Operating System 2
Dr. Khaled Morsy

Memory management
Review

Chapter 7

Memory Management
(Review)

Introduction

- Main memory is a large array of words or bytes, ranging in size from hundreds of thousands to billions.
- Each word has its own address.
- the main memory is generally the only large storage device that the CPU is able to address and access directly.

- The central processor reads instructions from main memory during the instruction-fetch cycle and both reads and writes data from main memory during the data-fetch cycle.

Introduction

- 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

- To improve both the utilization of the CPU and the speed of its response to users, a general-purpose computer must keep several processes in memory.
- There are several approaches (algorithms) for memory-management .

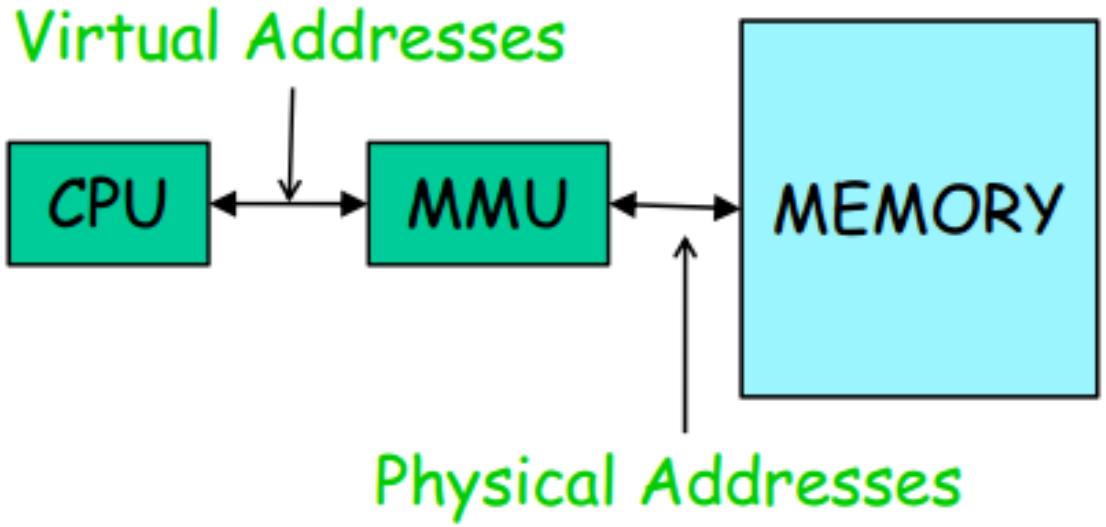
Memory management algorithms

- The effectiveness of each algorithm depends on the situation.
- Selection of a memory-management scheme for a system depends on many factors, especially on the *hardware* design of the system.

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 (loaded into MAR)**

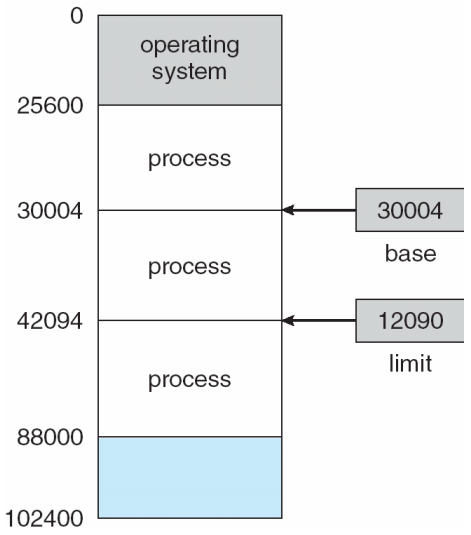
Memory translation and protection

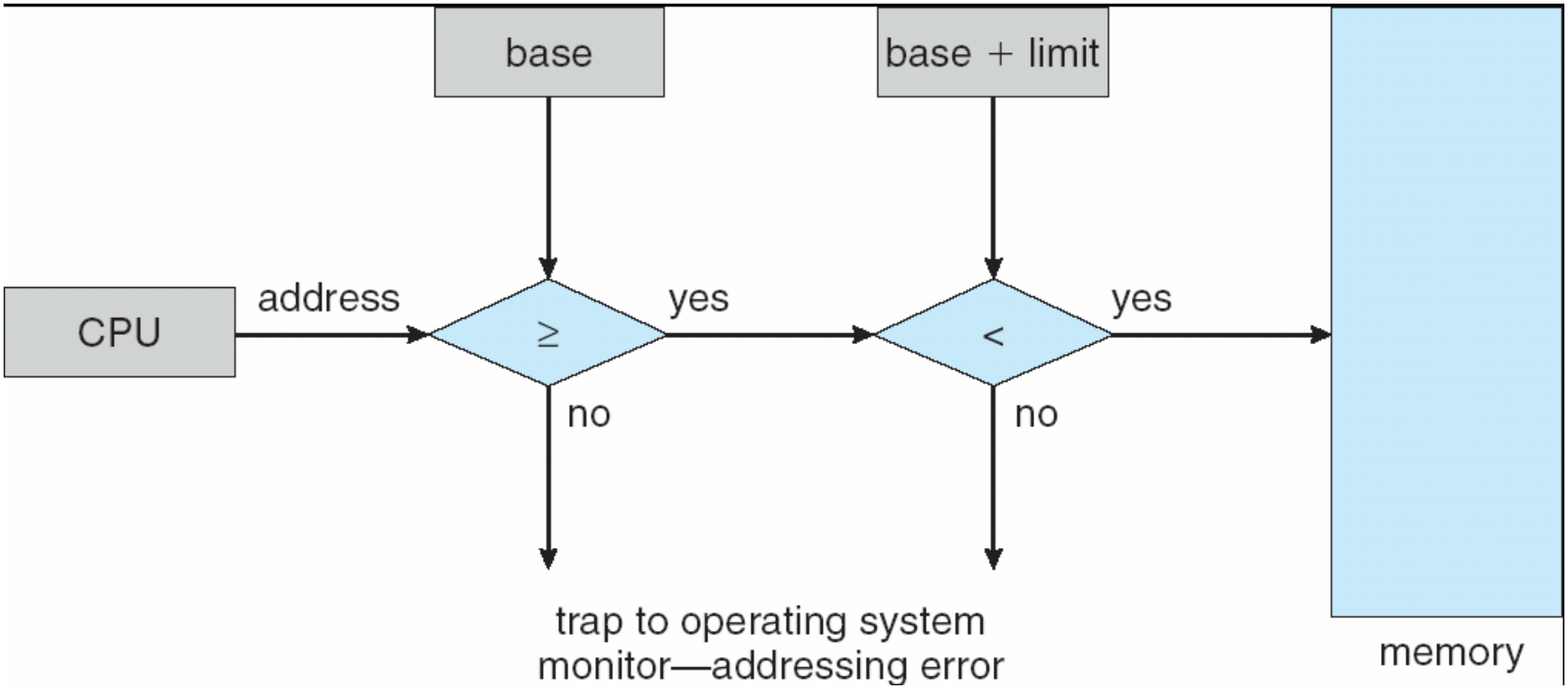


- Map program-generated address (virtual address) to hardware address (physical address) dynamically at every reference
  - MMU: Memory Management Unit
  - Controlled by OS

Using Base and Limit registers

- To have each process has a separate memory space, a base register is used to hold the smallest legal physical memory and limit register (max size allowed for the process in the memory)

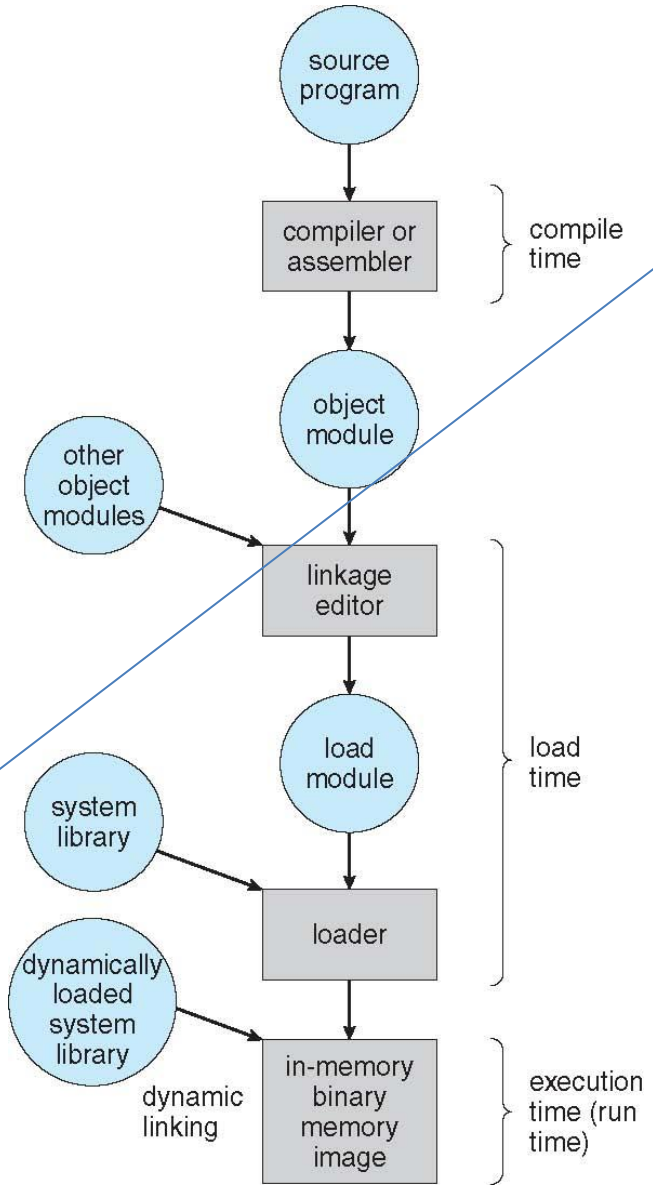



base
base + limit
CPU
address
≥
yes
<
yes
no
no
trap to operating system
monitor—addressing error
memory

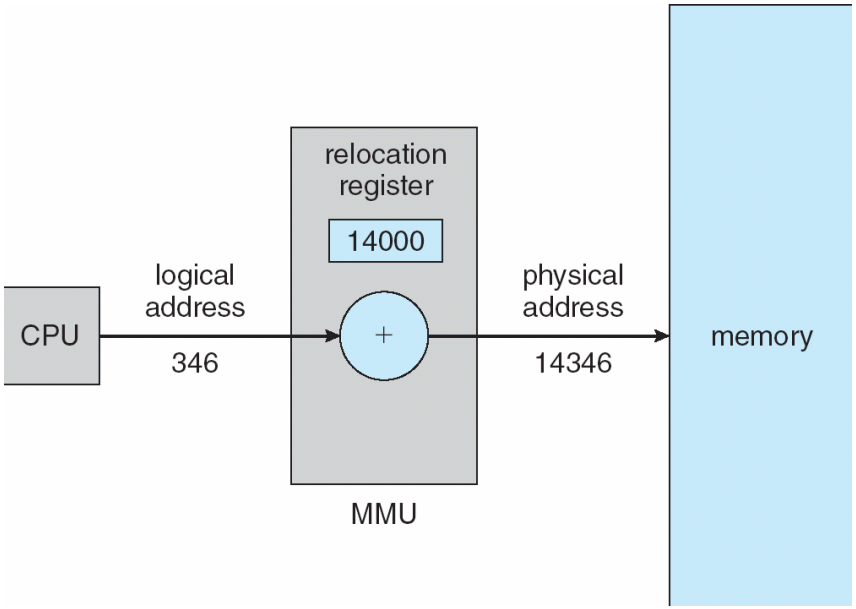
Memory-Management Unit (MMU)

- 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*

Multistep Processing of a User Program



Dynamic Relocation

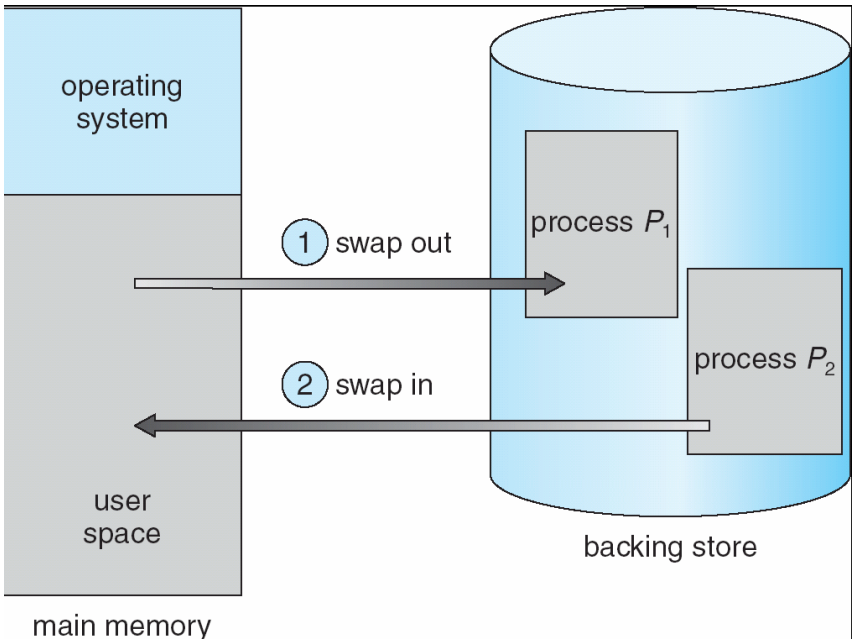


- **Using a relocation register (as a base register)**
- **Added to every *logical address generated by a user process at run time***
- **Make sure that each process has a separate memory space.**

Dynamic Linking

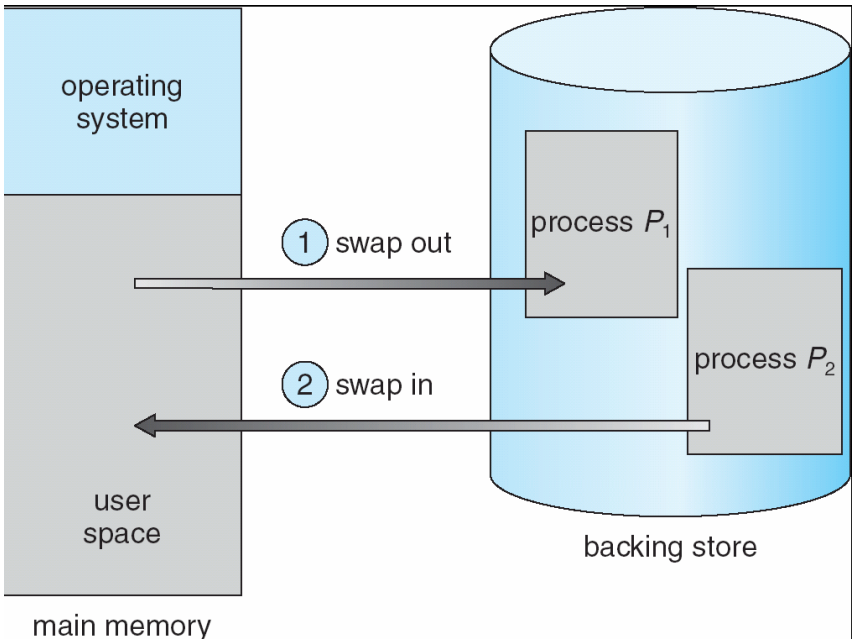
- Linking postponed until execution time
- Small piece of code, (*stub), used to locate the appropriate* memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system needed to check if routine is in processes' memory address
- **Dynamic linking is particularly useful for libraries needed for applications.**

Swapping



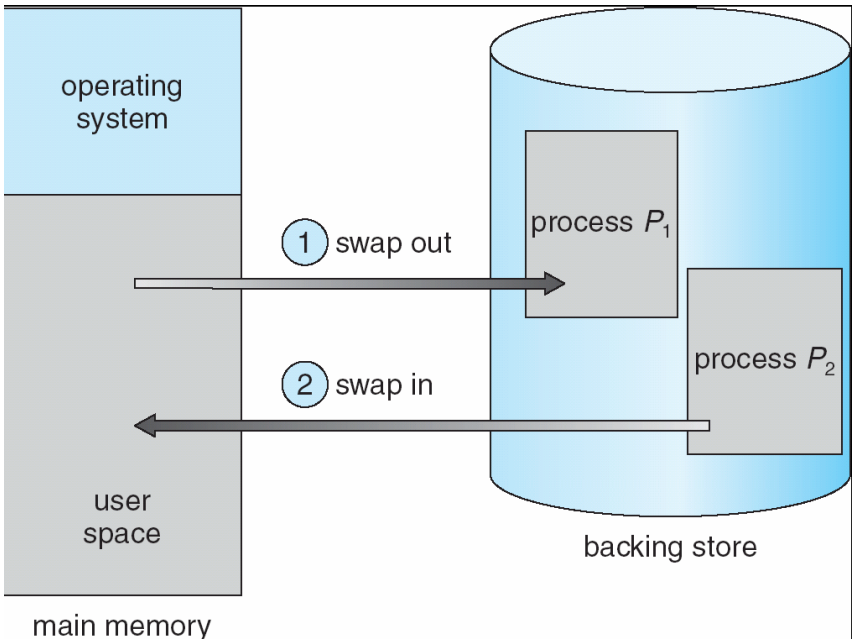
- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution.

Swapping



- **Backing store – fast disk large enough to accommodate copies of** all memory images for all users; must provide direct access to these memory images

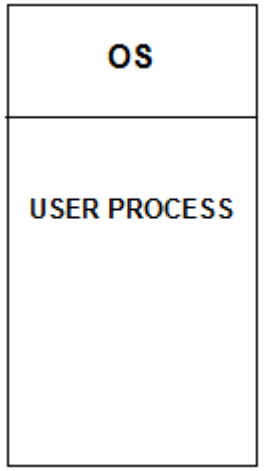
Swapping



- **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

Contiguous Memory Allocation

- Main memory usually divided into two partitions:
- Resident operating system, usually held in low memory with interrupt vector
- User processes , held in high memory



Contiguous Memory Allocation

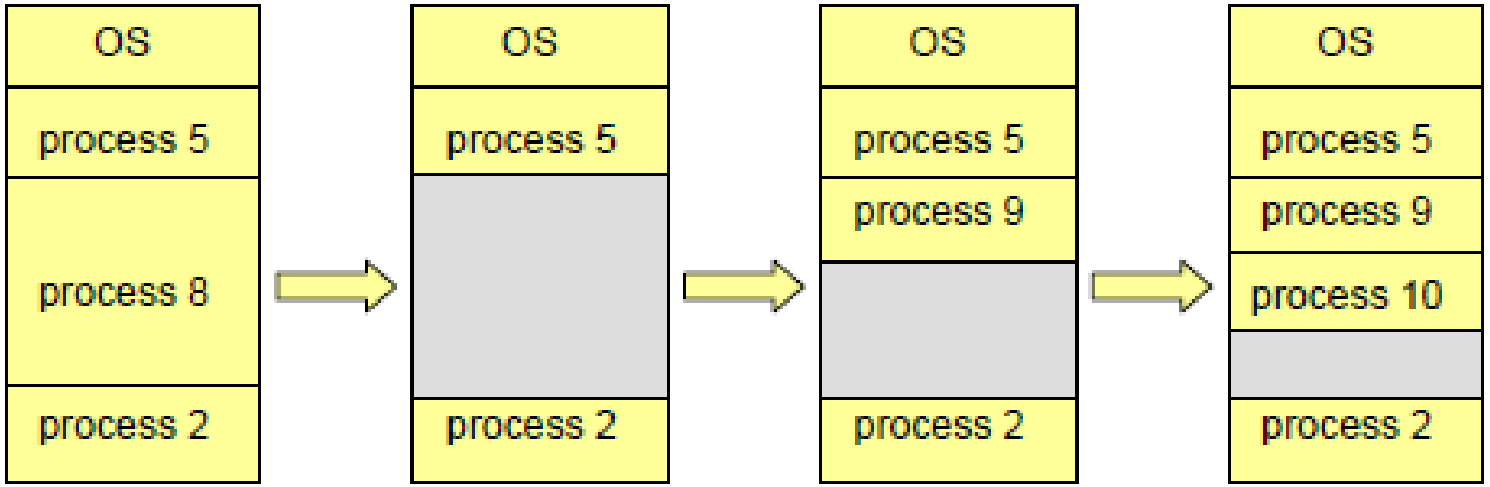
- **fixed-partition allocation**
- Memory is divided into several fixed-sized partitions
- To protect user processes from each other, and from changing operating-system code and data, Relocation-register scheme is used.

Contiguous Memory Allocation

- **fixed-partition allocation**
- Relocation register contains value of smallest physical address;
- limit register contains range of logical addresses.
- Each logical address must be less than the limit register

Contiguous Memory Allocation

- **fitted-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)



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

Example

- Free space: 2 blocks, size 20 and 15
- Workload 1: allocation requests: 10 then 20

Best fit 

First fit  Request of 20: fail!

Worse fit  Request of 20: fail!

- Workload 2: allocation requests: 8, 12, then 13

Best fit  Request of 13: fail!

First fit 

Worse fit  Request of 13: fail!

- Given six memory partitions of 300 KB, 600 KB, 350 KB, 200 KB, 750 KB,
  and 125 KB (in order), how would the first-fit, best-fit, and worst-fit
  place processes of size 115 KB, 500 KB, 358 KB, 200 KB, and375 KB (in order)?

- Rank the algorithms in terms of how efficiently they use memory.
- Answer:
- First-fit:
- 115 KB is put in 300 KB partition, leaving (185 KB, 600 KB, 350 KB,200 KB , 750 KB, 125 KB)
- 500 KB is put in 600 KB partition, leaving (185 KB, 100 KB, 350 KB,200 KB , 750 KB, 125 KB)
- 358 KB is put in 750 KB partition, leaving (185 KB, 100 KB, 350 KB,200 KB , 392 KB, 125 KB)
- 200 KB is put in 350 KB partition, leaving (185 KB, 100 KB, 150 KB,200 KB , 392 KB, 125 KB)
- 375 KB is put in 392 KB partition, leaving (185 KB, 100 KB, 150 KB,200 KB, 17 KB, 125 KB)

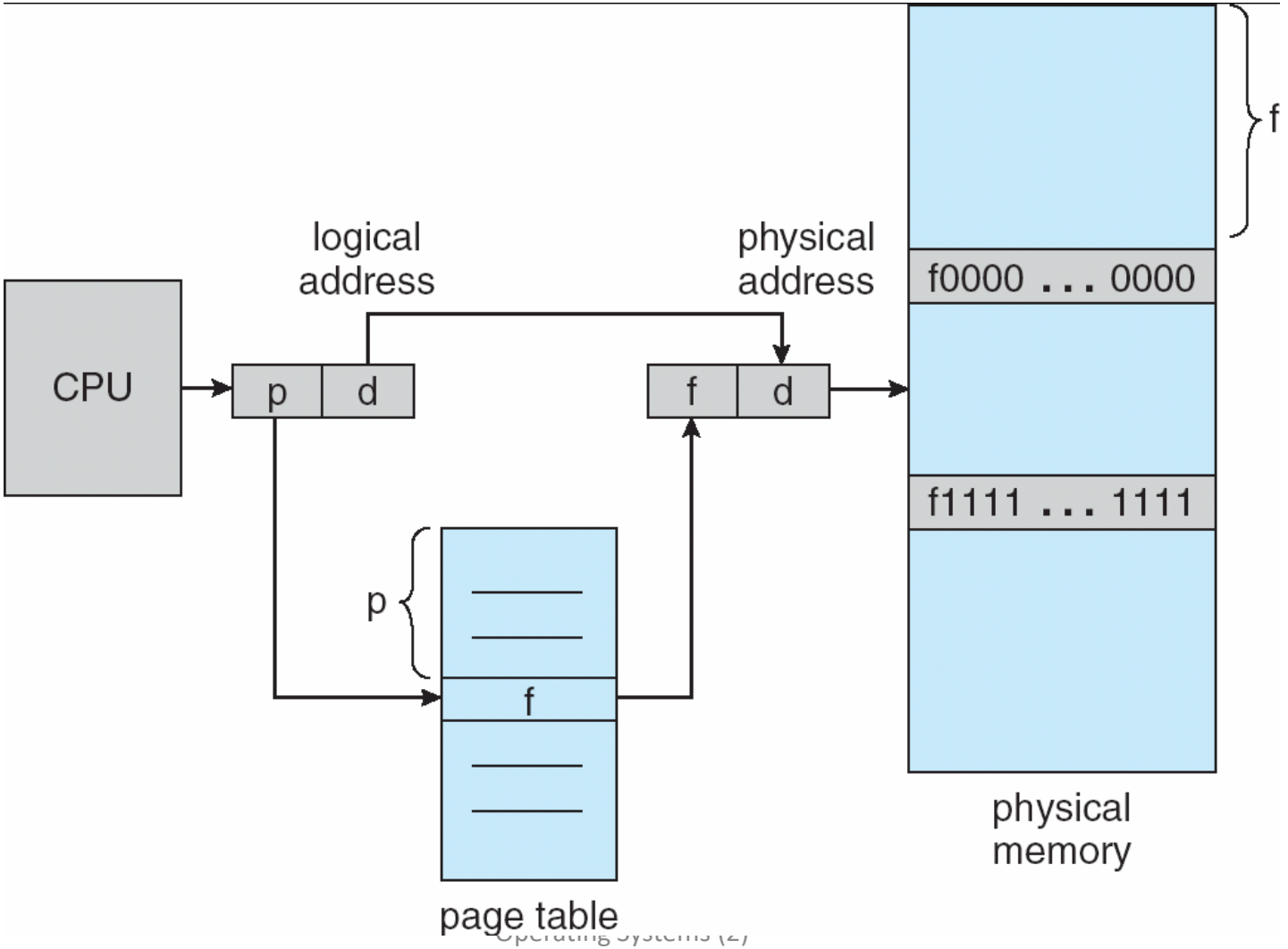
Paging

- 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 8192 bytes)
- Divide *logical memory into blocks of same size called* ***pages.***

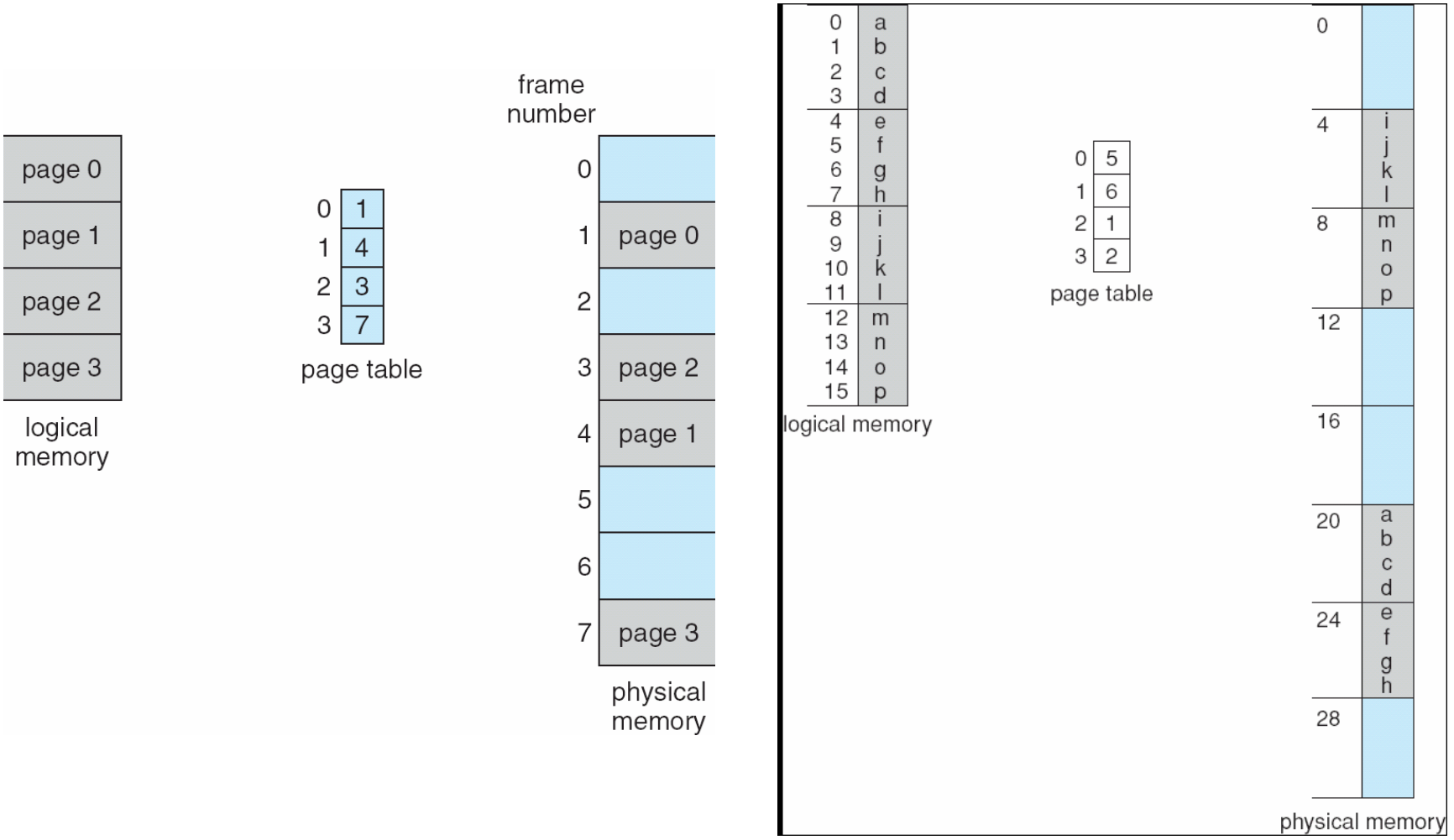
Paging

- 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.

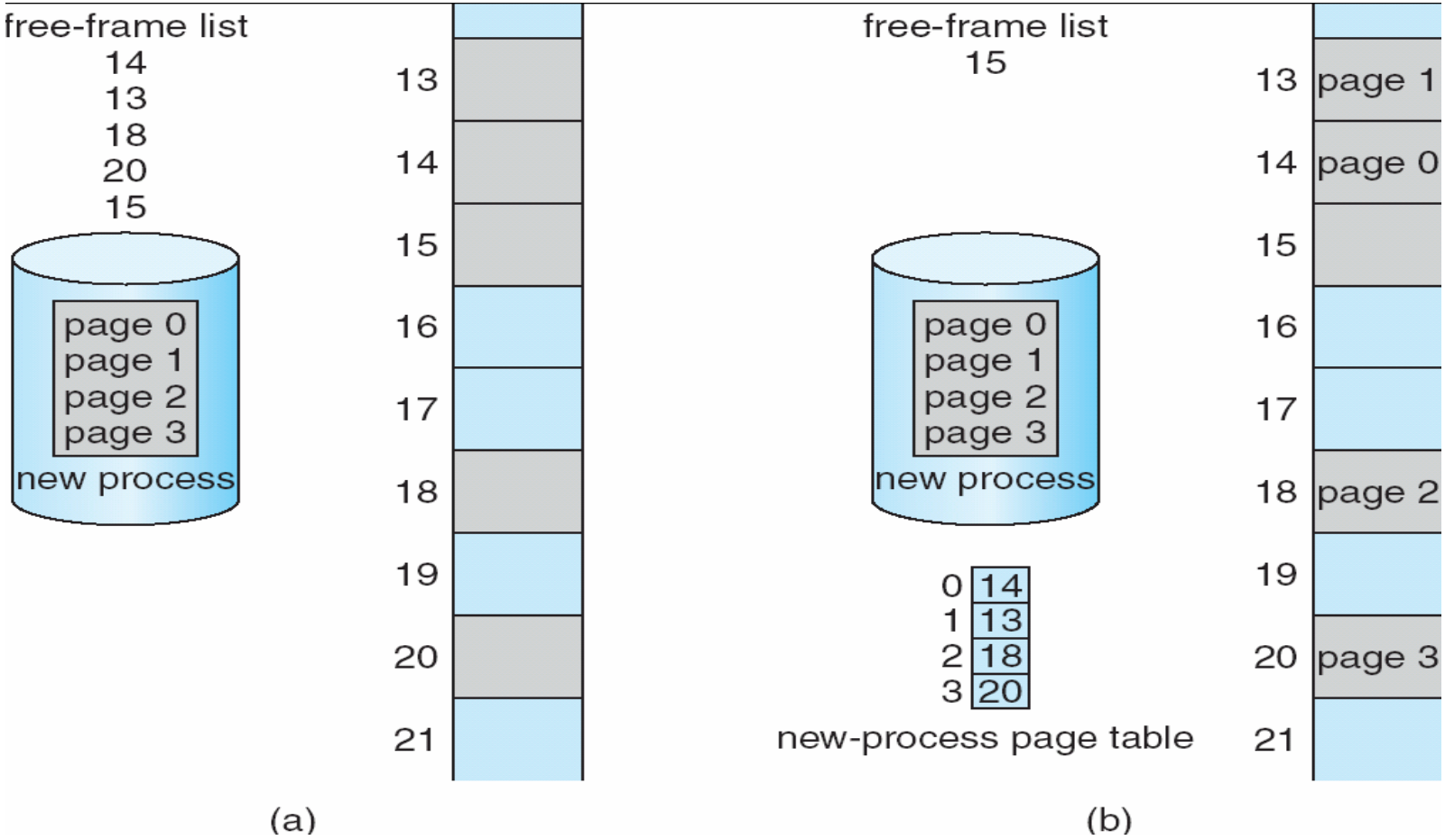
- Address generated by CPU is divided into:
- *Page number (p) – used as an index into a page table which contains* base address of each page in physical memory
- *Page offset (d) – combined with base address to define the physical* memory address that is sent to the memory unit.


CPU
logical address
p
d
physical address
f
d
f0000 . . . 0000
f1111 . . . 1111
f
p
f
page table
physical memory

Paging Example



Free Frames


free-frame list
14
13
18
20
15
page 0
page 1
page 2
page 3
new process
13
14
15
16
17
18
19
20
21
(a)
free-frame list
15
page 0
page 1
page 2
page 3
new process
0 14
1 13
2 18
3 20
new-process page table
13 page 1
14 page 0
15
16
17
18 page 2
19
20 page 3
21
(b)

Implementation of Page Table

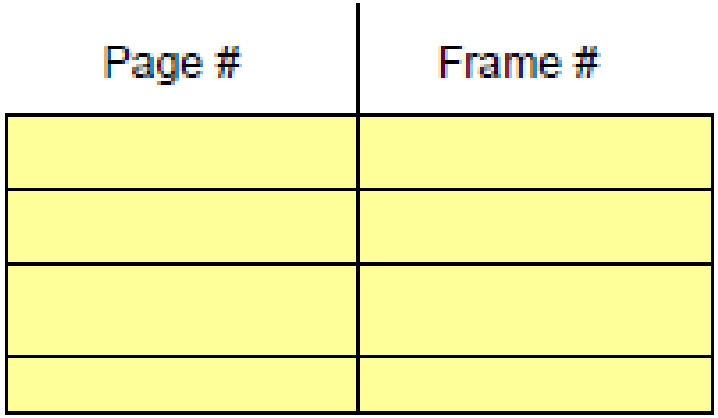
- Page table is kept in main memory
- *Page-table base register (PTBR) points to the page table*
- *Page-table length register (PRLR) indicates size of the page* table
- In this scheme every data/instruction access requires **two memory accesses.** One for the page table and one for the data/instruction.

Implementation of Page Table

- 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)**

Associative Memory

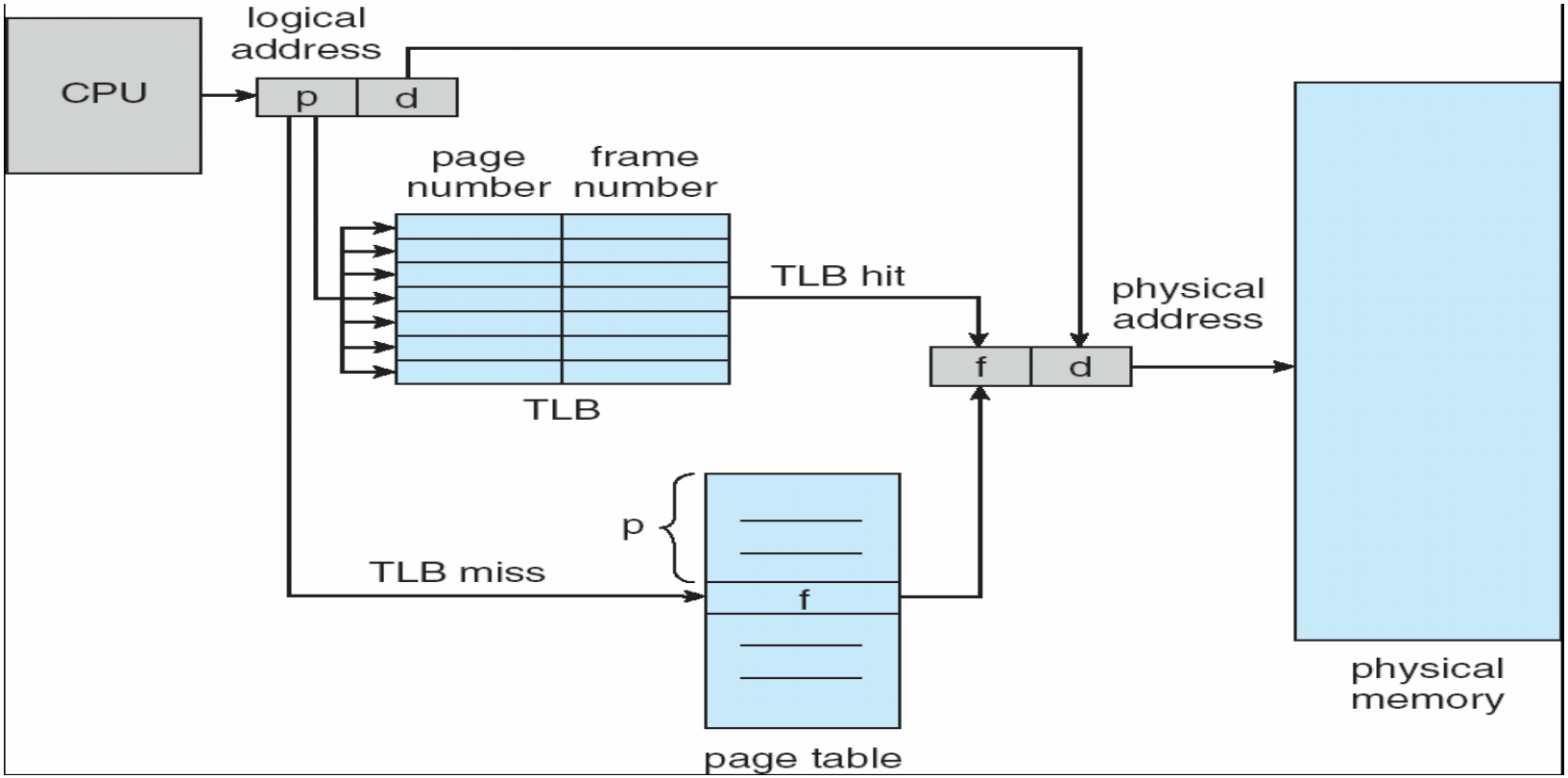
- Associative memory – parallel search



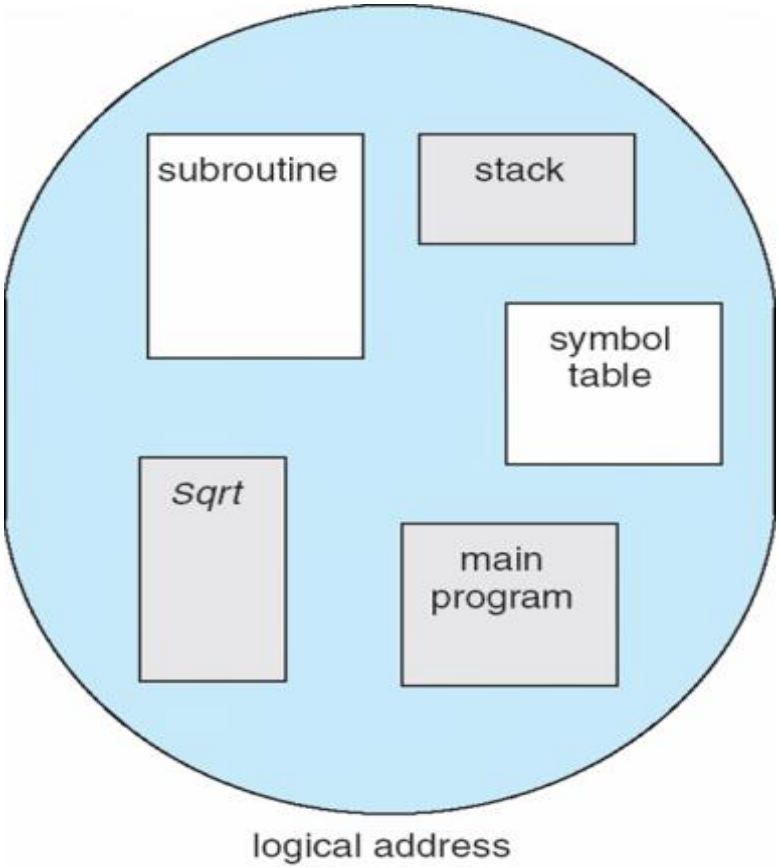
Address translation (A´, A´´)

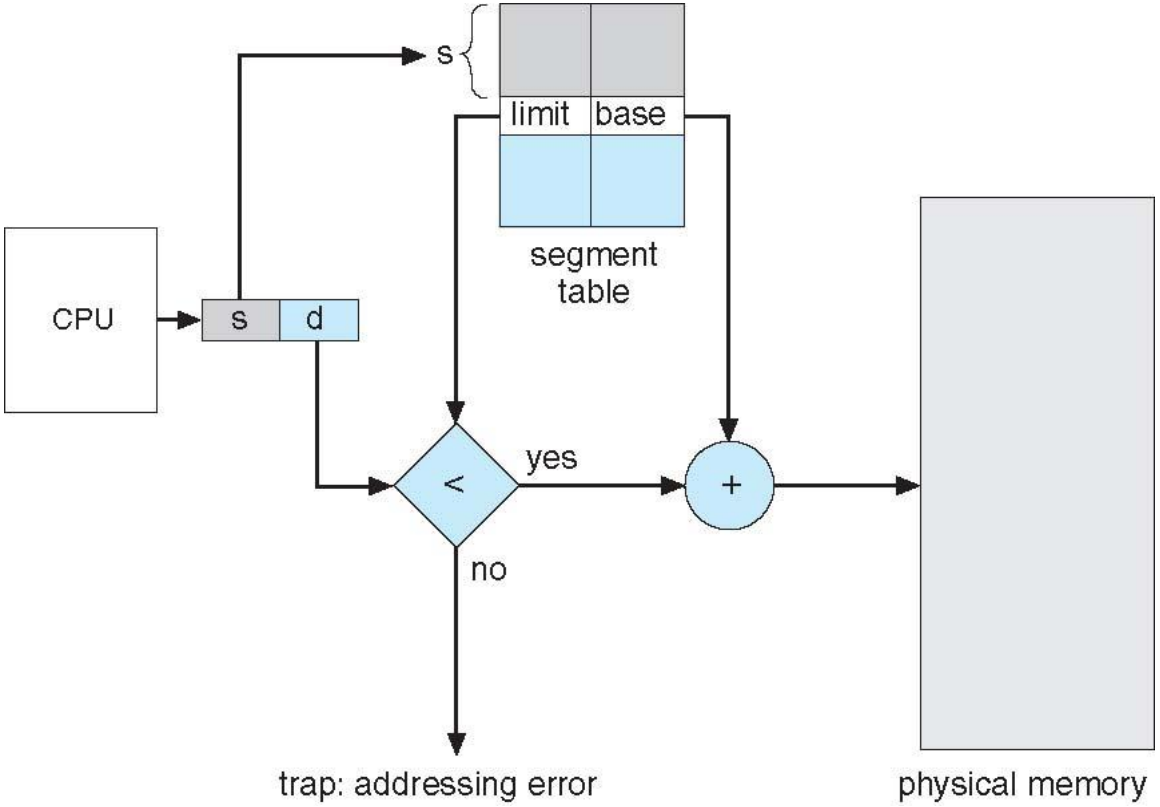
- If A´ is in associative register, get frame # out
- Otherwise get frame # from page table in memory

Paging Hardware With TLB



Segmentation




s
limit
base
segment
table
CPU
s
d
<
yes
+
no
trap: addressing error
physical memory

Virtual Memory

Dr. Khaled Morsy

Objectives

- To describe the benefits of a virtual memory system
- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames
- To discuss the principle of the working-set model

Background

- Code needs to be in memory to execute, but entire program rarely used:
  - Unusual Error code routines, large data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program:
  - **Program no longer constrained by limits of physical memory**
  - **Several programs could be allocated to physical memory at the same time.** More programs running concurrently
  - **Less I/O needed to load or swap processes**

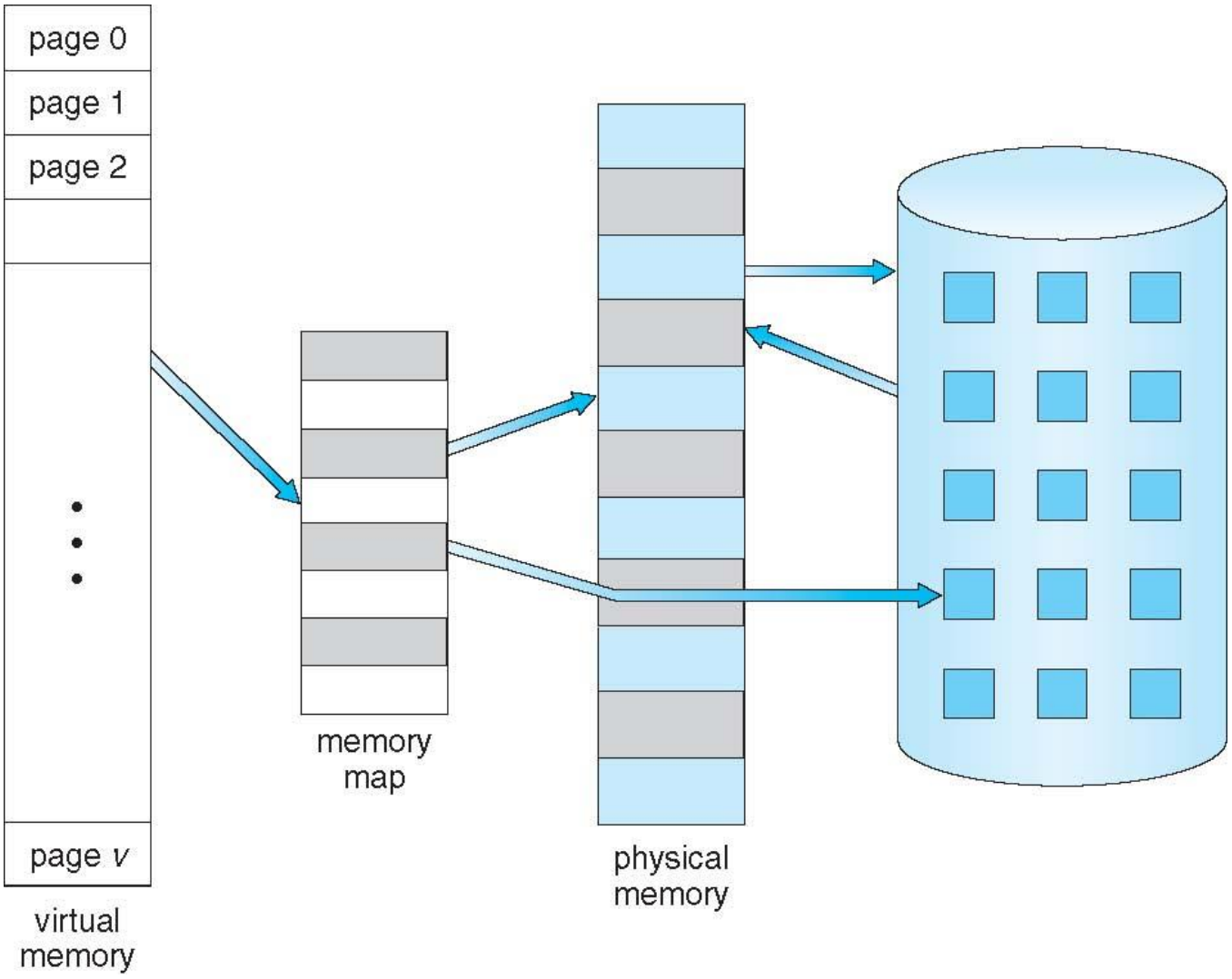
Benefits of Virtual Memory

- **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

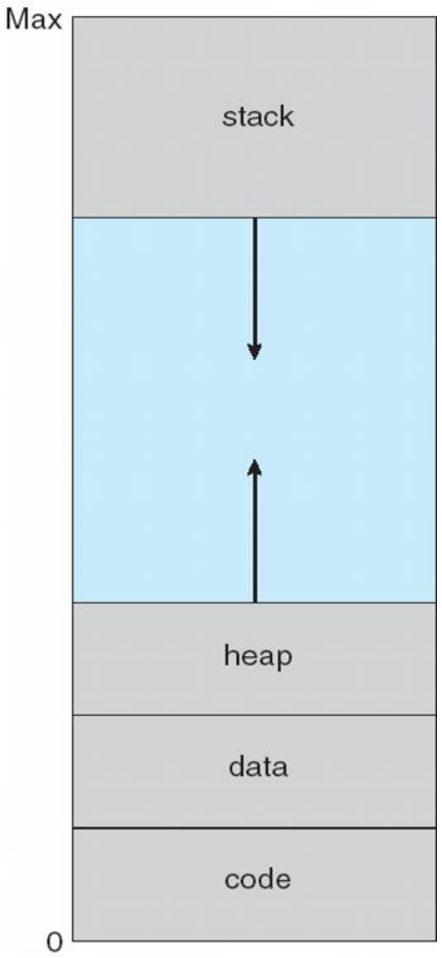
Implementation

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

Virtual Memory That is Larger Than Physical Memory

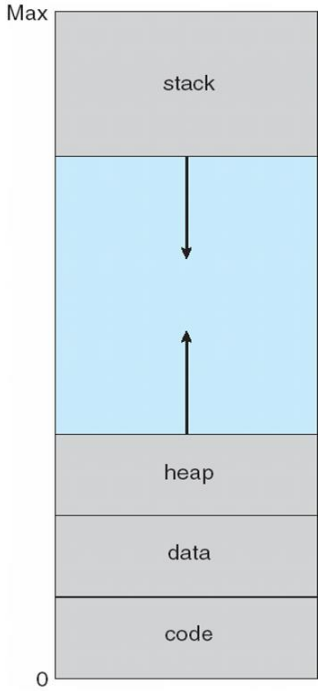


Virtual-address Space per process

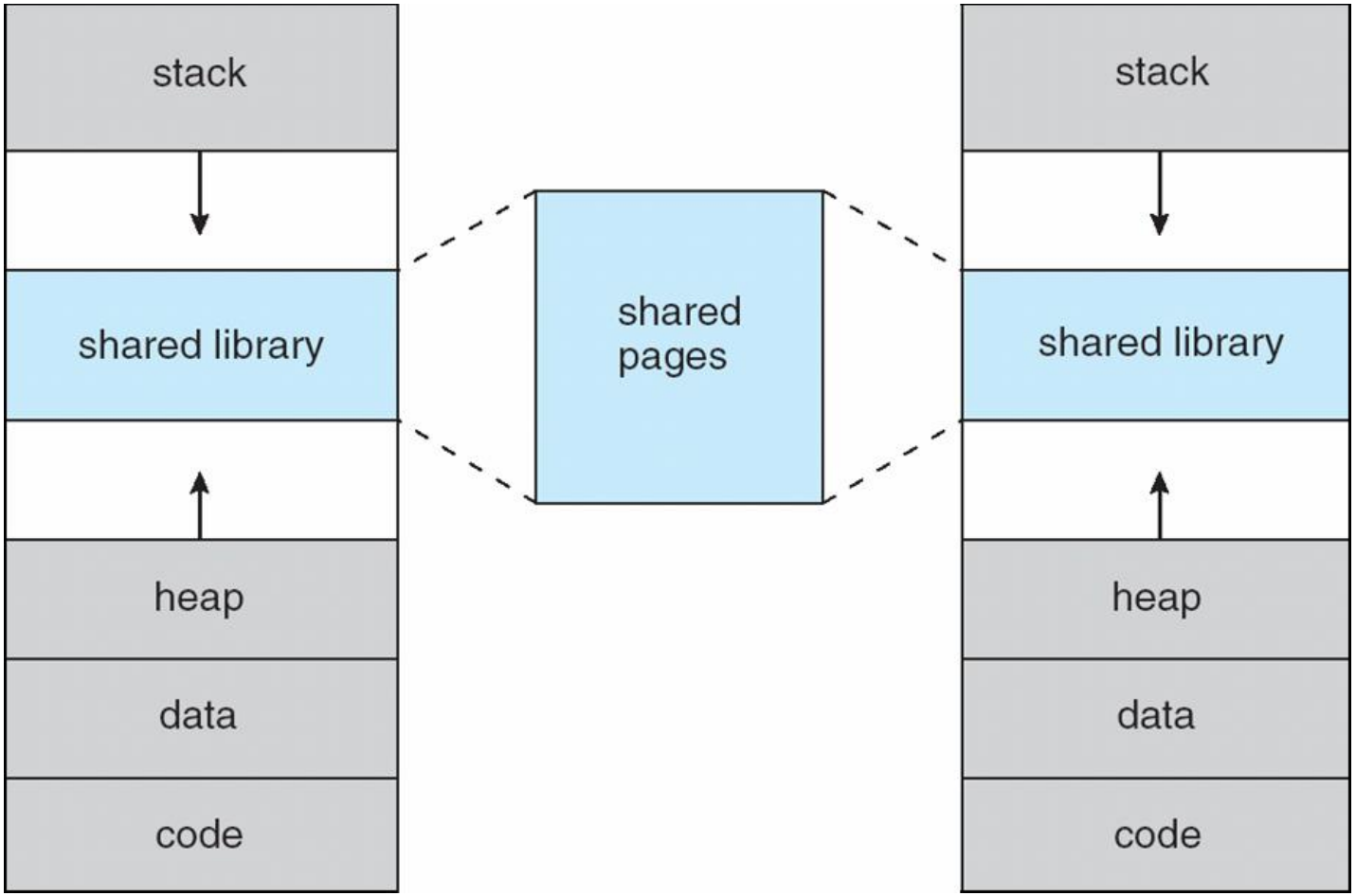


Virtual Address Space

- Starts at a certain address , say 0 – and exists in contiguous memory



Shared Library Using Virtual Memory



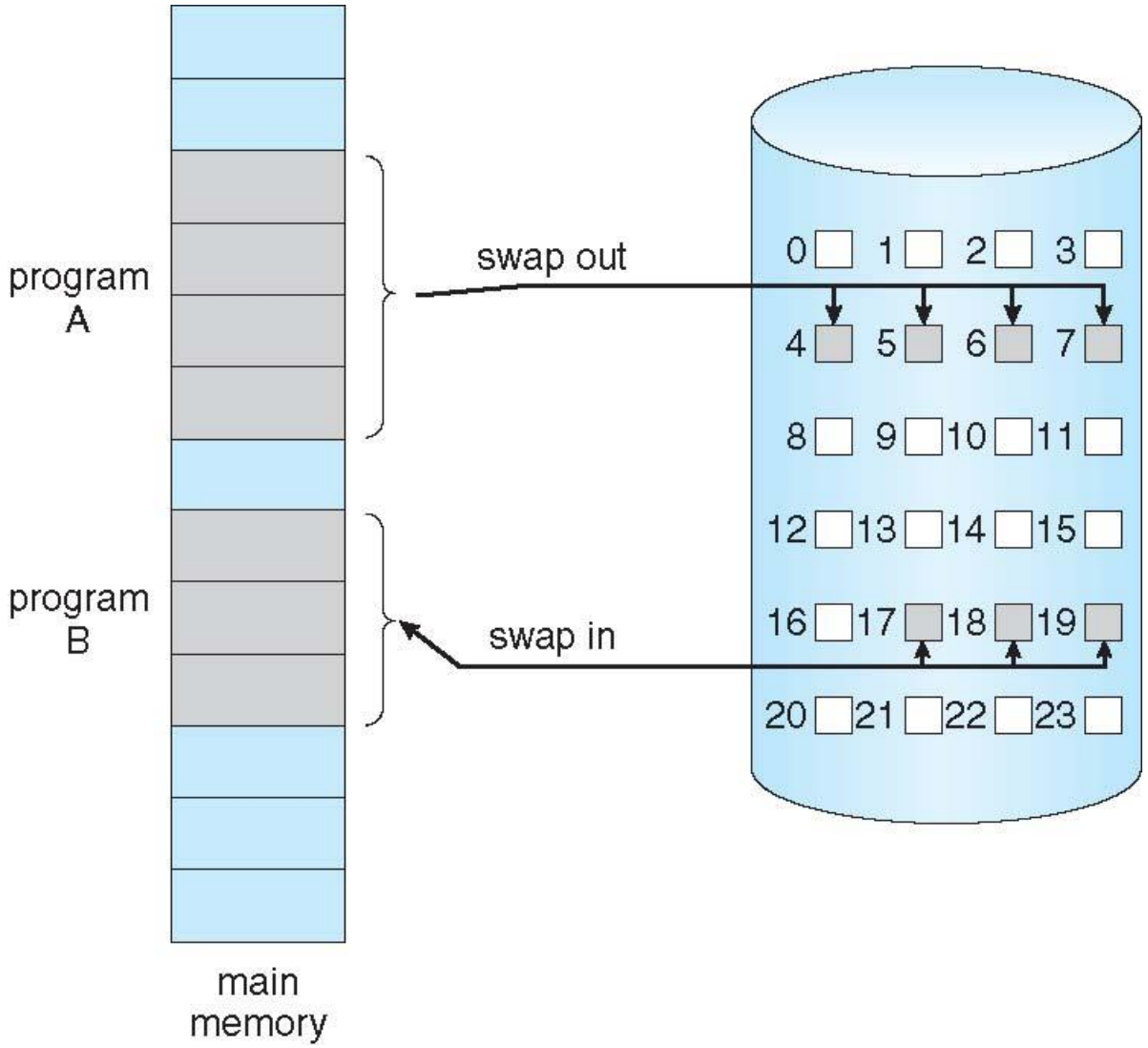
Demand Paging

- Consider how an executable program might be loaded from disk into memory :
- Loading the entire program in physical memory at run time (we may need not all the program modules)
- Or 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

Hardware to support demand paging

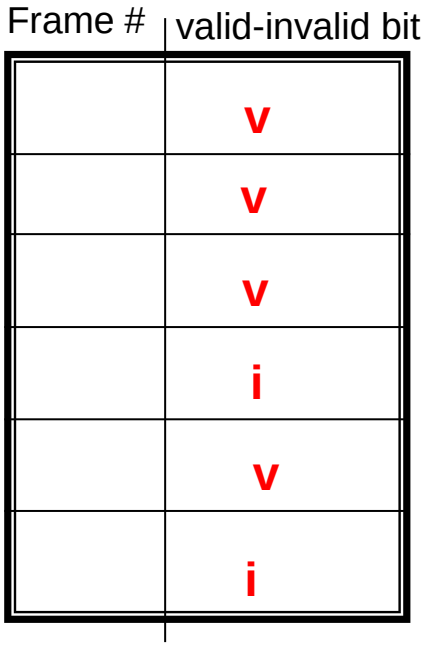
- Page table

Transfer of a Paged Memory to Contiguous Disk Space



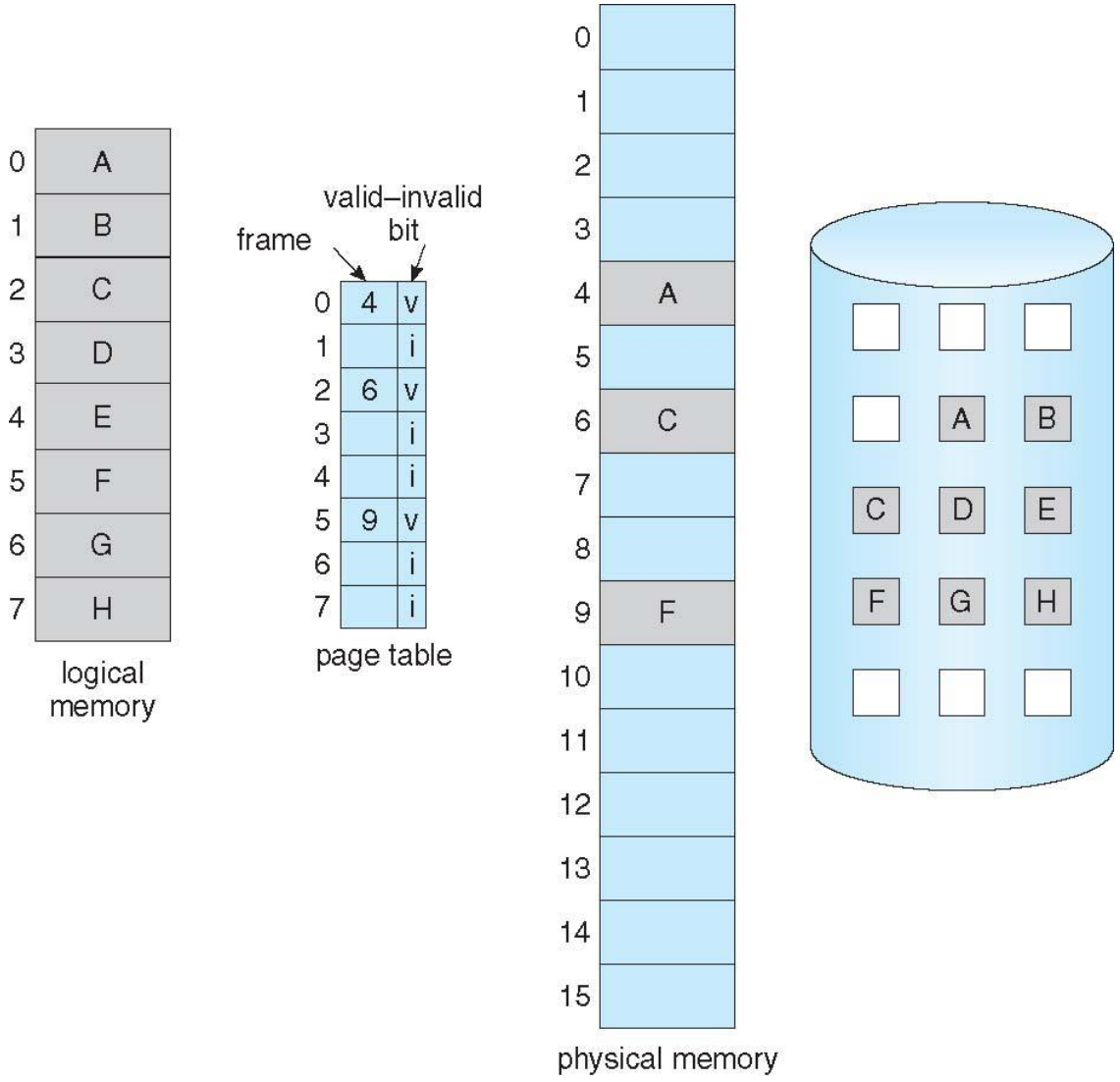
Valid-Invalid Bit

- 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
- Example of a page table snapshot:



page table

Page Table When Some Pages Are Not in Main Memory



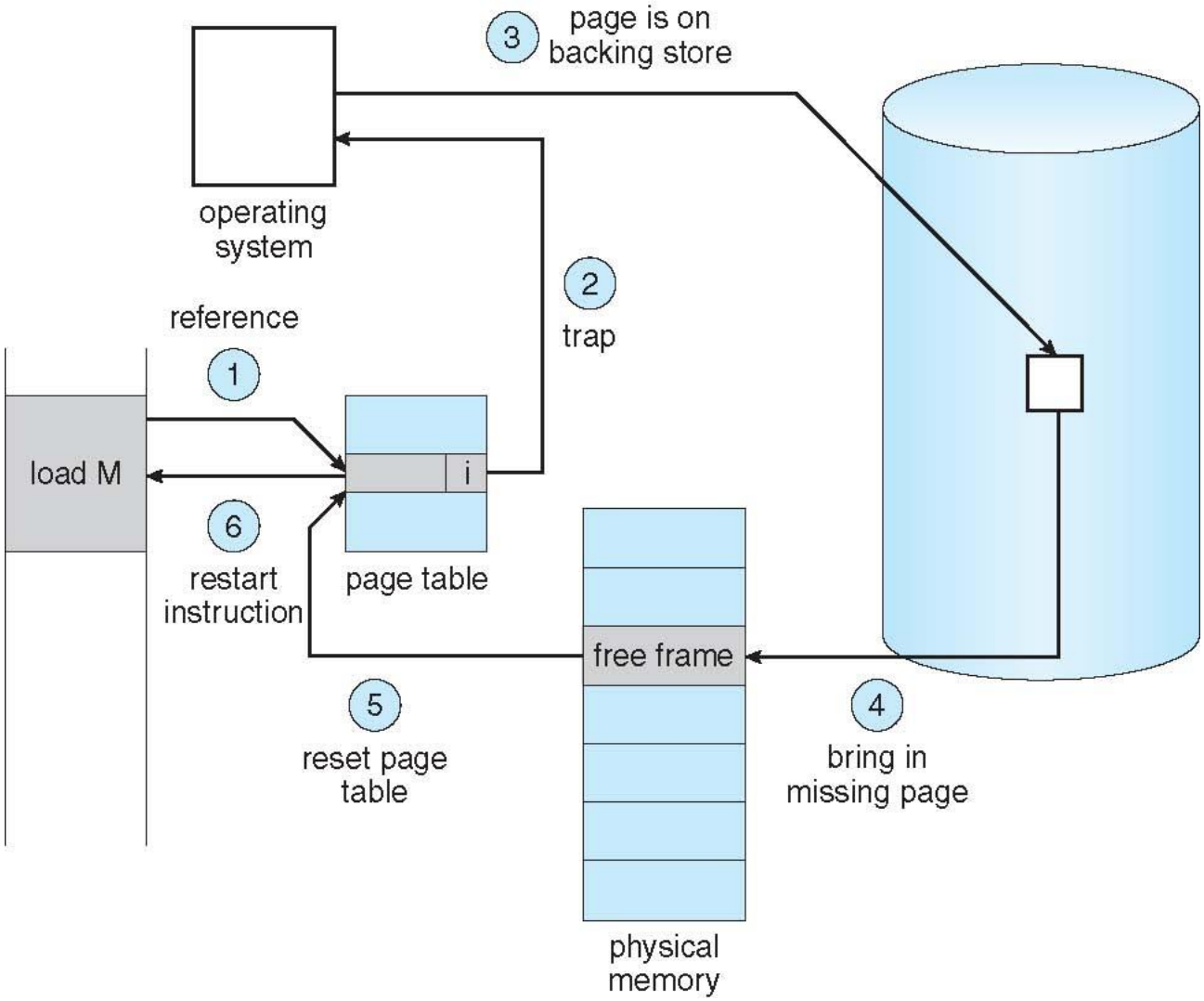
Page Fault

- 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 $\Rightarrow$ abort
   - Just not in memory
2. Get empty frame
3. Swap page into frame via scheduled disk operation

Steps in Handling a Page Fault



Performance of Demand Paging

- Stages in Demand Paging

1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame:
    1. Wait in a queue for this device until the read request is serviced
    2. Wait for the device seek and/or latency time
    3. Begin the transfer of the page to a free frame
6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Performance of Demand Paging (Cont.)

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

- Effective Access Time (EAT)

EAT = (1 – *p*) x memory access
+ *p* (page fault overhead
+ swap page out
+ swap page in
+ restart overhead)

Demand Paging Example

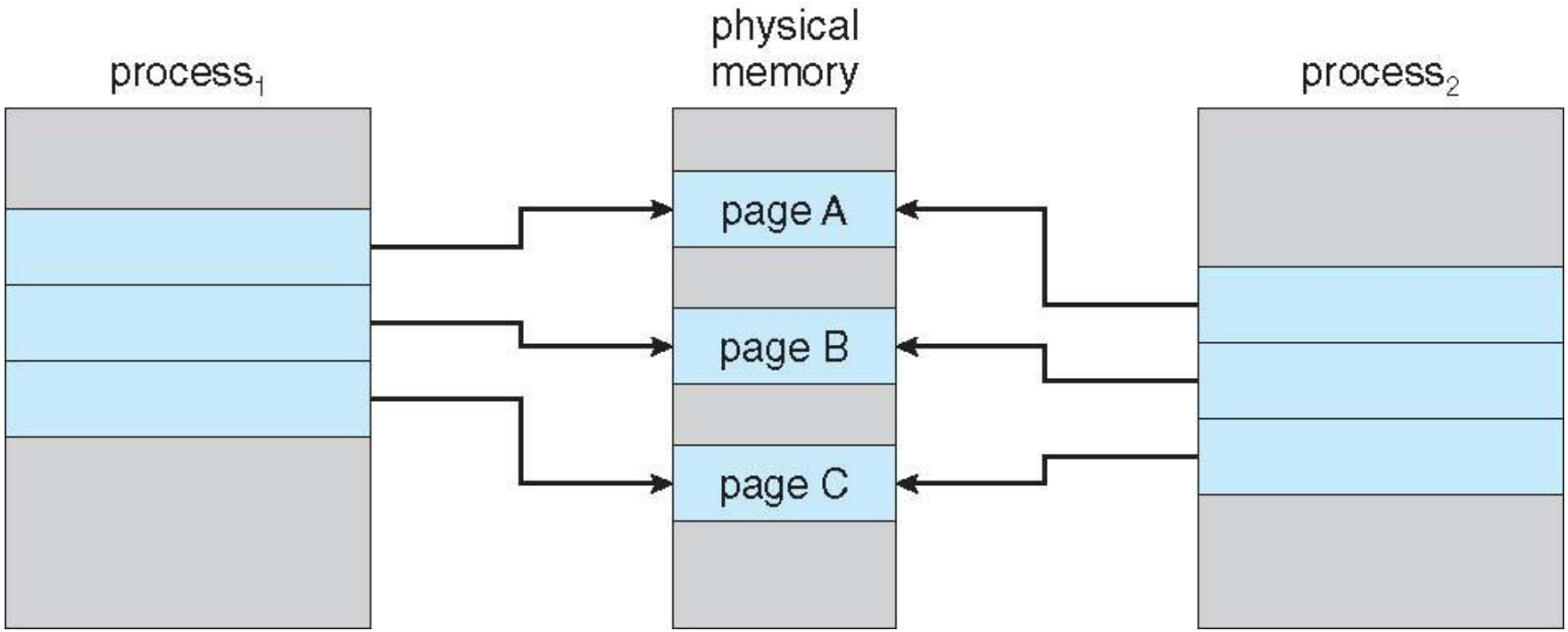
- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds

- EAT = (1 – p) x 200 + p (8 milliseconds)
  ====== millisecond = 1000,000 nanosecond
  = (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

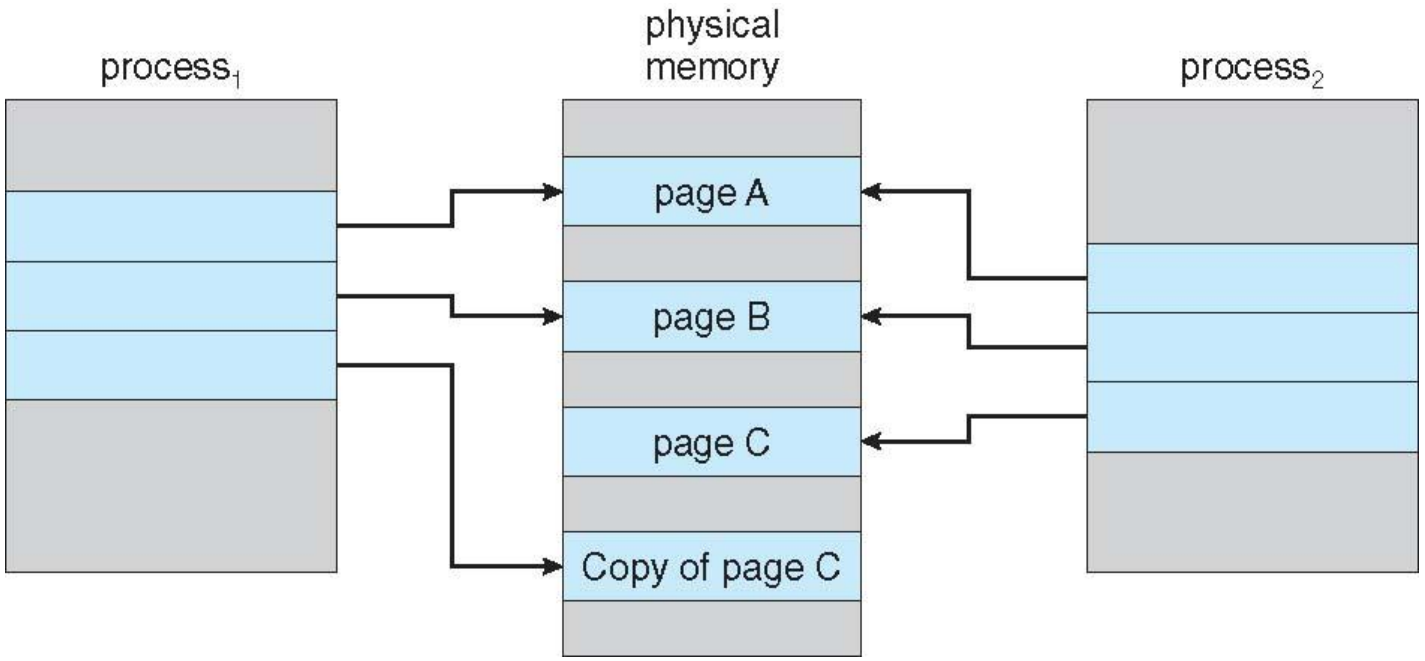
Copy-on-Write

- **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, then only this the page will be copied
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages
  - Why zero-out a page before allocating it?
- `vfork()` variation on `fork()` system

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



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

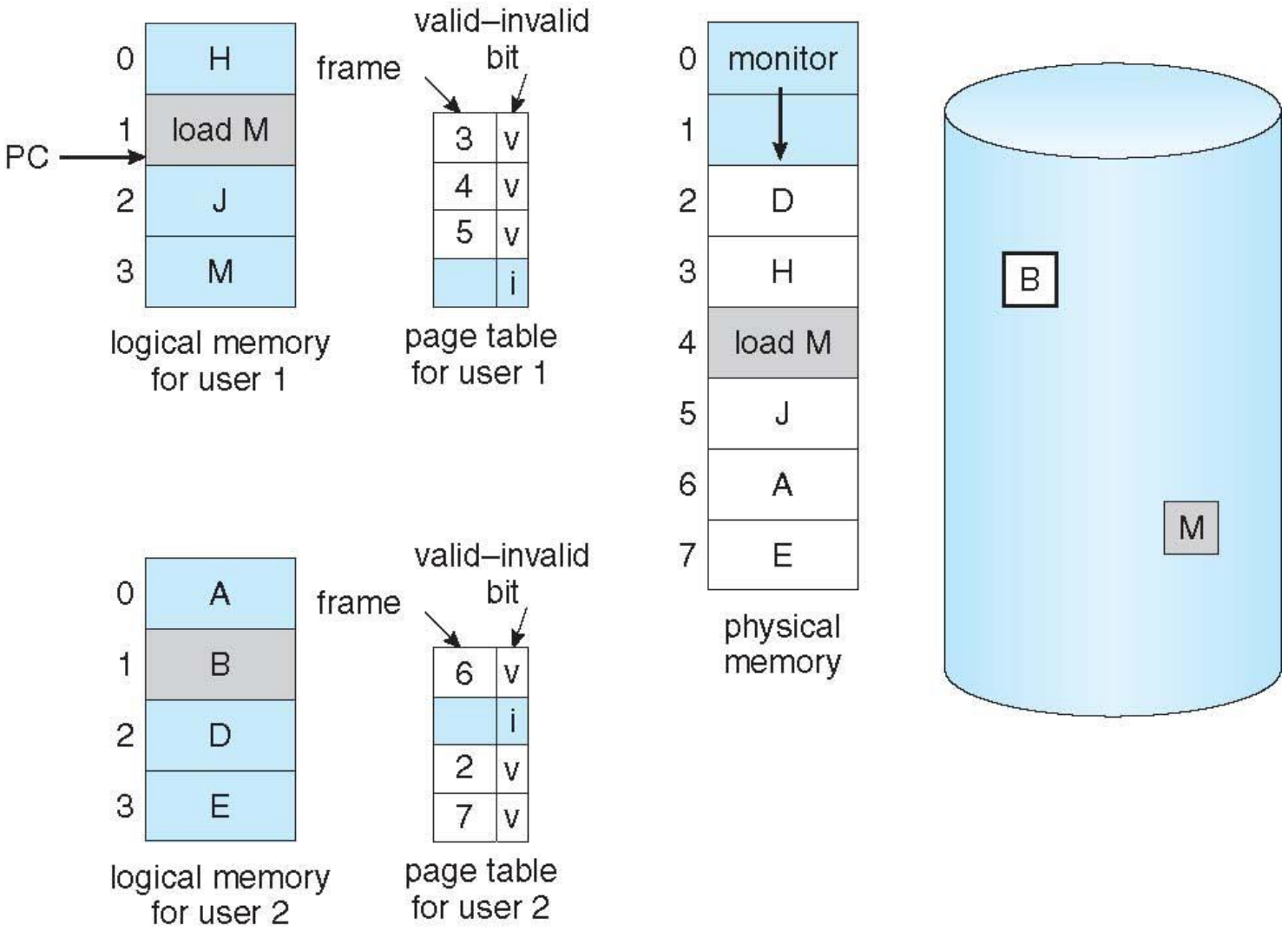
Page Replacement

- 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

Need For Page Replacement



Basic Page Replacement

1. Find the location of the desired page on disk

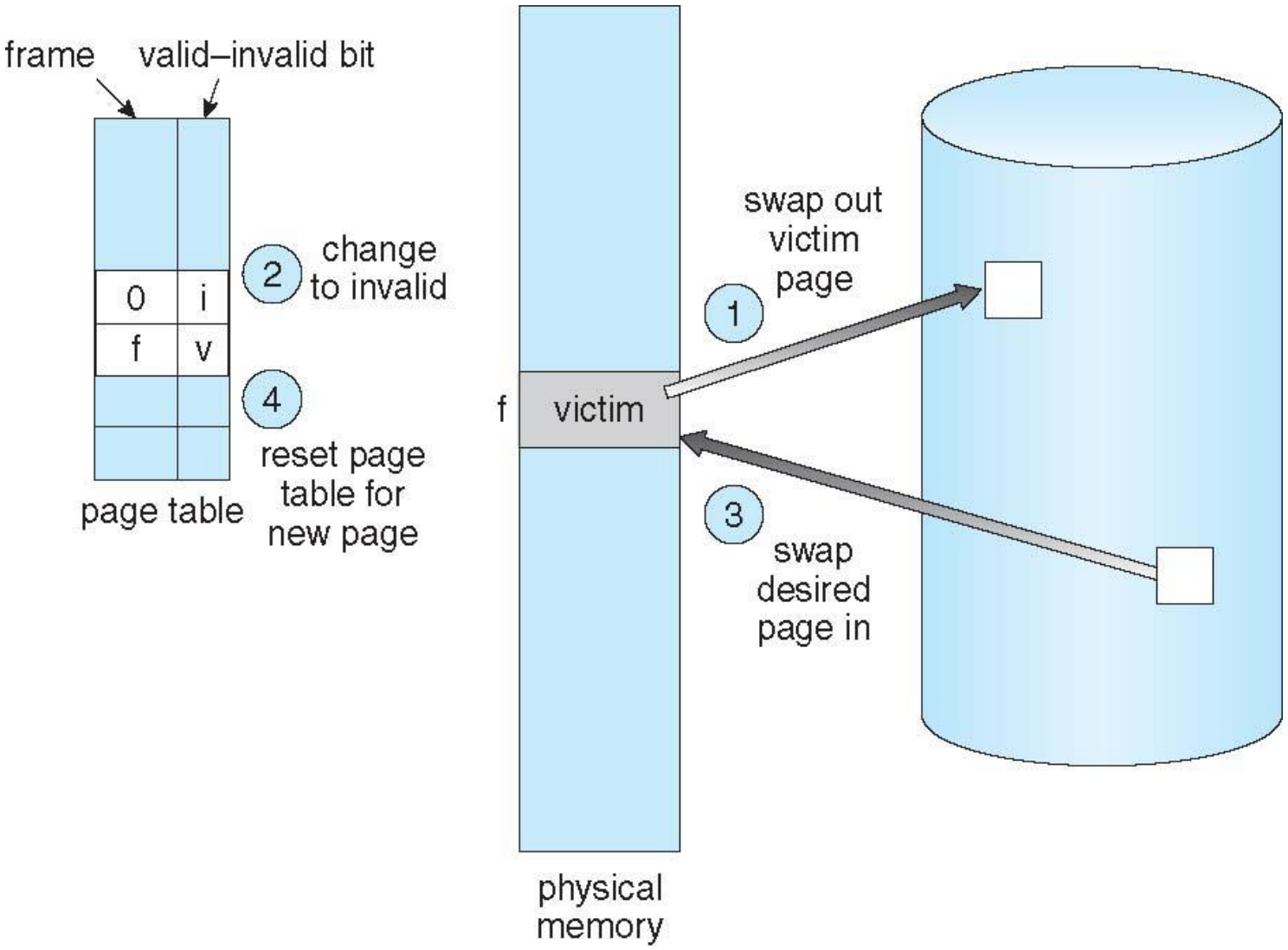
2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim frame**
     - Write victim frame to disk if dirty

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

Note now potentially 2 page transfers for page fault – increasing EAT

Page Replacement

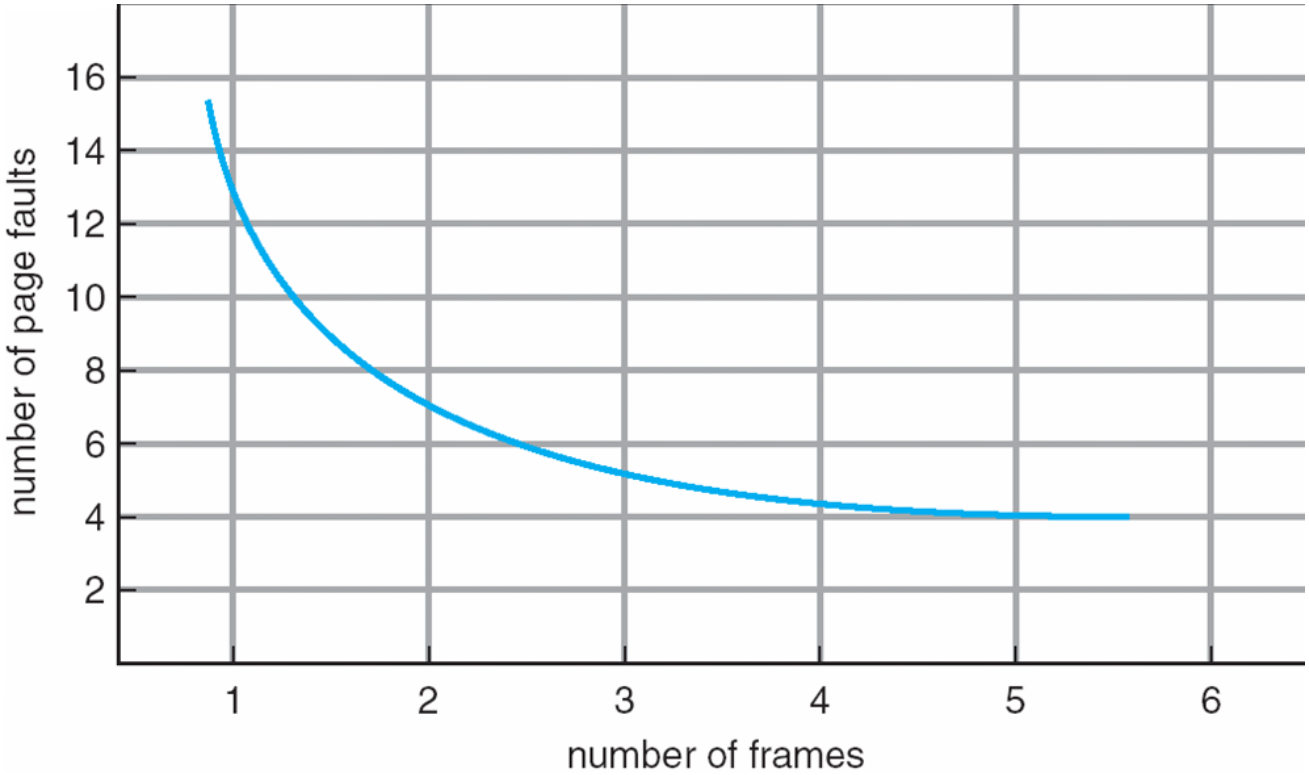


Page and Frame Replacement Algorithms

- **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

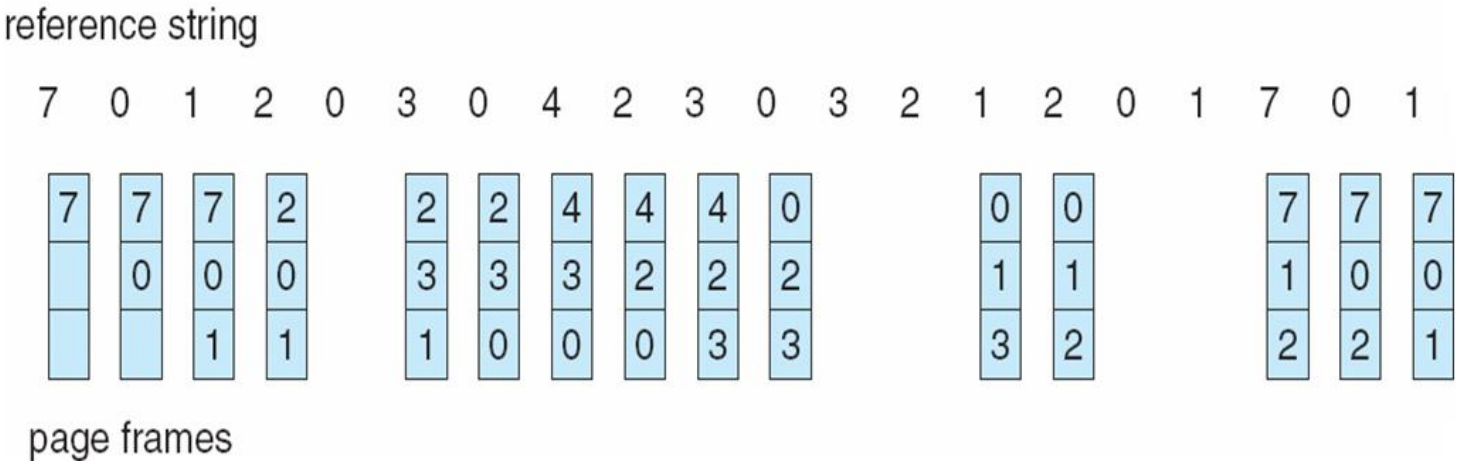
Graph of Page Faults Versus The Number of Frames



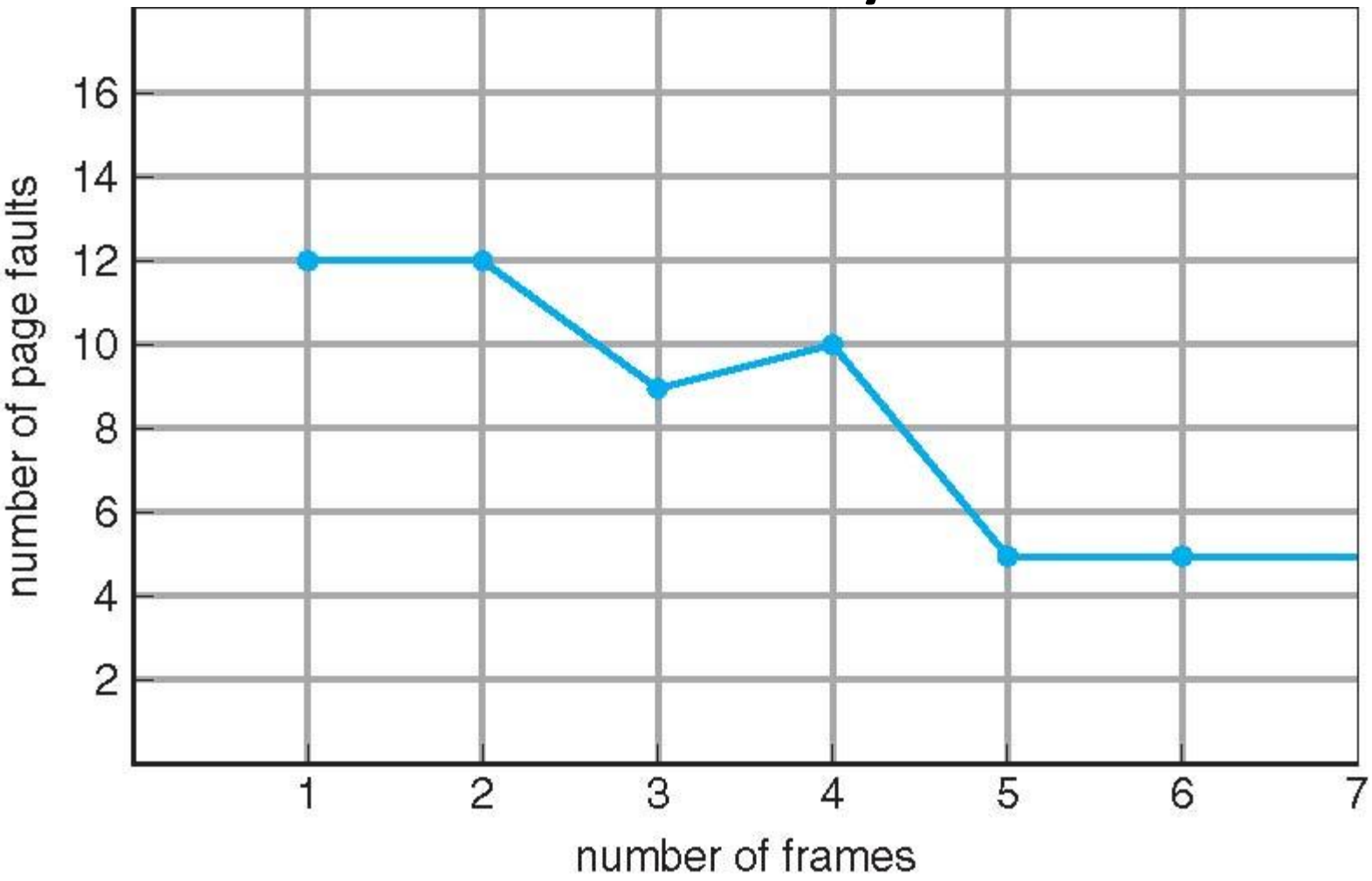
First-In-First-Out (FIFO) Algorithm

- 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)

FIFO Page Replacement



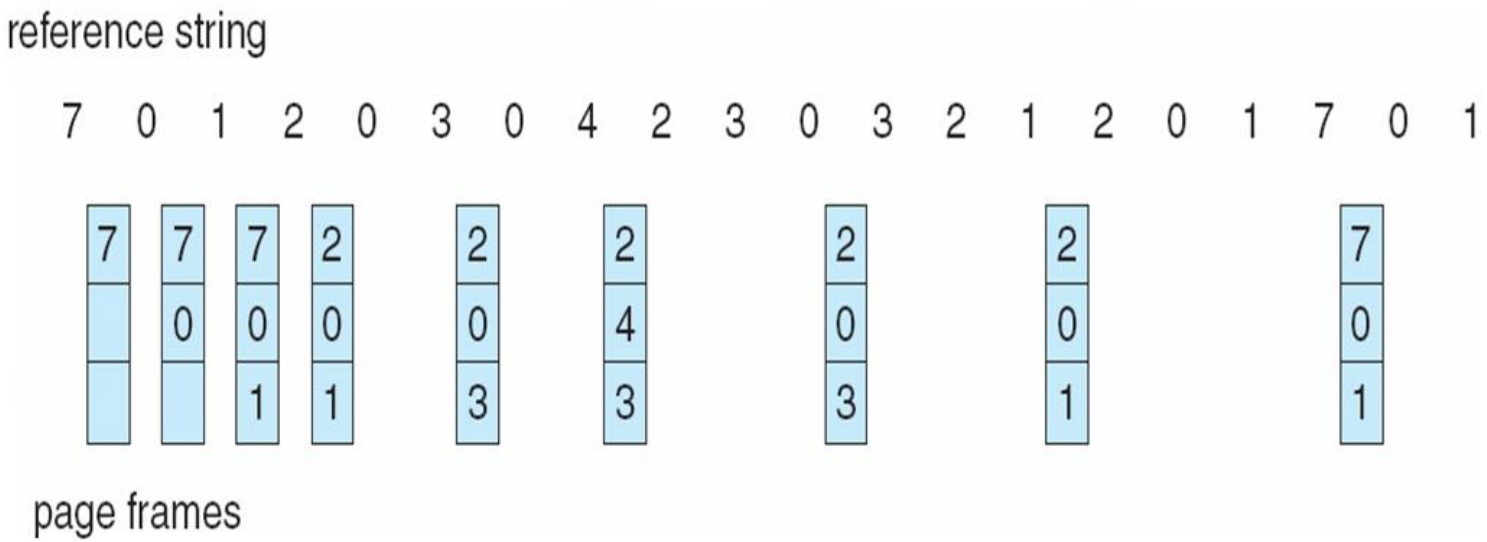
FIFO Illustrating Belady's Anomaly



Optimal Algorithm

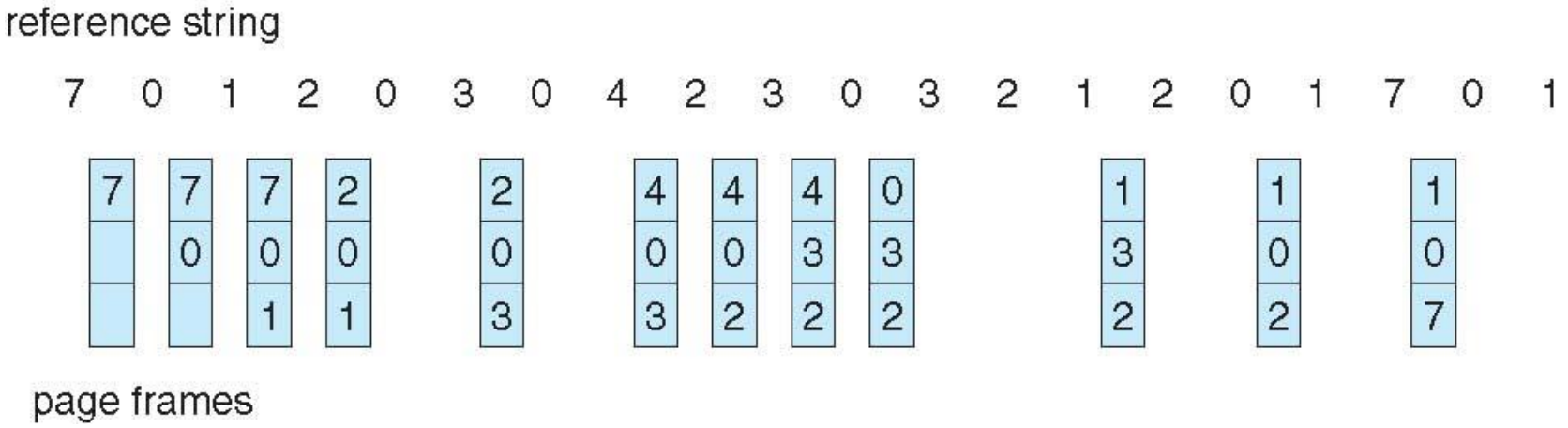
- Replace page that will not be used for longest period of time
  - 9 is optimal for the example on the next slide
- How do you know this?
  - Can't read the future
- Used for measuring how well your algorithm performs

Optimal Page Replacement



Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that has not been used in

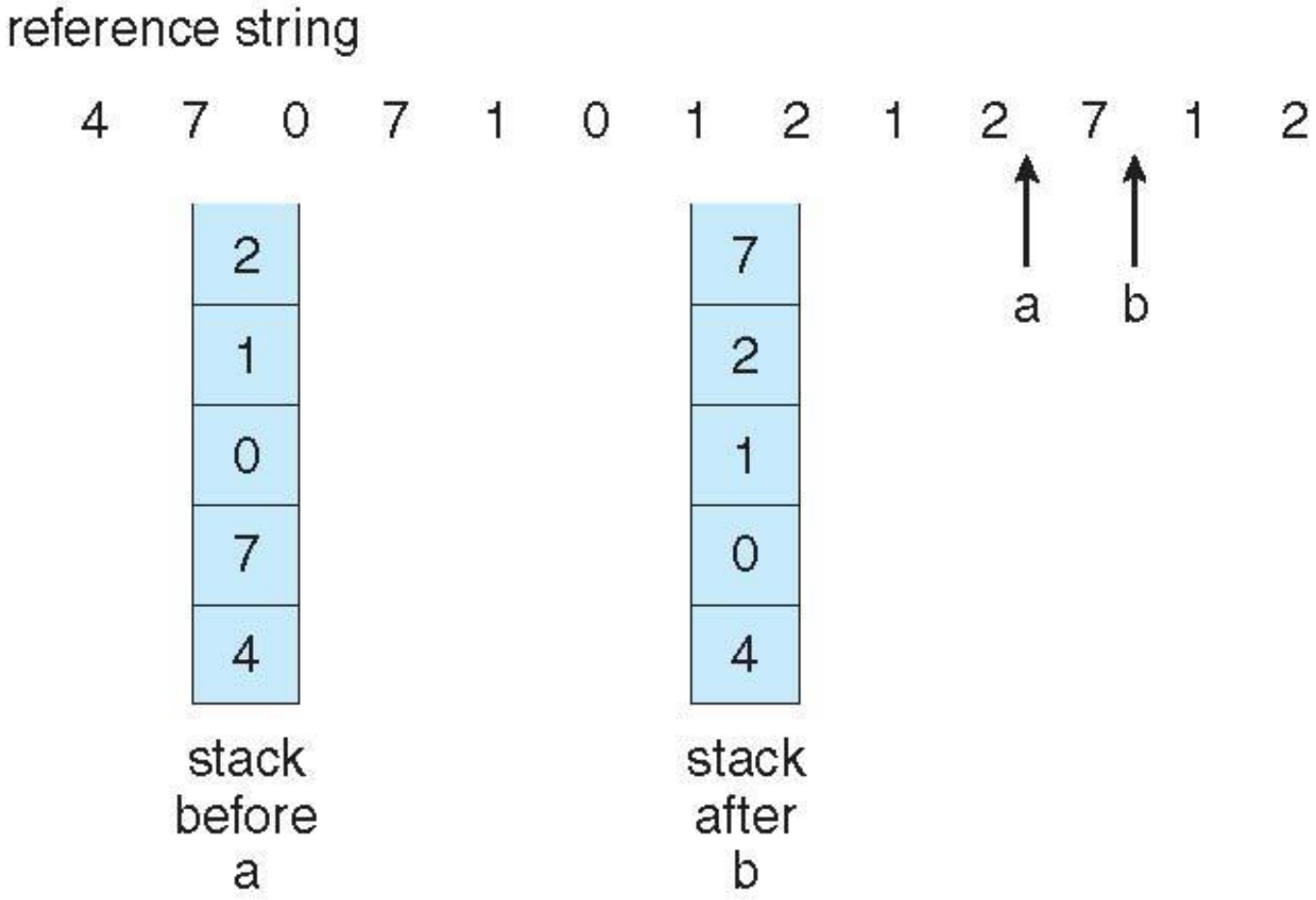


LRU Algorithm (Cont.)

- Counter implementation
  - Every page entry has a counter; every time page is used through this entry, increment 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 several pointers to be changed
  - But each update more expensive
  - No search for replacement

Use Of A Stack to Record The Most Recent Page References



Example 2

- Given the following sequence of page numbers and Given page reference string: 1,2,3,4,2,1,5,6,2,1,2,3,7,6,3,2,1,2,3,6
- Compare the number of page faults for LRU, FIFO and Optimal page replacement algorithm.
- Number of free frames is 3
-
-
-

FIFO

1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6	
1	1	1	4	4	4	4	6	6	6	6	3	3	3	3	2	2	2	2	6	
	2	2	2	2	1	1	1	2	2	2	2	7	7	7	7	1	1	1	1	
		3	3	3	3	5	5	5	1	1	1	1	6	6	6	6	6	3	3	

LRU :

1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
1	1	1	4	4	4	5	5	5	1	1	1	7	7	7	2	2	2	2	2
	2	2	2	2	2	2	6	6	6	6	3	3	3	3	3	3	3	3	3
		3	3	3	1	1	1	2	2	2	2	2	6	6	6	1	1	1	6