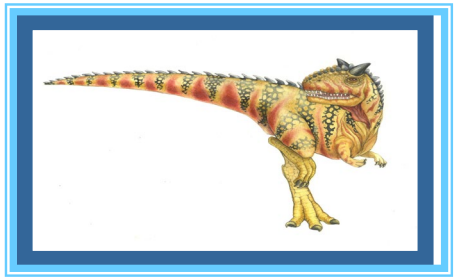
Chapter 8: Main Memory



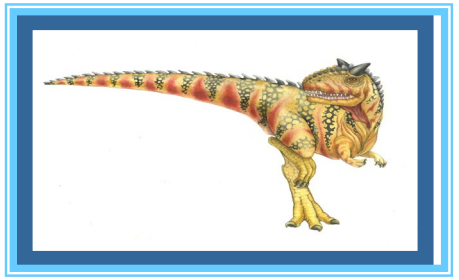
Chapter 8: Memory Management

- 8.1 Background
- 8.2 Swapping（交换）
- 8.3 Contiguous Memory Allocation（连续内存分配）
- 8.4 Paging（分页）
- 8.5 Segmentation（分段）
- 8.6 Segmentation with Paging（段页式）
- 8.7 Example: The Intel Pentium

Objectives

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging

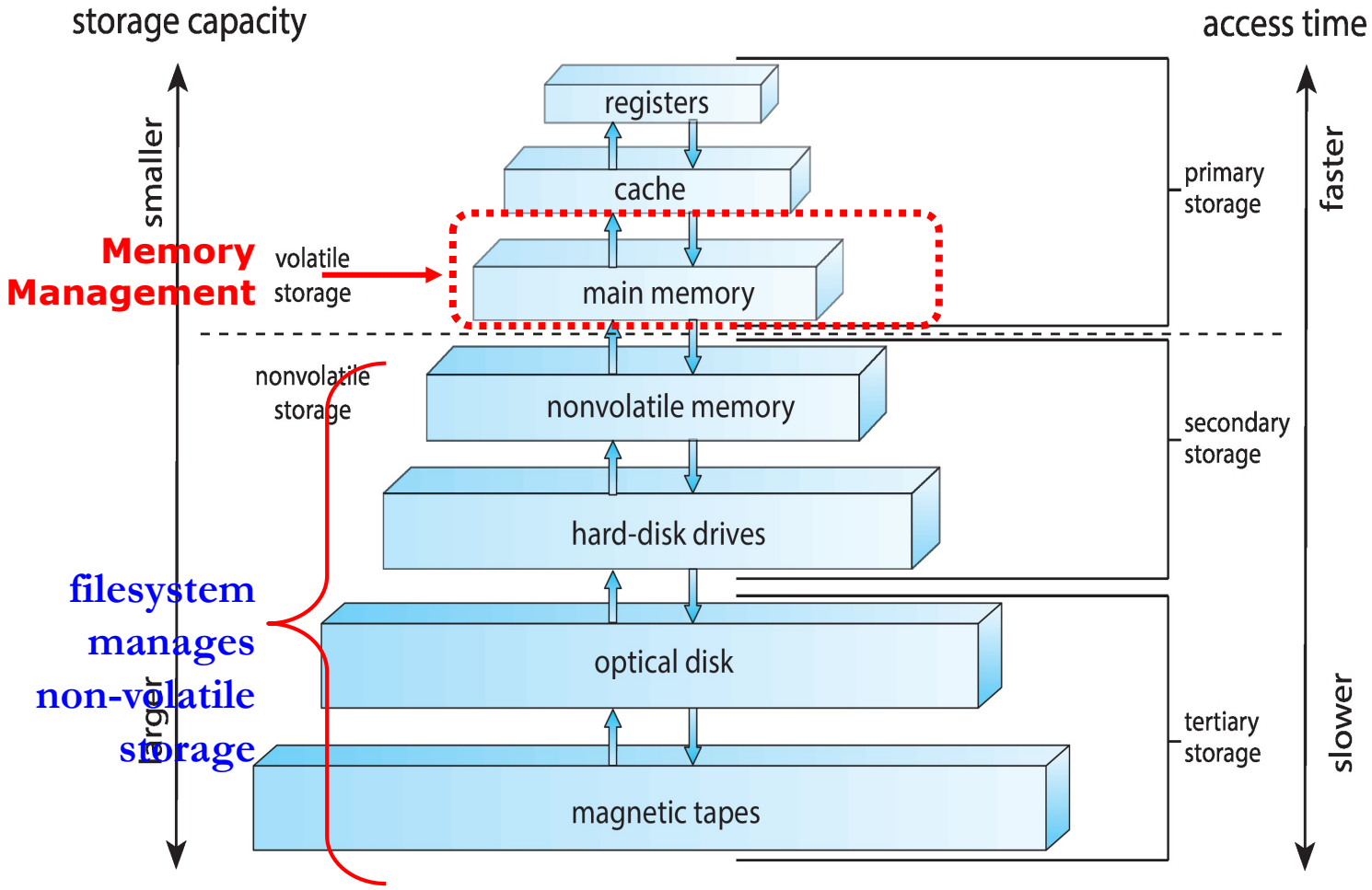
8.1 Background



Background

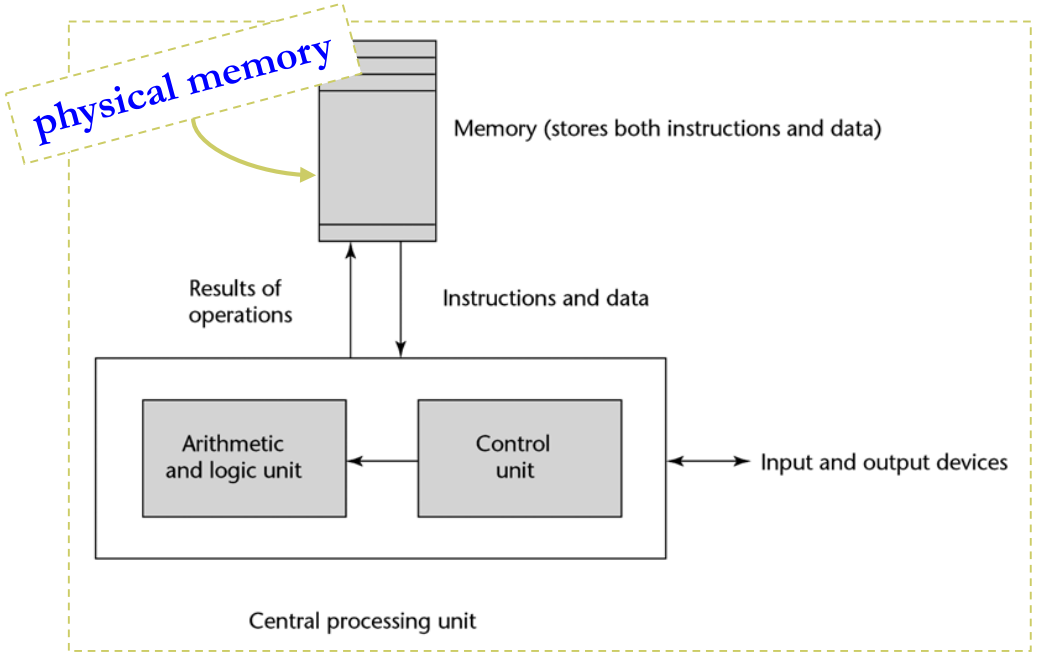
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Register access in one CPU clock (or less)
- Main memory can take many cycles
- Cache sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

Hierarchical Storage Architecture



Main Memory

- Based on the von Neumann architecture, data and program instructions exist in shared memory space

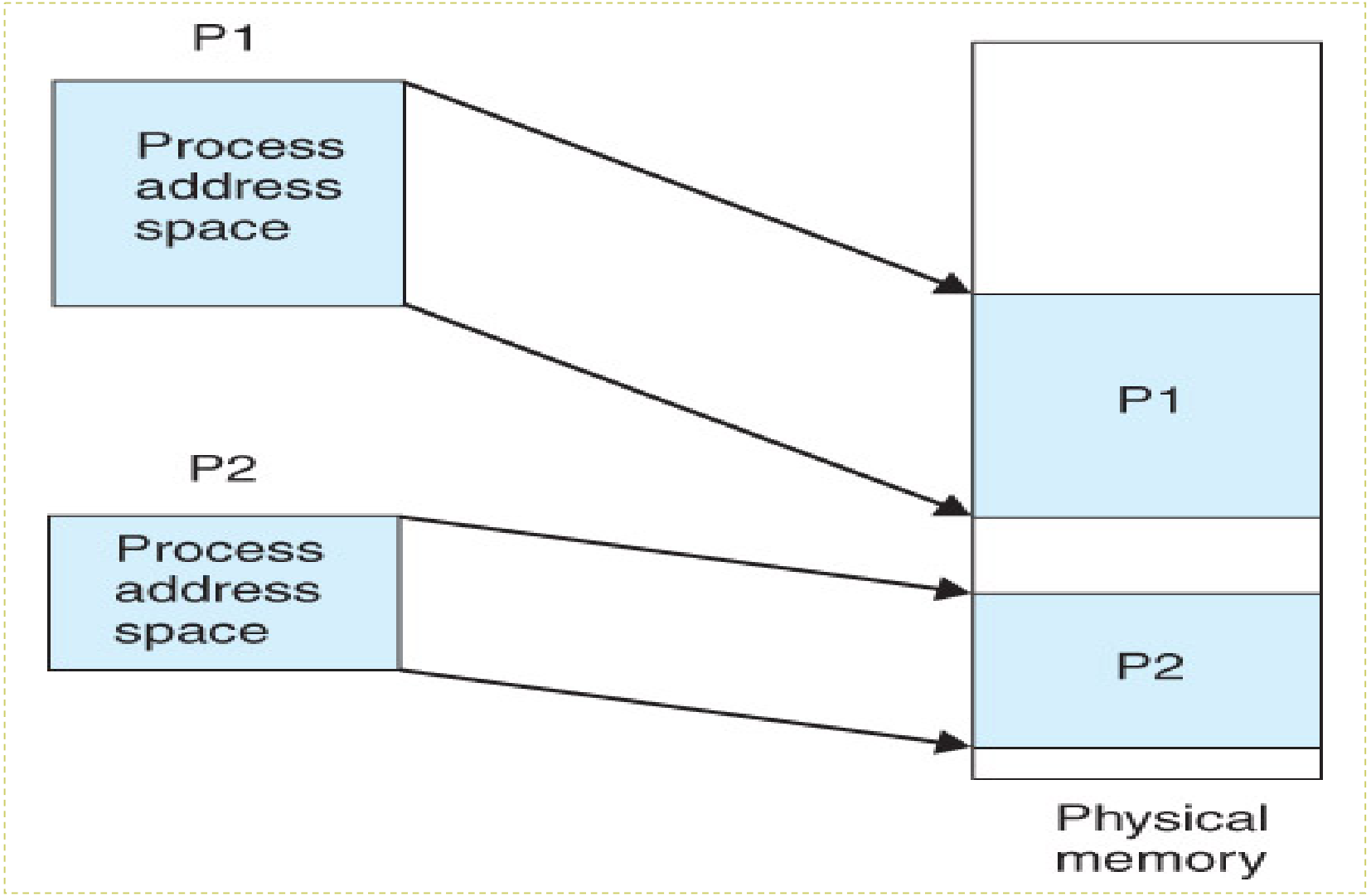


  - Repeatedly perform fetch-decode-execute cycles
  - The execute part often results in data fetch and store operations

Logical vs. Physical Address Space

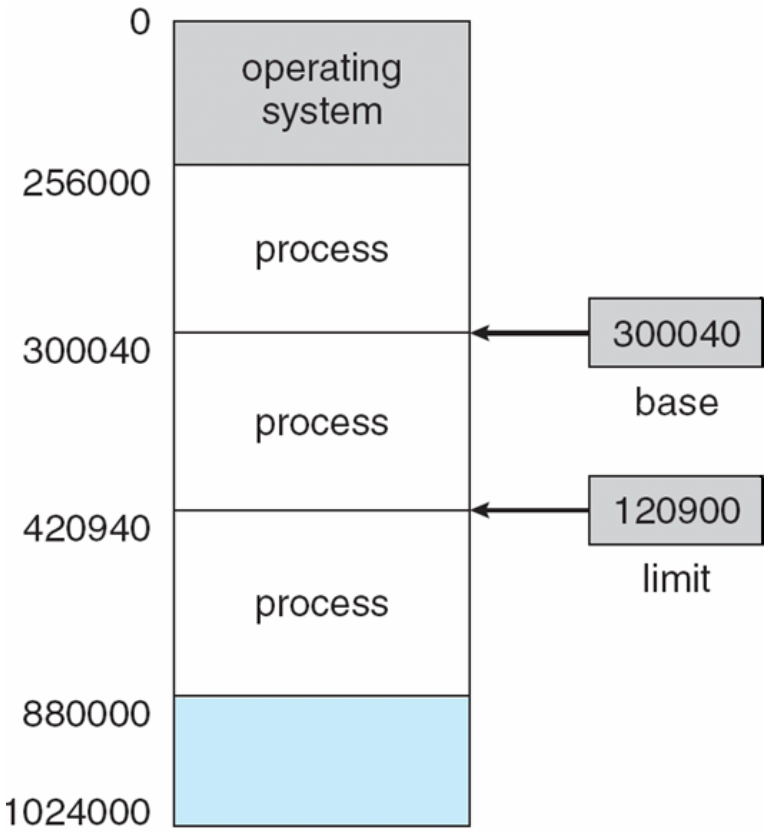
- The concept of a logical address space that is bound to a separate physical address space is central to proper memory management
  - Logical address（逻辑地址，相对地址，虚地址）– generated by the CPU; also referred to as virtual address
    - 用户的程序经过汇编或编译后形成目标代码，目标代码通常采用相对地址的形式。
    - 其首地址为0，其余指令中的地址都相对于首地址来编址。
    - 不能用逻辑地址在内存中读取信息。
  - Physical address（物理地址，绝对地址，实地址）– address seen by the memory unit
    - 内存中存储单元的地址。物理地址可直接寻址

Logical vs. Physical Address Space



Base and Limit Registers

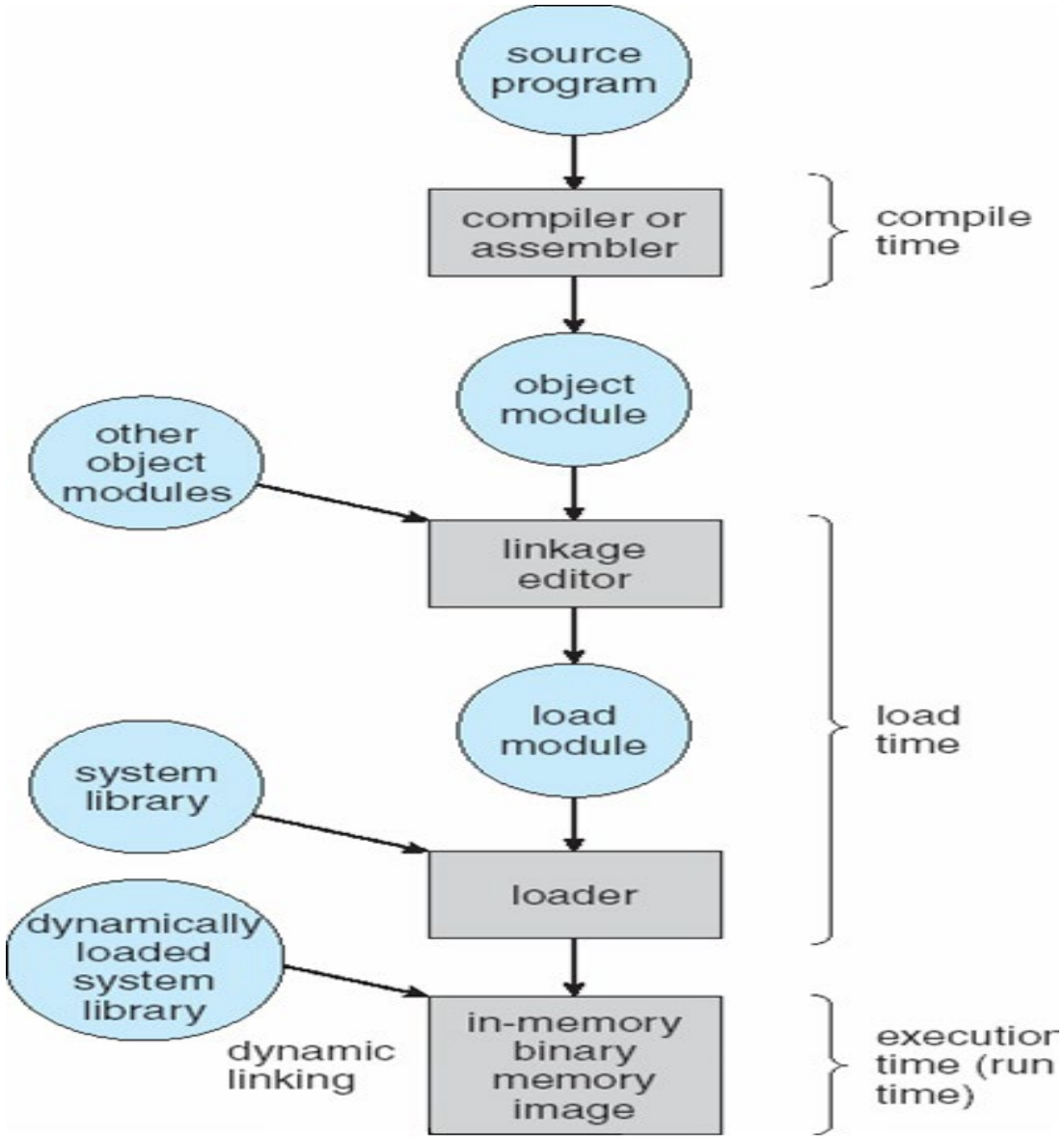
- A pair of base and limit registers（基址寄存器和限长寄存器）define the logical address space



Binding of Instructions and Data to Memory

- **Address binding**（地址绑定、地址映射、重定位）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: 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)

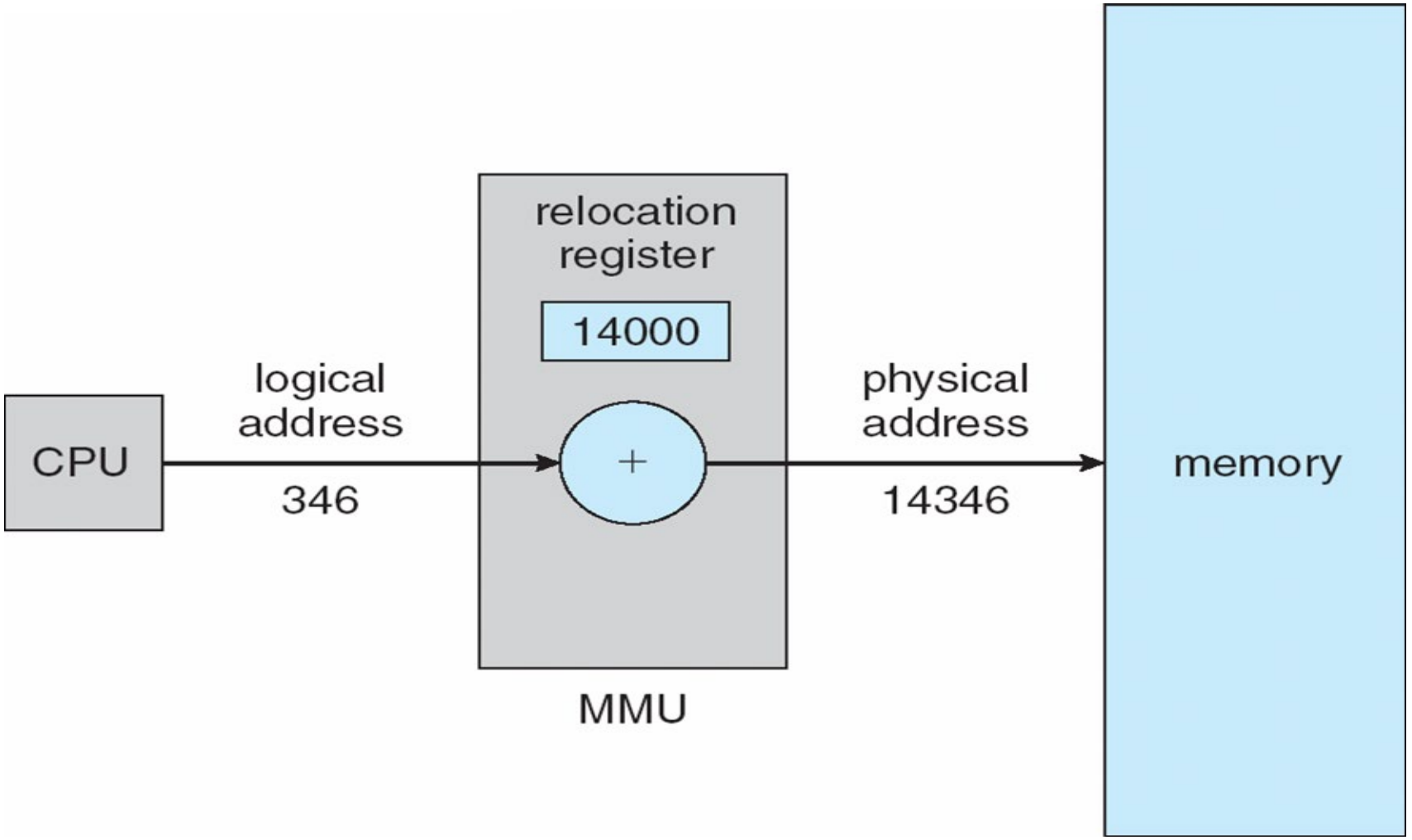
Multistep Processing of a User Program



Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address
- In MMU scheme, the value in the relocation register（重定位寄存器）is added to every address generated by a user process at the time it is sent to memory
- The user program deals with *logical* addresses; it never sees the *real* physical addresses

Dynamic relocation（动态重定位）



Dynamic Loading（动态转入）

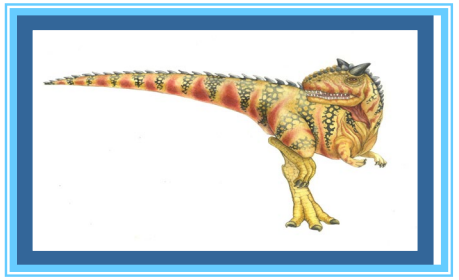
- Using dynamic loading, external libraries are not loaded when a process starts
  - Libraries are stored on disk in relocatable form
  - Libraries loaded into memory only when needed

- Better memory-space utilization; unused routine is never loaded
- 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

Dynamic Linking（动态链接）

- Using dynamic linking, external libraries can be preloaded into (shared) memory
  - When a process calls a library function, the corresponding physical address is determined
- 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 needed to check if routine is in processes' memory address
- Dynamic linking is particularly useful for libraries
- System also known as shared libraries
- **Dynamically Linked Library 动态链接库**

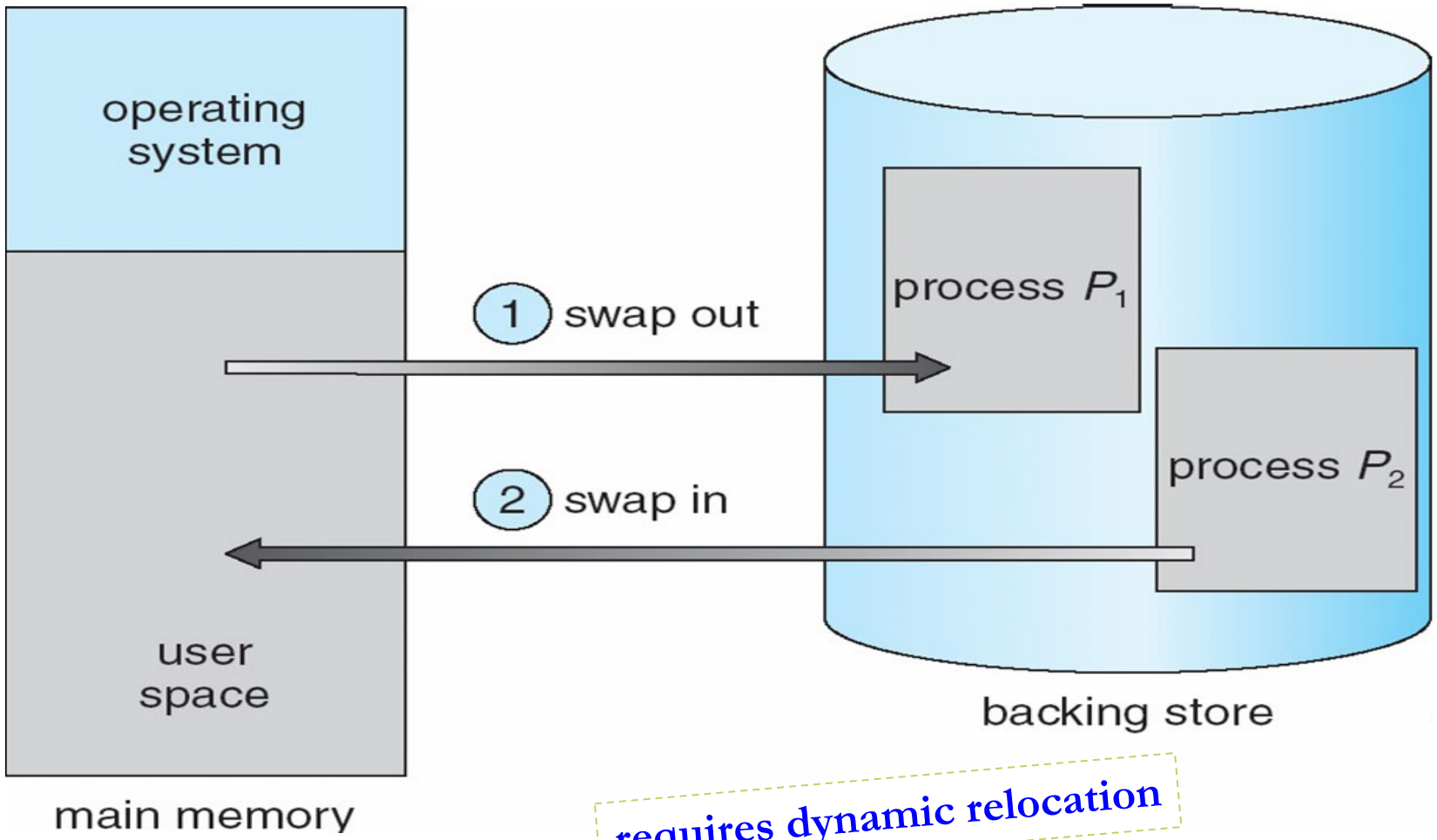
8.2 Swapping（交换技术）



Swapping（交换技术）

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
- Backing store – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
  - **Linux、UNIX—交换区**
  - **Windows—交换文件(pagefile.sys)**
- **Roll out, roll in（调出，调进）** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- **Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)**
- System maintains a ready queue of ready-to-run processes which have memory images on disk

Schematic View of Swapping



requires dynamic relocation

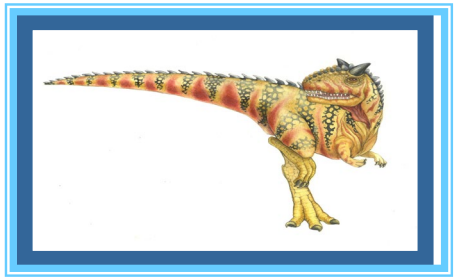
Swapping (Cont.)

- UNIX, Linux, and Windows:
  - 正常情况下，禁止交换
  - 当空闲内存低于某一个阈值时，启用交换换出
  - 当空闲内存增加一定数量时，停止换出

Swapping on Mobile Systems

- 移动系统不支持交换，Flash memory based：
  - 小空间
  - 闪存写次数限制
  - 在移动平台上闪存和CPU之间的吞吐量很低
- iOS要求应用程序自愿放弃分配的内存
  - 只读数据从系统中直接删除，已修改数据不会被删除
  - OS可以终止任何未能释放足够空间的应用
- Android如果空闲内存不足，会终止应用程序，但首先会将应用程序状态写入闪存，以便快速重启

8.3 Contiguous Allocation（连续分配）

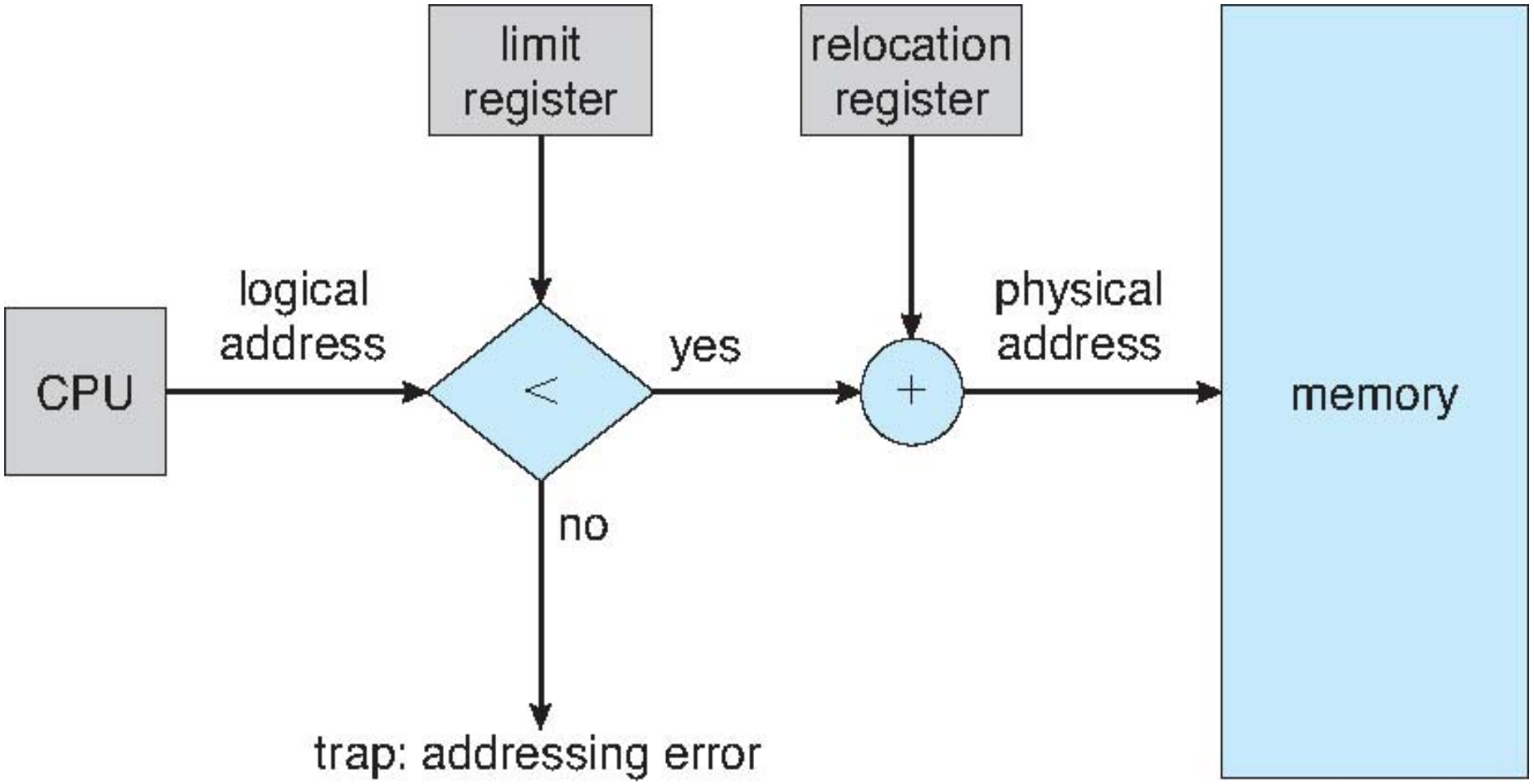


Contiguous Allocation（连续分配）

- Main memory usually into two partitions(单分区):
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory

- Relocation registers 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*

Hardware Support for Relocation and Limit Registers



Contiguous Allocation (Cont.)

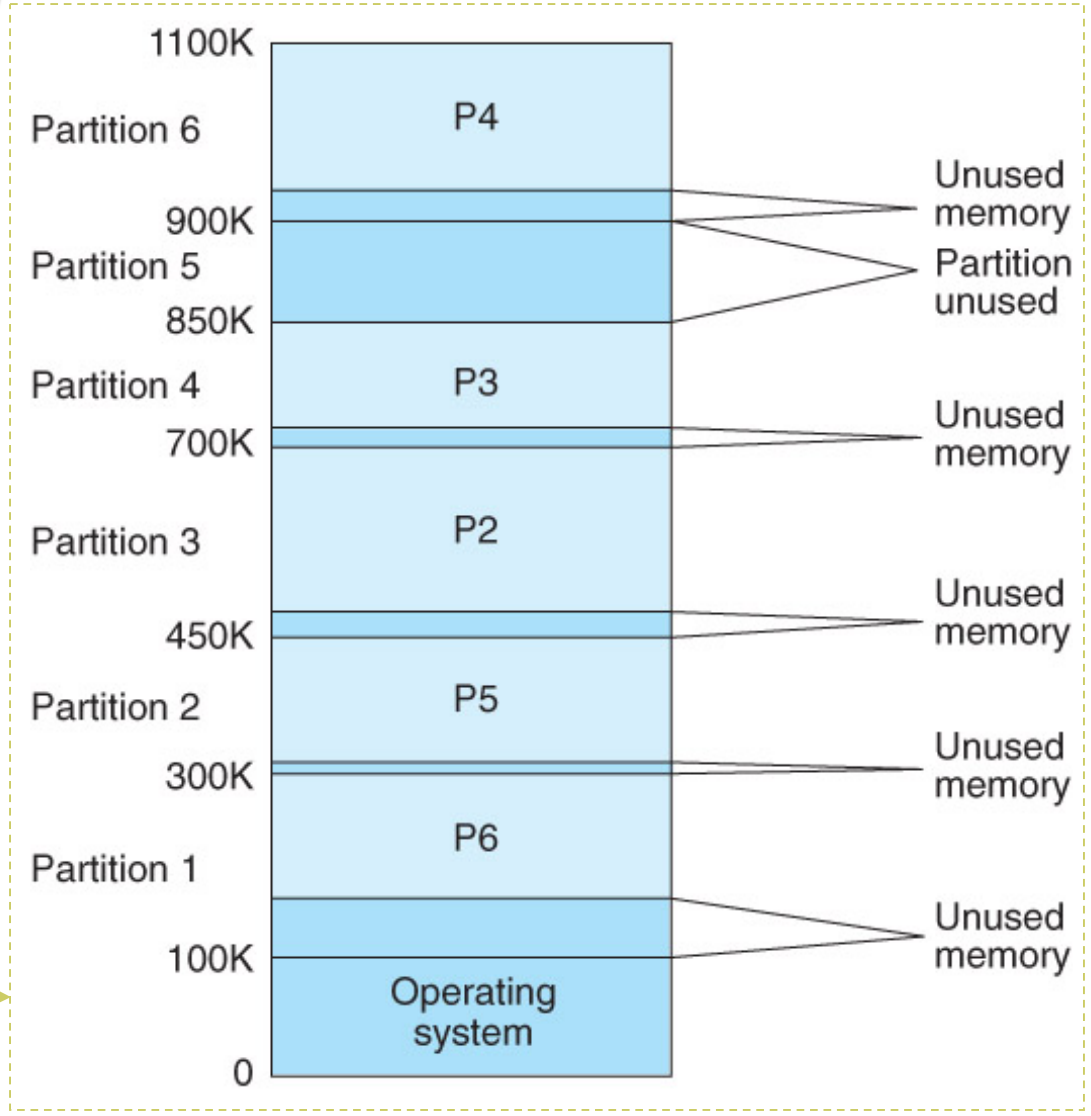
Multiple-partition allocation （多分区分配）

- 分区式管理的基本思想是将内存划分成若干个连续区域，称为分区。每个分区只能存放一个进程。
  - fixed partitioning（固定分区）
  - Dynamic Partitions（动态分区）

Contiguous Allocation (Cont.)

一、fixed partitioning（固定分区）

- Main memory is partitioned and allocated to the resident operating system and processes



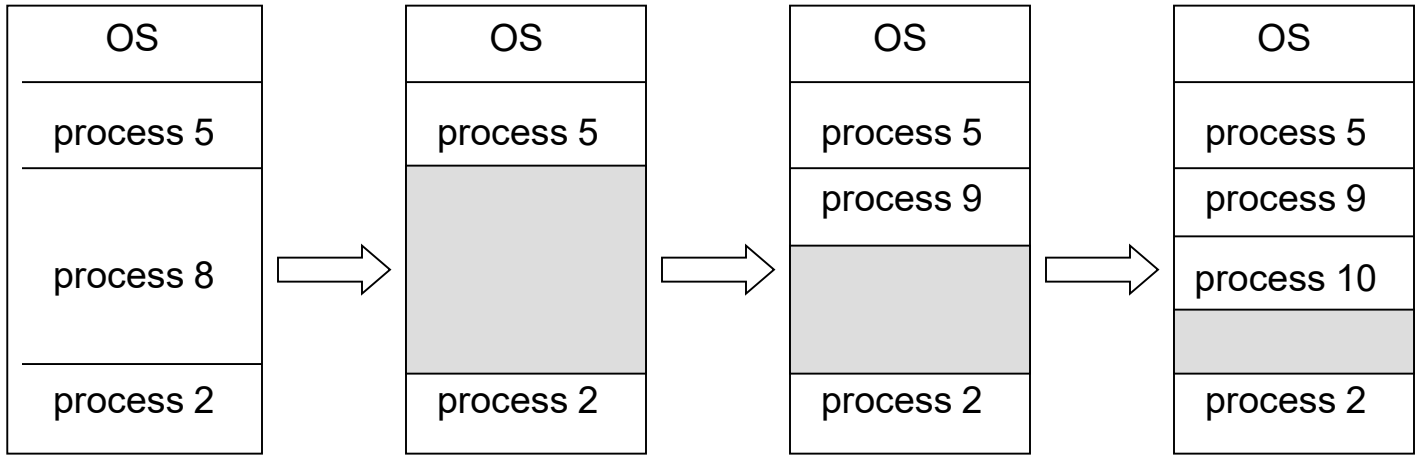
fixed partitioning scheme

Contiguous Allocation (Cont)

二、 Dynamic Partitions（动态分区）

- 动态划分内存，在程序装入内存时把可用内存“切出”一个连续的区域分配给该进程，且分区大小正好适合进程的需要。
- Operating system maintains information about:
  - allocated partitions （已分配的分区）
  - free partitions (hole) （空闲的分区）

operating system maintains allocated partitions and free partitions



Dynamic Storage-Allocation Problem

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

- First-fit: Allocate the *first* hole that is big enough
- Best-fit: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- Worst-fit: Allocate the *largest* hole; must also search entire list
  - Produces the largest leftover hole

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

Dynamic Storage-Allocation Problem

1. **First-fit**(首次适应): Allocate the *first* hole that is big enough.

- 设进程创建顺序： A：12K， B：10K，C： 3K

OS	OS	OS	OS
20K	进程A 12K	进程A 12K	进程A 12K
	8K	8K	进程C 3K
			5K
17K	17K	进程B 10K	进程B 10K
		7K	7K
初始状态	进程A创建	进程B创建	进程C创建

Dynamic Storage-Allocation

2. Best-fit（最佳适应）： Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole

- 进程创建顺序：A：12K，B：3K，C：10K

Dynamic Storage-Allocation

3. Worst-fit （最差适应） ： Allocate the *largest* hole; must also search entire list. Produces the largest leftover hole.

- 进程创建顺序：A：12K，B：3K，C：10K

Dynamic Storage-Allocation

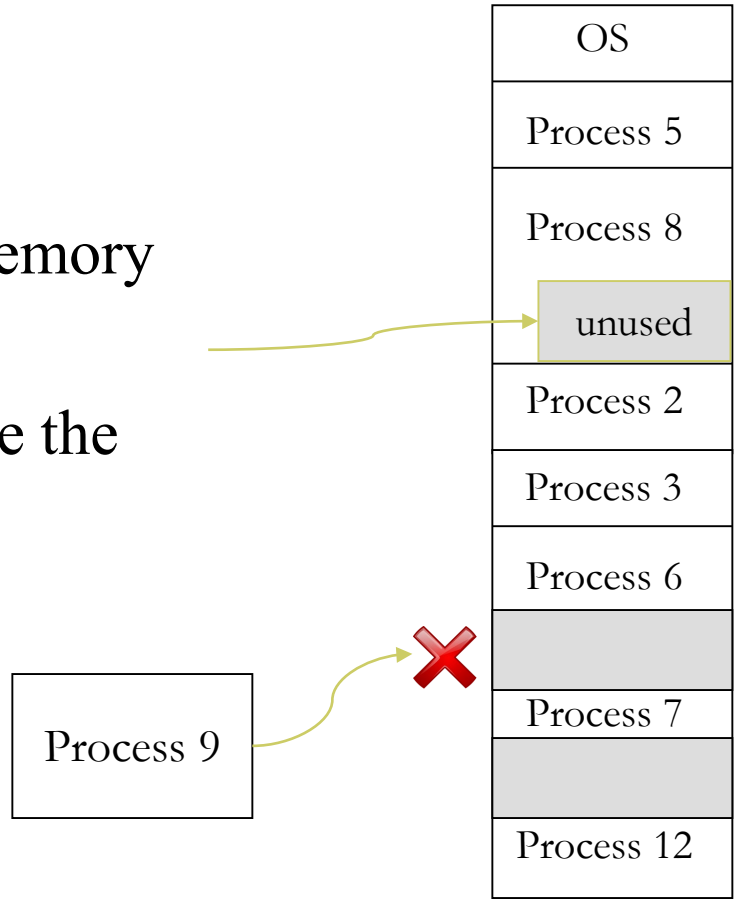
4. Next Fit（下次适应）：类似首次适应，每次分区时，总是从上次查找结束的地方开始，只要找到一个足够大的空白区，就把它划分后分配出去。
   - 进程创建顺序： A：12K， B：10K， C： 3K

Demo

Fragmentation

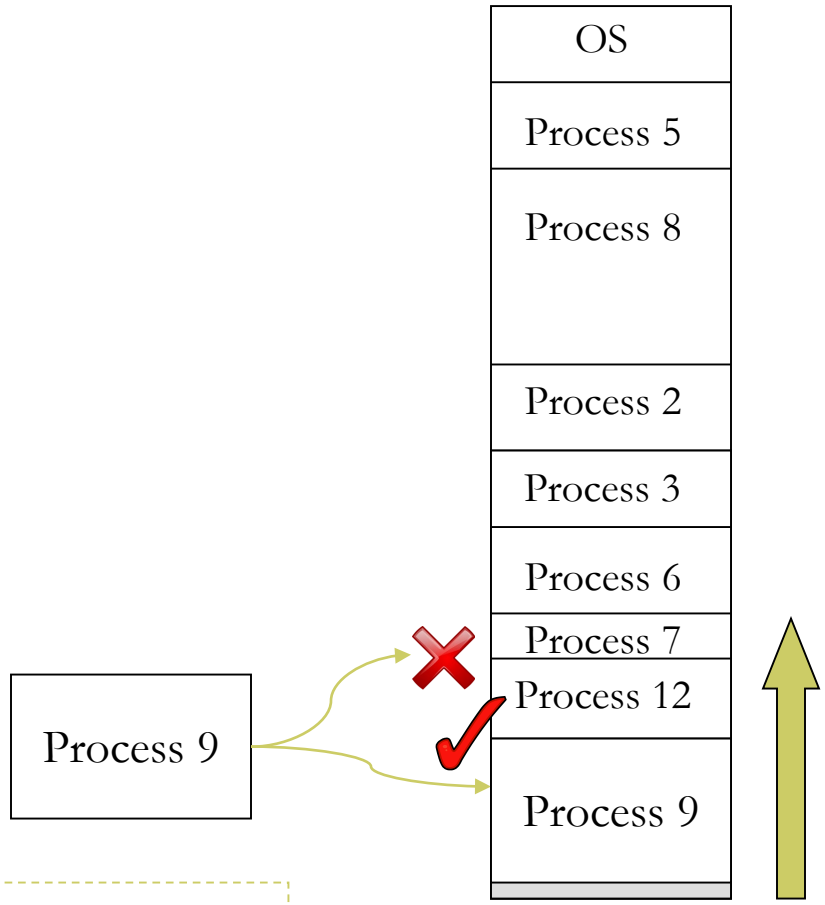
- Memory is wasted due to fragmentation, which can cause performance problems
  - Internal fragmentation (内碎片、内零头) is wasted memory *within* a single process memory space
  - External fragmentation（外碎片、外零头）can reduce the number of *runnable* processes
    - Total memory space exists

    to satisfy a request,but memory is not contiguous



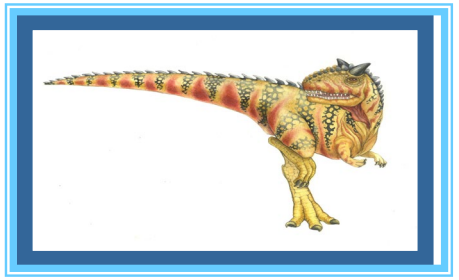
Fragmentation

- Reduce external fragmentation by compaction or defragmentation （紧缩，拼接）
  - Rearrange memory contents to organize all free memory blocks together in one large block
  - Compaction is possible only if relocation is dynamic and is done at execution time



defragmentation is expensive

8.4 Paging(分页，页式存储管理)



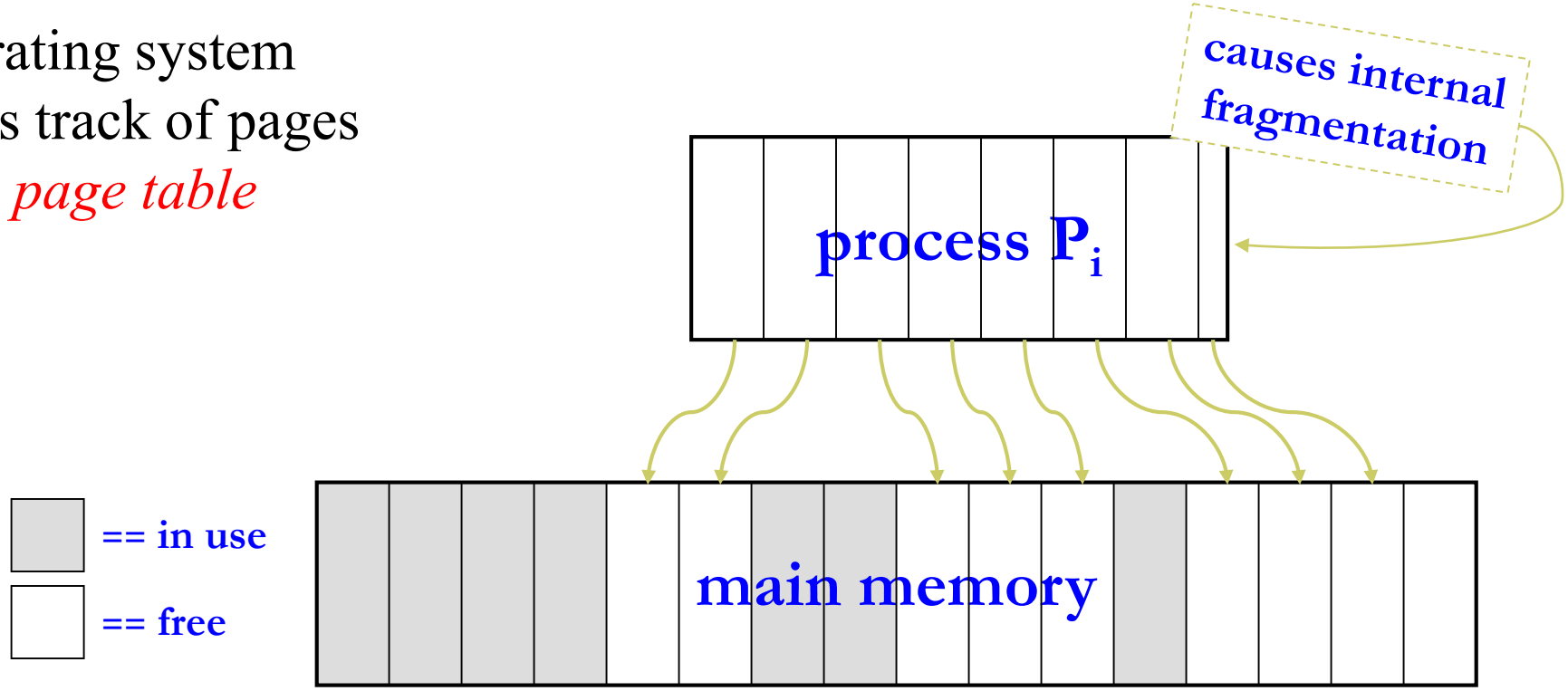
Paging

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Divide physical memory into fixed-sized blocks called **frames(帧、物理块、页框)** (size is power of 2, between 512 bytes and 8,192 bytes)
  - Linux 、Windows for x86: 4K
- Divide logical memory into blocks of same size called **pages（页）**
- 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（页表）** to translate logical to physical addresses

  page table列出了进程的逻辑页与其在主存中的物理帧间的对应关系
- Internal fragmentation

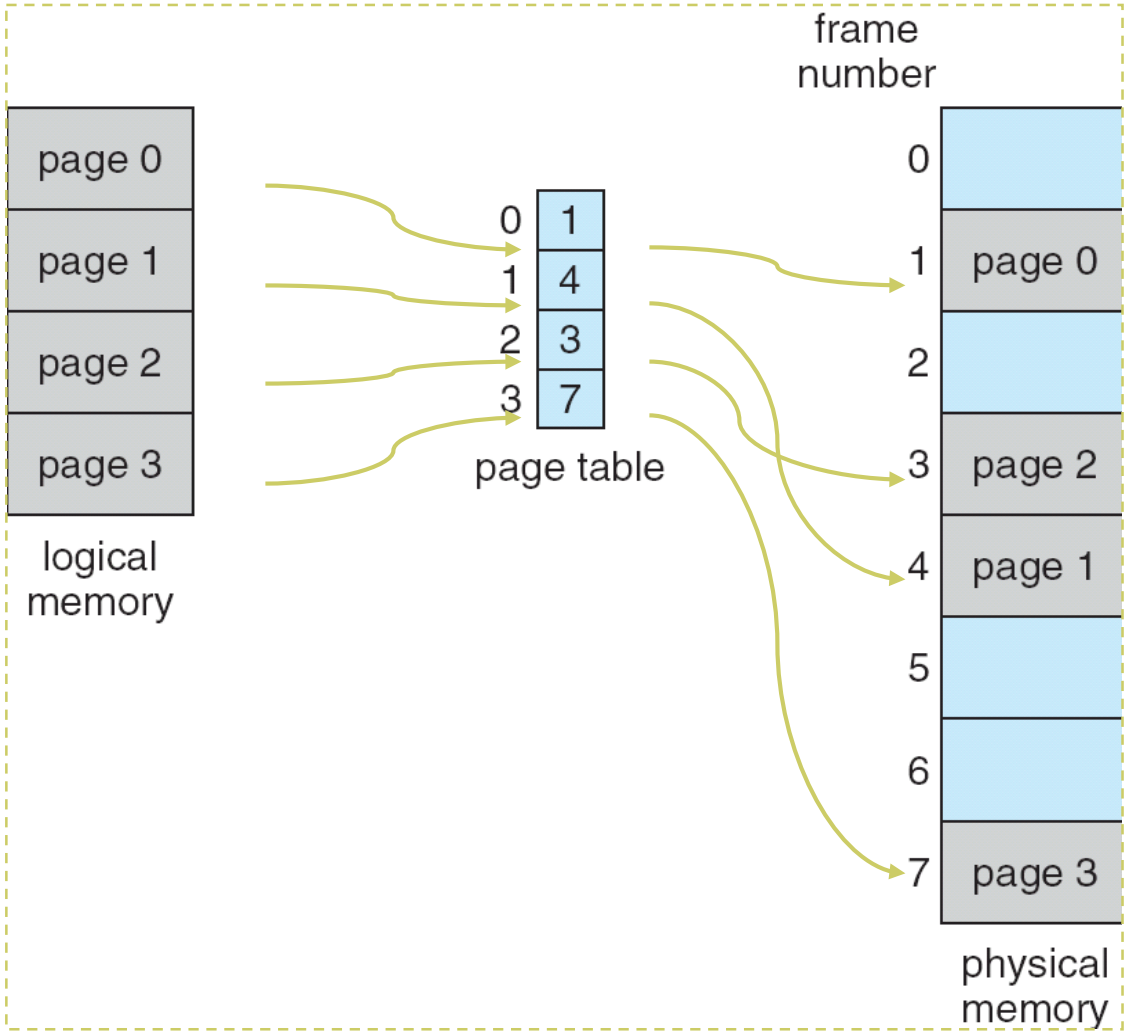
Noncontiguous Allocation

- When a process of size *n* pages is ready to execute, operating system finds *n* free frames
  - Operating system keeps track of pages via a *page table*



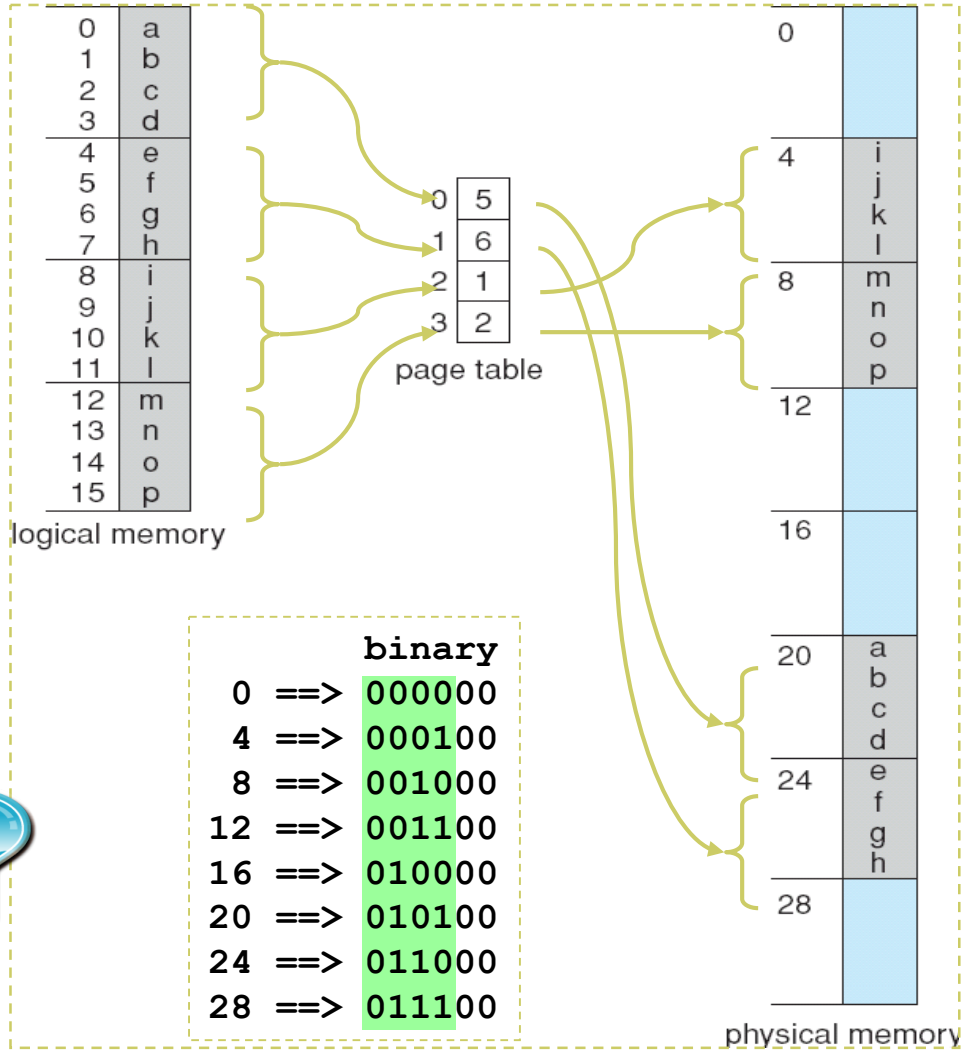
Paging Model of Logical and Physical Memory

- A page table maps logical memory addresses to physical memory addresses



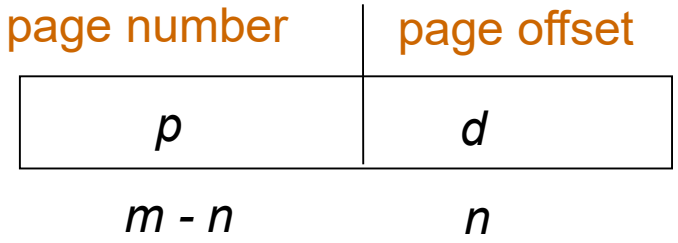
Paging Example

- Example process P_i requires 16 bytes of logical memory
  - Logical memory is mapped via a page table to a 32-byte memory with pages of size 4 bytes



Address Translation Scheme（地址变换）

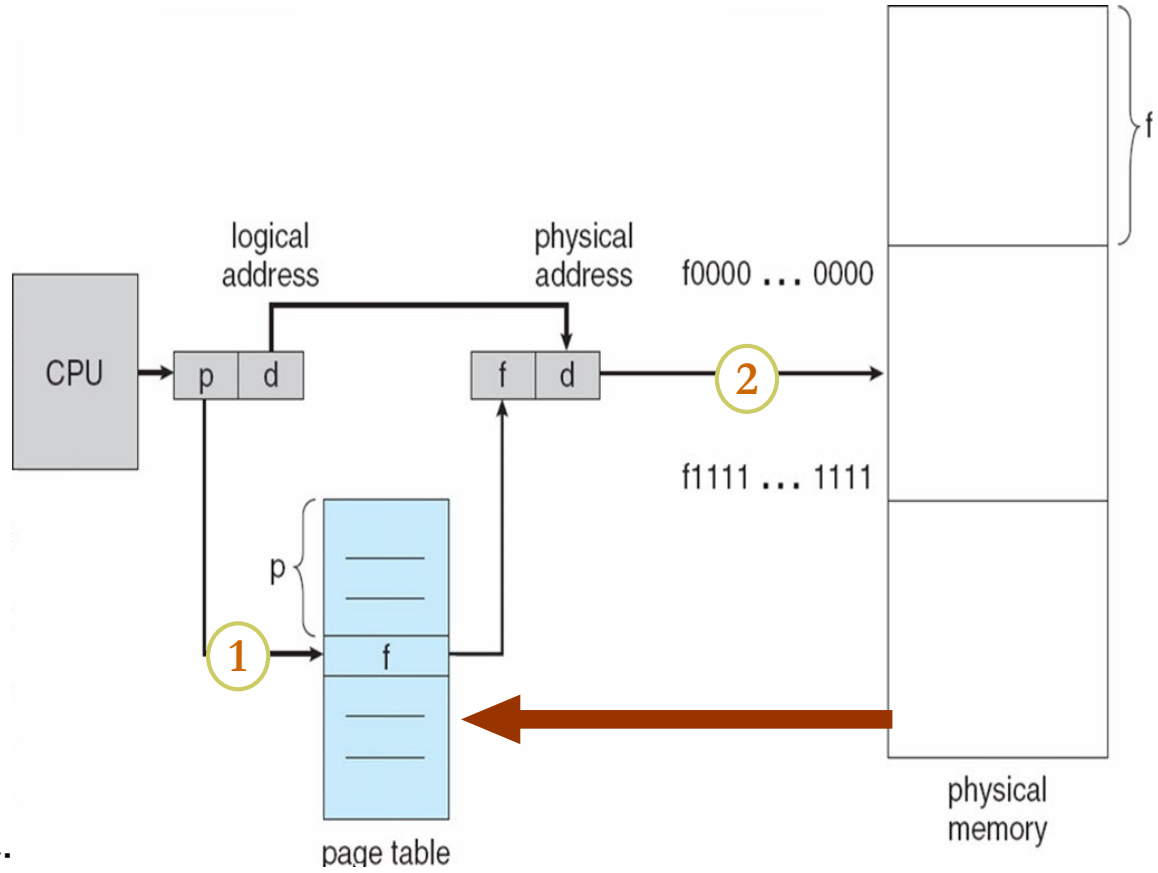
- Address generated by CPU is divided into:
  - Page number (*p*) （页号）– used as an index into a *page table* which contains base address of each page in physical memory
  - Page offset (d) （偏移）– combined with base address to define the physical memory address that is sent to the memory unit



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

Paging Hardware

Frames = physical blocks
Pages = logical blocks



HARDWARE

An address is determined by:

page number (index into table) + offset

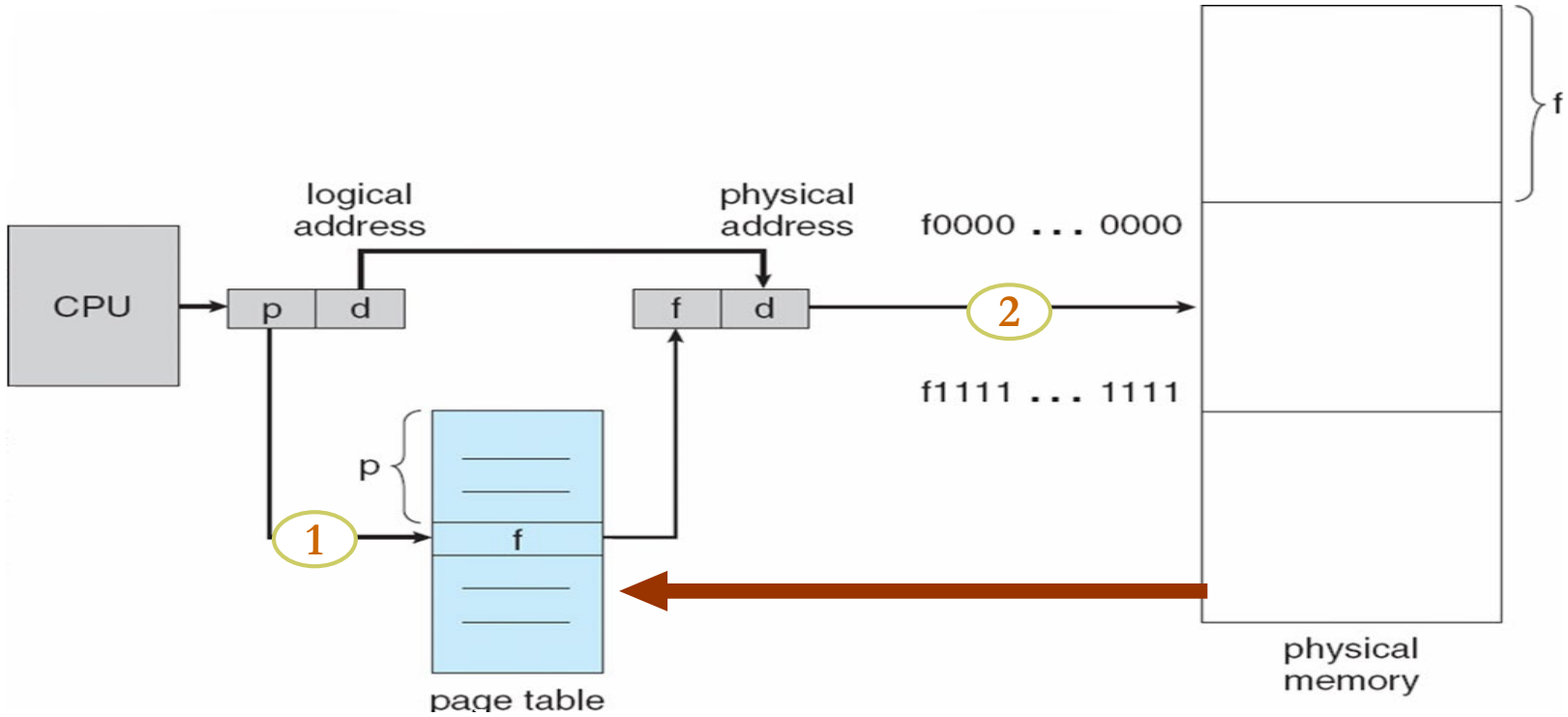
---> mapping into --->

base address (from table) + offset.

逻辑地址-物理地址转换

- 页面大小为4KB，虚地址2362H、1565H的物理地址分别是？页表如下

页号	页框 (Page Frame) 号
0	101H
1	102H
2	254H



Implementation of Page Table

- Page table is kept in main memory
- Page-table base register (PTBR) points to the page table， x86: cr3
- Page-table length register (PRLR) 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)（联想寄存器、快表）
- Some TLBs store address-space identifiers (ASIDs) in each TLB entry – uniquely identifies each process to provide address-space protection for that process

Associative Memory

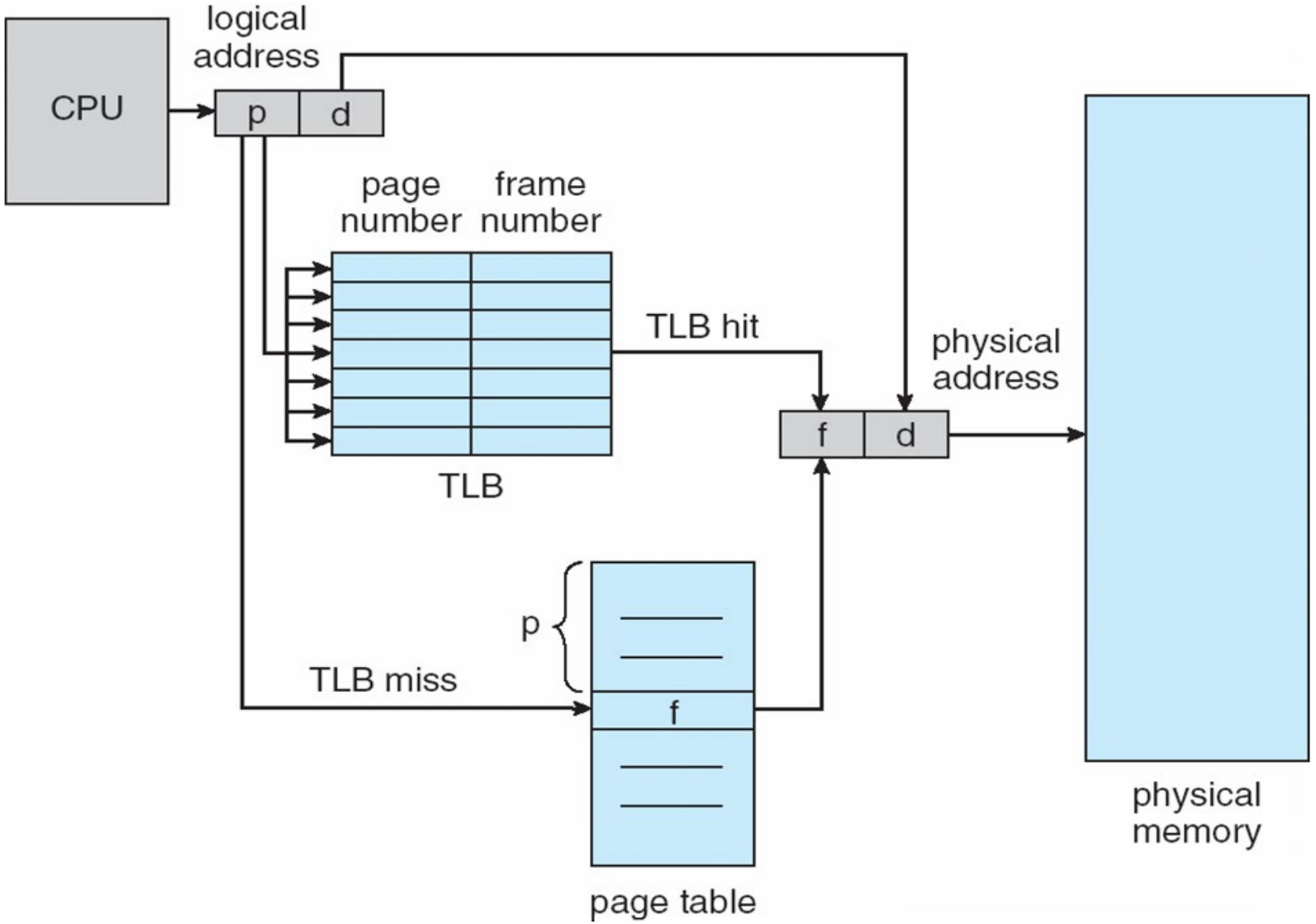
- Associative memory – parallel search

Page #	Frame #

Address translation (p, d)

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

Paging Hardware With TLB



Effective Access Time（有效访问时间）

- Associative Lookup = ε time unit （块表访问时间）
- Assume memory cycle time is t （内存访问时间）
- Hit ratio（命中率） – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Hit ratio = α
- Effective Access Time (EAT)

  $EAT = (t+\varepsilon)\,\alpha + (t + t + \varepsilon)\,(1 - \alpha)$

Effective Access Time

- Example: α(hit ratio) =80% , ε(TLB) =2ns, memory access time=100 ns
- Effective Access Time (EAT)

  EAT = (100 + 2)*0.8 + (100+100+2) *(1 – 0.8)

  = 122 ns (nanoseconds，纳秒)

TLB时间忽略不计的情况:120ns

- No associative registers

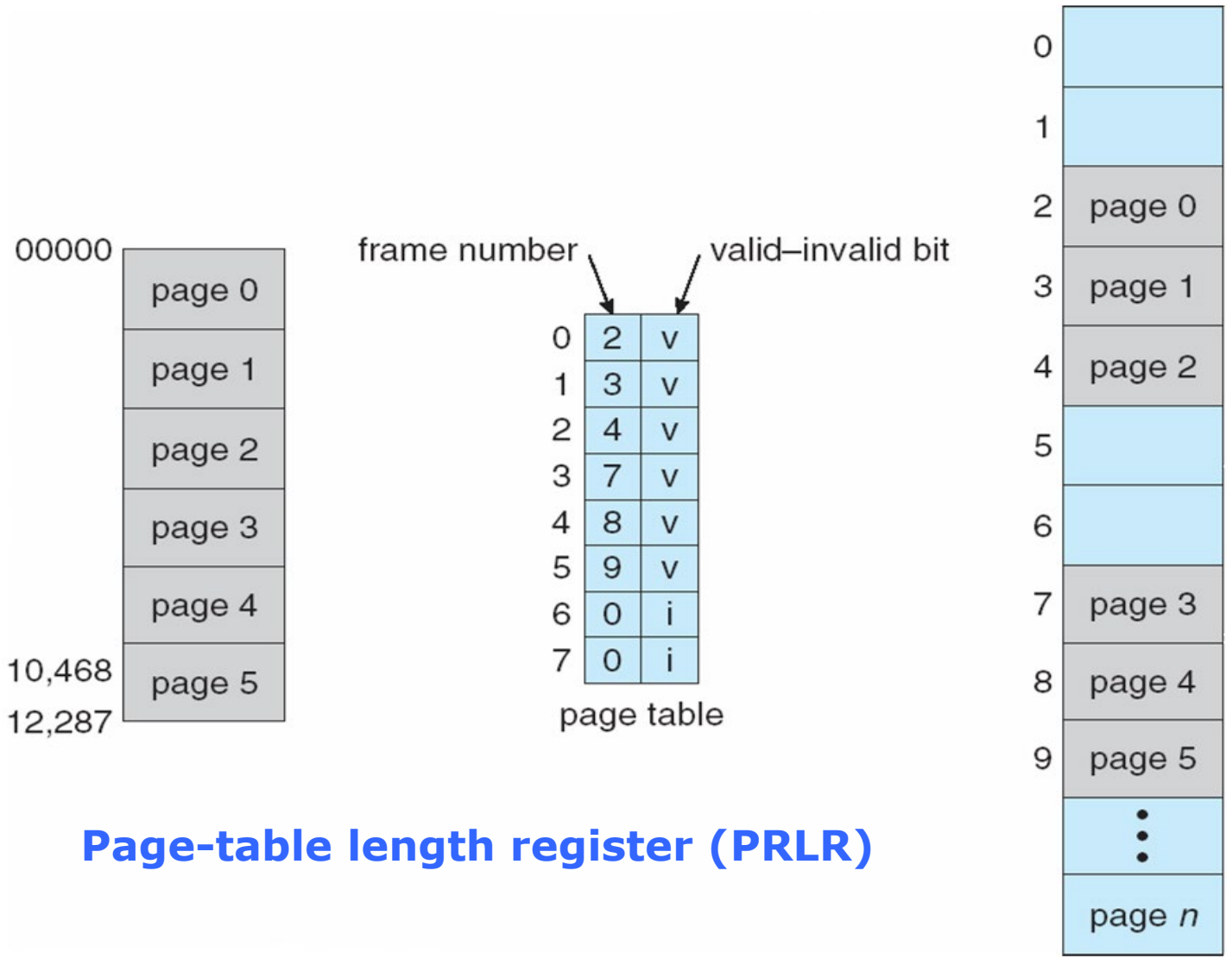
  EAT=2*(memory access time)=100+100=200 ns

microsecond（微秒，us）

Memory Protection

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

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

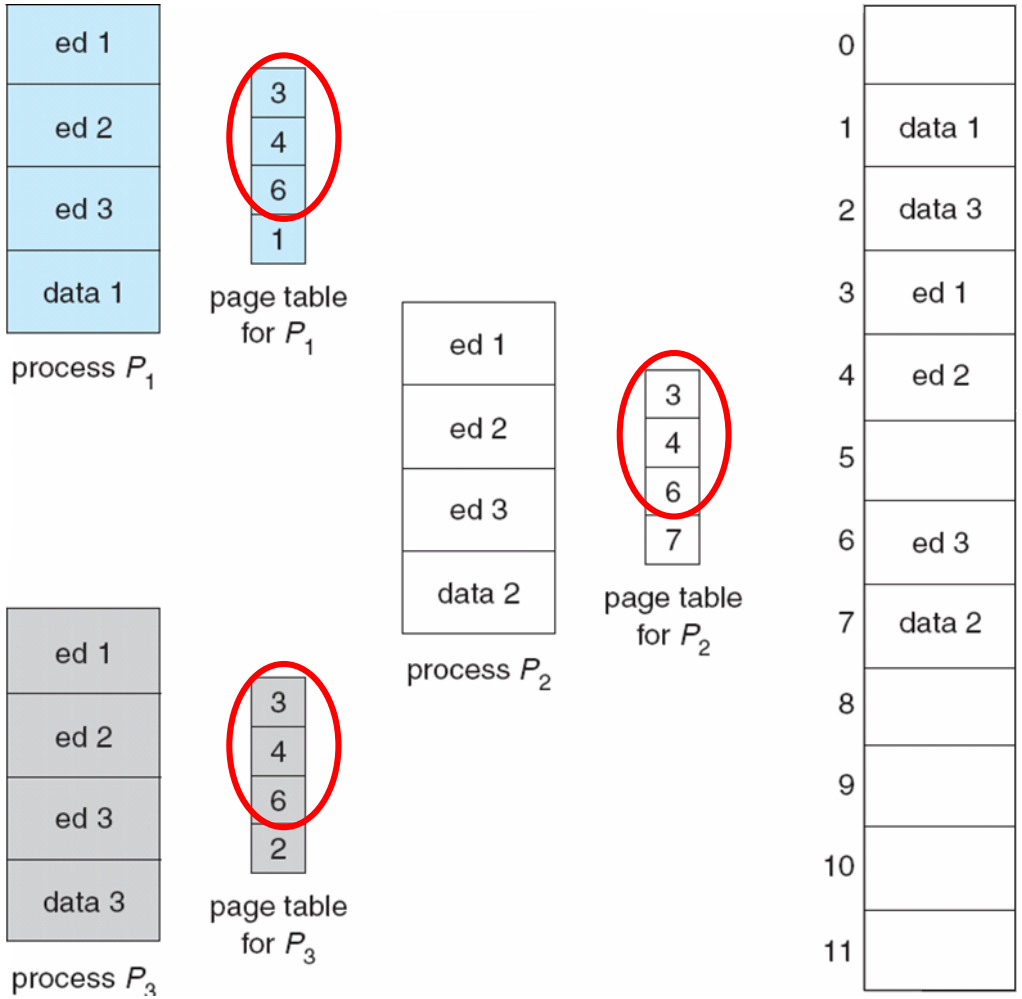


Page-table length register (PRLR)

Shared Pages

- Shared code
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes
- Private code and data
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space
- Reentrant Code(重入代码)、 Pure Code(纯代码)

Shared Pages Example



Structure of the Page Table

- 1. Hierarchical Paging　分级页表
- 2. Hashed Page Tables 哈希页表
- 3. Inverted Page Tables 反向(反置)页表

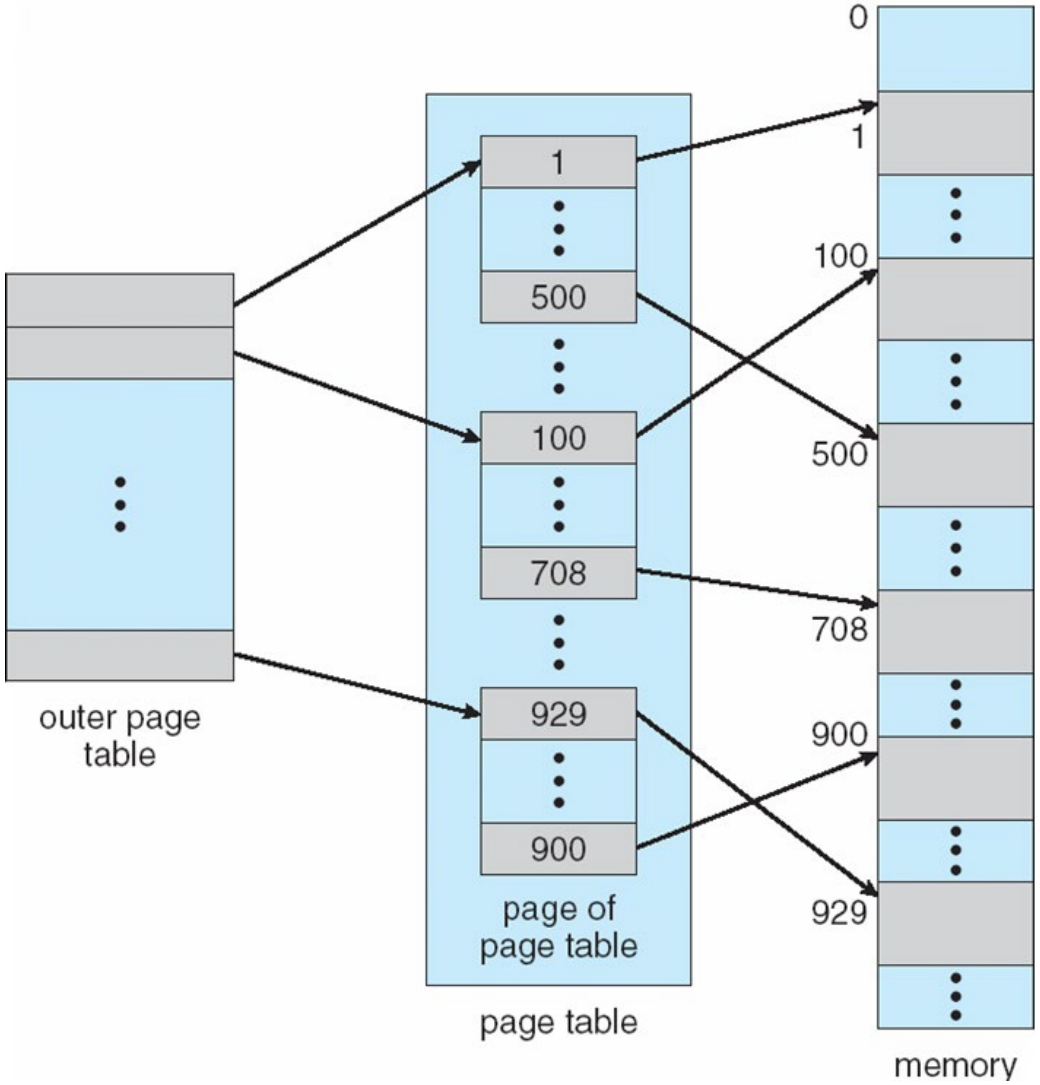
1. Hierarchical Page Tables

- x86的逻辑地址空间有2^{32}Byte，如页面大小为4KB（2^{12}Byte），则页表项最多有1M（2^{20}）个，每个页表项占用4Byte，故每个进程的页表占用4MB内存空间，还要求是连续的，显然这是不现实的。

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

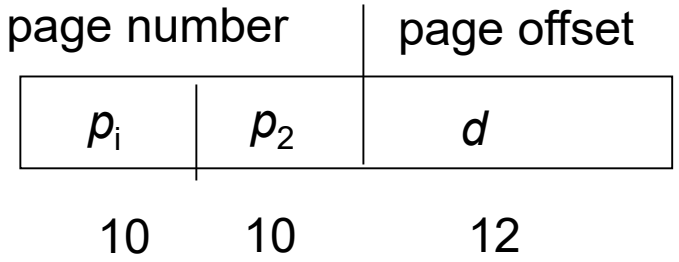
- Linux: four-level page table
- Windows: two-level page table

Two-Level Page-Table Scheme



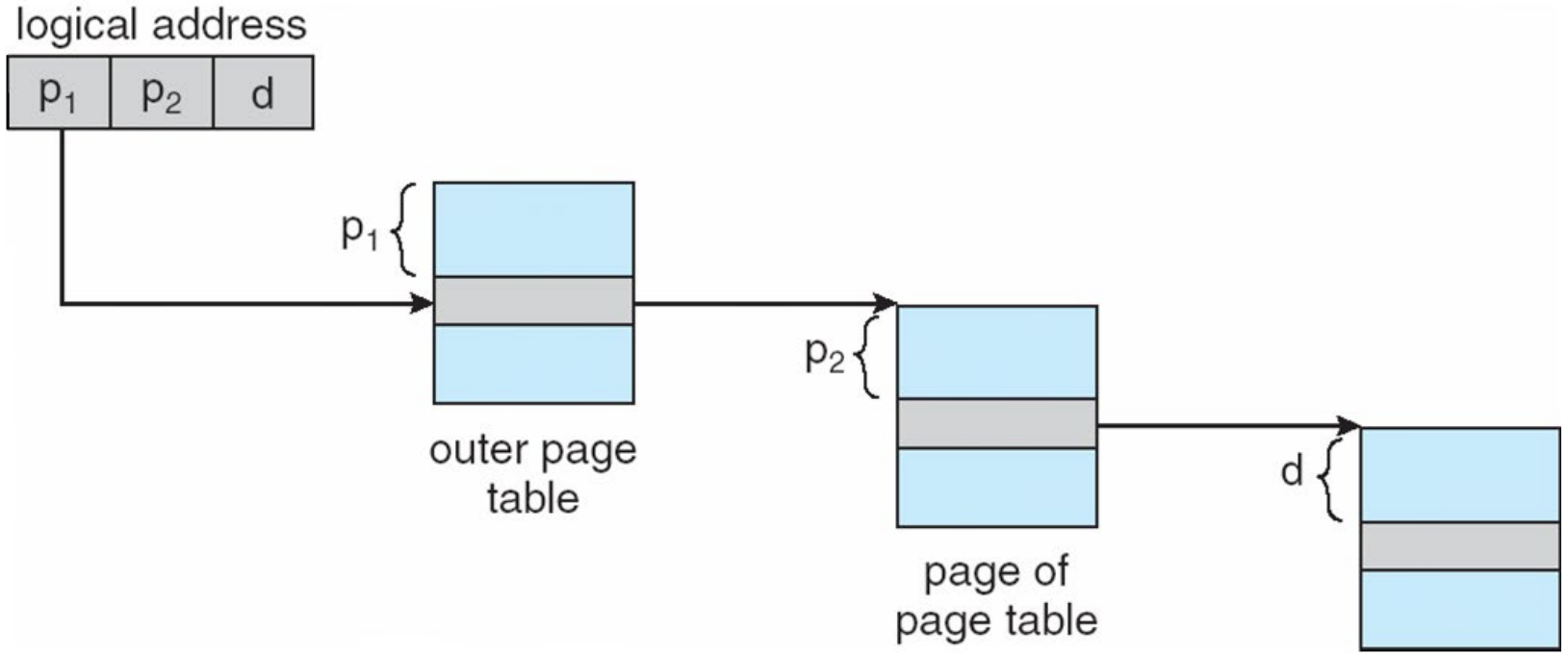
Two-Level Paging Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:

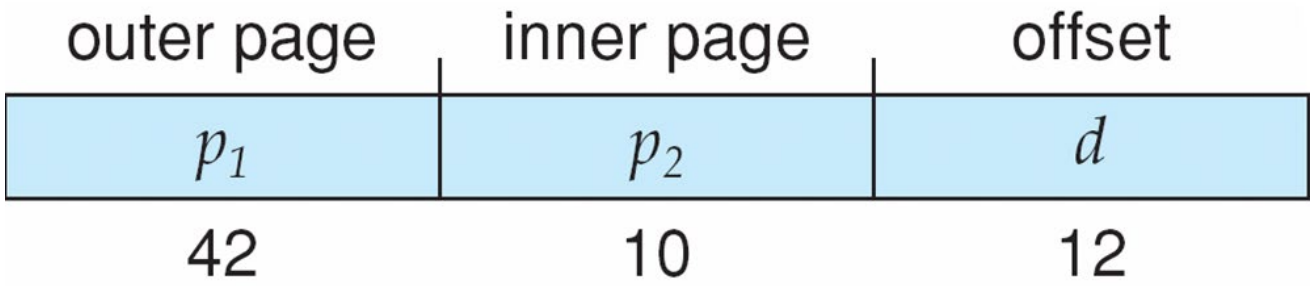


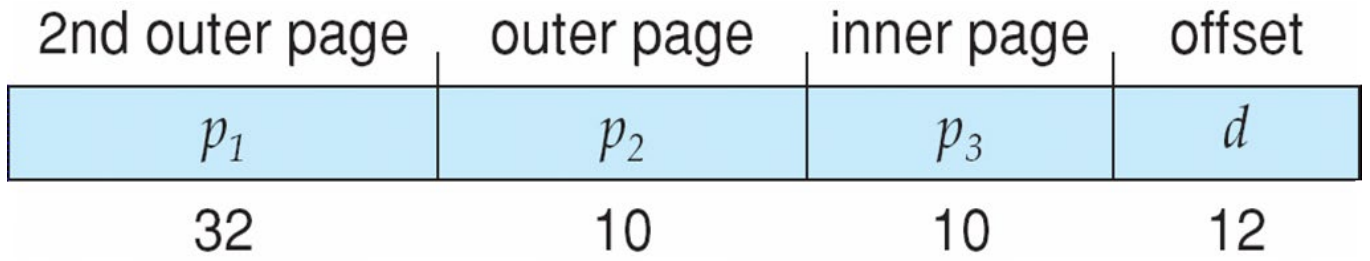
- where p_i is an index into the outer page table, and p_2 is the displacement within the page of the outer page table

Address-Translation Scheme



Three-level Paging Scheme



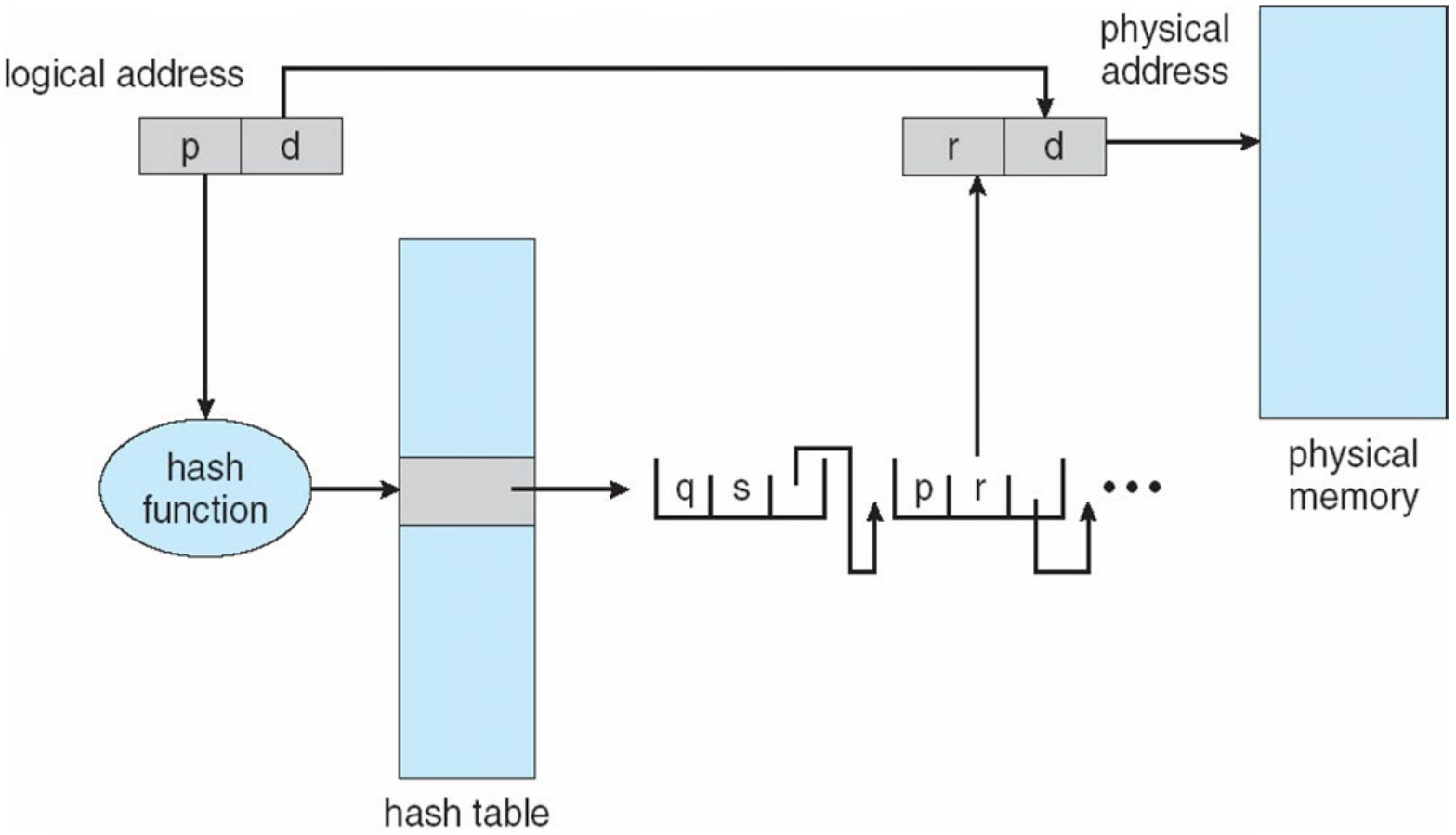


2. Hashed Page Tables

- 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
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted

- 例：SPARC Solaris (64位)

Hashed Page Table

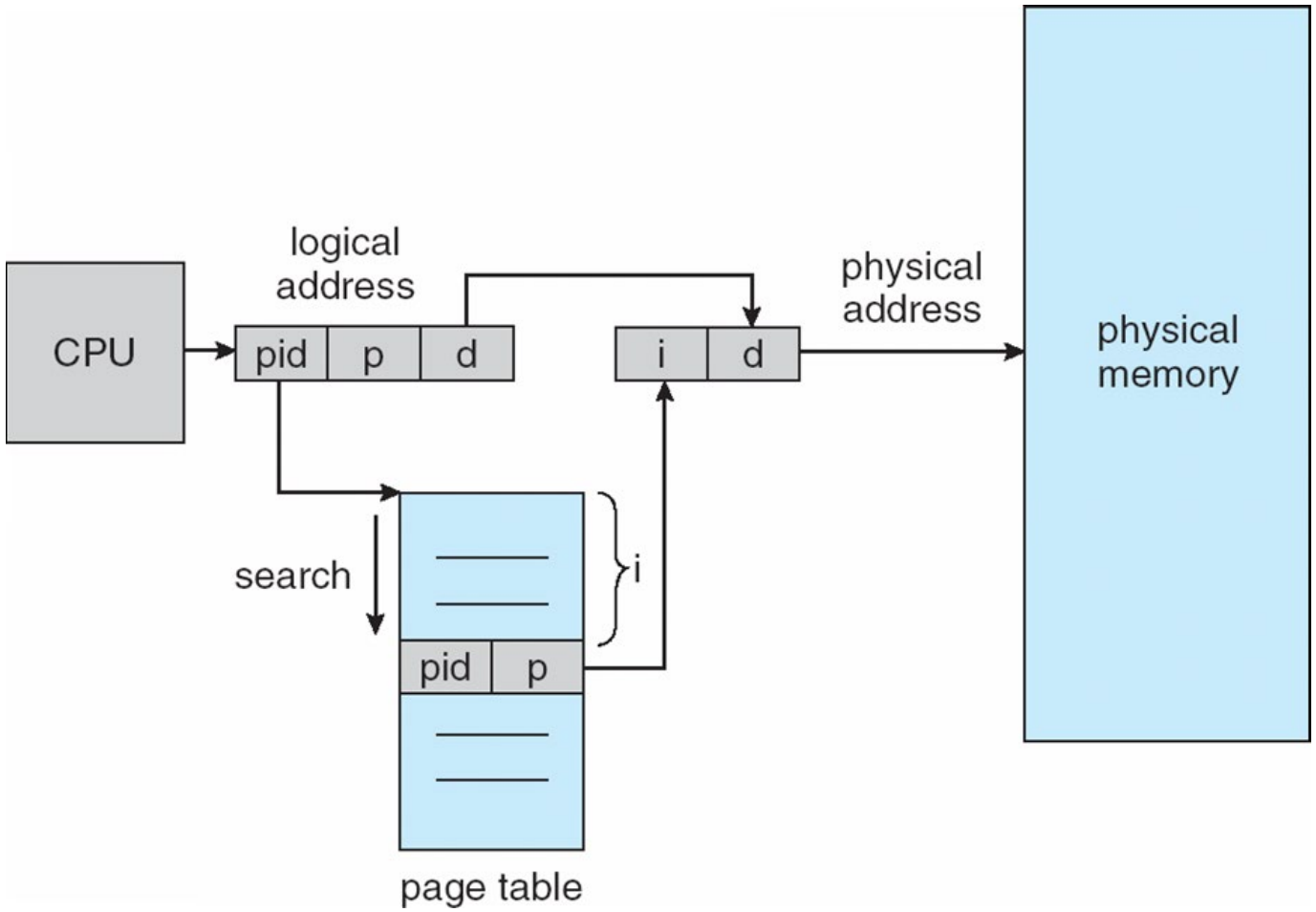


3. Inverted Page Table

- 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
- Use hash table to limit the search to one — or at most a few — page-table entries

- 例：UltraSPARC(64位)，PowerPC (64位)

Inverted Page Table Architecture



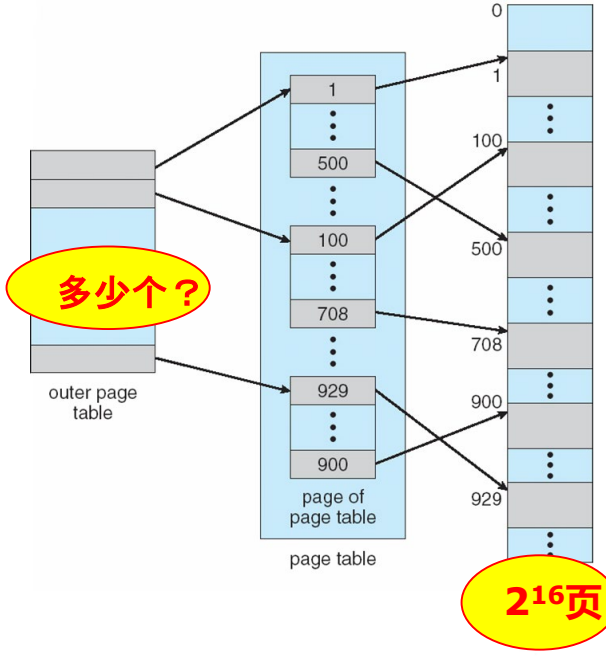
例题

- 某计算机采用二级页表的分页存储管理方式，按字节编址，页大小为2^{10}字节，页表项大小为2字节，逻辑地址结构为：

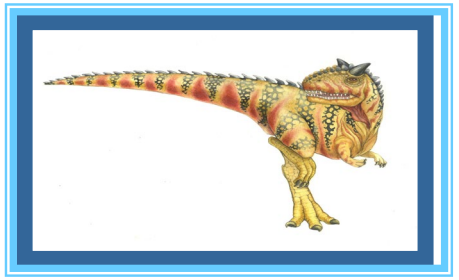
页目录号	页号	页内偏移量

逻辑地址空间大小为2^{16}页，则表示整个逻辑地址空间的页目录表中包含表项的个数至少是

A．64　　B．128　　C．256　　D．512



8.5 Segmentation (分段，段式管理)



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

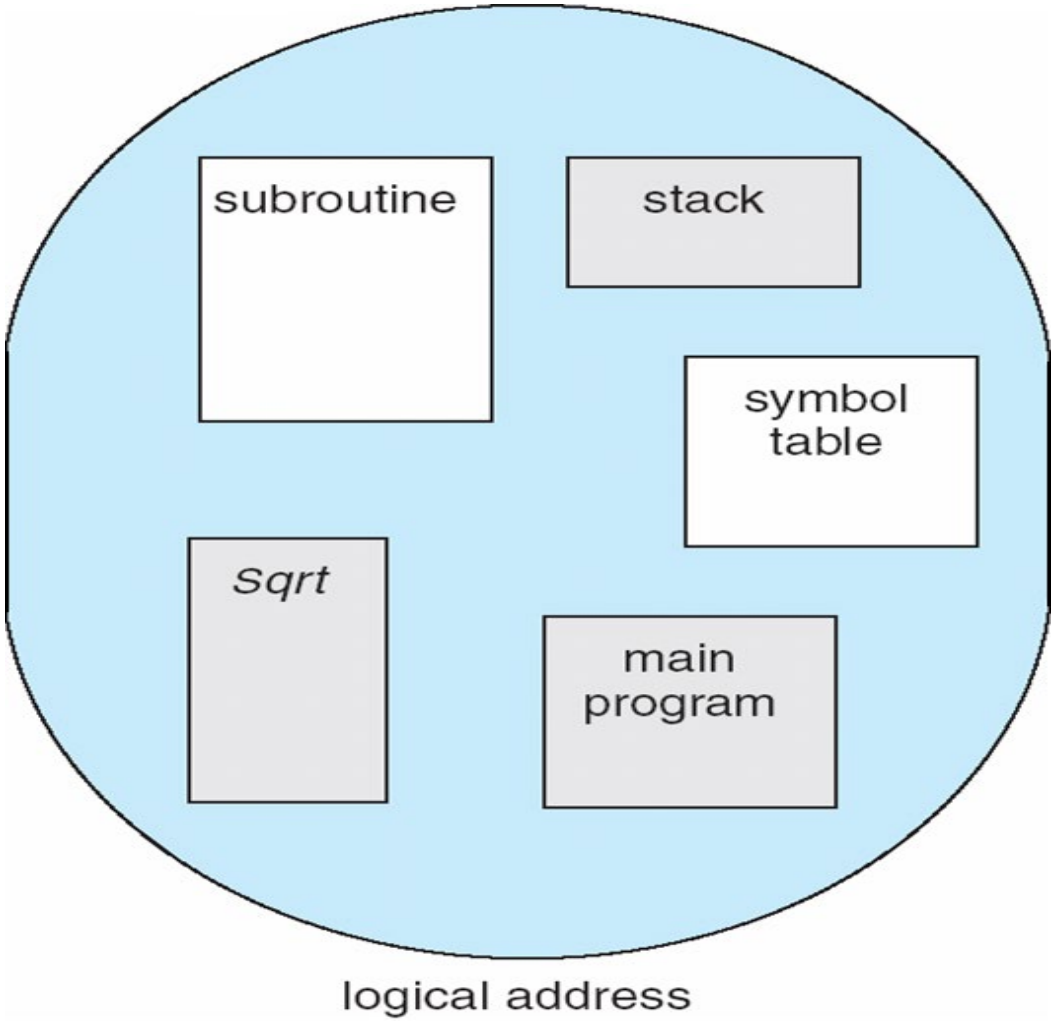
    common block

    stack

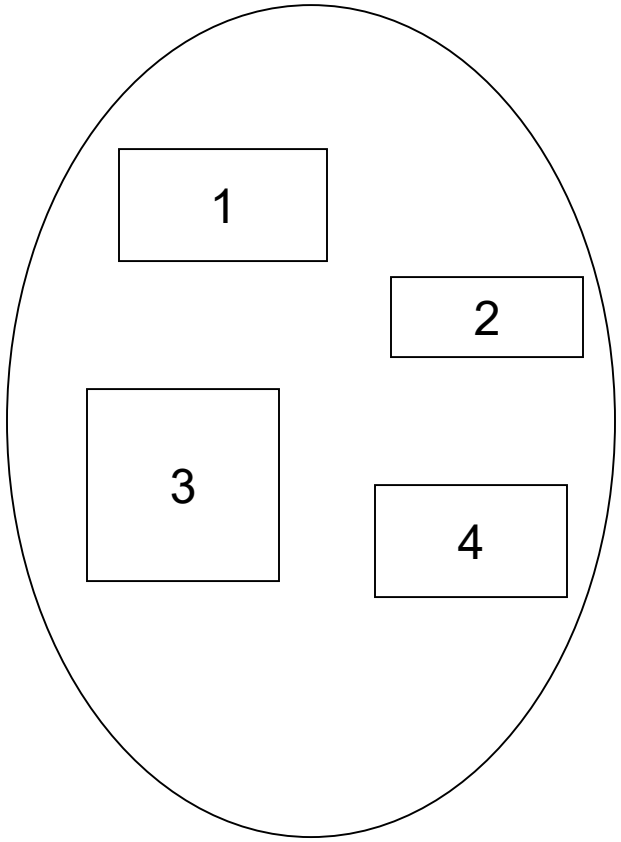
    symbol table

    arrays

User's View of a Program



Logical View of Segmentation



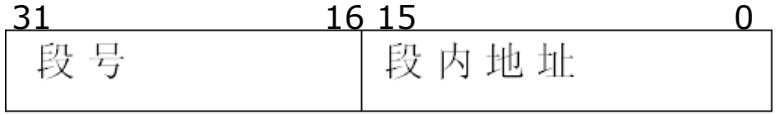
user space

physical memory space

Segmentation Architecture

外碎片

- Logical address consists of a two tuple:
  <segment-number, offset>,



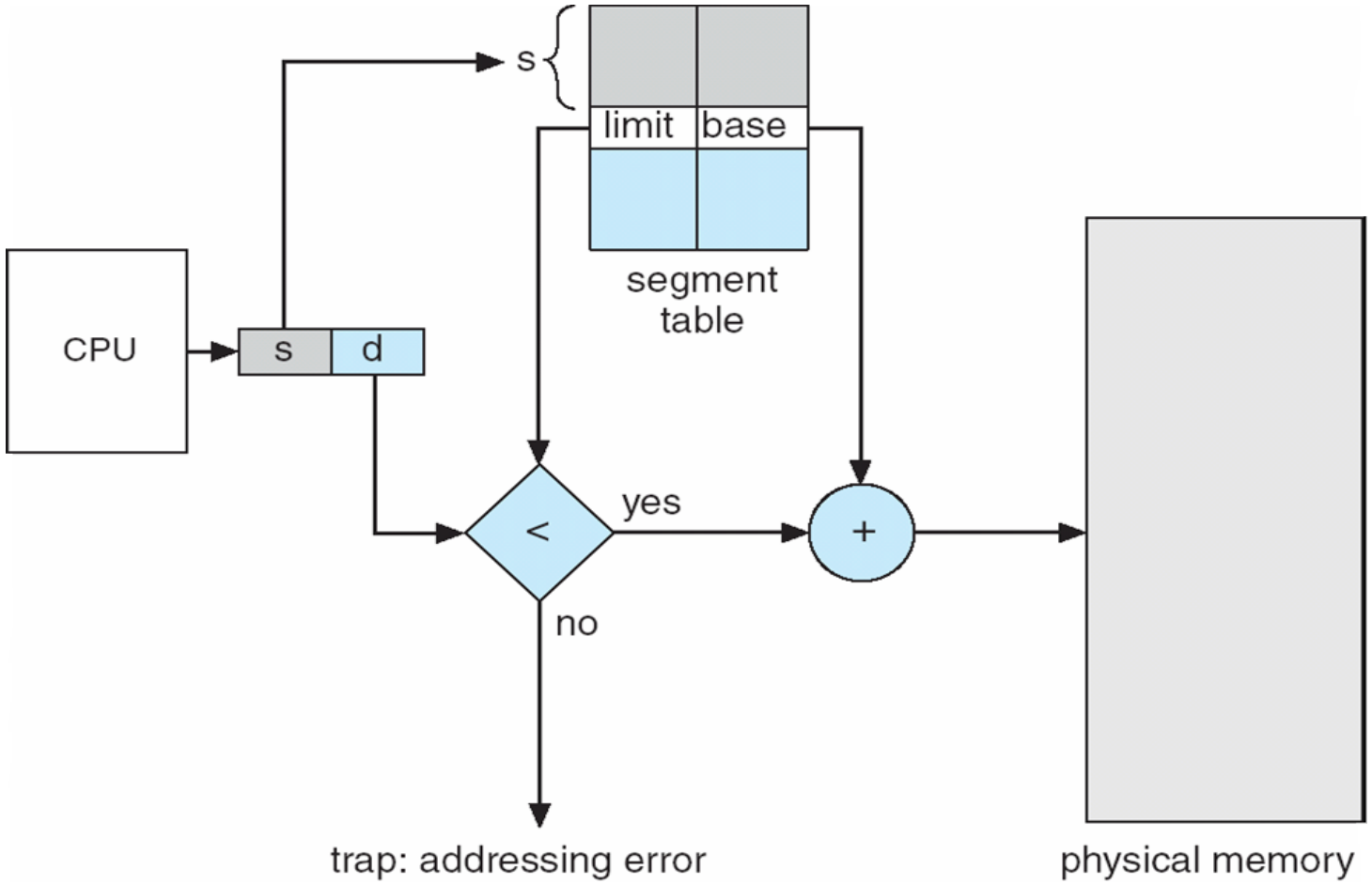
- Segment table（段表）– maps two-dimensional physical addresses; each table entry has:
  - base – contains the starting physical address where the segments reside in memory
  - limit – specifies the length of the segment
- Segment-table base register (STBR)（段表基址寄存器）points to the segment table's location in memory
- Segment-table length register (STLR)（段表限长寄存器）indicates number of segments used by a program;

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

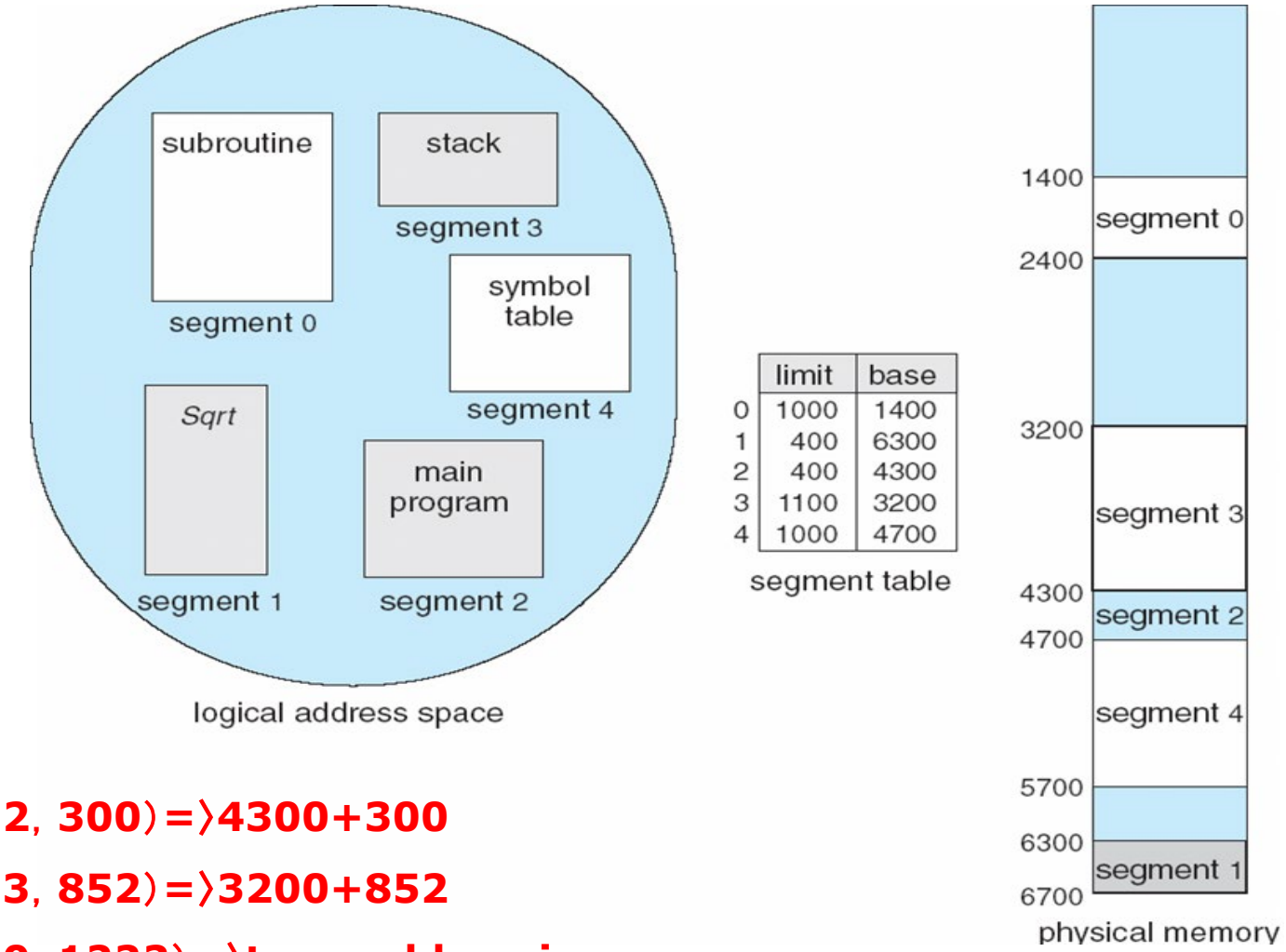
Segmentation Architecture (Cont.)

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



Example of Segmentation

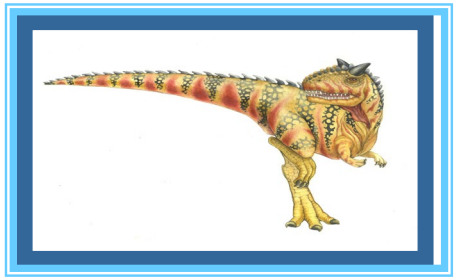


(2, 300)=〉4300+300

(3, 852)=〉3200+852

(0, 1222)=〉trap, addressing error

8.6 Example: The Intel Pentium

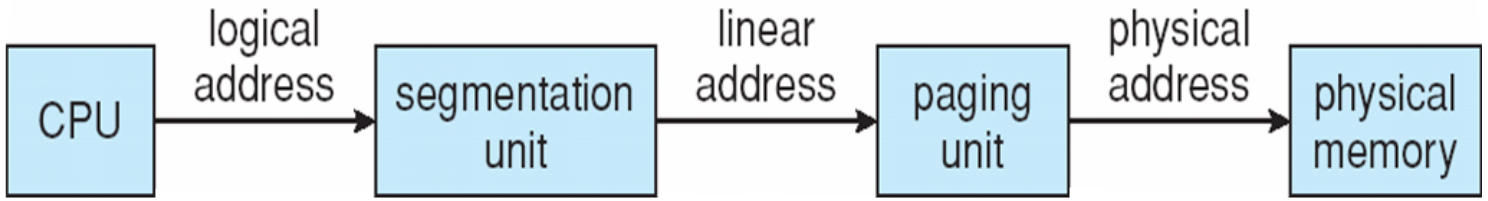


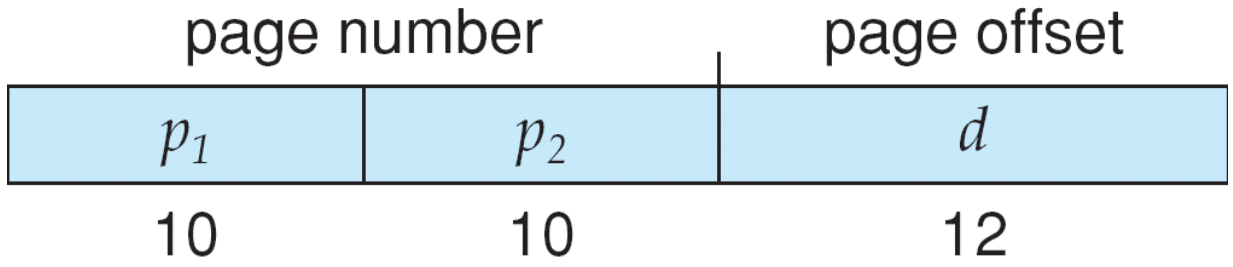
Example: The Intel Pentium

- Supports both segmentation and segmentation with paging
- CPU generates logical address
  - 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

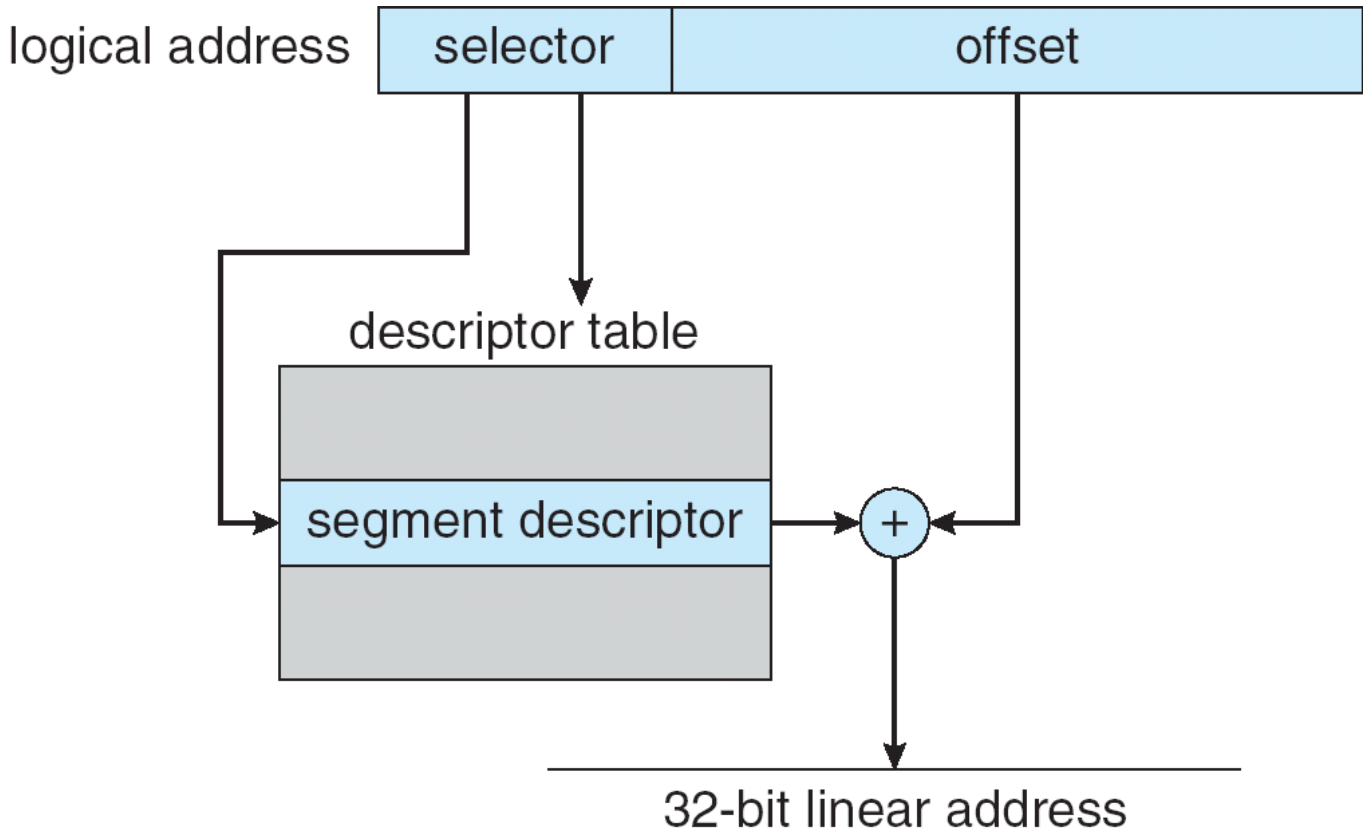
内碎片

Logical to Physical Address Translation in Pentium

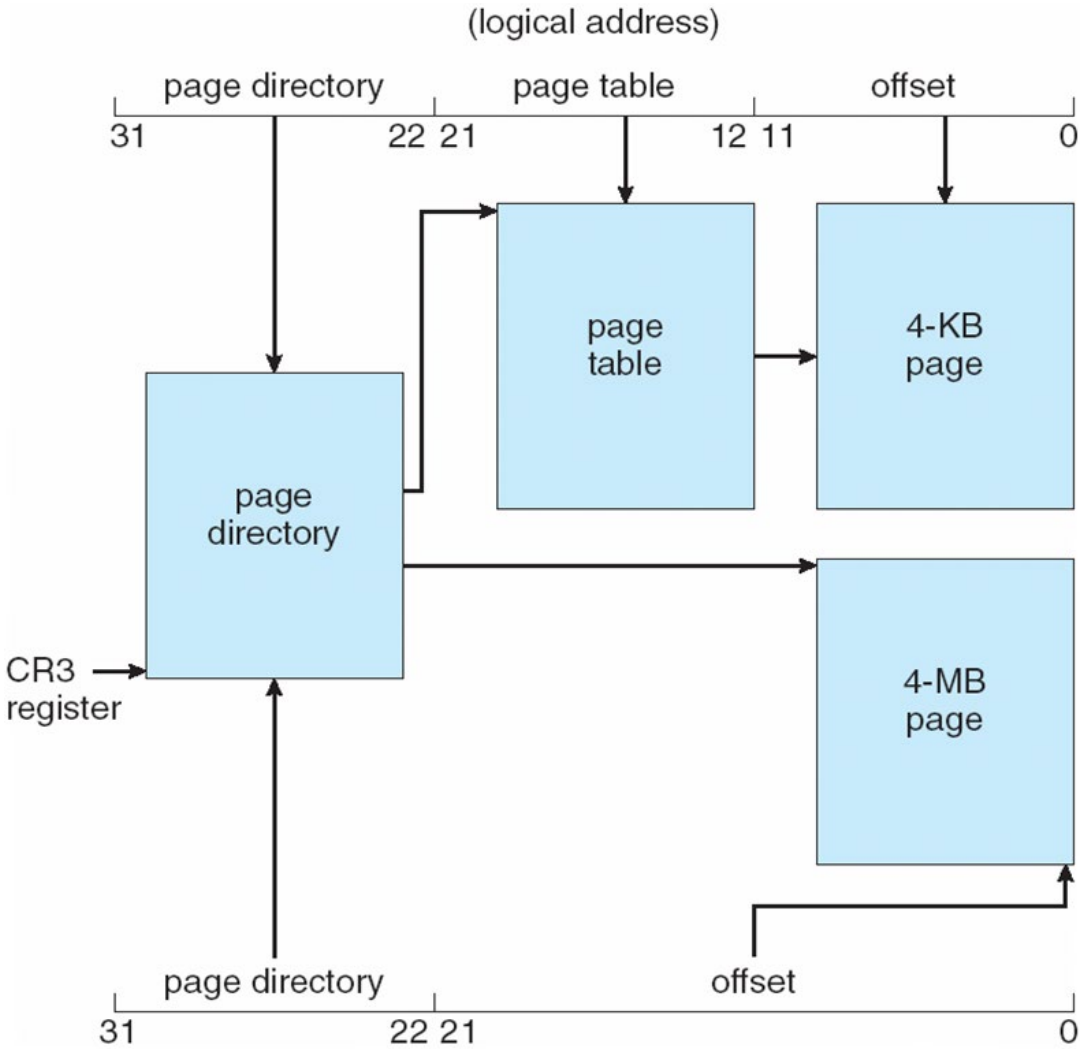




Intel Pentium Segmentation

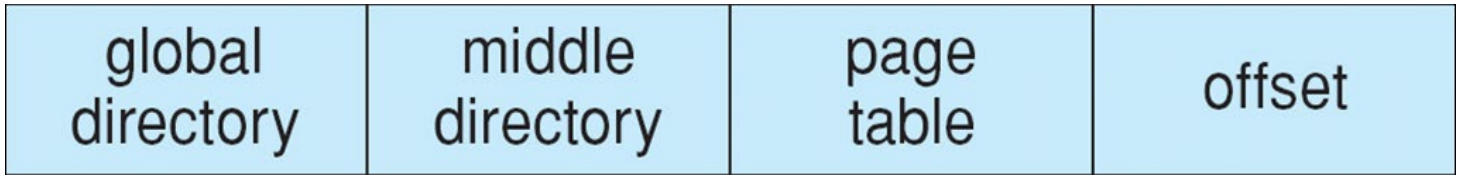


Pentium Paging Architecture

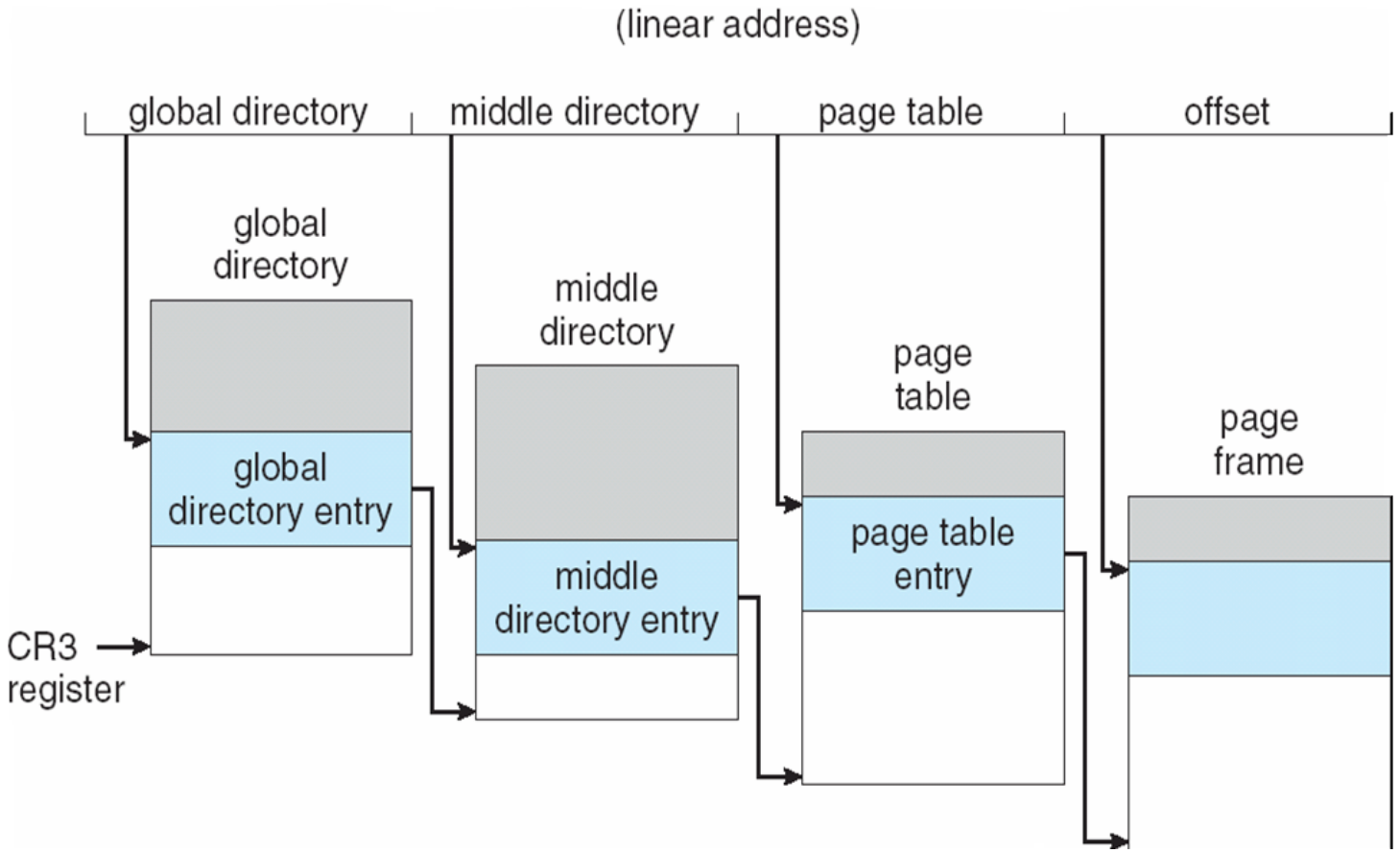


Linear Address in Linux

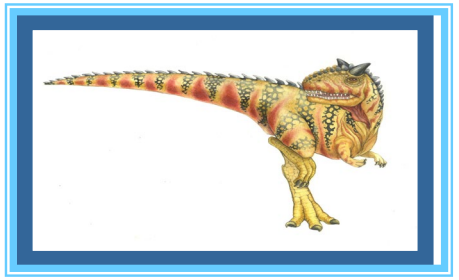
Broken into four parts:



Three-level Paging in Linux



8.8 Summary



8.8 Summary

- logical address, physical address
- contiguous allocation
  - fixed-size partition
  - variable-size partition
    - first-fit, best-fit, worst-fit
- paging
- segmentation
- segmentation with paging

Summary

- Hardware support
  - Address Translation Scheme
- Performance
  - translation look-aside buffer (TLB)
- Fragmentation
  - internal fragmentation：
    - fixed-size partition
    - paging
    - segmentation with paging
  - external fragmentation：
    - variable-size partition
    - segmentation

Summary

- relocation
  - compaction, save external fragmentation
  - require relocation at execution time
- swapping
- sharing
  - require paging or segmentation scheme
- protection
  - protection bits on page table/segment table

本章内容

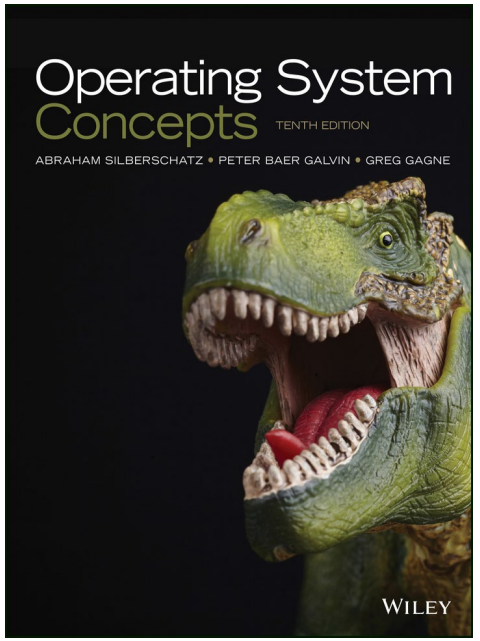
- 内存管理基本概念：源程序的常规处理流程，地址绑定，逻辑地址空间与物理地址空间；内存保护；MMU，动态加载，动态链接，交换，地址管理模型，模型指标。
- 交换与覆盖。
- 连续分配管理方式：单一连续分配算法，基地址寄存器，界限寄存器；动态分区管理，动态存储分配算法，外部碎片，内部碎片。
- 分页管理方式：页，页帧，页表，地址映射流程，硬件支持，页表实现，TLB，有效访问时间，页式管理的模型指标分析，页表结构，多级页表，哈希页表，反向页表。

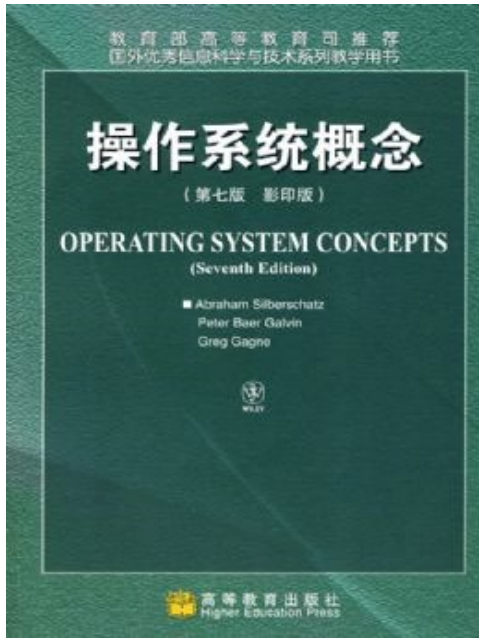
Homework

- 学在浙大

Reading Assignments

- Read for this week:
  - Chapters 8
    of the text book:
- Read for next week:
  - Chapters 9
    of the text book:





End of Chapter 8