CS162
Operating Systems and Systems Programming
Lecture 21

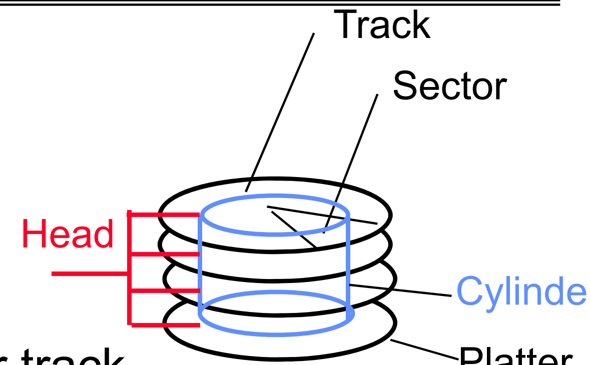
Filesystems 1: Performance, Queueing Theory, Filesystem Design

April 6th, 2023

Prof. John Kubiatowicz

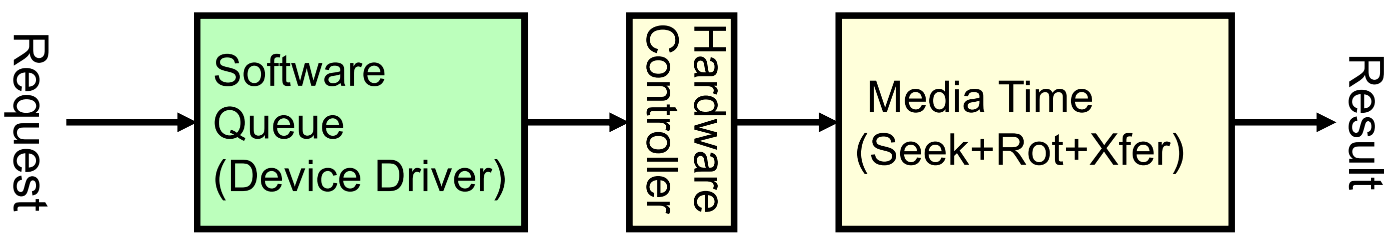
http://cs162.eecs.Berkeley.edu

Recall: Magnetic Disks

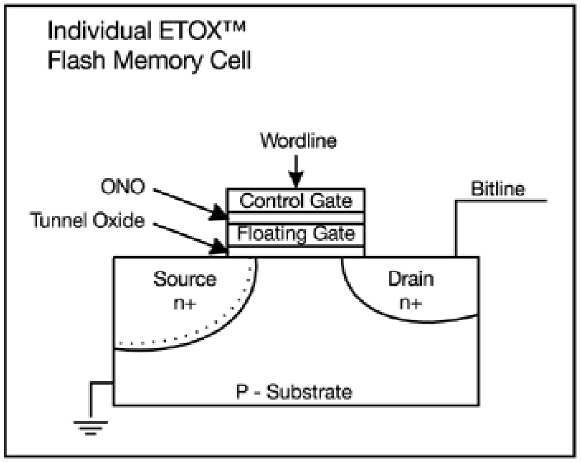


- Cylinders: all the tracks under the head at a given point on all surfaces
- Read/write data is a three-stage process:
  - Seek time: position the head/arm over the proper track
  - Rotational latency: wait for desired sector to rotate under r/w head
  - Transfer time: transfer a block of bits (sector)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

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



Recall: FLASH Memory



Samsung 2015: 512GB, NAND Flash

- Like a normal transistor but:
  - Has a floating gate that can hold charge
  - To write: raise or lower wordline high enough to cause charges to tunnel
  - To read: turn on wordline as if normal transistor
    - » presence of charge changes threshold and thus measured current
- Two varieties:
  - NAND: denser, must be read and written in blocks
  - NOR: much less dense, fast to read and write
- V-NAND: 3D stacking (Samsung claims 1TB possible in 1 chip)

Ways of Measuring Performance: Times (s) and Rates (op/s)

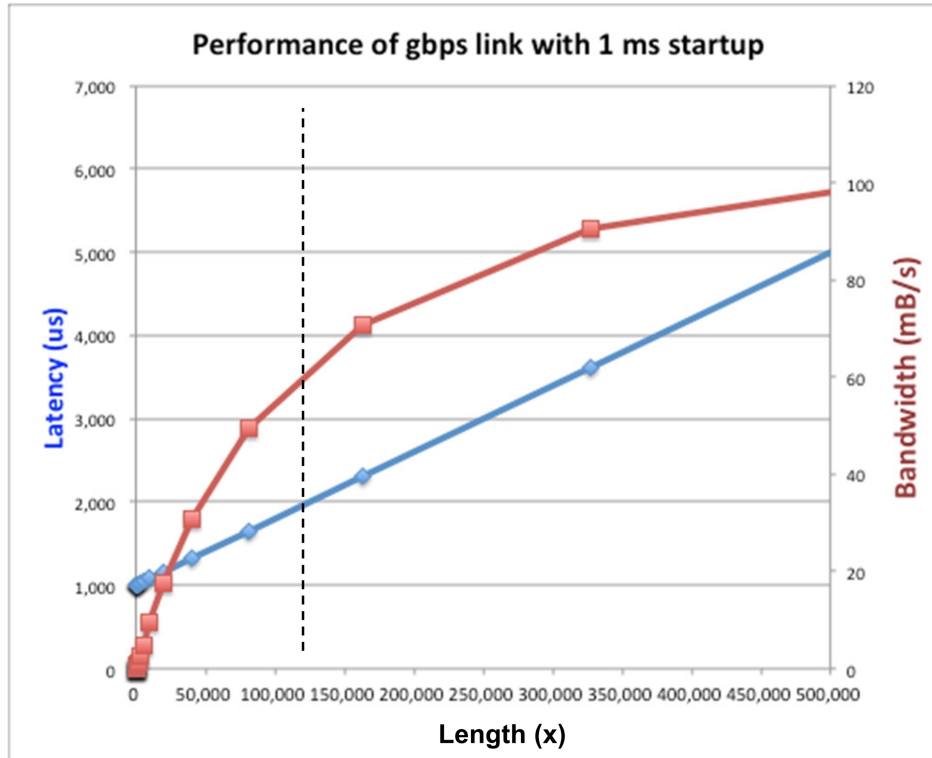
- ***Latency*** – time to complete a task
  - Measured in units of time (s, ms, us, …, hours, years)
- ***Response Time*** - time to initiate and operation and get its response
  - Able to issue one that *depends* on the result
  - Know that it is done (anti-dependence, resource usage)
- ***Throughput*** 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
  - Latency(b) = Overhead + b/TransferCapacity
- Performance???
  - Operation time (4 mins to run a mile…)
  - Rate (mph, mpg, …)

Example: Overhead in Fast Network

- Consider a $1\ \mathrm{Gb/s}$ link ($B_w = 125\ \mathrm{MB/s}$) with startup cost $S = 1\ \mathrm{ms}$
- Latency: $L(x) = S + \frac{x}{B_w}$
- Effective Bandwidth:

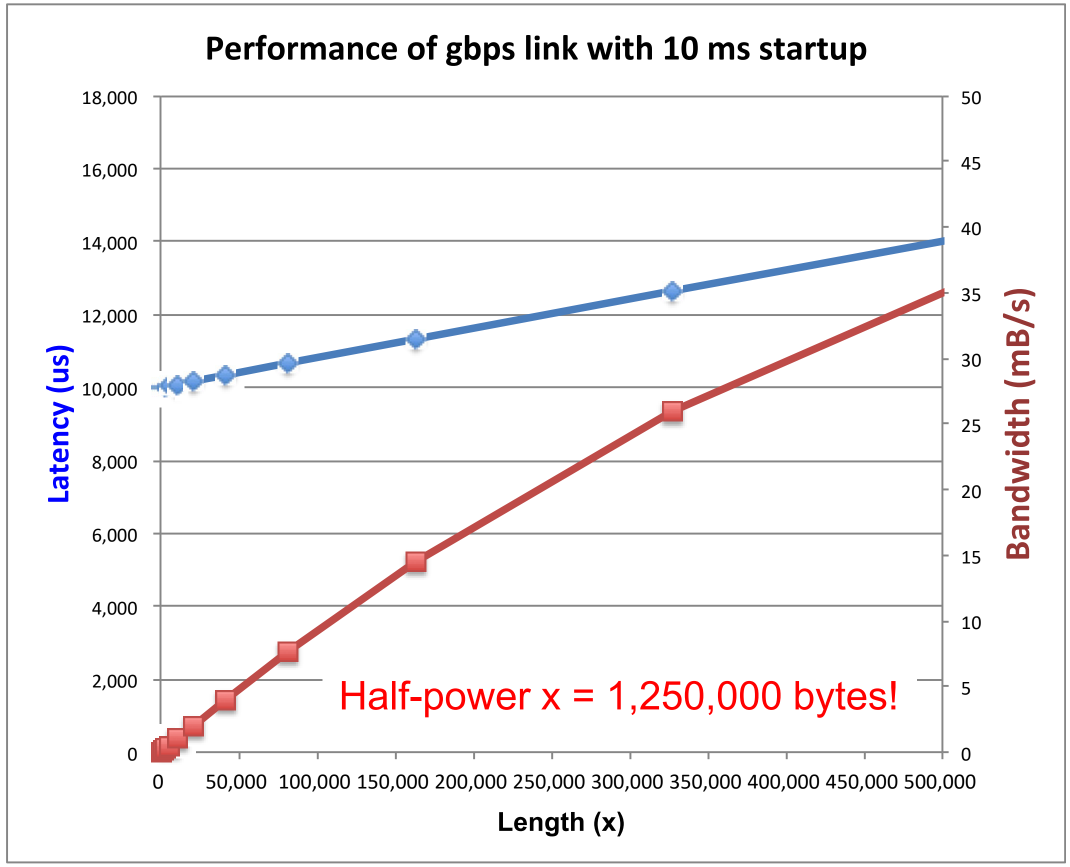
$$E(x) = \frac{x}{S + \frac{x}{B_w}} = \frac{B_w \cdot x}{B_w \cdot S + x} = \frac{B_w}{\frac{B_w \cdot S}{x} + 1}$$

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



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

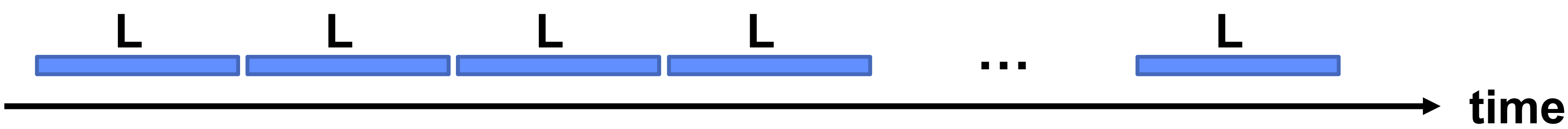
- Half-power bandwidth at $x = 1.25\ \mathrm{MB}$
- Large startup cost can degrade effective bandwidth
- Amortize it by performing I/O in larger blocks



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…

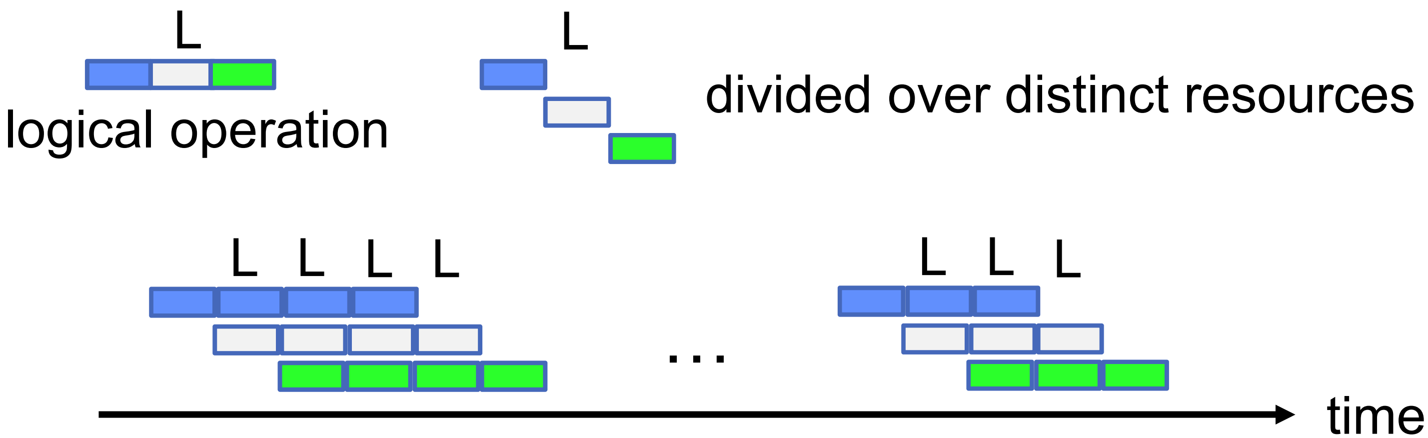
Sequential Server Performance



- Single sequential "server" that can deliver a task in time L operates at rate $\leq \frac{1}{L}$ (on average, in steady state, …)
  - $L = 10$ ms $\rightarrow B = 100$ $^{\text{op}}/_{\text{s}}$
  - $L = 2$ yr $\rightarrow B = 0.5$ $^{\text{op}}/_{\text{yr}}$

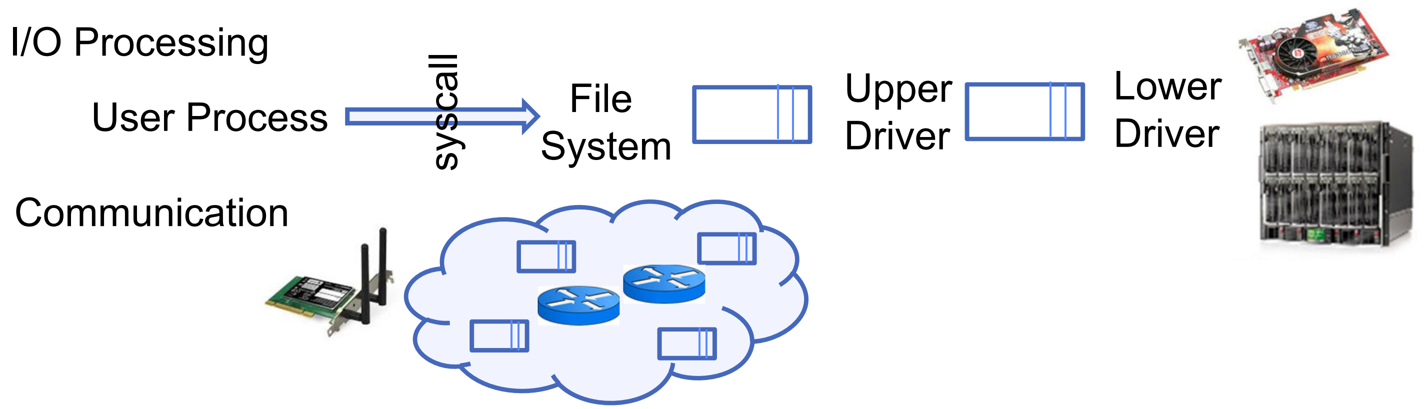
- Applies to a processor, a disk drive, a person, a TA, …

Single Pipelined Server



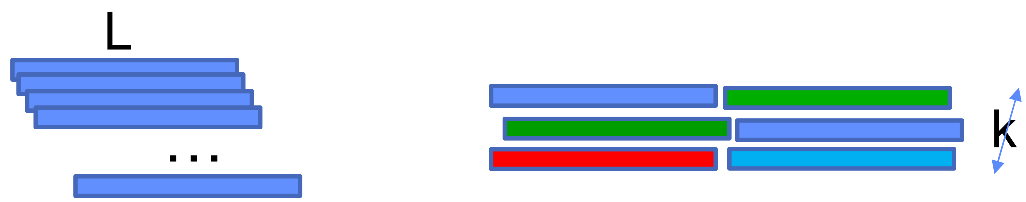
- Single pipelined server of k stages for tasks of length L (i.e., time $^{L}/_{k}$ per stage) delivers at rate $\leq {}^{k}/_{L}$.
  - $L = 10$ ms, $k = 4 \rightarrow B = 400\ ^{\text{op}}/_{\text{s}}$
  - $L = 2$ yr, $k = 2 \rightarrow B = 1\ ^{\text{op}}/_{\text{yr}}$

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

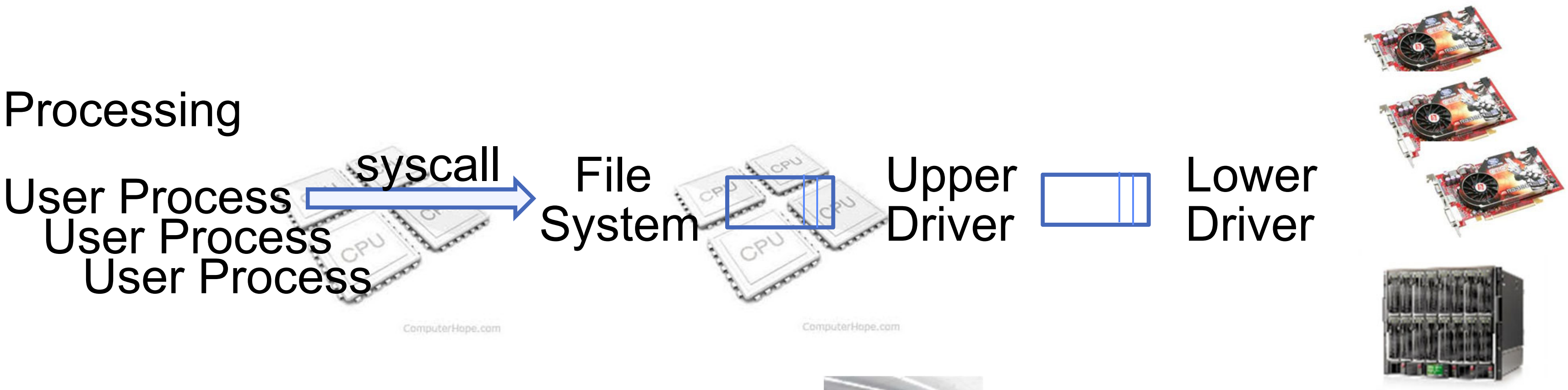
Multiple Servers



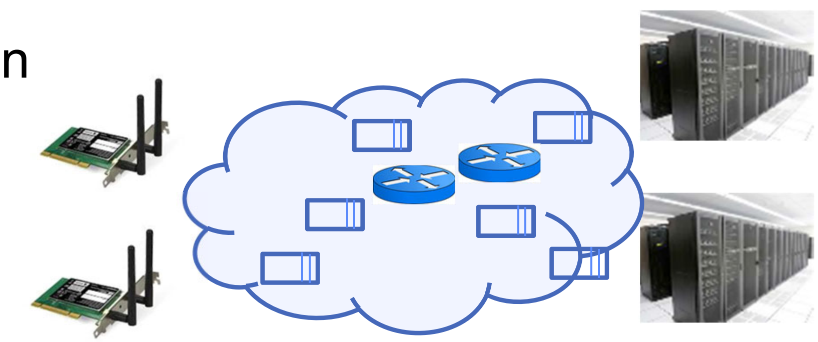
- k servers handling tasks of length L delivers at rate $\leq {}^{k}/_{L}$.
  - $L = 10$ ms, $k = 4 \rightarrow B = 400\ {}^{\text{op}}/_{\text{s}}$
  - $L = 2$ yr, $k = 2 \rightarrow B = 1\ {}^{\text{op}}/_{\text{yr}}$
- In 61C you saw multiple processors (cores)
  - Systems present lots of multiple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

Example Systems "Parallelism"

I/O Processing

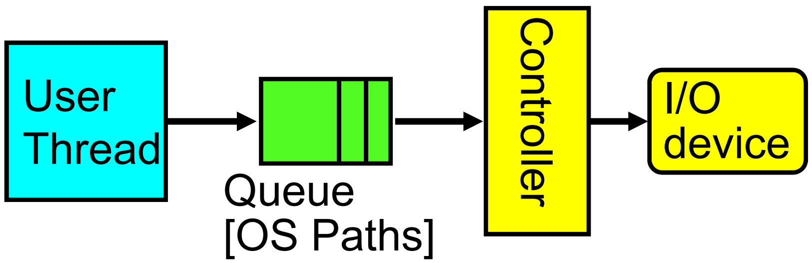


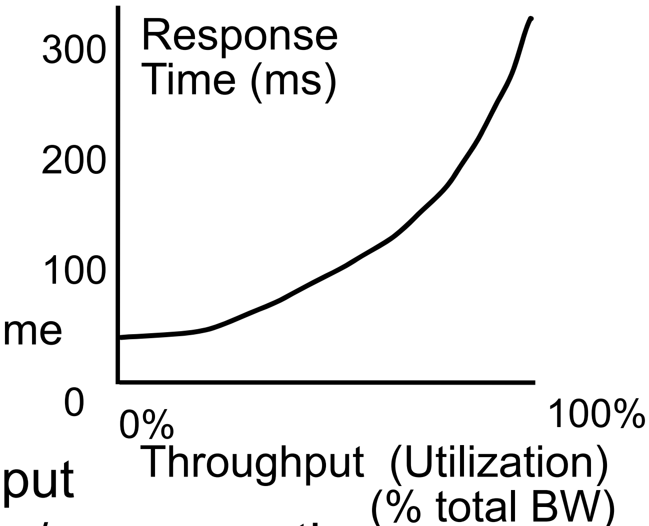
Communication



Parallel Computation, Databases, …

I/O Performance





Response Time = Queue + I/O device service time

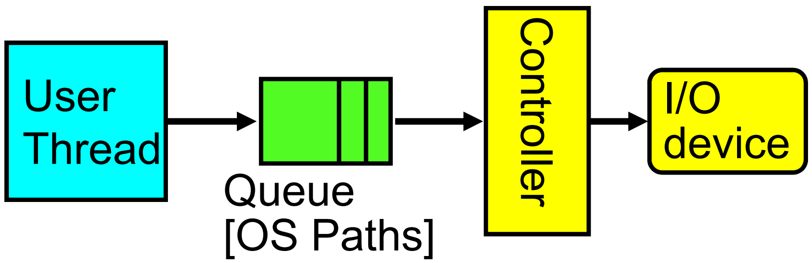
- Performance of I/O subsystem
  - Metrics: Response Time, Throughput
  - Effective BW per op = transfer size / response time
    - » EffBW(n) = n / (S + n/B) = B / (1 + SB/n)

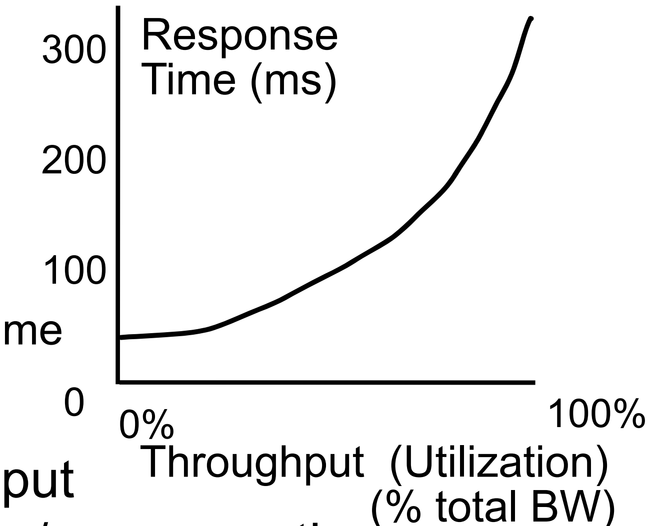
# of ops

Fixed overhead

time per op

I/O Performance

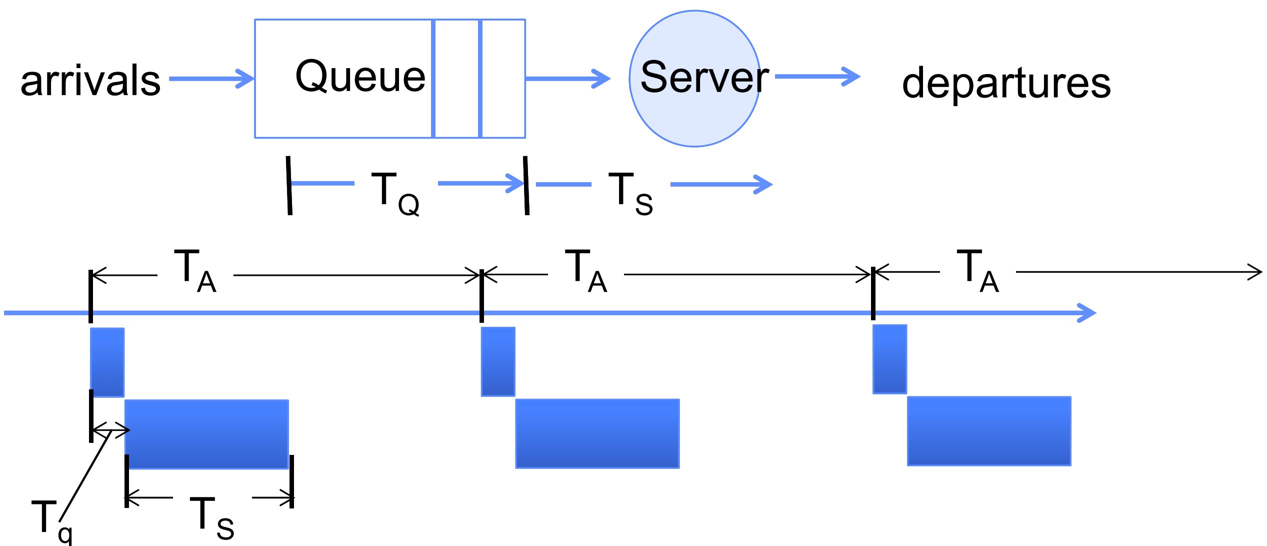




Response Time = Queue + I/O device service time

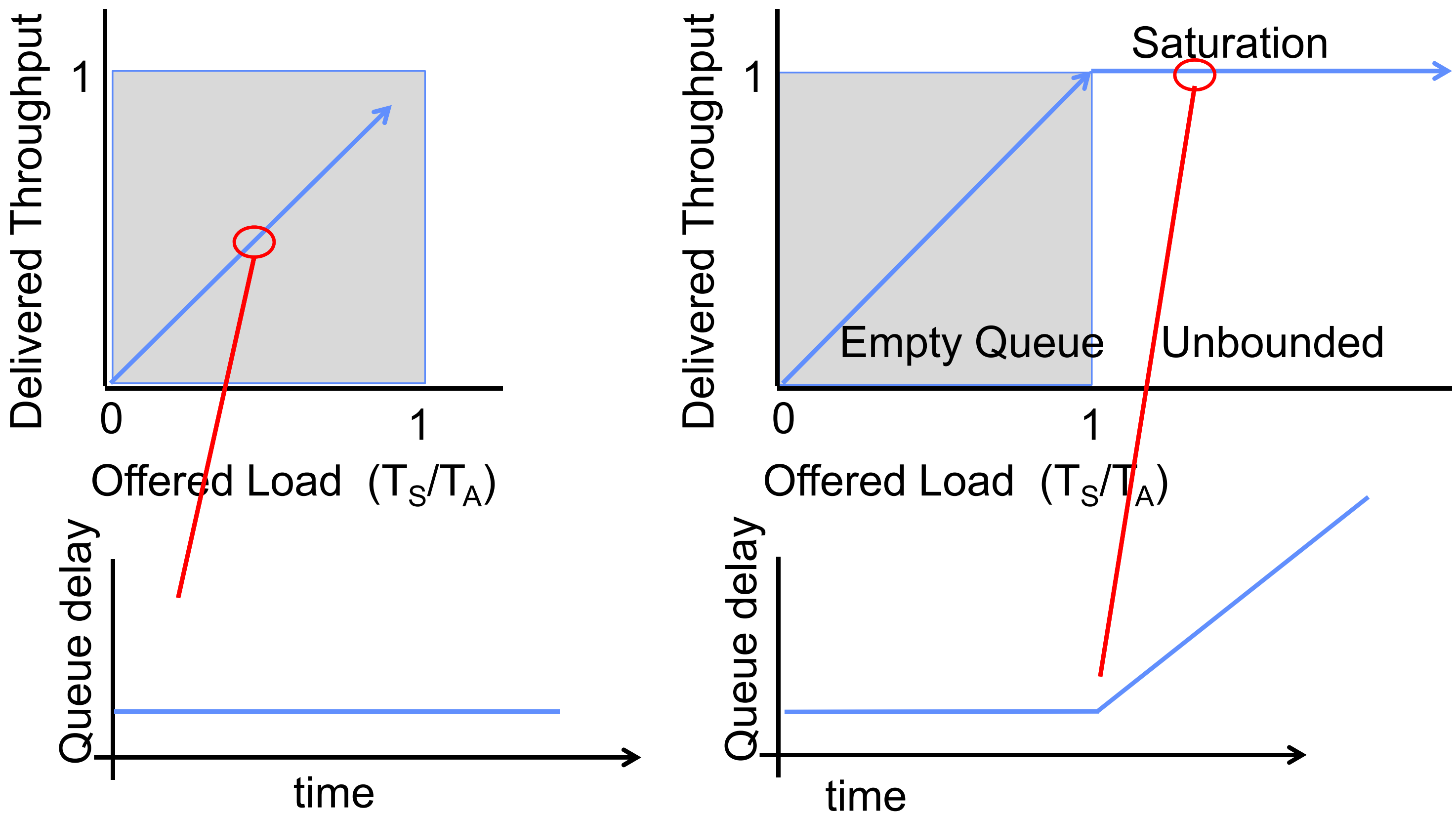
- Performance of I/O subsystem
  - Metrics: Response Time, Throughput
  - Effective BW per op = transfer size / response time
    - » EffBW(n) = n / (S + n/B) = B / (1 + SB/n)
  - Contributing factors to latency:
    - » Software paths (can be loosely modeled by a queue)
    - » Hardware controller
    - » I/O device service time
- Queuing behavior:
  - Can lead to big increases of latency as utilization increases
  - Solutions?

A Simple Deterministic World



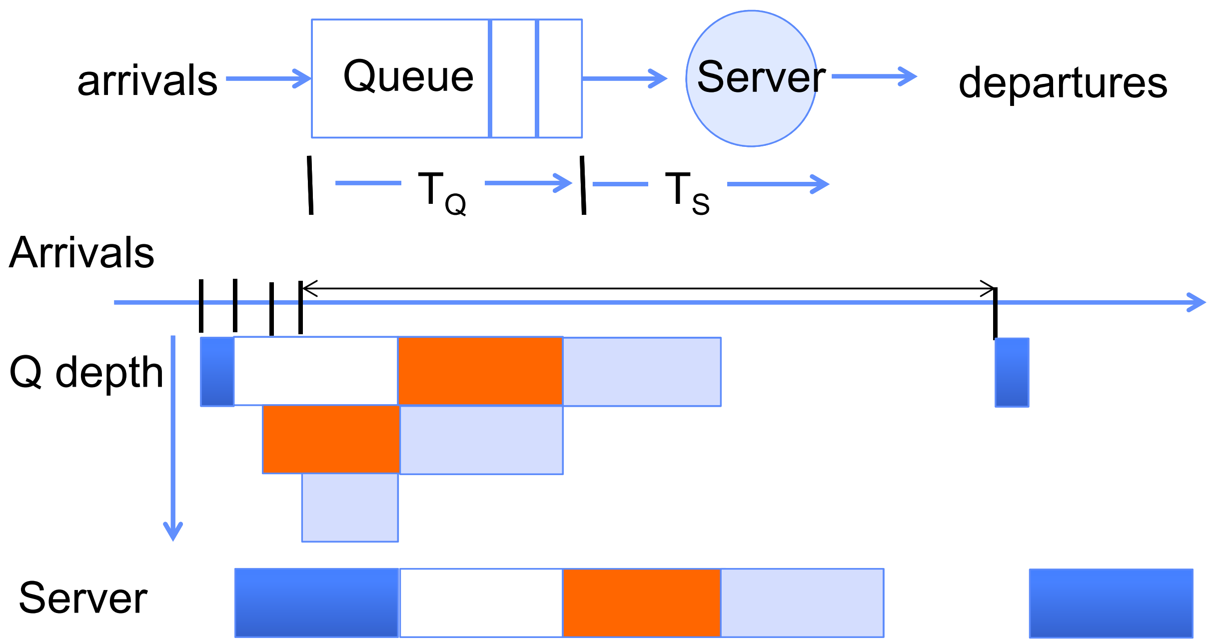
- Assume requests arrive at regular intervals, take a fixed time to process, with plenty of time between …
- Service rate ($\mu = 1/T_S$) - operations per second
- Arrival rate: ($\lambda = 1/T_A$) - requests per second
- Utilization: $U = \lambda/\mu$, where $\lambda < \mu$
- Average rate is the complete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

A Ideal Linear World



- What does the queue wait time look like?
  - Grows unbounded at a rate ~ (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 low

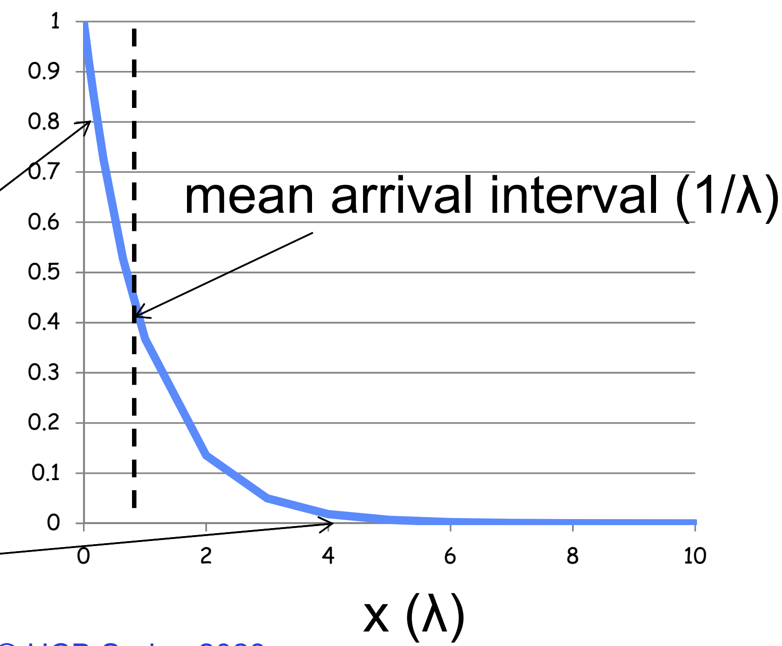
So how do we model the burstiness of arrival?

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

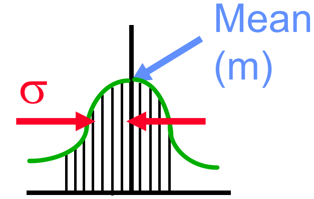
Likelihood of an event occurring is independent of how long we've been waiting

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

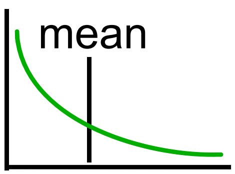
Few long gaps (i.e., low instantaneous rate)



Background: General Use of Random Distributions

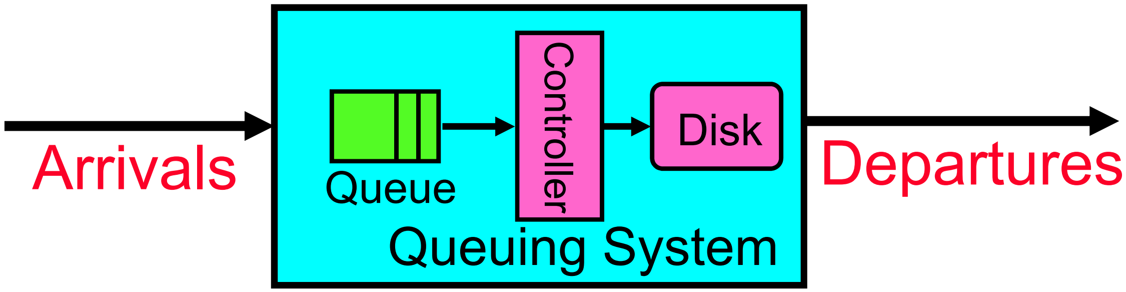


- Server spends variable time (T) with customers
  - Mean (Average) $m = \Sigma p(T) \times T$
  - Variance (stddev²) $\sigma^2 = \Sigma p(T) \times (T-m)^2 = \Sigma p(T) \times T^2 - m^2$
  - Squared coefficient of variance: $C = \sigma^2/m^2$
    Aggregate description of the distribution



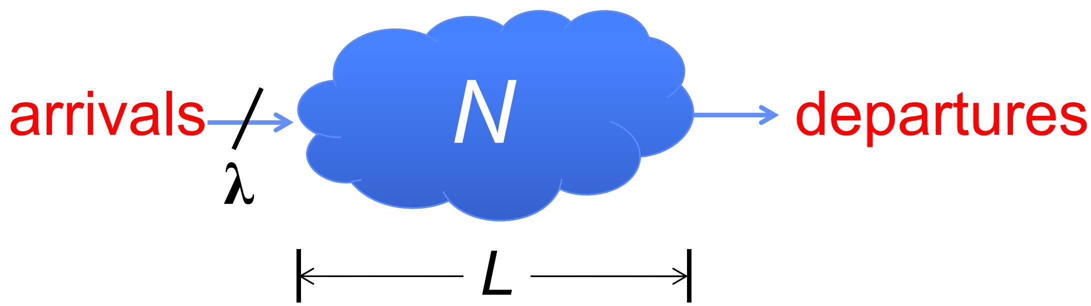
- Important values of C:
  - No variance or deterministic ⇒ C=0
  - "Memoryless" or exponential ⇒ C=1
    - » Past tells nothing about future
    - » Poisson process – *purely* or *completely* random process
    - » Many complex systems (or aggregates) are well described as memoryless
  - Disk response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

Introduction to Queuing Theory



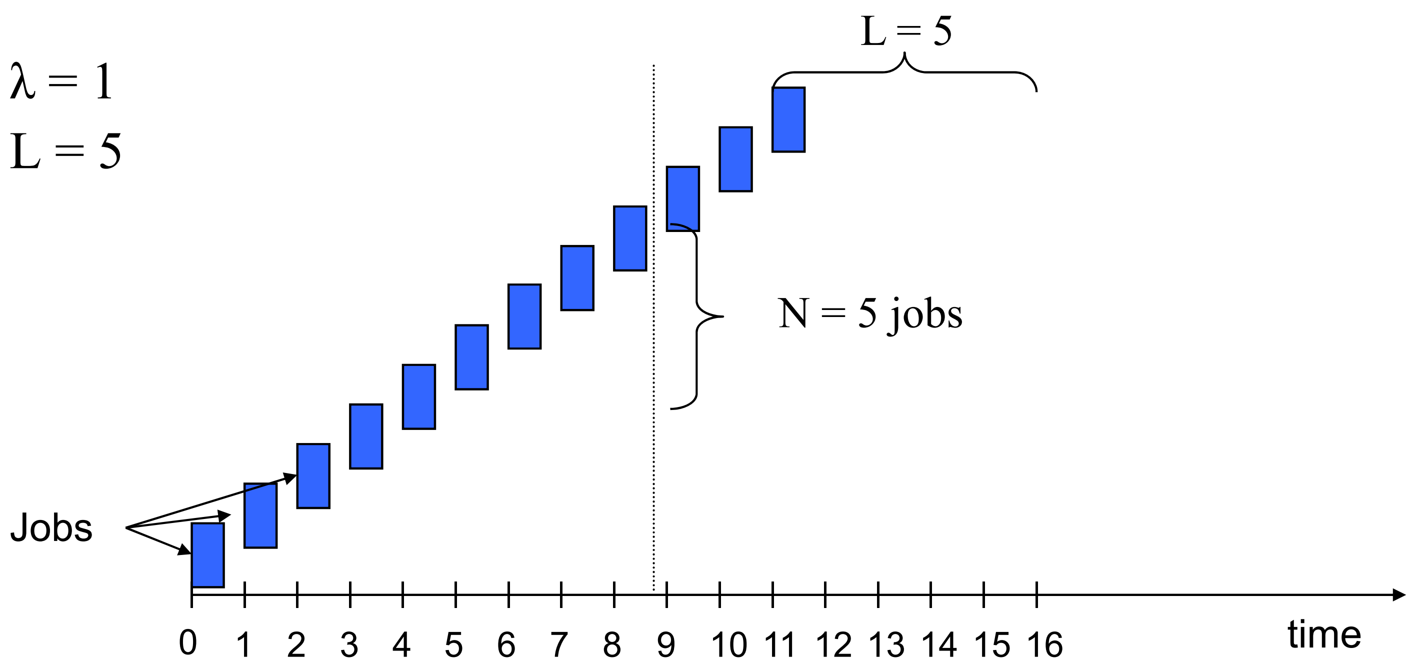
- What about queuing time??
  - Let's apply some queuing theory
  - Queuing Theory applies to long term, steady state behavior $\Rightarrow$ Arrival rate = Departure rate
- Arrivals characterized by some probabilistic distribution
- Departures characterized by some probabilistic distribution

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) x L (s)*
- Regardless of structure, bursts of requests, variation in service
  - Instantaneous variations, but it washes out in the average
  - Overall, requests match departures

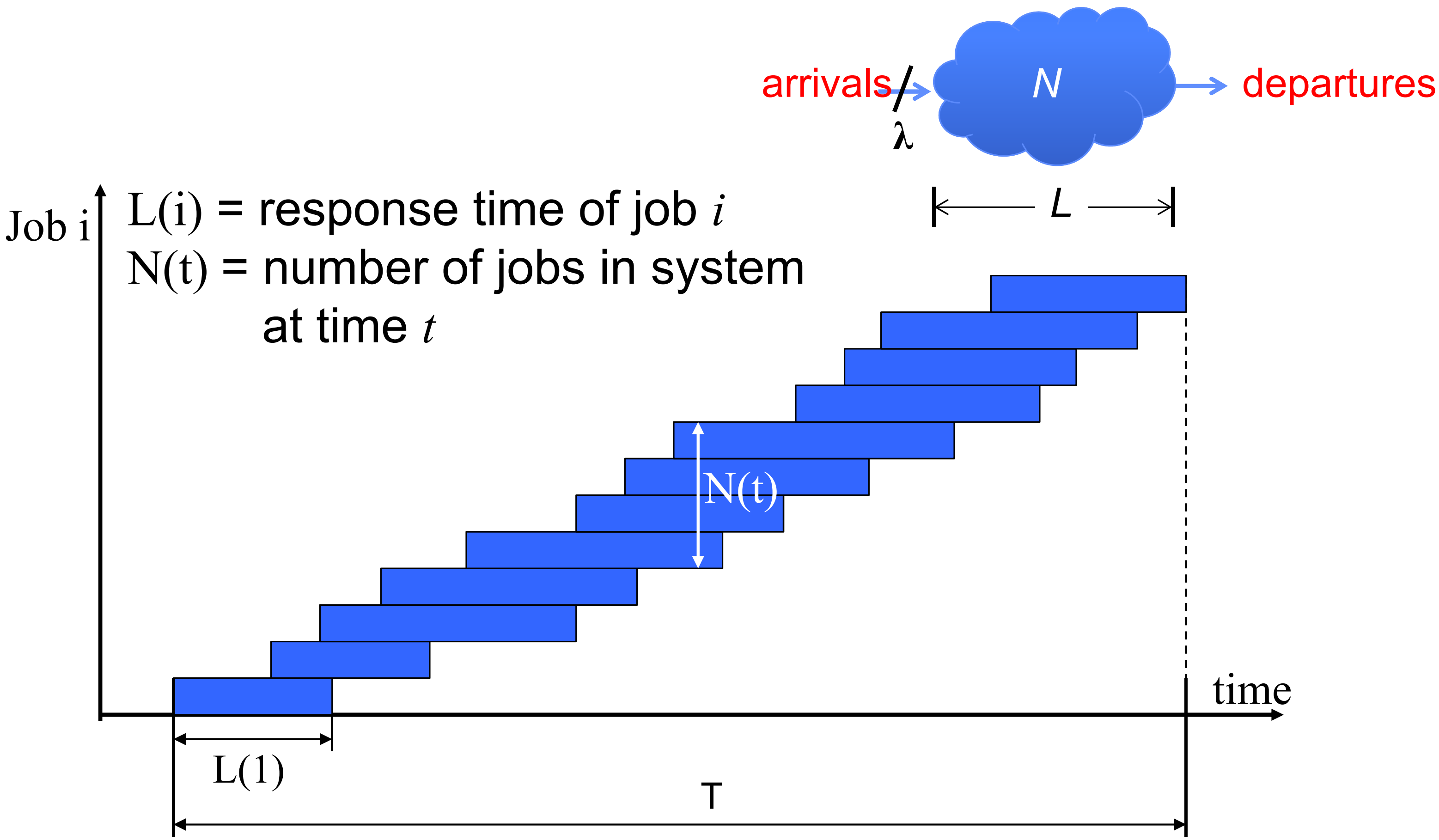
Example



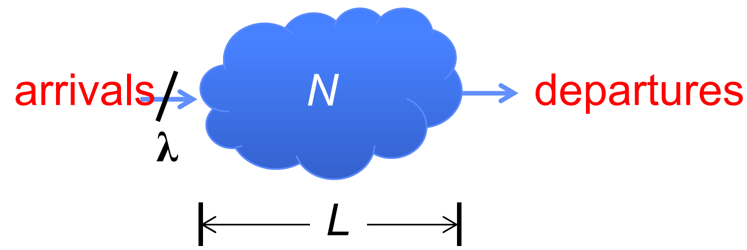
A: $N = \lambda \times L$

- E.g., $N = \lambda \times L = 5$

Little's Theorem: Proof Sketch

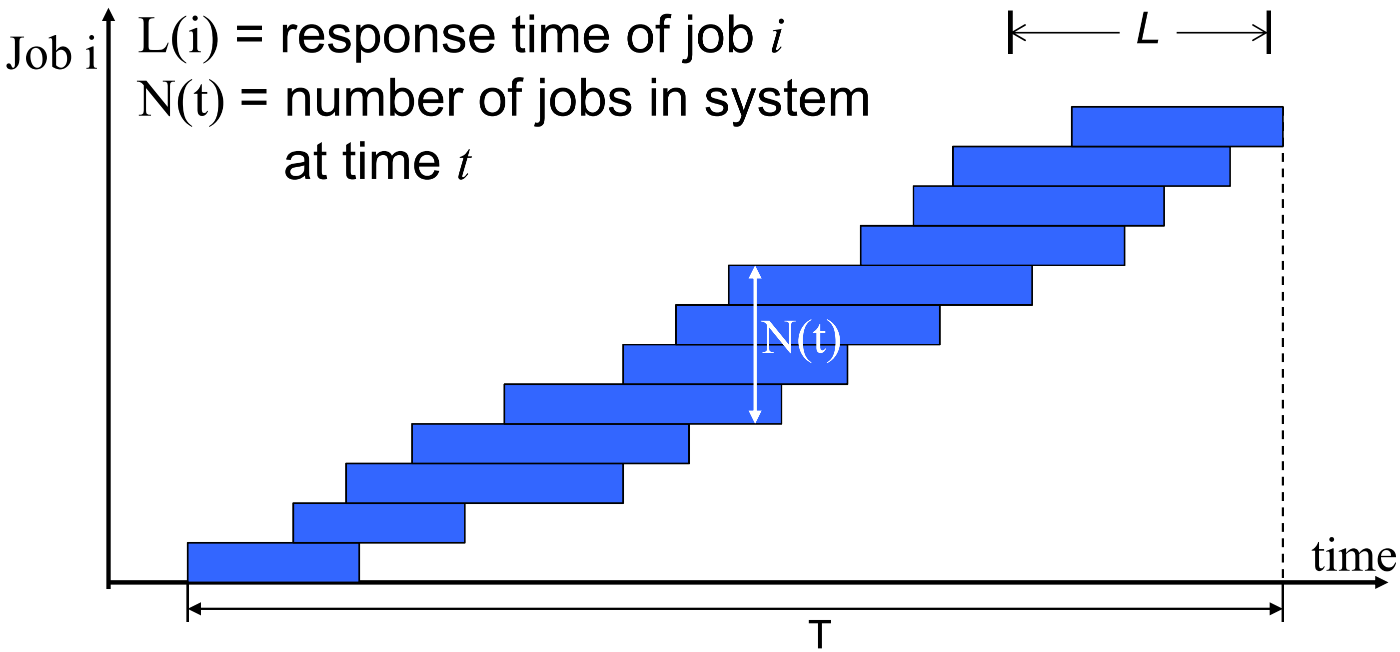


Little's Theorem: Proof Sketch



L(i) = response time of job *i*

N(t) = number of jobs in system at time *t*



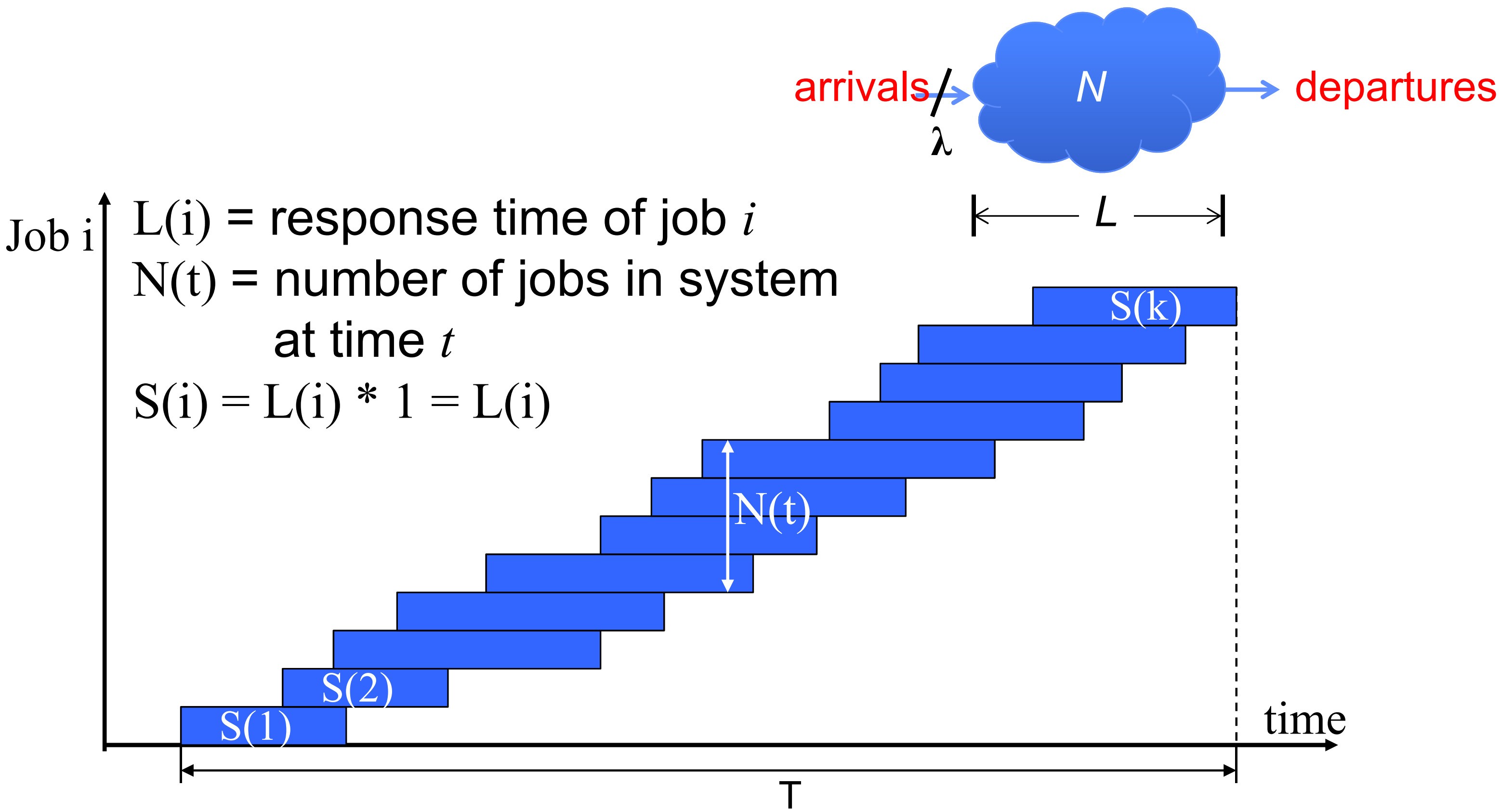
- What is the system occupancy, i.e., average number of jobs in the system?

Little's Theorem: Proof Sketch

L(i) = response time of job *i*

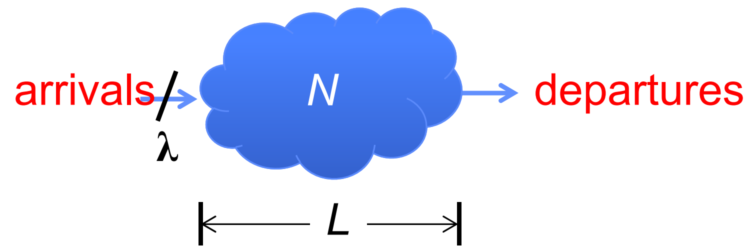
N(t) = number of jobs in system at time *t*

S(i) = L(i) * 1 = L(i)



$$S = S(1) + S(2) + \ldots + S(k) = L(1) + L(2) + \ldots + L(k)$$

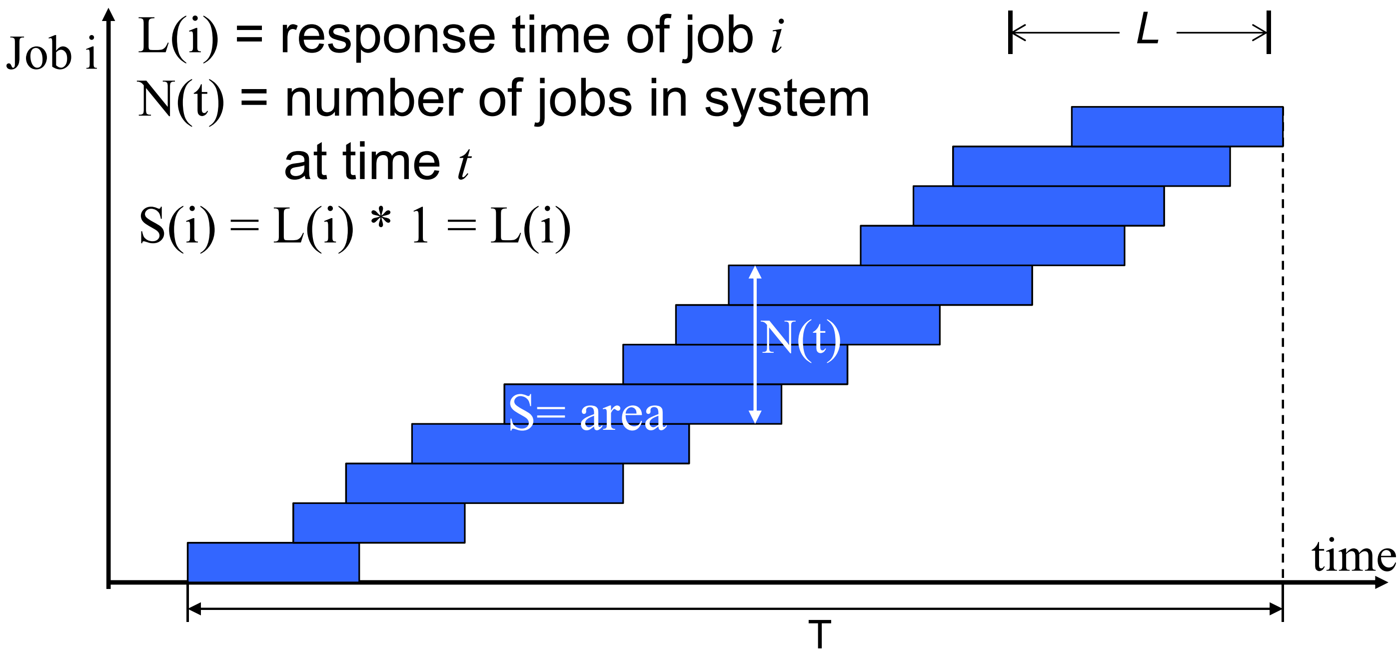
Little's Theorem: Proof Sketch



L(i) = response time of job *i*

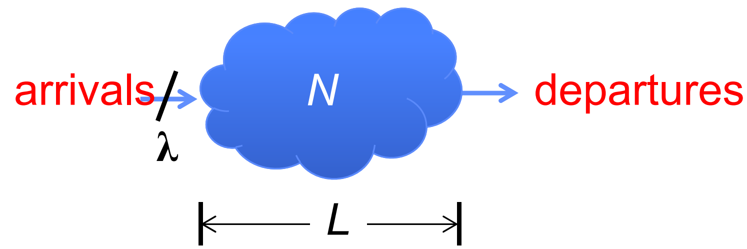
N(t) = number of jobs in system at time *t*

S(i) = L(i) * 1 = L(i)



Average occupancy (N_{avg}) = S/T

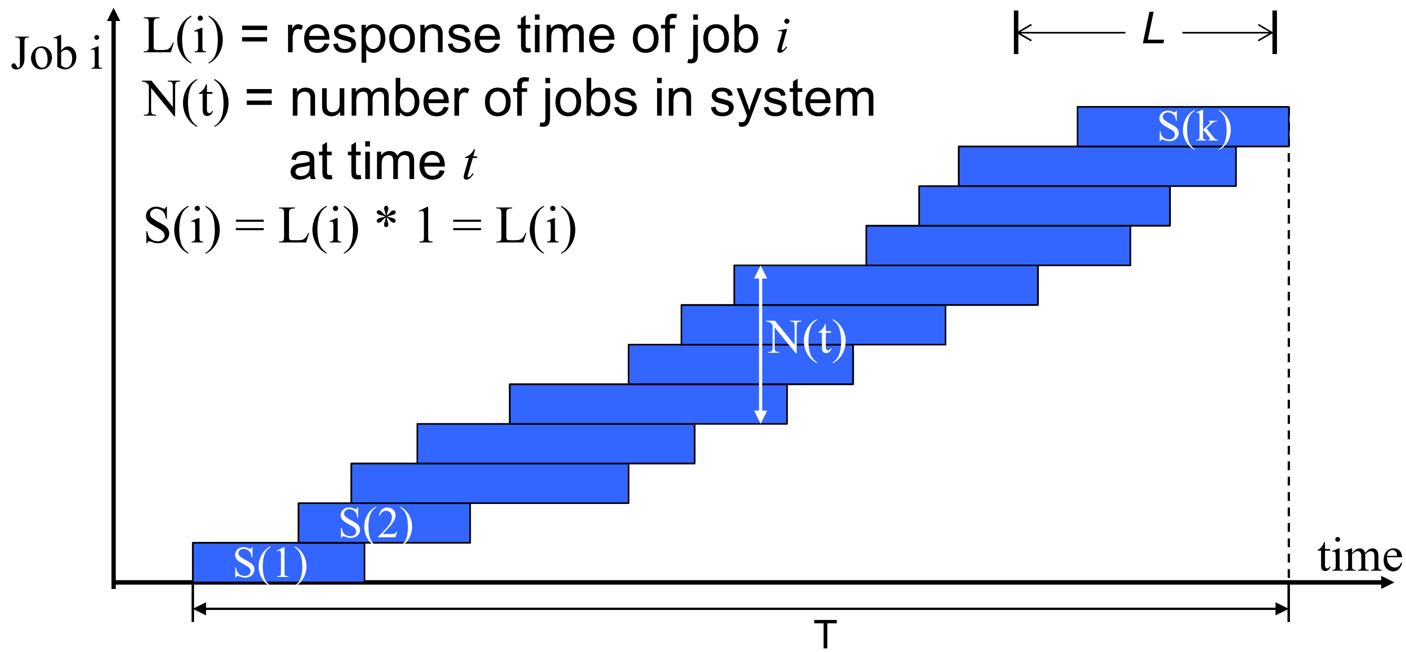
Little's Theorem: Proof Sketch



L(i) = response time of job *i*

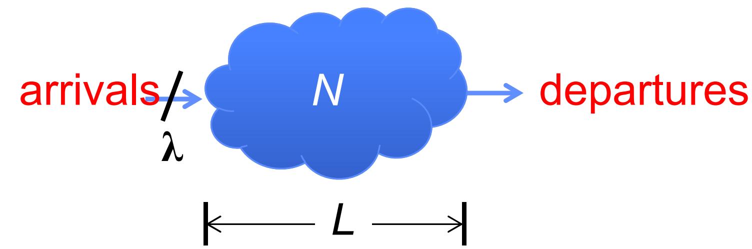
N(t) = number of jobs in system at time *t*

S(i) = L(i) * 1 = L(i)



$$N_{avg} = S/T = (L(1) + \ldots + L(k))/T$$

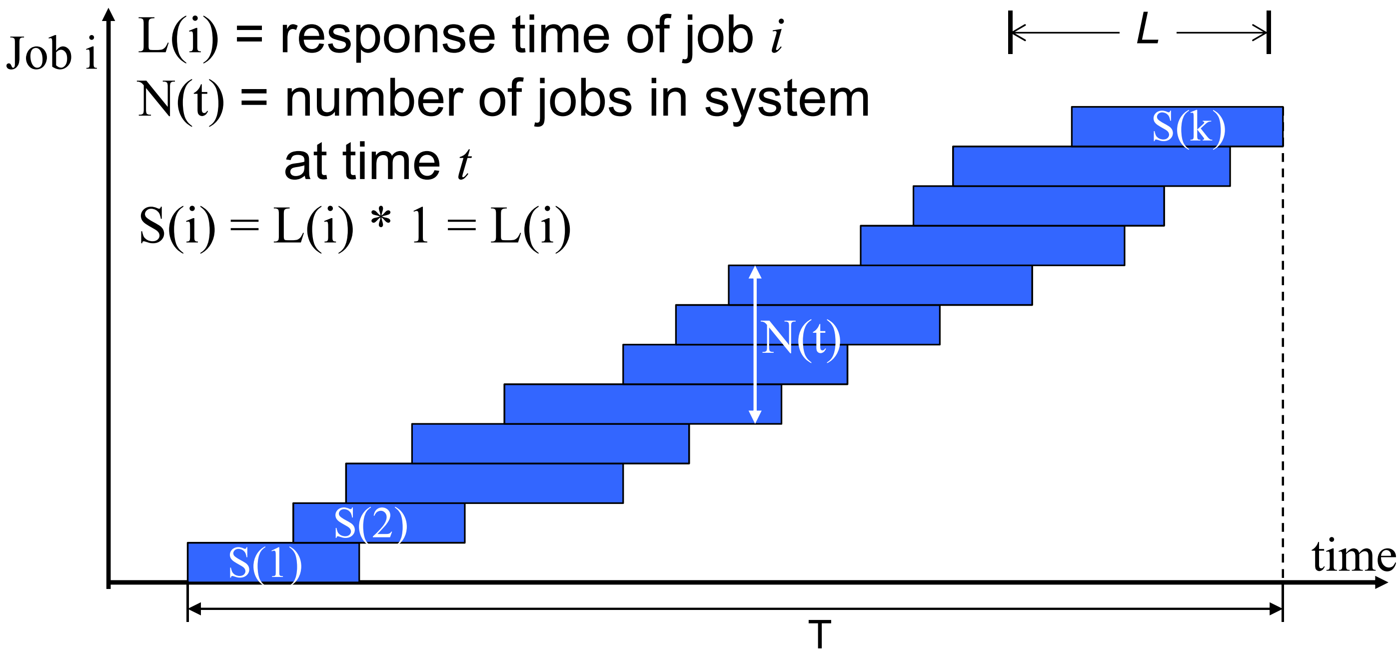
Little's Theorem: Proof Sketch



L(i) = response time of job i

N(t) = number of jobs in system at time t

S(i) = L(i) * 1 = L(i)



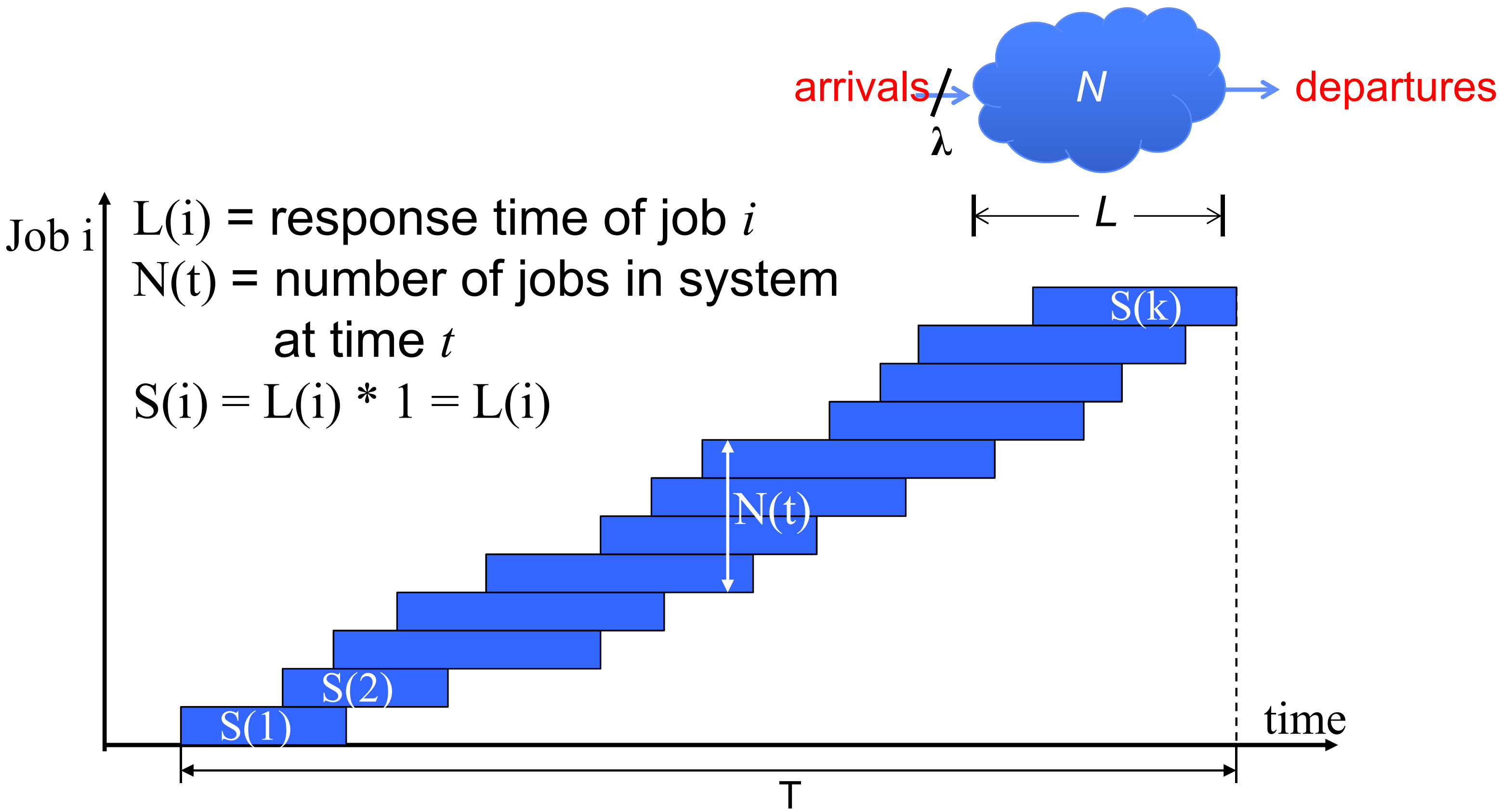
$$N_{avg} = (L(1) + \ldots + L(k))/T = (N_{total}/T)*(L(1) + \ldots + L(k))/N_{total}$$

Little's Theorem: Proof Sketch

L(i) = response time of job *i*

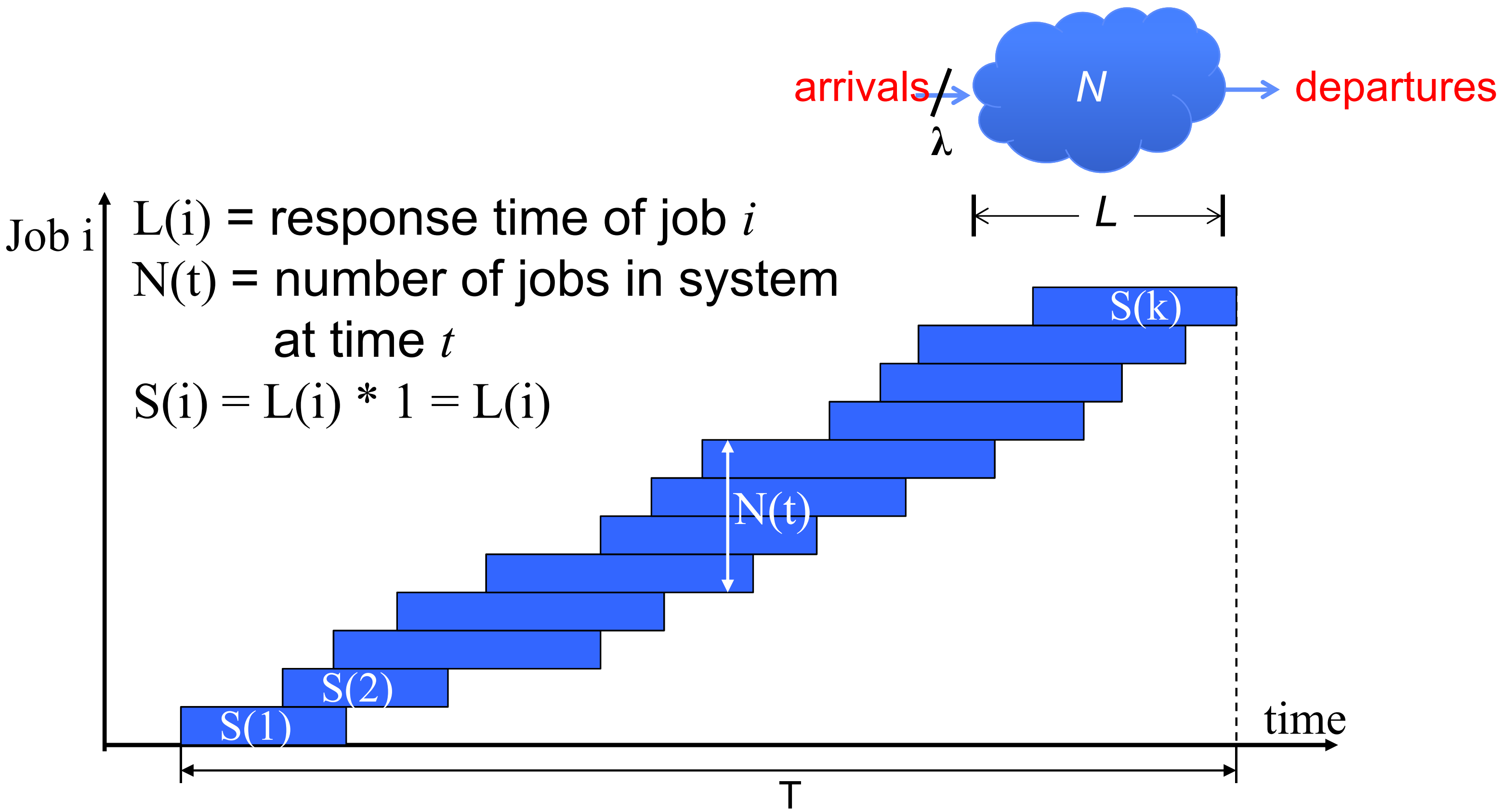
N(t) = number of jobs in system at time *t*

S(i) = L(i) * 1 = L(i)



$$N_{avg} = (N_{total}/T)*(L(1) + \ldots + L(k))/N_{total} = \lambda_{avg} \times L_{avg}$$

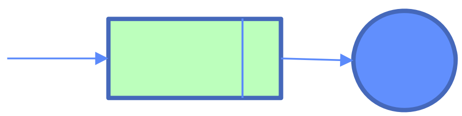
Little’s Theorem: Proof Sketch



$$N_{avg} = \lambda_{avg} \times L_{avg}$$

Little's Law Applied to a Queue

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



Average Arrival Rate

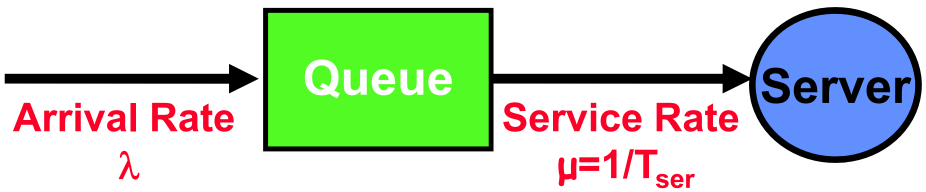
$$L_Q = \lambda T_Q$$

Average length of the queue

Average time "waiting" in queue

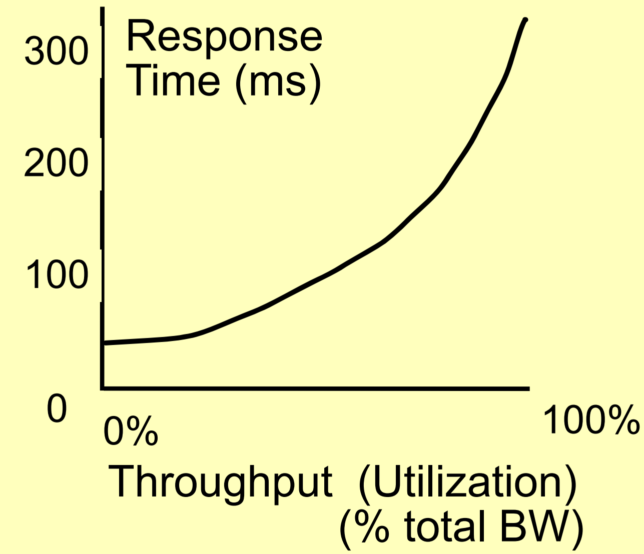
A Little Queuing Theory: Computing T_Q

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

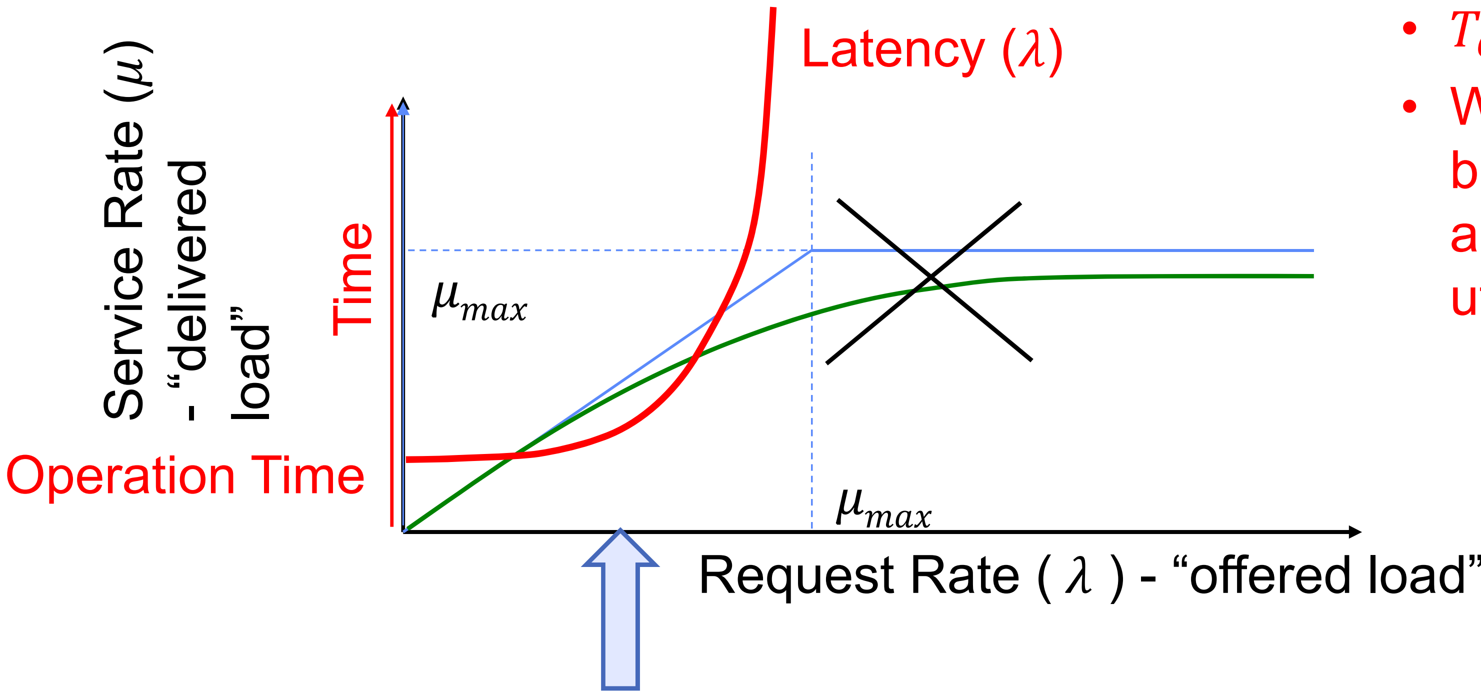


- Parameters that describe our system:
  - λ: mean number of arriving customers/seco
  - T_{ser}: mean time to service a customer (“m
  - C: squared coefficient of variance
  - μ: service rate = $1/T_{ser}$
  - u: server utilization ($0 \leq u$): $u = \lambda/$ $\lambda \times T_s$
- Results:
  - Memoryless service distribution (C = 1): (an “M/M/1 queue”):
    - » $T_q = T_{ser} \times u/(1 - u)$
  - General service distribution server (an “M/G/1 queue”):
    - » $T_q = T_{ser} \times \frac{1}{2}(1+C) \times u/(1 - u)$

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



System Performance In presence of a Queue

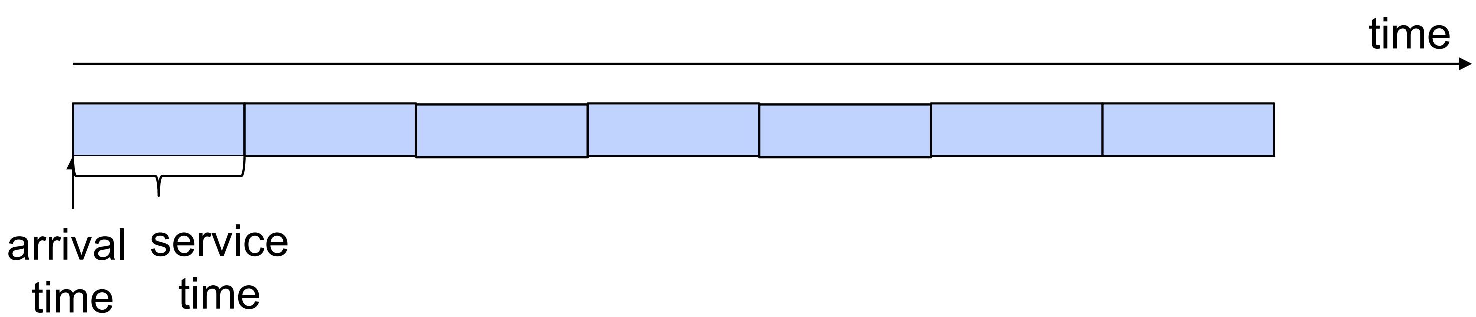


- $T_Q \sim \frac{u}{1-u}$, $u = \lambda/\mu_{max}$
- Why does latency blow up as we approach 100% utilization?
  - Queue builds up on each burst
  - But very rarely (or never) gets a chance to drain

"Half-Power Point" : load at which system delivers half of peak performance
- Design and provision systems to operate roughly in this regime
- Latency low and predictable, utilization good: ~50%

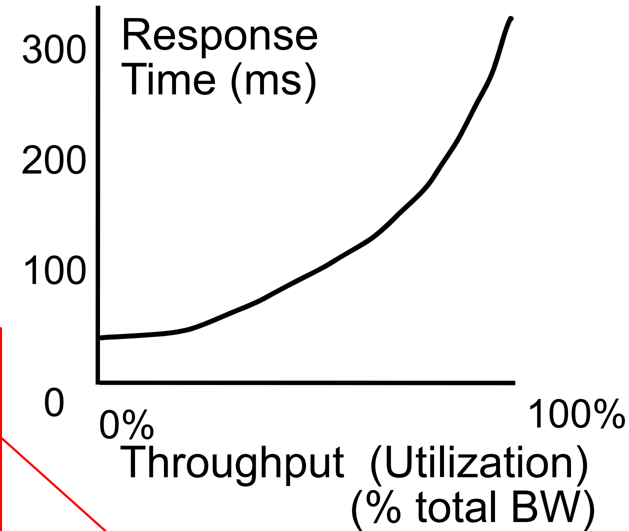
Why unbounded response time?

- Assume deterministic arrival process and service time
  – Possible to sustain utilization = 1 with bounded response time!

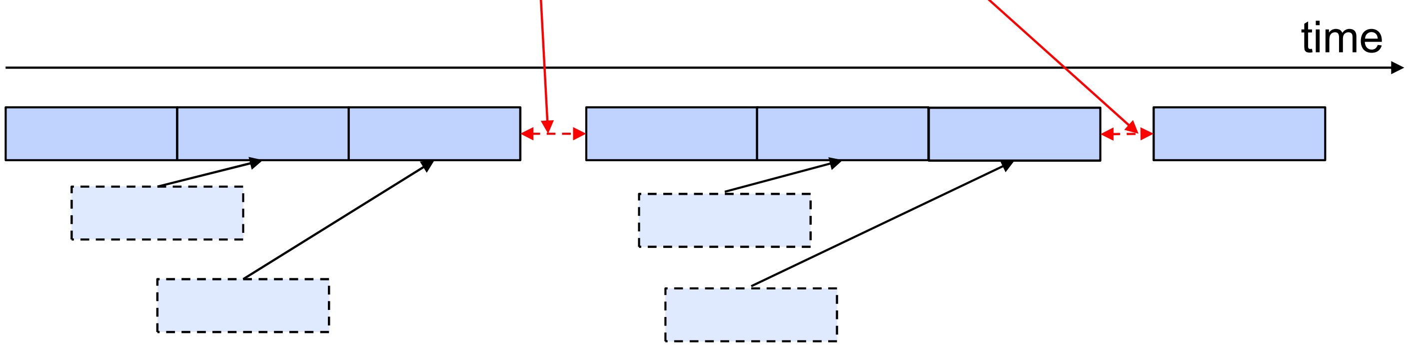


Why unbounded response time?

- Assume stochastic arrival process (and service time)
  - No longer possible to achieve utilization = 1



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



A Little Queuing Theory: An Example

- 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+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:

  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/customer 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 time/customer in system) =$T_q + T_{ser}$= 25 ms

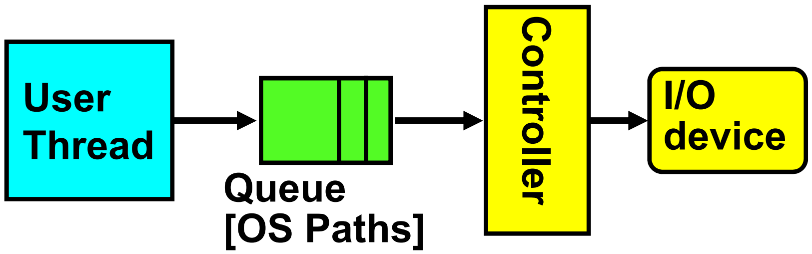
Queuing Theory Resources

- 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

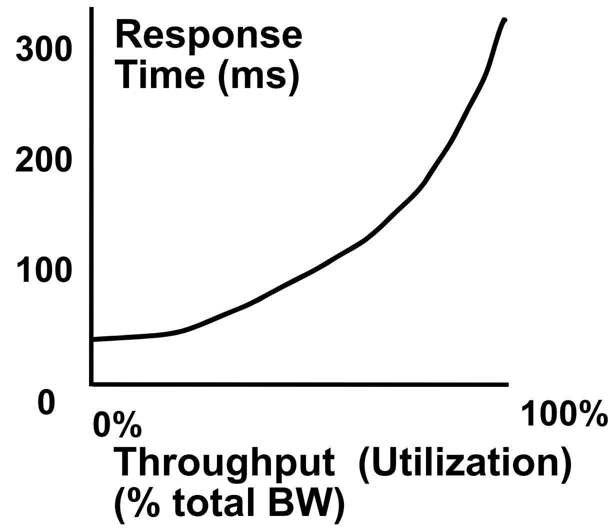
- Some previous midterms with queueing theory questions

- 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

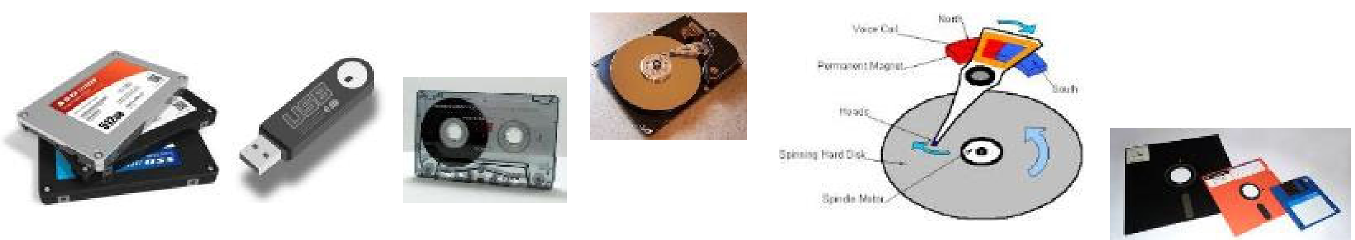
Application / Service

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

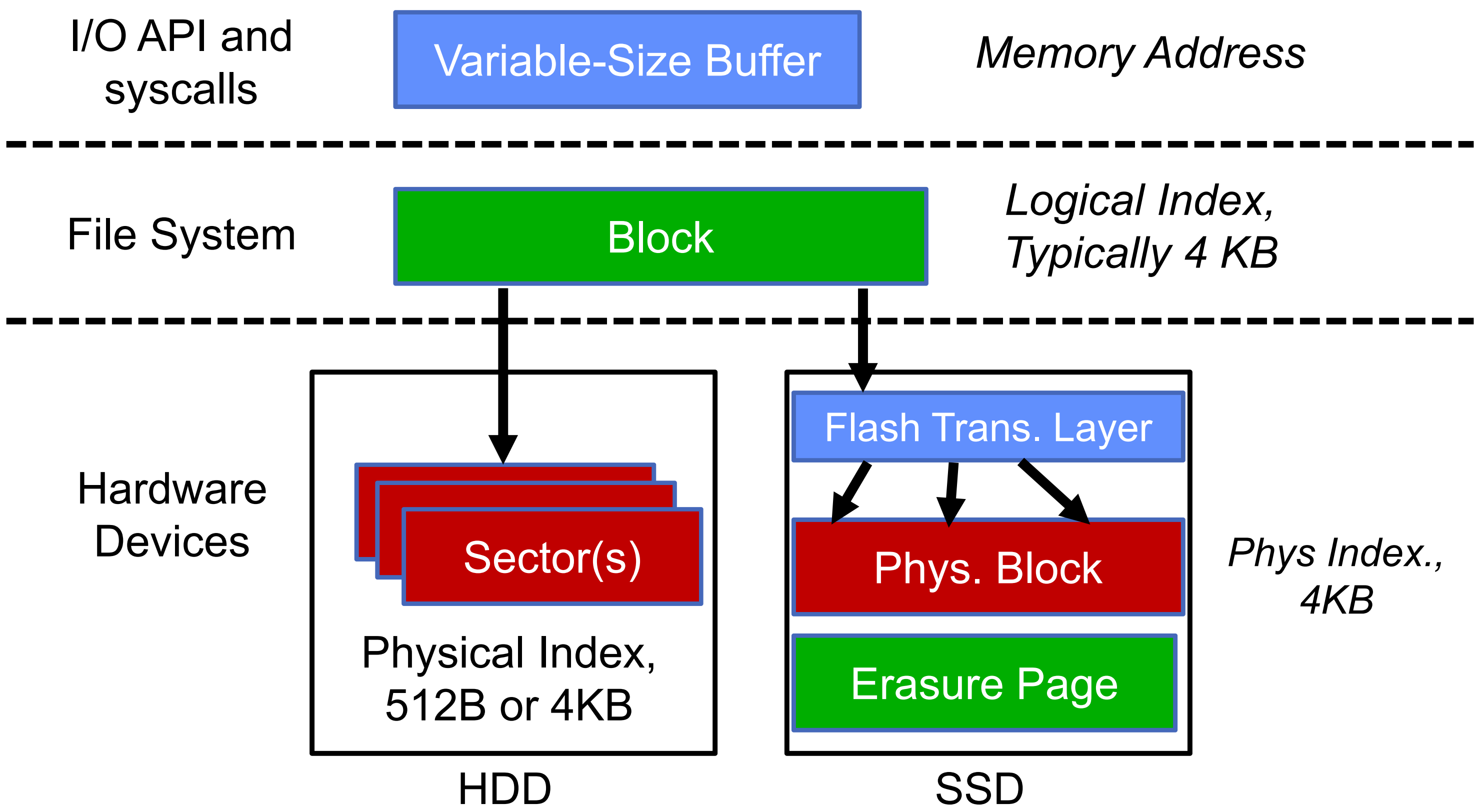
What we covered in Lecture 4

What we will cover next…

What we just covered…



From Storage to File Systems



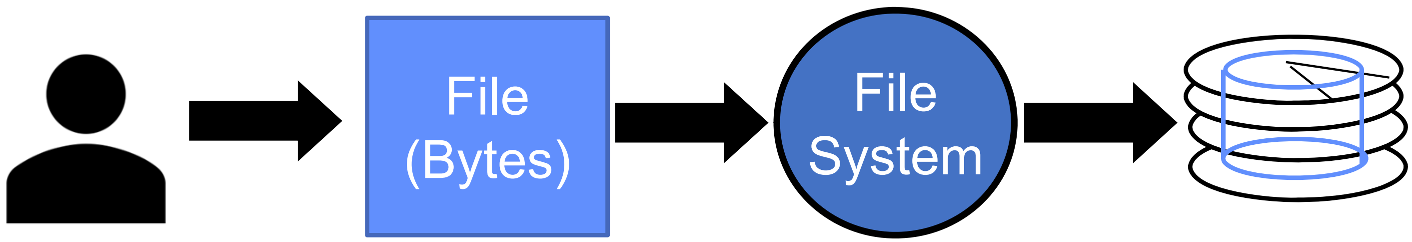
Building a File System

- 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
  – 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 ≥ 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 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

- Somewhat different from 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

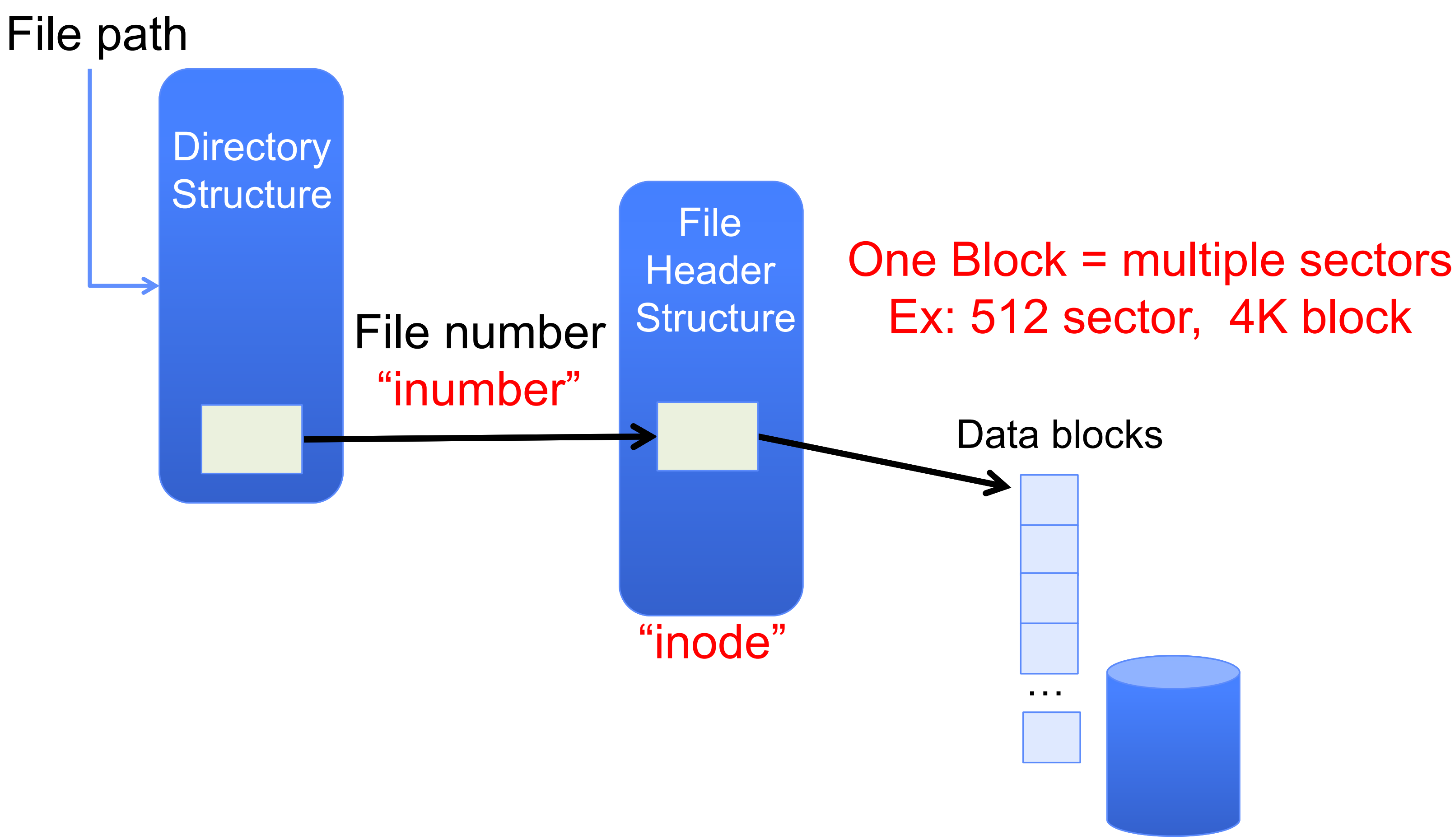
- Durability
  - Ideally, file system is in meaningful state upon shutdown
  - This obviously isn't always the case…

FILE SYSTEM DESIGN

Critical Factors in File System Design

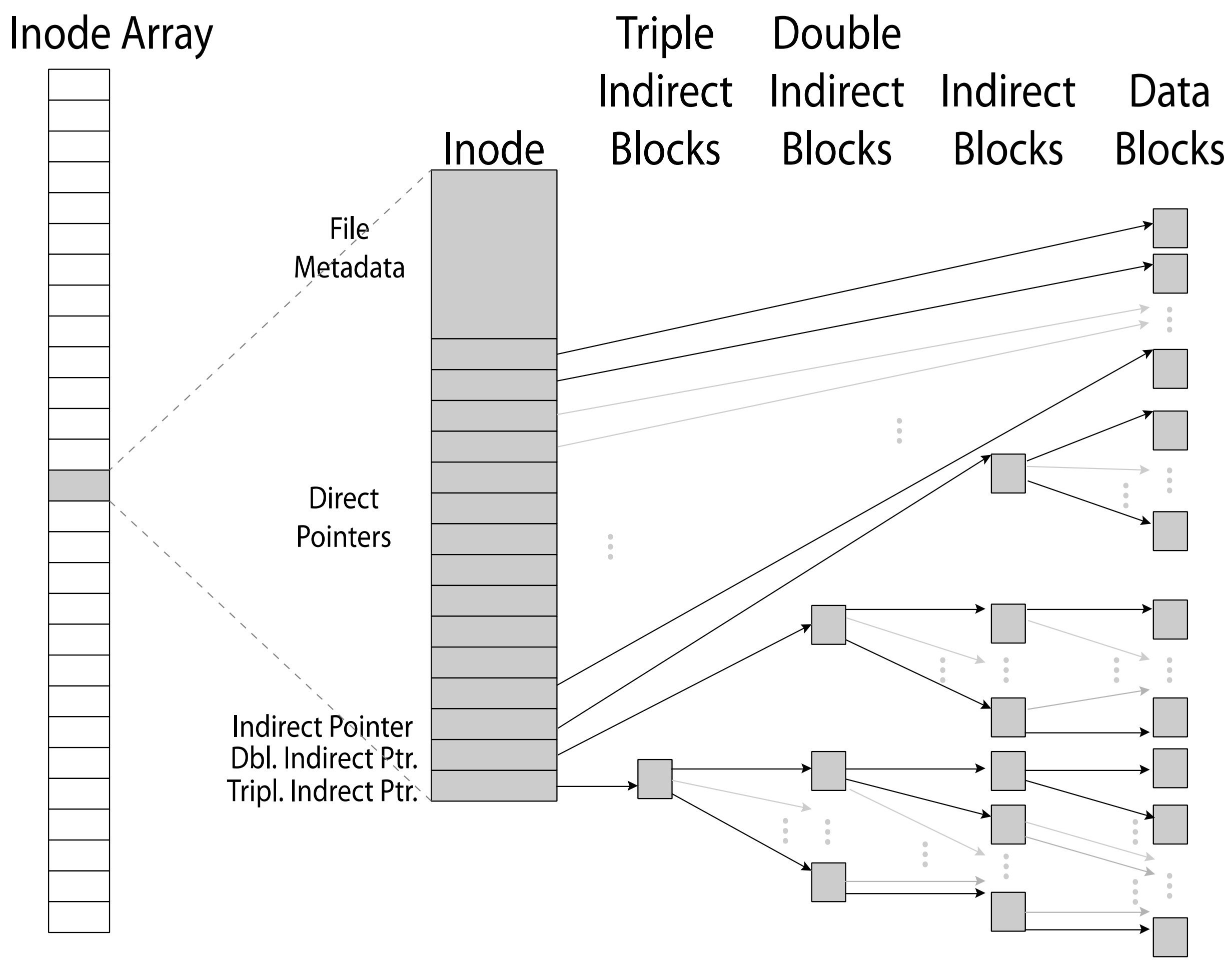
- (Hard) Disks Performance !!!
  – Maximize sequential access, minimize seeks
- Open before Read/Write
  – Can perform protection checks and look up where the actual file resource are, in advance
- Size is determined as they are used !!!
  – Can write (or read zeros) to expand the file
  – Start small 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

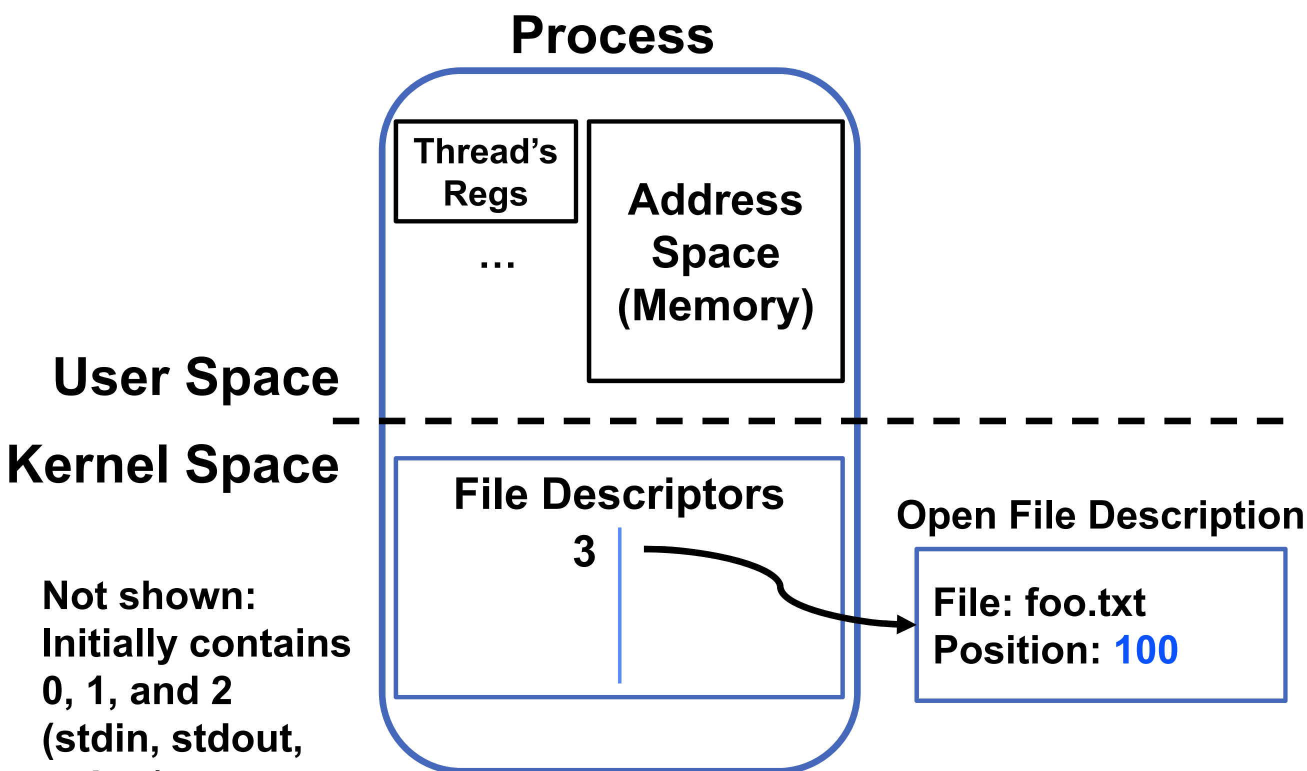


Example of BSD/Linux-like Inode structure

- Sample file in multilevel indexed format:
  - Data Stored in Blocks
  - 10 direct ptrs, 1K blocks
  - One Indirect block
  - One Doubly-indirect Block
  - One Triply-indirect Block
- Example accesses
  - How many accesses for block #23? (assume file header accessed on open)?
    » Two: One for indirect block, one for data
  - How about block #5?
    » One: One for data
  - Block #340?
    » Three: double indirect block, indirect block, and data



Recall: Abstract Representation of a Process

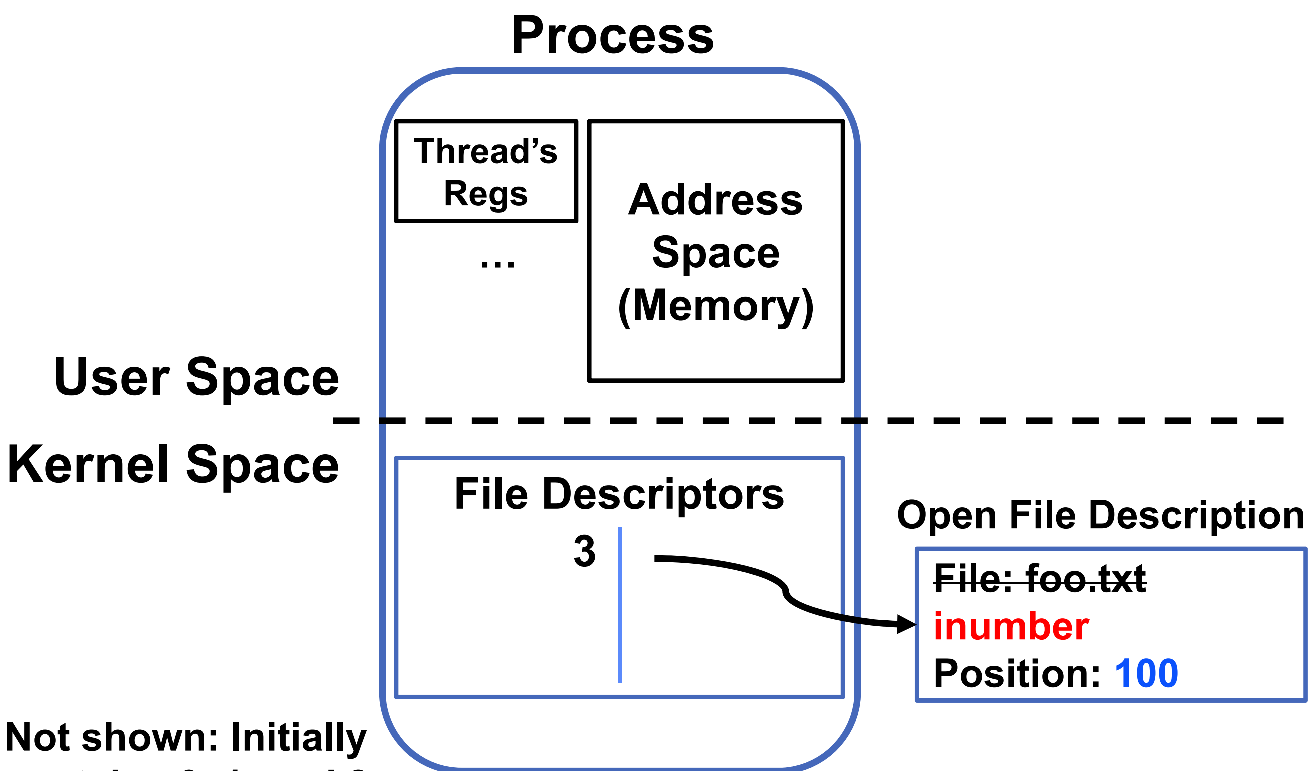


Not shown: Initially contains 0, 1, and 2 (stdin, stdout, stderr)

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

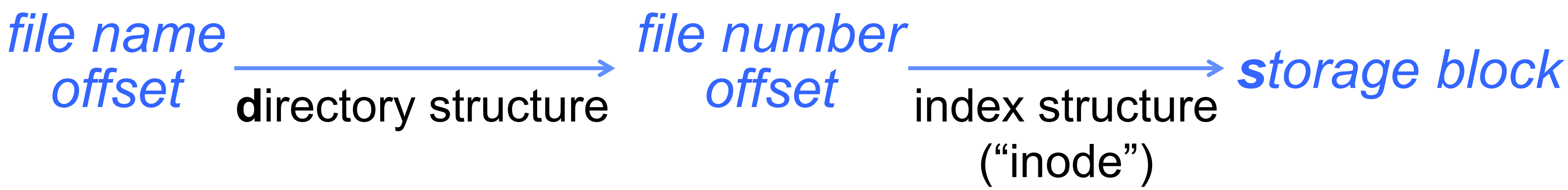
Components of a File System



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



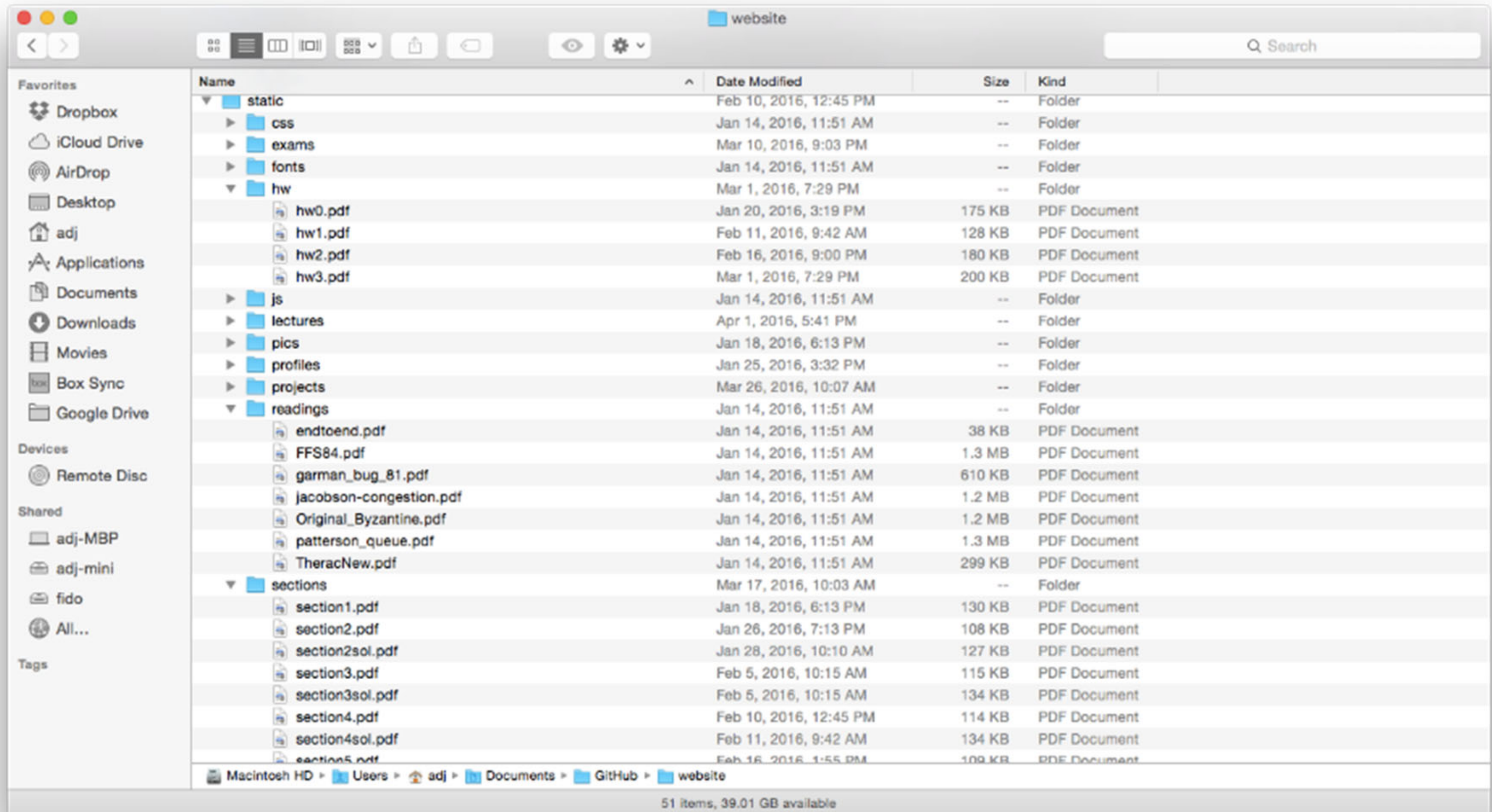
- Open performs *Name Resolution*
  - Translates path name into a "file number"
- Read and Write operate on the file number
  - Use file number as an "index" to locate the blocks

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

How to get the File Number?

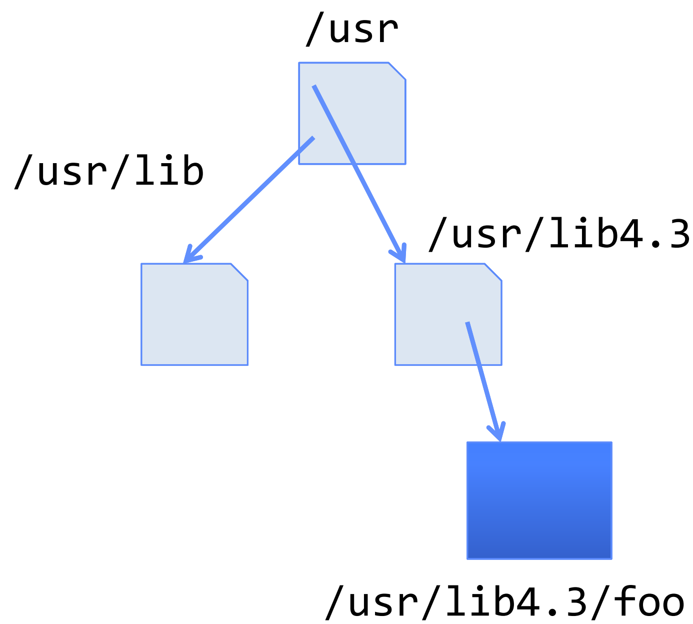
- Look up in ***directory structure***
- A directory is a file containing <file_name : file_number> mappings
  - File number could be a file or another directory
  - Operating system stores the mapping in the directory in a format it interprets
  - Each <file_name : file_number> mapping is called a directory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

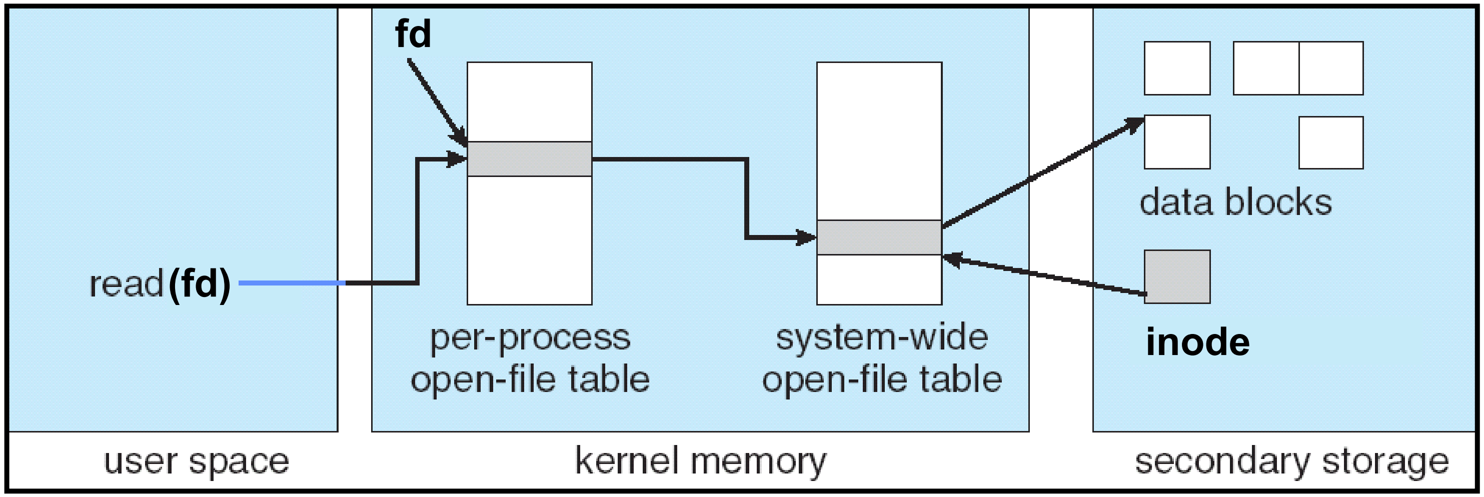


- libc support
  - `DIR * opendir (const char *dirname)`
  - `struct dirent * readdir (DIR *dirstream)`
  - `int readdir_r (DIR *dirstream, struct dirent *entry, struct dirent **result)`

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

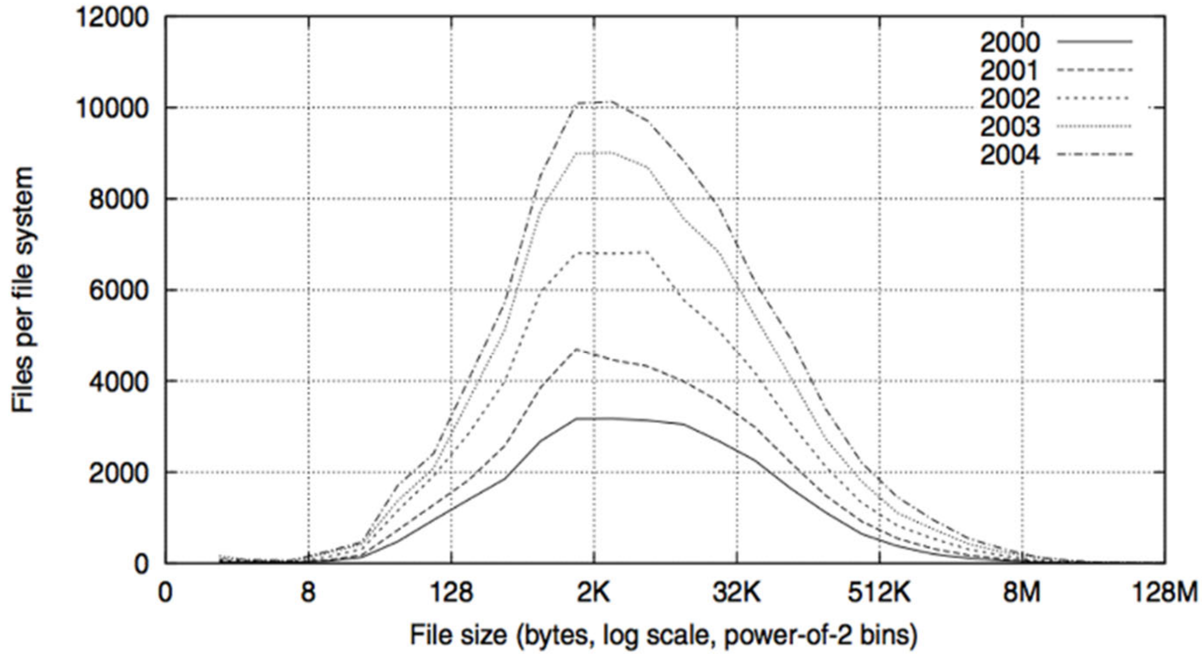


Fig. 2. Histograms of files by size.

Observation #2: Most Bytes are in Large Files

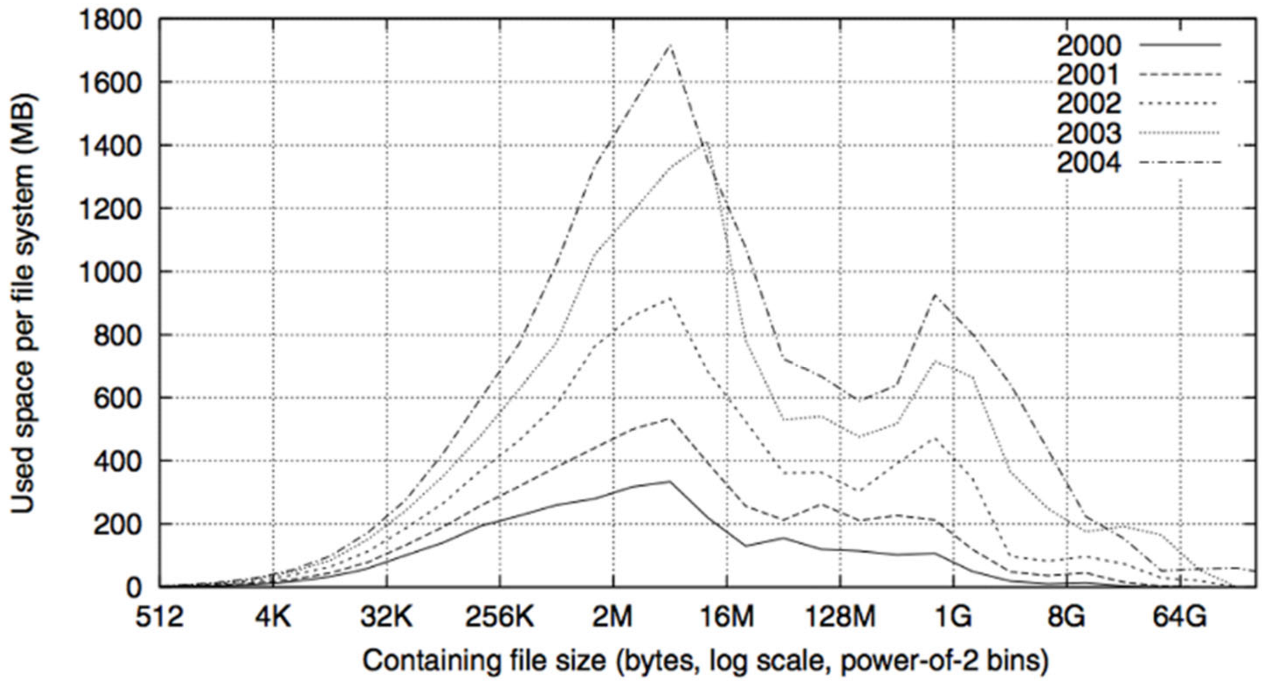


Fig. 4. Histograms of bytes by containing file size.

Conclusion

- 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