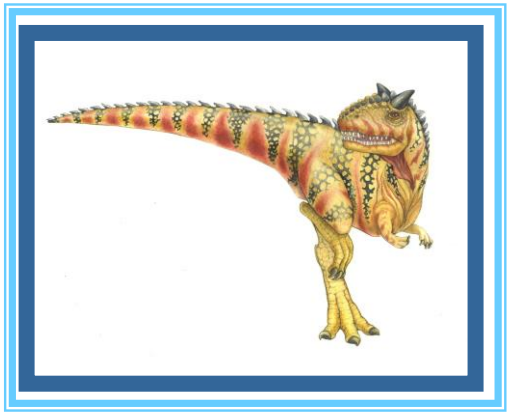
Chapter 8: Main Memory



Chapter 8: Memory Management

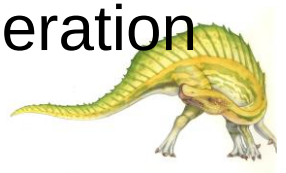
- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table
- Segmentation with Paging
- Example: The Intel Pentium,
- Example: ARMv8 Architecture

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

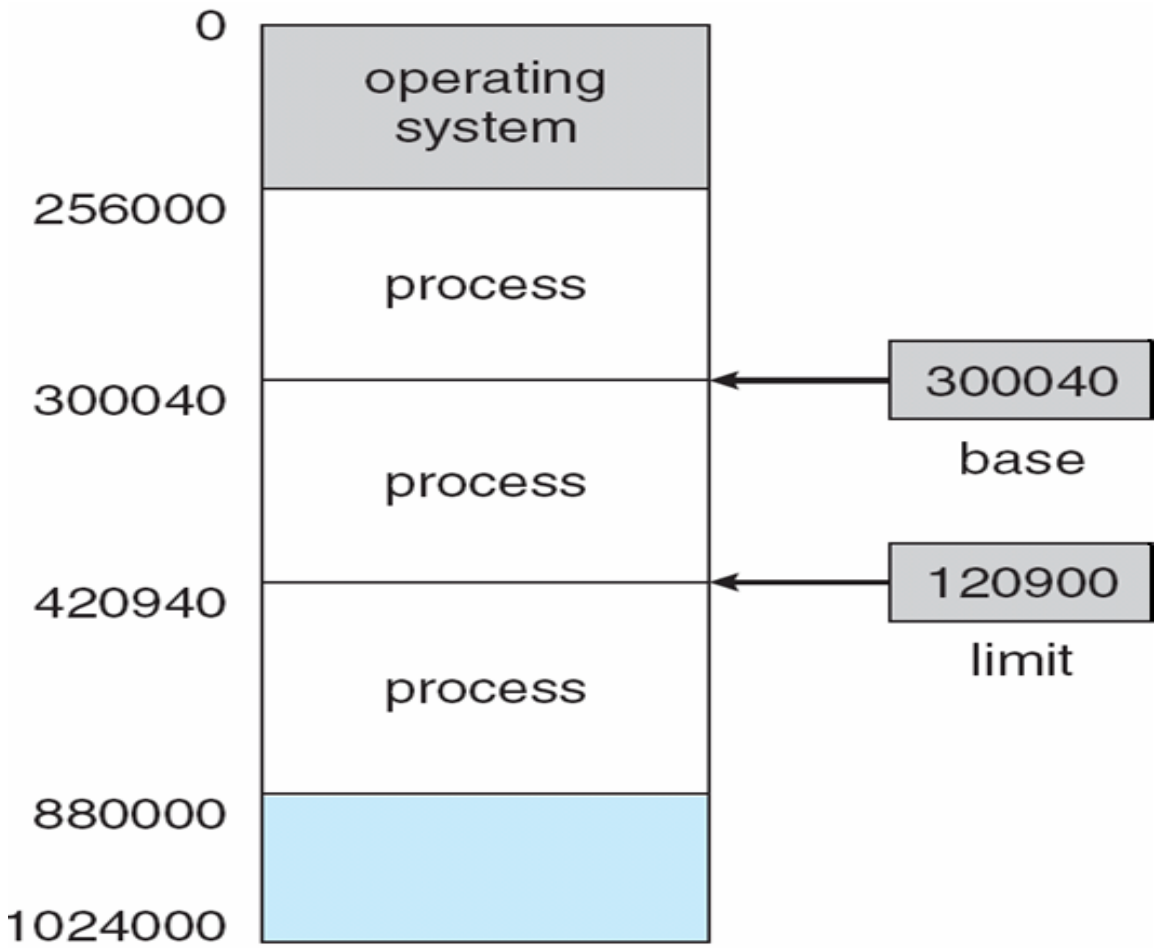
Background

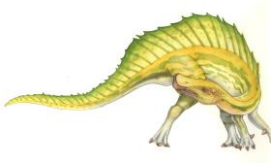
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Register access in one CPU clock (or less)
- Main memory can take many cycles
- **Cache** sits between main memory and CPU registers
-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



Base and Limit Registers

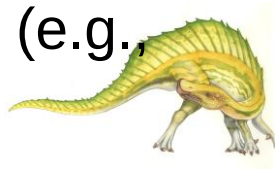
A pair of **base** and **limit** registers define the logical address space



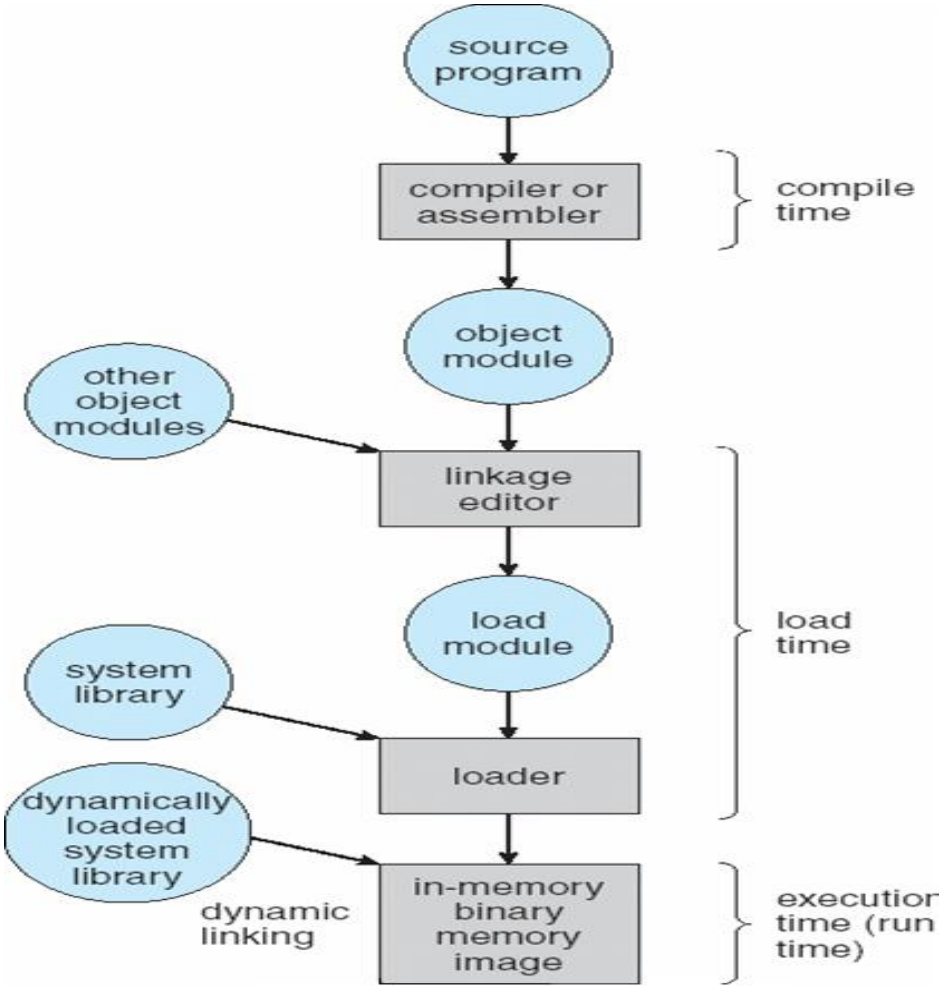


Binding of Instructions and Data to Memory

- 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**: 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



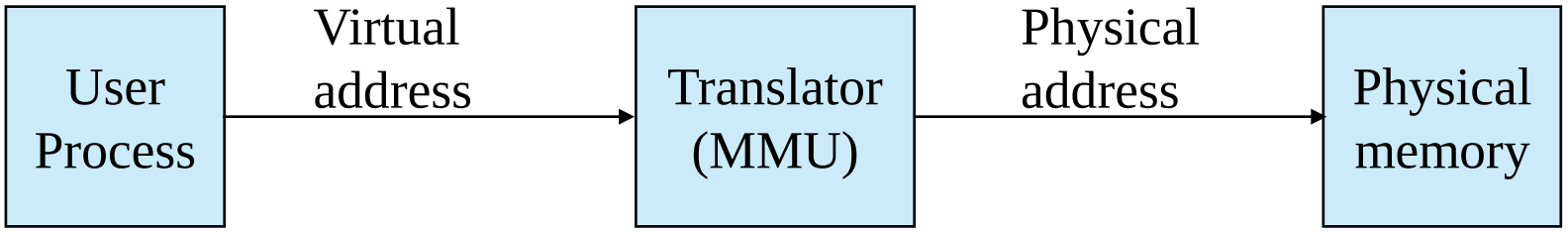
Logical vs. Physical Address Space

- 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

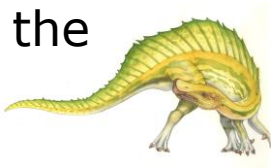
Dynamic Address Translation

- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- Virtual address:
  - An address viewed by the user process
  - The abstraction provided by the OS
- Physical address

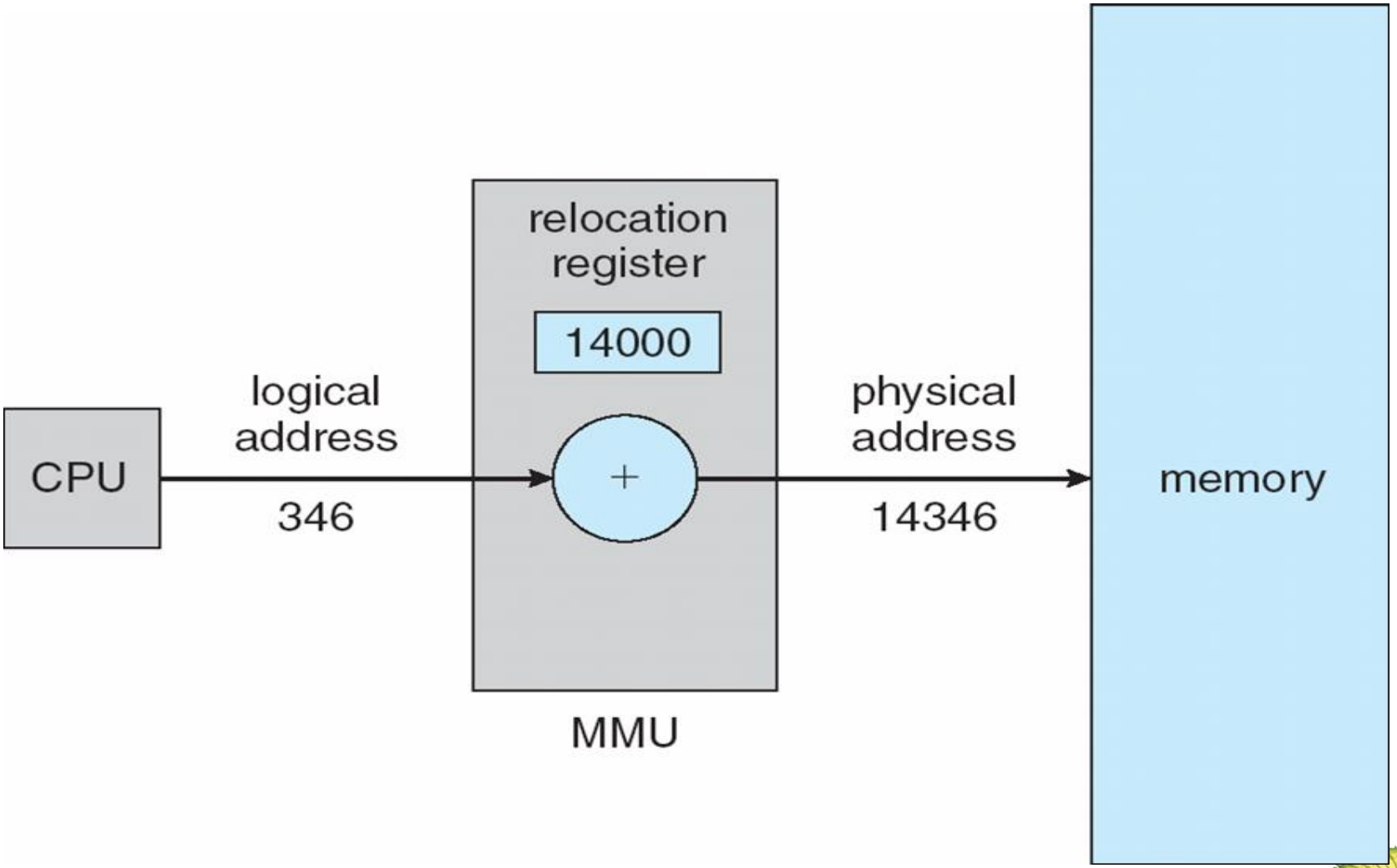
  An address viewed by the physical memory



The user program deals with *logical* addresses; it never sees the *real* physical addresses

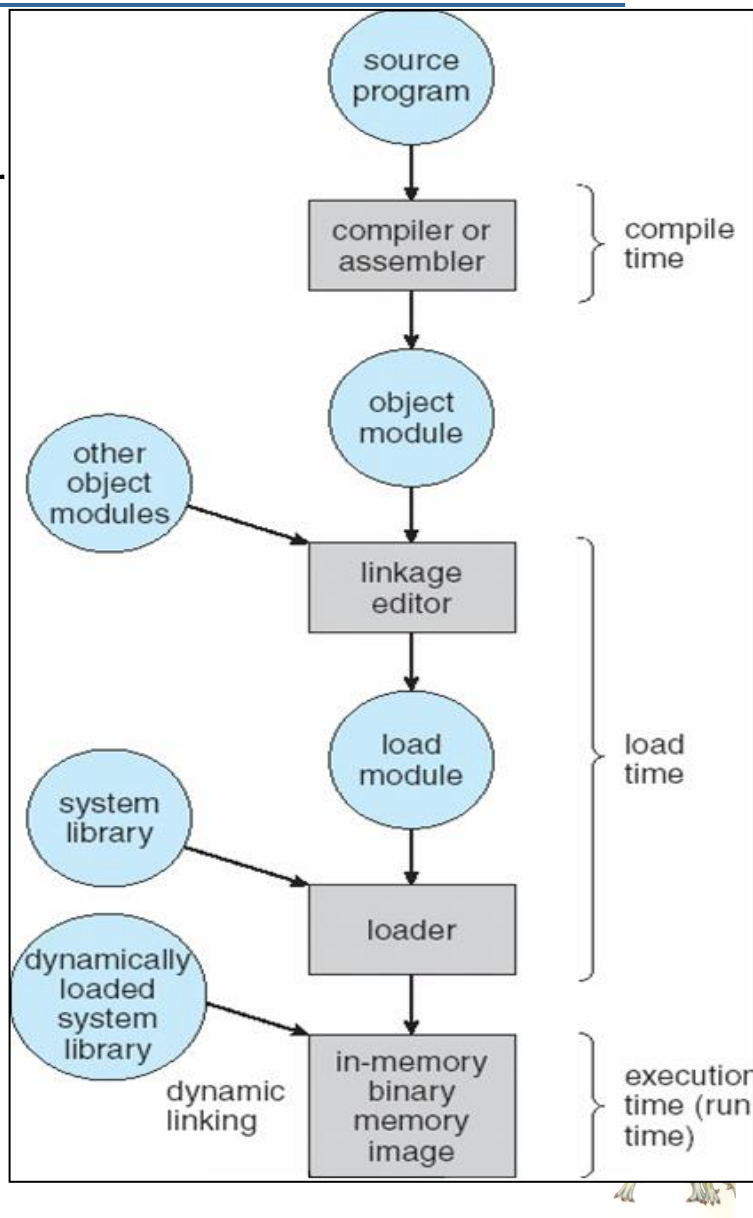


Dynamic relocation using a relocation register



Dynamic Loading

- Routine is not loaded until it is called, Better memory-space utilization; unused routine is never loaded
- Useful when large amounts of code are needed to handle infrequently occurring cases（代码量大使用频率低）
- No special support from the operating system is required implemented through program design



Dynamic Linking

- Linking postponed until execution time
- 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**

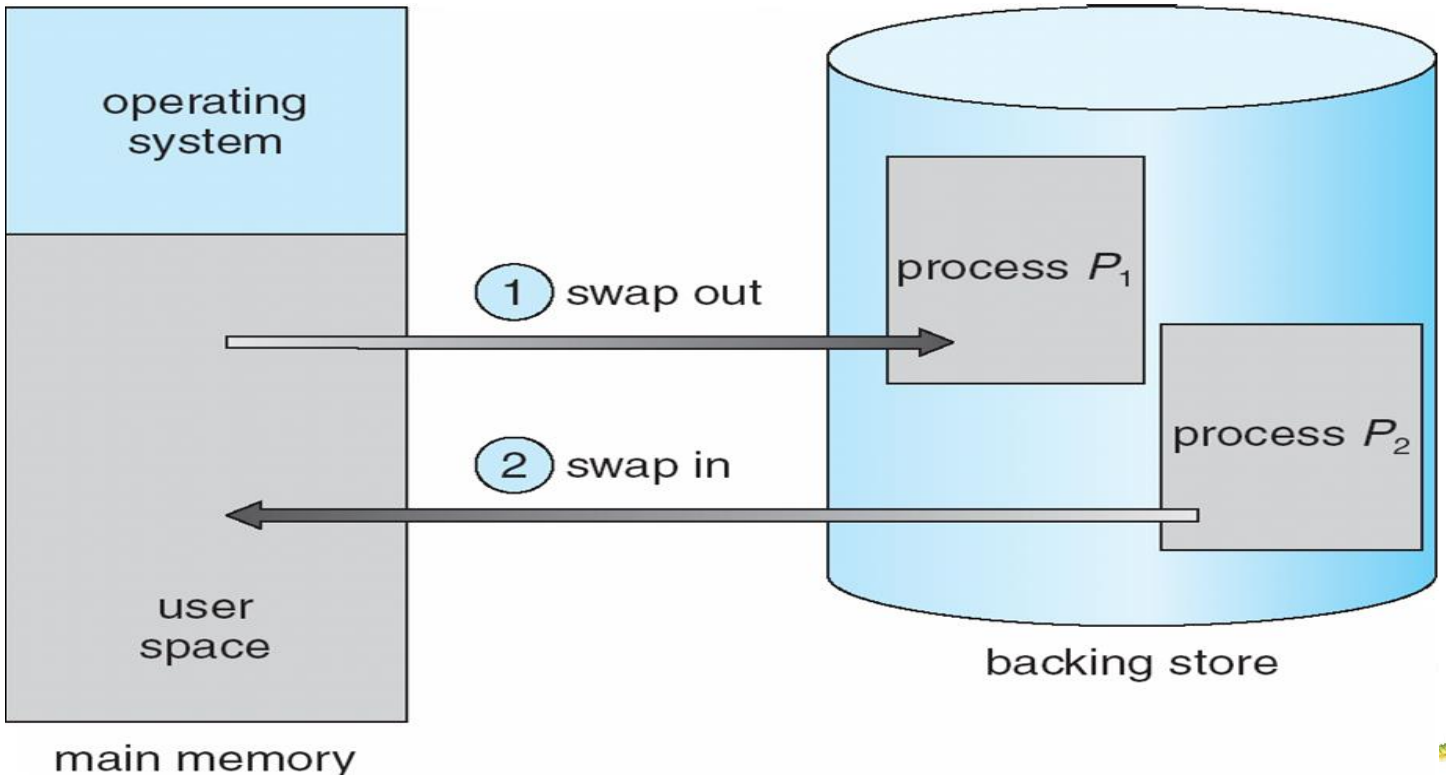
Swapping

- 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
- **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

Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)

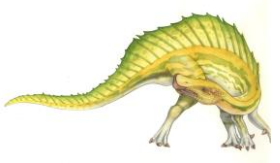
Schematic View of Swapping

- 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

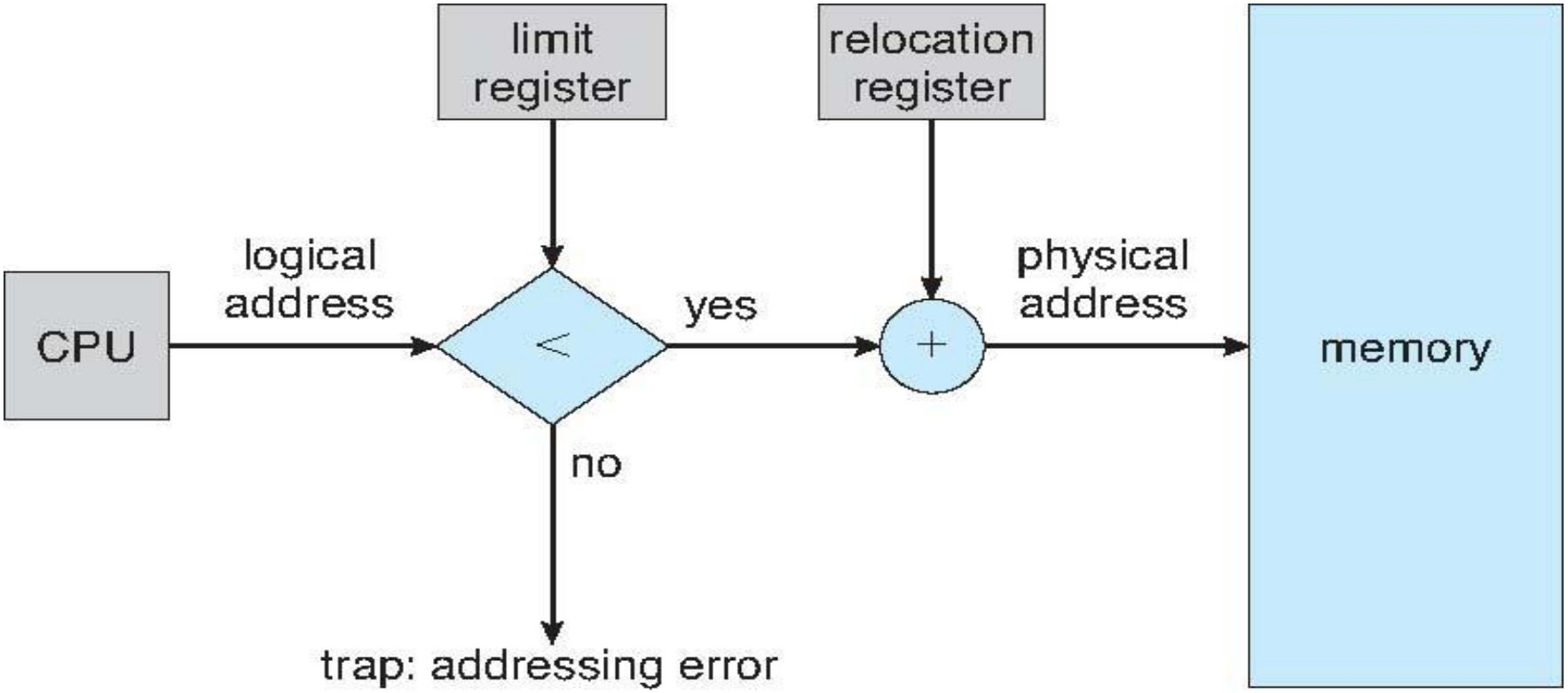


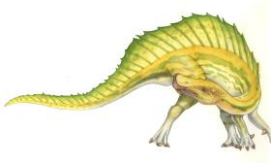
Contiguous Allocation

- Main memory usually into two partitions:
  - 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*



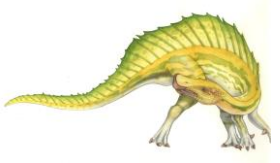
Hardware Support for Relocation and Limit Registers





Contiguous Allocation

- Fixed Partitioning
- Dynamic Partitioning
- Dynamic Relocationable Partitioning



Fixed Partitioning

Partition table

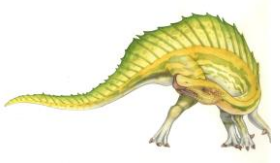
No.	size	Beginning address	Free or not
1	16K	20K	0
2	32K	36K	0
3	64K	68K	0
4	124K	132K	1

memory

- Divide memory into a number of fixed-sized partitions. Each partition may contain exactly one process.
- The degree of multiprogramming is bound by the number of partitions.
- The OS keeps a table indicating which parts of memory are available and which are occupied.

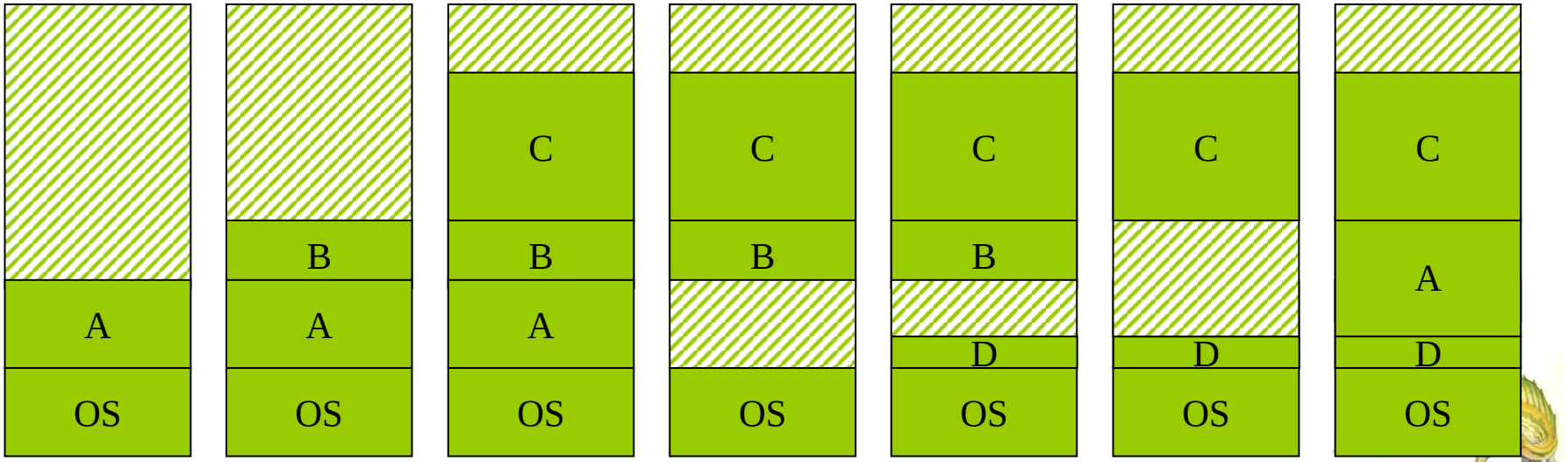
Fixed Partitioning

- It is easy to implement
- disadvantages:
  - Fixed partition size: internal fragmentation
  - Fixed partition number: The degree of multiprogramming is bound



Contiguous Allocation (Cont)

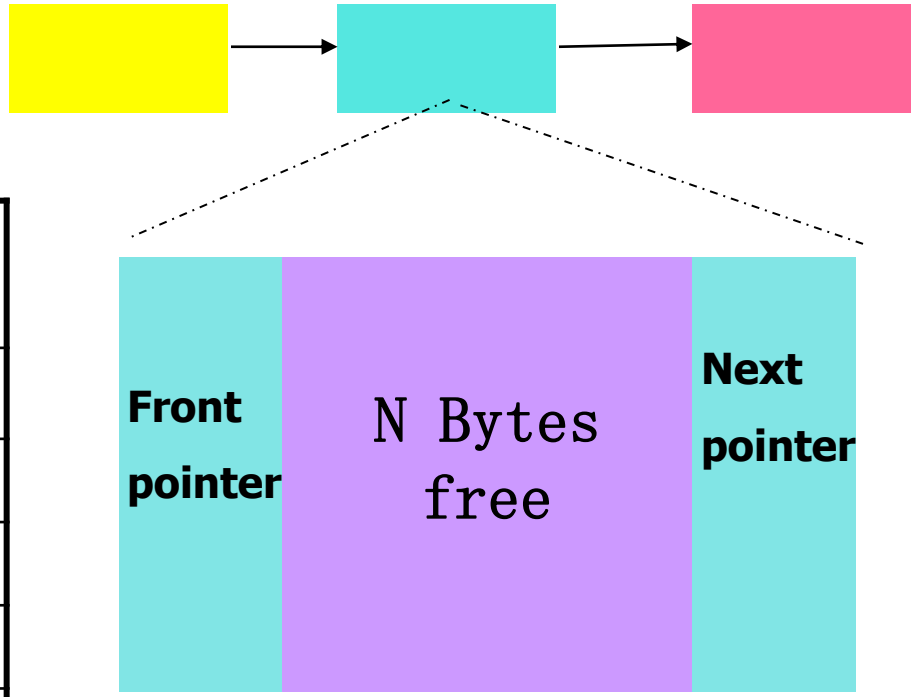
- Multiple-partition allocation
  - Hole – block of available memory; holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (hole)



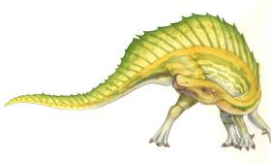
Partition management

No.	size	Beginning address	state
1	48K	116K	free
2	252K	260K	free
3	---	---	----
4	---	---	----
5	---	---	----

Free Partition table



Free partition list



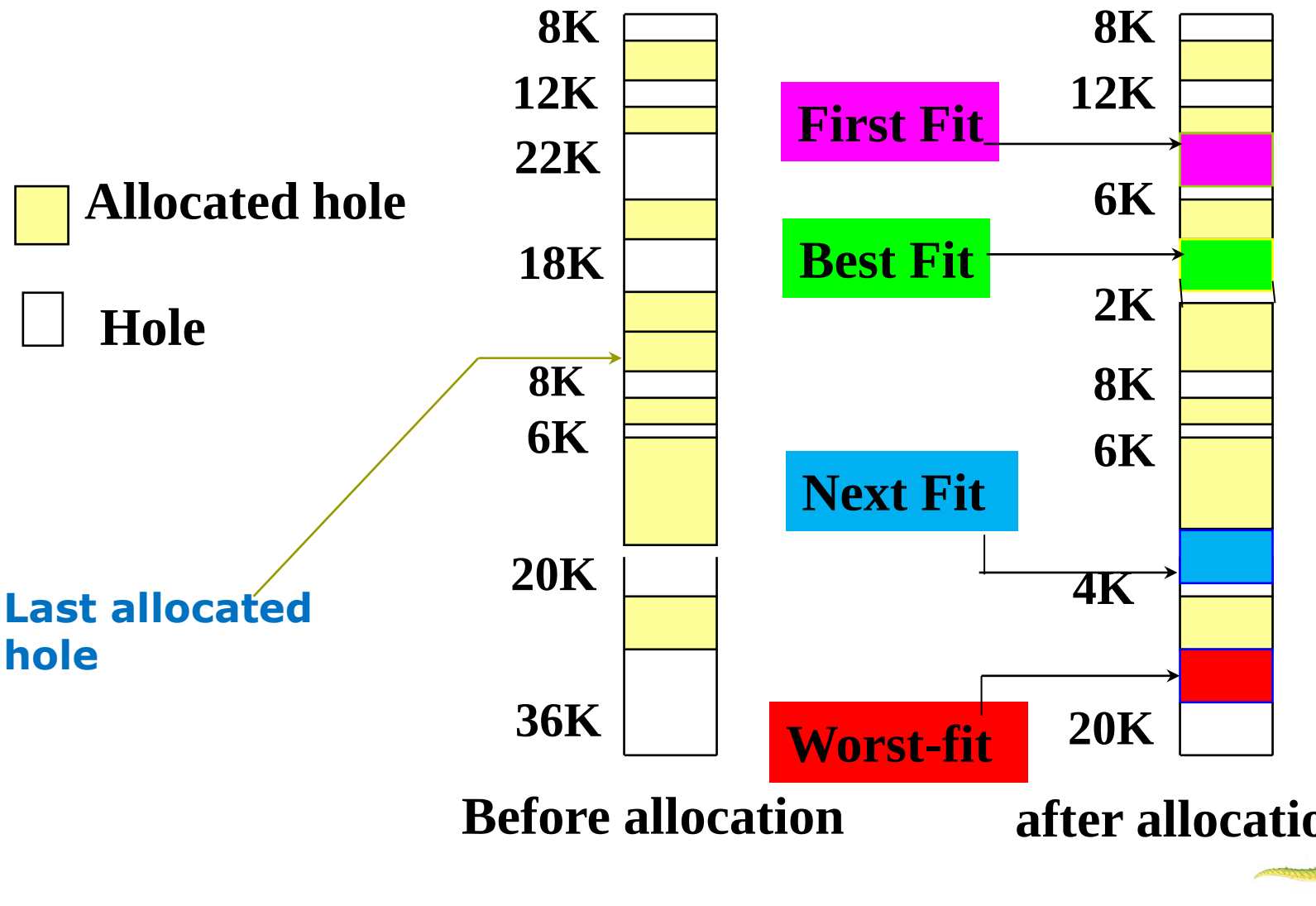
Dynamic Storage-Allocation Problem

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

- **First-fit**: Allocate the *first* hole that is big enough(**Next-fit**: Allocate a next hole that is big enough since the last time to have allocated hole)
- **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

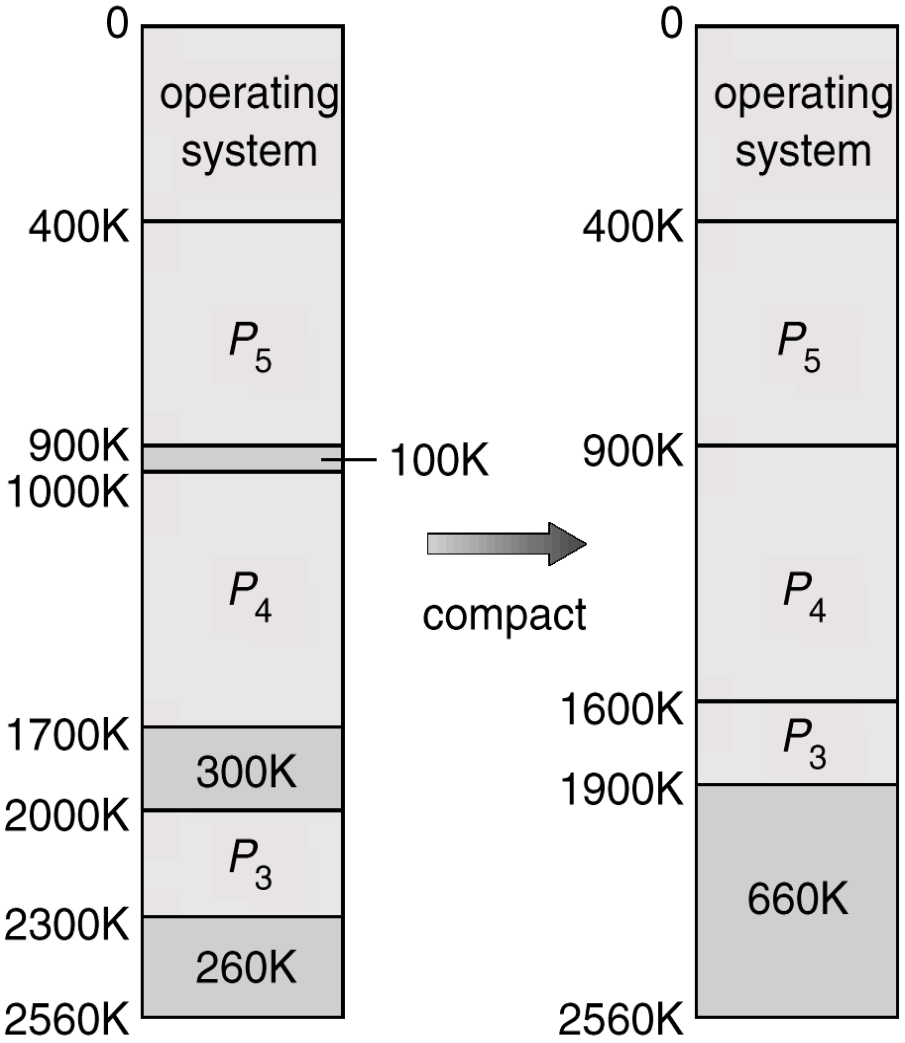
For example: Requests a 16 KB hole



Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used

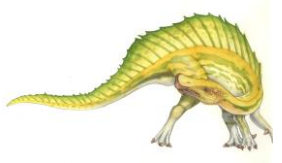
Reduce external fragmentation by compaction(紧凑)



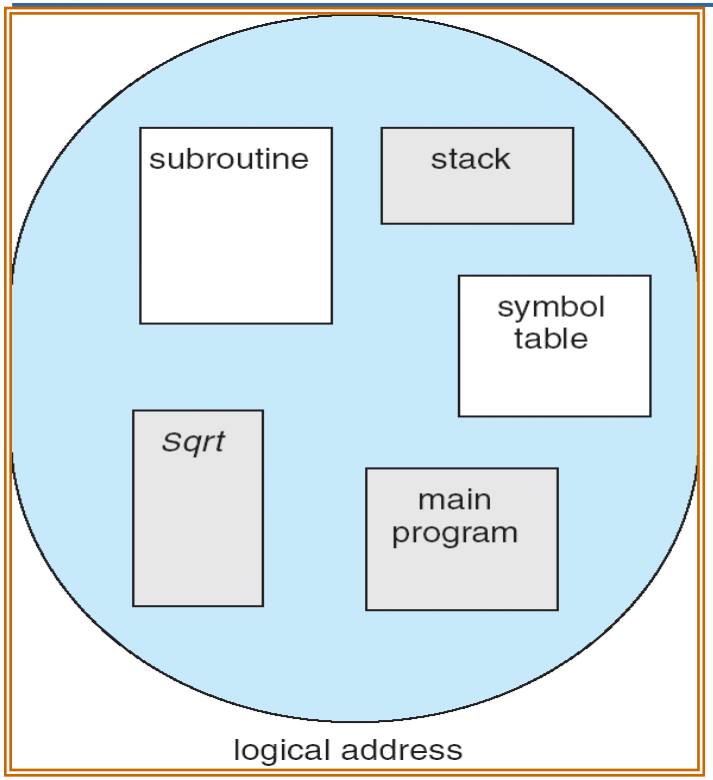
- Shuffle memory contents to place all free memory together in one large block
- Compaction is possible *only* if relocation is dynamic, and is done at execution time
- I/O problem:
  - Latch job in memory while it is involved in I/O
  - Do I/O only into OS buffers

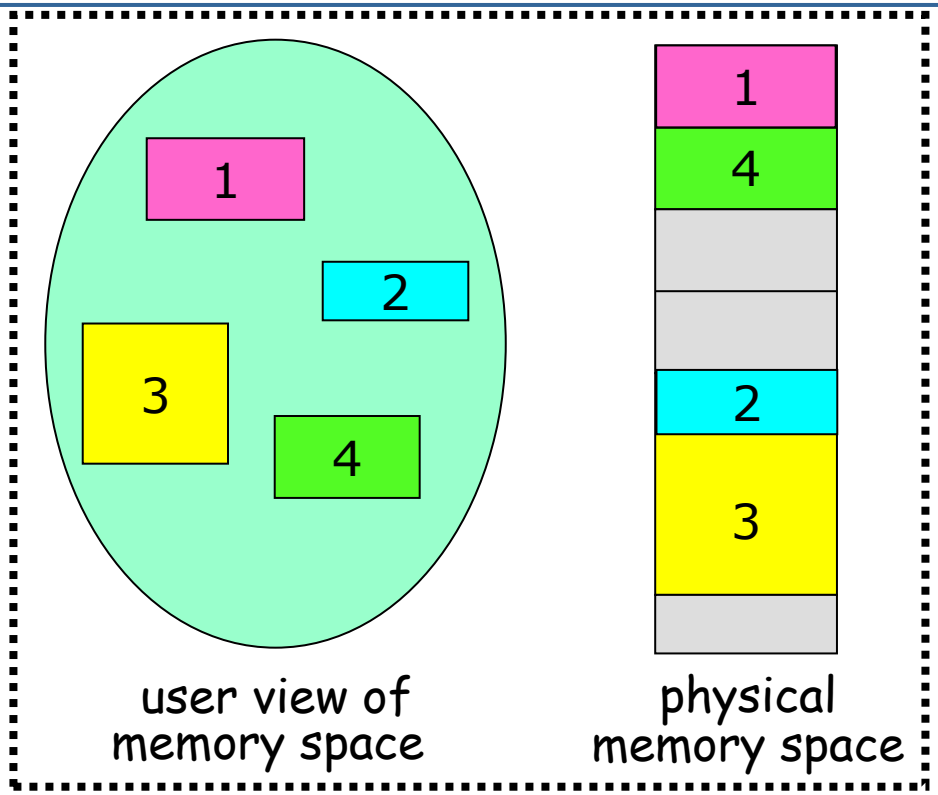
Segmentation

- 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



More Flexible Segmentation





- Logical View: multiple separate segments
  - Typical: Code, Data, Stack
  - Others: memory sharing, etc
- Each segment is given region of contiguous memory
  - Has a base and limit
  - Can reside anywhere in physical memory

Segmentation Architecture

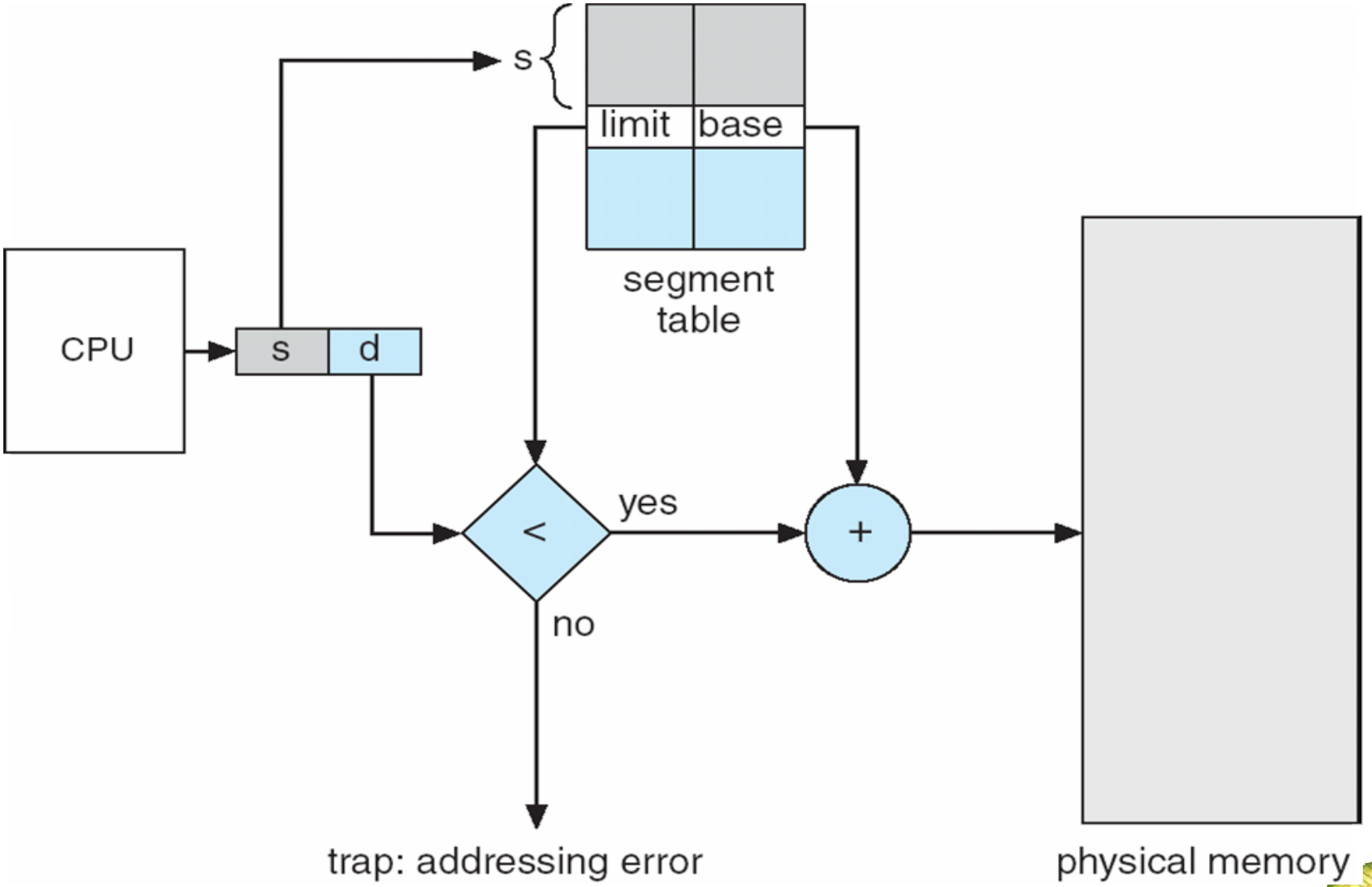
- Logical address consists of a two tuple:

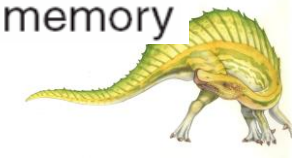
  <segment-number, offset>,

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

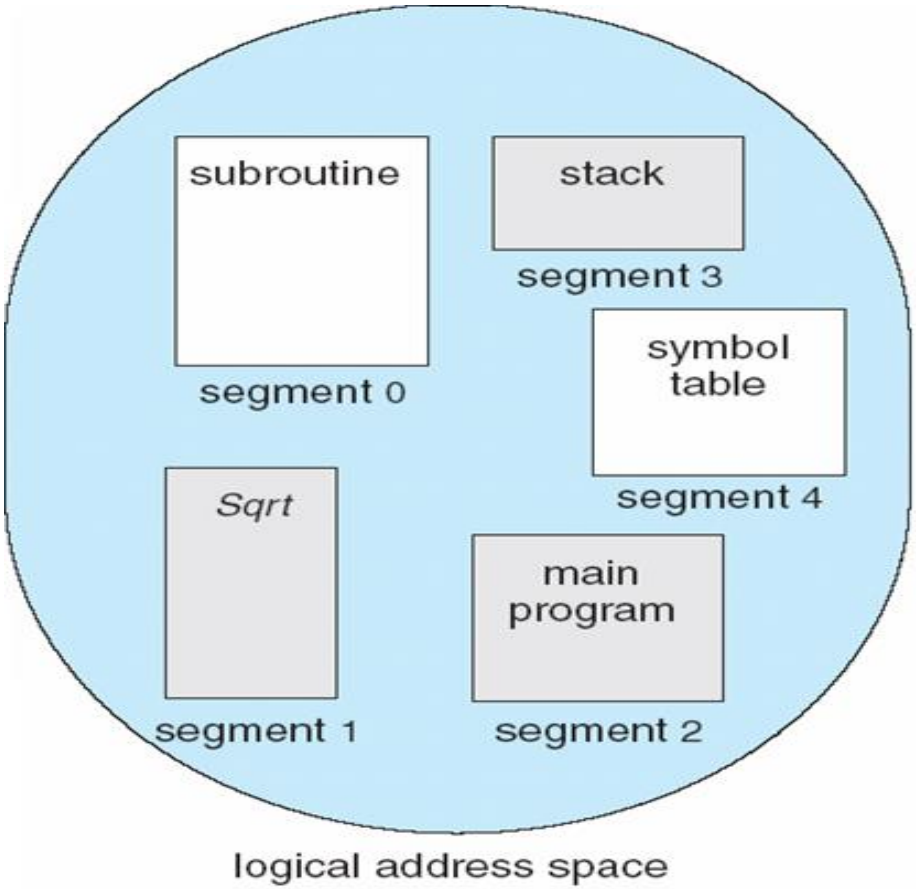
  segment number ***s*** is legal if ***s*** < **STLR**

Segmentation Hardware



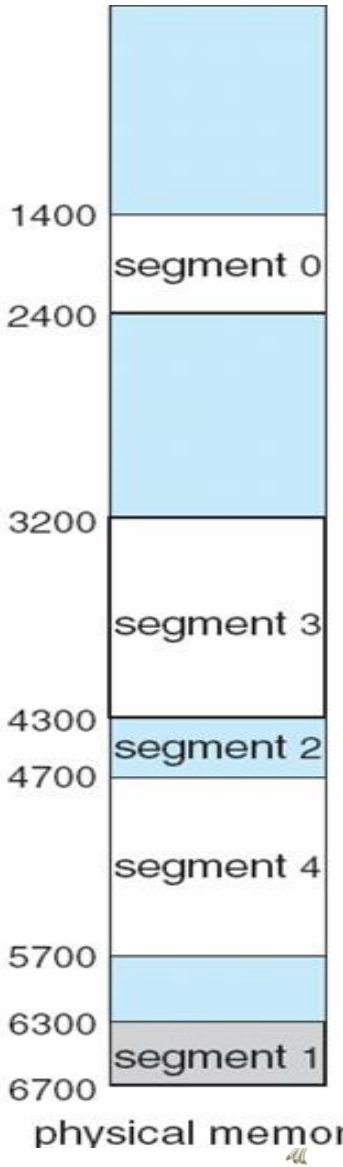


Example of Segmentation

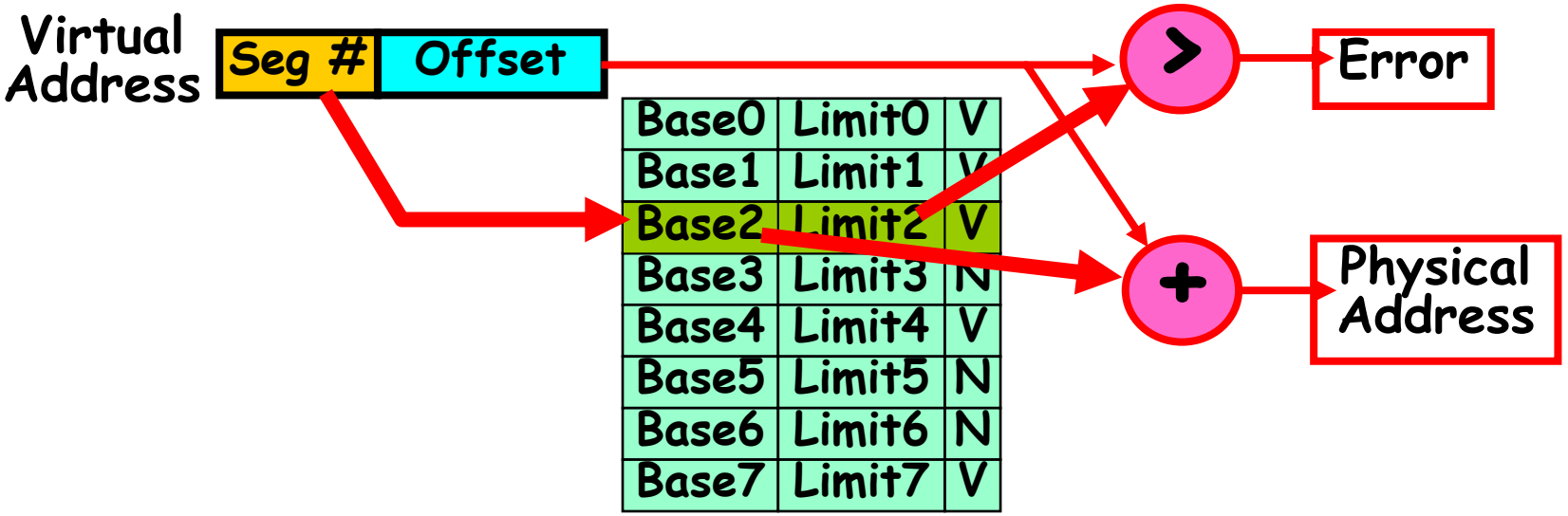


	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table



Implementation of Multi-Segment Model

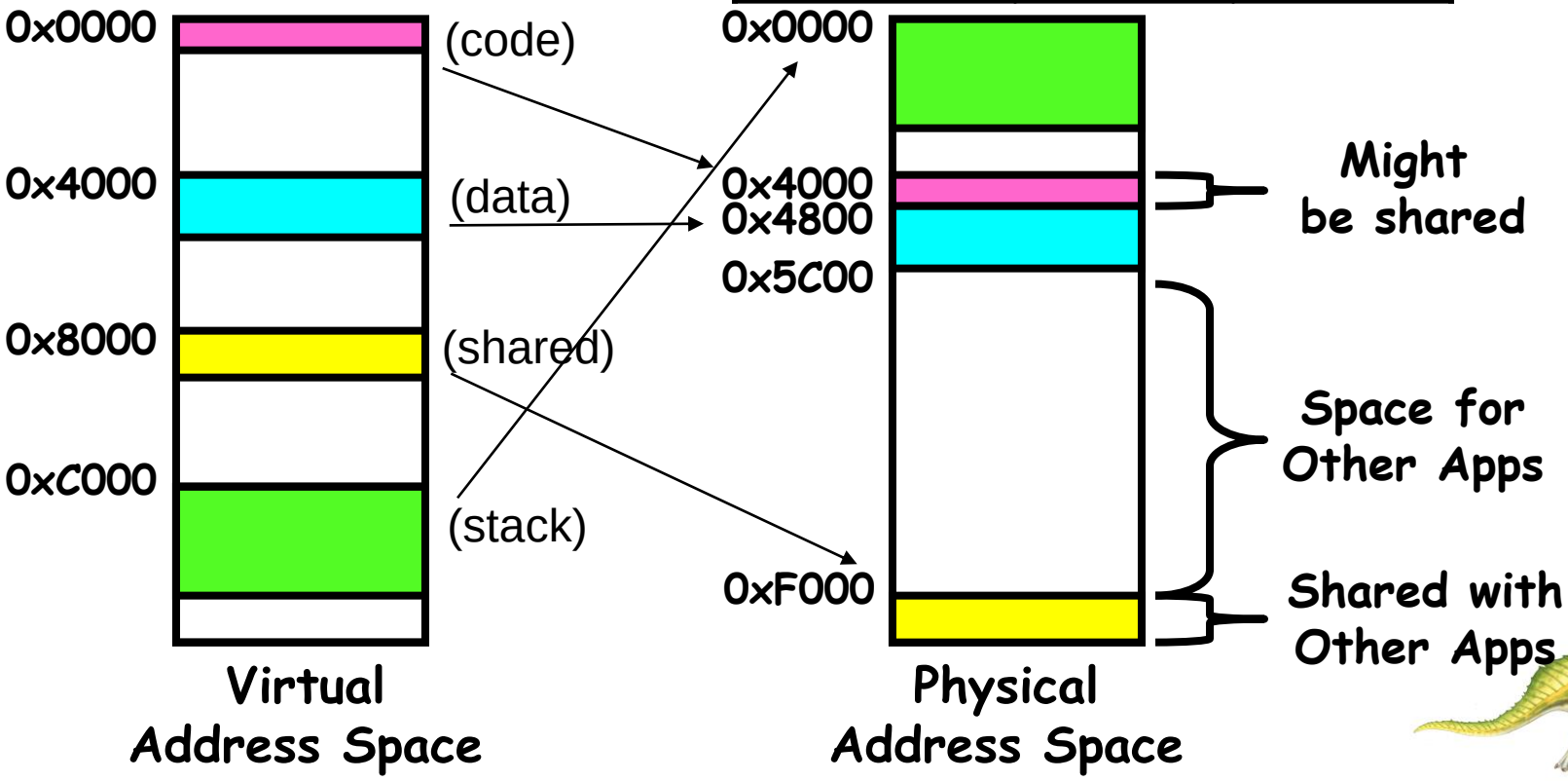


- Segment map resides in processor
  - Segment number mapped into base/limit pair
  - Base added to offset to generate physical address
  - Error check catches offset out of range
- As many chunks of physical memory as entries
  - Segment addressed by portion of virtual address
- What is “V/N”?
  - Can mark segments as invalid; requires check as well

Example: Four Segments (16 bit addr)

Virtual Address Format

Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000



Example of segment translation

```
0x240   main:   la $a0, varx
0x244           jal strlen
  …                 …
0x360   strlen: li   $v0, 0   ;count
0x364   loop:   lb   $t0, ($a0)
0x368           beq  $r0,$t1, done
  …                 …
0x4050  varx    dw   0x314159//data
```

Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data):4000	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000

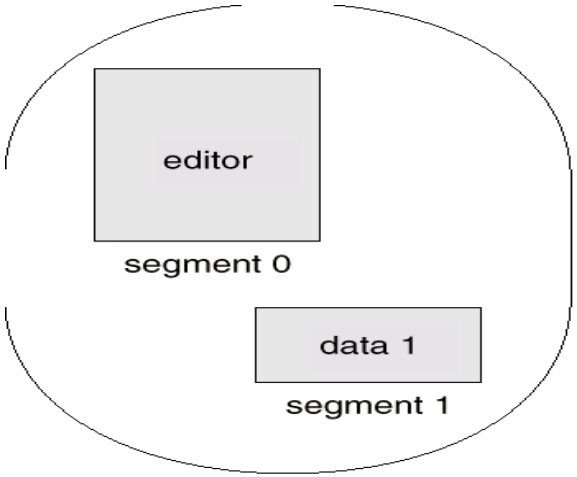
Let's simulate a bit of this code to see what happens (PC=0x240)://MIPS指令集

1. Fetch 0x240. Virtual segment #? 0; Offset? 0x240
   Physical address? Base=0x4000, so physical addr=0x4240
   Fetch instruction at 0x4240. Get "la $a0, varx"//varx地址0x4050送a0
   Move 0x4050 → $a0, Move PC+4→PC
2. Fetch 0x244.Translated to Physical=0x4244. Get"jal strlen"//跳转,$ra存返回地址0x248;
   Move 0x248 → $ra (return address!), Move 0x360 → PC)
3. Fetch 0x360. Translated to Physical=0x4360. Get "li $v0,0"
   Move 0x0000 → $v0, Move PC+4→PC
4. Fetch 0x364. Translated to Physical=0x4364. Get "lb $t0,($a0)"// 从$a0取一个字节0x50
   Since $a0 is 0x4050, try to load byte from 0x4050
   Translate 0x4050. Virtual segment #? 1; Offset? 0x50
   Physical address? Base=0x4800, Physical addr = 0x4850,
   Load Byte from 0x4850→$t0, Move PC+4→PC

Segmentation Protection

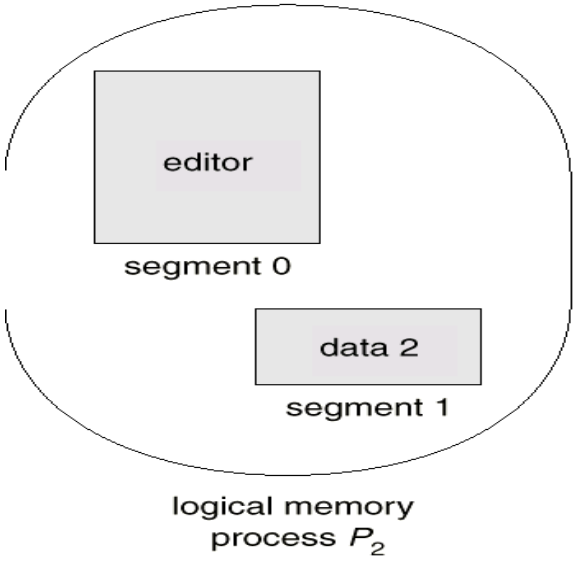
- Protection
  - With each entry in segment table associate:
    - validation bit = 0 $\Rightarrow$ illegal segment
    - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level

Segmentation Sharing



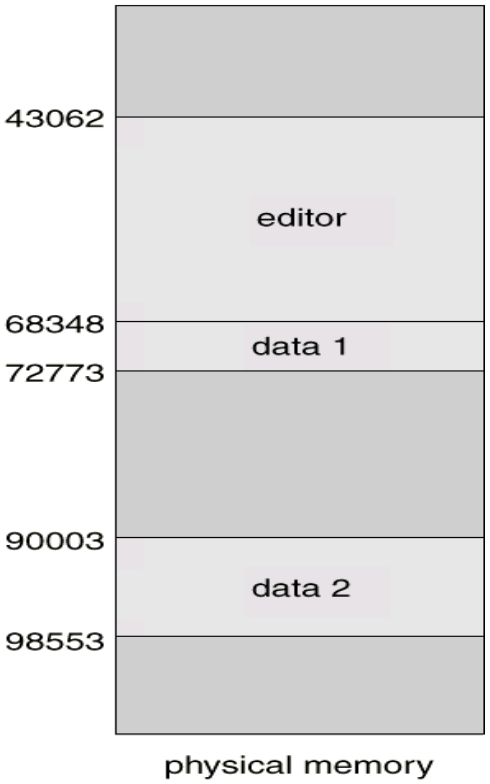
	limit	base
0	25286	43062
1	4425	68348

segment table
process P_1



	limit	base
0	25286	43062
1	8850	90003

segment table
process P_2

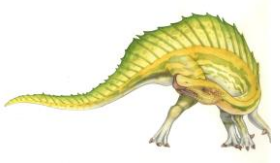


Observations about Segmentation

- Relocation.
  - Since segments vary in length, memory allocation is a dynamic storage-allocation problem
  - by segment table
- Sharing.
  - shared segments: code segment, data segment
  - code segment : same segment number
- Allocation.
  - first fit/best fit

Paging: Physical Memory in Fixed Size Chunks

- Problems with segmentation?
  - Must fit variable-sized chunks into physical memory
  - May move processes multiple times to fit everything
  - Limited options for swapping to disk
- Fragmentation: wasted space
  - External: free gaps between allocated chunks
  - Internal: don't need all memory within allocated chunks



Paging: Physical Memory in Fixed Size Chunks

- Solution to fragmentation from segments?
  - Divide physical memory into fixed-sized blocks called frames (size is power of 2, between 512 bytes and 8,192 bytes)
  - 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
  - Internal fragmentation

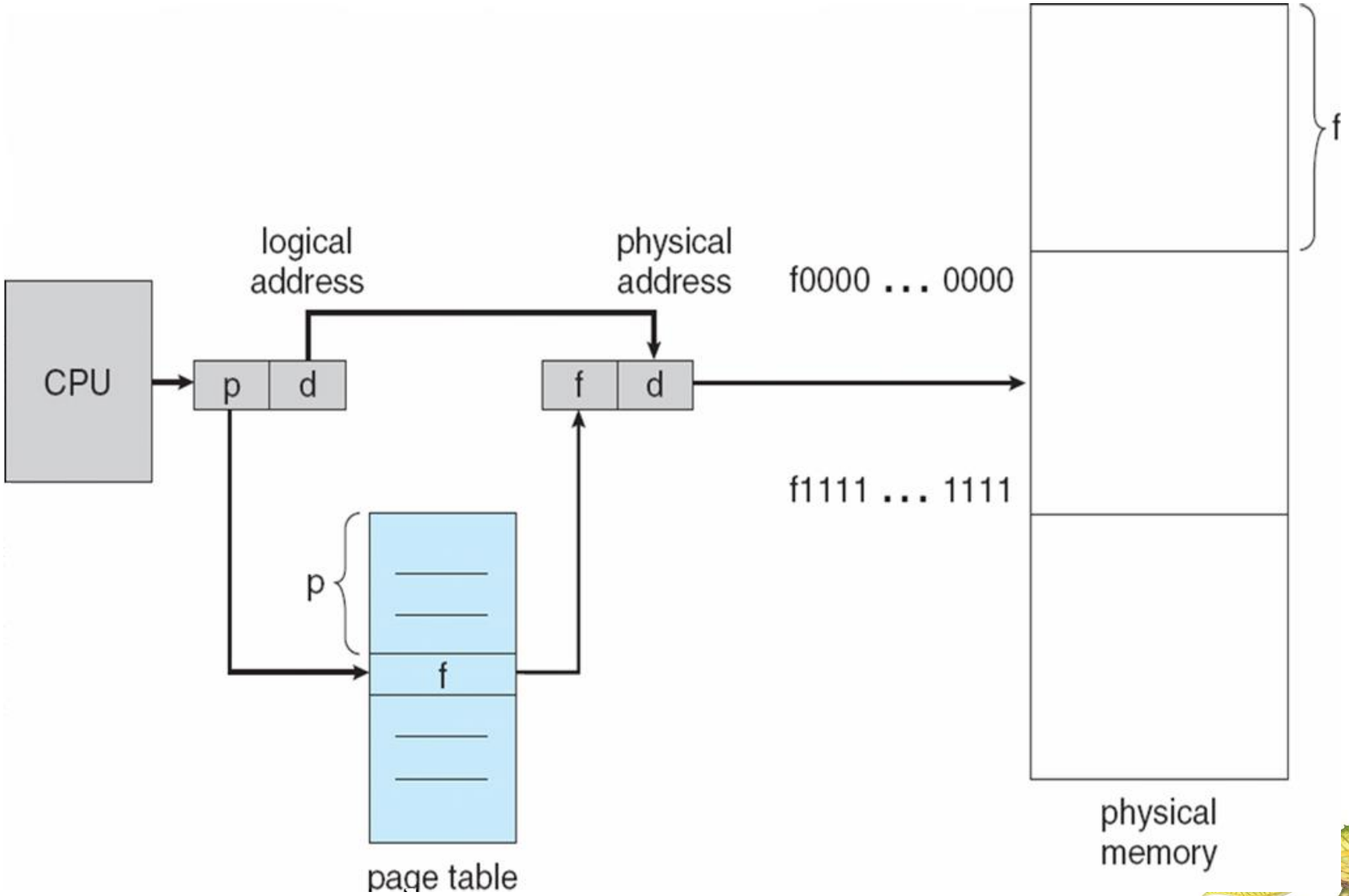
Address Translation Scheme

- 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

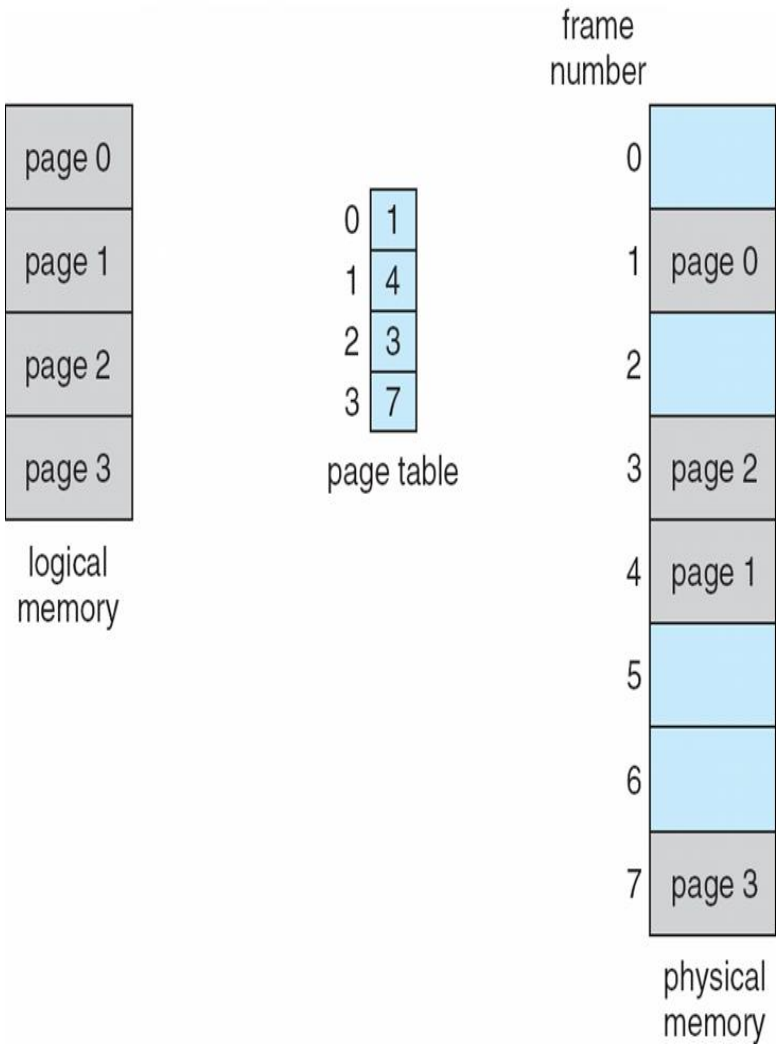
page number	page offset
p	*d*
m - *n*	*n*

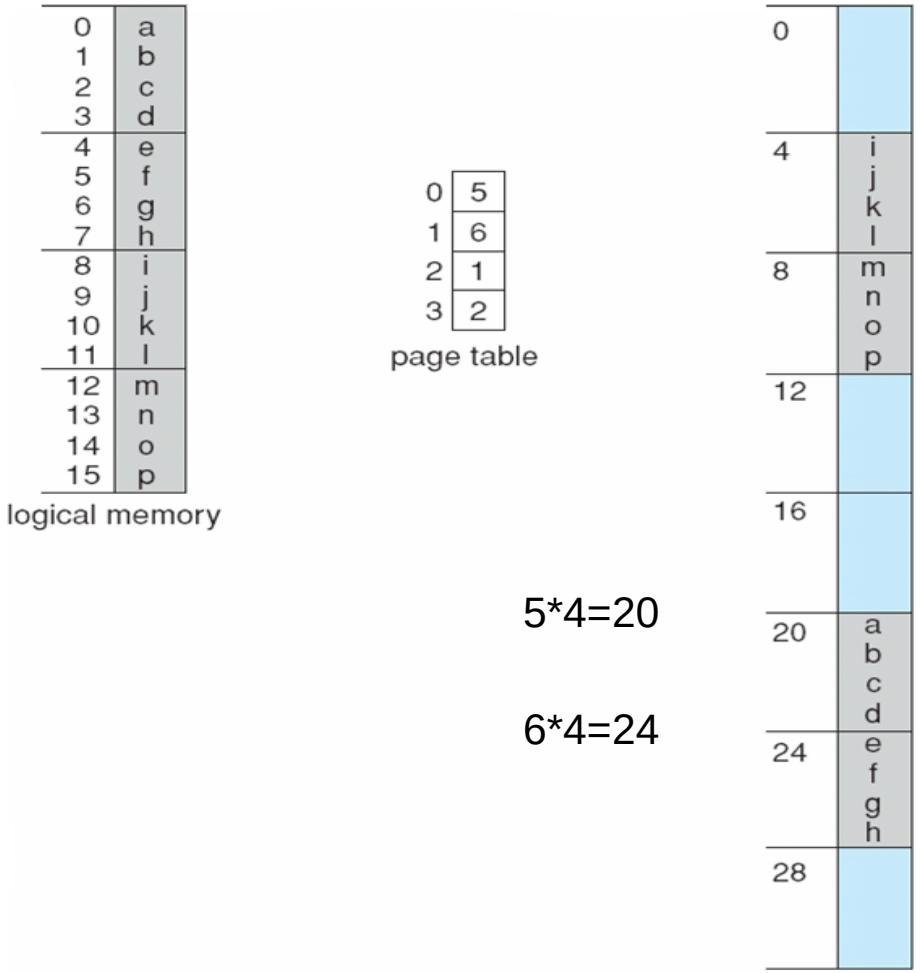
  - For given logical address space 2^m *and page size* 2^n

Paging Hardware



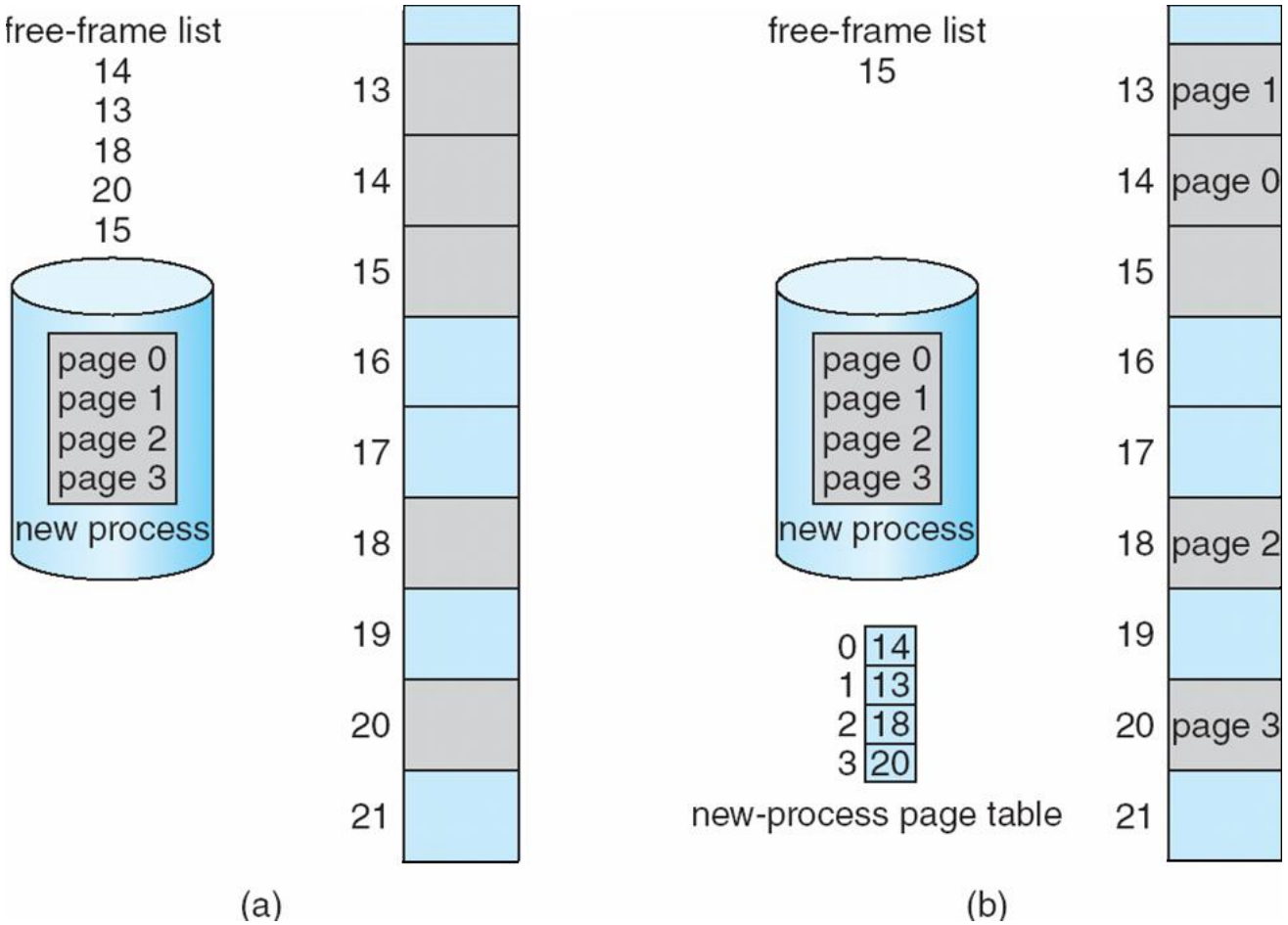
Paging Model of Logical and Physical Memory





32-byte memory and 4-byte pages

Free Frames

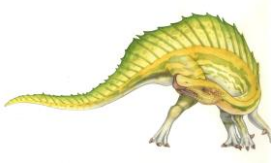


Before allocation

After allocation

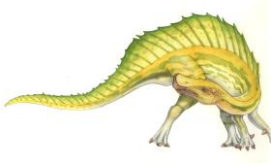
Implementation of Page Table

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



TLB

- 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)**
  - ✓ **Fast but expensive**
  - ✓ **Numbering between 64-1024**
- 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

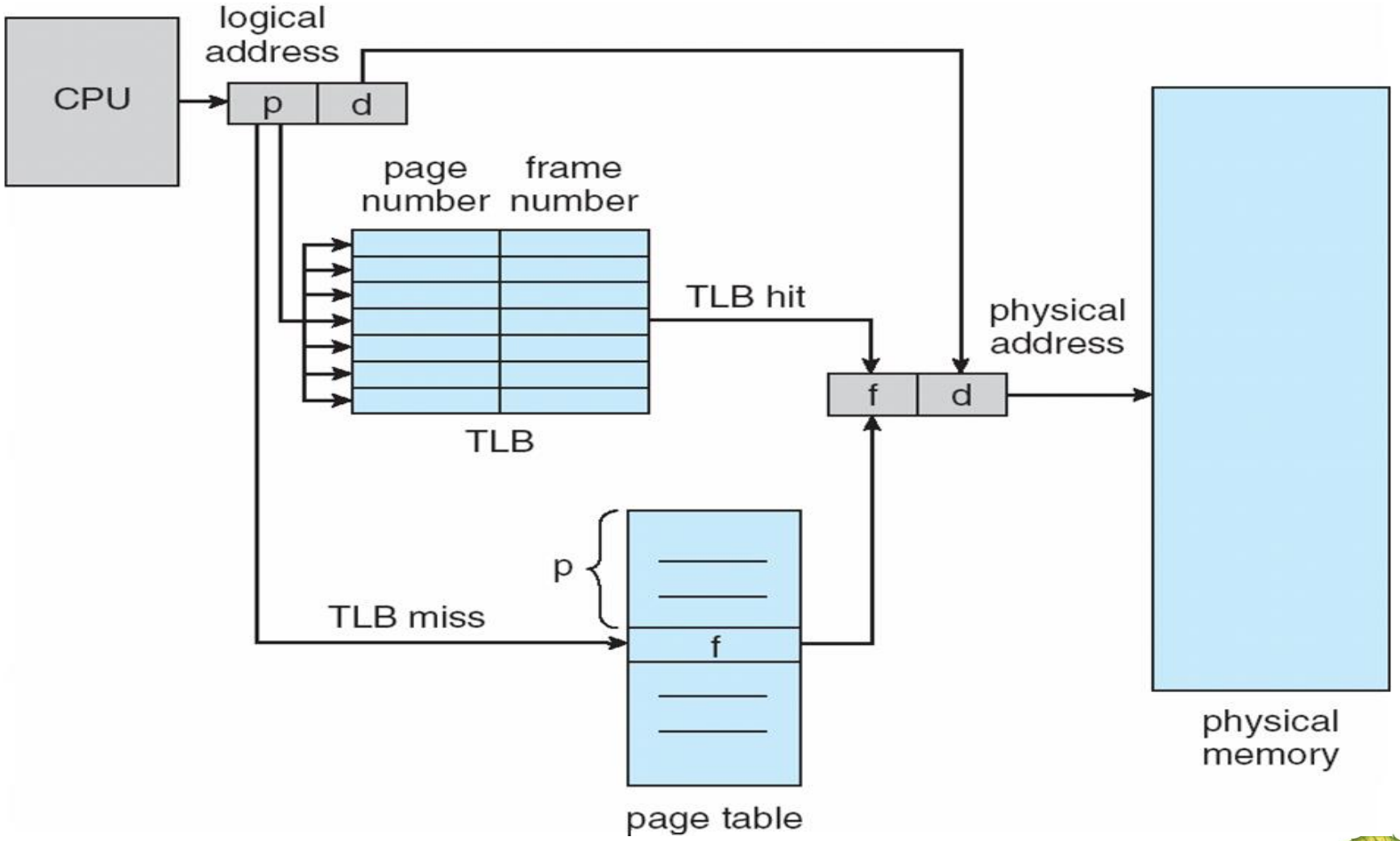
- Associative memory – parallel search

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

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

$$EAT = (1 + \varepsilon)\ \alpha + (2 + \varepsilon)(1 - \alpha) = 2 + \varepsilon - \alpha$$

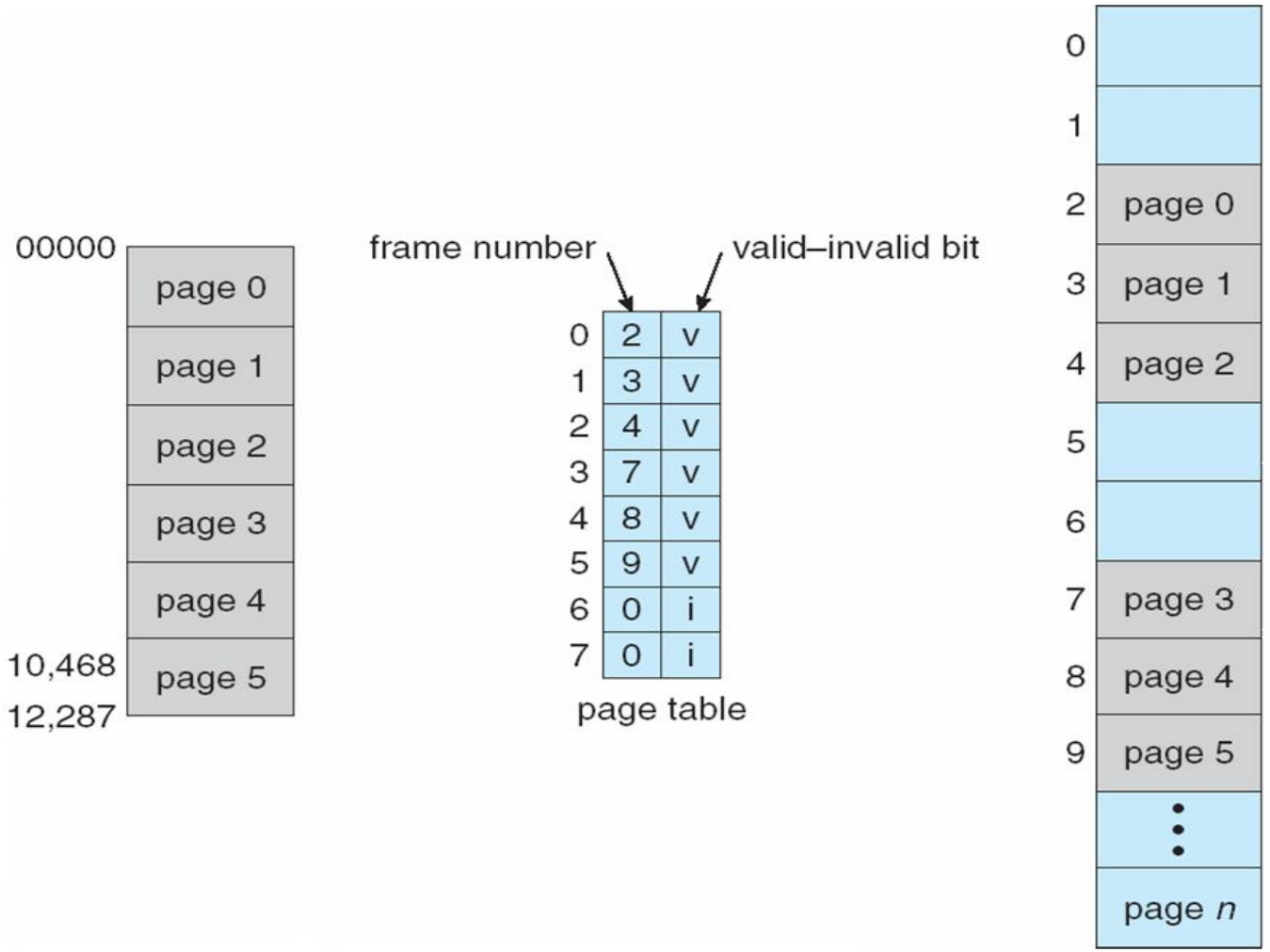
- Assume that associative lookup time is 20ns, memory cycle time is 100ns, the TLB hit ratio is 85%, then effective access time is as follow:

$$T=0.85*120+0.15*220=135ns.$$

Memory Protection

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

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

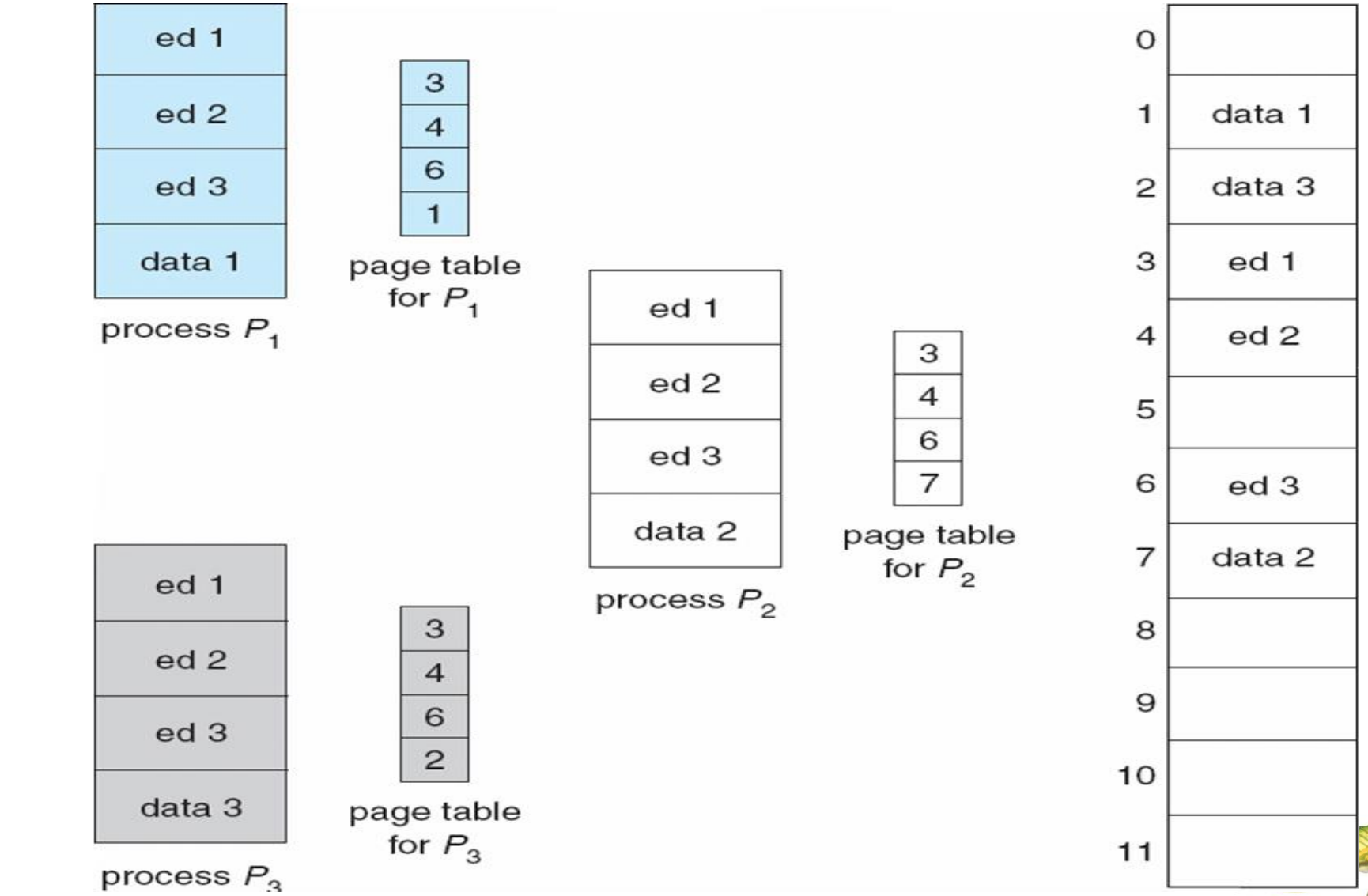


Shared Pages

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

- **Private code and data**
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space

Shared Pages Example



Structure of the Page Table

- 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

Two-Level Page-Table Scheme

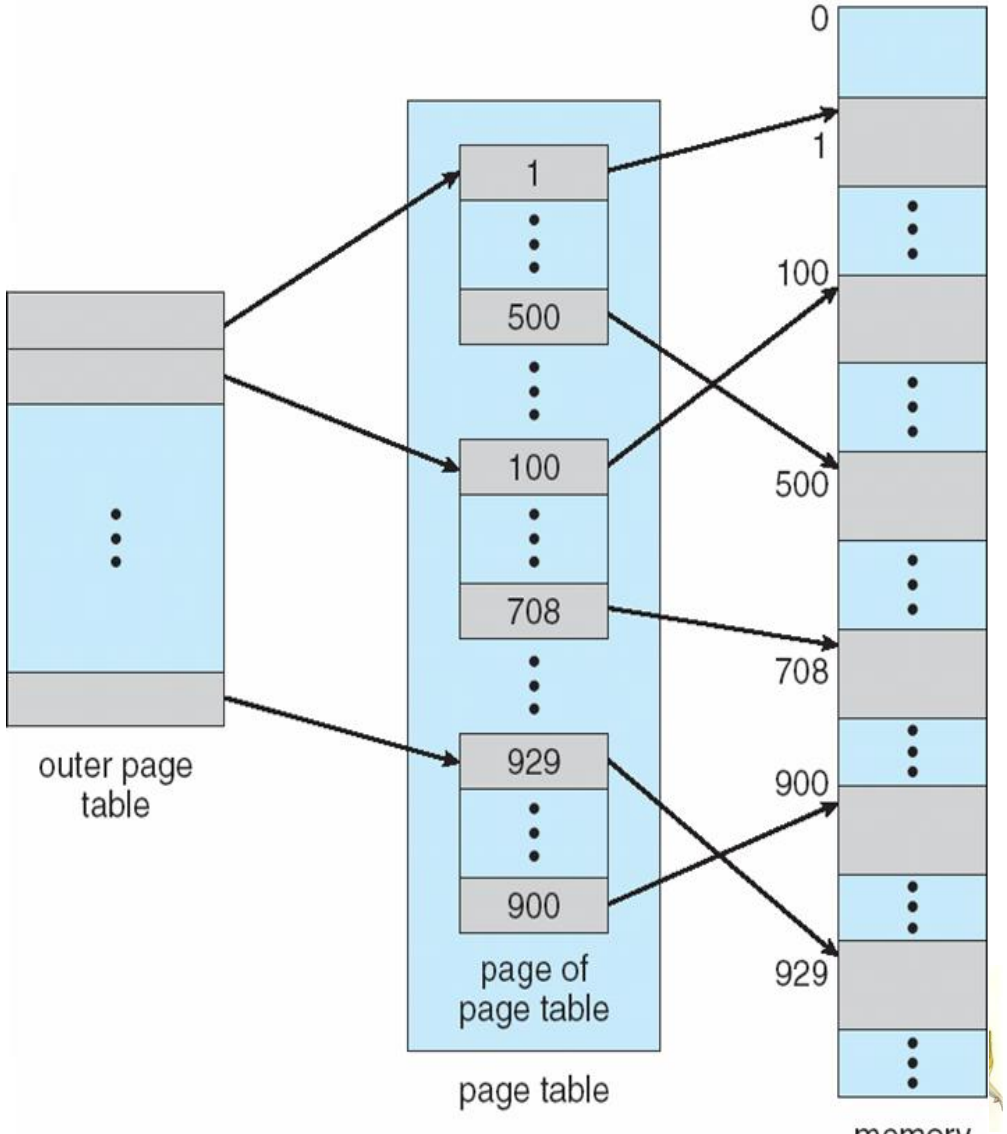
假设: logical address space=2^{32}B.

如果: page size=4KB（2^{12}B），

则: page table entries=2^{20}

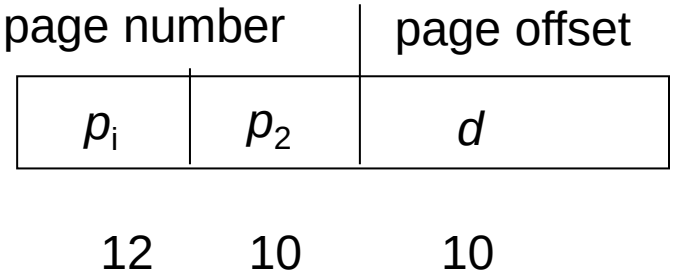
=4MB(如果每个项占4字节)

Solution: divide the page table into small pieces.



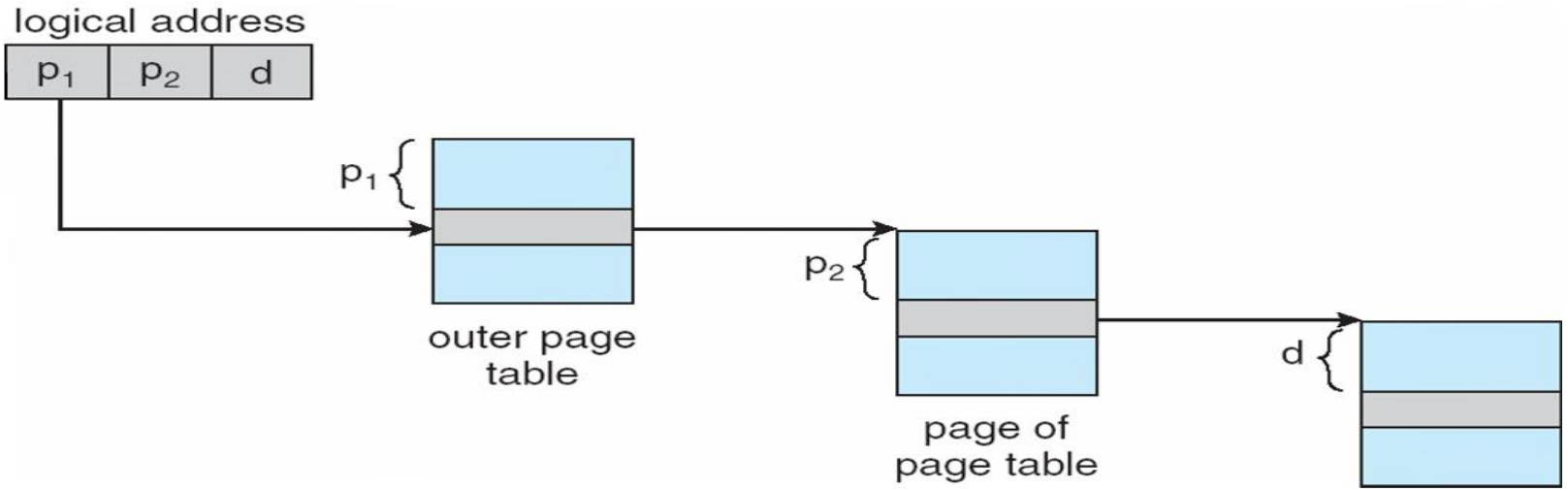
Two-Level Paging Example

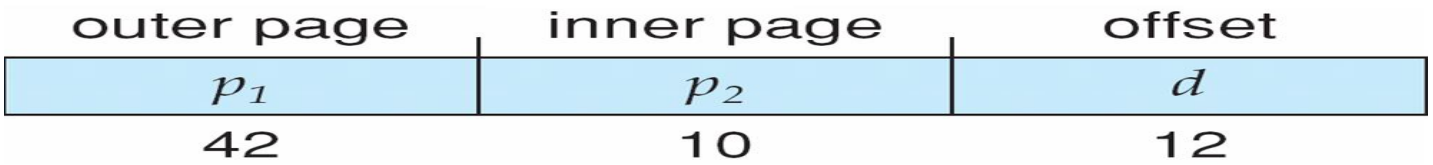
- A logical address (on 32-bit machine with 1K page size) is divided into:
  - ✓ a page number consisting of 22 bits
  - ✓ a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - ✓ a 12-bit page number
  - ✓ a 10-bit page offset
- Thus, a logical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



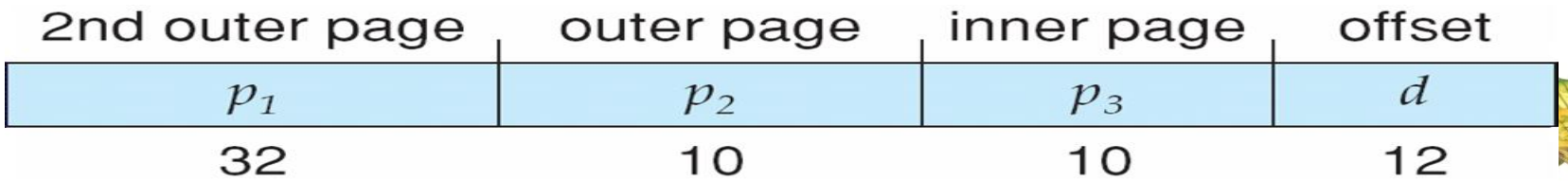
- where p_i is an index into the outer page table, and p_2 is the displacement within the page of the outer page table

Address-Translation Scheme





Three-level Paging Scheme:



Multilevel Paging and Performance

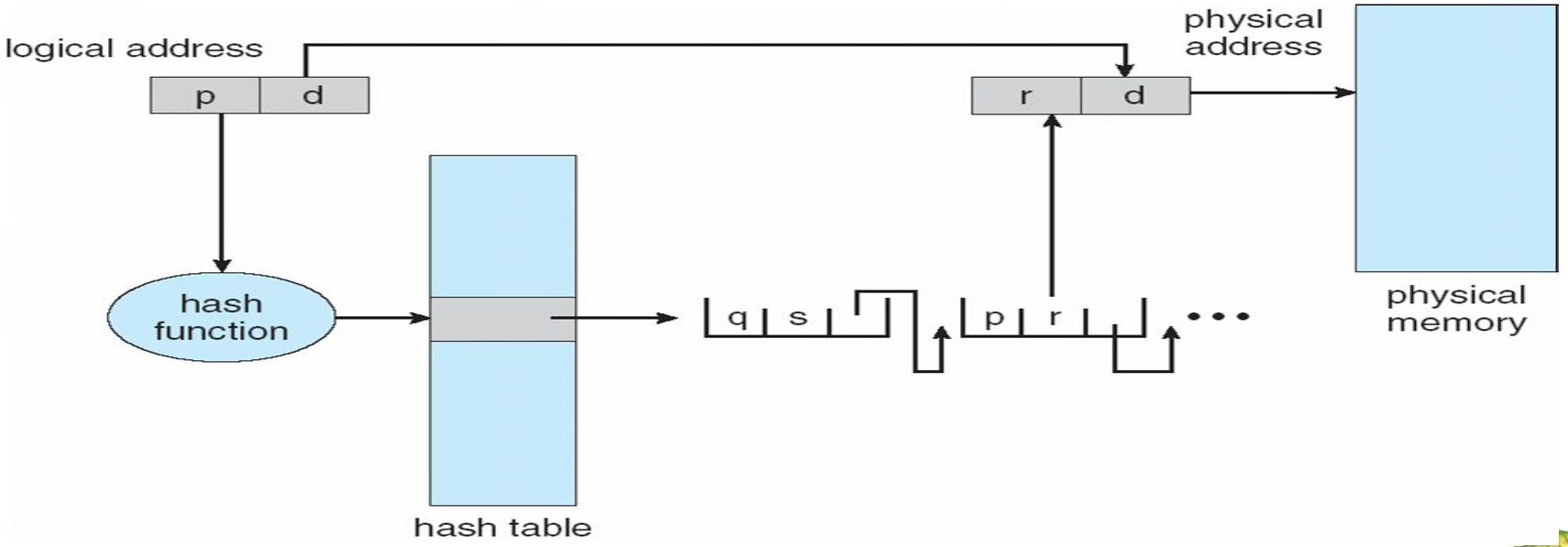
- Multilevel paging affects system performance.
- Given that each level is stored as a separate table in memory, converting a logical address to a physical one may take four memory access in a four level paging system.
- Cache hit rate of 98 percent yields:

  effective access time = 0.98 x 120 + 0.02 x 520= 128 ns

- which is only a 28 percent slowdown in memory access time.

Hashed Page Tables

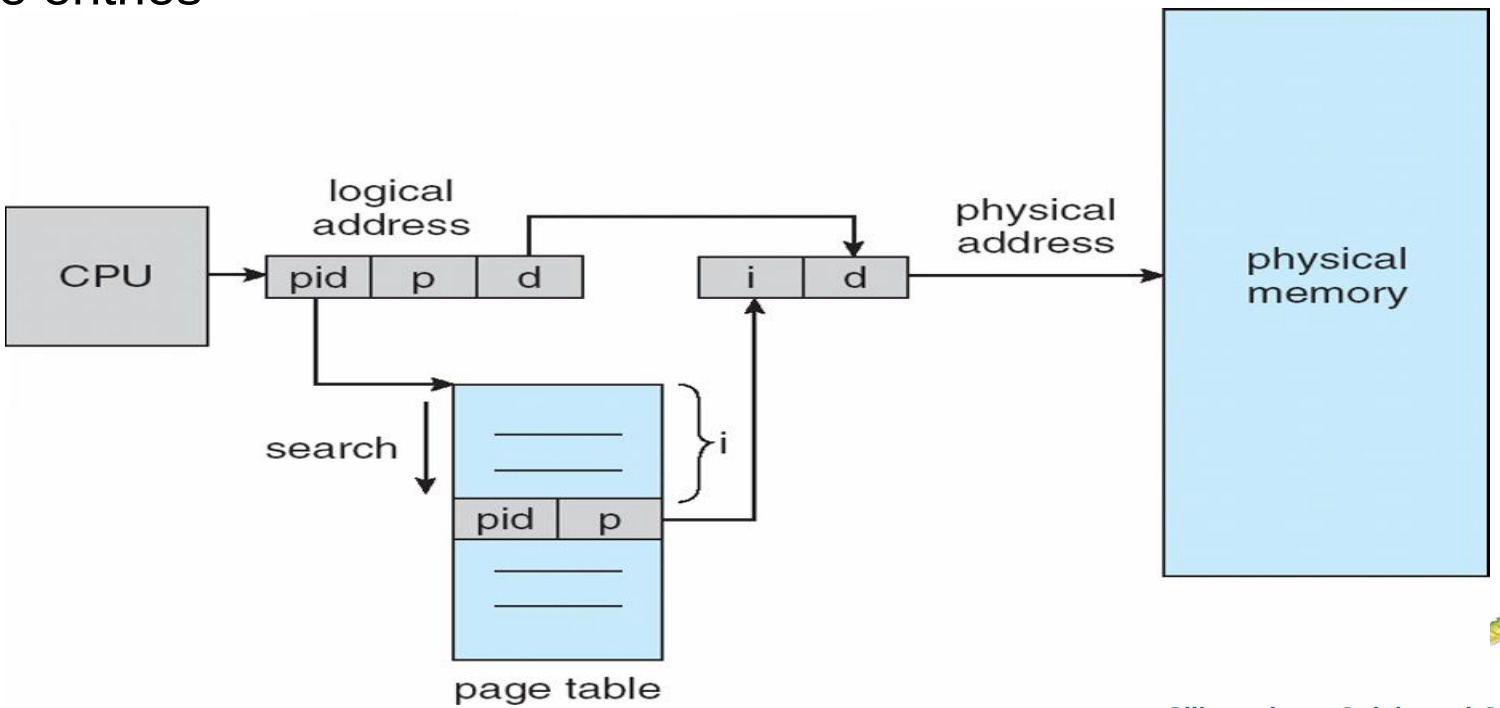
- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - ✓ This page table contains a chain of elements hashing to the same location
- Virtual page numbers are compared in this chain searching for a match
  - ✓ If a match is found, the corresponding physical frame is extracted



哈希运算:输入一个不限长度位串，返回一个固定长度的位串，不可逆。

Inverted Page Table

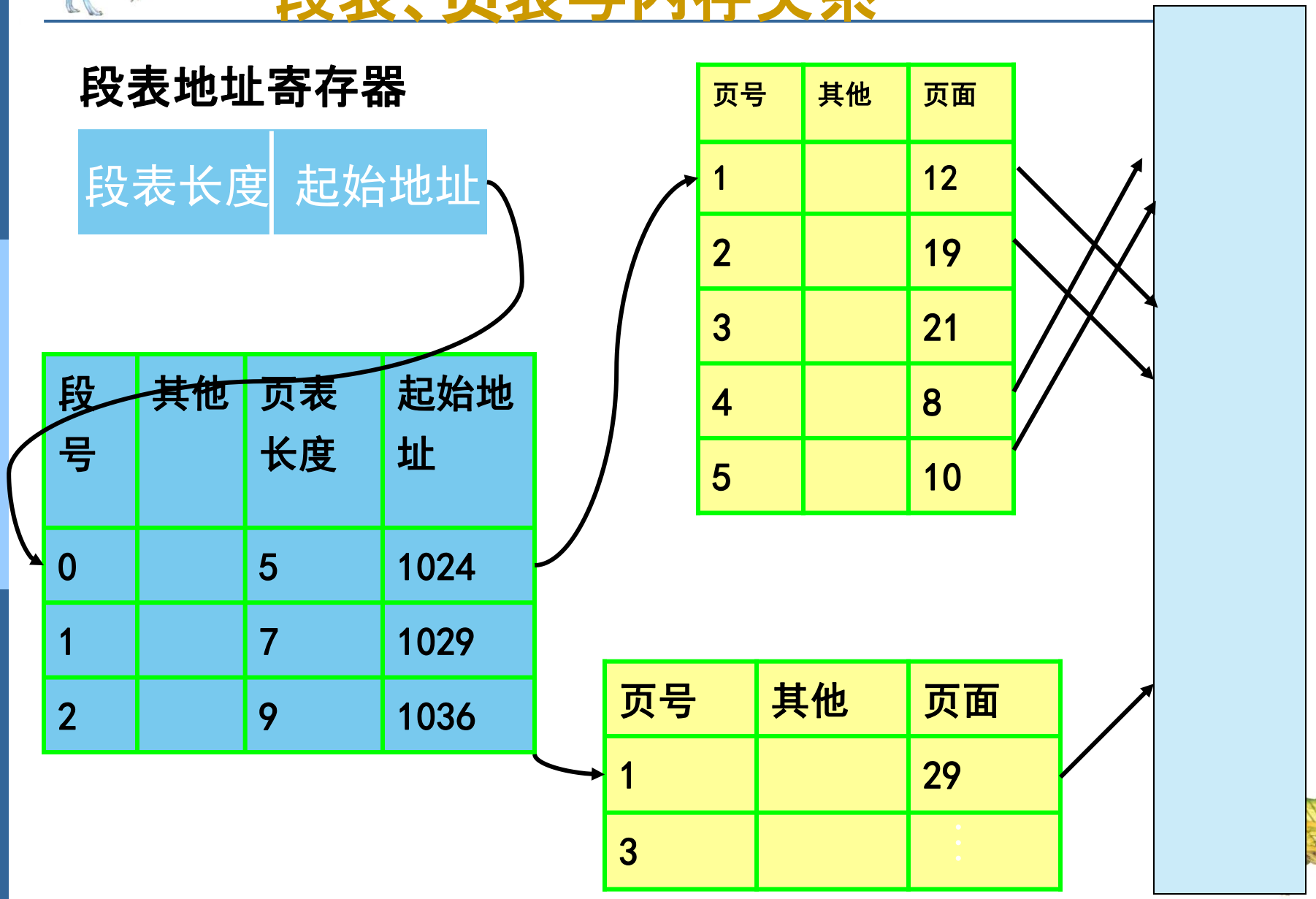
- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries



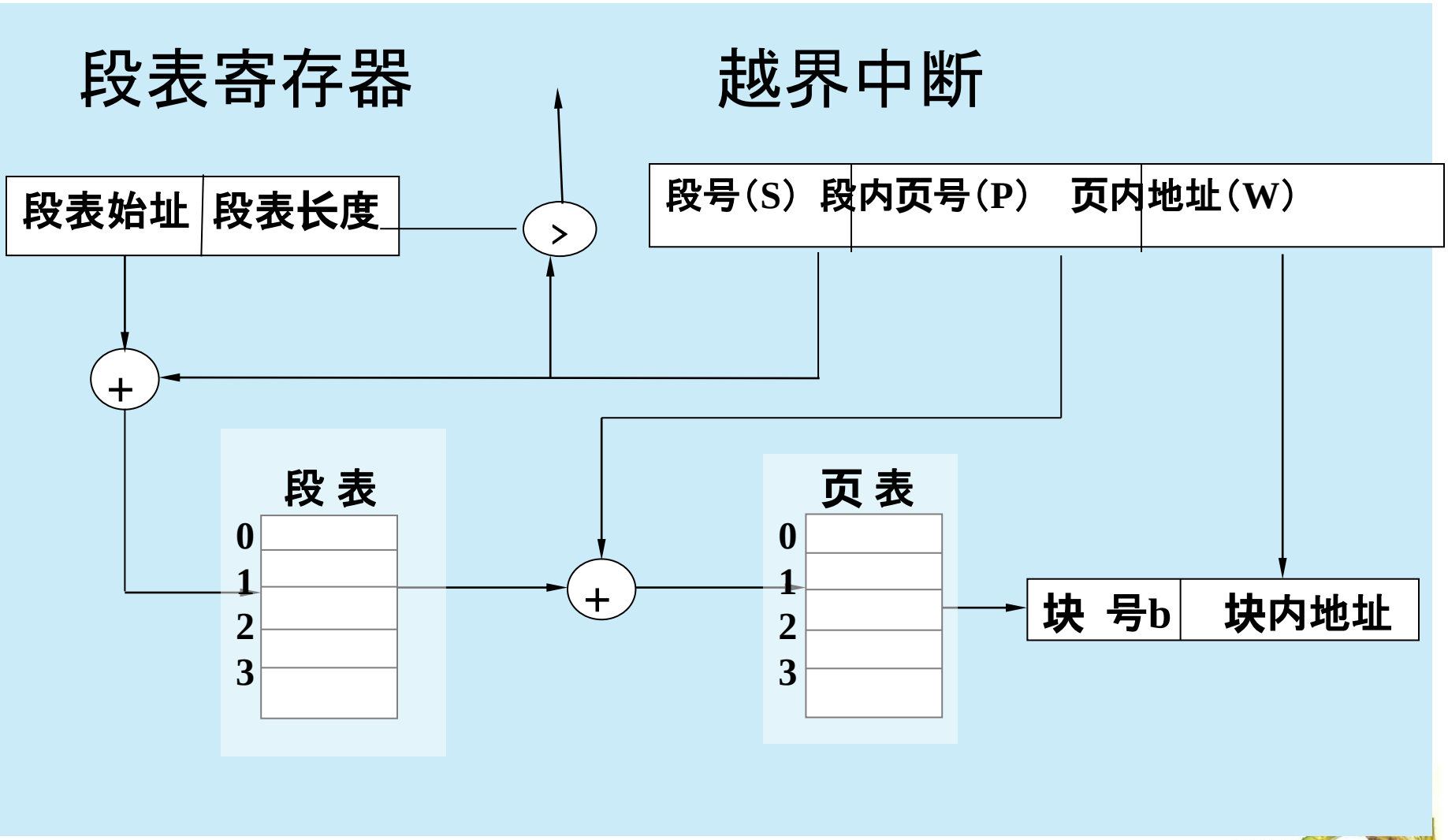
Segmentation with Paging

段号(S)	段内页号(P)	页内地址(W)

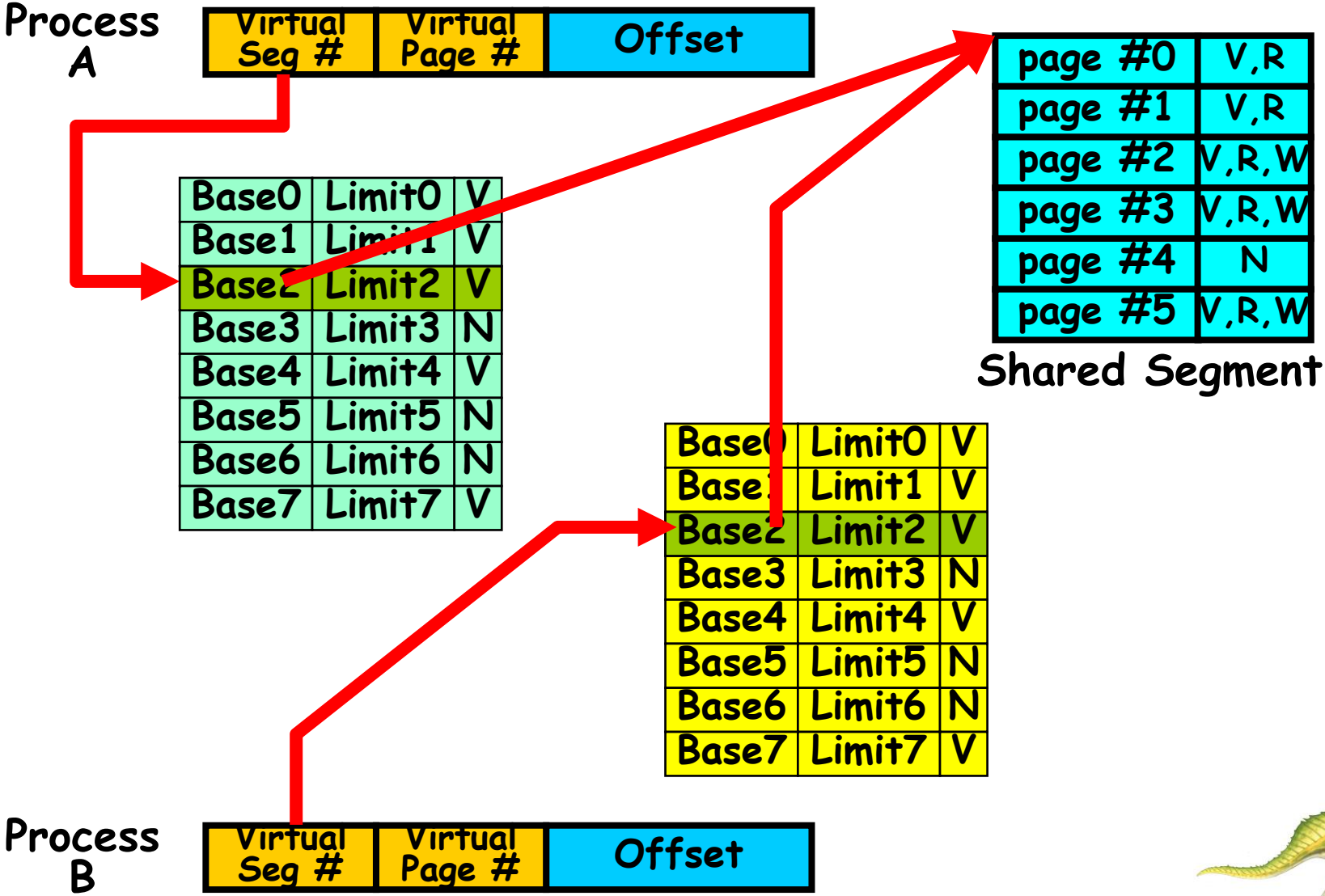
段表、页表与内存关系


段表地址寄存器
段表长度
起始地址
段号
其他
页表长度
起始地址
0
5
1024
1
7
1029
2
9
1036
页号
其他
页面
1
12
2
19
3
21
4
8
5
10
页号
其他
页面
1
29
3

Address translation in Segmentation with Paging



What about Sharing (Complete Segment)?

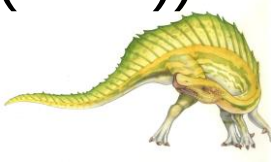


Example: The Intel 32 and 64-bit Architectures

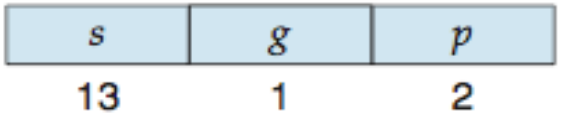
- Dominant industry chips
- Pentium CPUs are 32-bit and called IA-32 architecture
- Current Intel CPUs are 64-bit and called IA-64 architecture
- Many variations in the chips, cover the main ideas here

Example: The Intel IA-32 Architecture

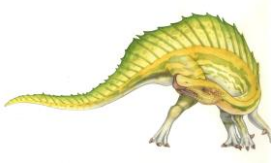
- Supports both segmentation and segmentation with paging
  - Each segment can be 4 GB
  - Up to 16 K segments per process
  - Divided into two partitions
    - First partition of up to 8 K segments are private to process (kept in **local descriptor table** (**LDT**))
    - Second partition of up to 8K segments shared among all processes (kept in **global descriptor table** (**GDT**))



Example: The Intel IA-32 Architecture (Cont.)

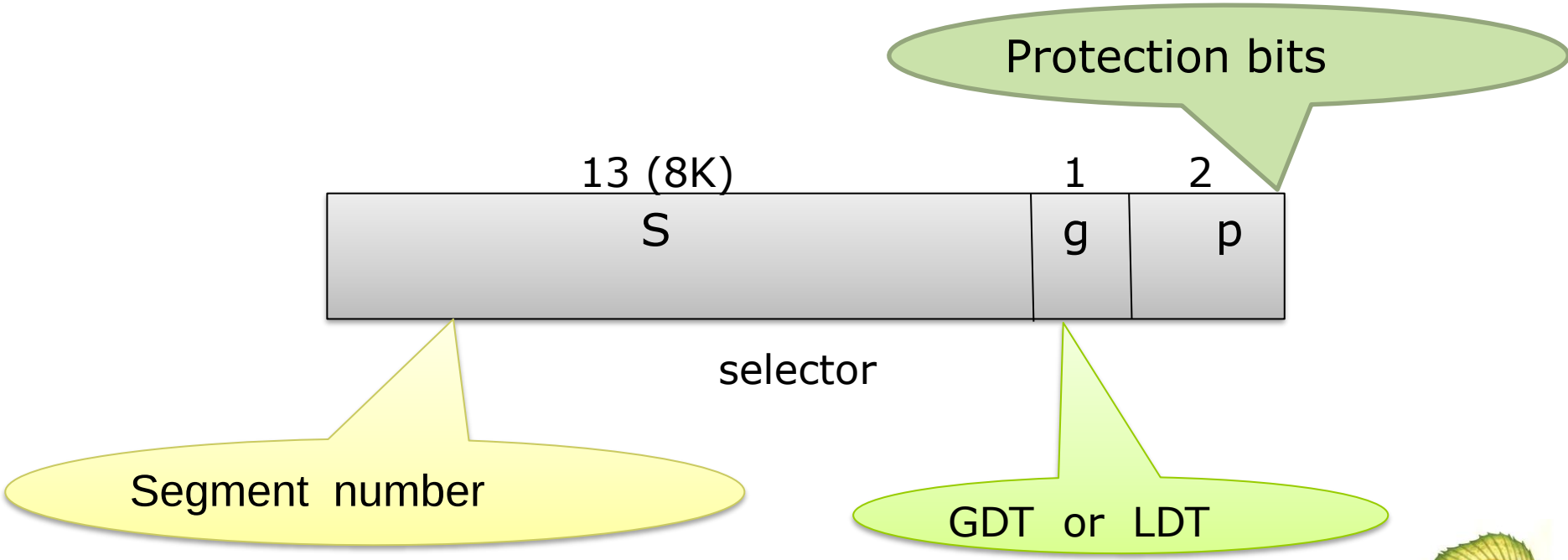
- CPU generates logical address
  - Selector given to segmentation unit
    - Which produces linear addresses
    - 

  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU
    - Pages sizes can be 4 KB or 4 MB



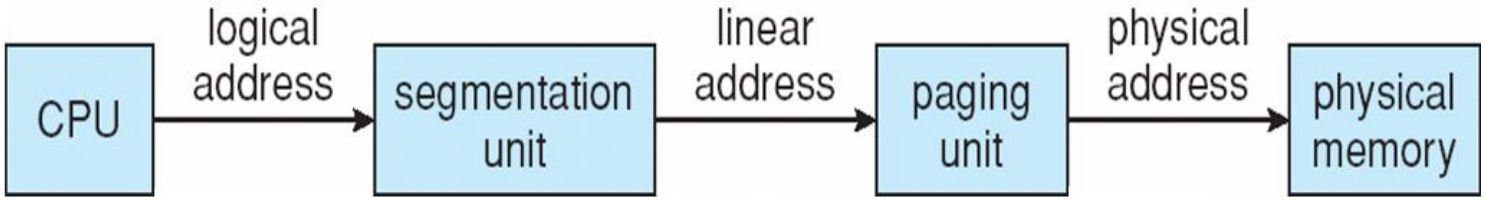
Example: The Intel IA-32 Architecture (Cont.)

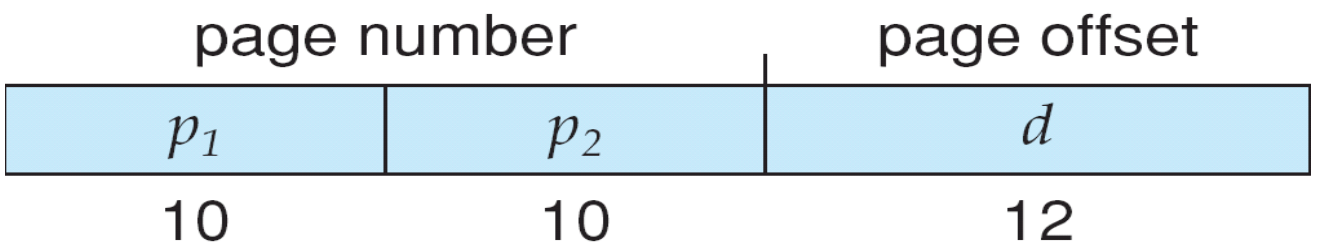
- The logical address is a pair (selector, offset), selector is a 16-bit number as follows

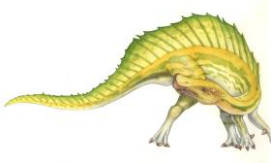


GDT/LDT : Global /Local Descriptor Table

Example: The Intel IA-32 Architecture (Cont.)



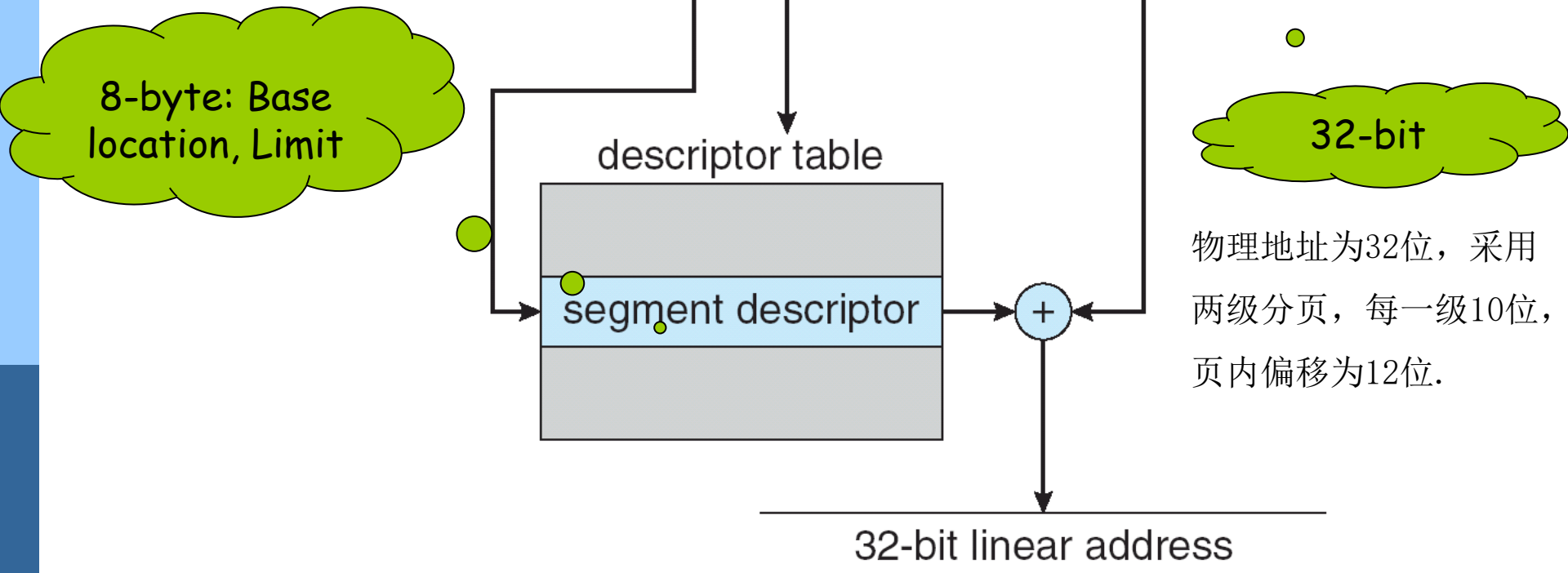




Example: The Intel IA-32 Architecture (Cont.)

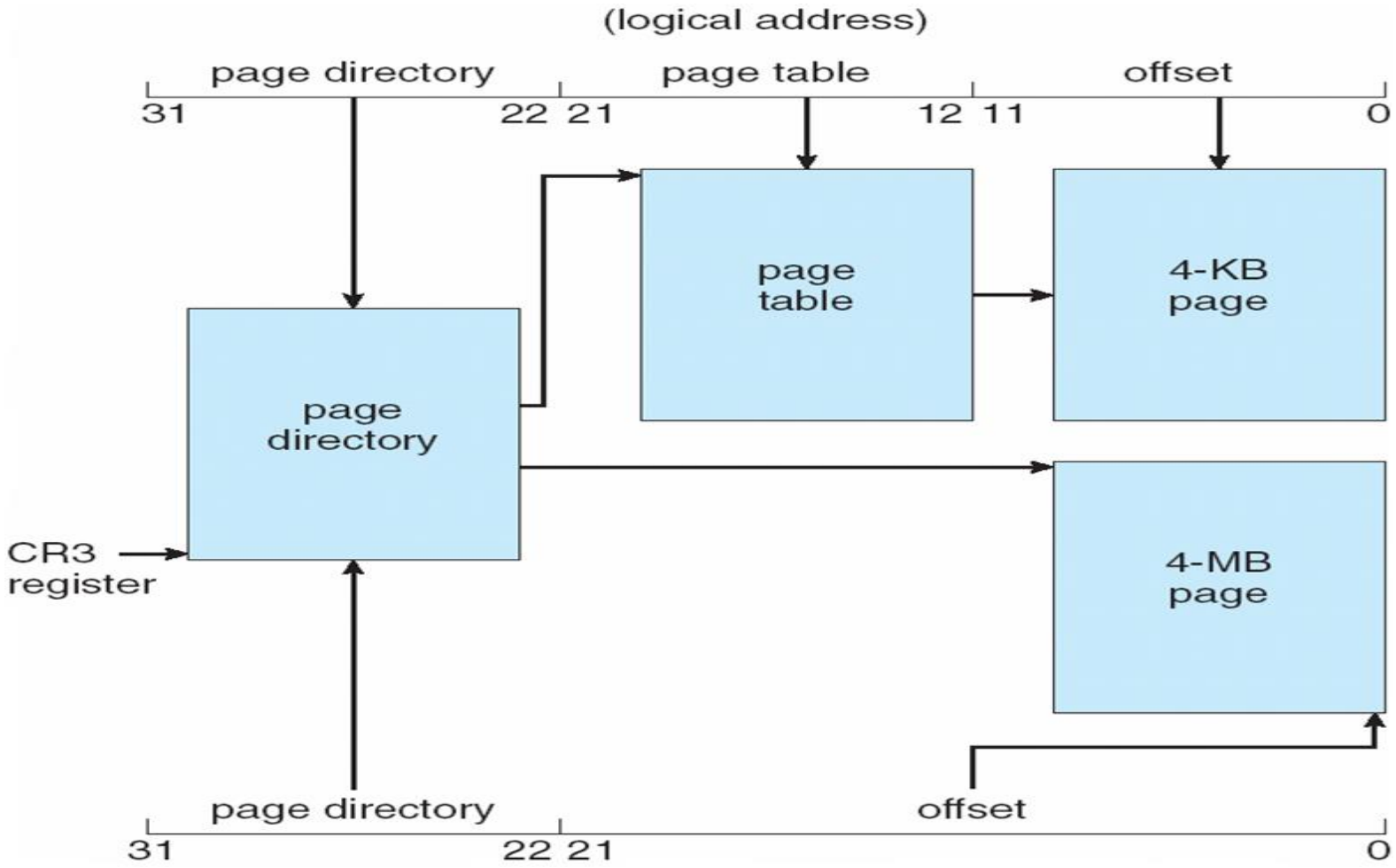
16-bit

8K个私有段
8K个公有段



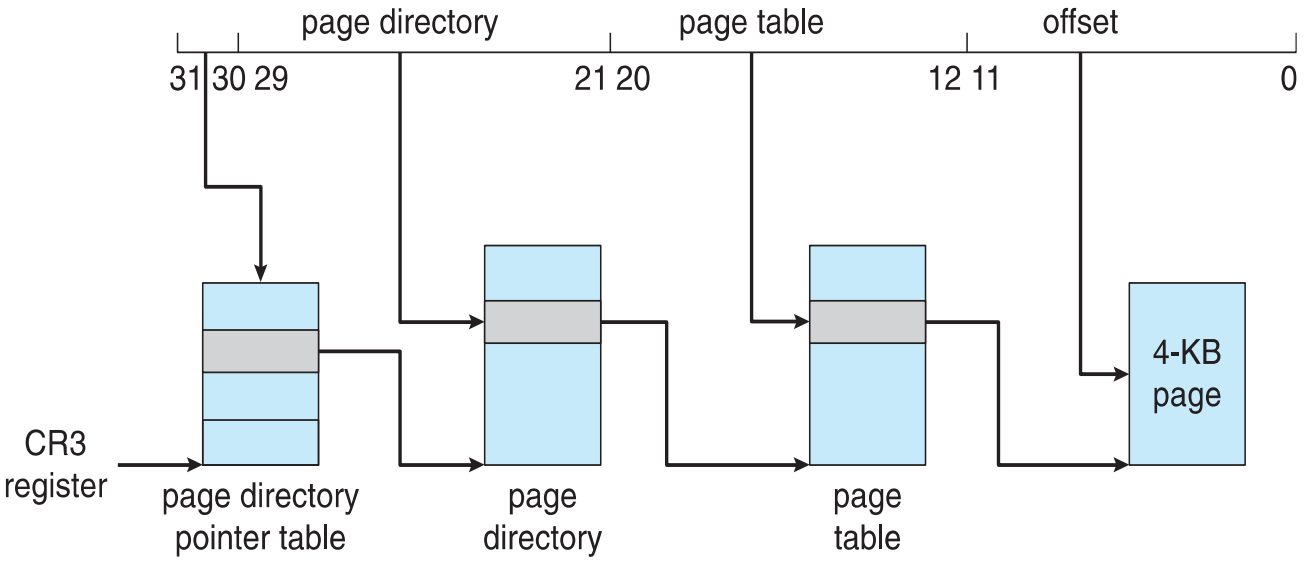
物理地址为32位，采用两级分页，每一级10位，页内偏移为12位.

Intel IA-32 Paging Architecture



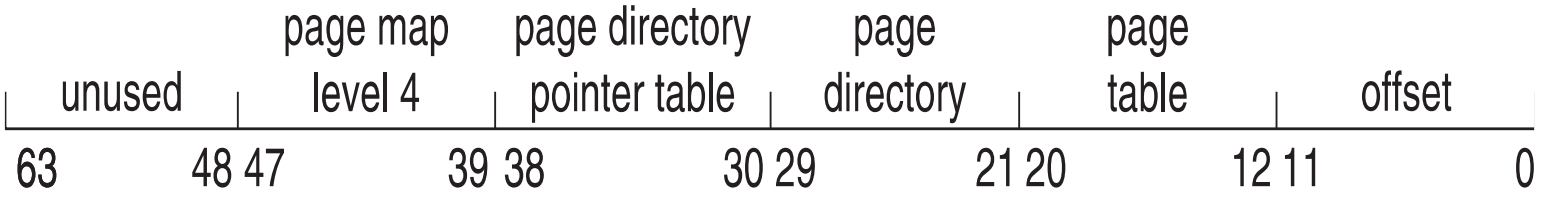
Intel IA-32 Page Address Extensions

- 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
  - Net effect is increasing address space to 36 bits – 64GB of physical memory



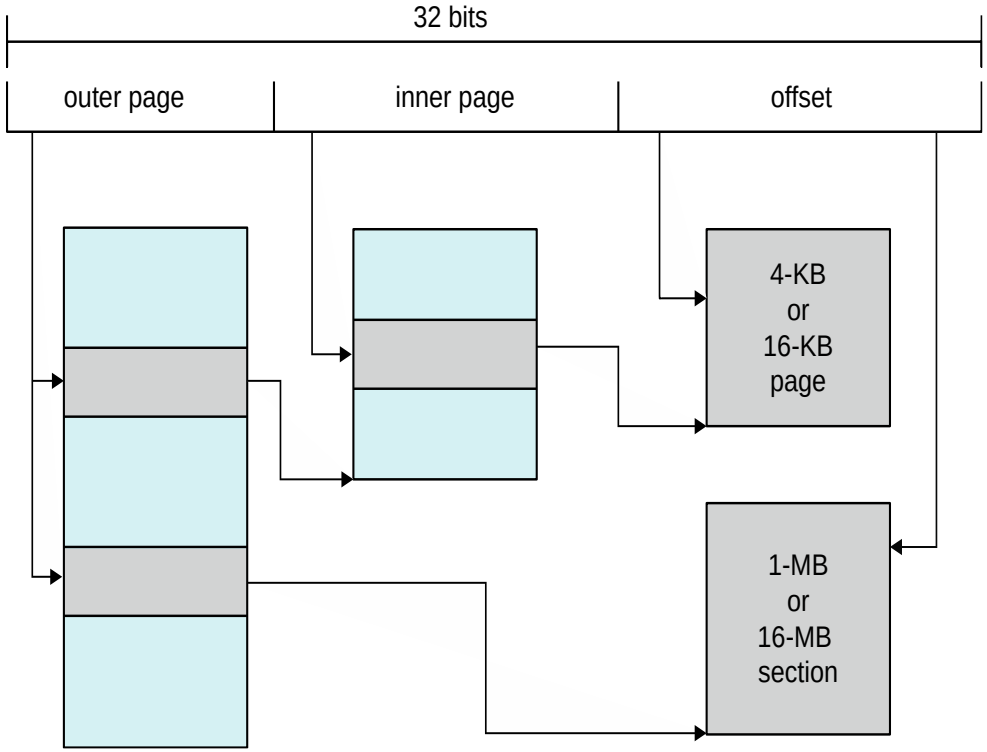
Intel x86-64

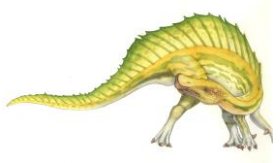
- Current generation Intel x86 architecture
- 64 bits is ginormous (> 16 exabytes)
- In practice only implement 48 bit addressing
  - Page sizes of 4 KB, 2 MB, 1 GB
  - Four levels of paging hierarchy
- Can also use PAE so virtual addresses are 48 bits and physical addresses are 52 bits



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

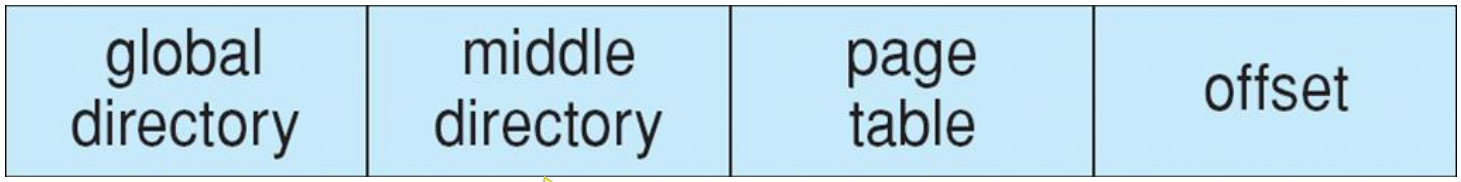




Linear Address in Linux

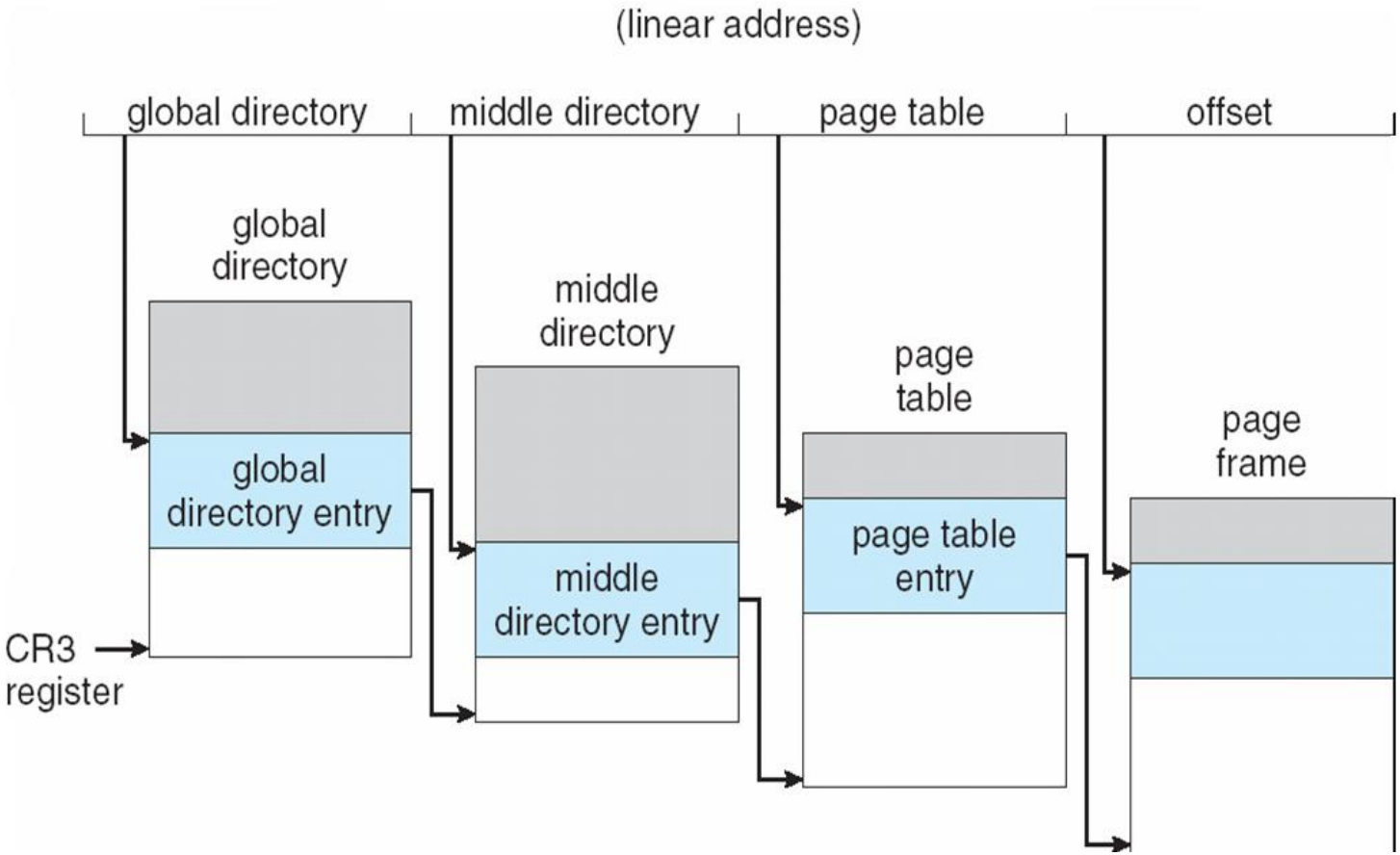
Broken into four parts to work well for both 32-bit and 64-bit architectures

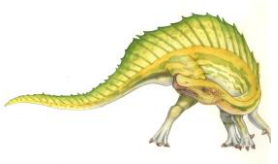
The number of bits in each part of the linear address varies according to the architecture



On Pentium system, the size of the middle directory is zero bits.

Three-level Paging in Linux





assignment

- 8.1
- 8.3
- 8.12
- 8.17

End of Chapter 8