CSE-3211, Operating System

Chapter 9: Virtual Memory

Md Saidur Rahman Kohinoor
Lecturer, Dept. of Computer Science
Leading University – Sylhet
kohinoor_cse@lus.ac.bd

Objectives

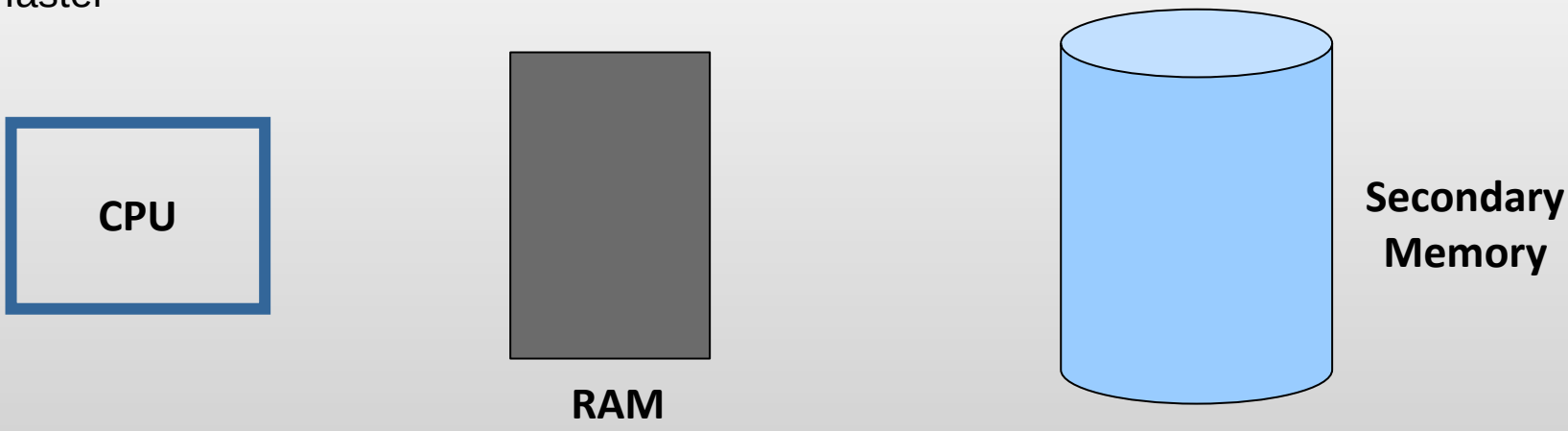
- To describe the benefits of a virtual memory system
- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames
- To examine the relationship between shared memory and memory-mapped files

Contents to be covered:

- Background
- Virtual Memory
- Demand Paging
- Copy-on-Write
- Page Replacement
- Disk Scheduling

Background

- Code needs to be in memory to execute, but the entire program is rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at the same time
- Consider the ability to execute a partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running -> more programs 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

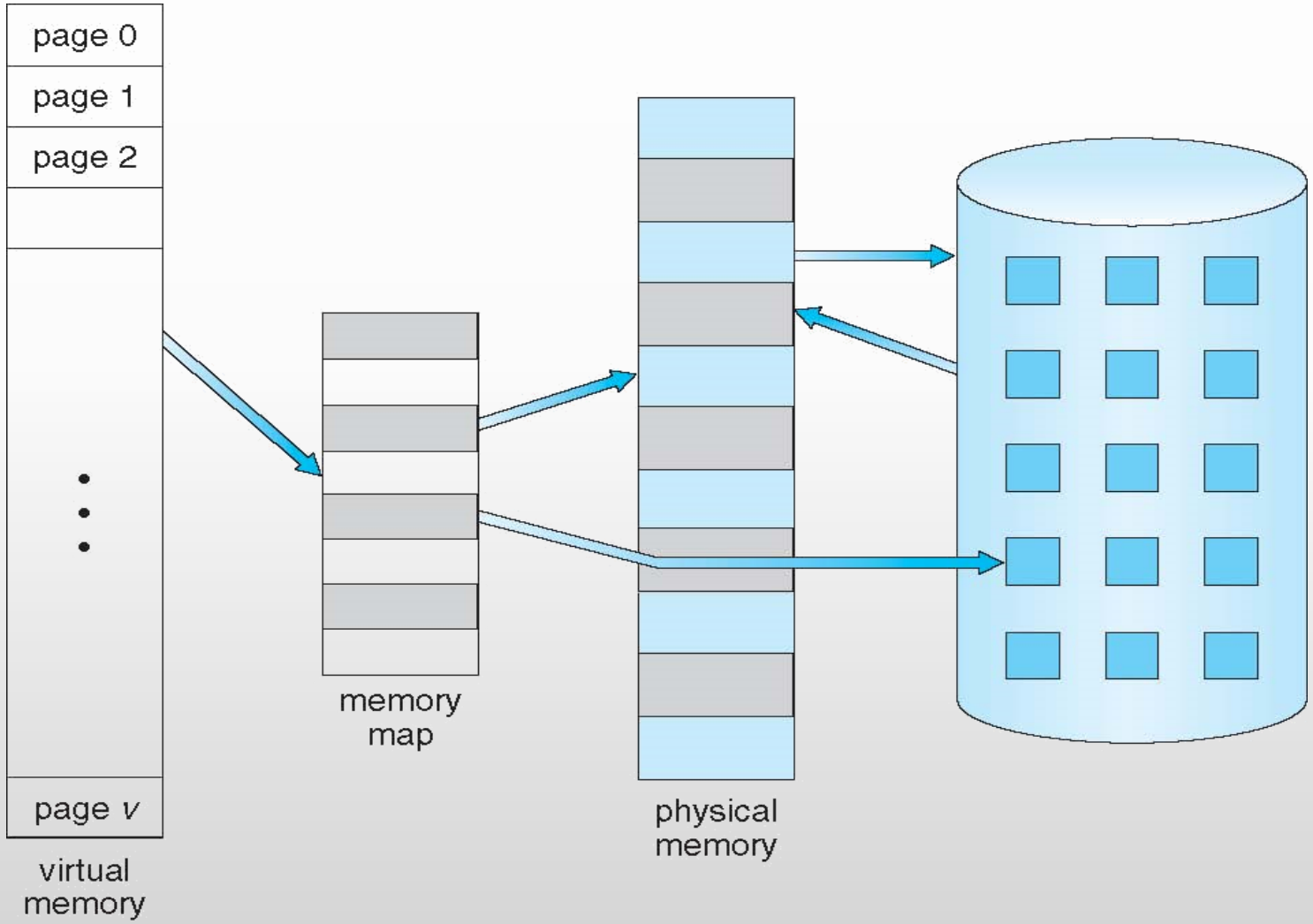


Virtual Memory

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

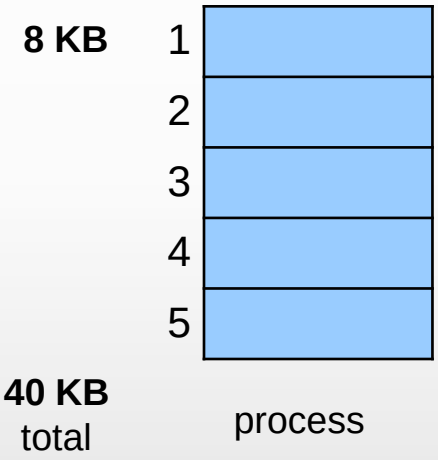
Virtual Memory (Cont.)

- Virtual Memory That is larger Than Physical Memory



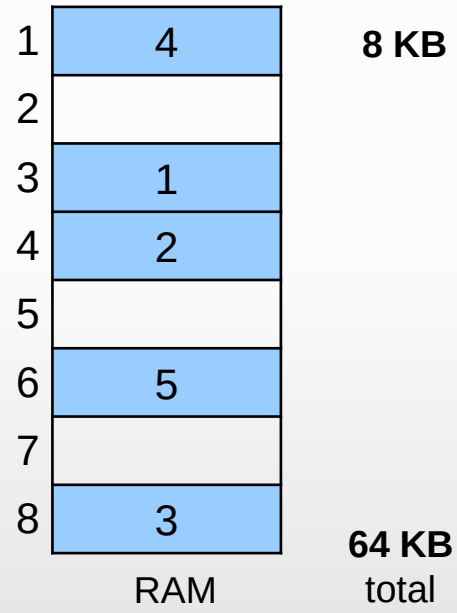
Virtual Memory (Cont.)

- Every time a memory location is accessed, the processor looks into the page table to identify the corresponding page frame number.



block	Page frames
1	3
2	4
3	8
4	1
5	6

Process page table



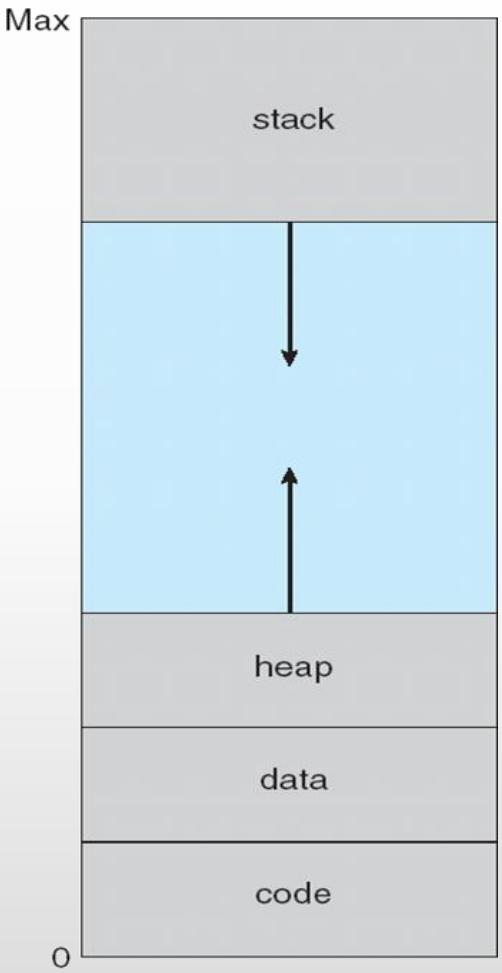
Process split into blocks of equal size

Block size = page frame size

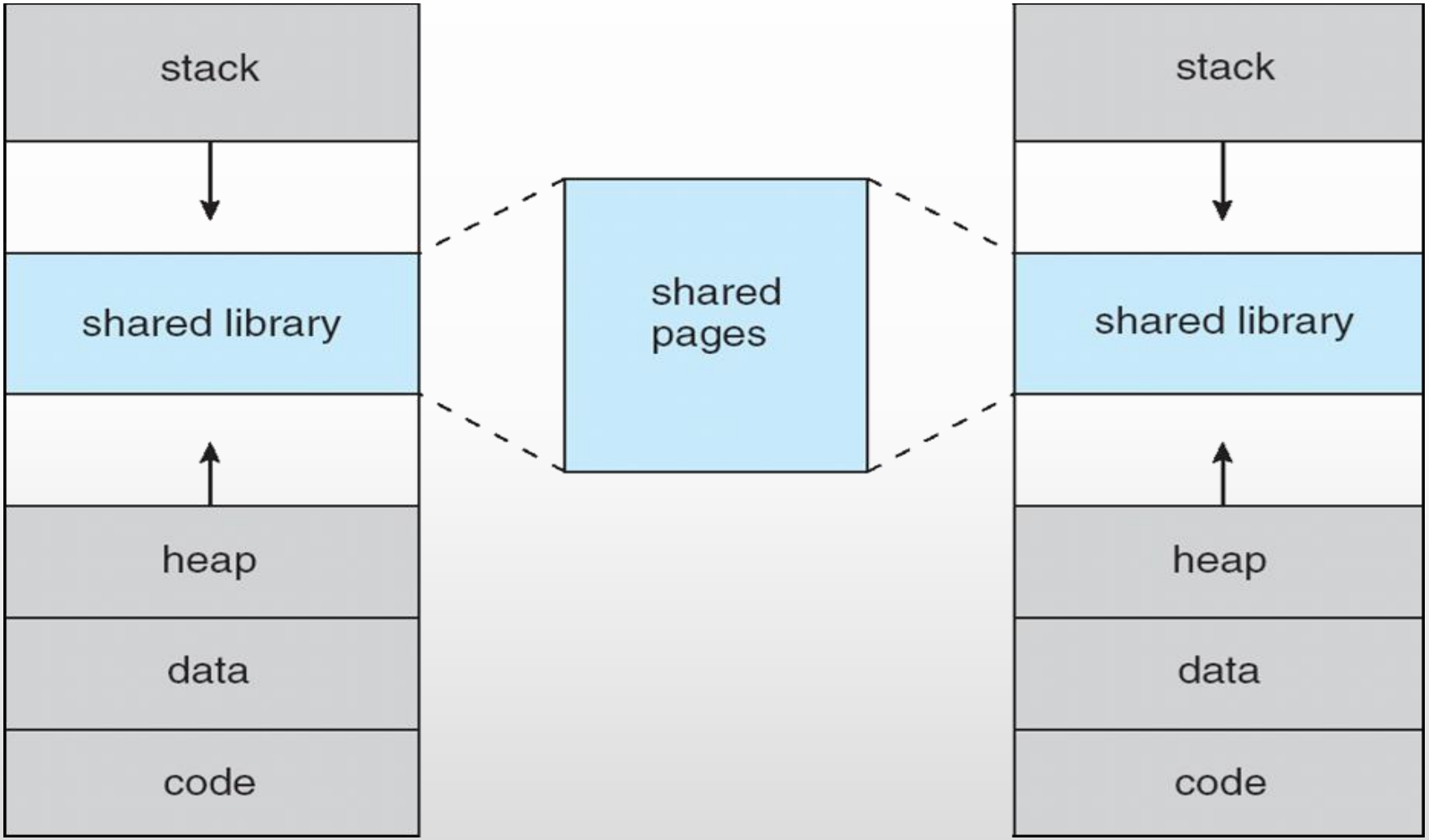
Physical memory is split into fixed-size partitions called page frames

Virtual Memory (Cont.)

- Usually design logical address space for a stack to start at Max logical address and grow “down” while heap grows “up”
  - Maximizes address space use
  - Unused address space between the two is a hole
    - No physical memory is needed until the heap or stack grows to a given new page
- 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

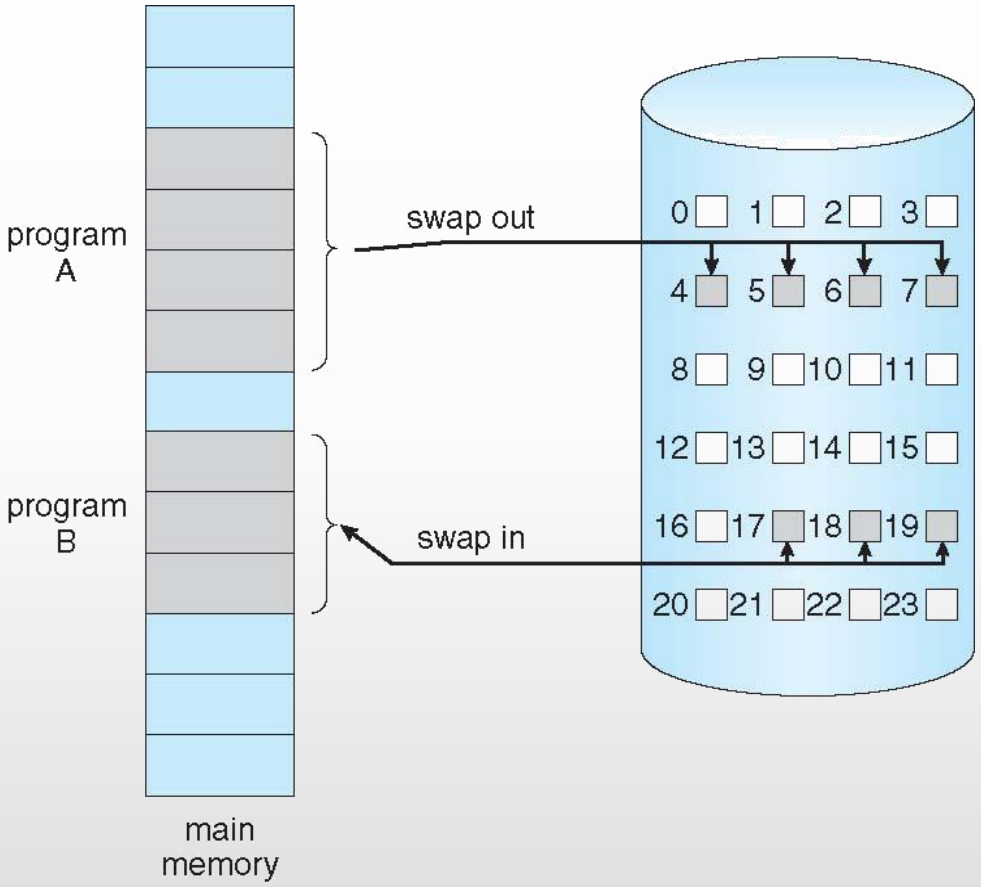


Shared Library Using Virtual Memory



Demand Paging

- Could bring the 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 the 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 the page will be needed
  - Swapper that deals with pages is a **pager**

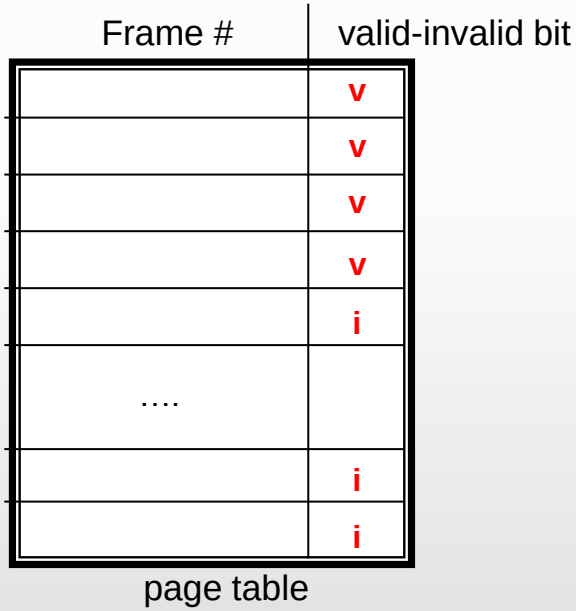


Basic Concepts

- With swapping, the pager guesses which pages will be used before swapping out again
- Instead, the 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 a page is needed and not memory resident
  - Need to detect and load the page into memory from storage
    - Without changing program behavior
    - Without the programmer needing to change code

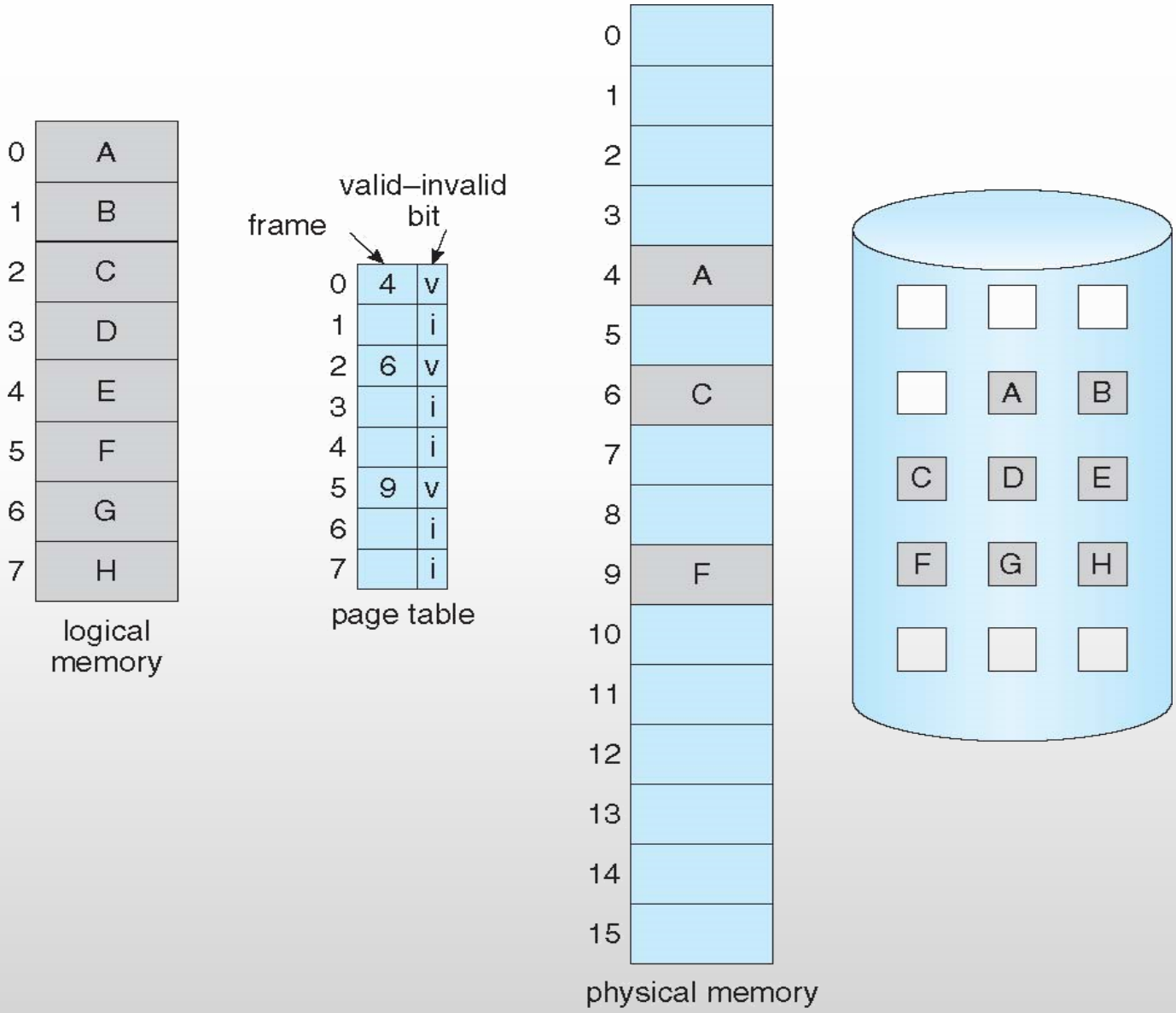
Valid-Invalid Bit

- 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 a valid–invalid bit in page table entry is **i** ⇒ page fault

Page Table When Some Pages Are Not in Main Memory



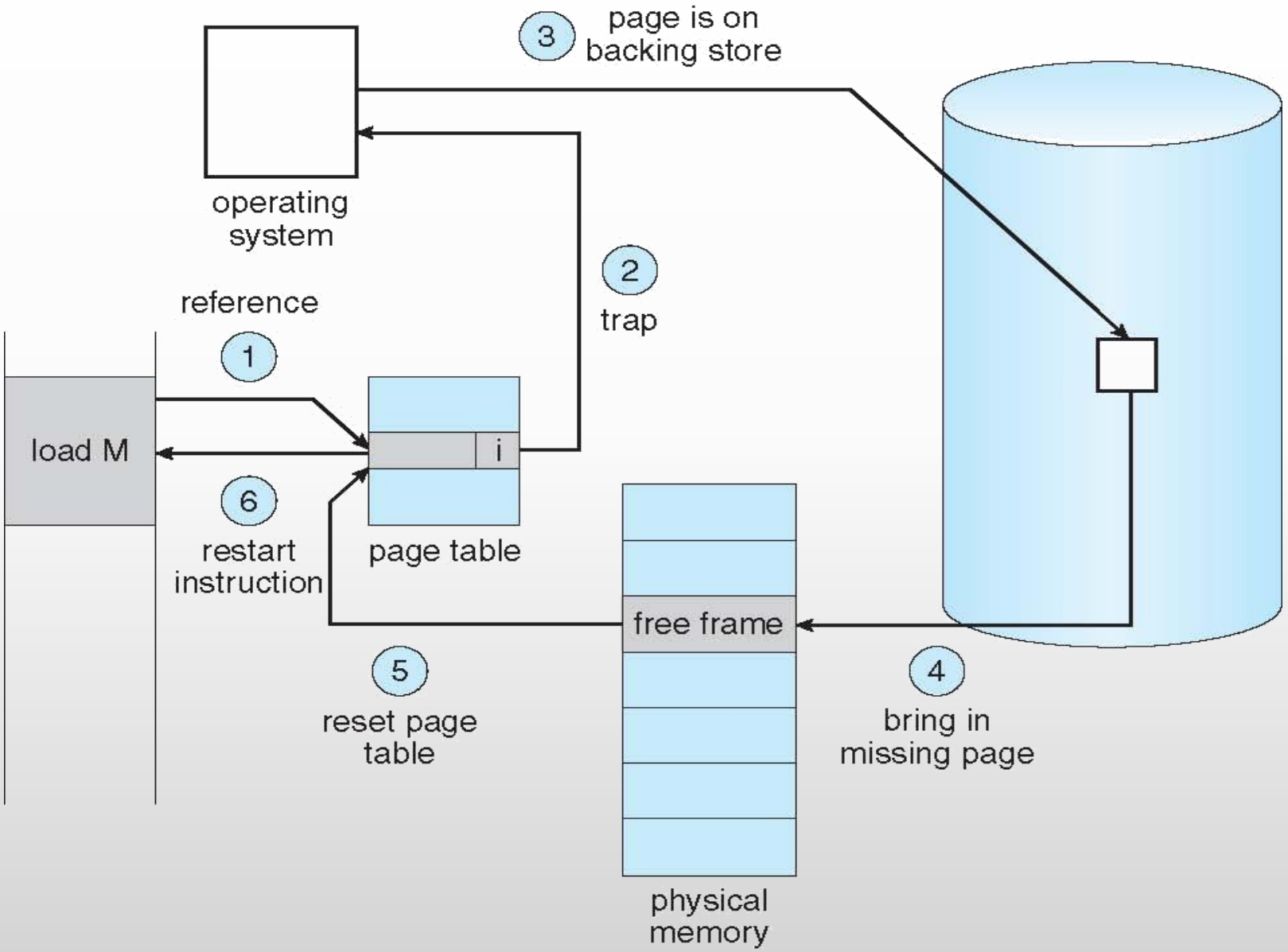
Page Fault

- If there is a reference to a page, first reference to that page will trap to operating system:

  page fault

- Operating system looks at another table to decide:
  - Invalid reference ⇒ abort
  - Just not in memory
- Find a free frame
- Swap page into the 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

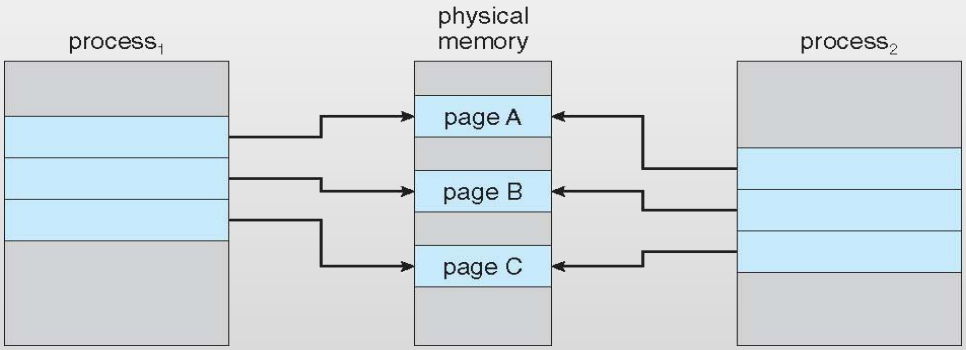
- Extreme case – start the process with *no* pages in memory
  - OS sets instruction pointer to the first instruction of process, non-memory-resident -> page fault
  - And for every other process page on the first access
  - **Pure demand paging**

- Actually, a given instruction could access multiple pages -> multiple page faults
  - Consider fetching and decoding of instruction which adds 2 numbers from memory and stores the result back to memory
  - Pain decreased because of the **locality of reference**

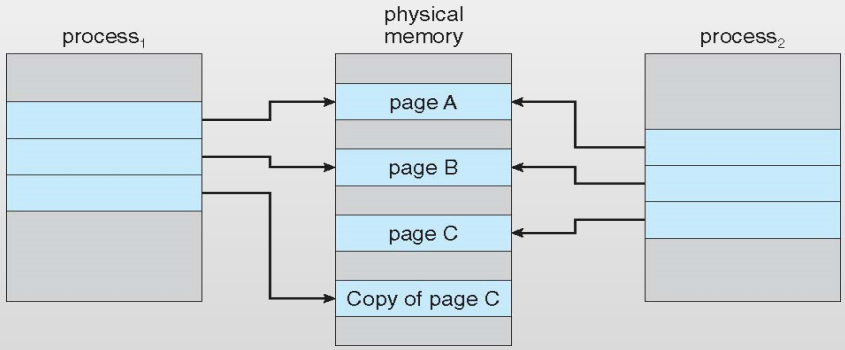
- Hardware support needed for demand paging
  - Page table with valid/invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart

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
  - Pool should always have free frames for fast demand page execution
    - Don't want to have to free a frame as well as other processing on page fault
  - 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



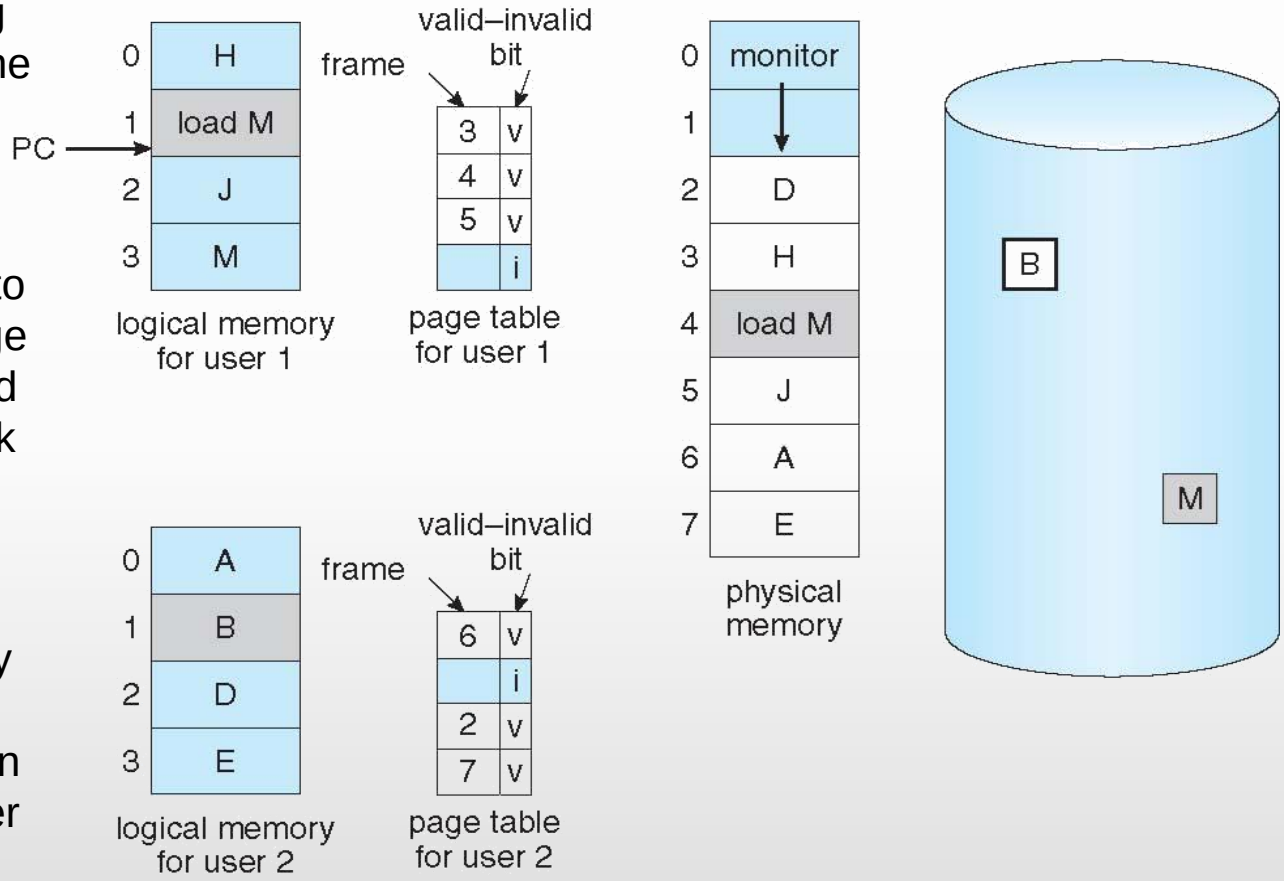
After Process 1 Modifies Page C

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 that will result in a minimum number of page faults
- Same page may be brought into memory several times

Page Replacement

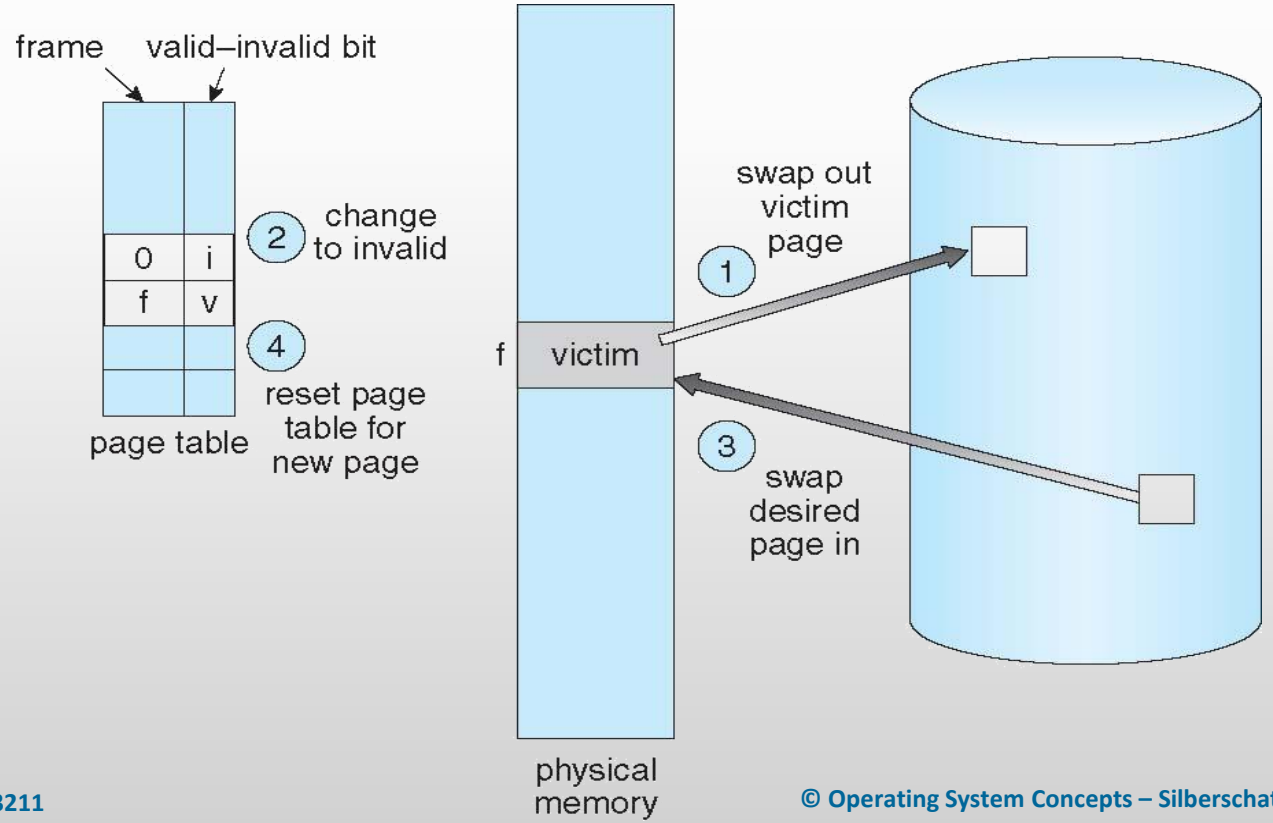
- Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement
- Use **modify** (**dirty**) **bit** to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory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



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

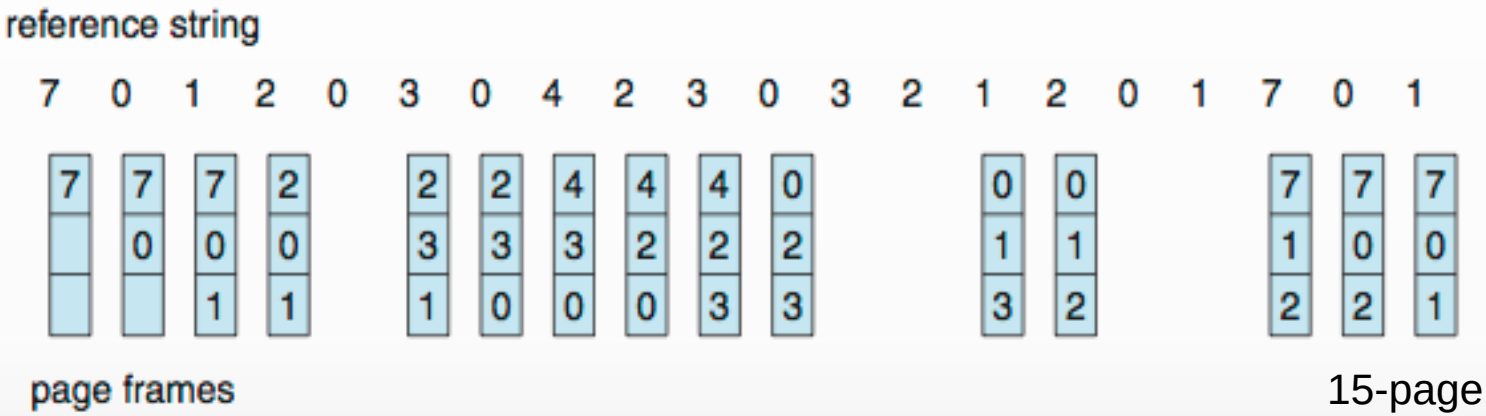
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
  - Results depend on number of 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

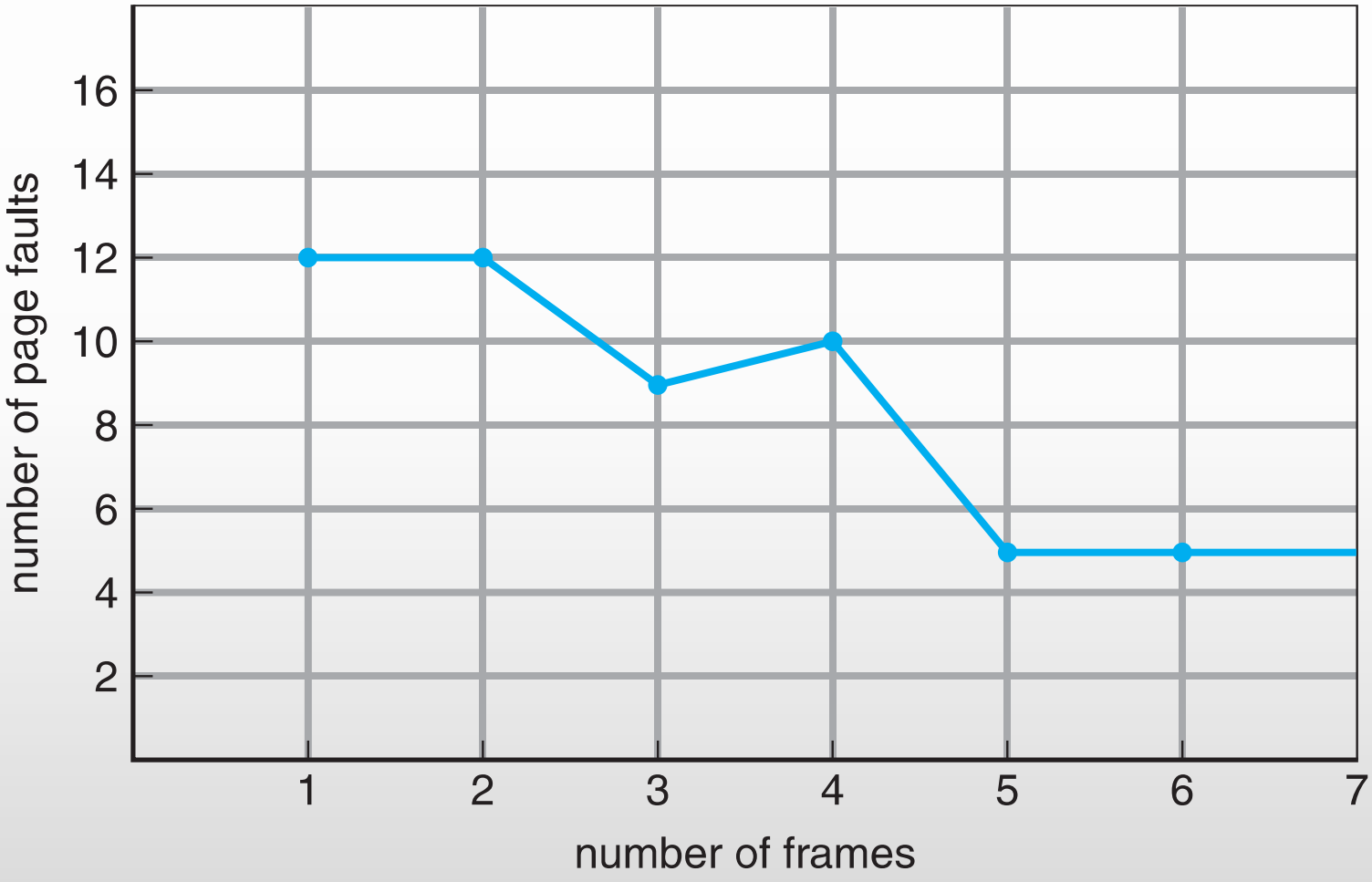
First-In-First-Out (FIFO) Algorithm

- 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)



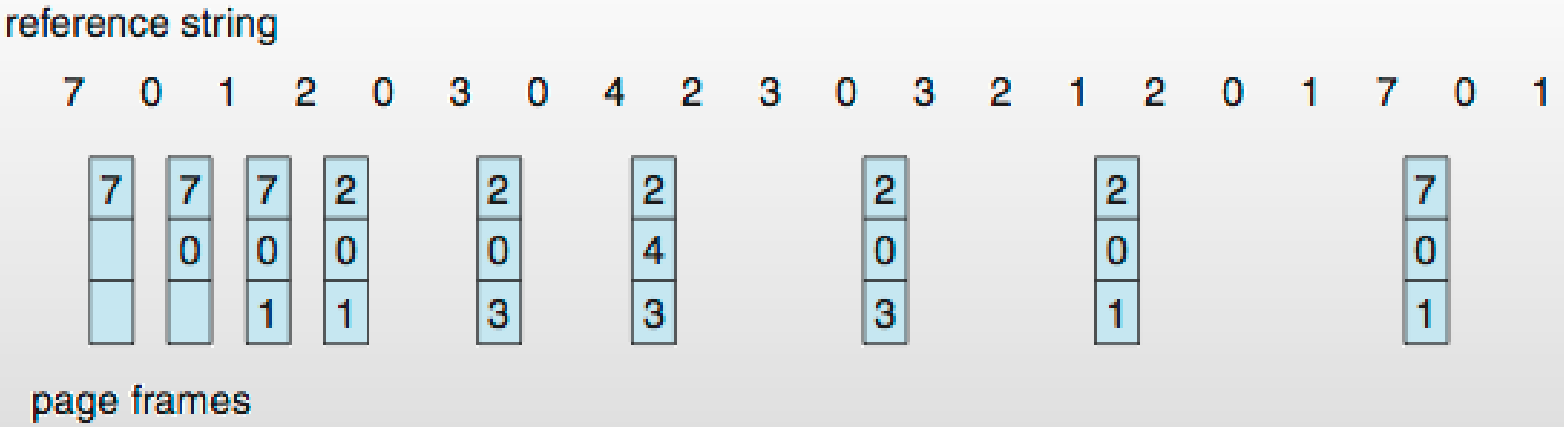
- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
  - Adding more frames can cause more page faults!
    - **Belady's Anomaly**
- How to track the ages of pages?
  - Just use a FIFO queue

FIFO Illustrating Belady's Anomaly



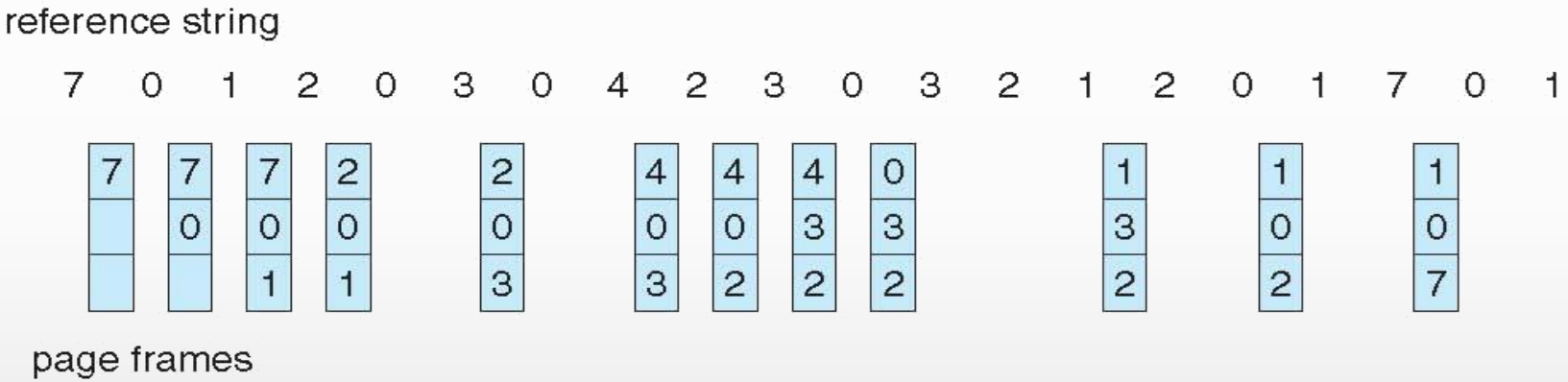
Optimal Algorithm

- Lowest page fault rate of all PRA and never suffer from Belady's anomaly.
- Replace the 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

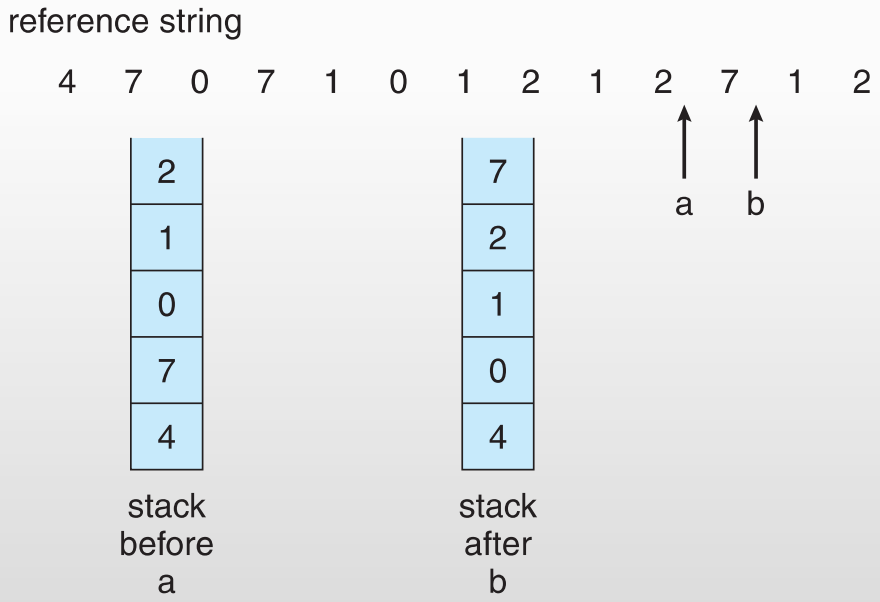
- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page



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

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 the smallest value
    - Search through the 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 is more expensive
  - No search for a replacement.
- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly



Disk Scheduling

- Magnetic disks provide bulk of secondary storage of modern computers
- **Disks** can be removable (Magnetic Disks, Solid-State Disks, Magnetic Tape etc.)
- **Disk drives** are addressed as large 1-dimensional arrays of **logical blocks**, where the logical block is the smallest unit of transfer
- The operating system is responsible for using hardware efficiently — for the disk drives, this means having a fast access time and disk bandwidth
- Minimize seek time, Seek time ≈ seek distance
- Disk **bandwidth** is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer

Disk Scheduling (Cont.)

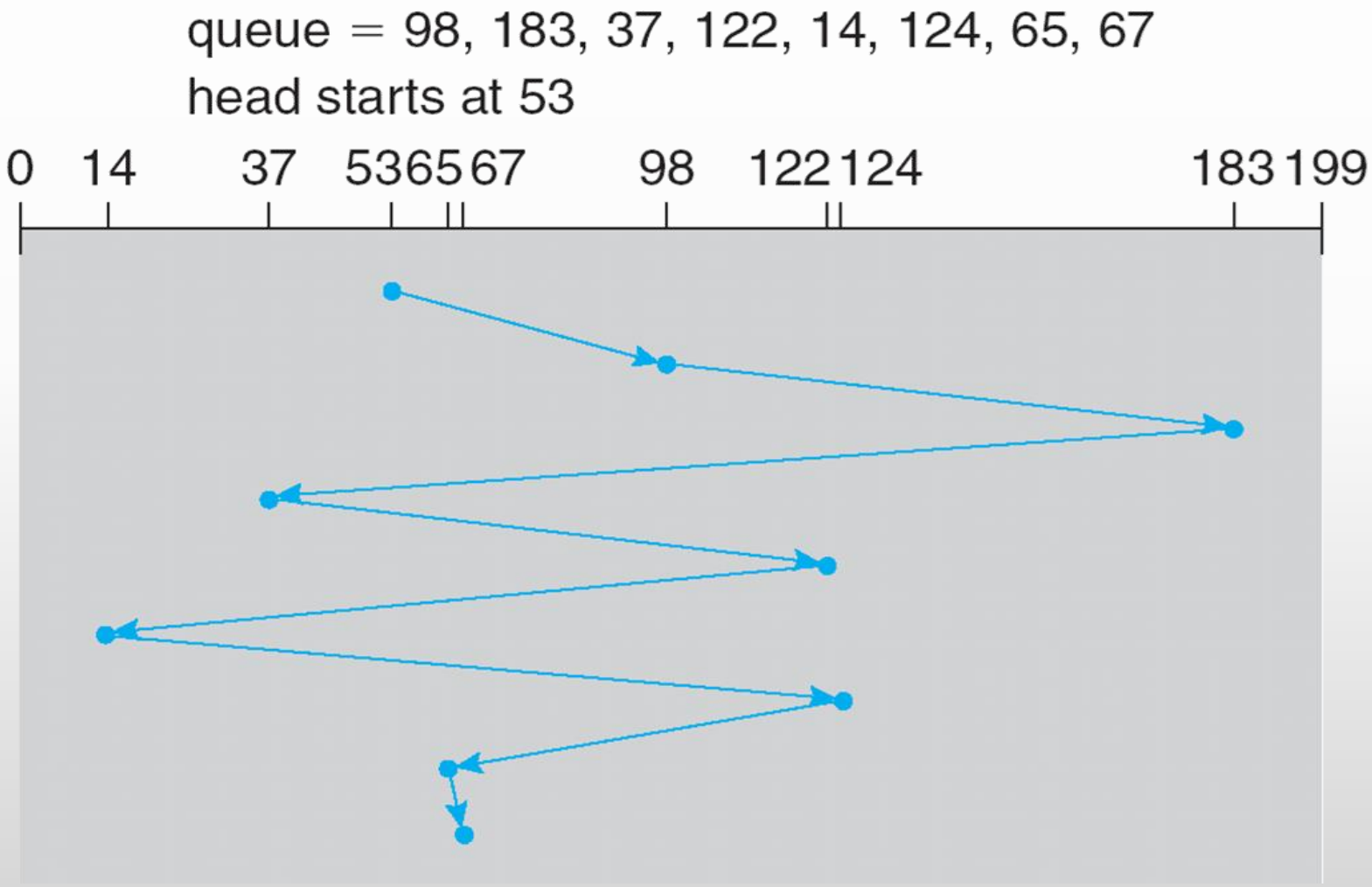
- There are many sources of disk I/O request
  - OS
  - System processes
  - User's processes
- I/O request includes input or output mode, disk address, memory address, number of sectors to transfer
- OS maintains queue of requests, per disk or device
- Idle disk can immediately work on I/O request, busy disk means work must queue
  - Optimization algorithms only make sense when a queue exists
- Note that drive controllers have small buffers and can manage a queue of I/O requests (of varying "depth")
- Several algorithms exist to schedule the servicing of disk I/O requests
- The analysis is true for one or many platters
- **We illustrate scheduling algorithms with a request queue (0-199)**

98, 183, 37, 122, 14, 124, 65, 67

Head pointer 53

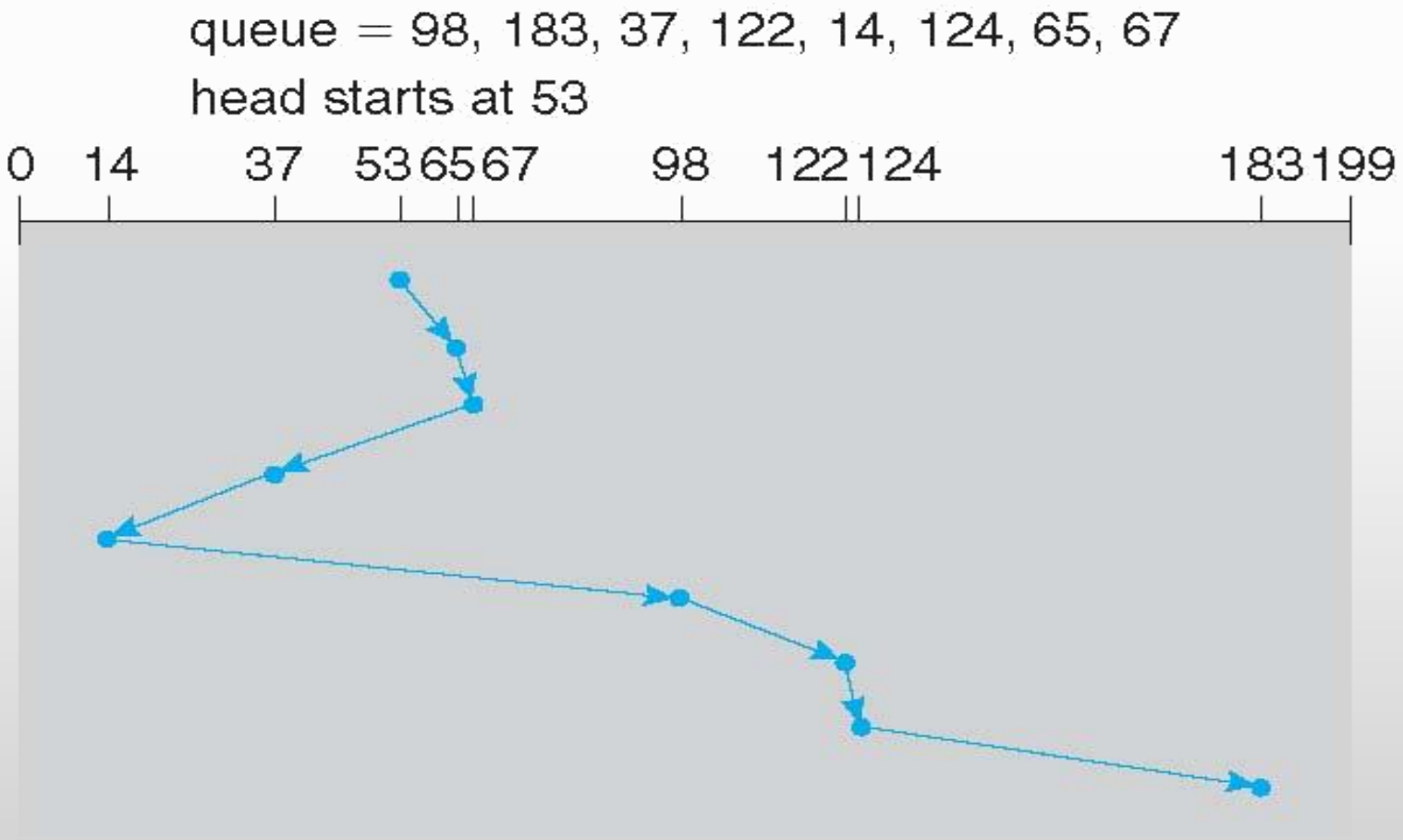
FCFS

- Illustration shows the total head movement of 640 cylinders



SSTF

- Shortest Seek Time First selects the request with the minimum seek time from the current head position
- SSTF scheduling is a form of SJF scheduling; may cause starvation of some requests
- Illustration shows total head movement of 236 cylinders

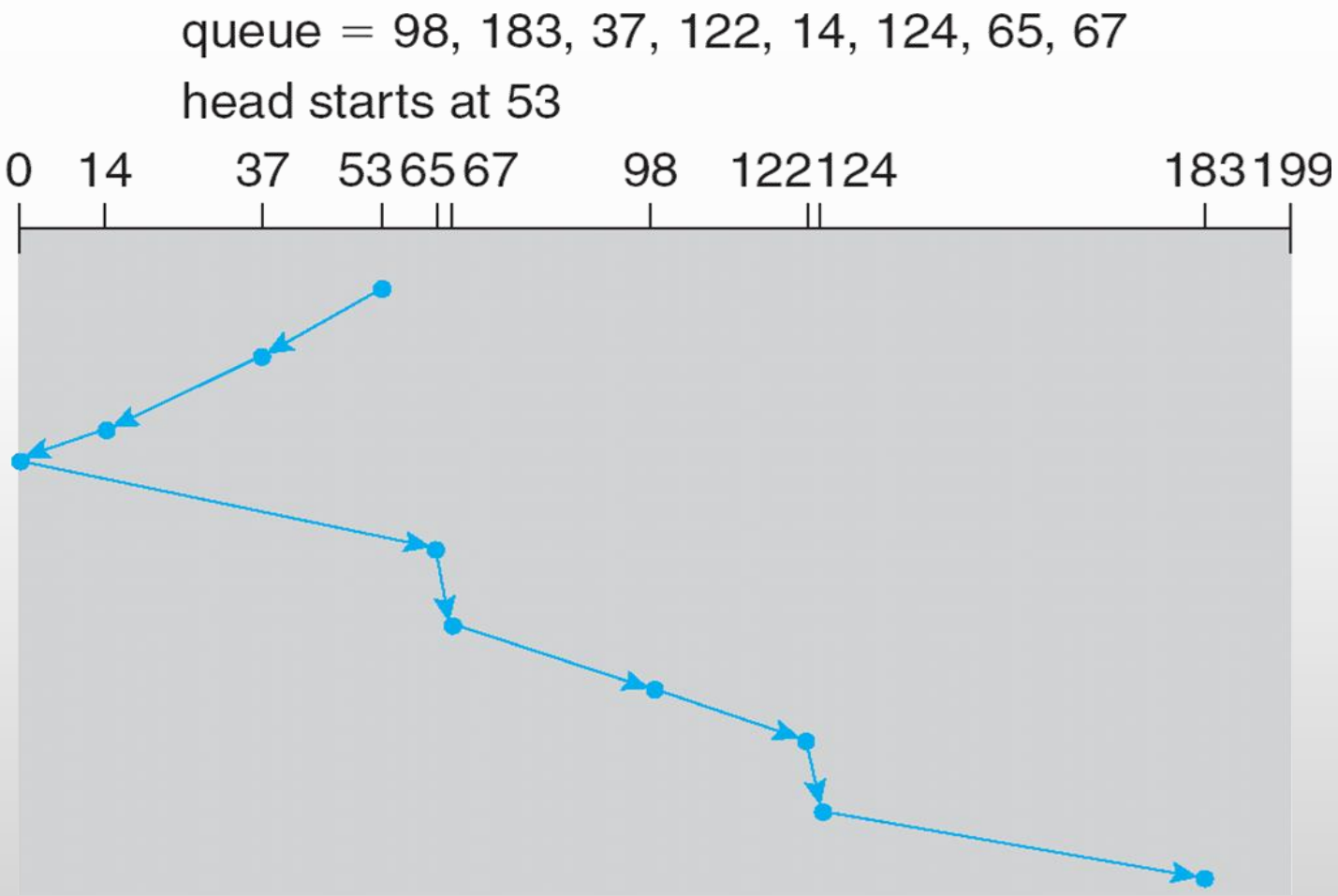


SCAN

- The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.

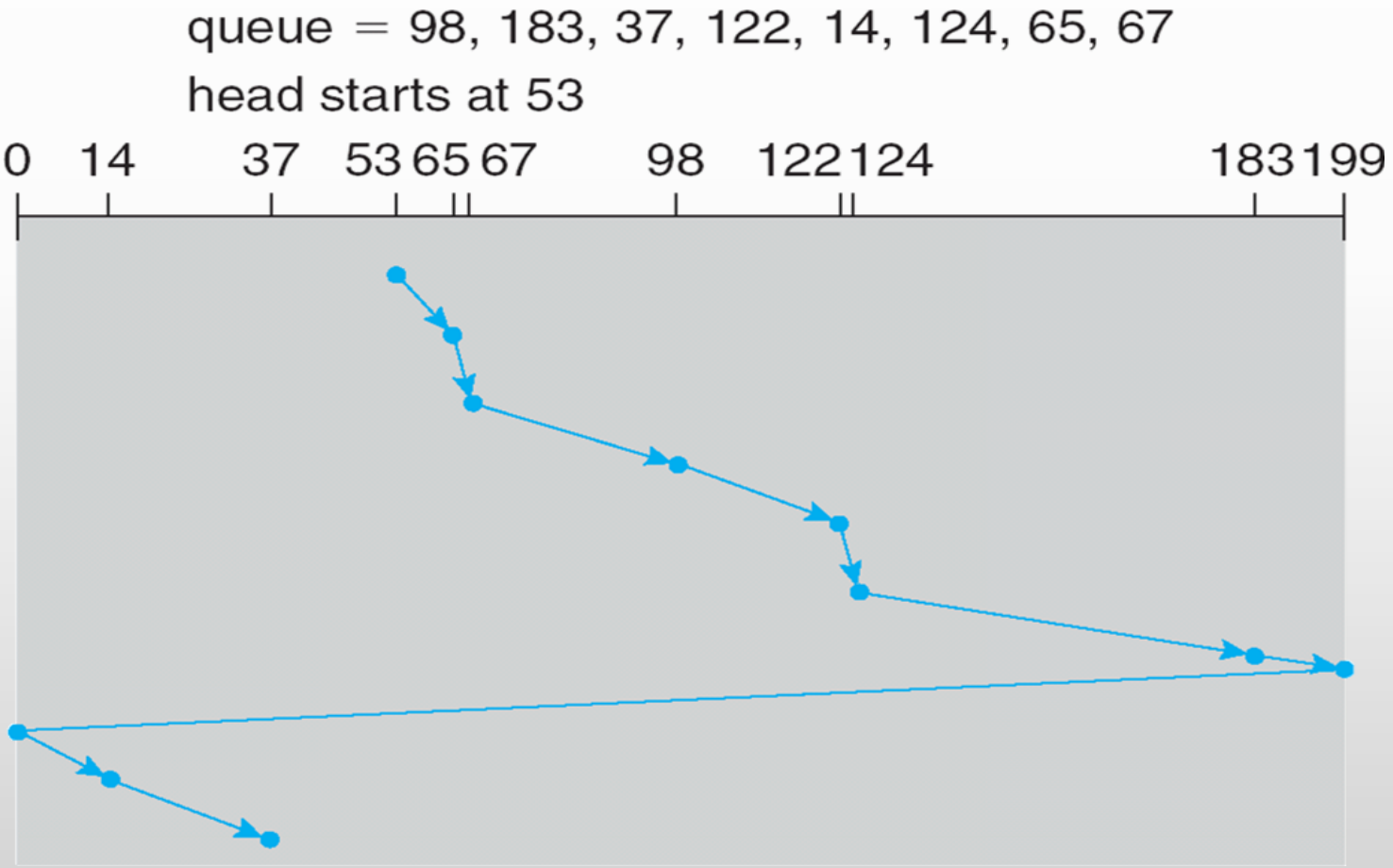
- **SCAN algorithm** Sometimes called the **elevator algorithm**

- Illustration shows total head movement of 208 cylinders



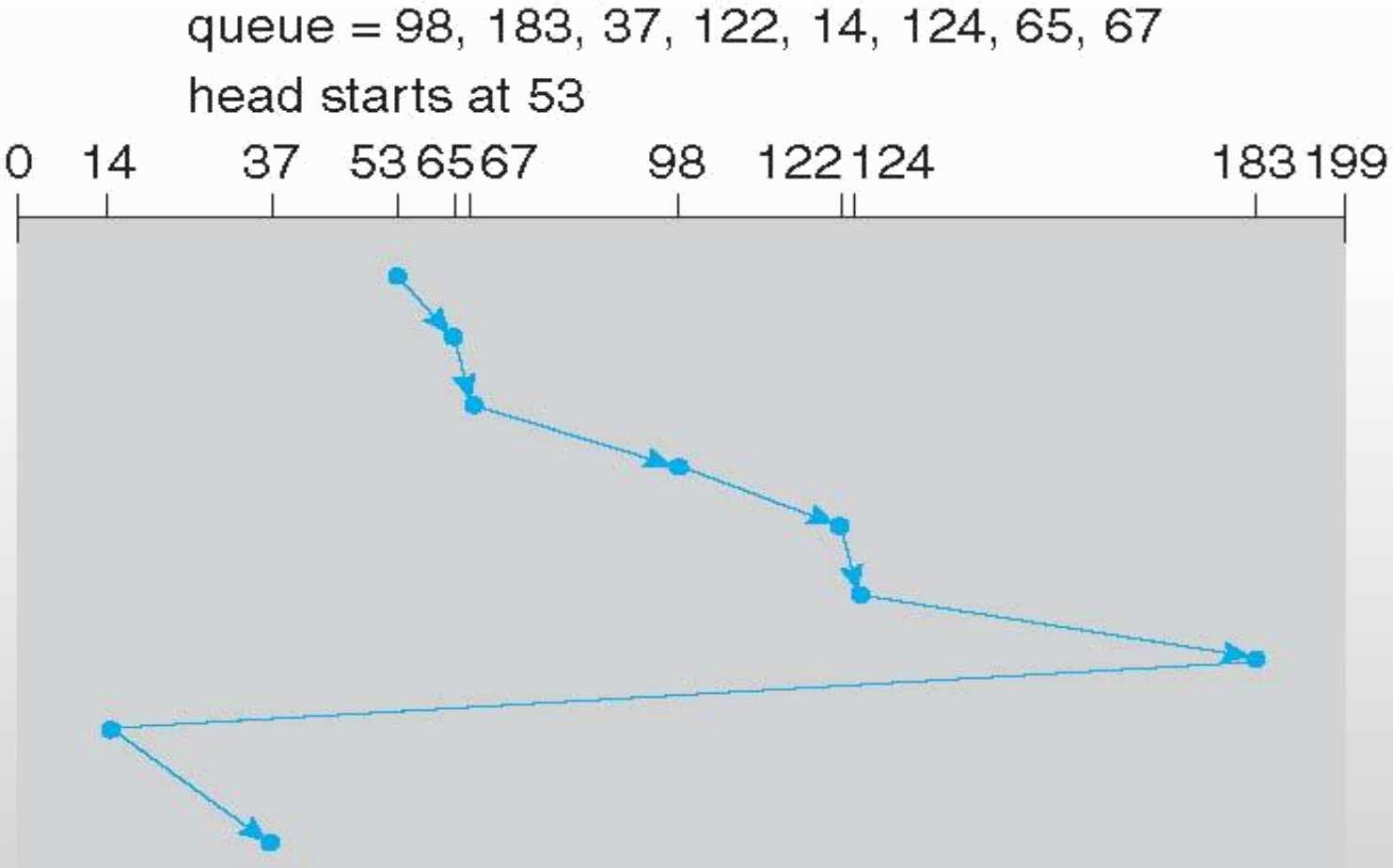
C-SCAN

- Provides a more uniform wait time than SCAN
- The head moves from one end of the disk to the other, servicing requests as it goes
  - When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip
- Treats the cylinders as a circular list that wraps around from the last cylinder to the first one



C-LOOK

- LOOK a version of SCAN, C-LOOK a version of C-SCAN
- Arm only goes as far as the last request in each direction, then reverses direction immediately, without first going all the way to the end of the disk
- Total number of cylinders?



Selecting a Disk-Scheduling Algorithm

- SSTF is common and has a natural appeal
- SCAN and C-SCAN perform better for systems that place a heavy load on the disk
  - Less starvation
- Performance depends on the number and types of requests
- The disk-scheduling algorithm should be written as a separate module of the operating system, allowing it to be replaced with a different algorithm if necessary
- Either SSTF or LOOK is a reasonable choice for the default algorithm
- What about rotational latency?
  - Difficult for OS to calculate

Question & Discussion

Task Assign

Thank You

kohinoor_cse@lus.ac.bd