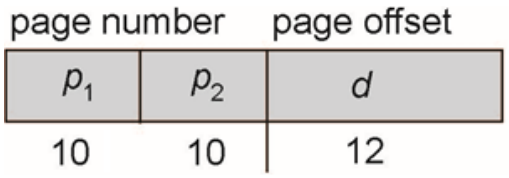
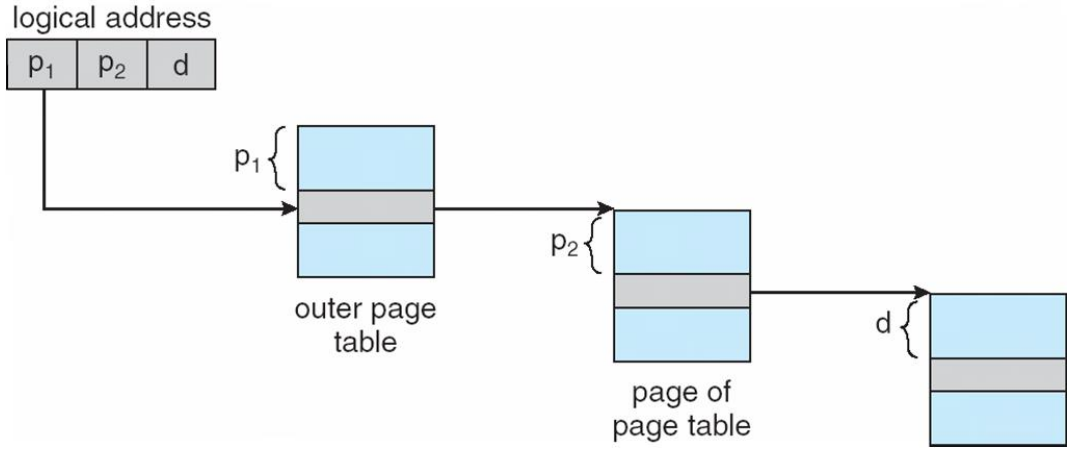
Two-Level Paging Example

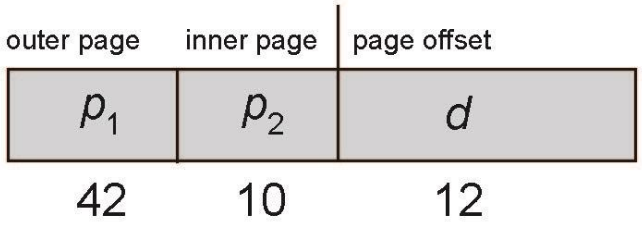
- A logical address (on 32-bit machine with 4K page size) has:
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits
- Since the page table is paged, the page number is further divided into p_1 and p_2.
- Where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table (**forward-mapped page table)**





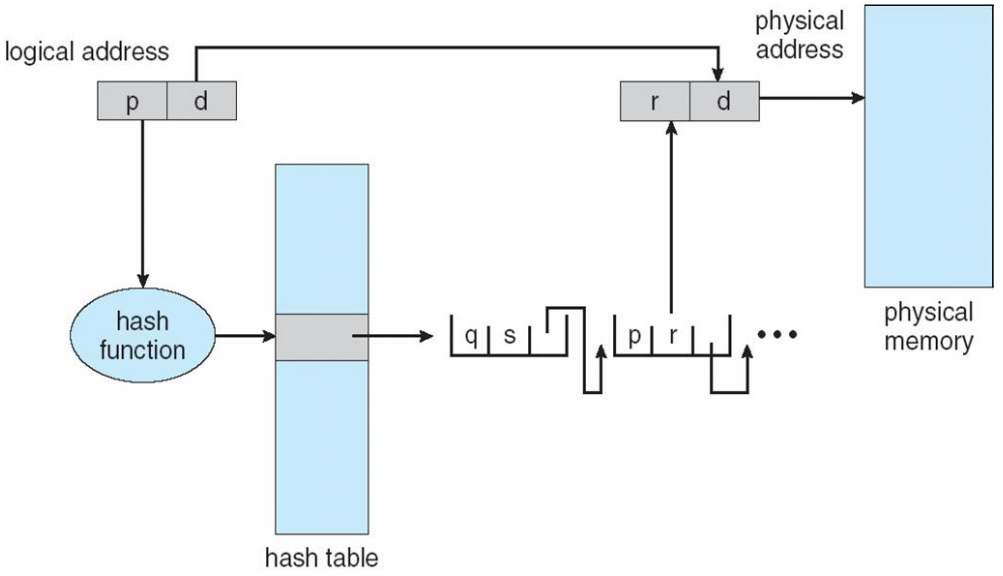
64-bit Logical Address Space

- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Outer page table has 2^{42} entries or 2^{44} bytes
  - One solution is to add a 2^{nd} outer page table
  - But in the following example the 2^{nd} outer page table is still 2^{34} bytes in size
    - And possibly 4 memory access to get to one physical memory location
- Even two-level paging scheme not enough



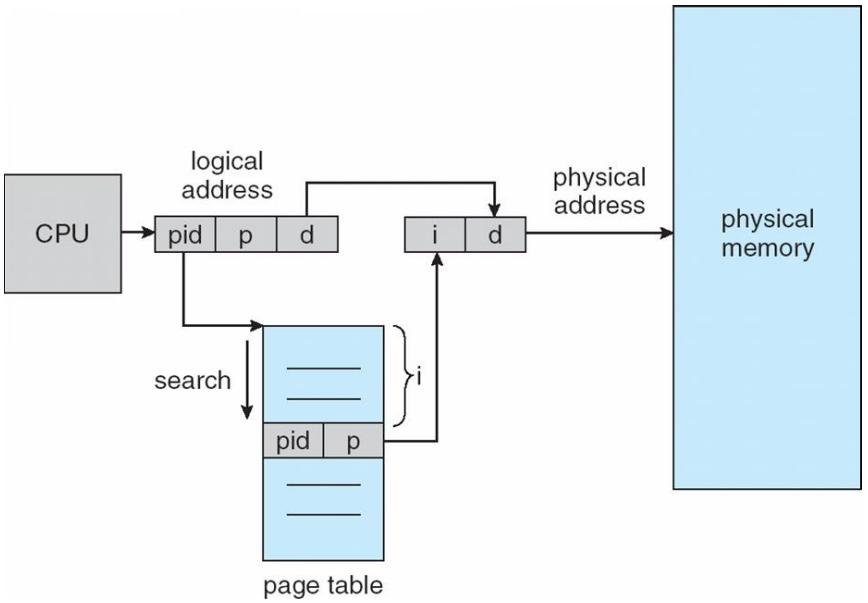
Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table containing a chain of elements hashing to the same location
- Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- Virtual page numbers are compared in this chain searching for a match



Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- One entry for each real page of memory
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- shared memory more difficult to implement



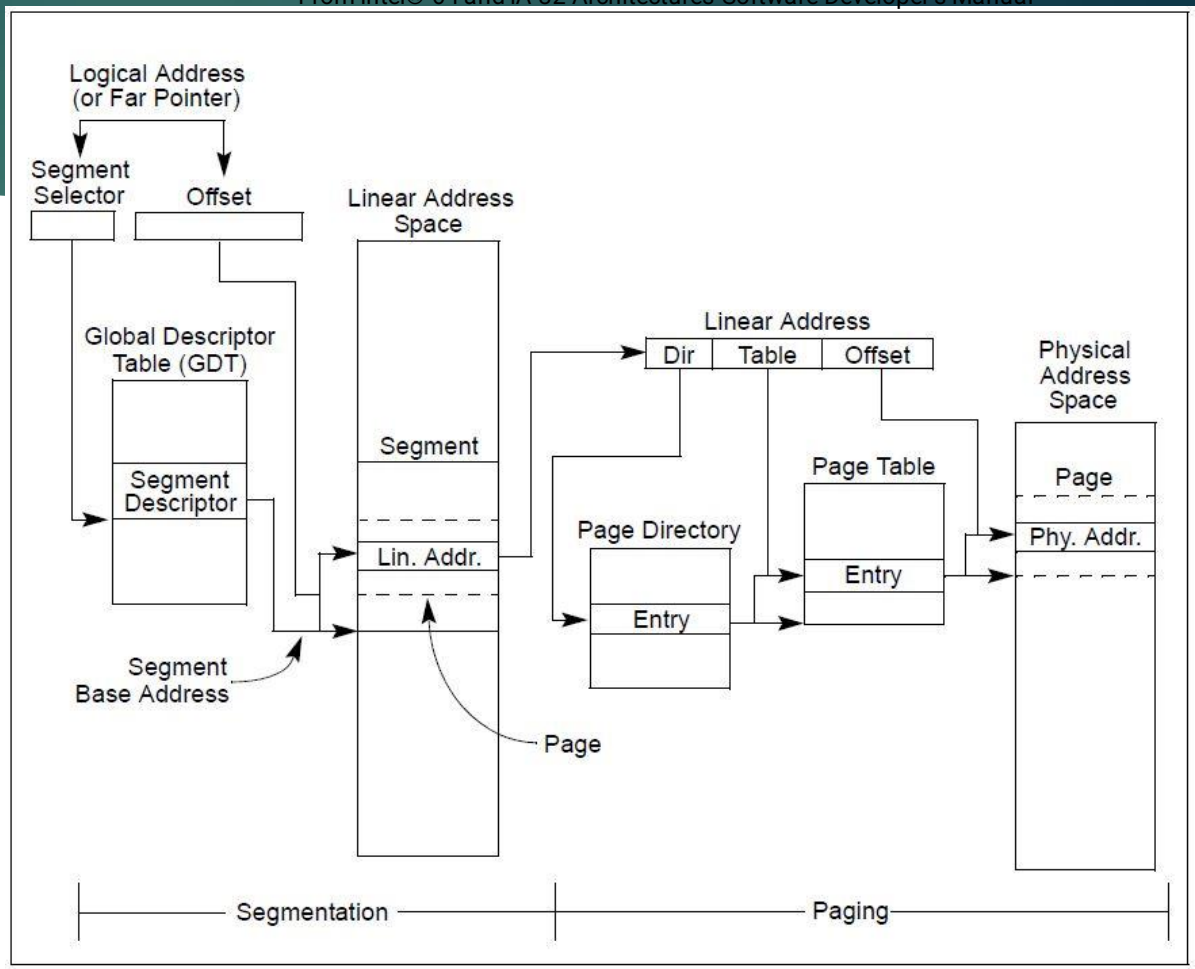
Structure of the Page Table

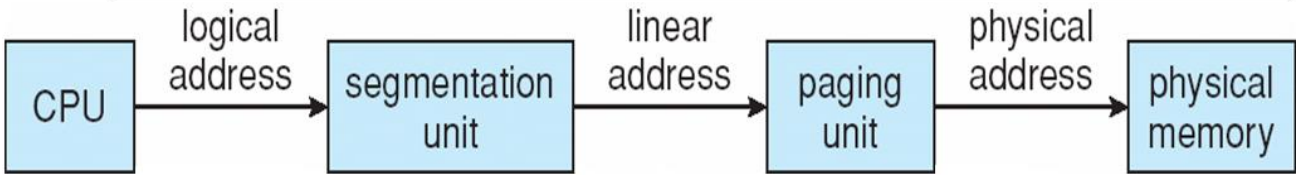
- Memory structures for paging can get huge using straight-forward methods
- Solutions:
  - Hierarchical Paging
  - Hashed Page Tables
  - Inverted Page Tables

Combining Segmentation and Paging (IA32)

- Each segment is organized as a set of pages.
- TLB caches page numbers (in linear address) to frame numbers (in physical address)

From Intel® 64 and IA-32 Architectures Software Developer's Manual





IV. Virtual Memory

SGG9: chapter 9
SGG10: chapter 10

- Virtual Mem.: demand paging, replacement algorithms, thrashing

TDIU11: Operating Systems

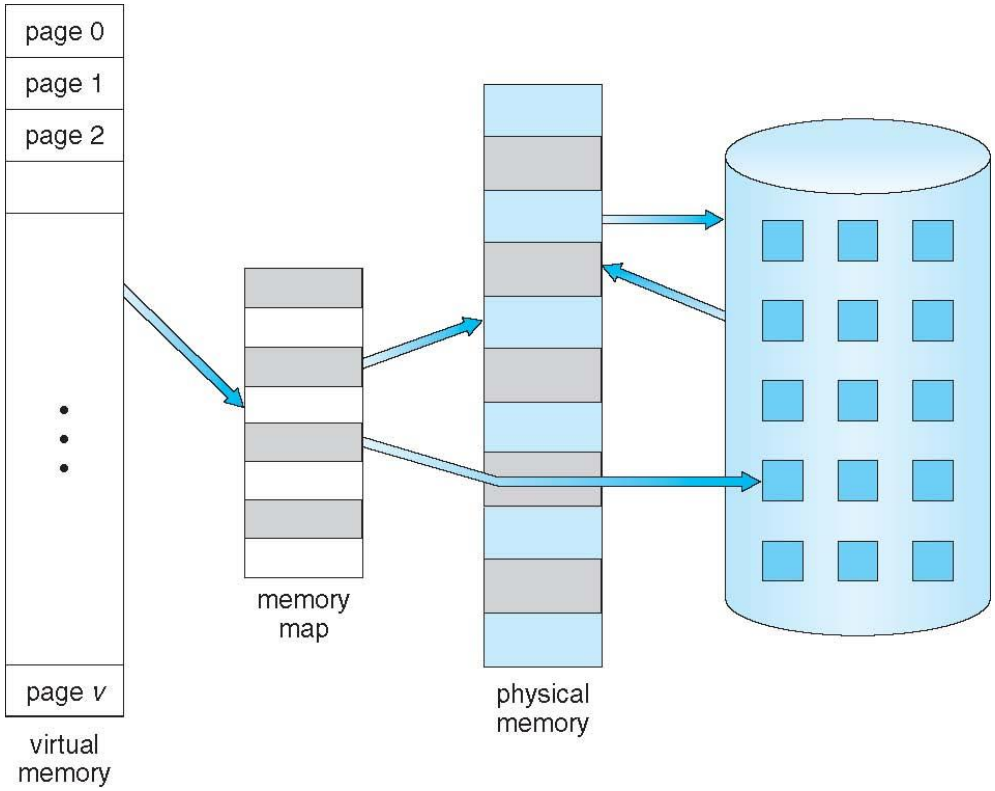
Ahmed Rezine, Linköping University

Virtual Memory: Background

- Entire program rarely used: error code, unusual routines, large data structures
- Entire program code not needed at same time. Execute partially-loaded 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, no increase in response time or turnaround time
- Less I/O needed to load or swap programs into memory. Each user program runs faster

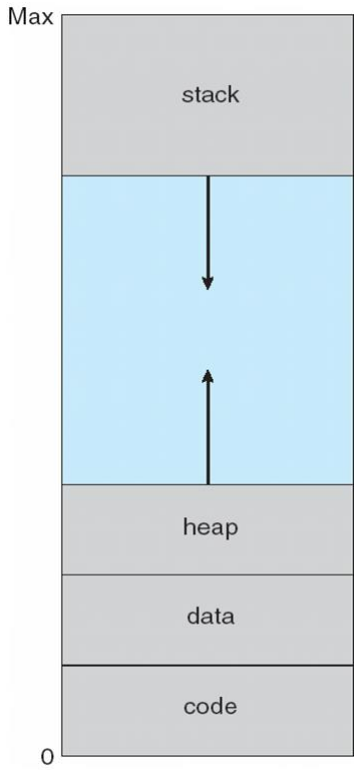
Virtual memory can be implemented via: demand paging or demand segmentation

Virtual Memory That is Larger than Physical Memory

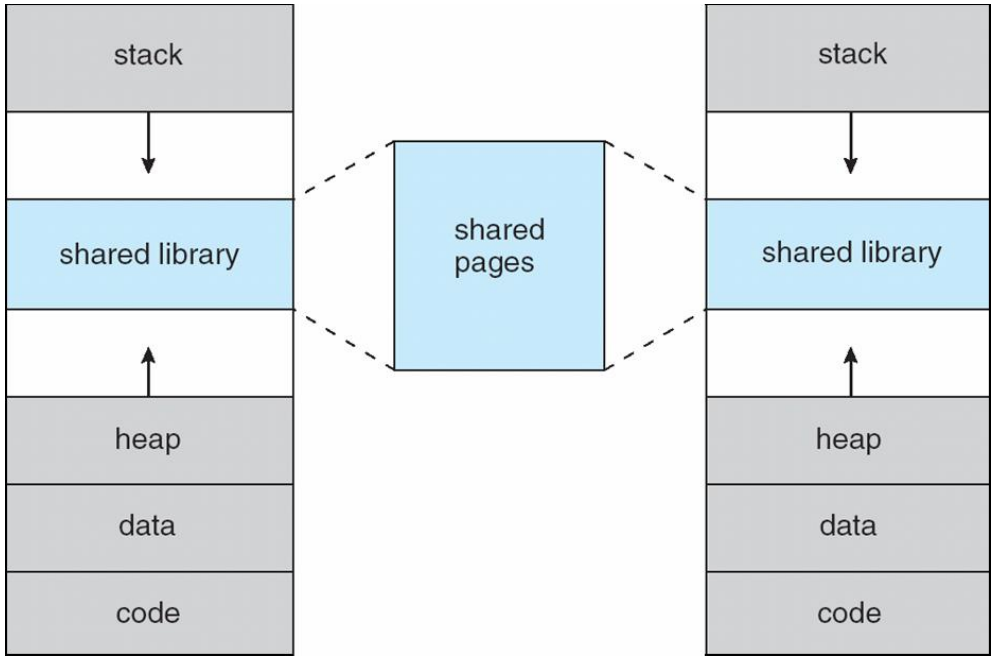


Virtual-address Space

- Usually, stack starts at Max logical address and grows "down" while 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
- 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

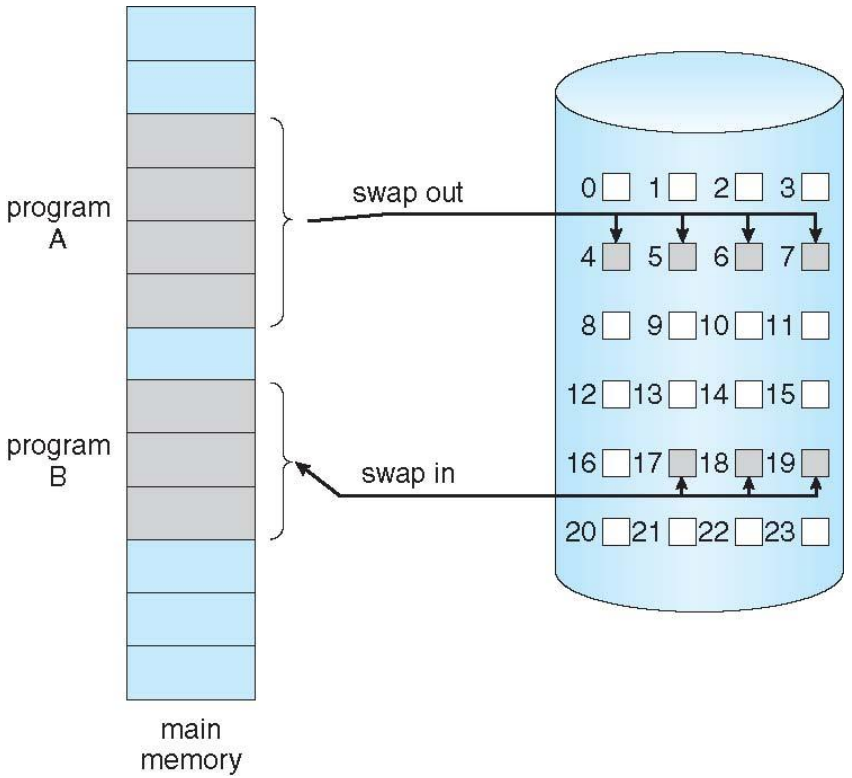


Shared Library Using Virtual Memory

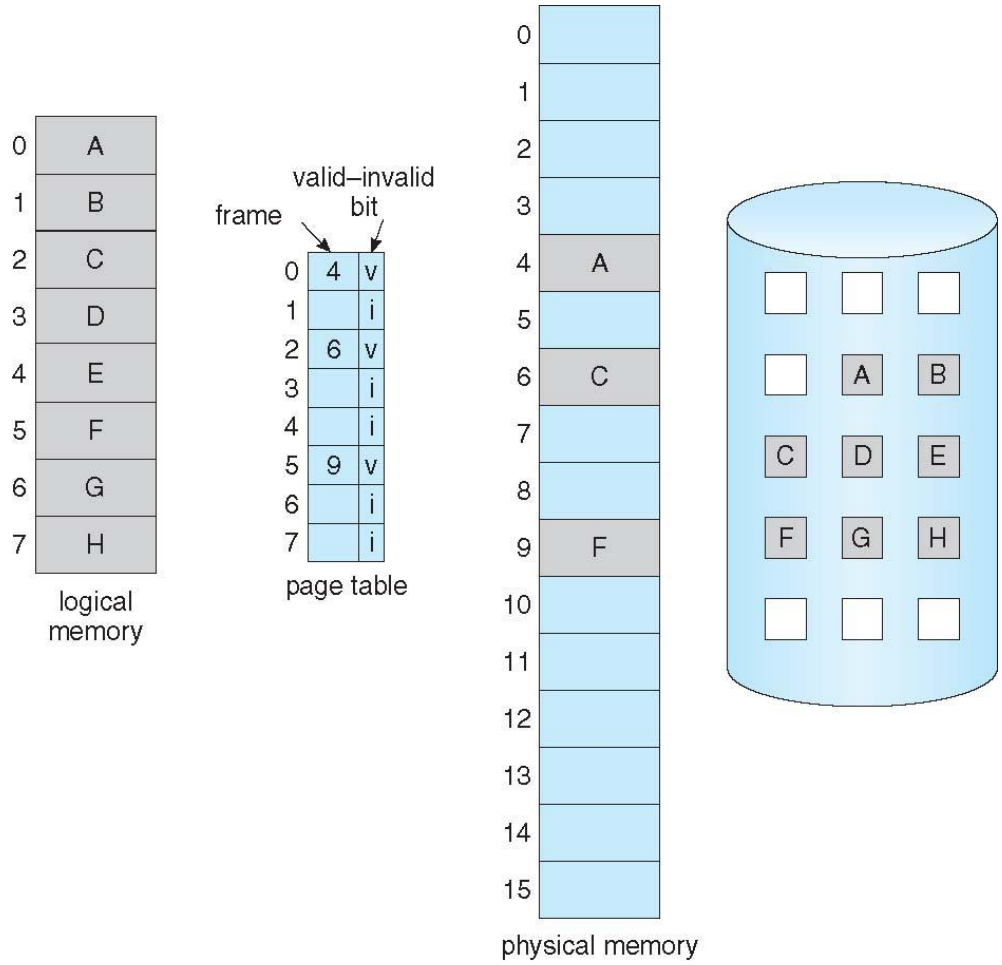


Virtual Memory via Demand Paging

- 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
- Like paging system with swapping
- 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 (instead of entire processes) is a **pager**

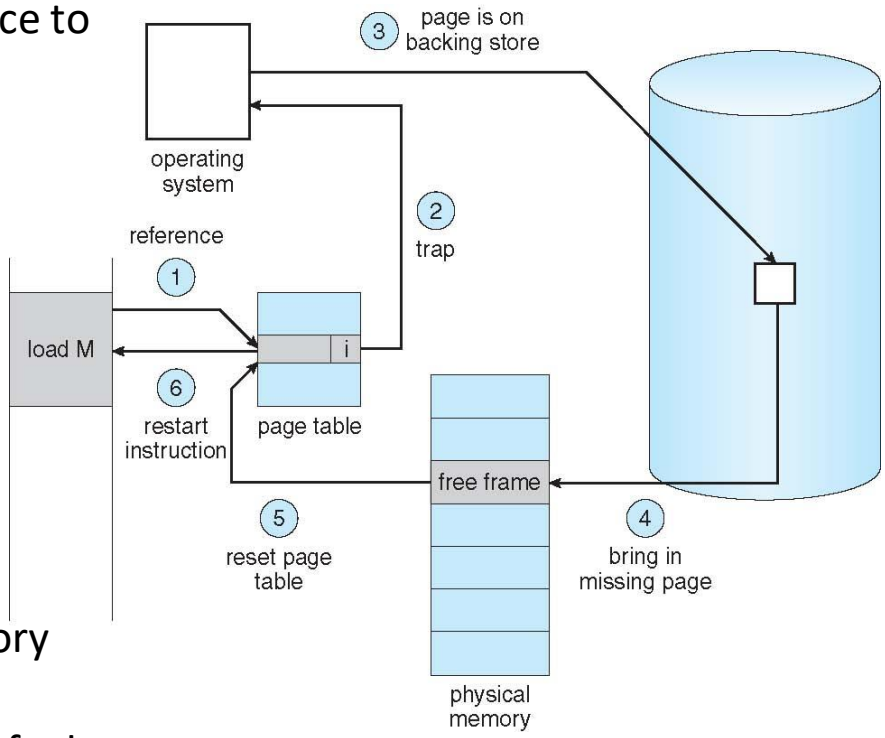


Page Table When Some Pages Are Not in Main Memory



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 $\Rightarrow$ abort
  - Just not in memory
- Find free frame
- Swap page into frame via scheduled disk operation
- Update tables to indicate page now in memory
  Set validation bit = **v**
- Restart the instruction that caused the page fault



Stages/Performance of Demand Paging

- Trap to the operating system
- Save the user registers and process state
- Determine that the interrupt was a page fault
- Check that the page reference was legal and determine disk page location
- Issue a read from the disk to a free frame:
  - Wait in a queue for this device until the read request is serviced
  - Wait for the device seek and/or latency time
  - Begin the transfer of the page to a free frame
- While waiting, allocate the CPU to some other user
- Receive an interrupt from the disk I/O subsystem (I/O completed)
- Save the registers and process state for the other user
- Determine that the interrupt was from the disk
- Correct the page table and other tables to show page is now in memory
- Wait for the CPU to be allocated to this process again
- Restore the user registers, process state, and new page table, and then resume the interrupted instruction

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.

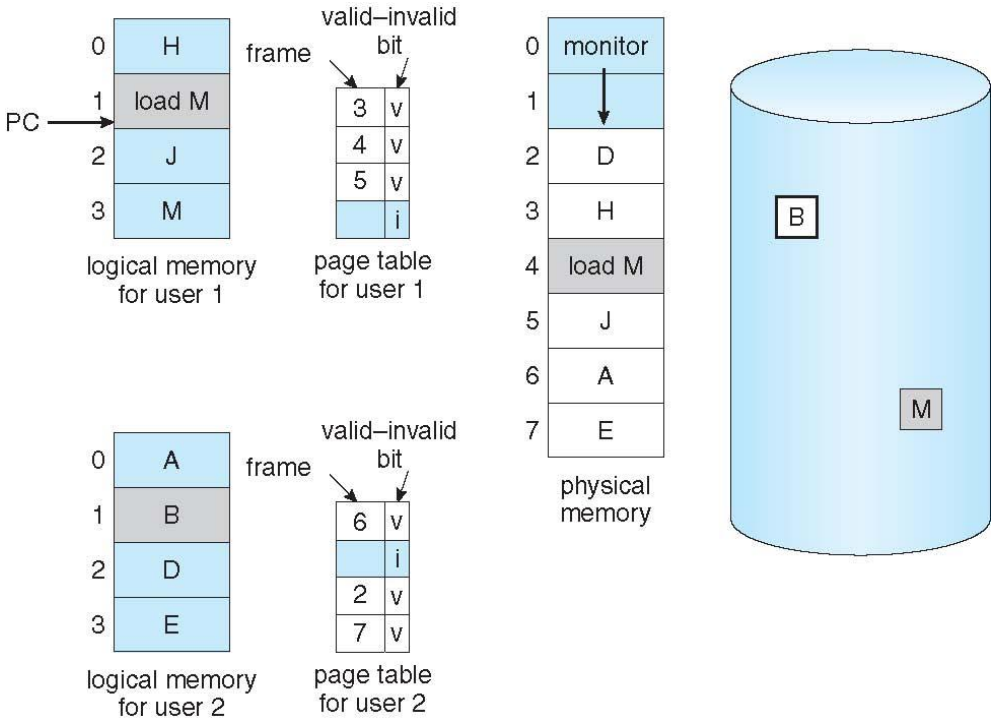
  This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 x p
  - 20 > 7,999,800 x p
  - p < .0000025

  Less than one page fault in every 400,000 memory accesses

Page Replacement

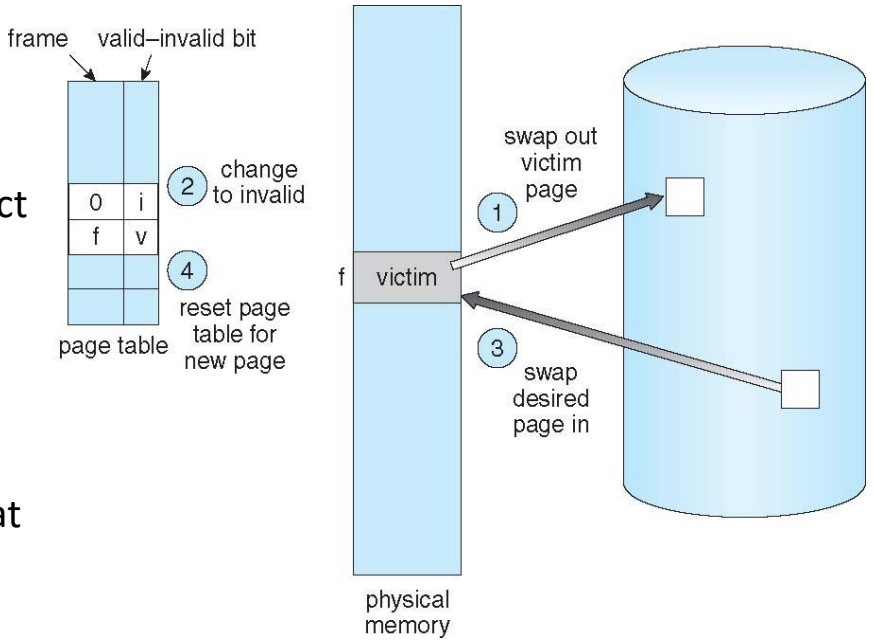
- Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement
- Use **modify** (**dirty**) **bit** to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory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

Need For Page Replacement



Basic Page Replacement

- Find the location of the desired page on disk
- Find a free frame:
  - If there is a free frame, use it
  - If not, 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; update the page and frame tables
- Continue the process by restarting the instruction that caused the trap

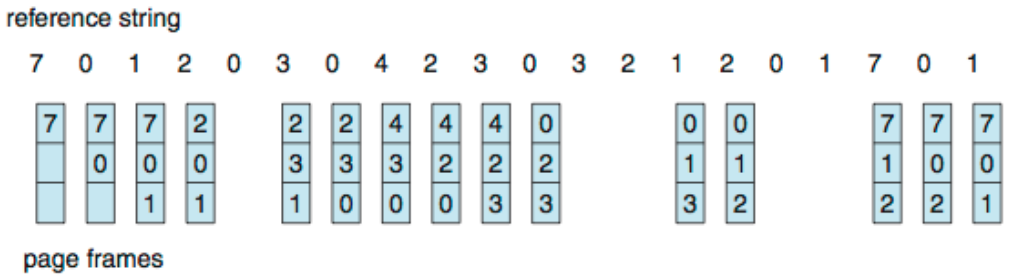


- Note now potentially 2 page transfers for page fault – increasing EAT

Page and Frame Replacement Algorithms

- **Frame-allocation algorithm**: how many frames to each process
- **Page-replacement algorithm**: lowest page-fault rate on first access and re-access
- Evaluate page-replacement 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

First-In-First-Out (FIFO) Algorithm

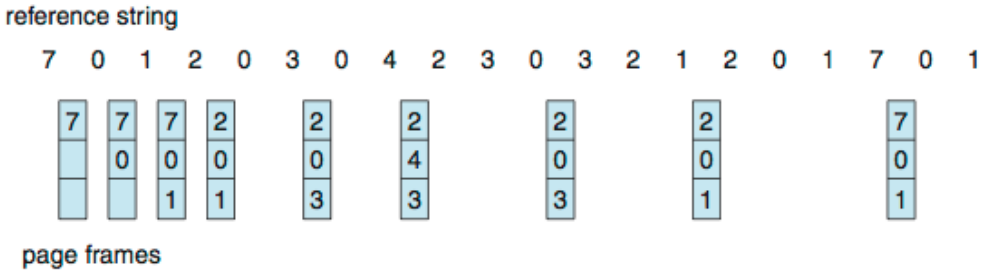


15-page faults

7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

- Adding more frames can cause more page faults!
- How to track ages of pages? Just use a FIFO queue

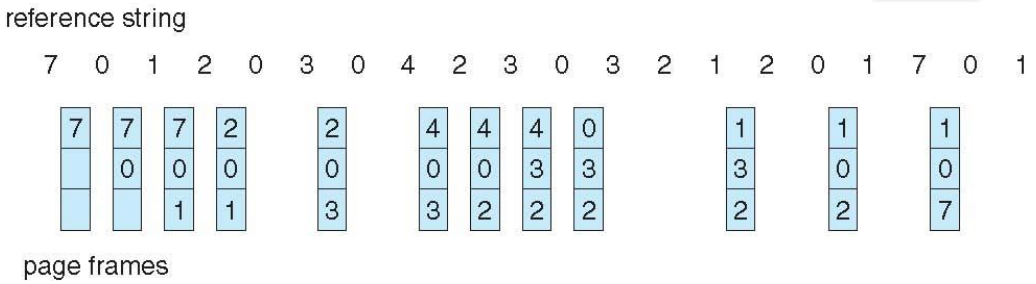
Optimal Algorithm



- Replace 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 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 Approximation Algorithms

- LRU needs special hardware and still slow
- **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced, bit set to 1
  - Replace any with reference bit = 0
- **Second-chance algorithm**
  - Generally, FIFO plus hardware-provided reference bit
  - **Clock** replacement
  - If page to be replaced has
    - Reference bit = 0 -> replace it
    - reference bit = 1 then:
      - set reference bit 0, leave in memory
      - replace next page, subject to same rules

Counting 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
- **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

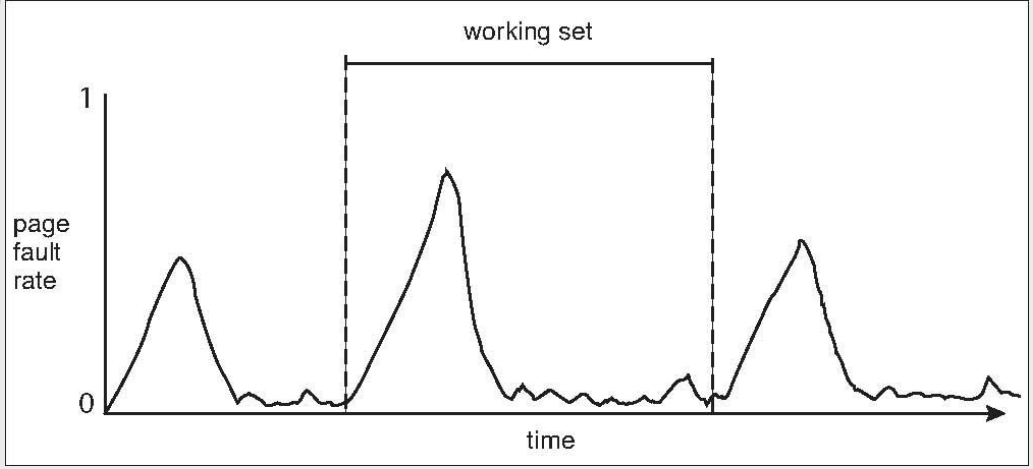
Page-Buffering Algorithms

- Keep a pool of free frames, always
  - 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

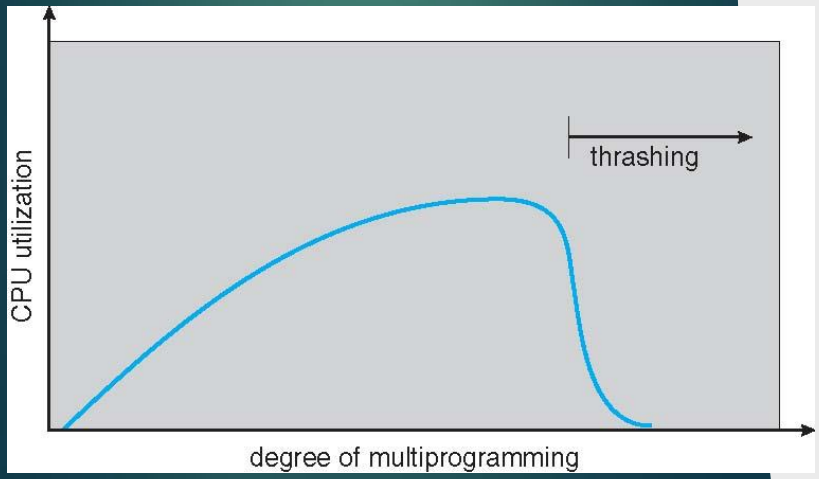
Allocation and Replacement of Frames

- Allocation:
  - 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,
  - Priority allocation: allocate proportionally to process priority
- **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

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



IV. B. Mass Storage Systems

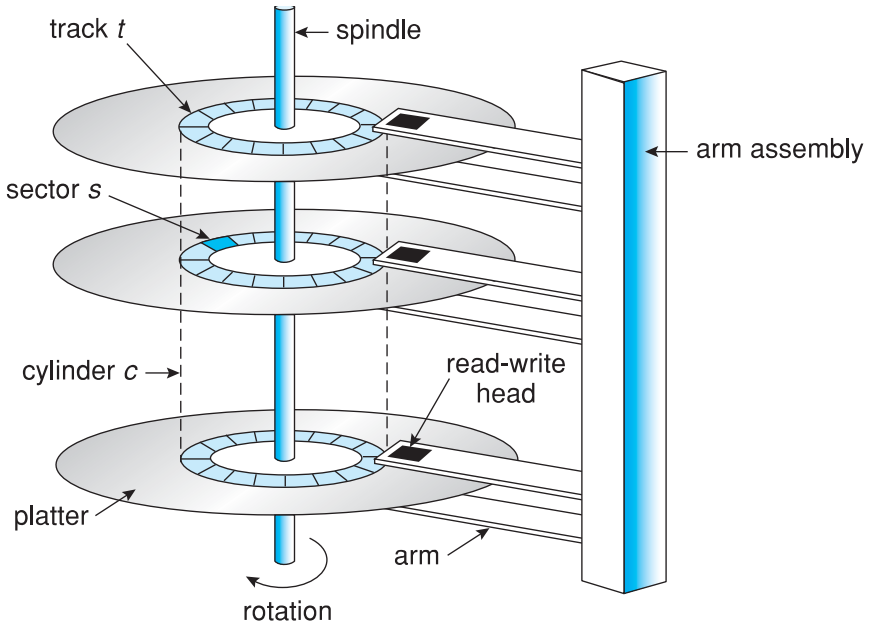
SGG9: chapters 12
SGG10: chapters 11

- Mass storage: hard disks, structure, disk scheduling

Overview of Mass Storage Structure

Magnetic disks bulk of secondary storage:

- Drives rotate at 60 to 250 times per second
- **Transfer rate** is rate at which data flow between drive and computer
- **Positioning time** (**random-access time**) is time to move disk arm to desired cylinder (**seek time**) and time for desired sector to rotate under the disk head (**rotational latency**)



Hard Disks

- Platters from .85” to 14”: commonly 3.5”, 2.5”
- Up to 10TB per drive
- Performance
  - Transfer Rate – around 1Gb/sec
  - Seek time from 3ms to 12ms – 9ms common for desktop drives
  - Average seek time measured or calculated based on 1/3 of tracks
  - Latency based on spindle speed
    - 1 / (RPM / 60) = 60 / RPM
  - Average latency = ½ latency

Spindle [rpm]	Average latency [ms]
4200	7.14
5400	5.56
7200	4.17
10000	3
15000	2

Hard Disk Performance

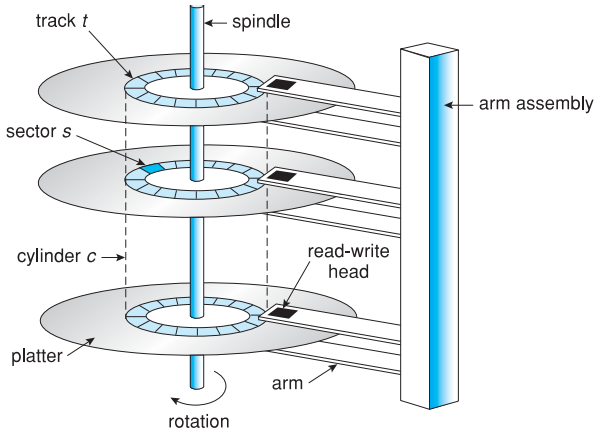
- **Access Latency** = **Average access time** = average seek time + average latency
  - For fastest disk 3ms + 2ms = 5ms
  - For slow disk 9ms + 5.56ms = 14.56ms
- Average I/O time = average access time + (amount to transfer / transfer rate) + controller overhead
- For example, to transfer a 4KB block on a 7200 RPM disk with a 5ms average seek time, 1Gb/sec transfer rate with a .1ms controller overhead =
  - 5ms + 4.17ms + 0.1ms + transfer time =
  - Transfer time = 4KB / 1Gb/s * 8Gb / GB * 1GB / 1024^2KB = 32 / (1024^2) = 0.031 ms
  - Average I/O time for 4KB block = 9.27ms + .031ms = 9.301ms

Solid-State Disks

- Nonvolatile memory used like a hard drive
  - Many technology variations
- Can be more reliable than HDDs
- More expensive per MB
- Maybe have shorter life span
- Less capacity
- But much faster
- No moving parts, so no seek time or rotational latency

Disk Structure

- Disk drives are addressed as large 1-dimensional arrays of **logical blocks**, where the logical block is the smallest unit of transfer
  - Low-level formatting creates **logical blocks** (typically 512B)
- The 1-dimensional array of logical blocks is mapped into the sectors of the disk sequentially
  - Sector 0 is the first sector of the first track on the outermost cylinder
  - Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost



Disk Scheduling

- 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, Users 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

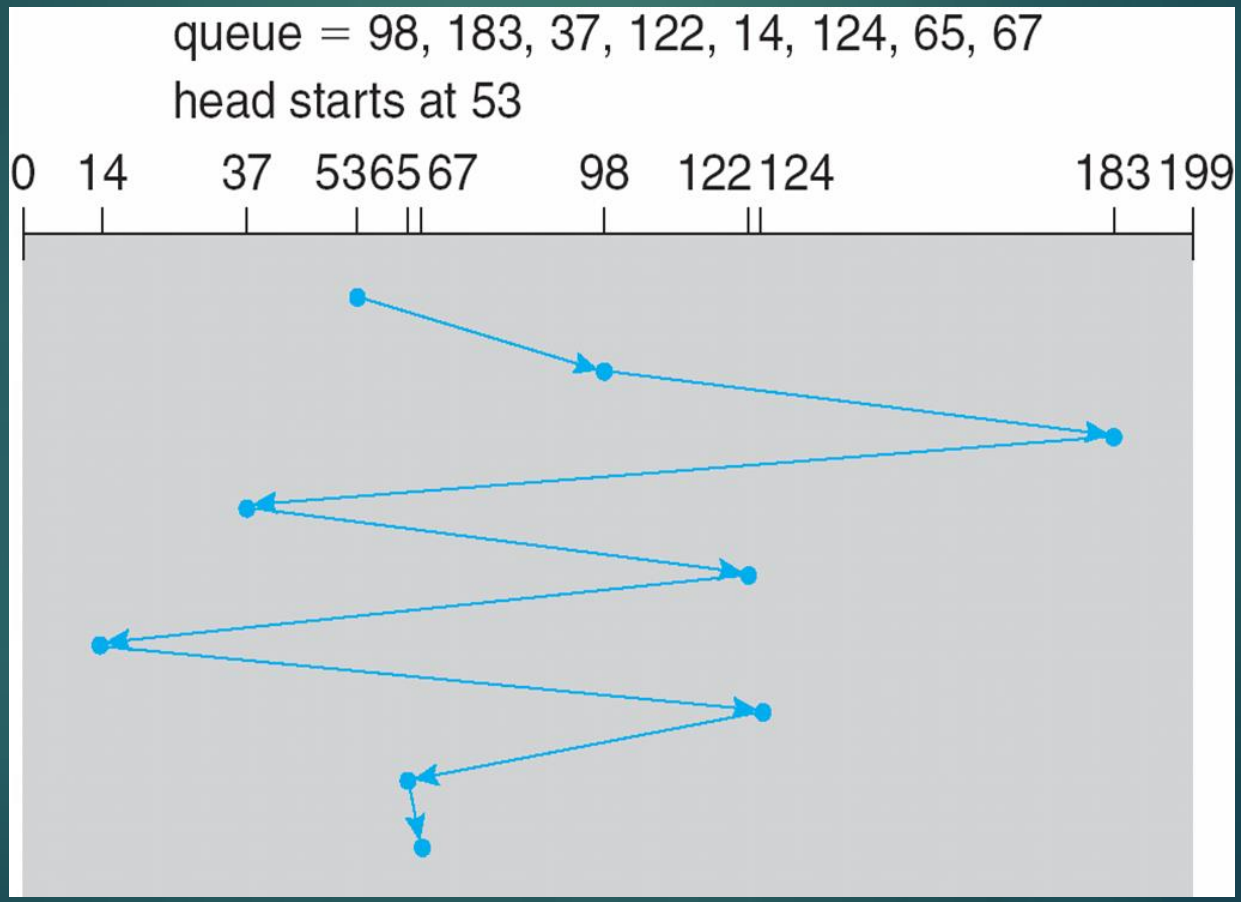
Disk Scheduling (Cont.)

- 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

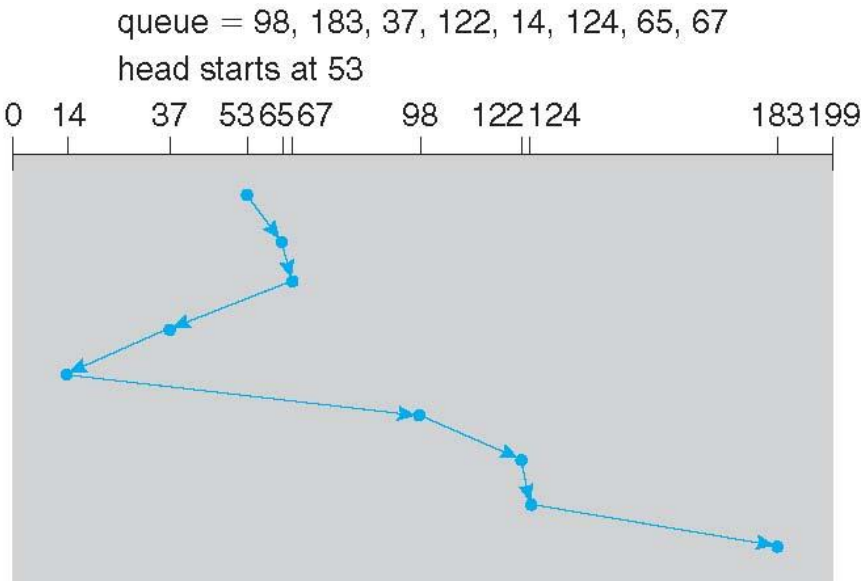
FCFS

Illustration shows total head movement of 640 cylinders



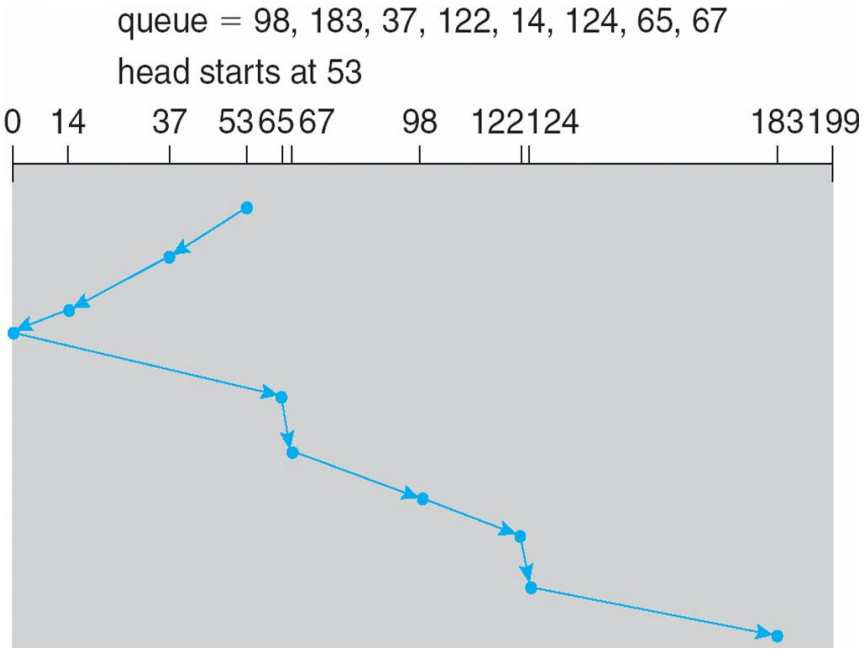
SSTF

- 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



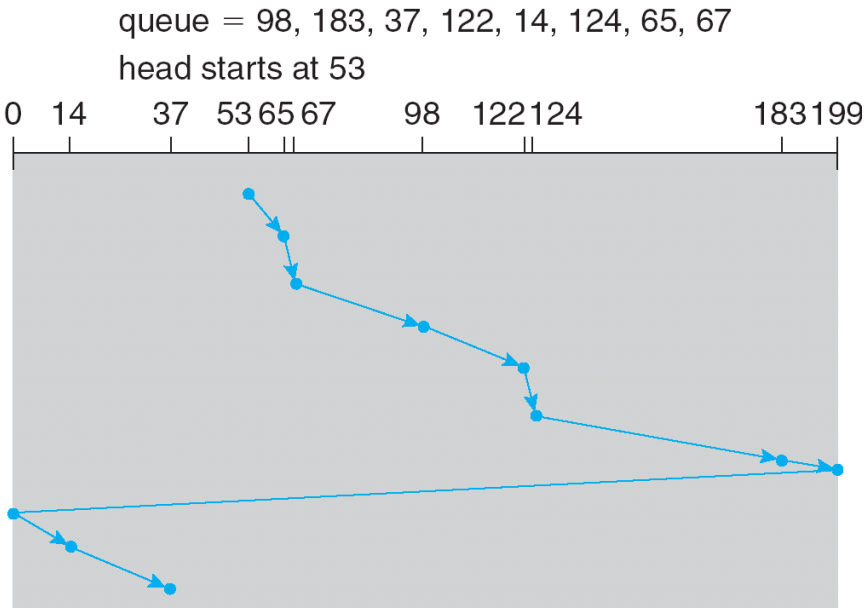
SCAN

- 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



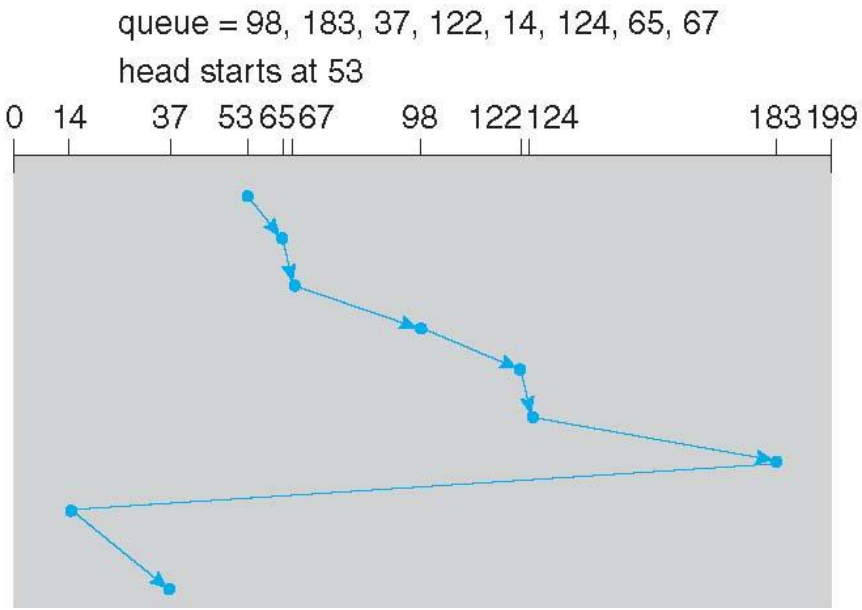
C-SCAN

- Provides a more uniform wait time than SCAN
- The head moves from one end of the disk to the other:
  - When it reaches the other end, however, it immediately returns to the beginning of the disk
- 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
- Requests for disk service can be influenced by the file-allocation method and metadata layout
- 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
- How does disk-based queueing effect OS queue ordering efforts?