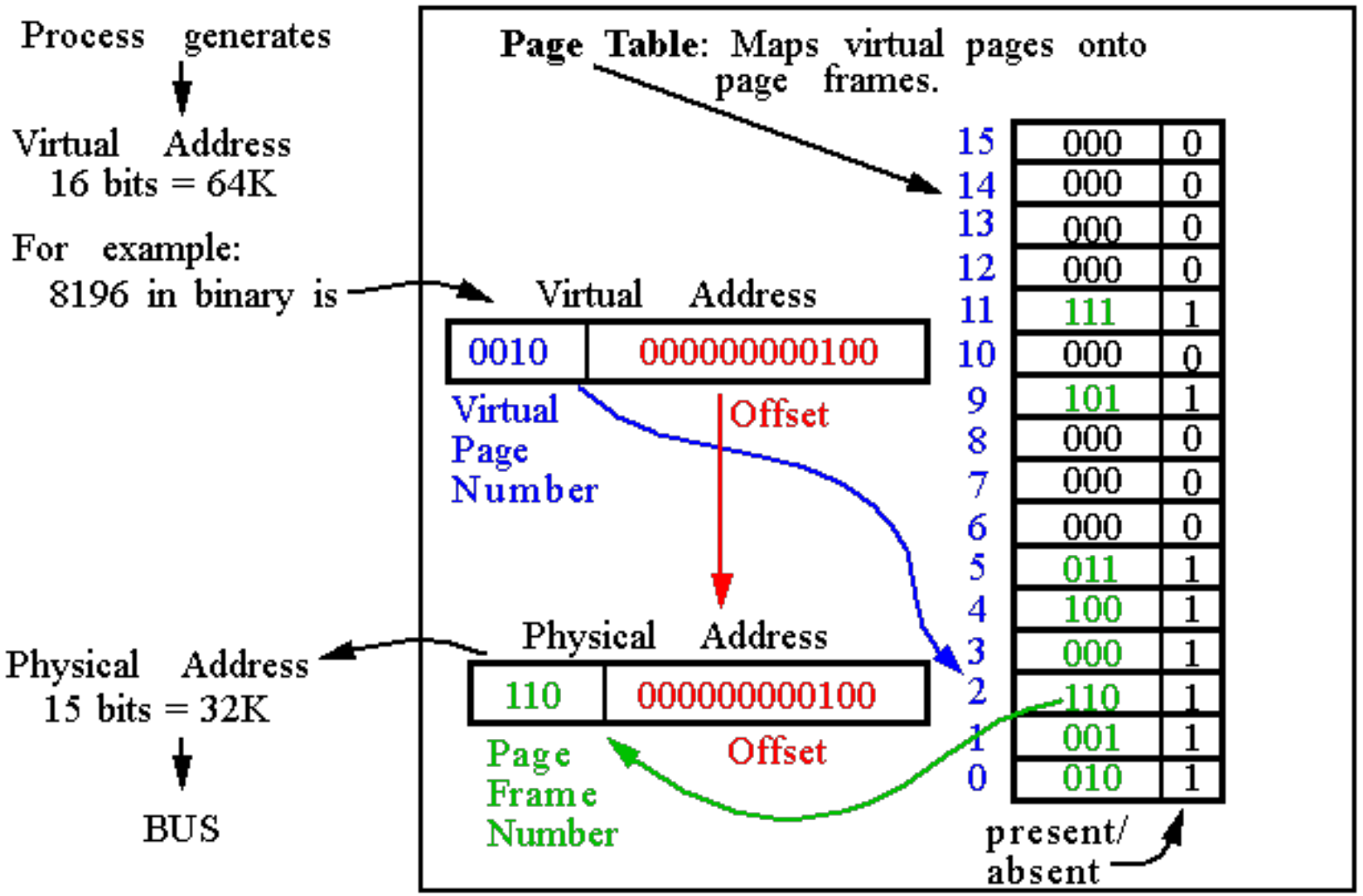


- d: page offset

- 물리 메모리는 frame이라는 동일크기 블록으로 나눠짐

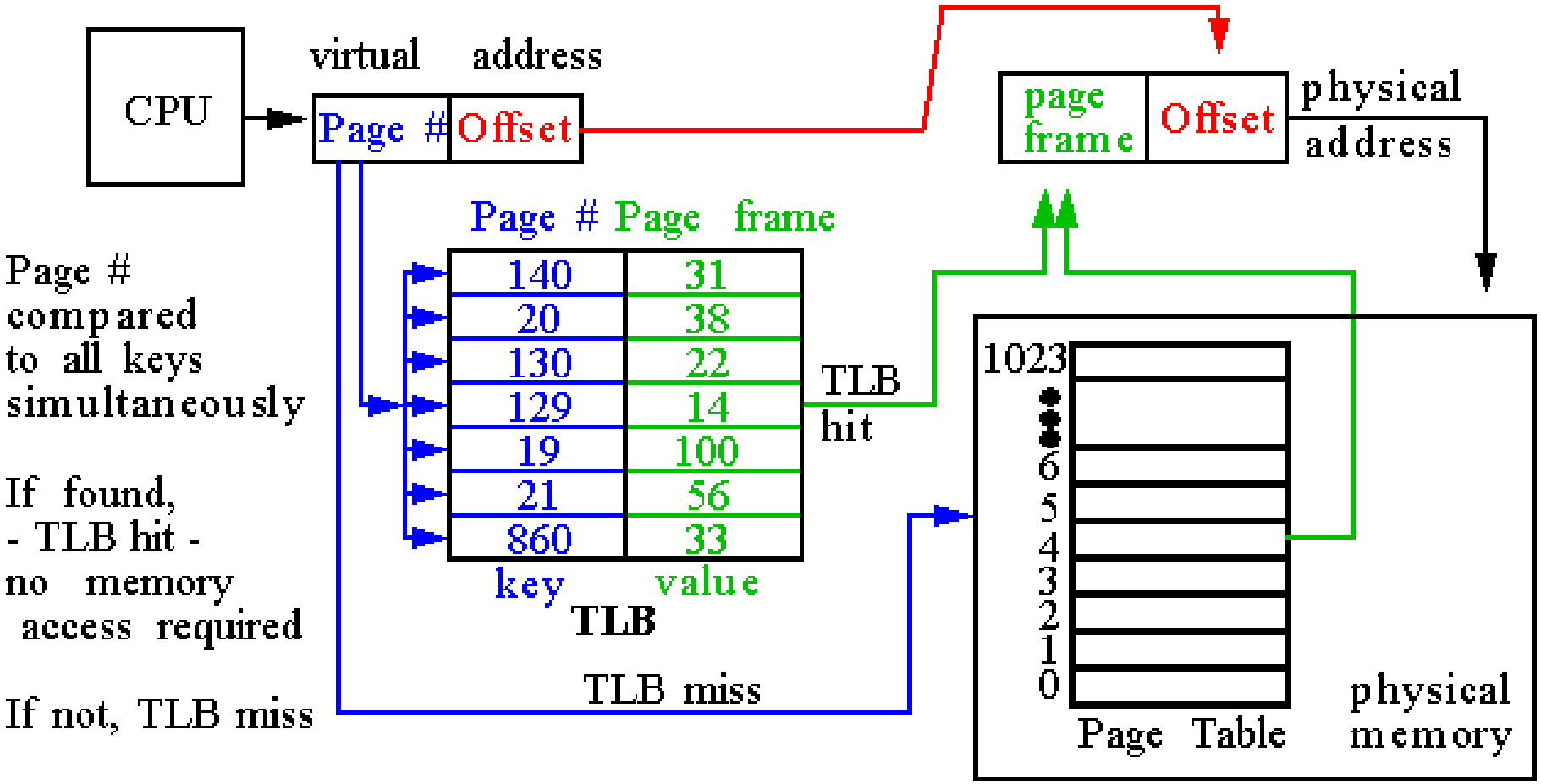
- 논리메모리는 page로 나눠짐(같은 크기 블록임)

Paging Hardware



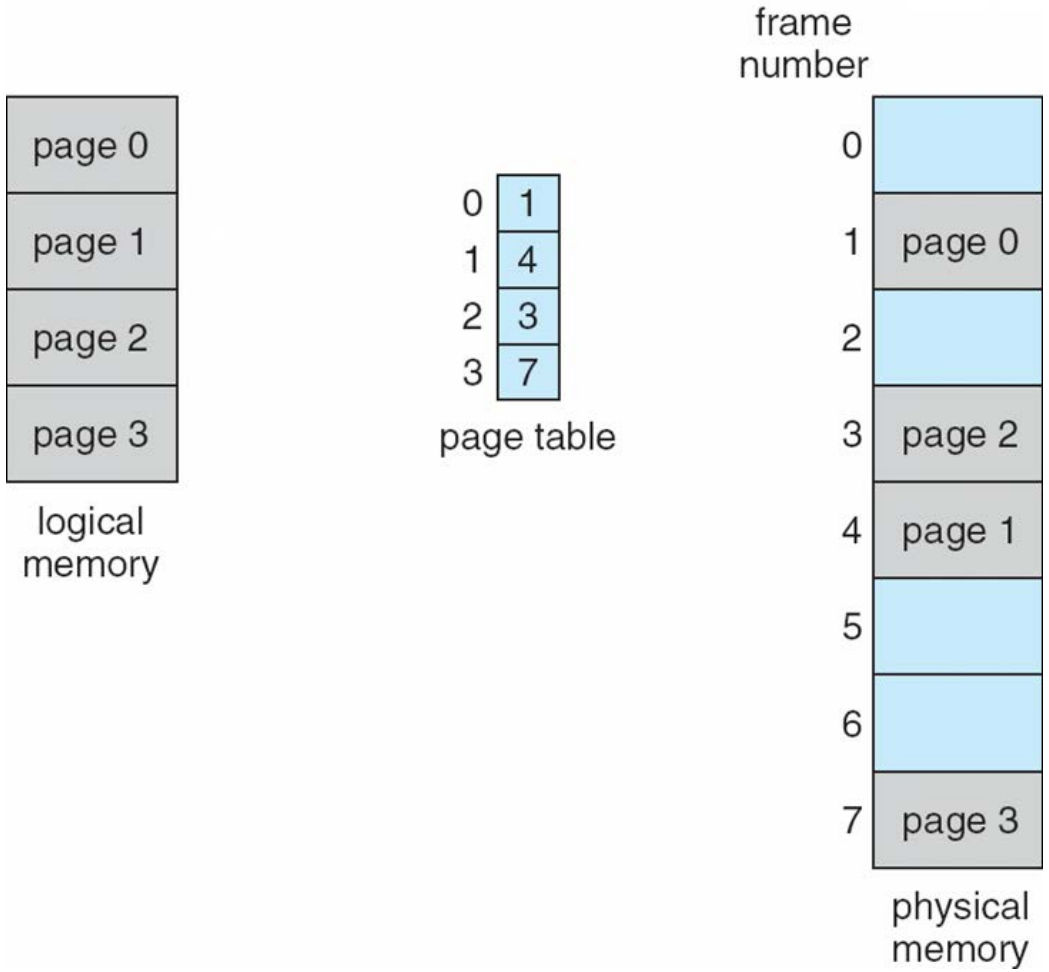
http://ece-research.unm.edu/jimp/310/slides/os_essentials.html

TLB(Translation Lookaside Buffers)



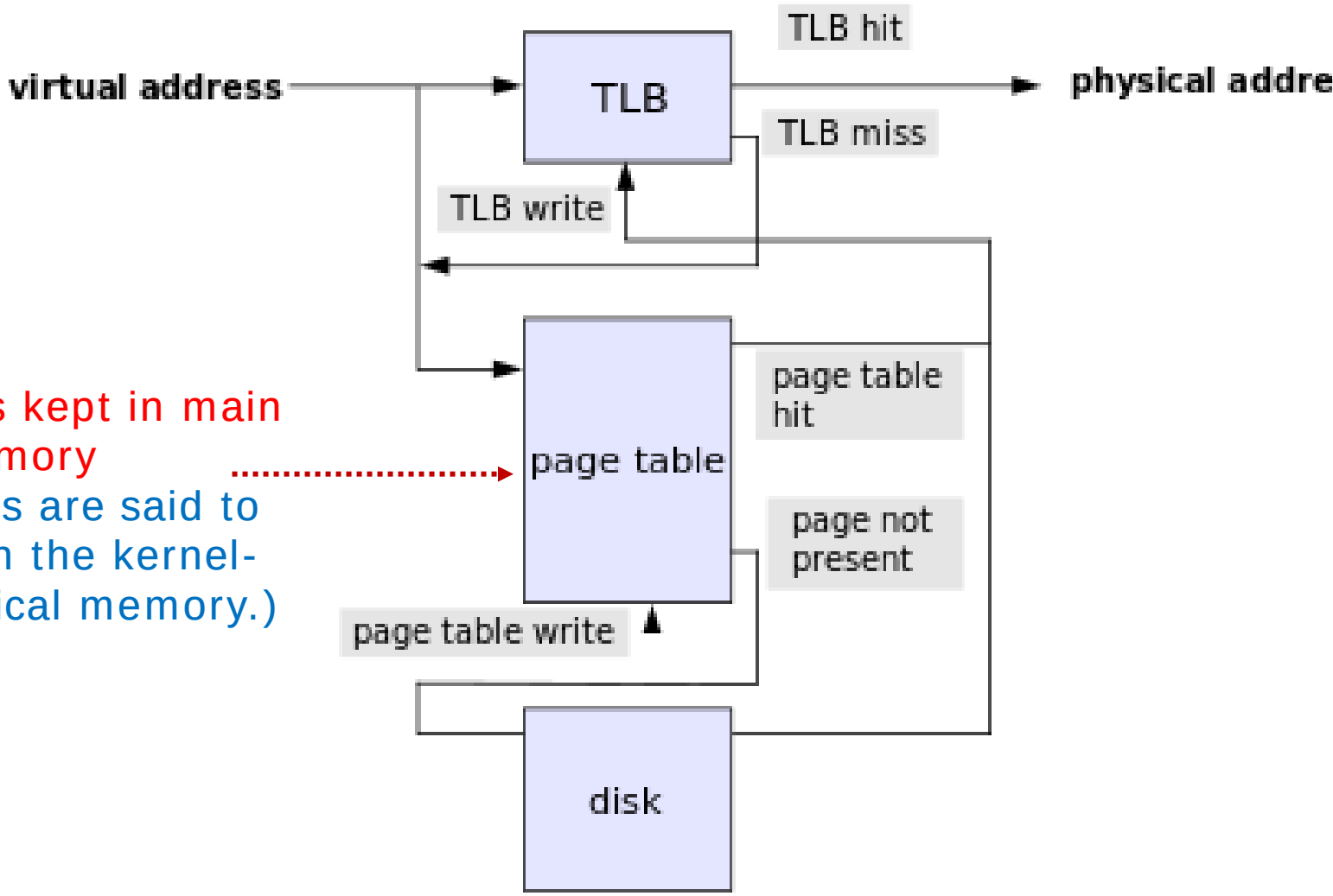
http://ece-research.unm.edu/jimp/310/slides/os_essentials.html

Paging Model of Logical and Physical Memory



Implementation of Page Table

Page table is kept in main memory
(page tables are said to be stored in the kernel-owned physical memory.)



Implementation of Page Table

- Page table is kept in main memory
- **Page-table base register** (**PTBR**) points to the page table
- **Page-table length register** (**PTLR**) indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses
  - One for the page table and one for the data / instruction
- 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**)

Implementation of Page Table (Cont.)

- Some TLBs store **address-space identifiers** (**ASIDs**) in each TLB entry – uniquely identifies each process to provide address-space protection for that process
  - Otherwise need to flush at every context switch
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
  - Replacement policies must be considered
  - Some entries can be **wired down** for permanent fast access

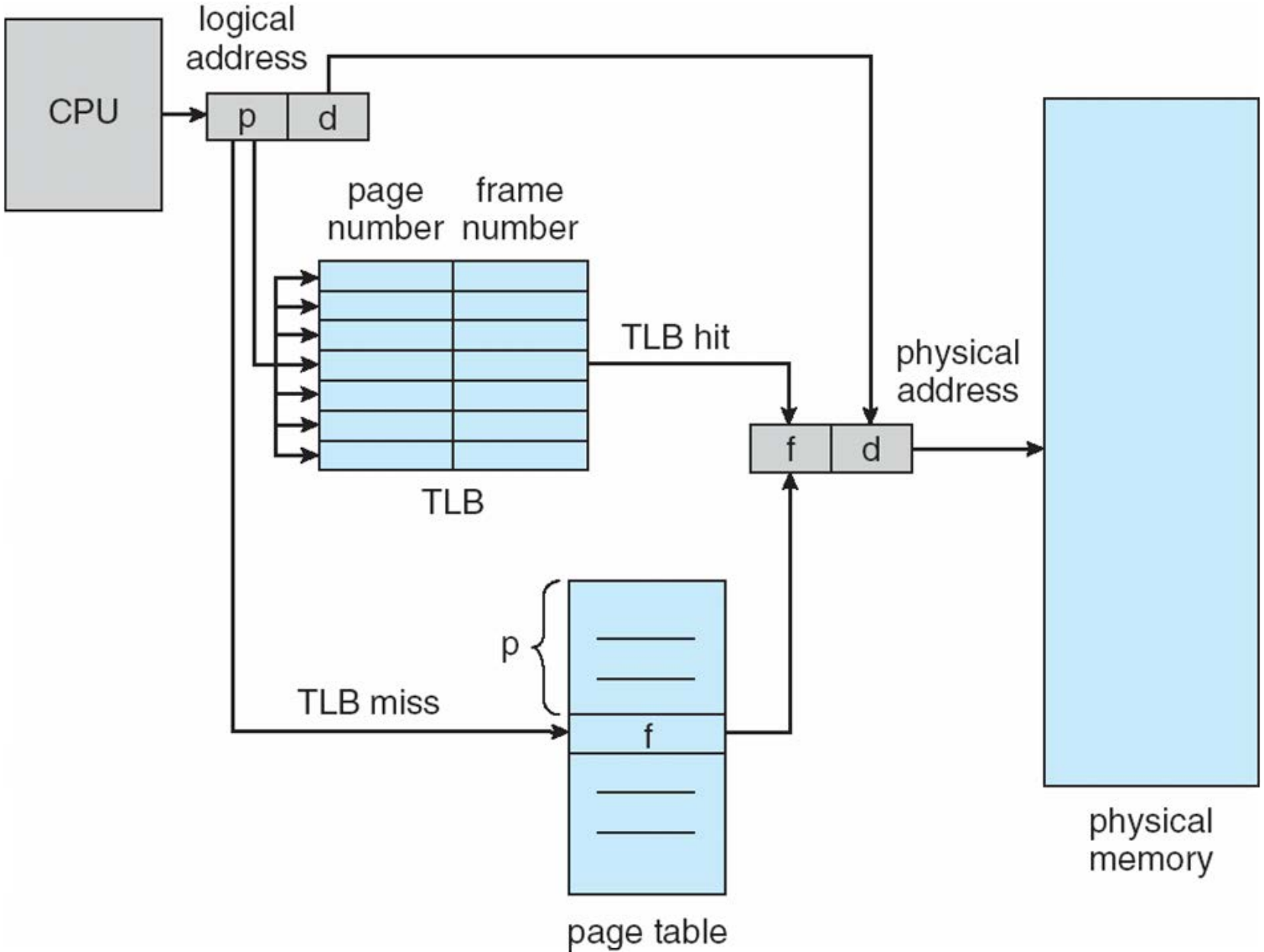
Associative Memory

- 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

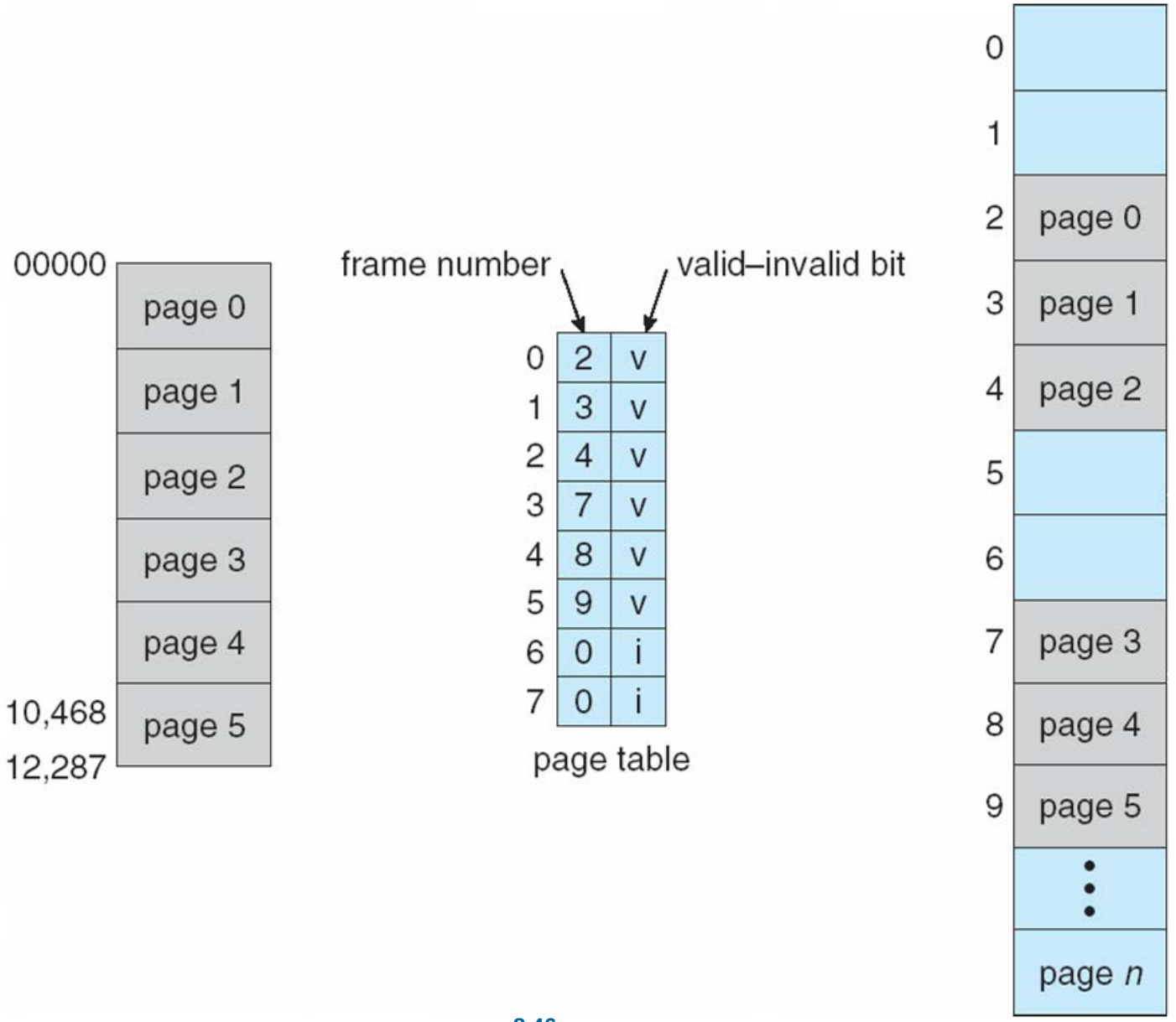
Paging Hardware With TLB



Memory Protection

- **Memory protection** implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
  - Can also add more bits to indicate page execute-only, and so on
- **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
  - Or use **page-table length register** (**PTLR**)
- Any violations result in a trap to the kernel

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

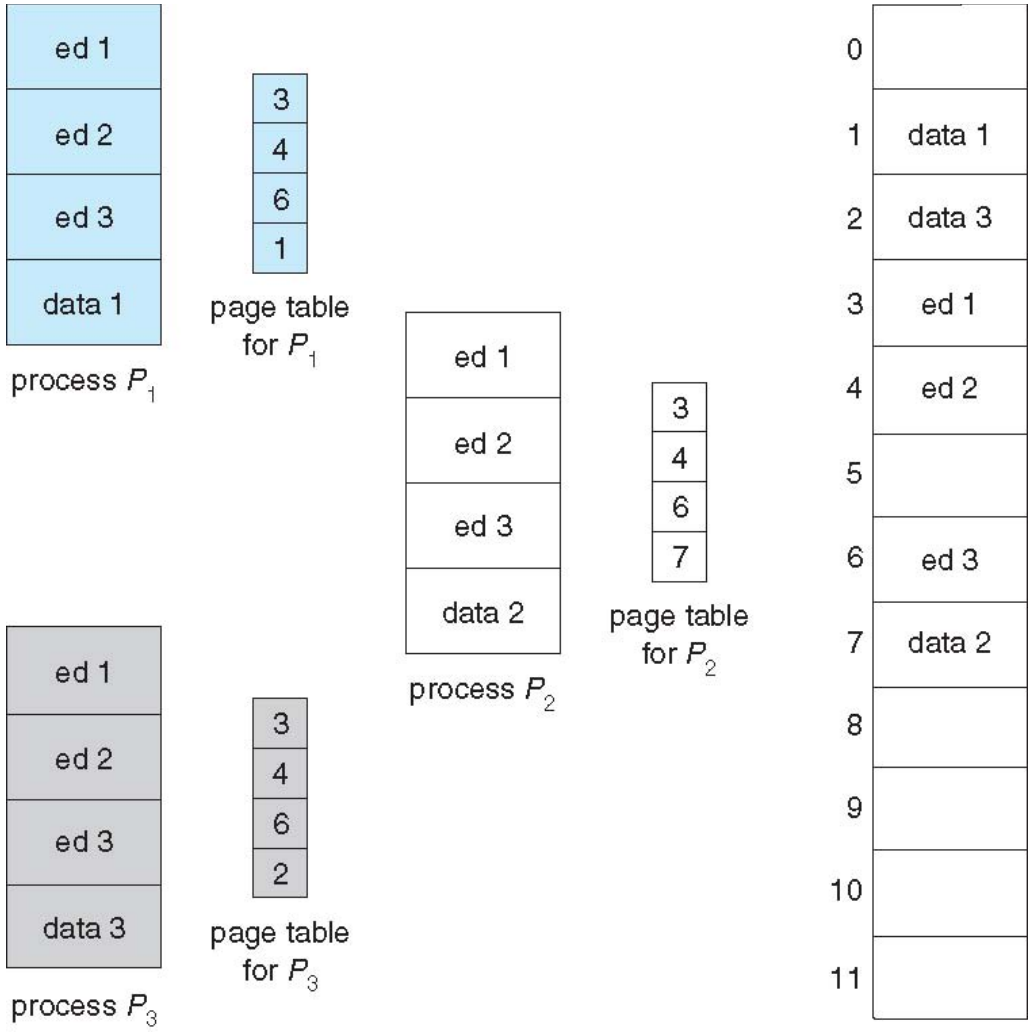


Shared Pages

- **Shared code**
  - One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
  - Similar to multiple threads sharing the same process space
  - Also useful for interprocess communication if sharing of read-write pages is allowed
- **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

Reentrant code can be safely re-entered, meaning that it can be called again even while a call to it is underway. (즉, 실행되는 동안 코드가 바뀌지 않는다는 것이 가장 큰 특성). Reentrant code keeps its state entirely in parameters and local variables, and doesn't use static variables or global variables, and doesn't share aliases to mutable objects with other parts of the program, or other calls to itself."

Shared Pages Example



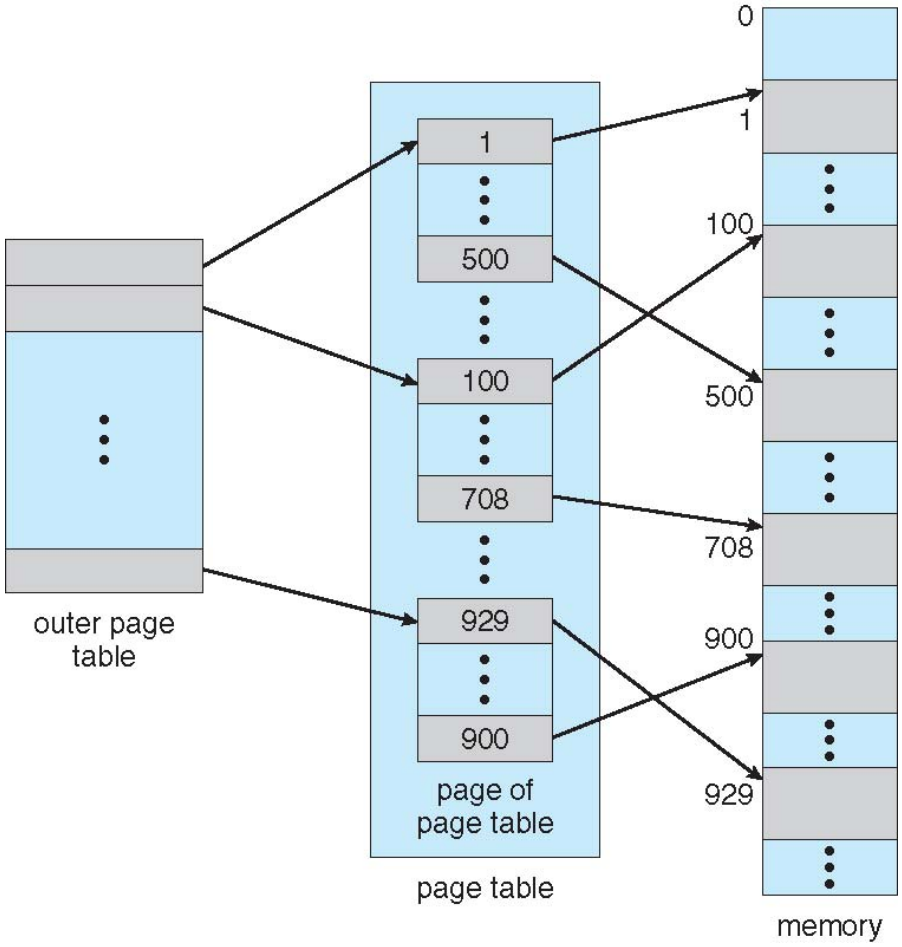
Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB (2^{12})
  - Page table would have 1 million(2^{20}) entries (← 2^{32} / 2^{12})
  - If each entry is 4 bytes -> 4 MB (← 4B * 2^{20}) of physical address space / memory for page table alone
    - That amount of memory used to cost a lot
    - Don't want to allocate that contiguously in main memory
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Page Tables

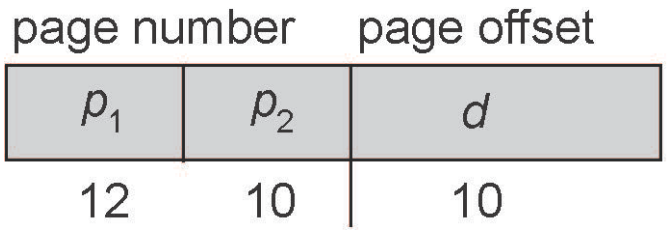
- **Break up the logical address space into multiple page tables**
- A simple technique is a two-level page table
- We then page the page table

Two-Level Page-Table Scheme



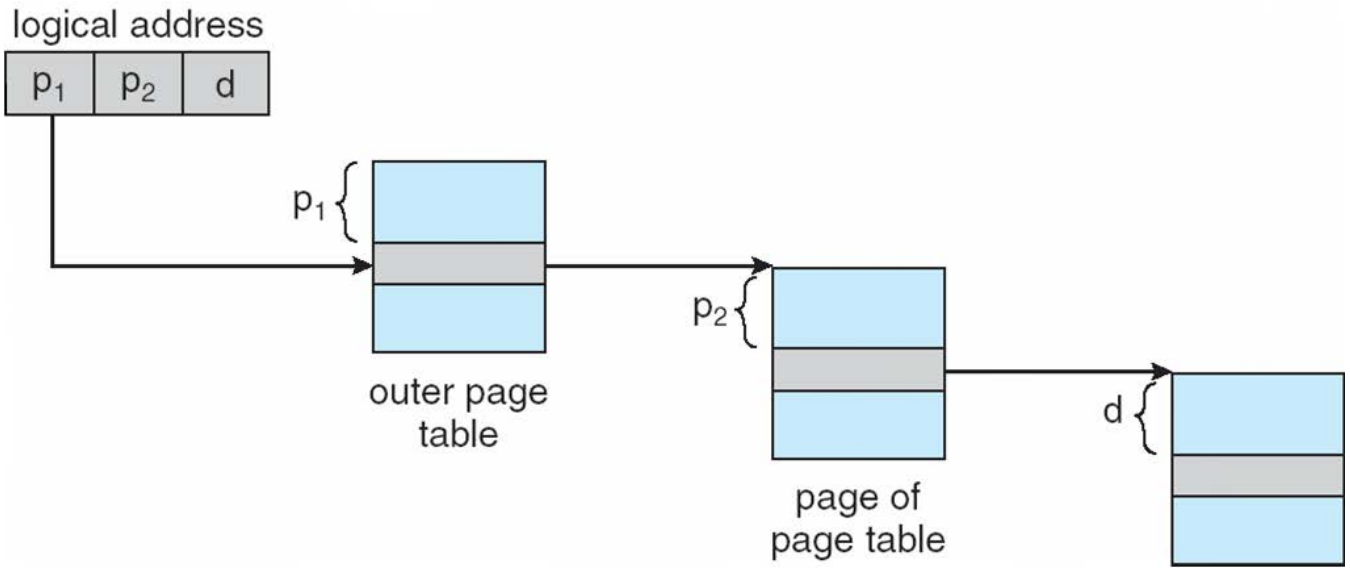
Two-Level Paging Example

- 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:



- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**

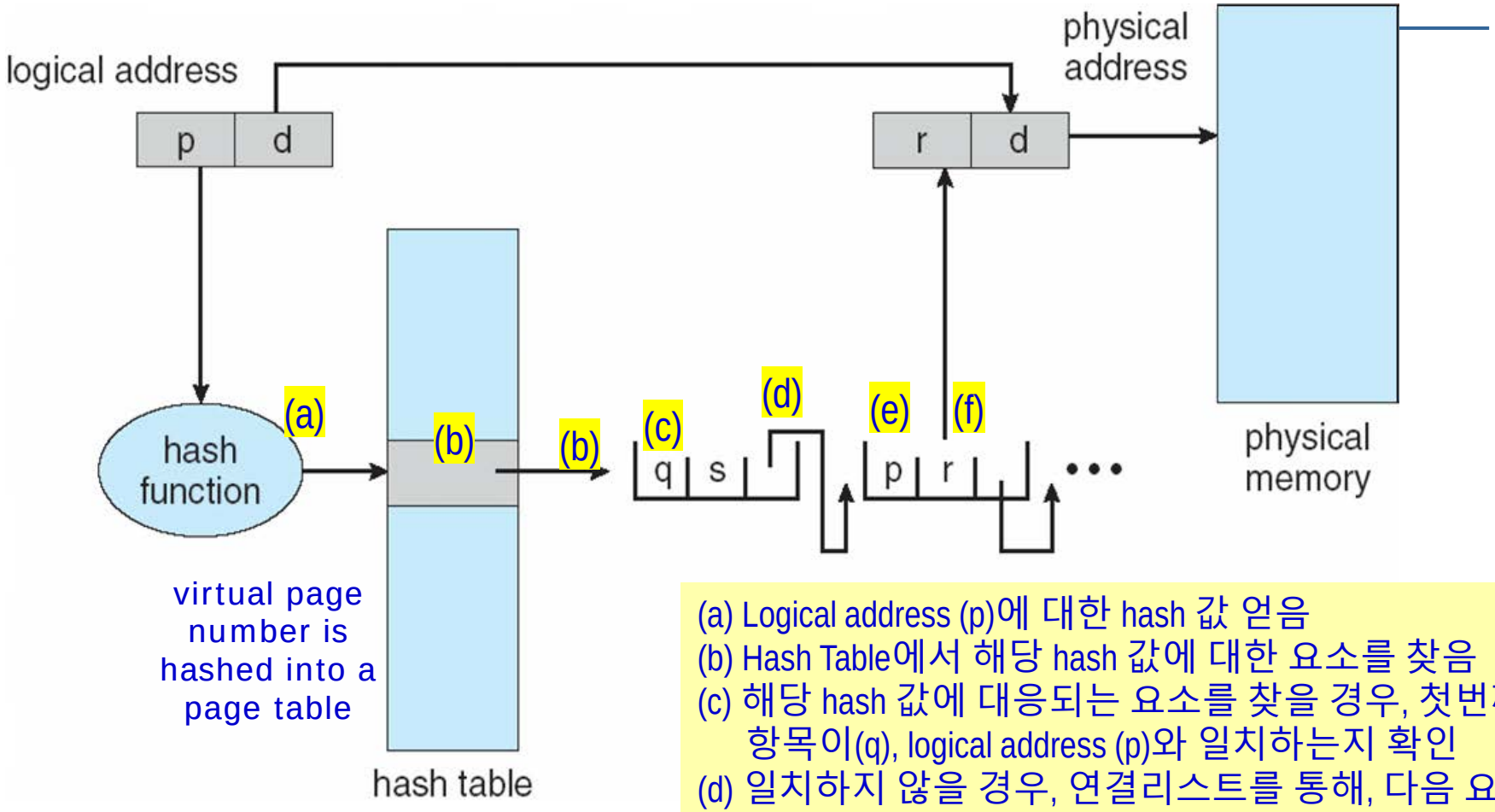
Address-Translation Scheme



Hashed Page Tables

- Hashed PT 기법은 address spaces가 32 bits보다 큰 경우 많이 사용됨
  - 즉, 주소 공간이 32bit보다 커지면 가상 주소를 해쉬화해서 사용하는 해쉬 페이지 테이블을 많이 사용함
- **The virtual page number is hashed into a page table**
- Each element contains **(1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element**
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted

Hashed Page Table



(1) the virtual page number : q, p
(2) the value of the mapped page frame : s, r
(3) a pointer to the next element

(a) Logical address (p)에 대한 hash 값 얻음
(b) Hash Table에서 해당 hash 값에 대한 요소를 찾음
(c) 해당 hash 값에 대응되는 요소를 찾을 경우, 첫번째 항목이(q), logical address (p)와 일치하는지 확인
(d) 일치하지 않을 경우, 연결리스트를 통해, 다음 요소를 찾음
(e) 동일하게 (1) 첫번째 값 비교.→ 동일함.
(f) 두번째 값이 page frame임. 이를 통해 physical address 얻음

Inverted Page Table

- **기존 Page table의 문제점**
  - Most of the Operating Systems implement a separate page table for each process, i.e. for ‘n’ number of processes running on a Multiprocessing/ Timesharing operating system, there are ‘n’ number of page tables stored in the memory.
  - Sometimes when a process is very large in size and it occupies virtual memory then with the size of the process, it’s page table size also increases substantially

만약, 이러한 경우,
(1)다중 프로세스가 있다면? → Page Table용 메모리 소비가 많음
(2)또한 multilevel Page Table을 사용한다면 ? 메모리 사용량 많음 !

- Page Table 크기 사례 :
  - A process of size 2 GB with:
  - Page size = 512 Bytes
  - Size of page table entry = 4 Bytes, then
  - Number of pages in the process = 2 GB / 512 B = 2^{22}
  - Page Table Size = $2^{22} * 2^{2} = 2^{24}$ bytes (16MBytes)

https://www.geeksforgeeks.org/inverted-page-table-in-operating-system/

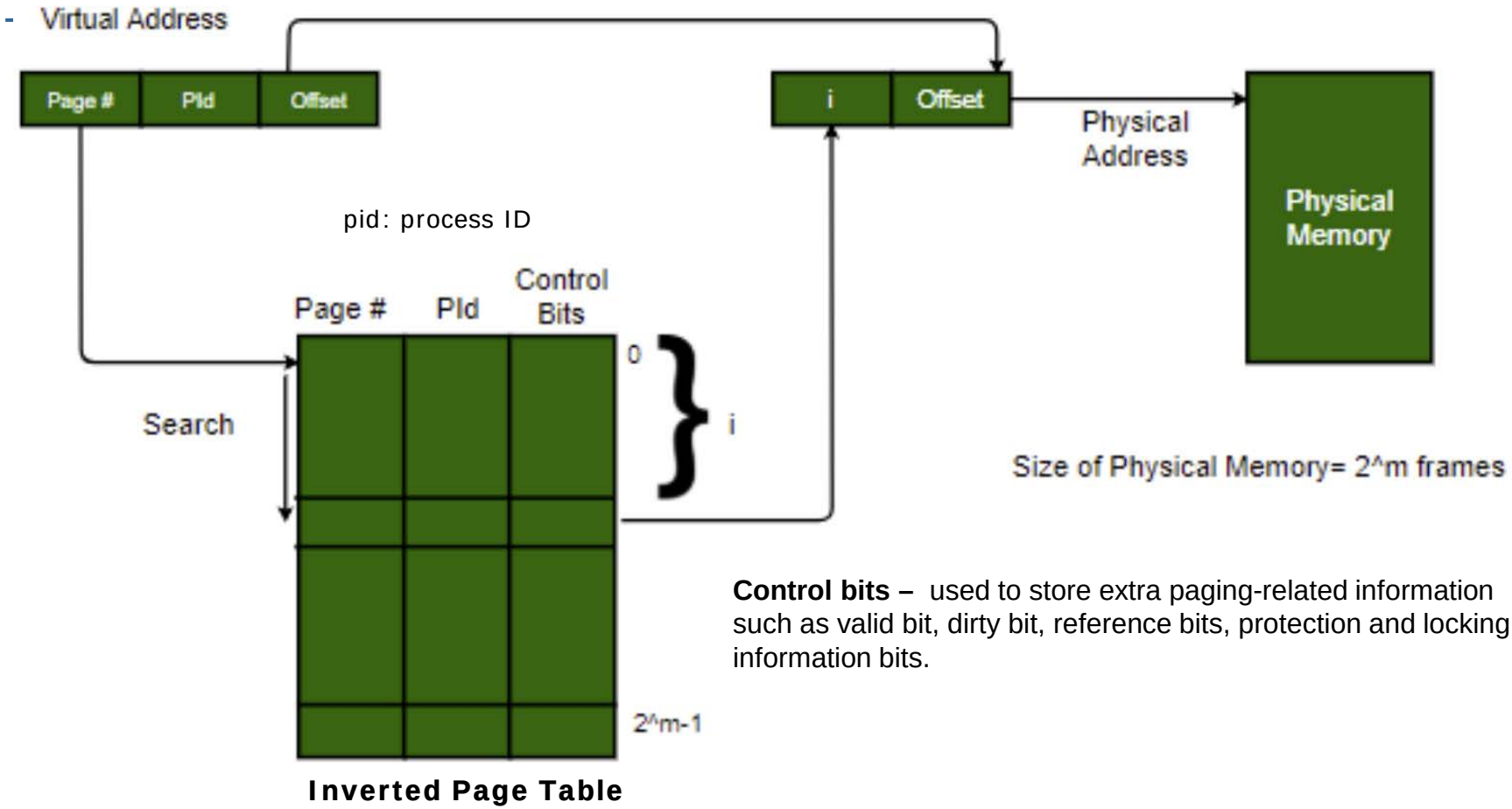
Inverted Page Table

- **Inverted Page Table 기법**
  - one-page table entry for every frame of the main memory.
  - The number of page table entries in the Inverted Page Table reduces to the number of frames in physical memory
  - A single page table is used to represent the paging information of all the processes.

- Through the inverted page table,
  - the overhead of storing an individual page table for every process gets eliminated and only a fixed portion of memory is required to store the paging information of all the processes together.
  - **"inverted paging"**이라는 것은 logical page number에 의해 인덱싱 되는 것이 아니라, physical memory frame number에 의해 인덱싱되기 때문임

https://www.geeksforgeeks.org/inverted-page-table-in-operating-system/

Inverted Page Table Architecture



Control bits – used to store extra paging-related information such as valid bit, dirty bit, reference bits, protection and locking information bits.

- The virtual address generated by the CPU contains the fields and each page table entry contains and the other relevant information required in paging related mechanism.
- When a memory reference takes place, this virtual address is matched by the memory-mapping unit and the Inverted Page table is searched to match the and the corresponding frame number is obtained.
- If the match is found at the i^{th} entry then the physical address of the process, *the value is sent as the real address otherwise if no match is found then Segmentation Fault is generated.*

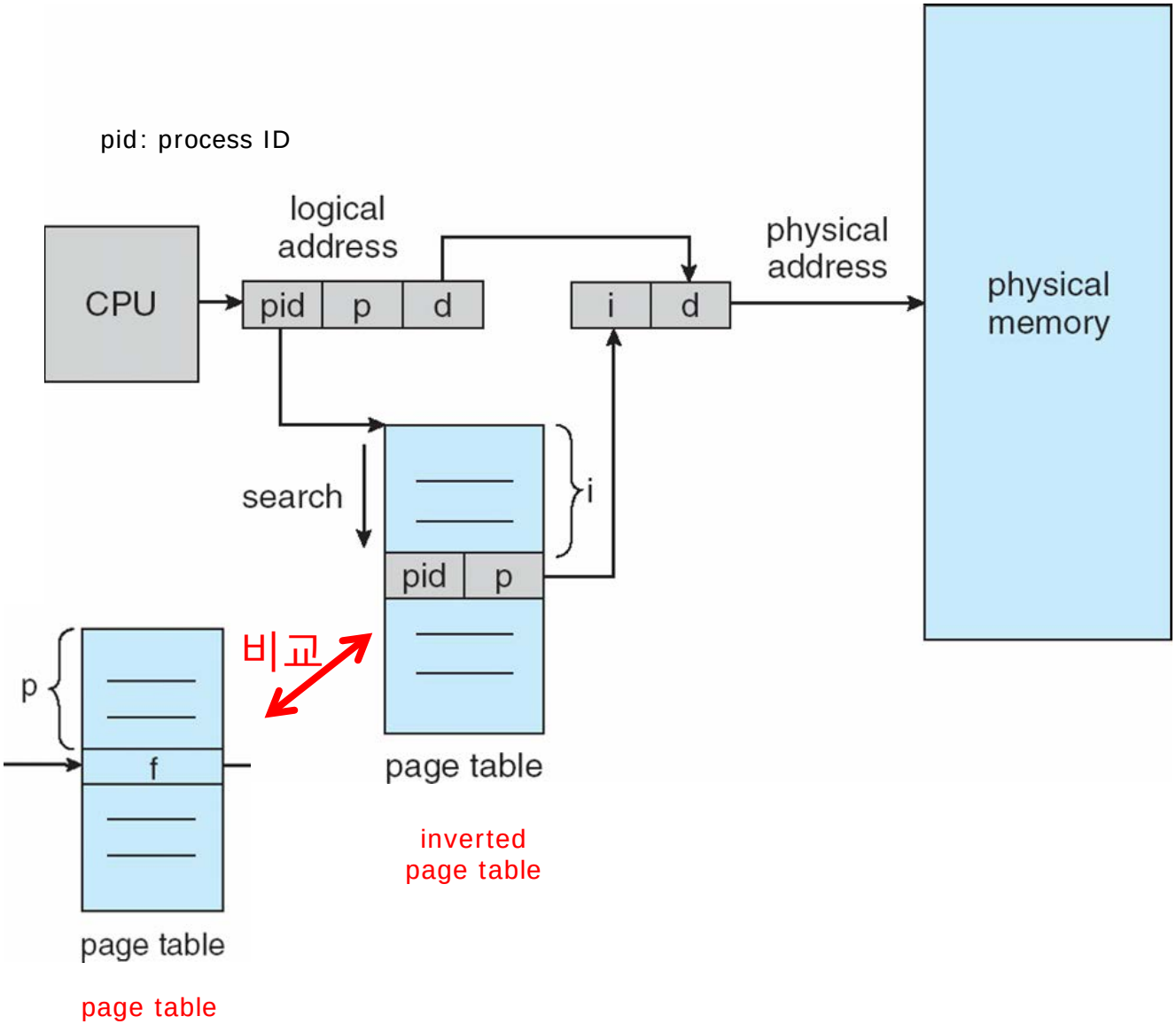
https://www.geeksforgeeks.org/inverted-page-table-in-operating-system/

Inverted Page Table Architecture

- **Advantages and Disadvantages of Inverted Page Table**
- **Reduced memory space :**
  - Inverted Page tables typically reduces the amount of memory required to store the page tables to a size bound of physical memory. The maximum number of entries could be the number of page frames in the physical memory.
- **Longer lookup time :**
  - Inverted Page tables are sorted in order of frame number but the memory look-up takes place with respect to the virtual address,
  - so, it usually takes a longer time to find the appropriate entry but often these page tables are implemented using hash data structures for a faster lookup.
- **Difficult shared memory implementation :**
  - As the Inverted Page Table stores a single entry for each frame, it becomes difficult to implement the shared memory in the page tables. Chaining techniques are used to map more than one virtual address to the entry specified in order of frame number.

https://www.geeksforgeeks.org/inverted-page-table-in-operating-system/

Inverted Page Table Architecture



End of Chapter 8