Chapter 10: Virtual Memory

CS 3423 Operating Systems

Fall 2019

National Tsing Hua University

Background

- No need to load entire program into mem. all at once
  - Error code, unusual routines, large data structures
- Partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running
    => more programs can run at the same time
    - Increased CPU utilization and throughput with no increase in response time or turnaround time
  - Less I/O needed to load or swap programs into memory
    => each user program runs faster

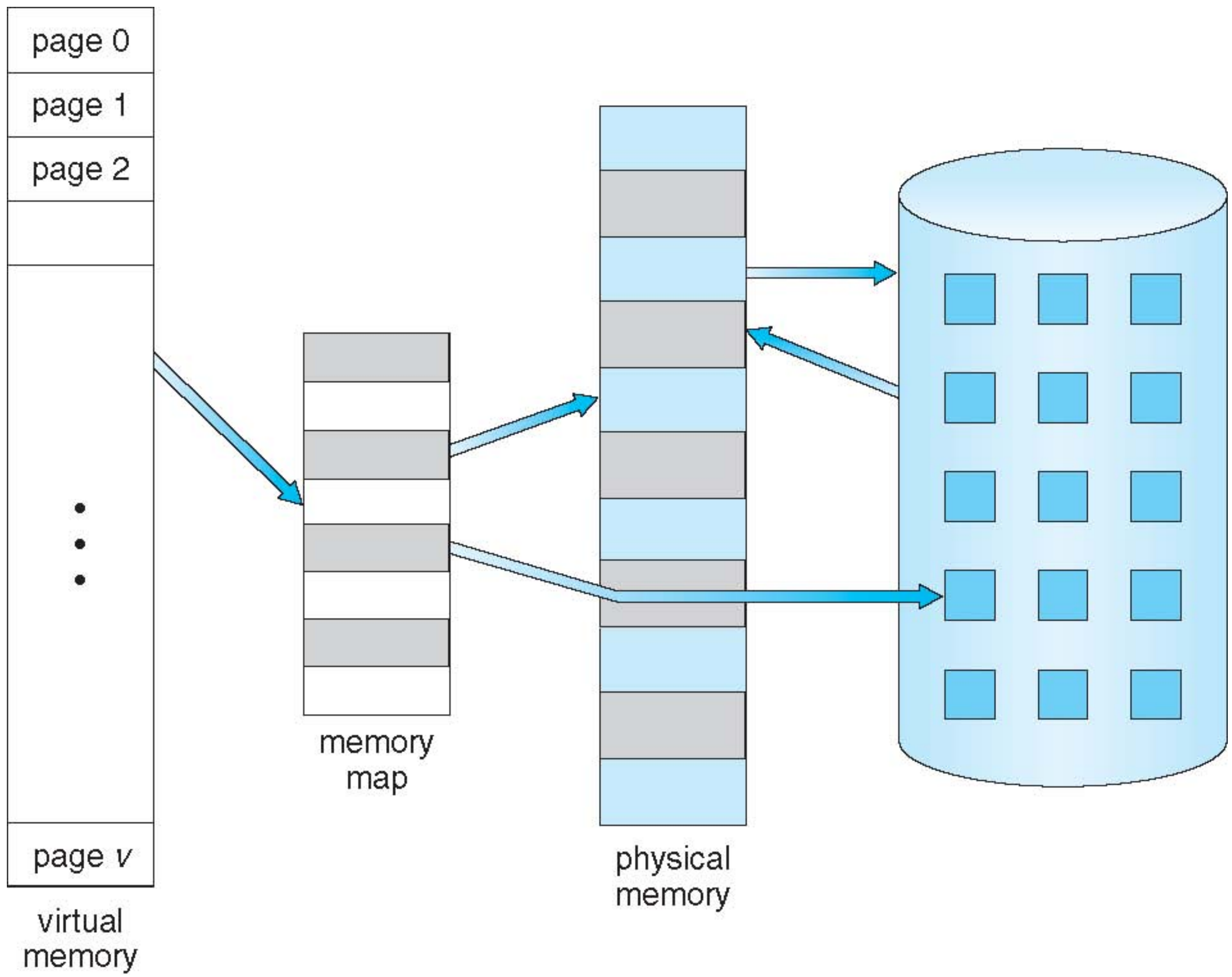
Background (Cont.)

- Virtual address space – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Virtual Memory advantages

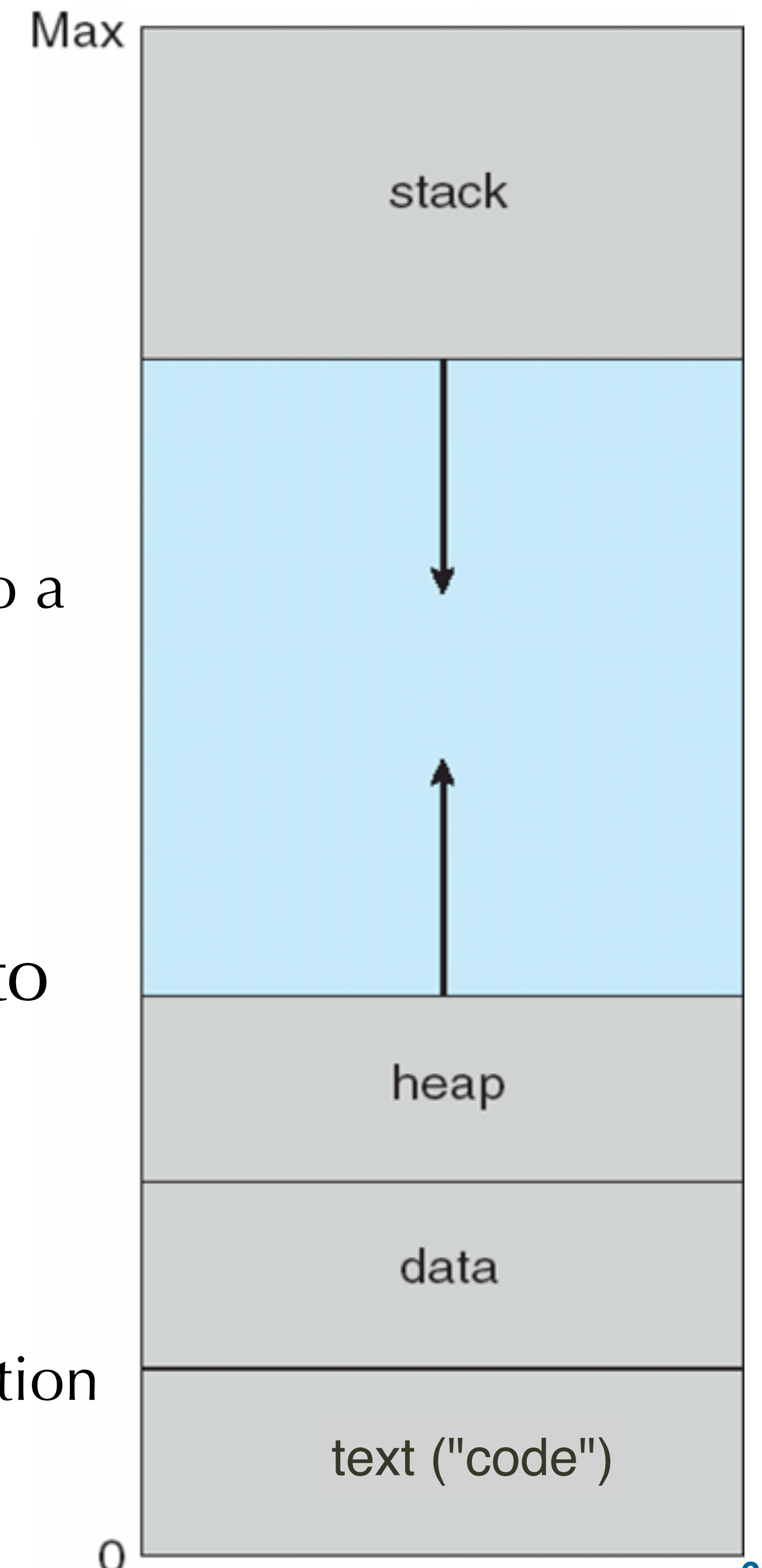
- Partial code loading
  - More programs running concurrently
  - Less I/O needed to load or swap processes
  - Allows for more efficient process creation
- Larger logical address space than physical
- Allows several processes to share memory

Virtual Memory that is Larger than Physical Memory

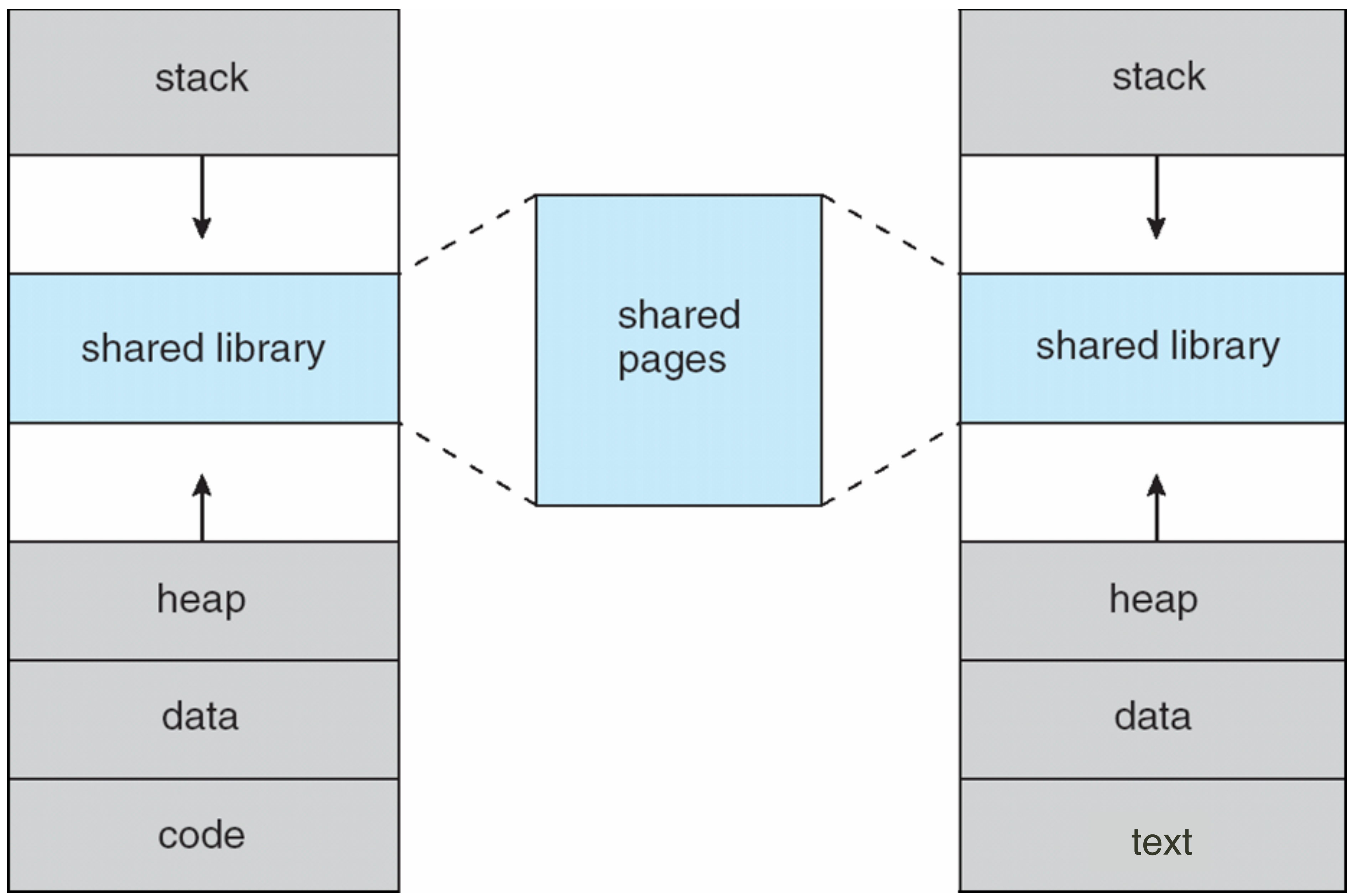


Virtual-address Space

- Maximizes address space use
  - Stack grows "down"
  - Heap grows "up"
  - Unused address space between the two is hole
  - No physical memory needed until heap or stack grows to a given new page
  - Enables sparse address spaces with holes left for growth, dynamically linked libraries, etc
- Shared memory by mapping pages read-write into virtual address space
  - System libraries shared via mapping into virtual address space
  - Pages can be shared during fork(), speeding process creation

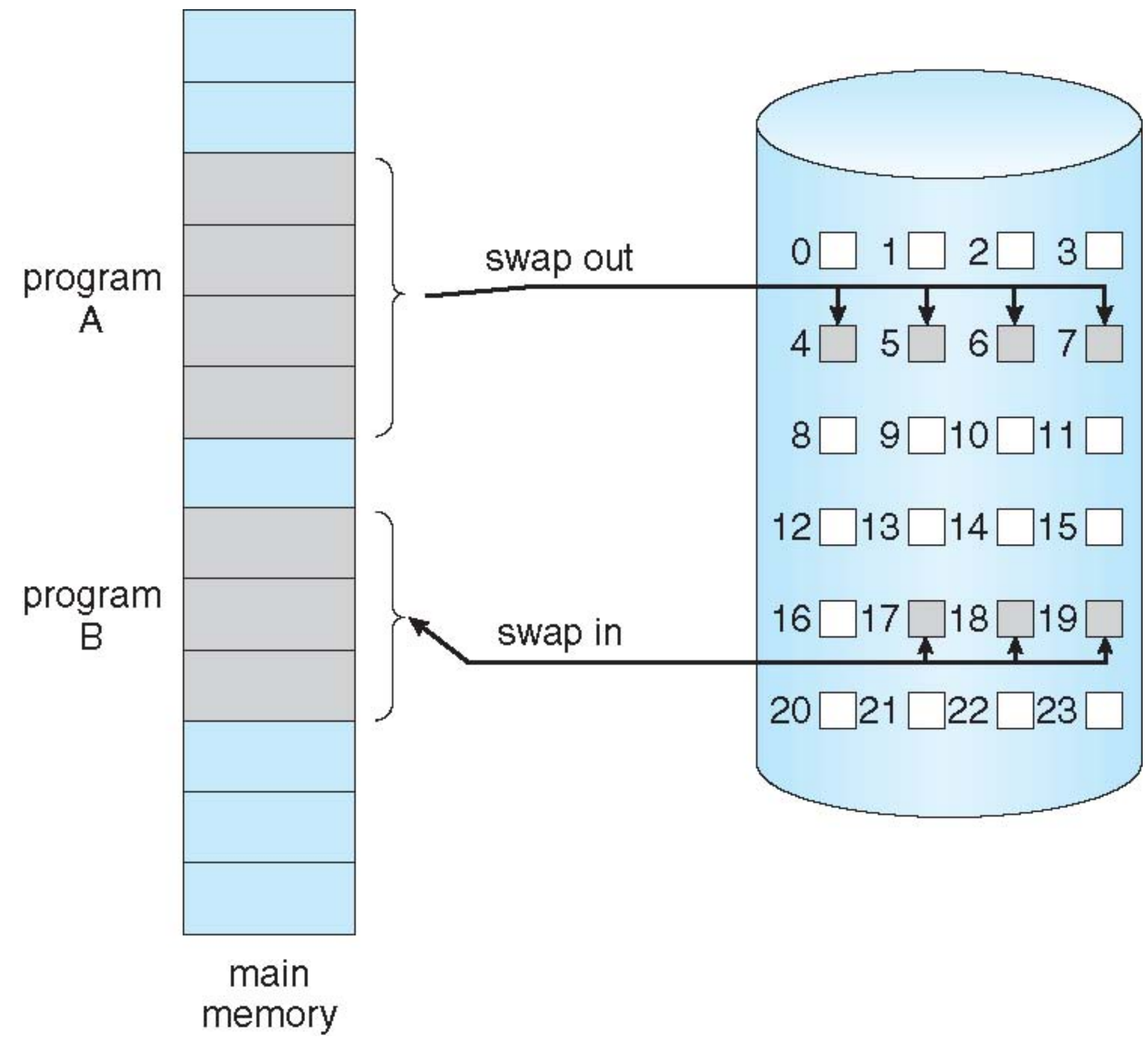


Shared Library Using Virtual Memory



Review: Swapper vs. Pager

- Swap out:
  - move process memory to disk
- Swap in:
  - move saved process from disk to memory
- Swapper that deals with pages is a pager
  - Page-in, Page-out instead of Swap-in, Swap-out



Demand Paging

- A way of implementing virtual memory
=> bring page into memory only when needed
    - page could be program code (read-only) or user data
- Benefits
    - Less I/O needed, no unnecessary I/O
    - Less memory needed
    - Faster response
    - More users

Basic Concepts

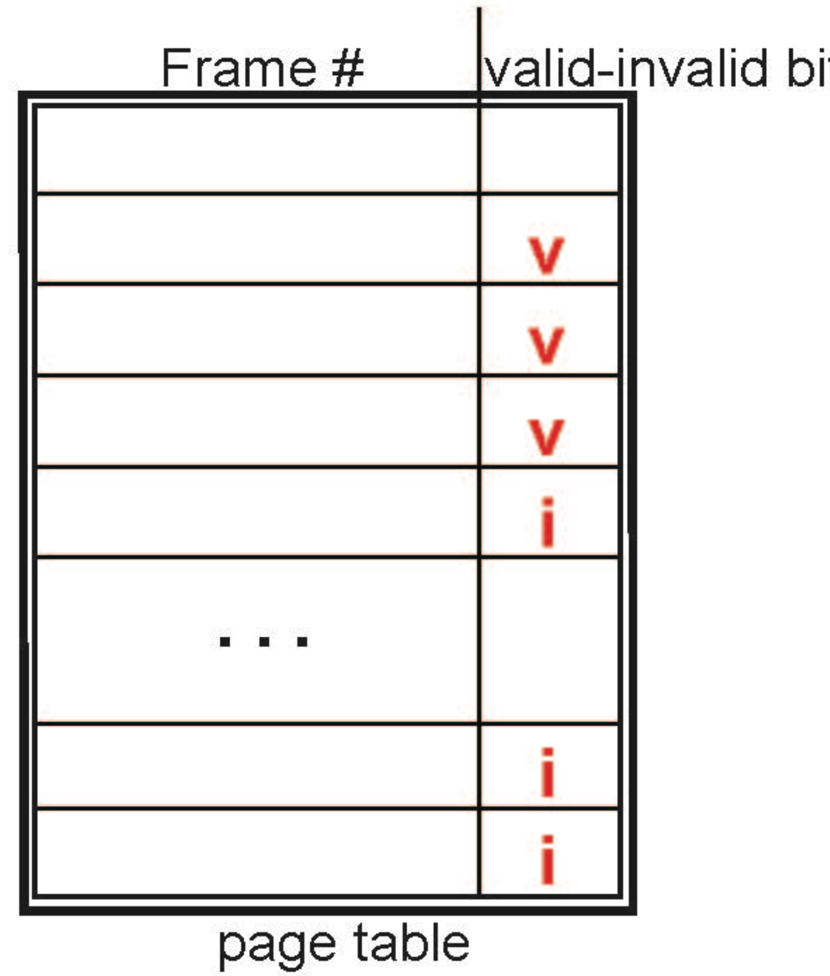
- Need new MMU functionality to implement demand paging
- If pages needed are already memory resident
  - No difference from non-demand-paging
- If page needed but not memory resident
  - Need to detect and load the page into memory from storage
- Abstraction provided by paging
  - Without changing program behavior
  - Without programmer needing to change code

Valid-Invalid Bit

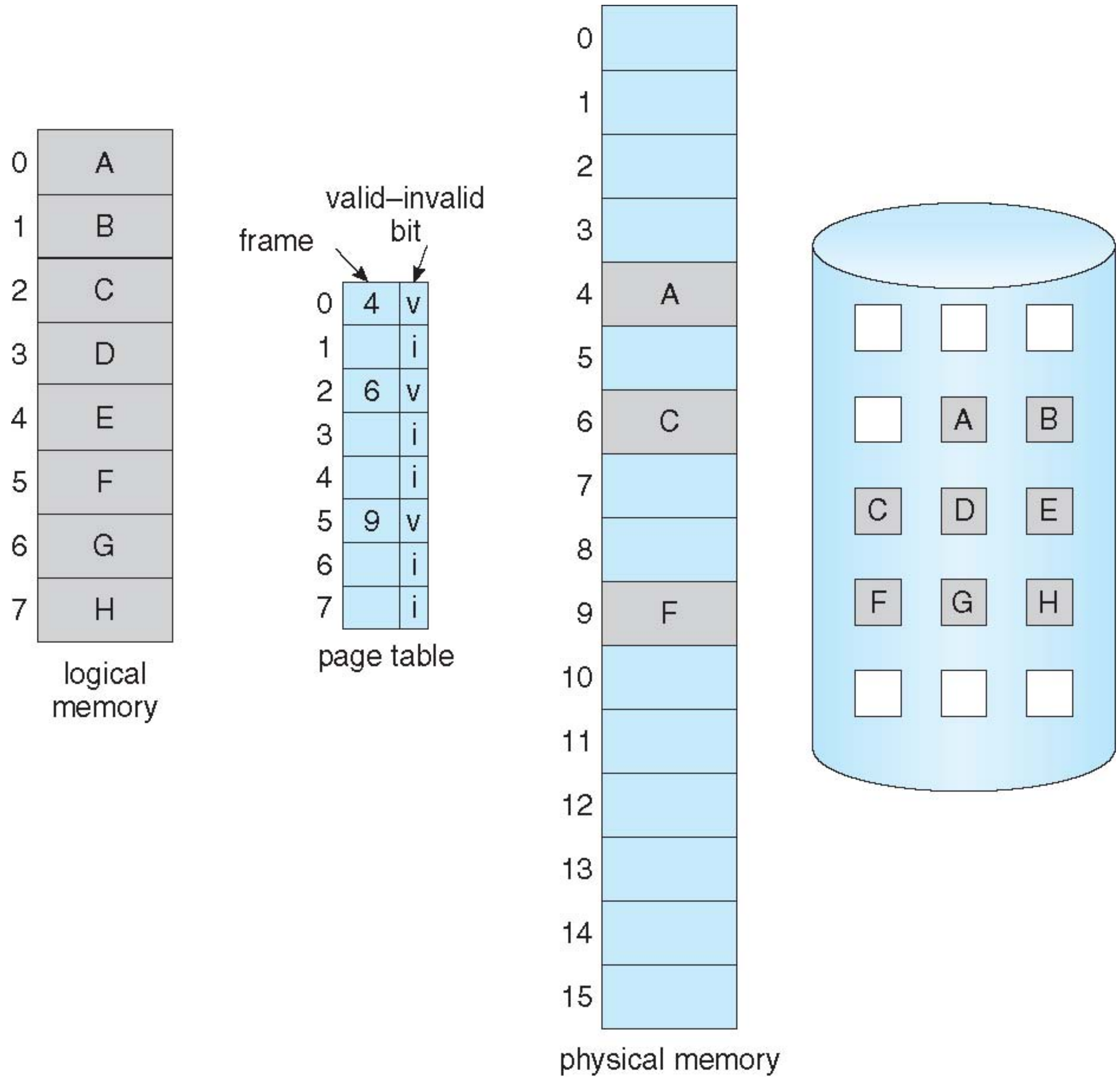
- Bit associated with each page table entry
- 'v' means in-memory
  - proceed
- 'i': two possibilities:
  - invalid reference => abort
  - not-in-memory => page fault, bring to memory

Valid-Invalid Bit

- Initialized to 'i' on all entries
- During MMU address translation, if valid-invalid bit in page table entry is i ⇒ page fault



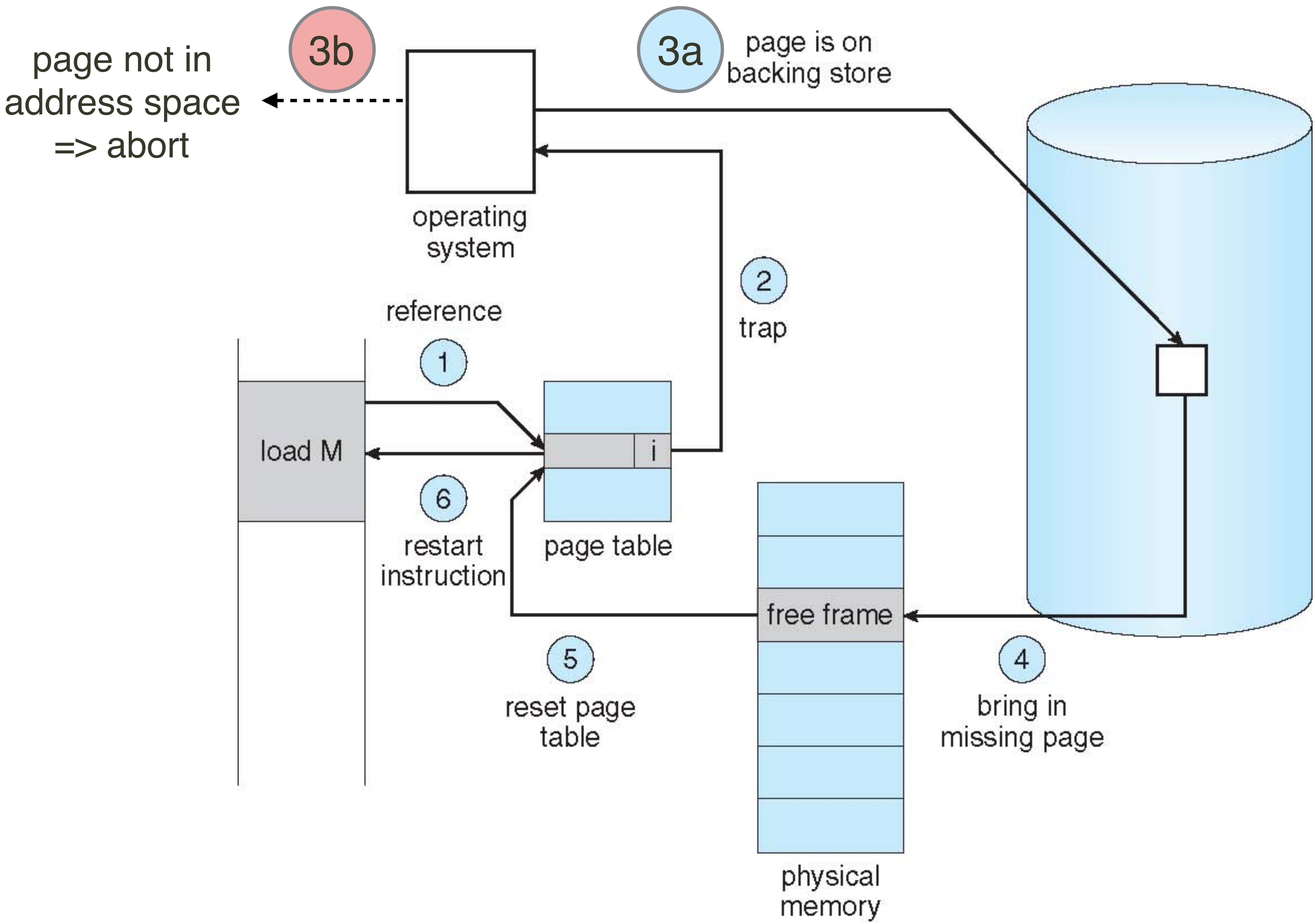
Page table when some pages are not in main memory



Page Fault

- First reference to a page causes page fault
  - trap to OS
- OS looks at another table (usually in PCB) to decide:
  - Invalid reference (outside process's address space) => abort
  - Nonresident page => handle as page fault
- Page fault handling for loading nonresident page from disk:
  - OS finds free frame (e.g, from free-frame list, or kick out some)
  - OS reads page into frame via scheduled disk operation
  - OS updates tables to indicate page now in memory
    Set valid bit = 'v'
  - OS restarts the instruction that caused the page fault

Steps in Handling a Page Fault

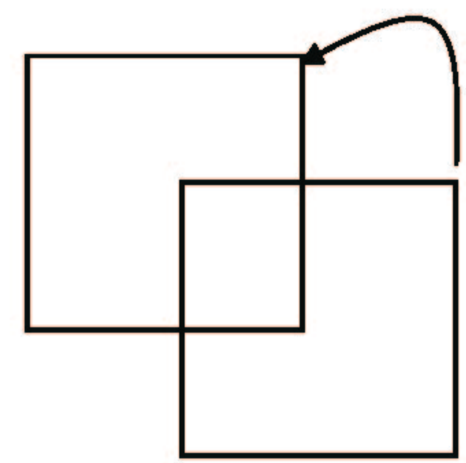


Aspects of Demand Paging

- **Pure demand paging** always get page faults when...
  - First instruction of process
  - First access of any page of the process
- One instruction could cause multiple page faults!
  - Example: add 2 numbers from memory, stores result back to memory
  - Pain decreased because of locality of reference
- Hardware support needed for demand paging
  - Page table in hardware with valid / invalid bit
  - Secondary memory (swap device with swap space)
  - Instruction restart after OS has loaded page into frame

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?

Stages in Demand Paging (worst case)

1. Page fault traps to the OS
2. OS saves the user registers and process state
3. OS checks if legal reference, determines location of page on disk
4. OS issues a read from disk to a free frame:
5. While waiting, OS allocates the CPU to some other user
6. OS gets interrupt from the disk on completion of transfer
7. OS updates the page table
8. OS allocates CPU to this process again
9. OS restores the user registers, process state, and new page table, and then resume the interrupted instruction

Non-Demand Paging vs. Demand Paging

- Non-demand paging
  - entirely transparent to process
  - mechanism between CPU and memory
  - page fault (bit == 'i') is a fatal error!
- Demand paging
  - CPU must support instruction restart after fault handling
  - page fault (bit == 'i') fatal only if outside address space, but not fatal if on-disk and not memory resident

Free-Frame List

- pool of free frames
  - uses linked list structure
- Zero-fill-on-demand
  - upon allocation, initialize entire page to 0
  - reason: privacy protection - don't want previous process's data to be seen by another process

Performance of Demand Paging

- Page fault overhead, excluding swapping
  - Service the interrupt
  - Restart the process
- Most time spent on disk transfer: swap-in and swap-out
- Page Fault Rate $0 \leq p \leq 1$
- Effective Access Time (EAT)

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

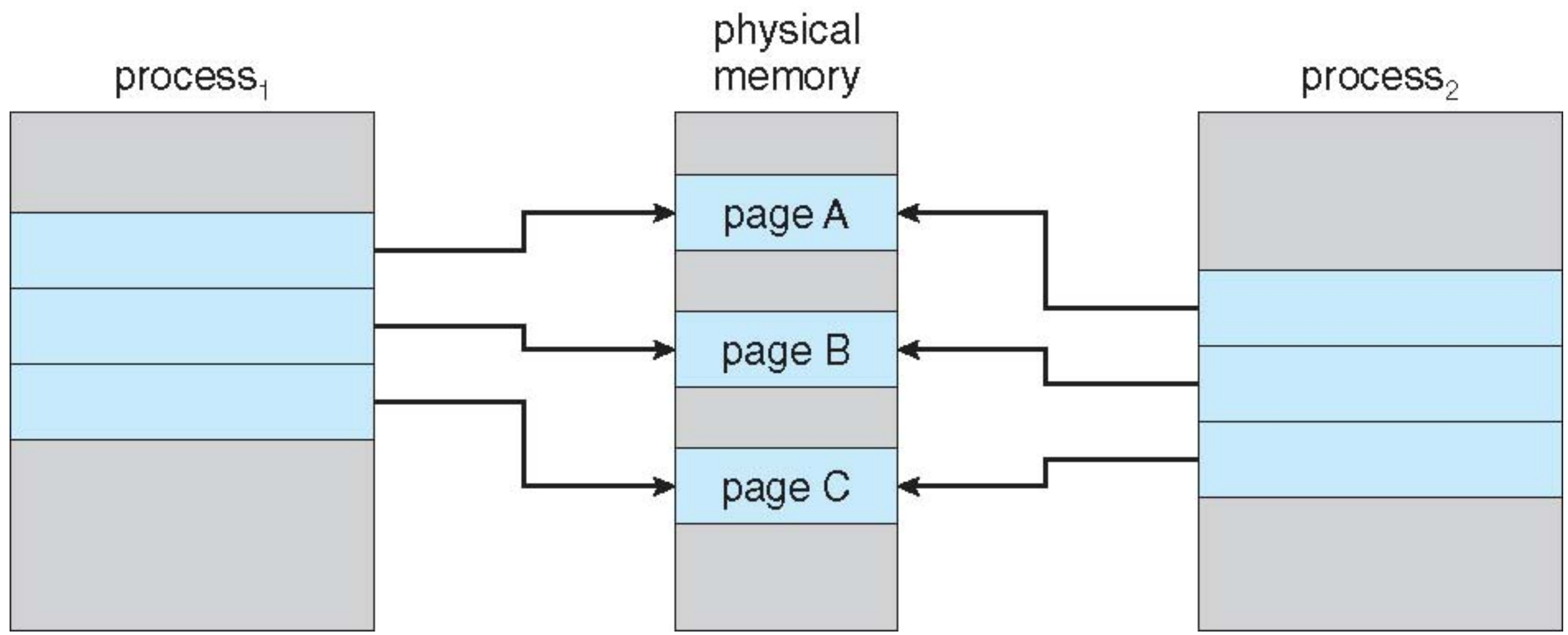
Demand Paging Example

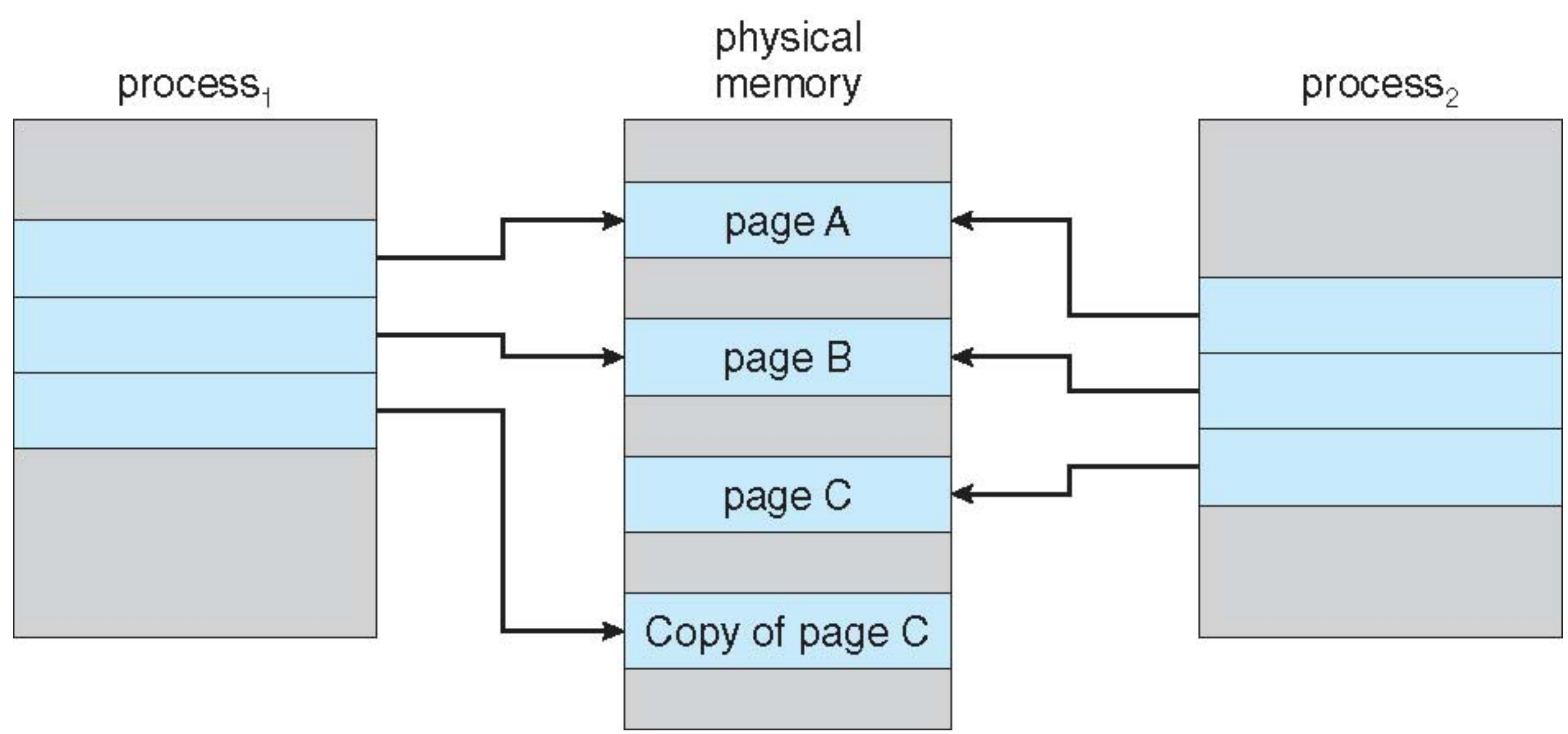
- Memory access time = 200 ns $\qquad$ *p* = page fault rate
- Average page-fault service time = 8 ms
- EAT = (1 – p) x 200 + *p* x (8 ms)
  = (1 – p x 200 + *p* x 8,000,000
  = 200 + *p* x 7,999,800
- If <u>1 in 1000</u> accesses causes a page fault, then
  EAT = 8.2 μs. => slowdown by a factor of 40!!
- If want performance degradation < 10%
  - 220 > 200 + 7,999,800 x p
    20 > 7,999,800 x p => *p* < .0000025
  - < <u>1 in 400,000</u> memory accesses per page fault

Copy-on-Write

- Allows both parent and child processes to initially share the same pages in memory when fork()
  - If either process modifies a shared page => OS makes copy of page first
  - but if no write => no need to copy!
- In general, OS allocates free <u>pages</u> from a pool of zero-fill-on-demand pages
  - Pool should always have free <u>frames</u> for fast demand page execution
  - Don't want to have to "free a frame" or do other processing upon page fault

Copy-on-write





alternative to copy-on-write: vfork()

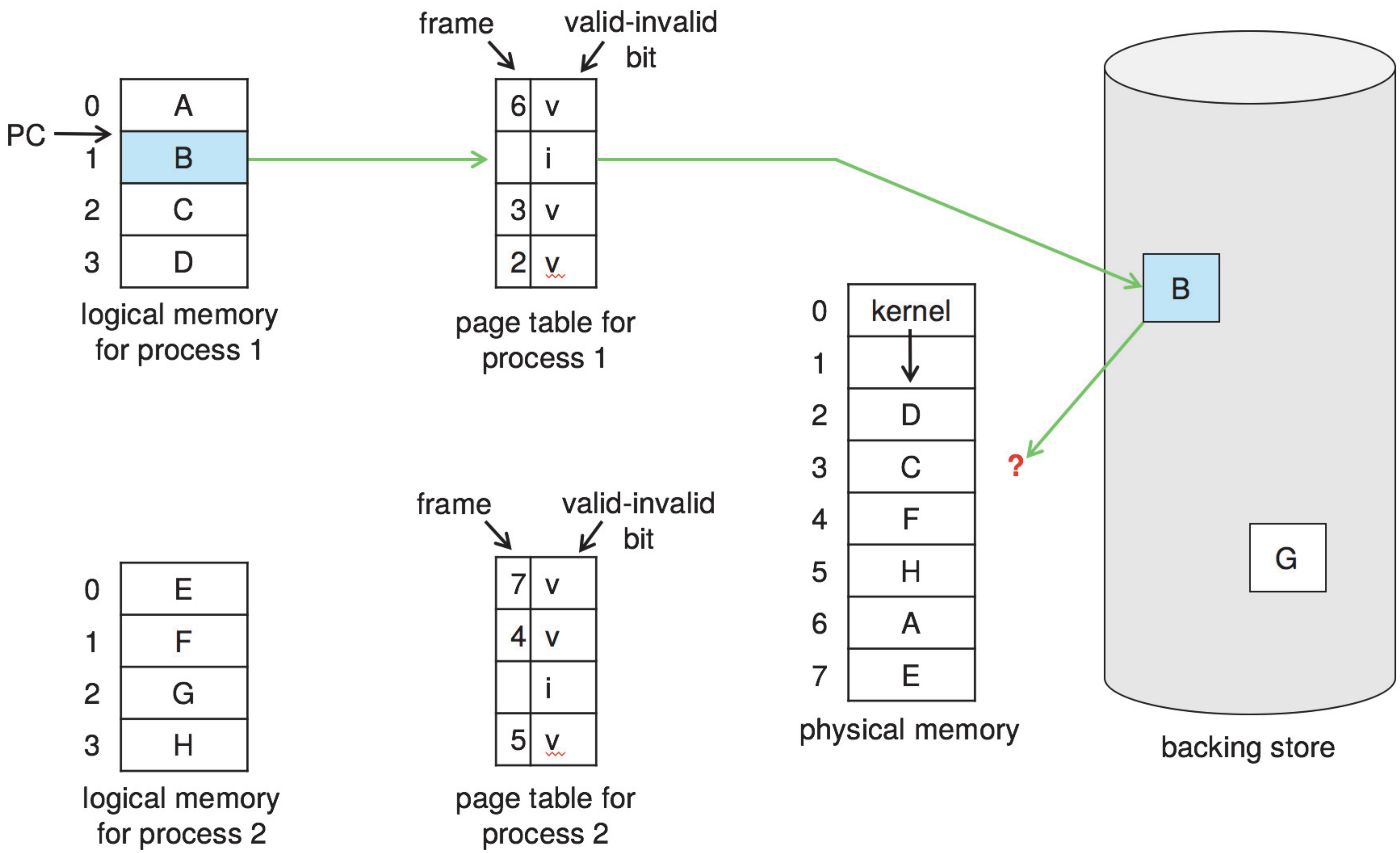
- vfork()
  - OS suspends parent while child uses parent's resources
  - Does NOT use copy-on-write!!!
- child changes will be visible to parent!
  - child needs to be very careful not to modify parent space
  - sharing stops when exec() is called.
- Purpose
  - useful for implementing command-line shells
    => child calls exec() immediately after creation.

Page Replacement

Page Replacement

- Need frame but no free frame available
  - find a *victim* page in memory to page out, free the frame
  - hardware *dirty-bit* (aka *modify bit*) to track modification
    => if dirty when page out, need to save to disk;
    => if not dirty, no need to save to disk (already on disk)
- Two problems in demand paging
  - Frame allocation: determine how many frames to allocate to a process
  - Page replacement: pick which frame to replace

Need For Page Replacement

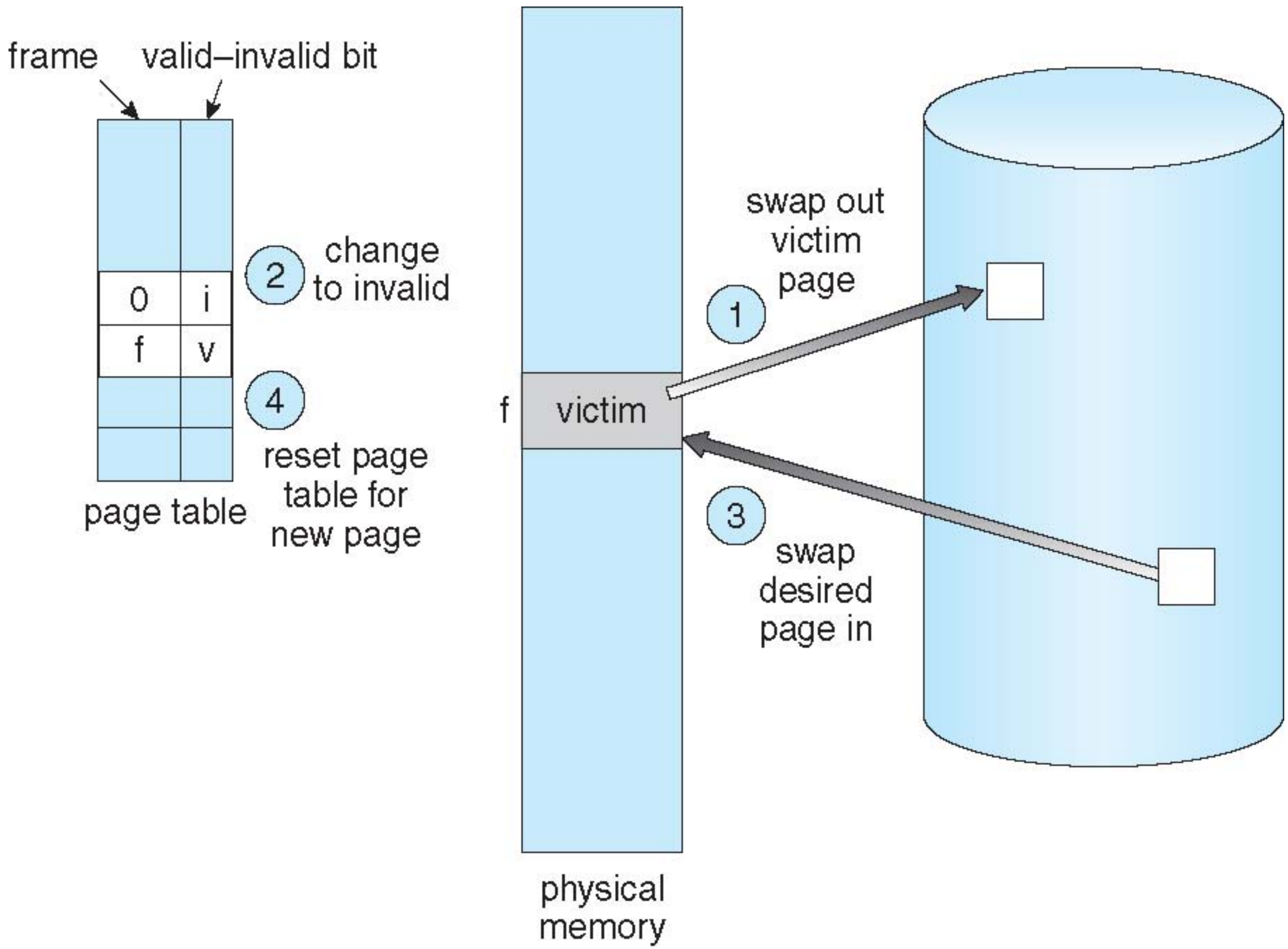


Steps in Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame:
   - If there is a free frame, use it
   - If no free frame, page replacement algorithm selects a **victim frame**
   - Write victim frame to disk <u>if dirty</u>
3. Bring desired page (from disk) into free frame (step2); update the page and frame tables
4. Restart the instruction that caused the trap

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

Page Replacement



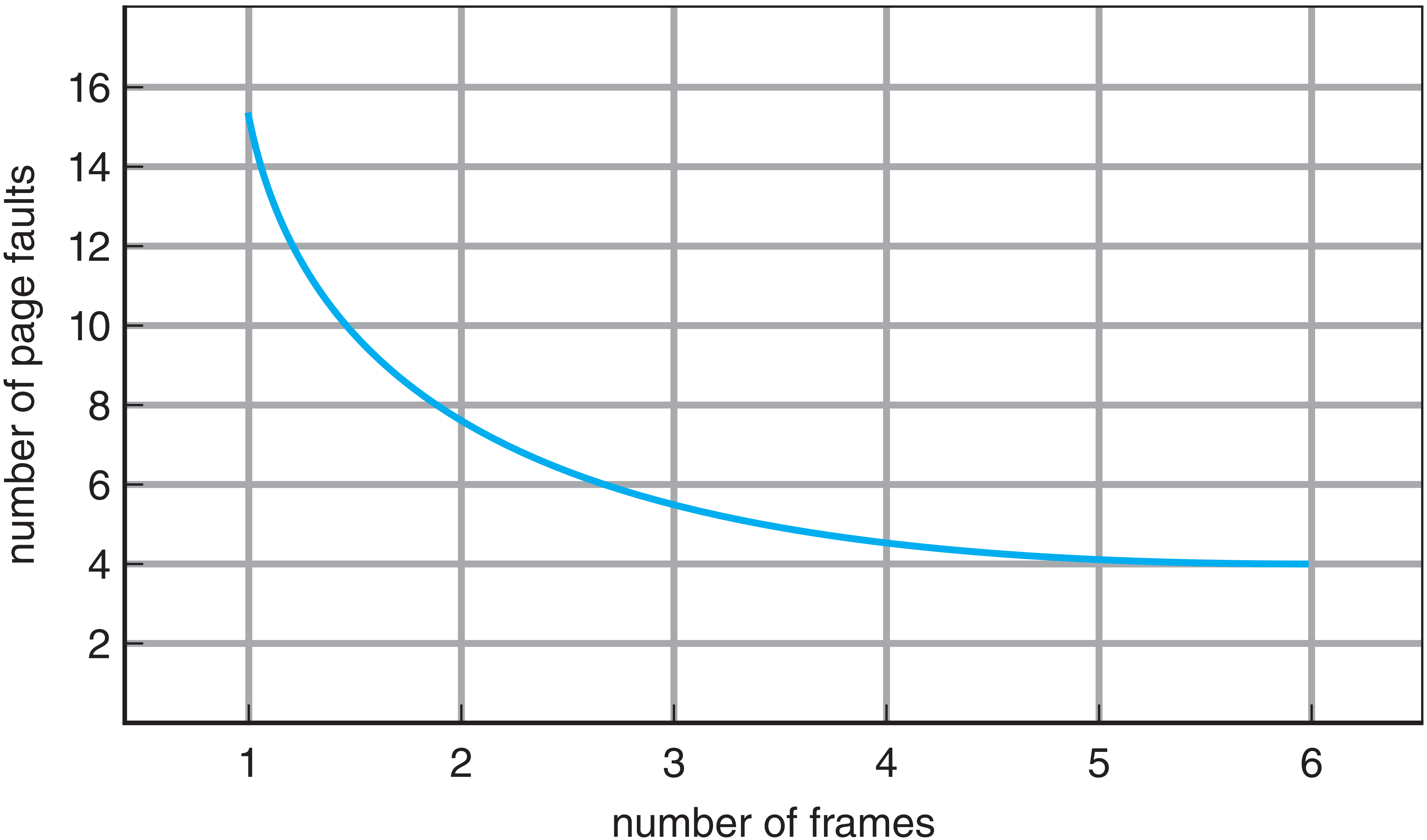
Page and Frame Replacement Algorithms

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

Evaluation of replacement algorithms

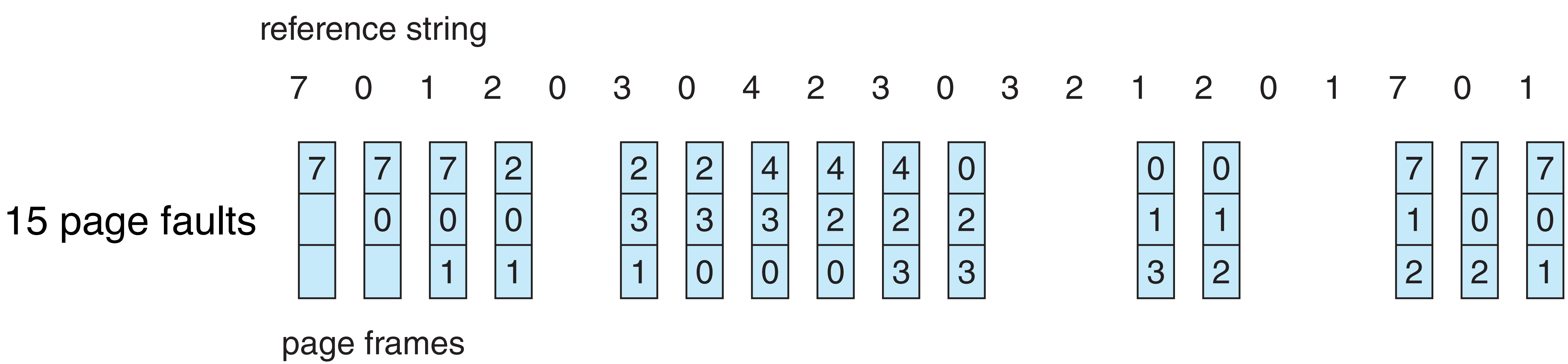
- Run on a particular string of memory references (reference string)
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
  - example: 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1
- Computing the number of page faults on that string
  - Results depend on number of frames available

Page Faults vs. # Frames



First-In-First-Out (FIFO) Algorithm

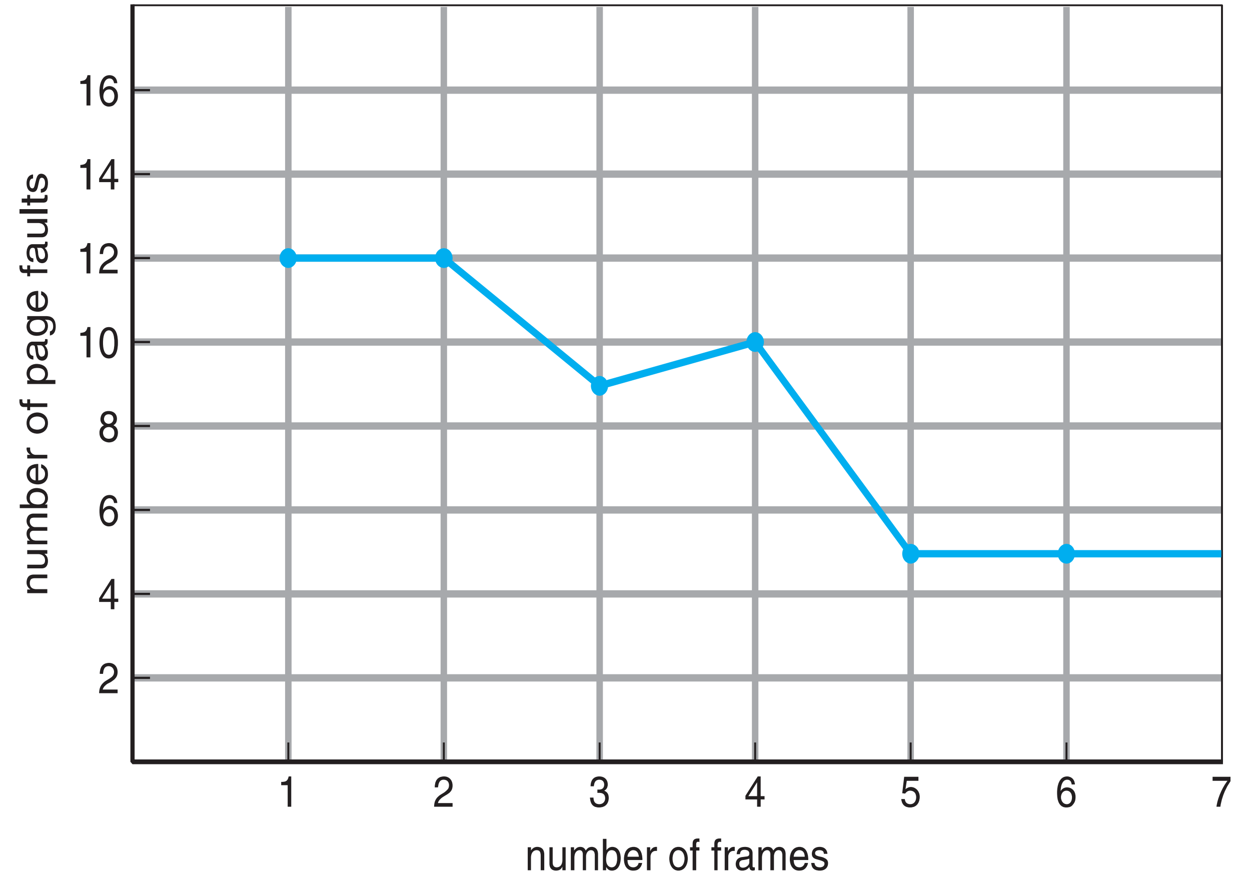
- Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1
  - 3 frames (3 pages can be in memory at a time per process)



  - Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
- To track ages of pages => Just use a FIFO queue

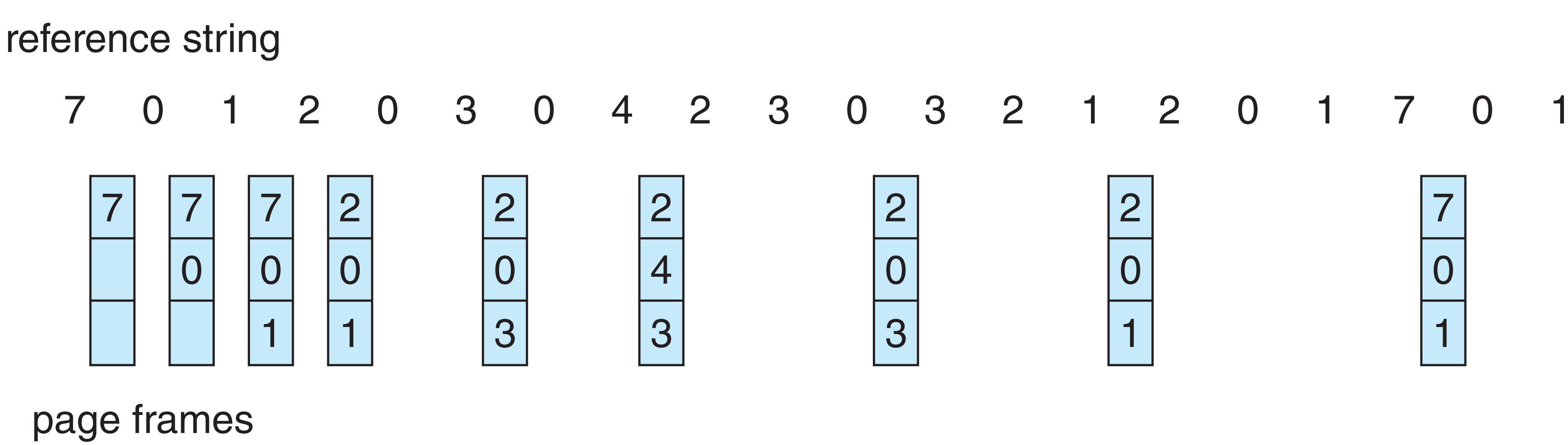
Belady's Anomaly

- Adding more frames can cause more page faults!
  - common in FIFO-based algorithms
- Bélády's /bɛˈleidɪ/
  - Hungarian Computer Scientist
  - IBM; then President & CEO of Mitsubishi Electric Research Labs
  - Known for OPT page replacement algorithm



Optimal page-replacement algorithm (OPT, aka MIN)

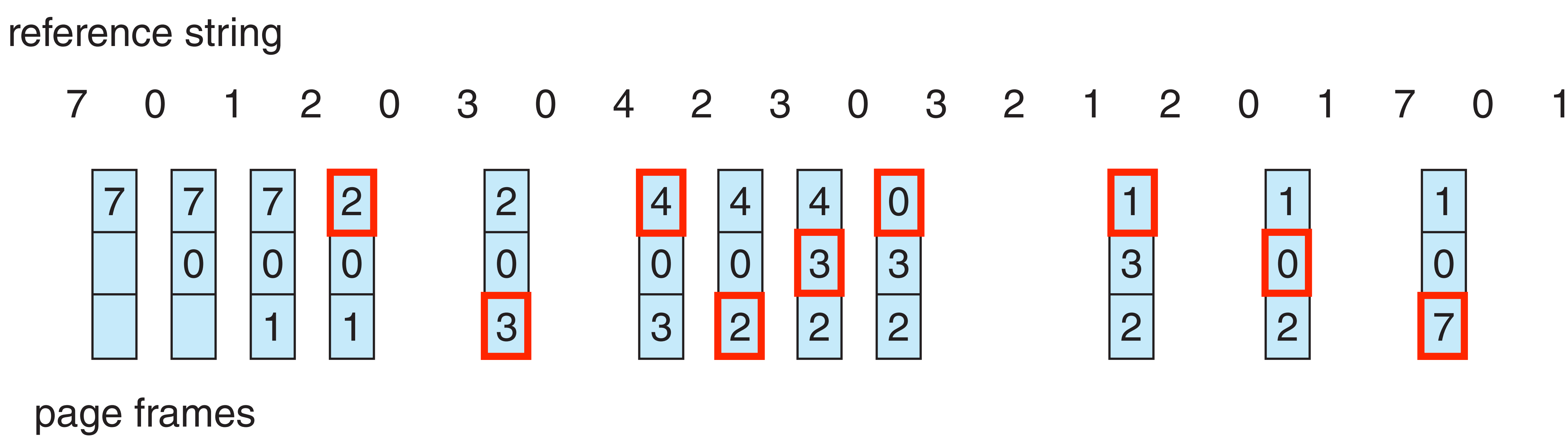
- Replace page that <u>will not be used</u> for longest period of time
  - 9 is optimal for the example
  - Does not have Belady's anomaly
- However, can't read the future…



Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that <u>has not been</u> used in the most amount of time
  - Associate time of last use with each page
  - "use" can be read or write
- Generally good algorithm and frequently used

LRU Example



- 12 faults – better than FIFO but worse than OPT

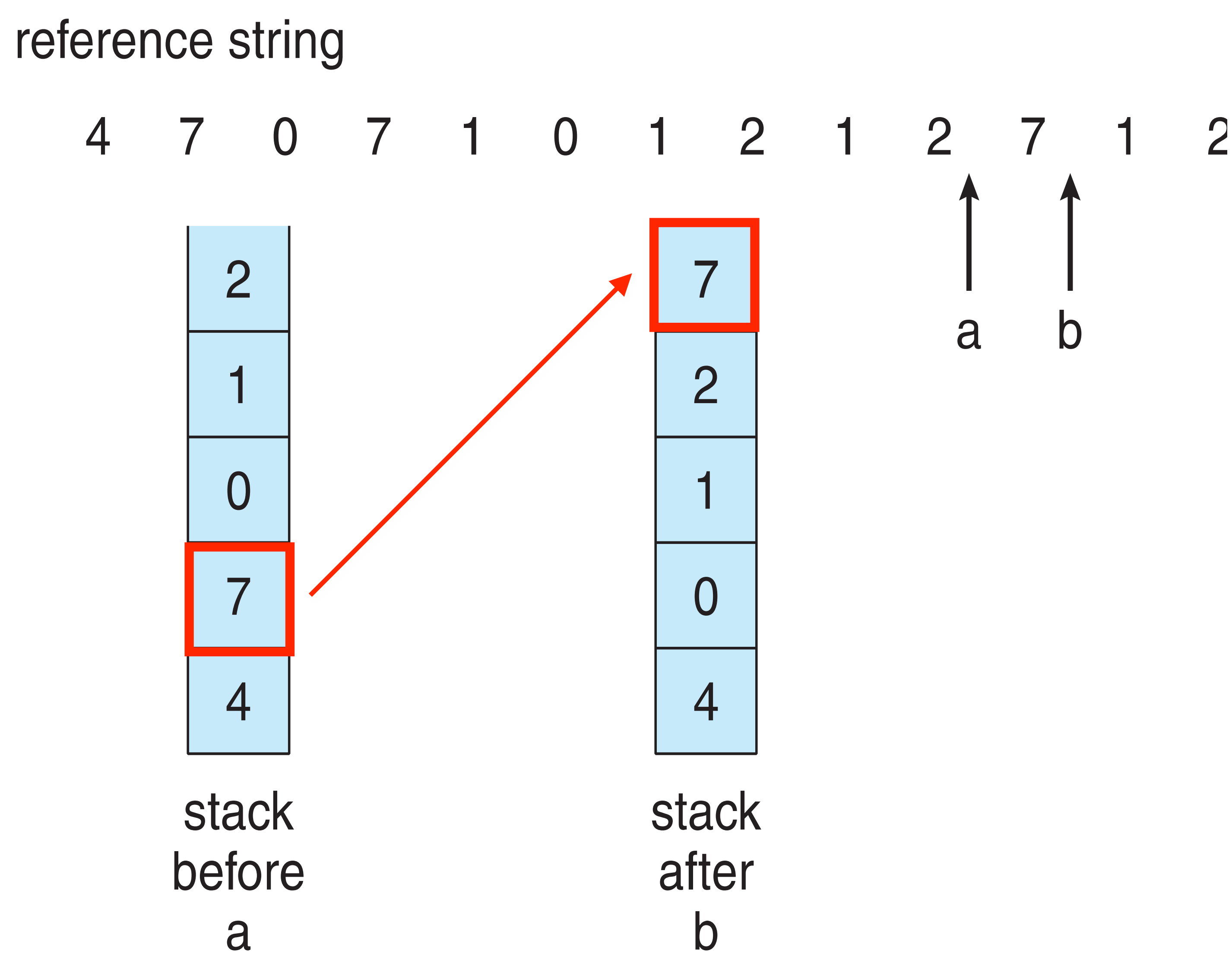
LRU implementation: Counter

- Every page entry has a "counter" ("time stamp")
  - a clock is incremented for every memory reference
  - 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

LRU implementation: Stack Algorithms

- Keep a stack of page numbers in a double link form:
  - Page referenced: move it to the top
- Requires 6 pointers to be changed
  - each update more expensive
  - No search for replacement
- Stack algorithms don't have Belady's Anomaly!
  - Examples: LRU and OPT

Use Of A Stack to Record Most Recent Page References



LRU Approximation Algorithms

- LRU needs special hardware
  - and still slow
- Variations
  - Single Reference bit
  - Additional Reference Bits
  - Second Chance

Single Reference Bit

- Single Reference Bit
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace any with reference bit = 0 (if one exists)
- Rough approximation
  - We don't know the order of use
  - Serves as basis for other algorithms

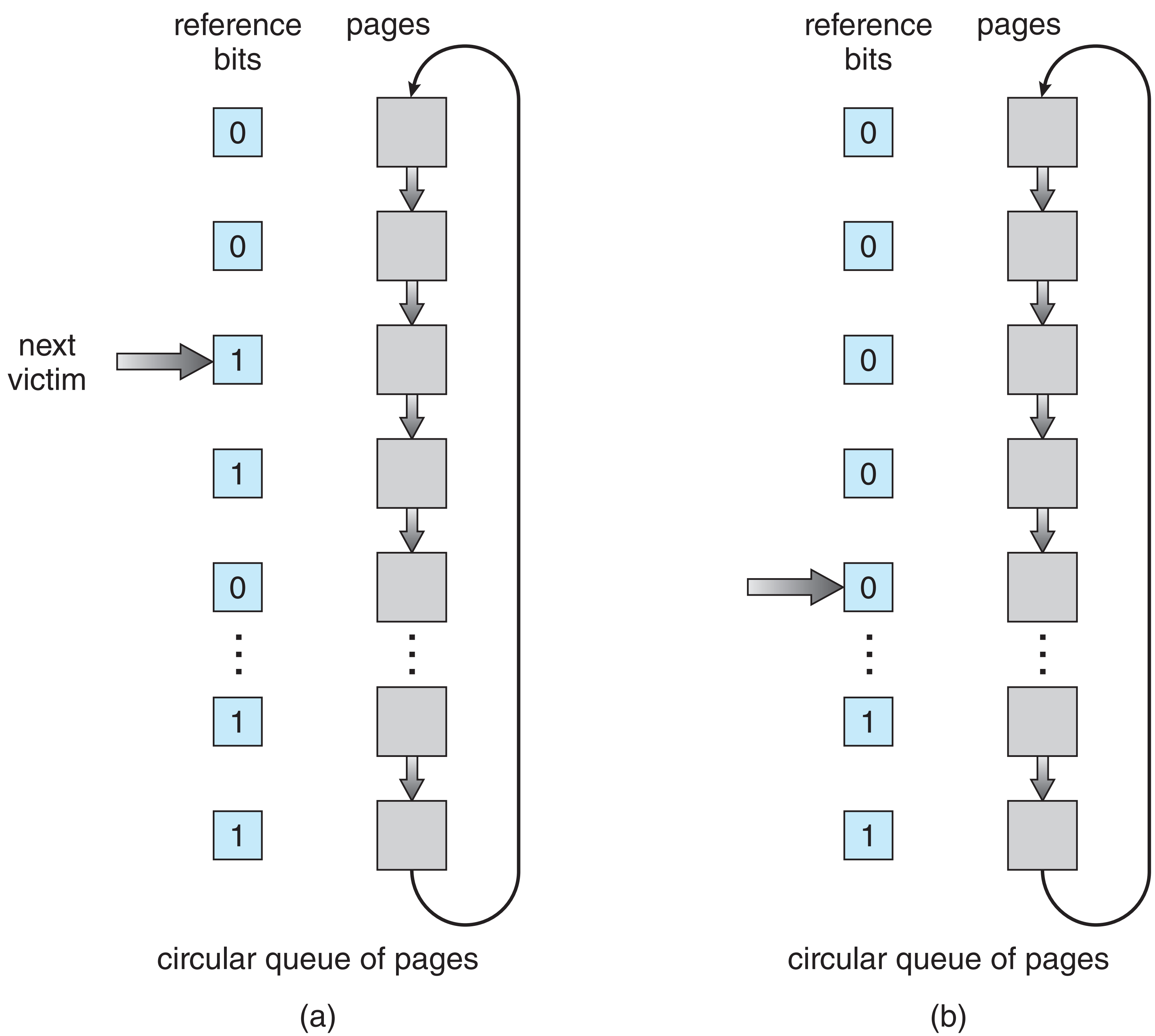
Additional Reference Bits

- e.g., 8 bits (unsigned) of history per page
- Sampled update (e.g., every 100 ms)
  - OS shifts reference bit for each page into 8-bit history (into most significant bit), shift right
  - e.g.,: 0000_0000 => has not been used 8 times
  - 1100_0100 more recent than 0111_0111
- Page with smallest value is picked as victim
  - multiple pages may have same history value...

LRU Approximation: Second-Chance Algorithm

- Generally FIFO, plus hardware-provided reference bit
  - **Clock** replacement
  - Algorithm maintains a pointer in circular order
- If page to be replaced has
  - reference bit = 0 -> found victim, replace it
  - reference bit = 1 then:
    - set reference bit = 0, leave page in memory
    - pointer++ % MEMSIZE, and check again

Second-Chance (clock) Page-Replacement Algorithm



Enhanced Second-Chance Algorithm

- Use both reference bit and modify bit (if available) in concert
- Take ordered pair (reference, modify)
  1. (0, 0) neither recently used nor modified – best page to replace
  2. (0, 1) not recently used but modified – not quite as good, must write out before replacement
  3. (1, 0) recently used but clean – probably will be used again soon
  4. (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement
- When page replacement called for, use the clock scheme but use the four classes to replace page in lowest non-empty class
  - Might need to search circular queue several times

Counting-based 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
  - problem: access may be heavy on startup but rarely used after => large count, can't get replaced easily.
  - Solution: shift count over time = exponential decaying average
- 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

Page-Buffering Algorithms

- Keep a pool of free frames, always
  - frame available when needed, not found at fault time
  - Read page into free frame and select victim to evict and add to free pool
  - <u>When convenient</u>, evict victim, not at the time of victim selection
- Extended idea 1: keep list of modified pages
  - When backing store idle, write pages there and set to non-dirty
- Extended idea 2: Keep free frame contents intact even when on free list 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 knowledge in optimizing Page Replacement

- Want to take advantage of application knowledge in paging
  - OS is just 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
- OS has direct access to the disk, getting out of the way of the applications
  - **Raw disk** mode - bypasses buffering, locking, etc

Allocation of Frames

- Each process needs ***minimum*** number of frames
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*
- ***Maximum*** = total frames in the system
- Two major allocation schemes
  - fixed allocation
  - priority allocation
- Many variations

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 size of process
  - Dynamic as degree of multiprogramming, process sizes change

— s_i = size of process p_i

— $S = \sum s_i$

— m = total number of frames

— a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$m = 64$

$s_1 = 10$

$s_2 = 127$

$a_1 = \frac{10}{137} \times 62 \approx 4$

$a_2 = \frac{127}{137} \times 62 \approx 57$

Priority Allocation

- Could work with equal or proportional allocation
- Define priority on process for allocation
  - higher priority process => likely to get more frames
  - lower priority process => likely to get replaced
- Upon page fault by a process
  - OS may need to select a frame for replacement, if no free frame available
  - => select victim from a process with lower priority

Global vs. Local Allocation upon page fault

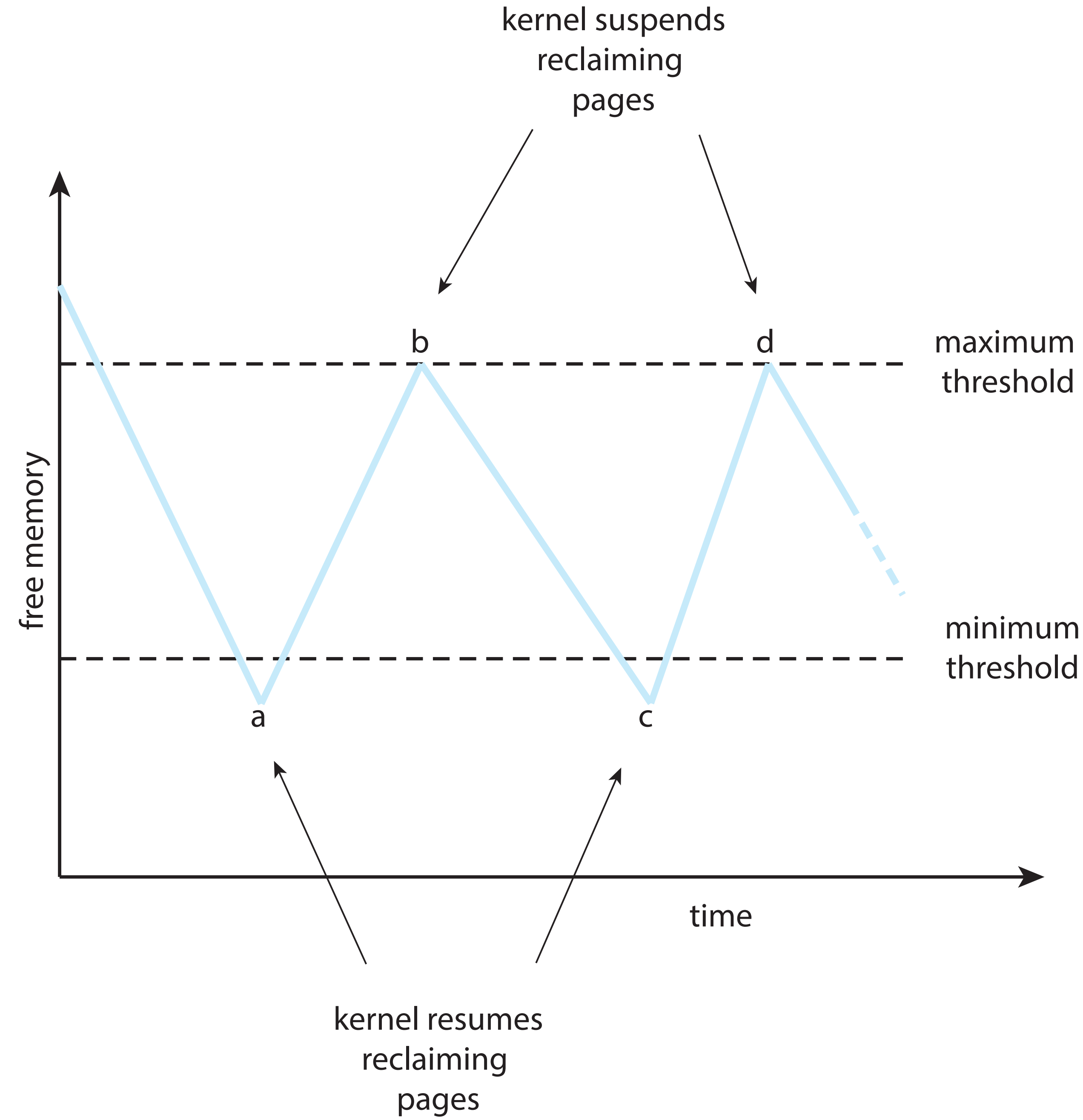
- Global replacement
  - A process can take a frame of another process
  - Advantage: greater throughput, better utilization => more common
  - Disadvantage: less predictable execution time
- Local replacement
  - A process replaces its own set of allocated frames
  - Advantage: More consistent per-process performance
  - Disadvantage: possibly underutilized memory

Major vs. Minor Page Faults

- Major fault
  - page is not in memory
  - need reading from backing store into free frame
- Minor fault
  - page is in memory but process does not have mapping
  - shared library - just update table
  - page on free frame list but not yet assigned to another page - content still intact!!

Reapers

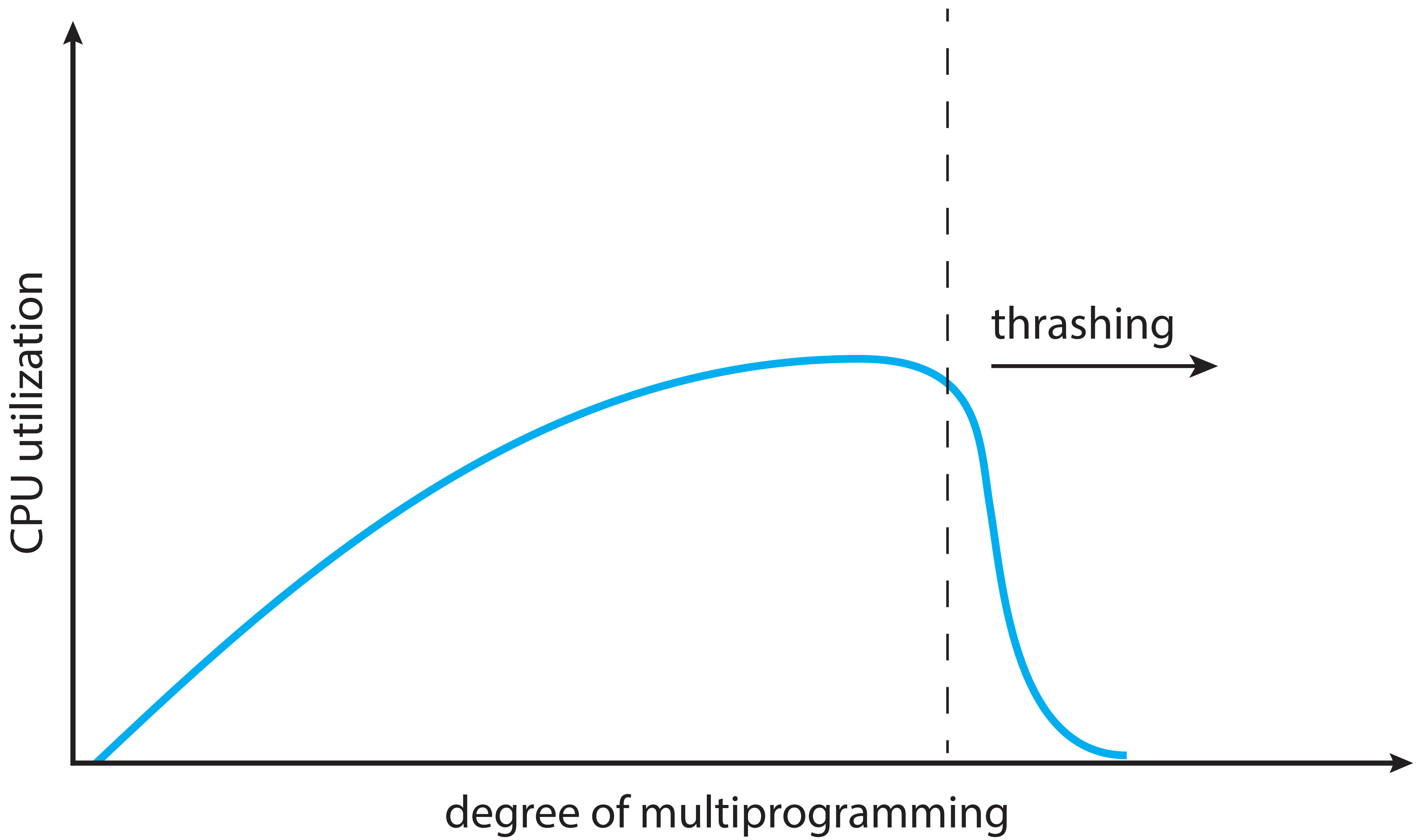
- kernel routines that reclaim pages
  - triggered when amount of free memory drops below some threshold
  - adds frames to free frame list
- May use different replacement policies
  - e.g., normally second chance, but when very low, switches to FIFO



Thrashing

- a process is busy swapping pages in and out
  - not getting real work done!
- Cause
  - a process does not have "enough" frames => high page-fault rate
  - Page fault to get page => Replace existing frame => need replaced frame back
- Potentially vicious cycle
  - Low CPU utilization
    => OS thinks it needs to increase the degree of multiprogramming
    => Another process added to the system

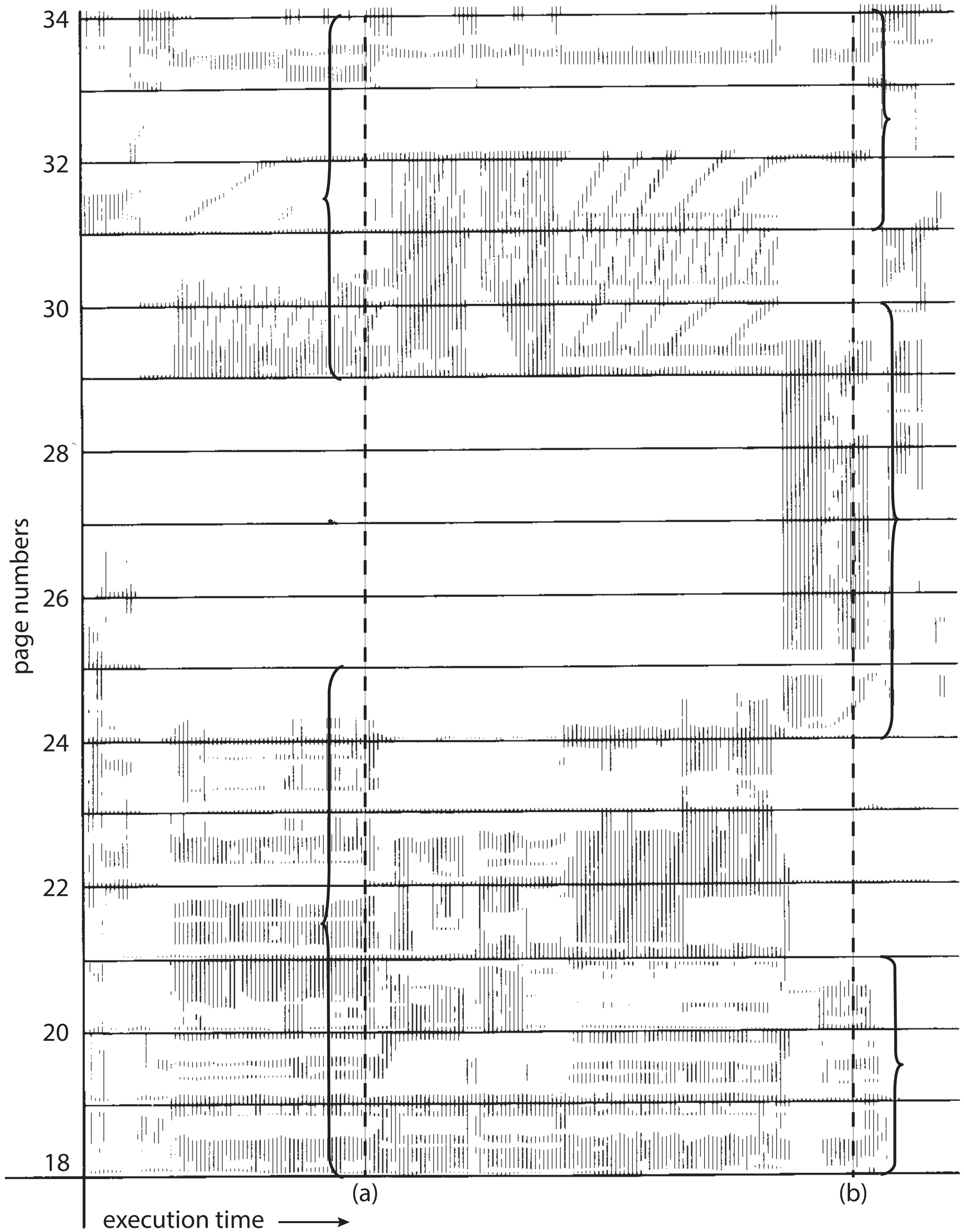
Thrashing (Cont.)



Demand Paging and Thrashing

- Demand paging depends on **Locality**
  - 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

Locality in a Mem.-Ref. Pattern

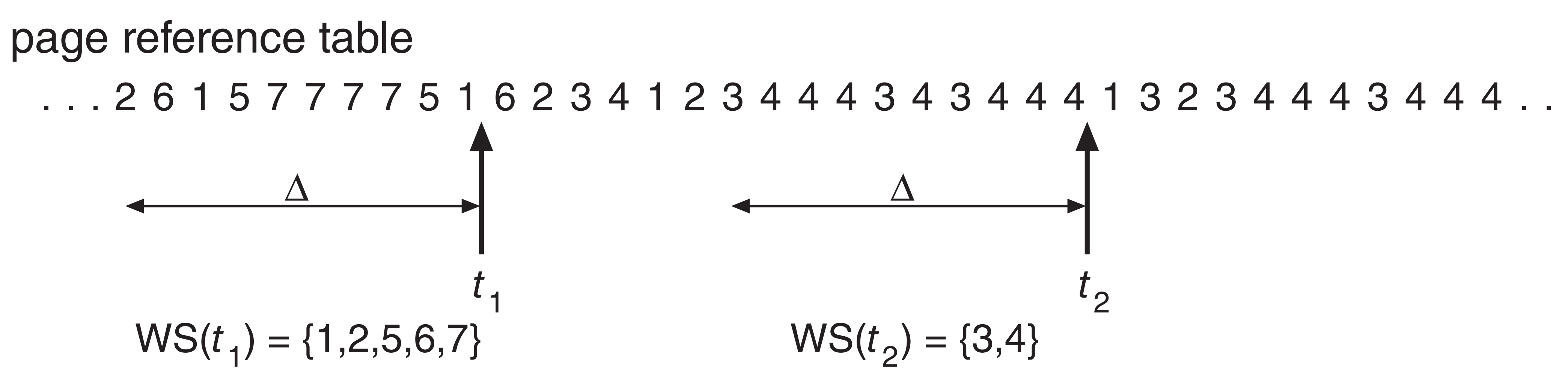


Working-Set Model

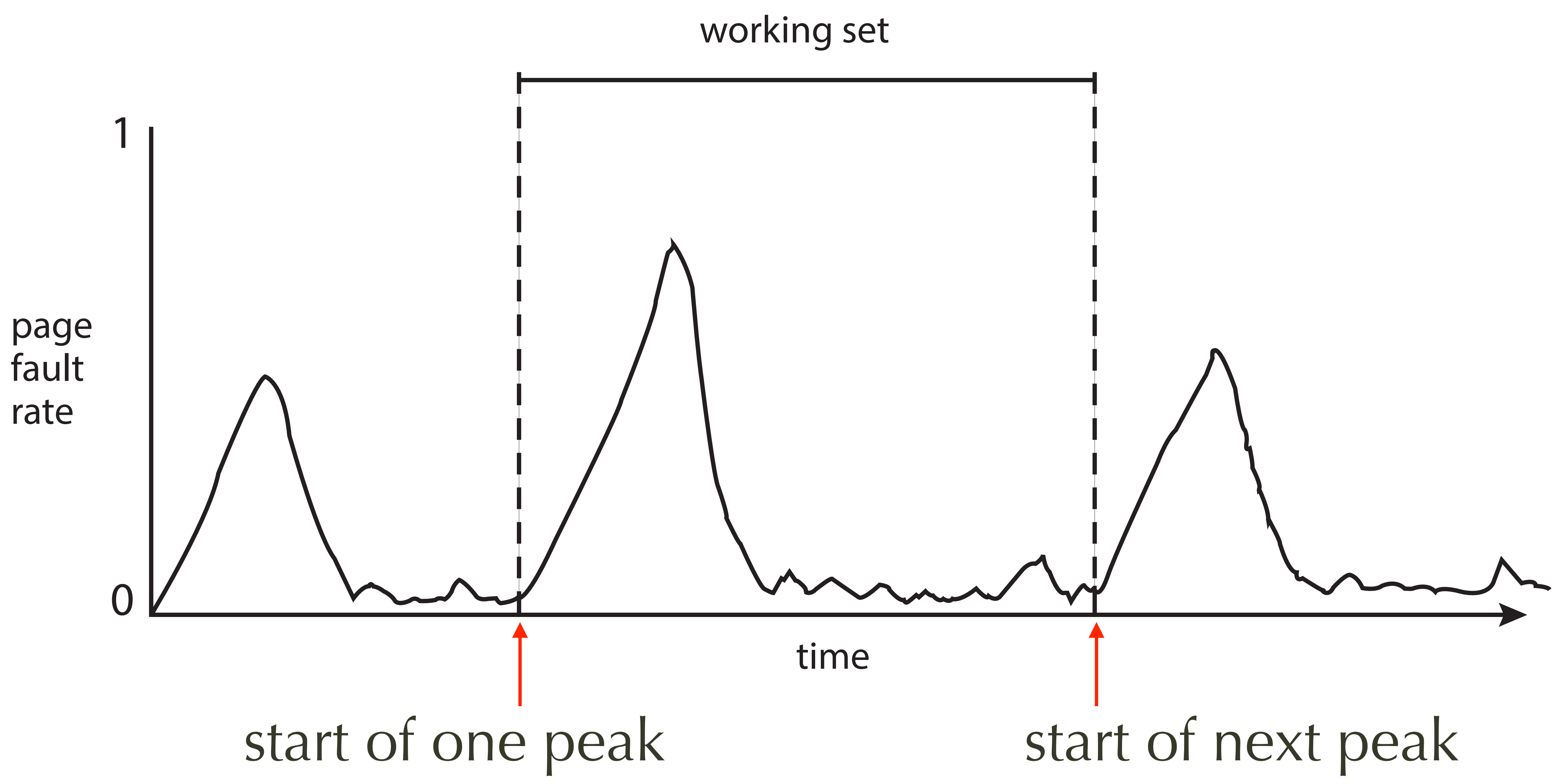
- $\Delta \equiv$ working-set window
  $\equiv$ a fixed #of page references in a time window
  - Example: 10,000 instructions
- WSS_i (working set of Process P_i)
  - = total #distinct pages referenced in the most recent Δ (varies in time)
  - if Δ too small => will not encompass entire locality
  - if Δ too large => will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program

Working-Set Model (cont'd)

- $D = \Sigma\, WSS_i \equiv$ total demand frames
  - Approximation of locality
- if $D > m \Rightarrow$ Thrashing (m = #avail. frames)
- Policy:
  - if $D > m$, suspend or swap out a **process**

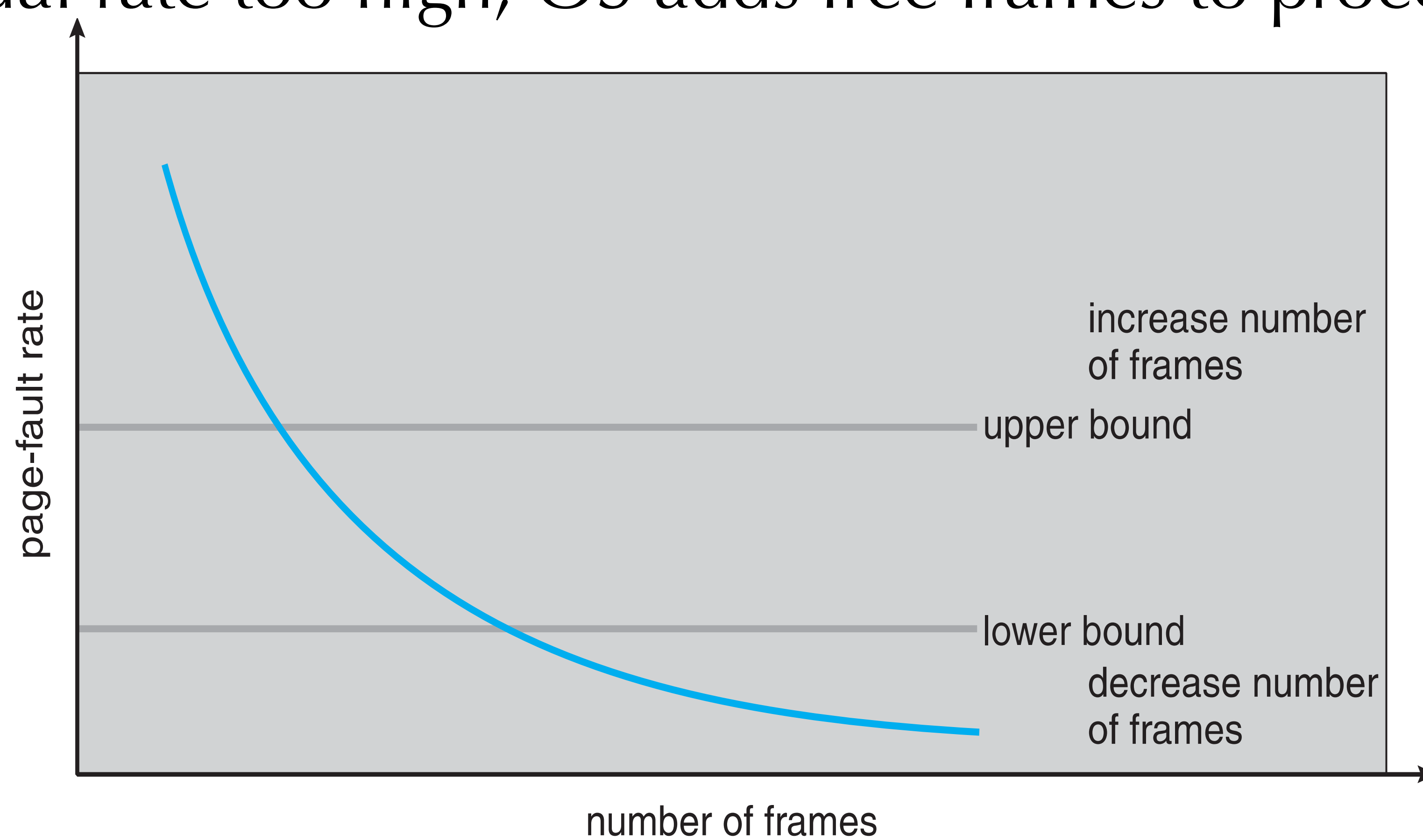


Transition from one working set to another



Page-Fault Frequency

- More direct approach than WSS
- Establish "acceptable" page-fault frequency (PFF) rate and use local replacement policy
  - If actual rate too low, OS takes frames from process
  - If actual rate too high, OS adds free frames to process



Allocating Kernel Memory (physically contiguous)

Allocating Kernel Memory

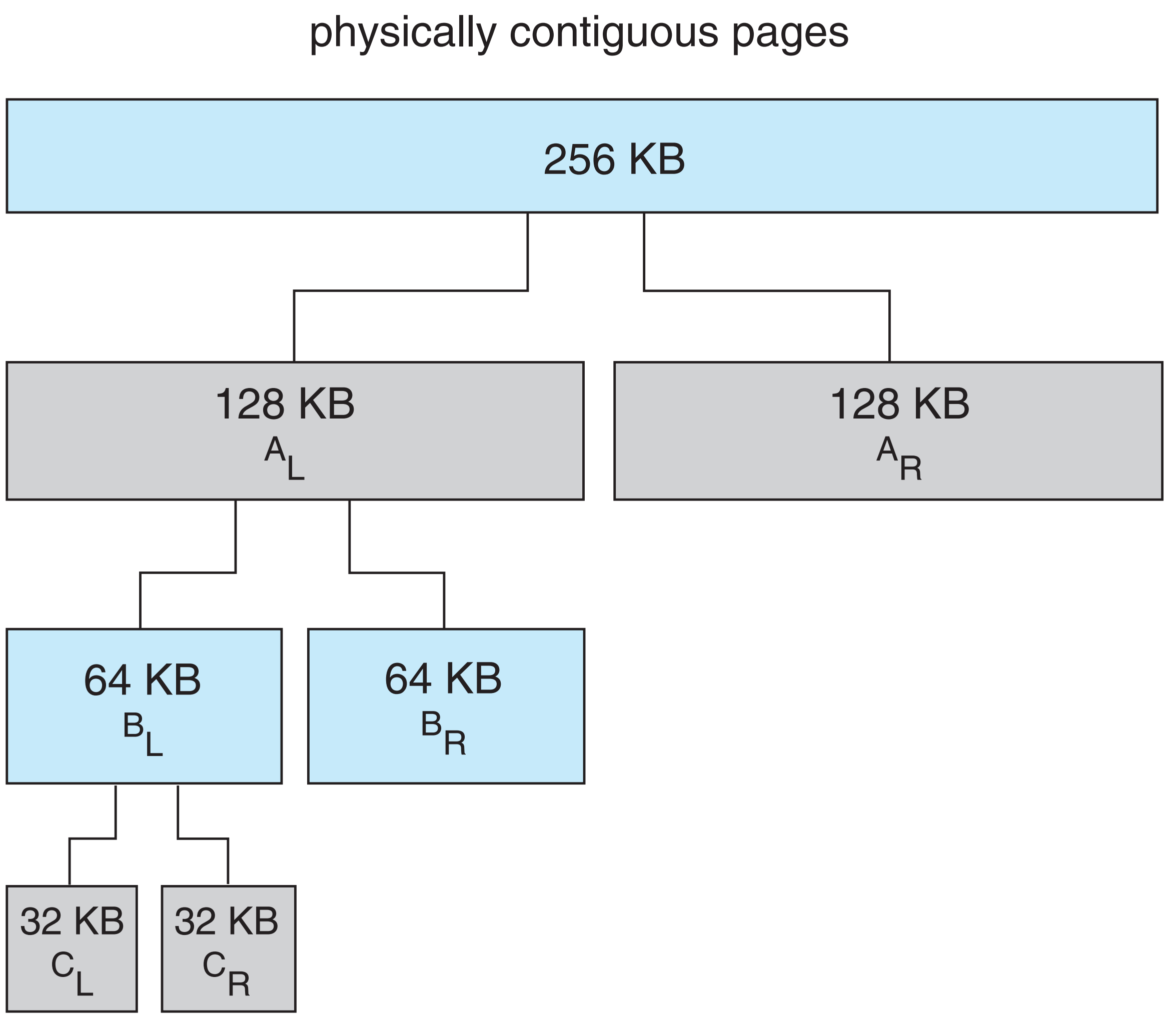
- Some kernel memory needs to be contiguous
  - i.e., for device I/O, hardware DMA
  - user process: logically contiguous, but not physically
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
- Strategies
  - buddy system
  - slab allocation

Buddy System

- using **power-of-2 allocator**
  - from fixed-size segment, physically-contiguous memory
- Properties
  - in units sized as power of 2
  - request rounded up to next highest power of 2
- When smaller allocation needed than is available
  - split current chunk into two buddies of next-lower power of 2
  - Continue until appropriate sized chunk available

Buddy System - example

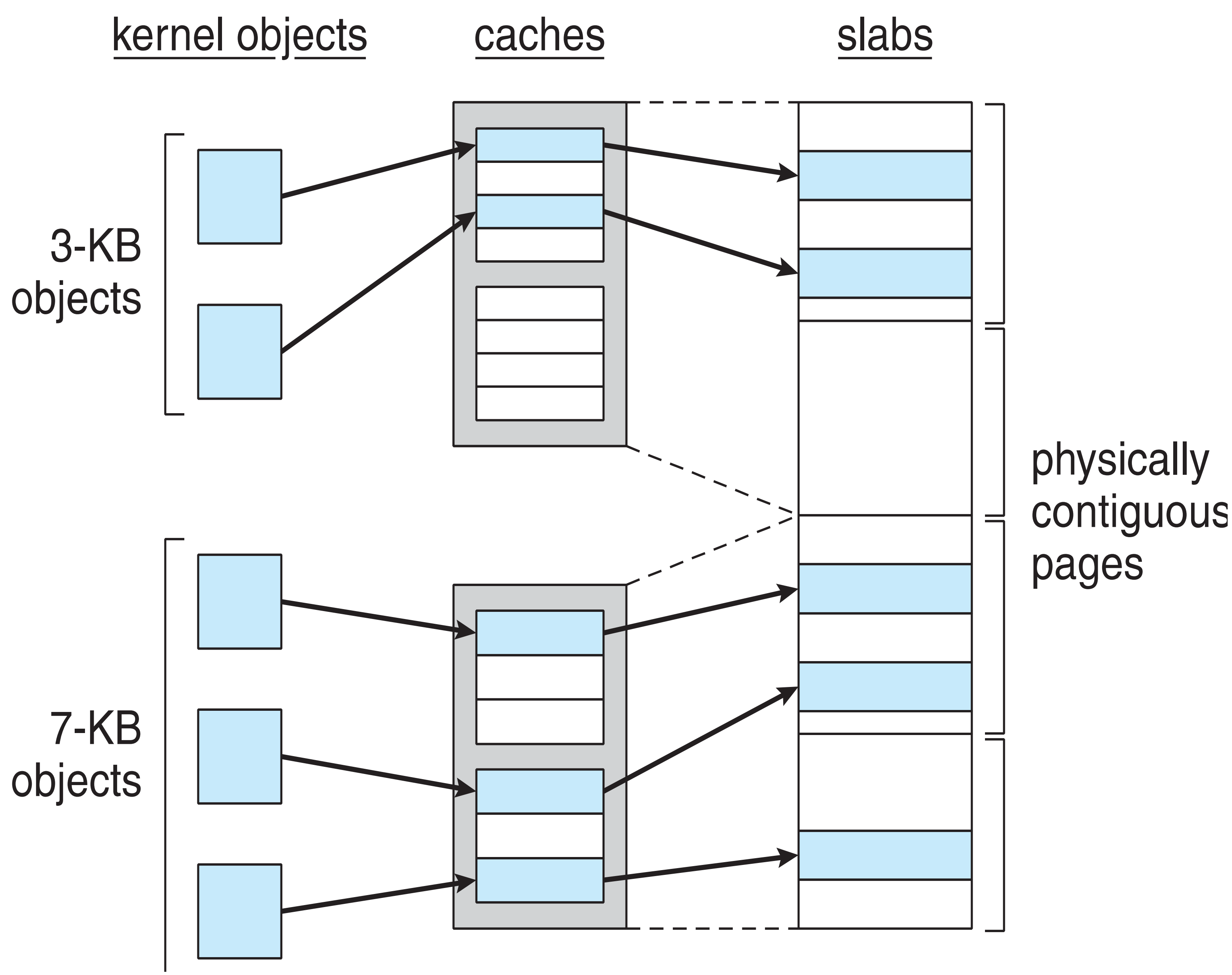
- 256KB chunk available, kernel requests 21KB
  - Split into A_L and A_R of 128KB each
  - One further divided into B_L and B_R of 64KB
  - One further into C_L and C_R of 32KB each – one just large enough used to satisfy request
- Advantage: quickly **coalesce** unused chunks into larger chunk
- Disadvantage: fragmentation



Slab Allocator

- Slab = one or more physically contiguous pages
  - Big enough to contain one or more instances of a given type of kernel data structure
- Cache = one or more slabs
  - One cache for each unique type of kernel data structure
  - e.g., PCB, semaphores, file descriptors, ...
- Objects = instantiations of the data structure
  - Initially, cache is filled with objects marked as free
  - When structures stored, objects marked as used

Slab Allocation



Slab Allocation

- Allocation
  - if slab of given type is full of used objects, allocate next object from empty slab
  - If no empty slabs, allocate new slab
- Benefits
  - no fragmentation - granularity is object, not page or buddy chunk
  - fast memory request satisfaction - recycle object memory through cache

SLAB, SLOB, SLUB in Linux

- SLOB: (list of simple objects)
  - K&R allocator (1991-1999)
  - small, medium, and large objects, first-fit
- SLAB (in Linux)
  - Solaris type allocator (1999-2008)
  - SLAB: As cache friendly as possible. Benchmark friendly.
- SLUB: (Linux 2.6.24) - replaces SLAB
  - Unqueued allocator (2008-today)
  - Simple and instruction cost counts.
  - Superior Debugging. Defragmentation. Execution time friendly.

Other Issues – Program Structure

- 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