

CS162
Operating Systems and
Systems Programming
Lecture 21

Filesystems 1: Performance,
Queueing Theory, Filesystem Design

April 14th, 2026

Prof. John Kubiatowicz

<http://cs162.eecs.Berkeley.edu>

Recall: Ways of Measuring Performance

- *Latency* – time to complete a task
 - Measured in units of time (s, ms, us, ..., hours, years)
- *Response Time* - time to initiate and operation and get its response
 - Able to issue one that *depends* on the result
 - Know that it is done (anti-dependence, resource usage)
- *Throughput* or *Bandwidth* – rate at which tasks are performed
 - Measured in units of things per unit time (ops/s, GFLOP/s)
- *Start up or “Overhead”* – time to initiate an operation
- Most I/O operations are roughly linear in b bytes
 - $\text{Latency}(b) = \text{Overhead} + b/\text{TransferCapacity}$
- Performance???
 - Operation time (4 mins to run a mile...)
 - Rate (mph, mpg, ...)

Example: Overhead in Fast Network

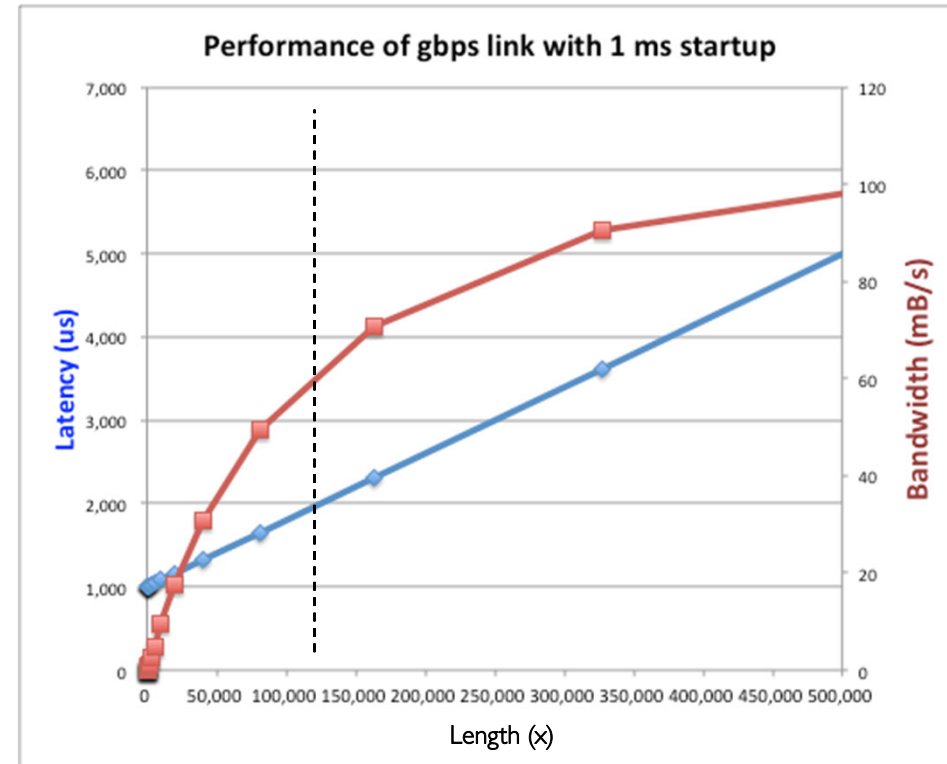
- Consider a 1 Gb/s link ($B_w = 125 \text{ MB/s}$) with startup cost $S = 1 \text{ ms}$

- Latency: $L(x) = S + \frac{x}{B_w}$

- Effective Bandwidth:

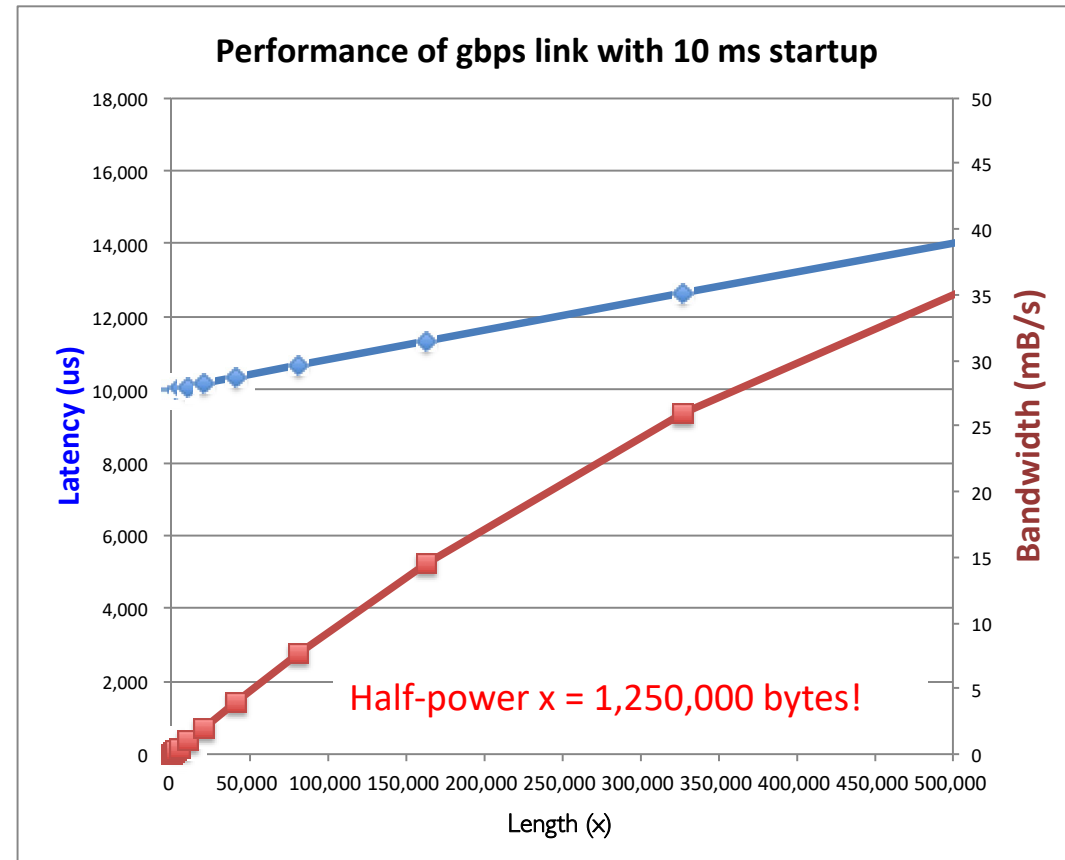
$$E(x) = \frac{x}{S + \frac{x}{B_w}} = \frac{B_w \cdot x}{B_w \cdot S + x} = \frac{B_w}{\frac{B_w \cdot S}{x} + 1}$$

- Half-power Bandwidth: $E(x) = \frac{B_w}{2}$
- For this example, half-power bandwidth occurs at $x = 125 \text{ KB}$



Example: 10 ms Startup Cost (e.g., Disk)

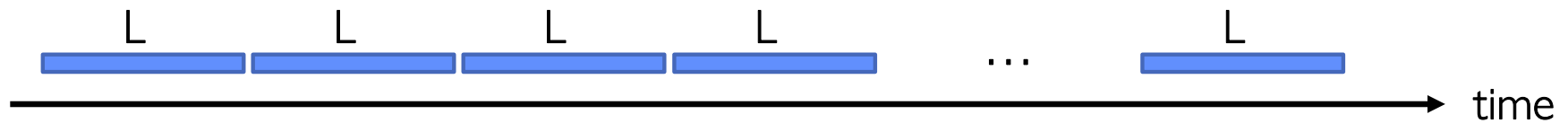
- Half-power bandwidth at $x = 1.25$ MB
- Large startup cost can degrade effective bandwidth
- Amortize it by performing I/O in larger blocks



What Determines Peak BW for I/O?

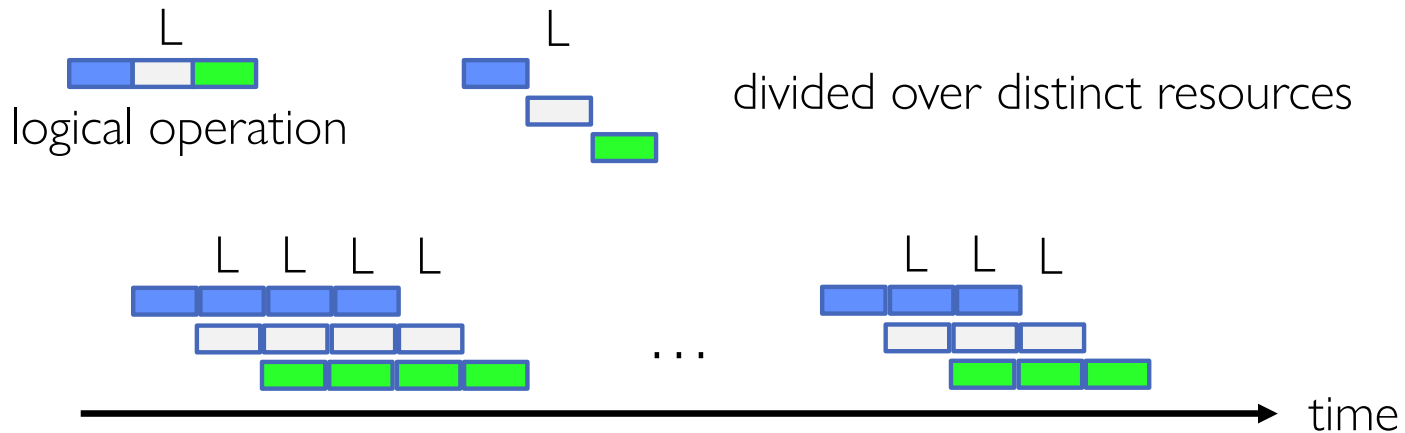
- Bus Speed
 - PCI-X: 1064 MB/s = 133 MHz x 64 bit (per lane)
 - ULTRA WIDE SCSI: 40 MB/s
 - Serial Attached SCSI & Serial ATA & IEEE 1394 (firewire): 1.6 Gb/s full duplex (200 MB/s)
 - USB 3.0 – 5 Gb/s
 - Thunderbolt 3 – 40 Gb/s
- Device Transfer Bandwidth
 - Rotational speed of disk
 - Write / Read rate of NAND flash
 - Signaling rate of network link
- Whatever is the bottleneck in the path...

Sequential Server Performance



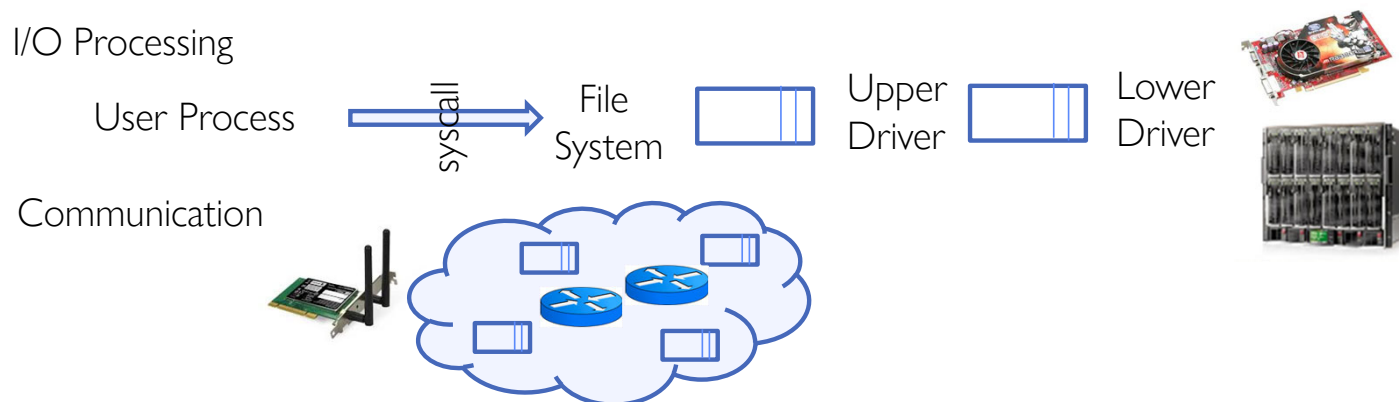
- Single sequential “server” that can deliver a task in time L operates at rate $\leq \frac{1}{L}$ (on average, in steady state, ...)
 - $L = 10 \text{ ms} \rightarrow B = 100 \text{ op/s}$
 - $L = 2 \text{ yr} \rightarrow B = 0.5 \text{ op/yr}$
- Applies to a processor, a disk drive, a person, a TA, ...

Single Pipelined Server



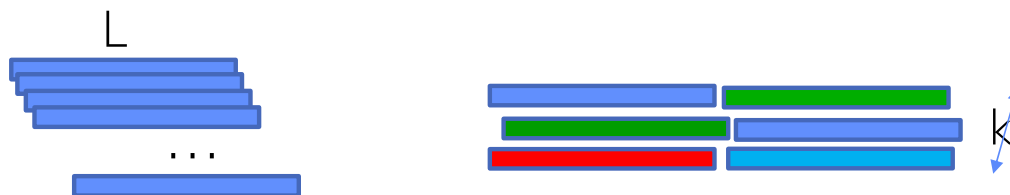
- Single pipelined server of k stages for tasks of length L (i.e., time L/k per stage) delivers at rate $\leq k/L$.
 - $L = 10$ ms, $k = 4 \rightarrow B = 400$ op/s
 - $L = 2$ yr, $k = 2 \rightarrow B = 1$ op/yr

Example Systems “Pipelines”



- Anything with queues between operational process behaves roughly “pipeline like”
- Important difference is that “initiations” are decoupled from processing
 - May have to queue up a burst of operations
 - Not synchronous and deterministic like in 61C

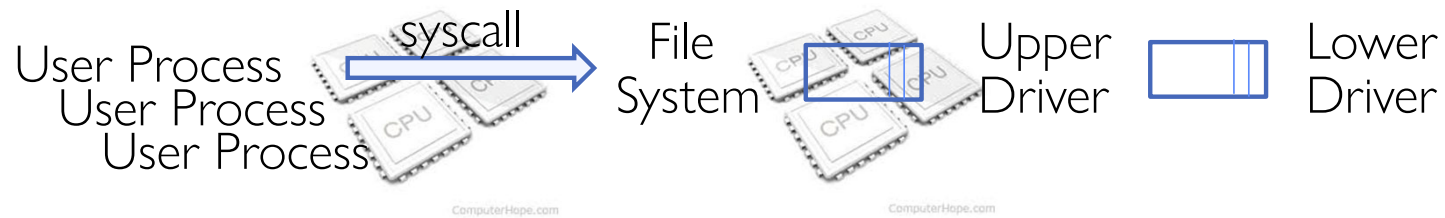
Multiple Servers



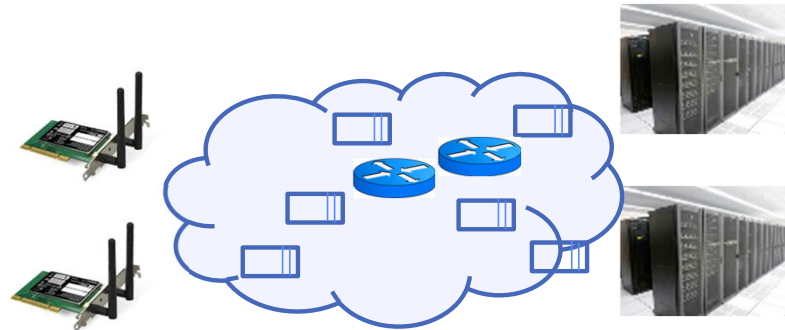
- k servers handling tasks of length L delivers at rate $\leq k/L$.
 - $L = 10$ ms, $k = 4 \rightarrow B = 400$ ^{op}/s
 - $L = 2$ yr, $k = 2 \rightarrow B = 1$ ^{op}/yr
- In 61C you saw multiple processors (cores)
 - Systems present lots of multiple parallel servers
 - Often with lots of queues

Example Systems “Parallelism”

I/O Processing

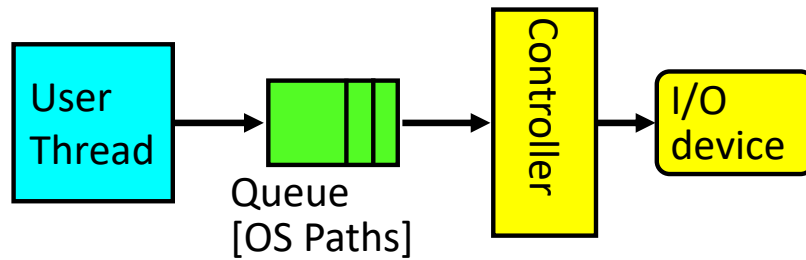


Communication



Parallel Computation, Databases, ...

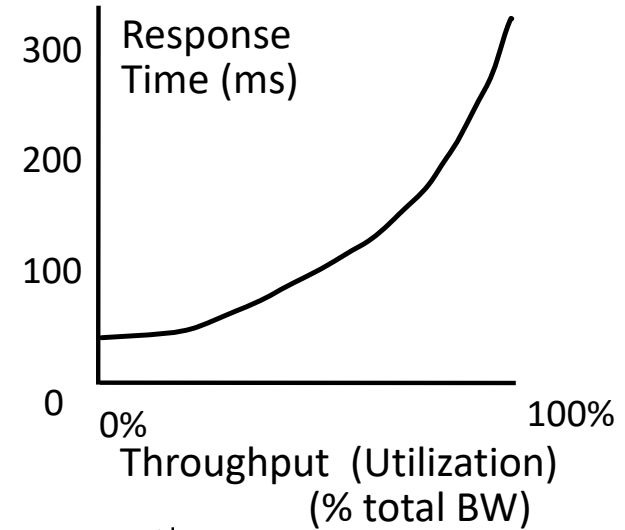
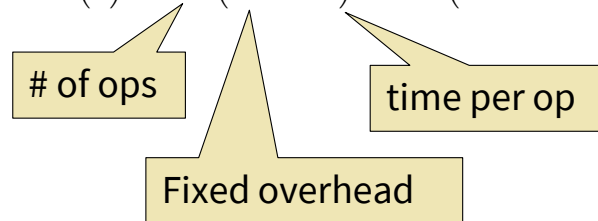
I/O Performance



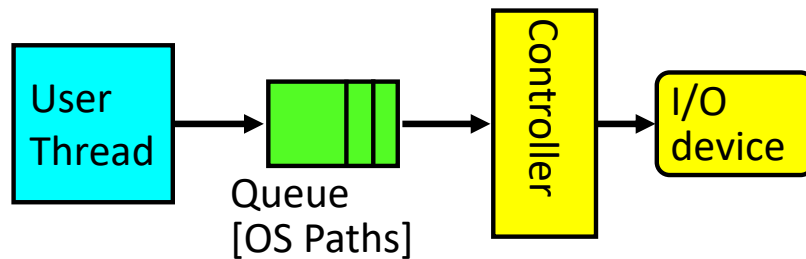
Response Time = Queue + I/O device service time

- Performance of I/O subsystem
 - Metrics: Response Time, Throughput
 - Effective BW per op = transfer size / response time

$$\gg \text{EffBW}(n) = n / (S + n/B) = B / (1 + SB/n)$$

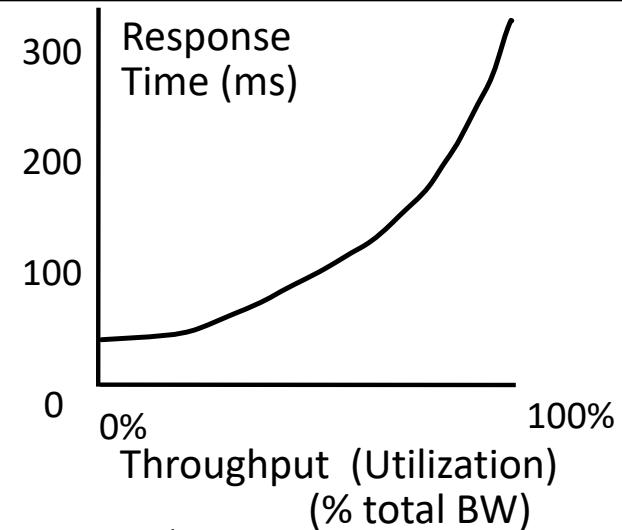


I/O Performance



Response Time = Queue + I/O device service time

- Performance of I/O subsystem
 - Metrics: Response Time, Throughput
 - Effective BW per op = transfer size / response time
 - » $\text{EffBW}(n) = n / (S + n/B) = B / (1 + SB/n)$
 - 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?



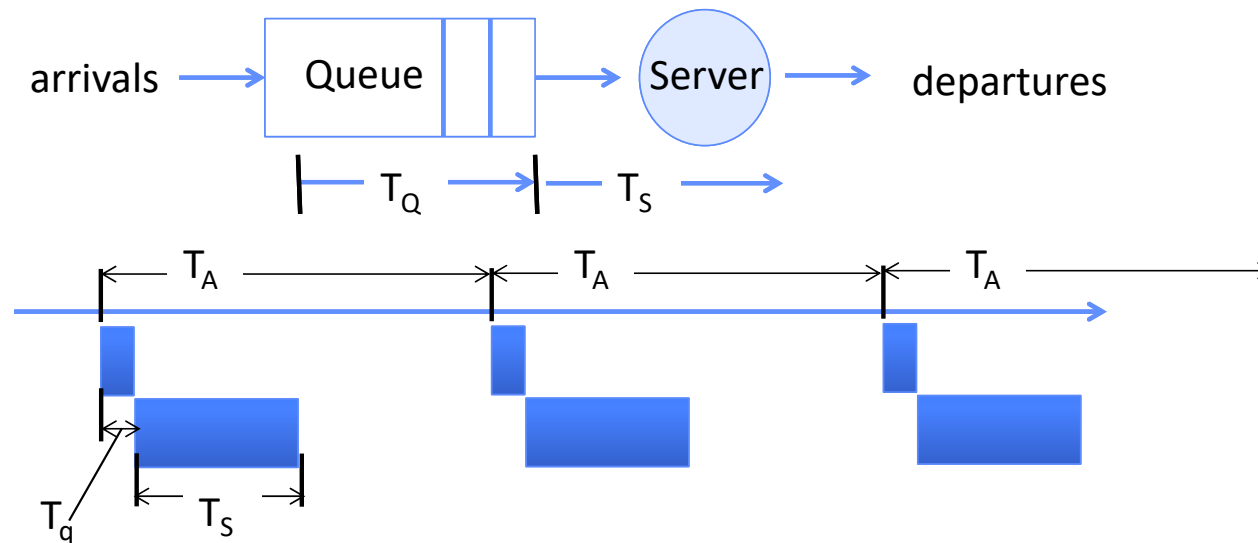
Administrivia

- Midterm 2 grading still in process
 - It has been promised that they will be out this week
 - I'm really pushing to have results before Thursday's class
- Midterm 3 on Thursday, April 30
 - All topics up to previous Tuesday (4/28) are in scope
 - Closed book, 3 pages, double-sided *handwritten* notes.
- Project 3 is out!
 - Give moving, since the end of term is approaching rapidly!
 - (That light you see is an approaching train!)
 - Design document due 4/17
- Other deadlines:
 - HW5 Checkpoint \Rightarrow Saturday 4/18
 - HW5 Due \Rightarrow Friday 4/24
 - HW6 Release \Rightarrow Saturday 4/25

Administrivia (Con't)

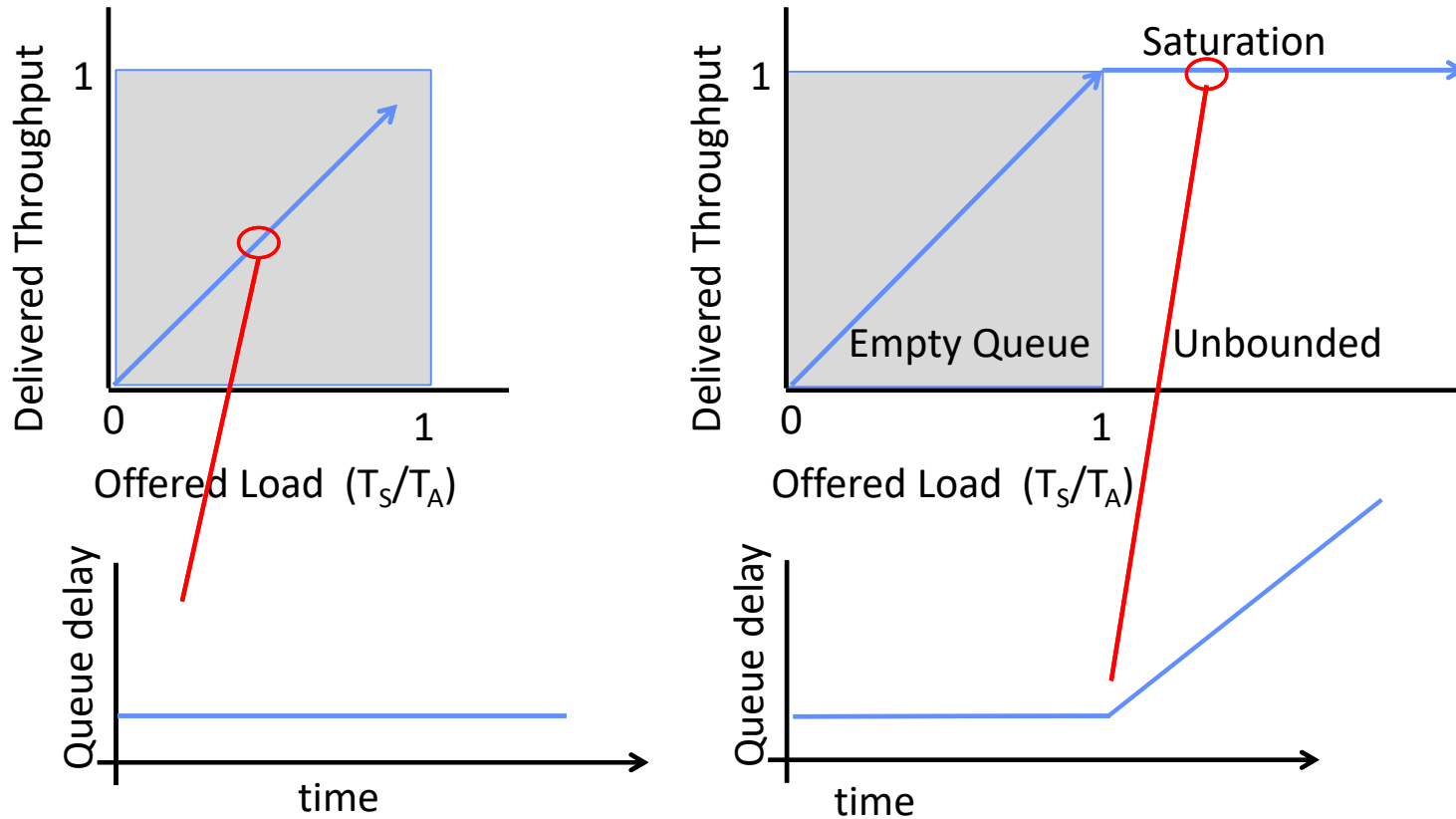
- Project 2 group evaluations:
 - Make sure to complete them *correctly*: They are crucial to our understanding of group dynamics
 - The grading is a zero-sum game – if you don't put in the work, it is possible we will give your points to other group members
 - Make sure your TA understands how your group is doing
- Many of you can't seem to follow directions:
 - You get 20 points for each of your partners (so, if you are in 4 person group, $20 \times 3 = 60$ pts)
 - Give these points out to your partners in way that adds to your total
 - Give a reason for your evaluation
 - You DO NOT evaluate yourself
- For those of you who cannot follow directions, we will be taking away 1 slip day point
 - We will give a temporary chance to correct your evaluations
 - It is up to you to check that you did it correctly

A Simple Deterministic World



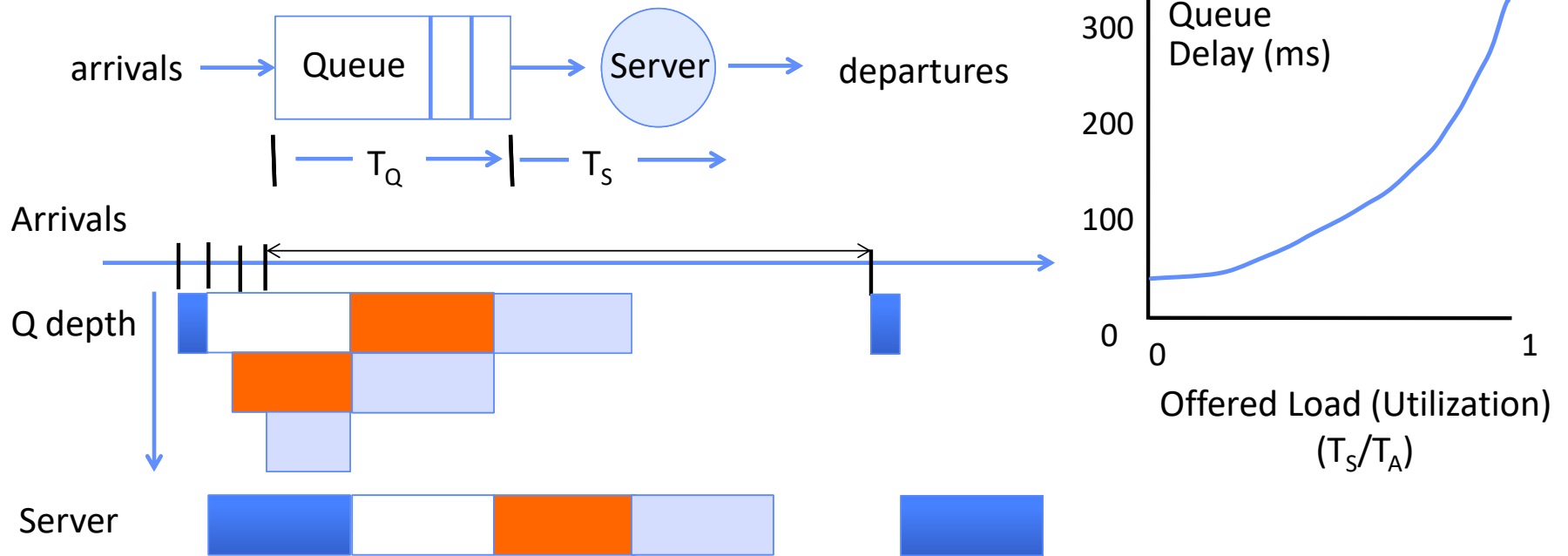
- Assume requests arrive at regular intervals, take a fixed time to process, with plenty of time between ...
- Service rate ($\mu = 1/T_S$) - operations per second
- Arrival rate: ($\lambda = 1/T_A$) - requests per second
- Utilization: $U = \lambda/\mu$, where $\lambda < \mu$
- Average rate is the complete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

A Ideal Linear World



- What does the queue wait time look like?
 - Grows *unbounded* at a rate $\sim (T_S/T_A)$ till request rate subsides

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 not too high
- Delay gets much worse as utilization $\Rightarrow 1$

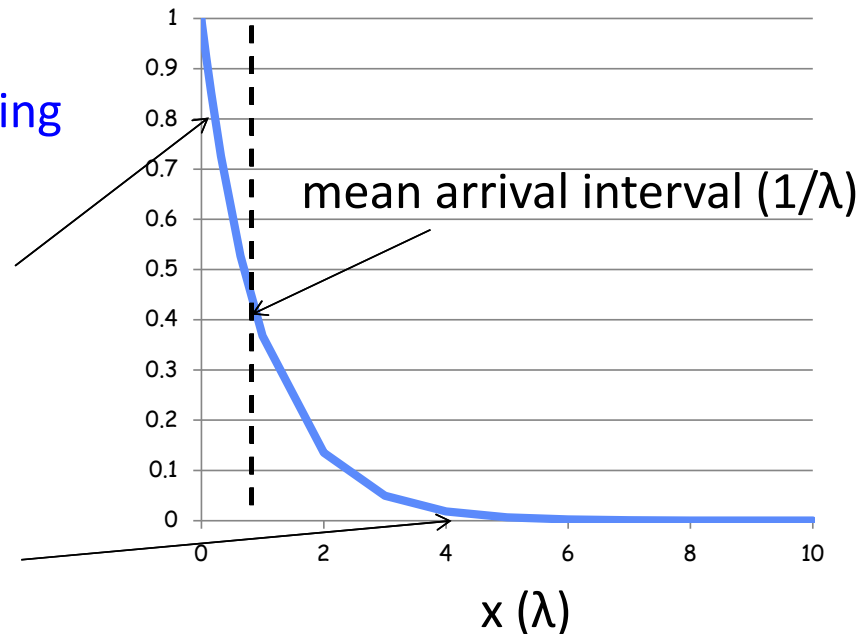
So how do we model the burstiness of arrival?

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

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

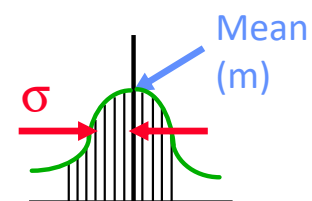
Lots of short arrival
intervals (i.e., high
instantaneous rate)

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

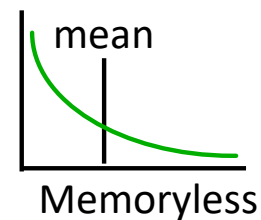


Background: General Use of Random Distributions

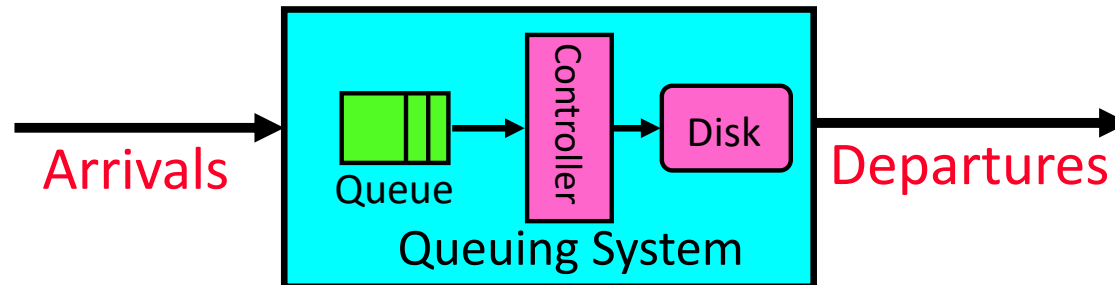
- Server spends variable time (T) with customers
 - Mean (Average) $m = \sum p(T) \times T$
 - Variance (stddev²) $\sigma^2 = \sum p(T) \times (T-m)^2 = \sum p(T) \times T^2 - m^2$
 - Squared coefficient of variance: $C = \sigma^2/m^2$
Aggregate description of the distribution
- Important values of C :
 - No variance or deterministic $\Rightarrow C=0$
 - “Memoryless” or exponential $\Rightarrow C=1$
 - » Past tells nothing about future
 - » Poisson process – *purely* or *completely* random process
 - » Many complex systems (or aggregates) are well described as memoryless
 - Disk response times $C \approx 1.5$ (majority seeks < average)



Distribution of service times

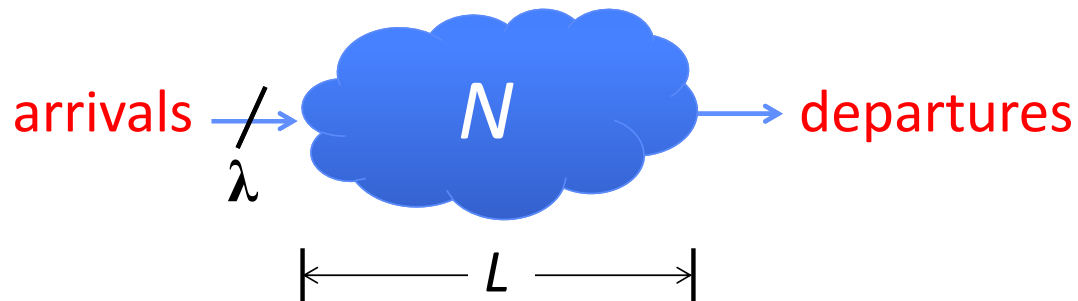


Introduction to Queuing Theory



- 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

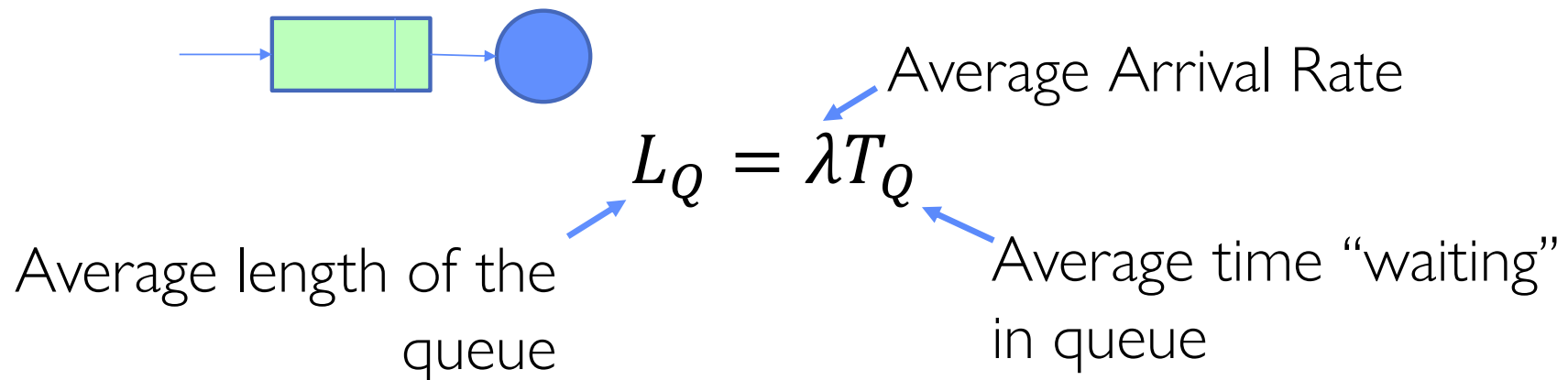
Little's Law



- In any *stable* system
 - Average arrival rate = Average departure rate
- The average number of jobs/tasks in the system (N) is equal to arrival time / throughput (λ) times the response time (L)
 - N (jobs) = λ (jobs/s) \times L (s)
- Regardless of structure, bursts of requests, variation in service
 - Instantaneous variations, but it washes out in the average
 - Overall, requests match departures

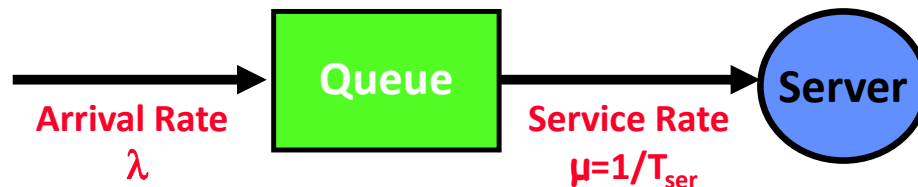
Little's Law Applied to a Queue

- When Little's Law applied to a queue, we get:

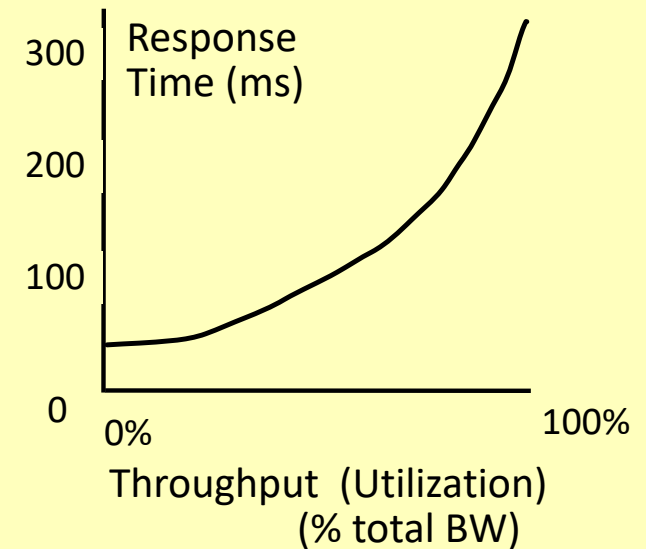


A Little Queuing Theory: Computing T_q

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



Why does response/queueing delay grow unboundedly even though the utilization is < 1 ?

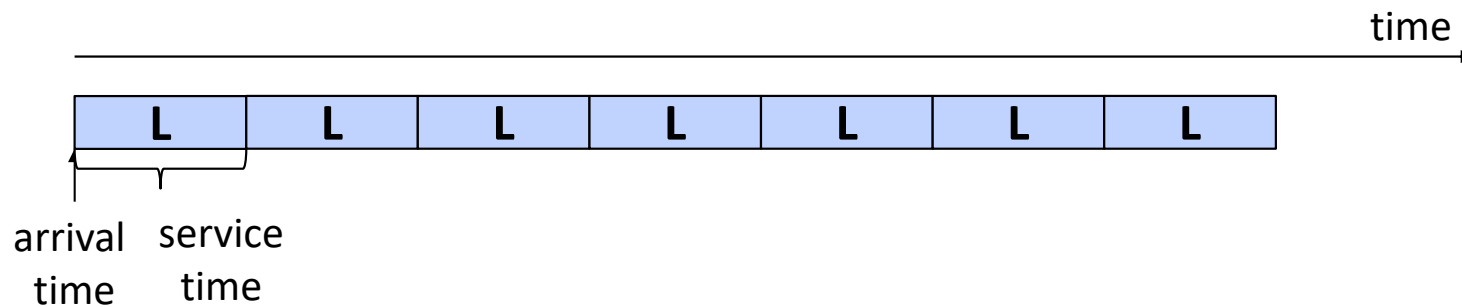


- Parameters that describe our system:
 - λ : mean number of arriving customers/second
 - T_{ser} : mean time to service a customer ("m1")
 - C : squared coefficient of variance = σ^2/μ^2
 - μ : service rate = $1/T_{ser}$
 - u : server utilization ($0 \leq u \leq 1$) = $\lambda/\mu = \lambda \times T_{ser}$

- Results:
 - Memoryless service distribution ($C = 1$) (an "M/M/1 queue"):
 - $T_q = T_{ser} \times u / (1 - u)$
 - General service distribution, 1 server (an "M/G/1 queue"):
 - $T_q = T_{ser} \times \frac{1}{2}(1+C) \times u / (1 - u)$

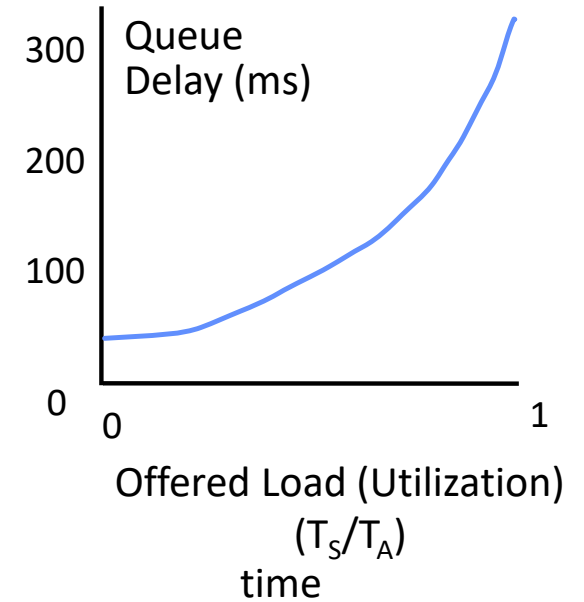
Why unbounded response time for special case $u = 1$?

- For $u > 1$, already see the problem:
 - More work arrives per unit time than can be serviced
- But: Consider $u = 1$
 - Assume deterministic arrival process and service time
 - Possible to sustain utilization = 1 with bounded response time!

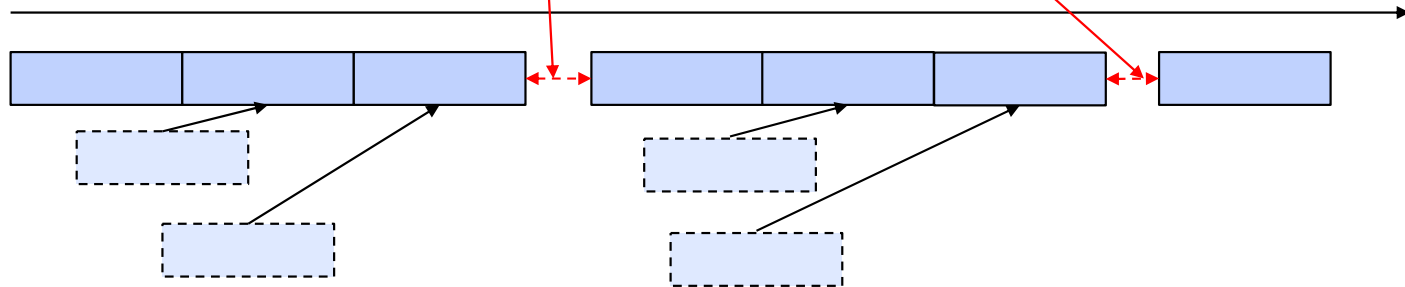


Why unbounded response time for $u=1$ with randomness?

- Assume stochastic arrival process
 - No longer possible to achieve $u = 1$
 - Queue grows in size without bound
- For $u < 1$, barely keep up \Rightarrow large queue



This wasted time can never be reclaimed!
So cannot achieve $u = 1$!



A Little Queuing Theory: An Example

- Example Usage Statistics:
 - User requests 10 × 8KB disk I/Os per second
 - Requests & service exponentially distributed (C=1.0)
 - Avg. service = 20 ms (From controller+seek+rot+trans)
- Questions:
 - How utilized is the disk?
 - » Ans: server utilization, $u = \lambda T_{ser}$
 - 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: $T_{sys} = T_q + T_{ser}$

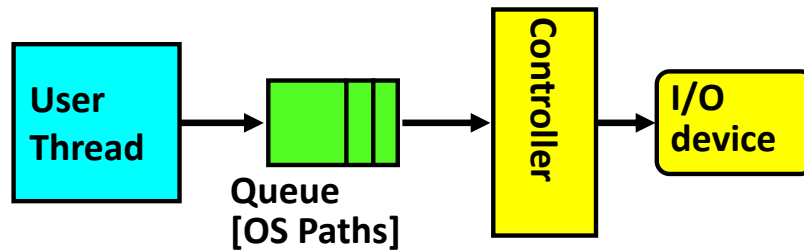
- Computation:

$$\begin{aligned}\lambda & \text{ (avg \# arriving customers/s) } = 10/\text{s} \\ T_{ser} & \text{ (avg time to service customer) } = 20 \text{ ms (0.02s)} \\ u & \text{ (server utilization) } = \lambda \times T_{ser} = 10/\text{s} \times .02\text{s} = 0.2 \\ T_q & \text{ (avg time/customer in queue) } = T_{ser} \times u / (1 - u) \\ & = 20 \times 0.2 / (1 - 0.2) = 20 \times 0.25 = 5 \text{ ms (0.005s)} \\ L_q & \text{ (avg length of queue) } = \lambda \times T_q = 10/\text{s} \times .005\text{s} = 0.05 \\ T_{sys} & \text{ (avg time/customer in system) } = T_q + T_{ser} = 25 \text{ ms}\end{aligned}$$

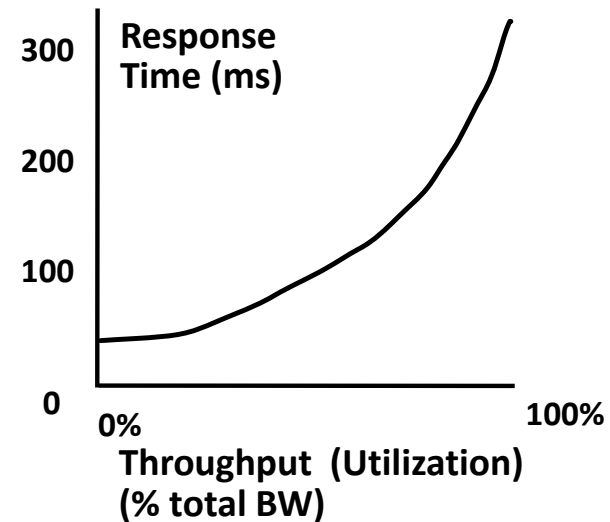
Queuing Theory Resources

- Resources page contains Queueing Theory Resources (under Readings):
 - Scanned pages from Patterson and Hennessy book that gives further discussion and simple proof for general equation:
https://cs162.eecs.berkeley.edu/static/readings/patterson_queue.pdf
 - A complete website full of resources: <http://web2.uwindsor.ca/math/hlynka/qonline.html>
- Some previous midterms with queueing theory questions
- Assume that Queueing Theory is fair game for Midterm III!

Optimize I/O Performance



**Response Time =
Queue + I/O device service time**



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

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*

What we covered in early Lectures

File System

Files/Directories/Indexes

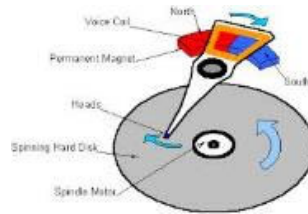
What we will cover next...

I/O Driver

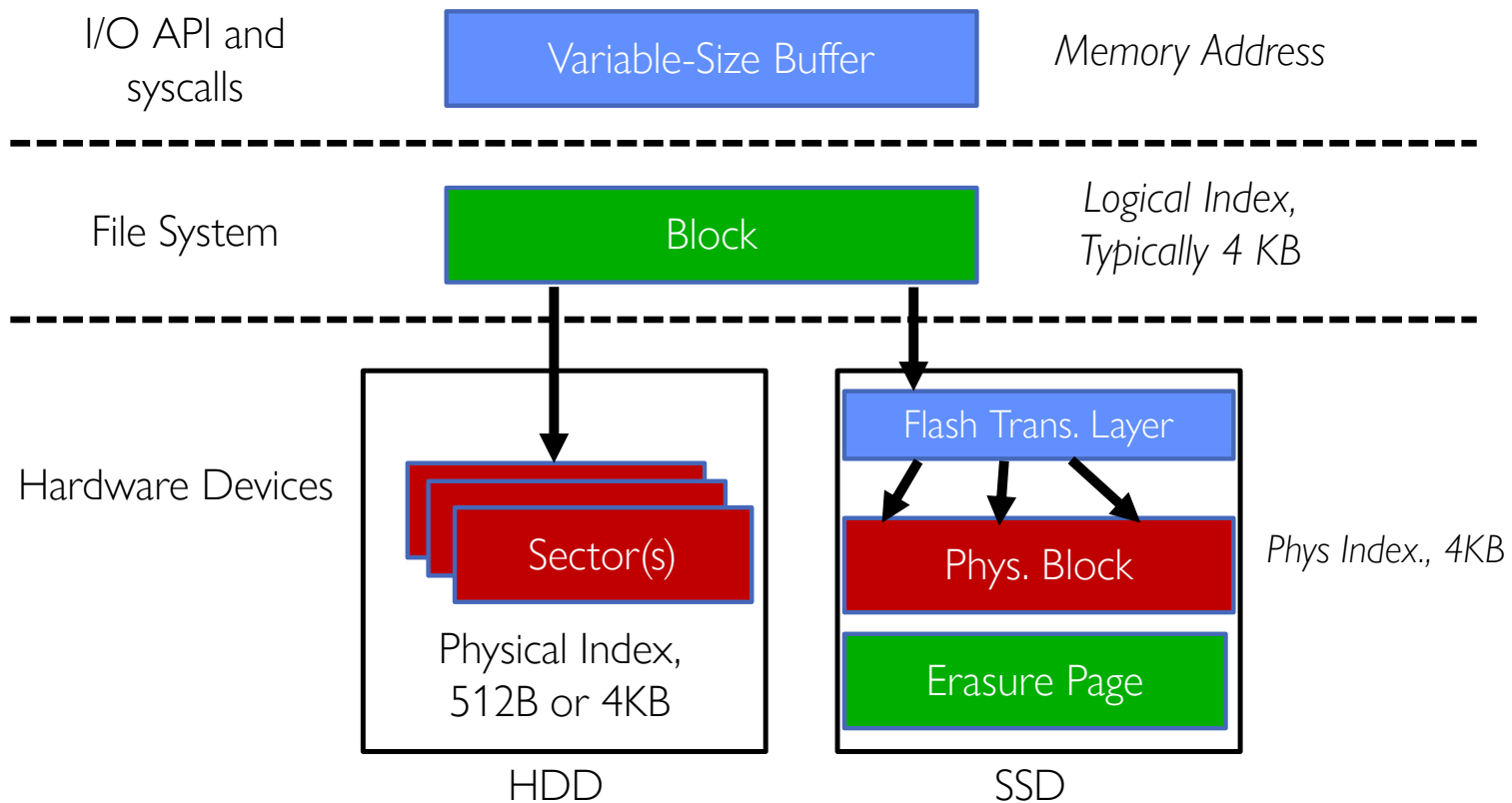
Commands and Data Transfers

Disks, Flash, Controllers, DMA

What we just covered...



From Storage to File Systems



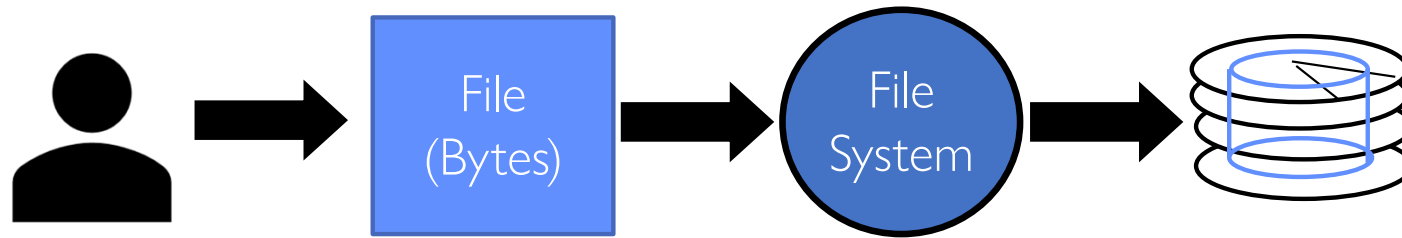
Building a File System

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

Recall: 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 typically 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 anymore
 - » 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 free disk blocks
 - Need to know where to put newly written data
- 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
- Where do we maintain all of this?
 - Somewhere on disk

Data Structures on Disk

- Different than data structures in memory
 - Must load from disk into memory to manipulate
 - Modifications to disk data are *really* expensive, so only change when needed
- 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...

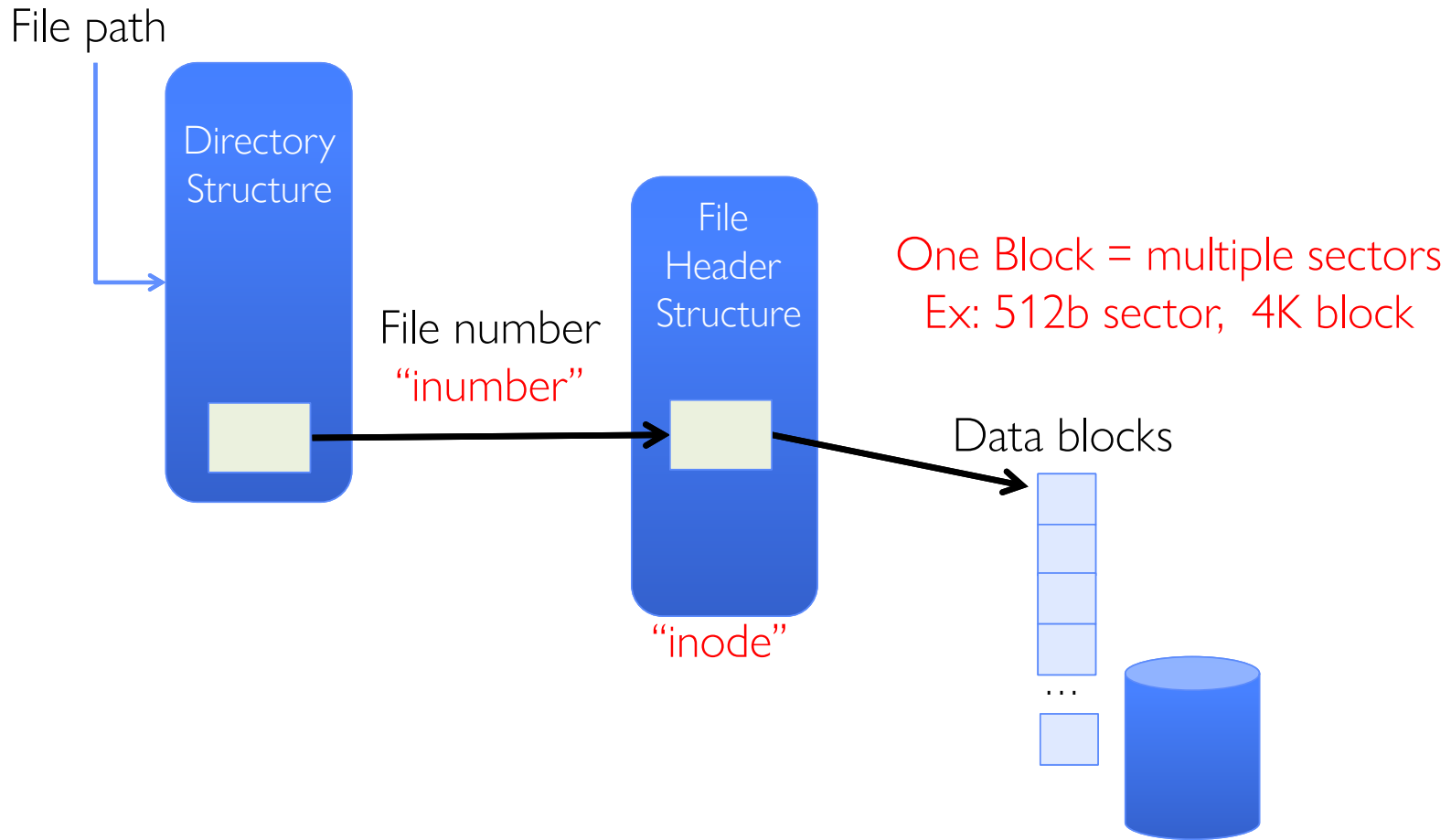


FILE SYSTEM DESIGN

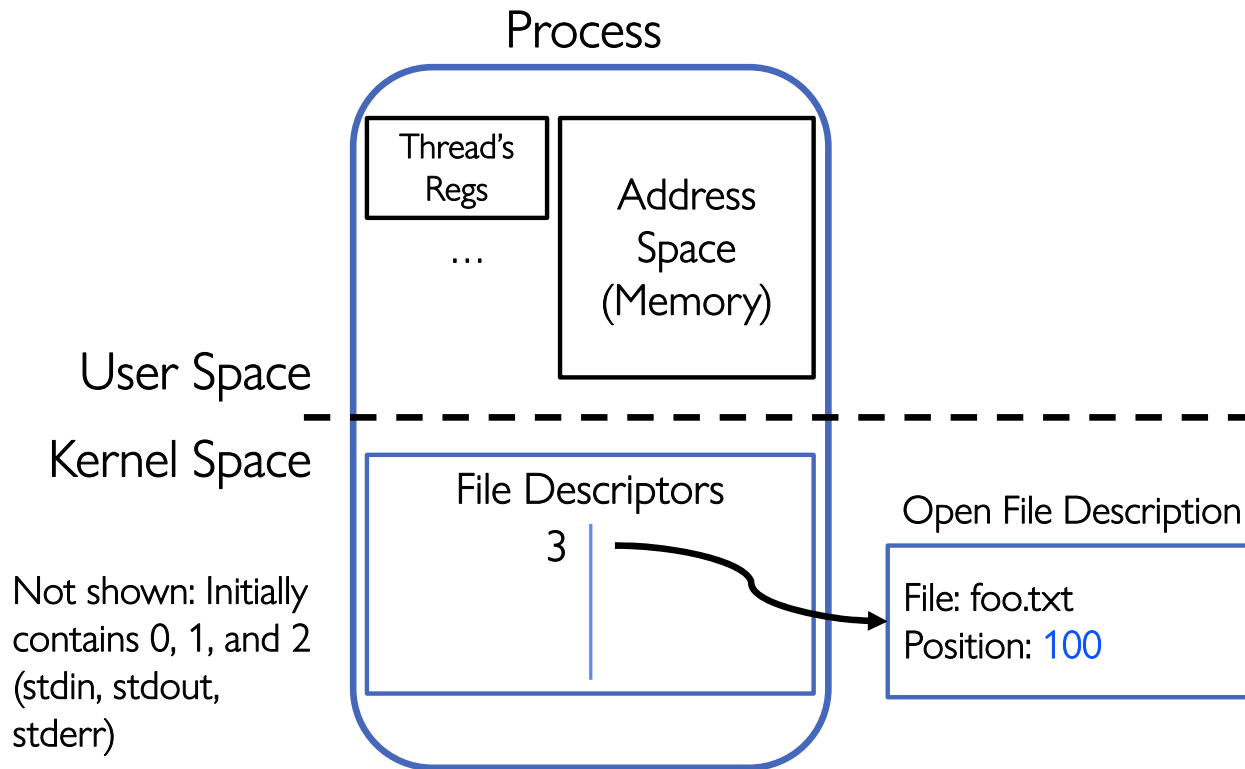
Critical Factors in File System Design

- (Hard) Disks 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 they are used !!!
 - Can write (or read zeros) 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

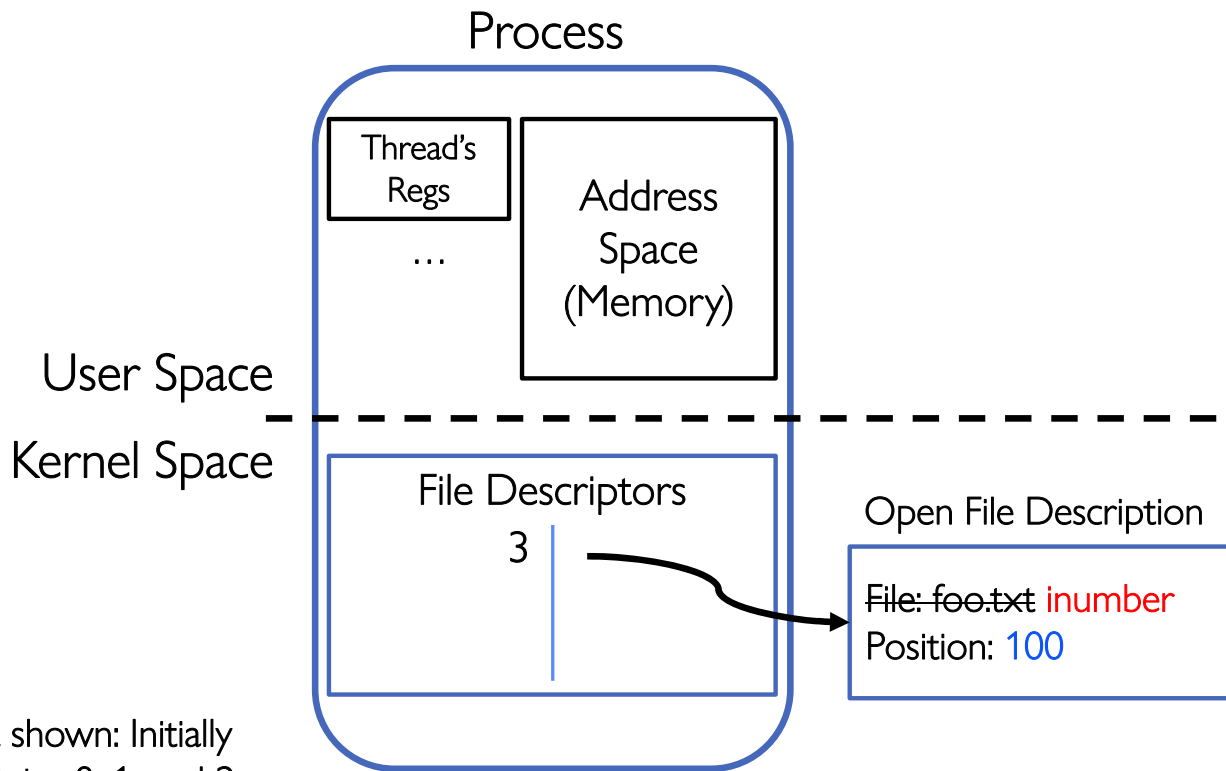


Suppose that we execute
`open("foo.txt")`
and that the result is 3

Next, suppose that we execute

`read(3, buf, 100)`
and that the result is 100

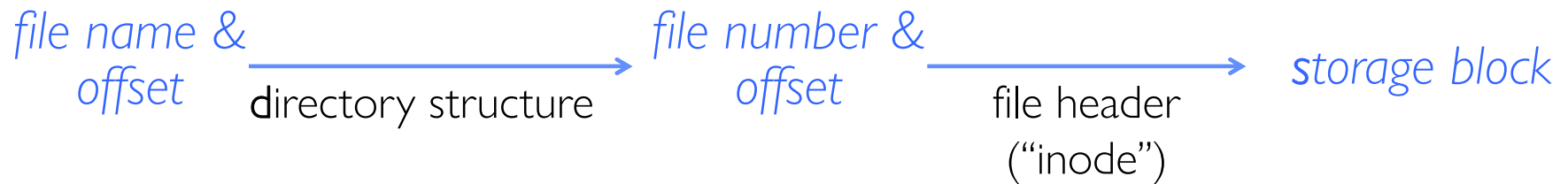
Components of a File System



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

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

Components of a File System

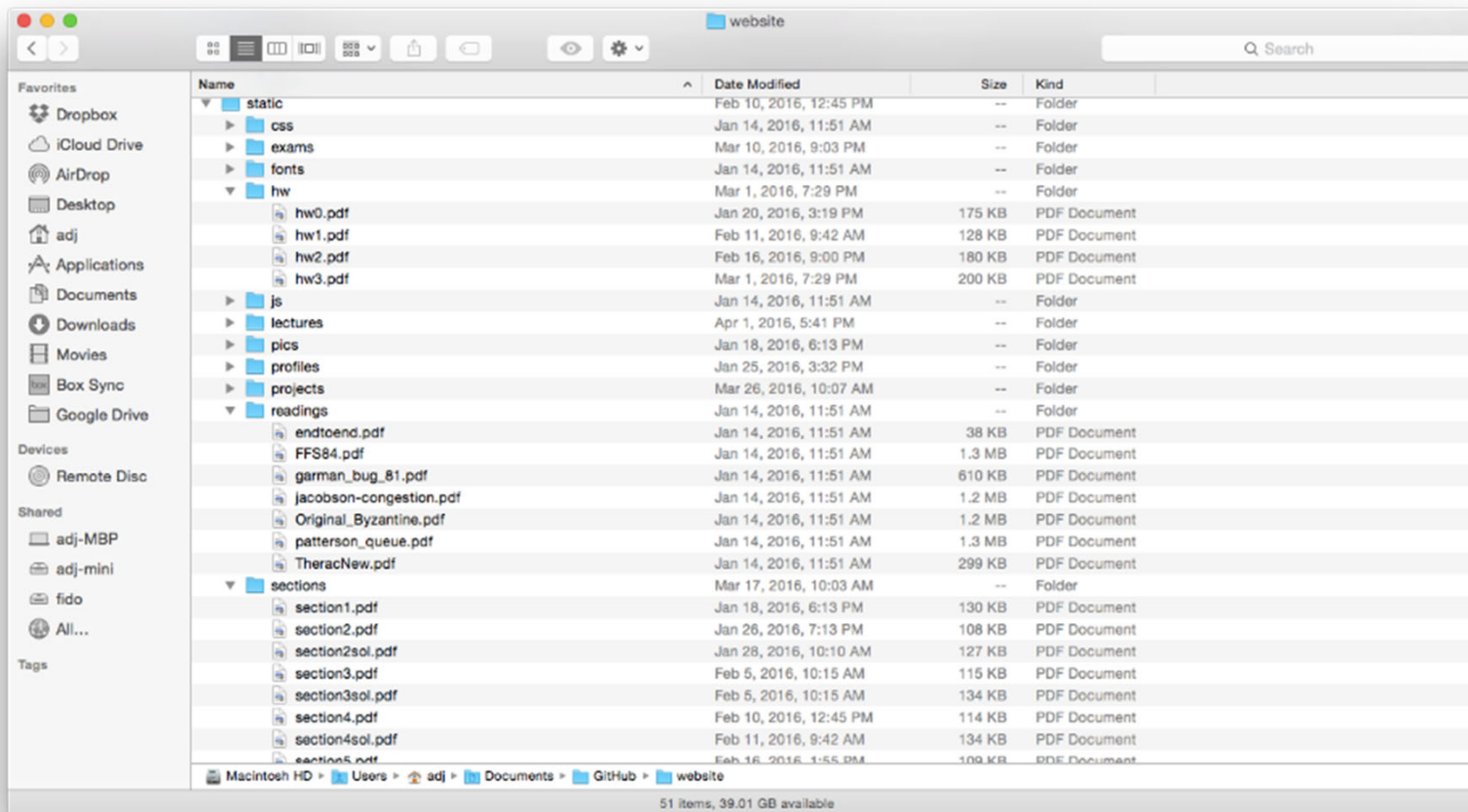


- Open performs *Name Resolution*
 - Translates path name into a *file number*
 - Checks access permissions (for file systems with permissions)
- File number is an index to *file header* structure describing the file
 - Often called an *inode*
 - File header gives us enough information to find all the blocks of the file – wherever they are on disk
- Read and Write operate on the file using the file header
 - Use file header as an “index” to locate the blocks
 - Much of the distinction between file systems is with respect to the structure of the file header and/or directories
- **4 components:**
 - **directory, file header, storage blocks, free space map**

How to get the File Number?

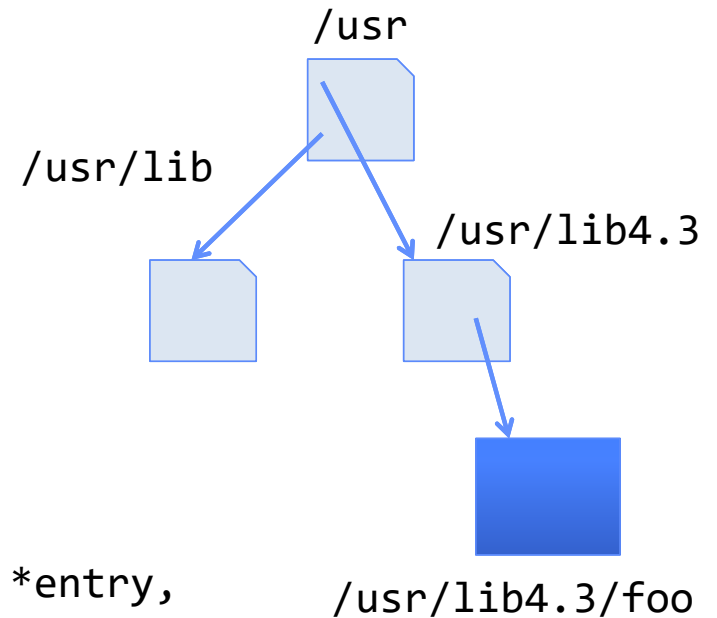
- Look up in *directory structure*
- A directory is a file containing <file_name : file_number> mappings
 - File number could be a file or another directory
 - Operating system stores the mapping in the directory in a format it interprets
 - Each <file_name : file_number> mapping is called a directory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

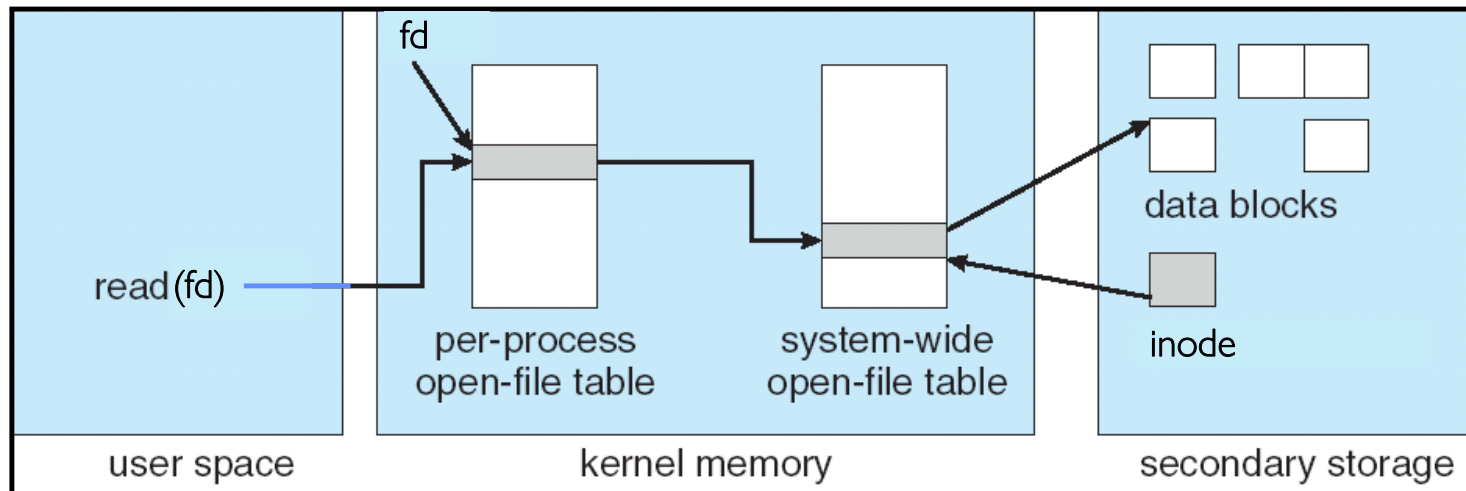
- Directories are specialized files
 - Contents: List of pairs <file name, file number>
- System calls to access directories
 - `open` / `creat` / `readdir` traverse the structure
 - `mkdir` / `rmdir` add/remove entries
 - `link` / `unlink` (`rm`)
- libc support
 - `DIR * opendir (const char *dirname)`
 - `struct dirent * readdir (DIR *dirstream)`
 - `int readdir_r (DIR *dirstream, struct dirent *entry, struct dirent **result)`



Directory Structure

- How many disk accesses to resolve “/my/book/count”?
 - Read in file header for root (fixed spot 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

Observation #1: Most Files Are Small

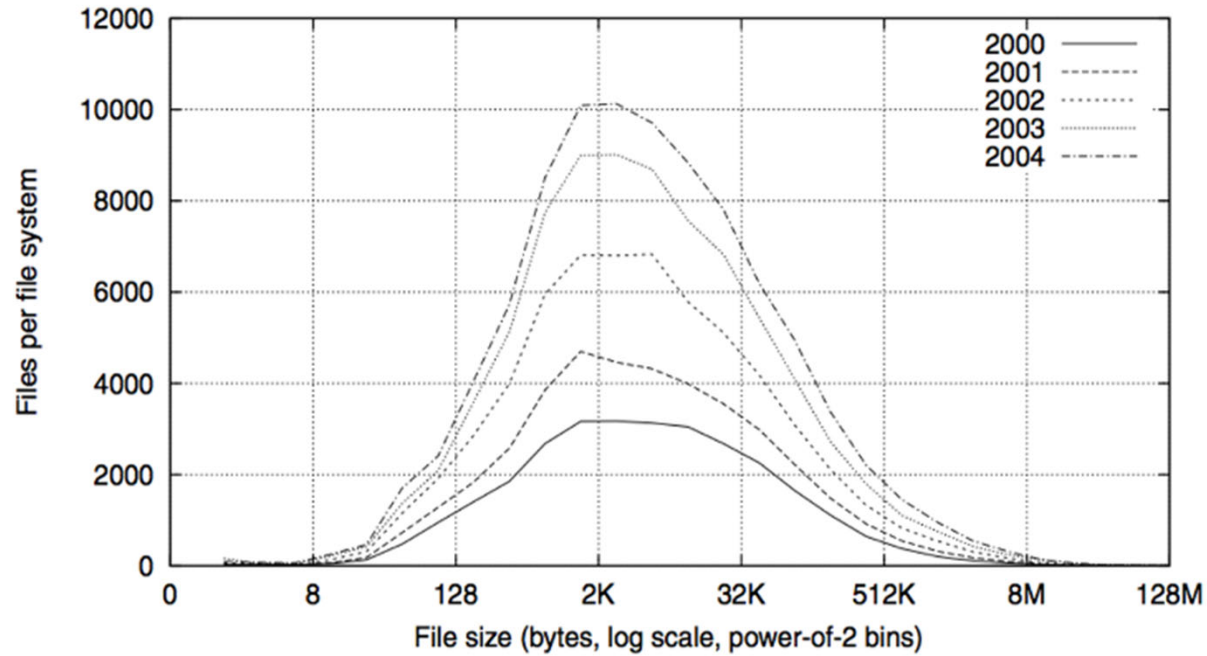


Fig. 2. Histograms of files by size.

Observation #2: Most Bytes are in Large Files

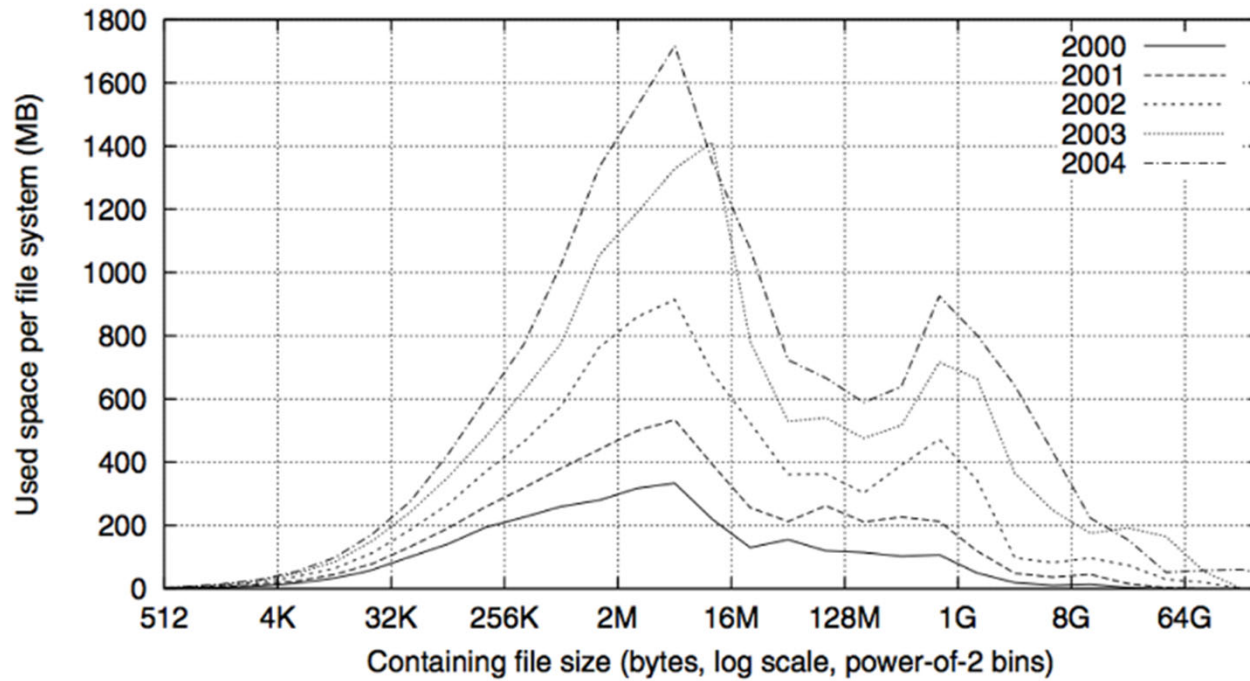


Fig. 4. Histograms of bytes by containing file size.

Conclusion

- Devices have complex interaction and performance characteristics
 - Response time (Latency) = Queue + Overhead + Transfer
 - Effective BW = $BW * T/(S+T)$
- Queuing Latency: Bursts & High Utilization introduce queuing delays
 - 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
- Next time: File Allocation Table (FAT) Scheme
 - Linked-list approach
 - Very widely used: Cameras, USB drives, SD cards
 - Simple to implement, but poor performance and no security