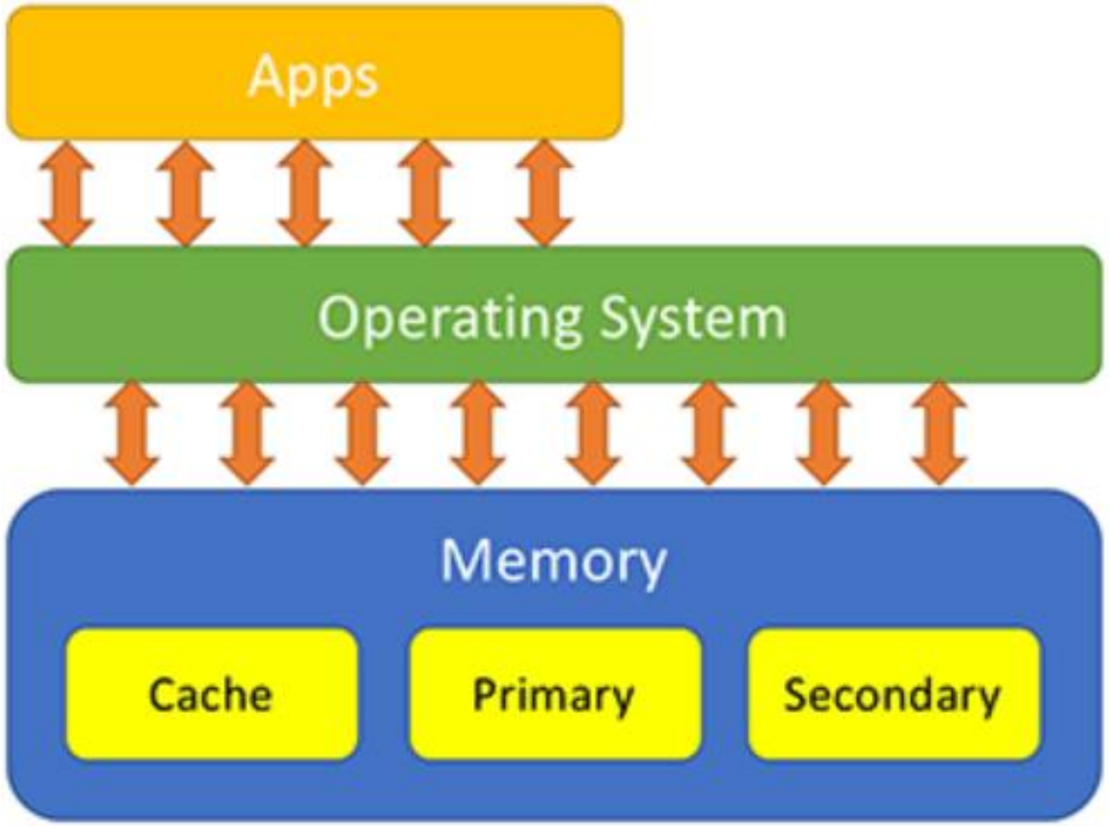
240-229: SDA (Operating Systems session)

Lecture 5: Memory Management

Associate Professor Dr. Sangsuree Vasupongayya
sangsuree.v@psu.ac.th
Department of Computer Engineering
Prince of Songkla University

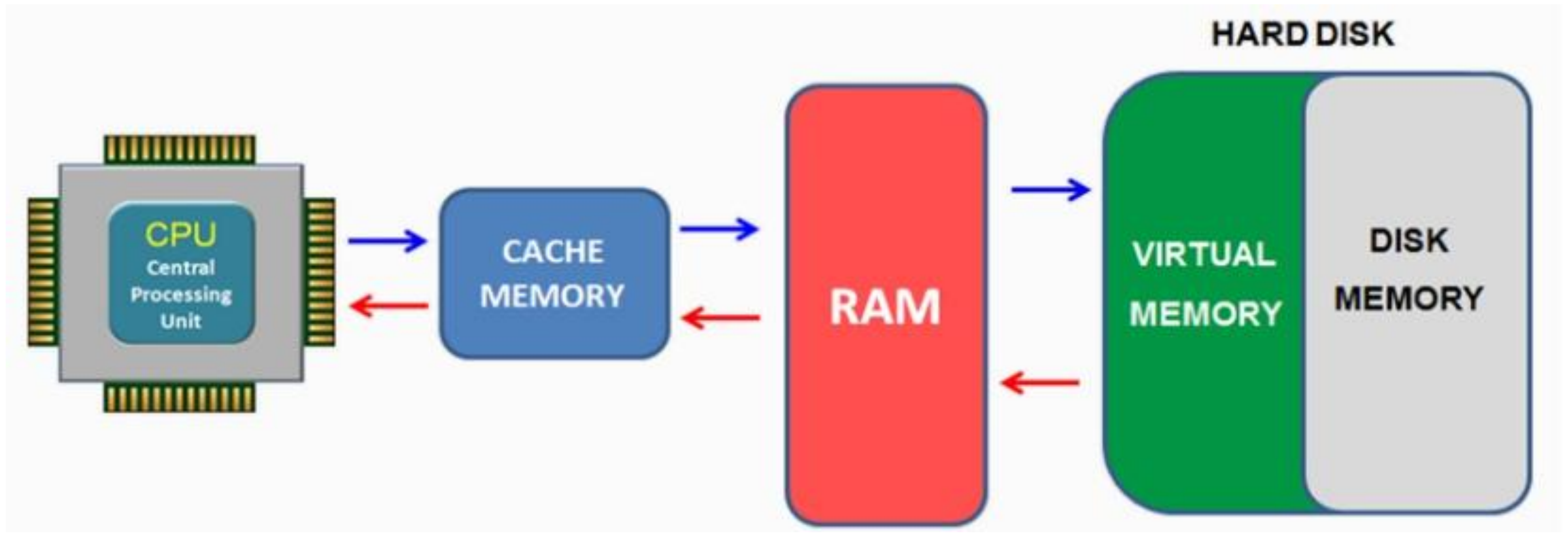
Outline

- Background
- Memory Management Requirements
- Paging
- Segmentation
- Virtual memory
- Replacement policy
- Kernel memory allocation



Background

- Program must be brought (from disk) into memory for it to be executed
  - Main memory and registers are the only storage that the CPU can access directly
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure the correctness of operations



Managing main memory

Objective

- Allow as many processes to be executed

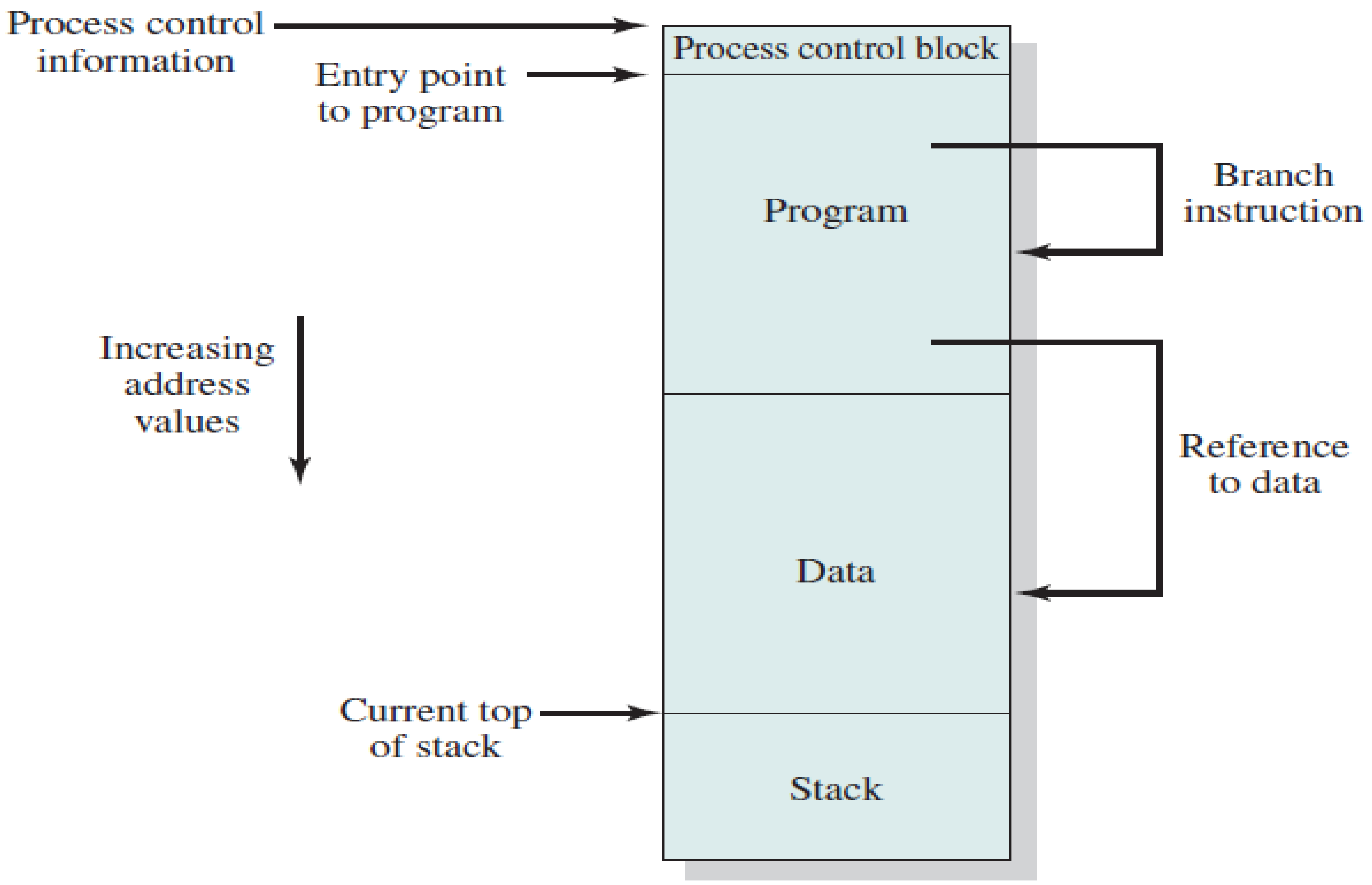
OS provides API functions to

- Allow program to allocate, access and return memory

When memory is allocated to processes, OS must keep track of

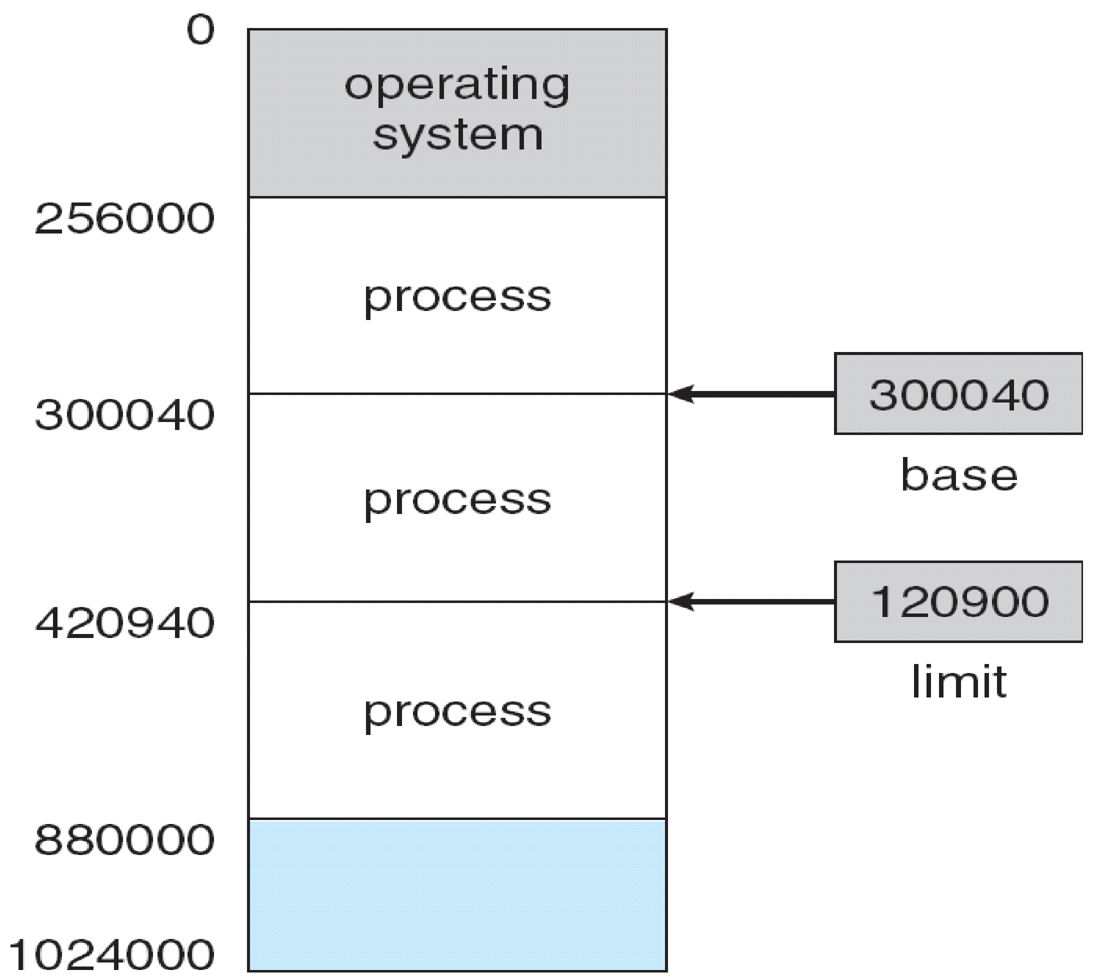
- which parts of the memory are free
- Which parts of the memory are allocated
- Allocated to which processes

Process in Memory

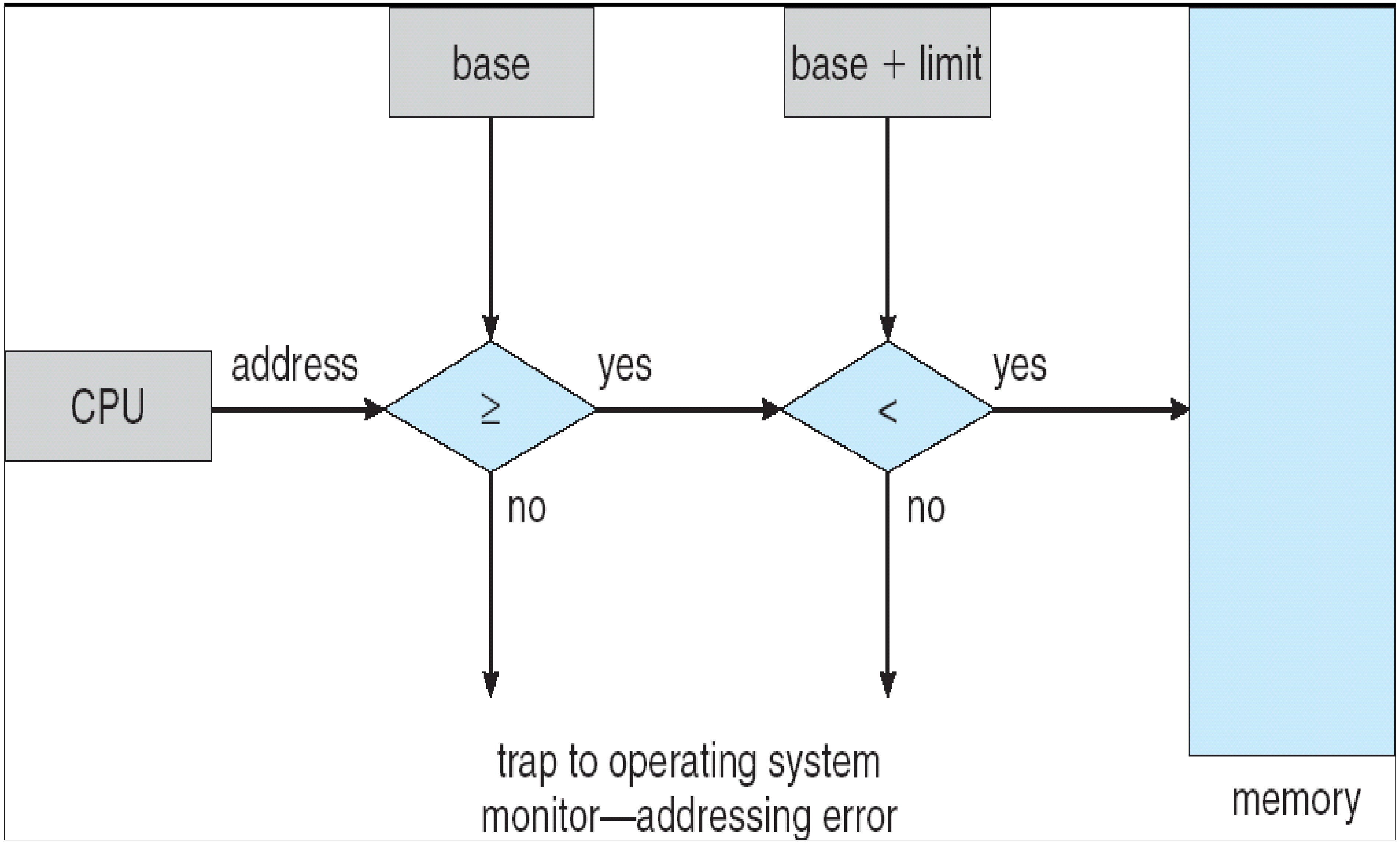


Base and Limit Registers

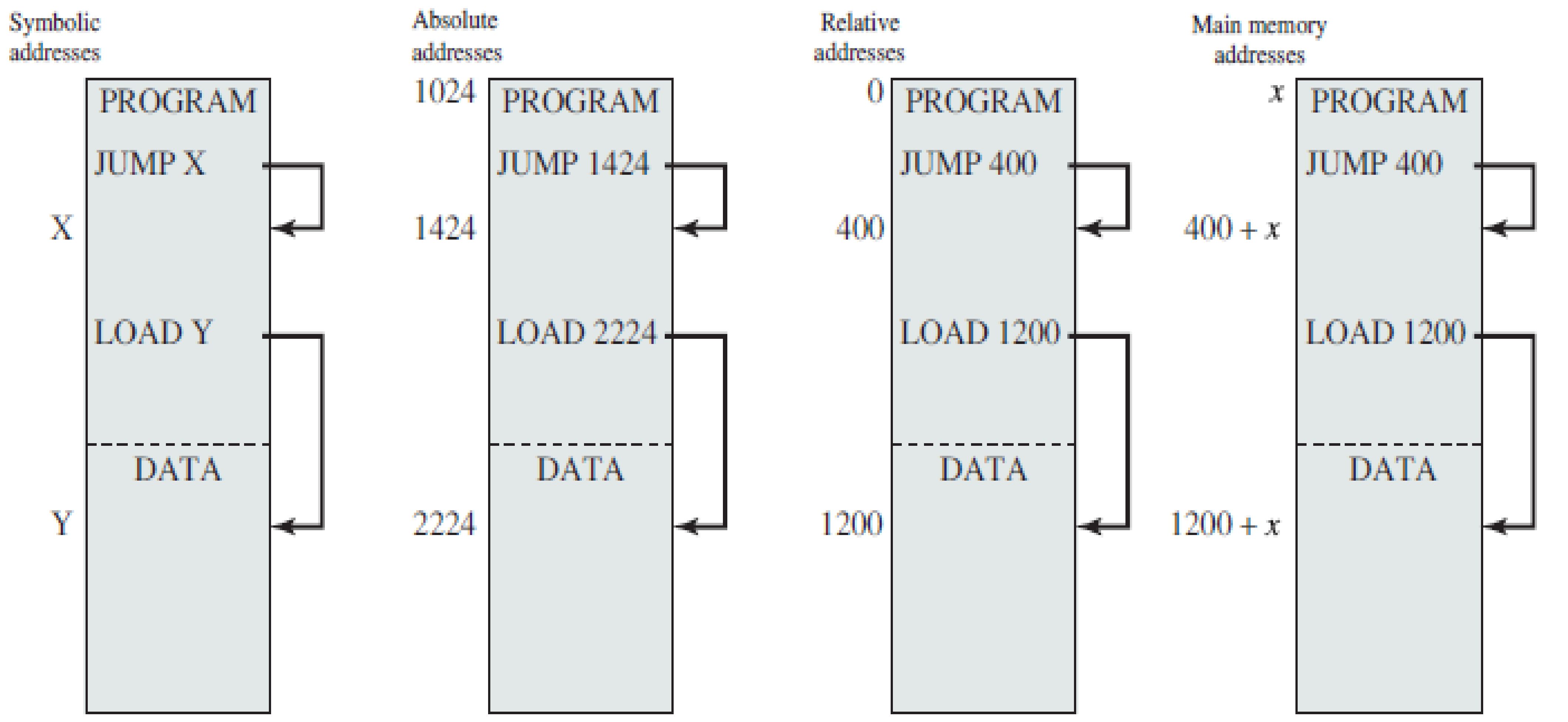
- A pair of **base** and **limit** registers define the logical address space



HW address protection with base and limit registers

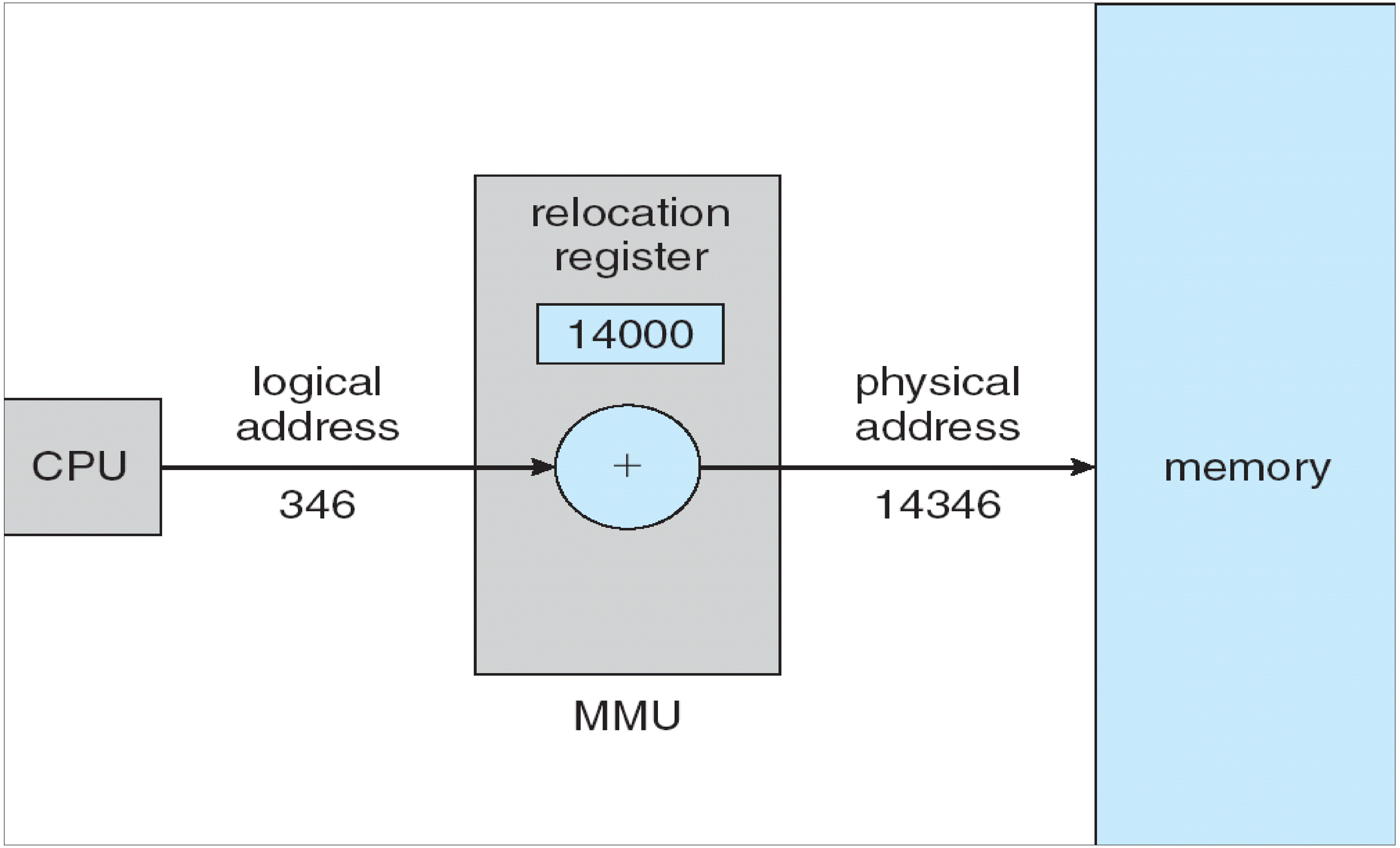


Absolute and relative address



(a) Object module (b) Absolute load module (c) Relative load module (d) Relative load module loaded into main memory starting at location *x*

Dynamic relocation using a relocation register



Memory Management Requirements

Relocation

- To support multiprogramming

Protection

- Against unwanted interferences

Sharing

- Flexibility

Logical organization

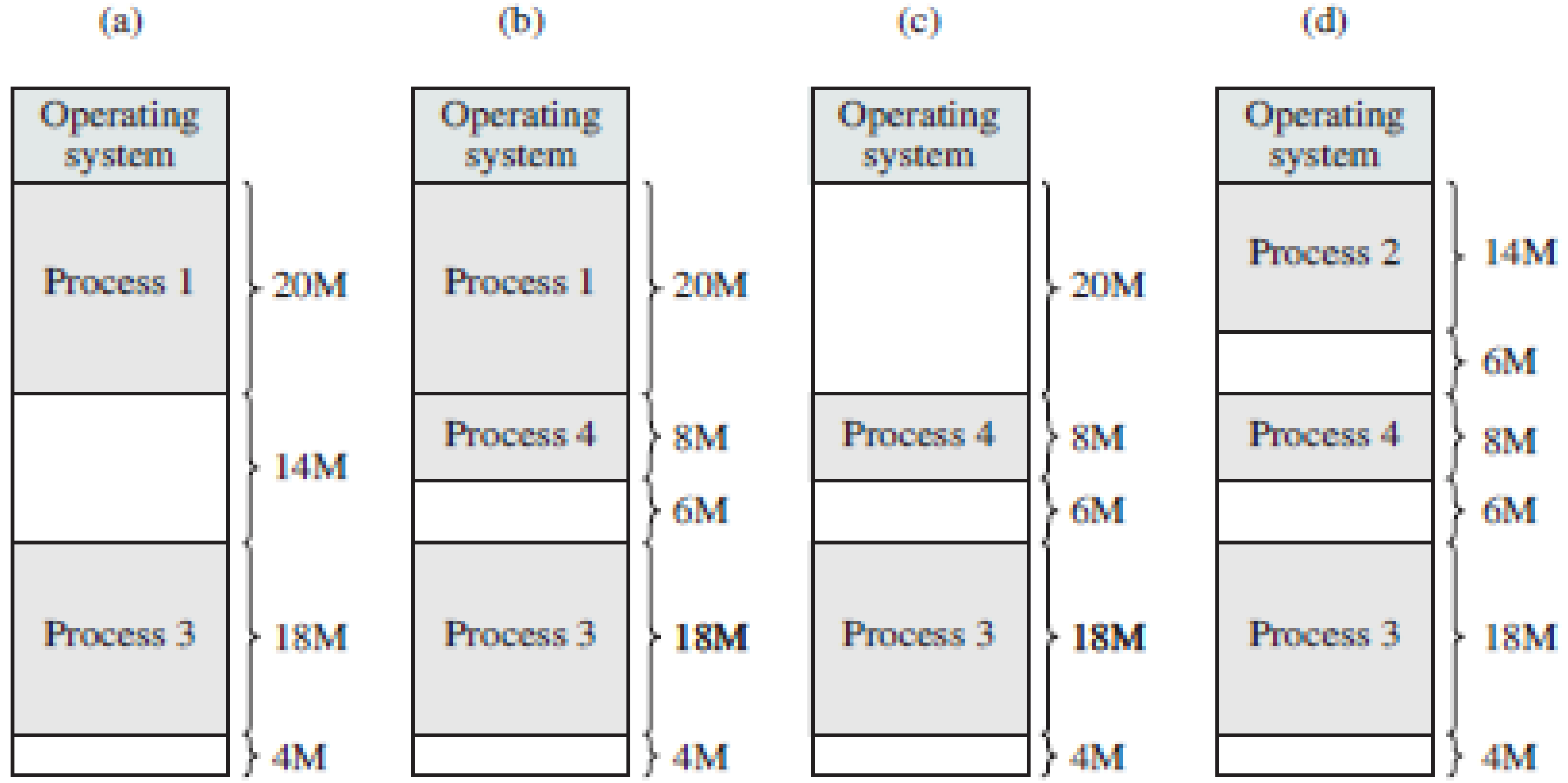
- Linear address space

Physical organization

- Space is limited

Contiguous Memory Allocation – dynamic partition

- Creating partitions dynamically of exactly the same size as the process
- No internal fragmentation
- Problem: External fragmentation



Fragmentation

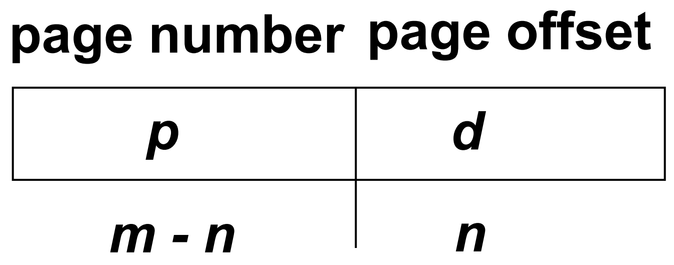
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- Reduce external fragmentation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

Paging

- Noncontiguous memory allocation
- Divide physical memory into fixed-sized blocks called **frames**
- Divide logical memory into blocks of same size called **pages**
- Set up a page table to translate logical to physical addresses
- To run a program of size ***n*** pages, need to find *n* free frames and load program
- Must keep track of all free frames
- Problem: internal fragmentation

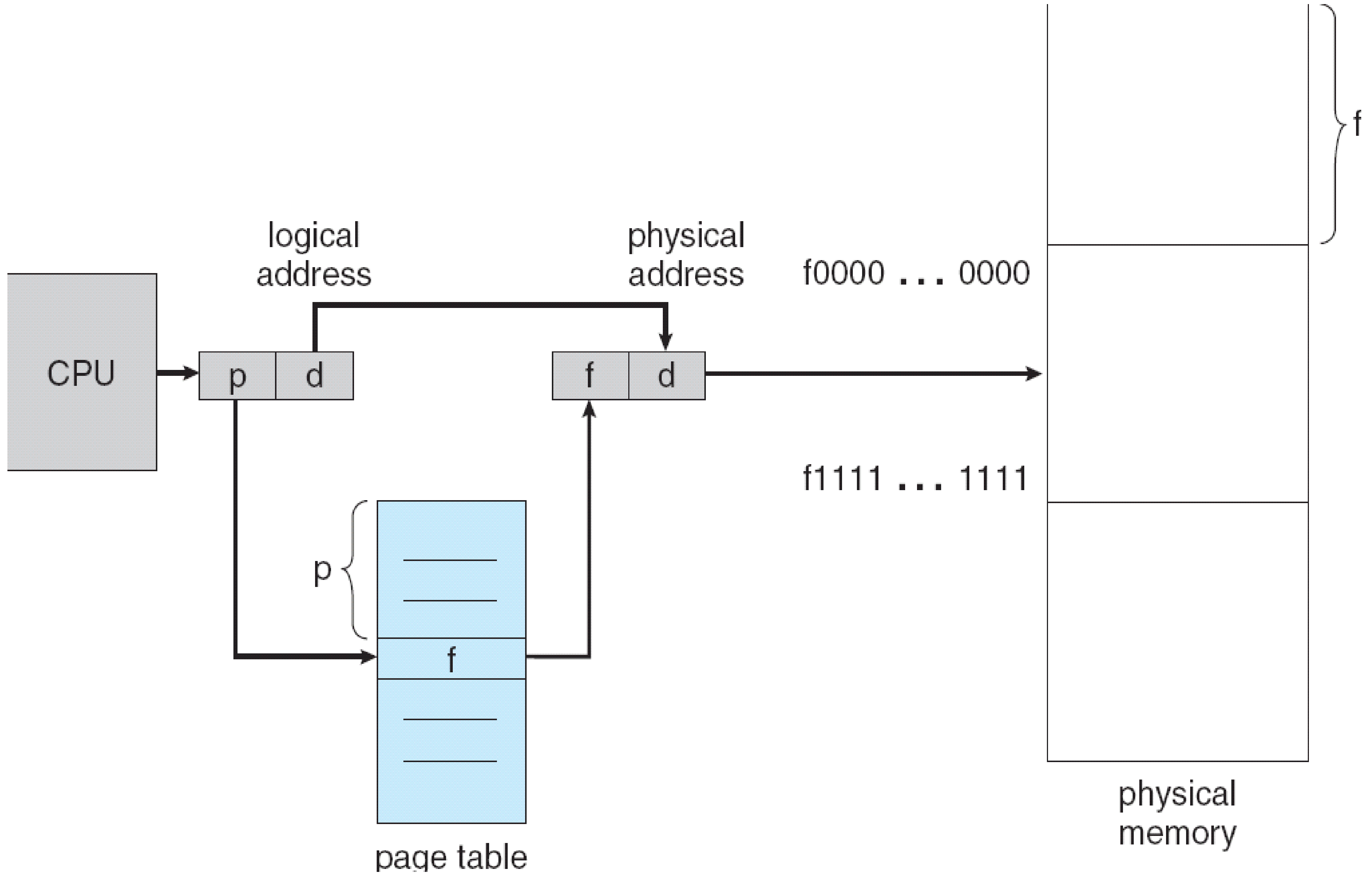
Address Translation Scheme

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

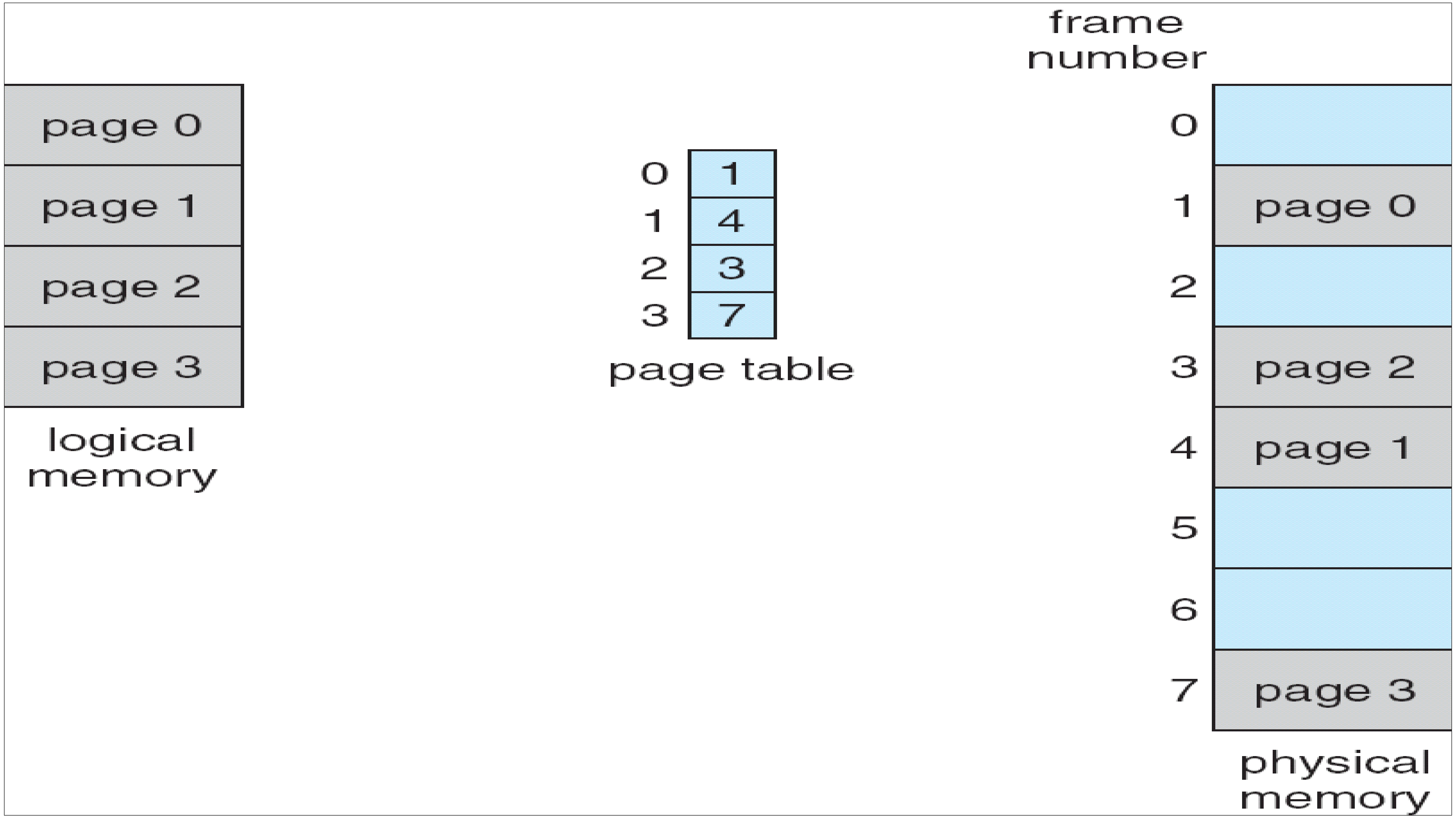


  - For given logical address space 2^m *and page size* 2^n

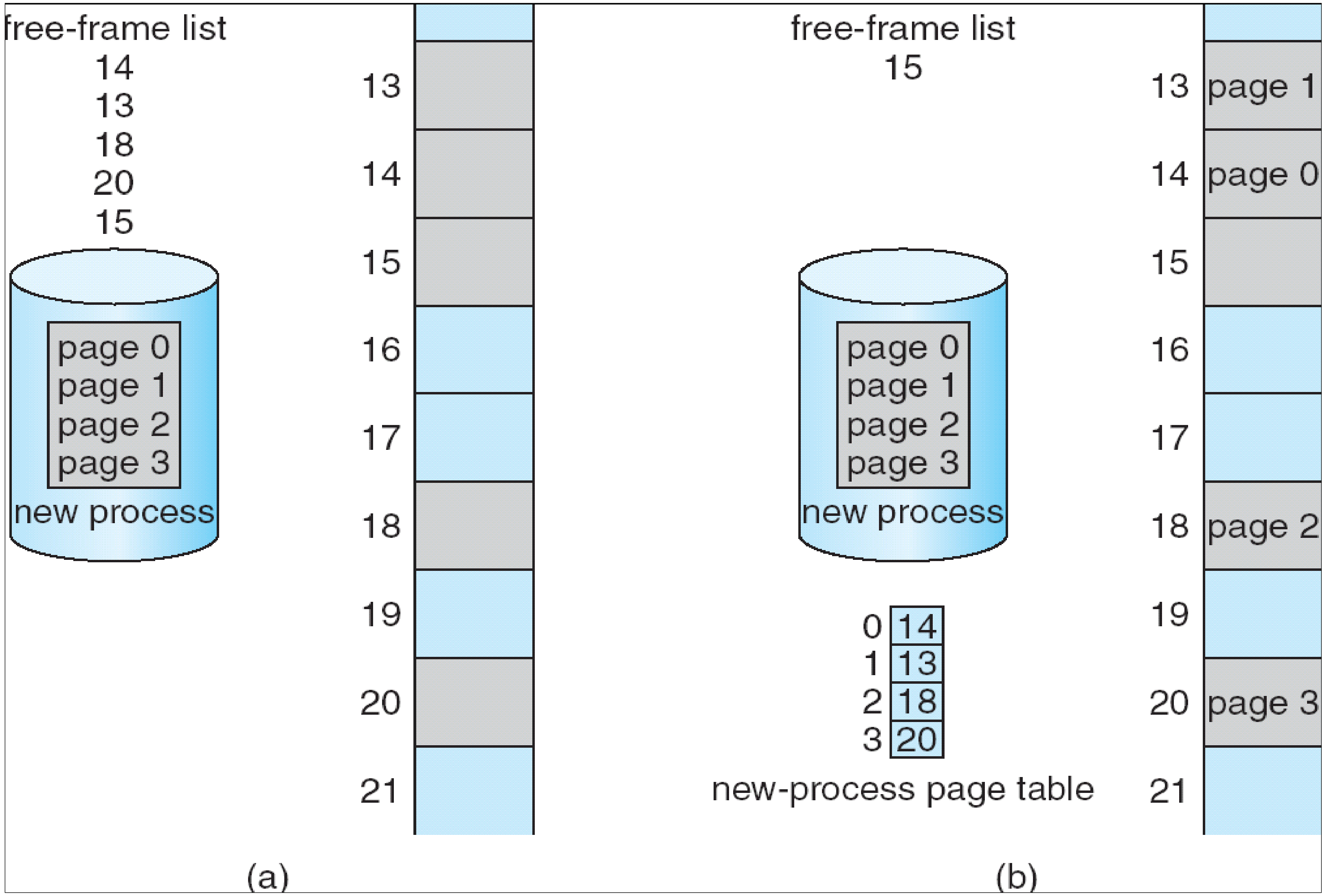
Paging Hardware



Paging Model of Logical and Physical Memory



Free Frames



Before allocation **After allocation**

Implementation of Page Table

- Scheme #1
  - 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 and one for the data/instruction.
- Scheme #2
  - 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)**

Translation look-aside buffers (TLBs)

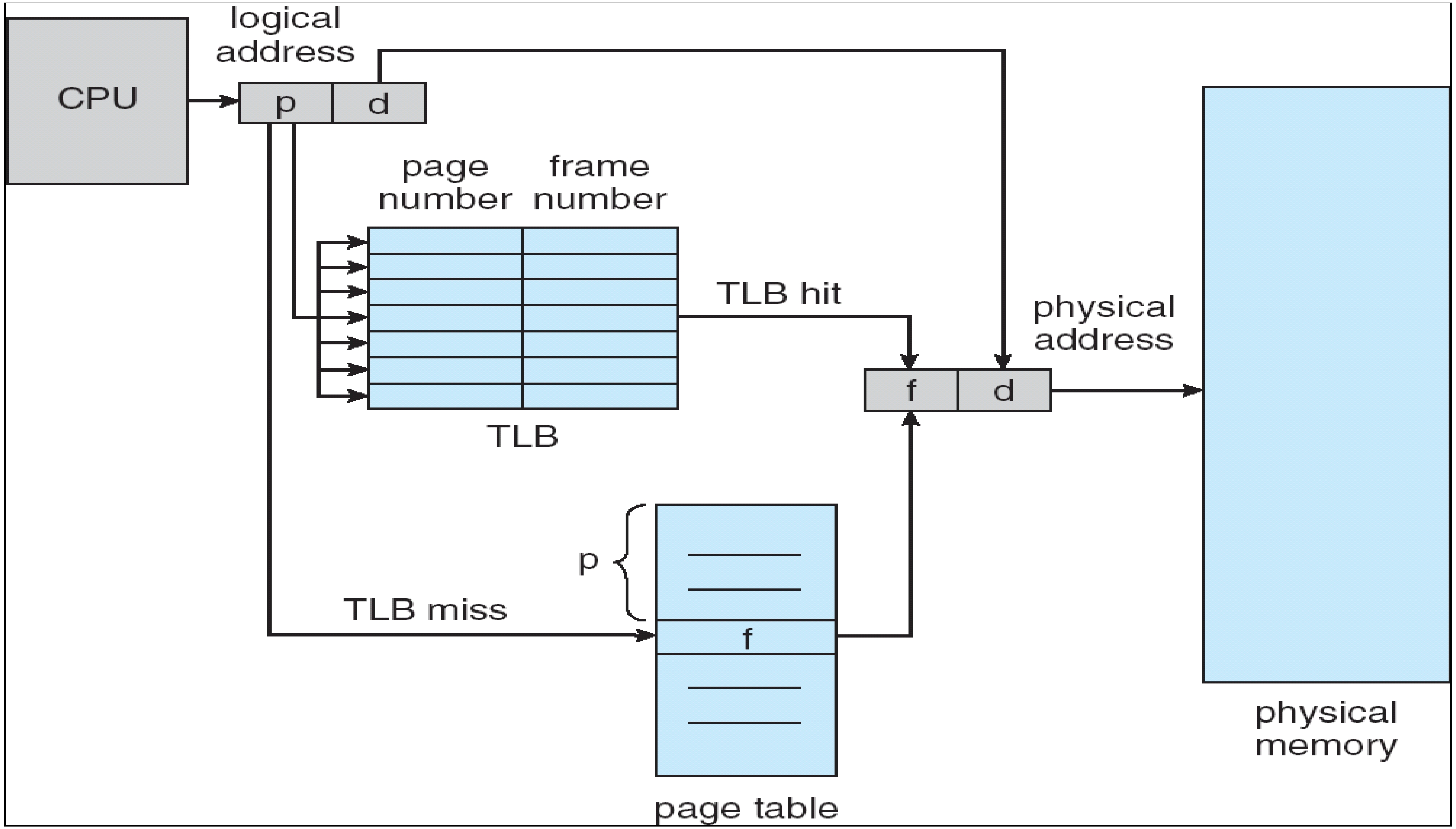
- Some TLBs store **address-space identifiers** (**ASIDs**) in each TLB entry – uniquely identifies each process to provide address-space protection for that process
  - Otherwise need to flush at every context switch
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into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

Page #	Frame #

Address translation (p, d)

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

Paging Hardware With TLB



Effective Access Time

- **Effective Access Time** (EAT)

  EAT = (TLBh * HitR) + (TLBm *(1 – HitR))

  - Hit ratio (HitR) – percentage of times that a page number is found in the associative registers
  - Time to access TLB and access memory when TLB hit (TLBh)
  - Time to access TLB and access page table in memory and access memory when TLB miss (TLBm)

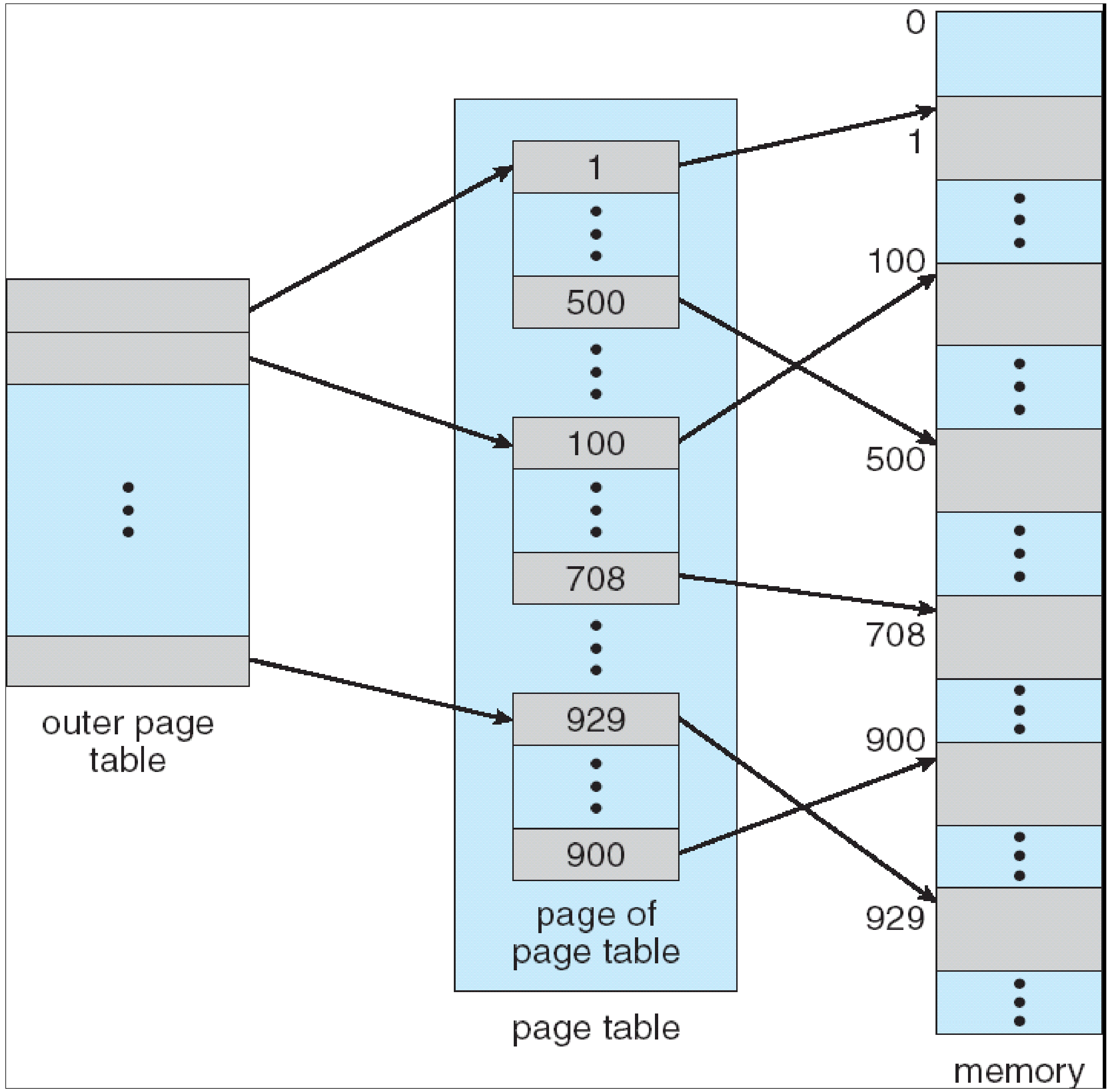
Effective Access Time (example)

- Consider HitR = 80%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.80 x 120 + 0.20 x 220
    = 96 + 44
    = 140ns

- Consider more realistic hit ratio -> HitR = 99%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.99 x 120 + 0.01 x 220
    = 118.8 + 2.2
    = 121ns

ε = Time for TLB search

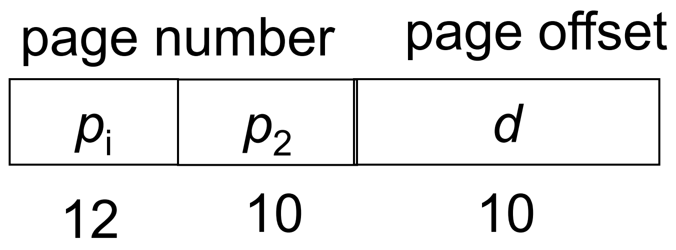
Hierarchical Page Tables



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

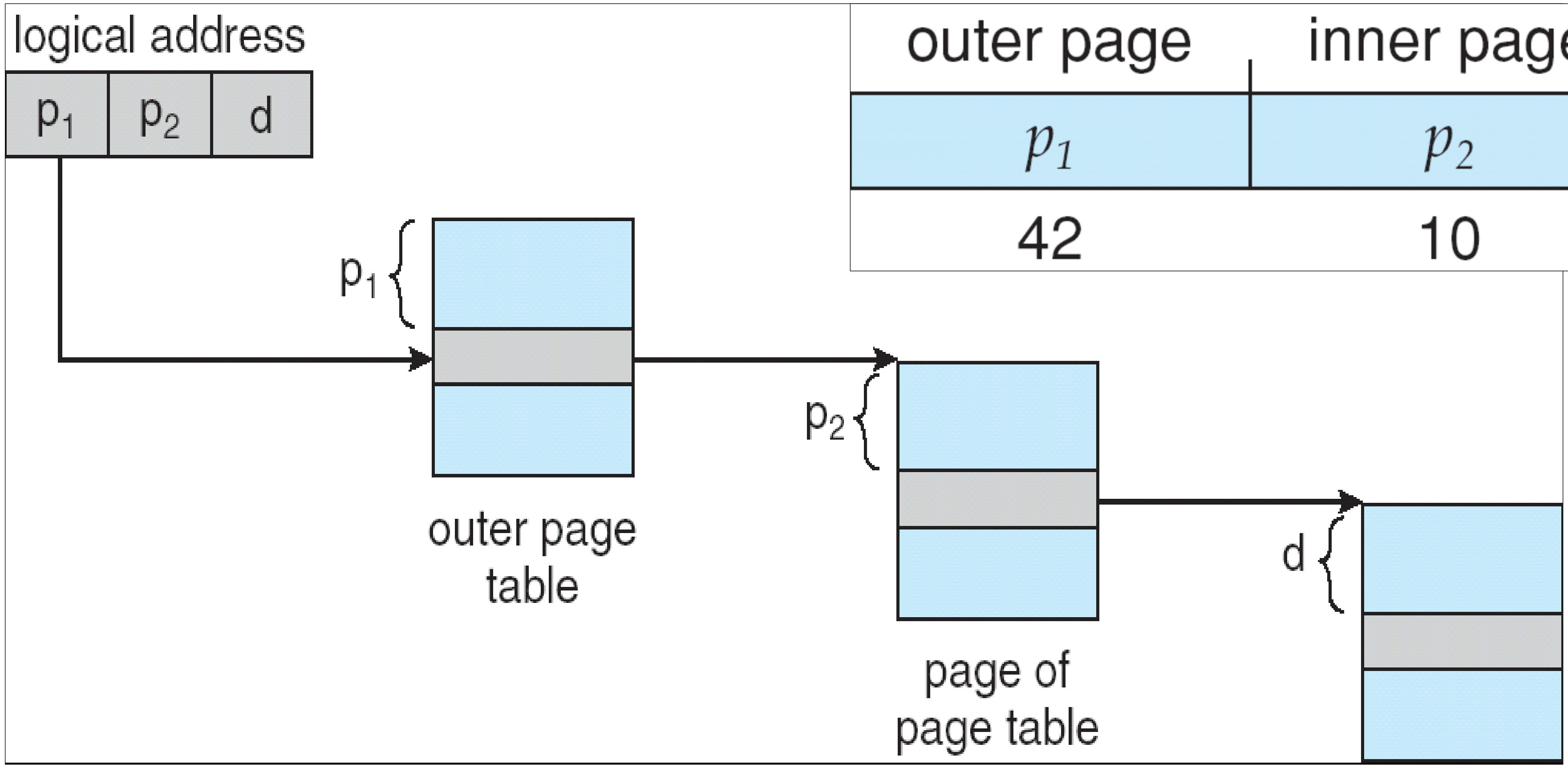
Two-Level Paging Example

- 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



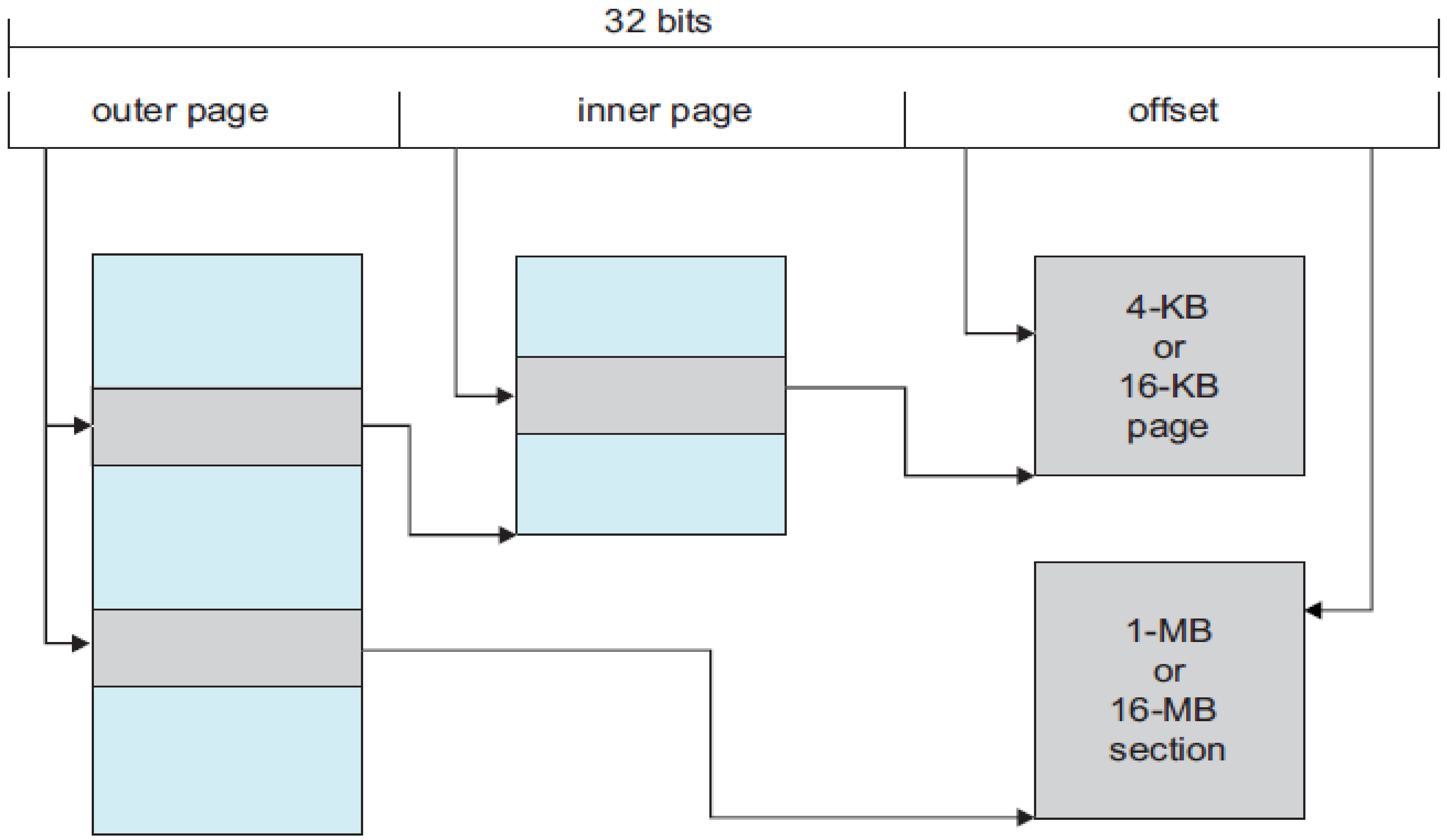
outer page	inner page	offset
p_1	p_2	d
42	10	12

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

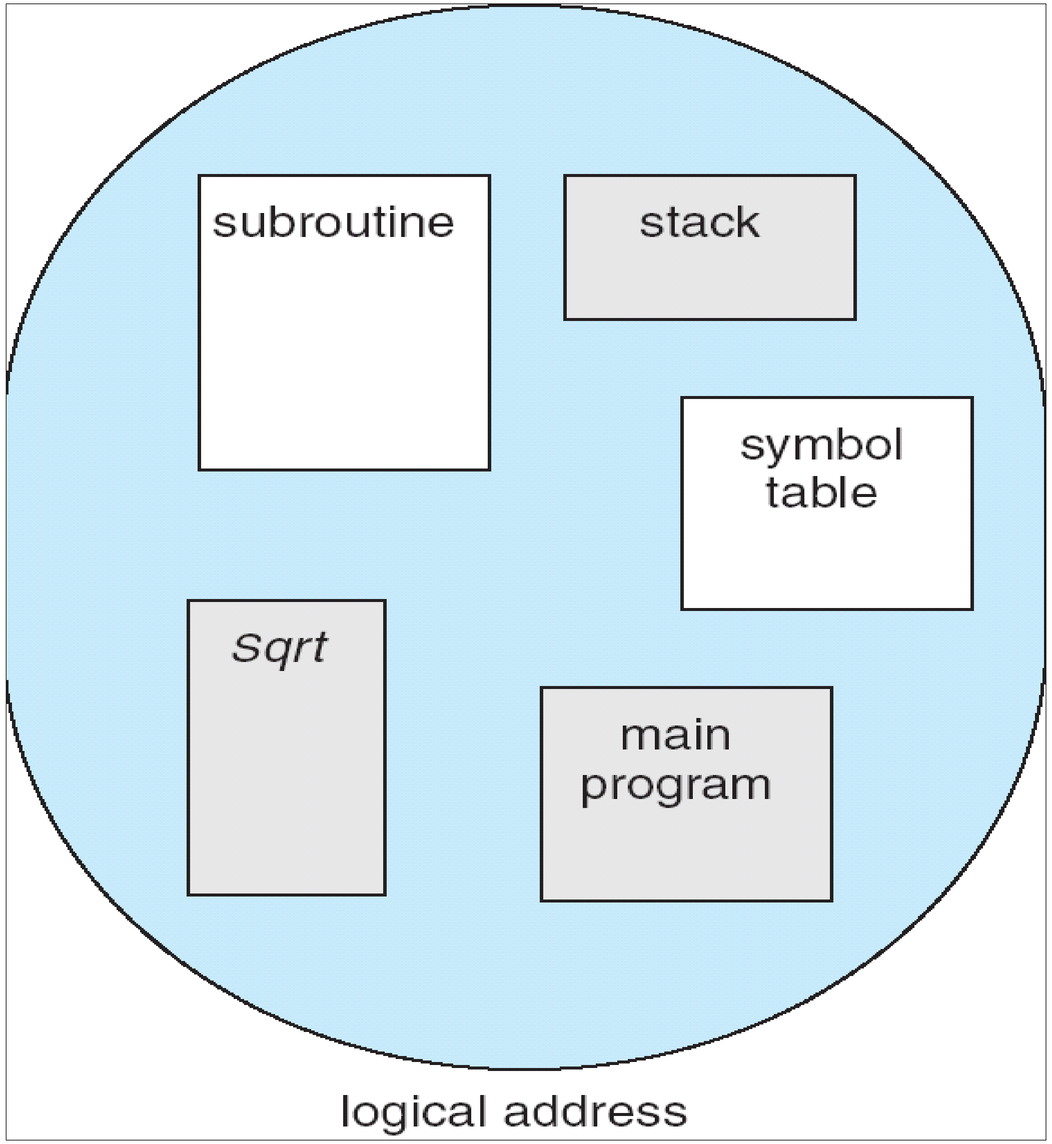
Paging example: ARM

- Dominant mobile platform chip
  - Apple iOS and Google Android devices
- 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
- Two levels of TLBs
  - Outer level has two micro TLBs (data, instruction)
  - Inner is single main TLB
- First inner is checked, on miss outers are checked, and on miss page table walk performed by CPU

Paging example: ARM (cont.)



Segmentation – user view

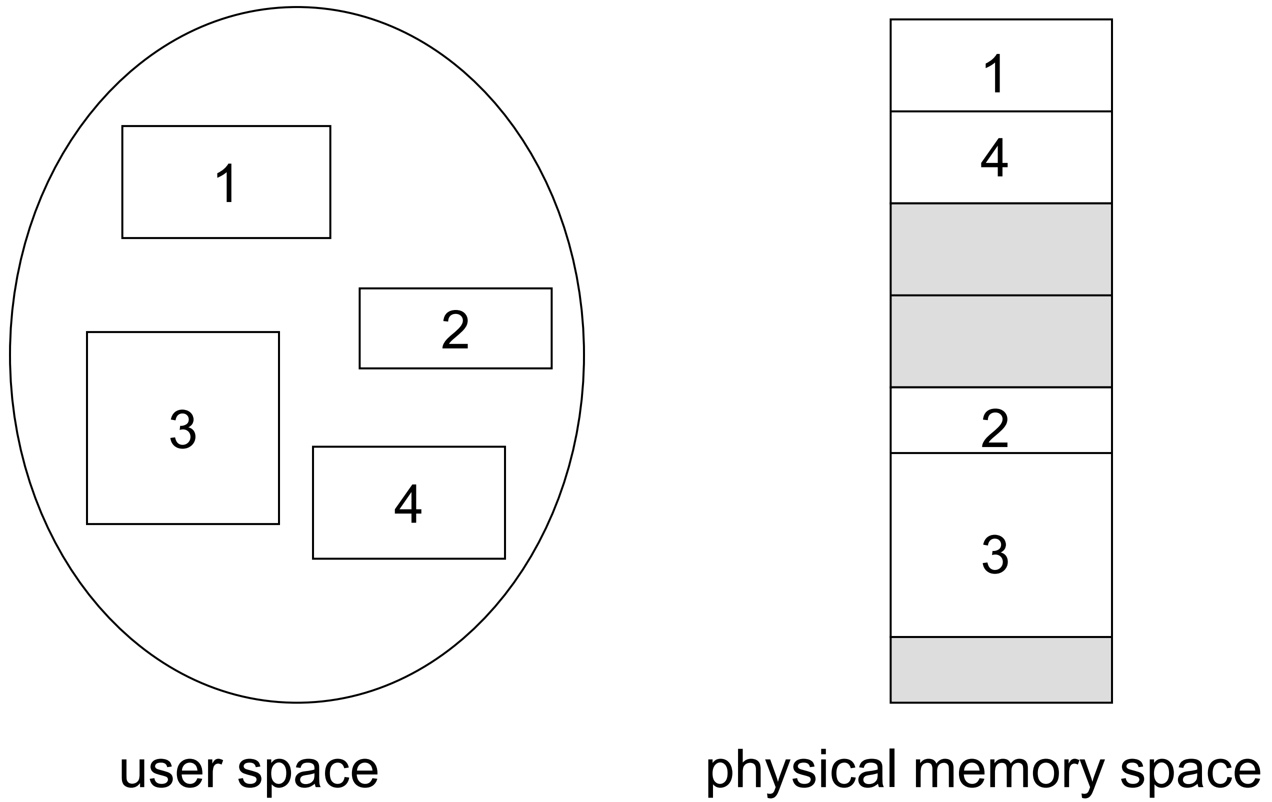


- Memory-management scheme that supports user view of memory
- A program is a collection of segments. A segment is a logical unit such as:
  - main program,
  - procedure,
  - function,
  - method,
  - object,
  - local variables, global variables,
  - common block,
  - stack,
  - symbol table, arrays

Segmentation – logical 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

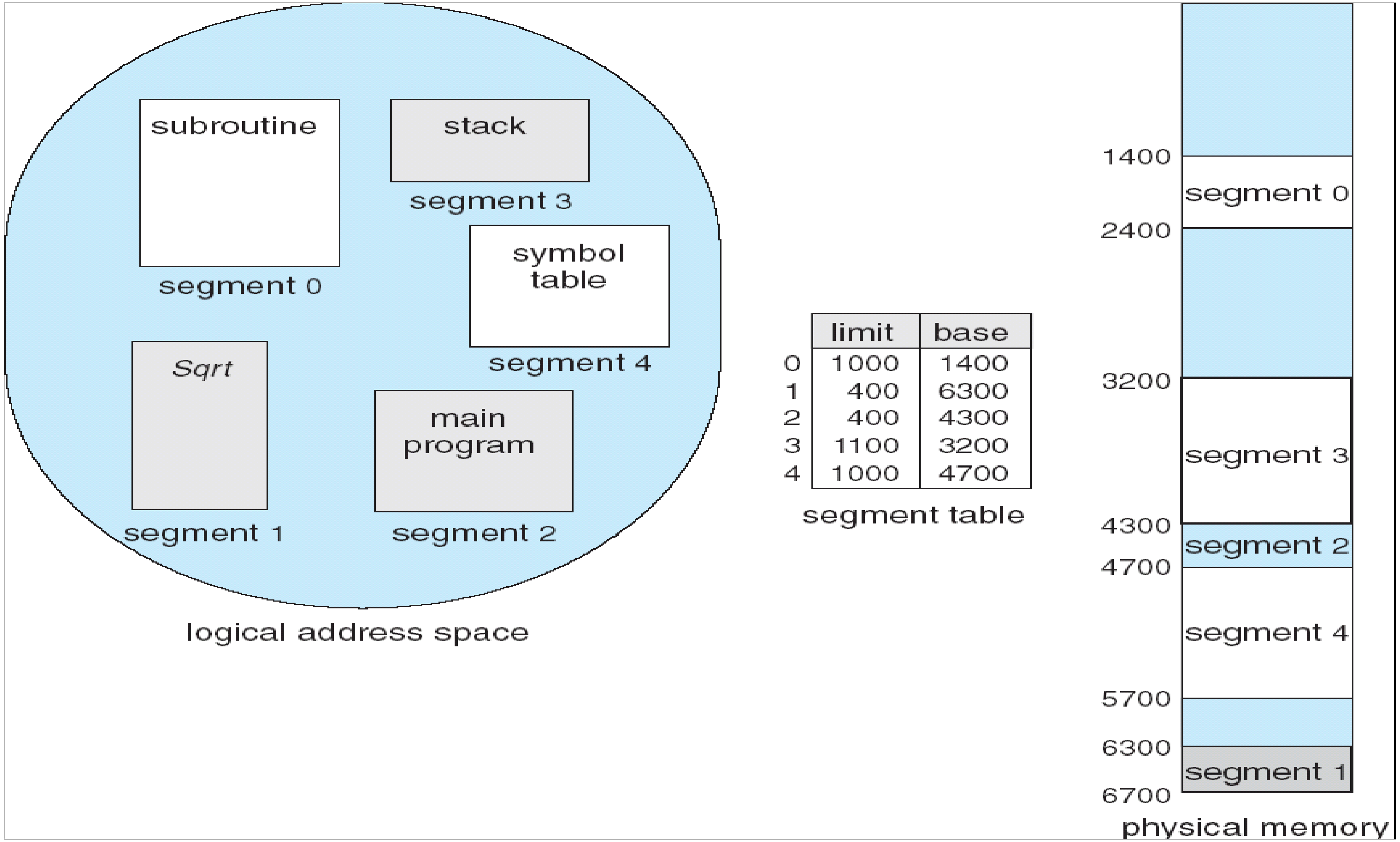


Segmentation Architecture

- E.g., 16-bit Intel 8086 & 8088
- 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**

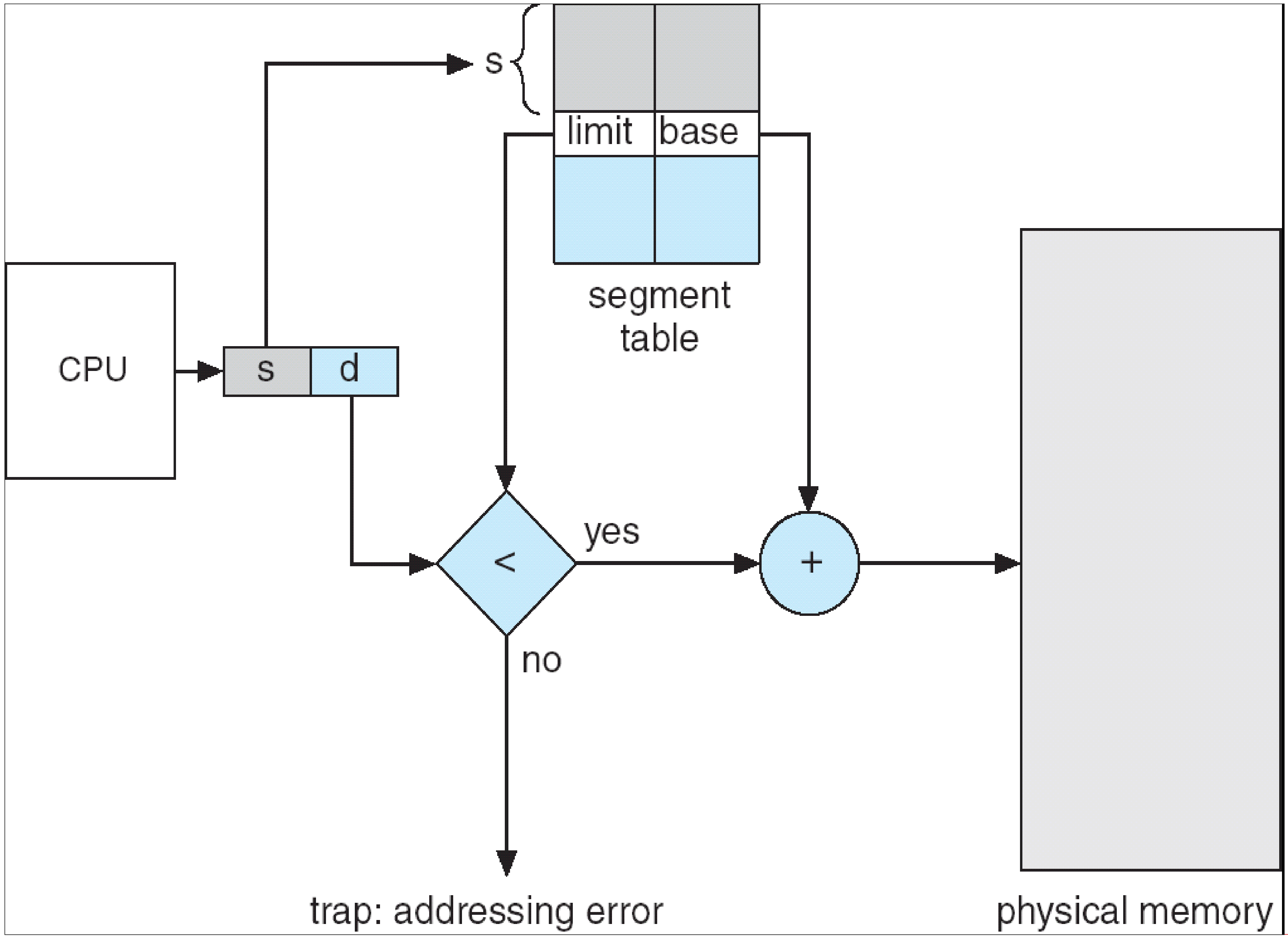
Example of Segmentation

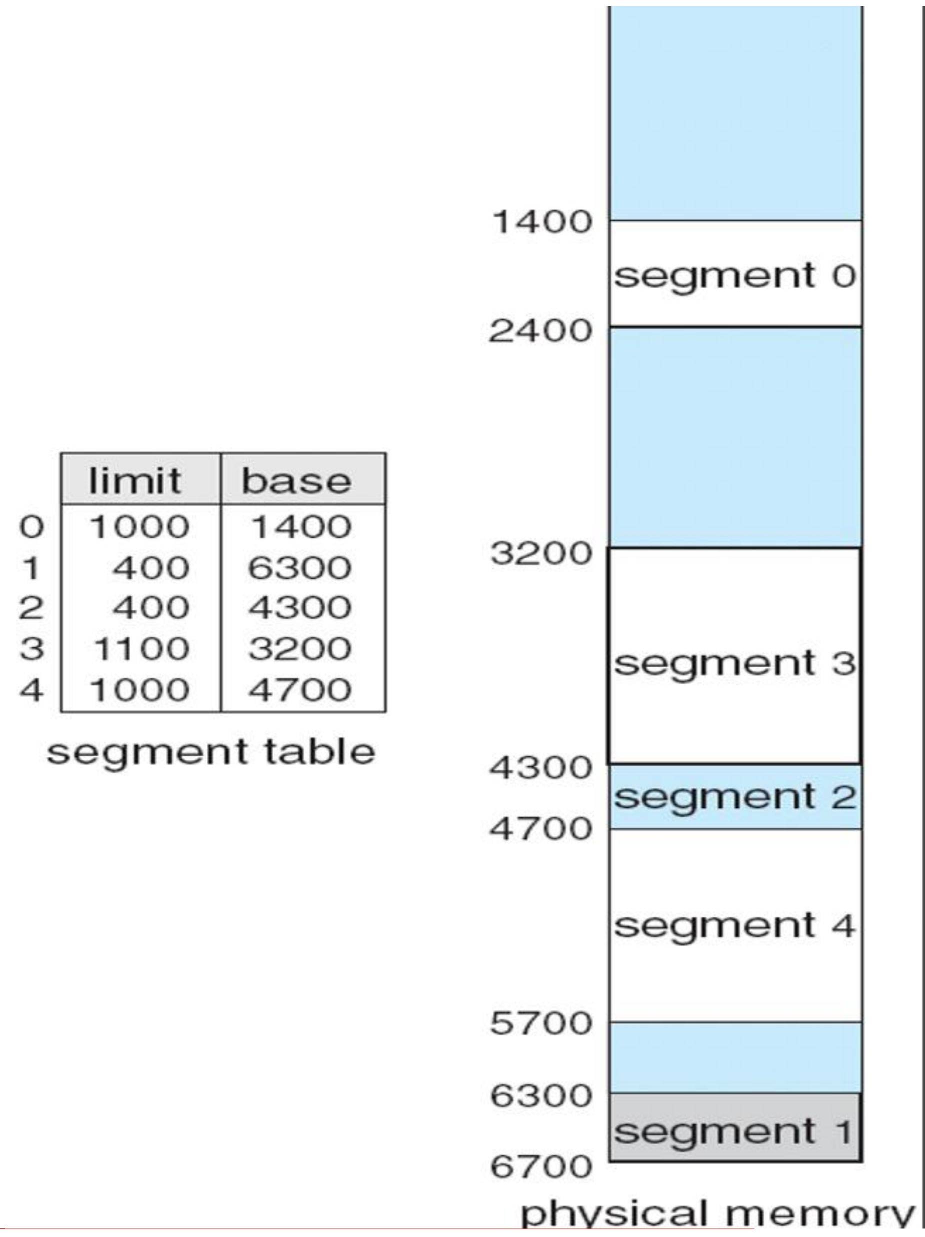


	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table

Segmentation Hardware





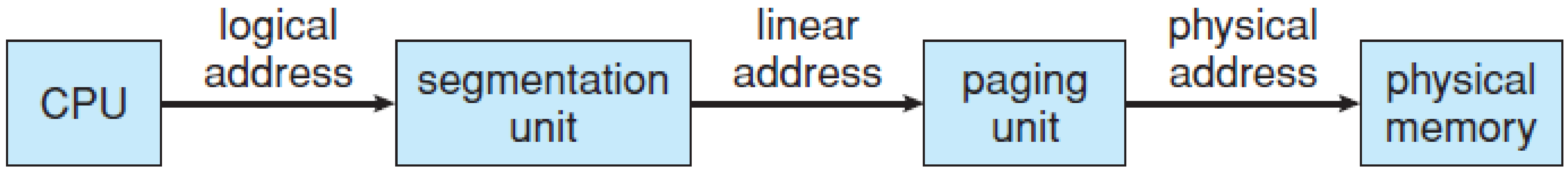
	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

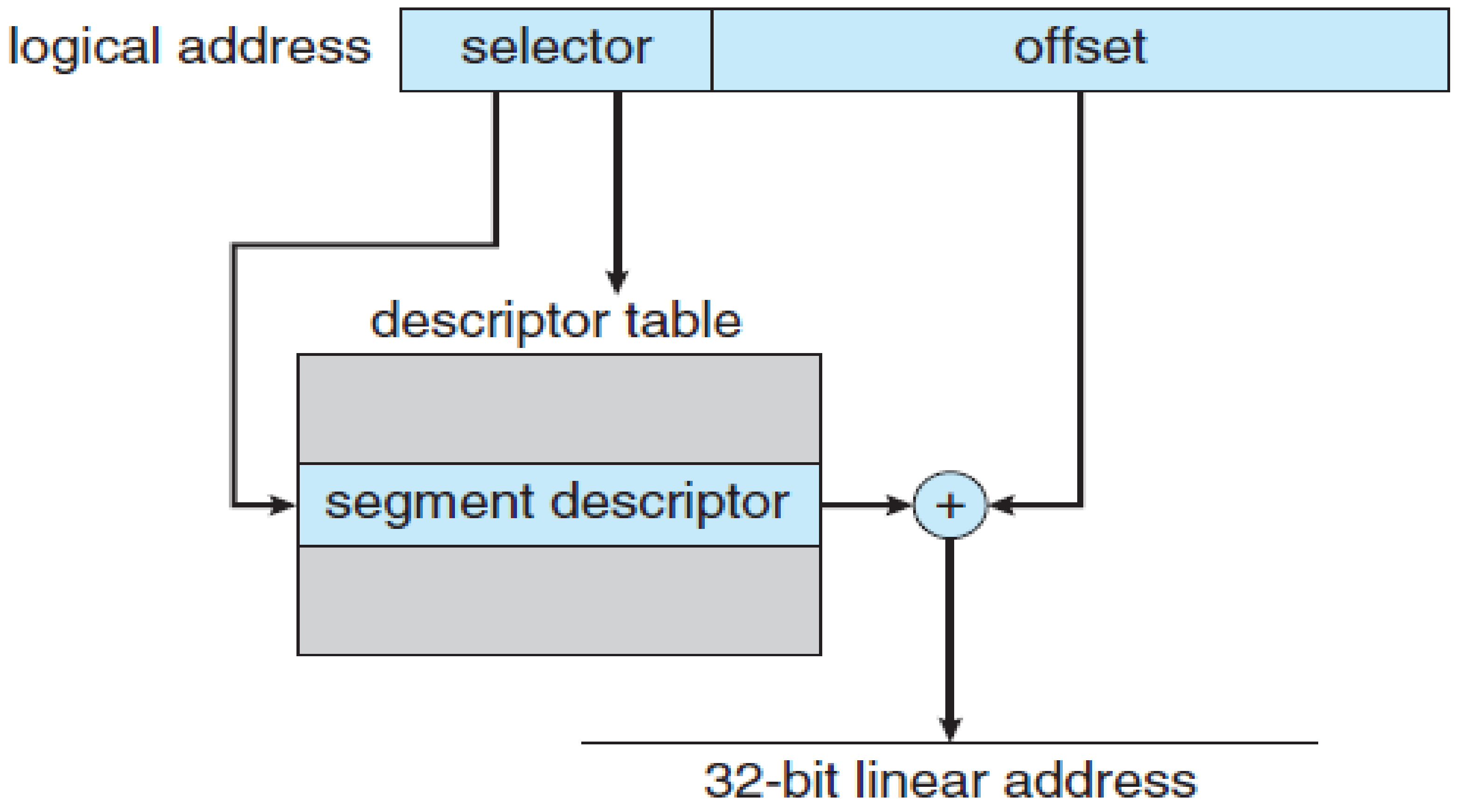
segment table

Segmentation with paging

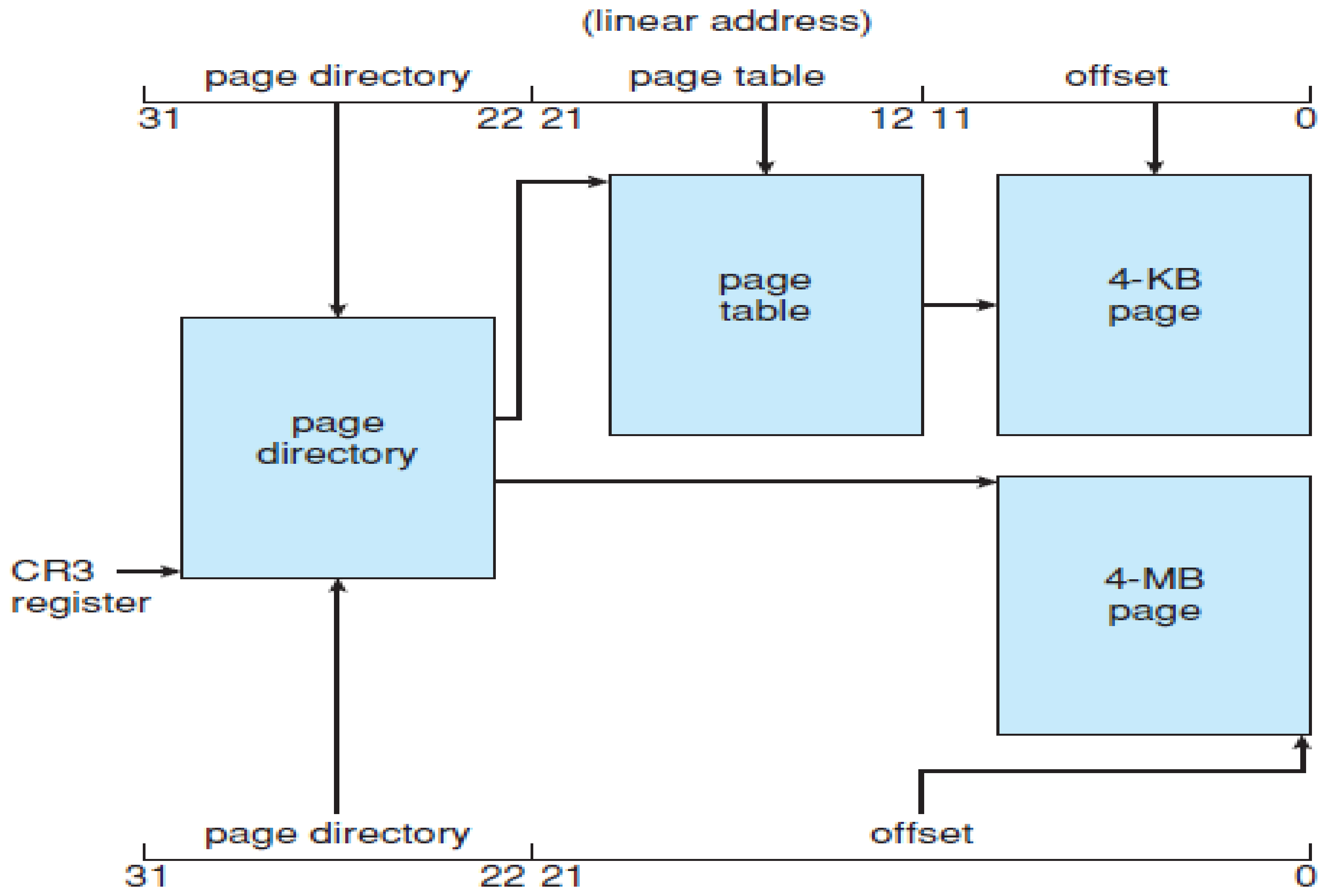
- Paging is transparent to the running process
- Segmentation requires that programs somehow be structured so that they are divided into logical parts with different address space
- With the proper hardware design a segmented program architecture can be run in combination with a paged memory architecture
  - Each segment is paged
- E.g., Intel 32-bit architecture (IA-32)

Intel 32-bit architecture (IA-32)





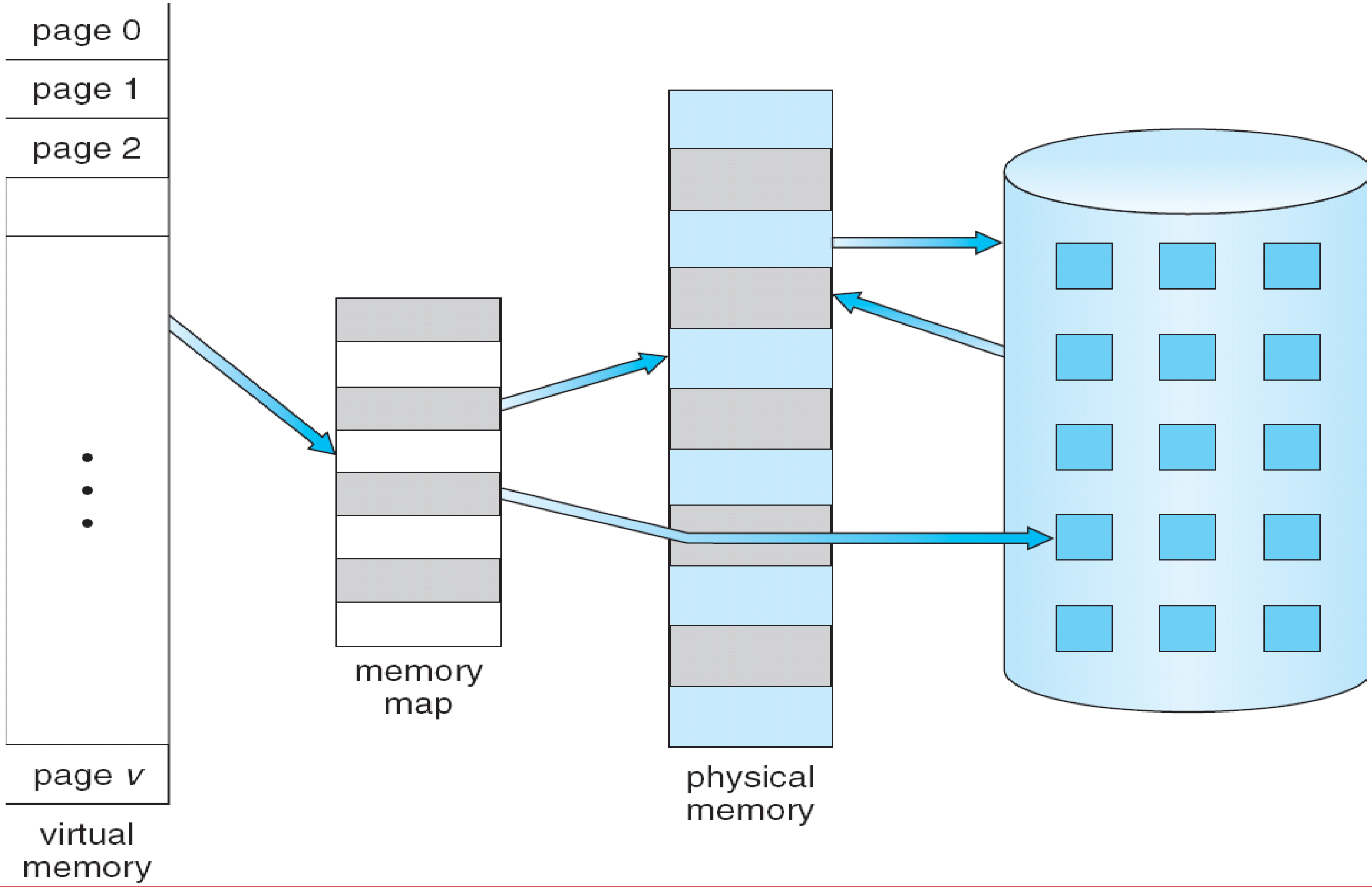
Intel 32-bit architecture (IA-32)



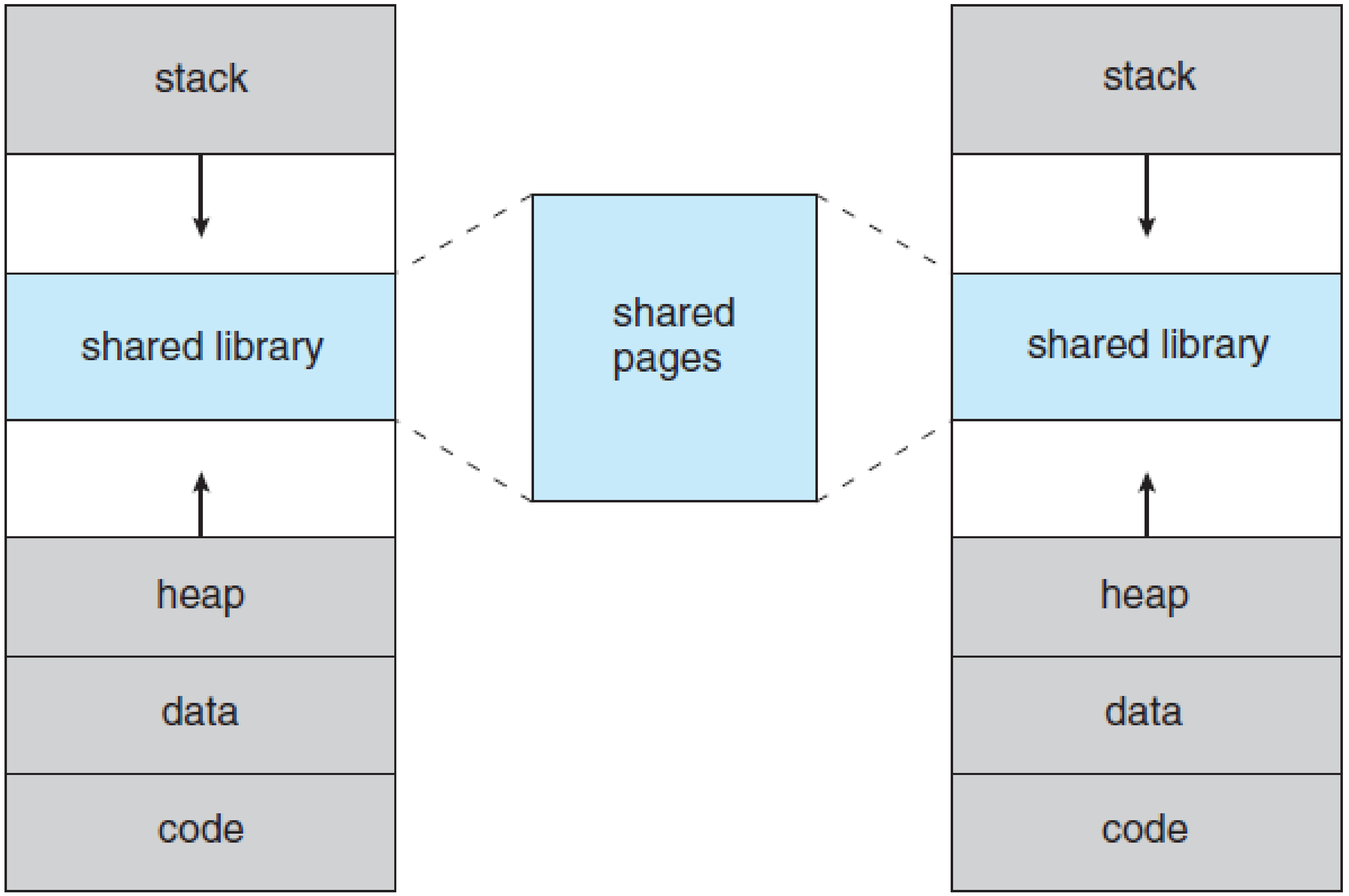
Virtual Memory

- Separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

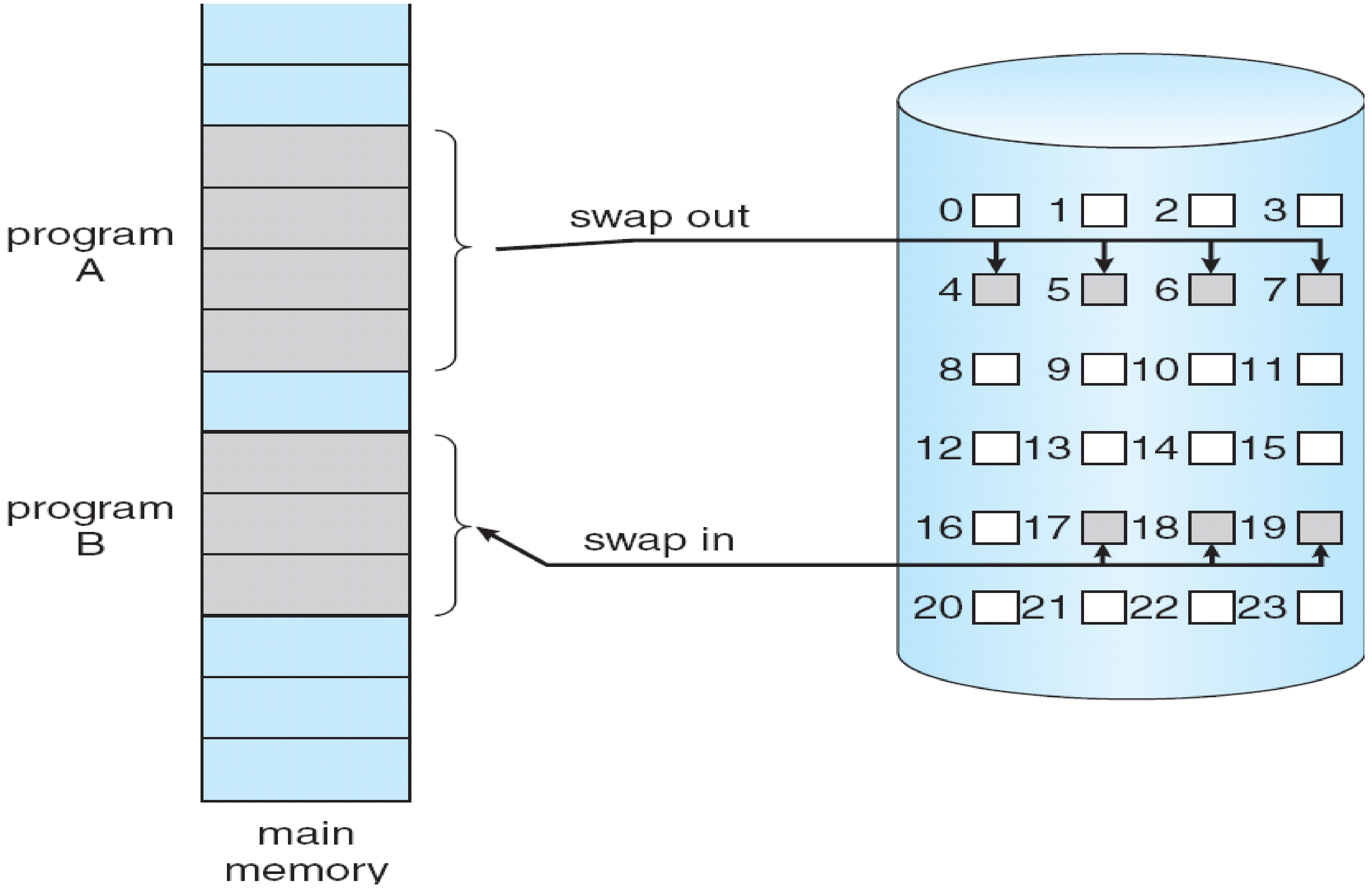
Virtual Memory That is Larger Than Physical Memory



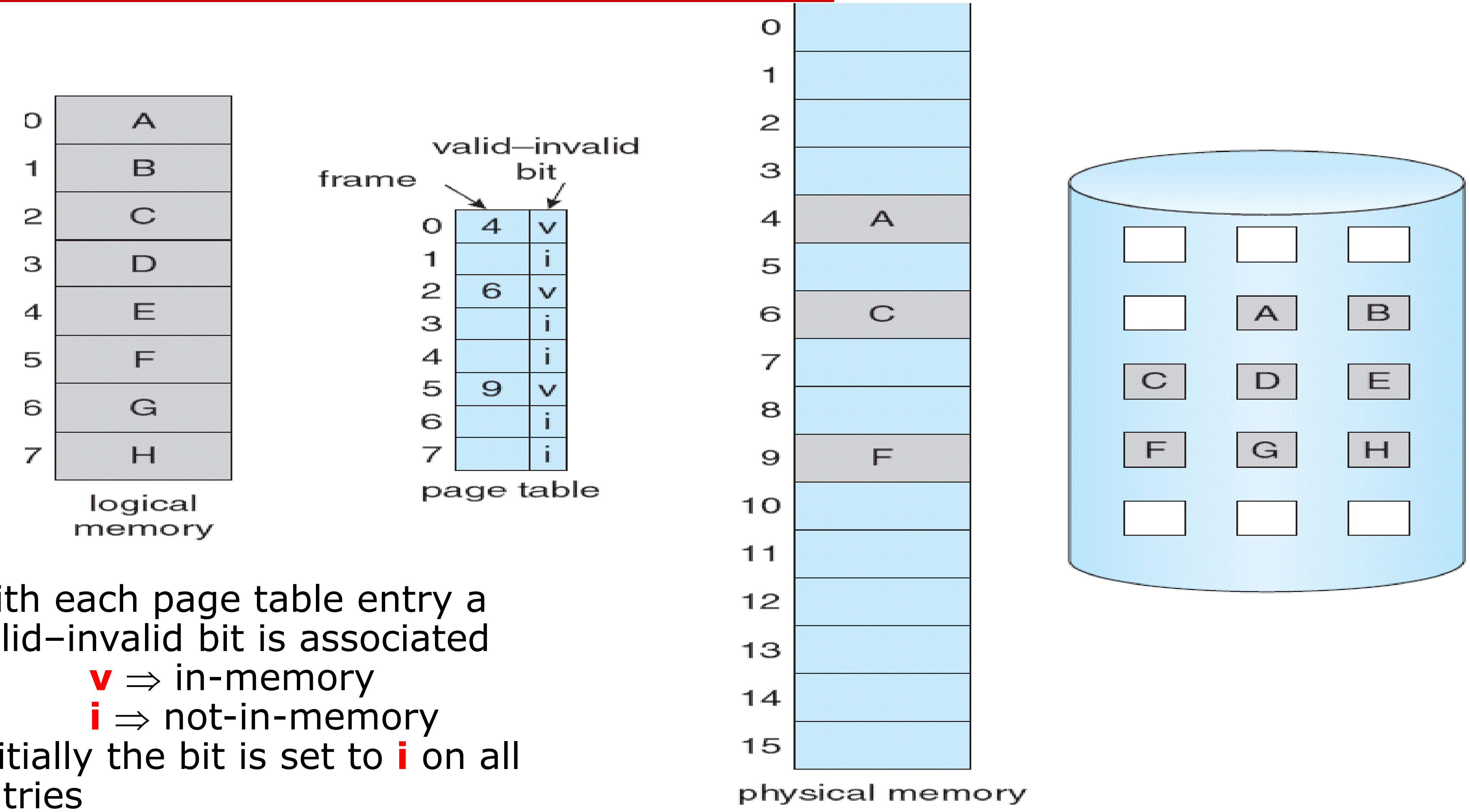
Shared pages with virtual memory



Transfer of a Paged Memory to Contiguous Disk Space



Page Table When Some Pages Are Not in Main Memory



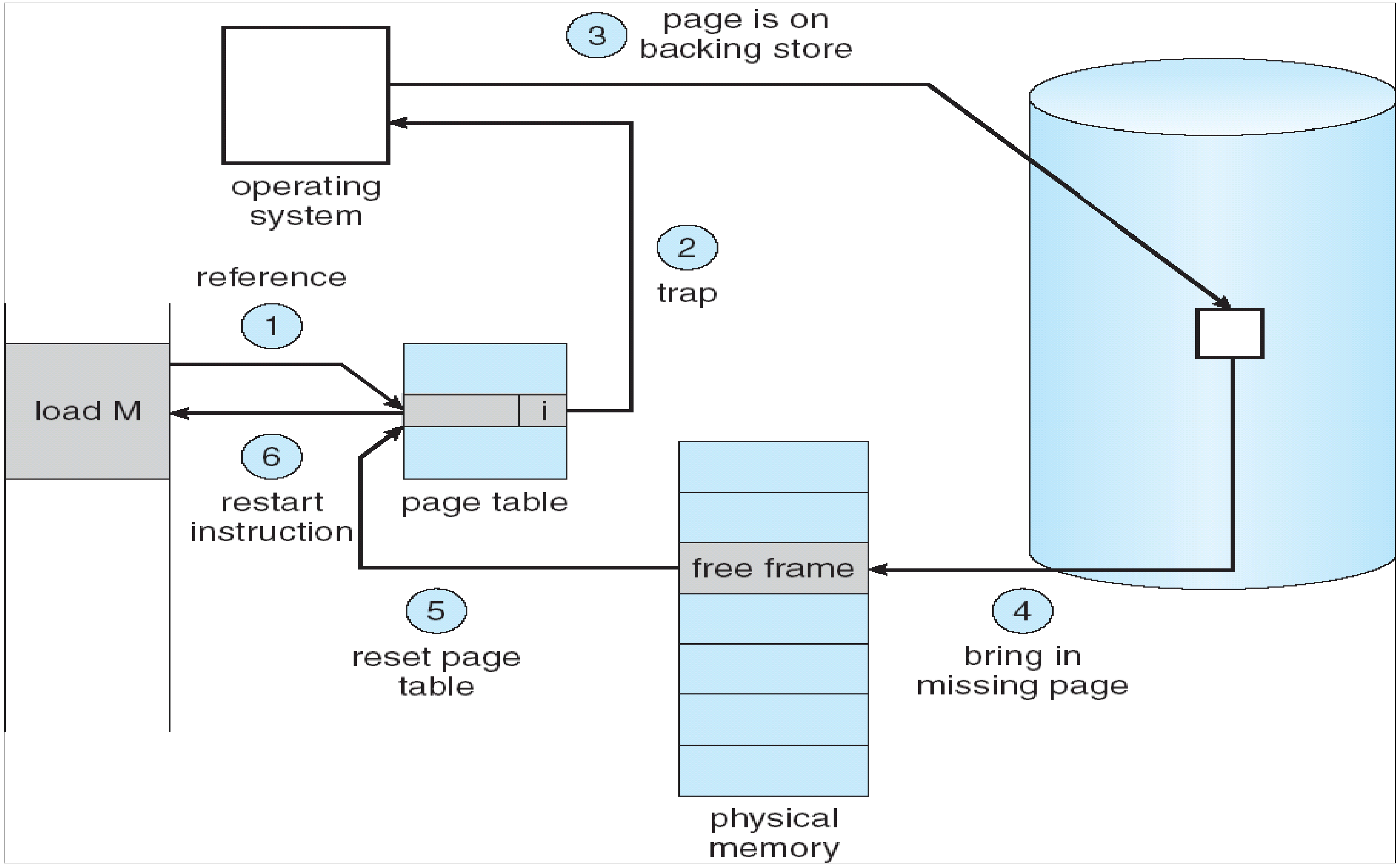
With each page table entry a valid–invalid bit is associated

v ⇒ in-memory

i ⇒ not-in-memory

Initially the bit is set to **i** on all entries

Steps in Handling a Page Fault



Performance of Demand Paging

- Page Fault Rate $0 \le p \le 1.0$
  - if p = 0 no page faults
  - if p = 1, every reference is a fault

- Effective Access Time (EAT)

$$\begin{aligned}\text{EAT} = {} & (1 - p) \times \text{memory access} \\ & + p\,(\text{page fault overhead} \\ & \qquad + \text{swap page out} \\ & \qquad + \text{swap page in} \\ & \qquad + \text{restart overhead}\ )\end{aligned}$$

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 + p (8 milliseconds)

  = (1 – p) x 200 + p x 8,000,000

  = 200 + p x 7,999,800
- If one access out of 1,000 causes a page fault, then

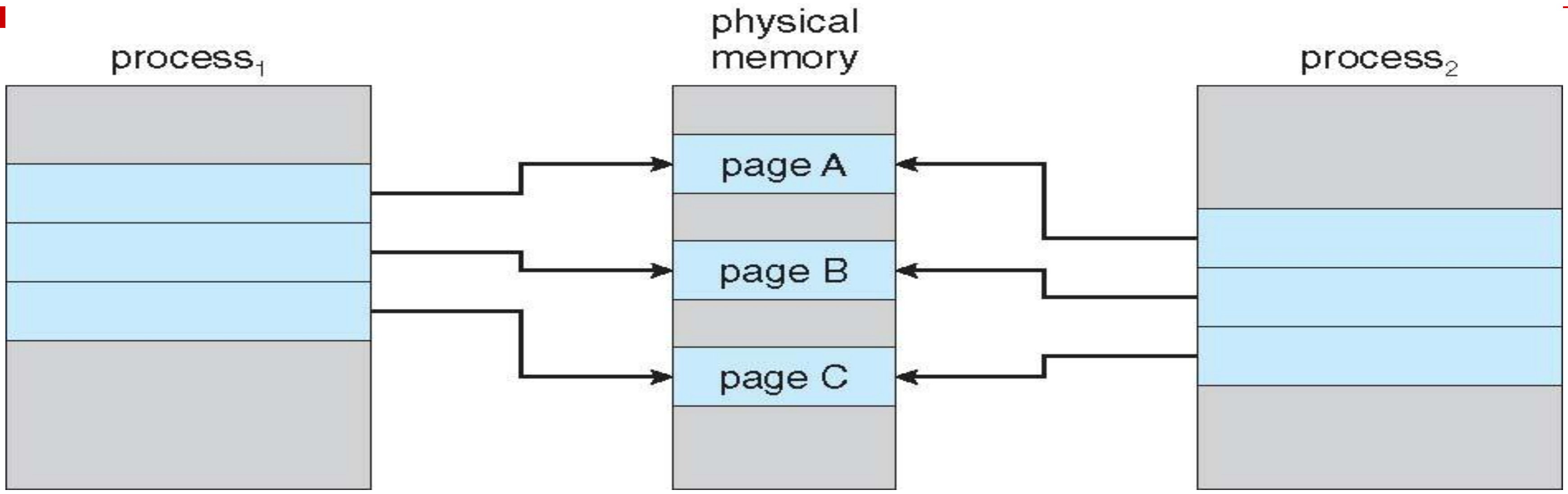
  EAT = 8.2 microseconds

  This is a slowdown by a factor of 40!!

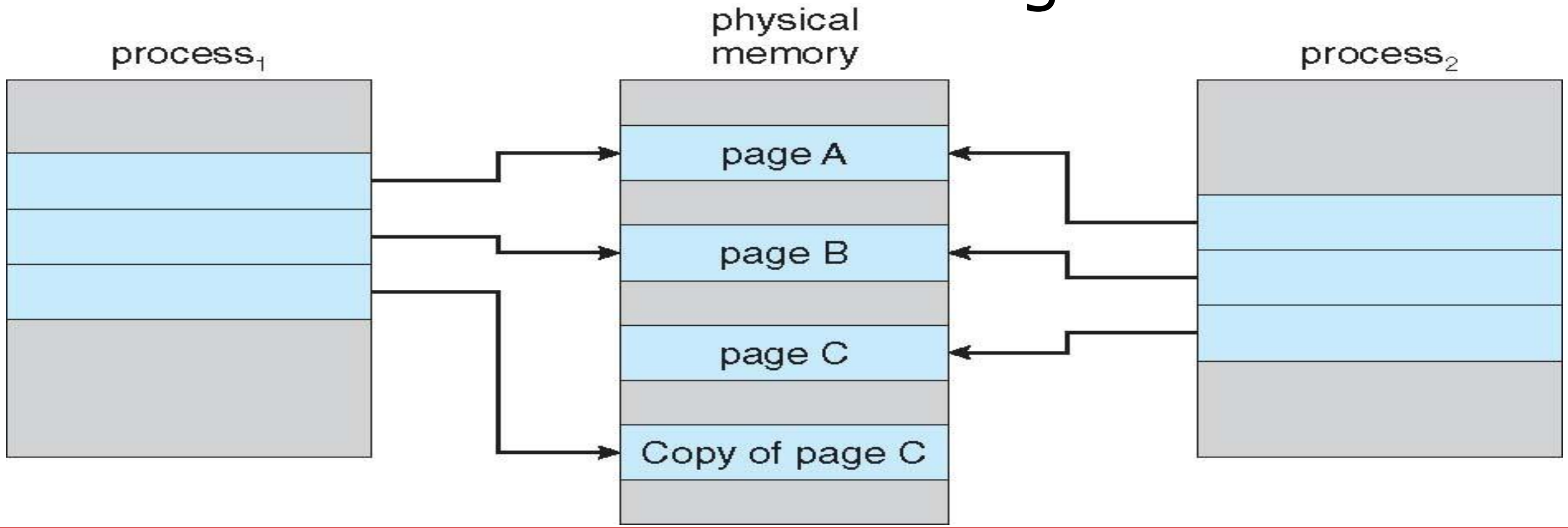
Copy-on-Write

- **Copy-on-Write** (COW) allows both parent and child processes to initially ***share*** the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- vfork() variation on fork() system call has parent suspend and child using copy-on-write address space of parent
  - Designed to have child call exec()
  - Very efficient

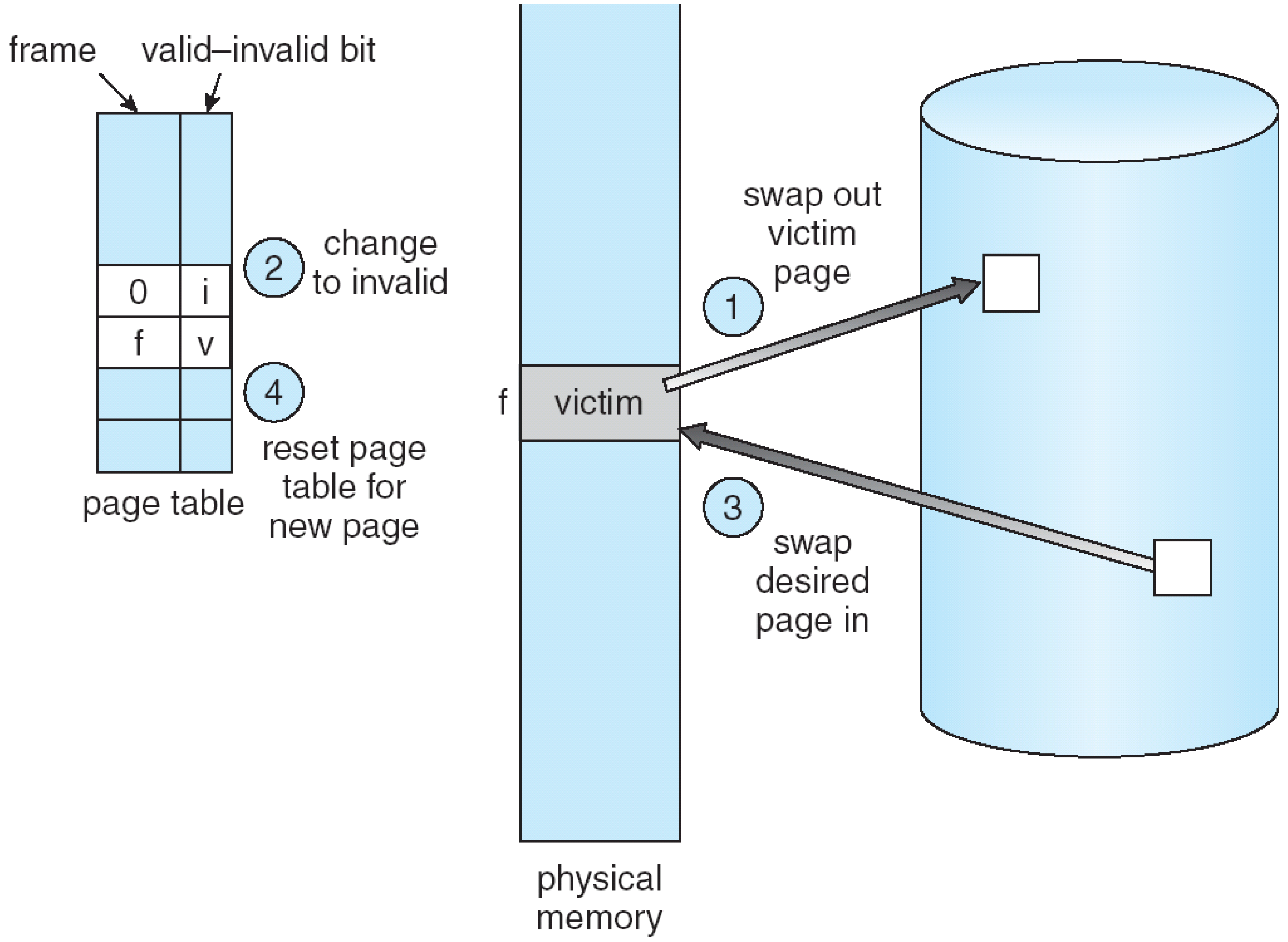
Before Process 1 Modifies Page C



After Process 1 Modifies Page C

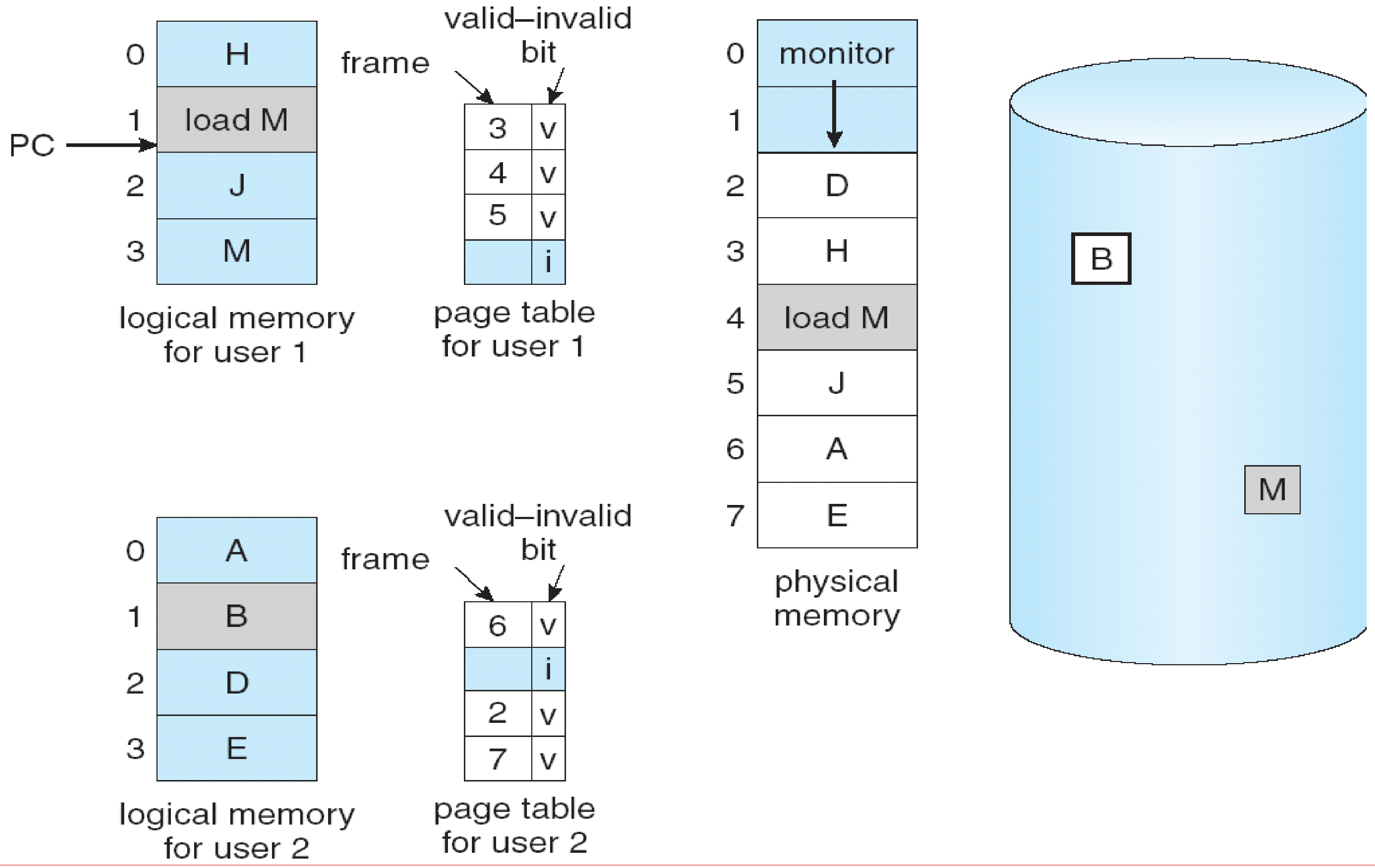


What happens if there is no free frame?



- Page replacement – find some page in memory, but not really in use, swap it out
  - algorithm
  - performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

Need For Page Replacement



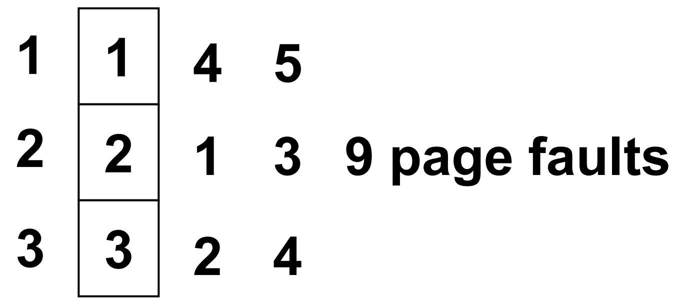
Page Replacement Algorithms

- Want lowest page-fault rate
- Algorithms
  - First-in First-out (FIFO)
  - Optimal Page Replacement
  - Least Recently Used (LRU)
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
- In all our examples, the reference string is

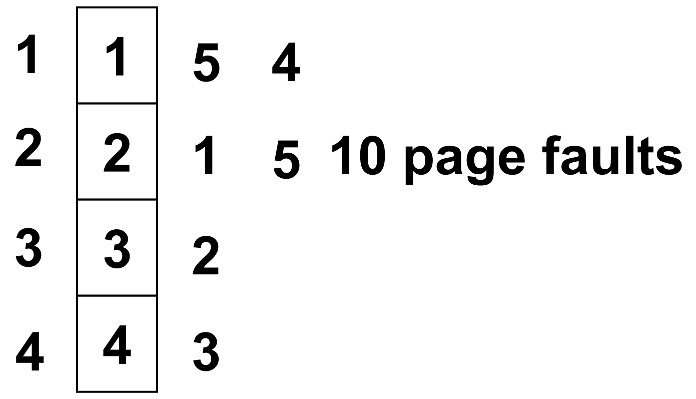
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

First-In-First-Out (FIFO) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)

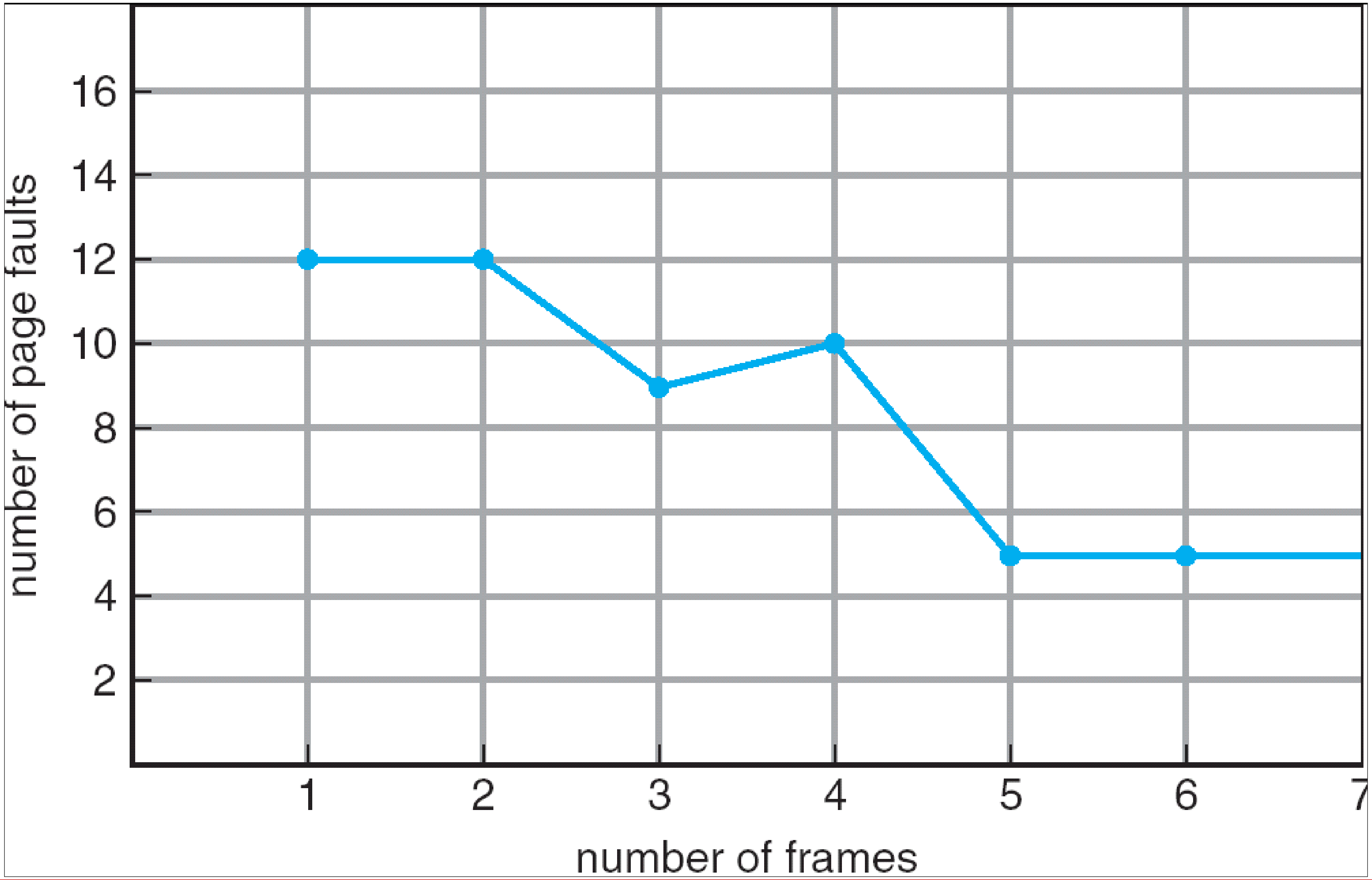


- 4 frames



Belady's Anomaly: more frames $\Rightarrow$ more page faults

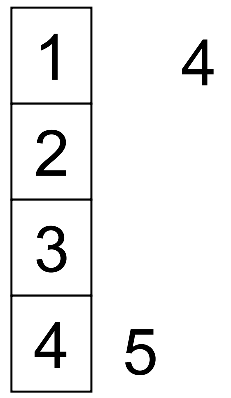
FIFO Illustrating Belady's Anomaly



Optimal Algorithm

- Replace page that will not be used for longest period of time
- 4 frames example

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



6 page faults

- How do you know this?
- Used for measuring how well your algorithm performs

Least Recently Used (LRU) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, **5**, 1, 2, **3**, **4**, **5**

1	1	1	1	**5**
2	2	2	2	2
3	**5**	5	**4**	4
4	4	**3**	3	3

- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to determine which are to change

Global vs. Local Allocation

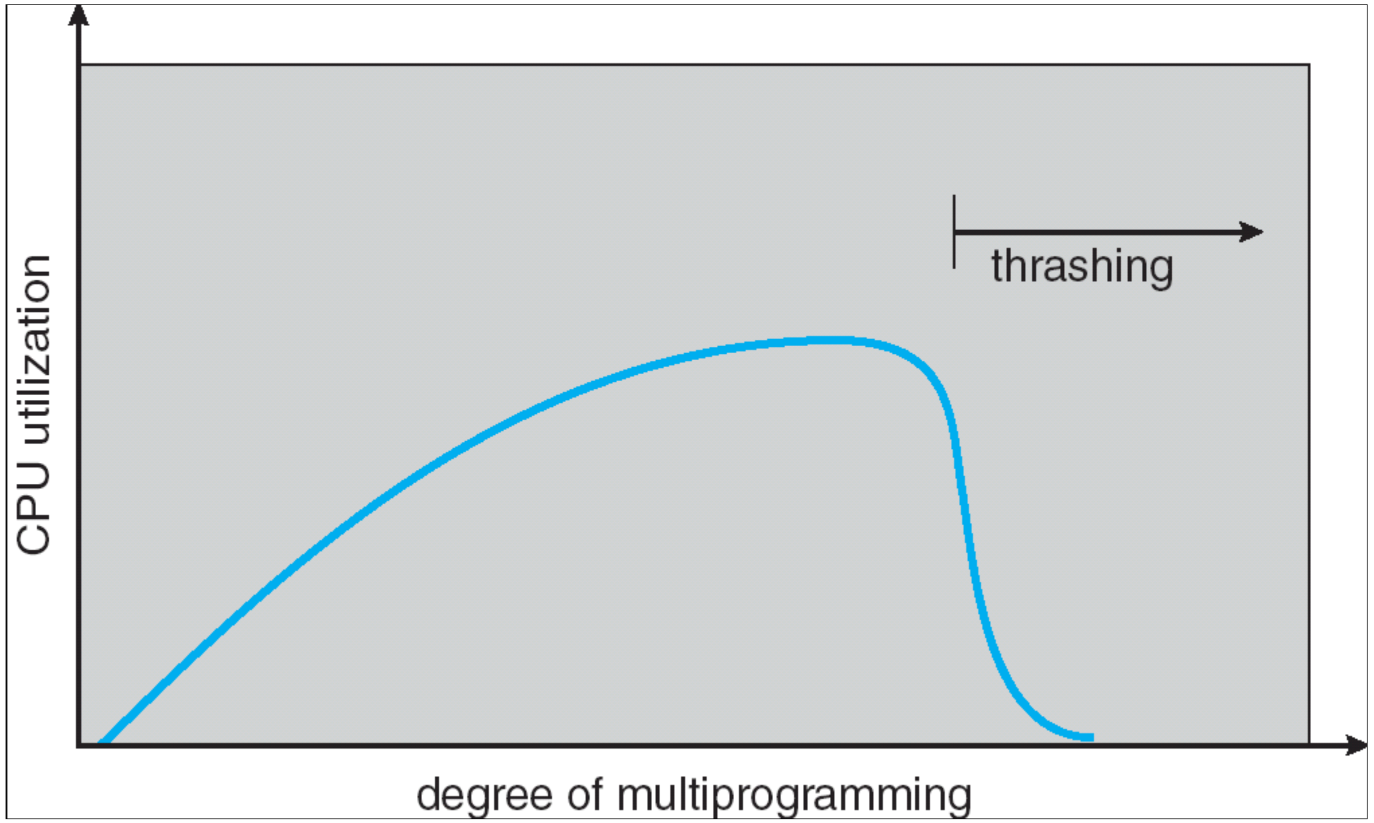
Global replacement

- process selects a replacement frame from the set of all frames
- process can take a frame from another

Local replacement

- each process selects from only its own set of allocated frames

Thrashing



Degree of multiprogramming = #processes

- If a process does not have "enough" pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system thinks that it needs to increase the degree of multiprogramming
  - another process added to the system
- **Thrashing** ≡ a process is busy swapping pages in and out

Kernel Memory Allocation

Treated differently from user memory

Often allocated from a free-memory pool

- requests memory for structures of varying sizes
- Some kernel memory needs to be contiguous
  - e.g., for device I/O

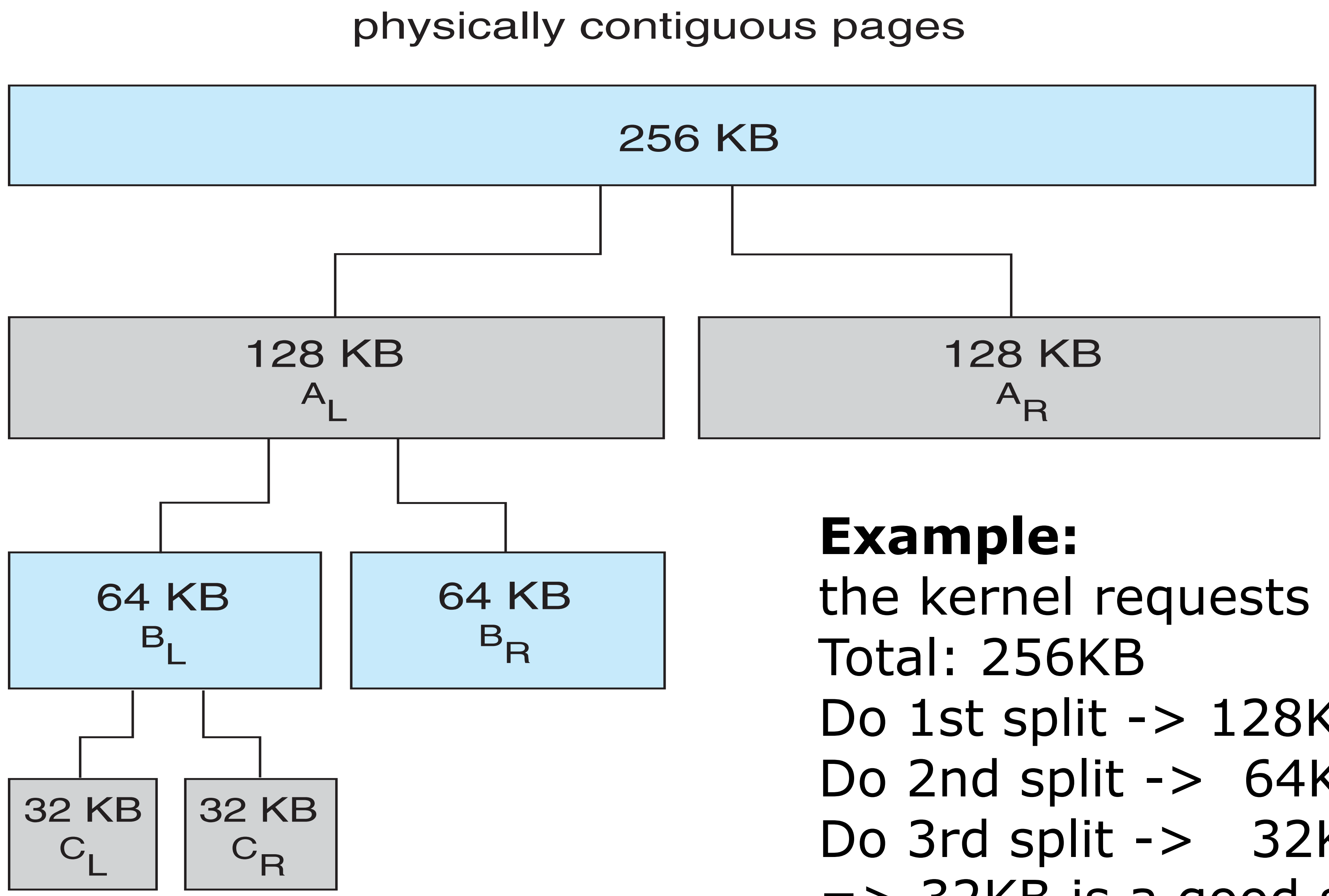
Two techniques

- Buddy
- Slab

Buddy System

- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using **power-of-2 allocator**
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2
  - When smaller allocation than the one that is available is needed, the current chunk is split into two buddies of next-lower power of 2
  - Continue until appropriate sized chunk available
- Advantage – quickly **coalesce** unused chunks into larger chunk
- Disadvantage - fragmentation

Buddy System Allocator

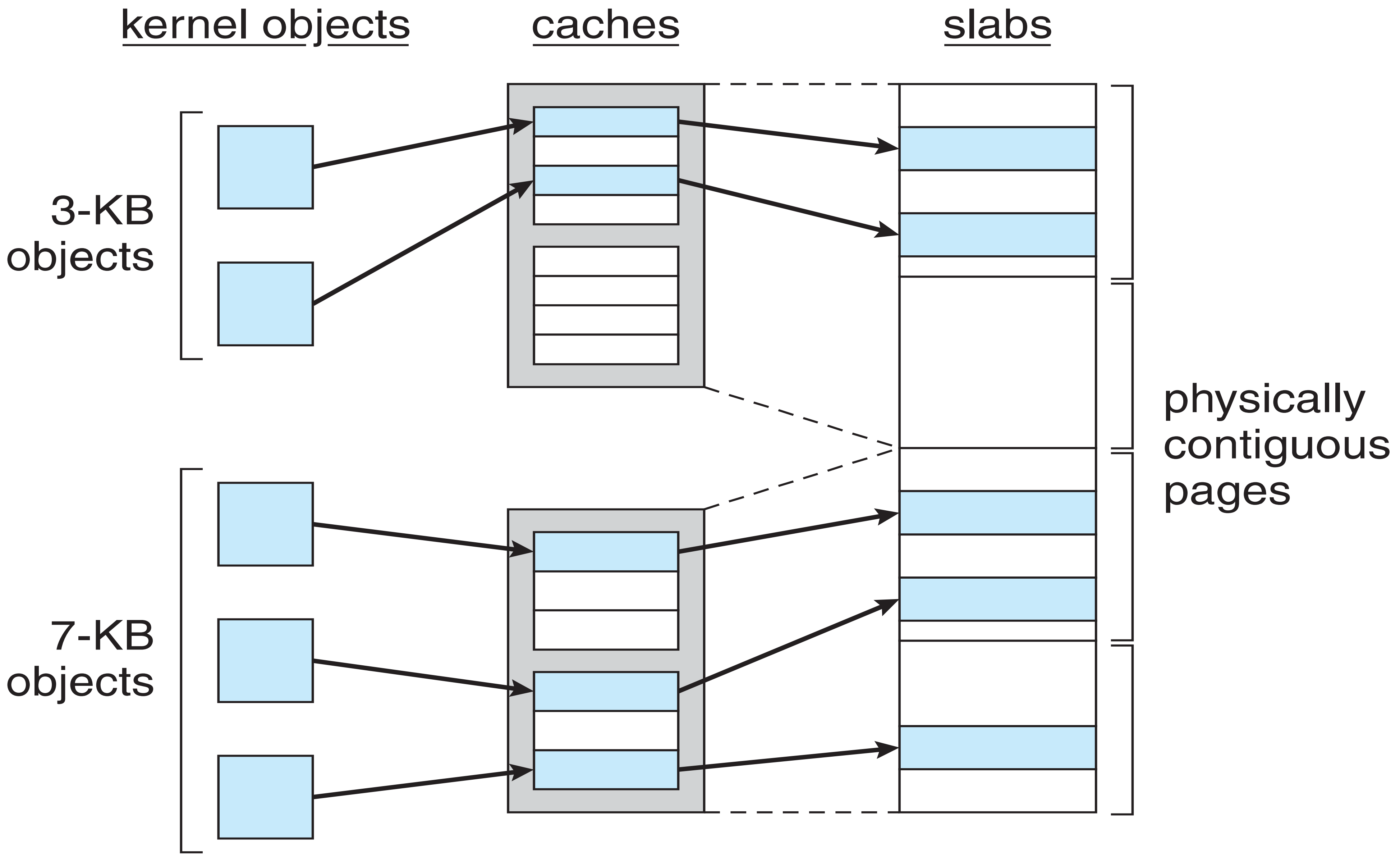


Example:
the kernel requests 21KB
Total: 256KB
Do 1st split -> 128KB
Do 2nd split -> 64KB
Do 3rd split -> 32KB
=> 32KB is a good size

Slab Allocator

- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure
  - Each cache filled with **objects** – instantiations of the data structure
- If slab is full of used objects, next object allocated from empty slab
  - If no empty slab, a new slab is allocated
- Benefits include
  - no fragmentation
  - fast memory request satisfaction

Slab Allocation


kernel objects
caches
slabs
3-KB objects
7-KB objects
physically contiguous pages