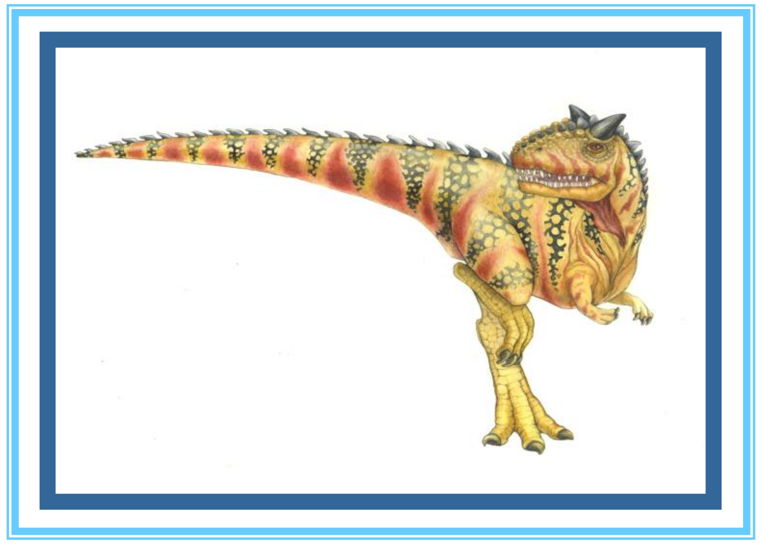
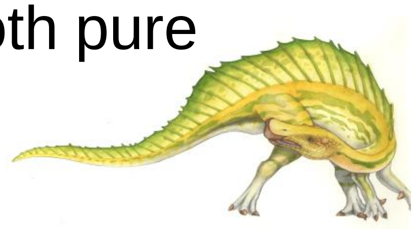
Chapter 8: Memory-Management Strategies



Chapter 8: Memory Management Strategies

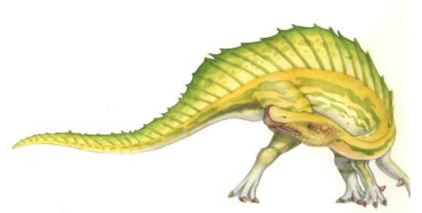
- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table
- Example: The Intel 32 and 64-bit Architectures
- Example: ARM Architecture

- **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



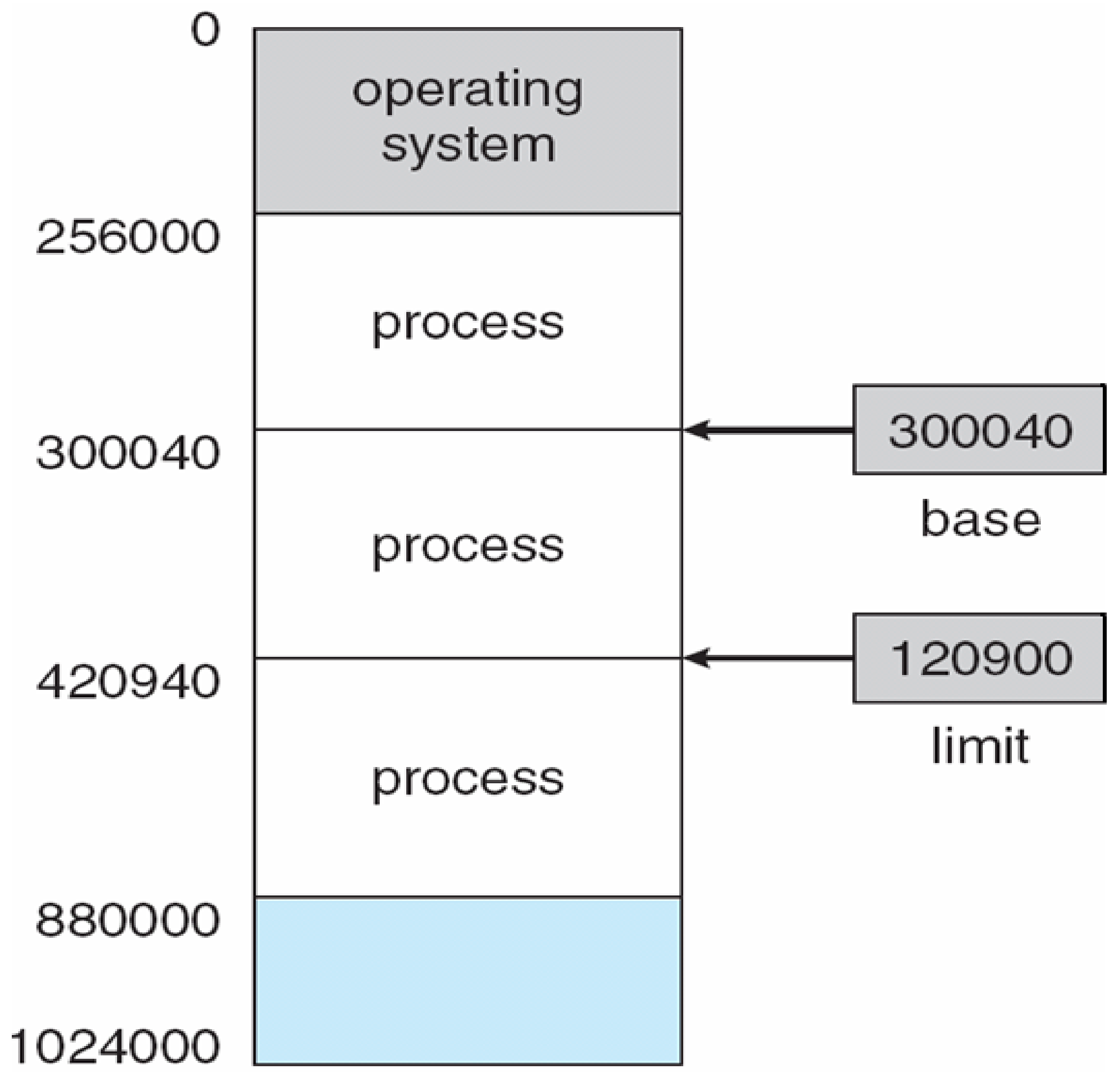
Background

- 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
- 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



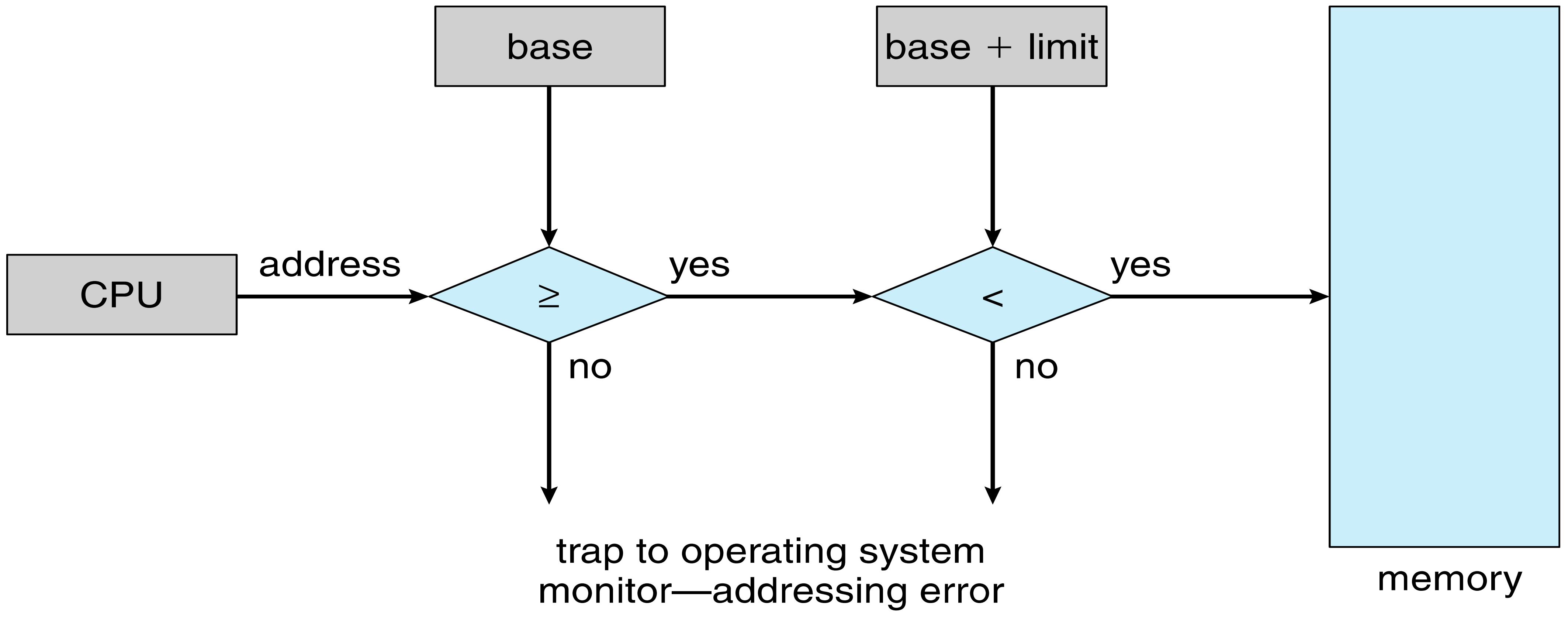
Base and Limit Registers

- In multiuser systems, we must separate per-process memory space to protect user processes from one another.
- One possible solution
  - A pair of **base** and **limit registers** define the logical address space
  - CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



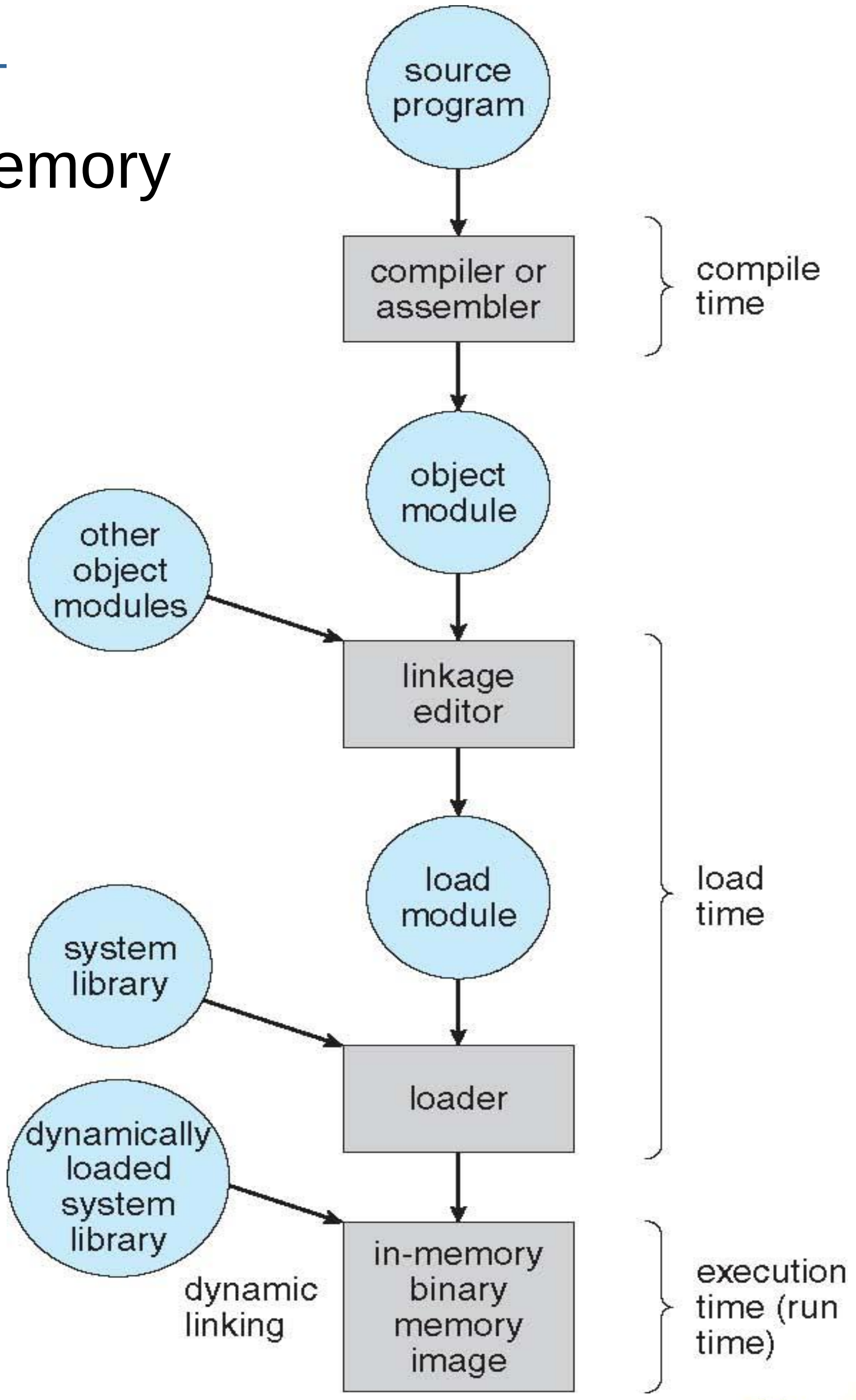
Hardware Address Protection with Base and Limit Registers

- The base and limit registers can be loaded only by the OS which uses a special privileged instruction.
- Any attempt by a program executing in user mode to access OS memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error.



Address Binding

- Address binding: Binding of instructions and data to memory addresses. It can happen at three different stages
  - **Compile time**: If memory location ***known*** a priori, **absolute addresses** can be generated at compile time. But, must recompile it if the location changes
    - e.g., MS-DOS .COM-format programs
  - **Load time**: If memory location is ***unknown*** at compile time, **compiler** must generate **relocatable code** (i.e. "14 bytes from beginning of this module"). When starts to run, **loader** will bind the relocatable addresses to **absolute addresses** (i.e., 74014)
  - **Execution time**: Binding delayed until run time if processes can be *moved during its execution* from one memory segment to another
    - Need hardware support for address maps
    - Most general-purpose OSs use this method.



Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate physical address space is central to **memory management**
  - **Logical address** – generated by CPU; also referred to as **virtual address**
    - *Logical address space*: the set of all logical addresses generated by a program
  - **Physical address** – address seen by the memory unit
    - *Physical address space*: the set of all physical addresses corresponding to those logical addresses belonging to that program.
- CPU generates for only ***logical* addresses** and *thinks that the process runs in locations 0 to max*; it never sees the *real* physical addresses
  - Hardware device that at run time maps logic to physical address (Many methods possible, covered in the rest of this chapter)
- For address binding at *compile-time* and *load-time*
  - Logical and physical addresses are the *same* (CPU generates absolute address)
- For address binding at *execution time*
  - logical (virtual) and physical addresses *differ* (need address mapping/translation)

Dynamic Loading

- **Dynamic loading**
  - All routines kept on disk in relocatable load format
    - It is not necessary for the entire program and all data of a process to be in physical memory for the process execute.
  - Routine is not loaded until it is called
    - When a routine needs to call another one, the calling routing first checks whether the other routing has been loaded. If not, the relocatable linking loader is called to load the desired routine info memory.
- Better memory-space utilization: unused routine is never loaded
- Useful when large amounts of code are needed to handle infrequently occurring cases
- Implemented through program design
  - OS can help by providing libraries to implement dynamic loading

Dynamic Linking

- **Static linking** – system libraries and program code combined into a binary image
- **Dynamic linking** –linking, rather than loading, is postponed until execution time
  - **Stub** – a piece of code to locate appropriate memory-resident library routine
    - It is included in the image for each library routine reference.
    - It replaces itself with the address of the system routine, and executes the code
  - Without this, each program must include a copy of the system library (or the routines referenced by the program) in the executable image.

    ➔ waste disk and memory spaces.
- Dynamic linking is particularly useful for **shared libraries**
  - All processes that share a library execute only one copy of the library code.
  - Version info shall be included in both program and library,
    - So that when a library is replaced by a new version, programs will not accidentally execute the new, incompatible version of libraries.

Chapter 8: Memory Management Strategies

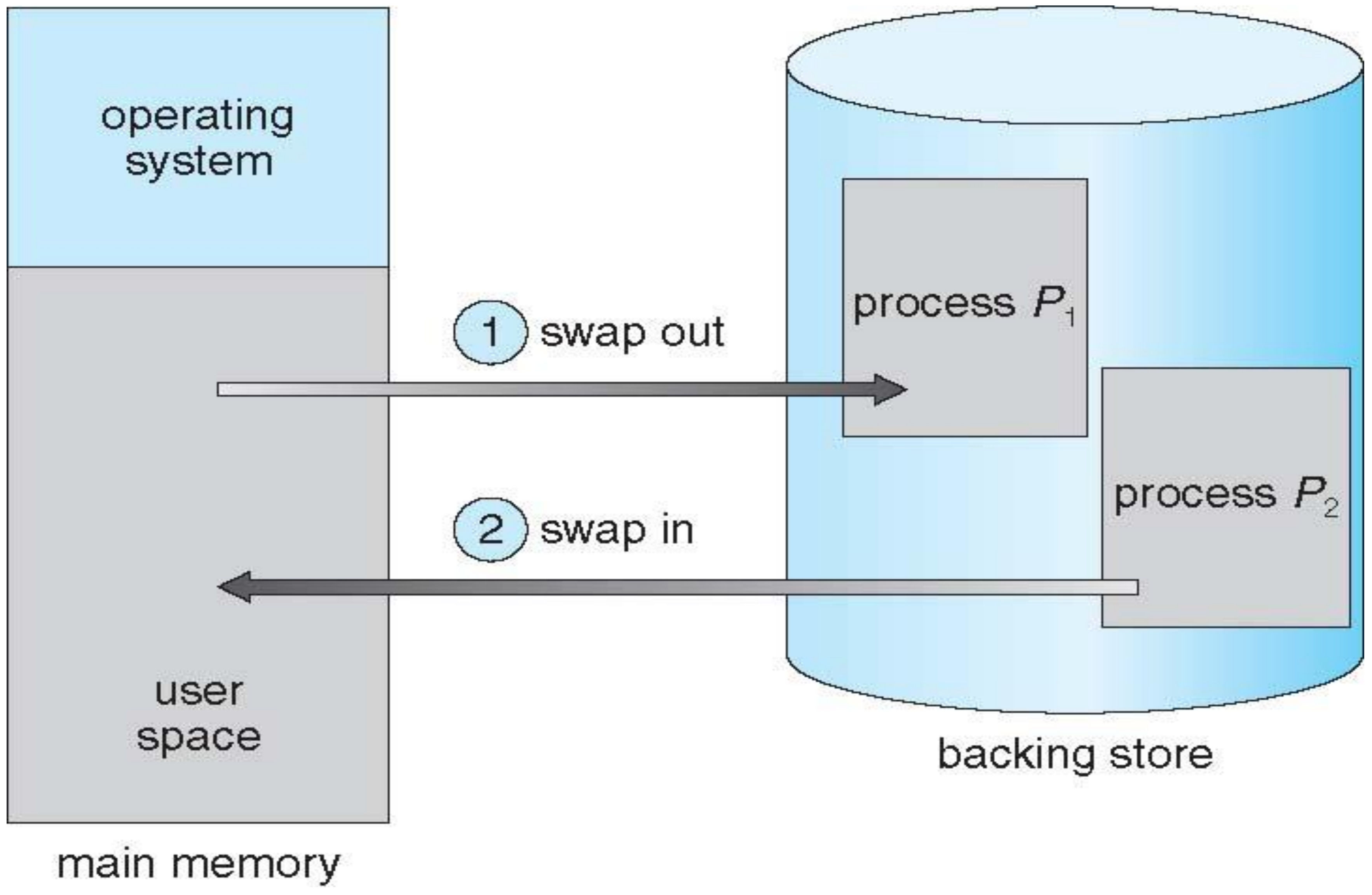
- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table
- Example: The Intel 32 and 64-bit Architectures
- Example: ARM Architecture

Swapping

- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - Allow total memory space needed by the processes to exceed physical memory
  - **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- Does the swapped out process need to swap back in to same physical addresses?
- Modified versions of swapping are found on many systems (UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold

Schematic View of Swapping

- CPU scheduler will call the **dispatcher** to check whether the next process 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.



Context Switch Time including Swapping

- If next processes to run 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 be reduced if reduce size of memory swapped – by knowing how much memory ***is*** using, not simply how much it ***might*** be using.
  - System calls to inform OS of memory use via `request memory()` and `release memory()`
- Other constraints as well on swapping
  - Never swap out a process with pending I/O 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
- Standard swapping not used in modern operating systems (too much swapping time)
  - But modified version common

Swapping on Mobile Systems

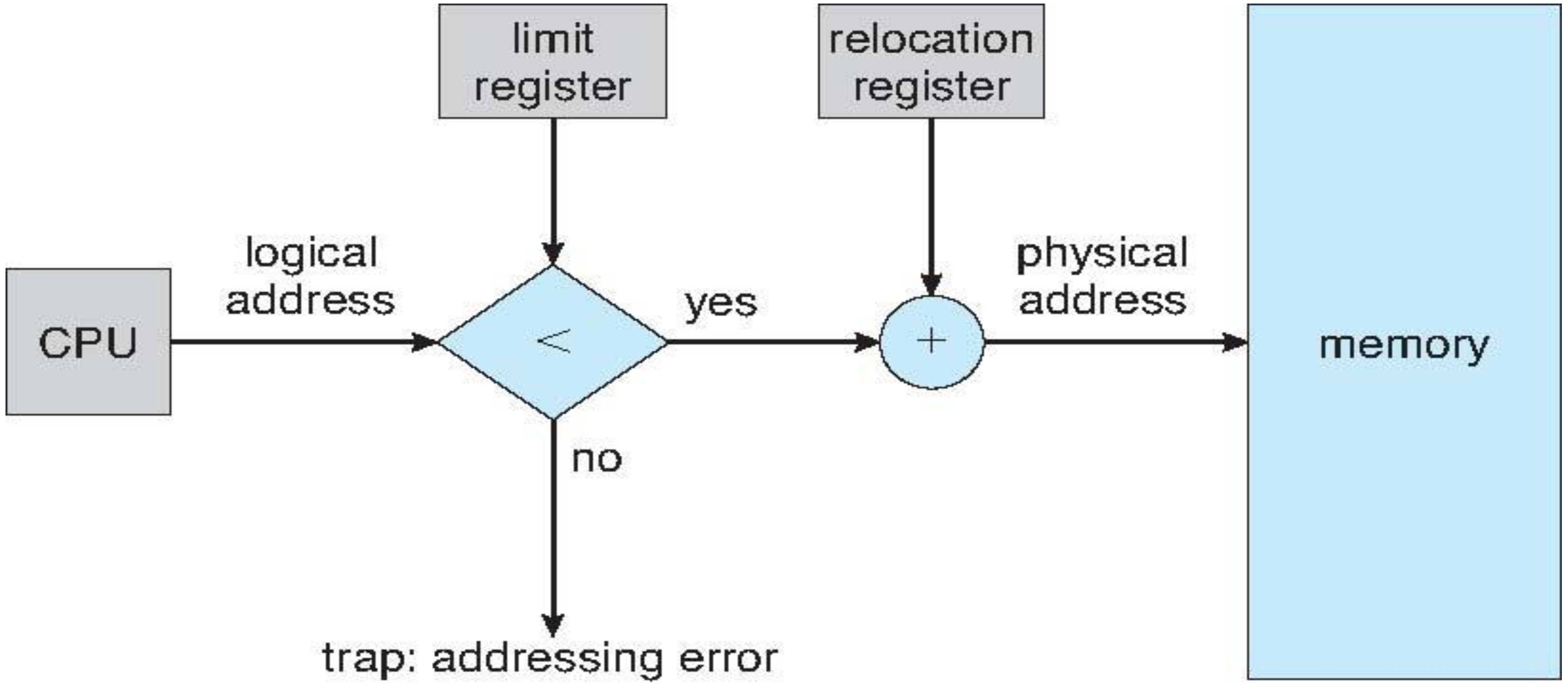
- Not typically supported
  - Flash memory based
    - Small amount of space
    - Limited number of write cycles
    - Poor throughput between flash memory and CPU on mobile platform
- Instead use other methods to free memory if low
  - iOS ***asks*** apps to voluntarily relinquish allocated memory
    - Read-only data (such as code) thrown out and reloaded from flash if needed; data that have been modified (such as stack) are never removed.
    - Failure to free can result in termination
  - Android terminates apps if low free memory, but first writes **application state** to flash for fast restart
  - Both OSes support paging as discussed below

Contiguous Allocation

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually into two **partitions**:
  - 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
- 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*
  - Can then allow actions such as kernel code being **transient** (i.e., the code comes and goes as needed) and kernel changing size

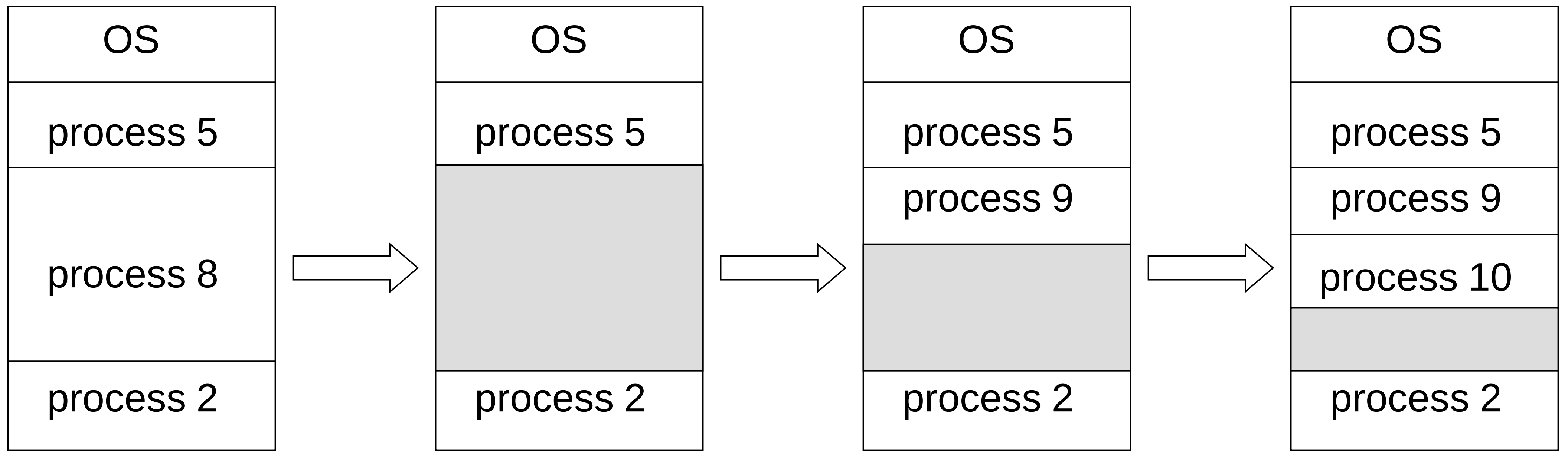
Hardware Support for Relocation and Limit Registers

The value in the **relocation register** is added to every address generated by a user process at the time it is sent to memory



Contiguous Allocation (Cont.)

- Multiple-partition allocation
  - Degree of multiprogramming limited by number of partitions
  - **Variable-partition** sizes for efficiency (sized to a given process' needs)
  - **Hole** – block of available memory; holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Process exiting frees its partition, adjacent free partitions combined
  - Operating system maintains information about:
    a) allocated partitions   b) free partitions (hole)

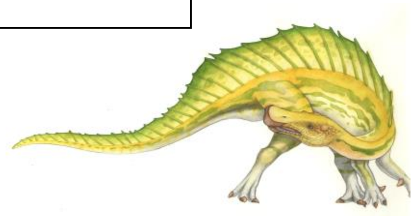


Dynamic Storage-Allocation Problem

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

- **First-fit**: Allocate the ***first*** hole that is big enough
- **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



Fragmentation

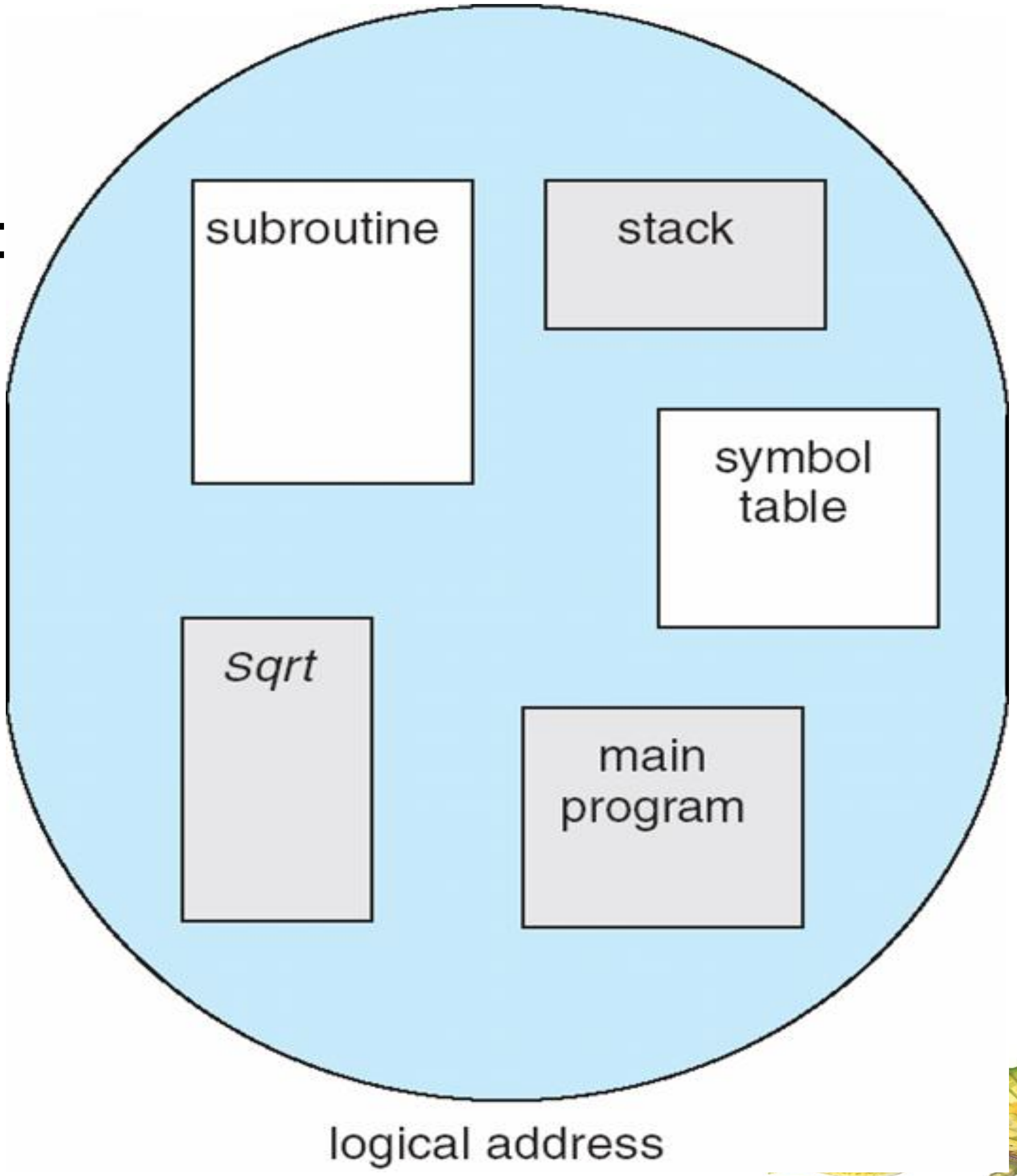
- **External Fragmentation** (caused by variable-sized allocation)
  - total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** (caused by fix-sized allocation)
  - allocated memory may be slightly larger than requested memory; this size difference results in unused memory that is internal to a partition
- First fit analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation ➔ **50-percent rule**
- Solution to external fragmentation ➔ **Compaction**
  - 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 run time
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers
- Another solution: **noncontiguous allocation** ➔ 1. segmentation 2. paging

Segmentation

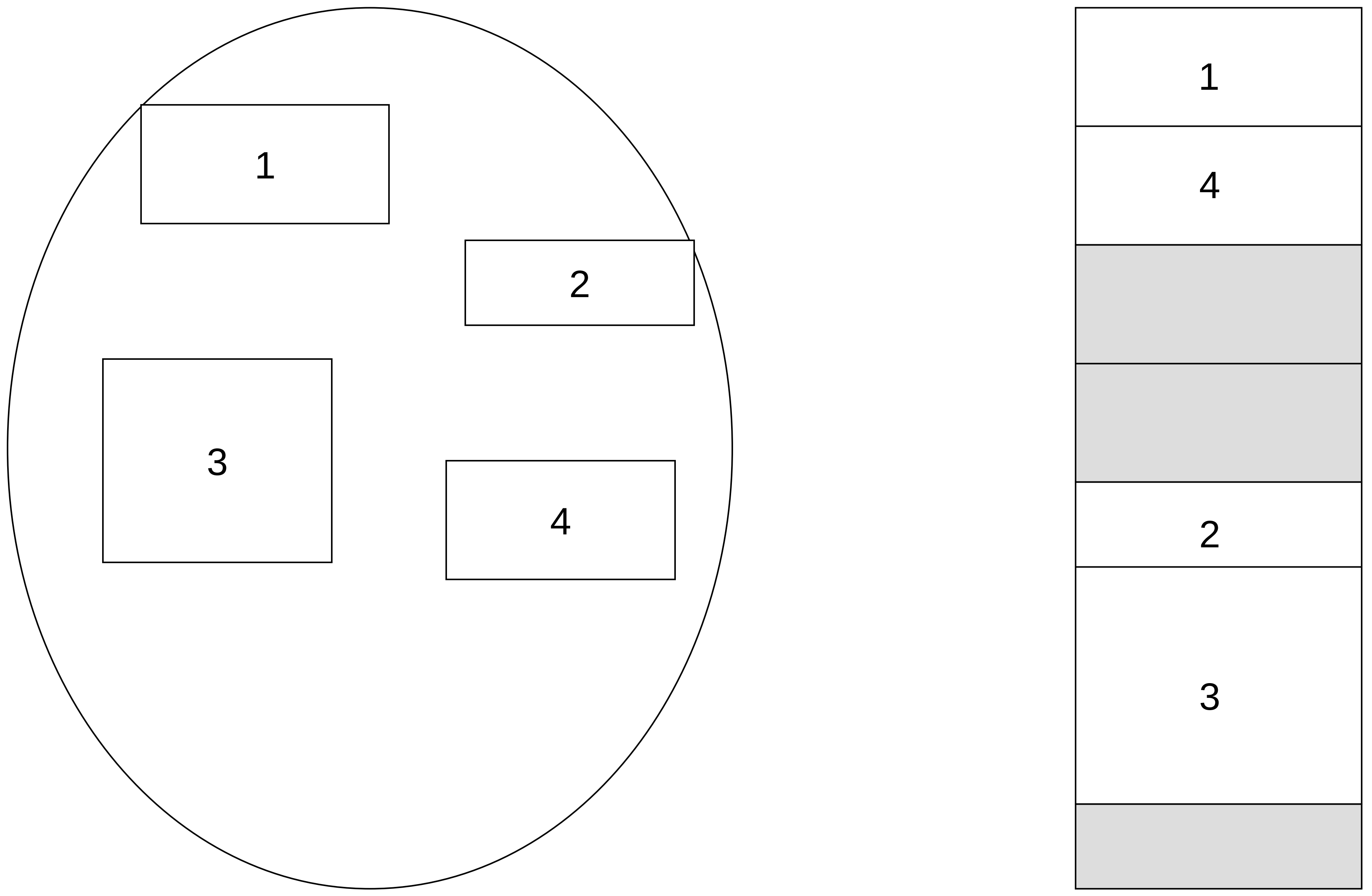
- Memory-management scheme that supports user view of memory
- A program is a collection of segments
  - A segment is a logical unit such as:

    main program
    procedure
    function
    method
    object
    local variables, global variables
    common block
    stack
    symbol table
    arrays

User's View of a Program



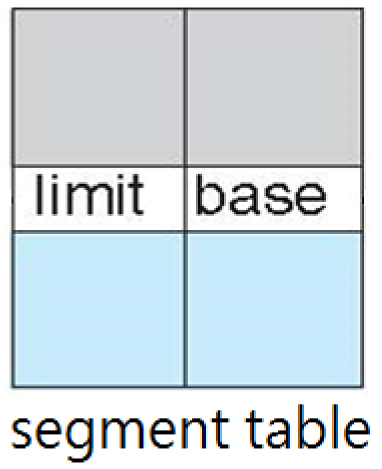
Logical View of Segmentation



user space   physical memory space

Segmentation Architecture

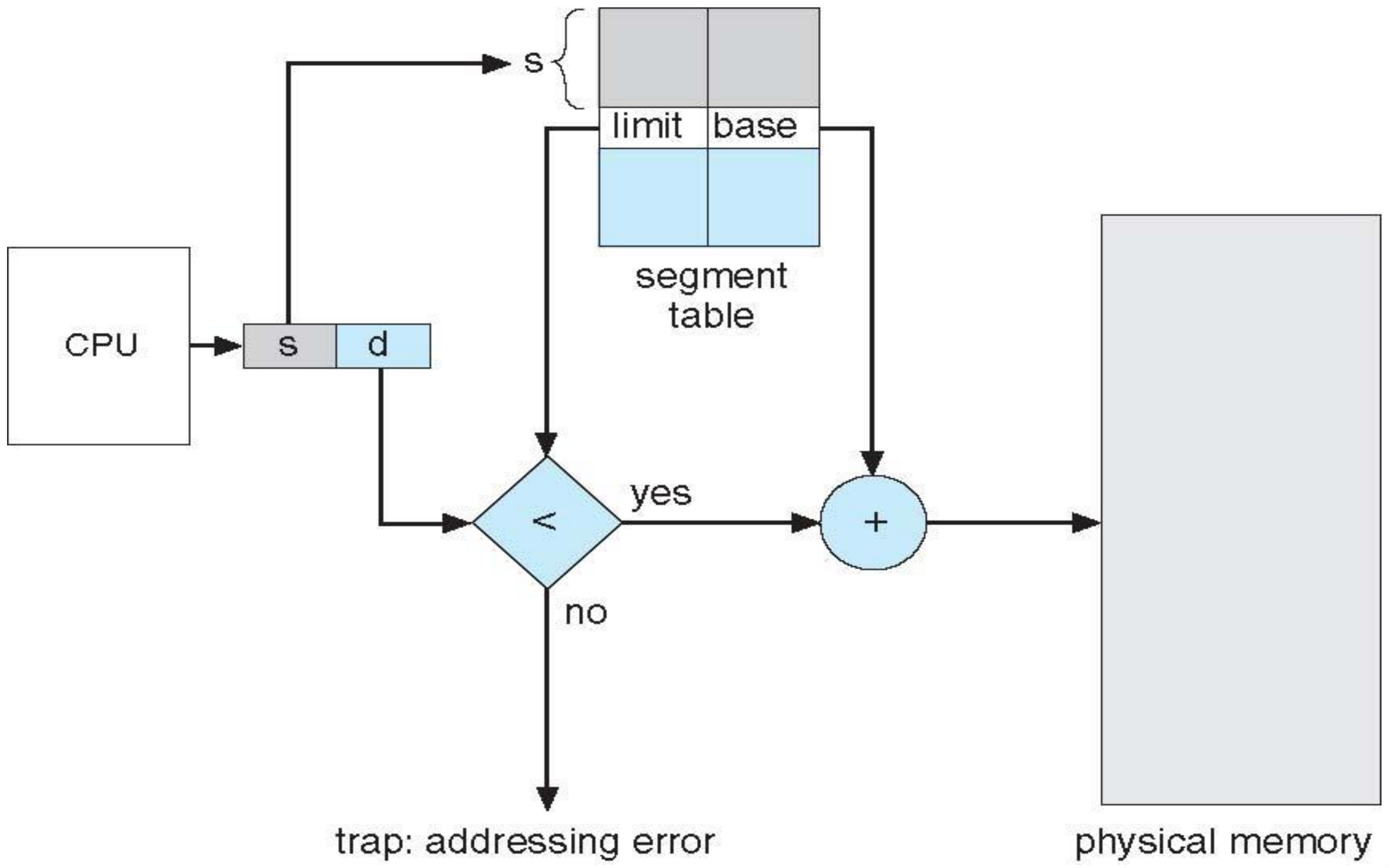
- Logical address consists of a two tuple: <segment-number, offset>
- When a program is compiled, the compiler constructs segments and the loader assigns them segment number. For example, a C compiler creates following segments
  - *Code, global variable, heap, stack, standard C library*
- **Segment table** – maps two-dimensional logical address to one-dimentional 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



Segmentation Architecture (Cont.)

- **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**
- **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
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram

Segmentation Hardware



Chapter 8: Memory Management Strategies

- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table
- Example: The Intel 32 and 64-bit Architectures
- Example: ARM Architecture

Paging

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

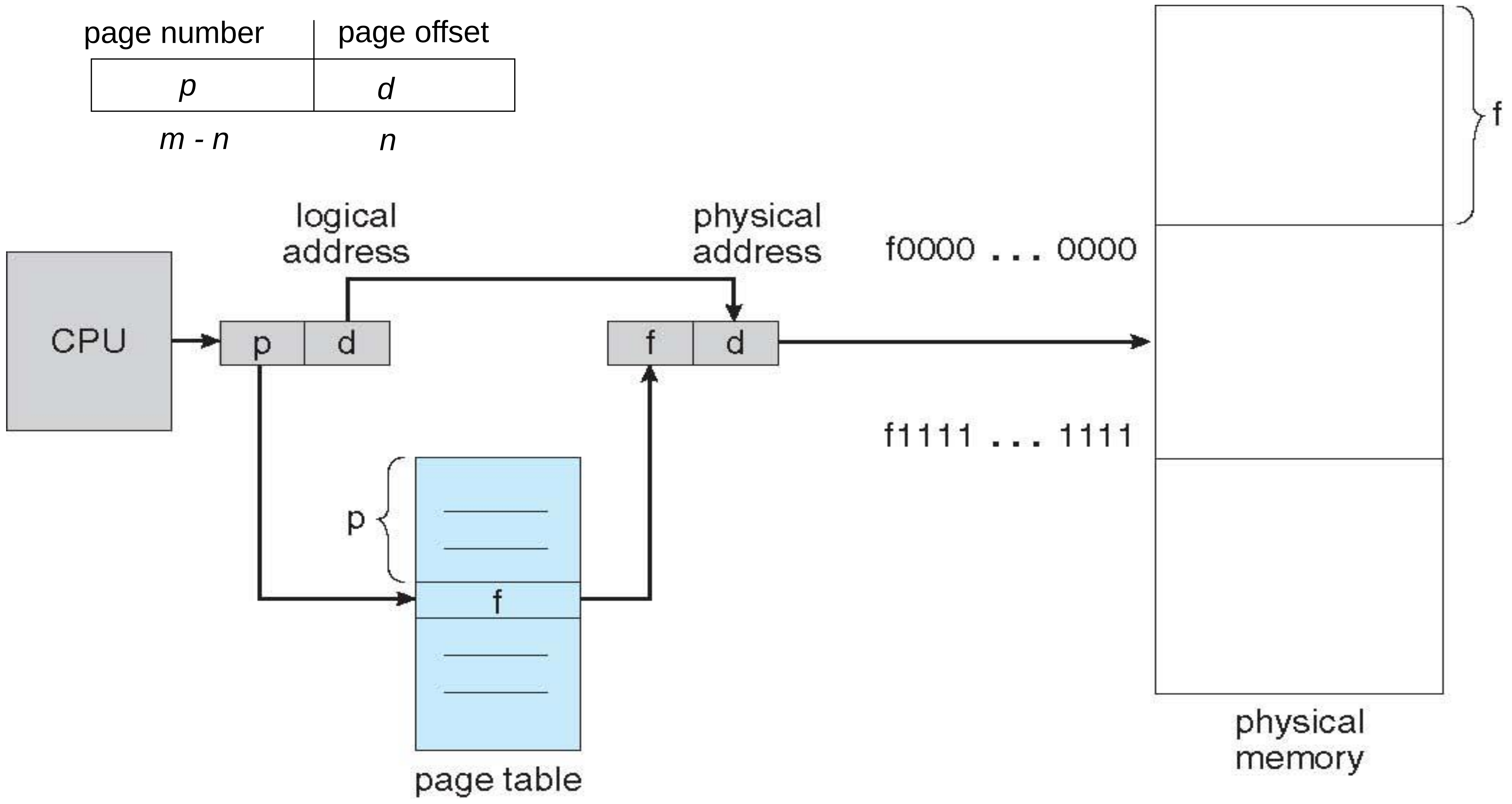
Address Translation Scheme

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

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

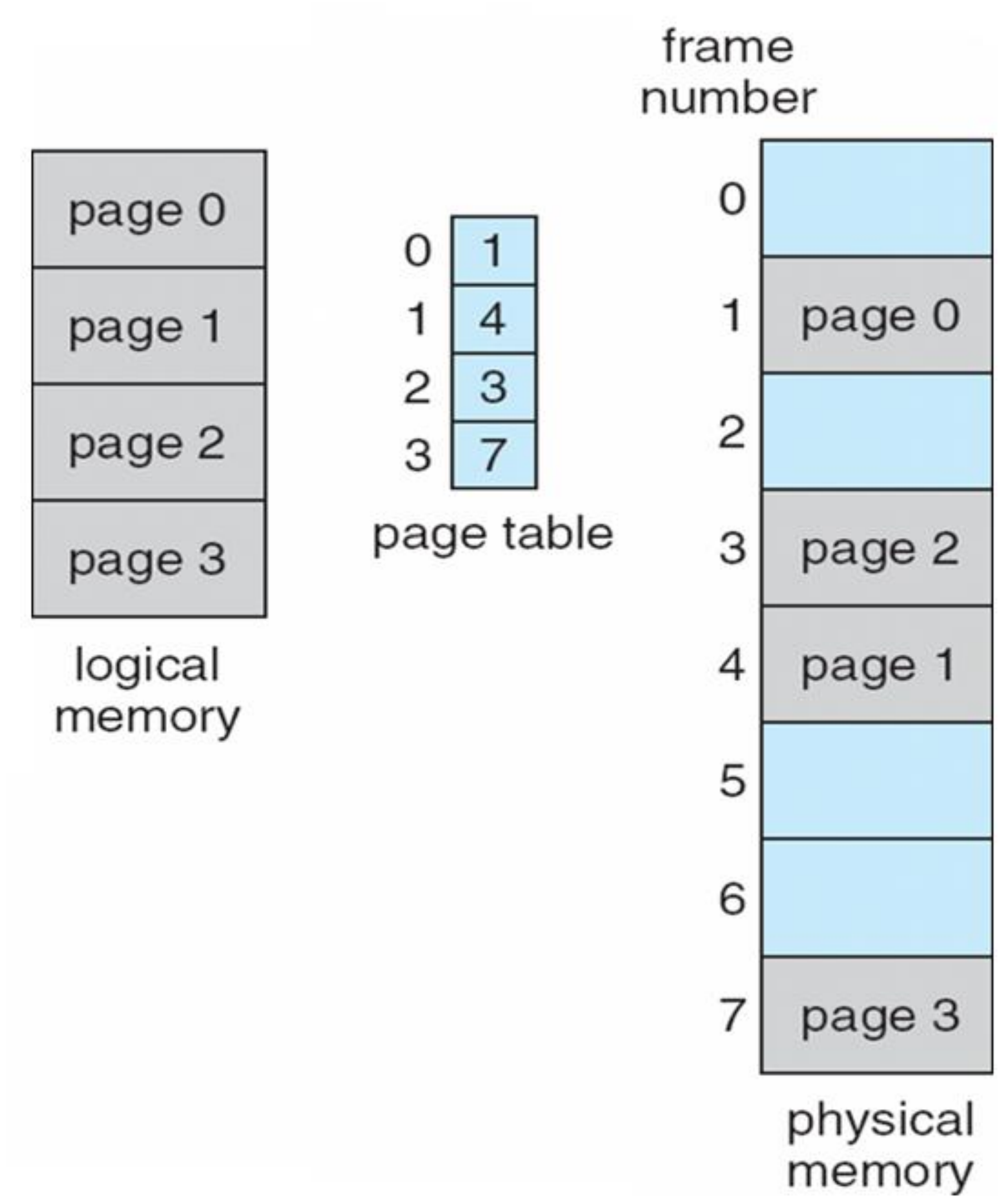
  - For given logical address space 2^m and page size 2^n

Paging Hardware



Paging Model

- Separation of programmer's view of memory and the actual physical memory.
  - The programmer's view: **logical address space** (i.e., view memory as one single space, containing only this program)
  - Actually, the program is scattered throughout memory in **physical address space**
- The mapping of logical address to physical address is provided by
  - address-translation hardware
  - OS (maintains page tables)



Paging Example

Logical address:
n=2, m=4
Page size:
4 bytes
Physical memory:
32 bytes (8 pages)

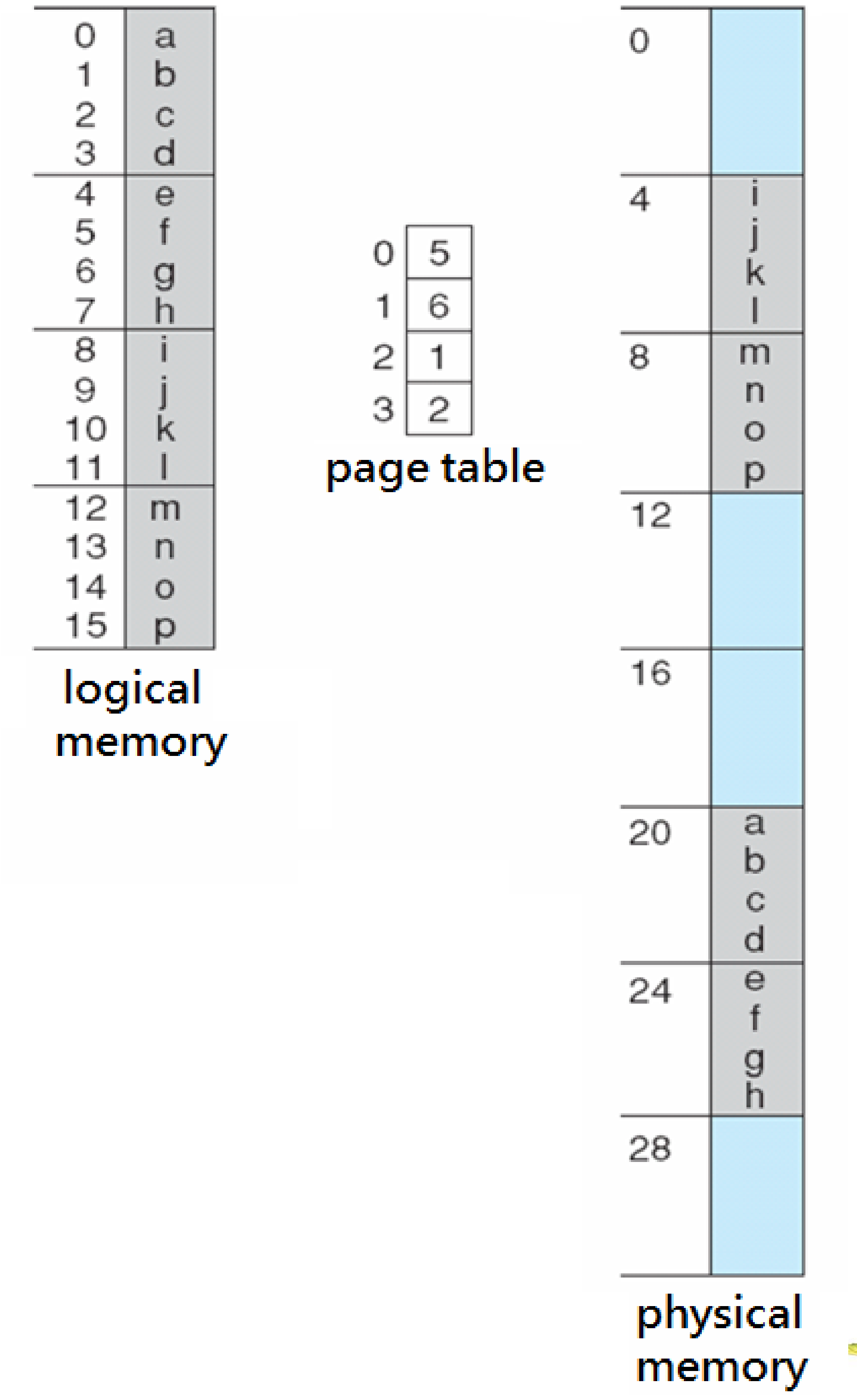
- Logical address : **0** (page 0, offset 0)

  ➔ Physical addr (5x4 + 0) = 20
- Logical addr : **3** (page 0, offset 3)

  ➔ Physical addr: (5x4 + 3) = 23
- Logical addr: **4** (page1, offset 0)

  ➔ Physical addr: (6x4 + 0) = 24
- Logical addr: **13**

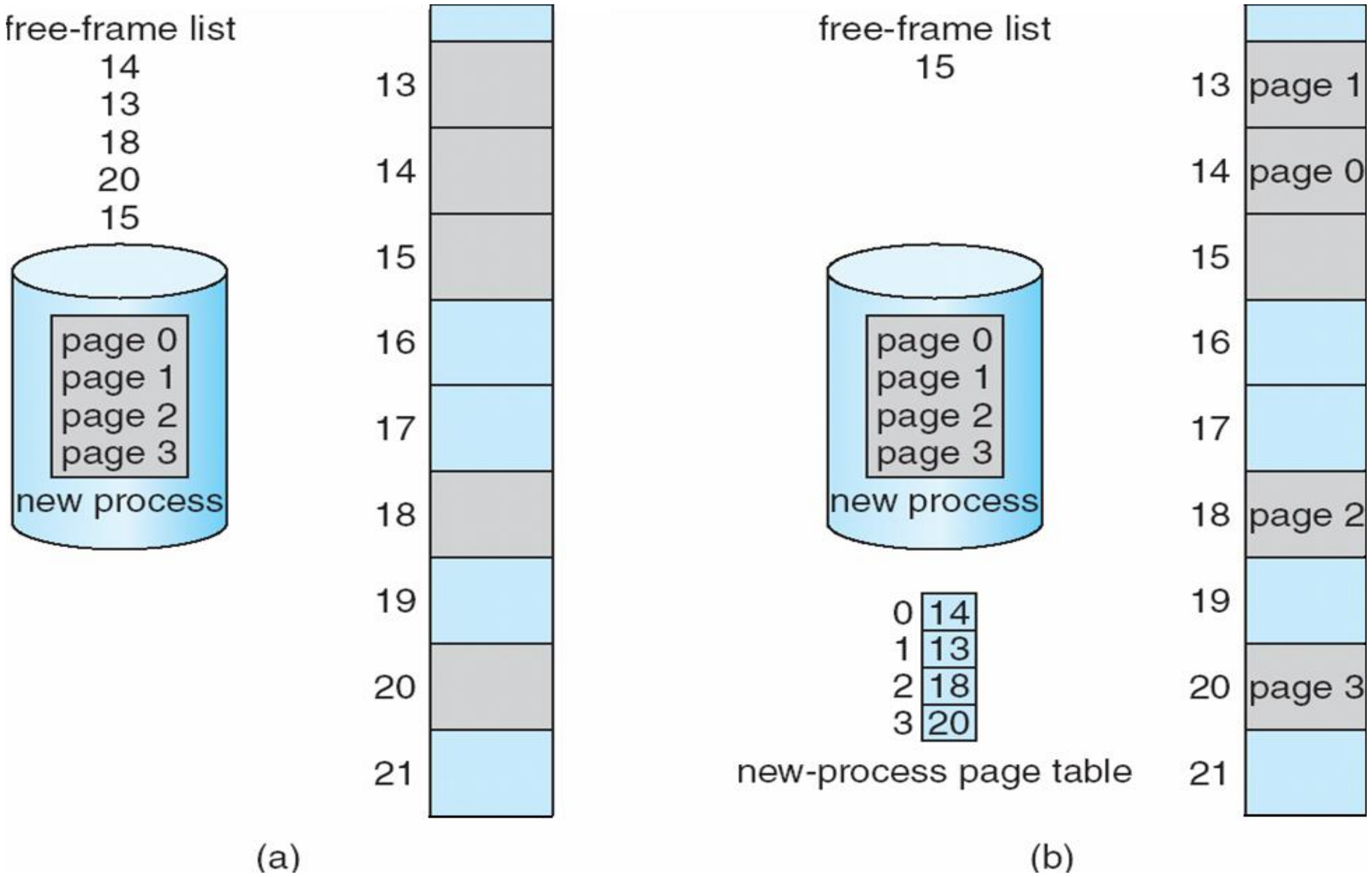
  ➔ Physical addr : ?



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

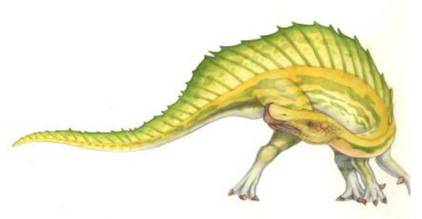
Free Frames



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



Translation Look-aside Buffer (TLB)

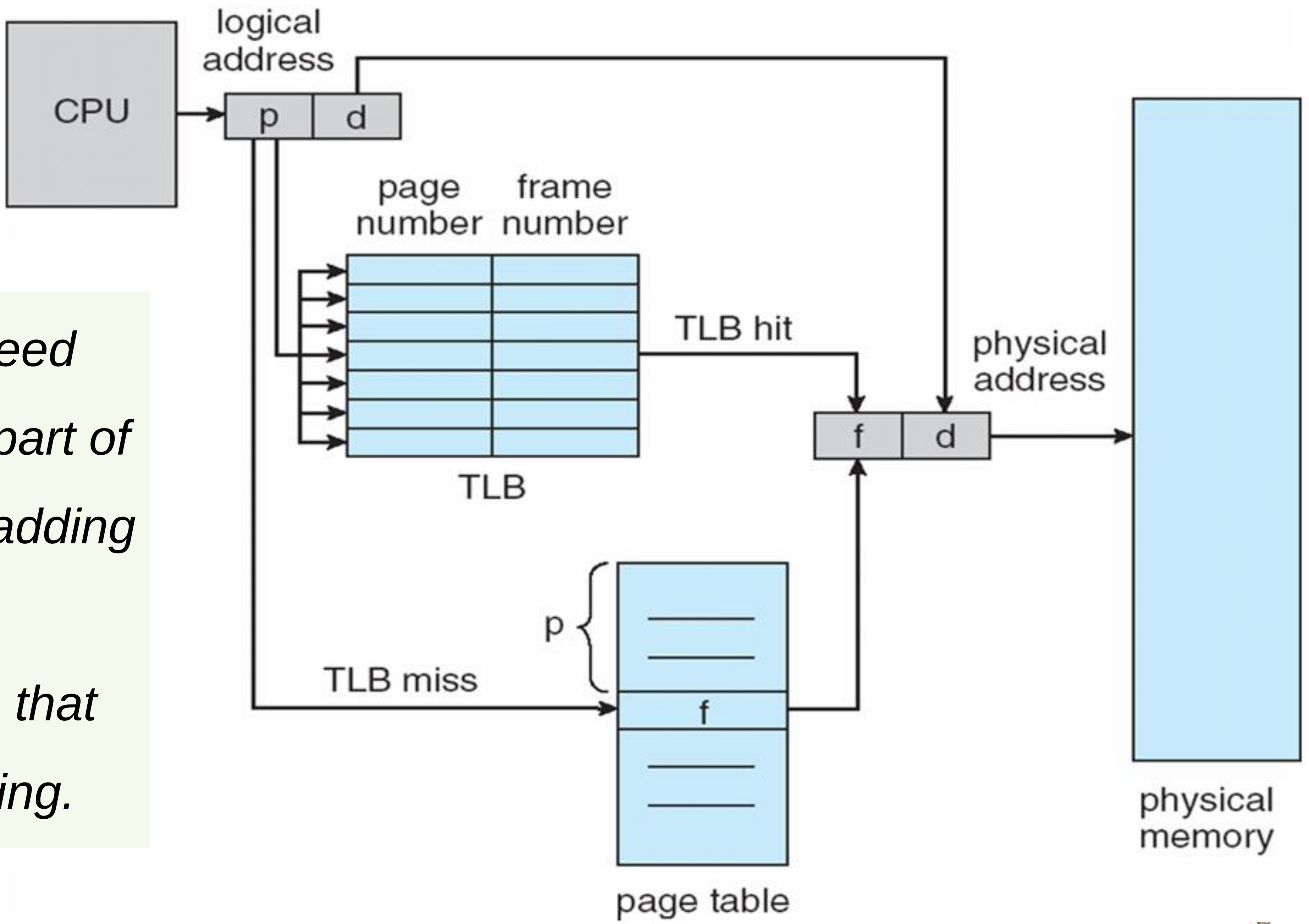
- TLB – parallel search

Page #	Frame #

- Address translation (p, d)
  - If p is in TLB, get frame # out
  - Otherwise get frame # from page table in memory
- TLB contains only a few of page table entries

Paging Hardware With TLB

- Page table is kept in main memory
- TLB must be small, containing only a few of page table entries



TLB is a special high-speed hardware which can be part of the instruction pipeline, adding no performance penalty compared with a system that does not implement paging.

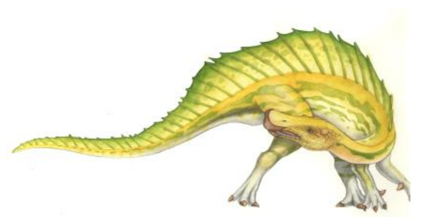
Effective Access Time

- TLB Lookup = ε time unit
  - Can be < 10% of memory access time
- 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**)
  - Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
    - EAT = 0.80 x 100 + 0.20 x 200 = 120ns
  - Consider more realistic hit ratio -> α = 99%, ε = 20ns for TLB search, 100ns for memory access
    - EAT = 0.99 x 100 + 0.01 x 200 = 101ns

Memory Protection

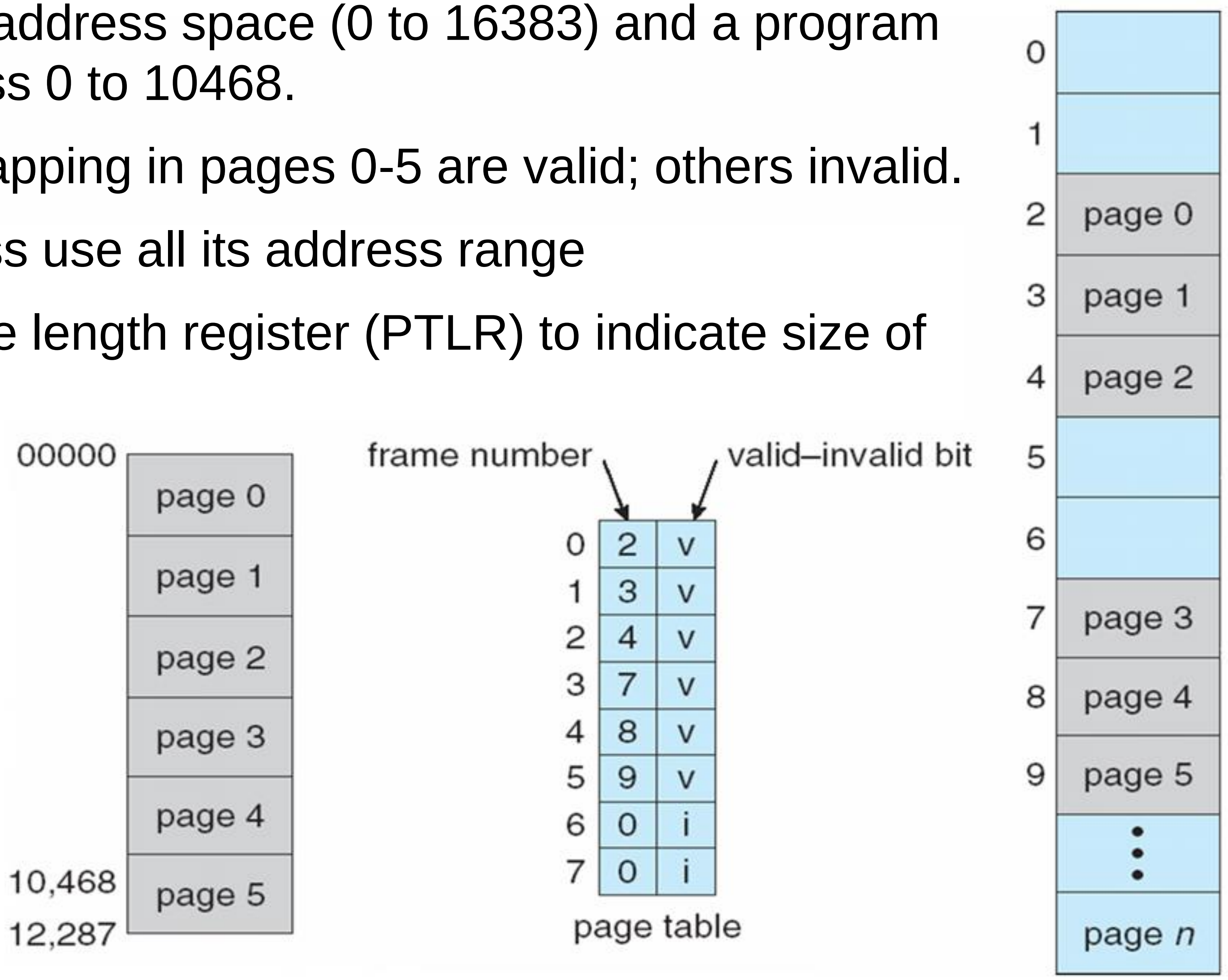
- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
  - Can also add more bits to indicate page execute-only, and so on
- **Valid-invalid** bit attached to each entry in the page table:
  - “valid” indicates that the associated 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

- A system with 14-bit address space (0 to 16383) and a program that uses only address 0 to 10468.
- Page size 2KB ➔ mapping in pages 0-5 are valid; others invalid.
- Rarely does a process use all its address range

    ➔ use a page-table length register (PTLR) to indicate size of page table.

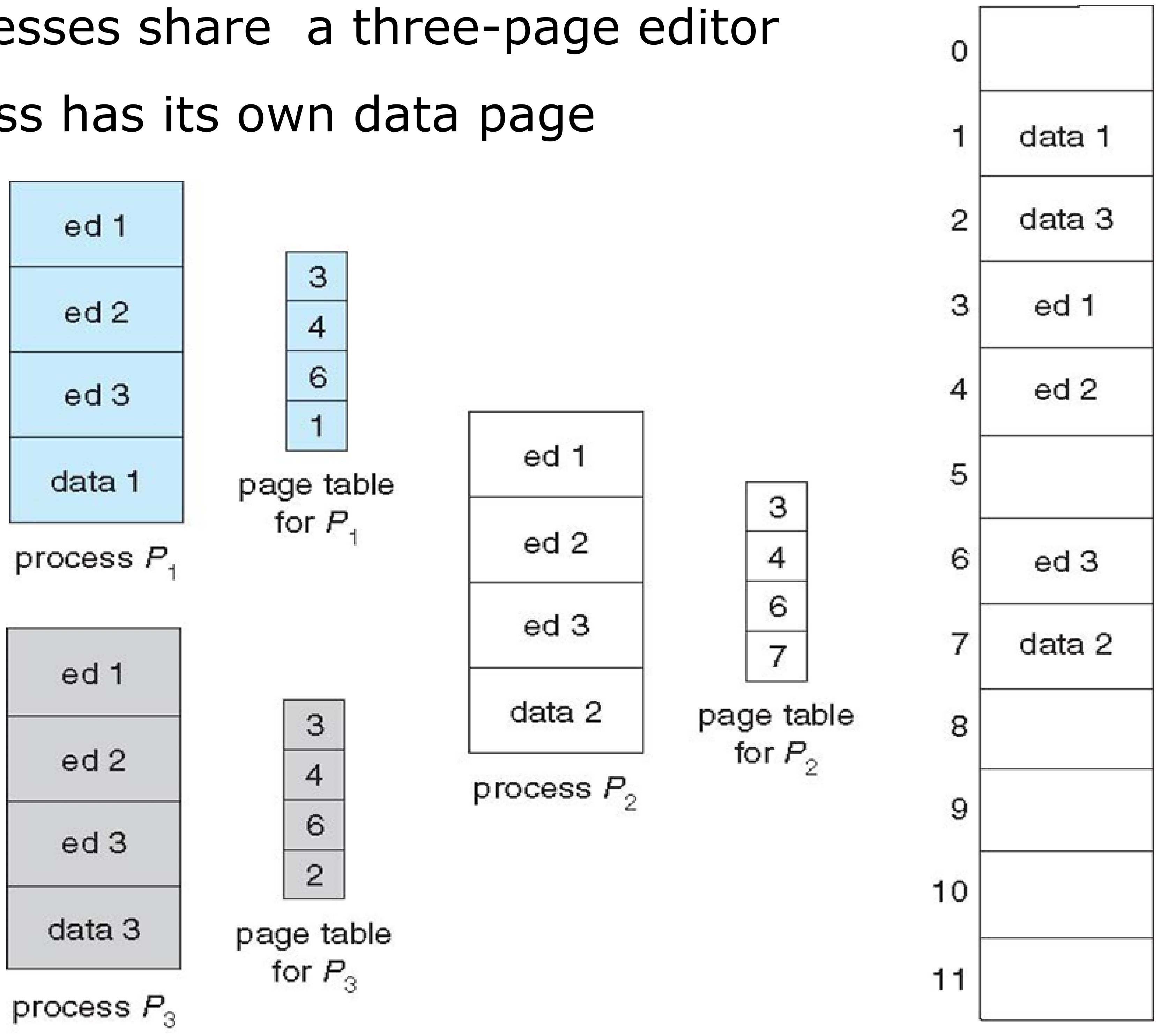


Shared Pages

- **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 inter-process 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

- Three processes share a three-page editor
- Each process has its own data page



Chapter 8: Memory Management Strategies

- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table
  - **Hierarchical Paging**
  - **Hashed Page Tables**
  - **Inverted Page Tables**
- Example: The Intel 32 and 64-bit Architectures
- Example: ARM Architecture

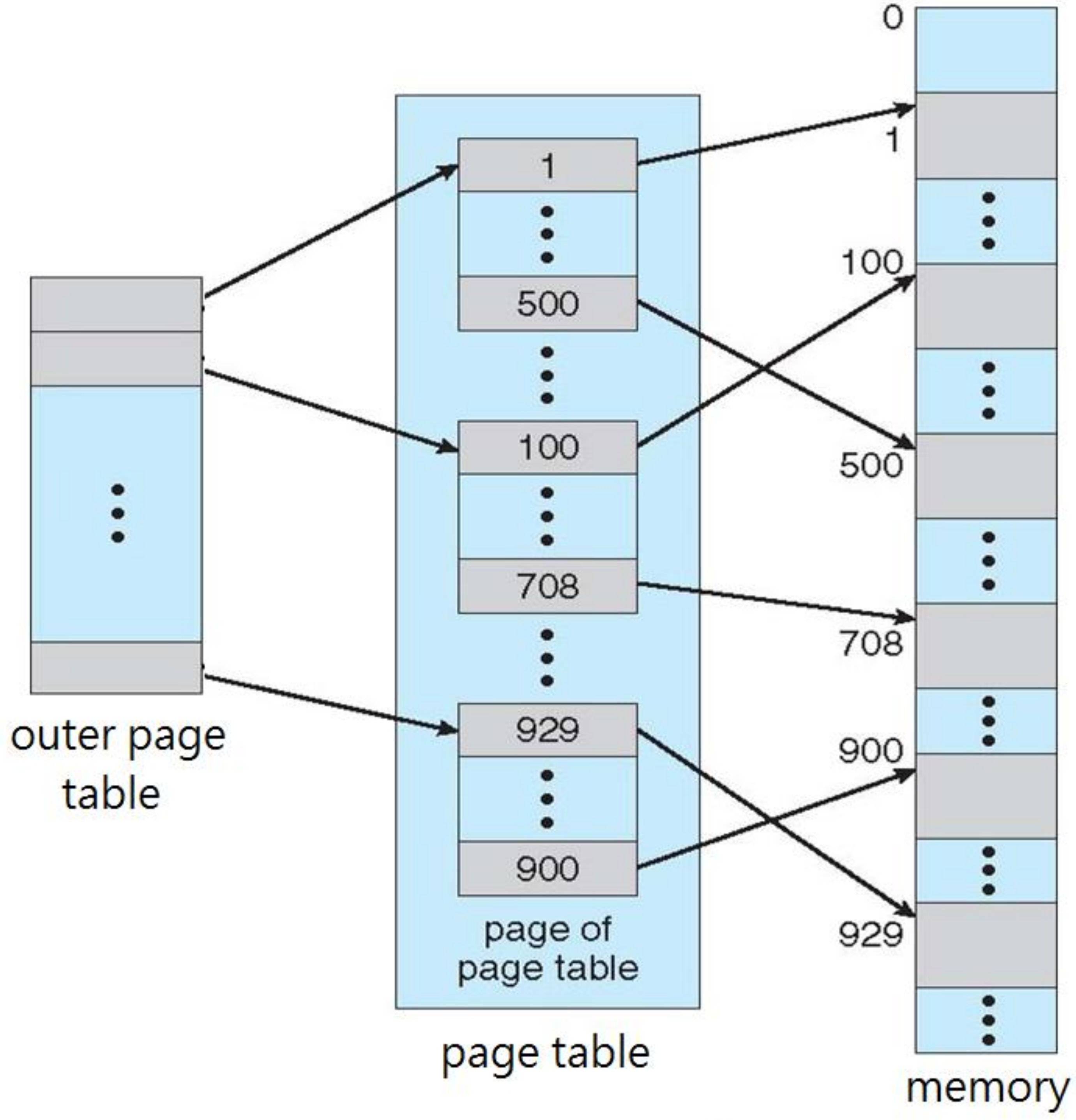
Structure of the Page Table

- 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
- **Hierarchical Paging**
- **Hashed Page Tables**
- **Inverted Page Tables**

Hierarchical Page Tables:
Two-Level Paging Example

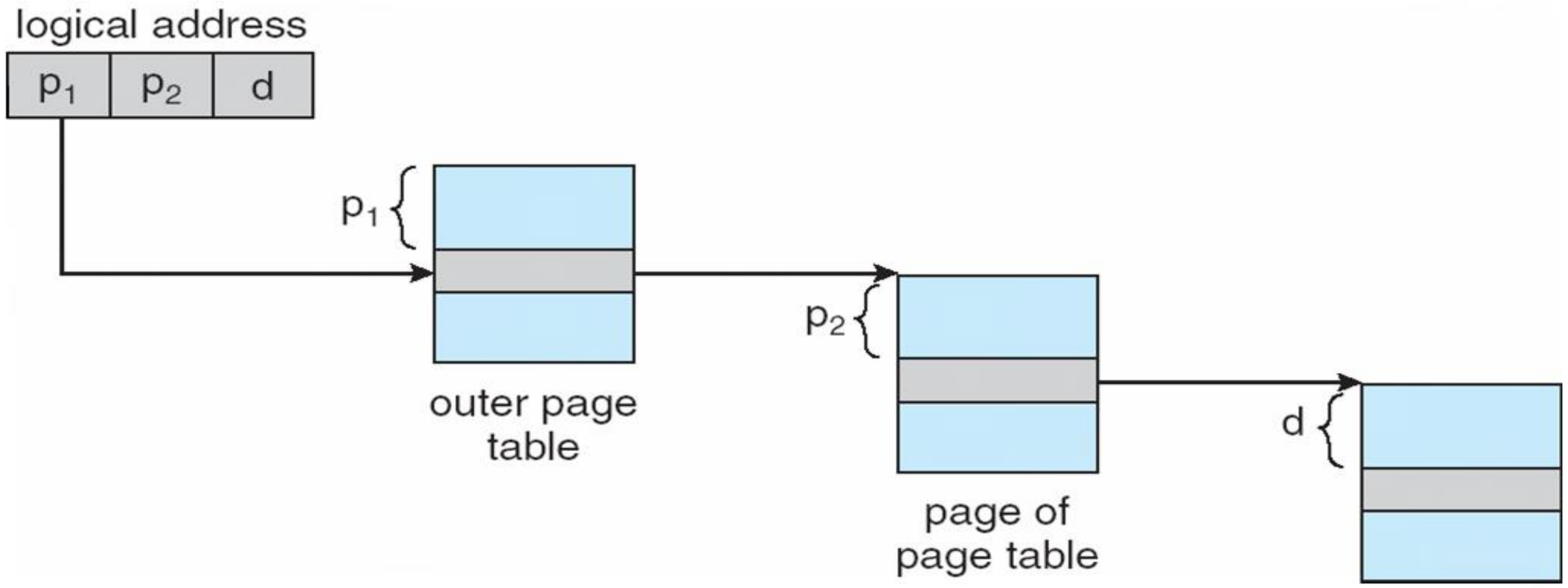
- Break up the logical address space into multiple page tables and the page table itself is also paged
- e.g., **Two-level page table**
- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bit
  - a page offset consisting of 10 bits
- Since the 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:

page number		page offset
p_1	p_2	d
12	10	10



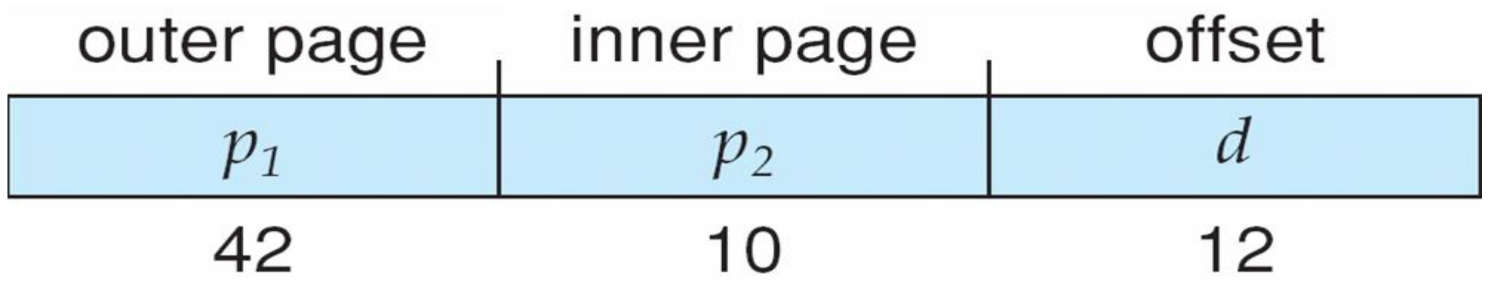
Address-Translation Scheme

- In two-level page-table scheme, p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- The address translation works from the outer page table inward, known as **forward-mapped page table**

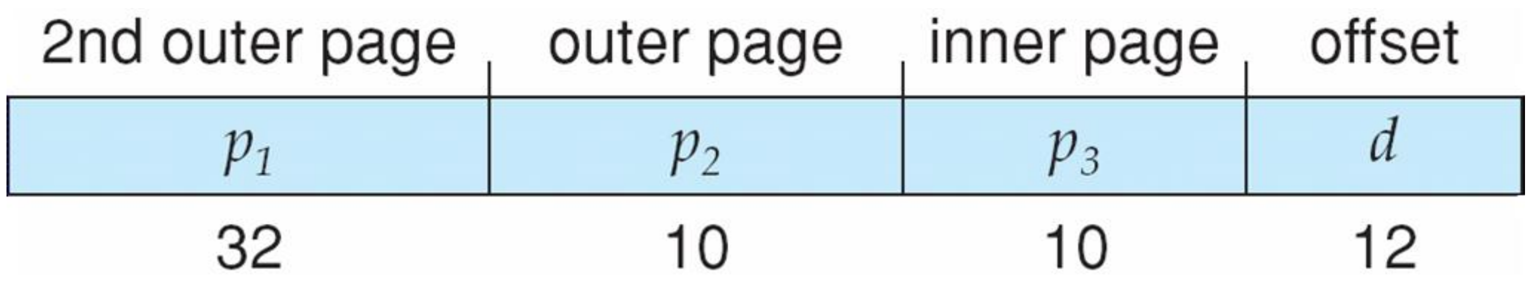


64-bit Logical Address Space

- 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 ➔ still too big.
  - **Solution: add a 2nd outer page table ➔ Three-level Paging Scheme**

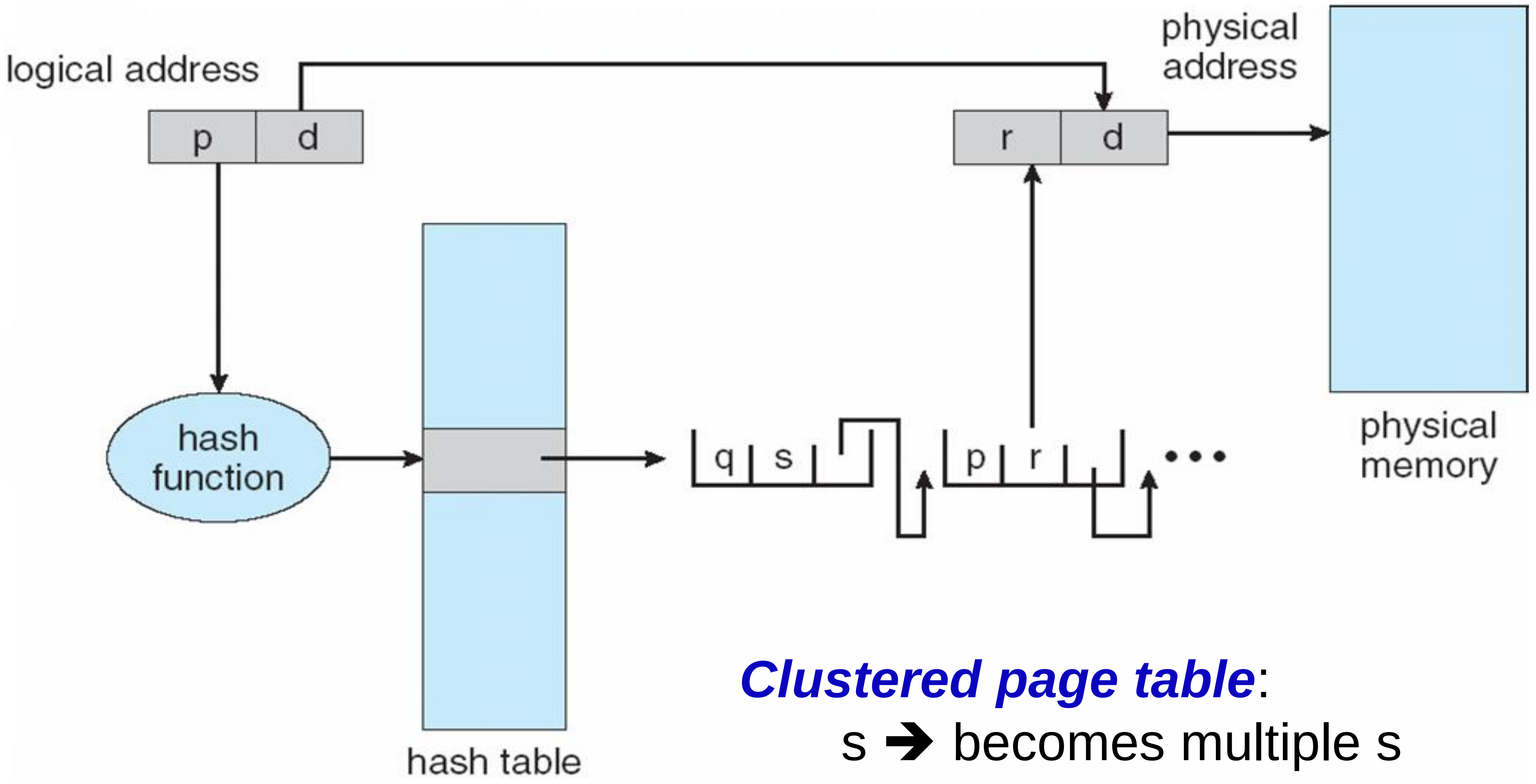


    - **The 2nd outer page table is still 2^{34} bytes in size**
    - **4 memory access to get to one physical memory location**

Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - The page table contains a chain of elements hashing to the same location (see next slide)
- 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



Clustered page table:

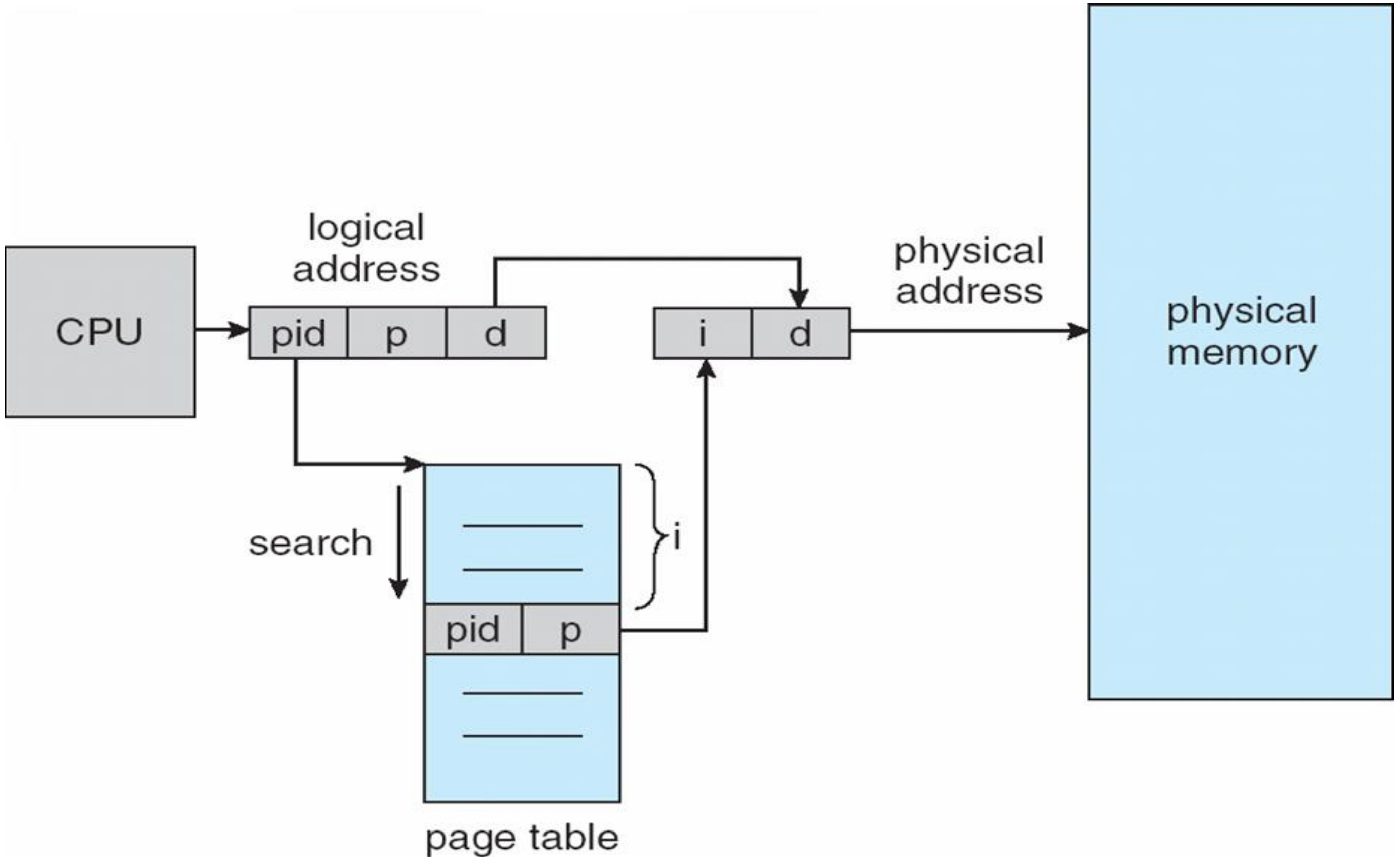
s ➔ becomes multiple s

r ➔ becomes multiple r

Inverted Page Table

- Rather than each process having a page table, **one inverted page table** in the system tracks 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
  - Inverted page-table entry: **<process-id, page-number>**
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
  - When a memory reference occurs, part of virtual address consisting of <process-id, page-number> is used to find a match in the inverted page table. Since the table is sorted by physical address, the whole table might need to be search before a match is found.
- 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?
  - One mapping of a virtual address to the shared physical address

Inverted Page Table Architecture



Chapter 8: Memory Management Strategies

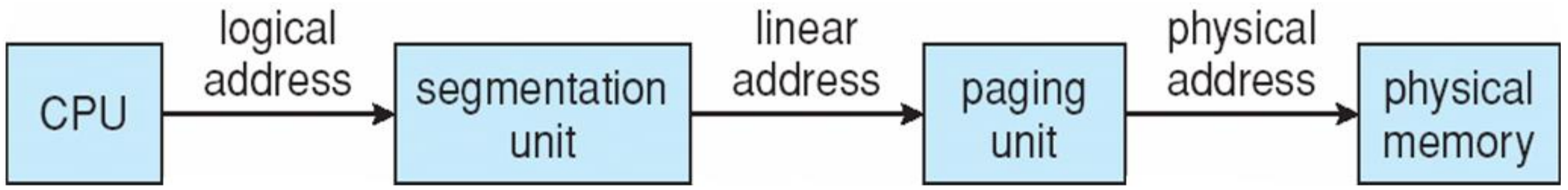
- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table
- Example: The Intel 32 and 64-bit Architectures
- Example: ARM Architecture

Example: The Intel 32 and 64-bit Architectures

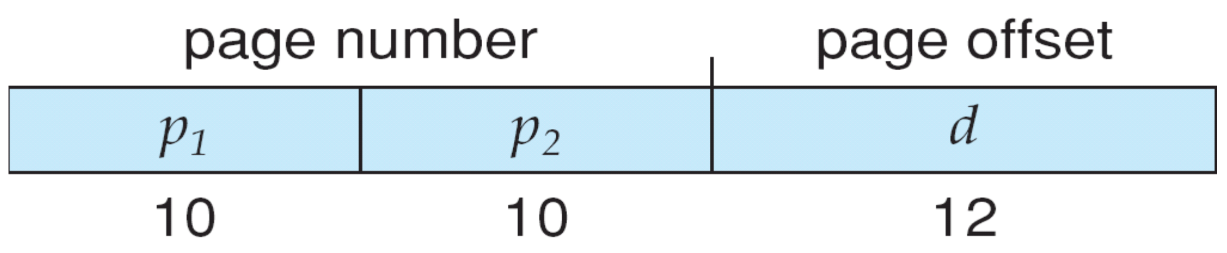
- Dominant industry chips
- Pentium CPUs are 32-bit and called IA-32 architecture
- Current Intel CPUs are 64-bit and called IA-64 architecture
- Many variations in the chips, cover the main ideas here

Example: The Intel IA-32 Architecture

- Memory management in IA-32 system is divided into segmentation and paging

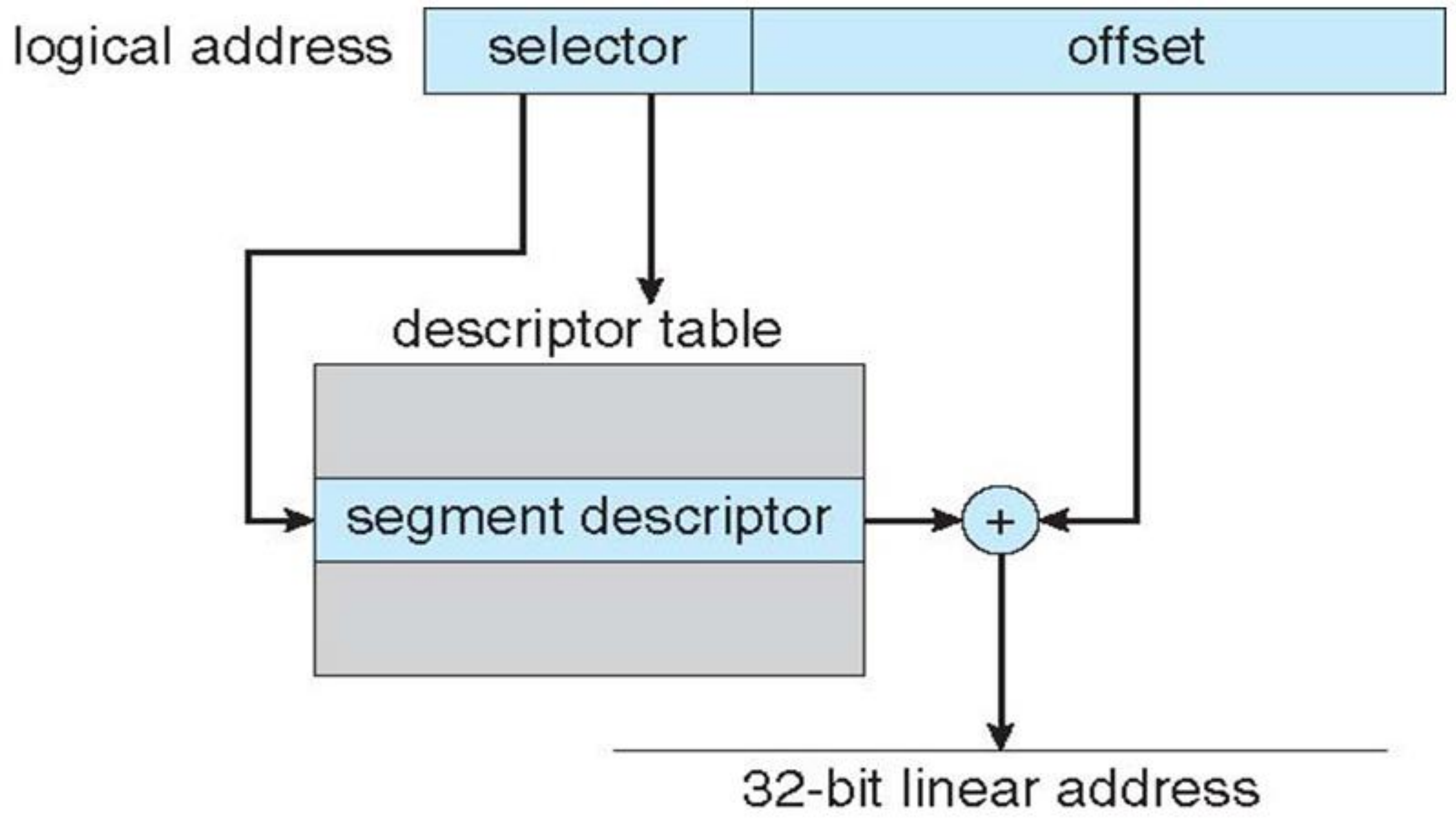


- CPU generates logical address which is a pair (selector , offset)
  - Selector given to segmentation unit
    - Which produces linear addresses
  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Pages sizes can be 4 KB or 4 MB
    - For 4KB pages, IA-32 uses a two-level paging scheme in which the division of 32bit linear address is as follows



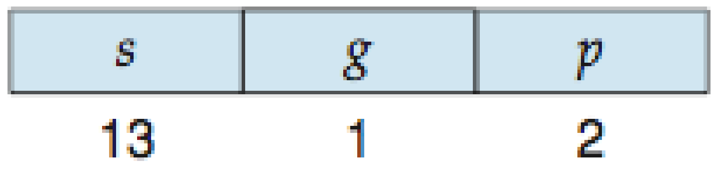
Intel IA-32 Segmentation

- 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:
    - 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**))
  - Entry of LDT/GDT: segment descriptor (base location, limit of the segment)



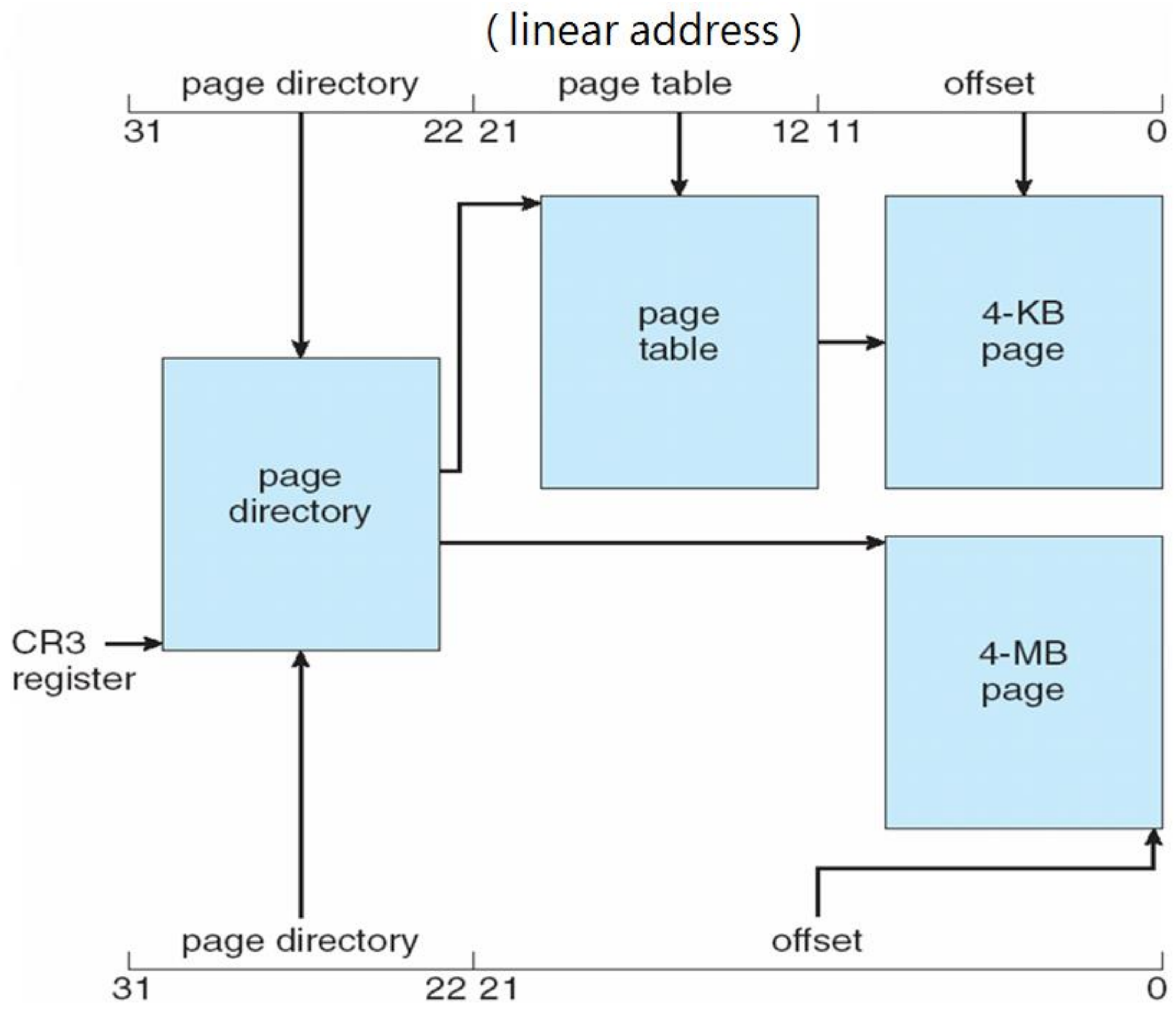
- CPU generates logical address: a pair (selector , offset)

  - Offset: 32 bits
  - Selector: 16 bits



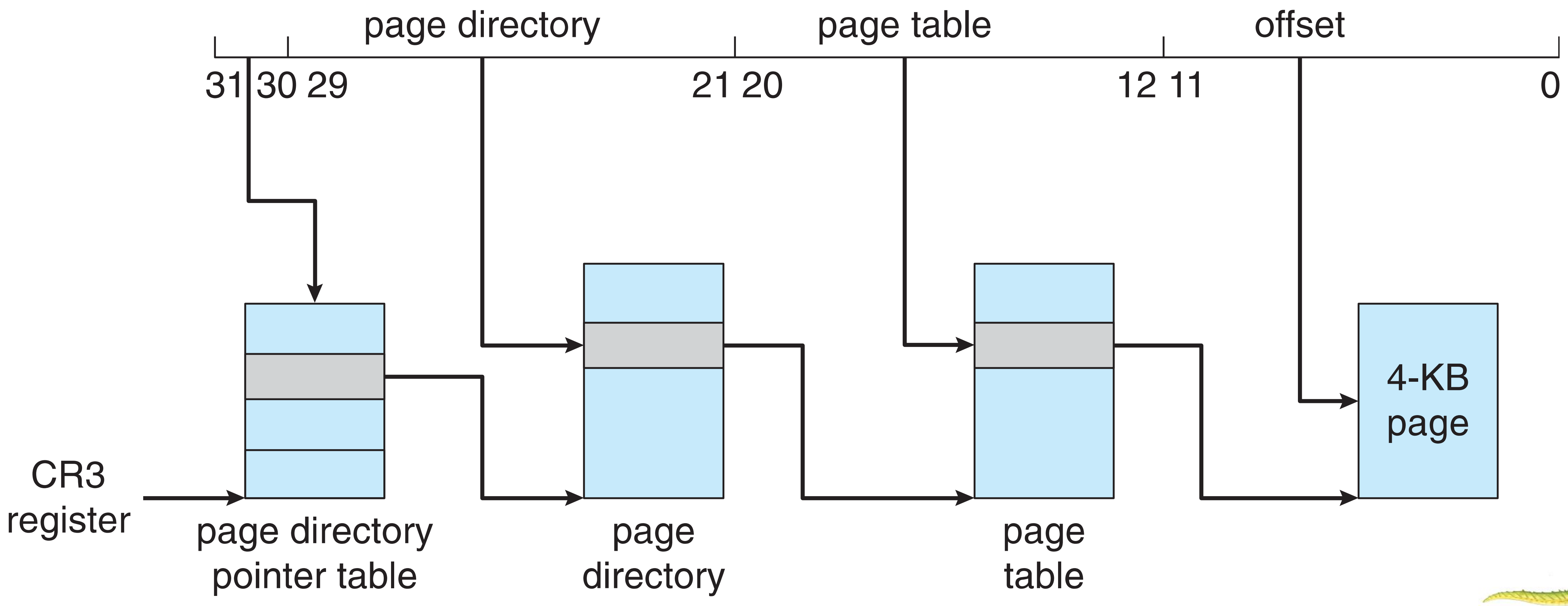
s: segment #
g: LDT or GDT
p: protection

Intel IA-32 Paging Architecture



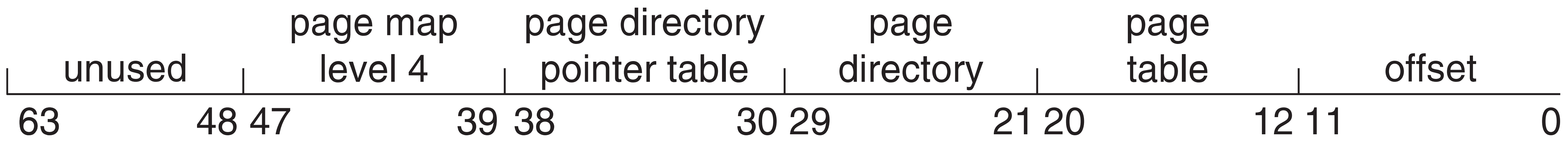
Intel IA-32 Page Address Extensions

- 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 from 20 to 24-bits
  - Net effect is increasing address space to 36 bits – 64GB of physical memory



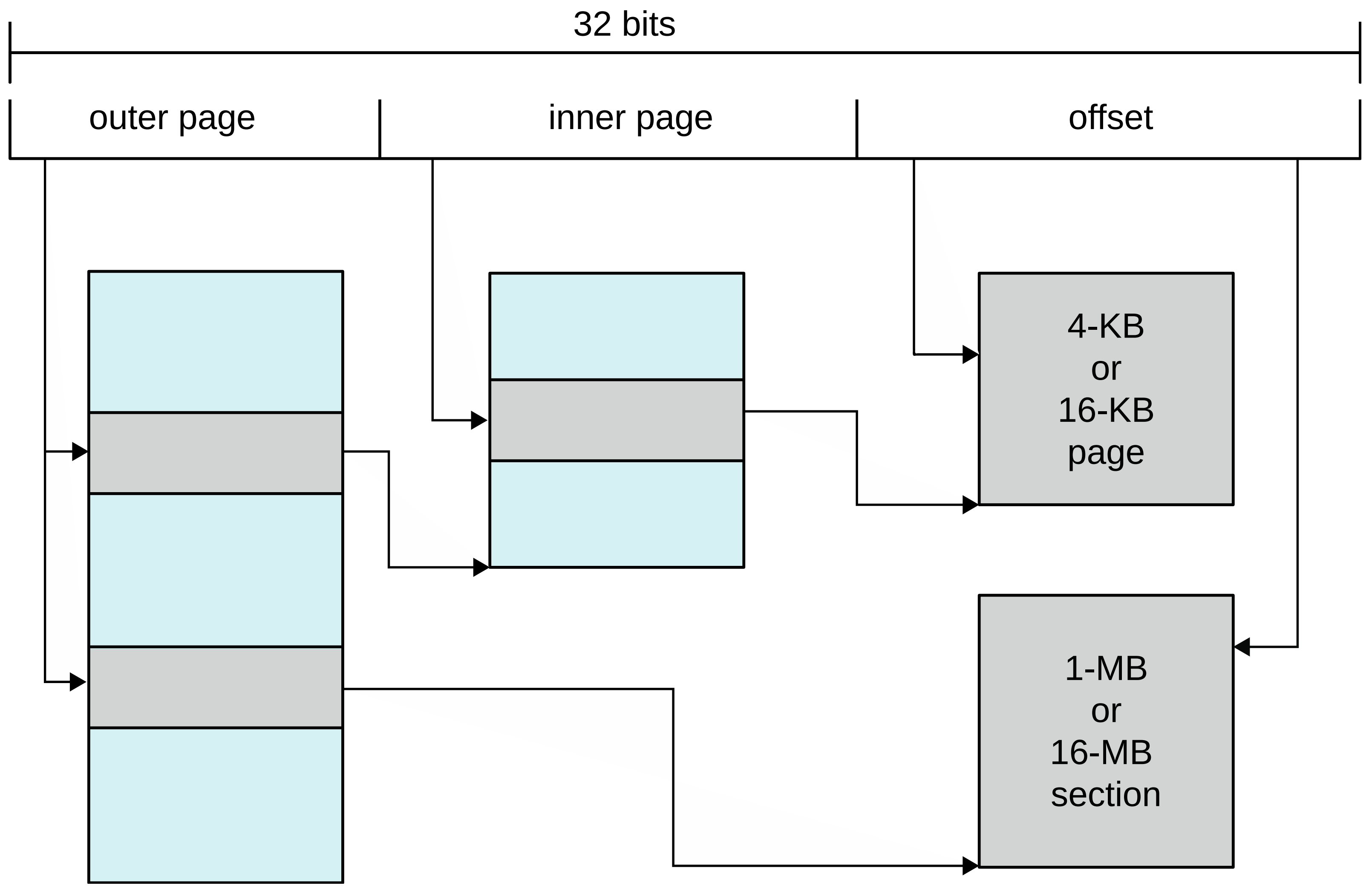
Intel x86-64

- 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 paging for smaller pages
- TLBs: two levels
  - Outer level has two micro TLBs (one data, one instruction)
  - Inner is single main TLB
  - Address translation begins at micro TLB. In the case of a miss, main TLB is then checked. If both yield misses, page table walk performed by CPU.


32 bits
outer page
inner page
offset
4-KB or 16-KB page
1-MB or 16-MB section