PART 3 MEMORY MANAGEMENT

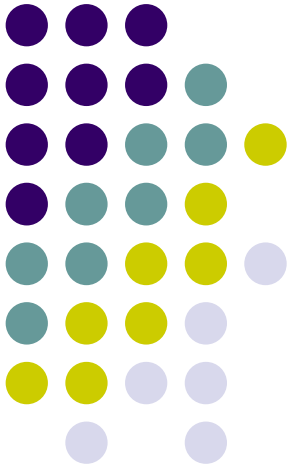
Chapter 8: Memory Management Strategies

WHAT'S AHEAD:

- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table
- Example: Intel IA-32

WE AIM:

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



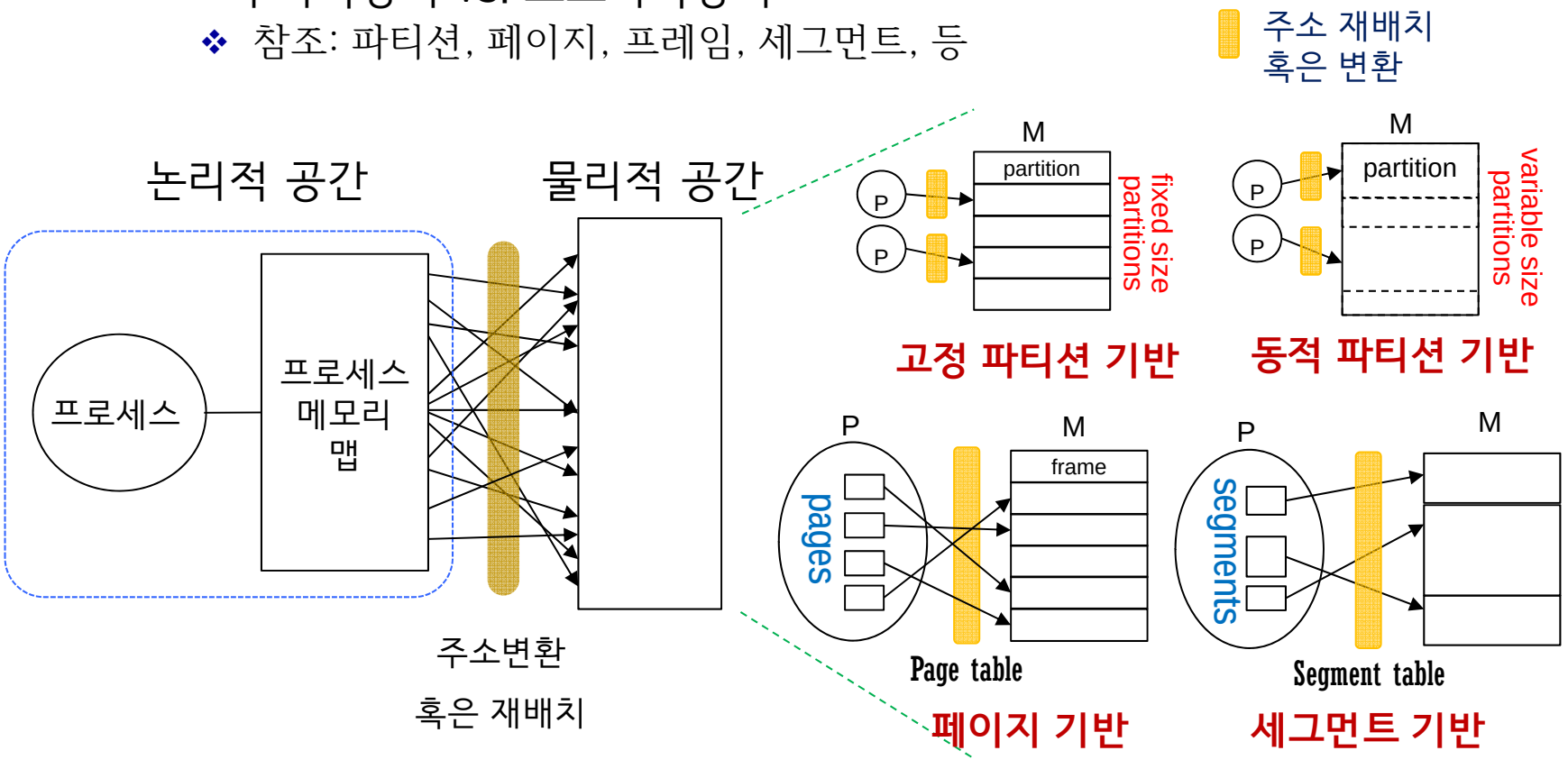
Note: These lecture materials are based on the lecture notes prepared by the authors of the book titled *Operating System Concepts*, 9e (Wiley)

핵심•요점 기억공간을 어떻게 조직화할 것인가

- **기억/주소 공간을 블록(block) 단위로 조직화한다**

  블록 ≡ 인접한 기억장소들의 집합

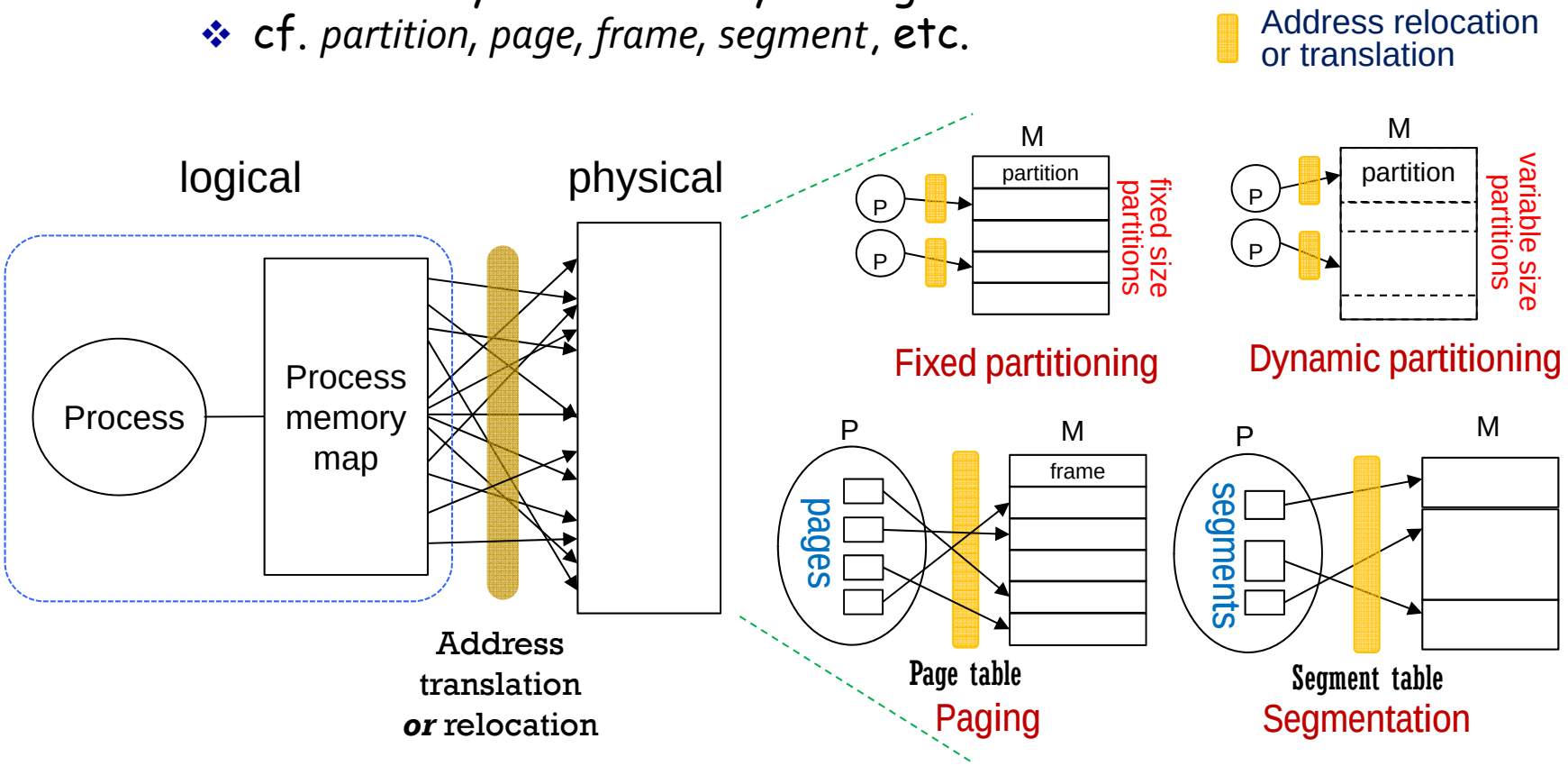
  - 고정 크기 vs. 가변 크기
  - 물리 기억(메모리) vs. 논리 주소공간
  - 주기억장치 vs. 보조기억장치
  - ❖ 참조: 파티션, 페이지, 프레임, 세그먼트, 등



Core Ideas

How to Organize Memory Space

- **We organize the memory/address space in blocks**
  - Block ≡ A set of contiguous memory locations
    - fixed size vs. variable size
    - physical memory vs. logical address space
    - main memory vs. secondary storage
    - ❖ cf. *partition*, *page*, *frame*, *segment*, etc.

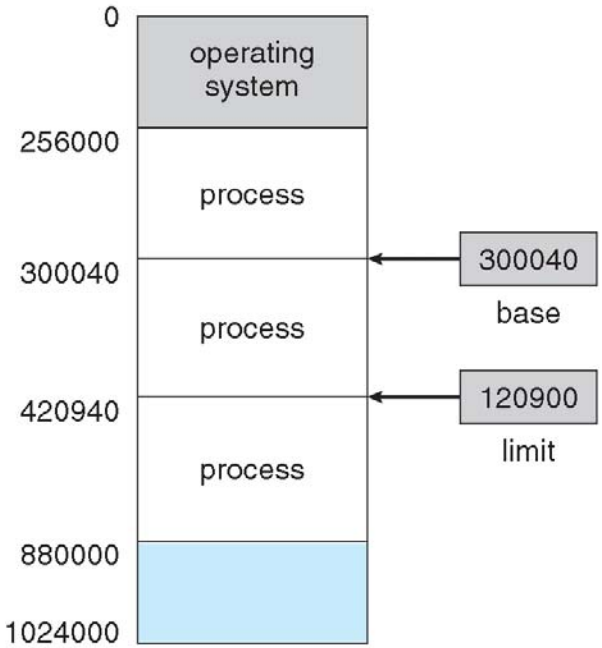


Background

- Program must be brought (from disk) into memory and placed within a process for it to be run (← von Neumann architecture)
- Main memory (including cache memory) and registers are only storage 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

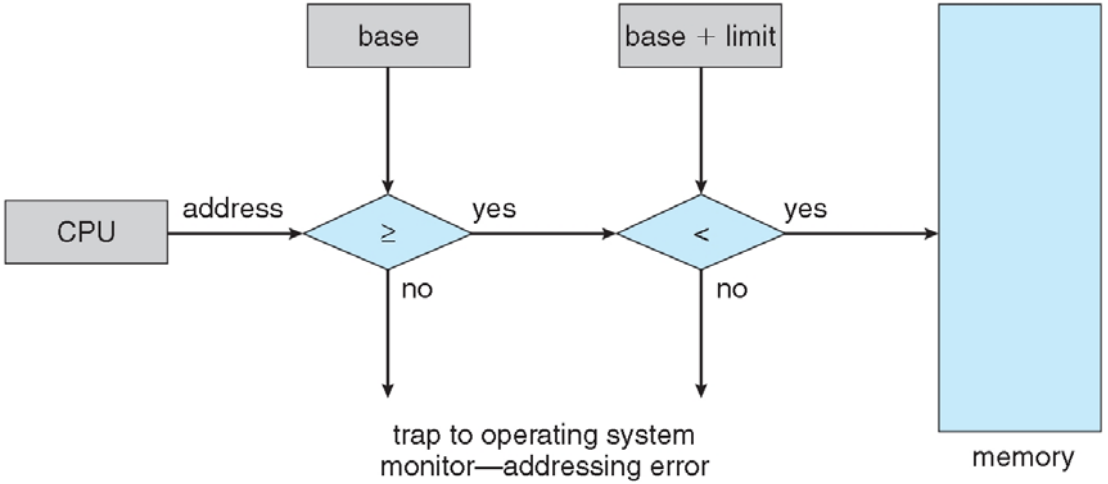
Base and Limit Registers

- Addressing: Where to get instruction/data?
- A pair of **base** and **limit registers** define the logical address space that a process deals with
- For address protection, CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



A base and a limit register define a logical address space

Hardware address protection with base and limit registers



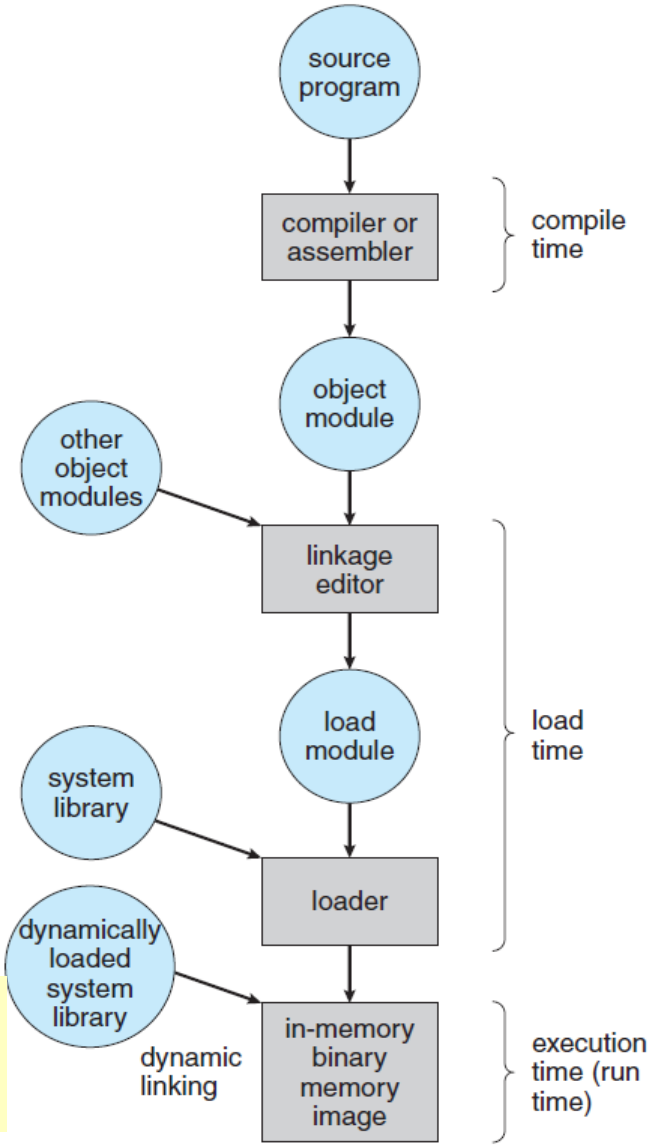
Address Binding

- Programs on disk, ready to be brought into memory to execute from an **input queue**
  - Without support, must be loaded into address 0000
- Inconvenient to have first user process physical address always at 0000
  - How can it not be?
- Further, addresses represented in different ways at different stages of a program's life
  - Source code addresses are usually symbolic
  - Compiled code addresses **bind** to relocatable addresses
    - i.e. "14 bytes from beginning of this module"
    - "relative addressing mode" in assembly code
  - Linker or loader will bind relocatable addresses to absolute addresses
    - i.e. 74014
  - Each binding maps one address to another

Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: If memory location known a priori, absolute code can be generated; must recompile code if starting location changes
  - **Load time**: Compiler must generate relocatable code if memory location is not known at compile time
  - **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another
    - Need hardware support for address maps (e.g., base and limit registers)

Multistep Processing of a User Program



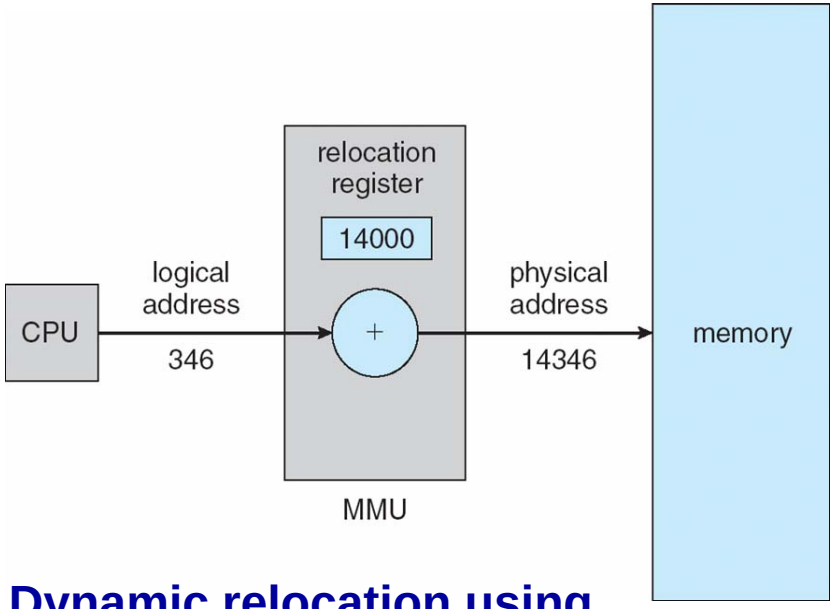
Dynamic link library (dll) or dynamically loaded system library is loaded by either loader or runtime environment.

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 addresses are dealt with in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by OS and/or hardware units

Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address
- Many methods possible, covered in the rest of this chapter
- To start with, 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
- 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

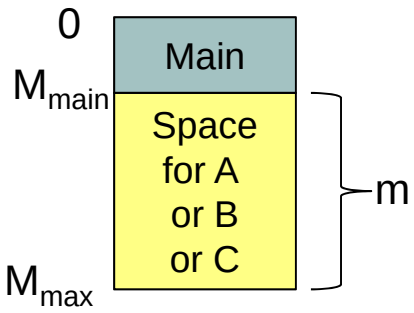


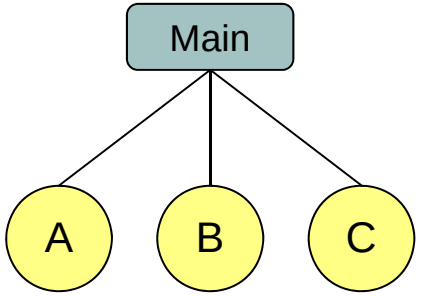
Dynamic relocation using a relocation register

Overlays – A Dynamic Loading Method

- What if a process needs more memory space than the main memory can provide?
  - We may opt to use the overlays memory management technique
- Overlays is to keep in memory only those instructions and data that are needed at any given time.
- When other routines are needed at runtime, they are loaded into space that was occupied previously by those routines that are no longer needed.
- No special support from the operating system is required
  - Implemented through program design by a programmer
  - Primary responsibility is given to the programmer

- A simple example
  - A program has four routines: *Main*, *A*, *B*, *C*
    - *Main* calls *A*, *B* & *C*, one at a time.
    - *A*, *B* & *C* do not call each other
  - Each of *A*, *B* and *C* fits in *m*, but no two routines can fit in *m*
    - $m = M_{max} - M_{Main}$
    - So only *Main* & *A*, *Main* & *B*, or *Main* & *C* need be in memory at any one time.
  - *A*, *B*, *C* can be kept on disk and copied into the same space when needed.
  - Now, the routines *A*, *B* & *C* can be overlayed.





Dynamic Linking

- **Static linking** – system libraries and program code combined by the linker into the binary program image
- Dynamic linking –linking postponed until load time or execution time
- For dynamic linking
  - Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
  - Stub replaces itself with the address of the routine, and executes the routine
  - Operating system checks if routine is in another process's memory space → if so, see if sharable
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**
- Consider applicability to patching system libraries
  - Versioning performed automatically without re-compilation and re-linking

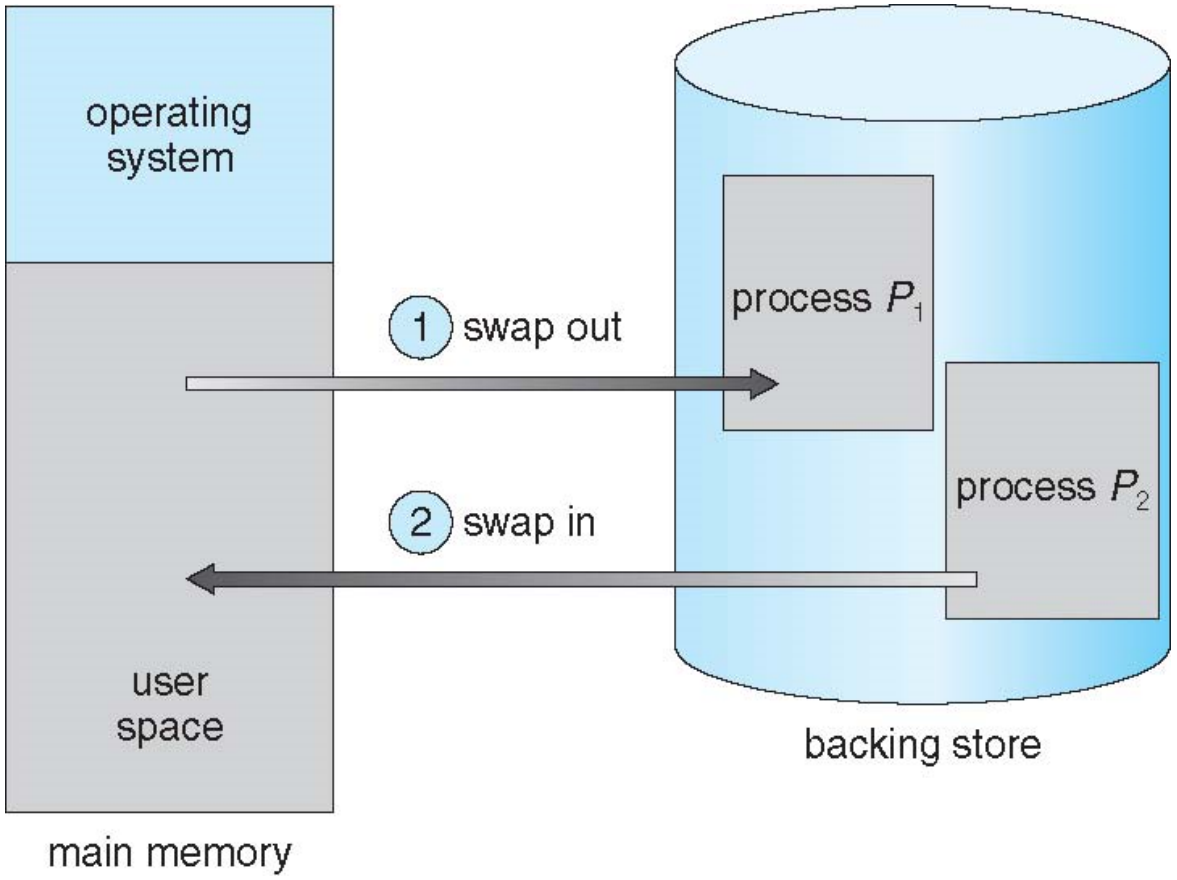
Swapping

- 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 all processes can exceed physical memory
  - System maintains a **ready queue** of ready-to-run processes which have memory images on disk
  - Whenever the CPU scheduler decides to execute a process, it calls the dispatcher.
  - The dispatcher checks to see whether the next process in the queue is in memory. If it is not, and if there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process.
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images

Swapping (Cont.)

- Major part of swap time is transfer time
  - transfer time = amount of data to transfer / transfer rate
  - transfer rate: rate at which data flow between disk drive and computer
- Does the swapped out process need to swap back in to same physical addresses?
  - Depends on address binding method
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if the amount of free memory falls below a threshold amount
  - Disabled again once the amount of free memory increases
- Standard swapping not used in modern OSs
  - But modified version common
    - Swap only when free memory extremely low

Schematic View of Swapping



Context Switch Time including Swapping

- If next process to be put on CPU is not in memory, need to swap out a process and swap in target process
  - Context switch time can then be very high
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
  - Swap out time of 2000 ms
  - Plus swap in of same sized process
  - Total context switch swapping component time of 4000ms (4 seconds)
- Can reduce if reduce size of memory swapped – by knowing how much memory really being used
  - System calls to inform OS of memory use via `request memory()` and `release memory()`
- Other constraints as well on swapping
  - Pending I/O – can't swap out as I/O would occur to wrong process
  - Or always transfer I/O to kernel space, then to I/O device
    - Known as **double buffering**, adds overhead

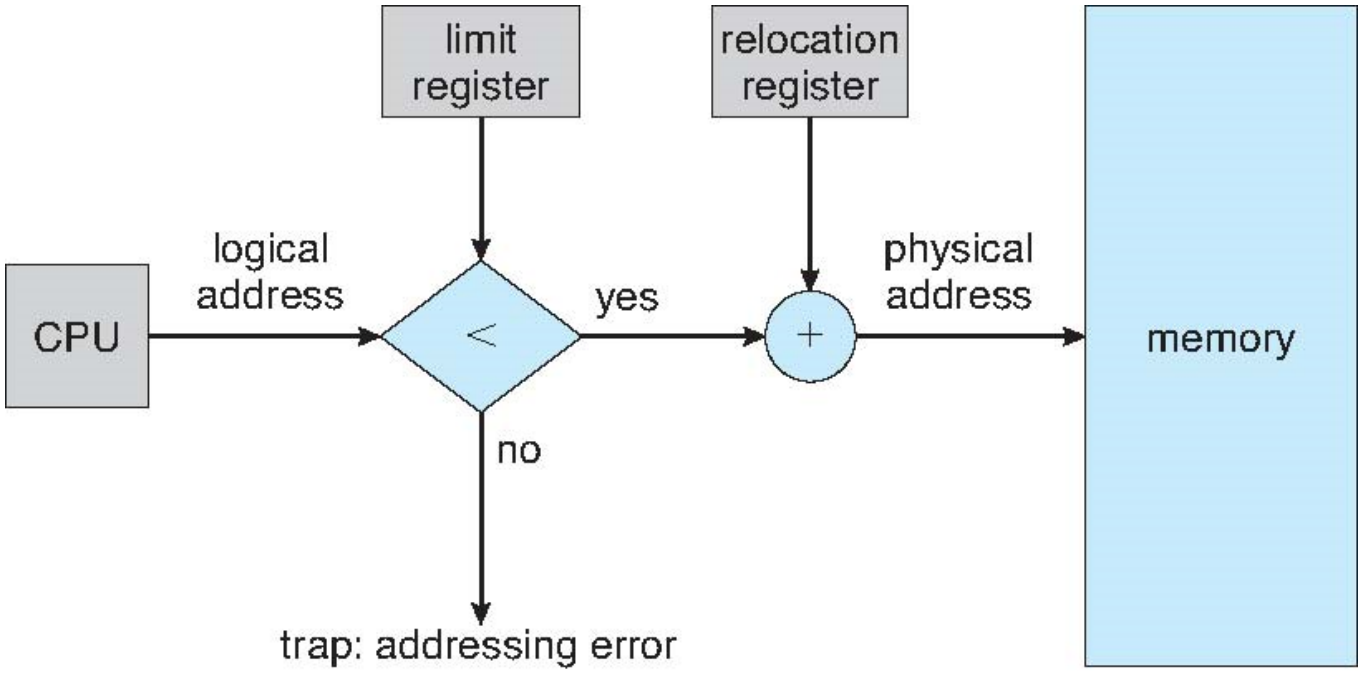
Swapping on Mobile Systems

- Not typically supported
  - Flash memory based
    - Small amount of space
    - Limited number of write cycles
    - Poor throughput between flash memory and CPU on mobile platform
- Instead use other methods to free memory if 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 later

Contiguous Memory Allocation

- Main memory must support both OS and user processes
  - Resident operating system, usually held in low memory with interrupt vector → User processes then held in high memory
  - Each process contained in single contiguous section of memory
  - Limited resource, must allocate efficiently
- **Contiguous memory allocation** is one early method
  - A process is assigned consecutive memory blocks, that is, memory blocks (bytes, words, etc.) having consecutive addresses
  - Fixed partitioning, dynamic partitioning
- Implementation of contiguous memory allocation
  - Two registers used to protect user processes from each other, and from changing OS code and data
  - **Base (or relocation) 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*
    - Physical Address = Base register address + Logical address

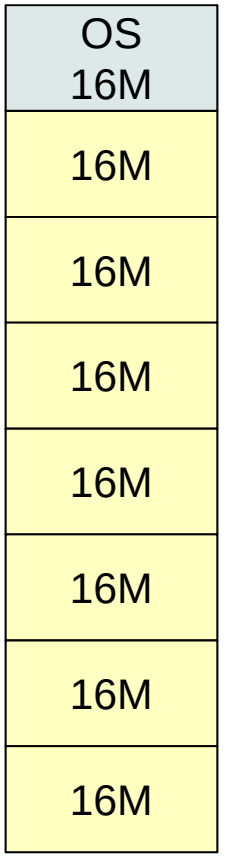
Hardware Support for Relocation and Limit Registers



relocation register = base register

Fixed Partitioning

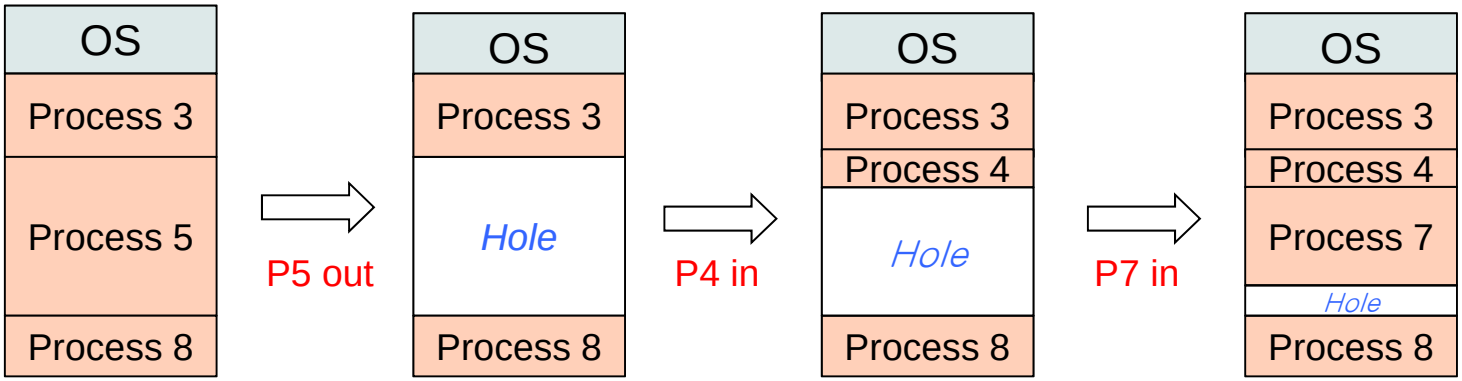
- Equal-size partitions
  - Any process whose size is less than or equal to the partition size can be loaded into an available partition
- OS can swap out a process if all partitions are full and the process is not in the *ready* or *running* state
- Disadvantages
  - A program may be too big to fit in a partition → Program needs to be designed with the use of overlays
  - Main memory utilization is inefficient
    - Any program, regardless of size, occupies an entire partition → may lead to internal fragmentation
    - *Internal fragmentation* : wasted space due to the block of data loaded being smaller than the partition
    - Partitions may be designed to be of unequal size



Example of fixed partitioning of a 128-MByte memory

Dynamic Partitioning

- Partitions of variable size
  - Partitions are of variable length and number
  - Process is allocated exactly as much memory as it requires
- Dynamic storage allocation
  - Degree of multiprogramming limited by number of partitions
  - **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 Partitioning - Placement

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

- **First-fit**: Allocate the ***first*** hole that is big enough; scan from the beginning; generally superior
  - **Next-fit**: Variation. Allocate the ***next available*** hole that is big enough; scan form the location of the last placement

- **Best-fit**: Allocate the ***smallest*** hole that is big enough; must search entire list, unless ordered by size → time-consuming
  - Produces the smallest leftover hole
  - But create many small holes that may not be recycled easily

- **Worst-fit**: Allocate the ***largest*** hole; must also search entire list
  - Produces the largest leftover hole

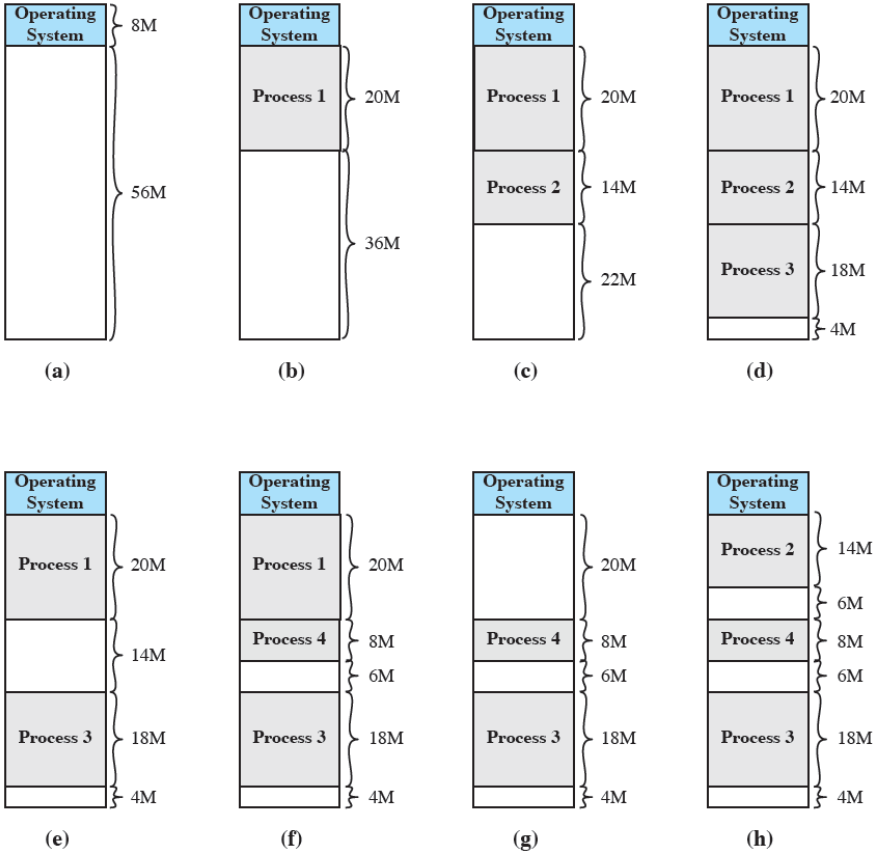
→ First-fit and best-fit better than worst-fit in terms of speed and storage utilization

Dynamic Partitioning - Fragmentation

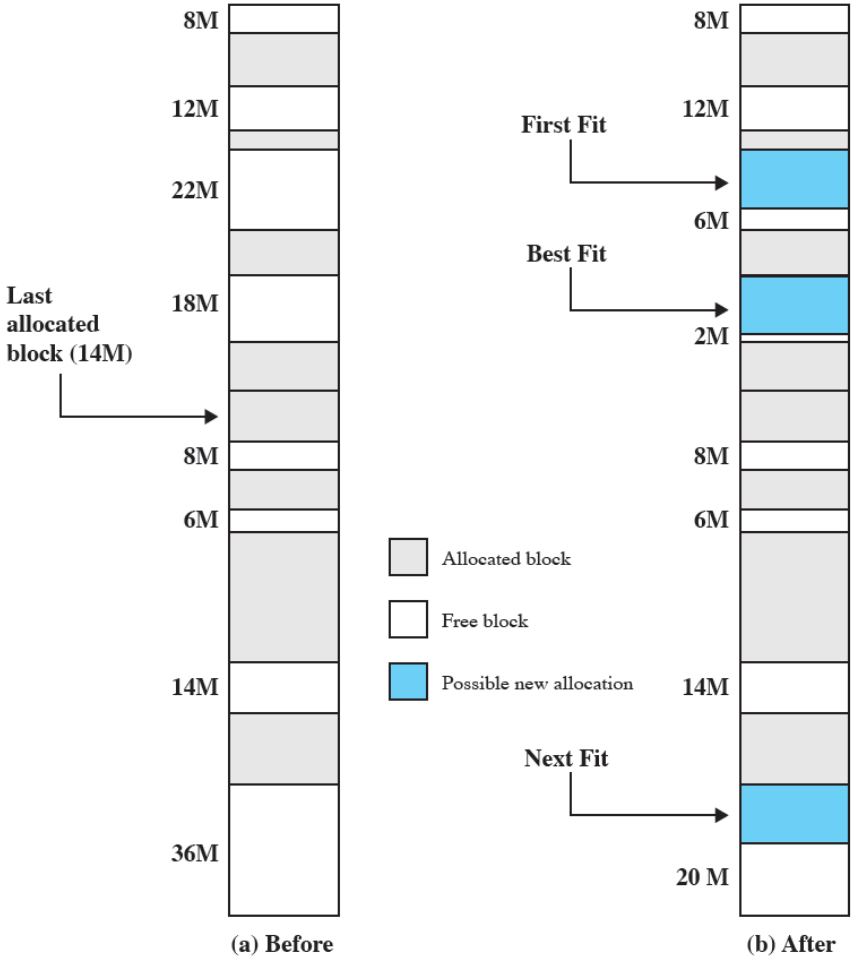
- **External Fragmentation** – total (free) memory space exists to satisfy a request, but it is not contiguous
  - Wasted space external to allocated partitions, visible to system
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory
  - Wasted space internal to an allocated partition, visible to process
- First fit analysis reveals that given N blocks allocated, another 0.5 N blocks lost to fragmentation
  - 1/3 (← 0.5/(1.0+0.5)) may be unusable -> **50-percent rule**
- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block (coalescing)
  - 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

Backing store (or disk) has the same fragmentation problems

Dynamic Allocation - Illustrated



Effect of dynamic storage allocation (external fragmentation)



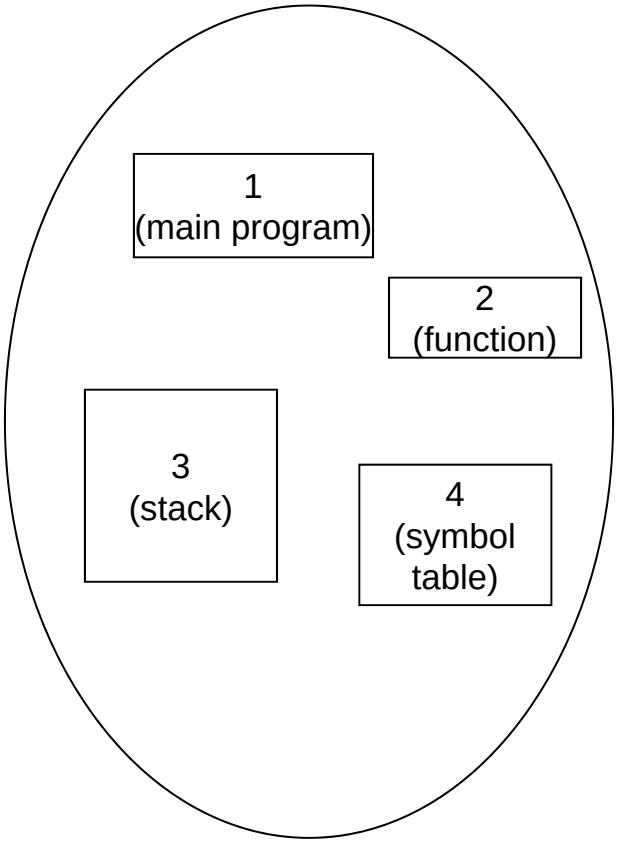
Example memory configuration before and after allocation of *16-Mbyte block*

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

Logical View of Segmentation



user space **physical memory space**

In parentheses are example segments that a programmer sees.

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

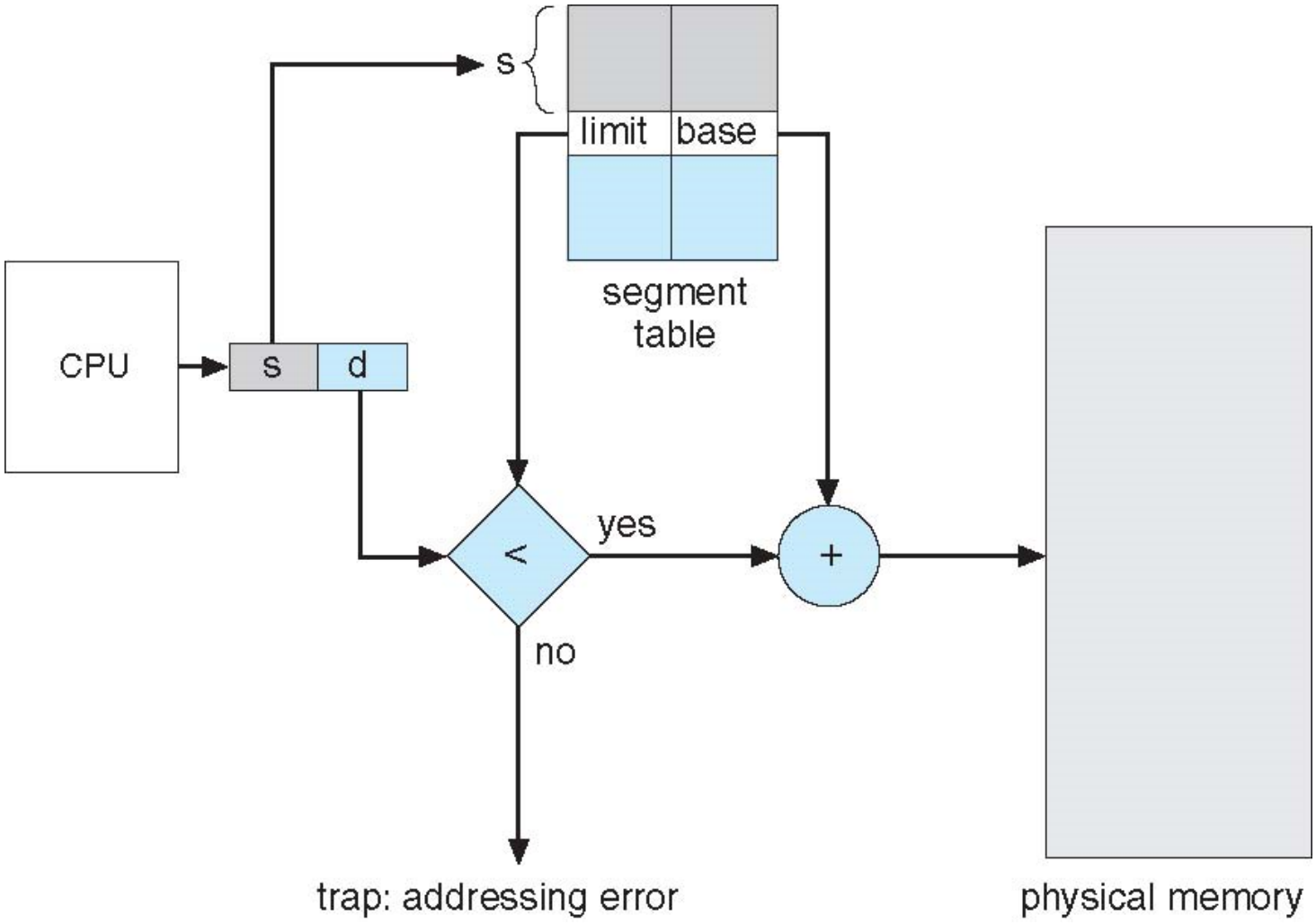
- **Segment-table length register (STLR)** indicates number of segments used by a program;

  segment number ***s*** is legal if ***s*** < **STLR**

Segmentation Architecture (Cont.)

- Protection
  - With each entry in segment table associate:
    - validation bit = 0 ⇒ invalid segment
    - read/write/execute privileges
    - protection bit
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram

Segmentation Hardware



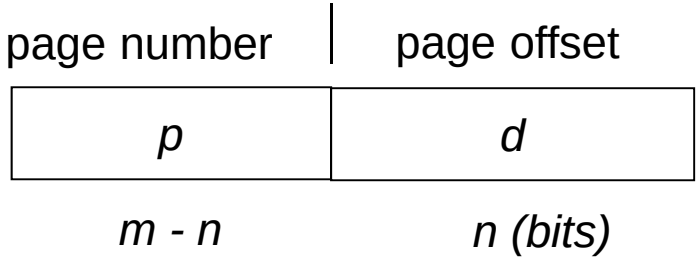
Paging

- Physical address space of a process can be noncontiguous as in the case of segmentation
- But unlike segmentation, paging
  - 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

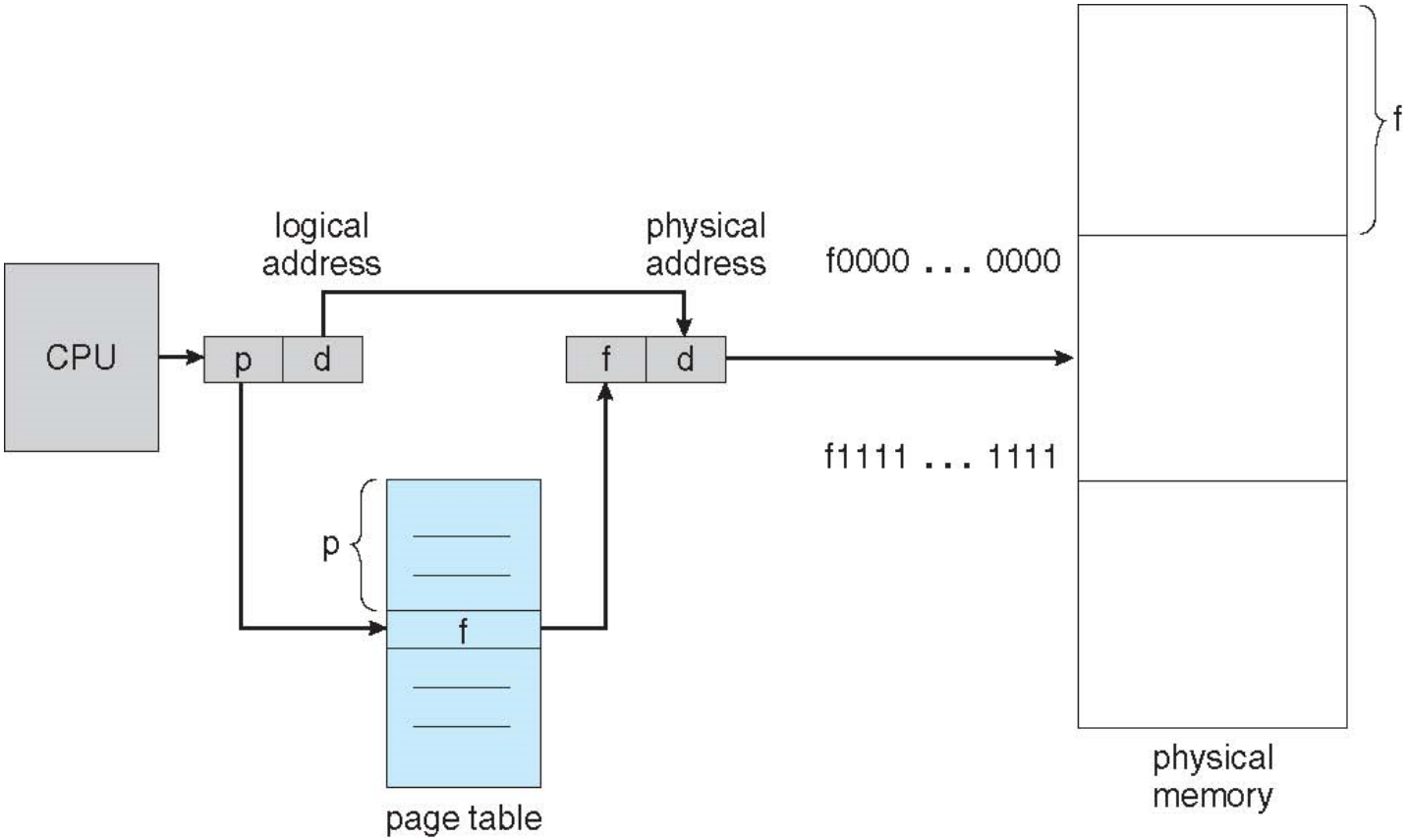
Address Translation Scheme

- 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

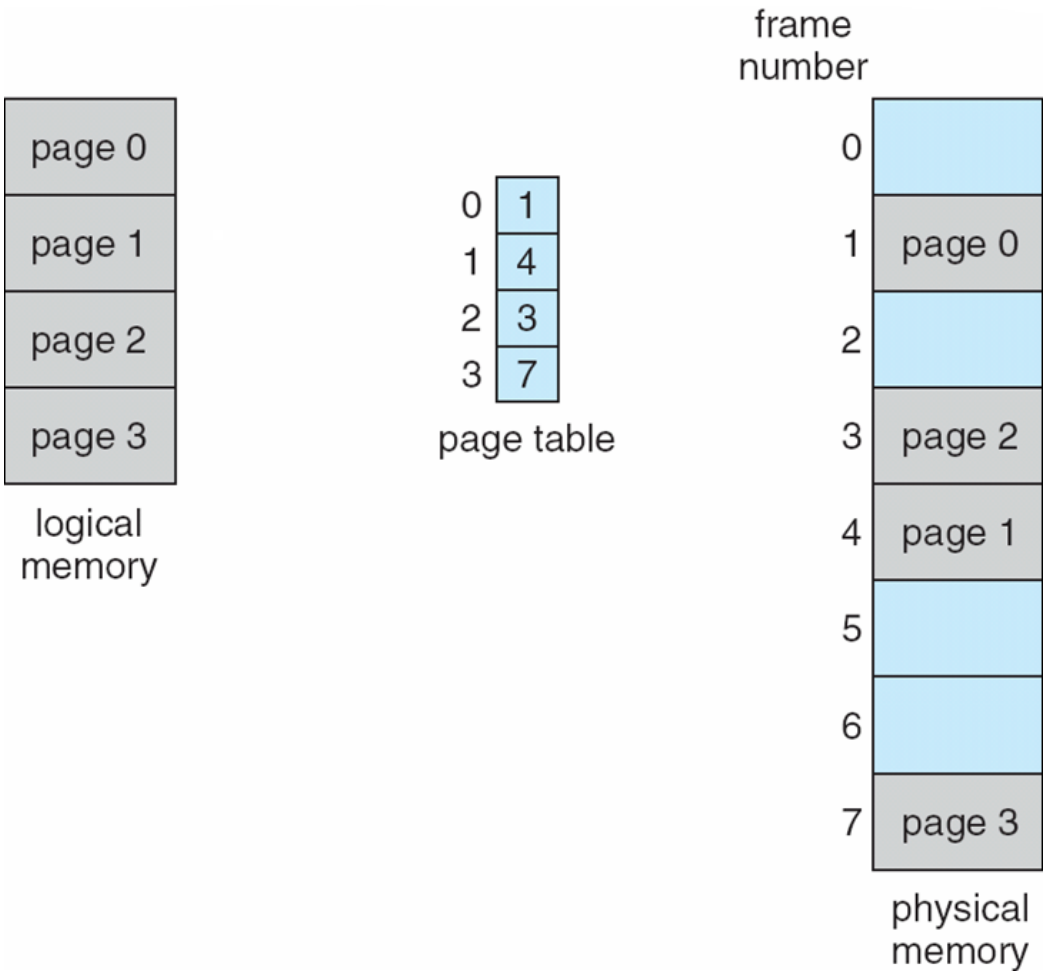
    - For given logical address space 2^m and page size 2^n



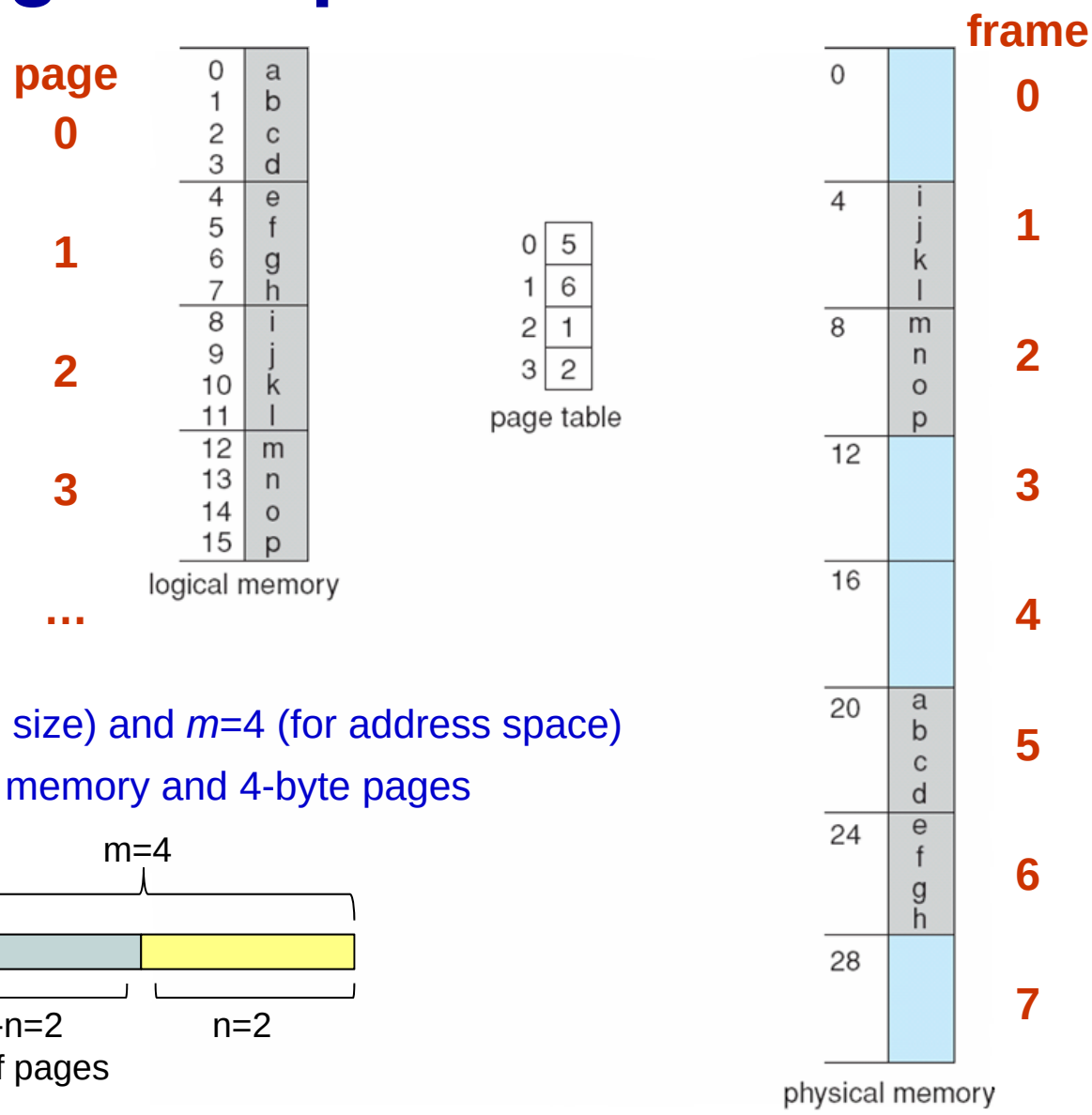
Paging Hardware



Paging Model of Logical and Physical Memory



Paging Example



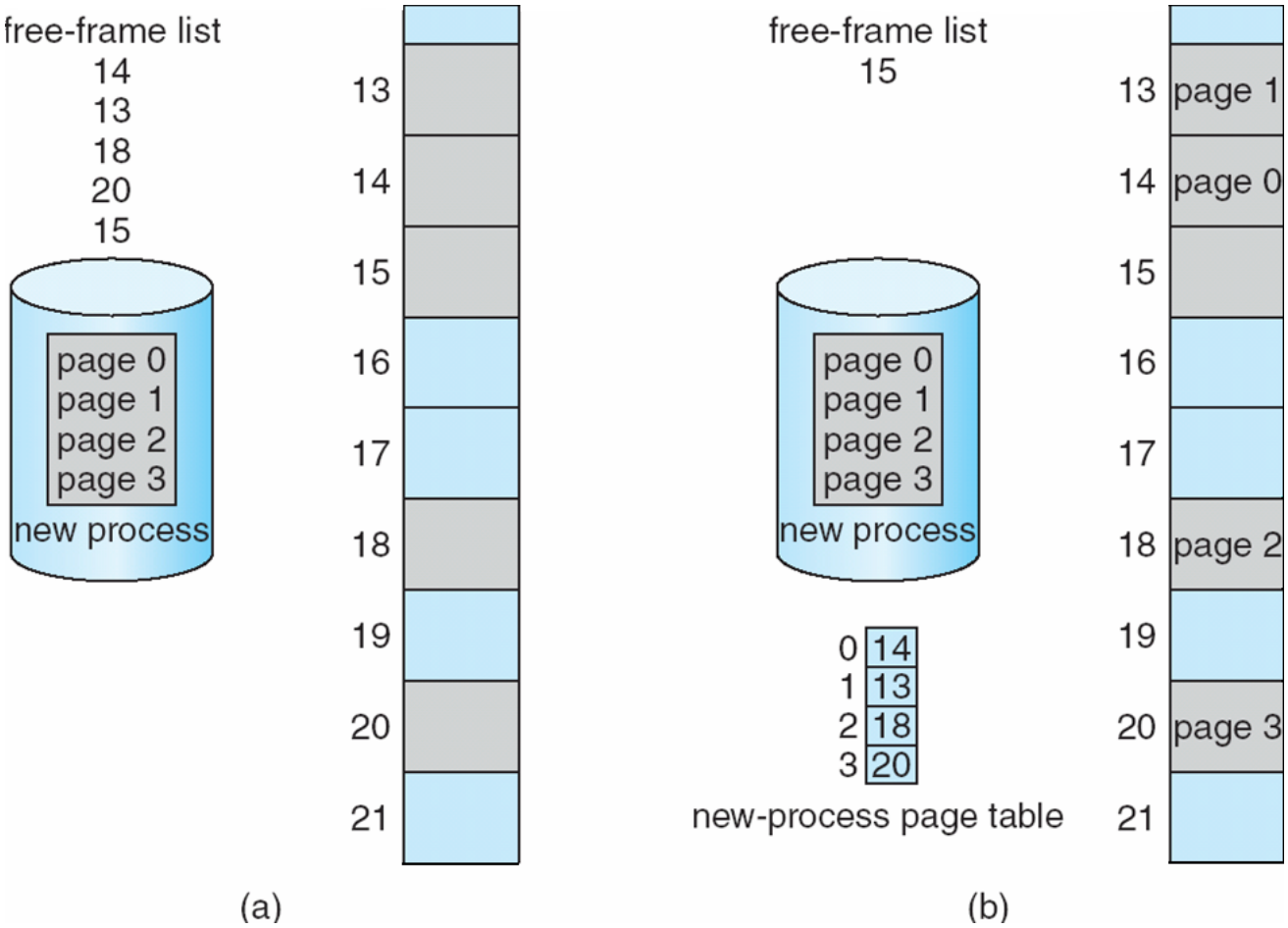
n=2 (for page size) and *m*=4 (for address space)

i.e., 32-byte memory and 4-byte pages

Paging (Cont.)

- Calculating internal fragmentation
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - 35 pages + 1,086 bytes
  - Internal fragmentation of 2,048 - 1,086 = 962 bytes
  - Worst case fragmentation = 1 frame – 1 byte
  - On average fragmentation = 1 / 2 frame size
  - So small frame sizes desirable?
  - But each page table entry takes memory to track
  - Page sizes growing over time
    - Solaris supports two page sizes – 8 KB and 4 MB
- Process view and physical memory now very different
- By implementation process can only access its own memory

Free Frames



Before allocation

After allocation

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**)
  - TLBs typically small (64 to 1,024 entries)
- 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
- 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

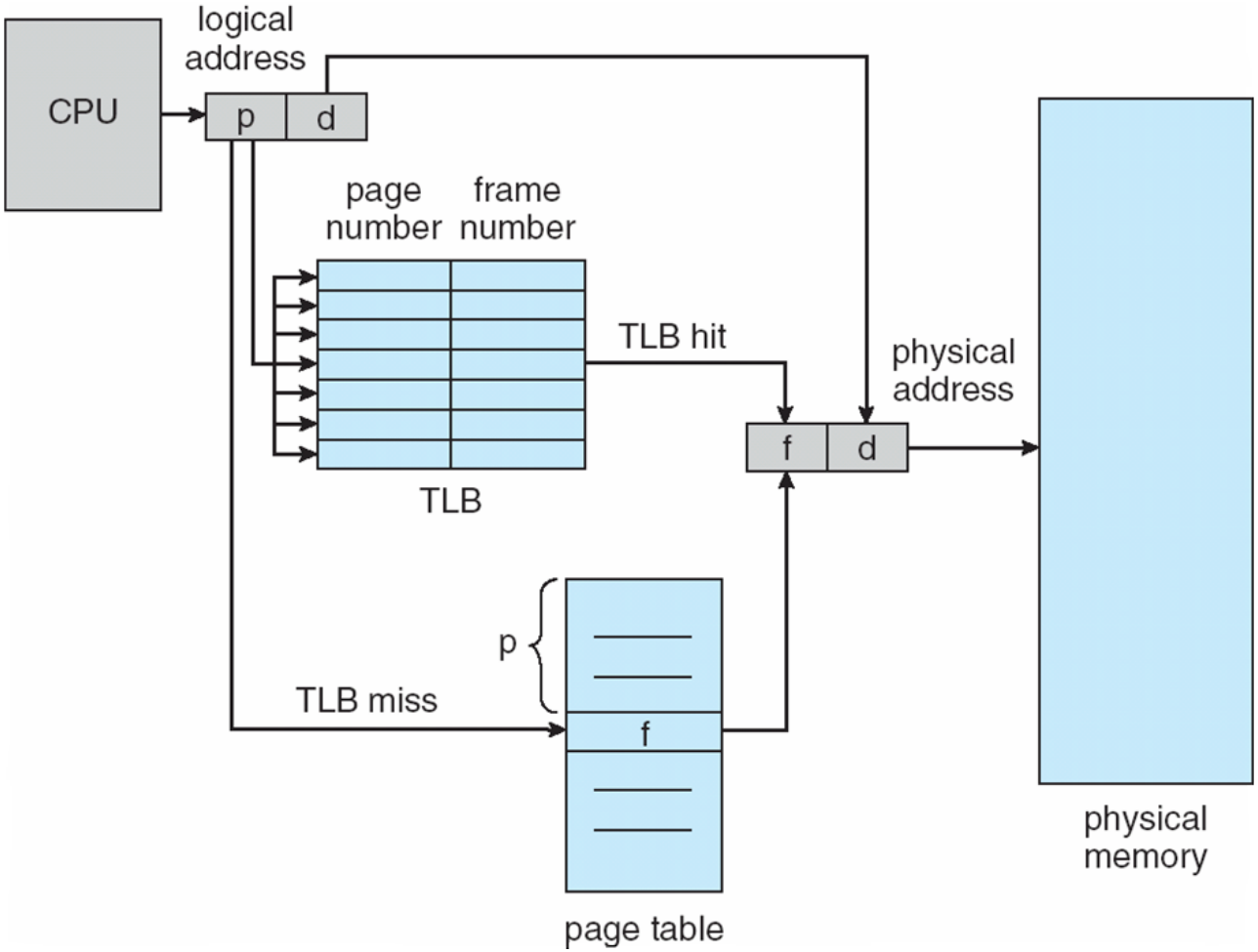
Associative Memory

- Associative memory – parallel search
  - Given the value or content, the associative memory searches for the given value and outputs the location or address at which the value is stored (if found).
  - The search operation is performed simultaneously across the given memory space.

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



Effective Access Time

- Associative Lookup = ε time unit
  - Can be < 10% of memory access time (say, τ)
- Hit ratio = α
  - Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- **Effective Access Time** (**EAT**)

  $EAT = (\tau + \varepsilon)\,\alpha + (2\tau + \varepsilon)(1 - \alpha)$
- Consider α = 80%, ε = 20ns for TLB search, τ = 100ns for memory access
  - EAT = 0.80 x (100+20) + 0.20 x (200+20) = 140ns
- Consider more realistic hit ratio -> α = 99%, ε = 20ns for TLB search, τ = 100ns for memory access
  - EAT = 0.99 x 120 + 0.01 x 220 = 121ns

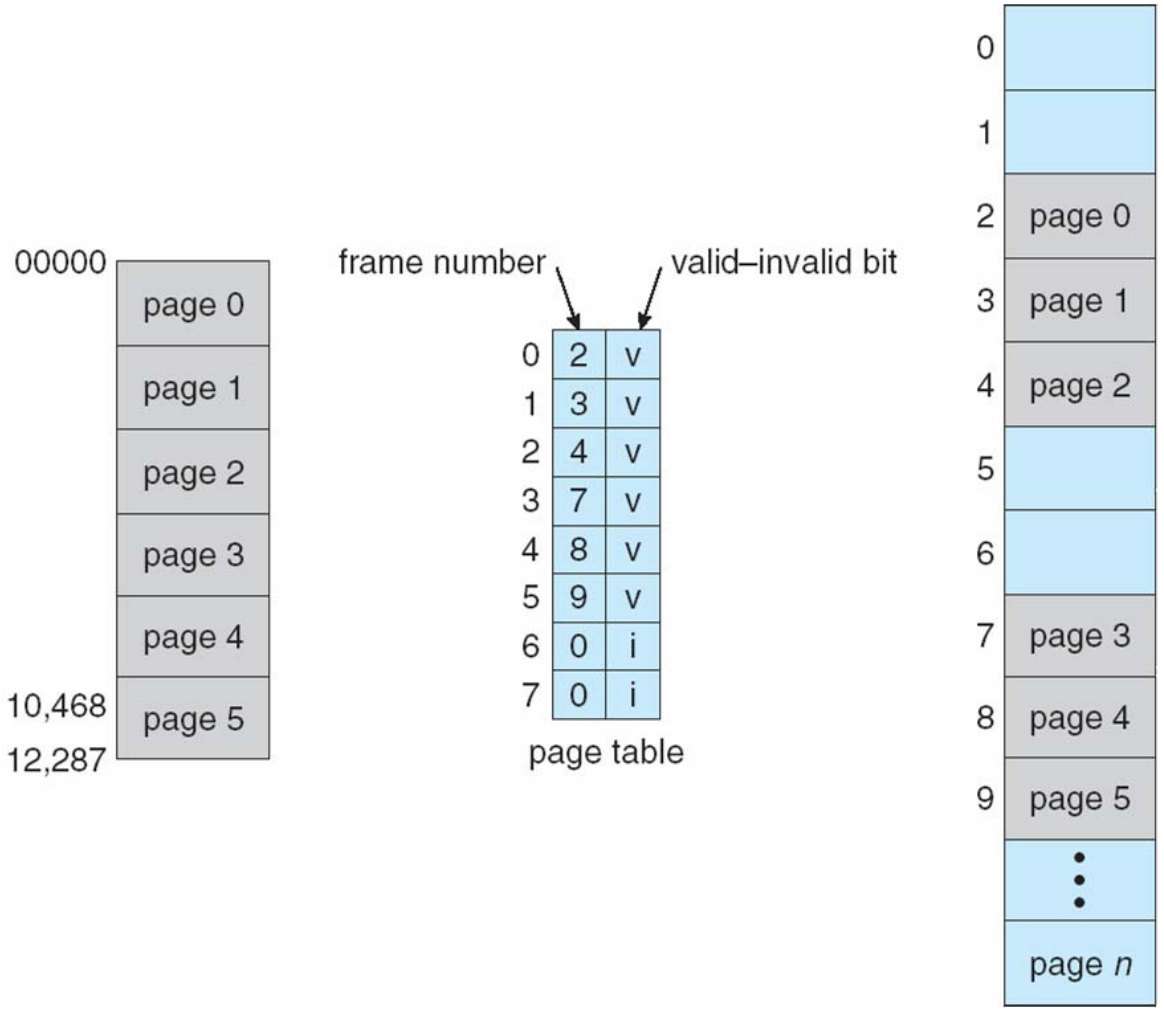
Memory Protection

- 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

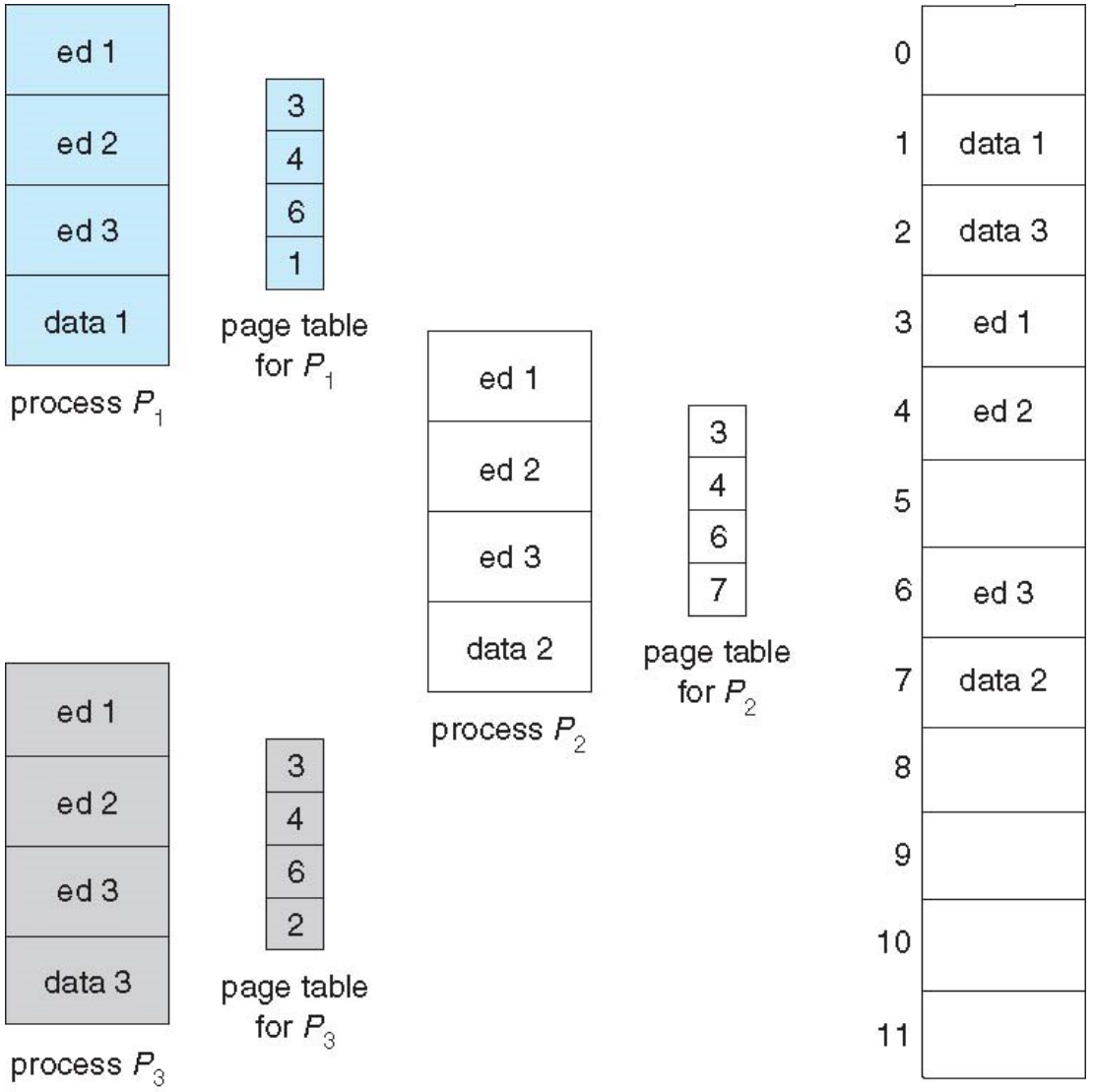


Shared Pages

- **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

Shared Pages Example



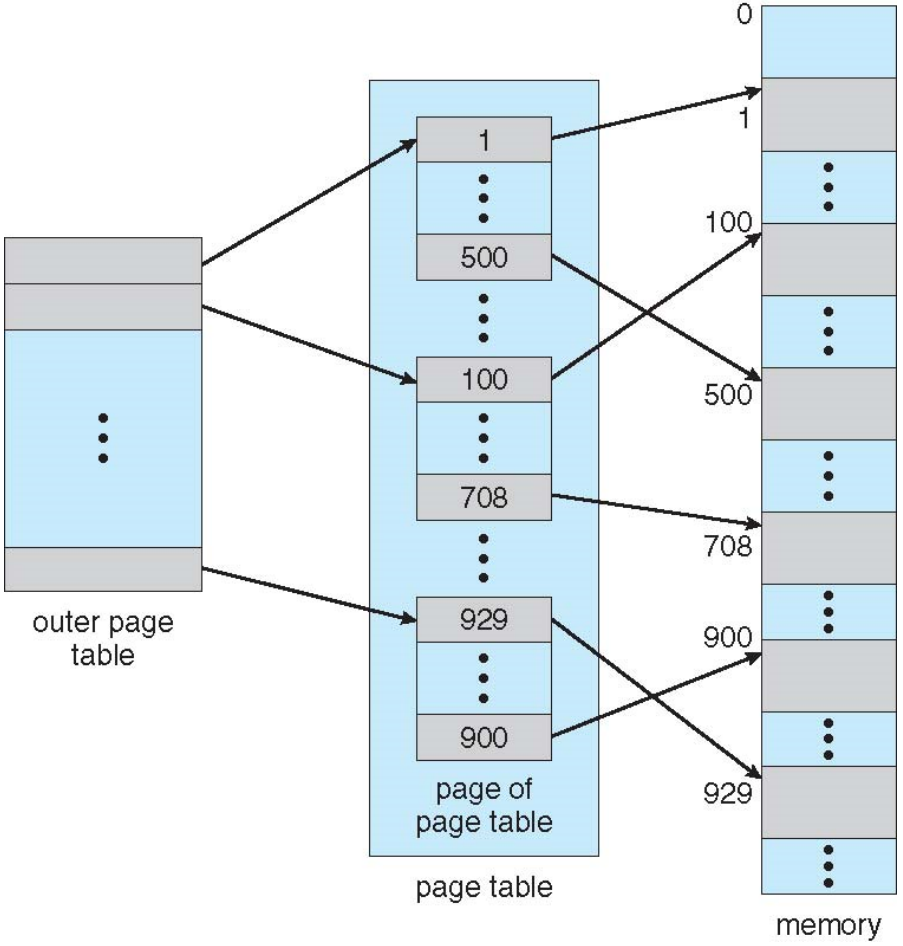
Structure of the Page Table

- 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 entries (2^{32} / 2^{12})
  - If each entry is 4 bytes -> 4 MB 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
- Approaches
  - Hierarchical Paging
  - Hashed Page Tables
  - Inverted Page Tables

Hierarchical Page Tables

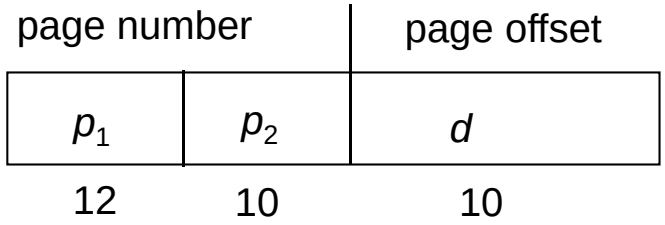
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table

Two-level page-table scheme



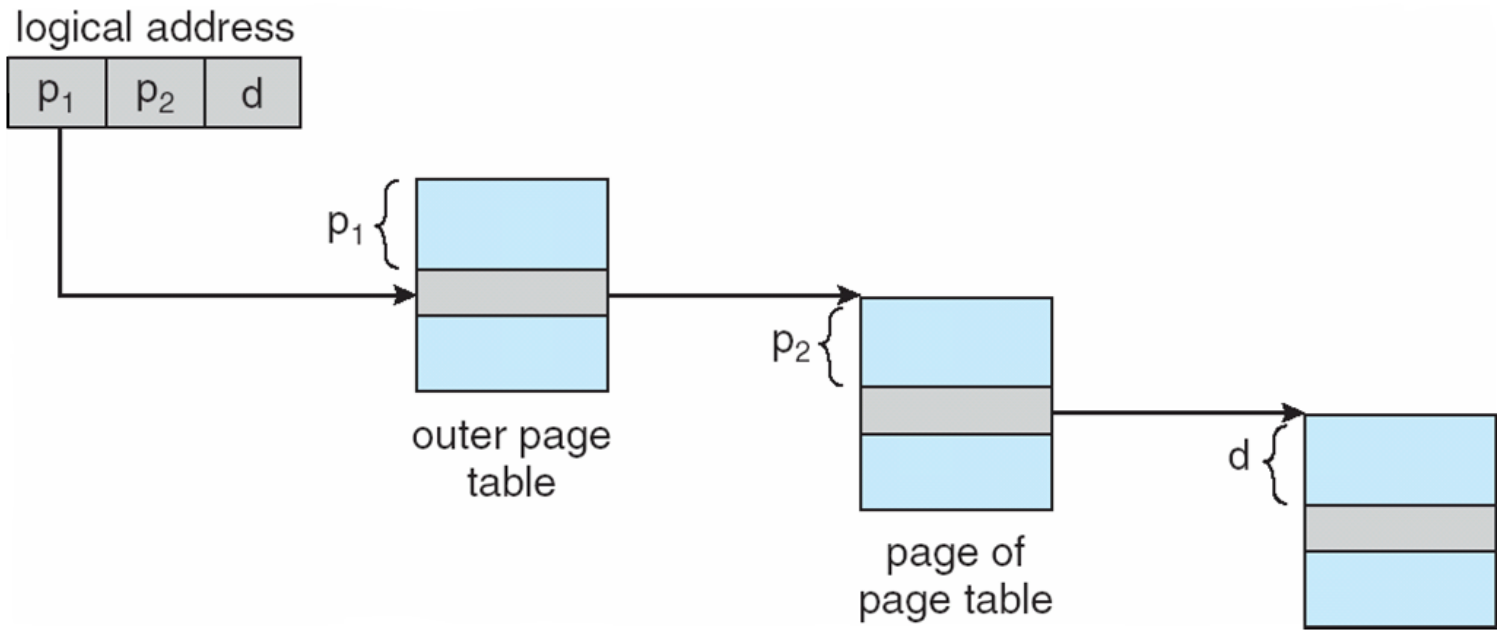
Two-Level Paging Example

- 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:



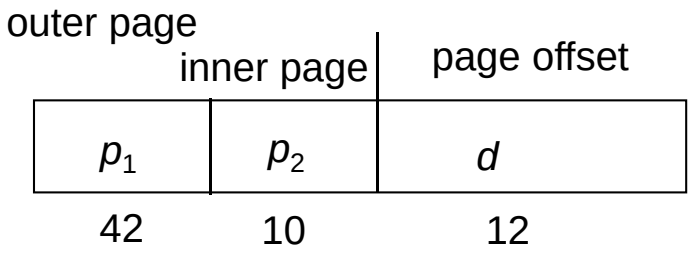
  - where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**

Address-Translation Scheme



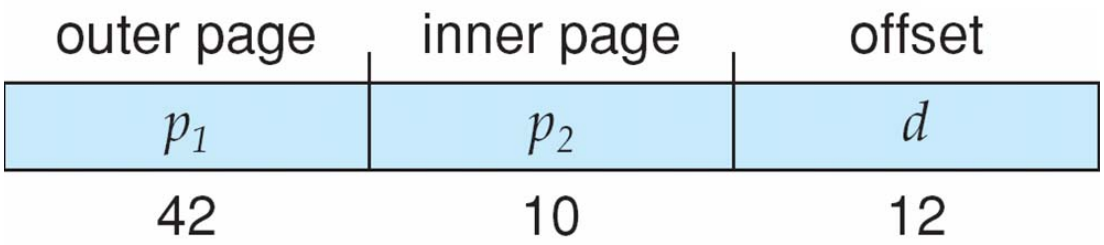
64-bit Logical Address Space

- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look like

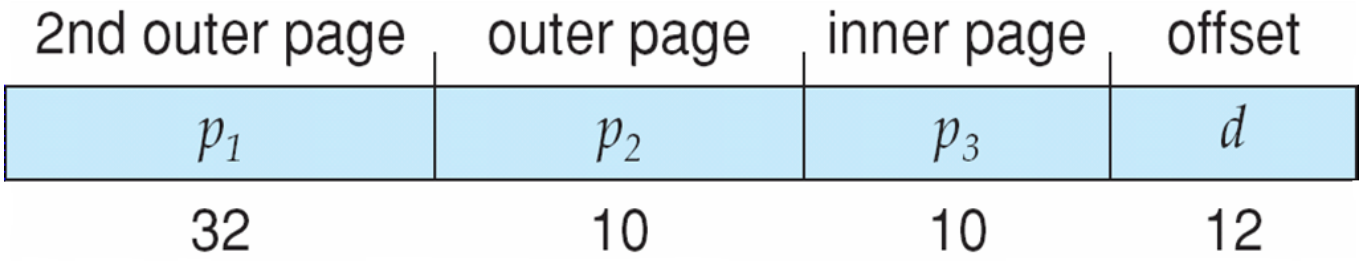


  - Outer page table has 2^{42} entries or 2^{44} bytes
  - One solution is to add a 2^{nd} outer page table
  - But in the following example the 2^{nd} outer page table is still 2^{34} bytes in size
    - And possibly 4 memory access to get to one physical memory location

Multi-level Paging Schemes



Two-level

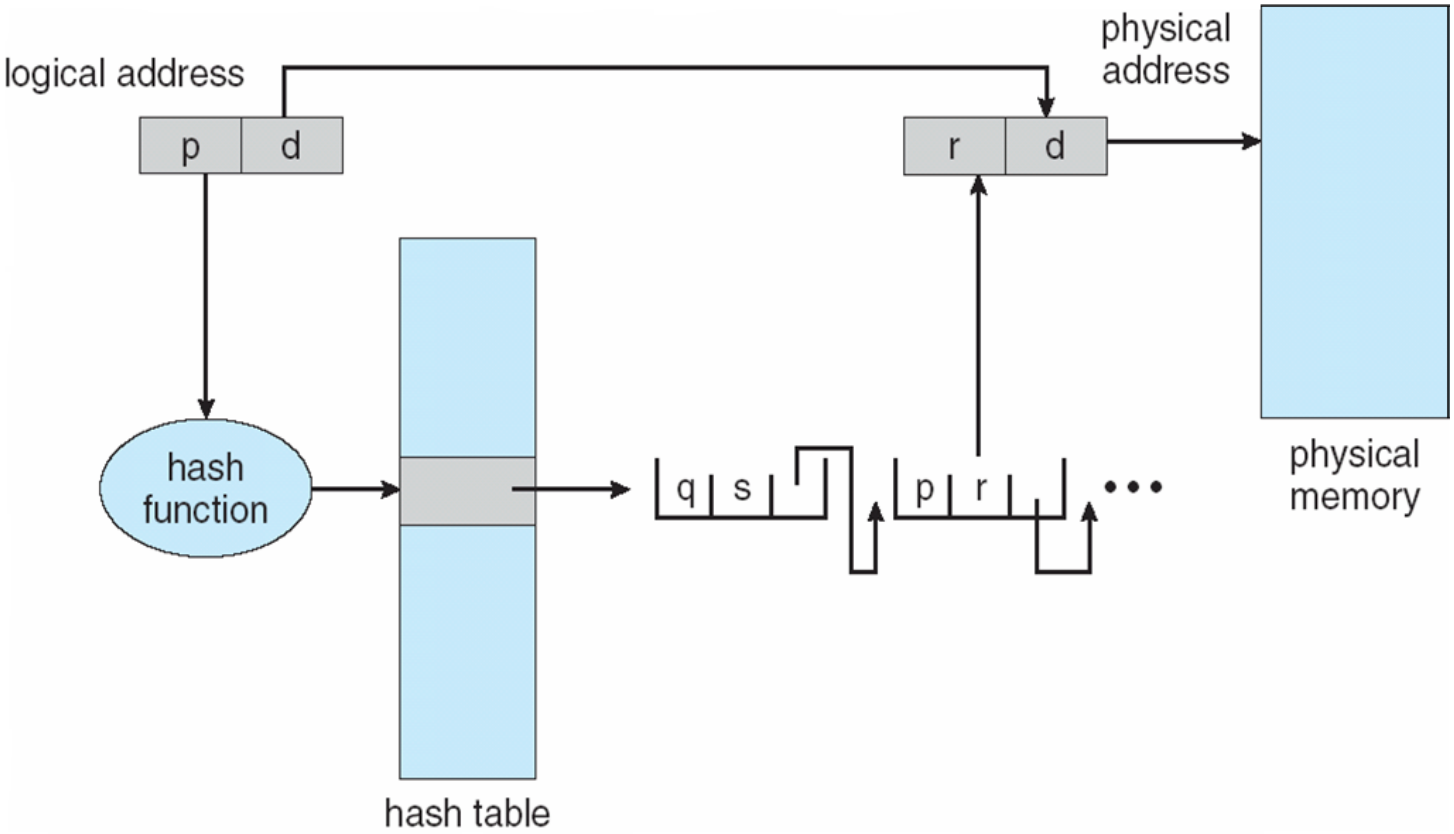


Three-level

Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
  - 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
- Variation for 64-bit addresses is **clustered page tables**
  - Similar to hashed but each entry refers to several pages (such as 16) rather than 1
  - Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)

Hashed Page Table

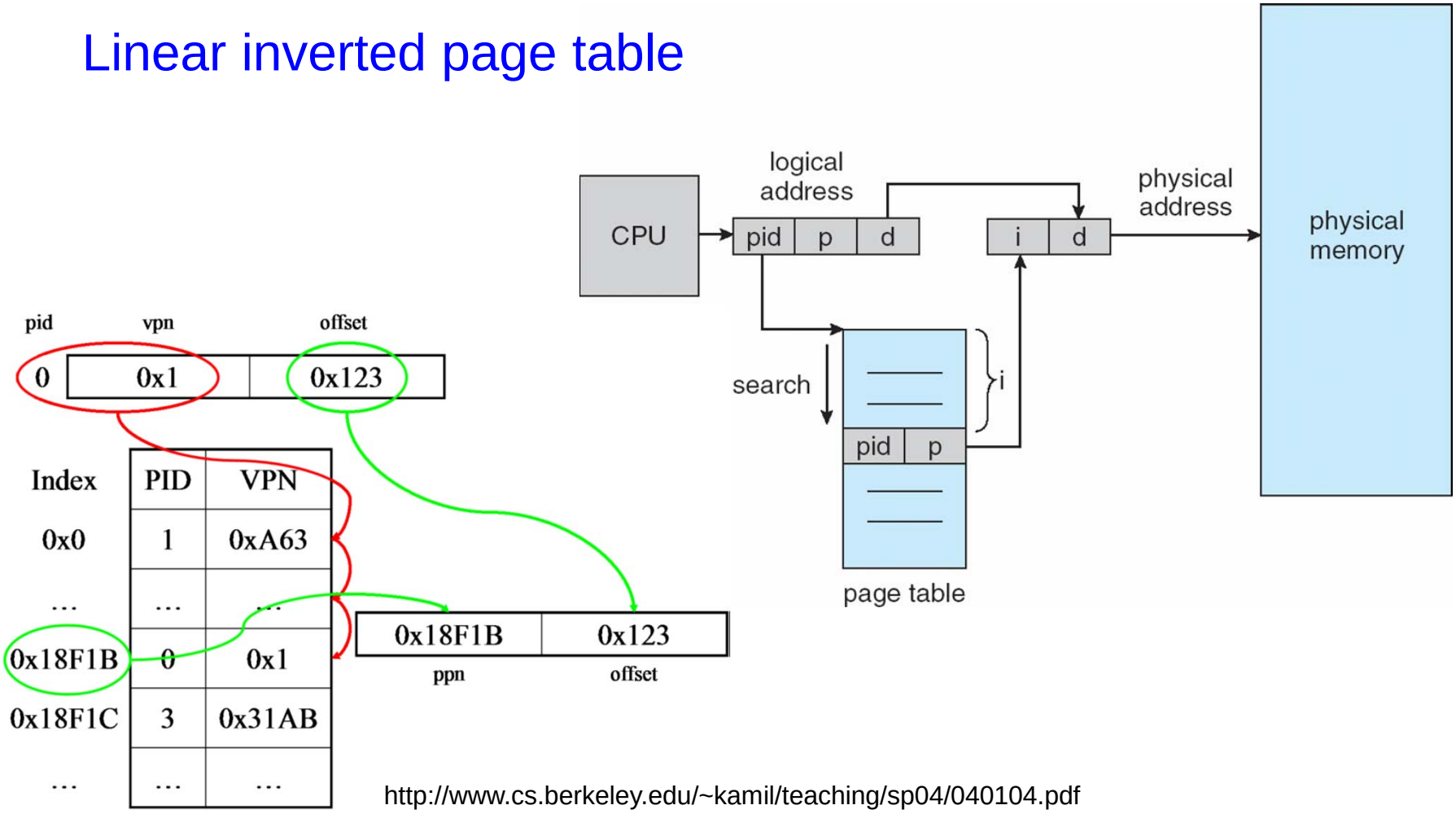


Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- May use hash table to limit the search to one — or at most a few — page-table entries
  - TLB can accelerate access
- But how to implement shared memory?
  - Simple approach: Allow the page table to contain only one mapping of a virtual address to the shared physical address

Inverted Page Table Architecture

Linear inverted page table

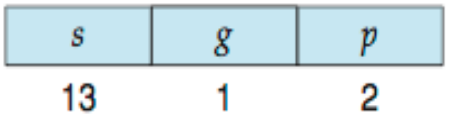


http://www.cs.berkeley.edu/~kamil/teaching/sp04/040104.pdf

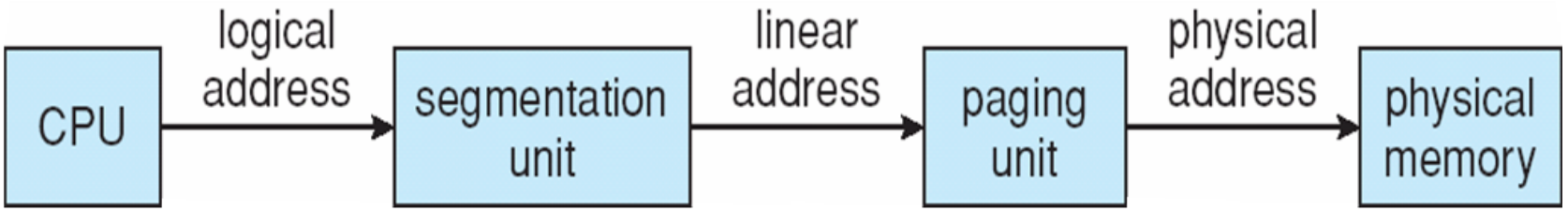
Translation procedure

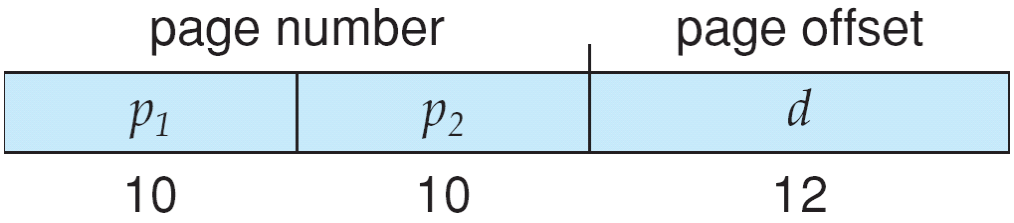
Example: The Intel 32-Bit Architecture

- 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
  - Selector given to segmentation unit
    - Which produces linear addresses
  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU
    - Page sizes can be 4 KB or 4 MB

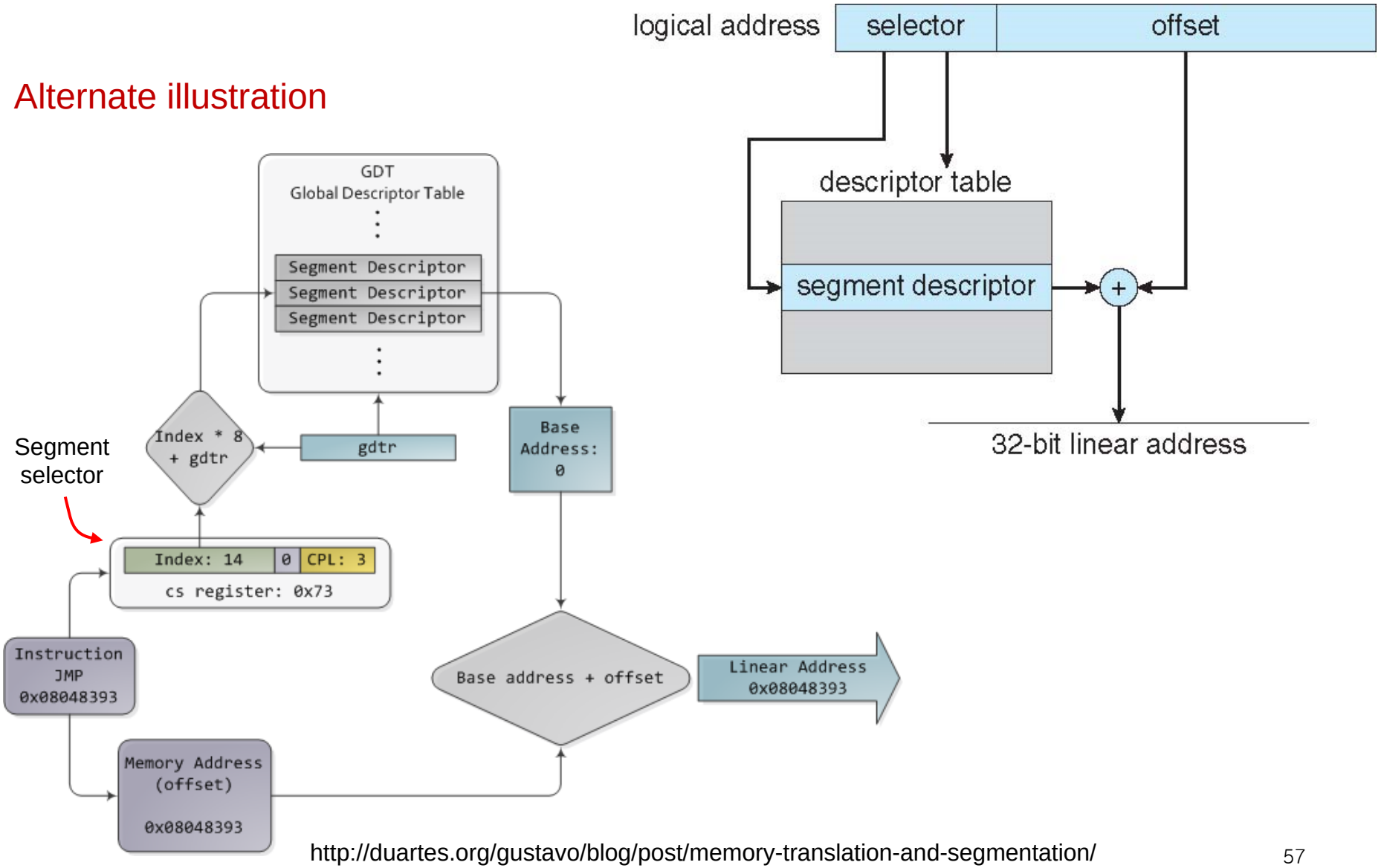


Logical to Physical Address Translation in IA-32



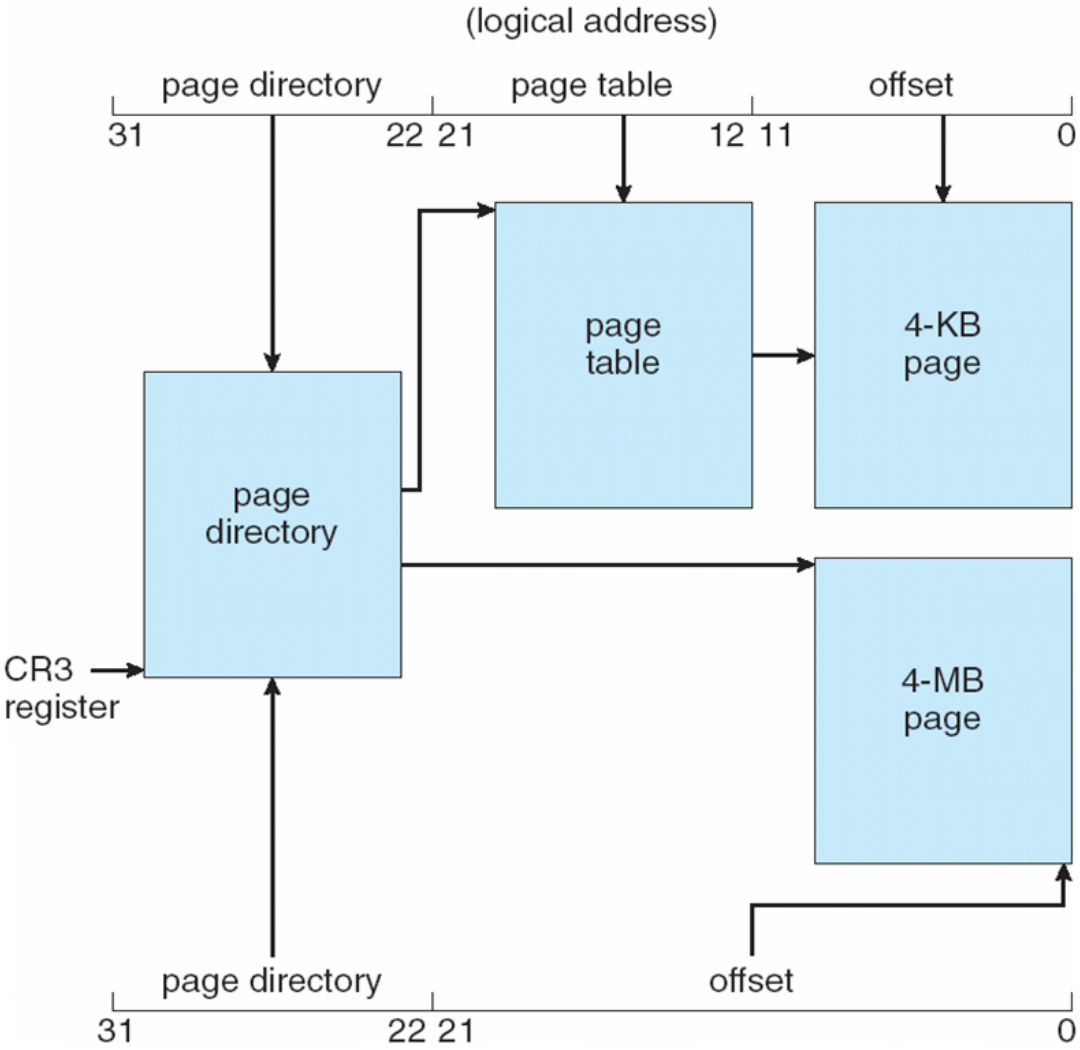


Intel IA-32 Segmentation



http://duartes.org/gustavo/blog/post/memory-translation-and-segmentation/

Intel IA-32 Paging Architecture

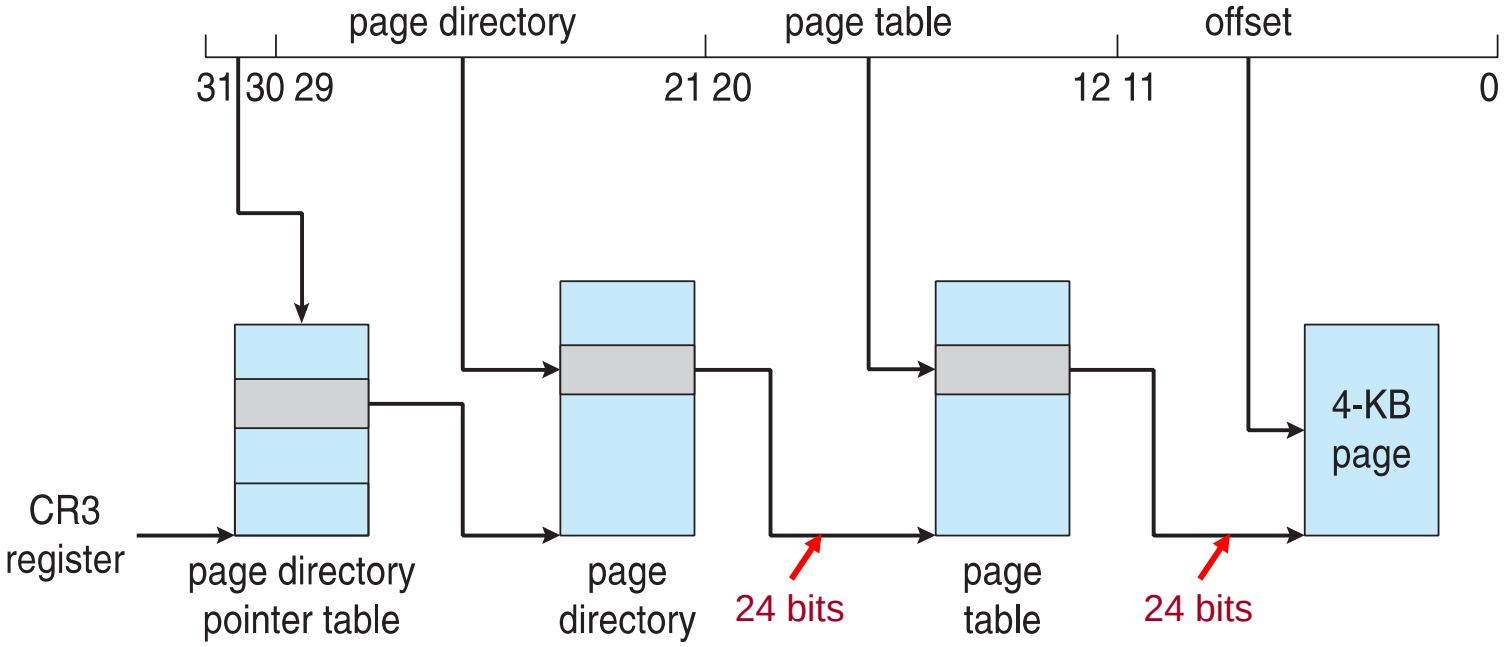


4 KB page

4 MB page

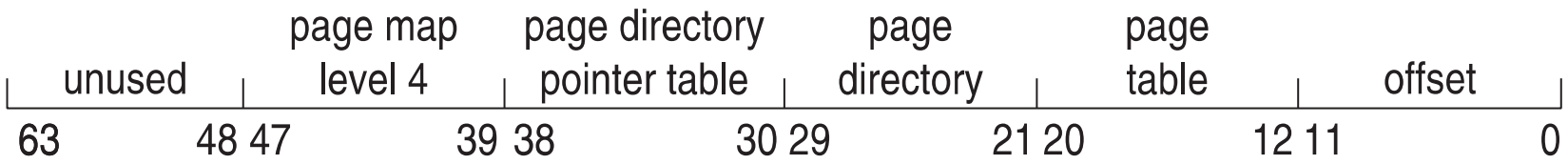
Intel IA-32 Page Address Extensions

- 32-bit address limits led Intel to create **page address extension** (**PAE**), allowing 32-bit apps access to more than 4GB of memory space
  - Paging went to a 3-level scheme
  - Top two bits refer to a **page directory pointer table**
  - Page-directory and page-table entries moved to 64-bits in size
    - the base address of page tables and page frames: 20 → 24 bits
  - Net effect is increasing address space to 36 bits – 64GB of physical memory



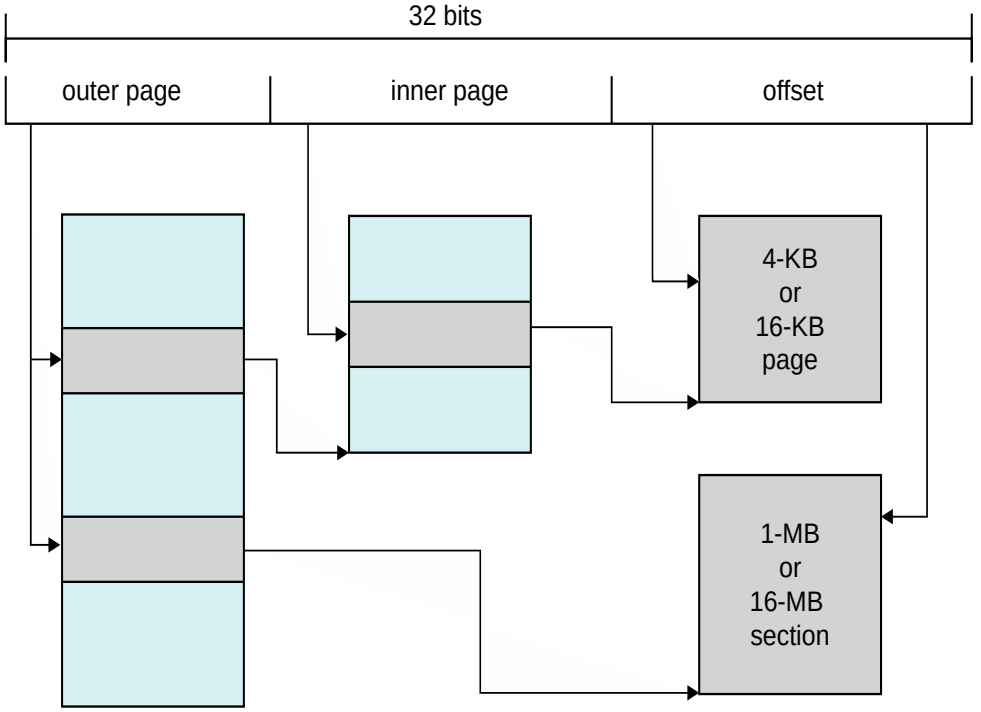
Intel x86 64-Bit Case

- Current generation Intel x86 architecture
- 64 bits is ginormous (> 16 exabytes)
- In practice only implement 48 bit addressing
  - Page sizes of 4 KB, 2 MB, 1 GB
  - Four levels of paging hierarchy
- Can also use PAE so virtual addresses are 48 bits and physical addresses are 52 bits



Example: ARM Architecture

- Dominant mobile platform chip (Apple iOS and Google Android devices for example)
- Modern, energy efficient, 32-bit CPU
- 4 KB and 16 KB pages
- 1 MB and 16 MB pages (termed **sections**)
- One-level paging for sections, two-level for smaller pages
- Two levels of TLBs
  - Outer level has two micro TLBs (one data, one instruction)
  - Inner is single main TLB
  - First inner is checked, on miss outers are checked, and on miss page table walk performed by CPU



Summary

- 폰노이만 구조
  - 반드시 코드와 데이터가 메모리에 위치해야 함
- 데이터의 위치, 주소
  - 프로세스가 보는 위치: 논리적 주소공간에서 정의됨
  - 실제 메모리에서의 위치: 물리적 주소공간에서 정의
  - base register, limit register
- 데이터 위치의 확정시점
  - 컴파일할 때
  - 링킹(linking)할 때
  - 적재(loading)할 때: 동적
  - 실행할 때: 동적
- 바인딩
  - 어떤 객체의 속성(이름, 주소, 등)을 확정짓는 것
  - 주소 바인딩(address binding)
- 재배치(relocation) 문제
- 오버레이 – 동적인 적재
  - 실행시, 필요한 모듈을 디스크에서 불러옴
  - 프로그래머가 일차적인 책임
- 동적인 링킹
  - 적재시 혹은 실행시 주소를 바인딩
- 교체(swapping)
  - 파일을 메모리에서 방출, 메모리에 적재: 디스크 이용
  - 성능문제

- von Neumann architecture
  - must have code/data in main memory
- Location and address of data
  - location seen by a process: defined in the logical address space
  - location seen by the memory: defined in the physical address space
  - base register, limit register
- When to bind the location of data
  - compile time
  - linking time
  - loading time: dynamic
  - execution time:; dynamic
- Binding
  - assigning a value to an attribute of object
  - address binding
- Relocation
- Overlays - Dynamic loading
  - at runtime, load a module needed next from disk
  - programmers assume primary responsibility
- Dynamic linking
  - address binding at load time or runtime
- Swapping
  - swap out data in the memory, and swap in data in the disk
  - performance problem

Summary (Cont.)

- 연속된 주소에 메모리 할당
  - contiguous allocation: 모두 같은 블록에 할당
  - 고정된 크기의 파티션 설정
  - 동적인 파티션 설정 – 가변
  - 동적인 블록 할당문제
  - 분열(fragmentation) 문제
- 세그먼트화(segmentation)
  - 프로그램의 각 세그먼트를 별도의 구역에 배치
  - 세그먼트 시작주소, 길이
- 페이징(paging)
  - 일정 크기의 페이지 단위로 데이터를 이동/배치/접근
  - 페이지 번호, 페이지 내의 오프셋
  - 주소변환(address translation): paging hardware
  - 페이지 테이블 (PT)의 구현
    - TLB: 페이지 테이블의 캐쉬 버전
    - 데이터에의 접근 시간 분석
    - 메모리 보호, 페이지의 공유
- 페이지 테이블의 구조
  - PT의 대형화 가능성 대비
  - 계층화(hierarchical PT): 2-level/3-level PT
  - 해쉉 (hashed PT)
  - 프레임 기반 (inverted PT)
- 사례
  - Intel IA-32, ARM

- Contiguous memory allocation
  - contiguous allocation: allocate all code/data in the same memory block
  - fixed partitioning
  - dynamic partitioning
  - dynamic storage allocation problem
  - fragmentation problem
- Segmentation
  - The segments of a program are placed separately
  - segment address, segment length
- Paging
  - Data are transferred, deployed or accessed by the page
  - page number, offset within a page
  - address translation: paging hardware
- Implementation of page table (PT)
  - TLB: a cache version of page table
  - analysis of data access time
  - memory protection, shared pages
- The structure of page table
  - to cope with a huge PT
  - hierarchical PT: 2-level/3-level PT
  - hashed PT
  - inverted PT
- Example
  - Intel IA-32, ARM

Summary (Cont.) Memory Management Schemes

	Technique	Description	Strengths	Weaknesses
Partitioning	Fixed partitioning	Main memory is divided into static partitions at system generation time. A process may be loaded into a partition of equal or greater size.	Simple to implement; little operating system overhead.	Inefficient use of memory due to internal fragmentation; max. number of active processes is fixed.
	Dynamic partitioning	Partitions are created dynamically, where a partition is of exactly the same size as the process needs.	No internal fragmentation; more efficient use of main memory.	Inefficient use of processor due to compaction to counter external fragmentation.
Paging	Simple paging	Main memory is divided into equal-size frames, and each process into equal-size pages of the same length as frames. A process is loaded by loading all pages into available frames.	No external fragmentation.	A small amount of internal fragmentation.
	VM paging	As with simple paging, except that it is not necessary to load all of the pages of a process. Nonresident pages that are needed are brought in later automatically.	No external fragmentation; higher degree of multiprogramming; large virtual address space.	Overhead of complex memory management.
Segmentation	Simple segmentation	Each process is divided into a number of segments. A process is loaded by loading all of its segments into dynamic partitions.	No internal fragmentation; improved memory utilization.	External fragmentation
	VM segmentation	As with simple segmentation, except that it is not necessary to load all of the segments of a process. Nonresident segments needed are brought in later automatically.	No internal fragmentation, higher degree of multiprogramming; large virtual address space; protection and sharing.	Overhead of complex memory management