UNIT 4
MEMORY MANAGEMENT

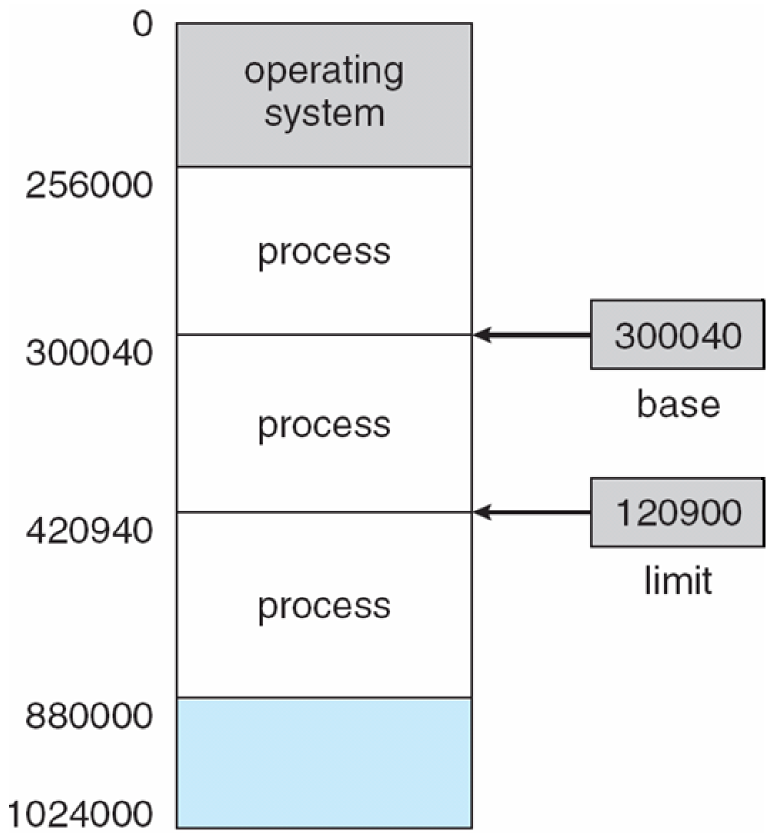
CONTENTS

4.1 BACKGROUND

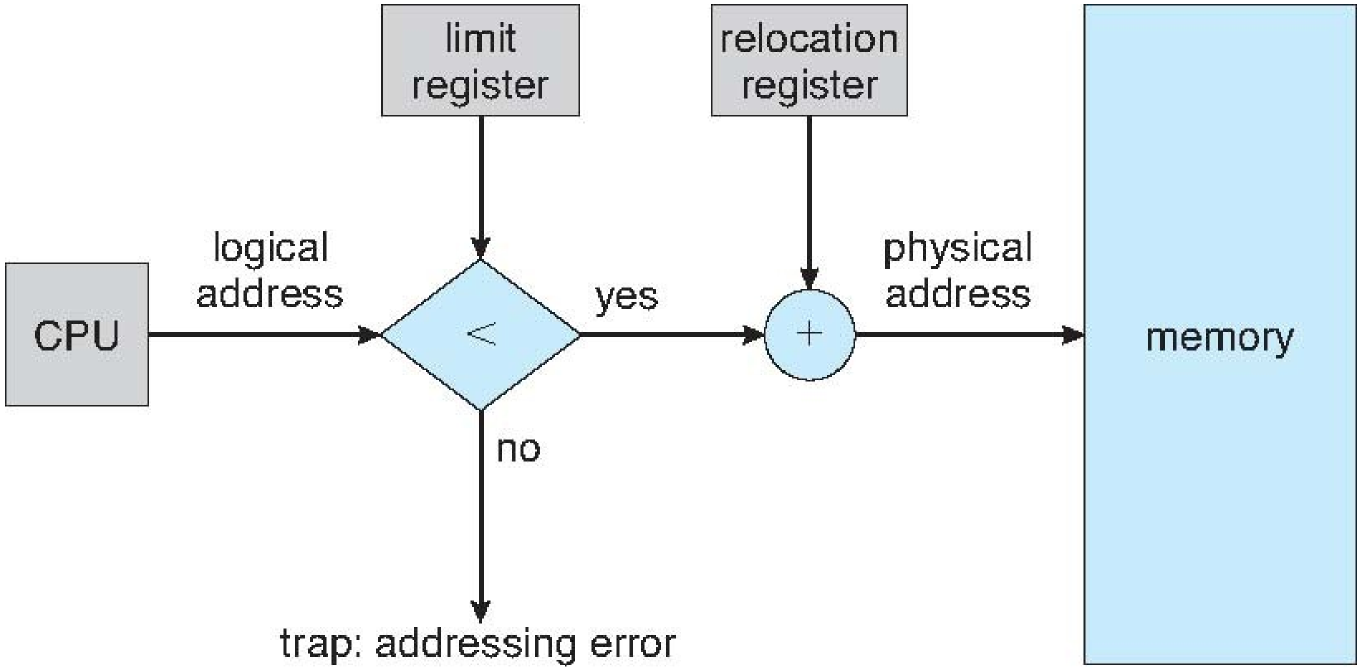
- Program must be brought (from disk) into memory and placed within memory for it to be run
- Main memory and registers are only storage CPU can access directly
- Register access in one CPU clock (or less)
- Main memory can take many cycles

BASE AND LIMIT REGISTERS

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



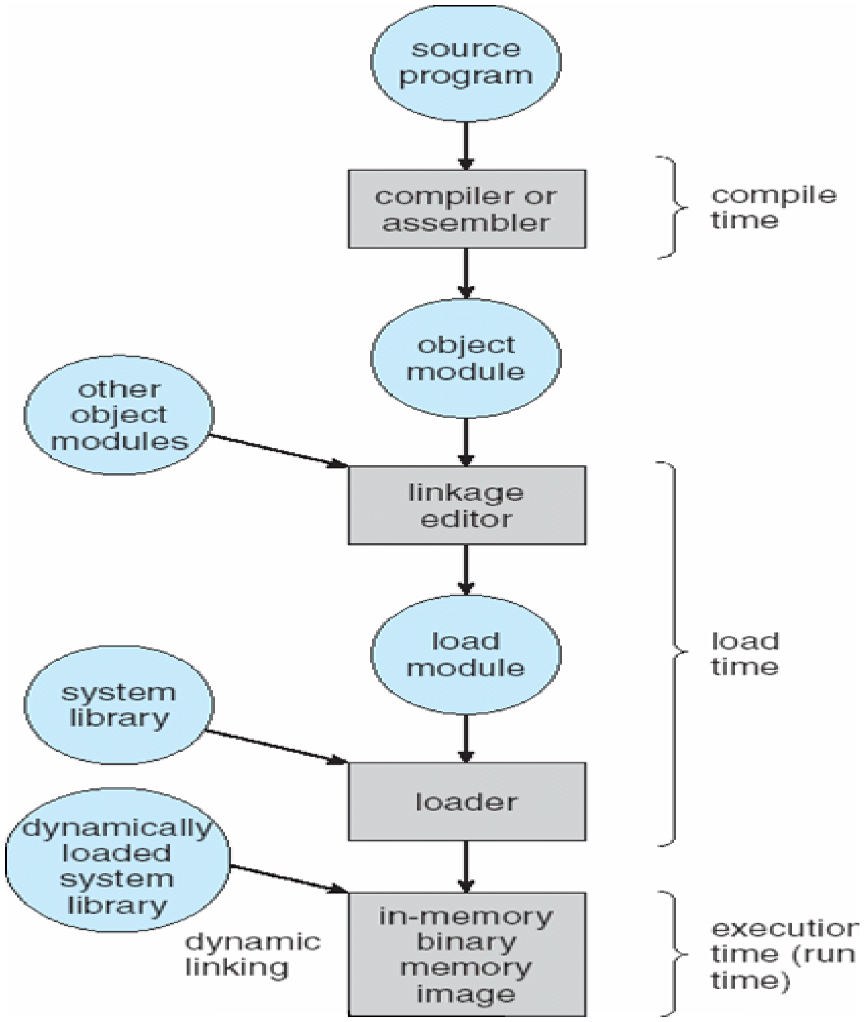
HARDWARE SUPPORT FOR RELOCATION AND LIMIT REGISTERS



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



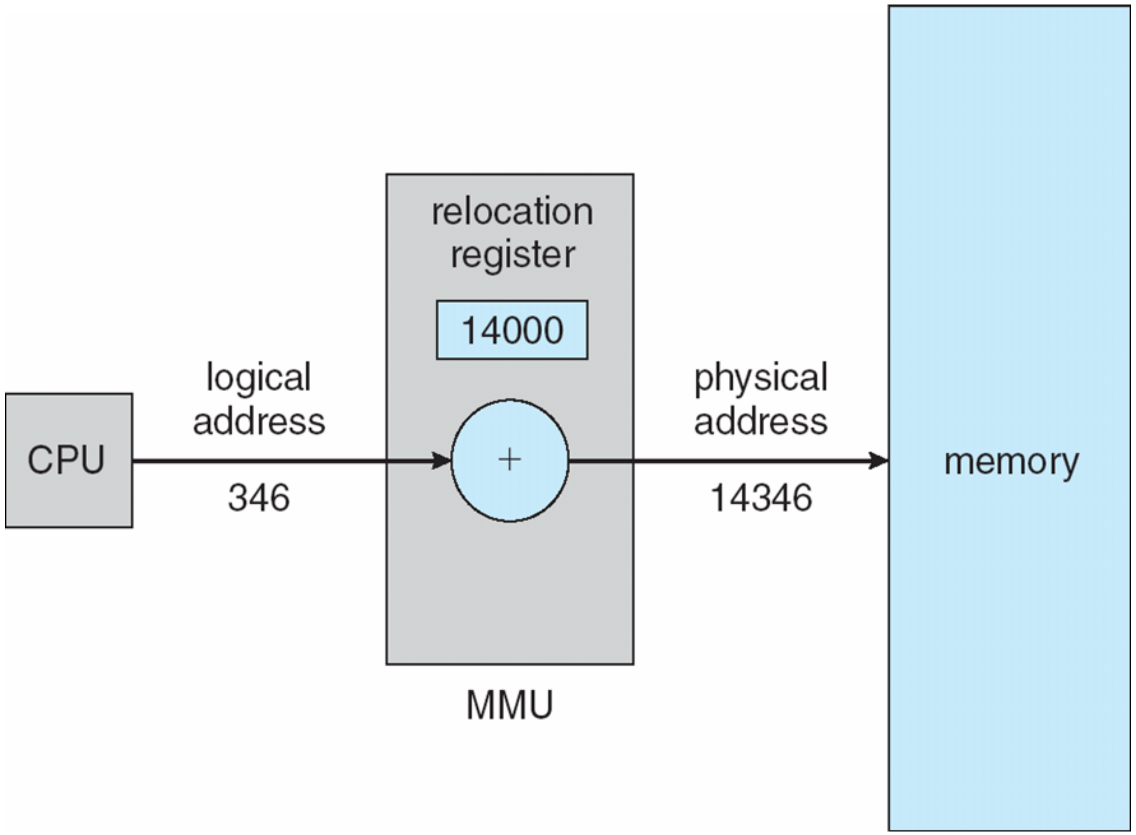
LOGICAL VS. PHYSICAL ADDRESS SPACE

- 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

MEMORY-MANAGEMENT UNIT (MMU)

- Hardware device that maps virtual to physical address
- 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
- The user program deals with *logical* addresses; it never sees the *real* physical addresses

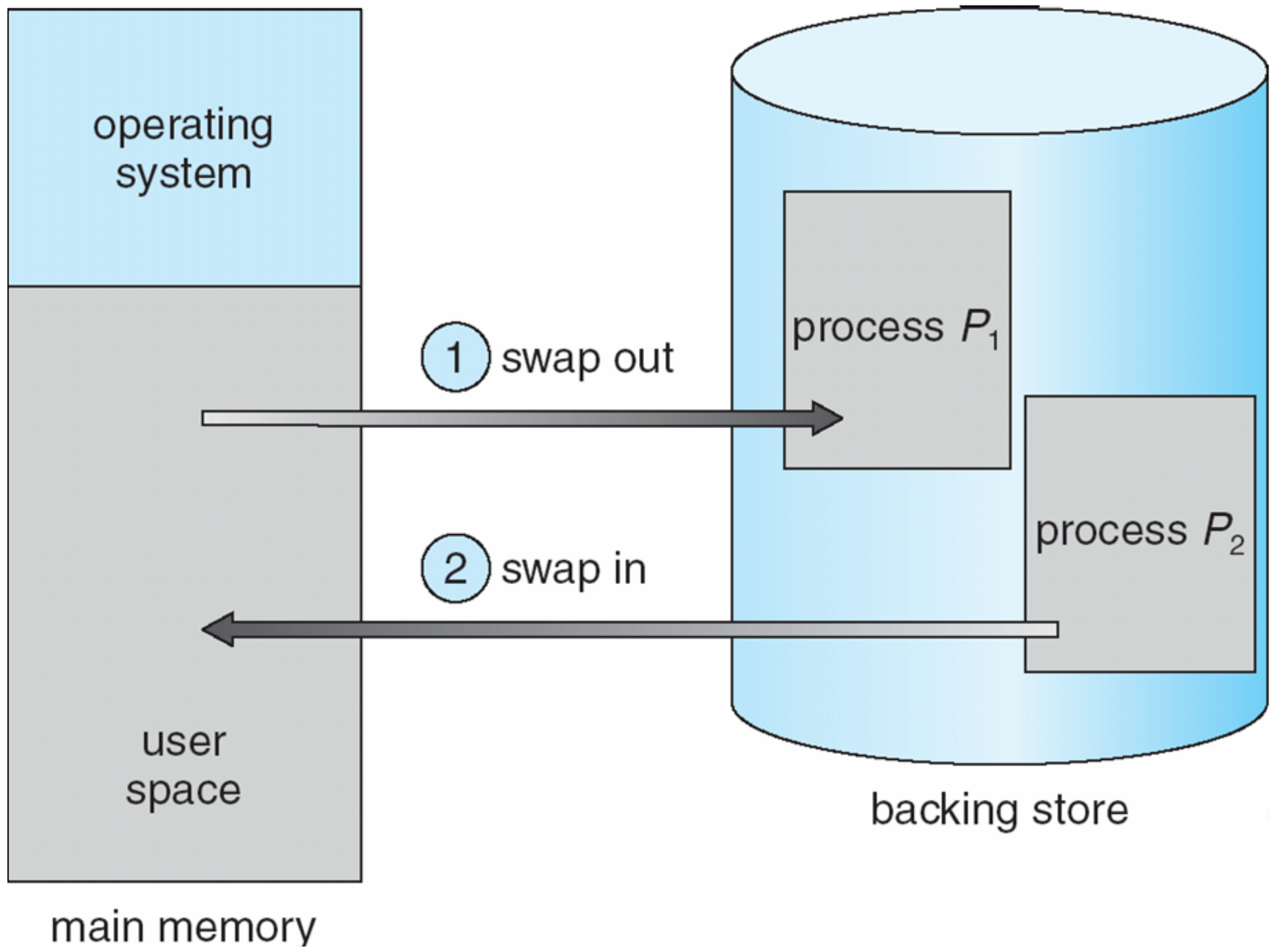
DYNAMIC RELOCATION USING A RELOCATION REGISTER



4.2 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
- 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

SCHEMATIC VIEW OF SWAPPING

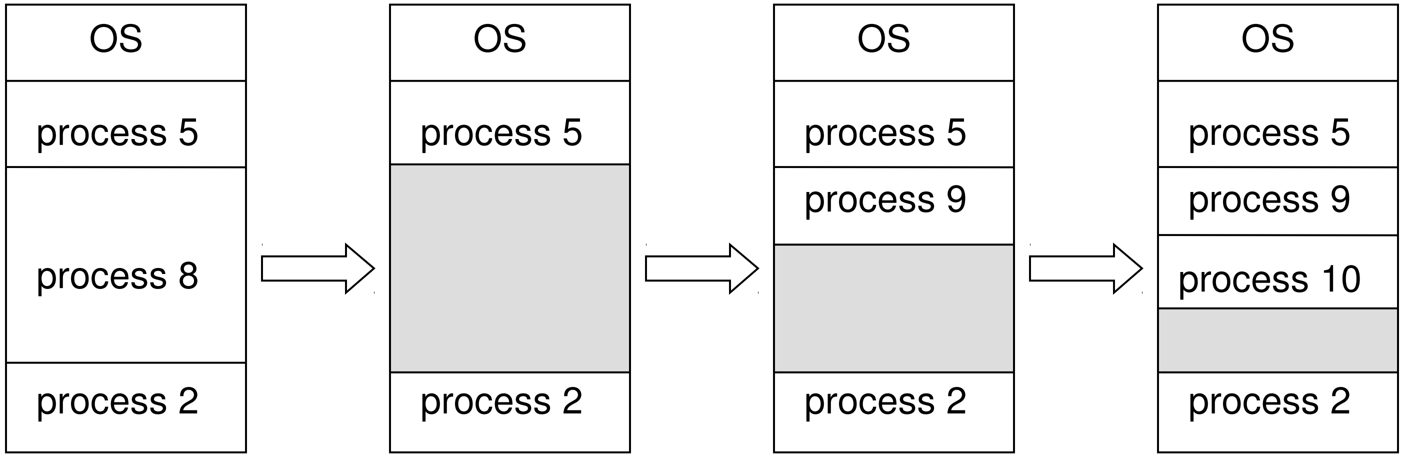


4.3 CONTIGUOUS ALLOCATION

- 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*

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)



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

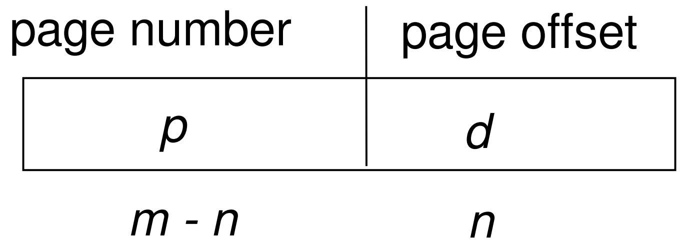
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- Reduce external fragmentation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

4.4 PAGING

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Divide physical memory into fixed-sized blocks called **frames** (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 may occur in the last frame

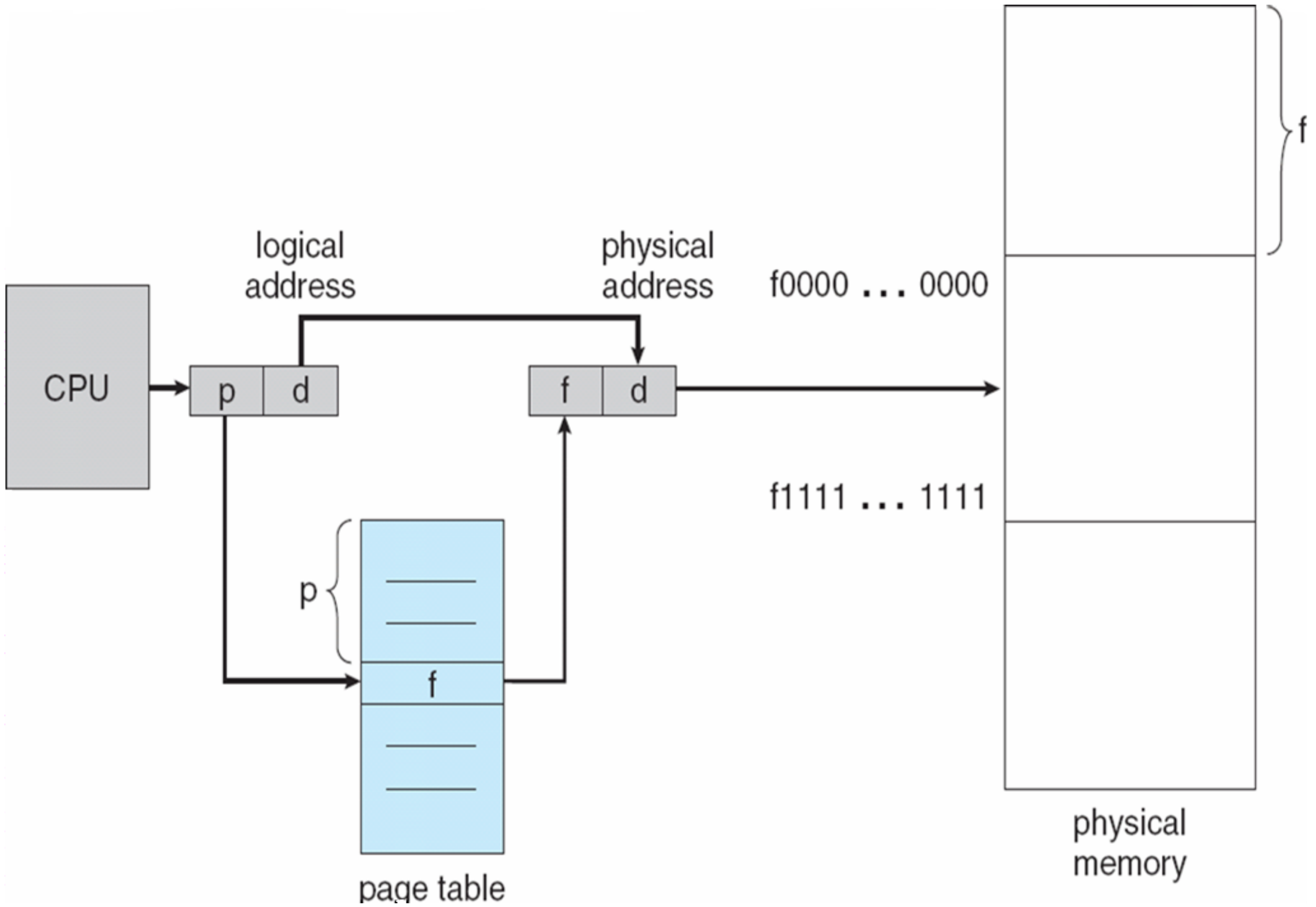
ADDRESS TRANSLATION SCHEME

- 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

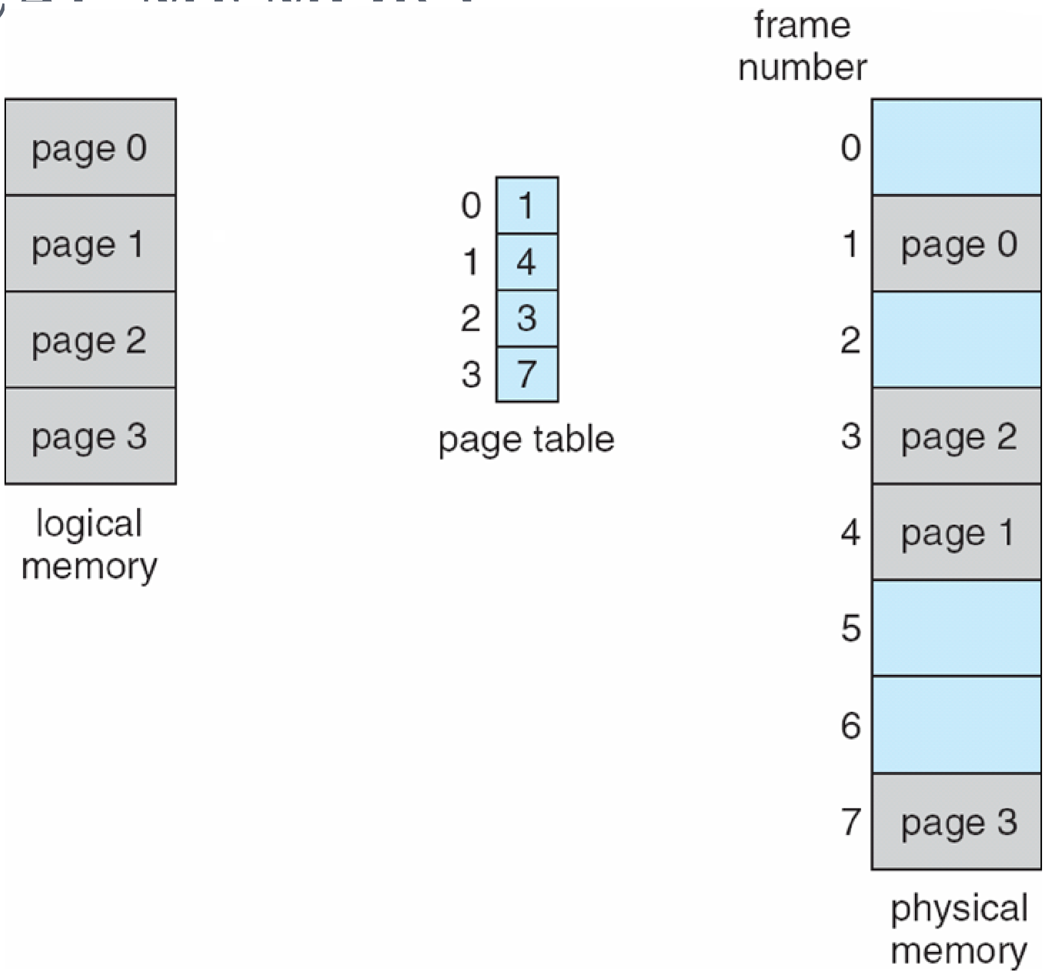


  - For given logical address space 2^m *and page size* 2^n

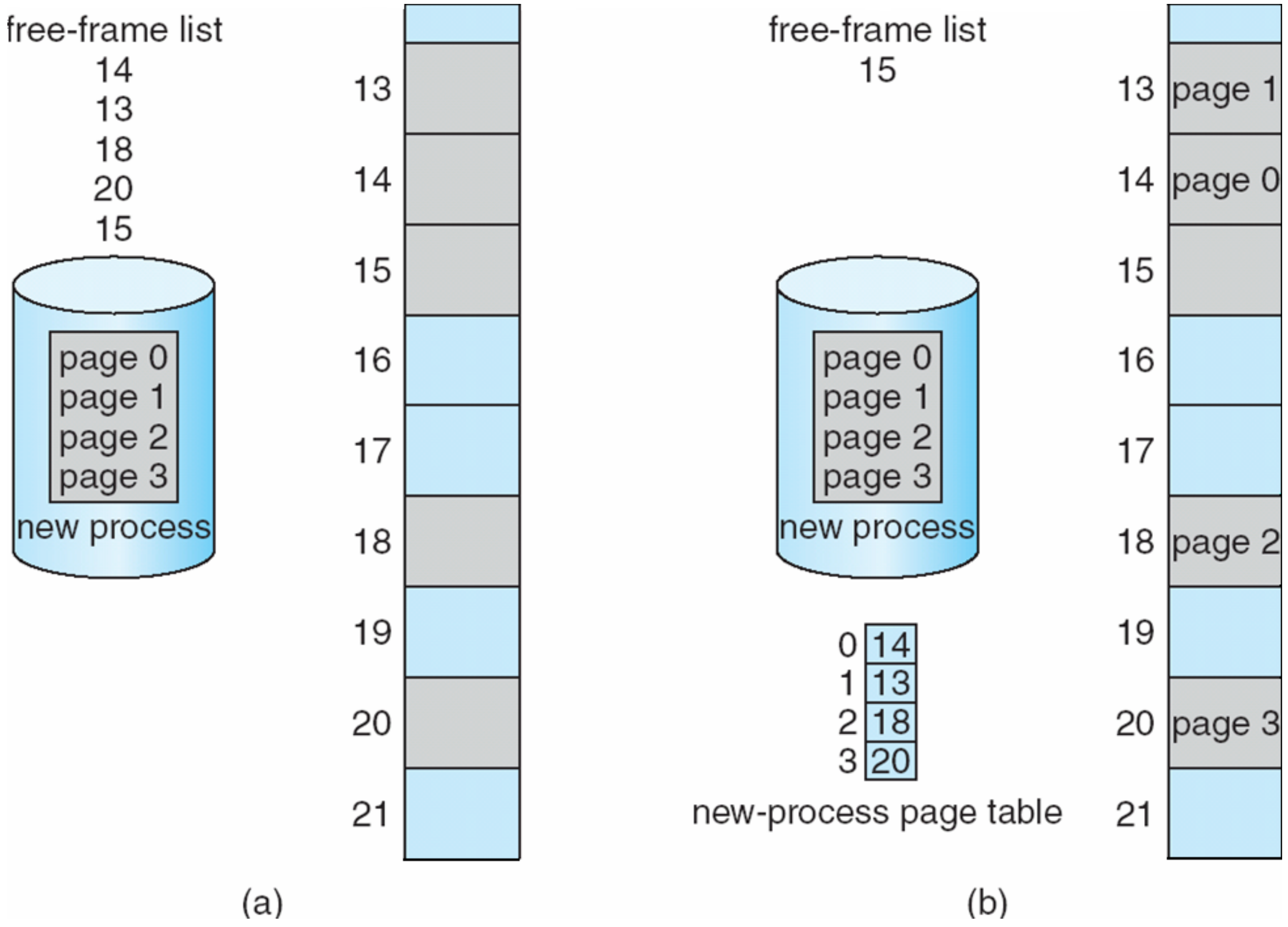
PAGING HARDWARE



PAGING MODEL OF LOGICAL AND PHYSICAL MEMORY



FREE FRAMES



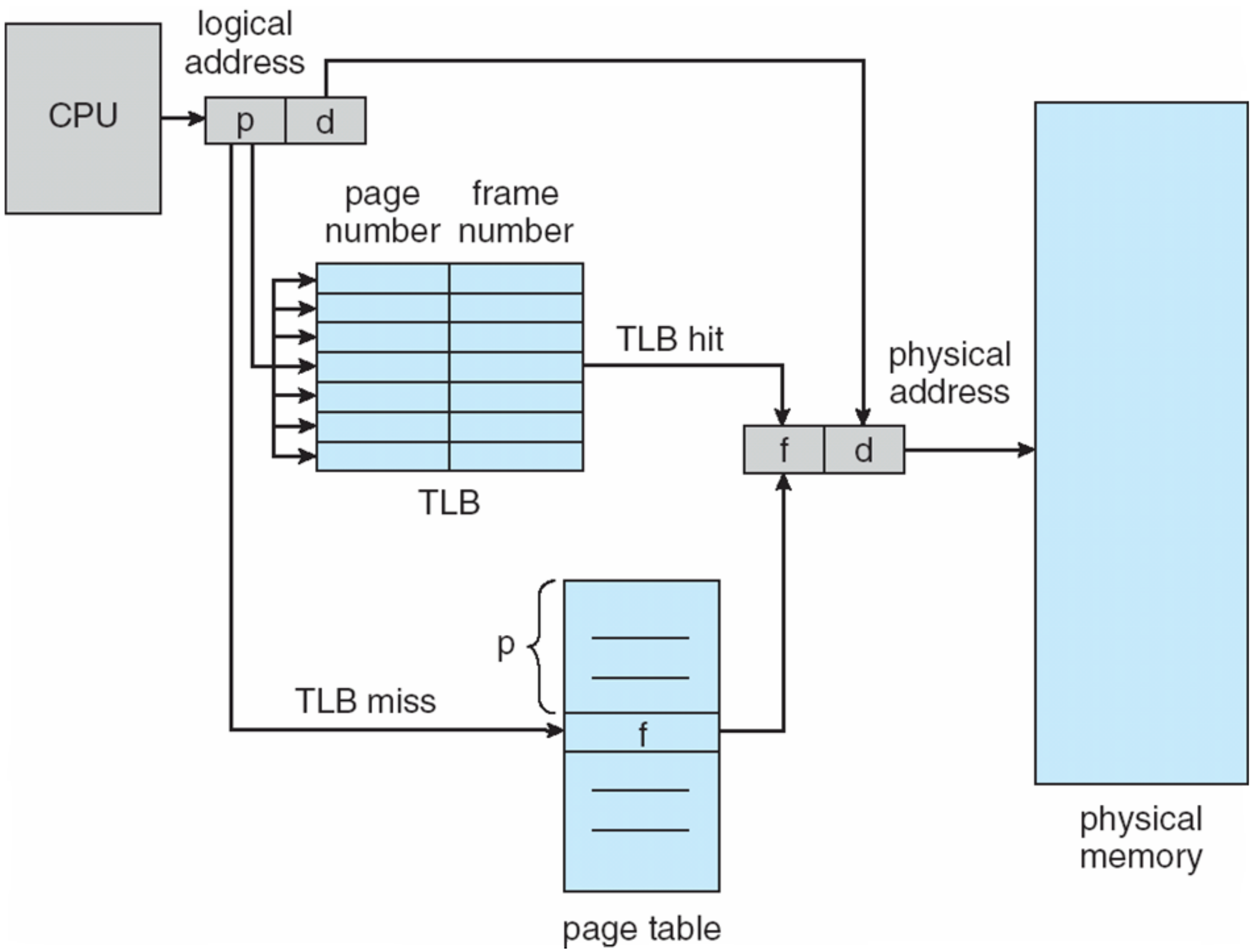
Before allocation

After allocation

IMPLEMENTATION OF PAGE TABLE

- 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.
- 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)**
- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process

PAGING HARDWARE WITH TLB



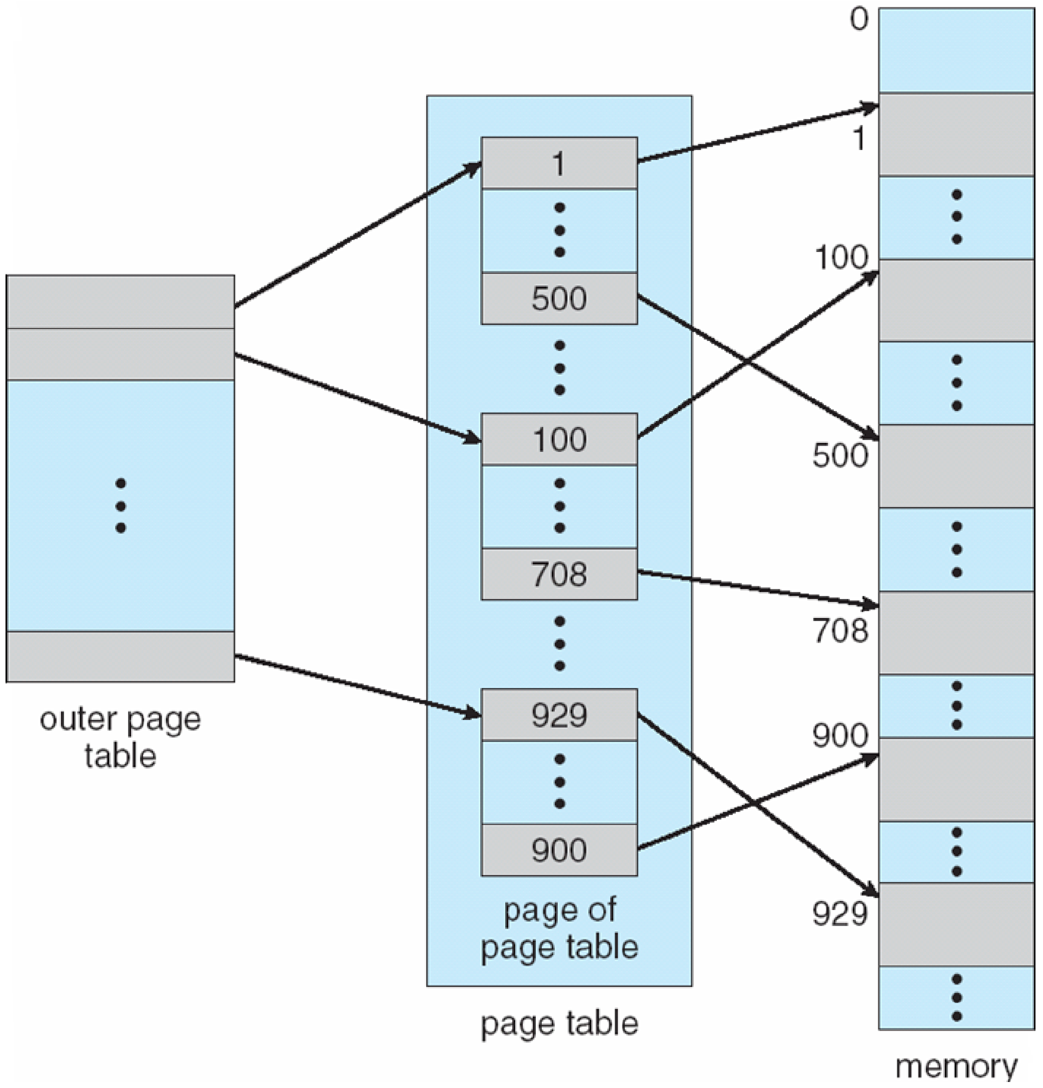
4.5 STRUCTURE OF THE PAGE TABLE

- Hierarchical Paging
- Hashed Page Table
- Inverted Page Table

HIERARCHICAL PAGE TABLES

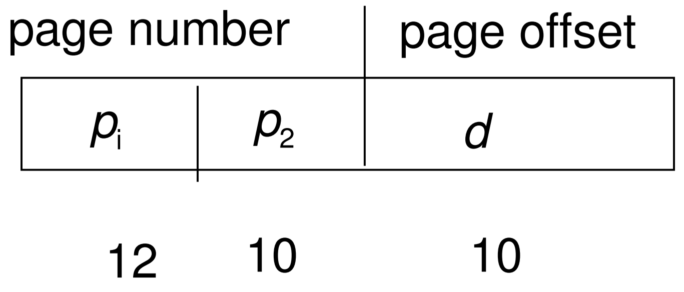
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table

TWO-LEVEL PAGE-TABLE SCHEME



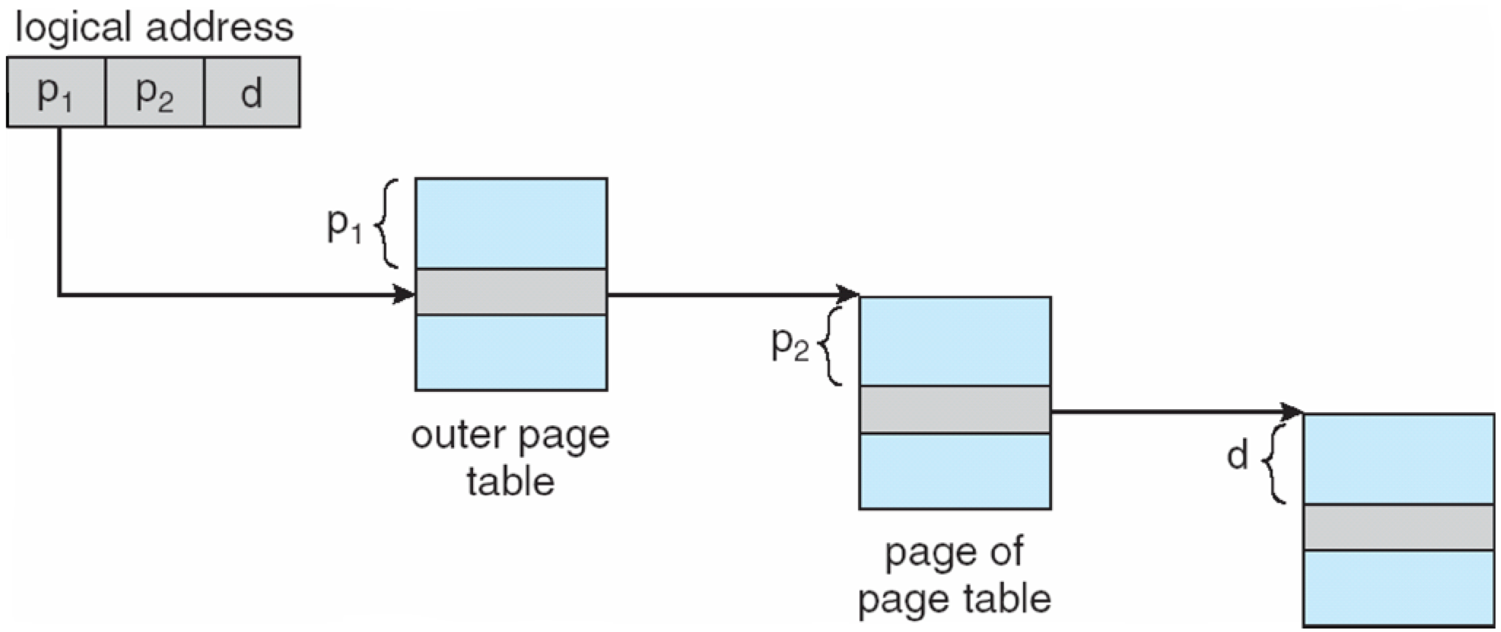
TWO-LEVEL PAGING EXAMPLE

- 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

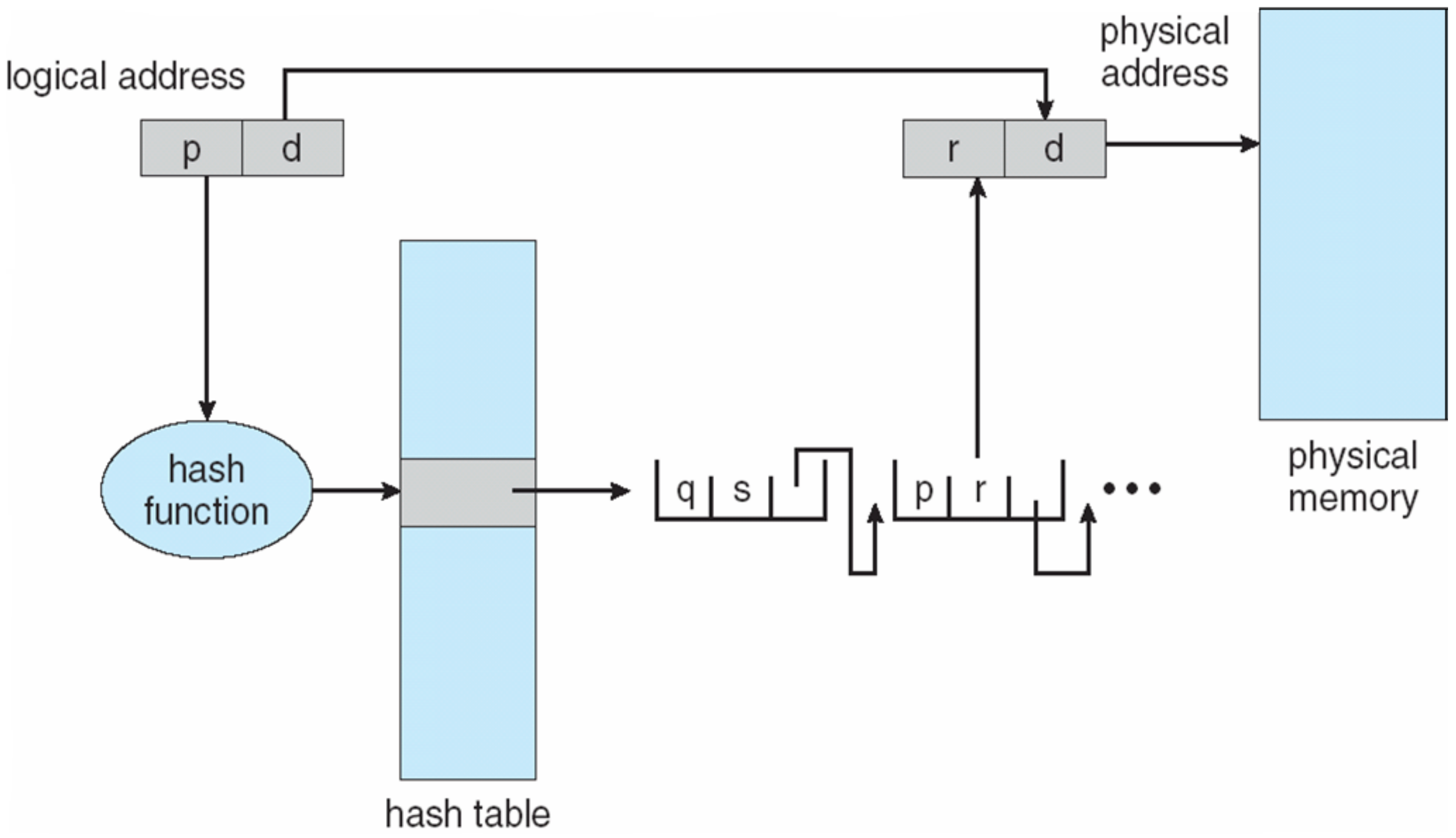
ADDRESS-TRANSLATION SCHEME



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

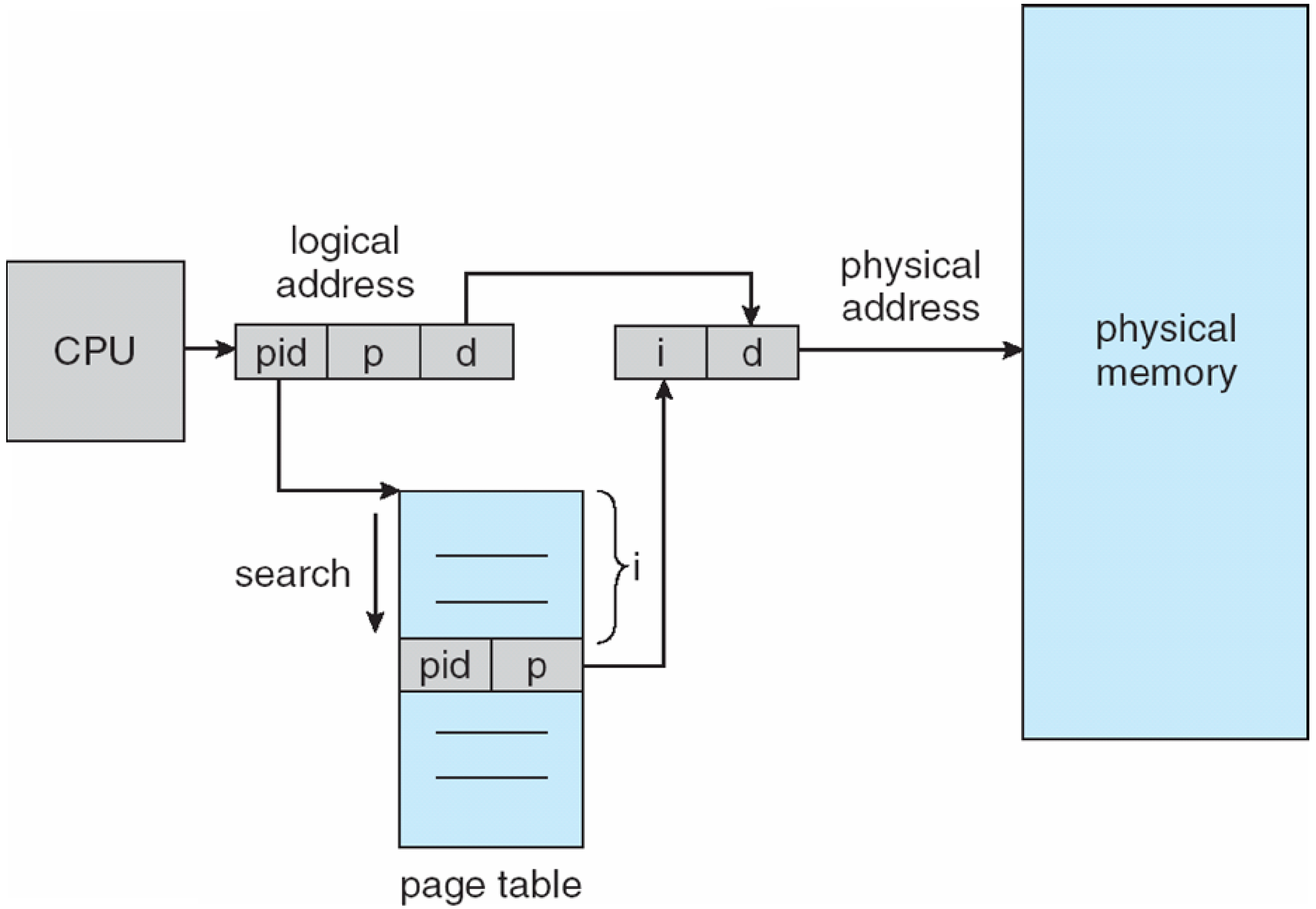
HASHED PAGE TABLE



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

INVERTED PAGE TABLE ARCHITECTURE



4.6 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

  function

  method

  object

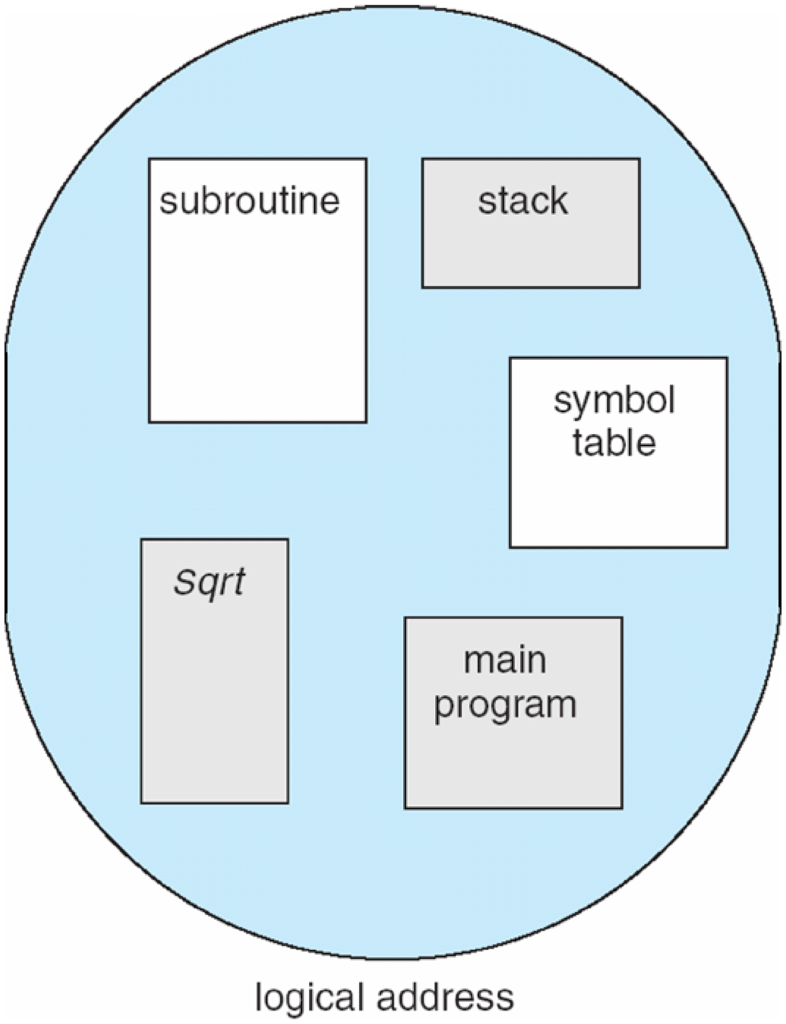
  local variables, global variables

  common block

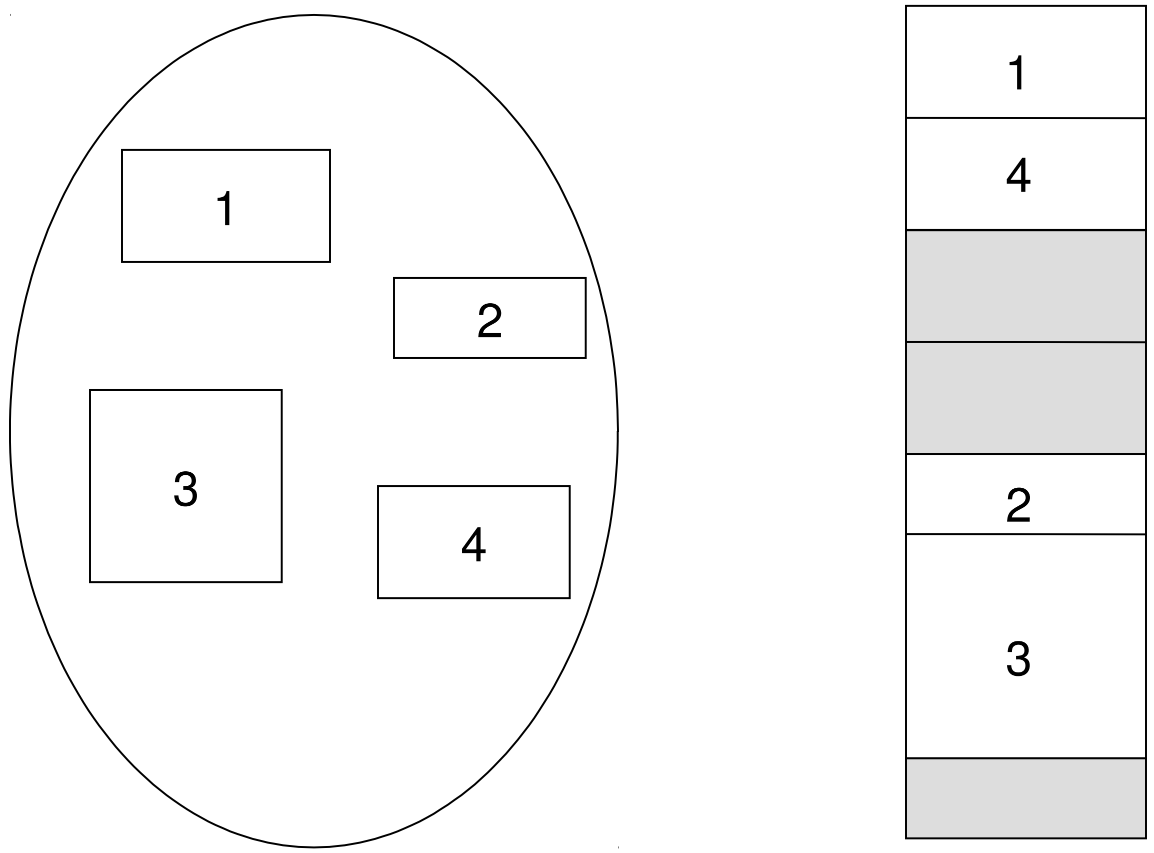
  stack

  arrays

USER'S VIEW OF A PROGRAM



LOGICAL VIEW OF SEGMENTATION



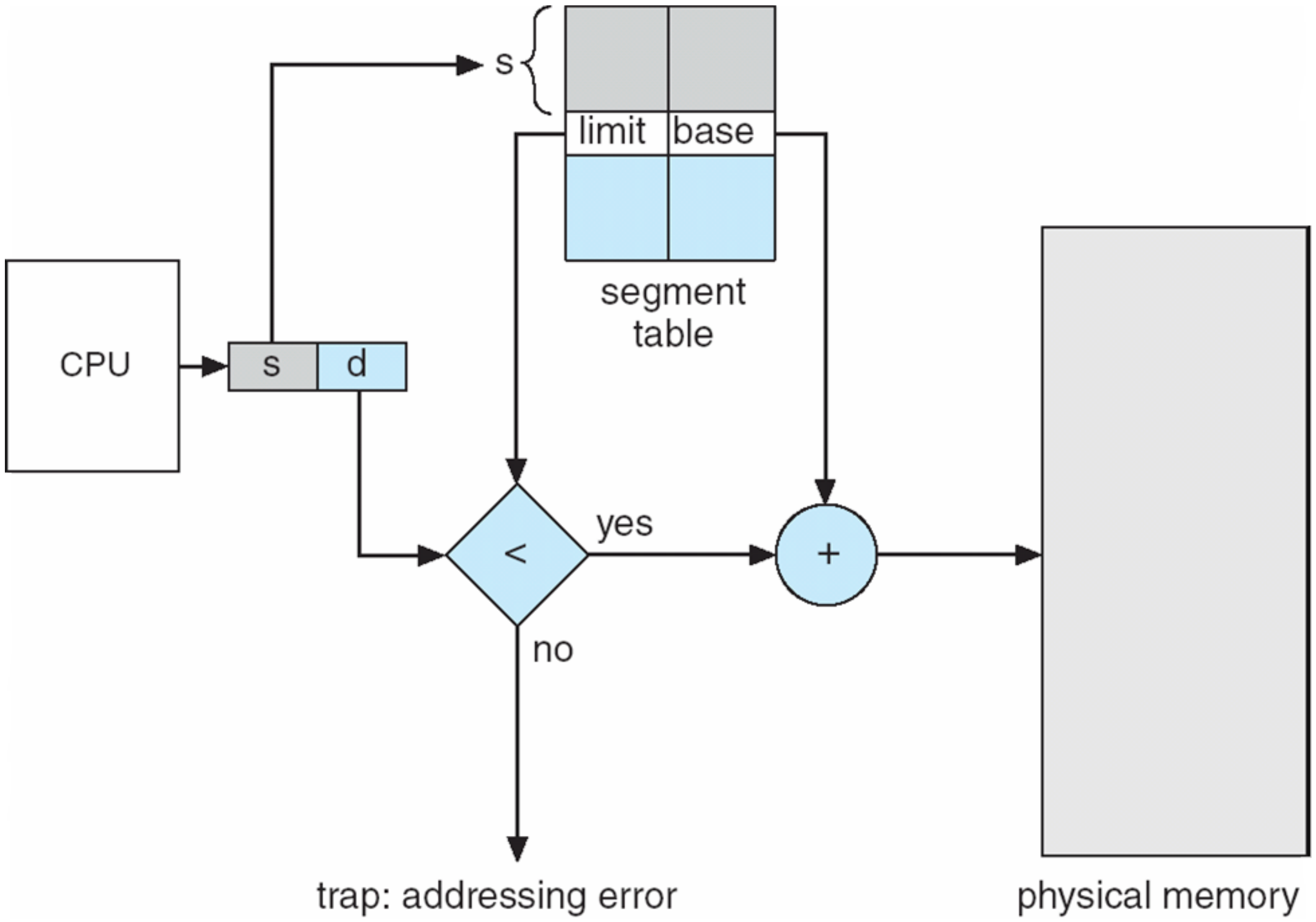
user space

physical memory space

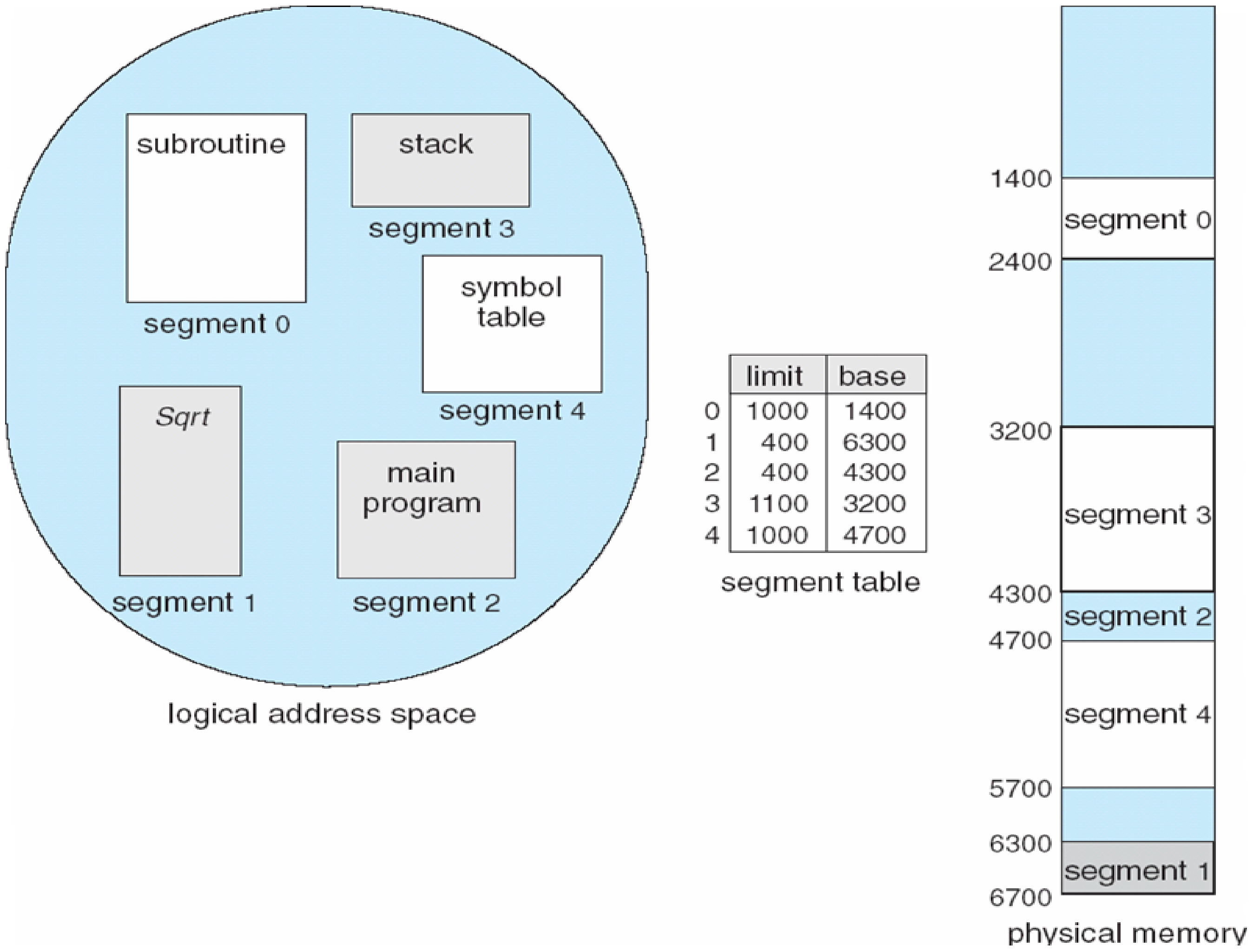
SEGMENTATION ARCHITECTURE

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

SEGMENTATION HARDWARE



EXAMPLE OF SEGMENTATION



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

VIRTUAL MEMORY

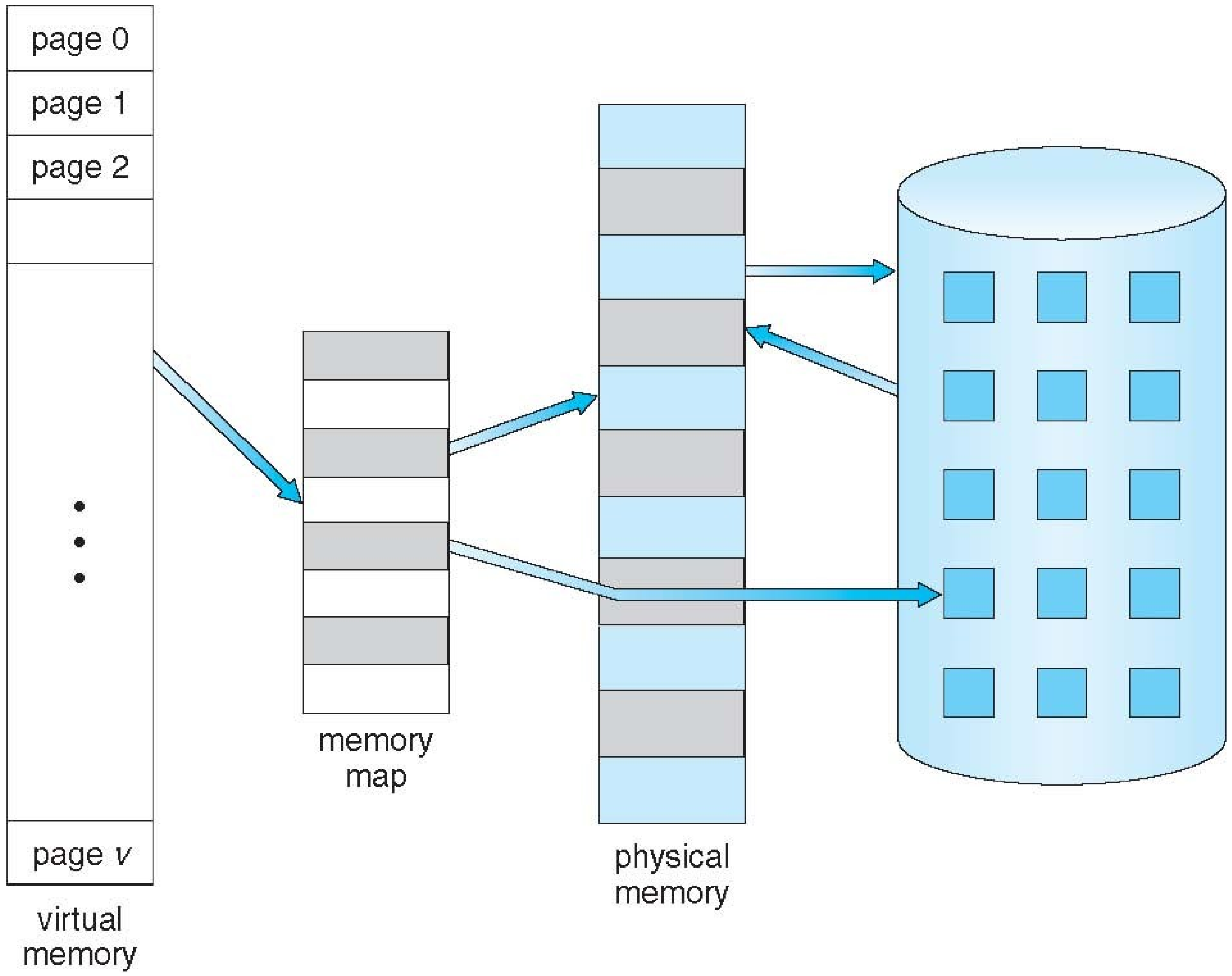
4.7 BACKGROUND

- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
- *Virtual memory* is imaginary memory: it gives you the illusion of a memory arrangement that's not physically there.

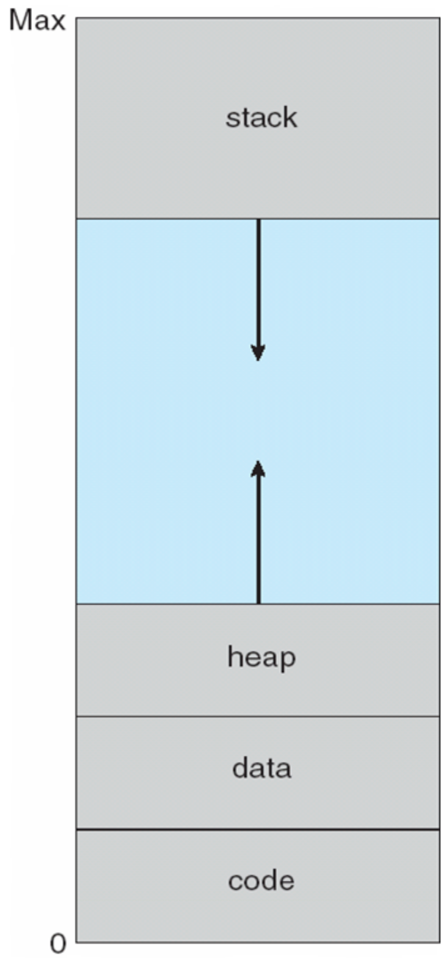
- **Virtual memory** – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes

- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

VIRTUAL MEMORY THAT IS LARGER THAN PHYSICAL MEMORY



VIRTUAL-ADDRESS SPACE



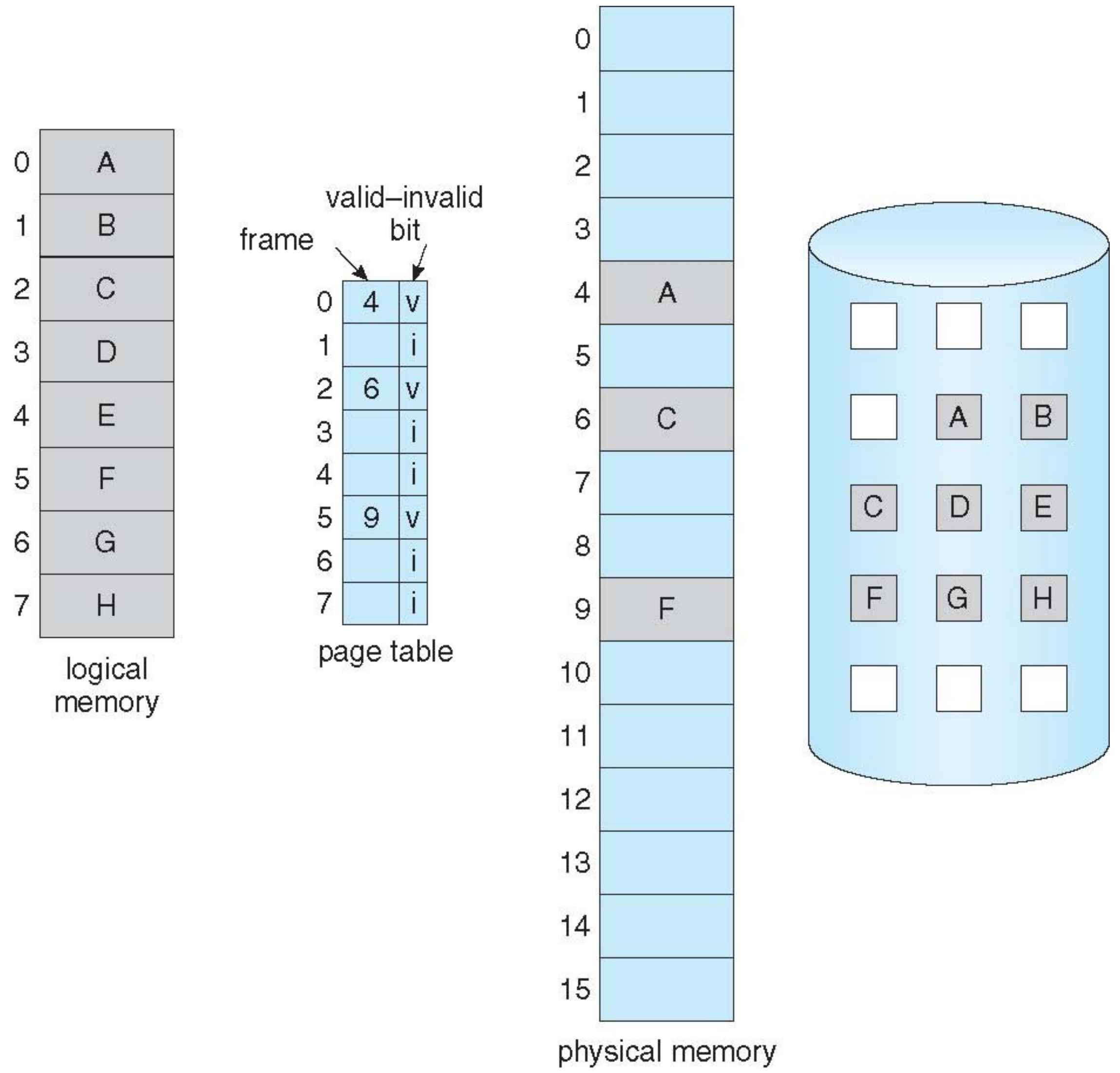
DEMAND PAGING

- Bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**

VALID-INVALID BIT

- With each page table entry a valid–invalid bit is associated
  (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- During address translation, if valid–invalid bit in page table entry
  is **i**⇒ page fault

PAGE TABLE WHEN SOME PAGES ARE NOT IN MAIN MEMORY



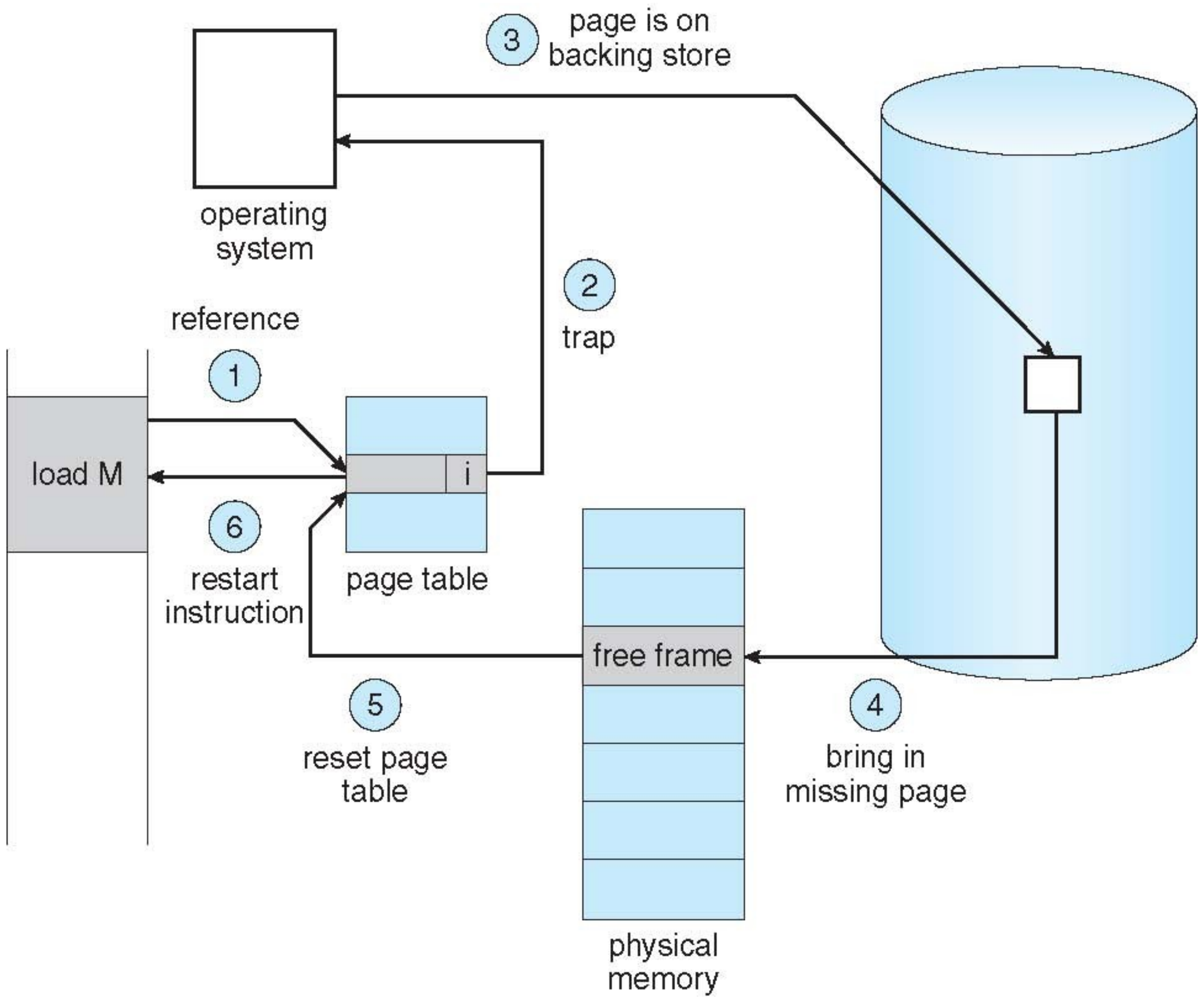
STEPS IN HANDLING PAGE FAULT

- If there is a reference to a page, first reference to that page will trap to operating system:

  page fault

1. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory
2. Get empty frame
3. Swap page into frame via scheduled disk operation
4. Reset tables to indicate page now in memory
   Set validation bit = **v**
5. Restart the instruction that caused the page fault

STEPS IN HANDLING A PAGE FAULT



ASPECTS OF DEMAND PAGING

- Extreme case – start process with *no* pages in memory - **Pure demand paging**
- Actually, a given instruction could access multiple pages -> multiple page faults -> Paging to done based on **locality of reference**
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart

WHAT HAPPENS IF THERE IS NO FREE FRAME?

- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?

- Page replacement – find some page in memory, but not really in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result in minimum number of page faults

- Same page may be brought into memory several times

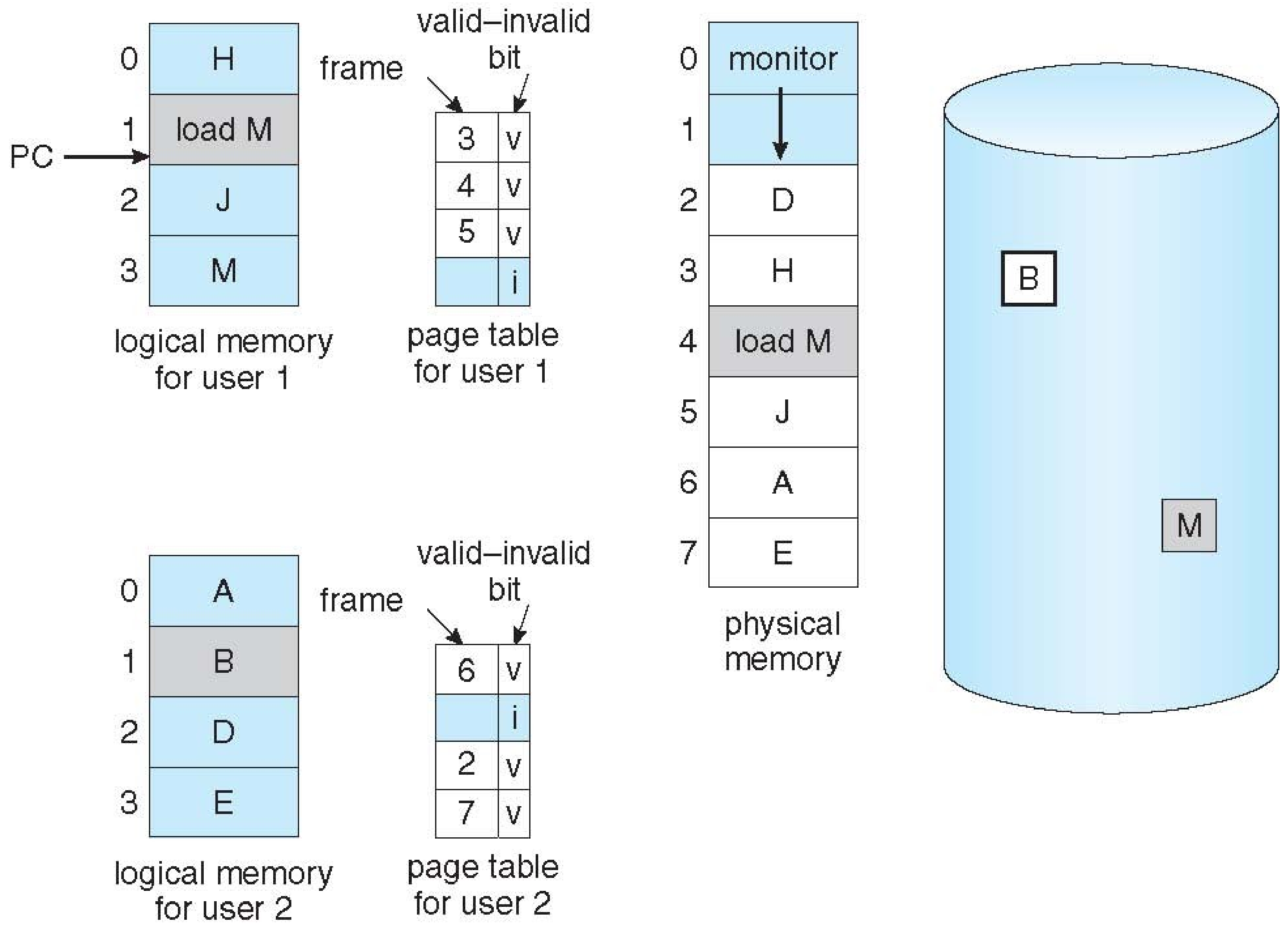
PAGE REPLACEMENT

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement

- Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk

- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

NEED FOR PAGE REPLACEMENT



BASIC PAGE REPLACEMENT

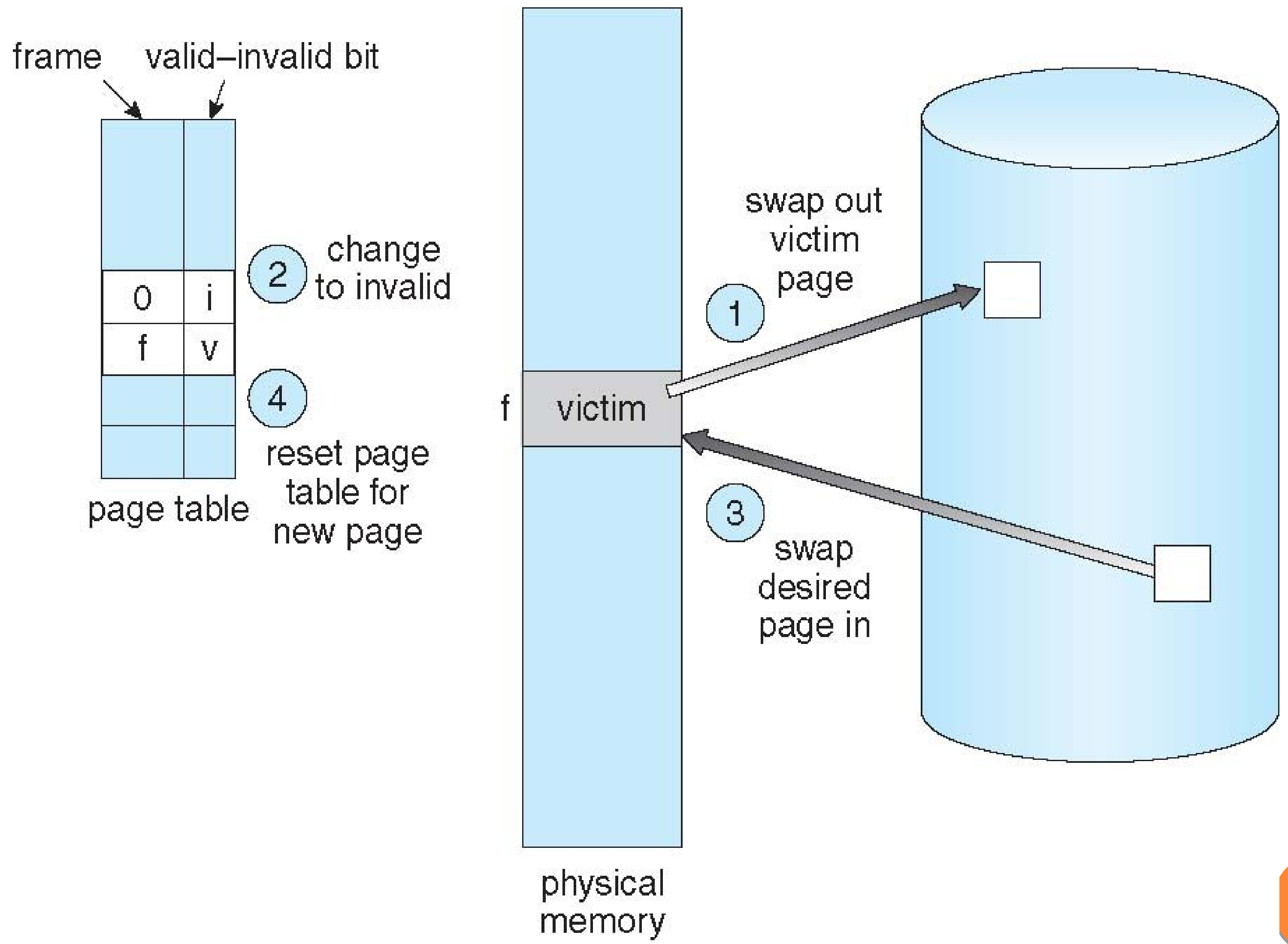
1. Find the location of the desired page on disk

2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim frame**
     - Write victim frame to disk if dirty

3. Bring the desired page into the (newly) free frame; update the page and frame tables

4. Continue the process by restarting the instruction that caused the trap

PAGE REPLACEMENT



PAGE AND FRAME REPLACEMENT ALGORITHMS

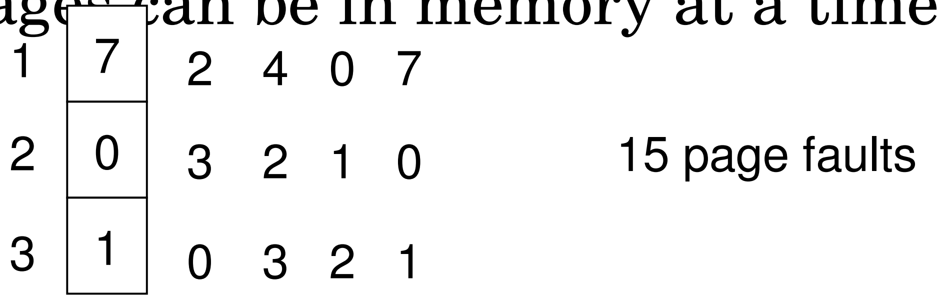
- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace
- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access

- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
- In all our examples, the reference string is

7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

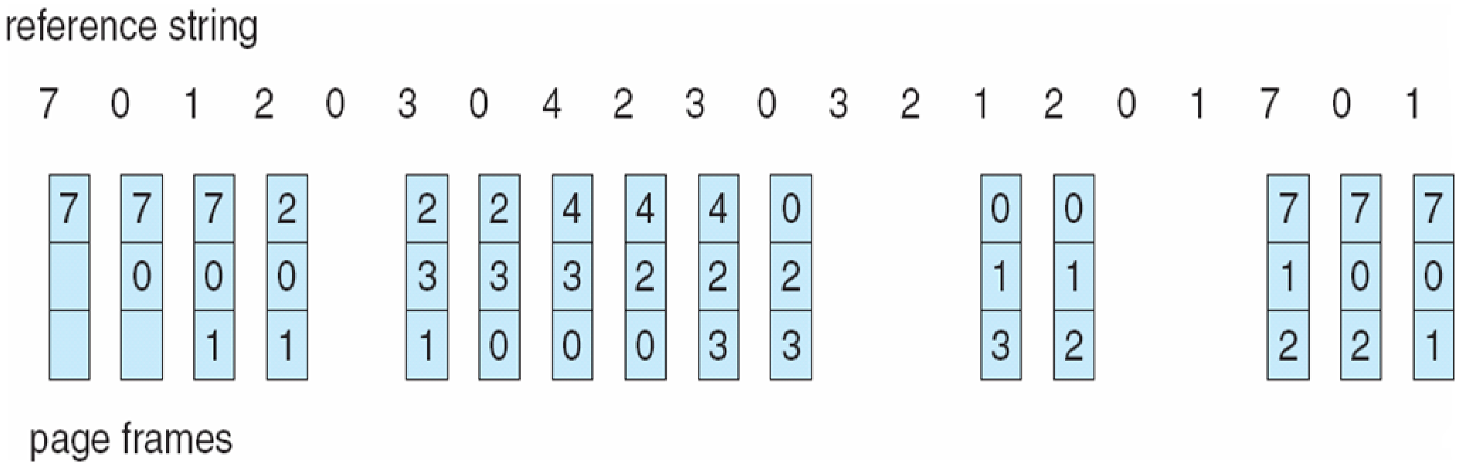
FIRST-IN-FIRST-OUT (FIFO) ALGORITHM

- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
  - Adding more frames can cause more page faults!
    - **Belady's Anomaly**
- How to track ages of pages?
  - Just use a FIFO queue

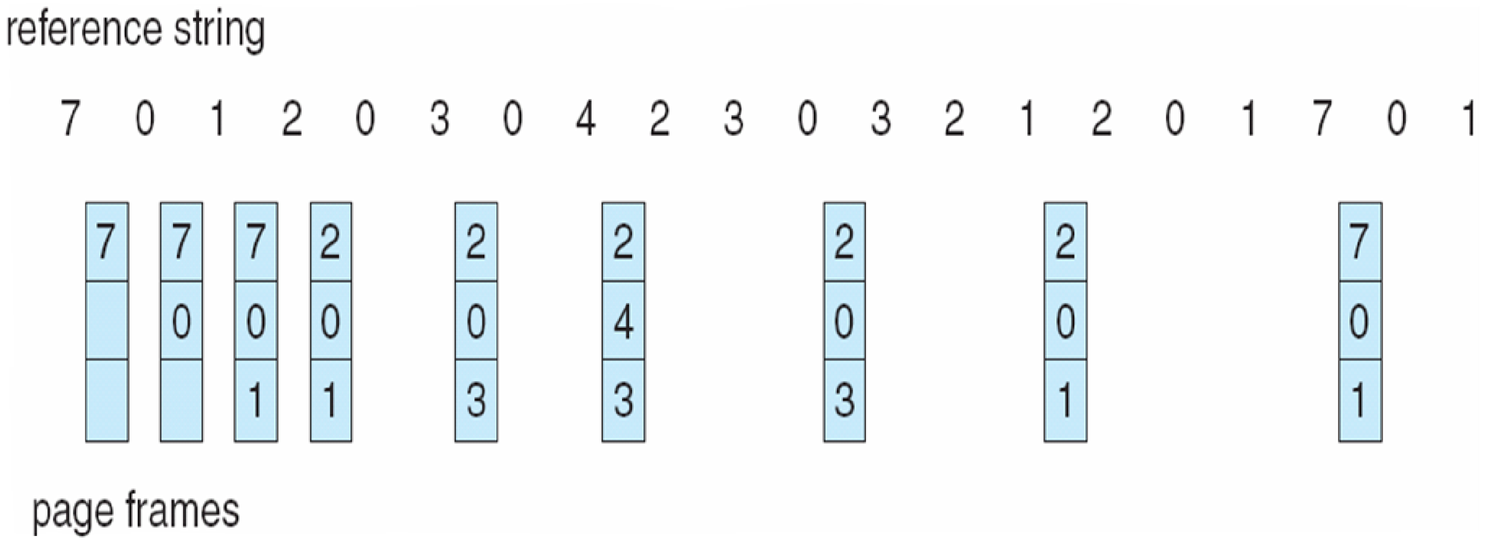
FIFO PAGE REPLACEMENT



OPTIMAL ALGORITHM

- Replace page that will not be used for longest period of time
  - 9 is optimal for the example on the next slide
- How do you know this?
  - Can't read the future
- Used for measuring how well your algorithm performs

OPTIMAL PAGE REPLACEMENT



LEAST RECENTLY USED (LRU) ALGORITHM

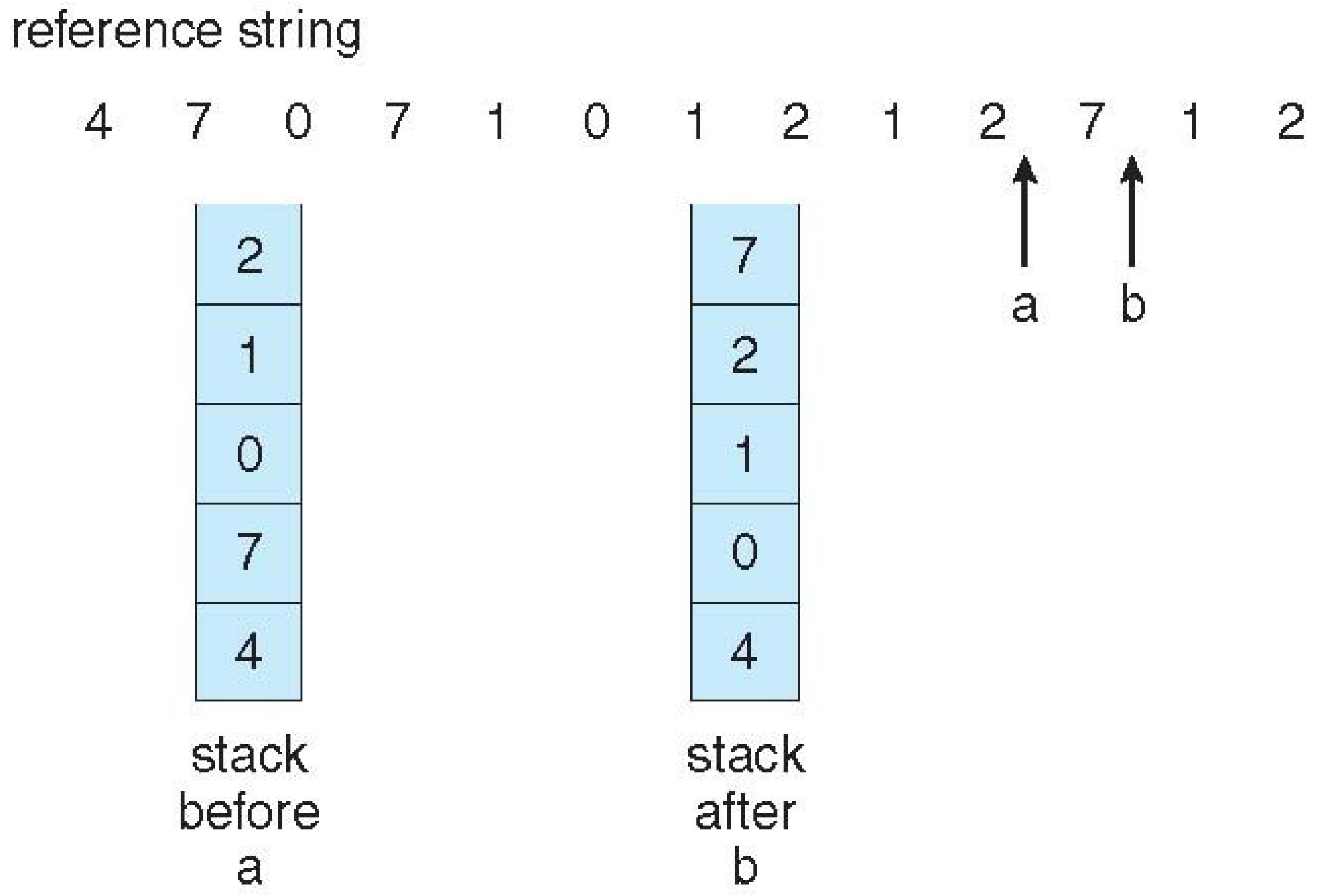
- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page

- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?

LRU ALGORITHM (CONT.)

- 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 find smallest value
    - Search through table needed
- Stack implementation
  - Keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - But each update more expensive
  - No search for replacement

USE OF A STACK TO RECORD THE MOST RECENT PAGE REFERENCES



ALLOCATION OF FRAMES

- Each process needs *minimum* number of frames
- Two major allocation schemes
  - fixed allocation
  - priority allocation

FIXED ALLOCATION

- Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
  - Keep some as free frame buffer pool
- Proportional allocation – Allocate according to the size_of process
  - Dynamic as degree of multiprogramming, process sizes change

—

$$m = 64$$
$$s_1 = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} \times 64 \approx 5$$
$$a_2 = \frac{127}{137} \times 64 \approx 59$$

PRIORITY ALLOCATION

- Use a proportional allocation scheme using priorities rather than size
- If process P_i generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority number

GLOBAL VS. LOCAL ALLOCATION

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
  - But then process execution time can vary greatly
  - But greater throughput so more common
- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

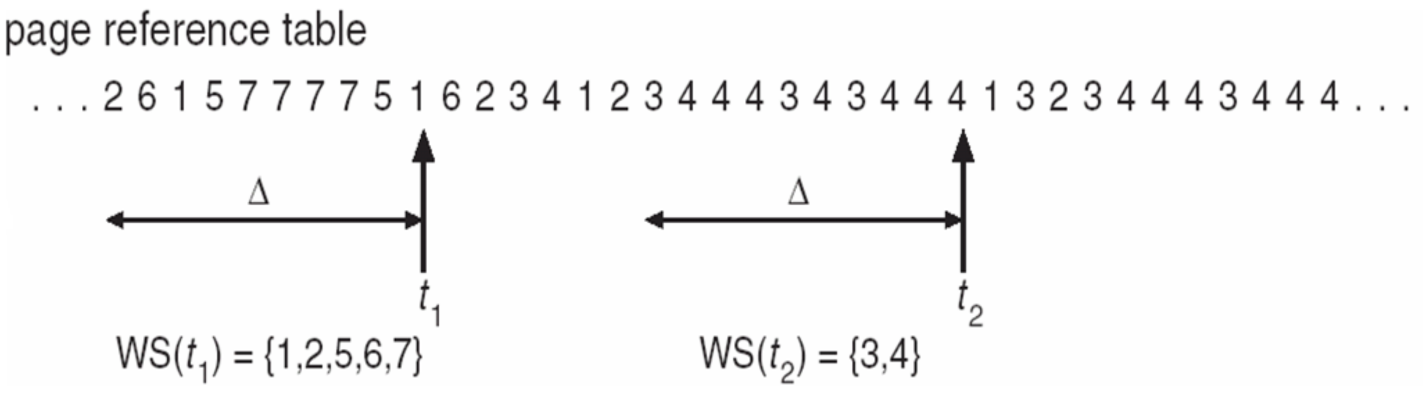
THRASHING

- If a process does not have “enough” frames for the pages, the page-fault rate is very high
- Assuming the CPU is idle, more programs are multiprogrammed and hence it worse the situation.
- This leads to Low CPU utilization.
- **Thrashing** ≡ Excessive Paging operation takes place or a process is busy swapping pages in and out

DEMAND PAGING AND THRASHING

- Why does demand paging work?
  Locality model
  - Process migrates from one locality to another
  - Localities may overlap
- Why does thrashing occur?
  Σ size of locality > total memory size
  - Limit effects by using local or priority page replacement

WORKING-SET MODEL


page reference table
. . . 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 . . .
Δ
t_1
WS(t_1) = {1,2,5,6,7}
Δ
t_2
WS(t_2) = {3,4}