Operating Systems
(Honor Track)

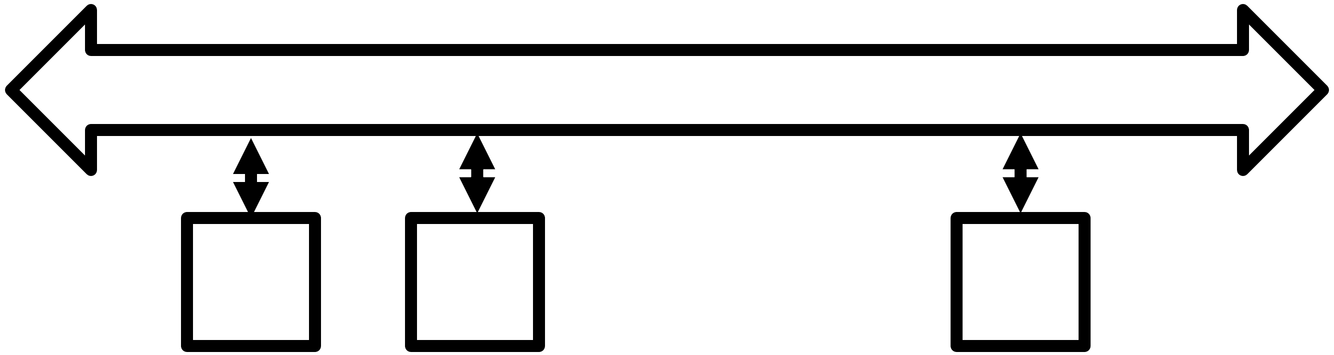
File System 1: IO Performance, File System Design

Xin Jin

Spring 2023

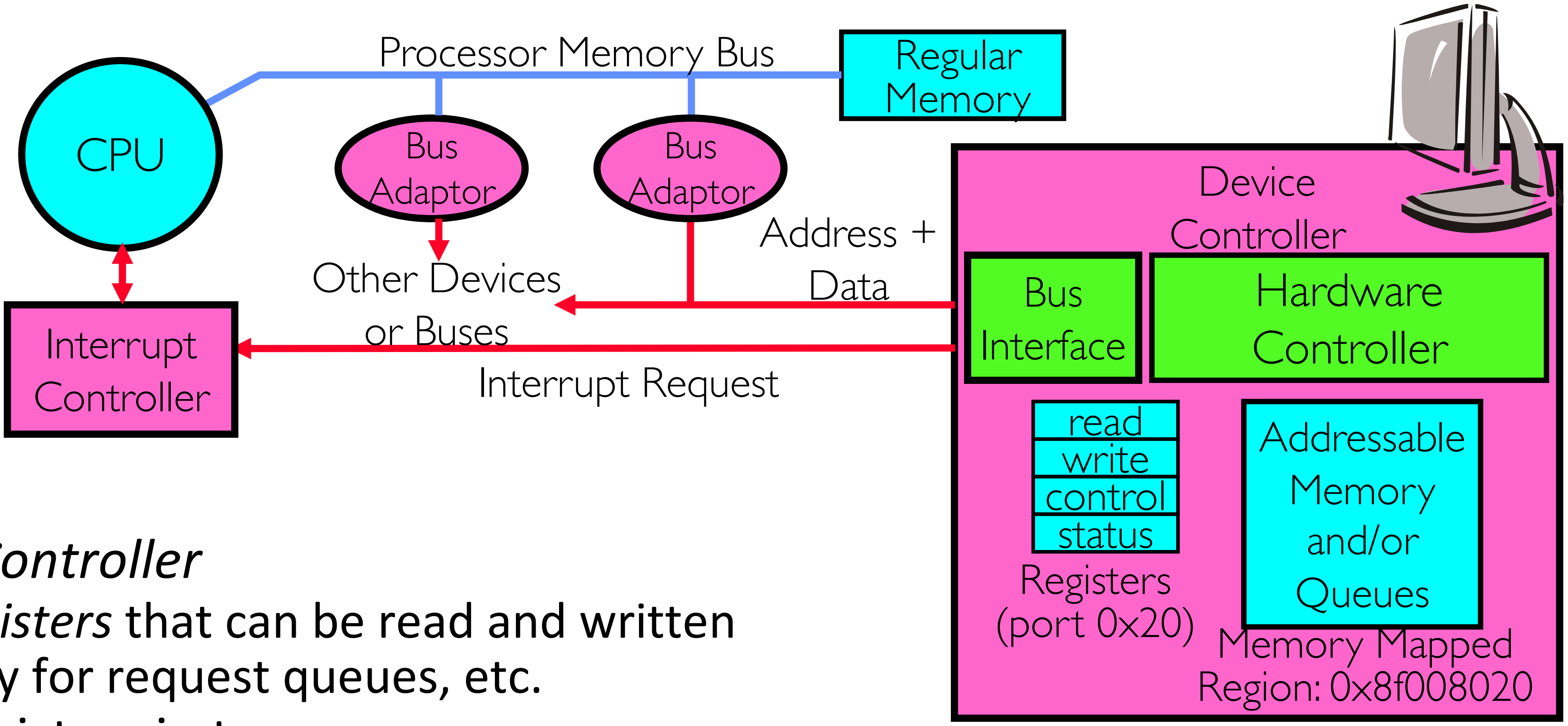
Acknowledgments: Ion Stoica, Berkeley CS 162

Recap: What's a bus?



- Common set of wires for communication among hardware devices plus protocols for carrying out data transfer transactions
  - Operations: e.g., Read, Write
  - Control lines, Address lines, Data lines
  - Typically, multiple devices
- Protocol: initiator requests access, arbitration to grant, identification of recipient, handshake to convey address, length, data
- Very high BW close to processor (wide, fast, and inflexible), low BW with high flexibility out in I/O subsystem

Recap: How does the Processor Talk to the Device?



- CPU interacts with a *Controller*
  - Contains a set of *registers* that can be read and written
  - May contain memory for request queues, etc.
- Processor accesses registers in two ways:
  - Port-Mapped I/O: in/out instructions
    - » Example from the Intel architecture: `out 0x21,AL`
  - Memory-mapped I/O: load/store instructions
    - » Registers/memory appear in physical address space
    - » I/O accomplished with load and store instructions

Recap: I/O Device Notifying the OS

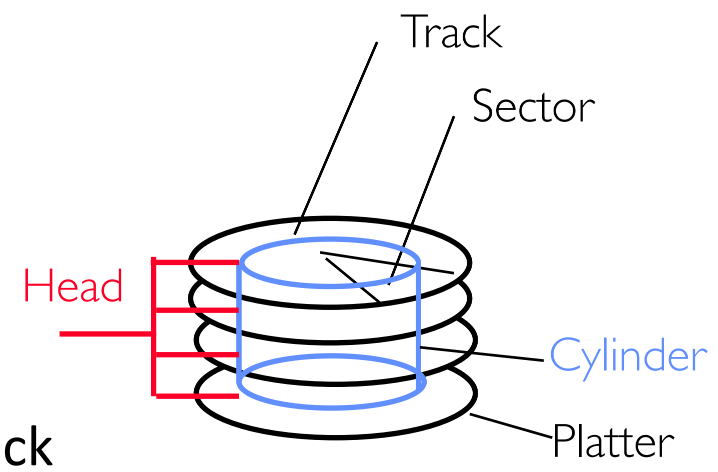
- The OS needs to know when:
  - The I/O device has completed an operation
  - The I/O operation has encountered an error
- I/O Interrupt:
  - Device generates an interrupt whenever it needs service
  - Pro: handles unpredictable events well
  - Con: interrupts relatively high overhead
- Polling:
  - OS periodically checks a device-specific status register
    - » I/O device puts completion information in status register
  - Pro: low overhead
  - Con: may waste many cycles on polling if infrequent or unpredictable I/O operations
- Actual devices combine both polling and interrupts
  - For instance – High-bandwidth network adapter:
    - » Interrupt for first incoming packet
    - » Poll for following packets until hardware queues are empty

Recap: How Does User Deal with Timing?

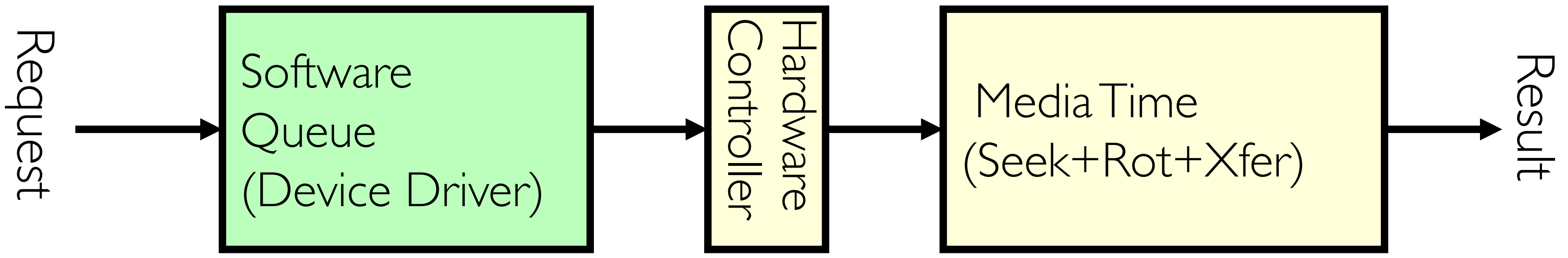
- Blocking Interface: "Wait"
  - When request data (e.g. `read()` system call), put process to sleep until data is ready
  - When write data (e.g. `write()` system call), put process to sleep until device is ready for data
- Non-blocking Interface: "Don't Wait"
  - Returns quickly from read or write request with count of bytes successfully transferred
  - Read may return nothing, write may write nothing
- Asynchronous Interface: "Tell Me Later"
  - When request data, take pointer to user's buffer, return immediately later kernel fills buffer and notifies user
  - When send data, take pointer to user's buffer, return immediately; later kernel takes data and notifies user

Recap: Magnetic Disks

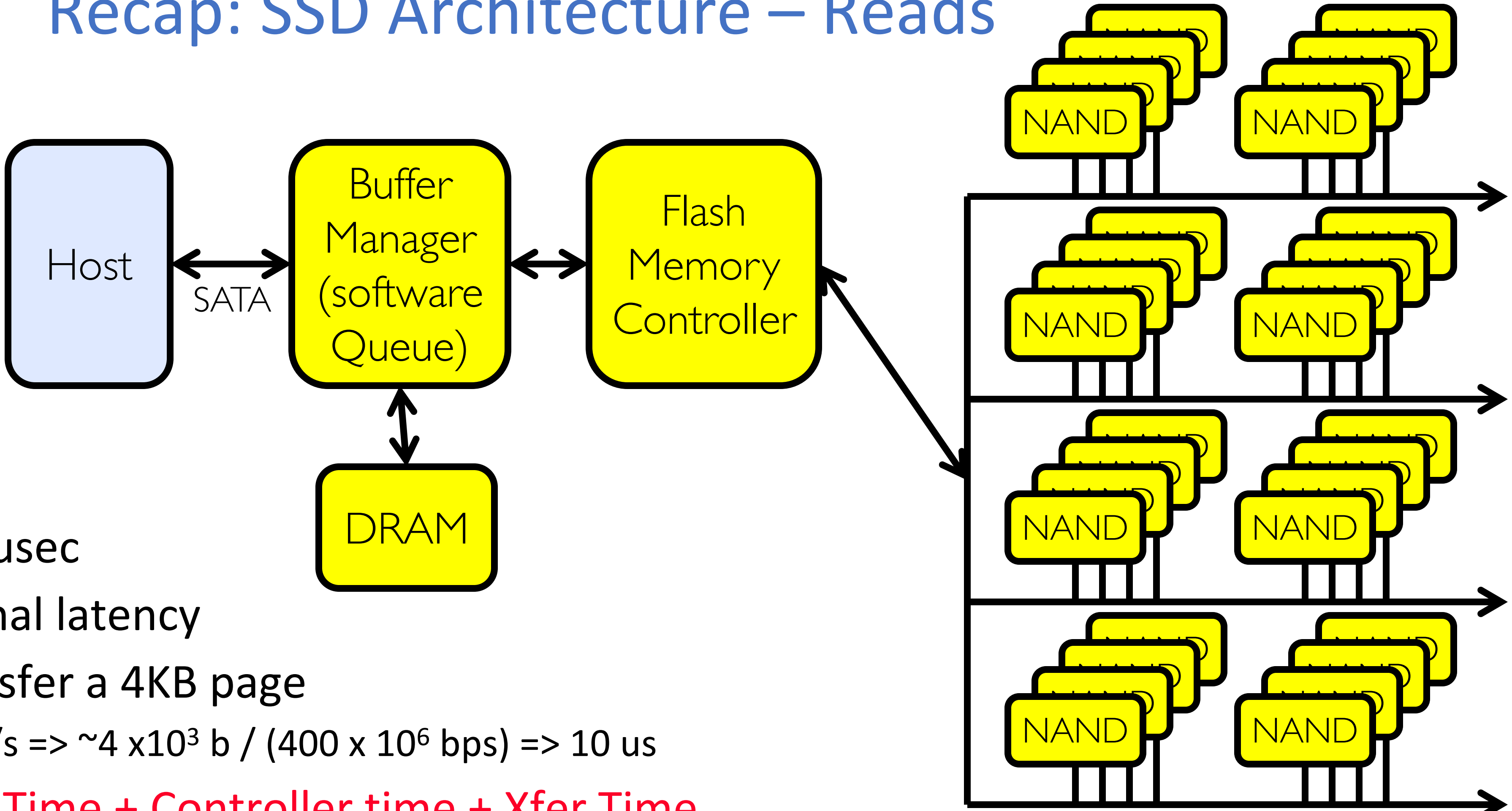
- Cylinders: all the tracks under the head at a given point on all surfaces
- Read/write data is a three-stage process:
  - Seek time: position the head/arm over the proper track
  - Rotational latency: wait for desired sector to rotate under r/w head
  - Transfer time: transfer a block of bits (sector) under r/w head



Disk Latency = Queueing Time + Controller time +
Seek Time + Rotation Time + Xfer Time



Recap: SSD Architecture – Reads

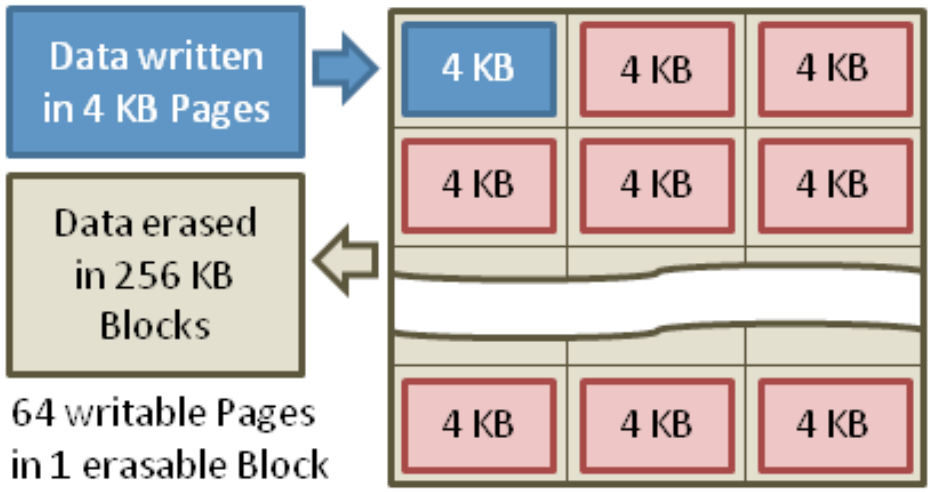


Read 4 KB Page: ~25 usec

- No seek or rotational latency
- Transfer time: transfer a 4KB page
  - SATA: 300-600MB/s => ~4 x10^3 b / (400 x 10^6 bps) => 10 us
- Latency = Queuing Time + Controller time + Xfer Time
- Highest Bandwidth: Sequential OR Random reads

Recap: SSD Architecture – Writes

- Writing data is complex! (~200µs – 1.7ms)
  - Can only write empty pages in a block
  - Erasing a block takes ~1.5ms
  - Controller maintains pool of empty blocks by coalescing used pages (read, erase, write), also reserves some % of capacity
- Rule of thumb: writes 10x reads, erasure 10x writes



Typical NAND Flash Pages and Blocks

https://en.wikipedia.org/wiki/Solid-state_drive

Recap: Solution – Two Systems Principles

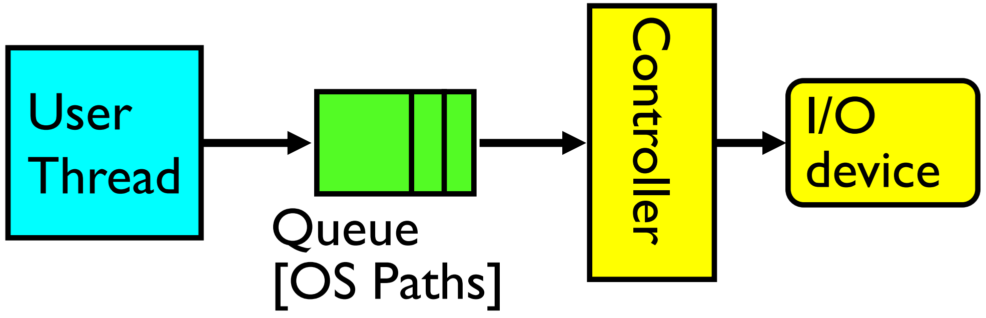
1. Layer of Indirection
   - Maintain a *Flash Translation Layer (FTL)* in SSD
   - Map virtual block numbers (which OS uses) to physical page numbers (which flash memory controller uses)
   - **Can now freely relocate data w/o OS knowing**

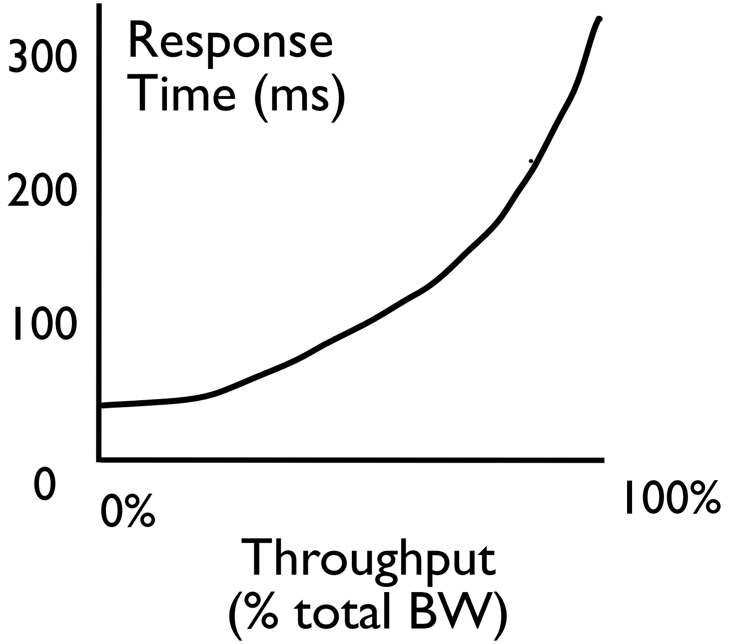
2. Copy on Write
   - Don't overwrite a page when OS updates its data (this is slow as we need to erase page first!)
   - Instead, write new version in a free page
   - Update FTL mapping to point to new location

Basic Performance Concepts

- *Response Time* or *Latency*: Time to perform an operation
- *Bandwidth* or *Throughput*: Rate at which operations are performed
  - Operations: op/s, Files: MB/s, Networks: Mb/s, Arithmetic: GFLOP/s

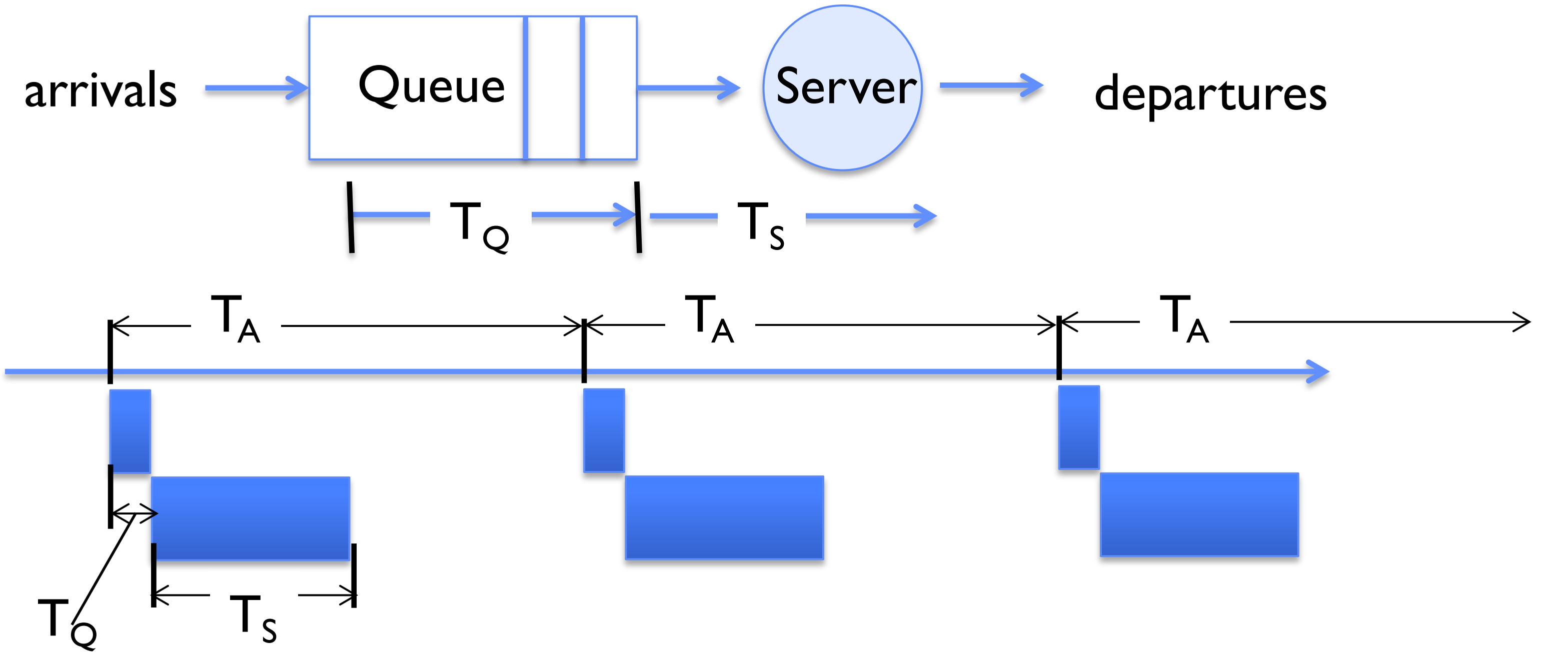
I/O Performance





- Performance of I/O subsystem
  - Metrics: Response Time, Throughput
  - Contributing factors to latency:
    - » Software paths (can be loosely modeled by a queue)
    - » Hardware controller
    - » I/O device service time
- Queuing behavior:
  - Can lead to big increases of latency as utilization increases
  - Solutions?

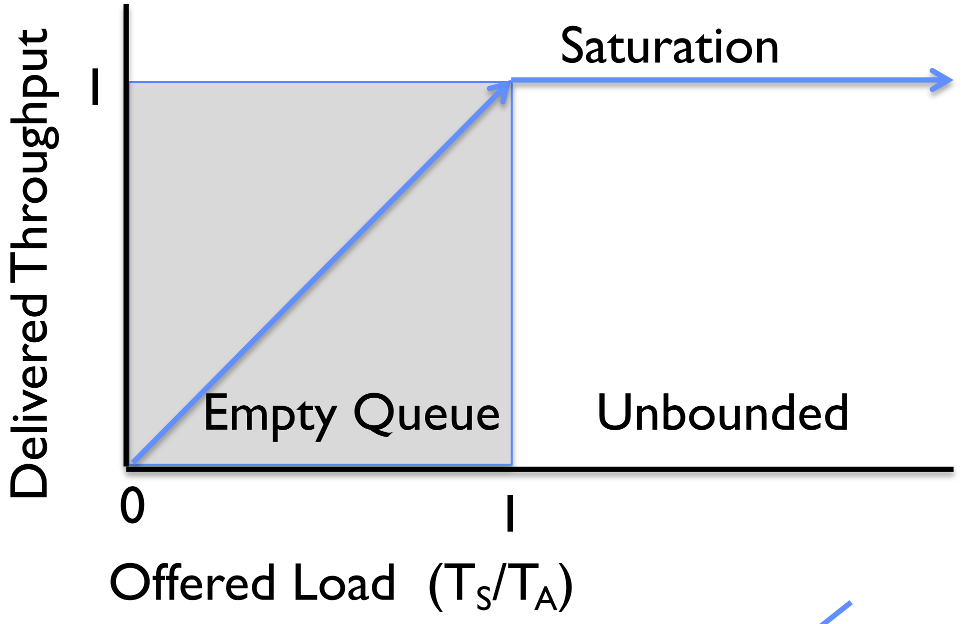
A Simple Deterministic World

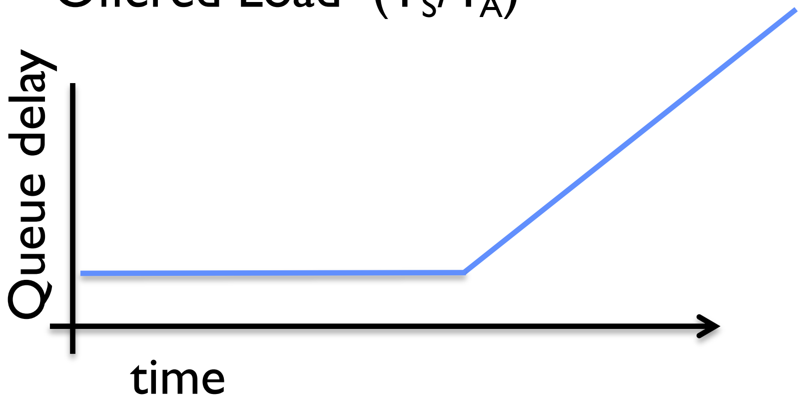


T_A = Interarrival time
T_S = Service time
T_Q = Queueing delay

- Assume requests arrive at regular intervals, take a fixed time to process, with plenty of time between …
- Service rate ($\mu = 1/T_S$) - operations per second
- Arrival rate: ($\lambda = 1/T_A$) - operations per second
- Utilization: $U = \lambda/\mu = T_S/T_A$, where $\lambda < \mu$

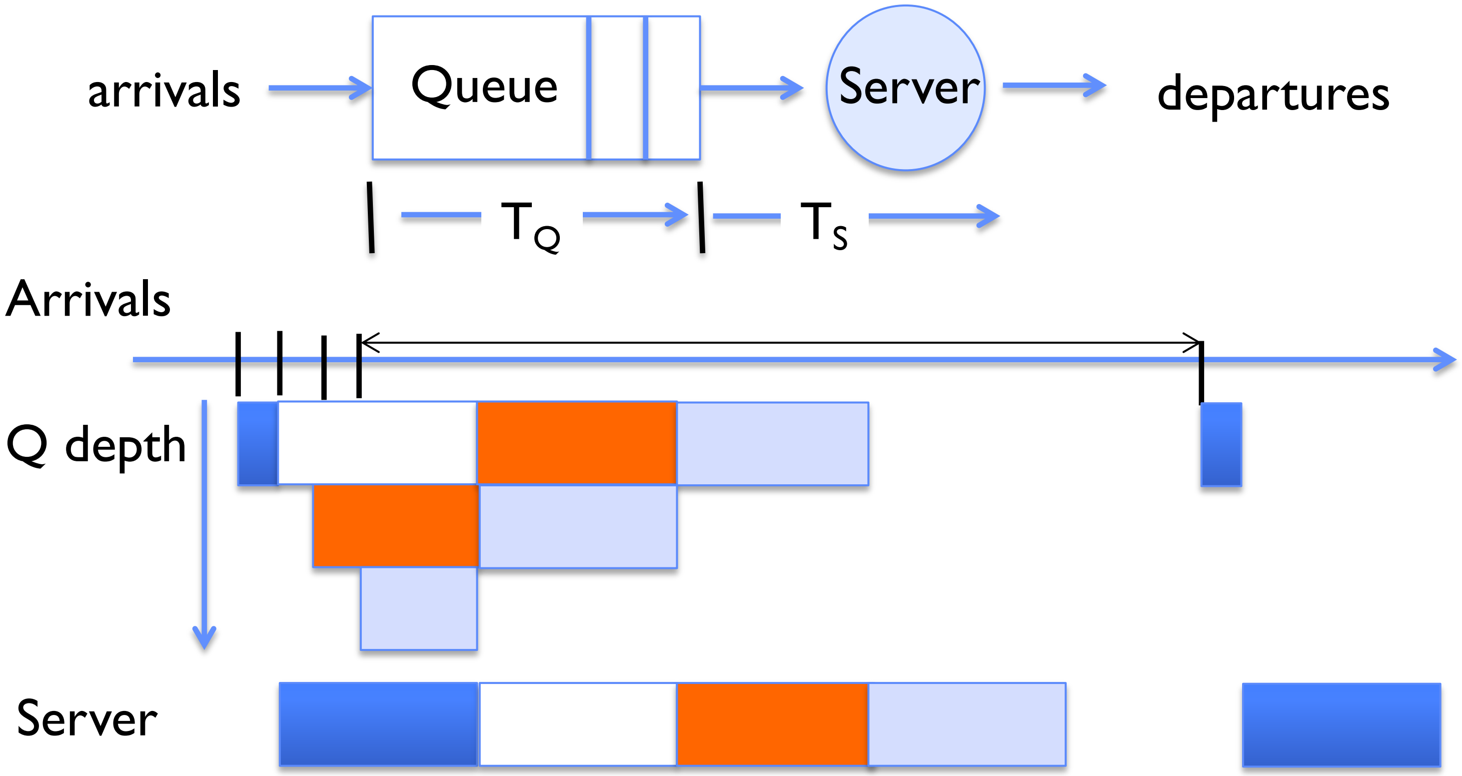
An Ideal Linear World





- What does the queue wait time look like?
  - Grows unbounded

A Bursty World



- Requests arrive in a burst, must queue up till served
- Same average arrival time, but almost all of the requests experience large queue delays
- Even though average utilization is low

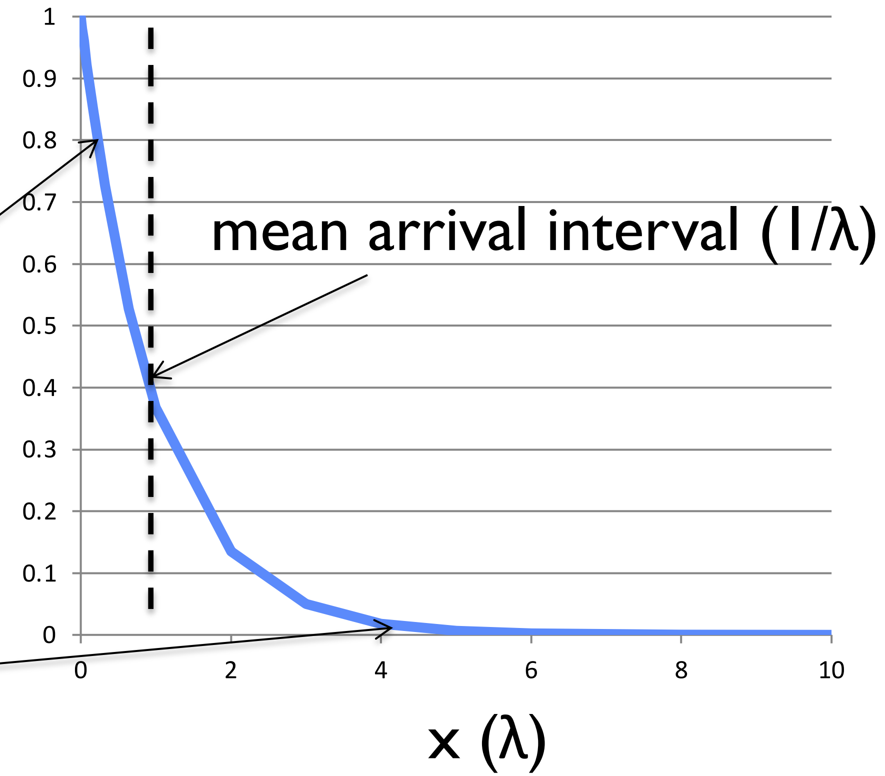
So how do we model the burstiness of arrival?

- Elegant mathematical framework if you start with *exponential distribution*
  - Probability density function of a continuous random variable with a mean of 1/λ
  - $f(x) = \lambda e^{-\lambda x}$
  - *"Memoryless"*

Likelihood of an event occurring is independent of how long we've been waiting

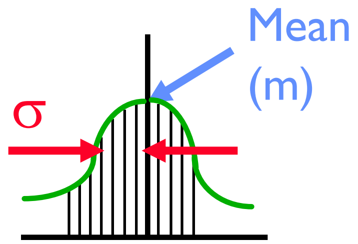
Lots of short arrival intervals (i.e., high instantaneous rate)

Few long gaps (i.e., low instantaneous rate)



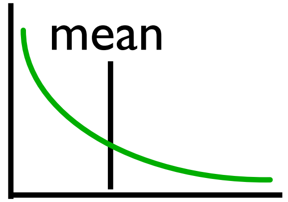
Background: General Use of Random Distributions

- Server spends variable time (T) with customers
  - Mean (Average) $m = \Sigma p(T) \times T$
  - Variance (stddev²) $\sigma^2 = \Sigma p(T) \times (T-m)^2 = \Sigma p(T) \times T^2 - m^2$
  - Squared coefficient of variance: $C = \sigma^2/m^2$
    Aggregate description of the distribution



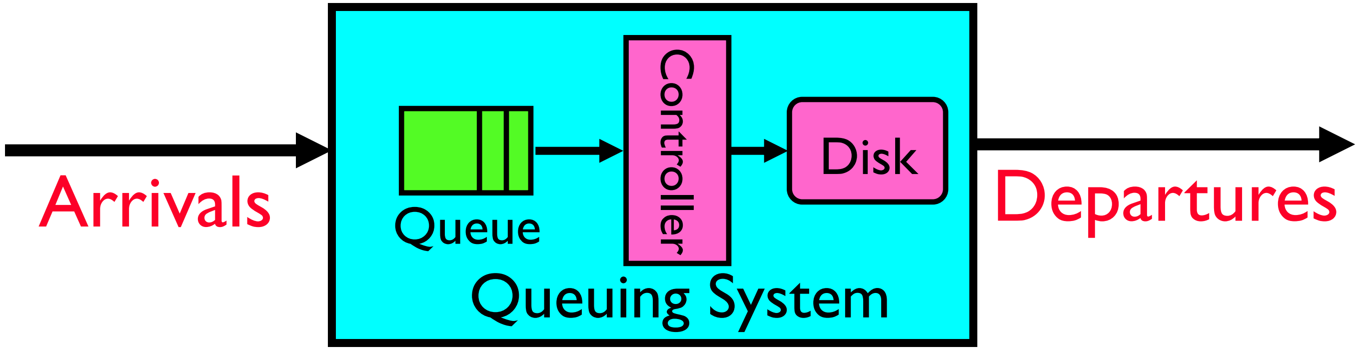
Distribution
of service times

- Important values of C:
  - No variance or deterministic ⇒ C=0
  - "Memoryless" or exponential ⇒ C=1
    - » Past tells nothing about future
    - » Many complex systems (or aggregates) are well described as memoryless
  - Disk response times $C \approx 1.5$ (majority seeks < average)



Memoryless

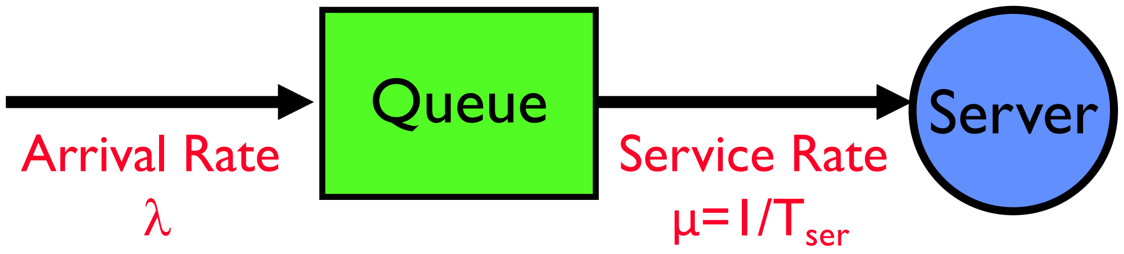
Introduction to Queuing Theory



- What about queuing time??
  - Let's apply some queuing theory
  - Queuing Theory applies to long term, steady state behavior $\Rightarrow$ Arrival rate = Departure rate
- Arrivals characterized by some probabilistic distribution
- Departures characterized by some probabilistic distribution

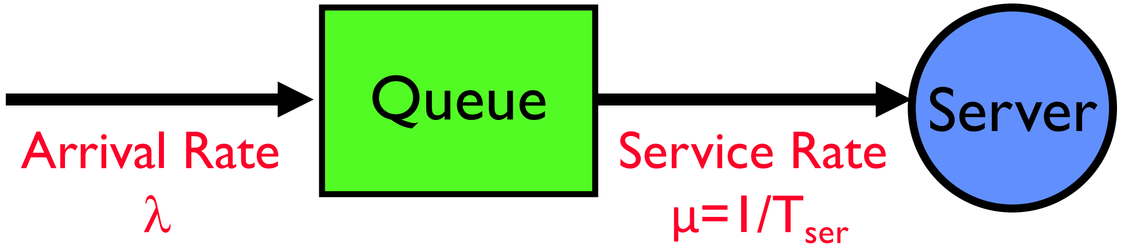
A Little Queuing Theory: Some Results (1/2)

- Assumptions:
  - System in equilibrium; No limit to the queue
  - Time between successive arrivals is random and memoryless



- Parameters that describe our system:
  - λ: mean number of arriving customers/second
  - T_{ser}: mean time to service a customer ("m")
  - C: squared coefficient of variance = σ^2/m^2
  - μ: service rate = $1/T_{ser}$
  - u: server utilization ($0 \leq u \leq 1$): $u = \lambda/\mu = \lambda \times T_{ser}$
- Parameters we wish to compute:
  - T_q: Time spent in the queue
  - L_q: Length of queue = $\lambda \times T_q$ (by Little's law)

A Little Queuing Theory: Some Results (2/2)



- Parameters that describe our system:
  - λ: mean number of arriving customers/second $\lambda = 1/T_A$
  - T_{ser}: mean time to service a customer ("m")
  - C: squared coefficient of variance = σ^2/m^2
  - μ: service rate = $1/T_{ser}$
  - u: server utilization ($0 \le u \le 1$): $u = \lambda/\mu = \lambda \times T_{ser}$
- Parameters we wish to compute:
  - T_q: Time spent in the queue
  - L_q: Length of queue = $\lambda \times T_q$ (by Little's law)
- Results (M: Poisson arrival process, 1 server):
  - Memoryless service time distribution (C = 1): Called an M/M/1 queue
    - » $T_q = T_{ser} \times u/(1 - u)$
  - General service time distribution (no restrictions): Called an M/G/1 queue
    - » $T_q = T_{ser} \times \frac{1}{2}(1+C) \times u/(1 - u)$

A Little Queuing Theory: An Example (1/2)

- Example Usage Statistics:
  - User requests 10 x 8KB disk I/Os per second
  - Requests & service exponentially distributed (C=1.0)
  - Avg. service = 20 ms (From controller + seek + rotation + transfer)
- Questions:
  - How utilized is the disk (server utilization)? Ans:, $u = \lambda T_{ser}$
  - What is the average time spent in the queue? Ans: T_q
  - What is the number of requests in the queue? Ans: L_q
  - What is the avg response time for disk request? Ans: $T_{sys} = T_q + T_{ser}$

A Little Queuing Theory: An Example (2/2)

- Questions:
  - How utilized is the disk (server utilization)? Ans:, $u = \lambda T_{ser}$
  - What is the average time spent in the queue? Ans: T_q
  - What is the number of requests in the queue? Ans: L_q
  - What is the avg response time for disk request? Ans: $T_{sys} = T_q + T_{ser}$

- Computation:

λ (avg # arriving customers/s) = 10/s

T_{ser} (avg time to service customer) = 20 ms (0.02s)

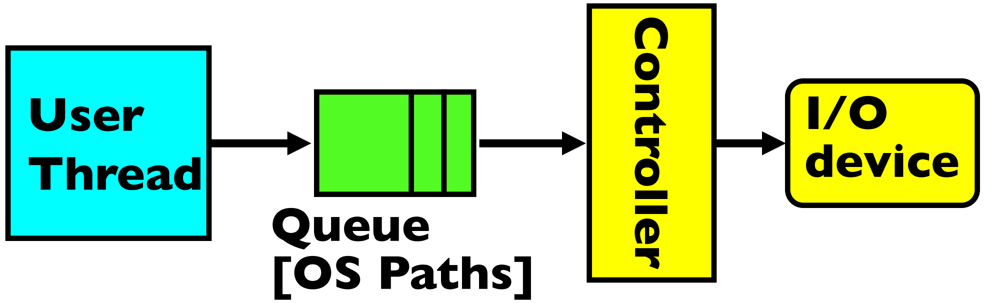
u (server utilization) = $\lambda \times T_{ser}$ = 10/s x .02s = 0.2

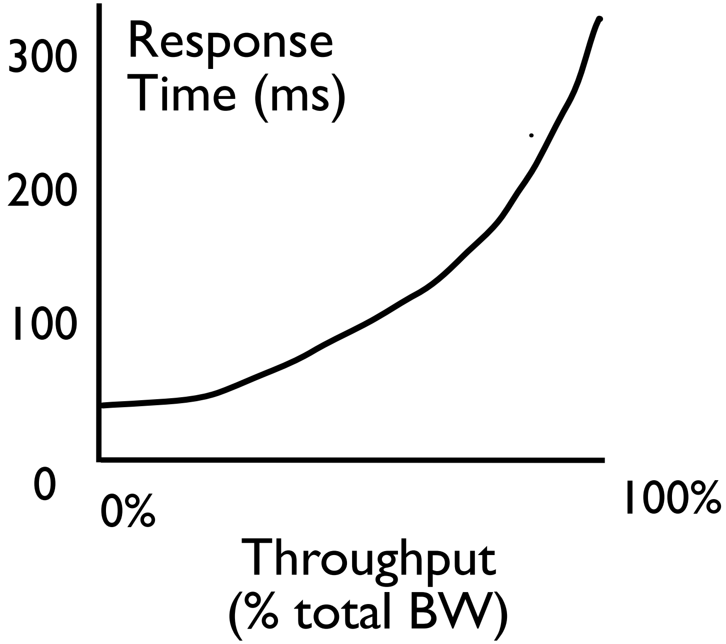
T_q (avg time spent in queue) = $T_{ser} \times u/(1 - u)$
= 20 x 0.2/(1-0.2) = 20 x 0.25 = 5 ms (0 .005s)

L_q (avg length of queue) = $\lambda \times T_q$=10/s x .005s = 0.05

T_{sys} (avg response time for disk request) =$T_q + T_{ser}$= 25 ms

Group Discussion: Optimize I/O Performance

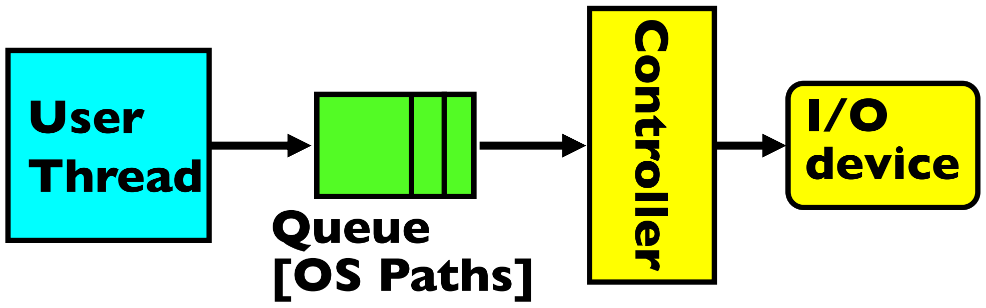


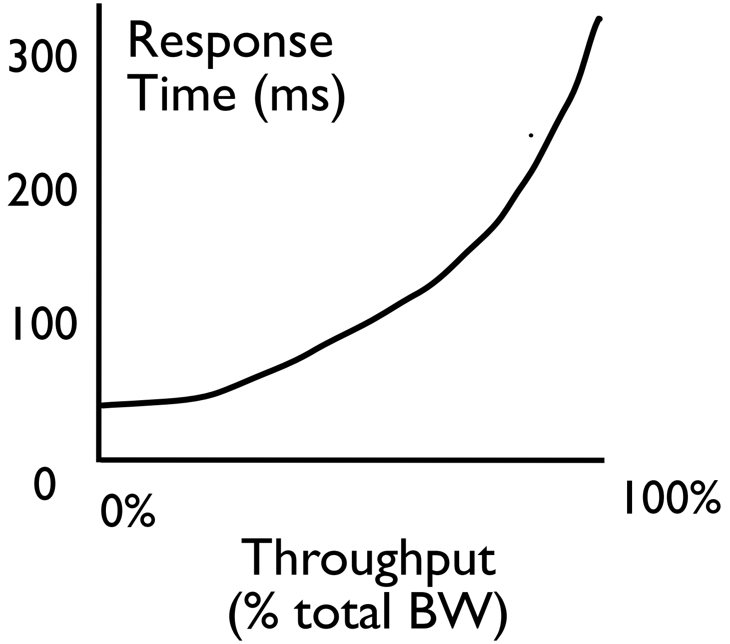


- How to improve performance?

- Discuss in groups of two to three students
  - Each group chooses a leader to summarize the discussion
  - In your group discussion, please do not dominate the discussion, and give everyone a chance to speak

Optimize I/O Performance





- How to improve performance?
  - Speed: make everything faster ☺
  - Parallelism: More Decoupled systems
    - » multiple independent buses or controllers
  - Overlap: do other useful work while waiting
  - Optimize the bottleneck to increase service rate
    - » Use the queue to optimize the performance
- Queues absorb bursts and smooth the flow
- Admissions control (finite queues)
  - Limits delays, but may introduce unfairness and livelock

When is Disk Performance Highest?

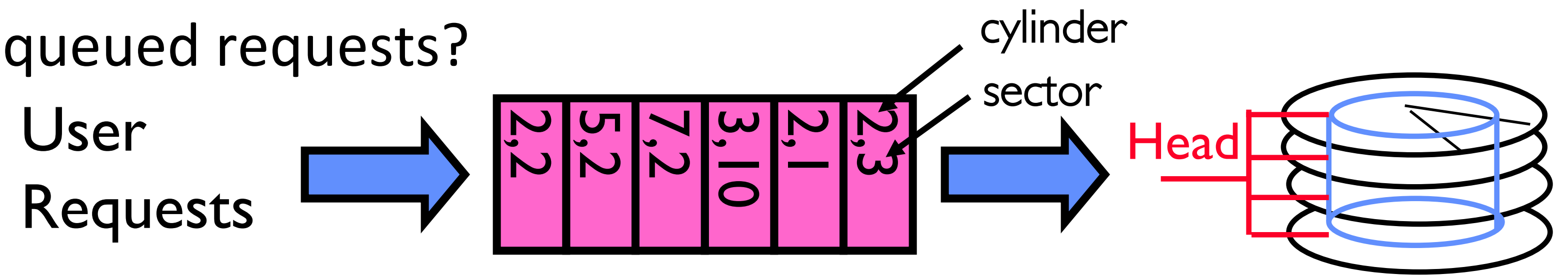
- When there are big sequential reads, or ….
- … when there is so much work to do so that they can be piggybacked (reordering queues—one moment)

- OK to be inefficient when things are mostly idle
- Bursts are both a threat and an opportunity
  - Treat: they can increase latency
  - Opportunity: enable piggyback (e.g., reordering of requests) & batching (e.g., one context switch to handle multiple requests)

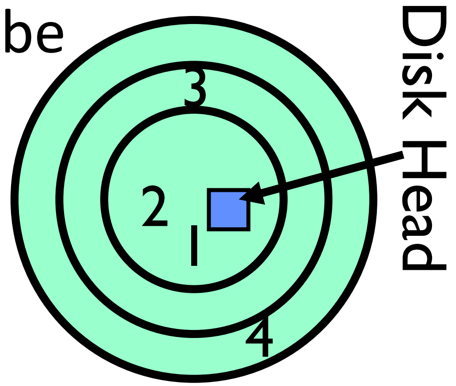
- Other techniques:
  - Reduce overhead through user level drivers (e.g., avoid context switching)
  - Reduce the impact of I/O delays by doing other useful work in the meantime

Disk Scheduling (1/3)

- Disk can do only one request at a time; What order do you choose to do queued requests?

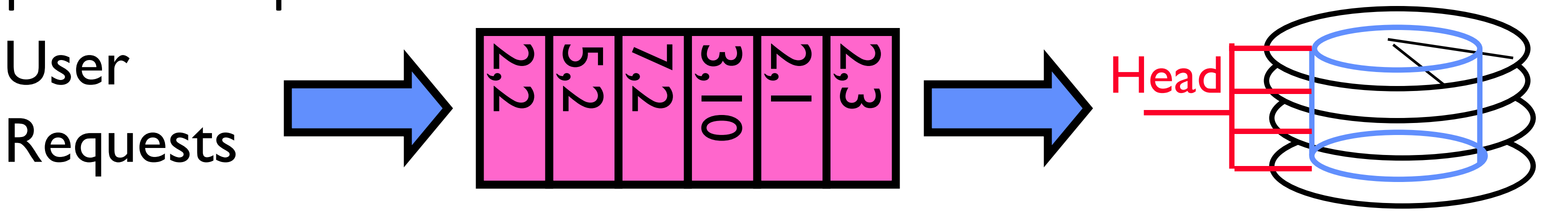


- FIFO Order
  - Fair among requesters, but order of arrival may be to random spots on the disk ⇒ Very long seeks
- SSTF: Shortest seek time first
  - Pick the request that's closest on the disk
  - Although called SSTF, today must include rotational delay in calculation, since rotation can be as long as seek
  - Con: SSTF good at reducing seeks, but may lead to starvation

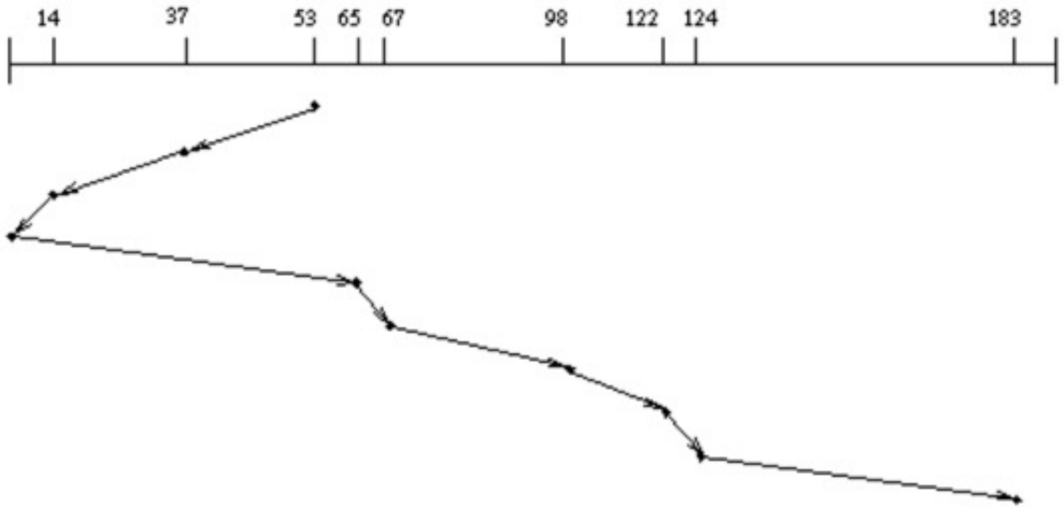


Disk Scheduling (2/3)

- Disk can do only one request at a time; What order do you choose to do queued requests?

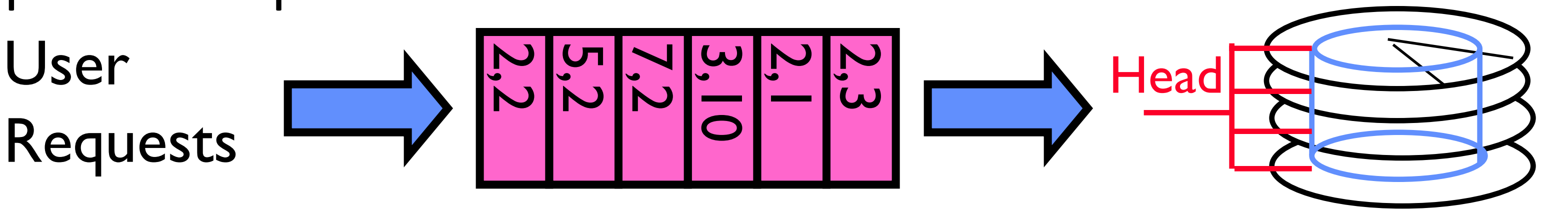


- SCAN: Implements an Elevator Algorithm: take the closest request in the direction of travel
  - No starvation, but retains flavor of SSTF

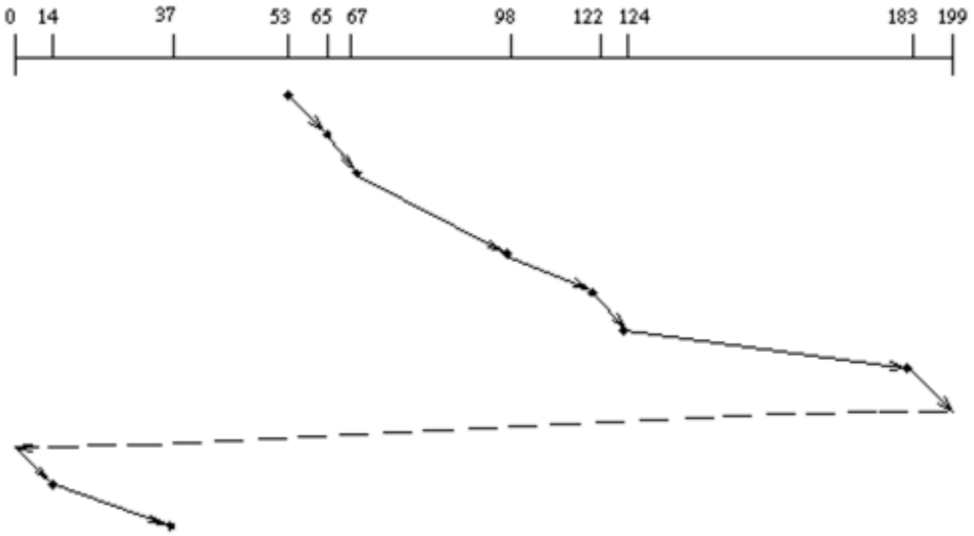


Disk Scheduling (3/3)

- Disk can do only one request at a time; What order do you choose to do queued requests?



- C-SCAN: Circular-Scan: only goes in one direction
  - Skips any requests on the way back
  - Fairer than SCAN, not biased towards pages in middle



Network IO

- Packets in network IO vs. blocks in disk IO, but the general principles apply

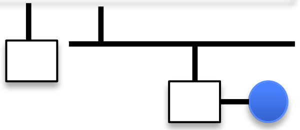
- Network IO is critical in modern cloud systems
  - Applications/systems are networked/distributed
  - Accessing to storage is via network IO!
    - » It is a common approach today to organize storage devices as a storage pool
    - » Accessing the storage pool via the datacenter network from compute nodes

- Approaches to improve network IO performance
  - Better abstractions for distributed applications, e.g., coflow
  - Optimize TCP/IP stack in the kernel
  - Kernel-bypass
    - » User-space network stack
    - » Offload to the NIC, e.g., RDMA, SmartNICs, and DPUs

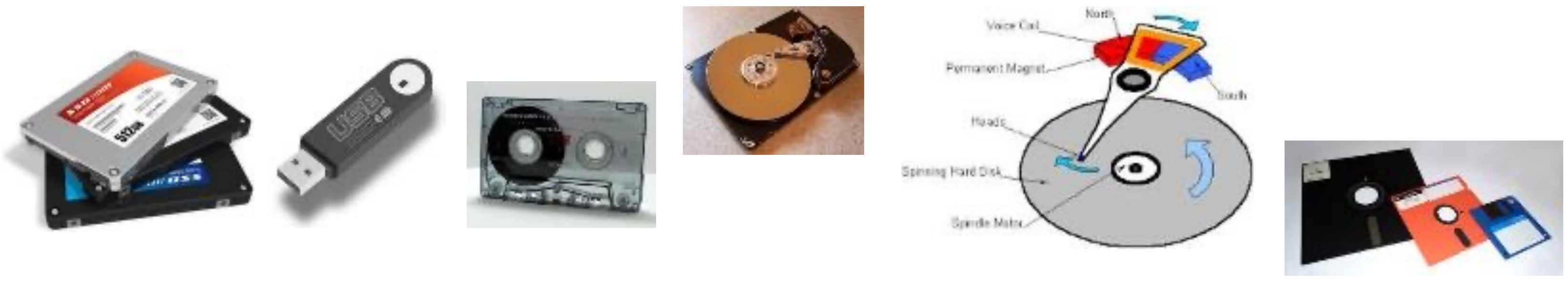
Recall: I/O and Storage Layers

Application / Service

High Level I/O	*Streams*
Low Level I/O	*File Descriptors*
Syscall	*open(), read(), write(), close(), …* *Open File Descriptions*
File System	*Files/Directories/Indexes*
I/O Driver	*Commands and Data Transfers*



Disks, Flash, Controllers, DMA

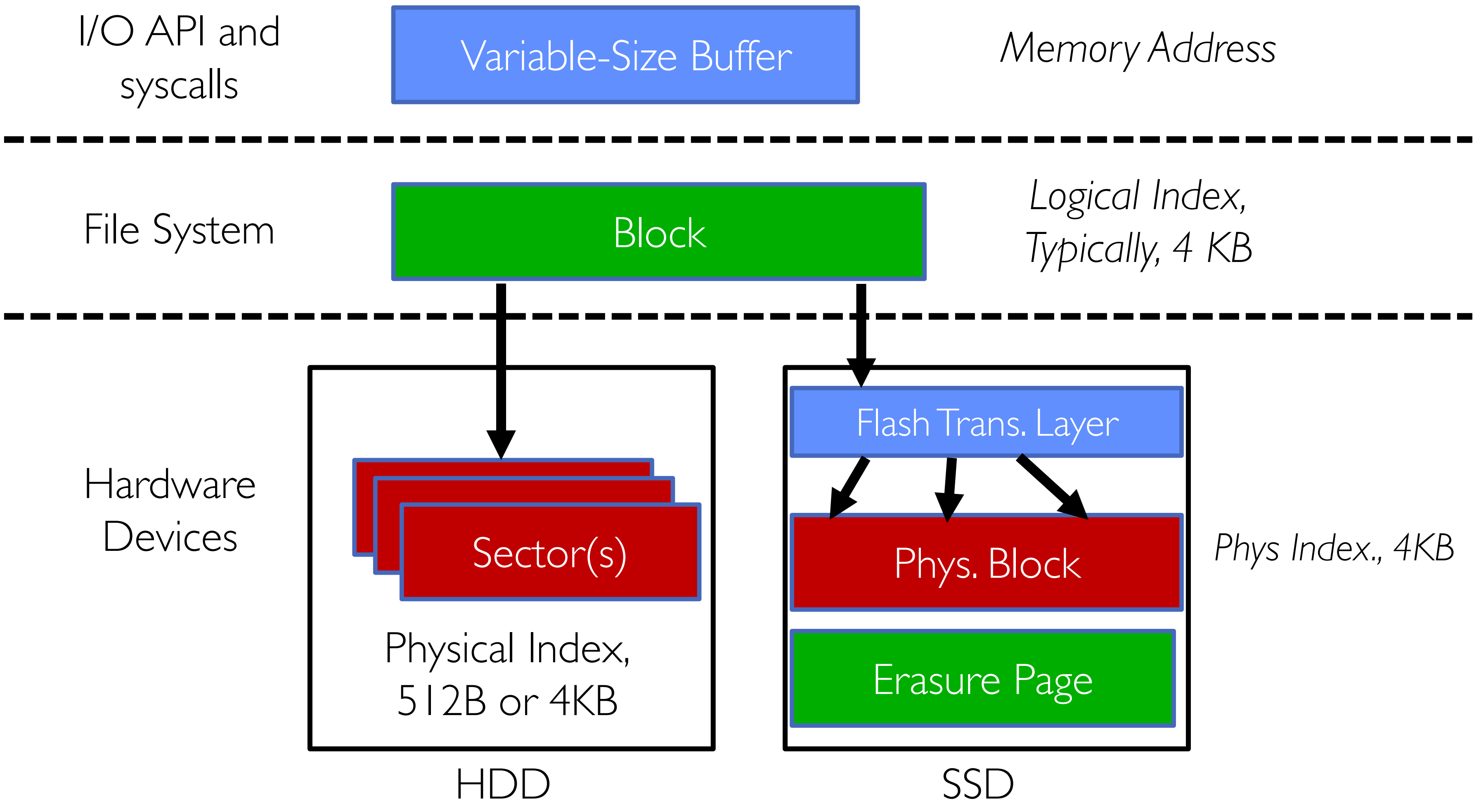


What we covered earlier

What we will cover next…

What we just covered…

From Storage to File Systems



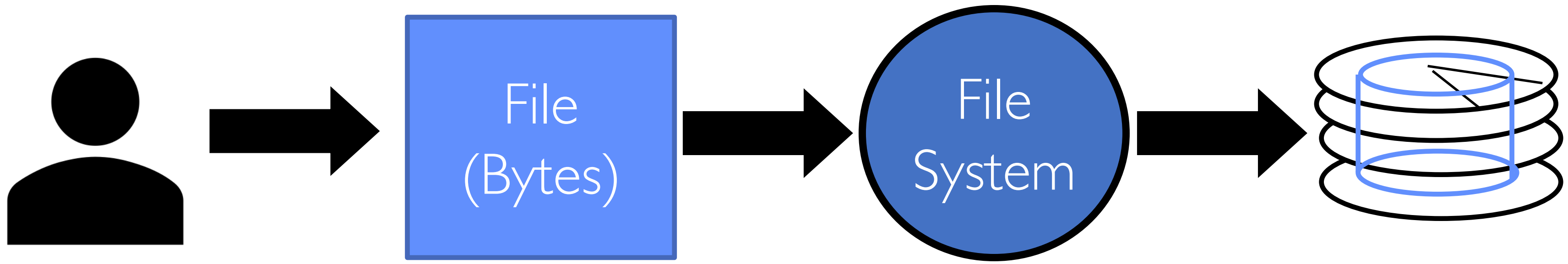
Building a File System

- File System: Layer of OS that transforms block interface of disks (or other block devices) into Files, Directories, etc.
- Classic OS situation: Take limited hardware interface (array of blocks) and provide a more convenient/useful interface with:
  - Naming: Find file by name, not block numbers
  - Organization:
    - File names in directories
    - Map files to blocks
  - Protection: Enforce access restrictions
  - Reliability: Keep files intact despite crashes, hardware failures, etc.

User vs. System View of a File

- User's view:
  - Durable Data Structures
- System's view (system call interface):
  - Collection of Bytes (UNIX)
  - Doesn't matter to system what kind of data structures you want to store on disk!
- System's view (inside OS):
  - Collection of blocks (a block is a logical transfer unit, while a sector is the physical transfer unit)
  - Block size $\geq$ sector size; in UNIX, block size is 4KB

Translation from User to System View



- What happens if user says: "give me bytes 2 – 12?"
  - Fetch block corresponding to those bytes
  - Return just the correct portion of the block
- What about writing bytes 2 – 12?
  - Fetch block, modify relevant portion, write out block
- Everything inside file system is in terms of whole-size blocks
  - Actual disk I/O happens in blocks
  - read/write smaller than block size needs to translate and buffer

Disk Management

- Basic entities on a disk:
  - File: user-visible group of blocks arranged sequentially in logical space
  - Directory: user-visible index mapping names to files
- The disk is accessed as linear array of sectors
- How to identify a sector?
  - Physical position
    - » Sectors is a vector [cylinder, surface, sector]
    - » Not used any more
    - » OS/BIOS must deal with bad sectors
  - Logical Block Addressing (LBA)
    - » Every sector has integer address
    - » Controller translates from address $\Rightarrow$ physical position
    - » Shields OS from structure of disk

What Does the File System Need?

- Track which blocks contain data for which files
  - Need to know where to read a file from
- Track files in a directory
  - Find list of file's blocks given its name
- Track free disk blocks
  - Need to know where to put newly written data
- Where do we maintain all of this?
  - **Somewhere on disk**

Data Structures on Disk

- Data structure on disk different than data structures in memory

- Access a block at a time
  - Can't efficiently read/write a single word
  - Have to read/write full block containing it
  - Ideally want sequential access patterns

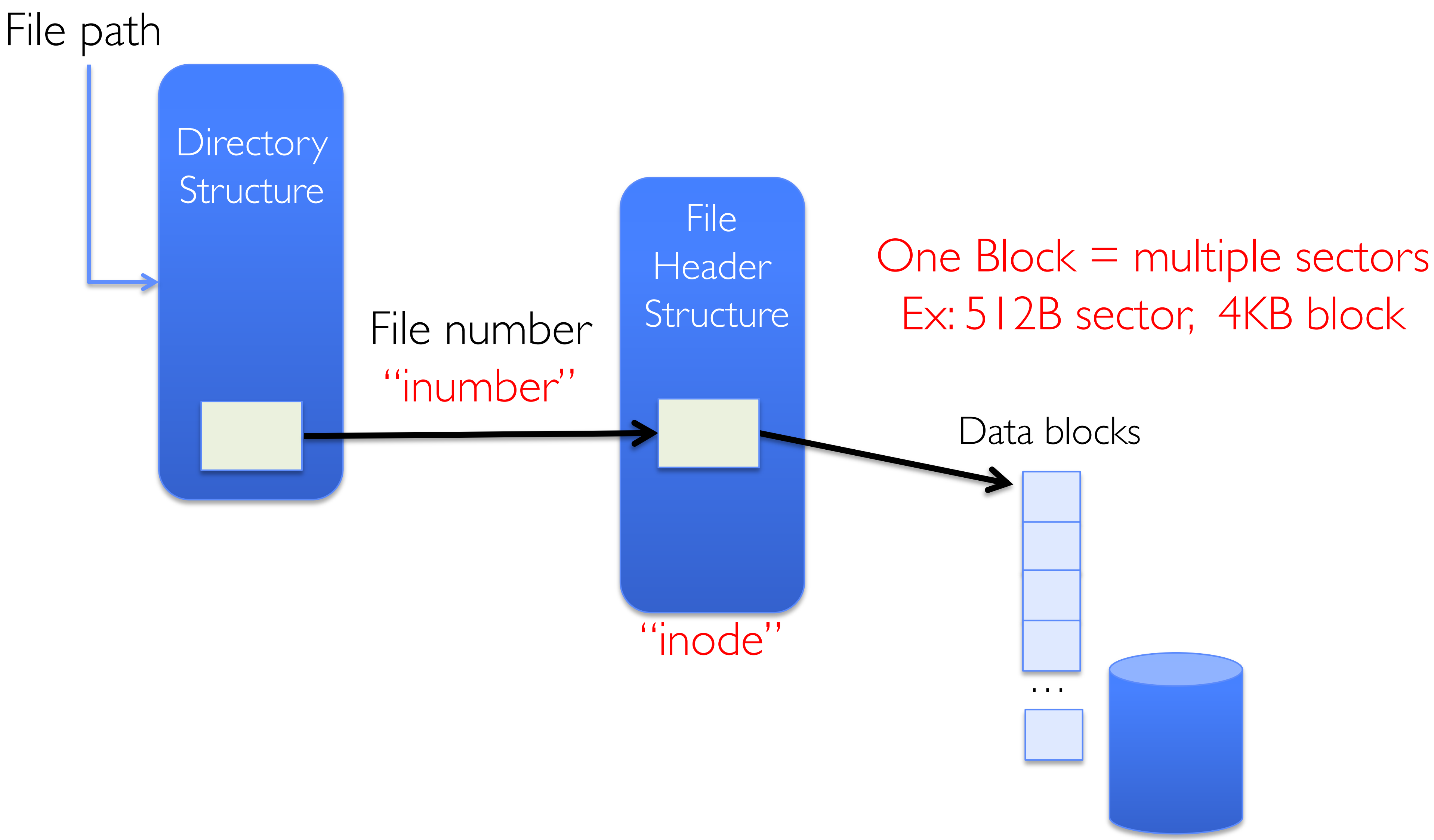
- Durability
  - Ideally, file system is in meaningful state upon shutdown
  - This obviously isn't always the case…

FILE SYSTEM DESIGN

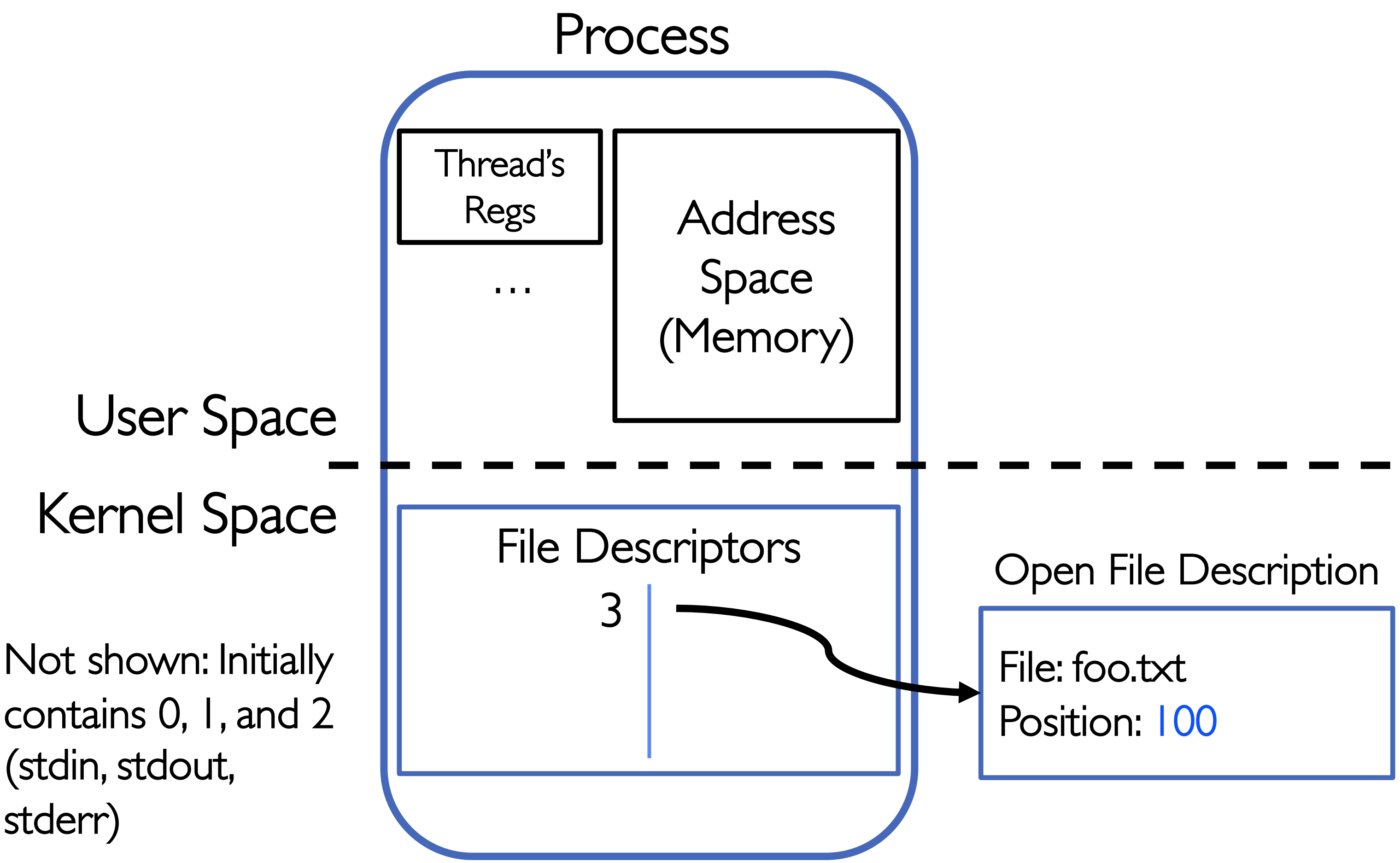
Critical Factors in File System Design

- (Hard) Disk Performance !!!
  - Maximize sequential access, minimize seeks
- Open before Read/Write
  - Can perform protection checks and look up where the actual file resource are, in advance
- Size is determined as files are used !!!
  - Can write to expand the file
  - Start small and grow, need to make room
- Organized into directories
  - What data structure (on disk) for that?
- Need to carefully allocate / free blocks
  - Such that access remains efficient

Components of a File System



Recall: Abstract Representation of a Process



Not shown: Initially contains 0, 1, and 2 (stdin, stdout, stderr)

Suppose that we execute

open("foo.txt")

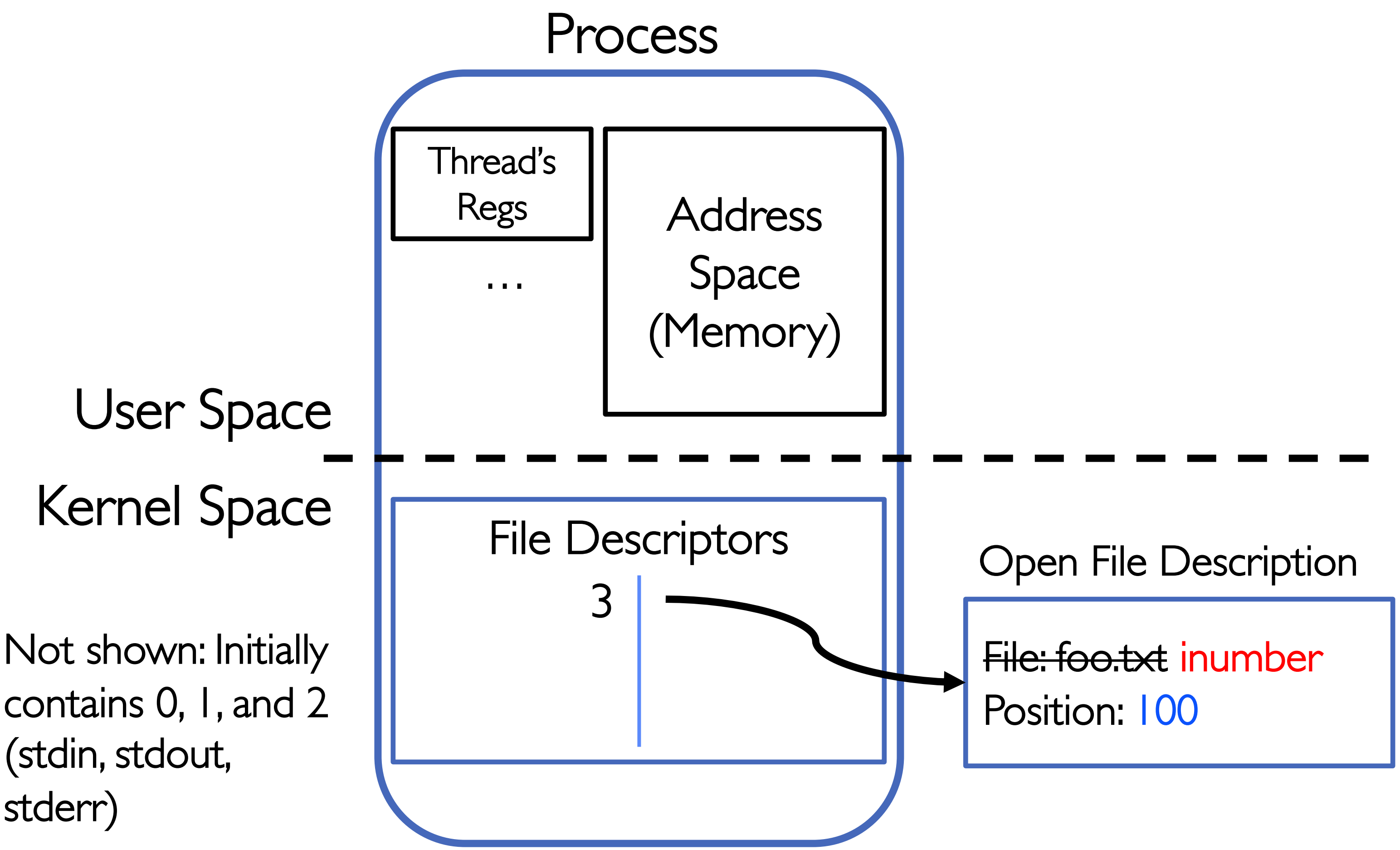
and that the result is 3

Next, suppose that we execute

read(3, buf, 100)

and that the result is 100

Components of a File System



Not shown: Initially contains 0, 1, and 2 (stdin, stdout, stderr)

Open file description is better described as remembering the inumber (file number) of the file, not its name

Components of a File System

file name
offset → directory structure → *file number*
offset → index structure (“inode”) → *storage block*

- Open performs *Name Resolution*
  - Translates path name into a “file number”
- Read and Write operate on the file number
  - Use file number as an “index” to locate the blocks

- 4 components:
  - directory, index structure, storage blocks, free space map

How to get the File Number?

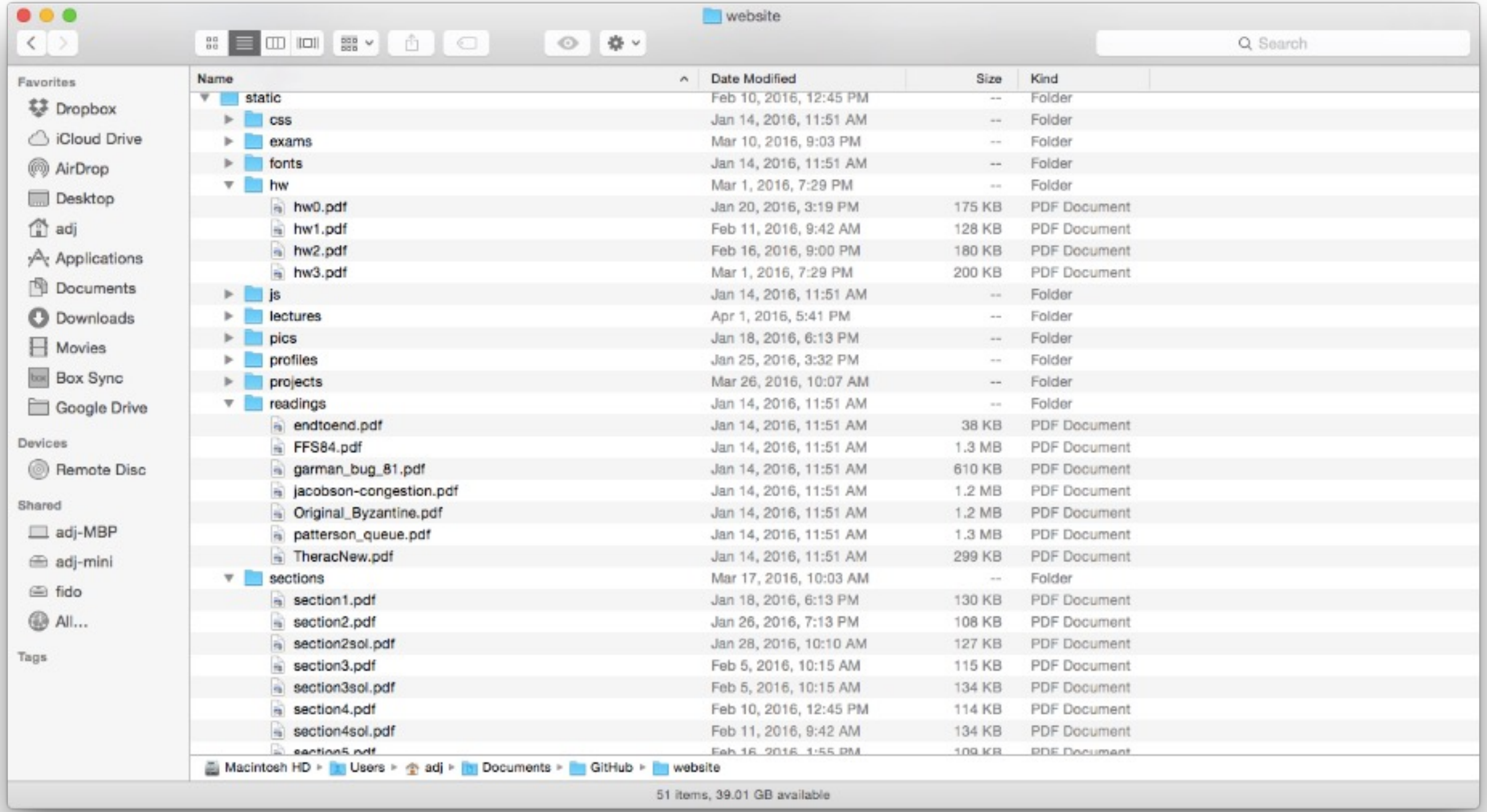
- Look up in ***directory structure***

- A directory is a file containing <file_name : file_number> mappings
  - File number could be a file or another directory
  - Operating system stores the mapping in the directory in a format it interprets
  - Each <file_name : file_number> mapping is called a **directory entry**

- Process isn't allowed to read the raw bytes of a directory
  - The `read` function doesn't work on a directory
  - Instead, see `readdir`, which iterates over the map without revealing the raw bytes

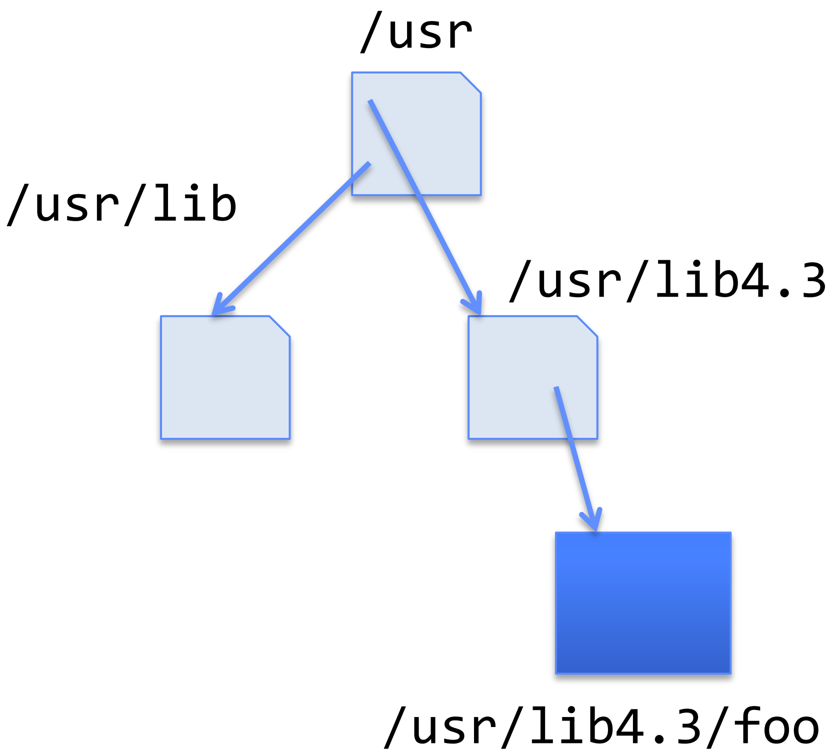
- Why shouldn't the OS let processes read/write the bytes of a directory?

Directories



Directory Abstraction

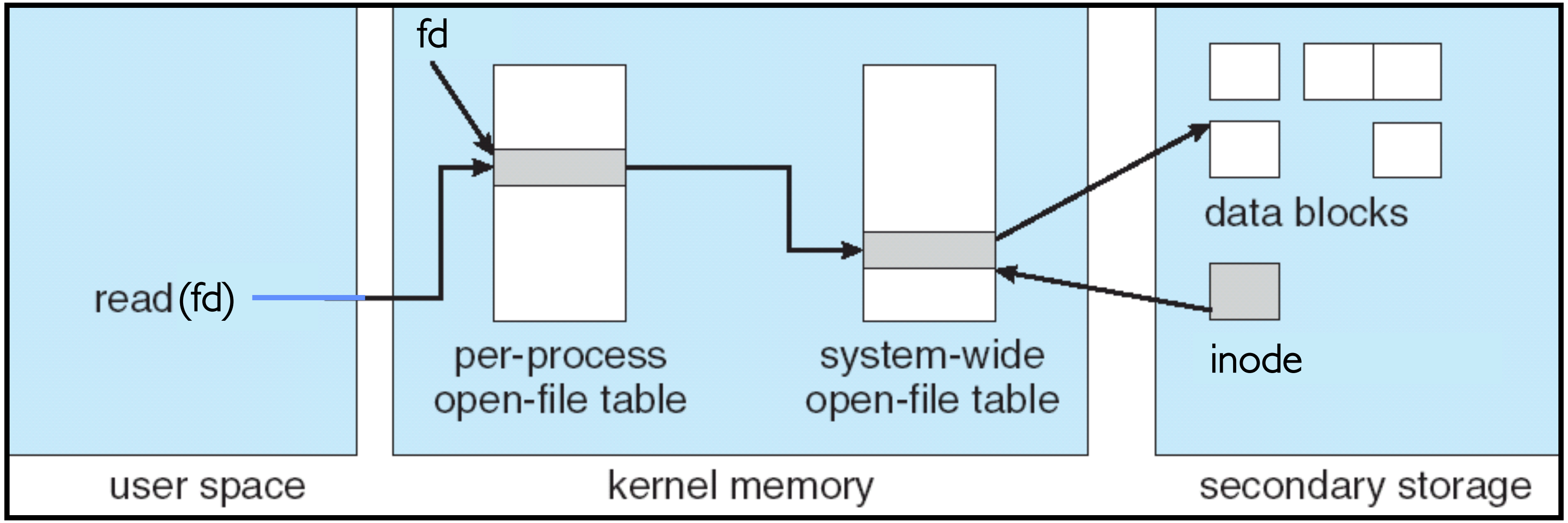
- Directories are specialized files
  - Contents: **List of pairs**
    <file name, file number>
- System calls to access directories
  - `open` / `creat` / `readdir` traverse the structure
  - `mkdir` / `rmdir` add/remove entries
  - `link` / `unlink`



Directory Structure

- How many disk accesses to resolve “/my/book/count”?
  - Read in file header for root (fixed position on disk)
  - Read in first data block for root
    - » Table of file name/index pairs.
    - » Search linearly – ok since directories typically very small
  - Read in file header for “my”
  - Read in first data block for “my”; search for “book”
  - Read in file header for “book”
  - Read in first data block for “book”; search for “count”
  - Read in file header for “count”
- Current working directory: Per-address-space pointer to a directory used for resolving file names
  - Allows user to specify relative filename instead of absolute path (say CWD=“/my/book” can resolve “count”)

In-Memory File System Structures



- Open syscall: find inode on disk from pathname (traversing directories)
  - Create "in-memory inode" in system-wide open file table
  - One entry in this table no matter how many instances of the file are open
- Read/write syscalls look up in-memory inode using the file handle

Characteristics of Files

A Five-Year Study of File-System Metadata

NITIN AGRAWAL
University of Wisconsin, Madison
and
WILLIAM J. BOLOSKY, JOHN R. DOUCEUR, and JACOB R. LORCH
Microsoft Research

Published in FAST 2007

annual snapshots of file-system metadata from over 60,000 Windows PC file systems in a large corporation

Observation #1: Most Files Are Small

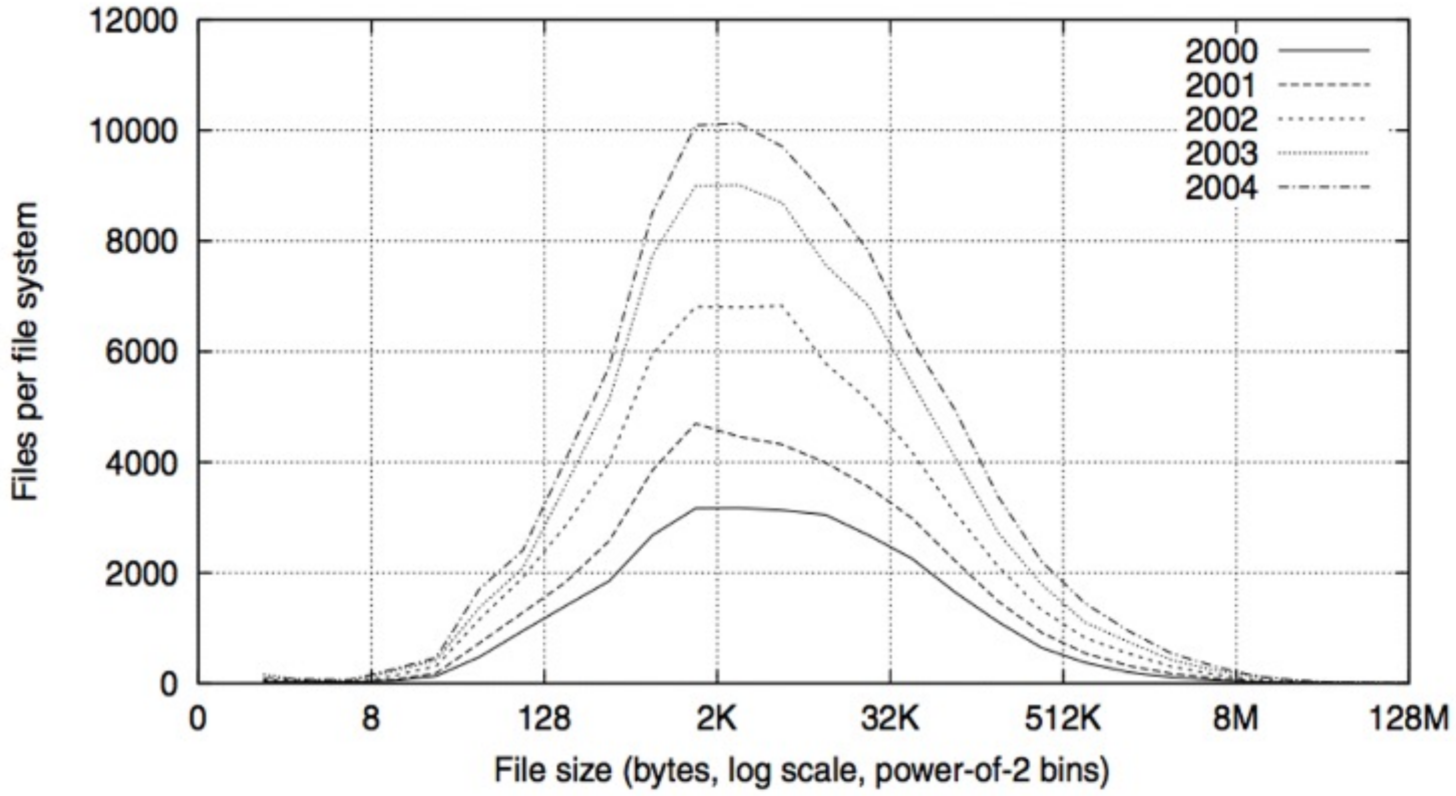


Fig. 2. Histograms of files by size.

Observation #2: Most Bytes are in Large Files

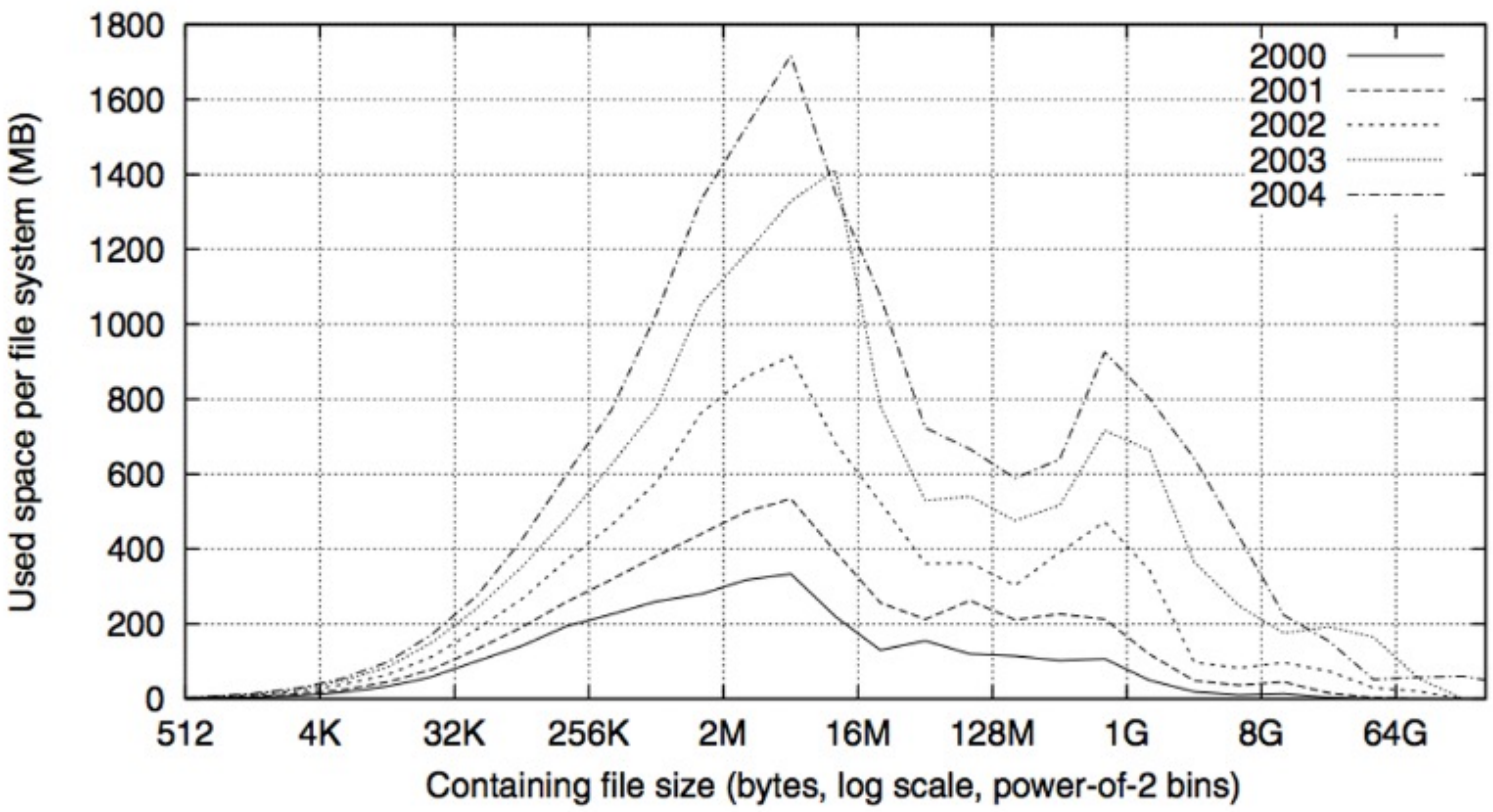


Fig. 4. Histograms of bytes by containing file size.

Conclusion

- Systems (e.g., file system) designed to optimize performance and reliability
  - Relative to performance characteristics of underlying device
- Bursts & High Utilization introduce queuing delays
- Queuing Latency:
  - M/M/1 and M/G/1 queues: simplest to analyze
  - As utilization approaches 100%, latency $\rightarrow \infty$

$$T_q = T_{ser} \times \frac{1}{2}(1+C) \times u/(1 - u)$$

- File System:
  - Transforms blocks into Files and Directories
  - Optimize for access and usage patterns
  - Maximize sequential access, allow efficient random access
- File (and directory) defined by header, called "inode"
- Naming: translating from user-visible names to actual sys resources
  - Directories used for naming for local file systems
  - Linked or tree structure stored in files