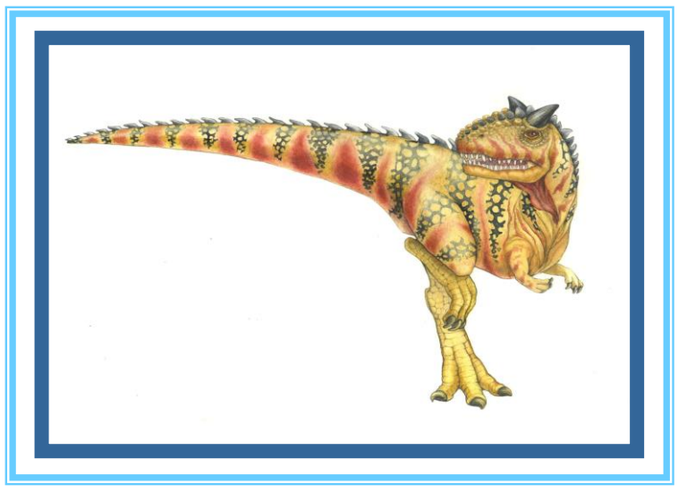
Chapter 8: Memory-Management Strategies



Chapter 8: Memory Management Strategies

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

Objectives

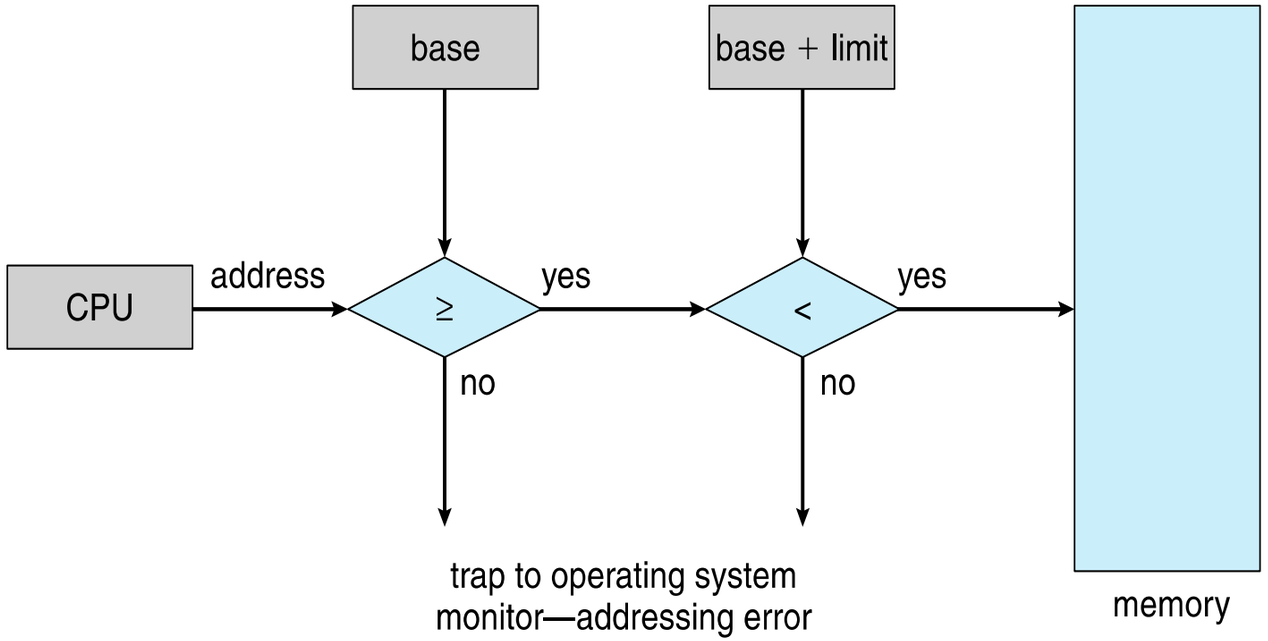
- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging

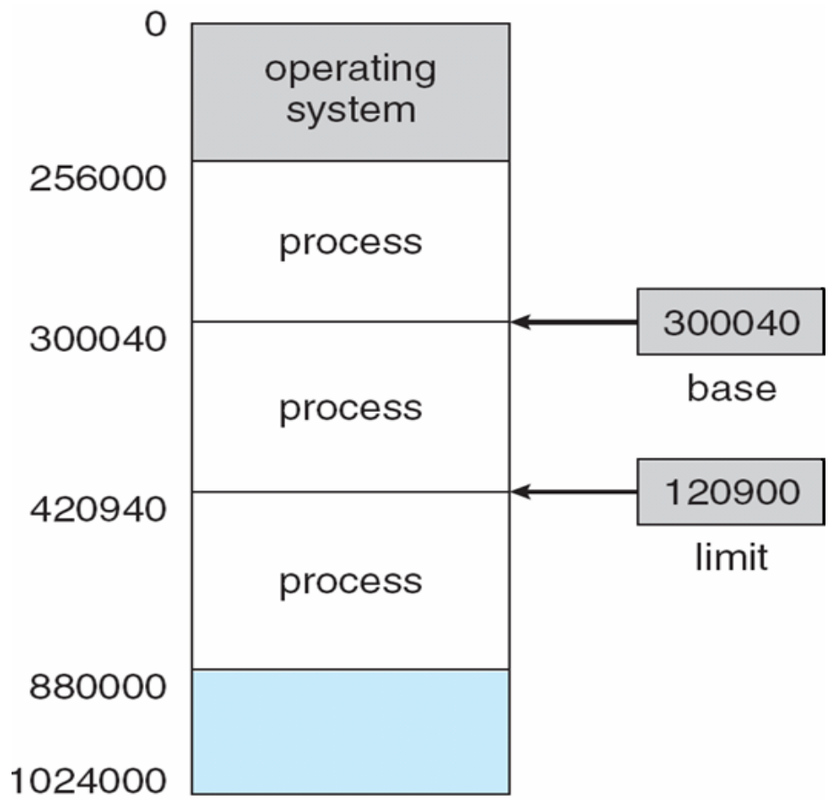
Background

- Program must be brought (from disk) into memory and placed within a process for it to run
- Main memory (including cache) and registers are the only storage CPU can access directly; in another word, CPU cannot access secondary storage (disk) directly, but through I/O controllers
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register can be accessed in one CPU clock (cycle)
- Accessing main memory may take many CPU cycles, causing a **memory stall,** when it does not yet have the data required to complete the instruction it is executing
- **Cache** sits between main memory and CPU registers, on the CPU chip for fast access
- Protection of memory required to ensure correct operation

Hardware Address Protection with Base and Limit Registers

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





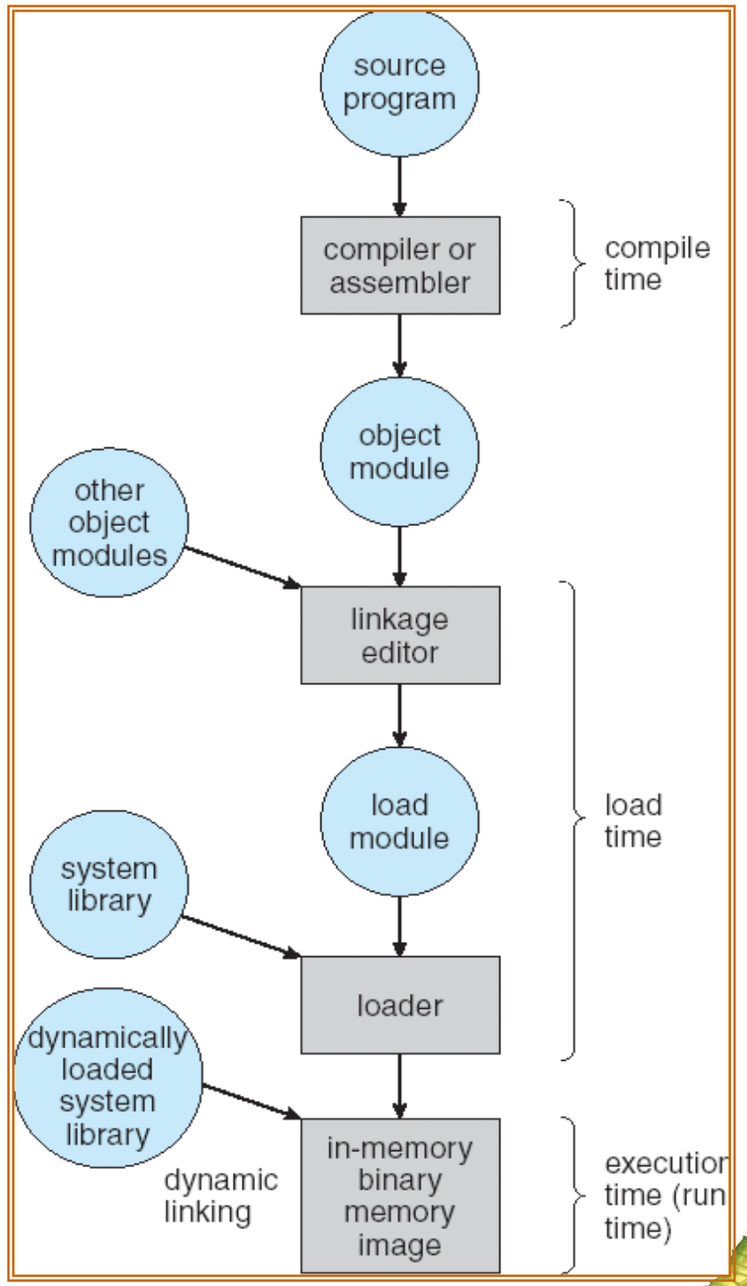
Address Binding

- Program must be brought into memory and placed within a process for it to be run
- **Input queue** – collection of processes on the disk that are waiting to be brought into memory to run the program – Long-term scheduler
  - Without proper support, it will be loaded into address 0000
- Inconvenient to have the first user process physical address always at 0000
  - How can it not be?
- Further, addresses represented in different ways at different stages of a program's life
  - Source code addresses usually symbolic
  - Compiled code addresses **bind** to relocatable addresses
    - i.e. "14 bytes from beginning of this module"
  - Linker or loader will bind relocatable addresses to absolute addresses
    - i.e. 74014
  - Each binding maps one address space to another address space

Multistep Processing of a User Program

Address binding of instructions and data to memory addresses can happen at three different stages

- **Compile time**: If memory location is known a priori, absolute code can be generated; must recompile code if starting location changes (e.g., “gcc”). MS-DOS uses this
- **Link or Load time**: Compiler must generate relocatable code if memory location is not known at compile time (e.g, Unix “ld” does link). The binding is delayed until load time; we need only reload the user code to incorporate this changed value
- **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. This needs hardware and operating system support for address maps (e.g., base and limit registers), e.g., dynamic libs. Most general-purpose operating systems use this method



Address Translation and Protection

- In the old days of uni-programming, only one program runs at the time, it occupies the entire physical memory
  - No address translation needed and protection

- In the earlier stage of multi-programming such as MS Window 3.1 (1990-1992)
  - No address translation, binding occurs at link or loader time – address adjusted when programs are loaded into memory. Bugs in any program can crash the system
  - Protection was added using base and limit registers, preventing illegal access

Logical vs. Physical Address Space

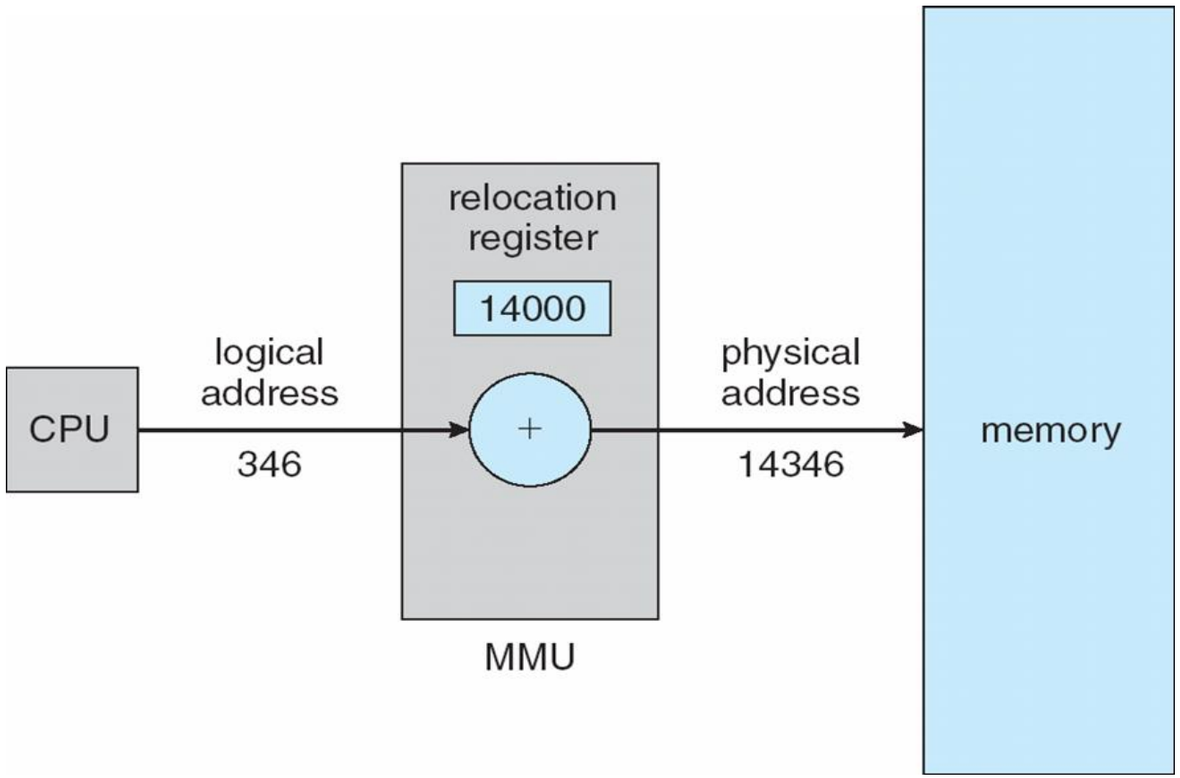
- Recall: address space
  - All the addresses and space a process can “touch”
  - Each process has its own unique address space, different from kernel address space
- Consequently, there are two view of memory
  - Logical address – generated by the CPU; also referred to as virtual address
  - Physical address – address seen by the memory
  - The translation is done by memory management unit or MMU
- Translation makes it much easier to implement protection
  - To ensure a process can not access another process’s address space.
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ only in execution-time address-binding scheme
  - Logical address space is the set of all logical addresses generated by a program
  - Physical address space is the set of all physical addresses generated by a program

Memory-Management Unit (MMU)

- MMU is the hardware device that at run time maps virtual address to physical address
- Many methods possible, covered in the rest of this chapter
- To start, consider a simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - Base register now called relocation register
  - MS-DOS on Intel 80x86 used 4 relocation registers
- The user program deals with logical addresses; it never sees the real physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses

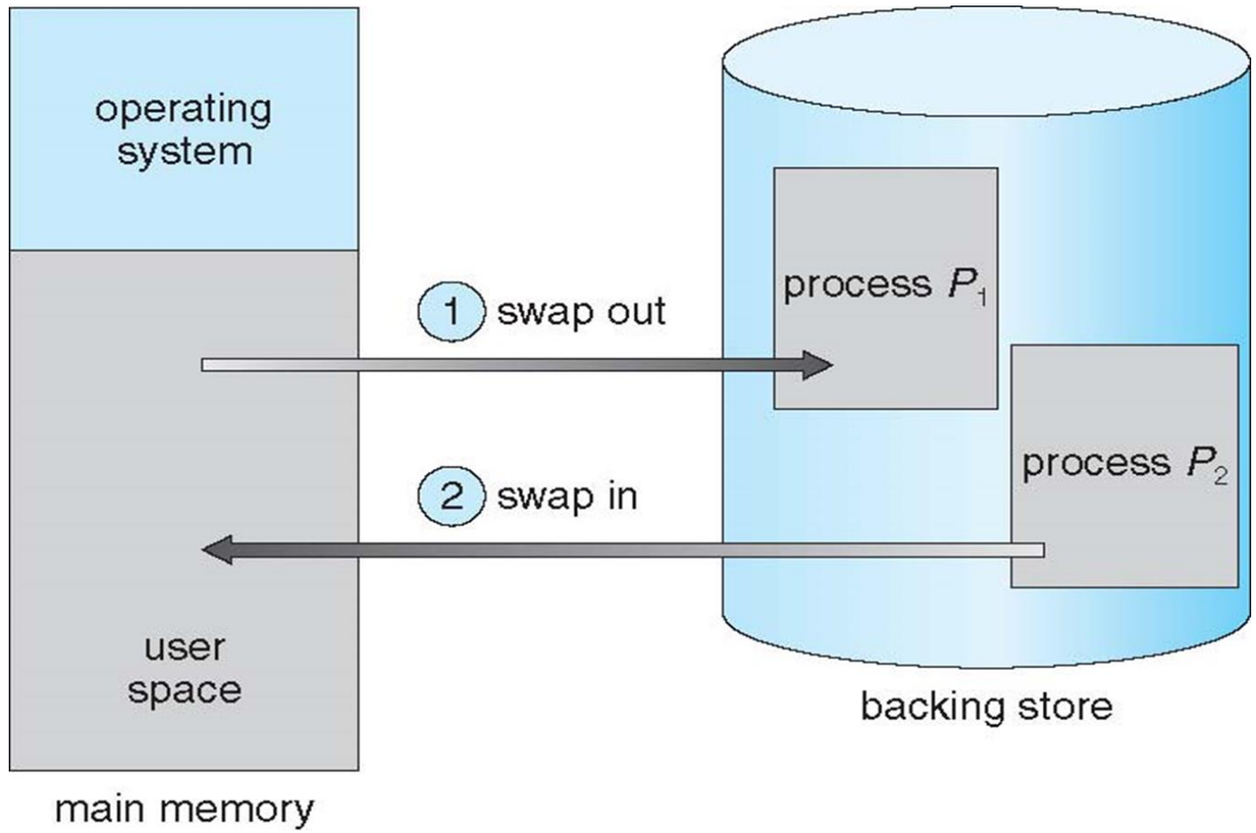
Dynamic Relocation Using a Relocation Register

- Dynamic Loading: a routine is not loaded until it is called. All routines are kept on disk in a relocated format.
- Better memory-space utilization; unused routine is never loaded
- Useful when large amounts of code are needed to handle infrequently occurring cases – handing errors for instance
- No special support from the operating system is required
  - Implemented through program design
  - OS can help by providing libraries to implement dynamic loading



Swapping

- A process can be swapped temporarily out of memory to a backing store (disk), and then brought back into memory for continued execution
- Total memory space of all processes can exceed physical memory, thus increasing the degree of multiprogramming
- Backing store – fast disk large enough to accommodate copies of all memory images of 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 can be swapped out so higher-priority process can be loaded and executed



Swapping (Cont.)

- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory being swapped.
- Suppose the a user process has 100 MB size, and backing store is a standard hard disk with a transfer rate of 50 MB/s. Thus the transfer time of the 100-MB process is 2 seconds, which is 2,000 millisecond (fairly high).
- System maintains a ready queue of ready-to-run processes with memory images on disk
- Does the swapped out process need to swap back in to same physical addresses?
- Depends on address binding method
  - Plus consider pending I/O to/from process memory space
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold

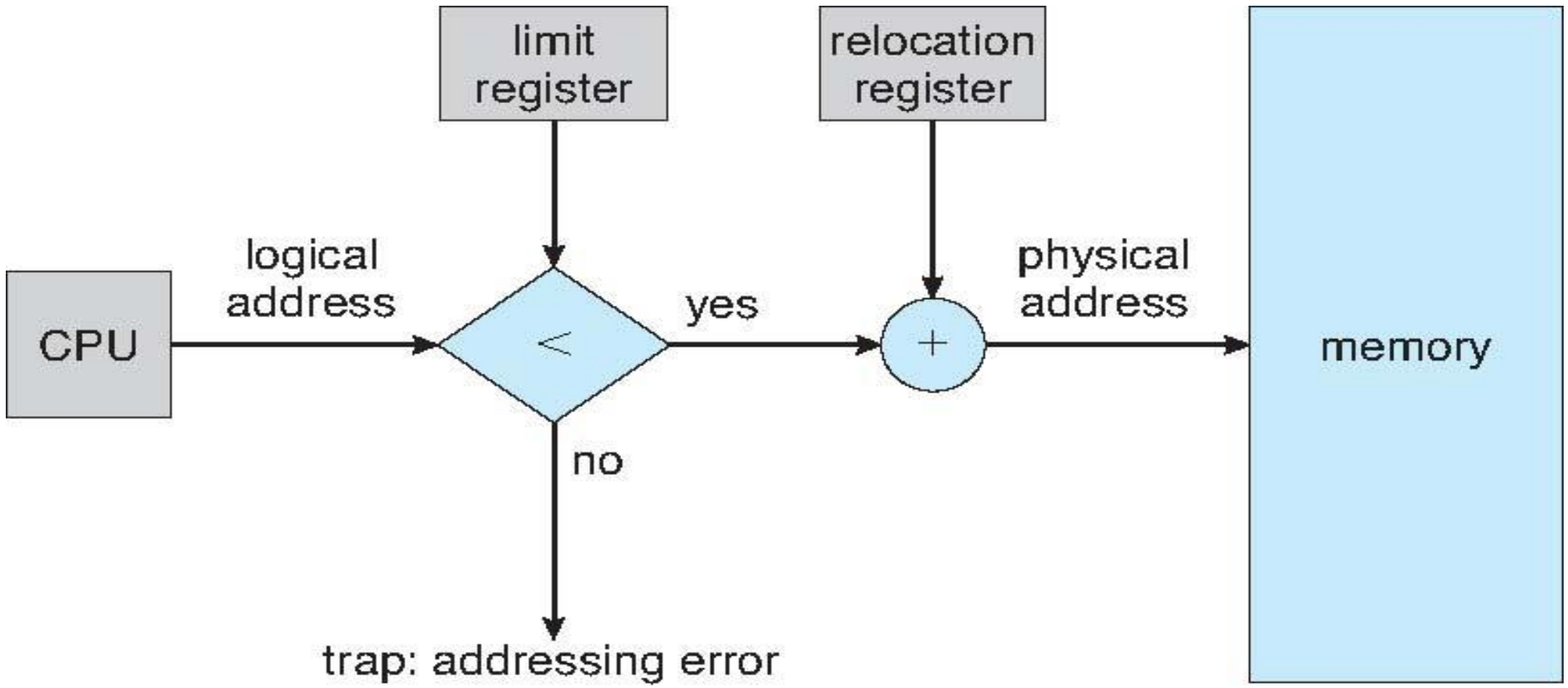
Contiguous Allocation

- Main memory must support both kernel processes and user processes
- Limited resource must be allocated efficiently
- Contiguous allocation is one of the early memory allocation methods

- Main memory is usually divided into two partitions:
  - Resident operating system, usually held in low memory with interrupt vectors
  - User processes then held in high memory
  - Each process contained in single contiguous section of memory

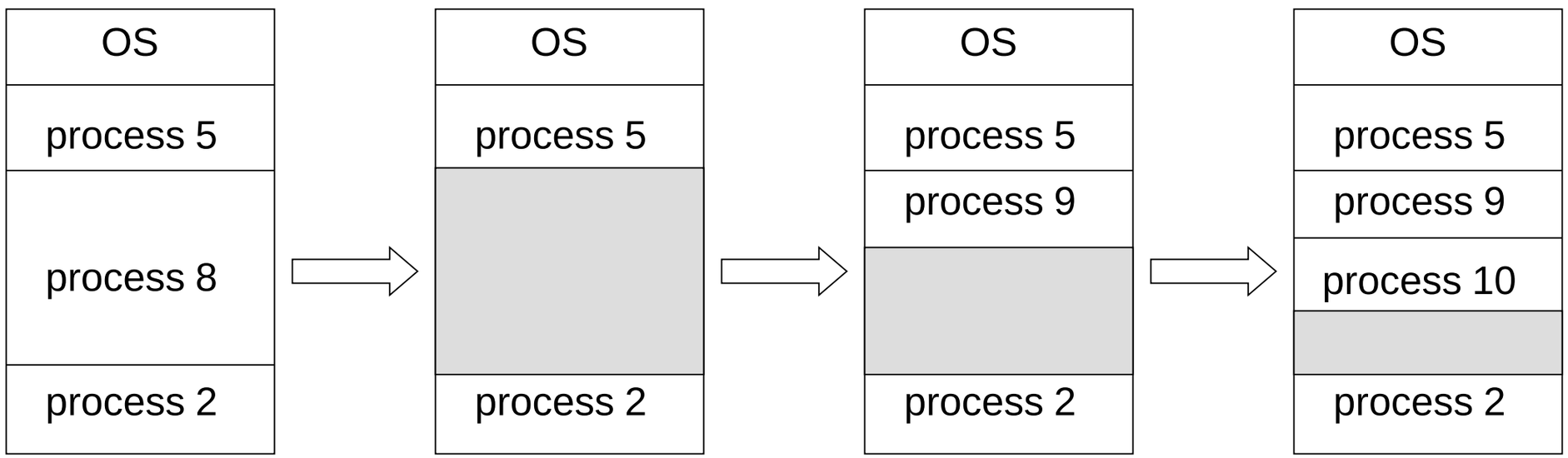
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*

Hardware Support for Relocation and Limit Registers



Contiguous Allocation (Cont.)

- Multiple-partition allocations
  - Degree of multiprogramming is bounded by number of partitions
  - Variable-partition sizes (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 – intended not use it
- Worst-fit: Allocate the *largest* hole
  - Must also search entire list
  - Produces the largest leftover hole – intended to reuse the remaining hole

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

Fragmentation

- External Fragmentation – total memory space exists to satisfy a request, but it is not contiguous – scattered holes

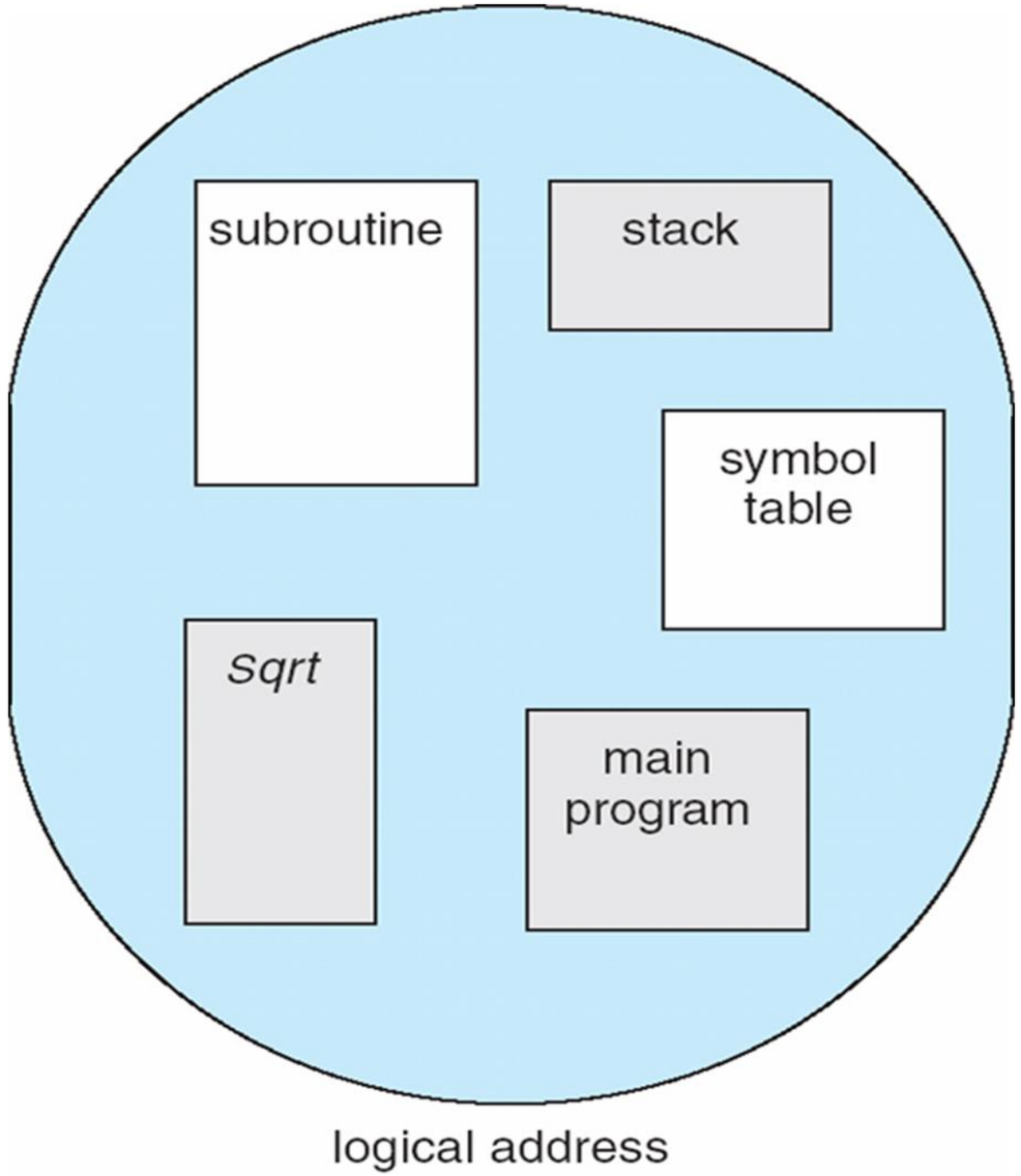
- Internal Fragmentation – memory allocated to a process may be larger than requested memory; this size difference is memory internal to a partition, but not being used

Fragmentation (Cont.)

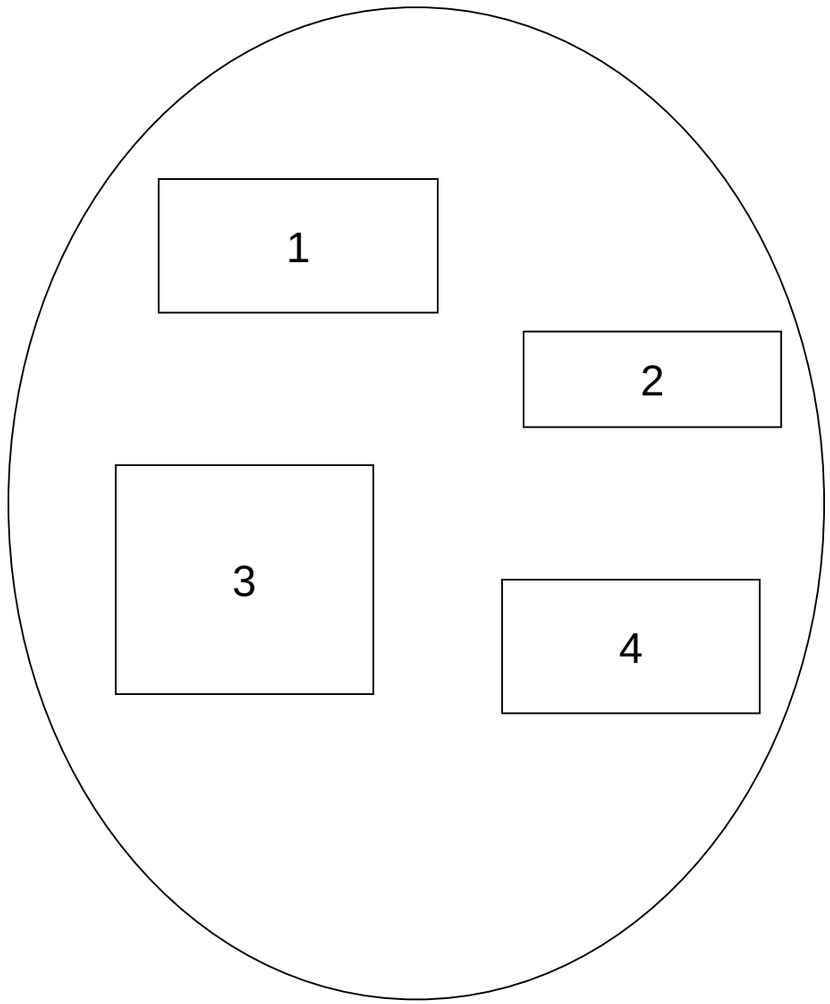
- Reduce external fragmentation by compaction
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time. In another word, if relocation is static and is done at assembly or load time, compaction cannot be done
  - The compaction is expensive (time-consuming)
- The backing store has similar fragmentation problems, will be discussed in Chapters 10-12
- Another solution is to permit the logical address space of the processes to be non-contiguous, thus allowing a process to be allocated physical memory wherever such memory is available. These techniques include
  - Segmentation
  - Paging

Segmentation- User View

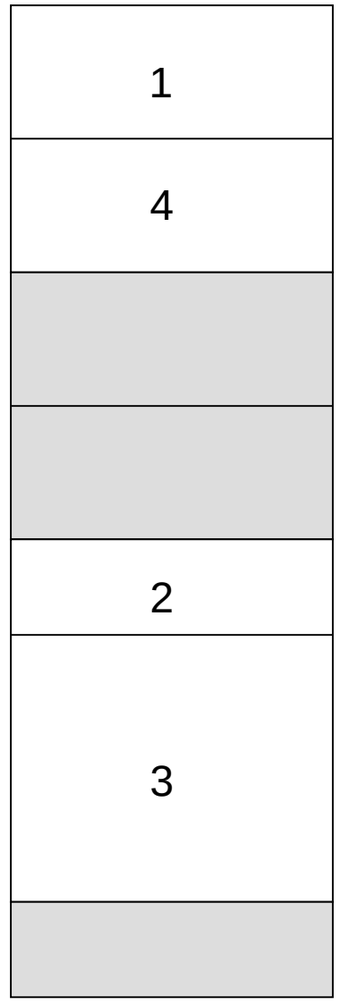
- Memory-management scheme that supports user view of memory
- A program is a collection of segments
  - A segment is a logical unit such as:
    - main program
    - procedure
    - function
    - method
    - object
    - local variables, global variables
    - common block
    - stack
    - symbol table
    - arrays



Segmentation: Logical vs. Physical



user space



physical memory space

Segmentation Architecture

- Logical address consists of a *two tuple*:

  <segment-number, offset>,

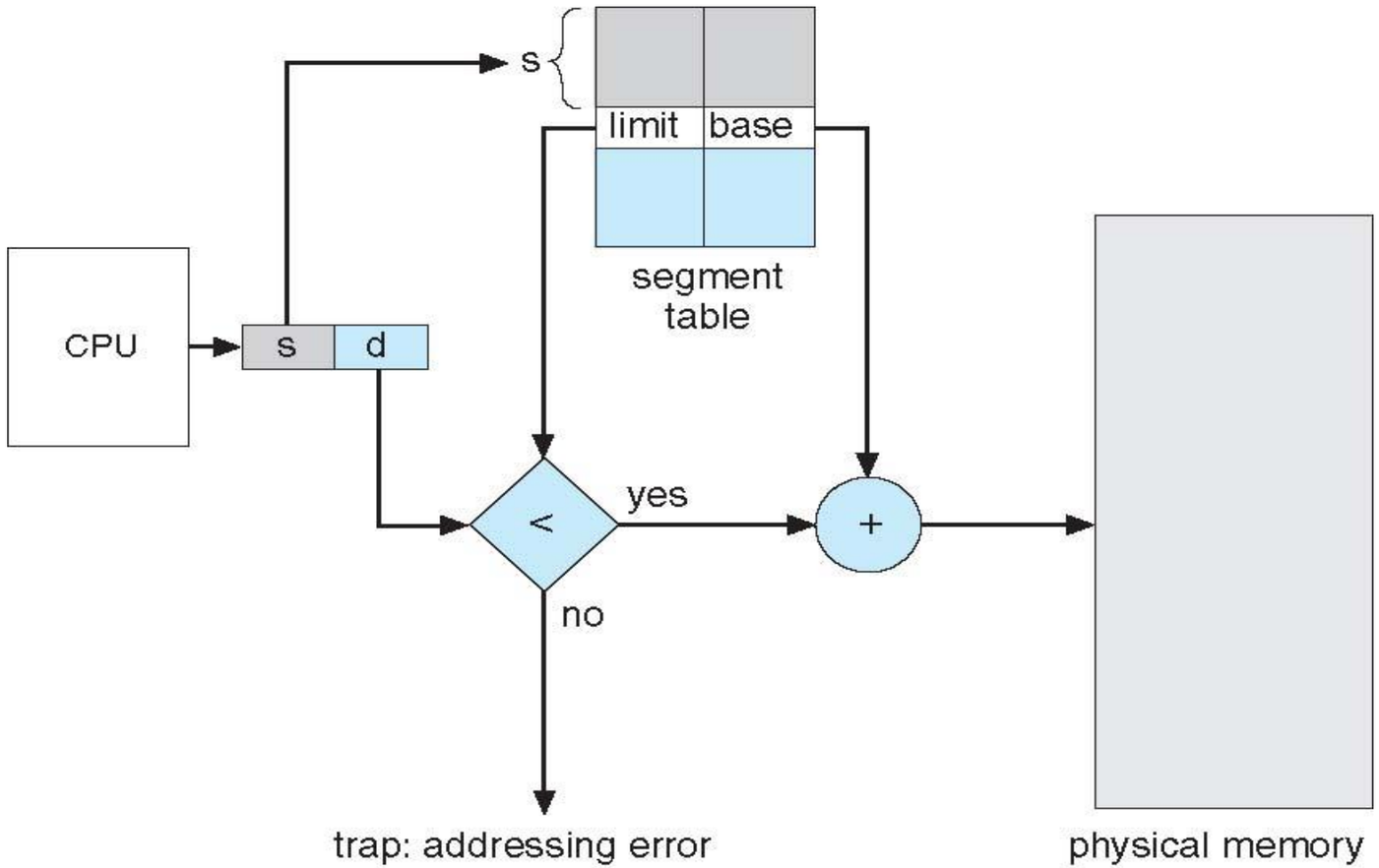
- Segment table – maps two-dimensional programmer-defined addresses (virtual address) into one-dimensional physical addresses; each table entry has:
  - base – contains the starting physical address where the segments reside in memory
  - limit – specifies the length of the segment

- Both STBR and STLR stored in the PCB of a process
- Segment-table base register (STBR) points to the segment table's location in memory
- Segment-table length register (STLR) indicates the number of segments used by a program;

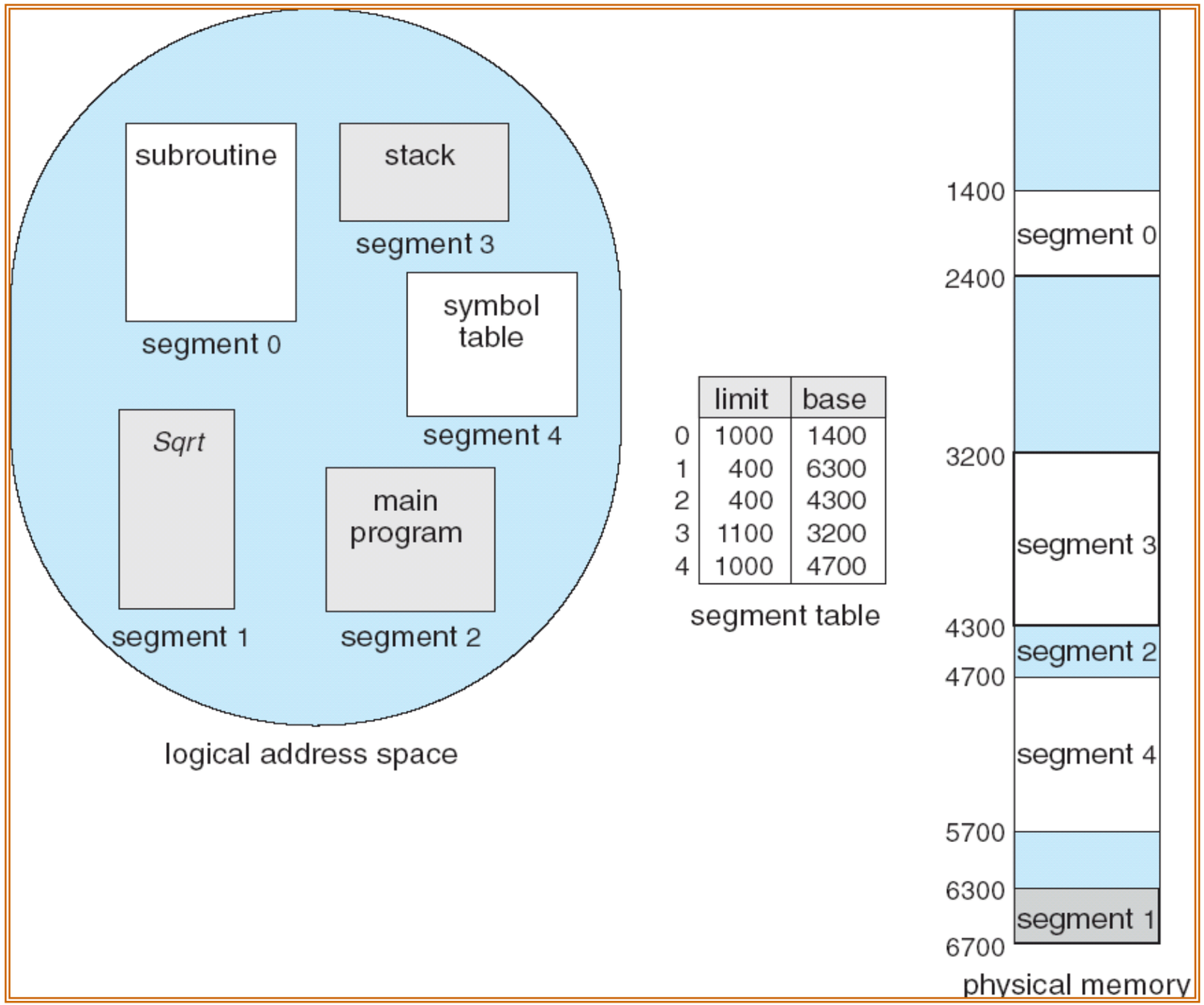
  segment number s is legal if s < STLR

Segmentation Architecture (Cont.)

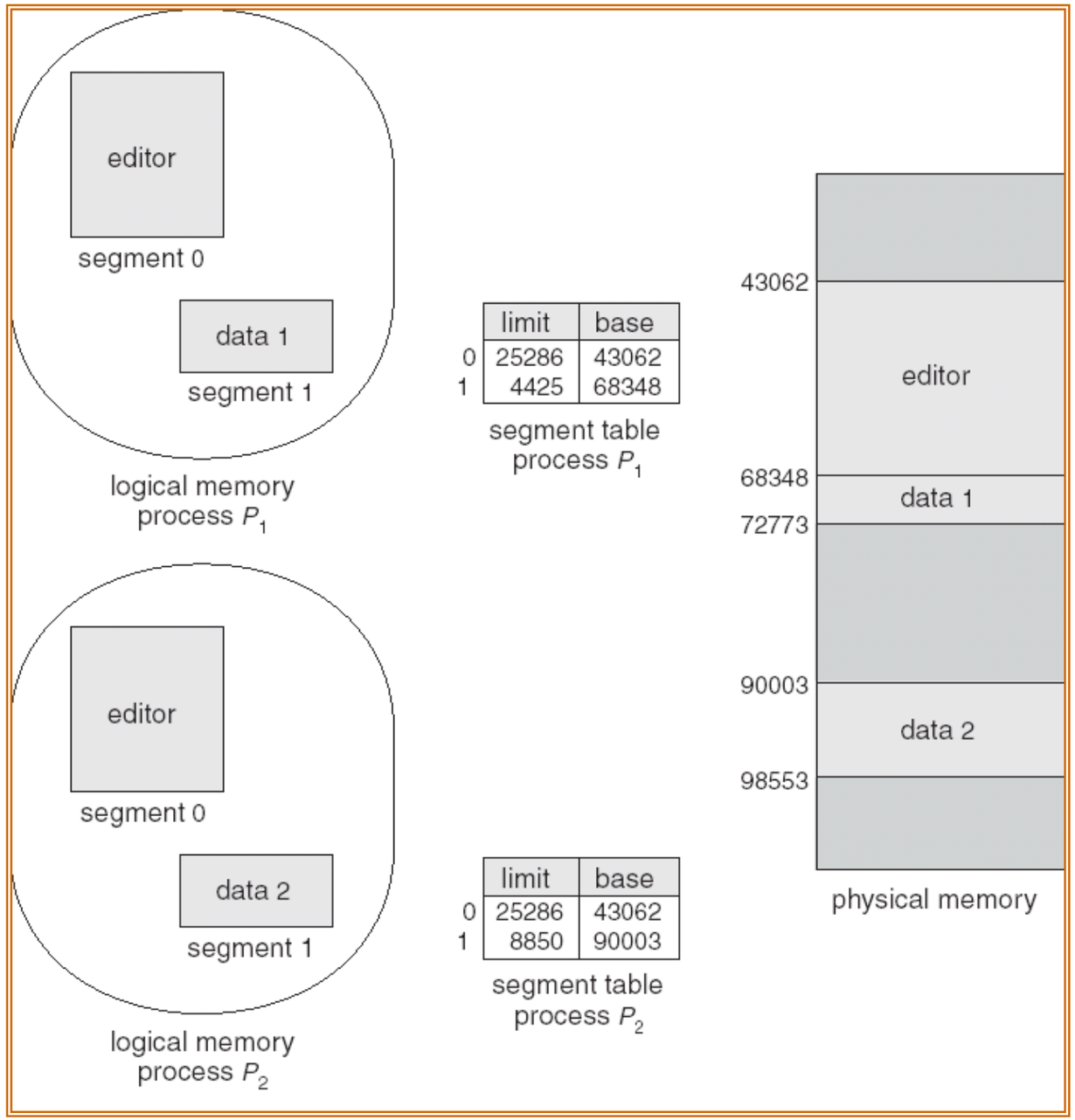
- Protection. each entry in segment table associates with:
  - validation bit = 0 ⇒ illegal segment
  - read/write/execute privileges
- A protection bit is associated with each segment; code sharing occurs at segment level
- Since segments vary in length, memory allocation for a segment is a dynamic storage-allocation problem
- External fragmentation still exists, but musch less severe than that in a contiguous allocation



Example of Segmentation



Sharing of Segments



Summary - Segmentation

- Protection is easy in segmentation scheme – at segment table
  - Code segment would be read-only, data and stack would be read-write (stores allowed), shared segment could be read-only or read-write
- Can address outside valid range?
  - This is how the stack and heap are allowed to grow. For instance, stack takes fault, system automatically increases size of stack
- The main problem with segmentation
  - Must fit variable-sized chunks into physical memory
  - May move processes multiple times to fit everything
  - Limited options for swapping to disk (multiple segments)
  - External fragmentation

Paging

- Physical address space of a process can be non-contiguous; process is allocated physical memory whenever the physical memory space is available
  - Avoids external fragmentation
  - Avoids the problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called frames
  - Size is power of 2, usually between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called pages - the same size of a frame
- Needs to keep track of all free frames available in main memory
- To run a program of size N pages, need to find N free frames and load the program
- A page table is used to translate logical to physical addresses – keep track of allocated frames
- Backing store likewise split into pages or frames
- This, however, suffers from internal fragmentation – last page/frame

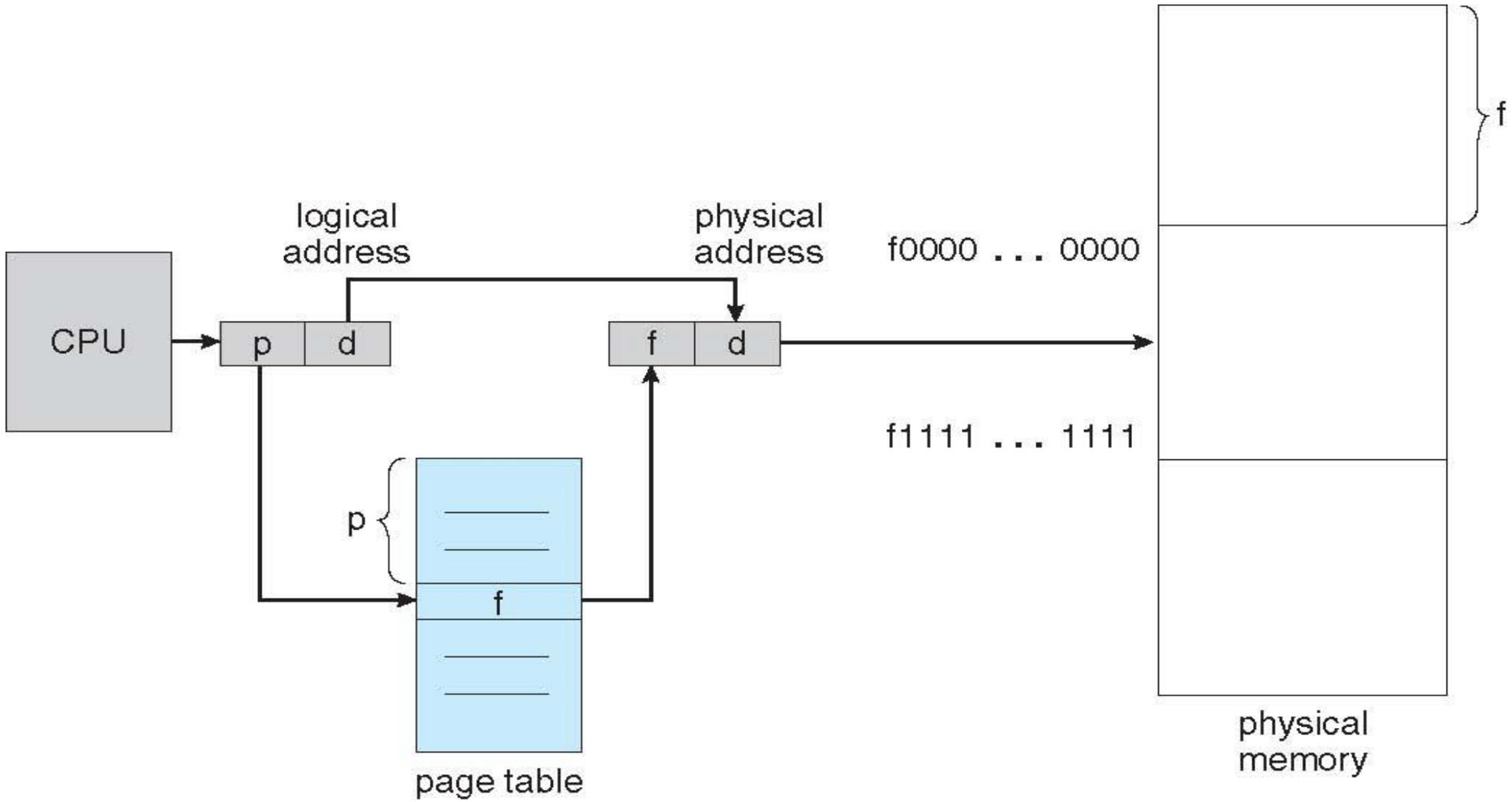
Address Translation Scheme

- 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

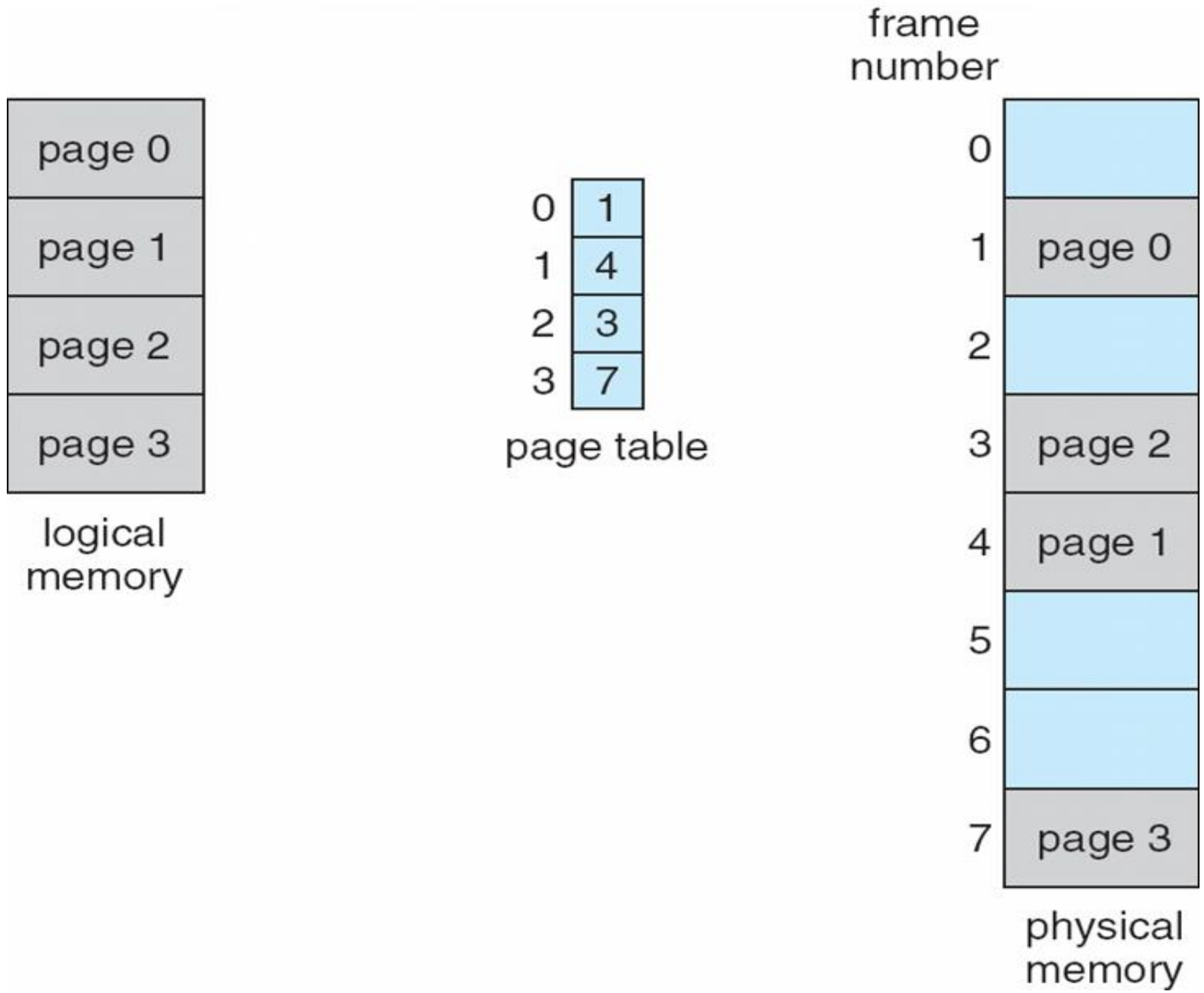
page number	page offset
p	*d*
m - n	*n*

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

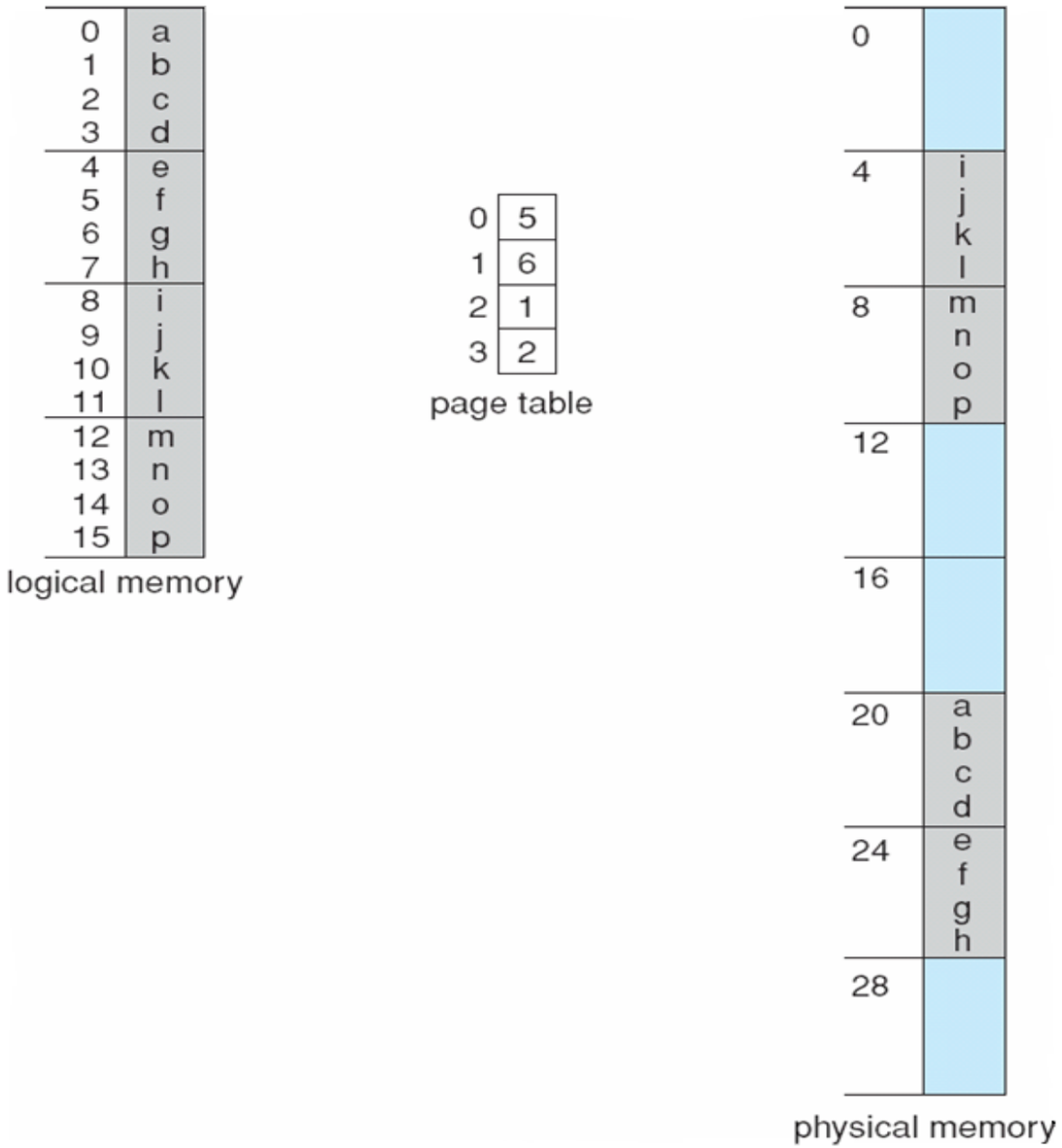
Paging Hardware



Paging Model of Logical and Physical Memory



Paging Example

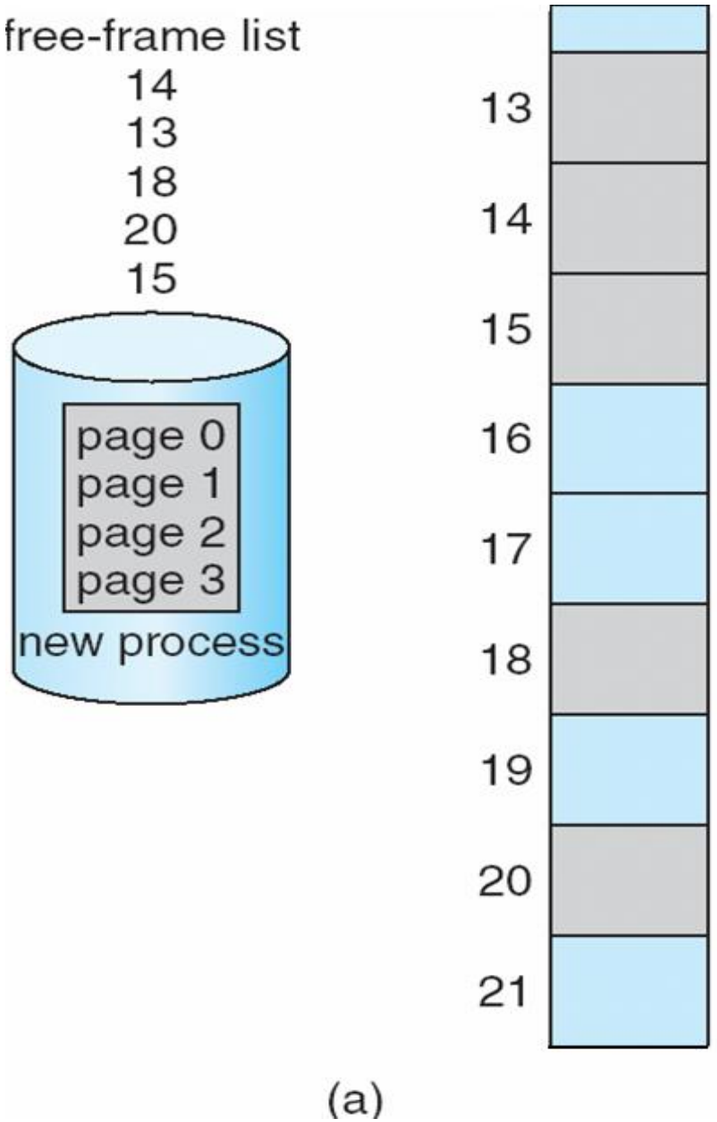


n=2 and *m*=4 32-byte memory and 4-byte pages

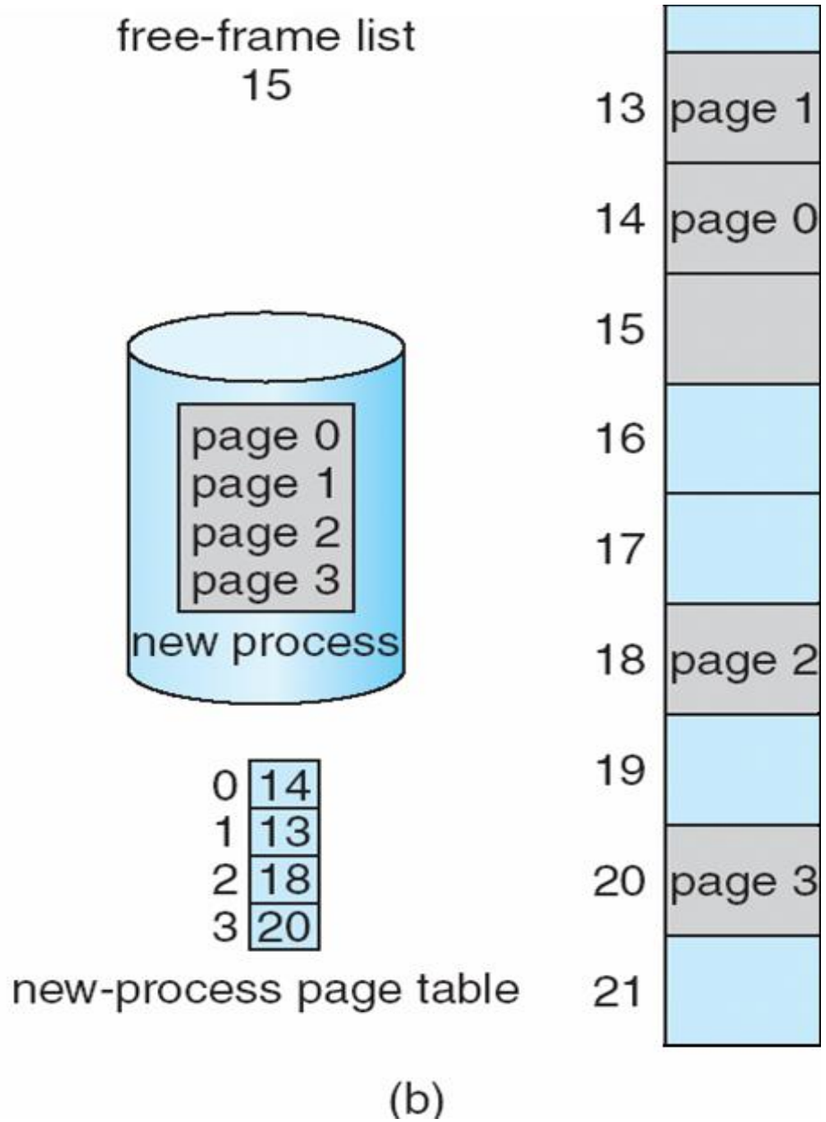
Paging (Cont.)

- Calculating internal fragmentation
  - Page size = 2,048 bytes (2KB)
  - 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, smaller page size leads to larger page table
  - 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 (starting address)
- 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 (translation) and one for fetching the data / instruction

- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called associative memory or translation look-aside buffers (TLBs)

- TLBs typically small (64 to 1,024 entries). Some CPUs implement separate instruction and data address TLBs

- On a TLB miss (if the page number is not in the TLB), value is loaded into the TLB for faster access next time. This can be done by hardware (MMU) or software (running kernel codes)
  - Replacement policies must be considered
  - Some entries can be wired down for permanent fast access, for example TLB entries for key kernel code for fast access

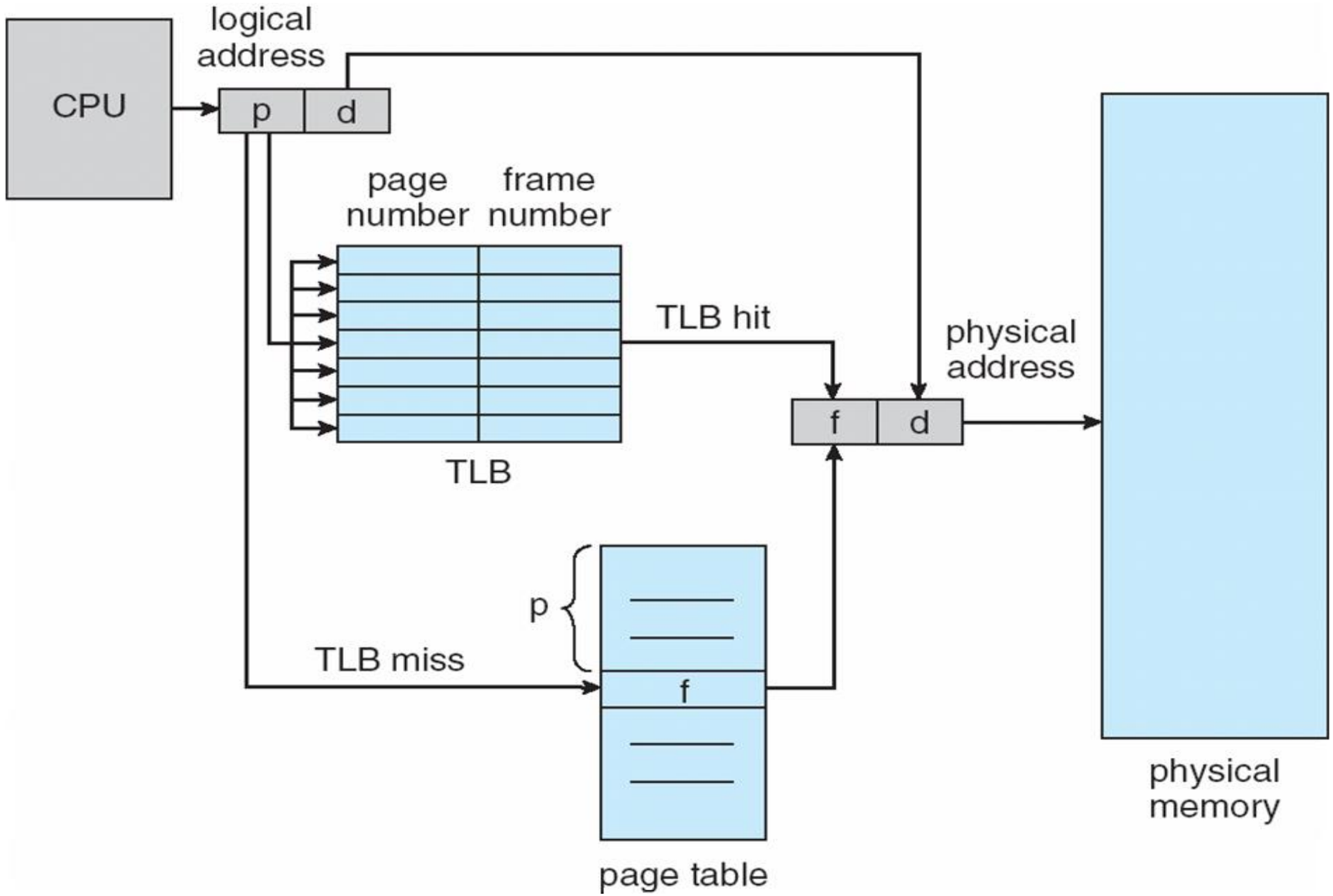
Associative Memory

- Associative memory (TLB) – parallel search

Page #	Frame #

- Address translation (p, d)
  - Check all entries in parallel (hardware)
  - If p is in associative register, get frame # out
  - Otherwise get frame # from page table in memory, and also bring this entry to the TLB
- Locality on TLB
  - Instruction usually stays on the same page (sequential access nature)
  - Stack definitely has locality (pop in and out)
  - Data less locality, still quite a bit

Paging Hardware With TLB



Effective Access Time

- Associative Lookup = ε time unit
  - Usually < 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)

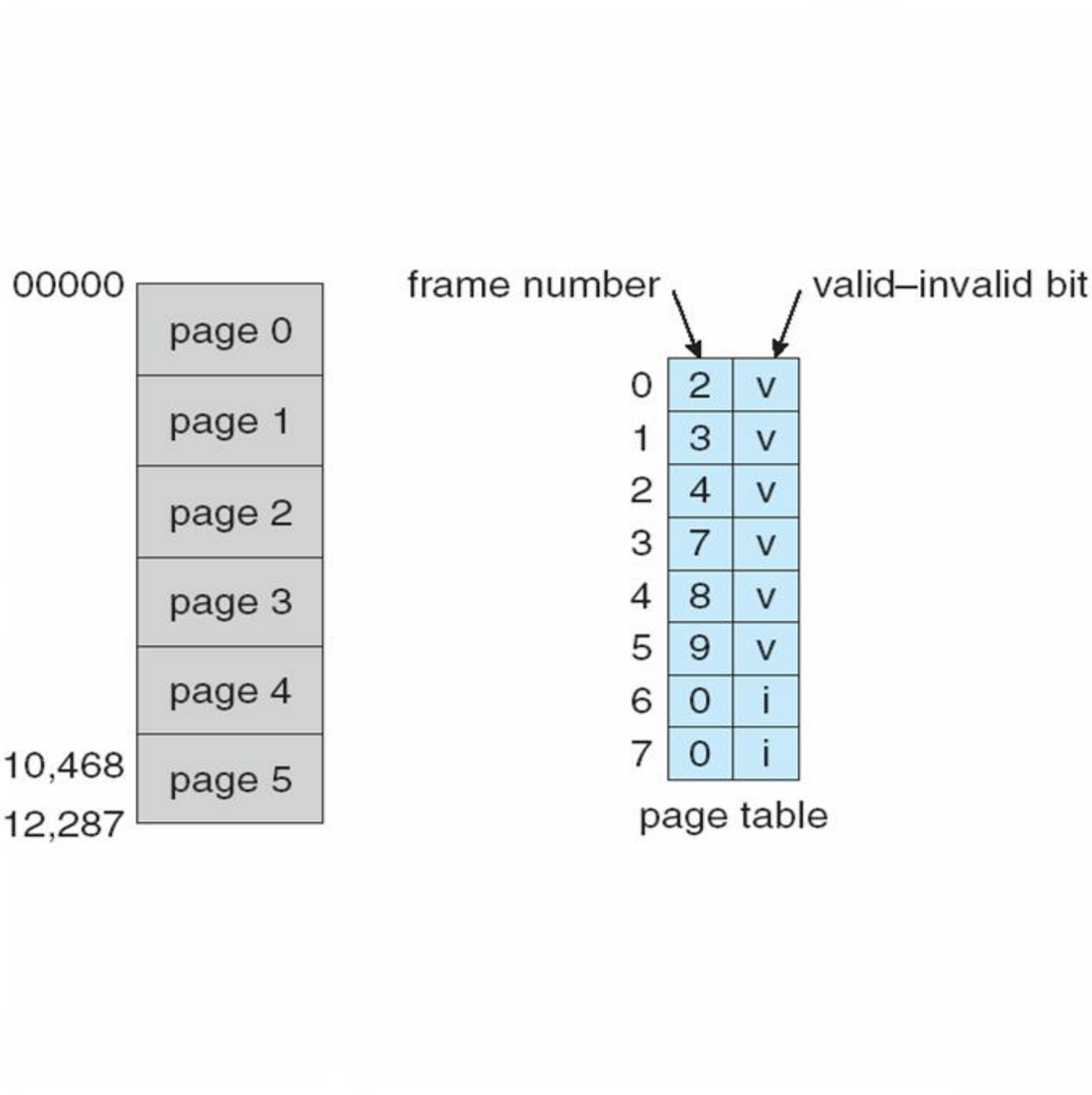
$$\begin{aligned} EAT &= (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha) \\ &= 2 + \varepsilon - \alpha \end{aligned}$$

- Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.80 x 120 + 0.20 x 220 = 140ns
- Consider more realistic hit ratio -> α = 99%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.99 x 120 + 0.01 x 220 = 121ns

Memory Protection

- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
  - Can also add more bits to indicate page execute-only, and so on
- Valid-invalid bit attached to each entry in the page table:
  - “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page
  - “invalid” indicates that the page is not in the process’ logical address space
  - The operating system sets this bit for each page to allow or disallow access to the page
- Any violations result in a trap to the kernel
  - Example: a 14-bit address space (0 to 16383). If a program only uses address 0 to 10468, with a page size 2KB, pages 0-5 are valid, pages 6-7 are invalid.
- Rarely does a process use all it address range, usually only a small fraction of the address space available. It would be wasteful to create a page table with entries for every page in the address range, and most of the table entries are unused but would take up memory space
  - use page-table length register (PTLR) to indicate the size of the page

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



An Example: Intelx86 Page Table Entry

- What is in a Page Table Entry or PTE?
  - For page translation, each page table consists of a number of PTEs
  - Permission bits: valid, read-only, read-write, write-only
- Example: Intel x86 architecture PTE:
  - Address same format previous slide (10, 10, 12-bit offset)
  - Intermediate page tables called “Directories”

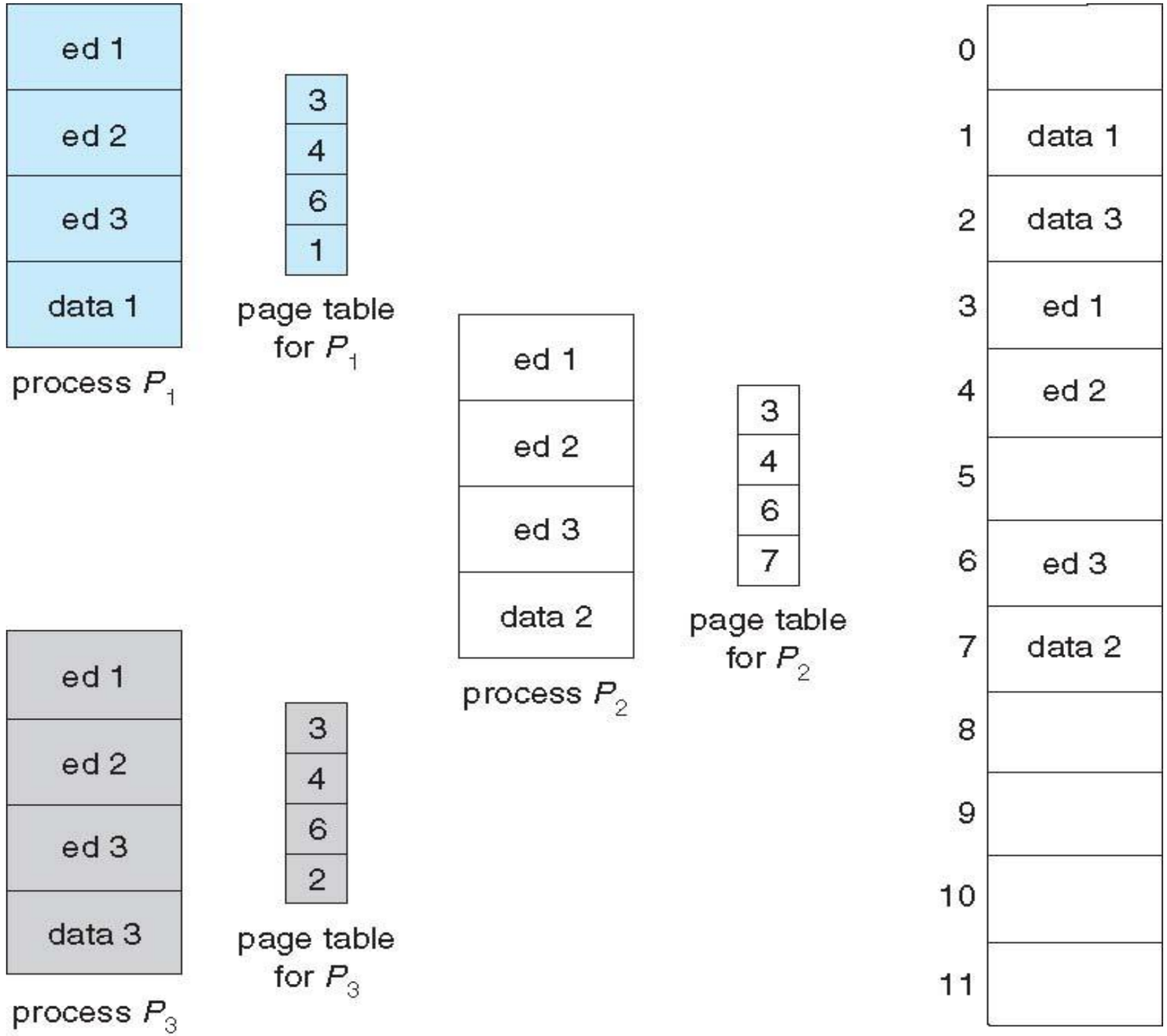
Page Frame Number (Physical Page Number)	Free (OS)	0	L	D	A	PCD	PWT	U	W	P
31-12	11-9	8	7	6	5	4	3	2	1	0

P: Present (same as “valid” bit in other architectures)
W: Writeable
U: User accessible
PWT: Page write transparent: external cache write-through
PCD: Page cache disabled (page cannot be cached)
A: Accessed: page has been accessed recently
D: Dirty (PTE only): page has been modified recently
L: L=1⇒4MB page (directory only).
Bottom 22 bits of virtual address serve as offset within a page

Shared Pages

- Shared code
  - One copy of read-only (re-entrant) code shared among processes (i.e., text editors, compilers, window systems)
  - Re-entrant code is non-self-modifying code; it never changes during execution
  - 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



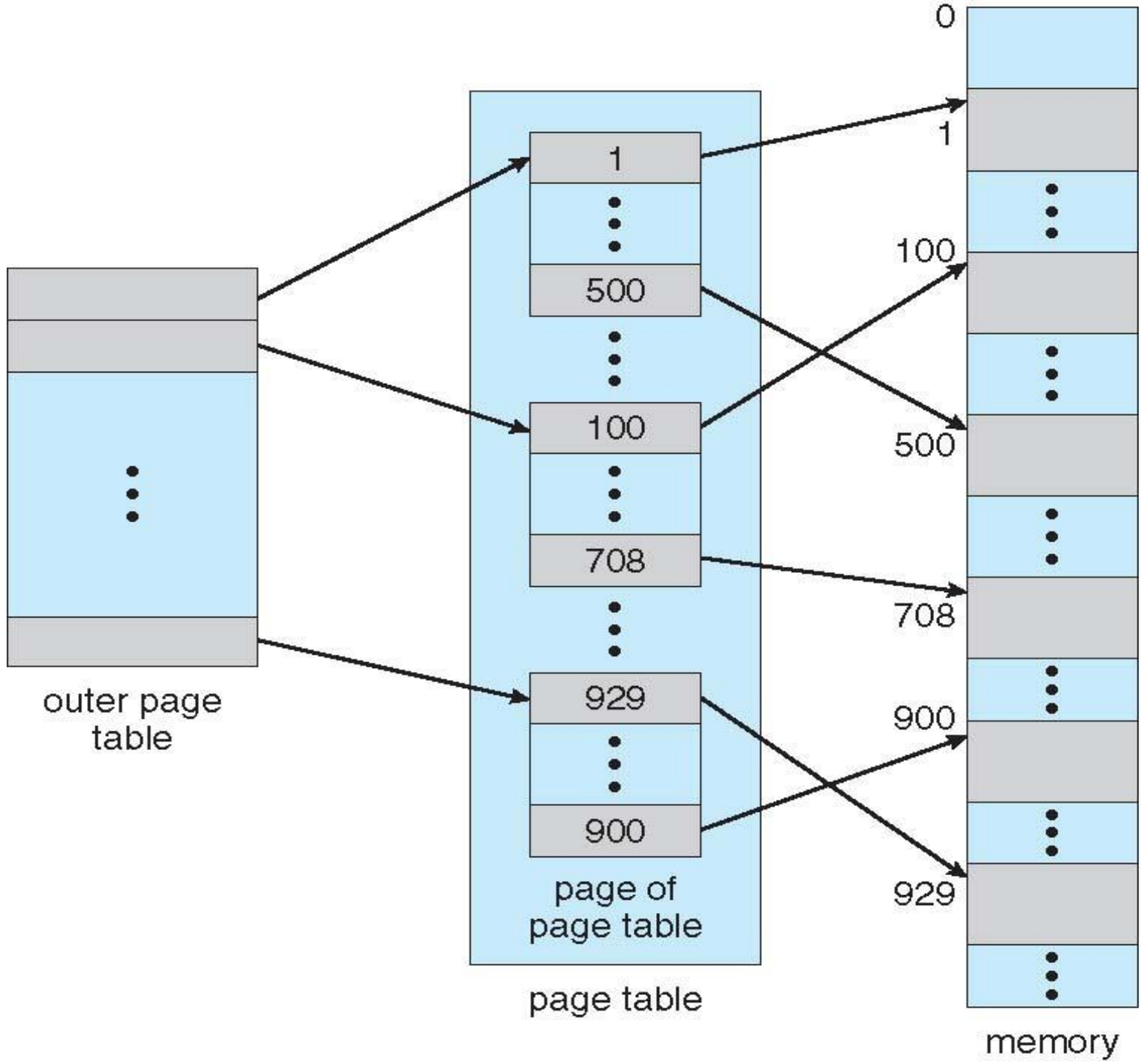
Structure of the Page Table

- Memory structures for paging can grow 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
    - Page table cannot be allocated contiguously in main memory either, which will be allocated into multiple pages (frames)
  - Page size 4 MB (2^{22}) results in a page table with 1,000 entries, less of a problem
  - What about 64-bit logical address?
- Hierarchical Paging

Hierarchical Page Tables

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

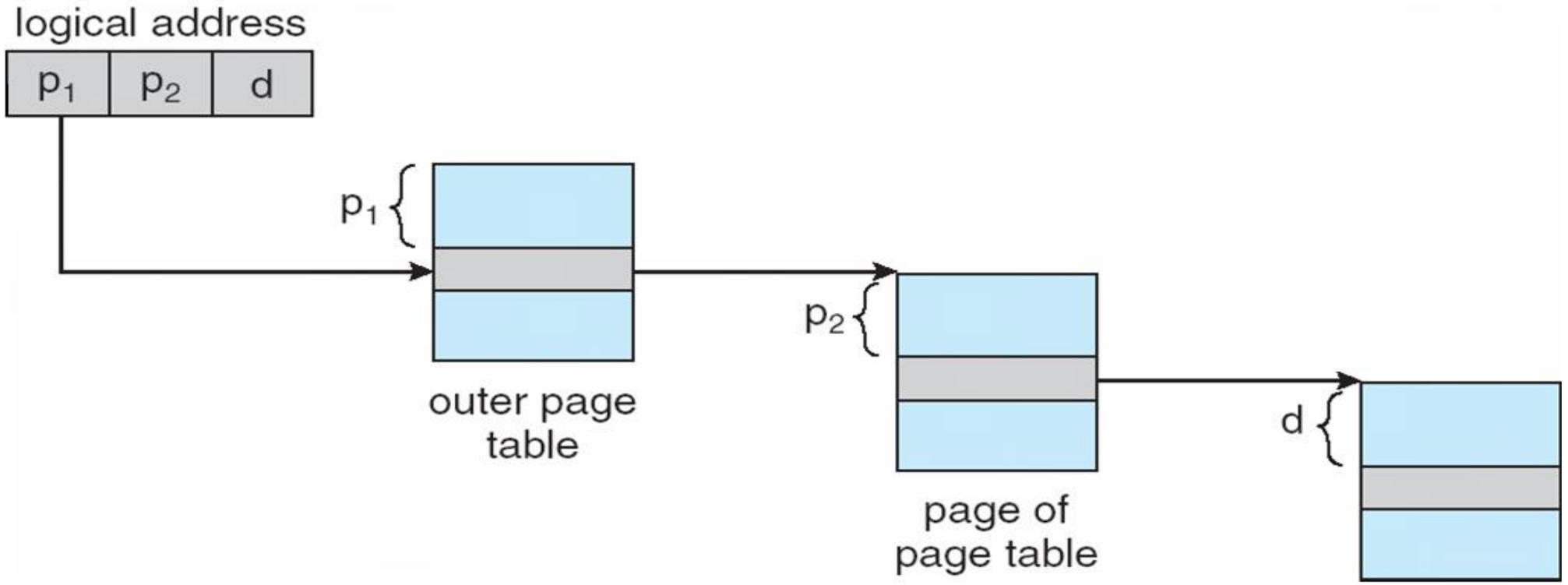
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number
  - a 10-bit page offset

- Thus, a logical address is as follows:

page number		page offset
p_1	p_2	d
10	10	12

  - where p_1 is an index into the outer page table (in Intel architecture, this is called “directories”, and p_2 is the displacement within the page of the inner page table
  - because the address translation works from the outer page table inward, this scheme is also known as a forward-mapped page table

Address-Translation Scheme



64-bit Logical Address Space

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

outer page	inner page	page offset
p_1	p_2	d
42	10	12

  - Outer page table has 2^{42} entries or 2^{44} bytes
  - One solution is to add a 2^{nd} outer page table
  - But in the following example the 2^{nd} outer page table is still 2^{34} bytes (16 GB) in size
    - And possibly 4 memory access to get to one physical memory location
    - The 64-bit UltraSPARC would require seven levels of paging – a prohibitive number of memory accesses – to translate each logical address

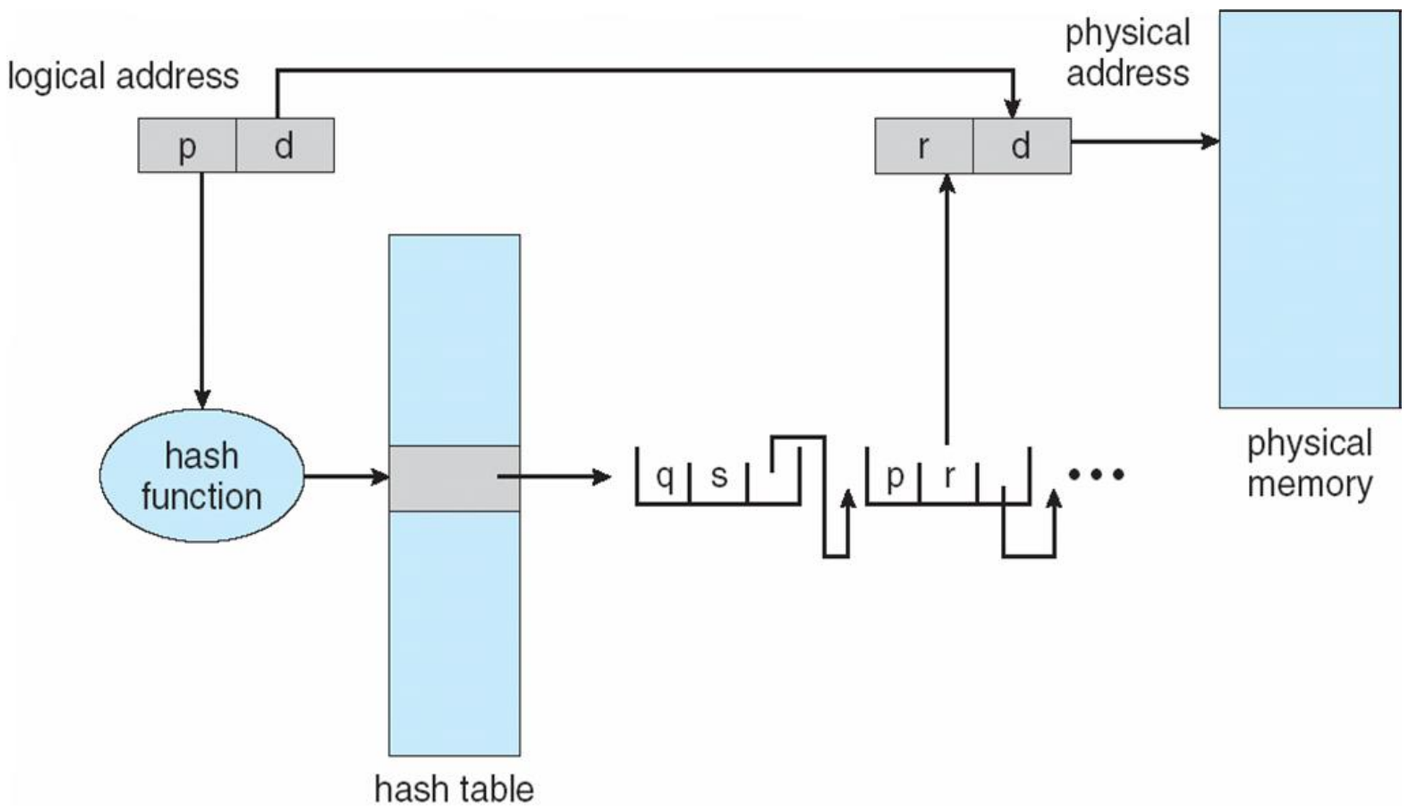
Three-level Paging Scheme

outer page	inner page	offset
p_1	p_2	d
42	10	12

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted



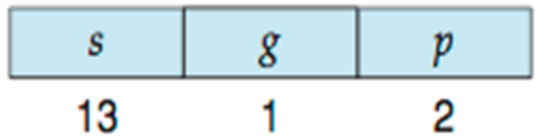
Example: The Intel 32 and 64-bit Architectures

- Dominant industry chips
  - 16-bit Intel 8086 (late 1970s) and 8088 was used in original IBM PC
- Pentium CPUs are 32-bit and called IA-32 architecture
  - It supports both segmentation and paging
- Current Intel CPUs are 64-bit and called IA-64 architecture
  - Currently most popular PC operating systems run on Intel chips, including Windows, MacOS, and Linux (of course Linux runs on several architectures as well)
  - Intel's dominance has not spread to mobile systems, where they mainly use ARM architecture
- Many variations in the chips, cover the main ideas here

Example: The Intel IA-32 Architecture

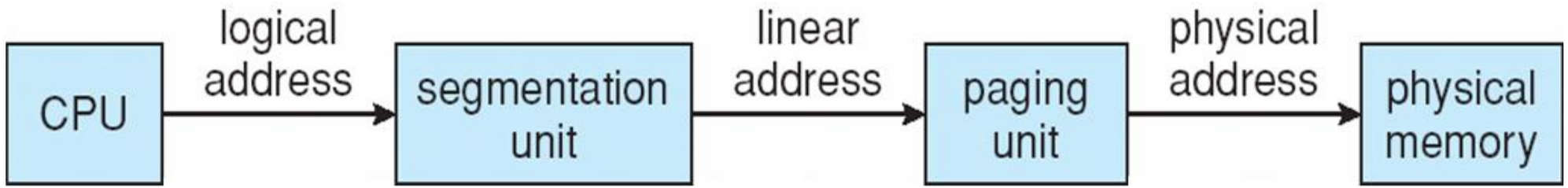
- Supports both segmentation and segmentation with paging
  - Each segment can be 4 GB, up to 16 K segments per process, divided into two partitions
    - First partition of up to 8 K segments are private to process (kept in local descriptor table (LDT))
    - Second partition of up to 8K segments shared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

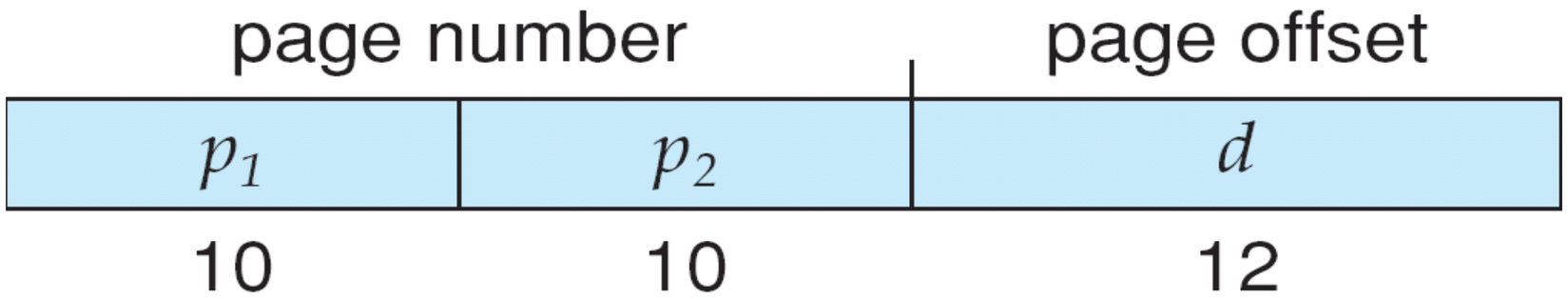
- CPU generates logical address
  - Selector given to segmentation unit
    - Which produces linear addresses



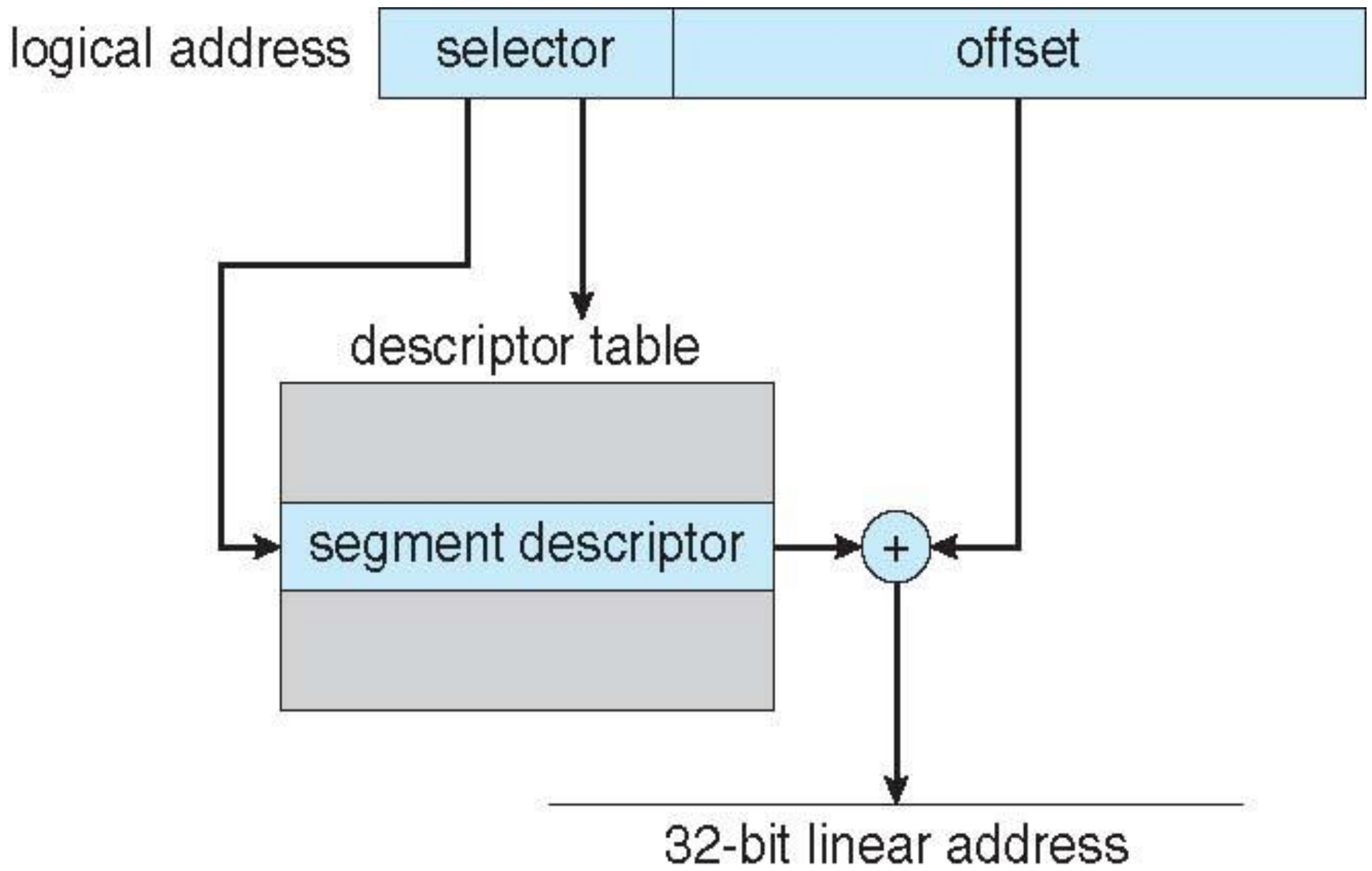
  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU
    - Pages sizes can be 4 KB or 4 MB

Logical to Physical Address Translation in IA-32

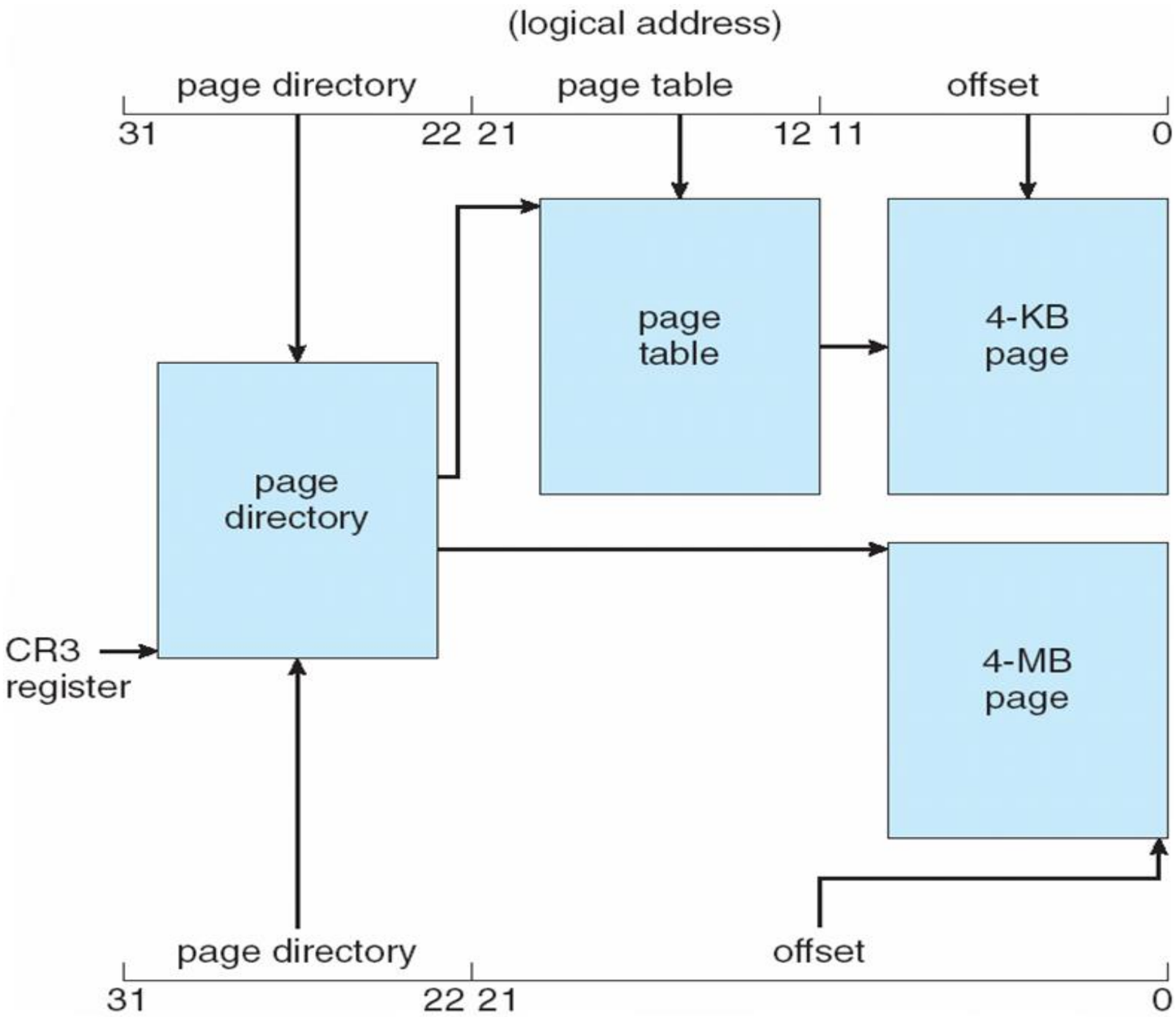




Intel IA-32 Segmentation

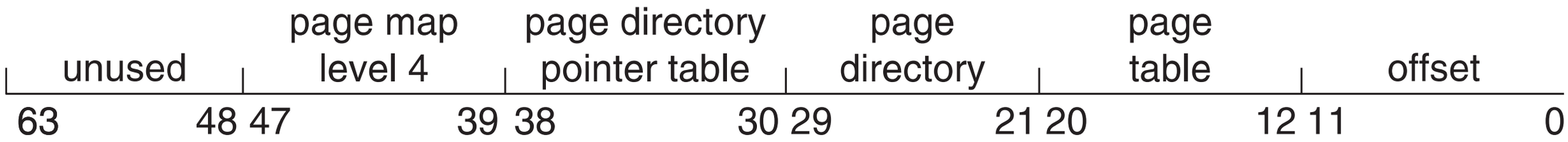


Intel IA-32 Paging Architecture



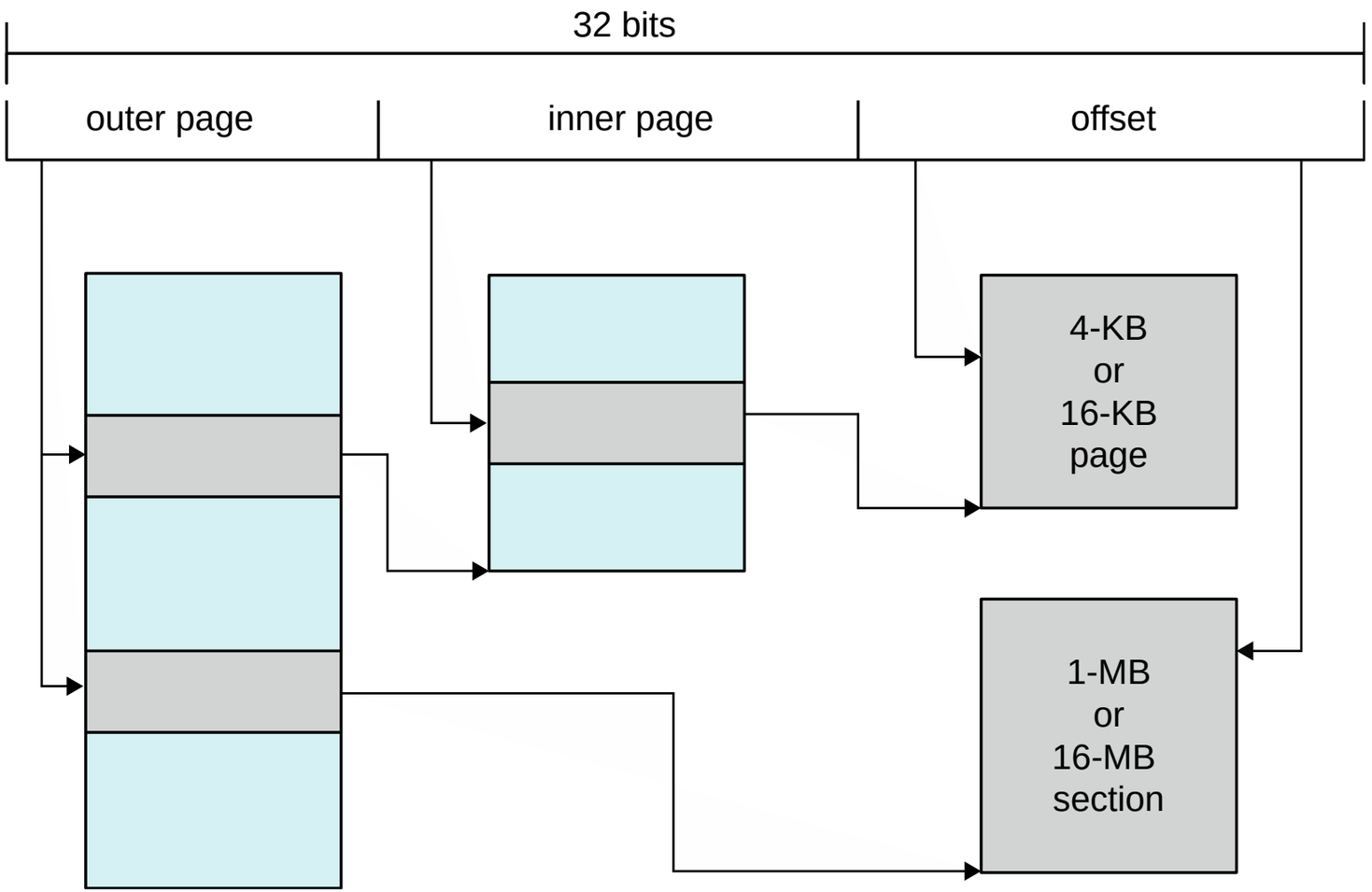
Intel x86-64

- Current generation Intel x86-64 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 (page address extension) so virtual addresses are 48 bits and physical addresses are 52 bits (4096 terabytes)



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



End of Chapter 8