OPERATING SYSTEMS-1
CS 241, SPRING 2020
VIRTUAL MEMORY: Lec 6

Ass. Prof. Ghada Ahmed
ghada_khoriba@fci.helwan.edu.eg

Edited slides,
main reference

Operating system concepts, 10th edition, by abraham silberschatz, greg gagne, peter B. Galvin

BACKGROUND

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

LOGICAL VS. PHYSICAL

- **Logical address** – generated by the CPU; also referred to as **virtual address**
- **Physical address** – address seen by the memory unit

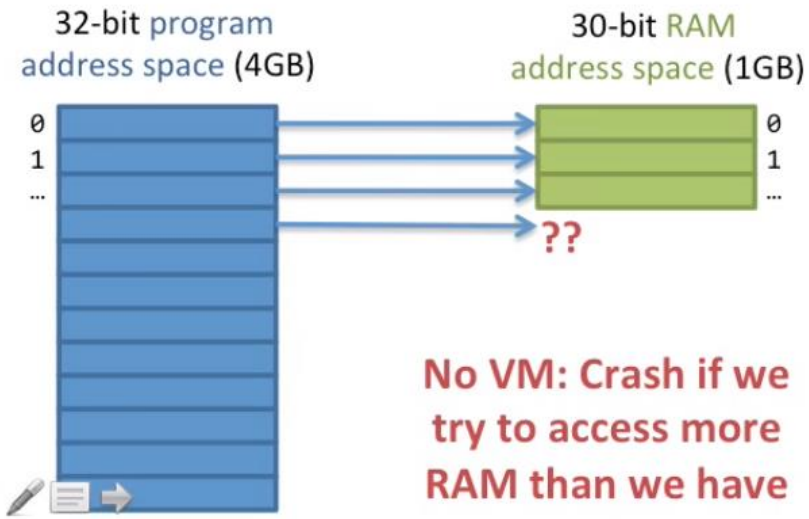
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program

Main Goal:

Simultaneously keep several processes in memory to facilitate multiprogramming

Without Virtual Memory

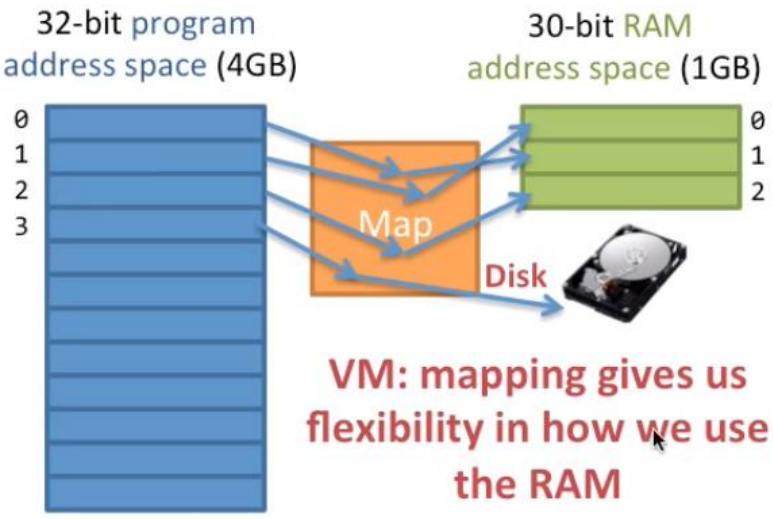
Program Address = RAM Address



No VM: Crash if we try to access more RAM than we have

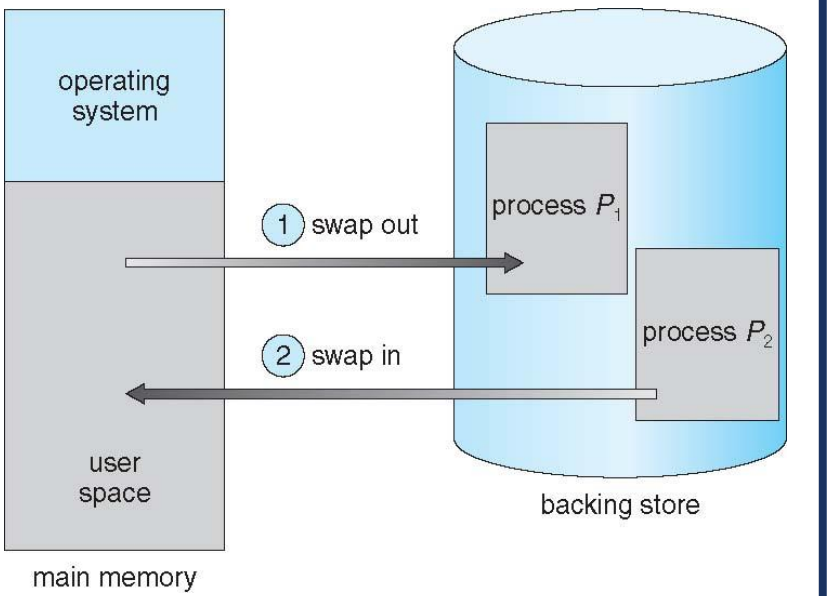
With Virtual Memory

Program Address Maps to RAM Address



VM: mapping gives us flexibility in how we use the RAM

TERMINOLOGIES



- **Memory Management Unit (MMU):** Hardware device that at run time maps virtual to physical address

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

SWAPPING ON MOBILE SYSTEMS

Not typically supported

- Flash memory based
  - Small amount of space
  - Limited number of write cycles
  - Poor throughput between flash memory and CPU on mobile platform

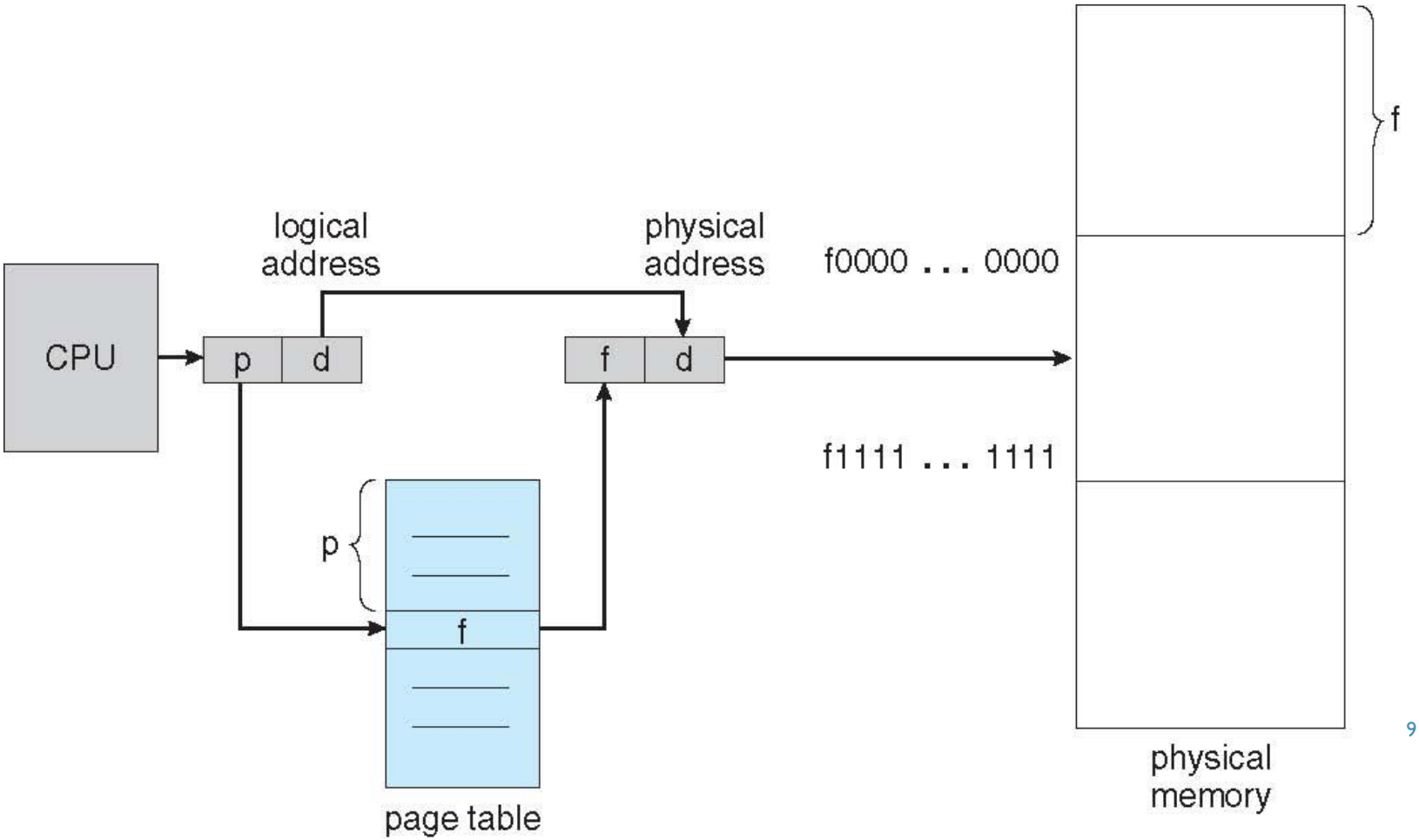
Instead use other methods to free memory if low

- **iOS *asks*** apps to voluntarily relinquish allocated memory
  - Read-only data thrown out and reloaded from flash if needed
  - Failure to free can result in termination
- **Android** terminates apps if low free memory, but first writes **application state** to flash for fast restart
- Both OSes support paging

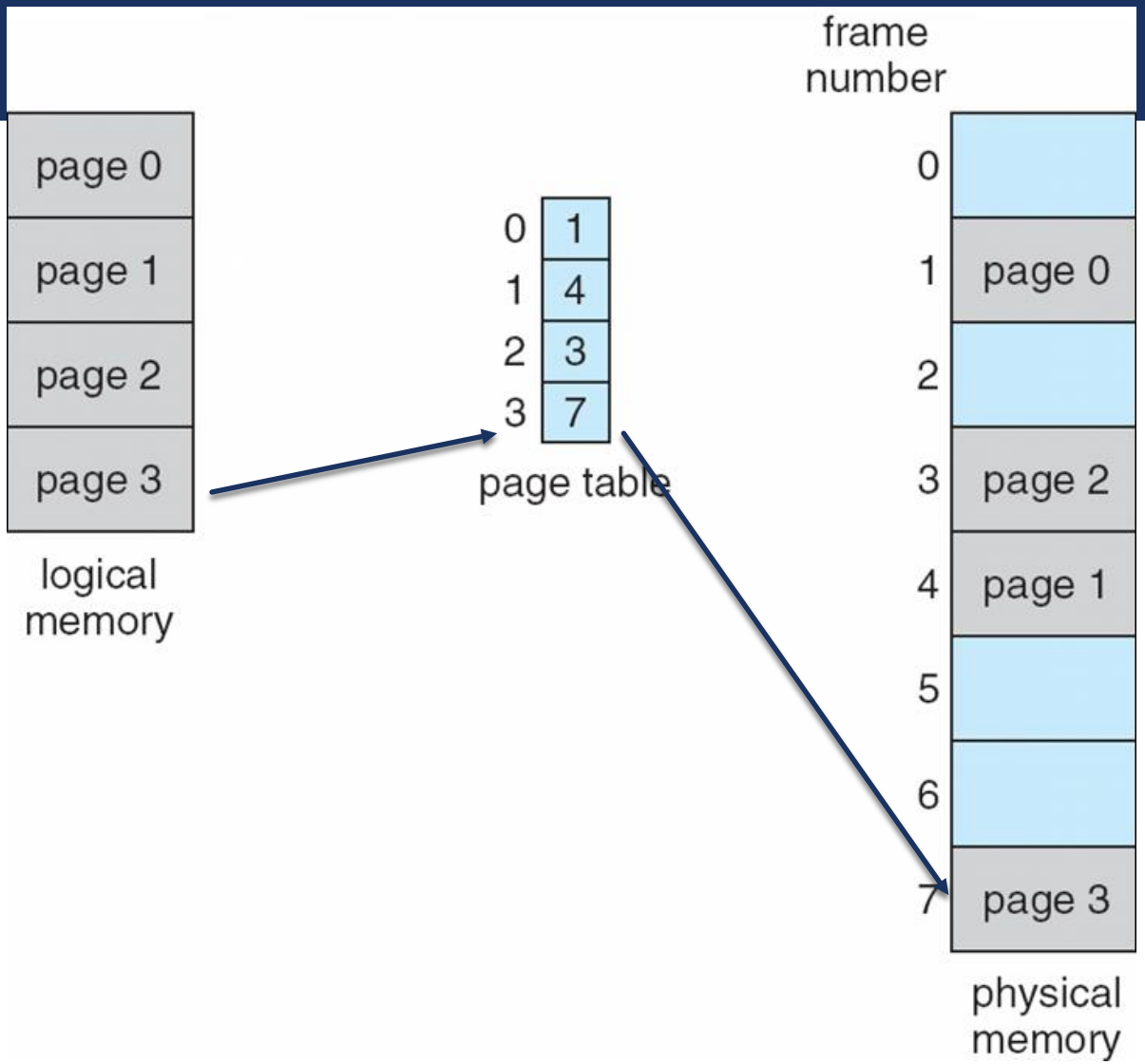
PAGING

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

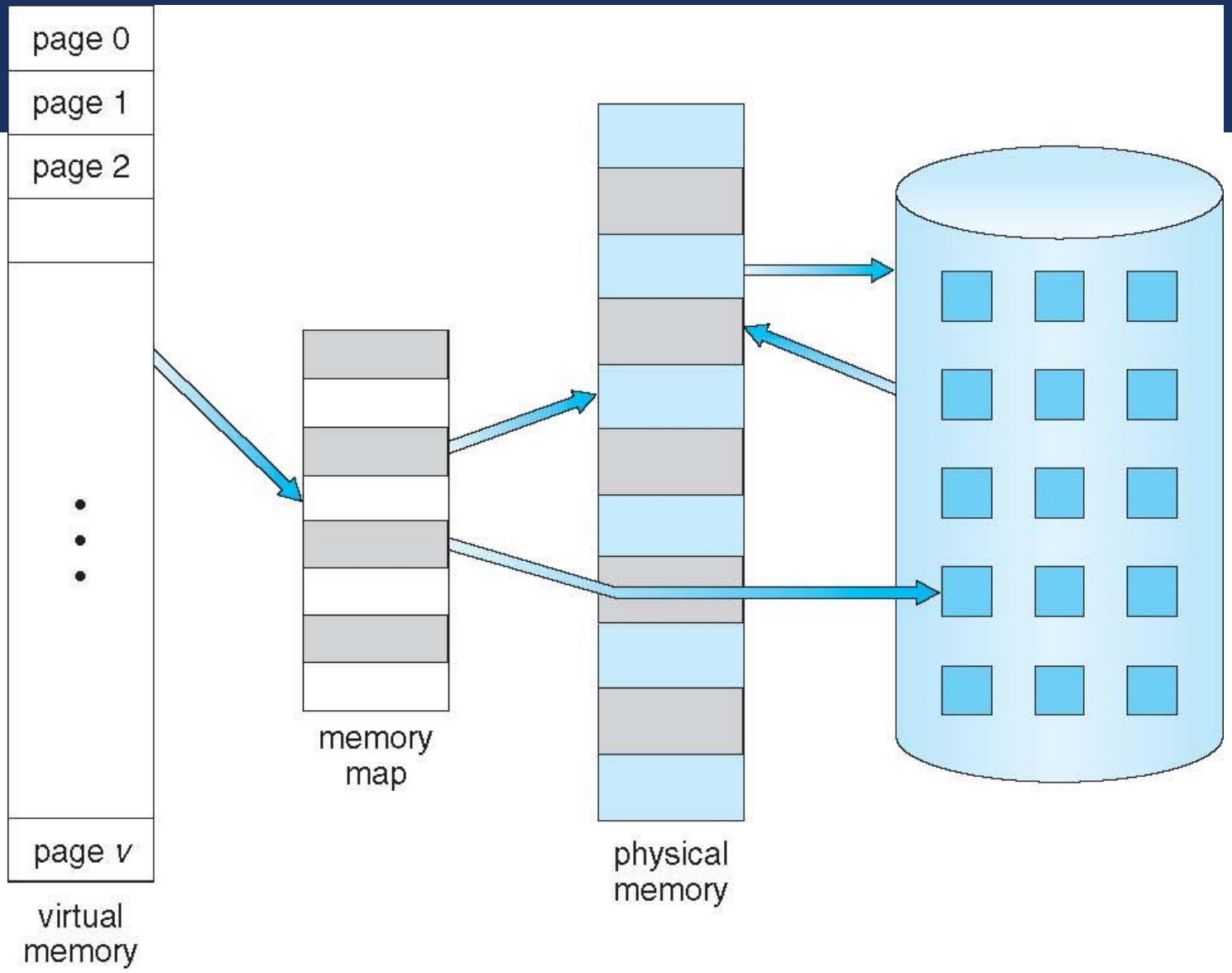
PAGING HARDWARE



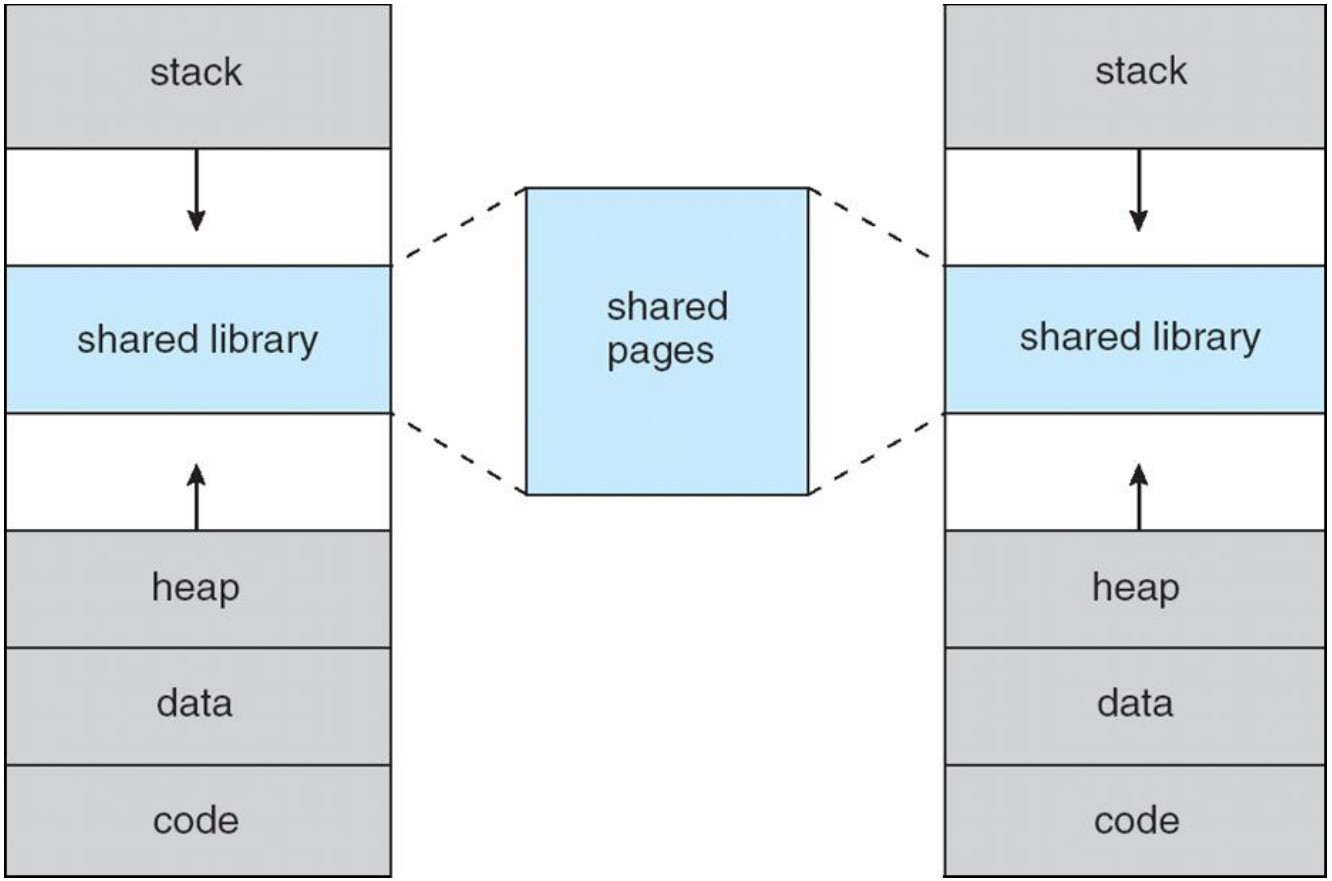
PAGING MODEL OF LOGICAL AND PHYSICAL MEMORY



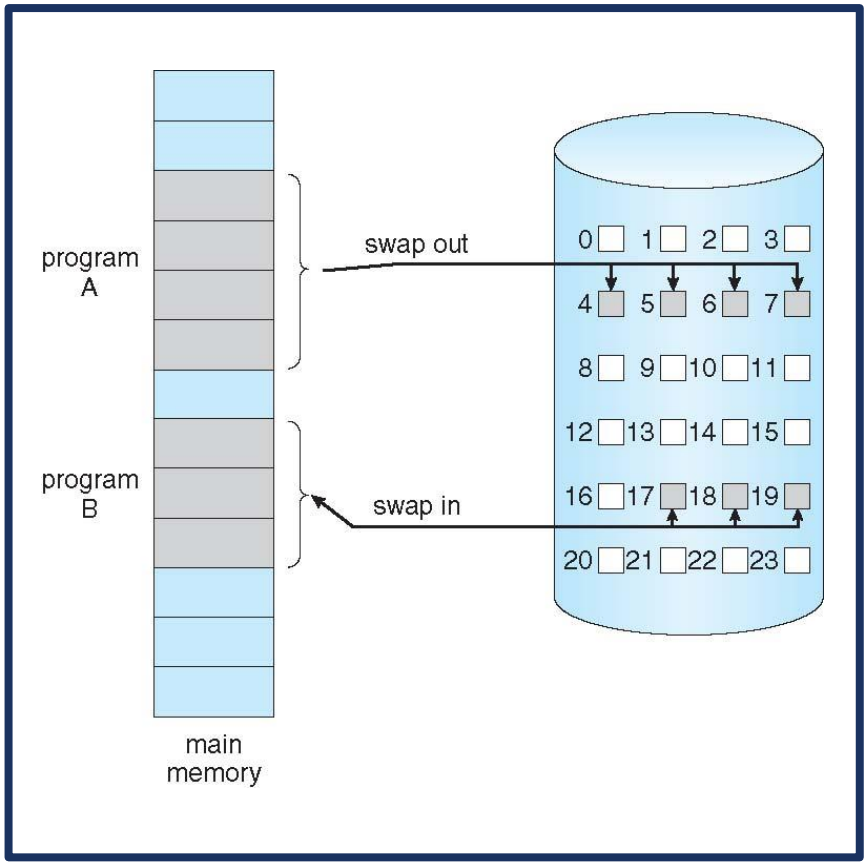
VIRTUAL MEMORY THAT IS LARGER THAN PHYSICAL MEMORY



SHARED LIBRARY USING VIRTUAL MEMORY



DEMAND PAGING



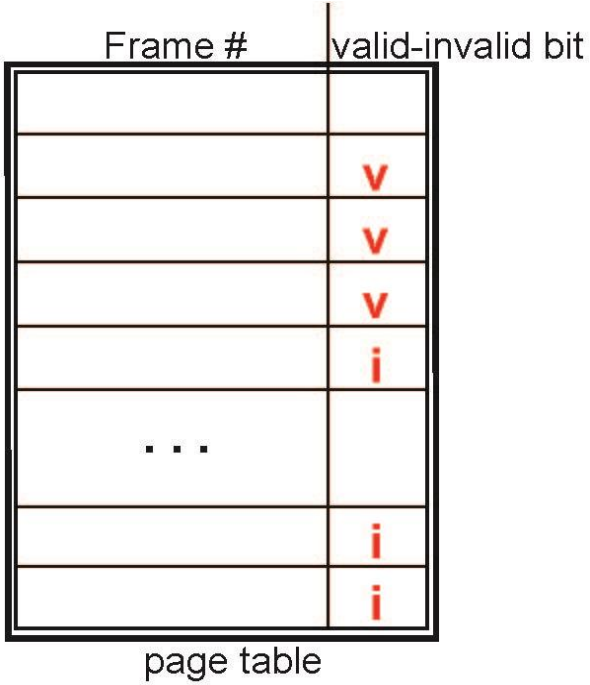
- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Similar to paging system with swapping
- Page is needed ⇒ reference to it
  - invalid reference ⇒ 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**

BASIC CONCEPTS

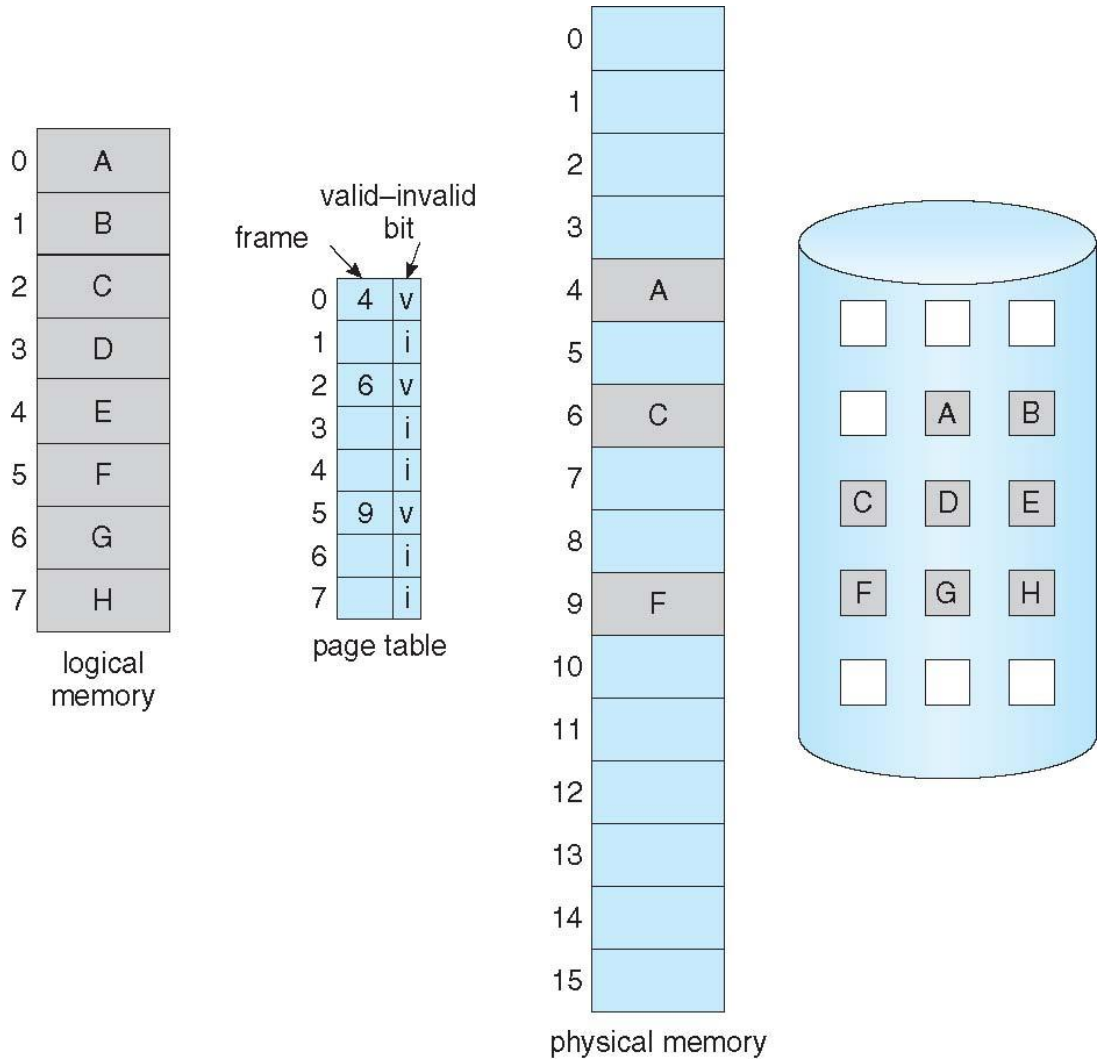
- With swapping, pager guesses which pages will be used before swapping out again
- Instead, pager brings in only those pages into memory
- How to determine that set of pages?
  - Need new MMU functionality to implement demand paging
- If pages needed are already **memory resident**
  - No difference from non demand-paging
- If page needed and not memory resident
  - Need to detect and load the page into memory from storage
    - Without changing program behavior
    - Without programmer needing to change code

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:
- During MMU address translation, if valid–invalid bit in page table entry is **i** ⇒ **page fault**



Page Table When Some Pages Are Not in Main Memory



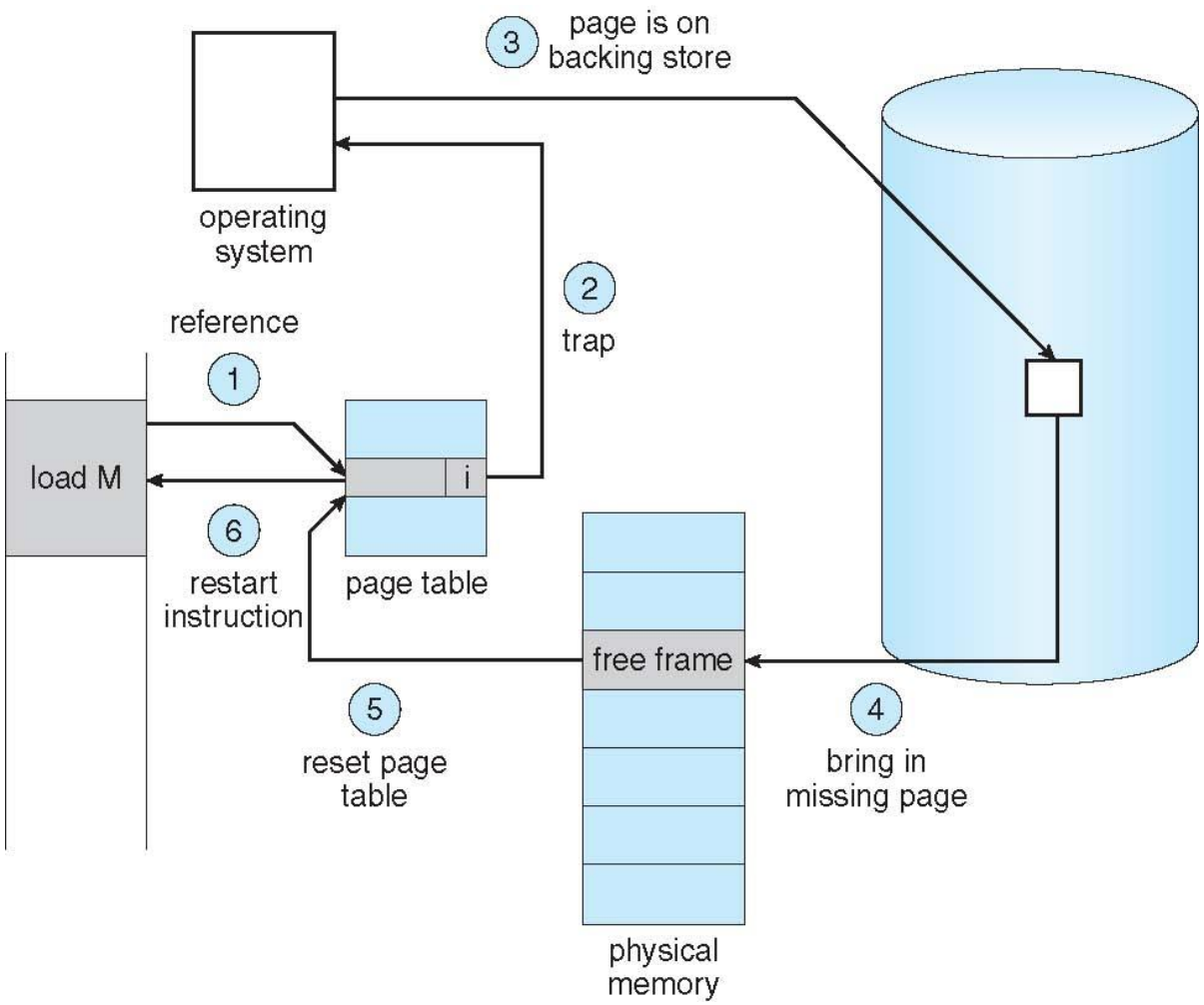
PAGE FAULT

- 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 ⇒ abort
   - Just not in memory
2. Find free 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



PERFORMANCE OF DEMAND PAGING

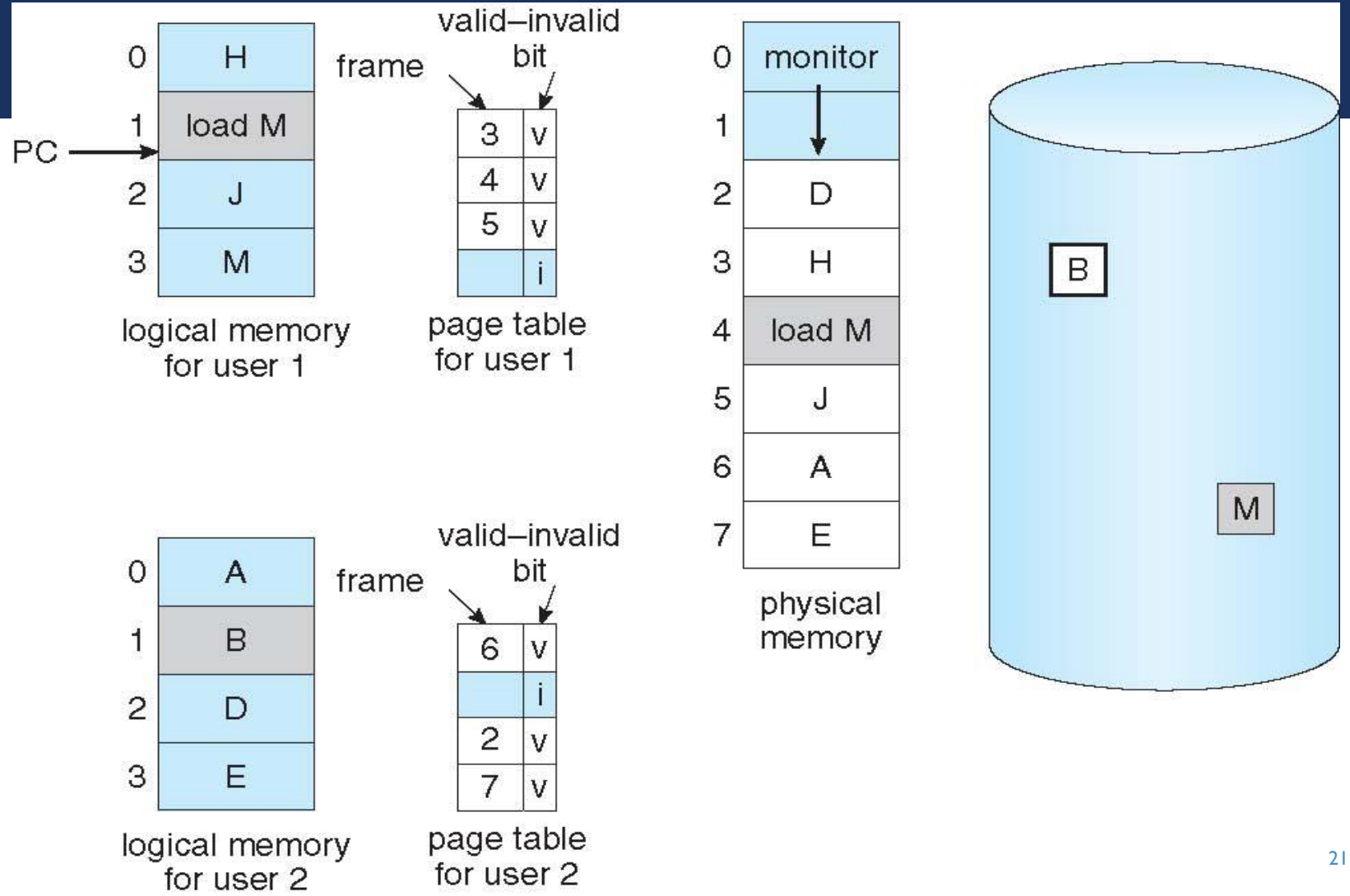
- Three major activities
  - Service the interrupt – careful coding means just several hundred instructions needed
  - Read the page – lots of time
  - Restart the process – again just a small amount of time
- Page Fault Rate $0 \leq p \leq 1$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- **Effective Access Time (EAT)**

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

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

NEED FOR PAGE REPLACEMENT



BASIC PAGE REPLACEMENT

1. Find the location of the desired page on disk

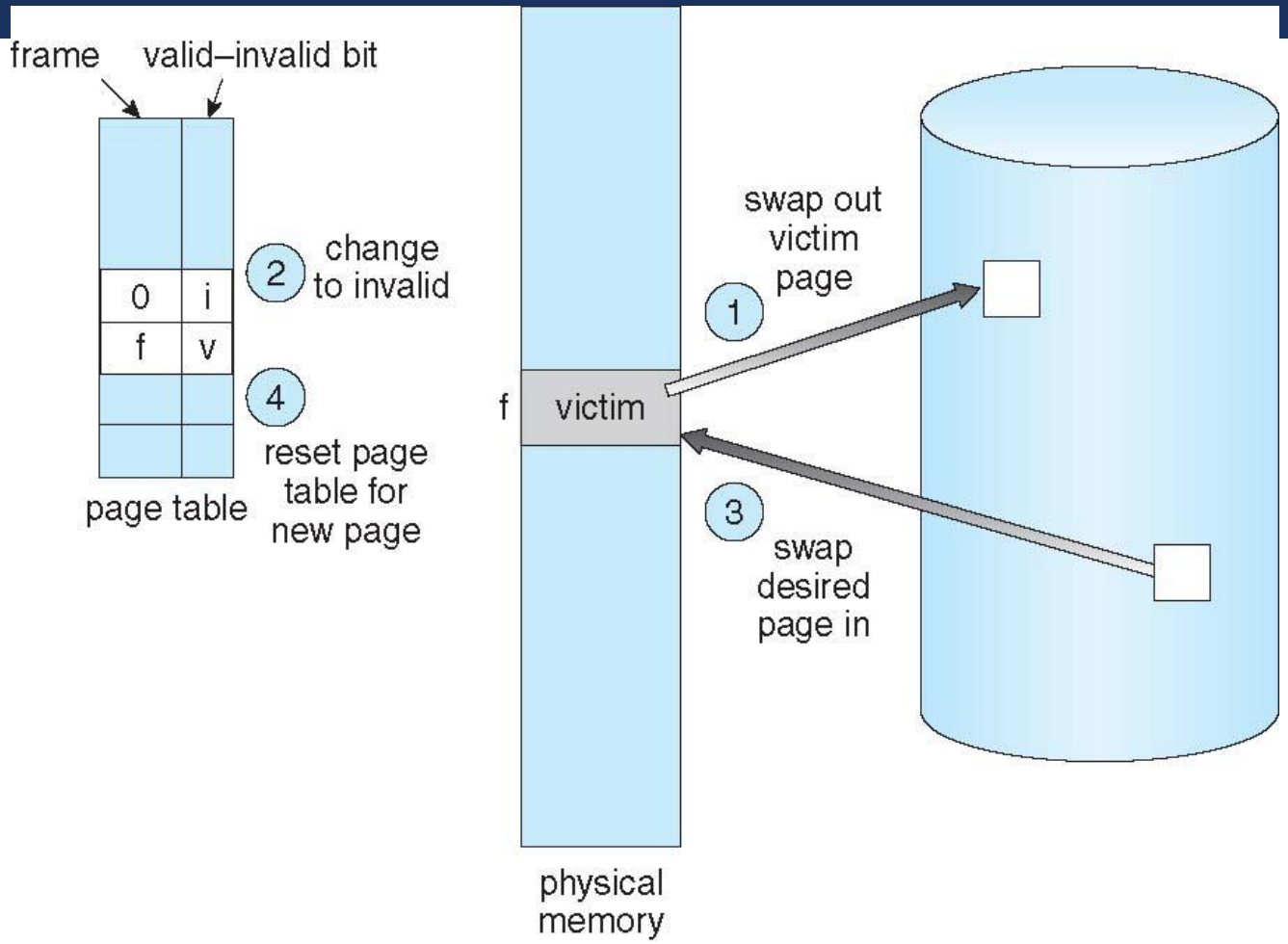
2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim frame**
   - Write victim frame to disk if dirty

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

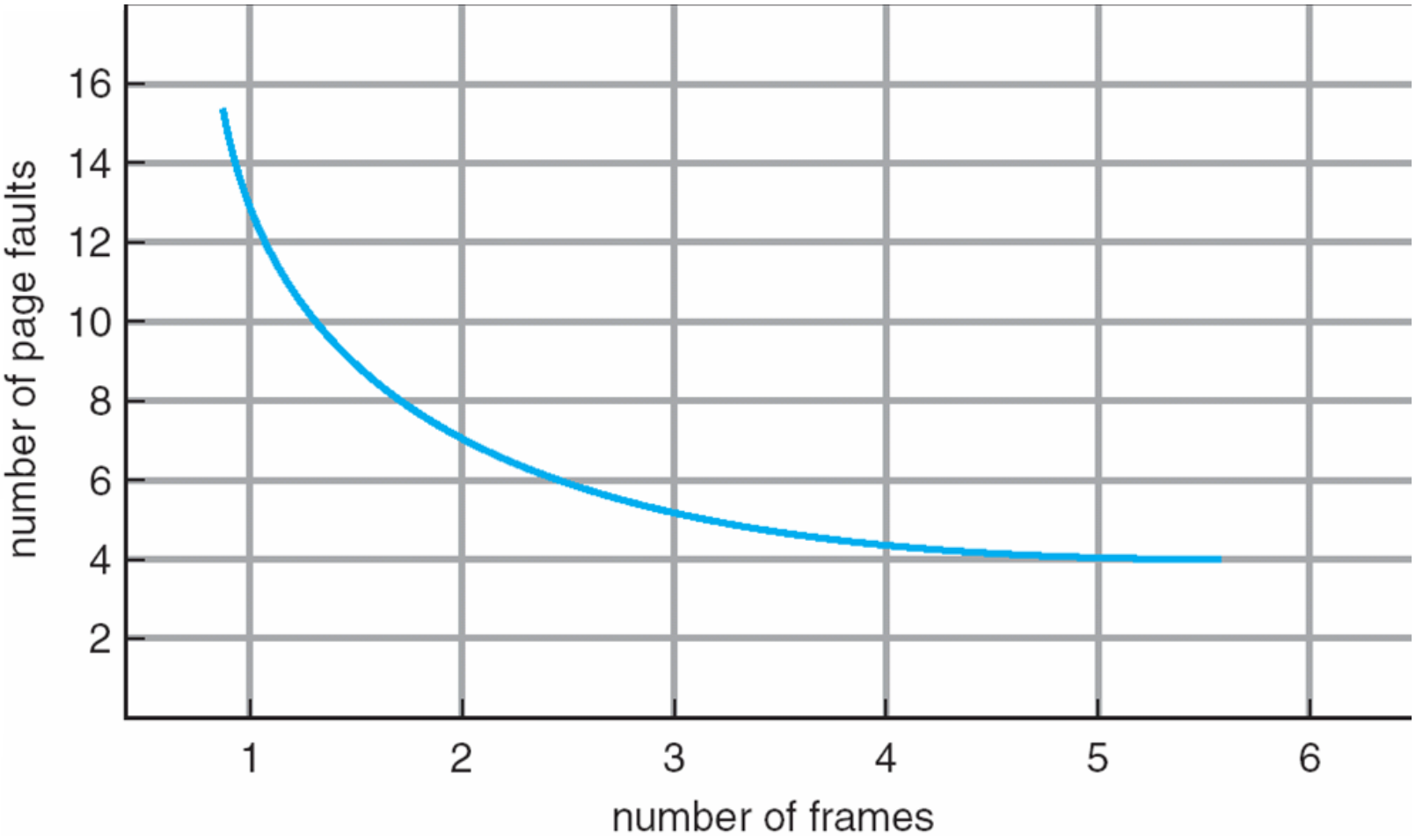


PAGE AND FRAME REPLACEMENT ALGORITHMS

- **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
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
  - Results depend on number of frames available
- In all our examples, the **reference string** of referenced page numbers is

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

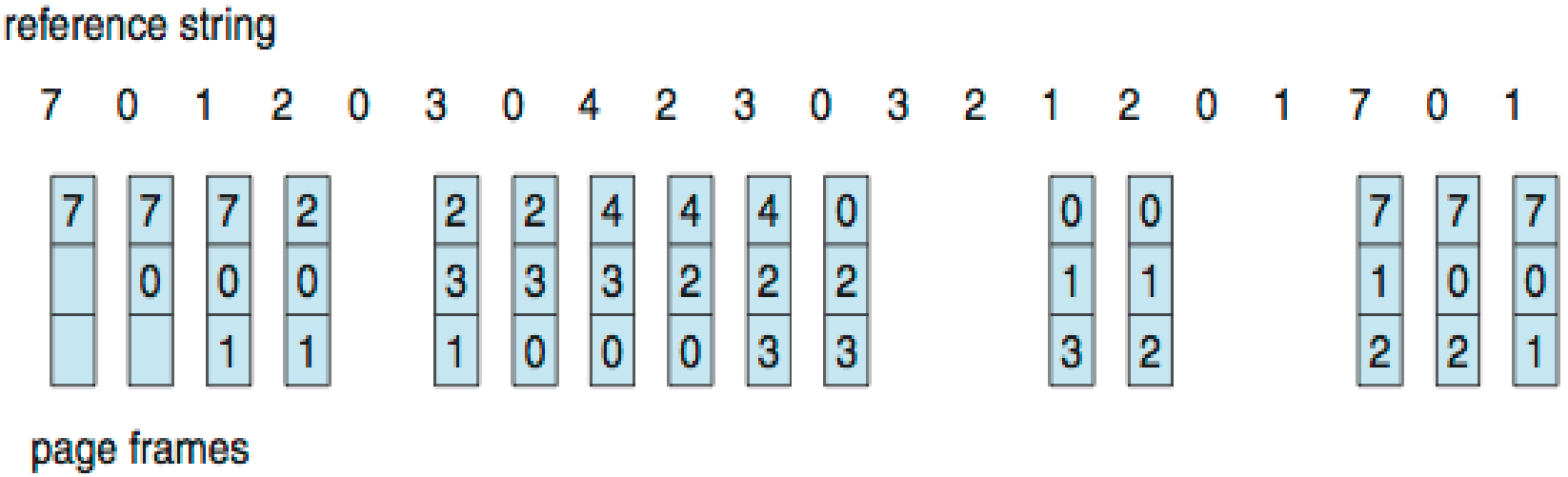
GRAPH OF PAGE FAULTS VERSUS THE NUMBER OF FRAMES



FIRST-IN-FIRST-OUT (FIFO) ALGORITHM

15 page faults

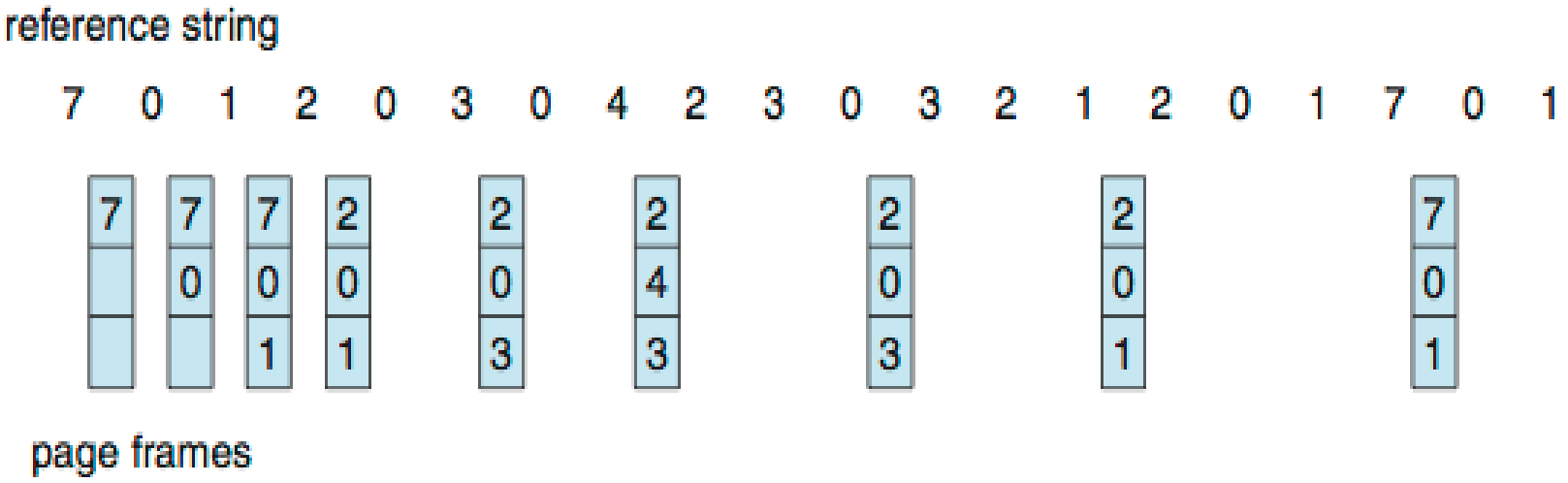
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



- How to track ages of pages?
  - Just use a FIFO queue

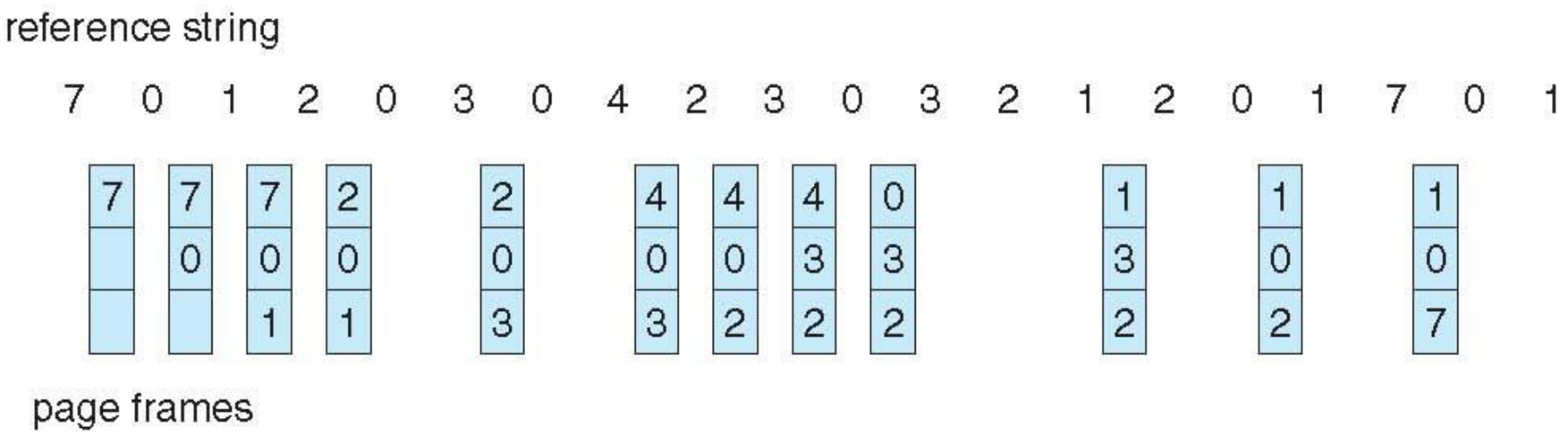
OPTIMAL ALGORITHM

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



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

COUNTING ALGORITHMS

- Keep a **counter of the number of references** that have been made to each page
  - Not common
- **Lease Frequently Used** (**LFU**) **Algorithm**: replaces page with smallest count
- **Most Frequently Used** (**MFU**) **Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

APPLICATIONS AND PAGE REPLACEMENT

- All of these algorithms have OS guessing about future page access
- Some applications have better knowledge – i.e. databases
- Memory intensive applications can cause double buffering
  - OS keeps copy of page in memory as I/O buffer
  - Application keeps page in memory for its own work
- Operating system can given direct access to the disk, getting out of the way of the applications
  - **Raw disk** mode
- Bypasses buffering, locking, etc

PROGRAM STRUCTURE && PAGE FAULT

- Program structure
  - `int[128,128] data;`
  - Each row is stored in one page
  - Program 1

```
for (j = 0; j <128; j++)
    for (i = 0; i < 128; i++)
        data[i,j] = 0;
```

128 x 128 = 16,384 page faults

  - Program 2

```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i,j] = 0;
```

128 page faults

R U OK?

Consider the following page reference string:

1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6.

How many page faults would occur for the following replacement algorithms, assuming one, two, three, four, five, six, and seven frames? Remember that all frames are initially empty, so your first unique pages will cost one fault each.

- LRU replacement
- FIFO replacement
- Optimal replacement

ANSWER

Number of frames	LRU	FIFO	Optimal
1	20	20	20
2	18	18	15
3	15	16	11
4	10	14	8
5	8	10	7
6	7	10	7
7	7	7	7