# Virtual Memory

# Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples

# 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
  - Error code, unusual 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
  - Program and programs could be larger than physical memory

# Background

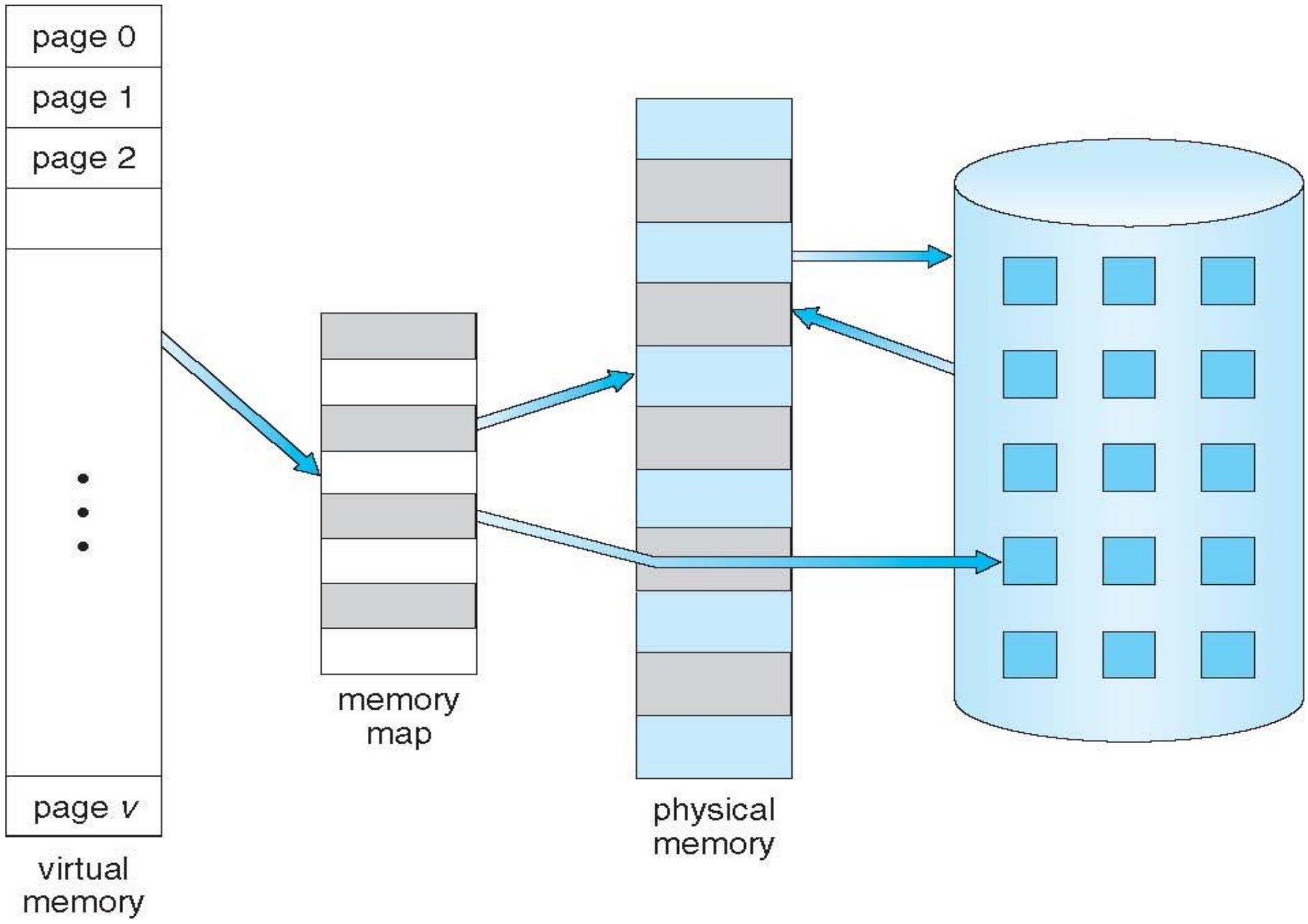
- **Virtual memory** – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for 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

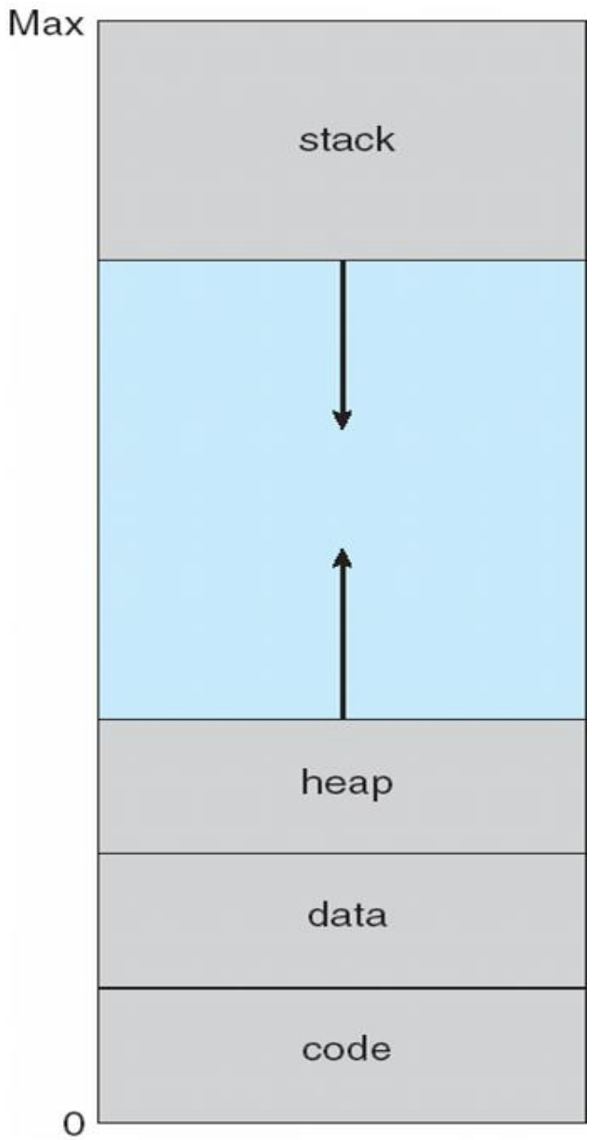
# Virtual Memory

- This is an enhancement of simple paging wherein the pages are brought into the primary memory from the secondary memory (disk) only on demand. Thus, the entire process is not loaded into the memory at one stretch. It is loaded part by part and executed and then swapped back to the disk so that some other blocks of the same process can be loaded in that place. This gives an illusion to the user that the memory can accommodate and execute a process of any size. Since the full process is not loaded at one stretch, the process size can exceed the total memory size and still be executed. This is known as the virtual memory concept.

# Virtual Memory That is Larger Than Physical Memory



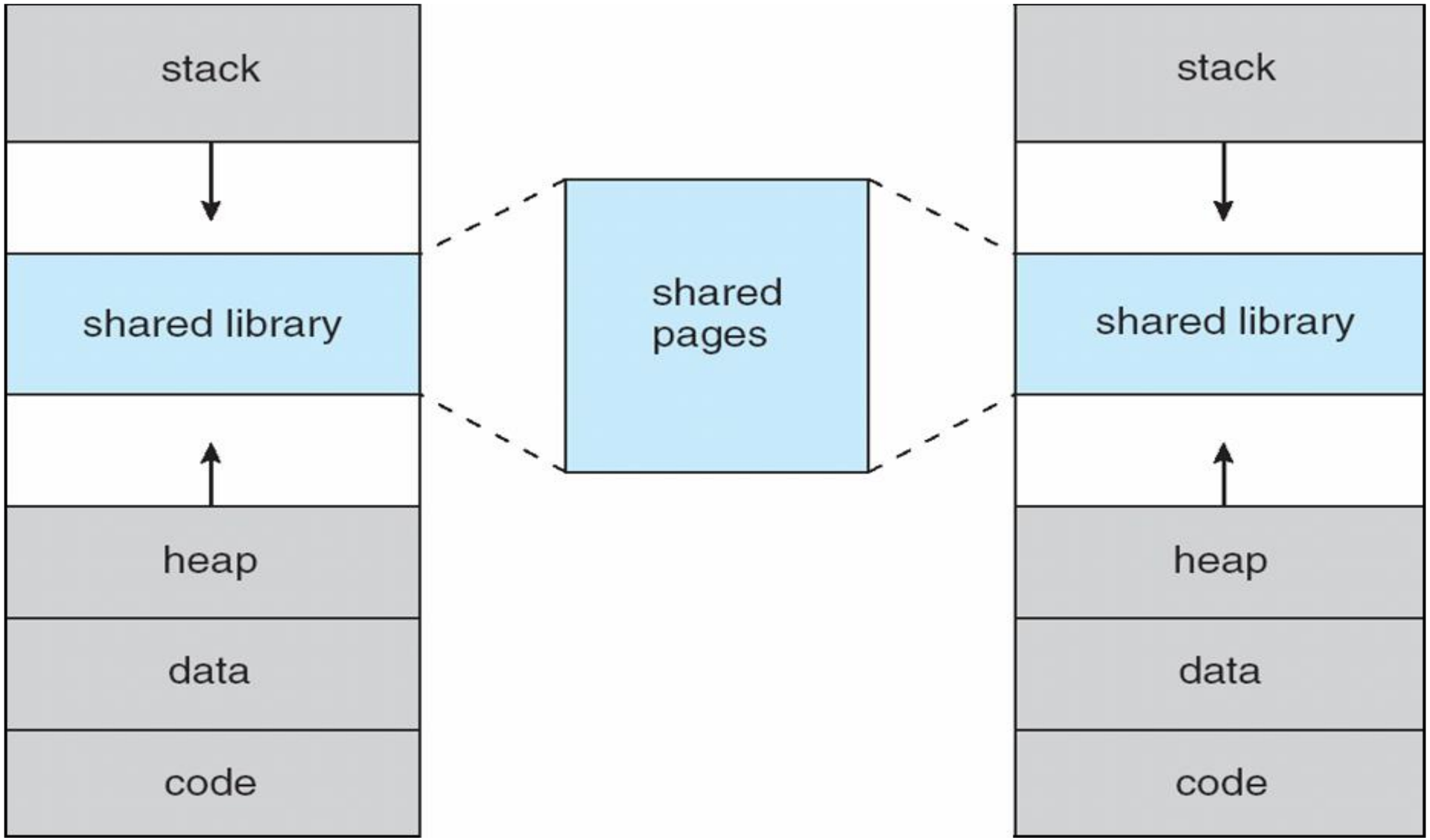
# Virtual-address Space



# Virtual Address Space

- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during `fork()`, speeding process creation

# Shared Library Using Virtual Memory



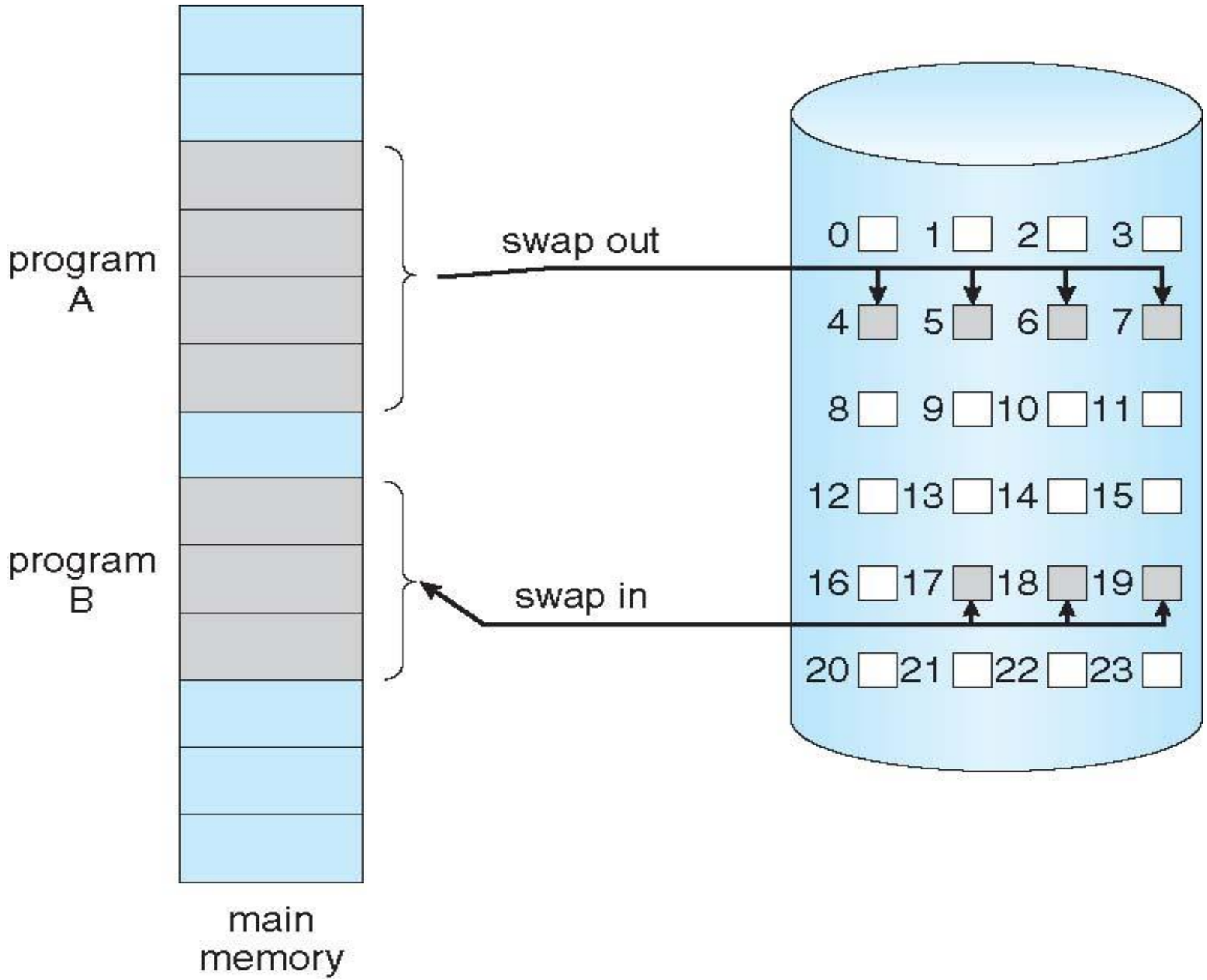
# Demand Paging

- 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

- 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**

# Transfer of a Paged Memory to Contiguous Disk Space



# Valid-Invalid Bit

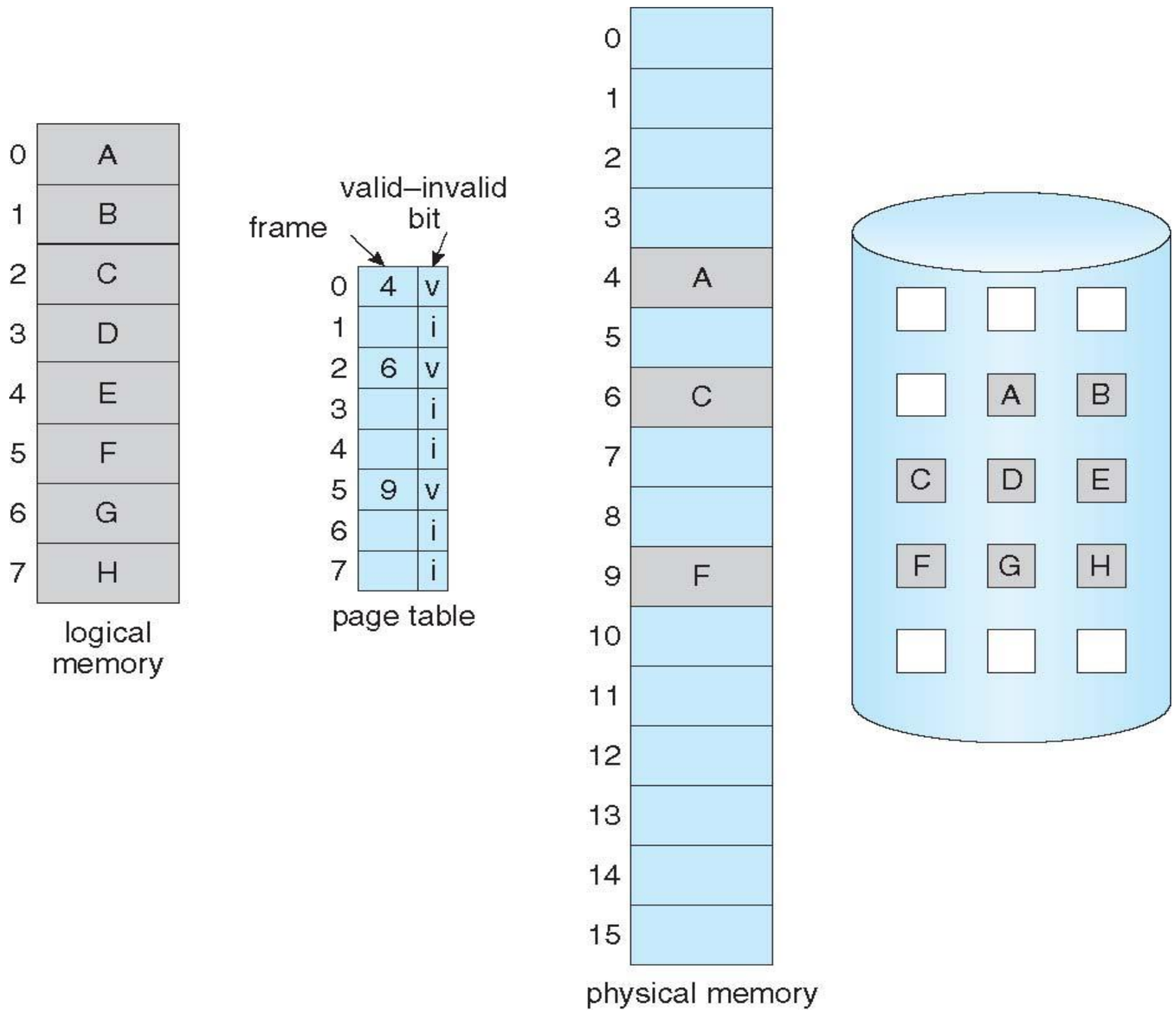
- With each page table entry a valid–invalid bit is associated
  (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

| Frame # | valid-invalid bit |
|---|---|
| | **v** |
| | **v** |
| | **v** |
| | **i** |
| …. | |
| | **i** |
| | **i** |

page table

- 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



# 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

# Page Fault and Page Replacement

- Page Fault: If a user job accesses a page and the page is not available in the main memory, a page fault is said to occur

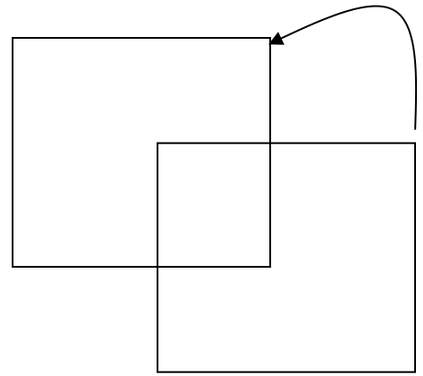
- Page Replacement: If the memory is full then the inactive pages which are not needed currently for execution are removed and are replaced by those pages from the secondary device which are to be executed. This is called Page Replacement.

# Aspects of Demand Paging

- Extreme case – start process with *no* pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - And for every other process pages on first access
  - **Pure demand paging**
- Actually, a given instruction could access multiple pages -> multiple page faults
  - Pain decreased because of **locality of reference**
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart

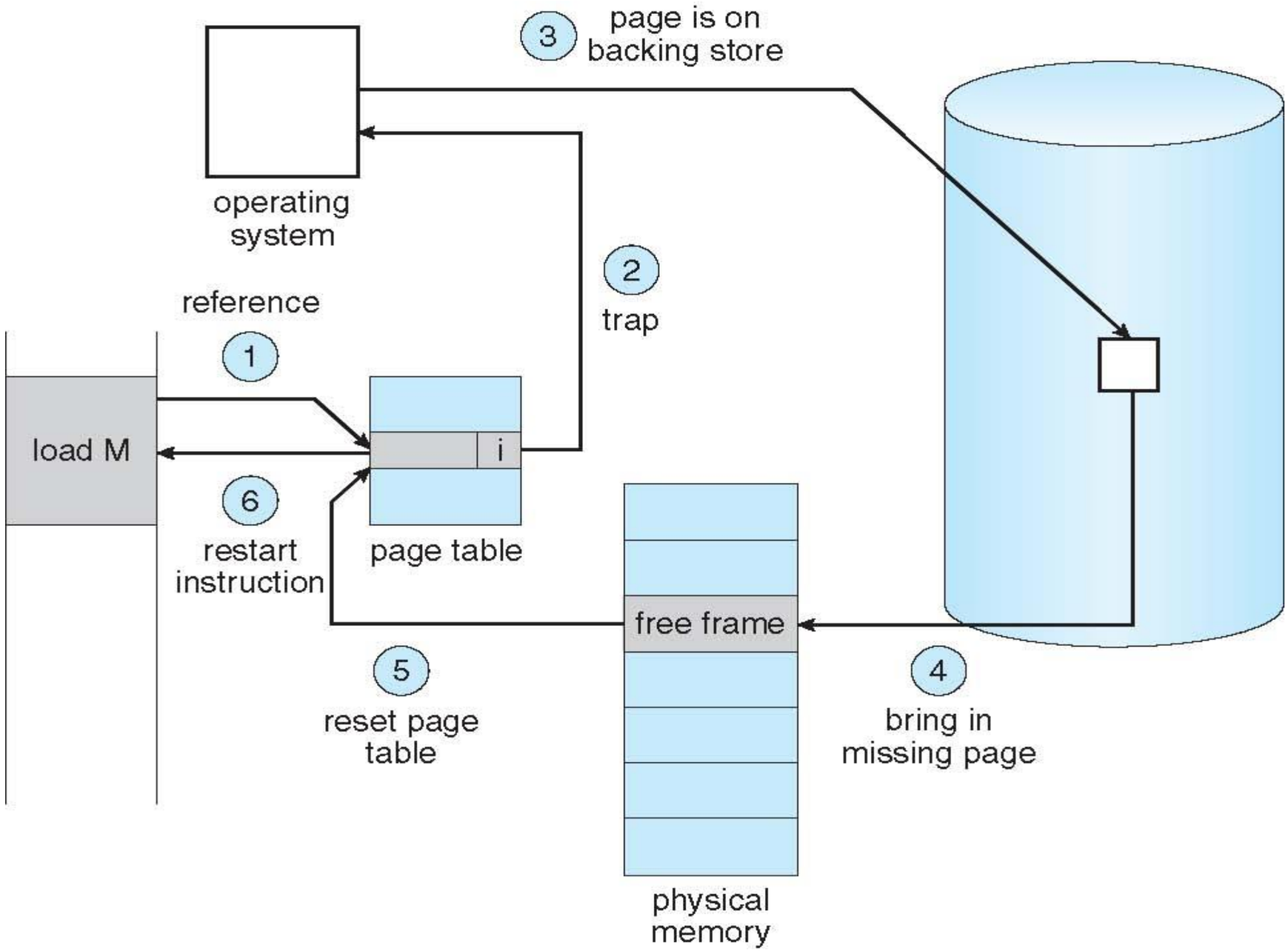
# Instruction Restart

- Consider an instruction that could access several different locations
  - block move



  - auto increment/decrement location
  - Restart the whole operation?
    - What if source and destination overlap?

# 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 $0 \le p \le 1$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault

- Effective Access Time (EAT)

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

# 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!!
- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 x p
    20 > 7,999,800 x p
  - p < .0000025
  - < one page fault in every 400,000 memory accesses

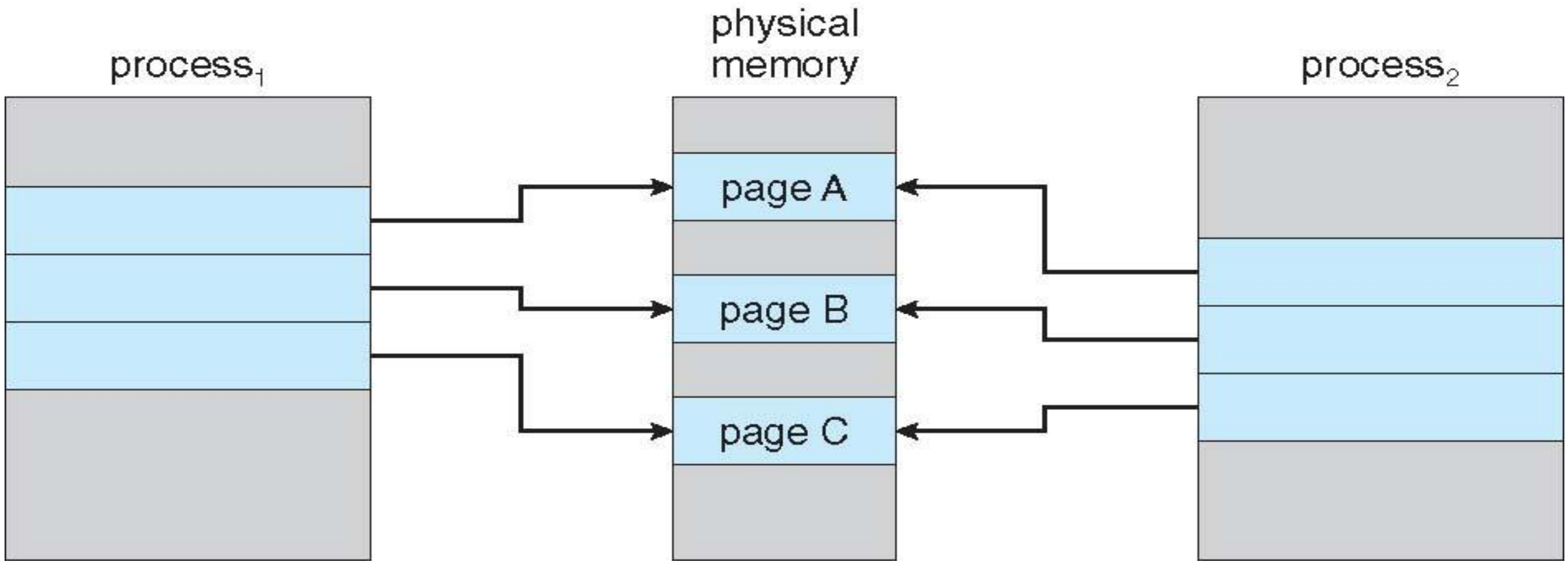
# Demand Paging Optimizations

- Copy entire process image to swap space at process load time
  - Then page in and out of swap space
  - Used in older BSD Unix

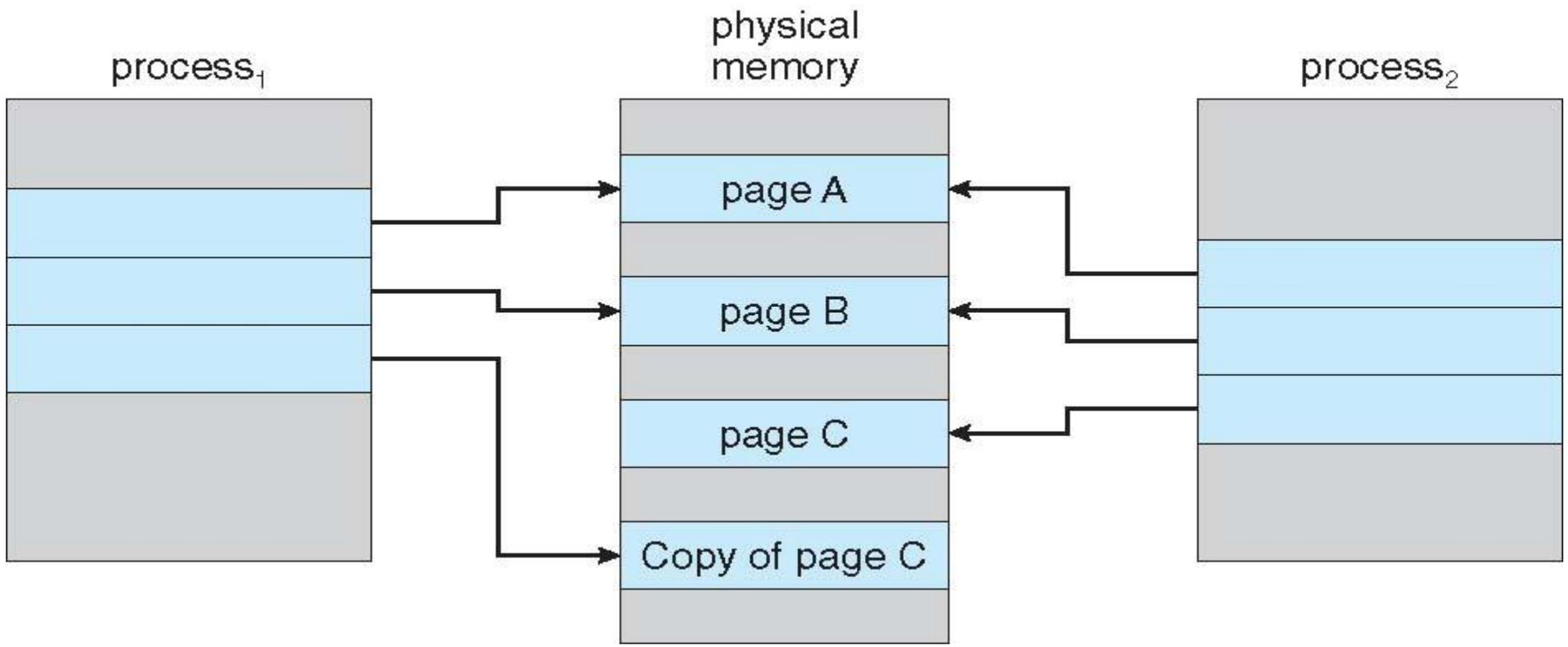
# Copy-on-Write

- **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
- 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 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?

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

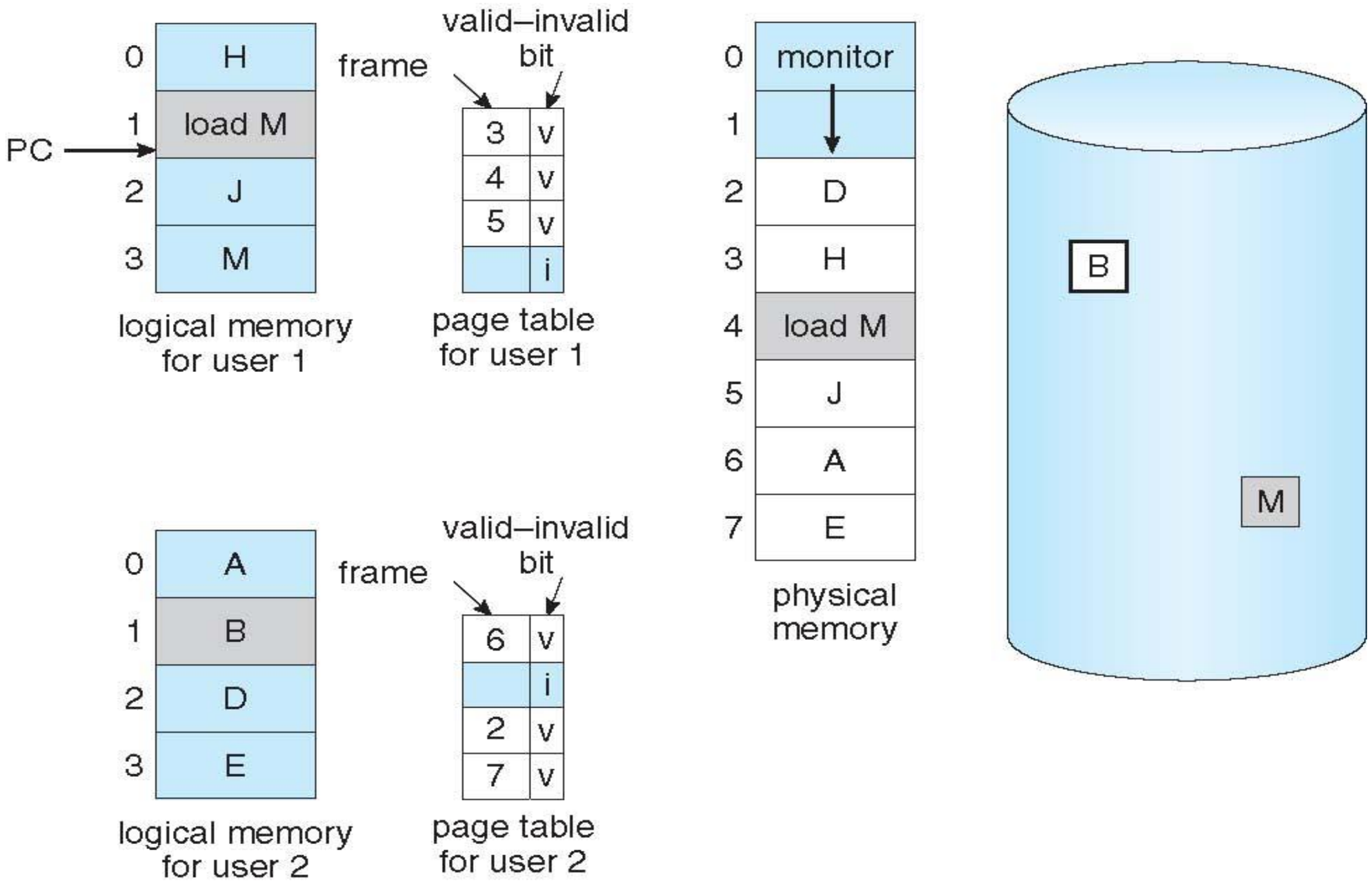
- Page replacement – find some page in memory, but not really in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result in minimum number of page faults

- 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

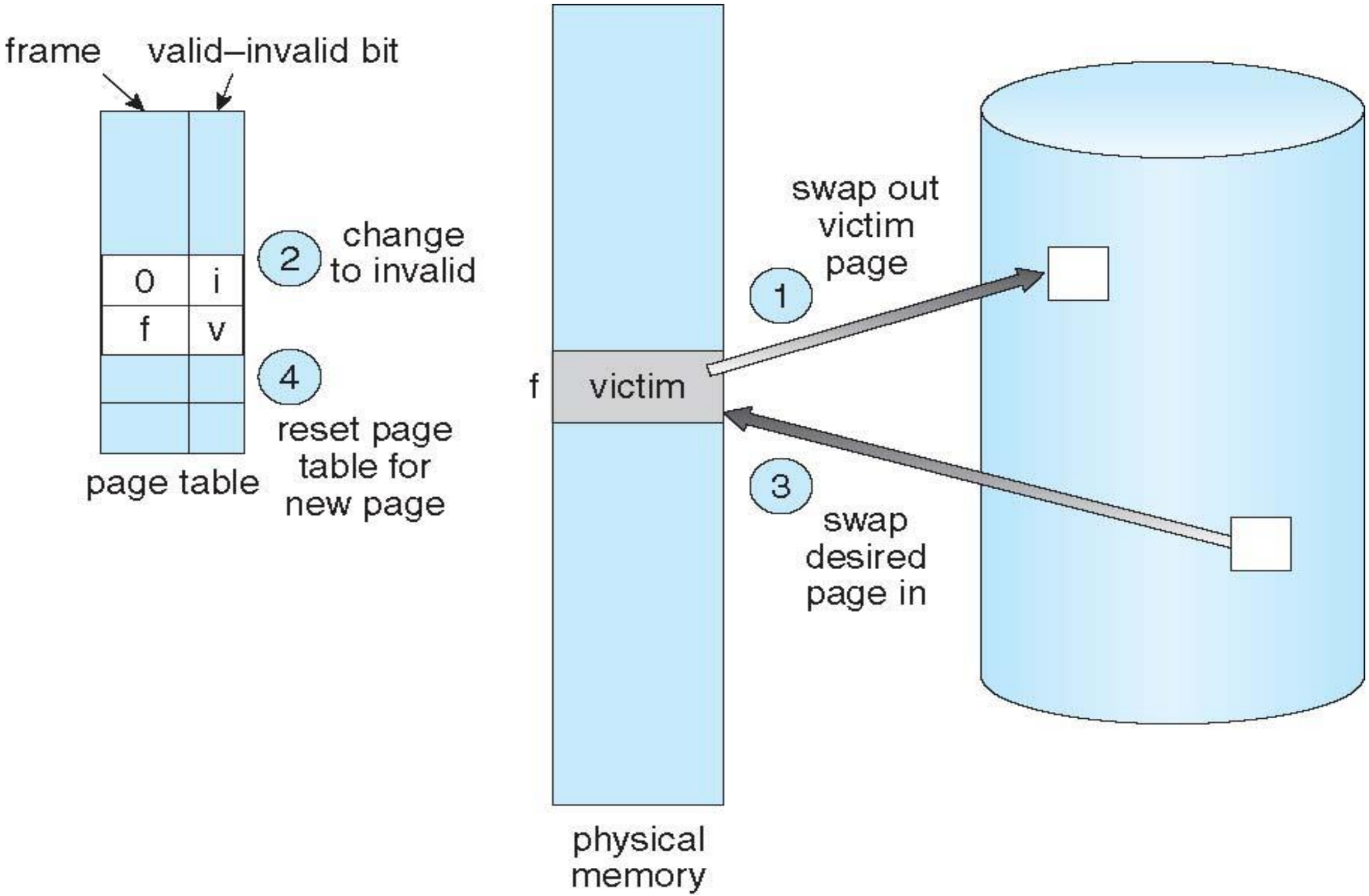
2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim frame**
     - Write victim frame to disk if dirty

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

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

# Page Replacement



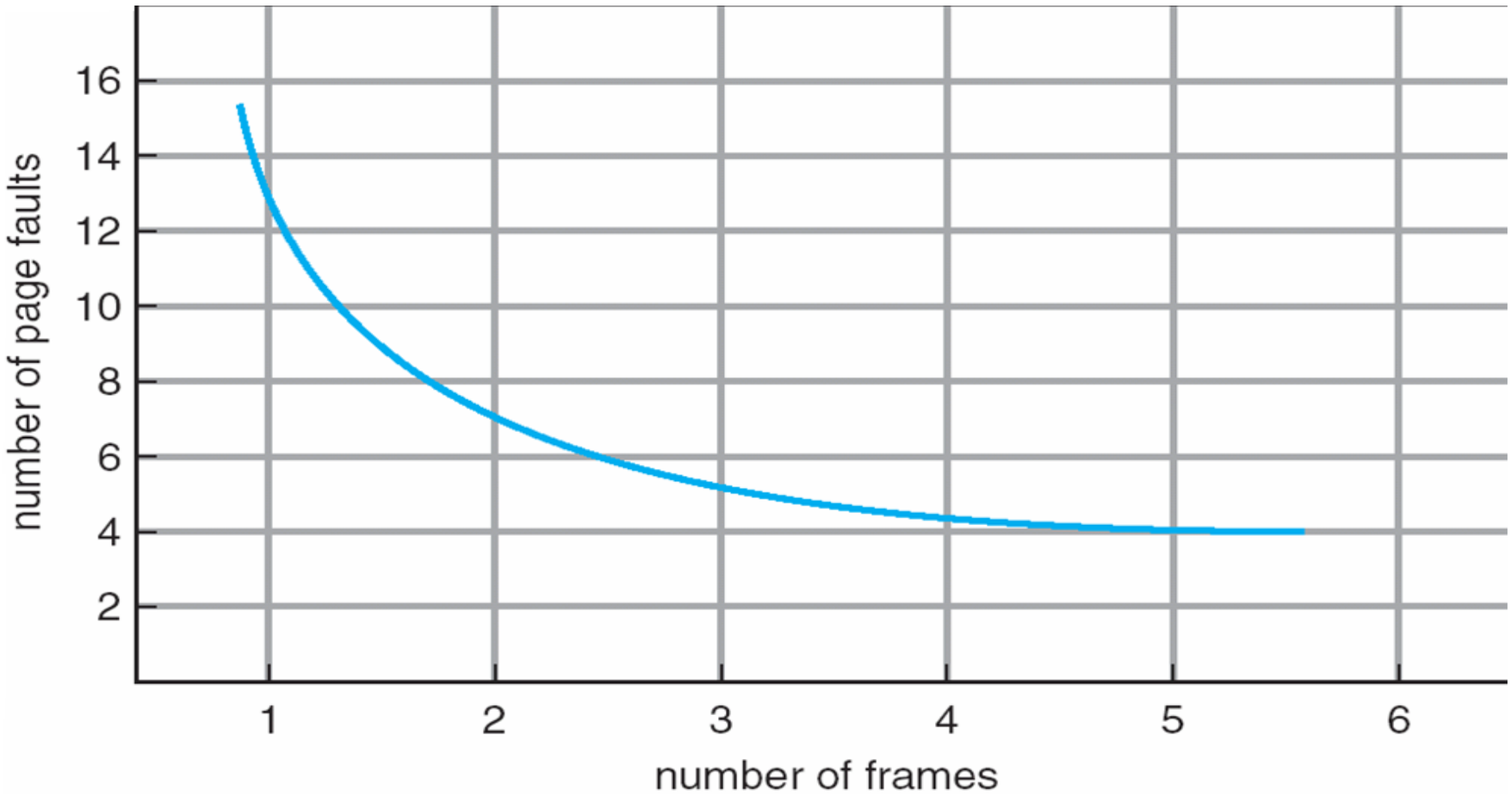
# Page and Frame Replacement Algorithms

- **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
- 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**

# Graph of Page Faults Versus The Number of Frames

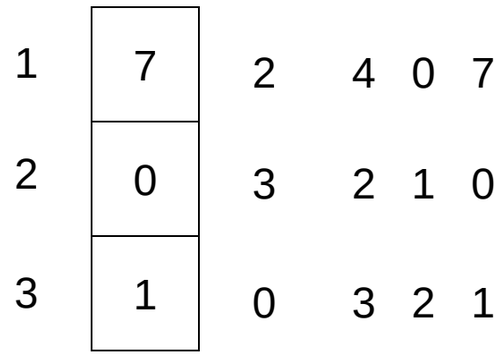


# Page Replacement Algorithms

- LFU: If the algorithm decides to move a page from main memory and store it in secondary memory based on the fact that it is not used often, then it is called LFU. For every page a reference counter is maintained in the judgement field.
- LRU: If the algorithm decides to move a page from main memory and store it in secondary memory based on the fact that it is not used often in the recent times, then it is called LRU. For every page a timestamp is maintained in the judgement field.
- NRU: If the algorithm decides to move a page from main memory and store it in secondary memory based on the fact that it is not used at all in the recent times, then it is called NRU. A reference bit is associated with each page.
- FIFO: If the algorithm decides that the page has been first moved to the memory should be moved out to secondary memory first, then it is using FIFO.

# First-In-First-Out (FIFO) Algorithm

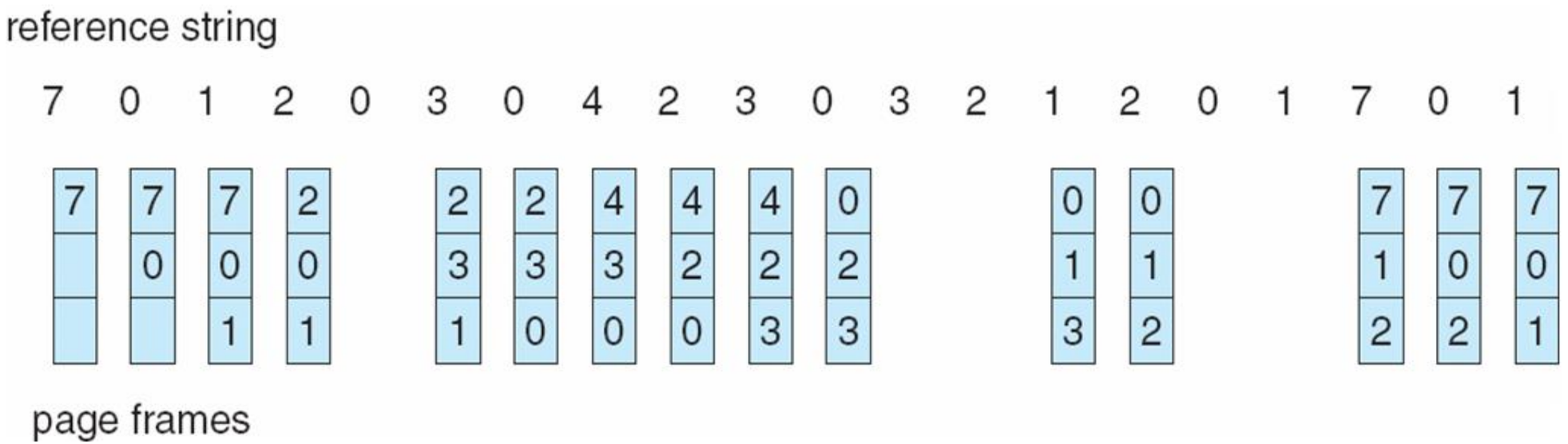
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



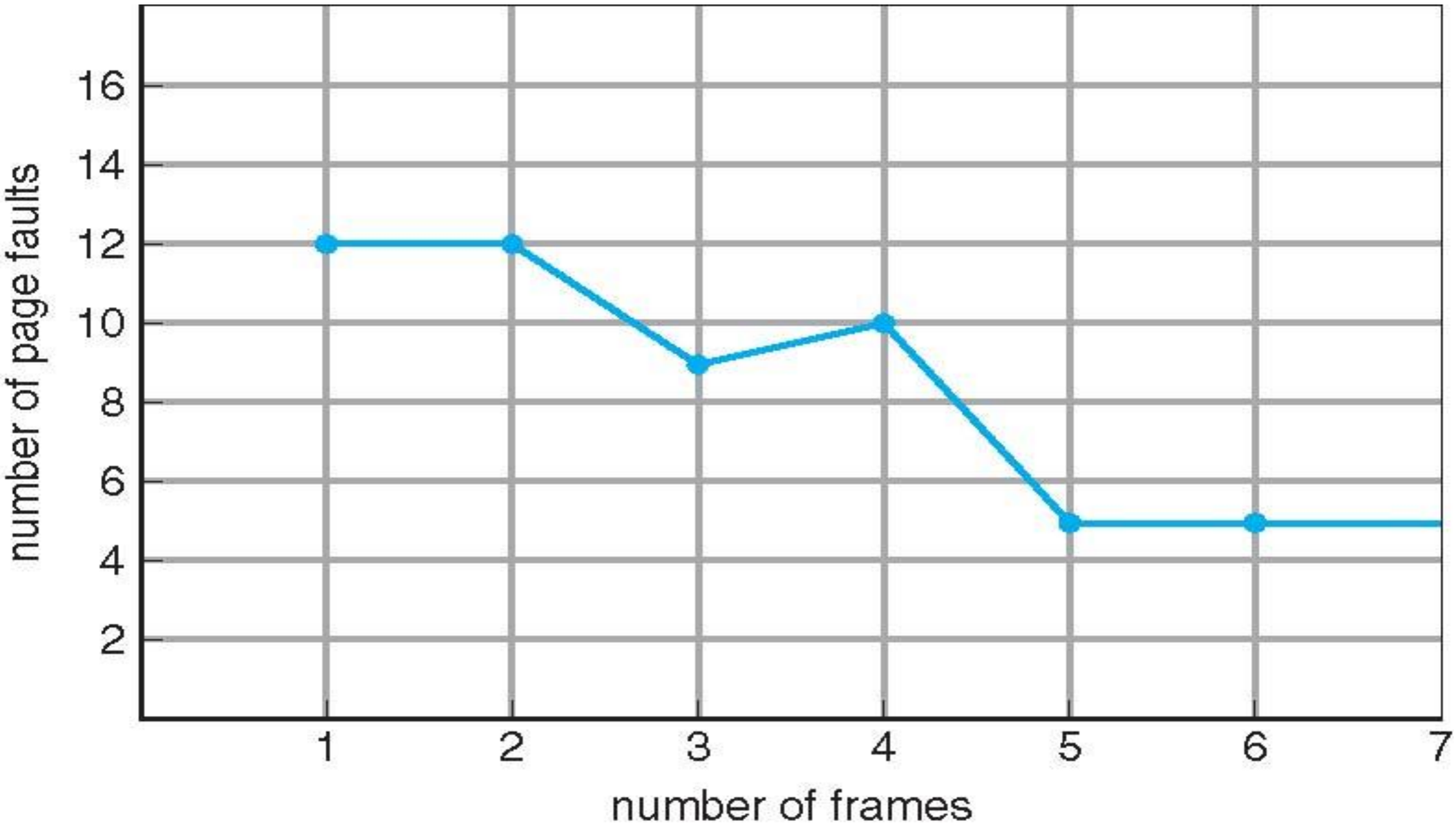

15 page faults

- 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



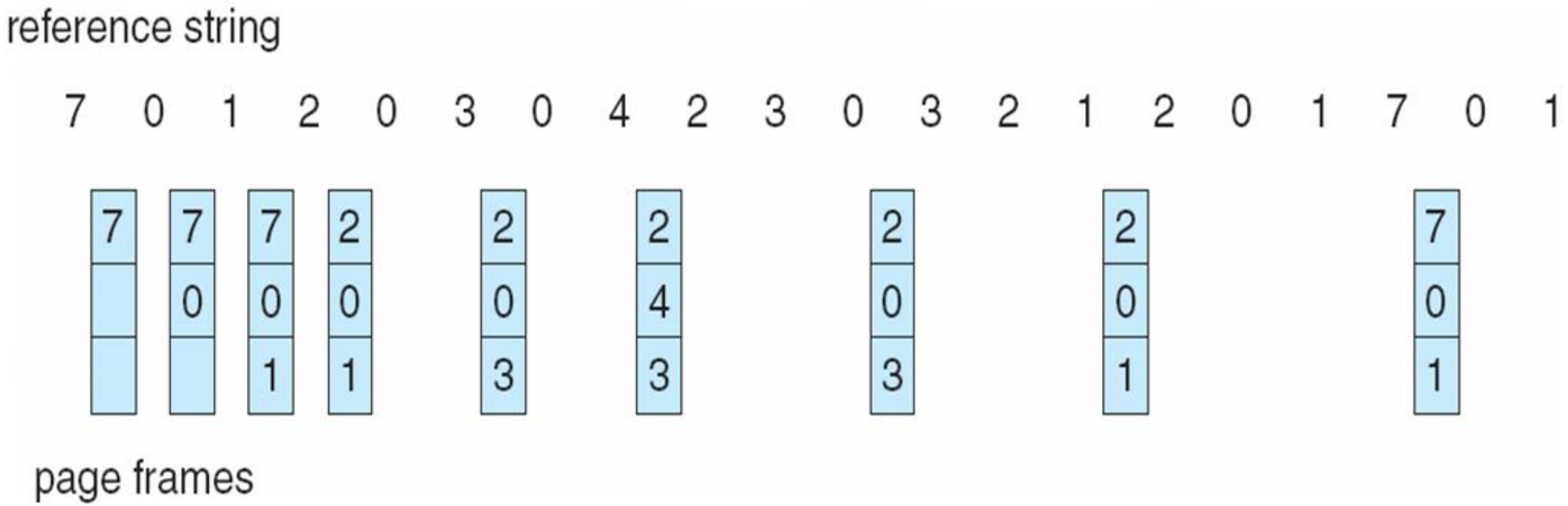
# FIFO Illustrating Belady's Anomaly



# Optimal Algorithm

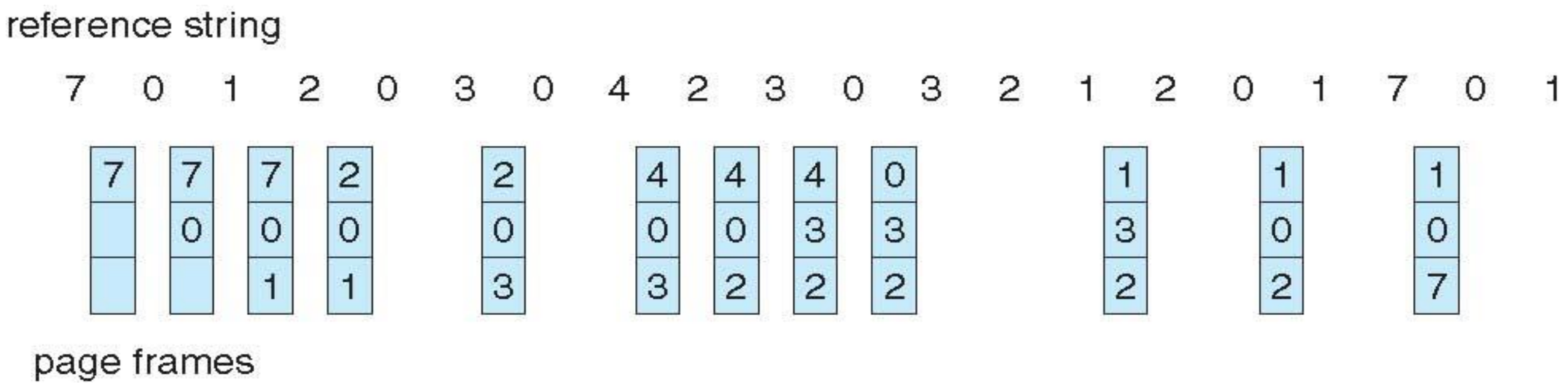
- Replace page that will not be used for longest period of time
  - 9 is optimal for the example on the next slide
- How do you know this?
  - Can’t read the future
- Used for measuring how well your algorithm performs

# Optimal Page Replacement



# Least Recently Used (LRU) Algorithm

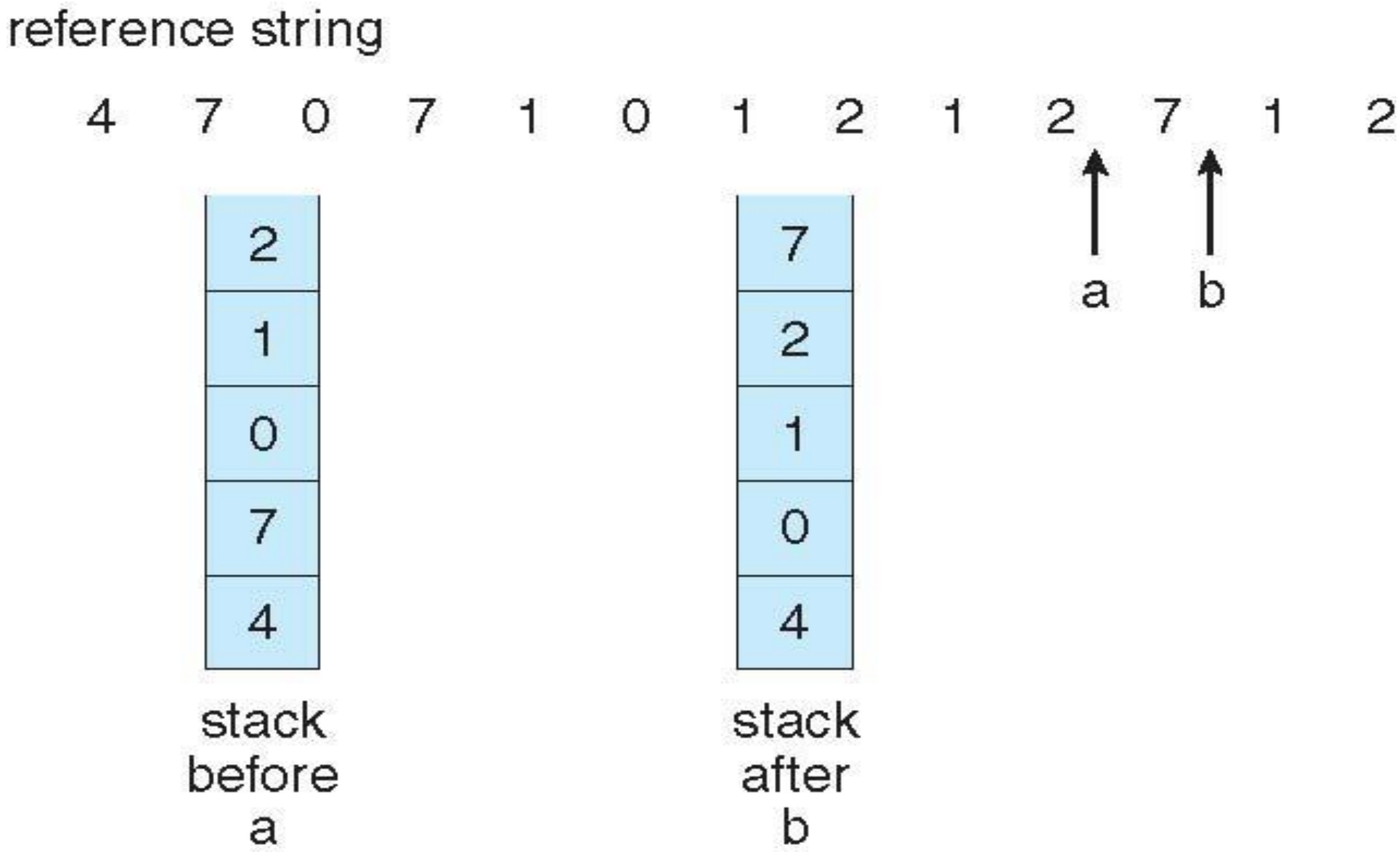
- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page



# 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
- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly

# Use Of A Stack to Record The Most Recent Page References



# Page-Buffering Algorithms

- Keep a pool of free frames, always
  - Then frame available when needed, not found at fault time
  - Read page into free frame and select victim to evict and add to free pool
  - When convenient, evict victim
- Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages there and set to non-dirty
- Possibly, keep free frame contents intact and note what is in them
  - If referenced again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected

# 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

# 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

# Principle of Locality

- A program tends to operate within a particular logical module, drawing its instructions from a single procedure and its data from a single data area.

- Therefore program references tend to be grouped into small localities of address space. Loops make localization even stronger : the tighter the loop the smaller the spread of references.

- This behavior is known as “Operating in Context”, and leads to the “Principle of Locality”, i.e. : program references tend to be grouped into small localities of address space, and these localities tend to change only intermittently.

# Working Set Model(1)

- Program behavior can be modeled through the “Working Set Model”.
- Competition among processes for memory space:
  - Degree of multiprogramming is proportional to the number of processes present in memory
  - Processor utilization: is % of time the processor is busy executing, i.e. not idle waiting for I/O.
  - As the degree of multi-programming increases, the processor utilization increases
  - However: there is a critical degree of multiprogramming  beyond which processor utilization starts to decrease. The processor starts to spend more time swapping pages in and out of memory than executing processes.

# Working Set Model(2)

- Implications:
  - The high degree of multiprogramming makes it impossible for every process to keep sufficient pages in memory to avoid generating a large number of faults.
  - The disc channel becomes saturated.
  - Most processes are blocked awaiting a page transfer.
  - The processor is under utilized.
- This is THRASHING, and is directly dependent on memory size : the larger the memory the larger is the number of processes before thrashing starts.
- Each process needs a minimum number of pages, called its ‘working set’, in memory to make effective use of the processor.
- If process has < working set : continually interrupted by paged faults → Thrashing.

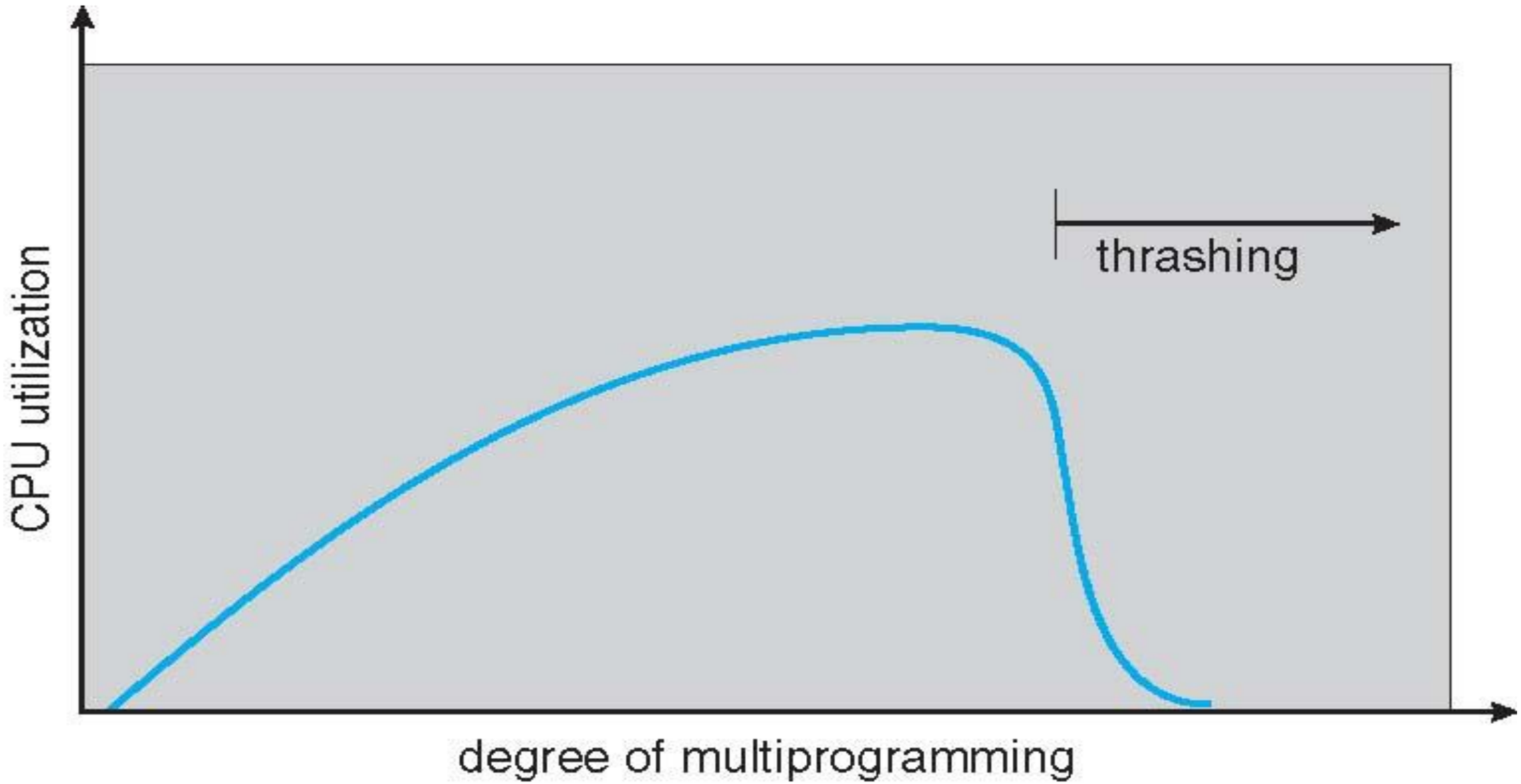
# Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking 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

# Thrashing

- Most of the time is being spent in either swapping in or swapping out of the pages from main memory to secondary memory, instead of doing useful work.
- This high paging activity when happens frequently is called THRASHING
- To overcome thrashing ,system schedules some pages to be removed from memory in the background (Page Stealing) and continues till a certain level of page frames are free
- It is the phenomenon in virtual memory schemes when the processor spends most of its time swapping pages, rather than executing instructions.

# Thrashing (Cont.)



# 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

# Thank You