Virtual Memory

Virtual Memory

- **Logical memory** allowed us to do two things when multiple programs are concurrently loaded into the memory
  - Protect one program from the other program ⇒ prevent one program from accidentally or maliciously manipulating memory held by the other program ⇒ known as **memory protection**
  - Relocate one program or portions of it in memory efficiently (at execution time) while retaining memory protection ⇒ known as memory relocation
- What else could we need from memory management? We want to **overload** programs into memory ⇒ load more programs than what the physical memory would allow us to do

Memory Overloading

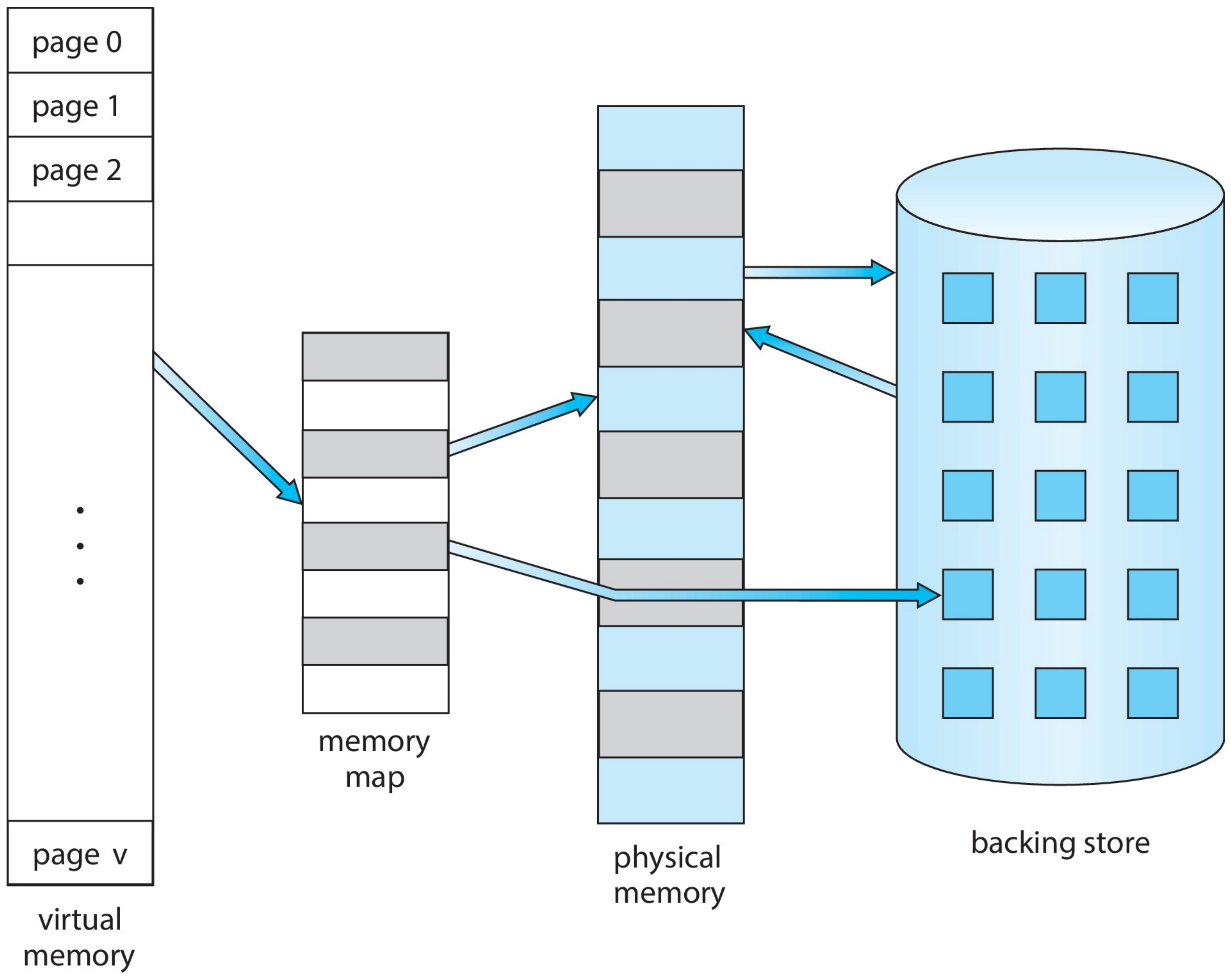
- How can we do overloading?
- We did overloading with overlays $\Rightarrow$ program was split into portions and not all portions were brought into the memory at once. We loaded portions of the program as they became needed
- Key observation: programs tend to have locality $\Rightarrow$ spend lot of time in a specific portion of the code and not much elsewhere $\Rightarrow$ there are highly used functions, loops, etc. Loading those heavily used portions is good to run the program in most cases.

Virtual Memory: Basic Idea

- **Virtual memory**: separation of logical memory from physical memory
  - Only part of the program needs to be in the memory for execution (similar to overlays - but handled by OS)
  - Logical address space can be much larger than the physical memory space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation - don’t need to load the whole process at once - incremental loading is allowed
  - More programs running concurrently
  - Less IO or swap needed to load or swap processes

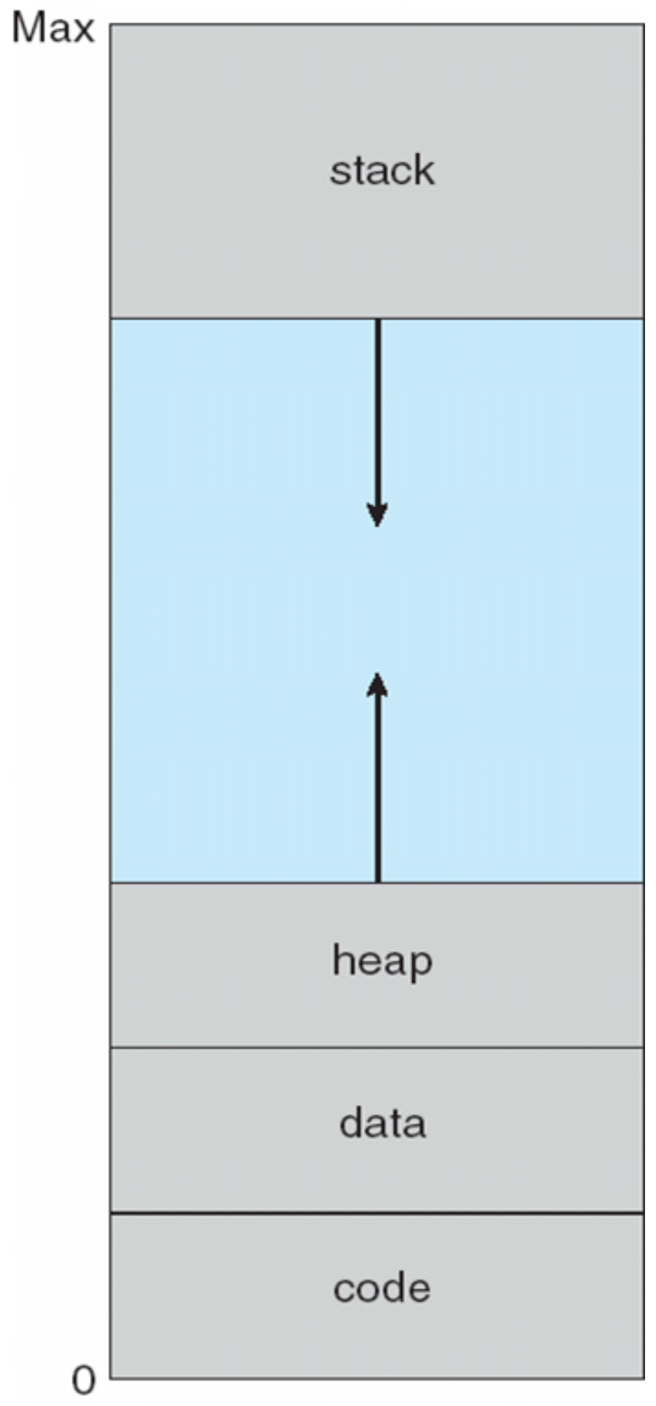
Virtual Memory: Basic Idea

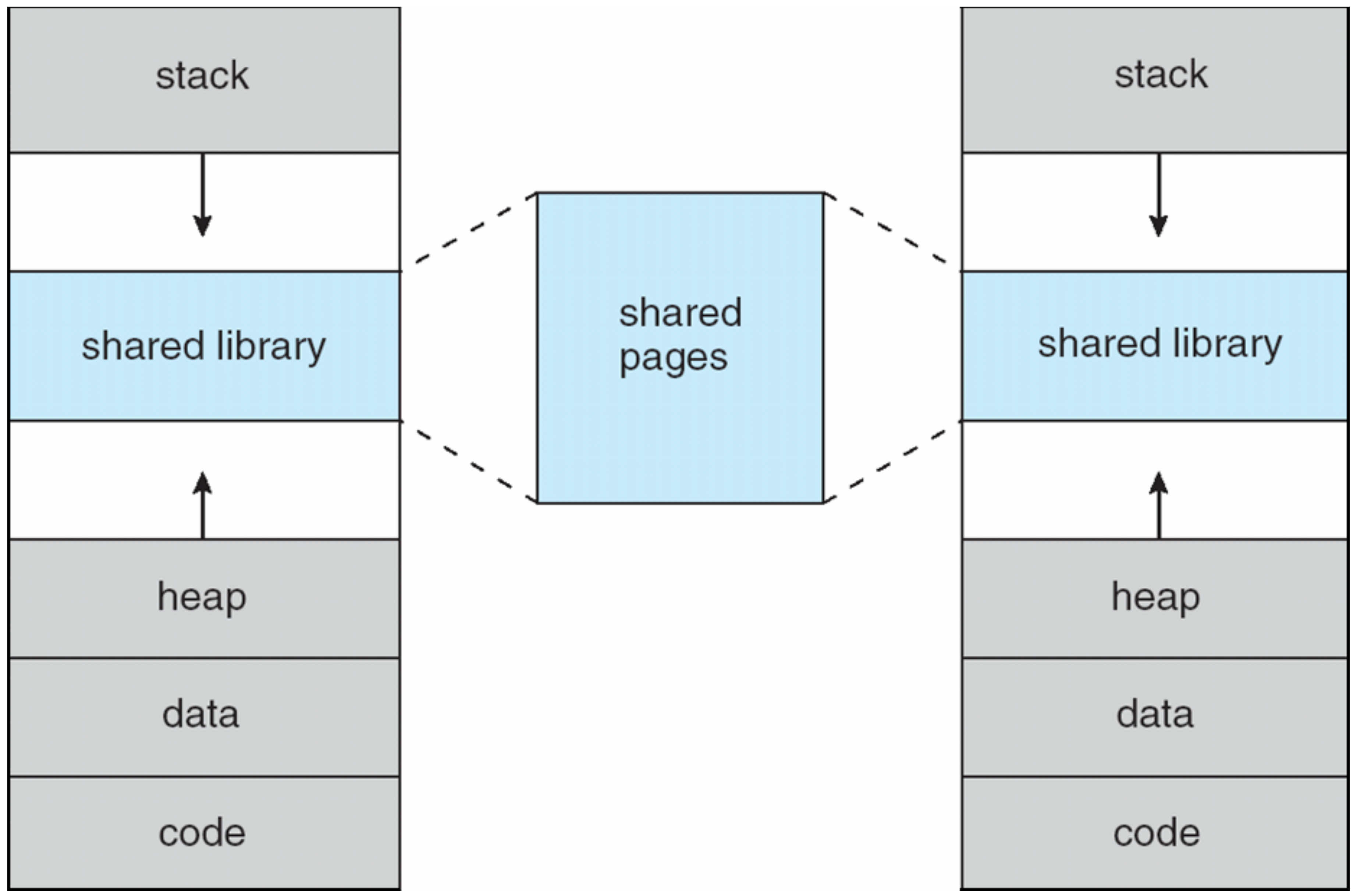
- Logical (virtual) memory is split into pages
- Frames corresponding to pages in logical memory can be in:
  - Physical memory
  - Backing store
  - Not allocated
- It takes time to bring pages from backing store to memory - process need to wait and then resume afterwards



Virtual Memory of a Process

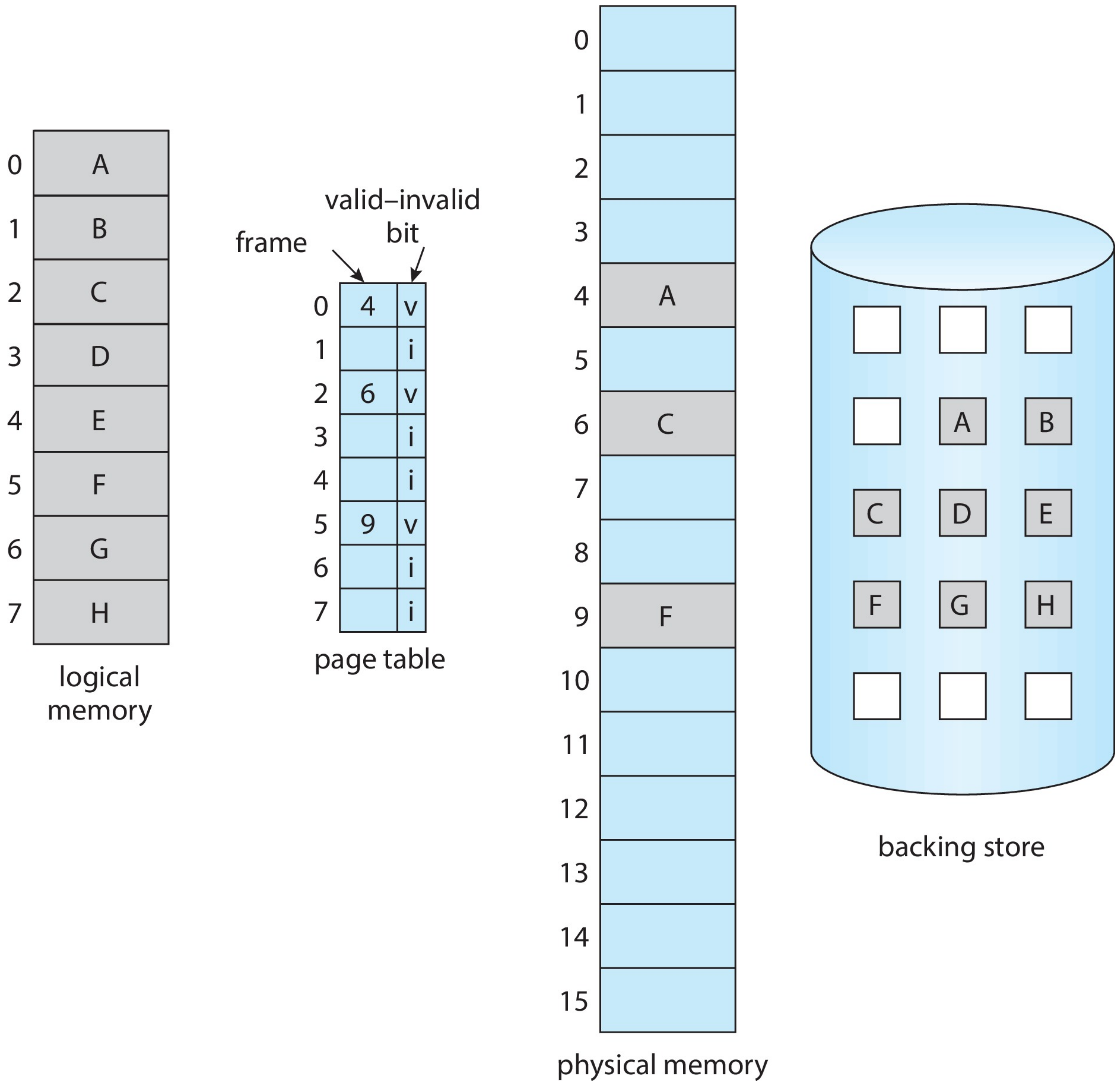
- A process' memory is quite sparse - heap & stack grow and shrink. Initial memory allocation is going to change as the segments vary in size.
- Virtual memory also helps in sharing libraries among different processes — a library is mapped read-only to multiple processes
- Processes' can also share memory with each other — use such shared memory to share input data and results





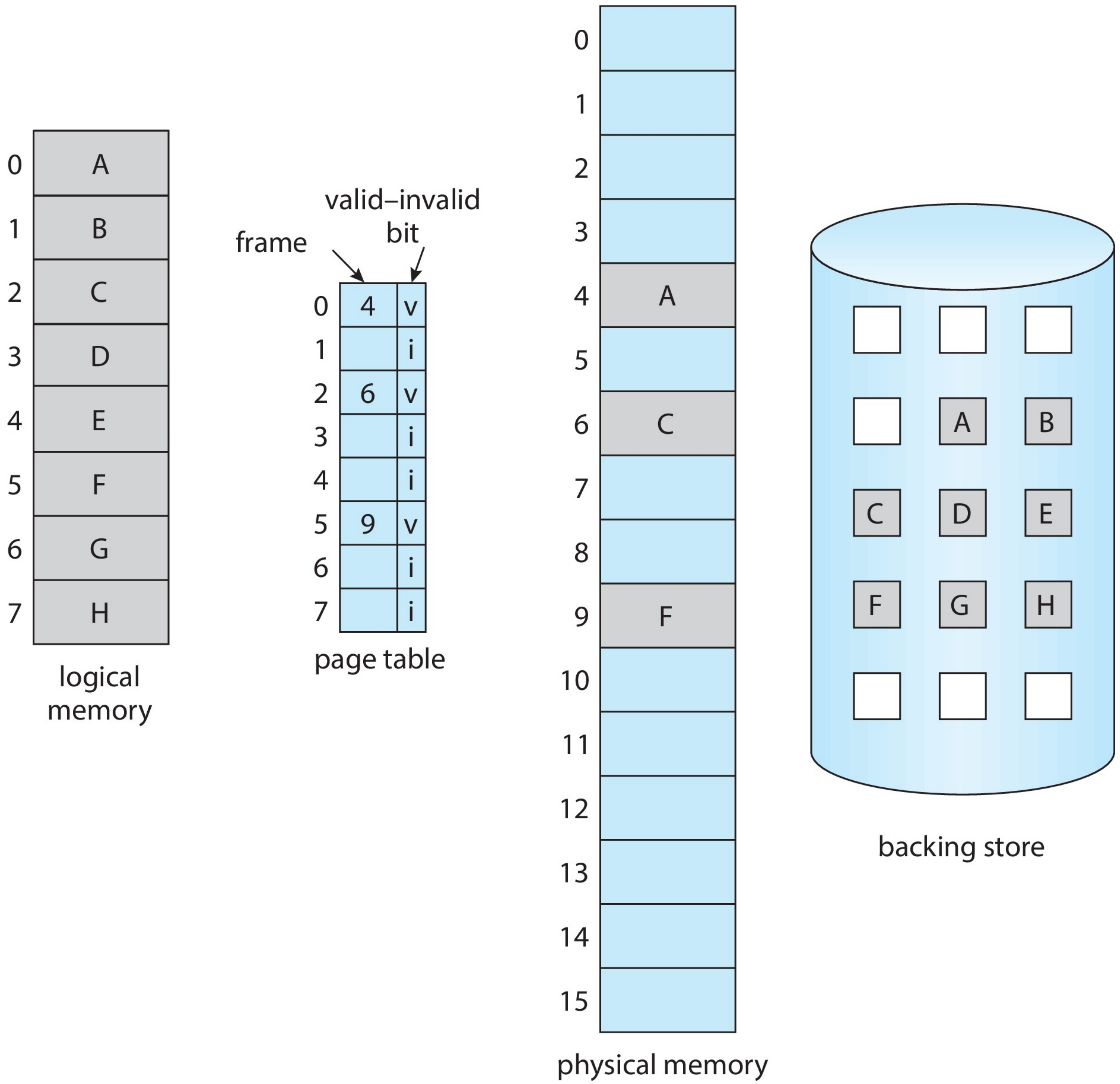
When to Bring into Physical Memory?

- Process can use A—H pages. At a given time (shown) we have loaded A, C, F
- We would have started the process with no pages and let it ask for A, then load it, so on.
- Wait for process to ask for a page is called — **demand paging (pure - zero initial pages)**
- Pages not needed are not loaded into memory
- Program waits while a page is being loaded — **page fault** processing overhead



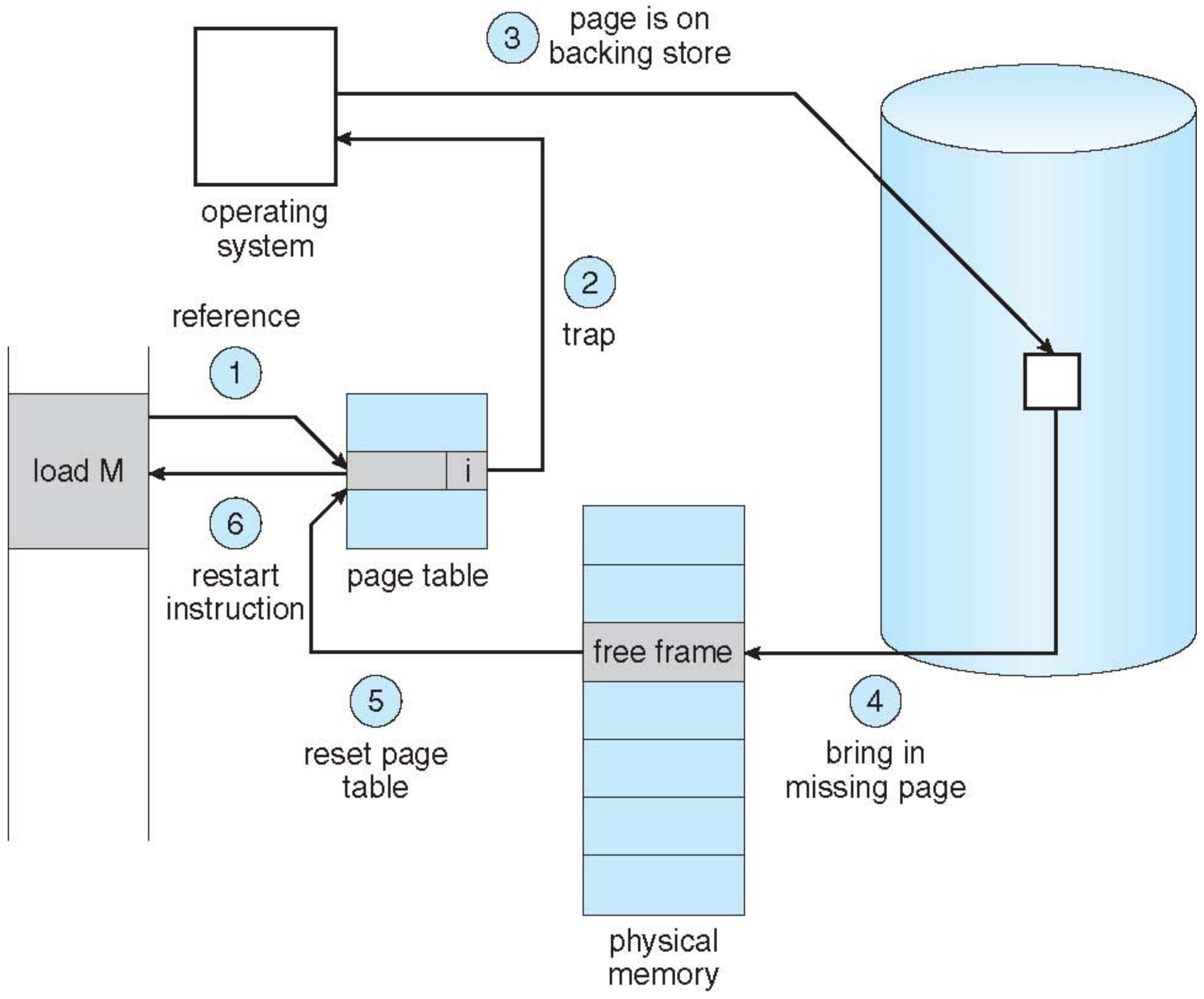
Demand Paging

- Demand paging has an **valid/invalid bit** in the page table
- Bit is valid ⇒ page is already loaded, just proceed with the access
- Bit is invalid ⇒ page is not loaded — either in backing store or unallocated in the logical address space)
- Initially all entries have invalid bit set
- Process accesses an invalid page, we have a page fault — trap into the OS to load the page



Page Fault Process for Demand Paging

1. Check if the reference is invalid/valid in the page table
2. If invalid, we suspend the process and trap into OS
3. OS Does page fault processing and pull page from backing store
4. Find a free page frame and load the page from backing store
5. Change invalid to valid in the page table
6. Restart the faulting process — by resuming from the point of suspension



Demand Paging Challenges

- Demand paging needs us to restart the execution of an instruction that caused the page fault
- Some instructions can have problems restarting due to partial executions
- For example, DBNZ R1, #567733 — decrement register and non zero branch to the given address
- When the fault occurs, we might have already decremented once
- By rerunning it again from the beginning, we would decrement it again — to avoid this problem, we need to resume from the partially executed state of the instruction

Free Frames

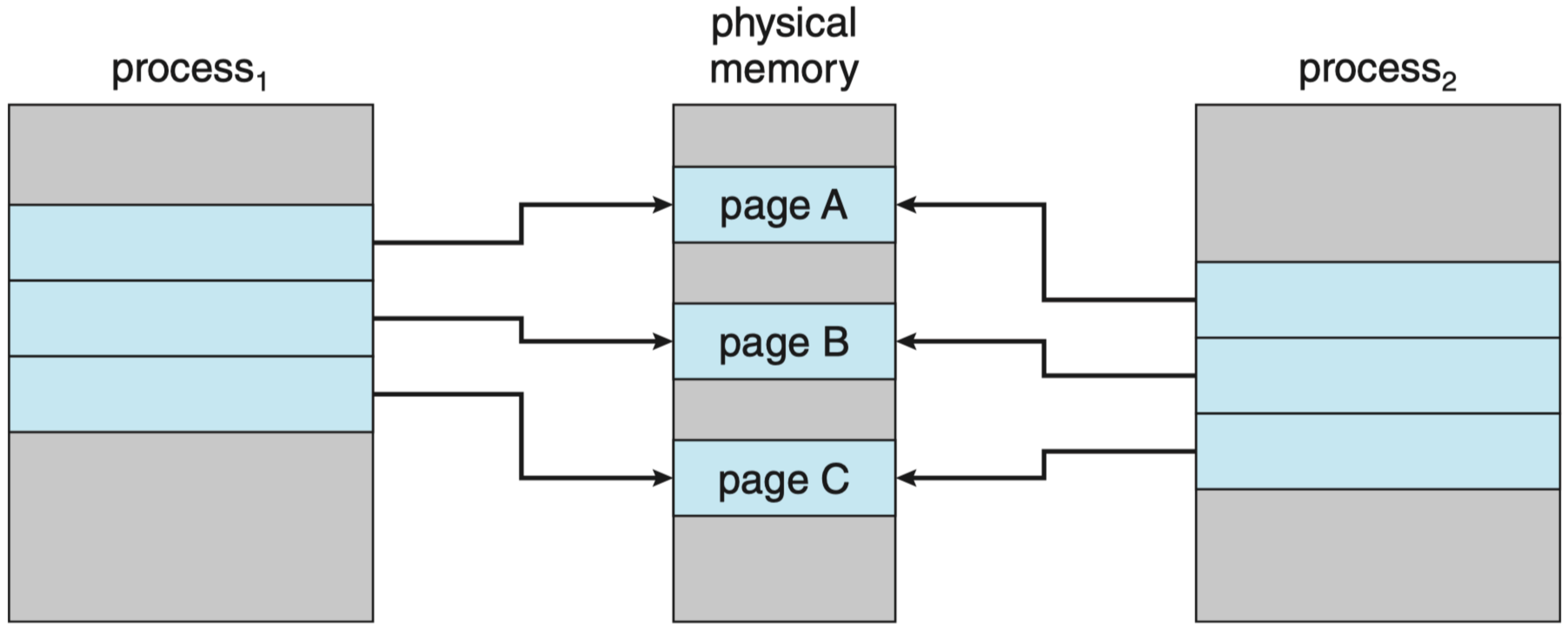
- At page fault, we bring in the missing frame from backing store
- Where do we put that in the memory? We need free frames
- OS maintains a free-frame list, a pool of free frames
- OS uses a zero-fill-on-demand to zero out the page content before reusing a page - for security purposes - this way a process would not see data from a previous process

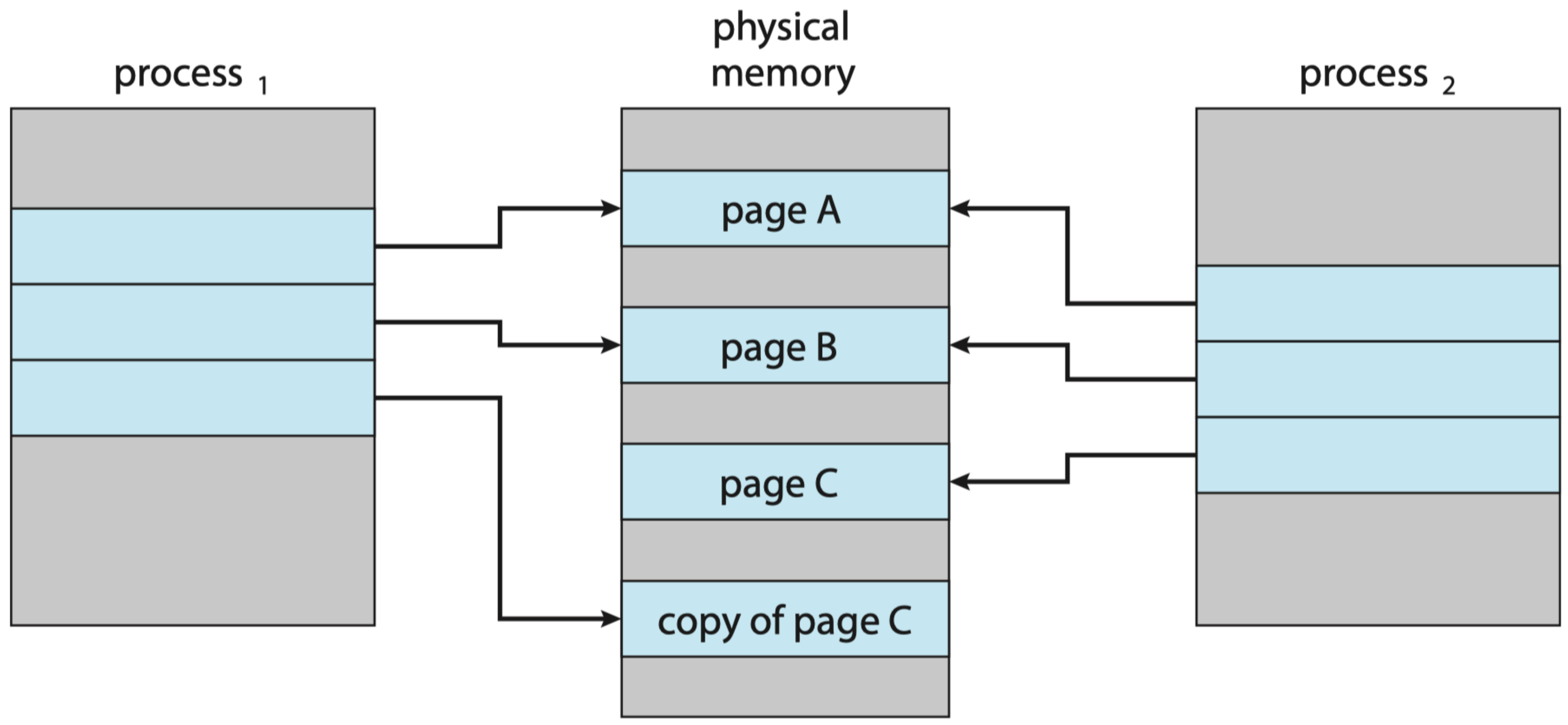
Performance of Demand Paging

- Let memory access time be t_m ns and t_f (in ns) be the page fault processing time and p be the probability of page fault
- Effective access time $(1-p) \times t_m + p \times t_f$ ns
- The page fault processing time is determined by the times for the following (see book for a full list)
  - Trap to the operating system
  - Save faulting process state
  - Issue a read to the backing storage for free frame
  - Get the retrieved frame and setup the page tables
  - Restore the faulting process
- Consider an example: $t_m = 200$ ns, $t_f = 8$ ms. $t_e = (1-p) \times 200 + p \times 8000000 = 200 + 7999800 \times p$
- If one access out of 1000 causes page fault, $p = 1/1000 \Rightarrow t_e = 8.2$ us
- If we want to keep performance degrade to 10% of less, $220 > 200 + 79998000 \times p \Rightarrow p < 0.00000025$

Copy-on-Write

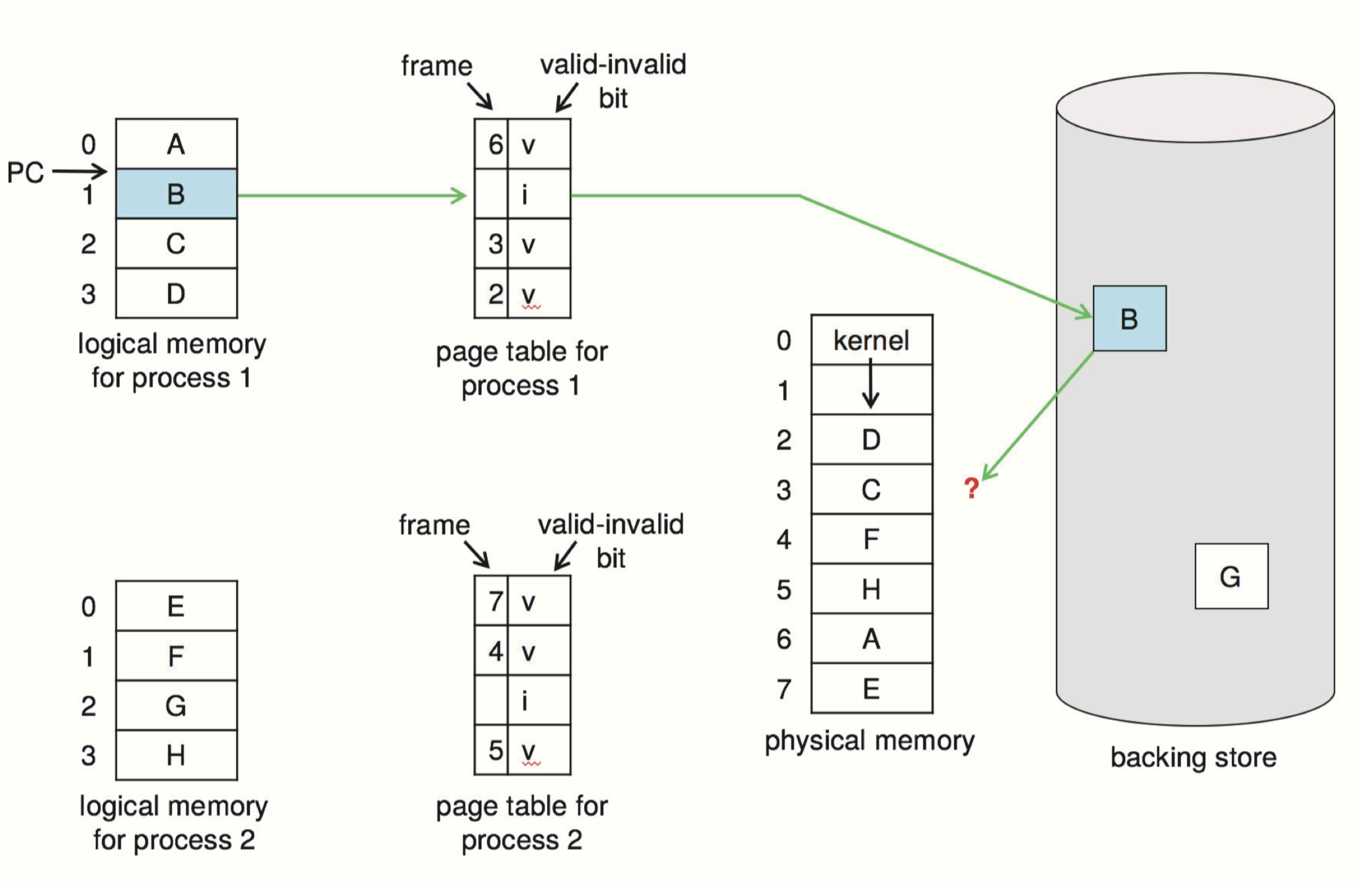
- At fork() Process_2 (child) has a copy of all of Process_1 pages
- The alternate is to use demand paging and let Process_2 request its own pages from scratch
- Process_2 is given a read-only copy of Process_1 pages
- When Process_2 tries to make modifications it makes it's own copy of the shared page
- If no modification, original is shared
- Copy happens if Process_1 tries to write as well
- Copy-on-write is necessary for writeable pages - code pages are not included in CoW





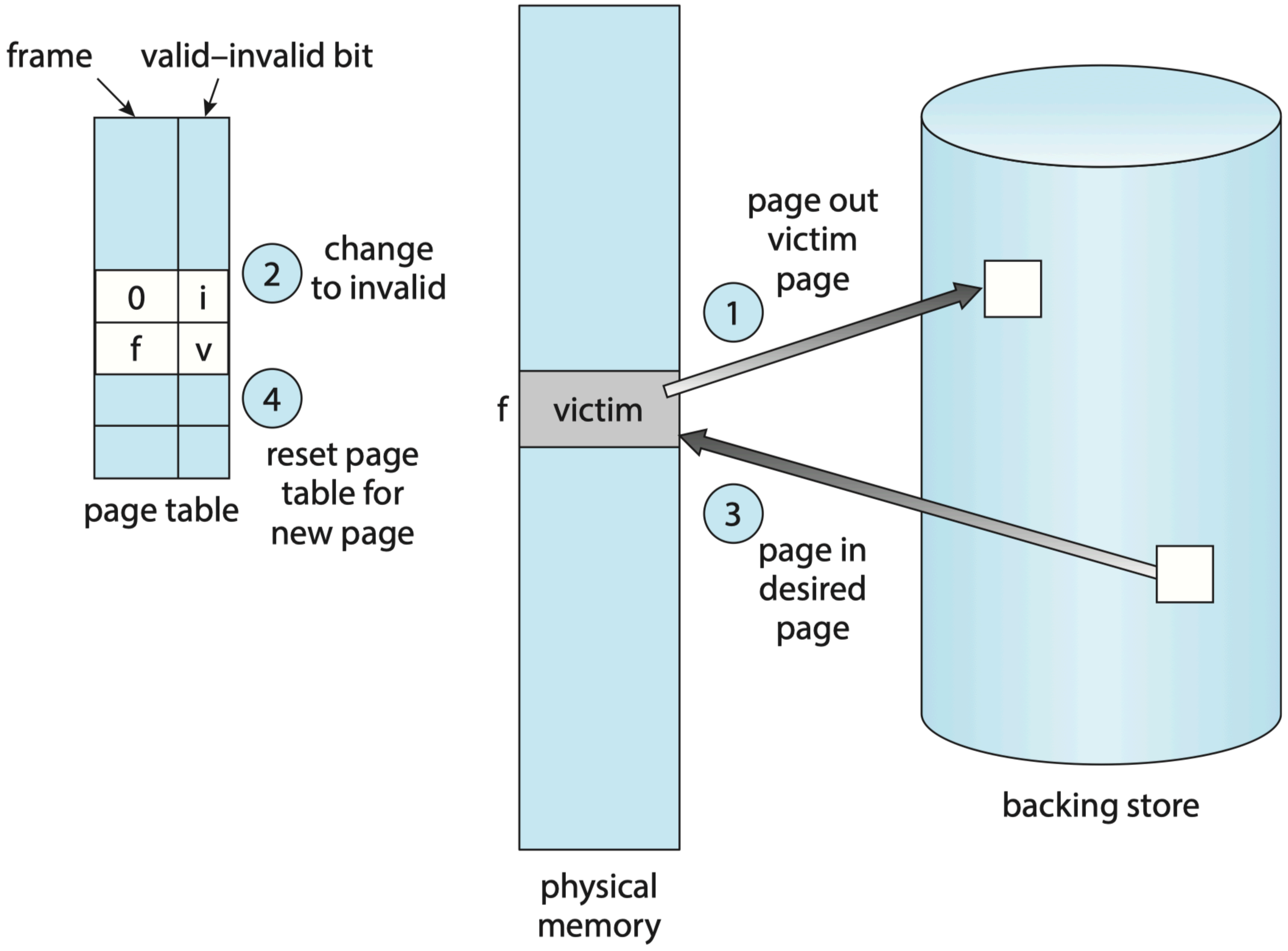
Page Replacement Problem

- Process_1 needs 4 pages and Process_2 needs 4 pages
- There are only 6 page frames in memory for user data after kernel taking 2 page frames
- A, C, D of Process_1 and E, F, H of Process_2 are loaded
- If we want to bring B for Process_1, a loaded page needs to be evicted (copied to backing store)
- **Page replacement** is necessary to determine the page to evict



Basic Page Replacement

- Find a free frame
  - If there is free frame use it
  - Otherwise, use page_replacement() to find a victim
  - Write the victim out to backing store
- Read the new page (from backing store) into new freed frame
- Change the page tables
- Resume the page faulted process

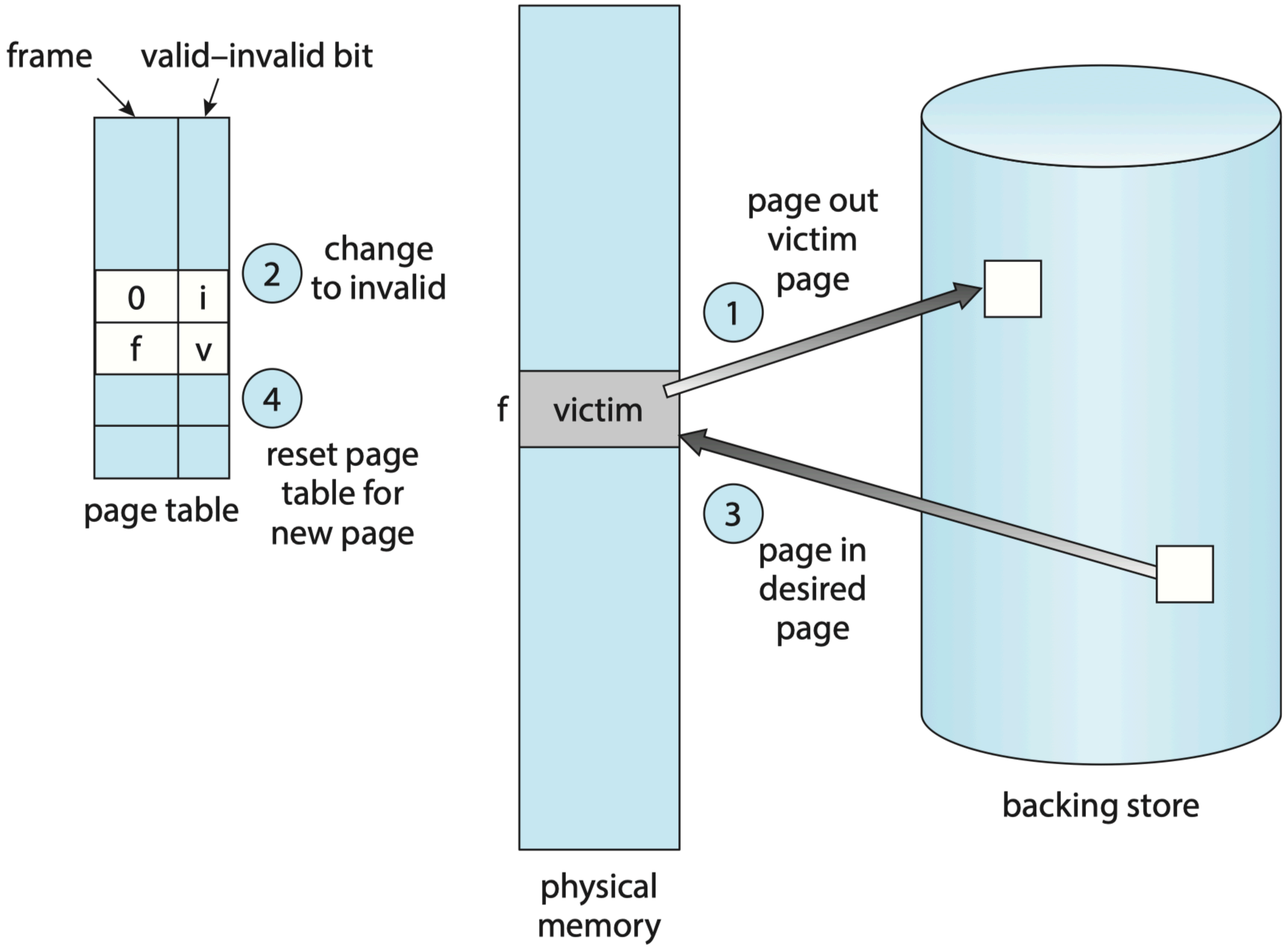


Two memory access on a page fault

Page Replacement Optimization

- Page is not modified, we don't need to write it back the backing store
- Prefer **unmodified pages** as victims ⇒

  Need to know the modified status ⇒ Page table can have a modified (dirty) bit
- Page replacement is also affected by the **frame-allocation algorithm** (this decides how many frames must be allocated for each process when multiple processes are loaded into memory)



Reference String

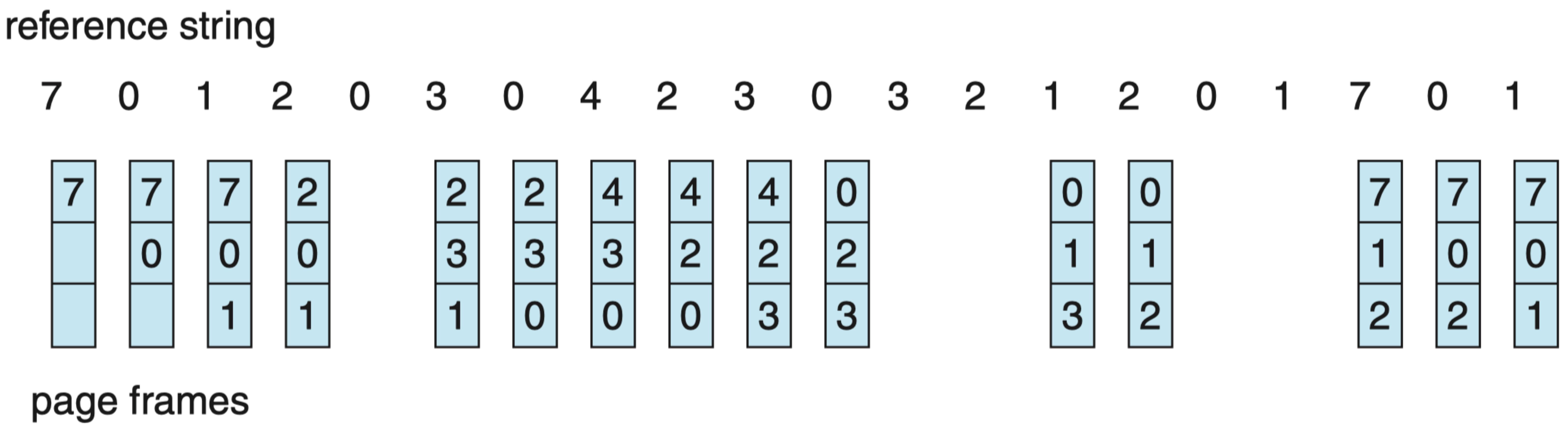
- Demand paging algorithms have to be evaluated by checking how well they are doing on representative applications - how many pages faults are generated for the different algorithms
- Application's memory access is represented by a reference string - a record of memory accesses made by the process
- Each memory access will be in a page, once the page is loaded it will be in memory (until evicted) and further accesses would not cause a miss
- Example: 100 bytes per page

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105

1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

FIFO Page Replacement

- First-in-first out replacement: Oldest page is replaced
- Strict times are not necessary, we can maintain the order of page induction and use in the victim selection
- FIFO is easy to understand and program, performance not good

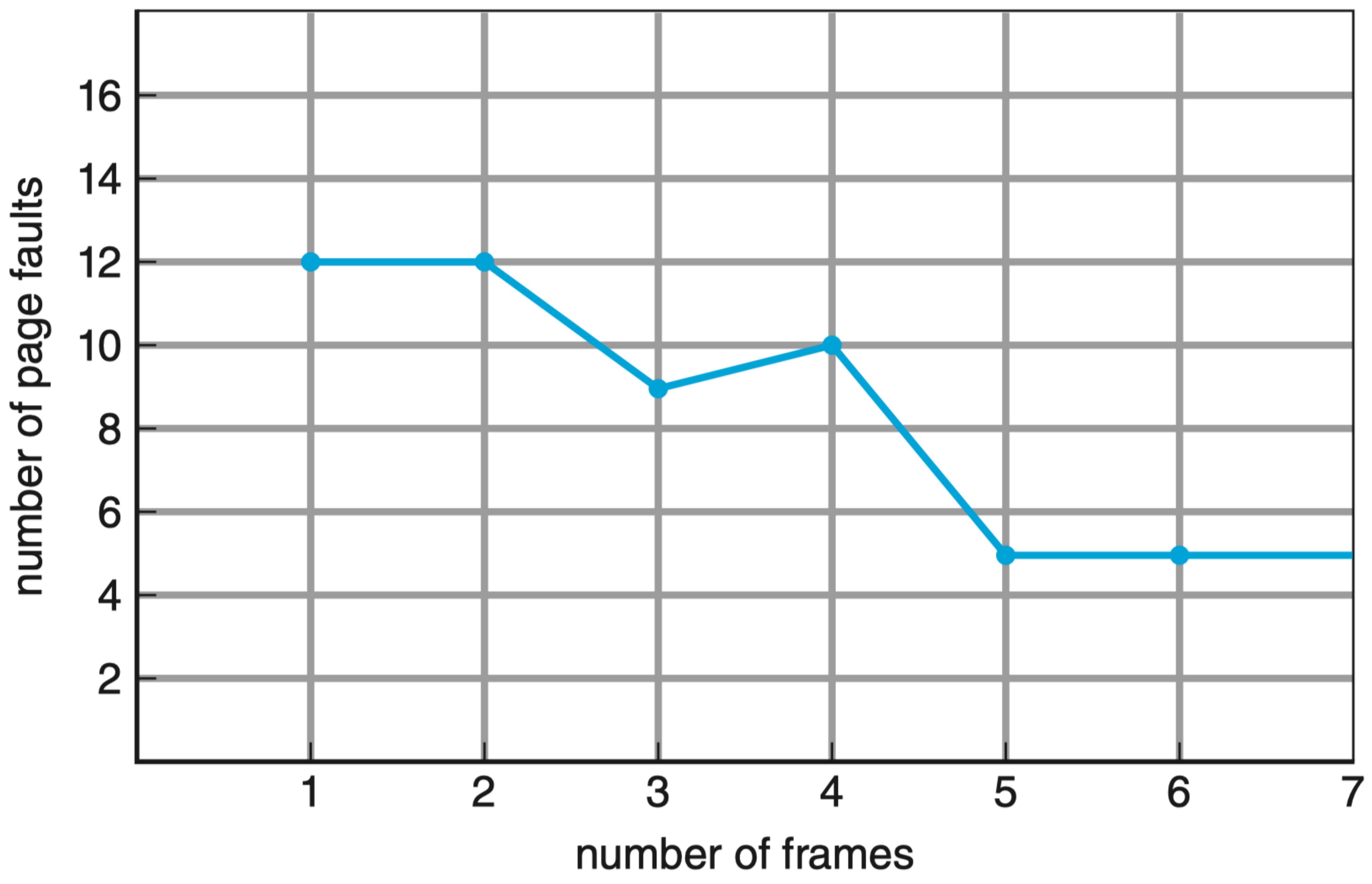


15 page faults

Belady’s Anomaly

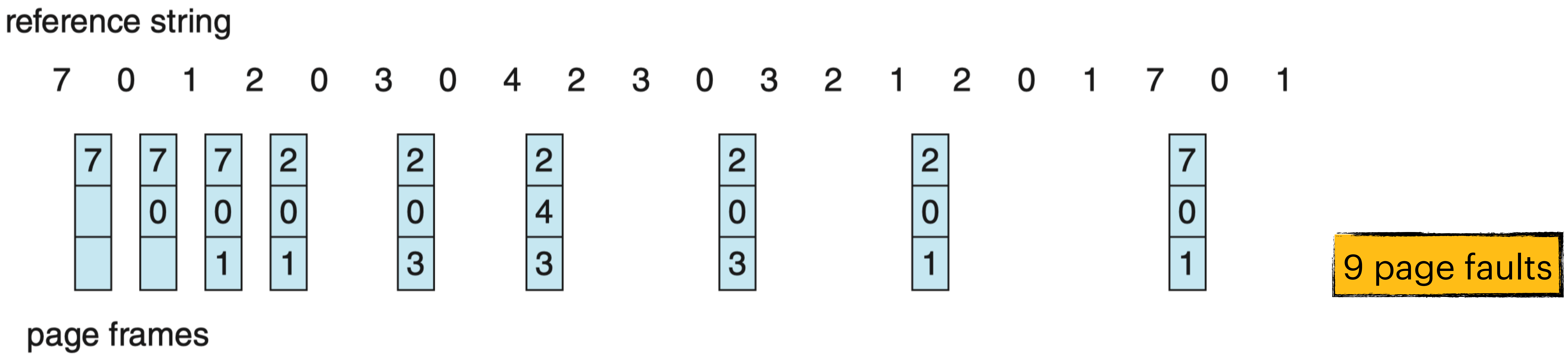
- More memory, you expect less page faults
- FIFO suffers from an anomaly where more memory can increase page faults for certain reference strings

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



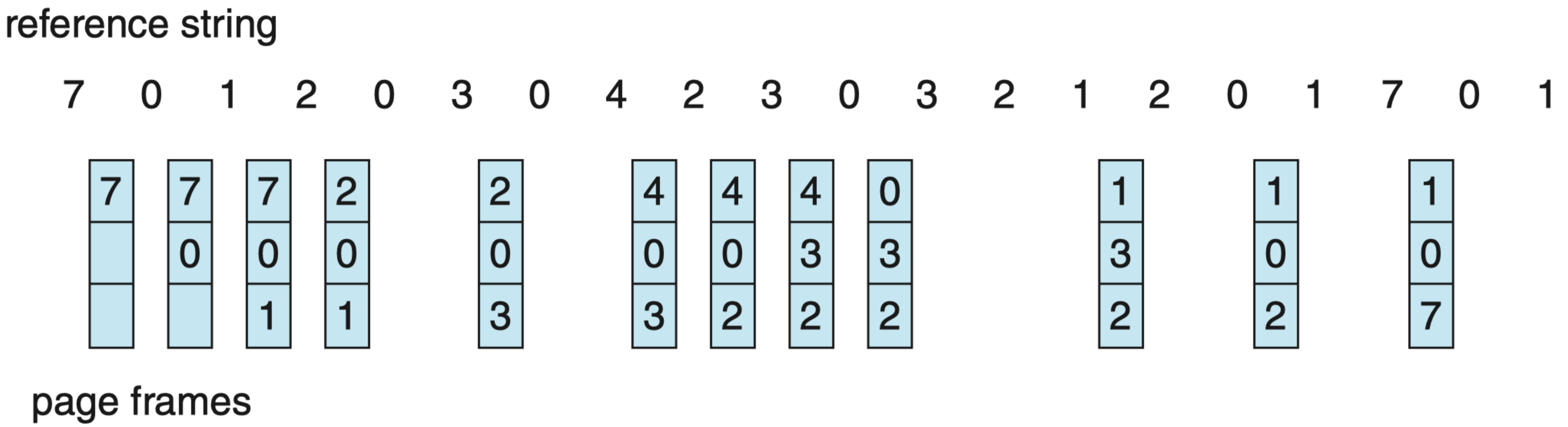
Optimal Page Replacement

- Optimal page replacement should give the lowest page-fault rate of all algorithms, never suffer from Belady's anomaly
- Replace the page that is not going to be used for the longest period of time into the future
- Optimal is not possible to implement - need to know future access
- Can be used for comparison studies



Least Recently Used (LRU) Page Replacement

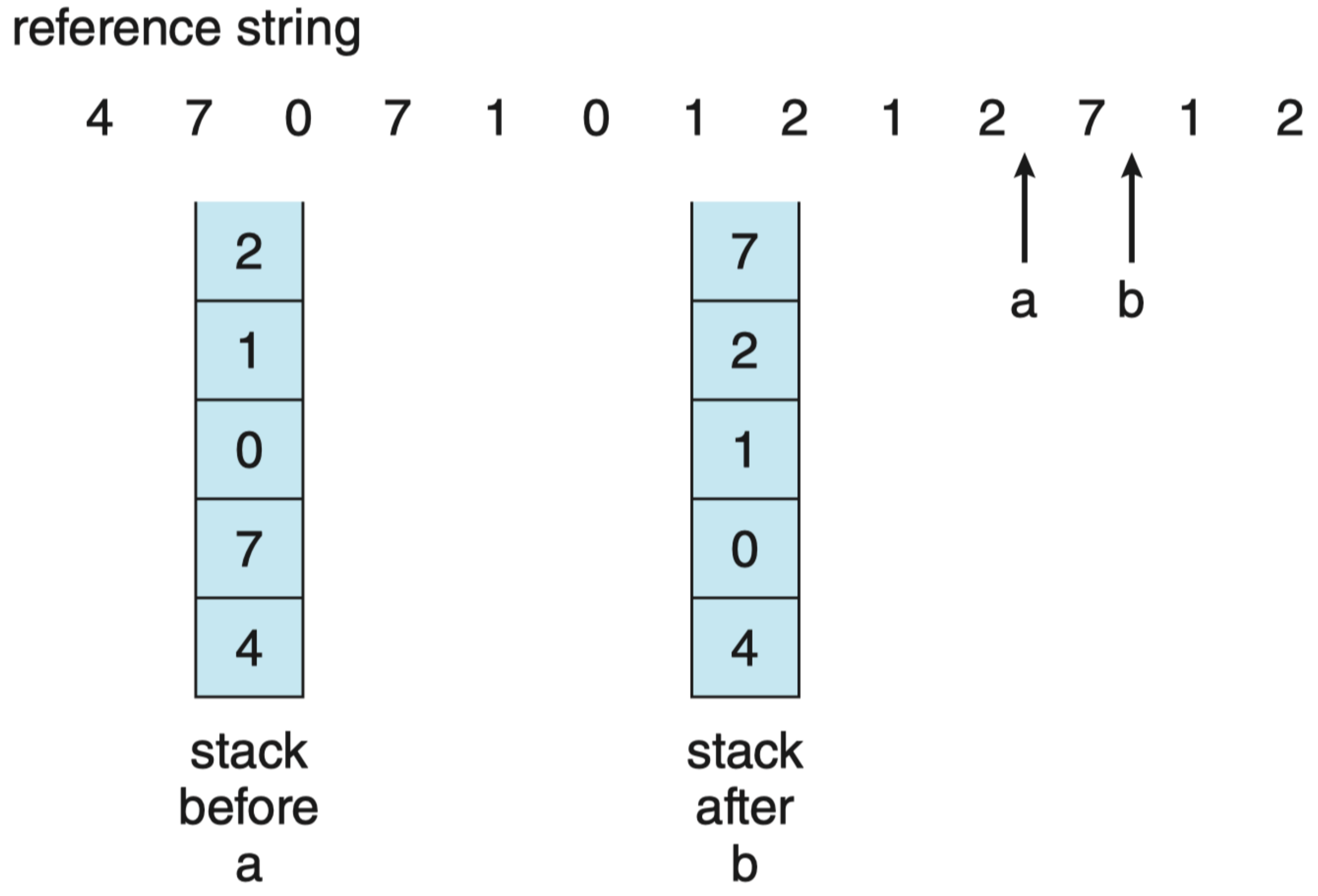
- Optimal is not possible to implement because we have to look into the future
- How about approximate the future by looking into the past? This is the LRU approach!
- LRU maintains with each page when it was used
- When a victim is needed, select the page that has not been used in the longest time in the past



12 page faults

Implementing LRU Page Replacement

- LRU is a difficult algorithm to implement - need to order the frames by the time of last use
- **Counter Approach:**
  - Maintain a time-of-use field with each page
  - We have a logical clock or counter that increments at each memory reference
  - When a page is referenced, we copy the clock into the time-of-use field
  - A page with the smallest time-of-use field is the victim
- **Stack Approach:**
  - Keep a stack of page numbers used in the past
  - Bubble up the recent access to the top of the stack

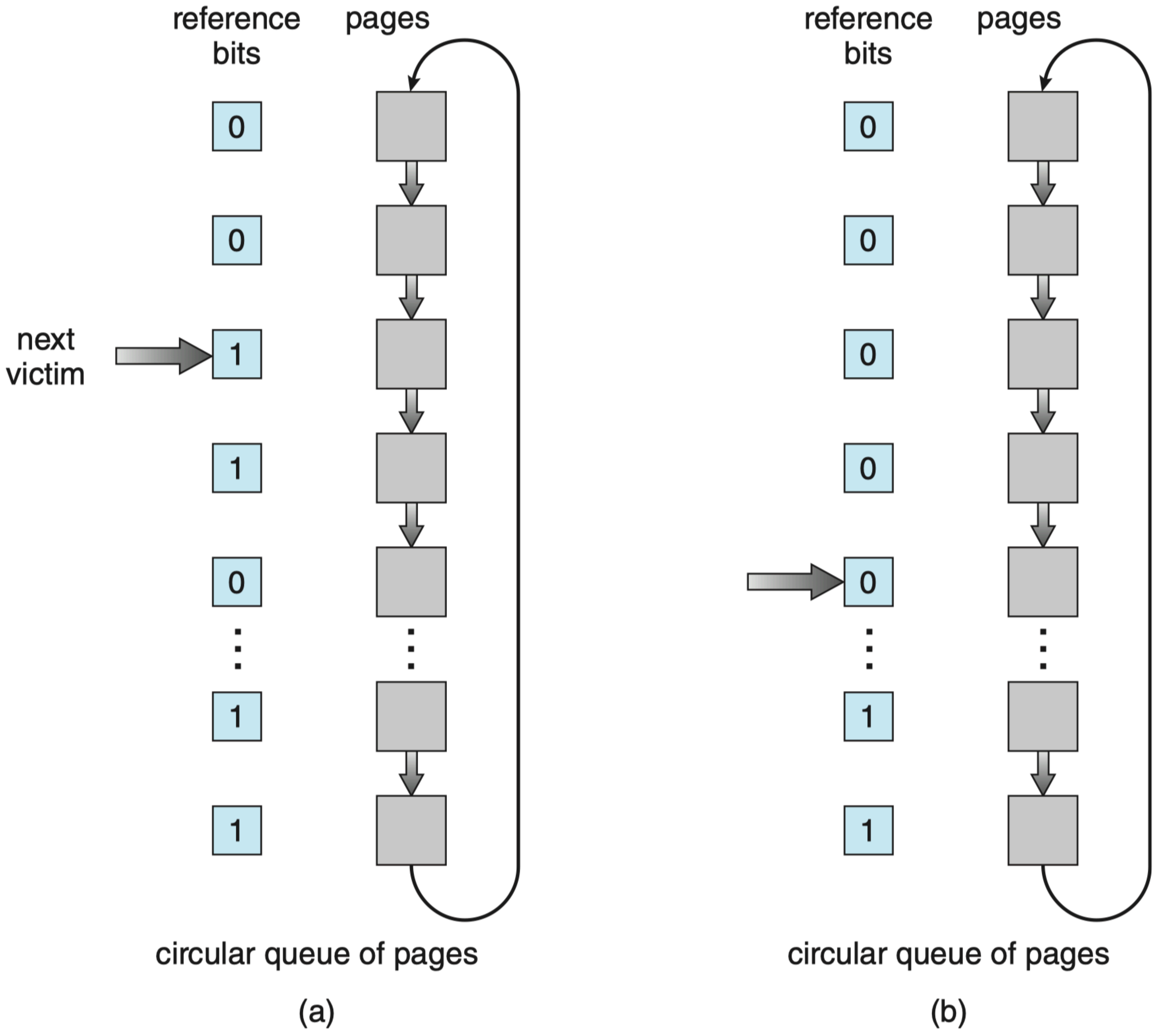


LRU-Approximation Page Replacement

- Many hardware systems provide a **reference bit** in the page table to aid page replacement algorithms
- Initially, all reference bits are cleared (0)
- If a page is accessed, reference bit is set to 1
- We can determine the pages that are used by examining the reference bit - not the access order

Second Chance Algorithm

- Second chance is a FIFO replacement algorithm with enhancements
- When a page is selected as victim, we examine the reference bit
  - If it is set, we reset it and skip to the next
  - If it is not set, we return it as the victim
- A victim that is referenced gets a second chance to remain in memory!
- If all the pages are referenced, second chance becomes a FIFO



Least Frequently Used Page Replacement

- LFU requires that the page with least frequently used count be replaced
- Actively used page should have high "frequently used count" and would remain in memory
- How do we maintain the frequently used count?
- Maintain a counter and increment the counter at each use
- Shift the counter right by 1 bit to decay the counter