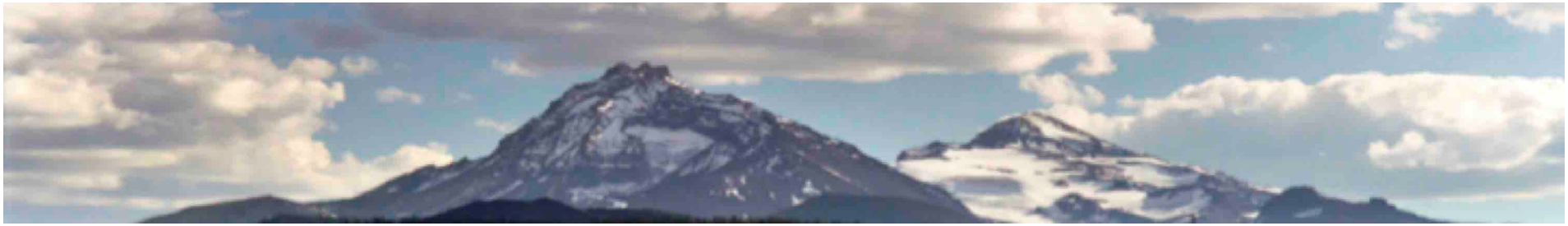


CS 415
Operating Systems
Virtual Memory

Prof. Jieyang Chen
Department of Computer and Information Science
Fall 2025

UNIVERSITY OF OREGON

Logistics

- Read Chapter 10

Outline

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

Objectives

- To describe the benefits of 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
- To examine the relationship between shared memory and memory-mapped files
- To explore how kernel memory is managed

Memory Management Review

- *Logical address space*: range of addresses that a process **can** reference (e.g., $0 \leq r \leq 2^{32}-1$, for 32-bit addresses)
- *Logical memory*: amount of data at addresses that a process **does** reference (e.g., 1,037,944 bytes)
  - Logical memory size ≤ logical address space
- OS memory management is trying to allocate physical memory to satisfy the logical memory requirements
- Assumption thus far is that we are allocating memory for the **entire** logical memory of a process
  - Contiguous allocation
  - Non-contiguous allocation
  - What is we do not?

Background

- Code needs to be in memory to execute, but entire program is rarely used
  - Error code, unusual routines, large data structures
  - Entire program code not really needed all the time
- What about the program's data?
  - Is all the data the program ever uses needed all the time?
- Consider executing a *partially-loaded* program
  - Only part of the program's code and data is in memory
  - Each program has less than all its memory while running
  - This allows better memory management outcomes
    - more programs can run at the same time (have more memory free)
    - increases CPU utilization and throughput
  - Will need to manage the memory allocated to a program!

Memory Allocation/Management per Process

- We want to be able to run a process when it does not have ALL of its code/data (that it will ever use) in memory at the same time
  - Each process will receive only an allocation
  - It will be less than what the process uses in total
- Problems

  1) In addition to global memory management, each process allocation will need to be managed

  2) What if a process reads/writes a logical memory address which has no physical memory assigned?

Virtual Memory

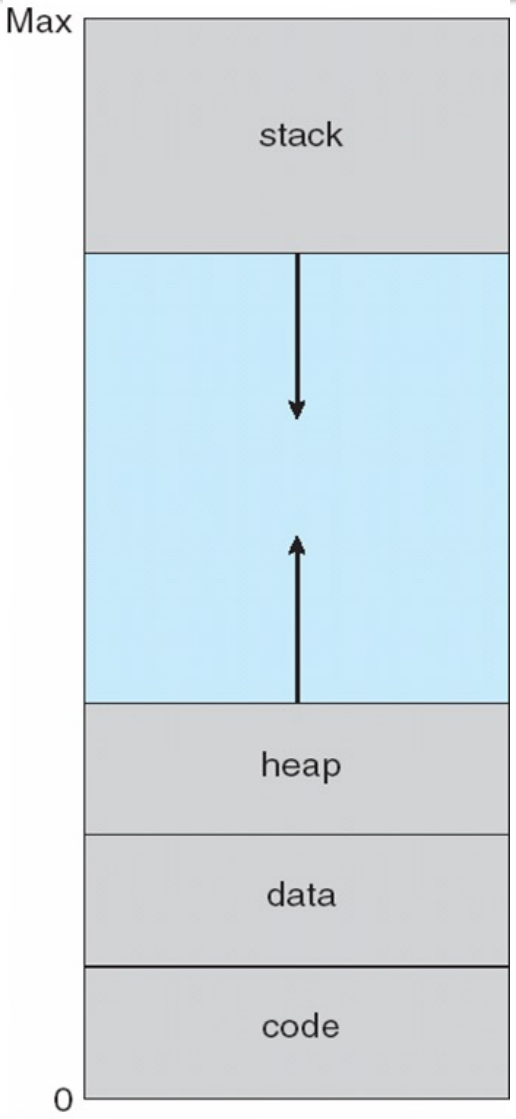
- Suppose you can figure out how to have only part of the program's logical memory in physical memory for execution
  - Can break dependence of logical memory size from physical memory size
- Several benefits
  - Programs can have memory requirements larger than the actual physical memory
  - Allows for more efficient process memory allocation
  - Allows more programs/processes to run concurrently
  - Less I/O needed to load or swap processes
- We call this "memory" *virtual memory*
  - Memory provided to the process for its execution
  - Beyond limits of physical memory

Virtual Address Space

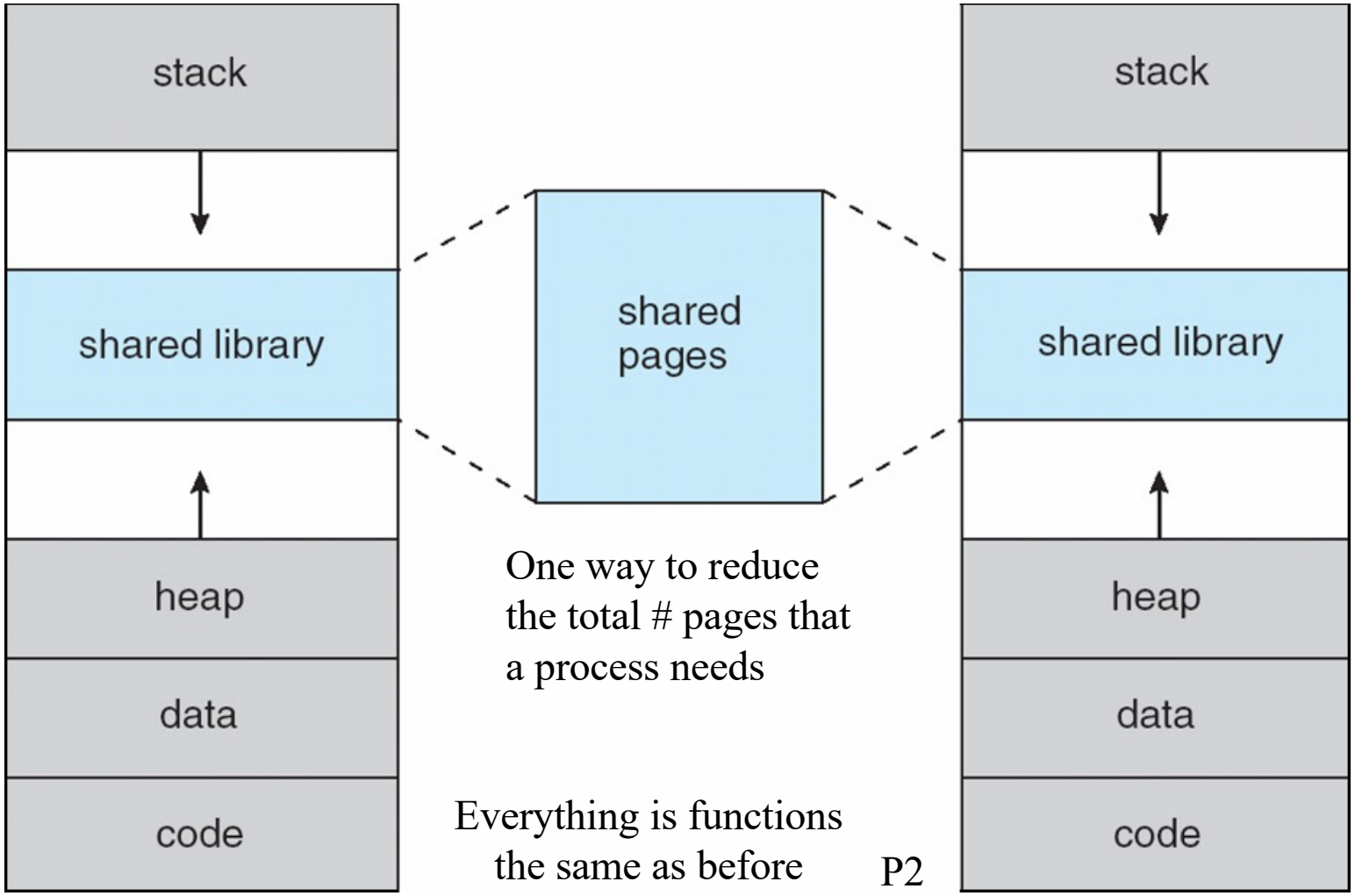
- Each process has a logical address space
  - Usually start at address 0, contiguous addresses until end of space defined by # bits in the address
- The *virtual address space* of a process is exactly the same as the logical address space
- Memory management problem is the same
  - Memory management unit (MMU) must map logical address space to a physical address space
  - Look at our memory management mechanisms
- What is different from our discussion before?
  - Allow the amount of logical memory used (by a process or all processes) to be greater than all physical memory
  - Allow a process to run without all its memory in physical memory … Where is it? … It is in virtual memory!

Virtual Address Space

- Usually design logical address space for stack to start at *Max* logical address and grow "down" while heap grows "up"
  - Maximizes address space use
  - Unused address space between the two
    - 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, …
- Exactly like the logical address space
- How to partition the virtual address space?
  - Can partition into pages, just like before



Shared Library Using Virtual Memory



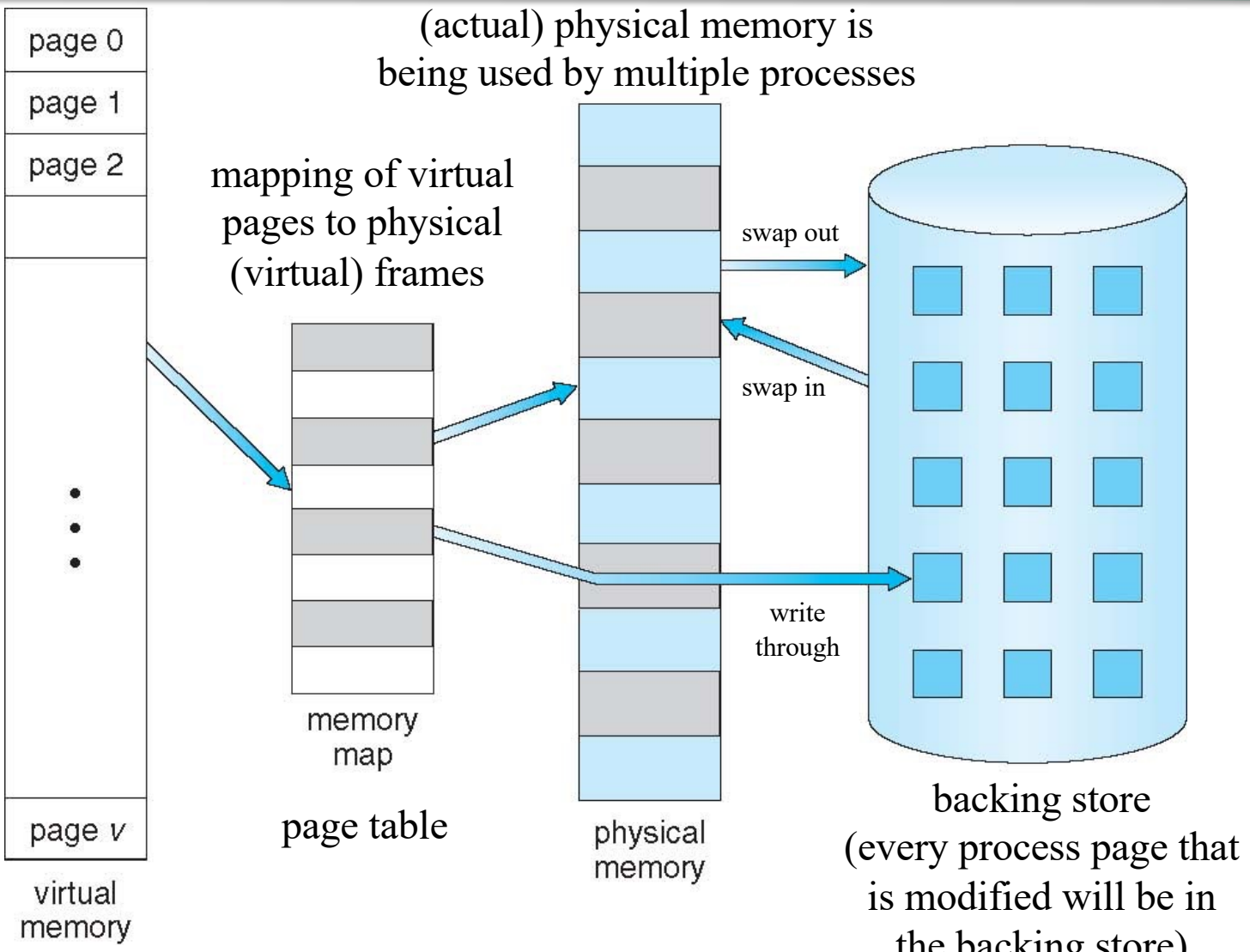
One way to reduce the total # pages that a process needs

Everything is functions the same as before

Virtual Memory and Physical Memory (1)

- How do we allocate physical memory?
- Use same page-based memory management
- Do not have to allocate a physical frame for all of the logical pages being used
- Think about allocating a *virtual frame* instead
- Where is this virtual frame?
  - Either in a physical frame (i.e., in physical memory)
  - Or on a storage system (*backing store, paging disk*)
- Now can allocate a virtual frame for all logical pages used (assuming large enough storage)
- A virtual frame must be in a physical frame to be used

Virtual Memory and Physical Memory (2)

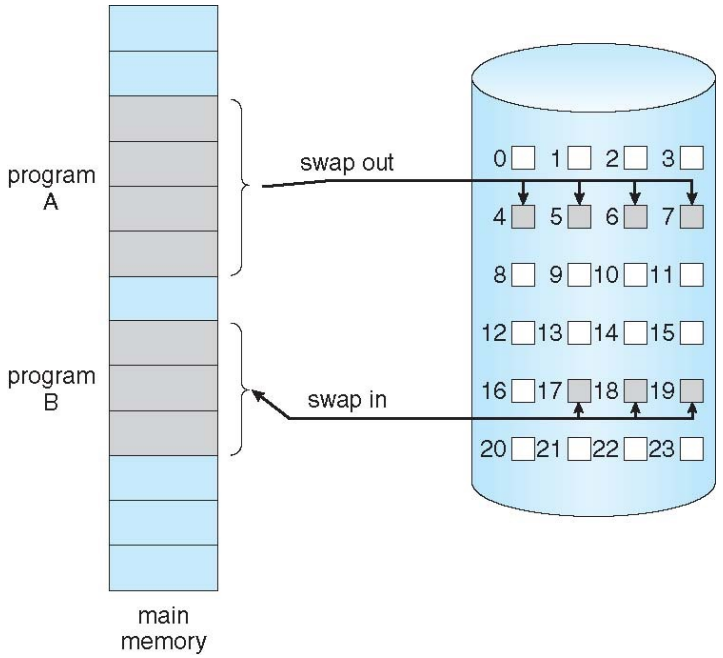


How is physical memory allocated?

- Consider using paging (assume for rest of lecture)
- Load all pages a process will ever need into frames
  - Bring entire process into memory at load time
  - What if there are too many (i.e., not enough frames)
- Load only what a process needs now into frames
  - At this point in its execution
  - How much is that?
  - How do we know what is needed?
- Load pages one at a time into a frame
  - What triggers this?

Demand Paging

- 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 (diagram on right)
- 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



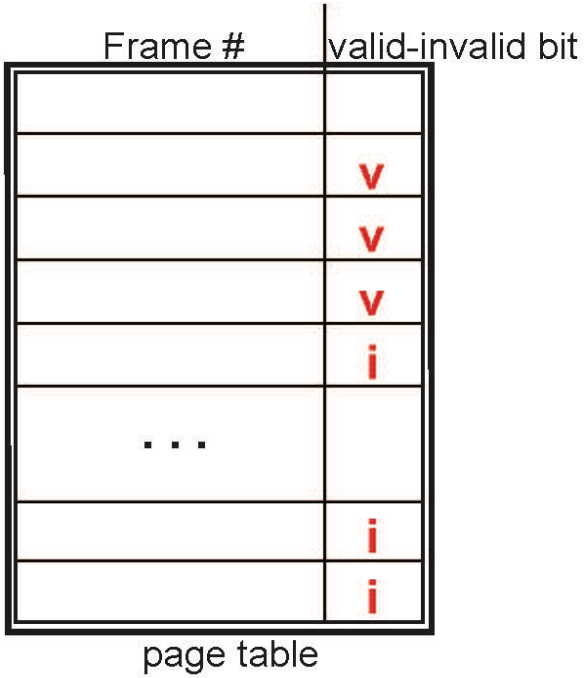
Swapping within a process at a page level!

Basic Concepts

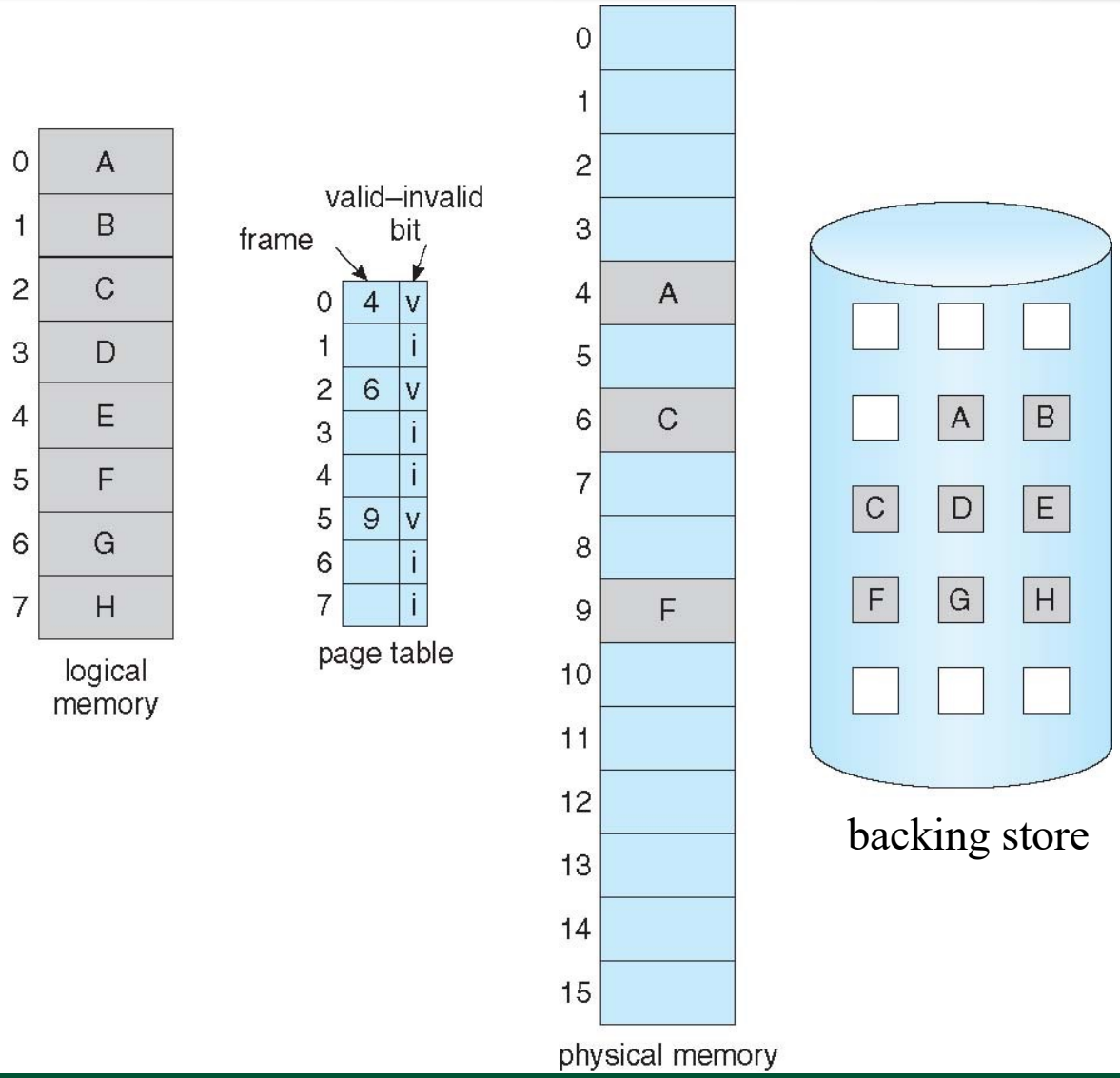
- With swapping, pager guesses which pages will be used before swapping out again
- Instead, pager brings in only those pages into memory
- How to determine that set of pages?
  - Need new MMU functionality to implement demand paging
- If pages needed are already memory resident
  - Already appears in the page table
- 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

- Each page table entry has a *valid–invalid* bit
  - $v \Rightarrow$ in-memory (memory resident)
  - $i \Rightarrow$ not-in-memory
- Initially valid–invalid bit is set to *i* on all entries
- During MMU address translation, if valid–invalid bit in page table entry is *i*, a page fault occurs



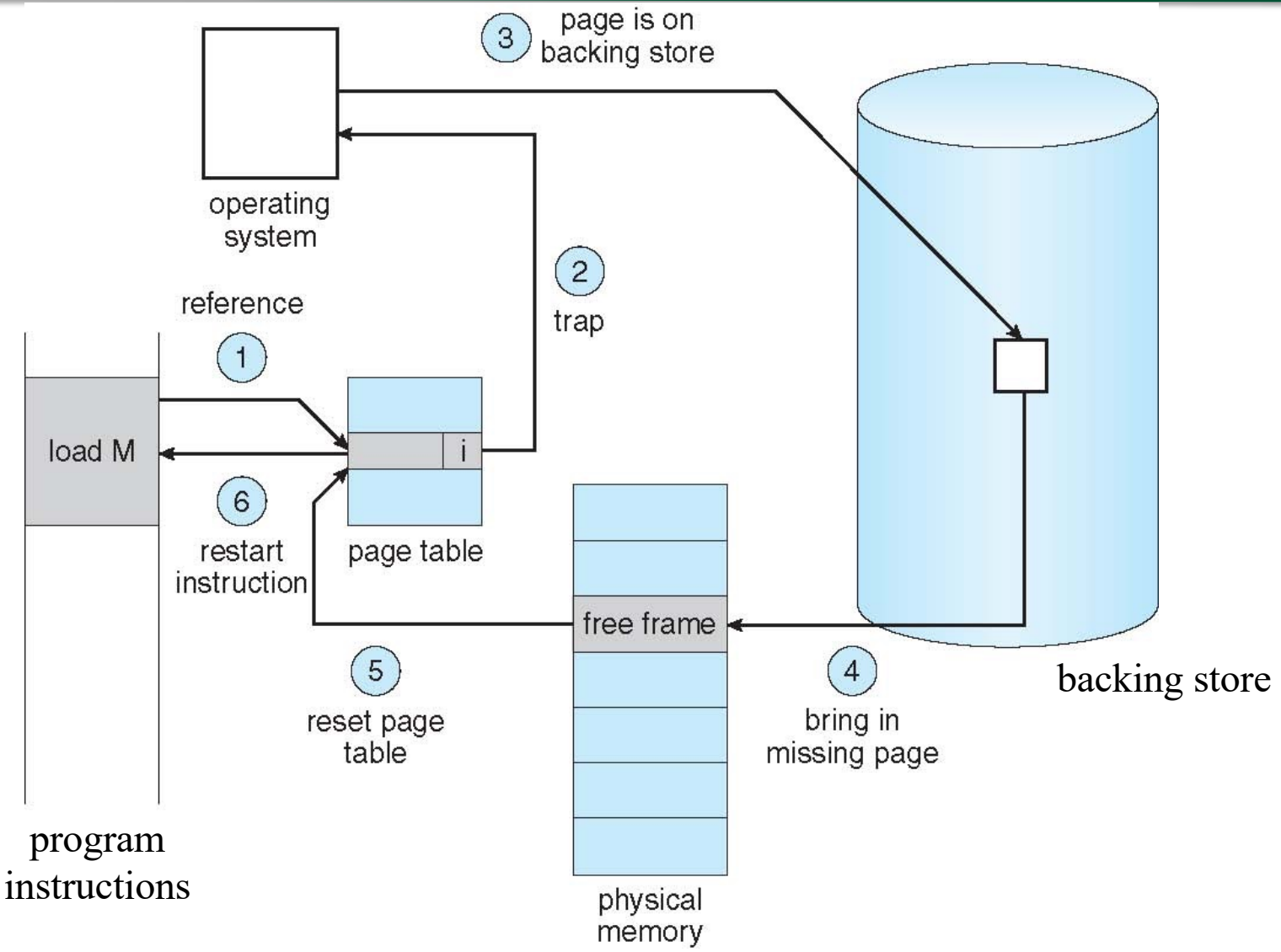
When Some Pages Are Not in Main Memory



Page Fault Processing

- If there is a reference to a page, first reference to that page will trap to operating system … why?
- Operating system looks at another table to decide:
  - Invalid reference ⇒ abort
  - Just not in memory (*page fault*) (first reference is)
- Find free frame
- Swap page into frame via scheduled disk operation
- Reset tables to indicate page now in memory
  - Set validation bit = *v*
- Restart the instruction that caused the page fault

Steps in Handling a Page Fault



Aspects of Demand Paging

- Extreme case – start process with no pages in memory
  - OS sets instruction pointer to first instruction of process
  - Immediately a page fault occurs … why?
  - First access to any page will generate a page fault
  - *Pure demand paging*
- Actually, a given instruction could access multiple pages, generating multiple page faults
  - Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
  - 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

Stages in Demand Paging (Worst Case)

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:
    a. Wait in a queue for this device until the read request is serviced
    b. Wait for the device seek and/or latency time
    c. 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

- 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)*

$$EAT = (1 - p) * \text{memory access} + p * (\text{page fault overhead} + \text{swap out} + \text{swap in} + \text{memory access})$$

Demand Paging Example

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

  $EAT = (1 - p) \times 200 + p\ (8\ milliseconds)$
  $= (1 - p \times 200 + p \times 8{,}000{,}000$
  $= 200 + p \times 7{,}999{,}800$
- 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

  $220 > 200 + 7{,}999{,}800 \times p$
  $20 > 7{,}999{,}800 \times p$
  $p < .0000025$
  < one page fault in every 400,000 memory accesses

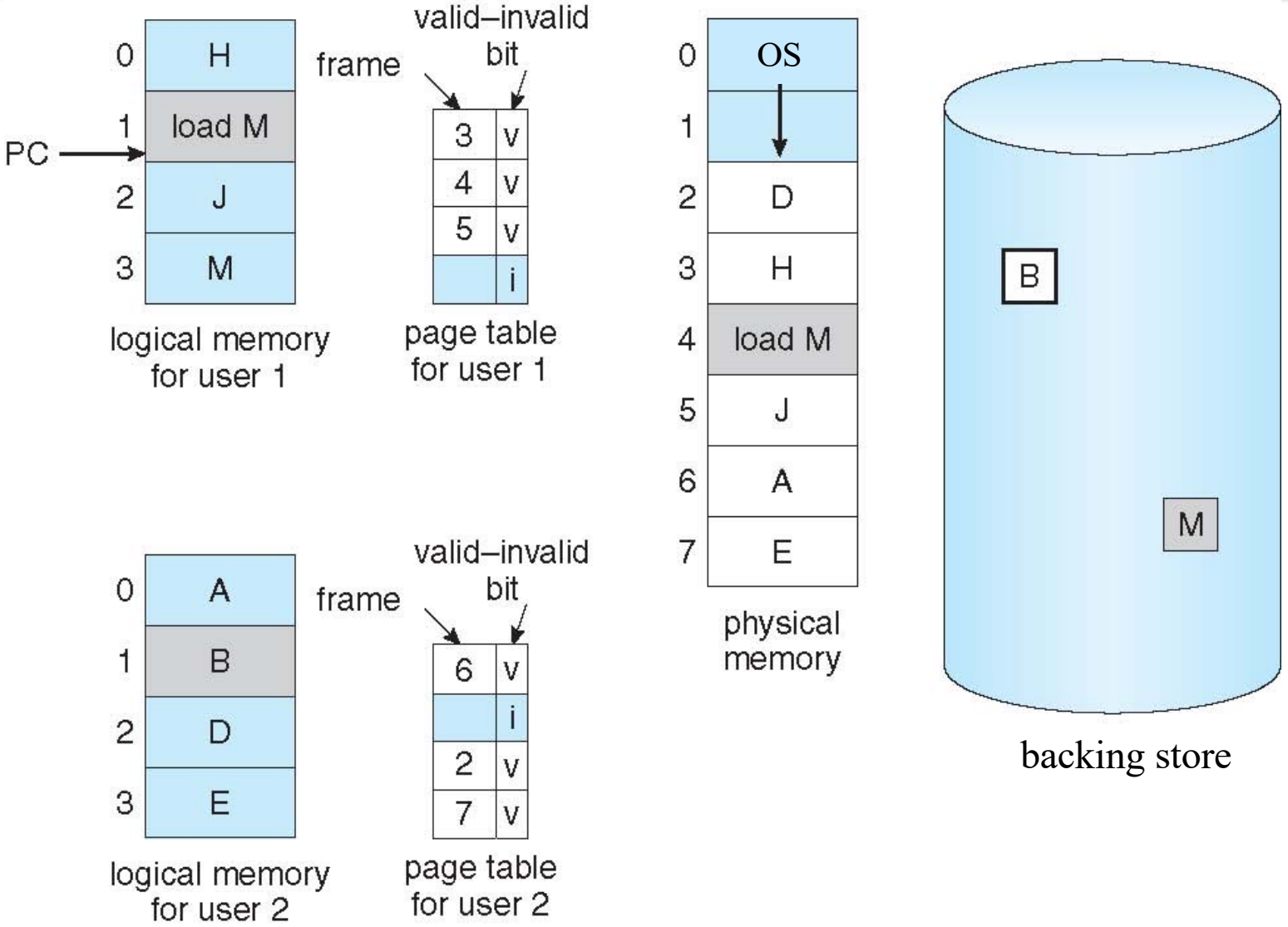
What happens if there is no free frame?

- Used up by process pages
- Memory is also in demand from the kernel, I/O buffers, and so on
- How much to allocate to each?
- *Page replacement*
  - Find some page in memory (what page?)
  - 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 during the lifetime of a process

Page Replacement

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
- Use modify (*dirty*) bit to reduce overhead
  - Page transfers
  - Only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory
  - Large virtual memory > smaller physical memory

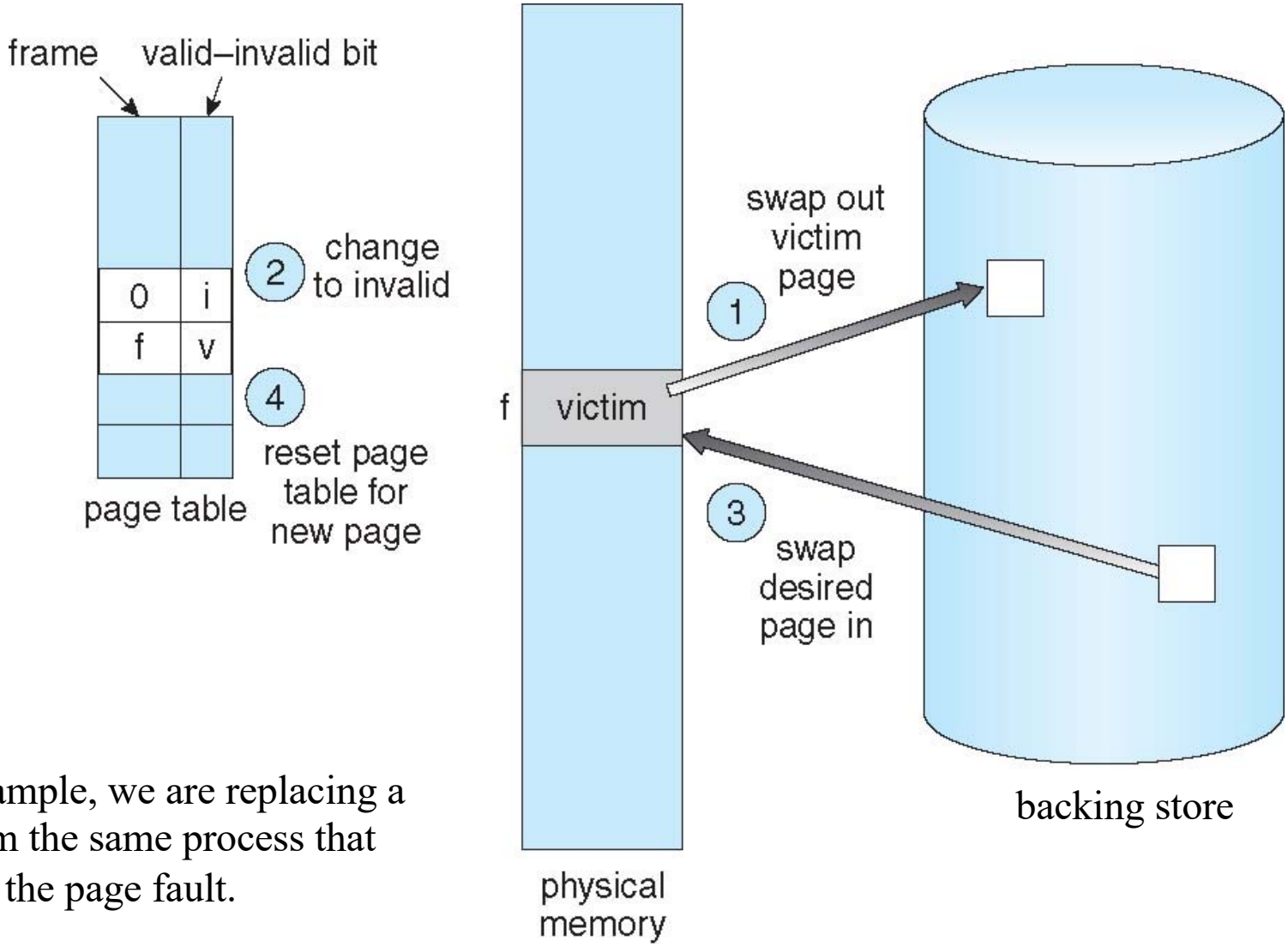
Need For Page Replacement



Basic Page Replacement

- Find the location of the desired page on disk
- 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
- Bring the desired page into the (newly) free frame
  - Need to update the page and frame tables
- Continue the process by restarting the instruction that caused the trap (i.e., page fault)
- Note now potentially 2 page transfers for page fault
  - Increases EAT

Page Replacement

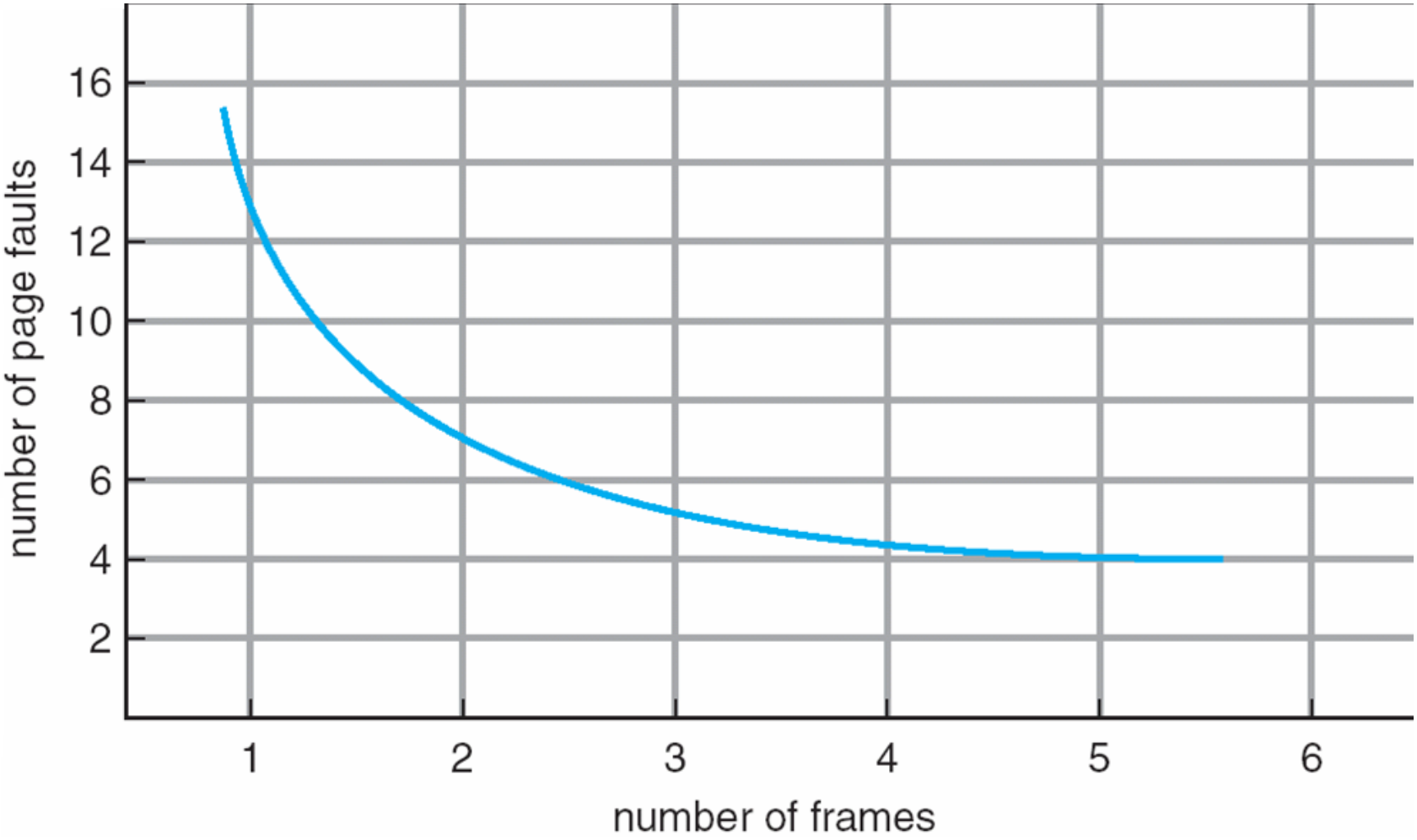


In this example, we are replacing a pages from the same process that generated the page fault.

Page and Frame Replacement Algorithms

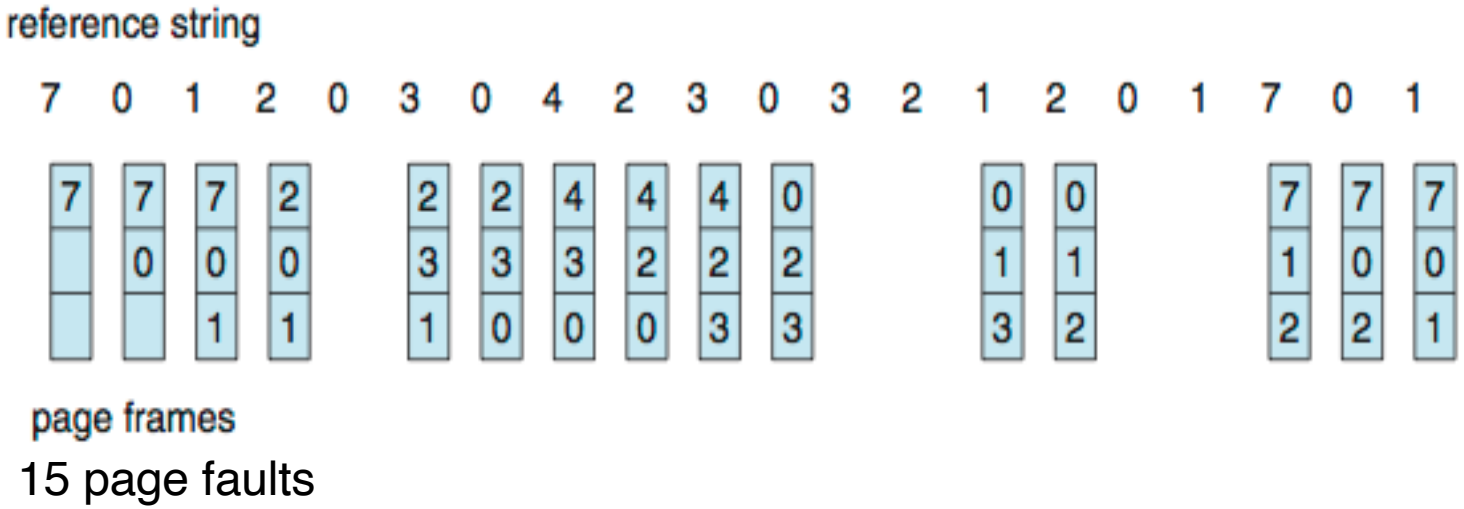
- Frame-allocation algorithm determines
  - How many frames to give each process
  - Which frames to replace
- Page-replacement algorithm
  - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string) 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

Page Faults Versus The Number of Frames



First-In-First-Out (FIFO) Algorithm

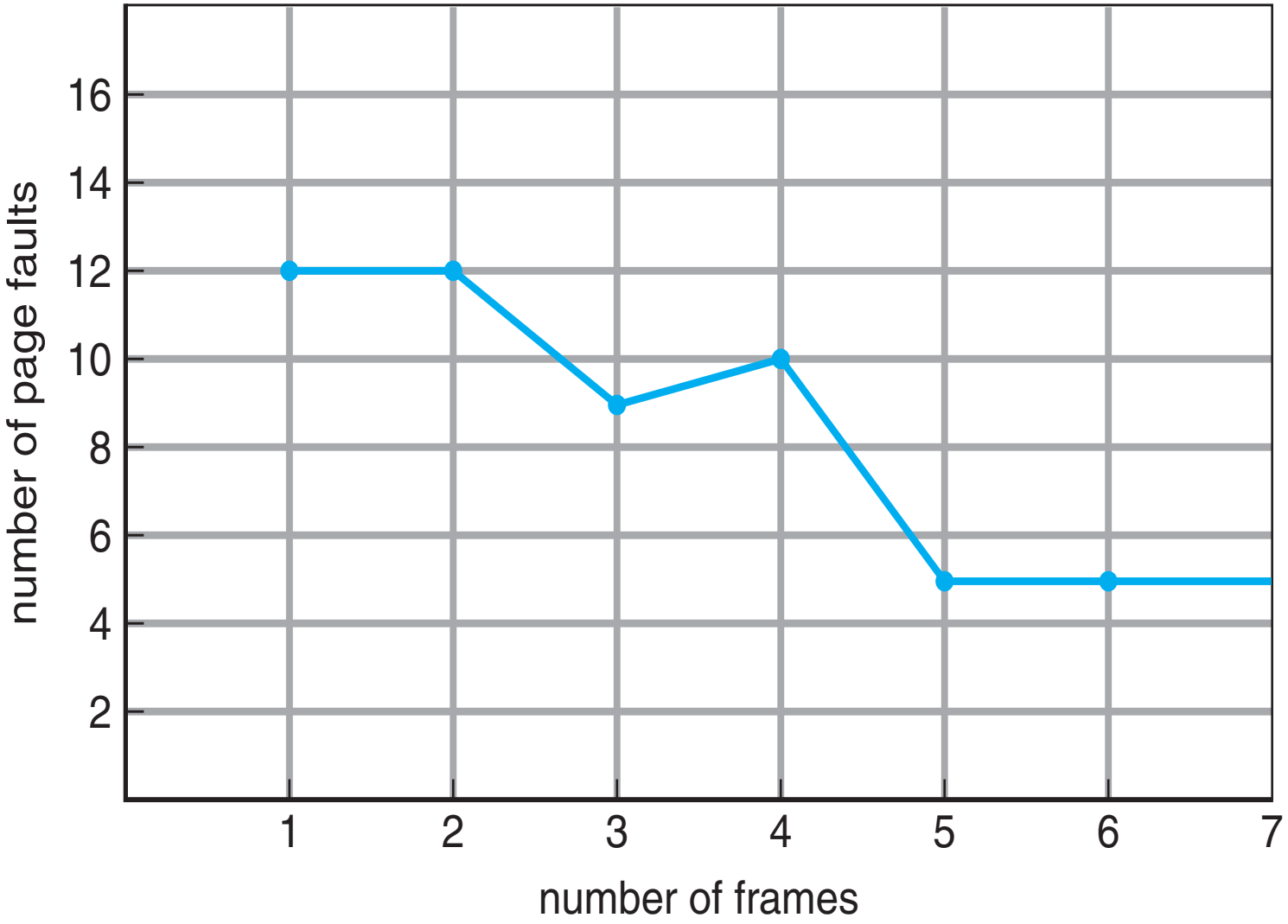
- Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1
- 3 frames (3 pages can be in memory at a time per process)



15 page faults

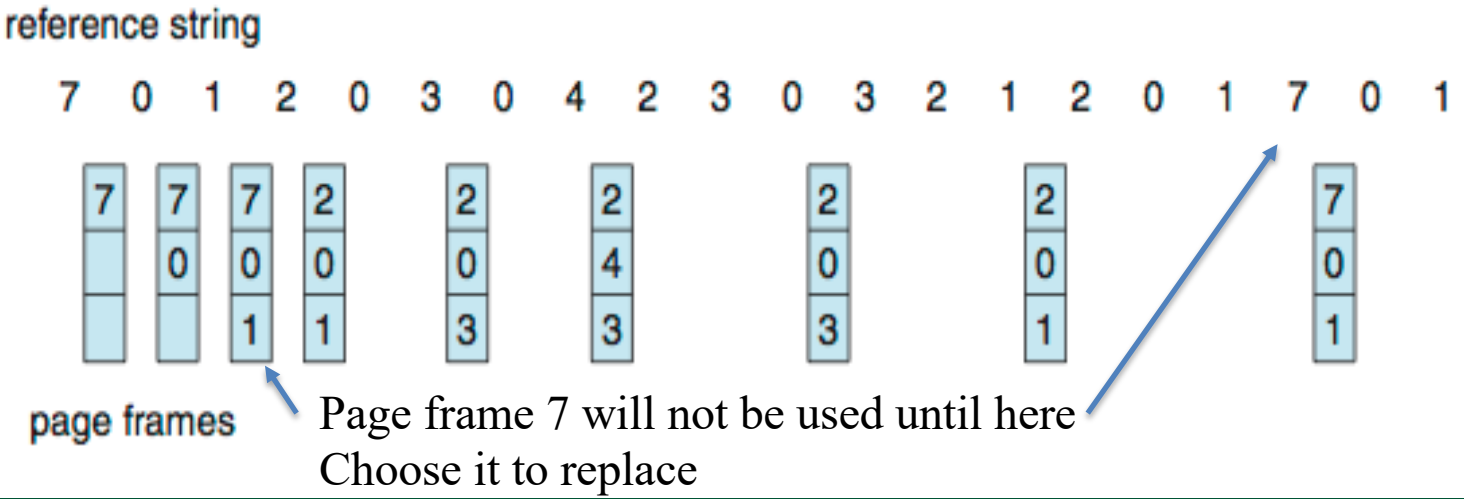
- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
  - Adding more frames can cause more page faults!
  - Belady's anomaly
- How to track ages of pages?
  - Just use a FIFO queue

FIFO Illustrating Belady's Anomaly



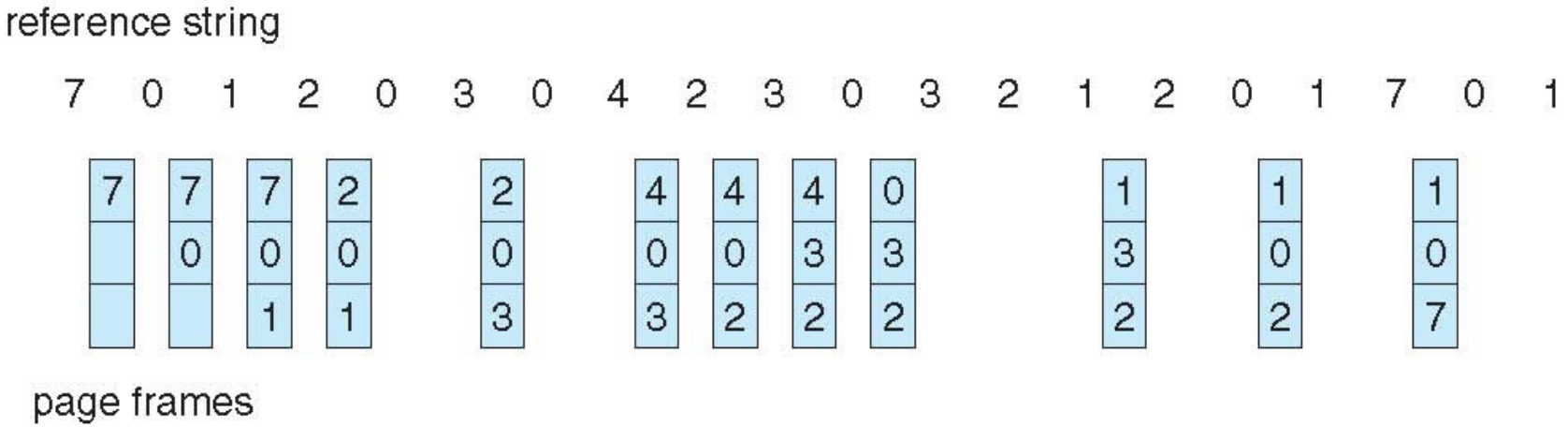
Optimal Algorithm

- Replace page that will not be used for longest period of time in the future
- How do you know this?
  - Can not read the future
- Used as a "best case" 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 (i.e., the one least recently used)
- Associate time of last use with each page

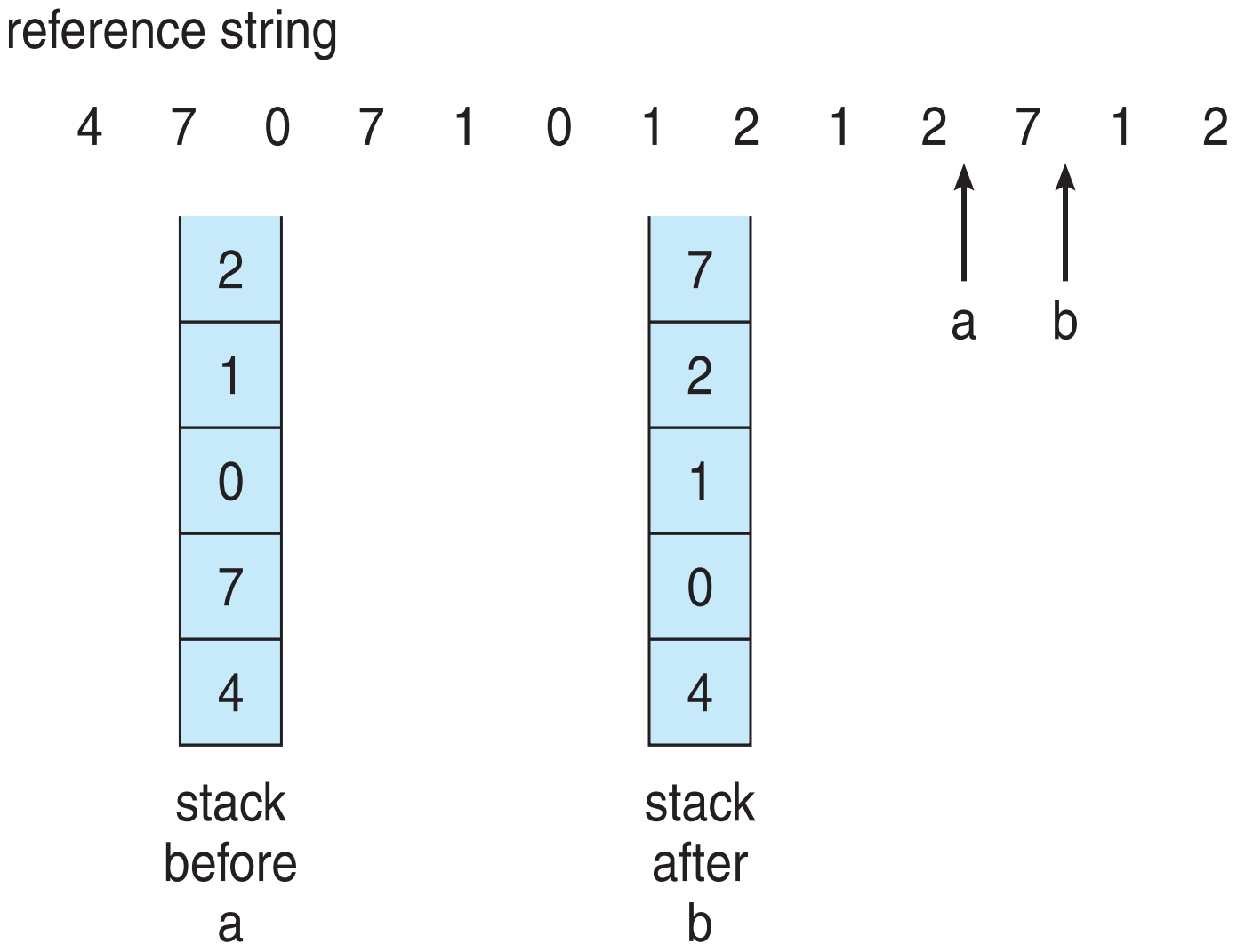


- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used in pratice
- But how to implement?

LRU Algorithm

- Counter implementation
  - Every page entry has a counter
  - Update counter with current clock when the page is referenced
  - When a page needs to be replaced, look at the counters to find smallest value (among the pages with frames)
    - search through table needed
- Stack implementation
  - Keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
  - Updates are more expensive than counters
  - No need to search for replacement (at bottome of stack)
- LRU and OPT are cases of stack algorithms that do not suffer Belady's Anomaly

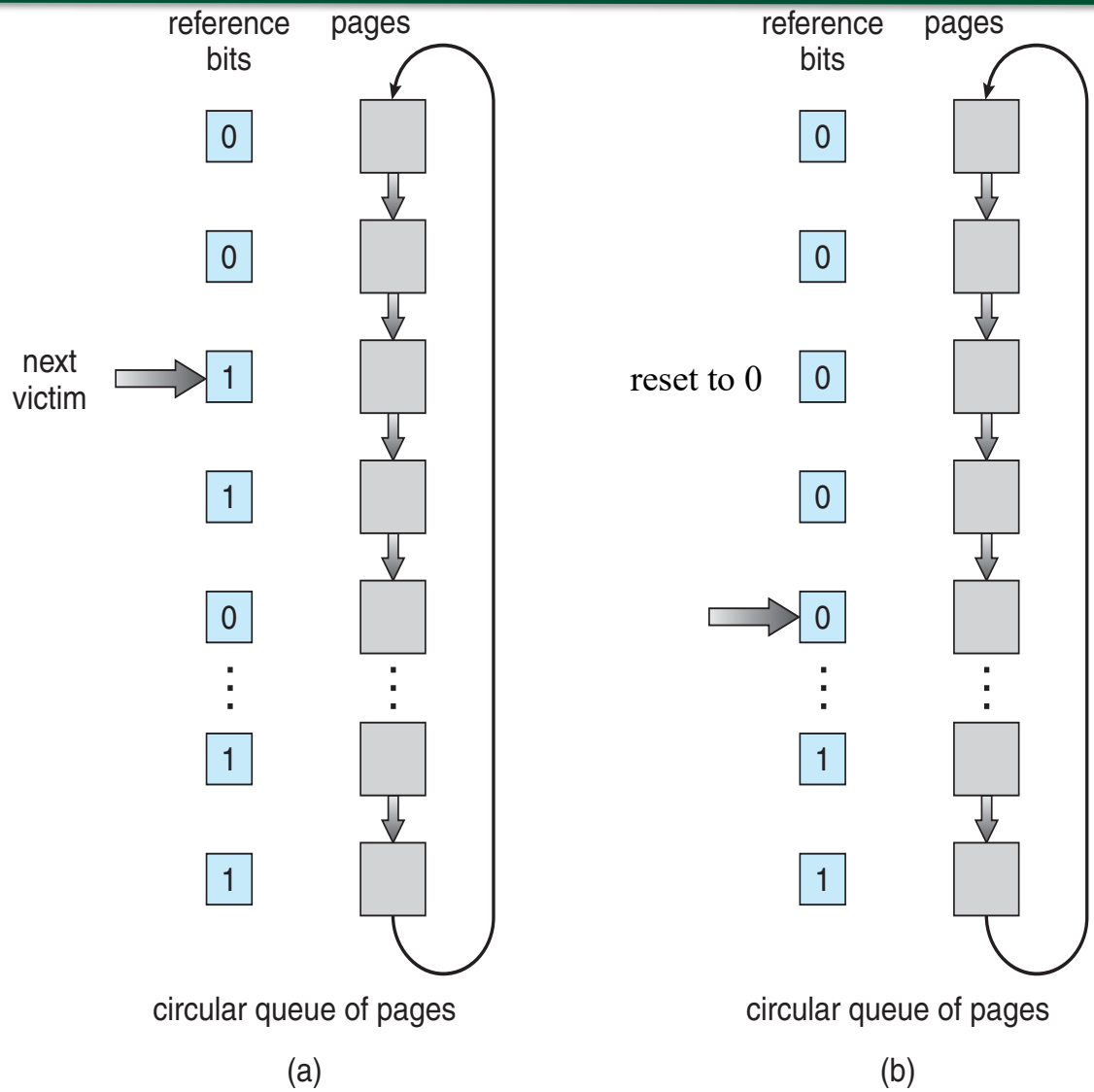
Record Most Recent Page References



LRU Approximation Algorithms

- LRU needs special hardware
- Algorithm 1: *Reference bit* (1-bit counter)
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace any page with reference bit = 0 (if one exists)
    - we do not know the order, however
- Algorithm 2: *Second-chance algorithm*
  - FIFO with hardware-provided reference bit
    - also known as *Clock replacement*
  - If page to be replaced has
    - reference bit = 0 then replace it
    - reference bit = 1 then:
      - set reference bit to 0, leave page in memory
      - replace next page, subject to same rules

Second-Chance (clock) Page-Replacement



Allocation of Frames

- Each process needs minimum number of frames
- Maximum of course is total frames in the system
- Two major allocation schemes
  - fixed allocation
  - priority allocation
- Many variations

Fixed Allocation

- Equal allocation
  - Each process gets the same # frames
- Proportional allocation
  - Allocate according to the size of process
- What happens when a process page faults?
  - Select a victim frame from among its own frames

Priority Allocation

- Suppose use priorities rather than size
- Processes of higher priority can get more frames
- If process P_i generates a page fault
  - Select for replacement one of its frames
  - Select for replacement a frame from a process with lower priority

Local vs. Global Replacement

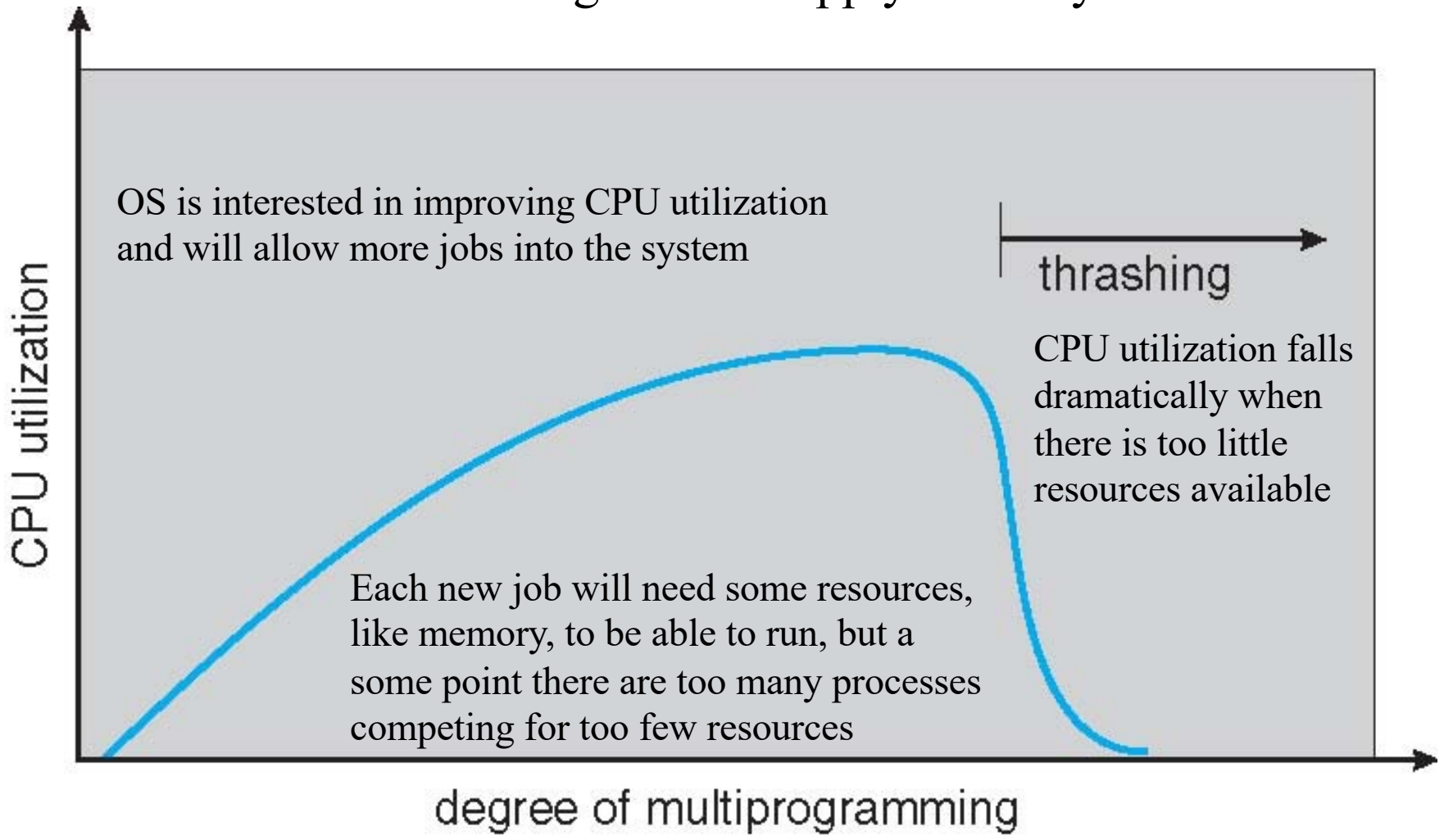
- *Local replacement*
  - Each process selects from only its own allocated frames
  - More consistent per-process performance
  - But possibly underutilizes memory
- *Global replacement*
  - Process selects a replacement frame from all frames
  - One process can take a frame from another
  - Can lead to greater throughput
  - But process execution time can vary greatly … Why?

Thrashing

- If a process does not have "enough" pages in memory, 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 rather than getting any work done

Thrashing Behavior

Thrashing can also apply to the system as a whole!



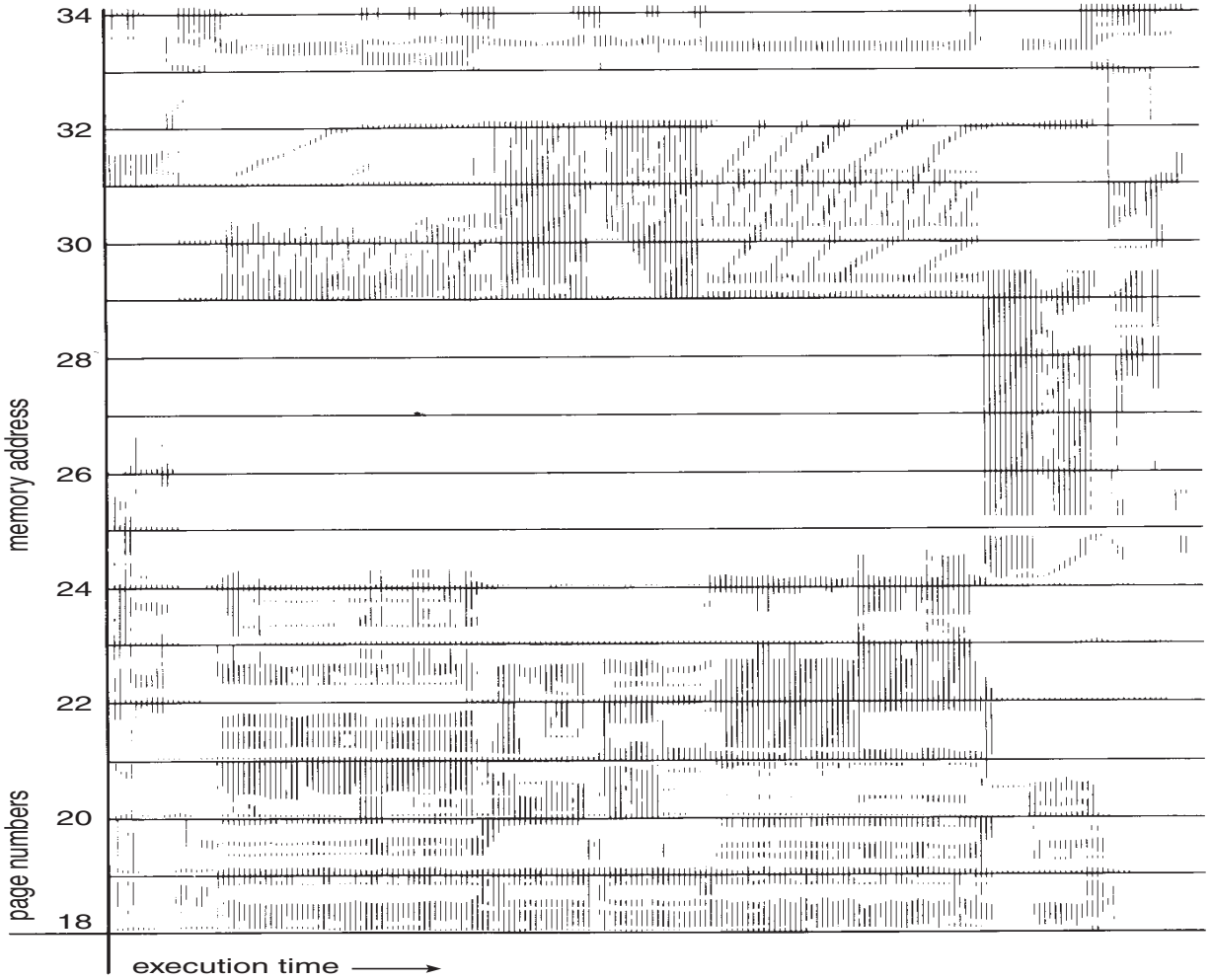
Demand Paging and Thrashing

- Why does demand paging work?
  - It brings pages being used NOW into memory
- Based on a model of *locality of reference*
  - Locality has to do with what logical addresses a process is using at a particular time
  - Process migrates from one locality to another
  - Localities may involve multiple address ranges
  - Want to have the pages covering the locality in memory
- Why does thrashing occur?

  Σ *process locality size > total memory size*
  - One or several processes might start thrashing

Locality In A Memory-Reference Pattern

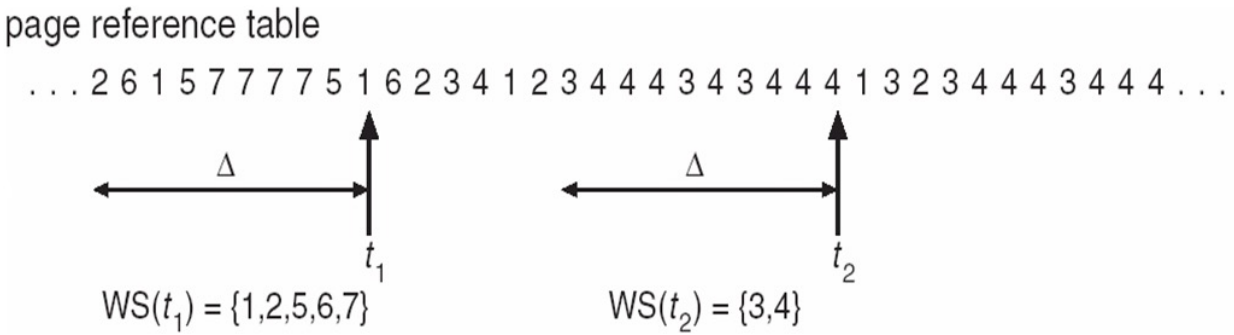
- Programs do not reference their memory randomly
- *Spatial locality*: memory addresses nearby to a reference are likely to be also addressed
- *Temporal locality*: a referenced memory address is likely to be referenced again in the near future



Shown are referenced addresses by a process over a period of time. Each reference is shown as a dot on the plot. Locality can be seen, both spatial and temporal.

Working-Set Model (Denning)

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
- WSS_i (working set of Process P_i) =
  total number of 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
- $D = \Sigma WSS_i \equiv$ total demanded frames
  - Approximation of locality
- Let m = total physical memory
  - If $D > m \Rightarrow$ thrashing is likely to occur
- What to do if $D > m$?
  - Need to suspend or swap out one of the processes until $D < m$



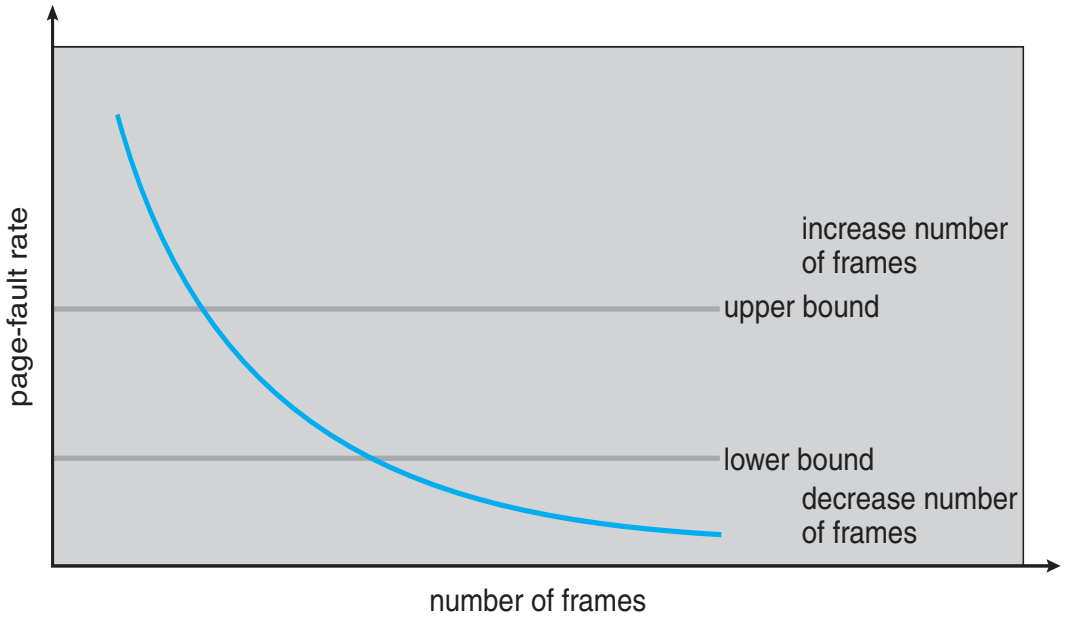
https://en.wikipedia.org/wiki/Working_set

How to keep track of the working set?

- Approximate with interval timer + a reference bit
- Example: $\Delta = 10{,}000$
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set
- Why is this not completely accurate?
- Improvement
  - 10 bits and interrupt every 1000 time units

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, process loses frame
  - If actual rate too high, process gains frame



Working Sets and Page Fault Rates

- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- Peaks and valleys over time