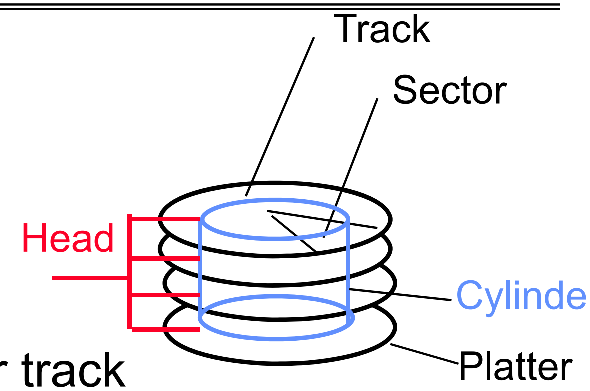
# CS162
# Operating Systems and Systems Programming
# Lecture 19

# Filesystems 1: Performance (Con't), Queueing Theory, Filesystem Design

April 5th, 2022

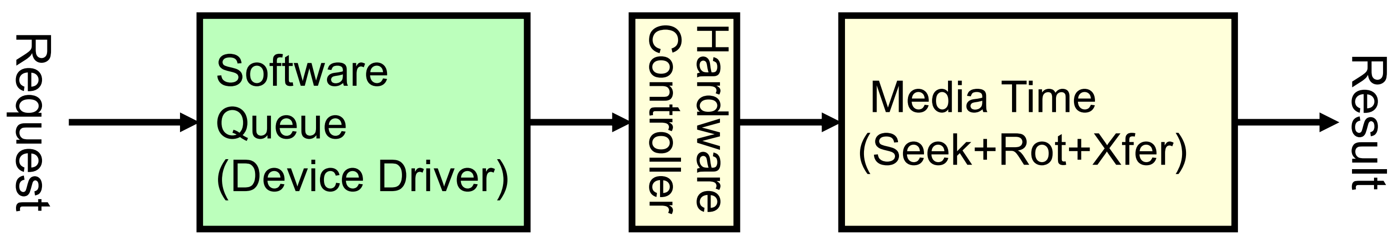
Prof. Anthony Joseph and John Kubiatowicz
http://cs162.eecs.Berkeley.edu

# 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

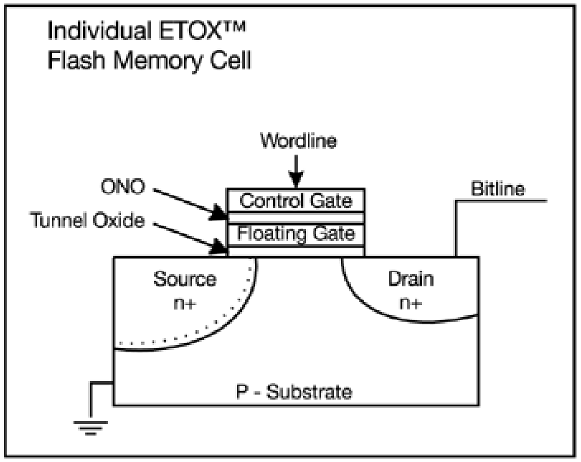
**Disk Latency = Queueing Time + Controller time + Seek Time + Rotation Time + Xfer Time**



# Recall: Typical Numbers for Magnetic Disk

| Parameter | Info/Range |
|---|---|
| Space/Density | Space: 18TB (Seagate), 9 platters, in 3½ inch form factor!<br>**Areal Density: ≥ 1 Terabit/square inch! (PMR, Helium, …)** |
| Average Seek Time | Typically 4-6 milliseconds |
| Average Rotational Latency | Most laptop/desktop disks rotate at 3600-7200 RPM (16-8 ms/rotation). Server disks up to 15,000 RPM.<br>Average latency is halfway around disk so 4-8 milliseconds |
| Controller Time | Depends on controller hardware |
| Transfer Time | Typically 50 to 250 MB/s. Depends on:<br>• Transfer size (usually a sector): 512B – 1KB per sector<br>• Rotation speed: 3600 RPM to 15000 RPM<br>• Recording density: bits per inch on a track<br>• Diameter: ranges from 1 in to 5.25 in |
| Cost | Used to drop by a factor of two every 1.5 years (or faster), now slowing down |

# 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)

# Recall: SSD Summary

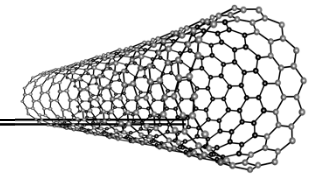
- Pros (vs. hard disk drives):
  - Low latency, high throughput (eliminate seek/rotational delay)
  - No moving parts:
    - » Very light weight, low power, silent, very shock insensitive
  - Read at memory speeds (limited by controller and I/O bus)
- Cons
  - Small storage (0.1-0.5x disk), expensive (3-20x disk)
    - » Hybrid alternative: combine small SSD with large HDD

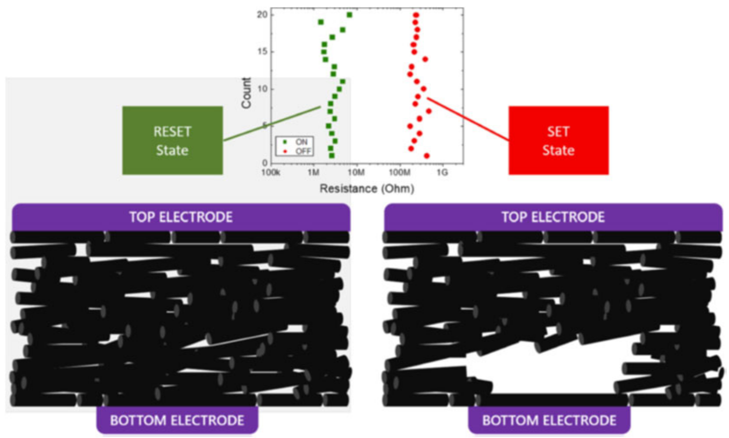
# Recall: SSD Summary

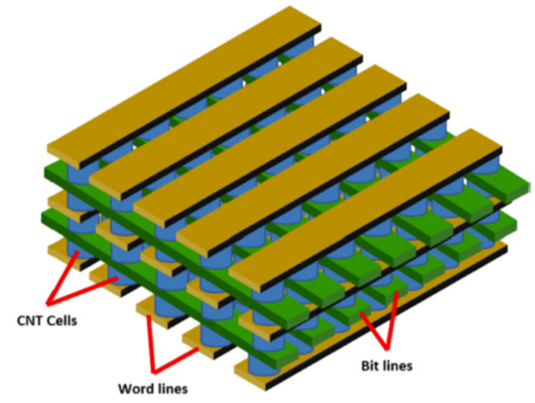
- Pros (vs. hard disk drives):
  - Low latency, high throughput (eliminate seek/rotational delay)
  - No moving parts:
    - » Very light weight, low power, silent, very shock insensitive
  - Read at memory speeds (limited by controller and I/O bus
- Cons
  - ~~Small storage (0.1-0.5x disk), expensive (3-20x disk)~~
    - » Hybrid alternative: combine small SSD with large HDD
  - Asymmetric block write performance: read pg/erase/write pg
    - » Controller garbage collection (GC) algorithms have major effect on performance
  - Limited drive lifetime
    - » 1-10K writes/page for MLC NAND
    - » Avg failure rate is 6 years, life expectancy is 9–11 years
- These are changing rapidly!

No longer true!

# Nano-Tube Memory (NANTERO)






Crosspoint

- Yet another possibility: Nanotube memory
  - NanoTubes between two electrodes, slight conductivity difference between ones and zeros
  - No wearout!
- Better than DRAM?
  - Speed of DRAM, no wearout, non-volatile!
  - Nantero promises 512Gb/dice for 8Tb/chip!  (with 16 die stacking)

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

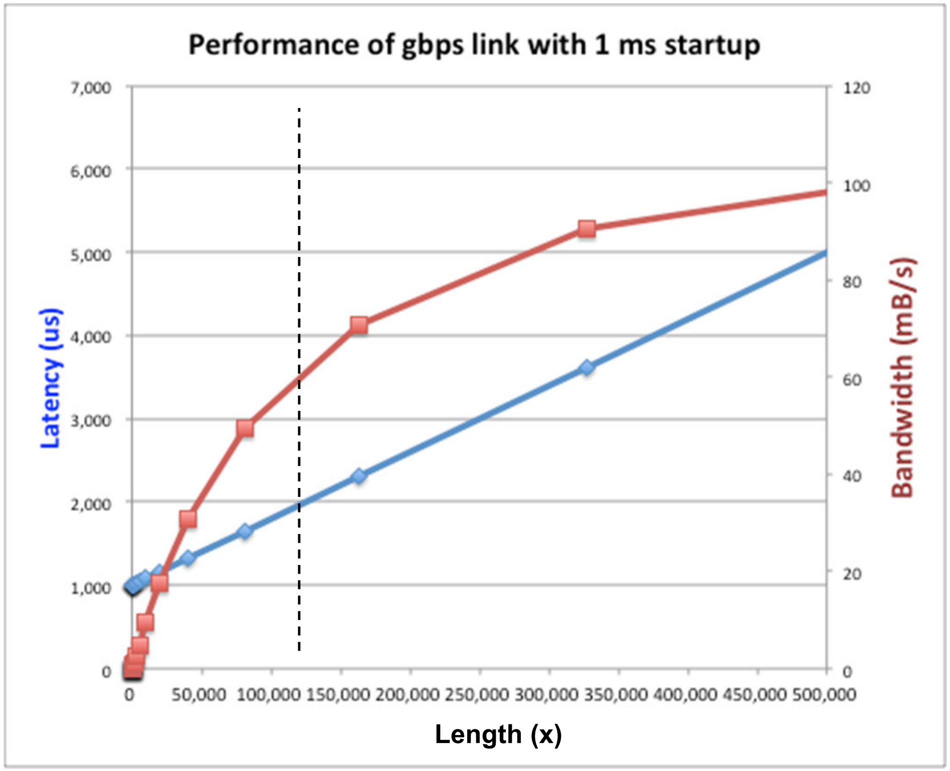
- ***Latency*** – time to complete a task
  - Measured in units of time (s, ms, us, …, hours, years)
- ***Response Time*** - time to initiate and operation and get its response
  - Able to issue one that *depends* on the result
  - Know that it is done (anti-dependence, resource usage)
- ***Throughput*** 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:

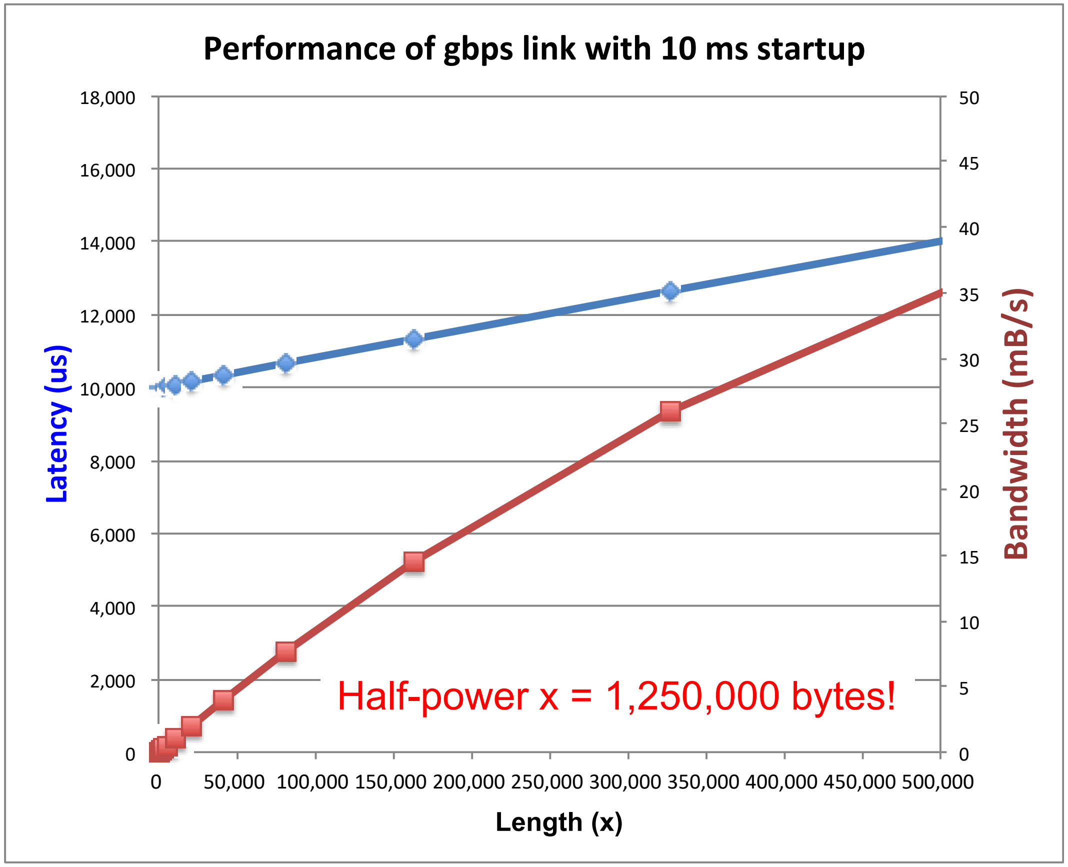
$$E(x) = \frac{x}{S + \frac{x}{B_w}} = \frac{B_w \cdot x}{B_w \cdot S + x} = \frac{B_w}{\frac{B_w \cdot S}{x} + 1}$$

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



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

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



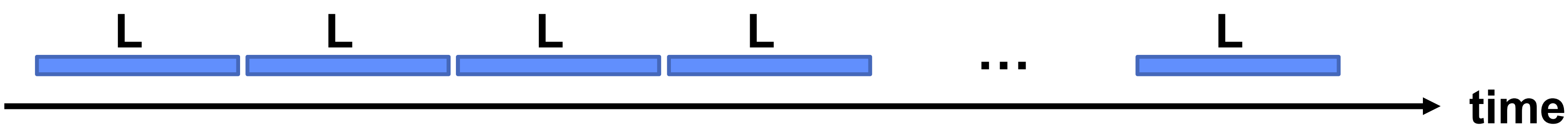
# What Determines Peak BW for I/O?

- 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…

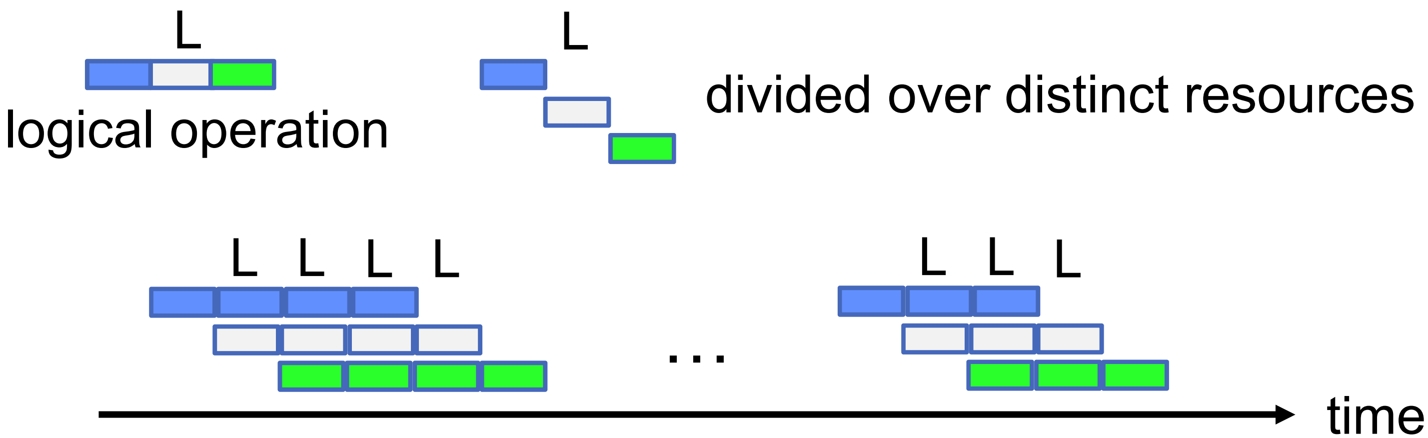
# Sequential Server Performance




- 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 $\rightarrow B = 0.5$ $^{\text{op}}/_{\text{yr}}$

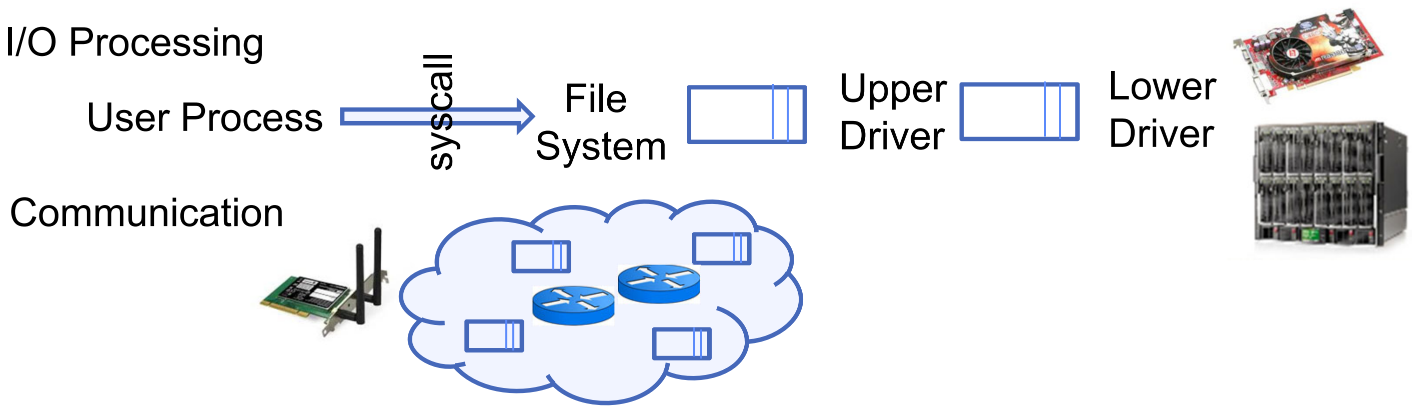
- Applies to a processor, a disk drive, a person, a TA, …

# Single Pipelined Server



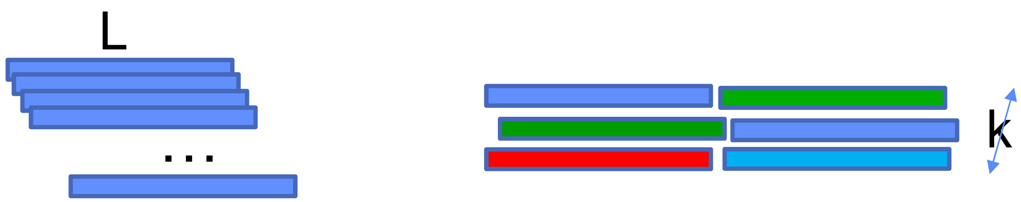

- Single pipelined server of $k$ stages for tasks of length $L$ (i.e., time $^L/_k$ per stage) delivers at rate $\leq {}^k/_L$.
  - $L = 10$ ms, $k = 4 \rightarrow B = 400$ $^{\text{op}}/_{\text{s}}$
  - $L = 2$ yr, $k = 2 \rightarrow B = 1$ $^{\text{op}}/_{\text{yr}}$

# 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

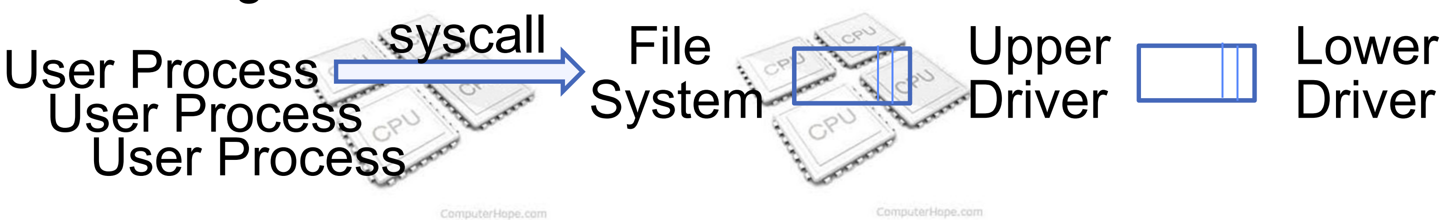
# Multiple Servers



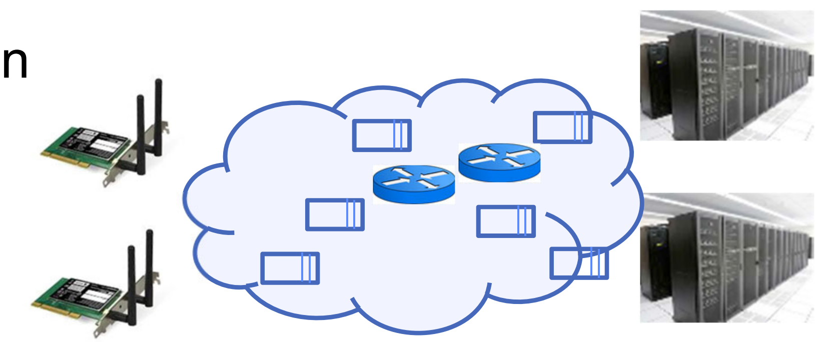

- $k$ servers handling tasks of length $L$ delivers at rate $\leq {}^{k}/_{L}$.
  - $L = 10$ ms, $k = 4 \rightarrow B = 400\ {}^{\text{op}}/_{\text{s}}$
  - $L = 2$ yr, $k = 2 \rightarrow B = 1\ {}^{\text{op}}/_{\text{yr}}$
- In 61C you saw multiple processors (cores)
  - Systems present lots of multiple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

# Example Systems "Parallelism"

I/O Processing

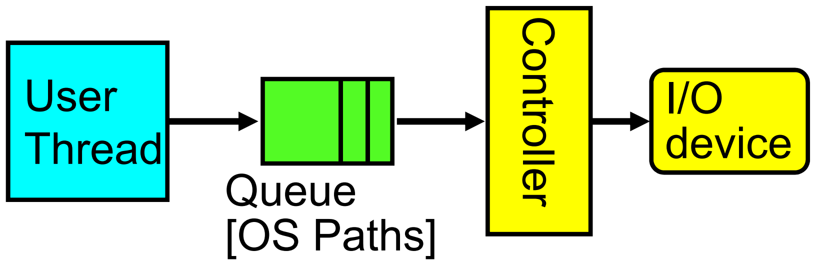



Communication

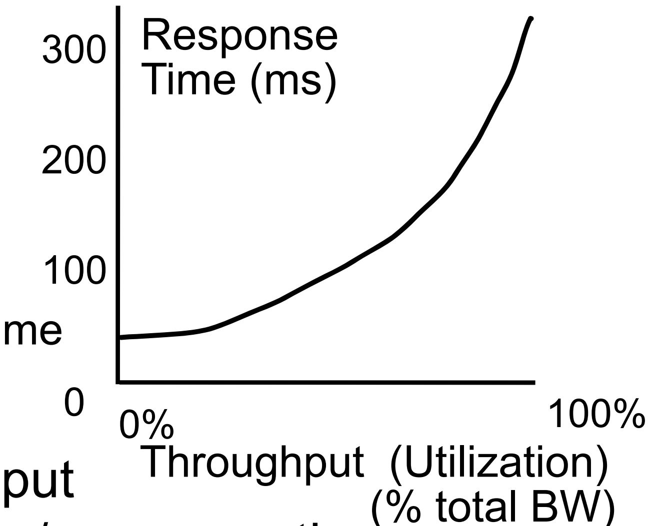


Parallel Computation, Databases, …

# I/O Performance




Response Time = Queue + I/O device service time



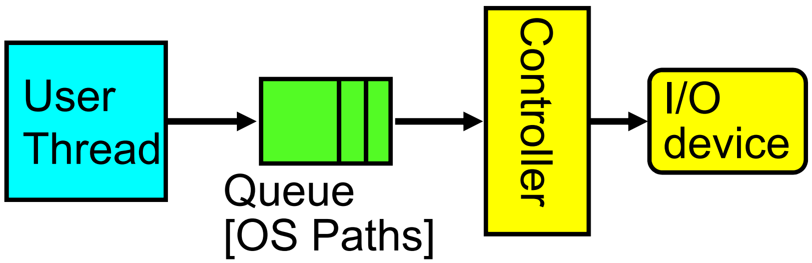

- Performance of I/O subsystem
  - Metrics: Response Time, Throughput
  - Effective BW per op = transfer size / response time
    - » EffBW(n) = n / (S + n/B) = B / (1 + SB/n )

# of ops

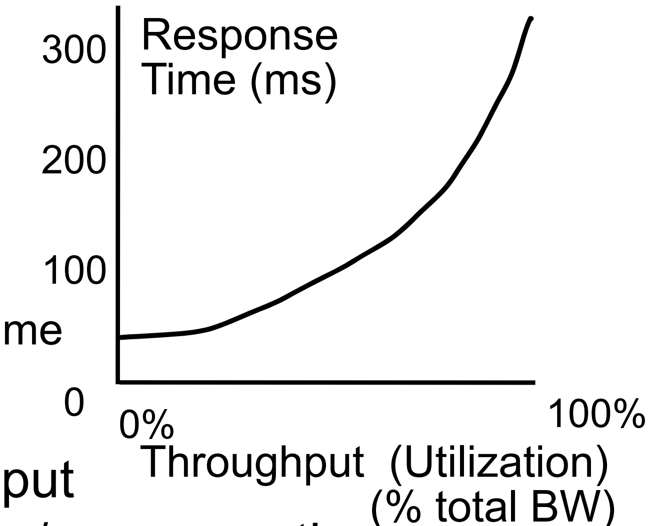
Fixed overhead

time per op

# I/O Performance

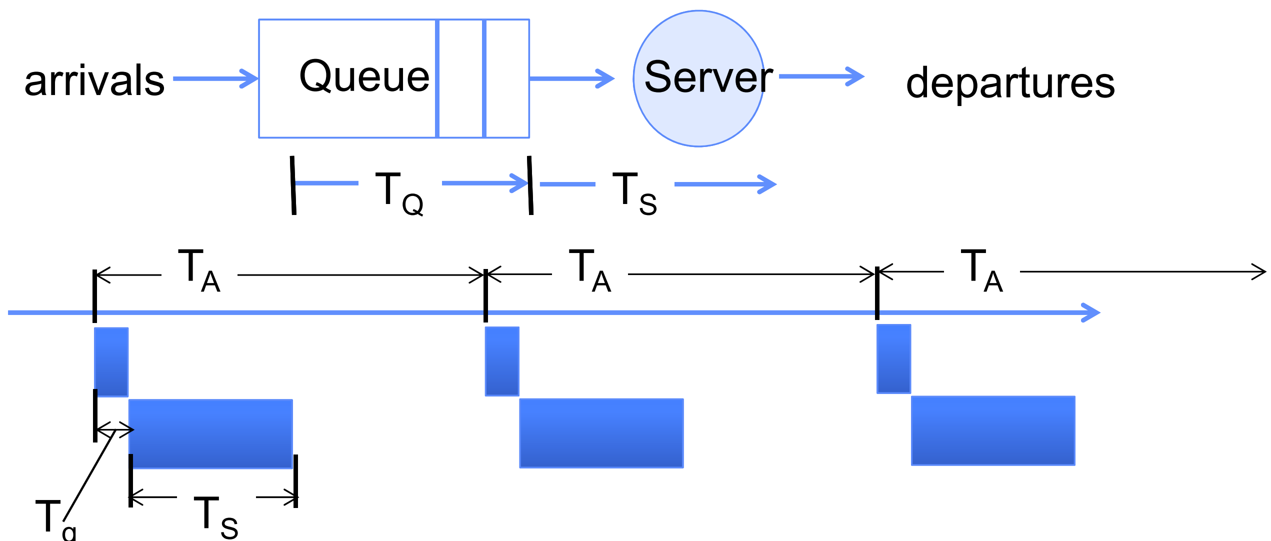



Response Time = Queue + I/O device service time



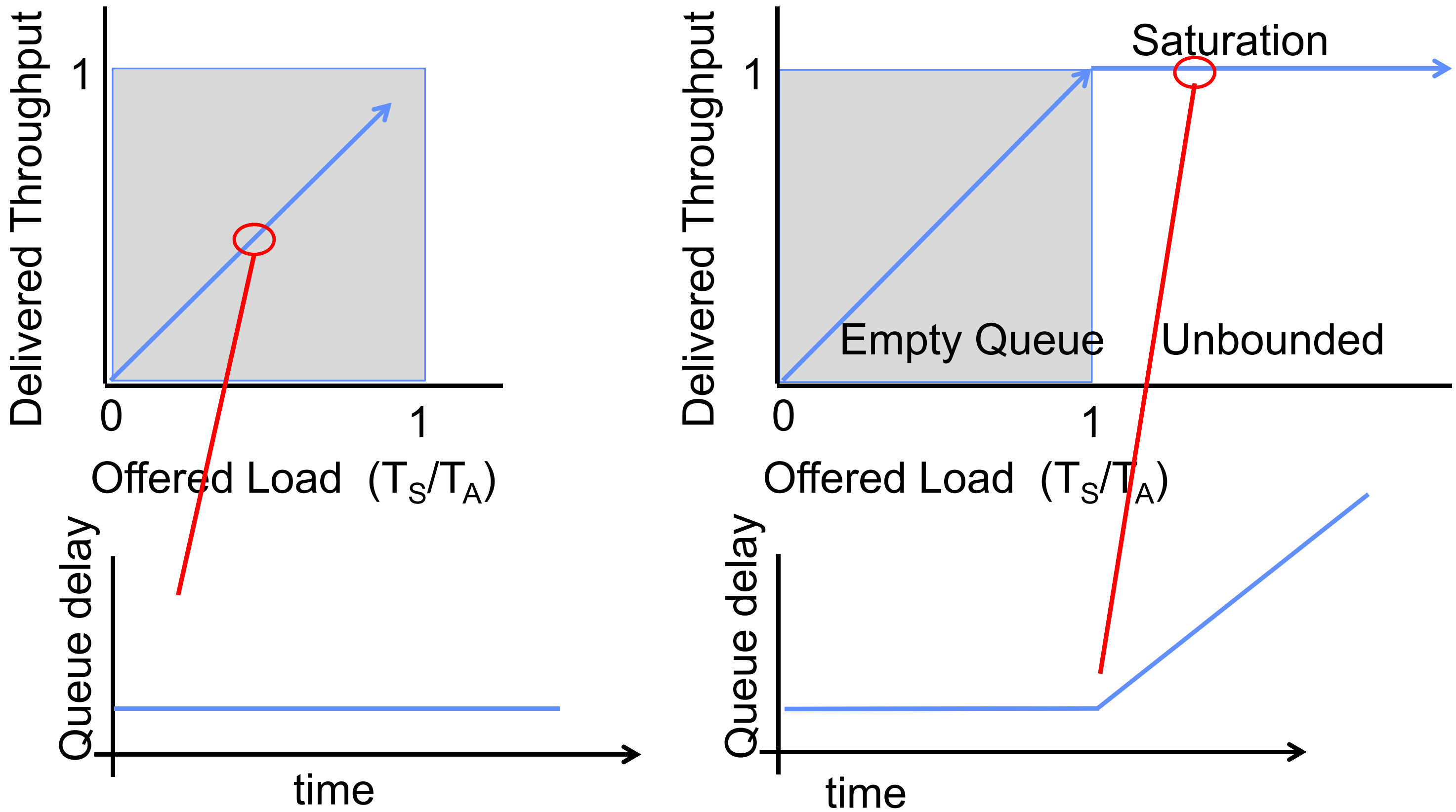

- Performance of I/O subsystem
  - Metrics: Response Time, Throughput
  - Effective BW per op = transfer size / response time
    - » EffBW(n) = n / (S + n/B) = B / (1 + SB/n )
  - Contributing factors to latency:
    - » Software paths (can be loosely modeled by a queue)
    - » Hardware controller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

# A Simple Deterministic World



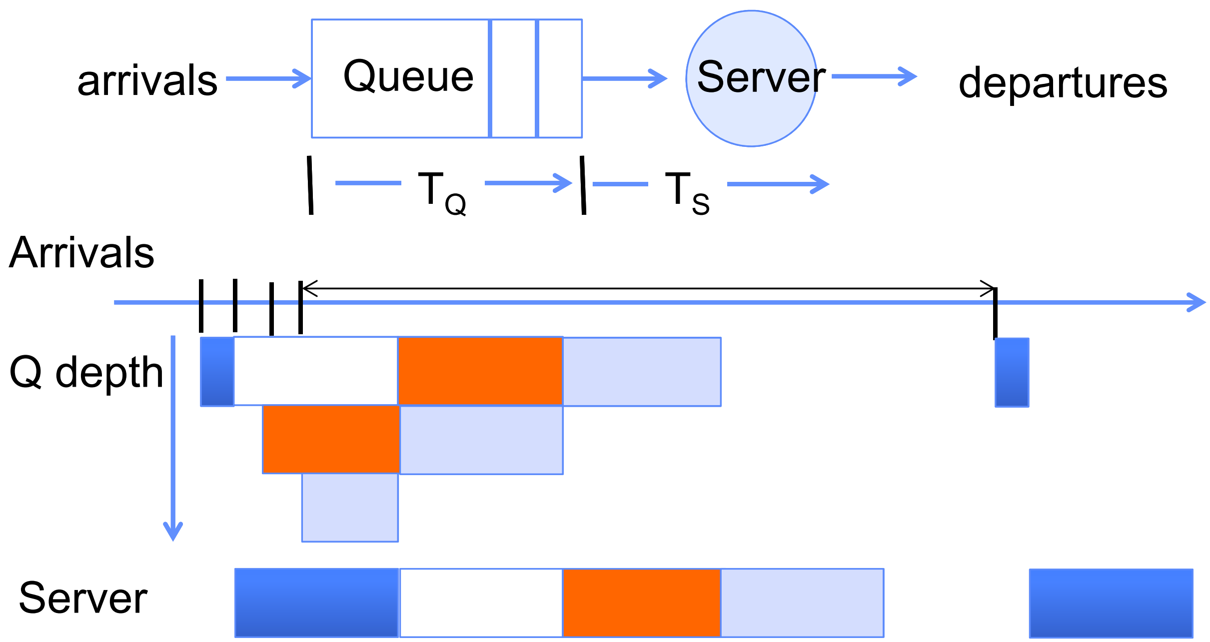

- Assume requests arrive at regular intervals, take a fixed time to process, with plenty of time between …
- Service rate ($\mu = 1/T_S$) - operations per second
- Arrival rate: ($\lambda = 1/T_A$) - requests per second
- Utilization: $U = \lambda/\mu$ , where $\lambda < \mu$
- Average rate is the complete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

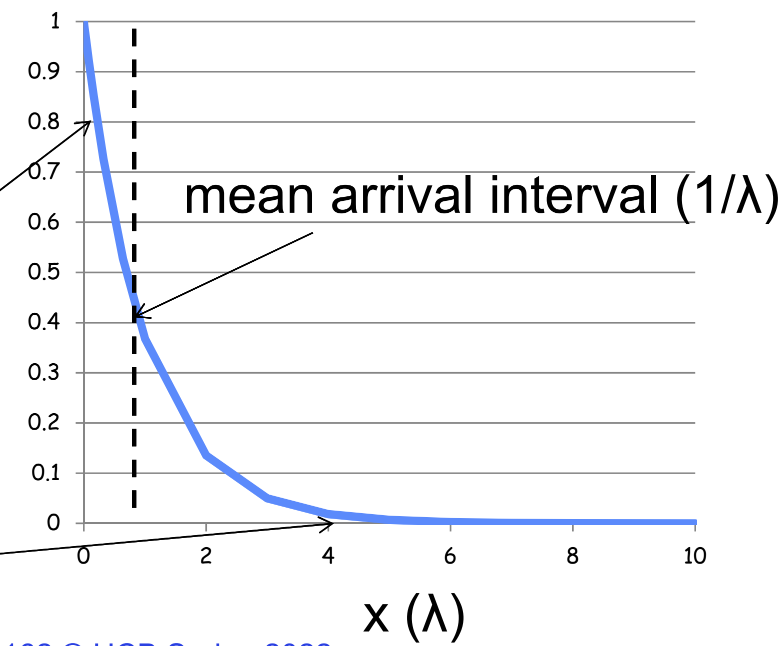
# So how do we model the burstiness of arrival?

- 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

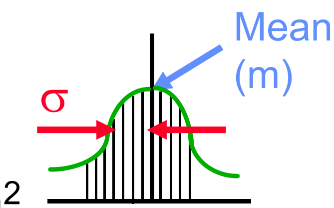
Lots of short arrival intervals (i.e., high instantaneous rate)

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



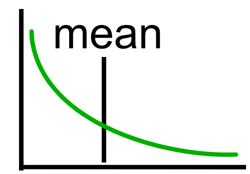
# Background:
# General Use of Random Distributions

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



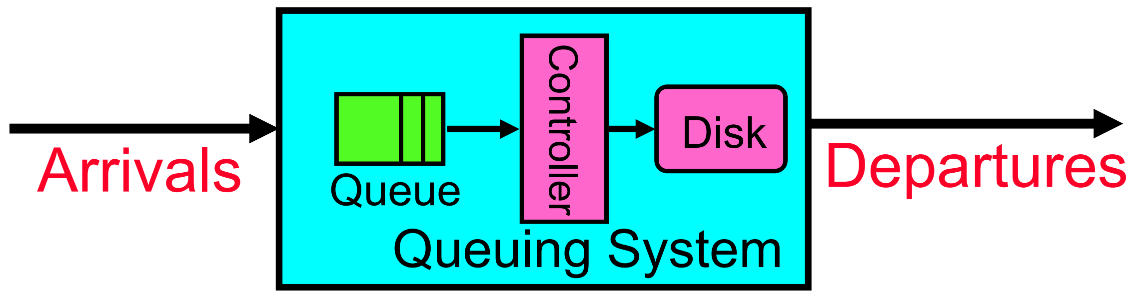

Distribution
of service times

- Important values of C:
  - No variance or deterministic $\Rightarrow$ 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)



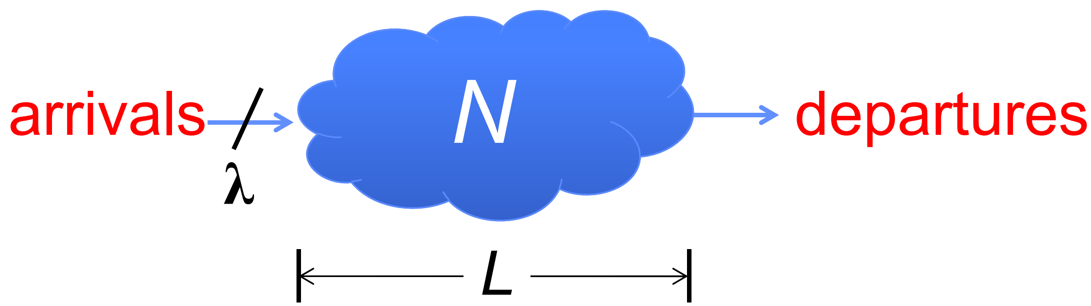

Memoryless

# Introduction to Queuing Theory



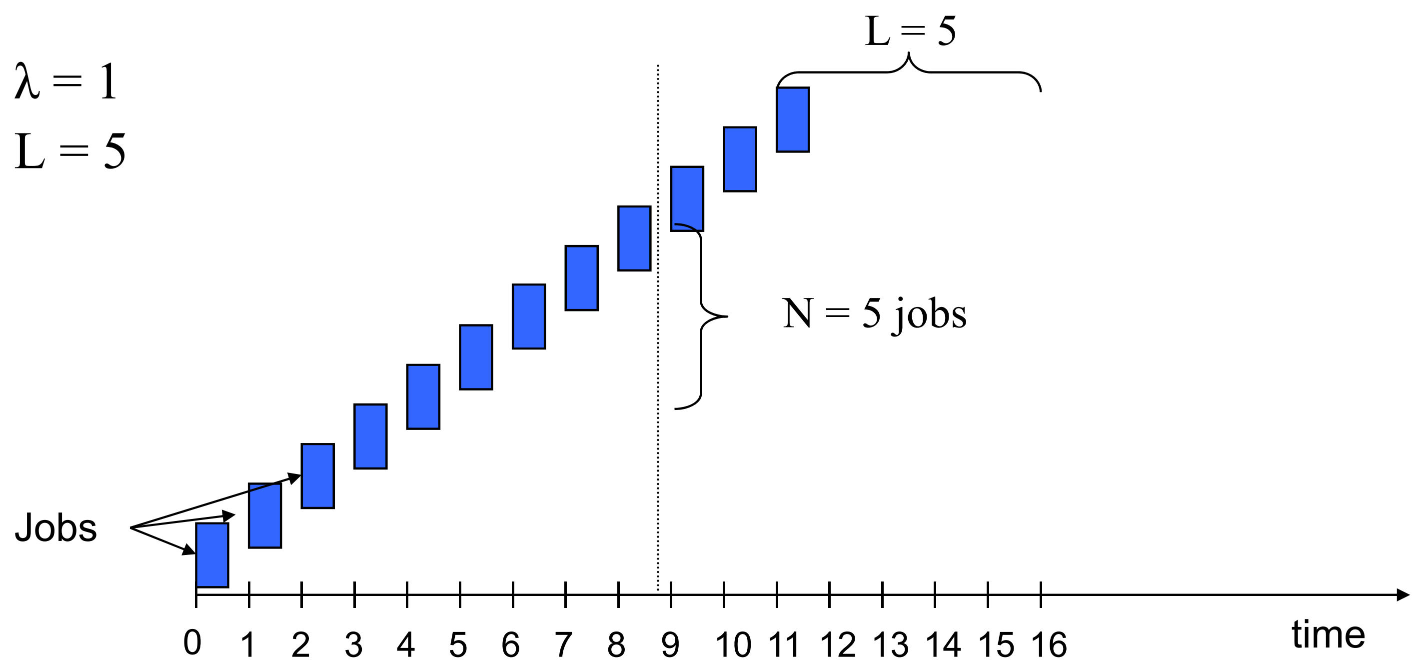

- What about queuing time??
  - Let's apply some queuing theory
  - Queuing Theory applies to long term, steady state behavior ⇒ Arrival rate = Departure rate
- Arrivals characterized by some probabilistic distribution
- Departures characterized by some probabilistic distribution

# 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 ($\lambda$) times the response time ($L$)
  - $N$ *(jobs)* = $\lambda$ *(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

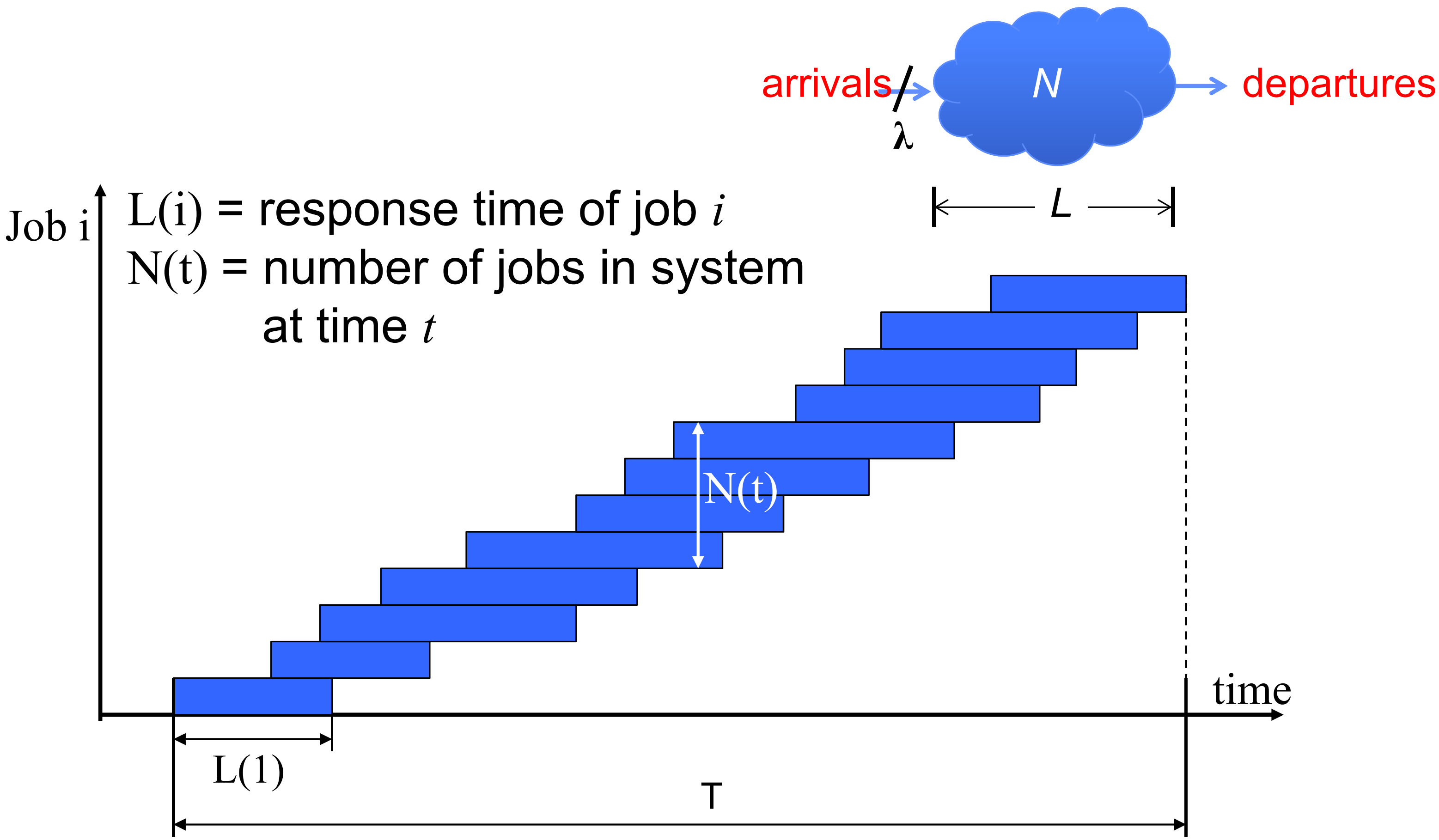
# Example



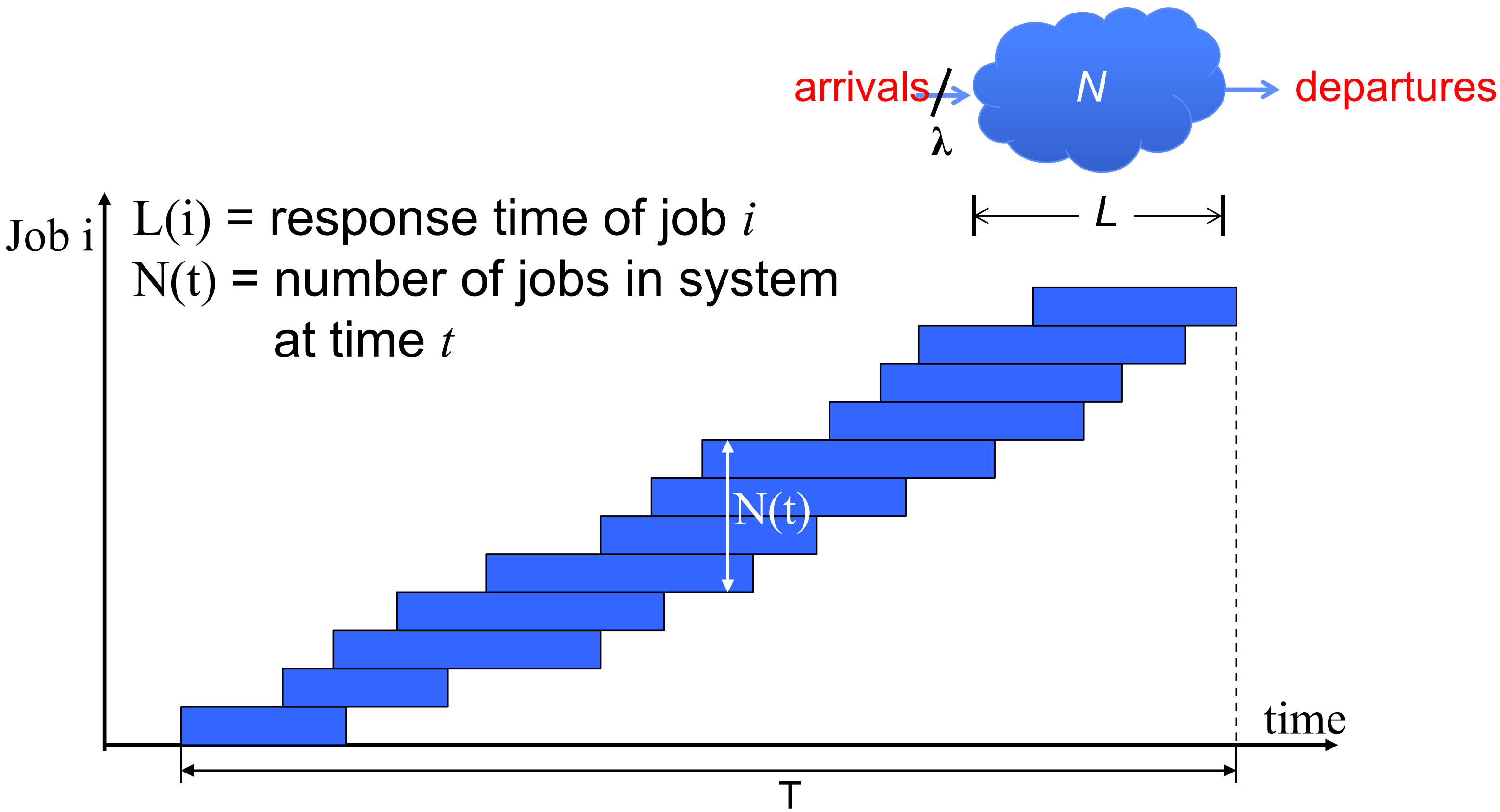

A: $N = \lambda \times L$

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

# Little's Theorem: Proof Sketch

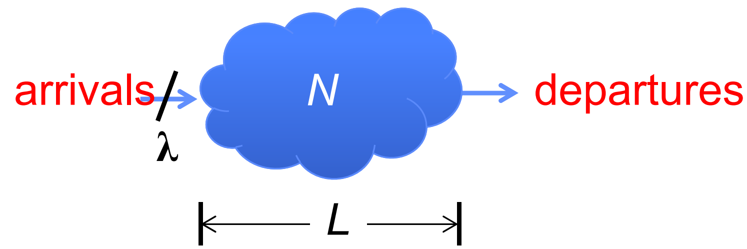


# Little's Theorem: Proof Sketch




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

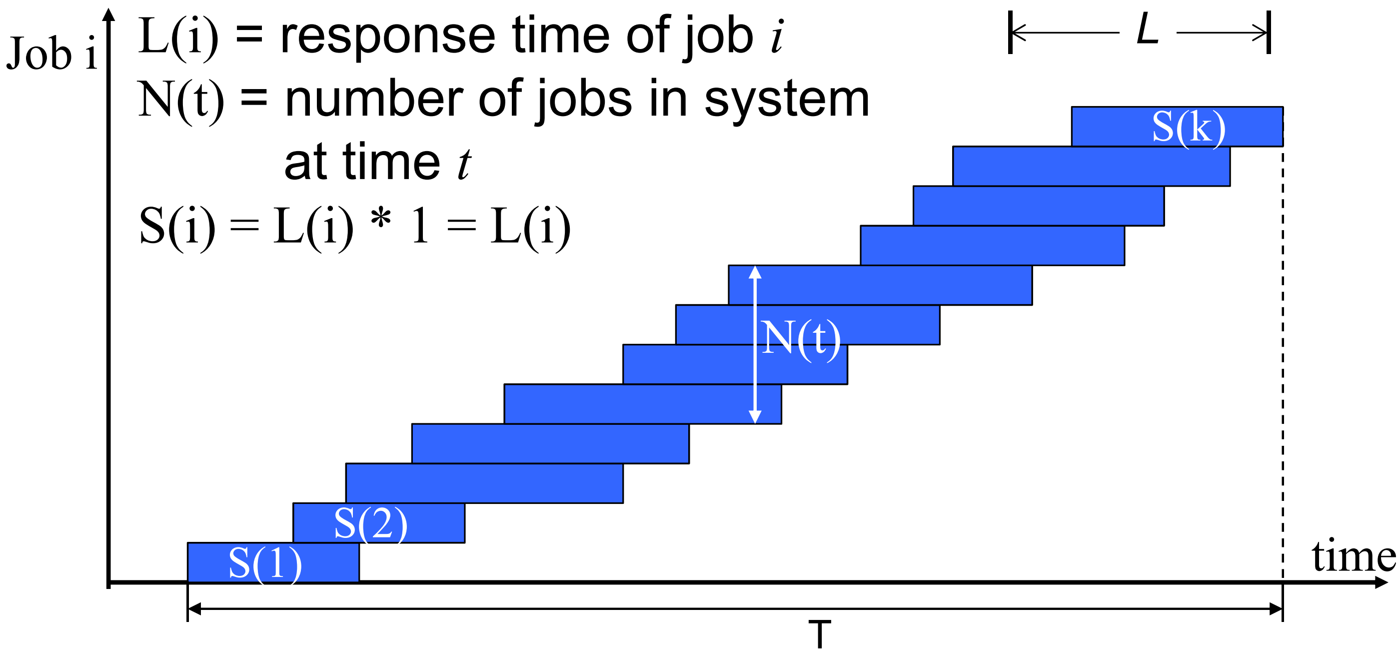
# Little's Theorem: Proof Sketch




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

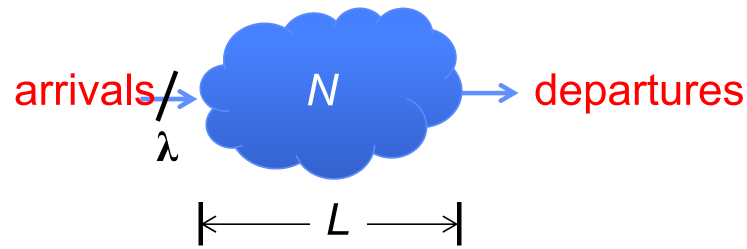
N(t) = number of jobs in system at time *t*

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




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

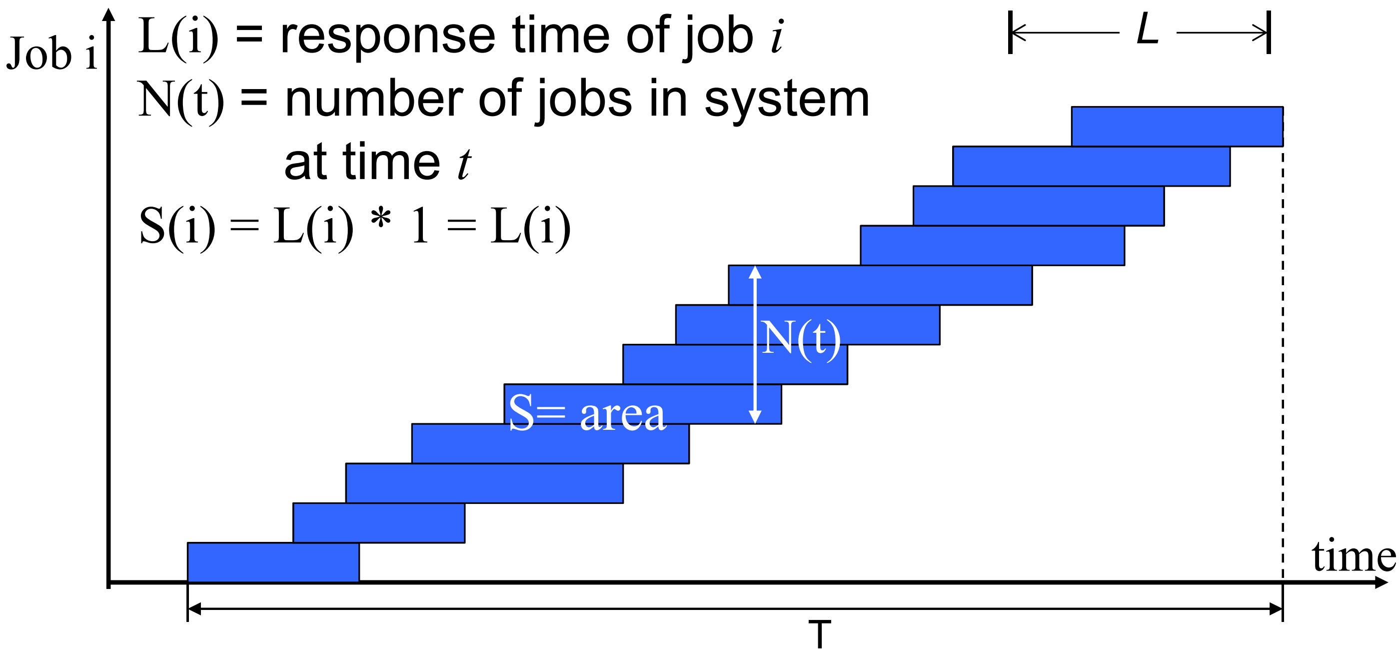
# Little's Theorem: Proof Sketch




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

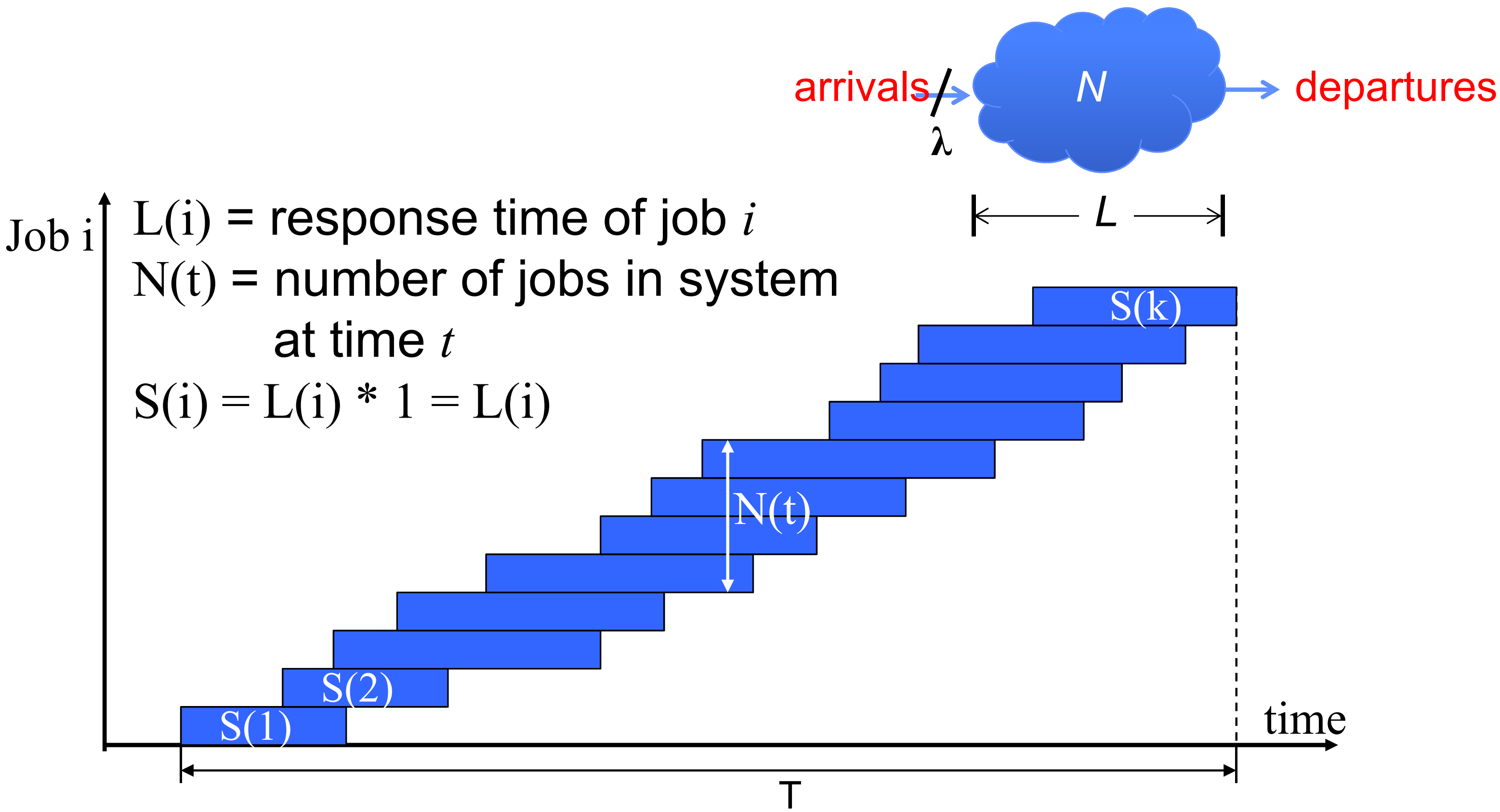
N(t) = number of jobs in system at time *t*

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




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

# Little's Theorem: Proof Sketch



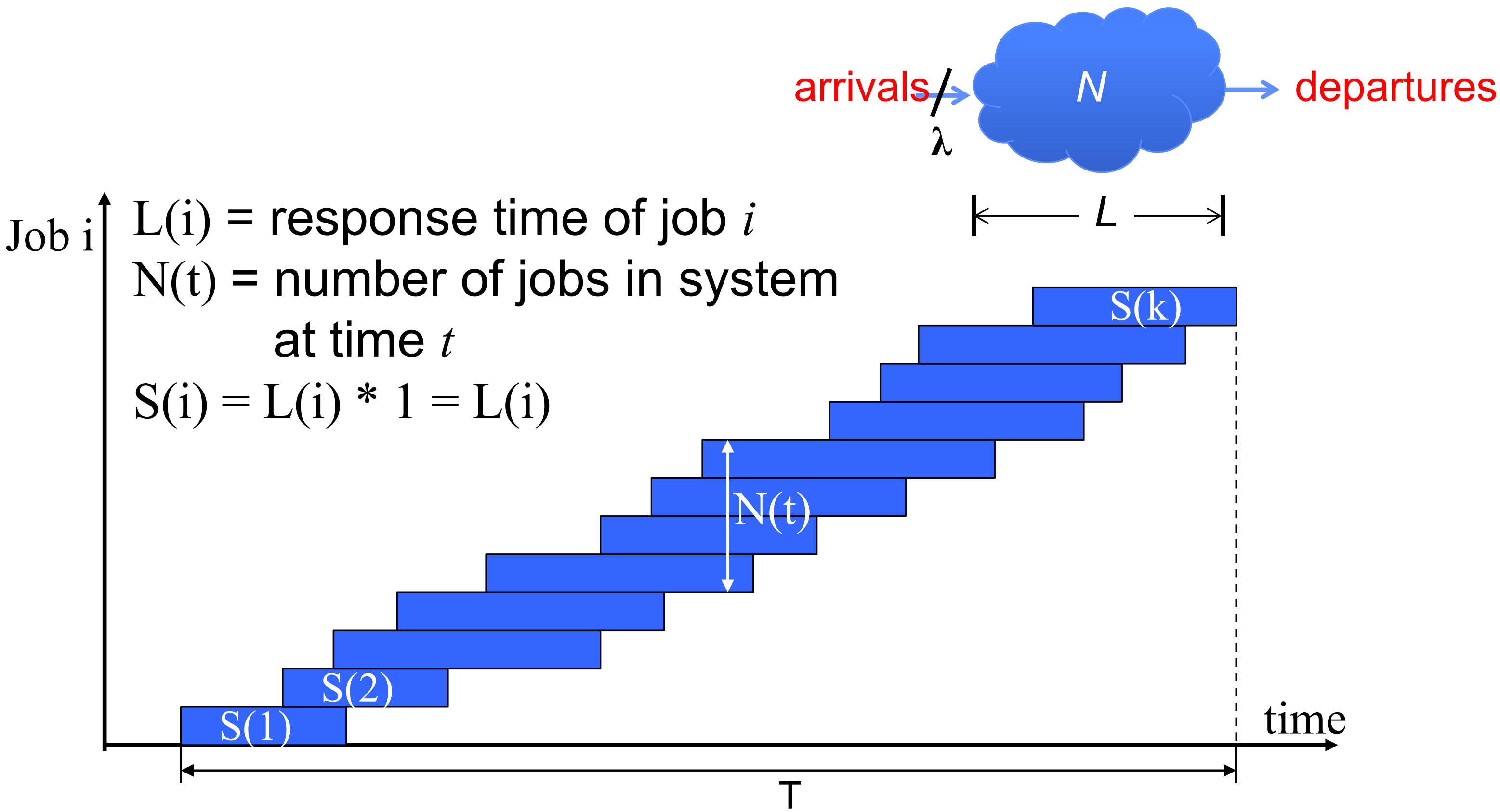

$$N_{avg} = S/T = (L(1) + \ldots + L(k))/T$$

# Little's Theorem: Proof Sketch

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

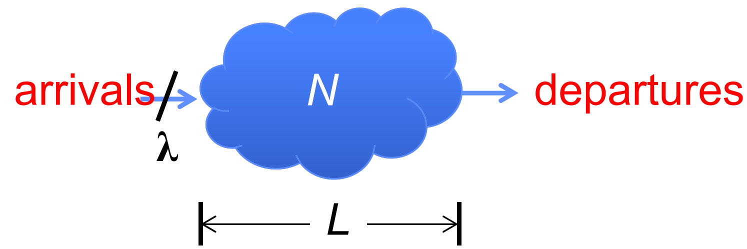
N(t) = number of jobs in system at time *t*

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




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

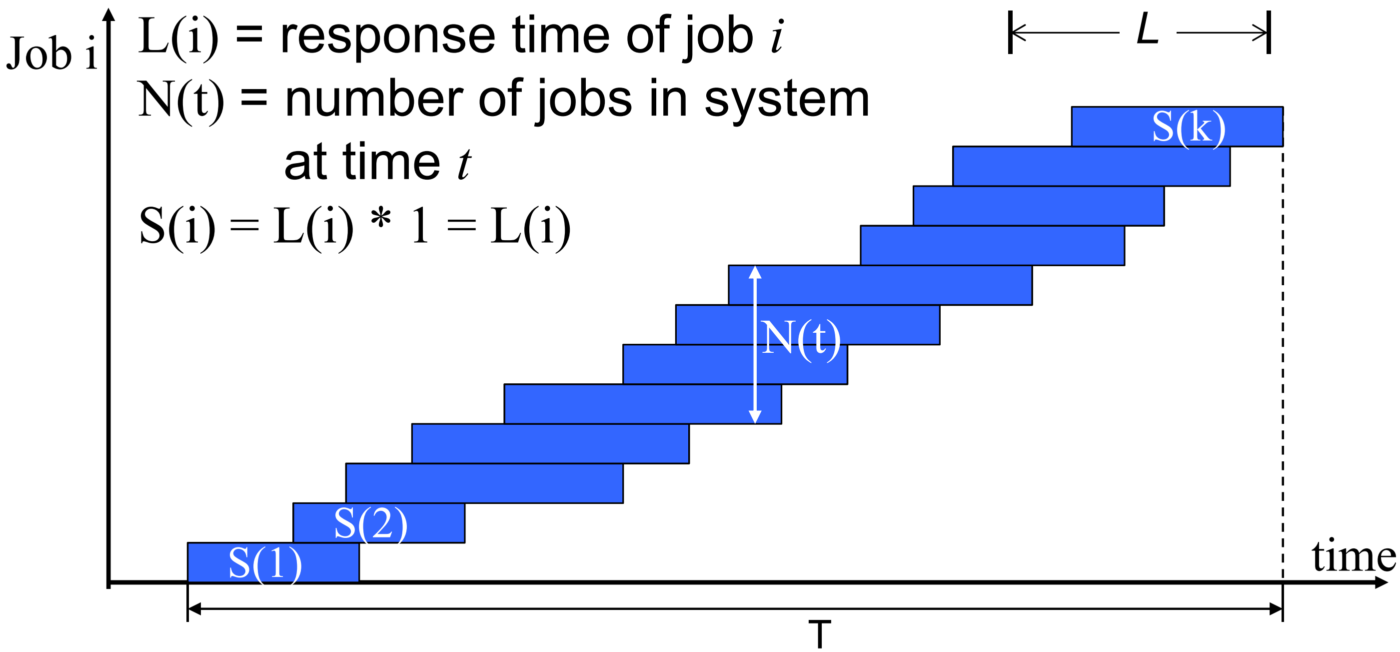
# Little's Theorem: Proof Sketch




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

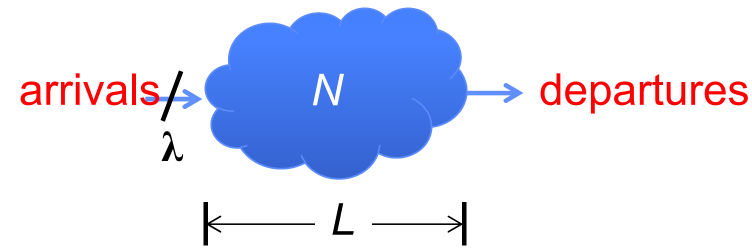
N(t) = number of jobs in system at time $t$

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




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

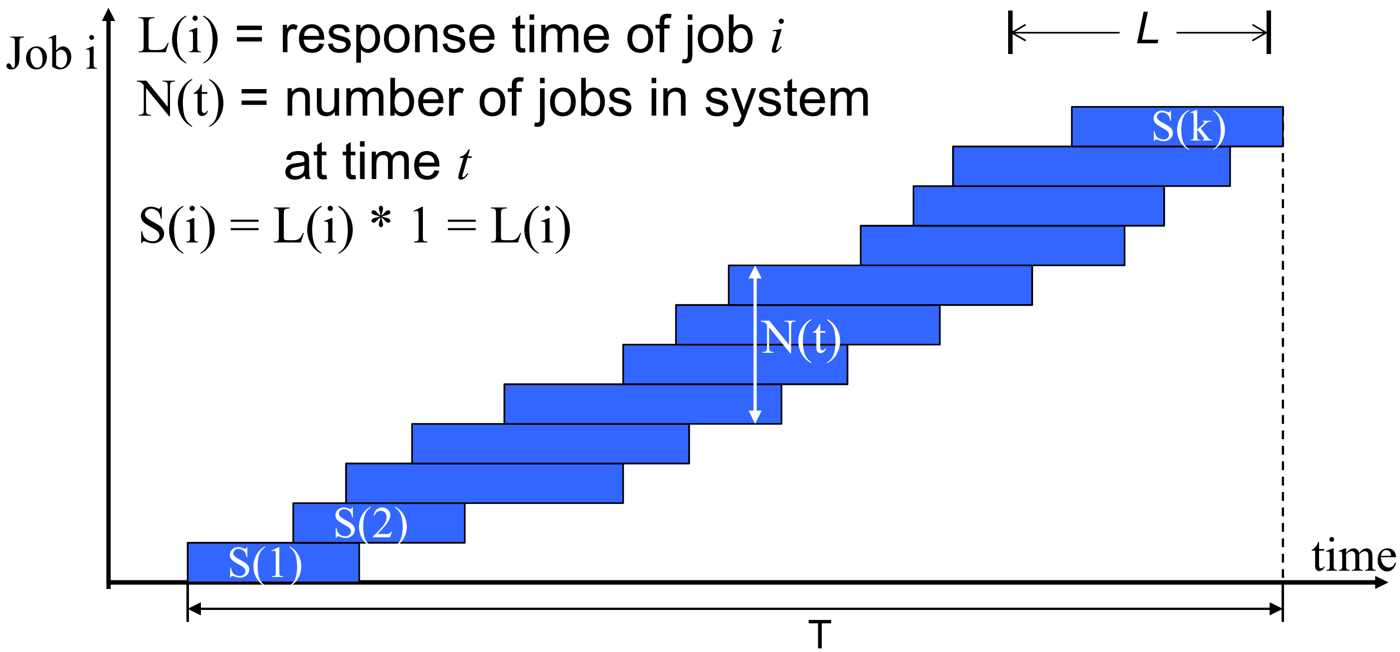
# Little's Theorem: Proof Sketch




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

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

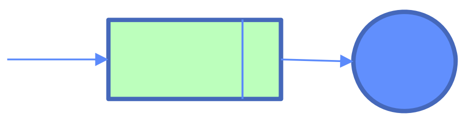
S(i) = L(i) * 1 = L(i)




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

# Little's Law Applied to a Queue

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



Average Arrival Rate

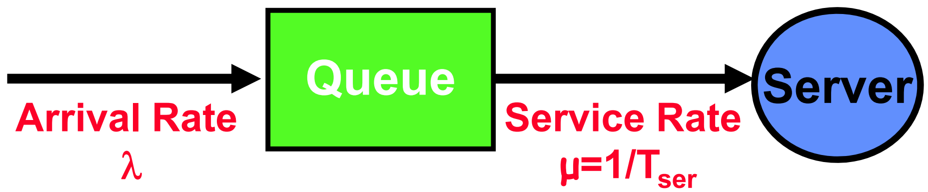
$$L_Q = \lambda T_Q$$

Average length of the queue

Average time "waiting" in queue

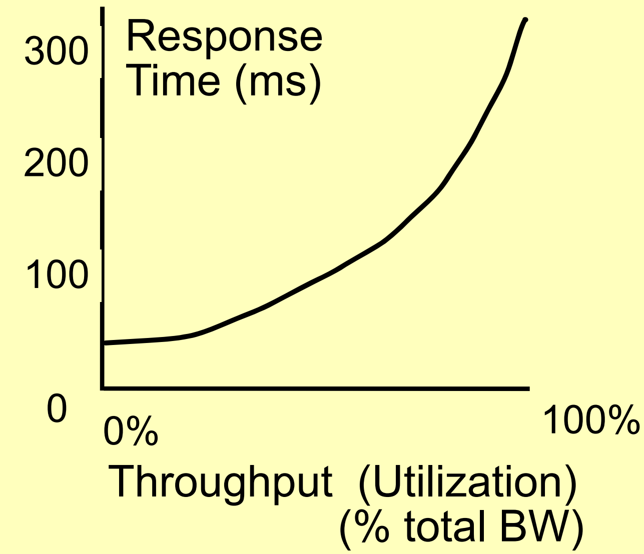
# A Little Queuing Theory: Computing $T_Q$

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

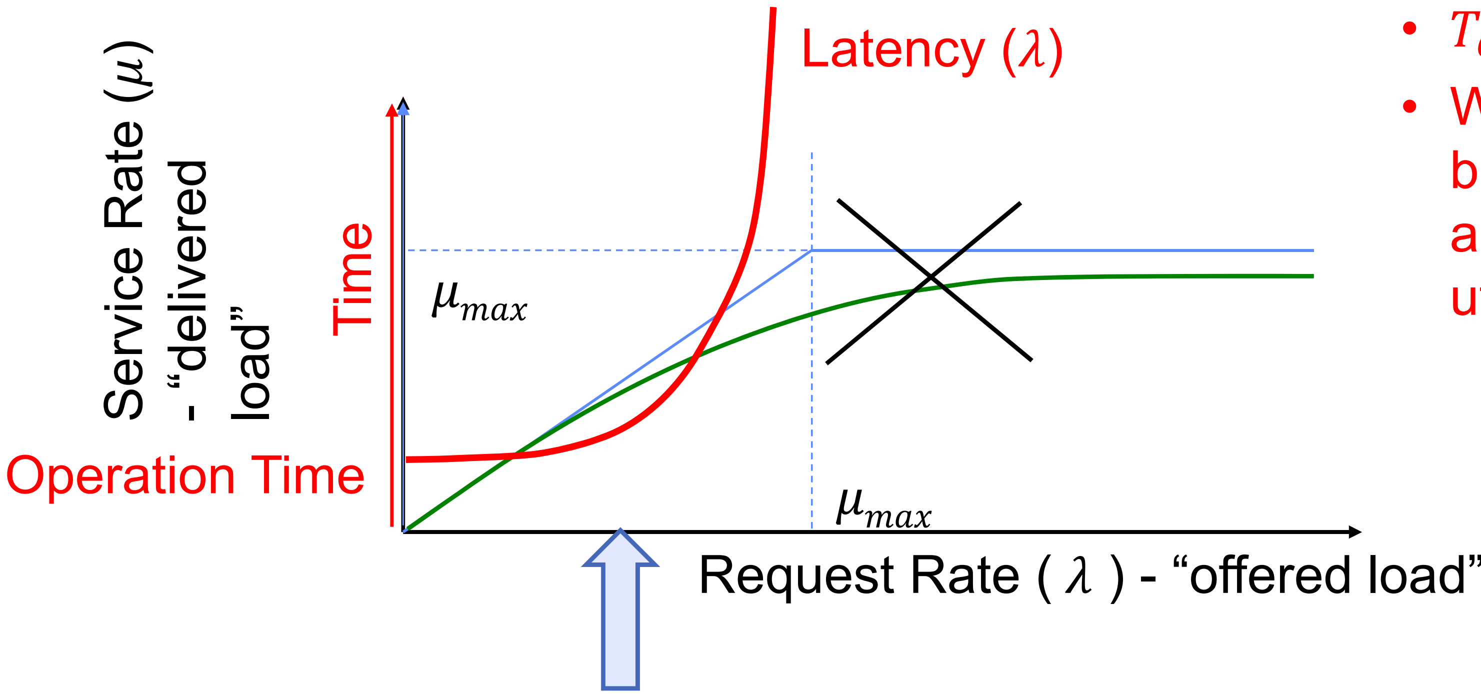



- Parameters that describe our system:
  – $\lambda$: mean number of arriving customers/seco
  – $T_{ser}$: mean time to service a customer ("m
  – C: squared coefficient of variance
  – $\mu$: service rate = $1/T_{ser}$
  – u: server utilization ($0 \le u$ ...): $u = \lambda/\mu = \lambda \times T_s$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

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



# System Performance In presence of a Queue

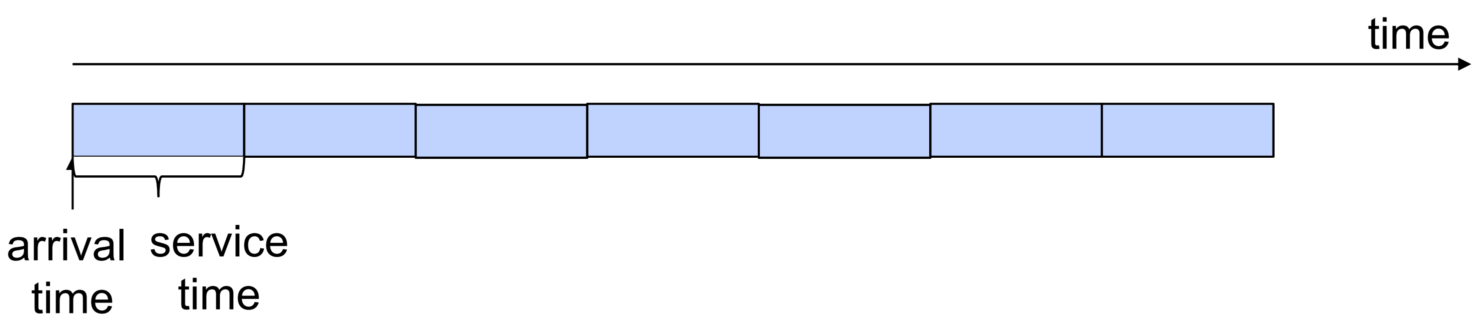



- $T_Q \sim \frac{u}{1-u}$, $\mathrm{u} = \lambda/\mu_{max}$
- Why does latency blow up as we approach 100% utilization?
  - Queue builds up on each burst
  - But very rarely (or never) gets a chance to drain

"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%

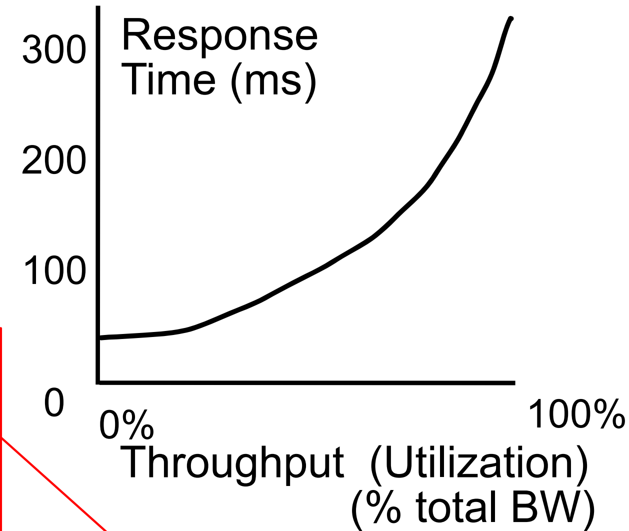
# Why unbounded response time?

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

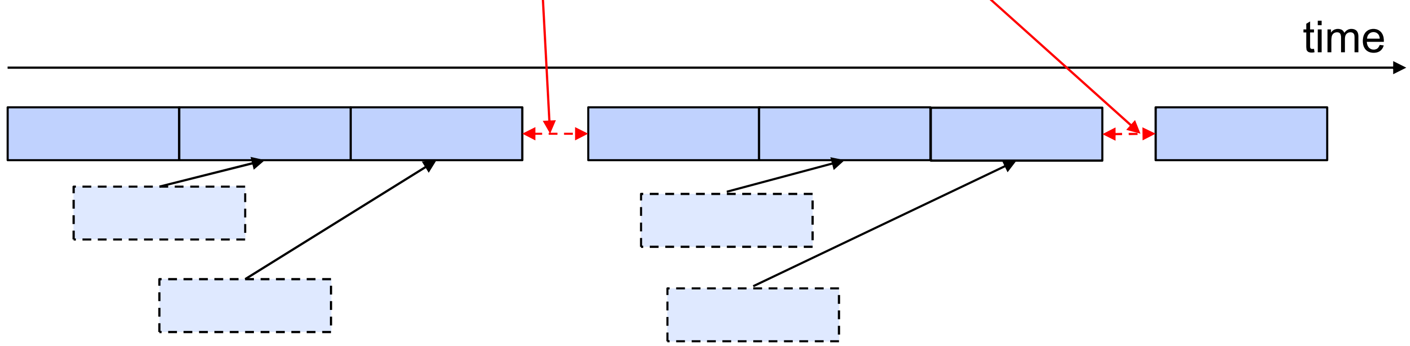


# Why unbounded response time?

- 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:

$\lambda$ (avg # arriving customers/s) = 10/s

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

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

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

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

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

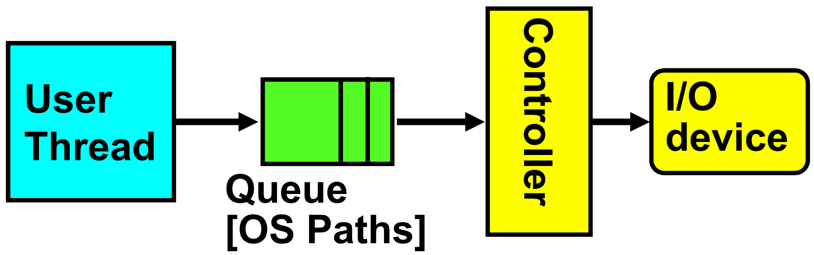
# Queuing Theory Resources

- 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

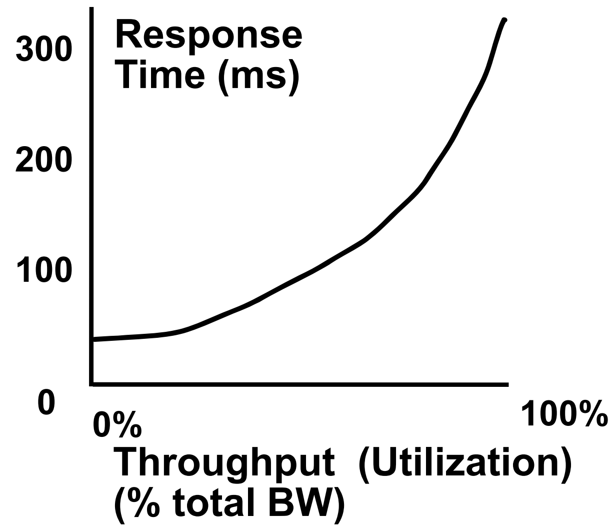
- Some previous midterms with queueing theory questions

- 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

# When is Disk Performance Highest?

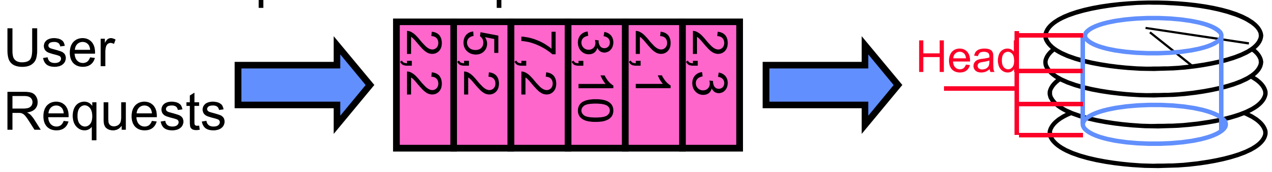
- When there are big sequential reads, or
- When there is so much work to do that they can be piggy backed (reordering queues—one moment)

- OK to be inefficient when things are mostly idle
- Bursts are both a threat and an opportunity
- <your idea for optimization goes here>
  – Waste space for speed?

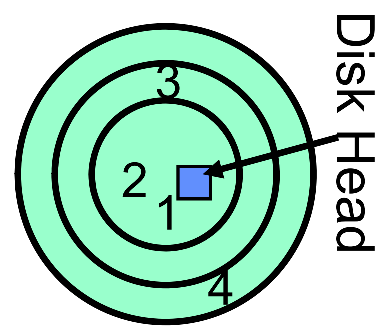
- Other techniques:
  – Reduce overhead through user level drivers
  – Reduce the impact of I/O delays by doing other useful work in the meantime

# Disk Scheduling (1/3)

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



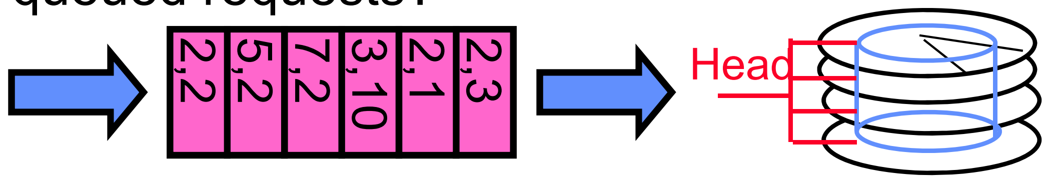

- FIFO Order
  - Fair among requesters, but order of arrival may be to random spots on the disk $\Rightarrow$ Very long seeks
- SSTF: Shortest seek time first
  - Pick the request that’s closest on the disk
  - Although called SSTF, today must include rotational delay in calculation, since rotation can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



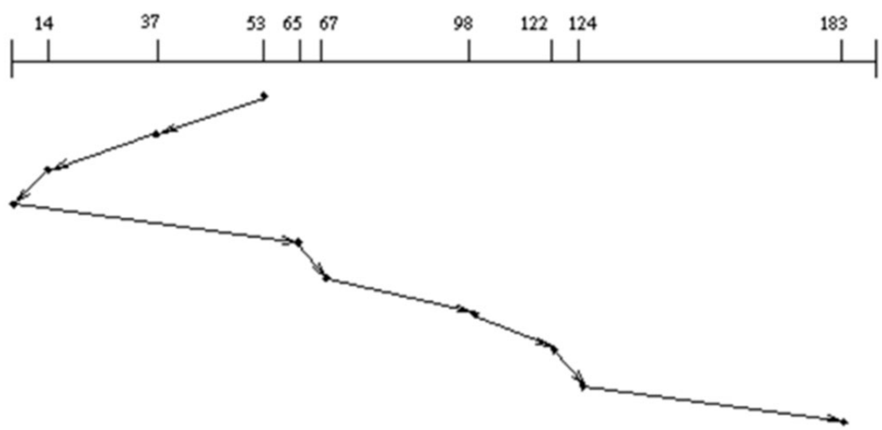
# Disk Scheduling (2/3)

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

User Requests

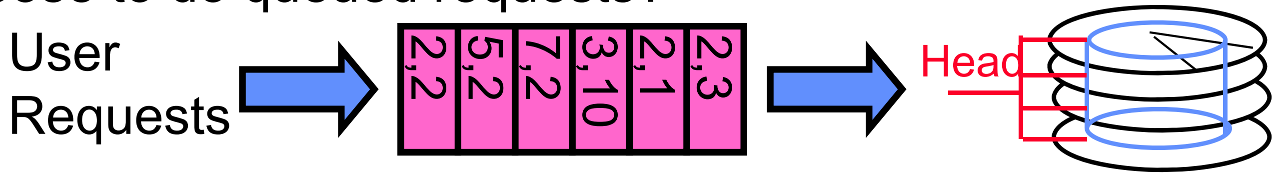



- SCAN: Implements an Elevator Algorithm: take the closest request in the direction of travel
  – No starvation, but retains flavor of SSTF

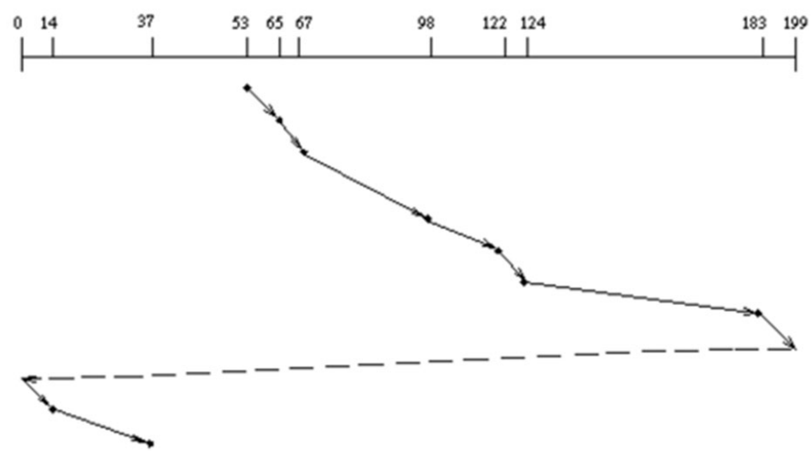


# Disk Scheduling (3/3)

- 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



# Recall: How do we Hide I/O Latency?

- 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 to kernel
  - Read may return nothing, write may write nothing
- Asynchronous Interface: "Tell Me Later"
  - When requesting data, take pointer to user's buffer, return immediately; later kernel fills buffer and notifies user
  - When sending data, take pointer to user's buffer, return immediately; later kernel takes data and notifies user

# 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* |

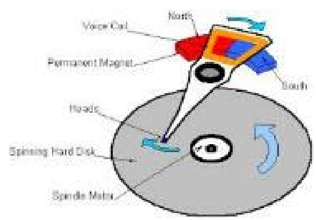
What we covered in Lecture 4

What we will cover next…

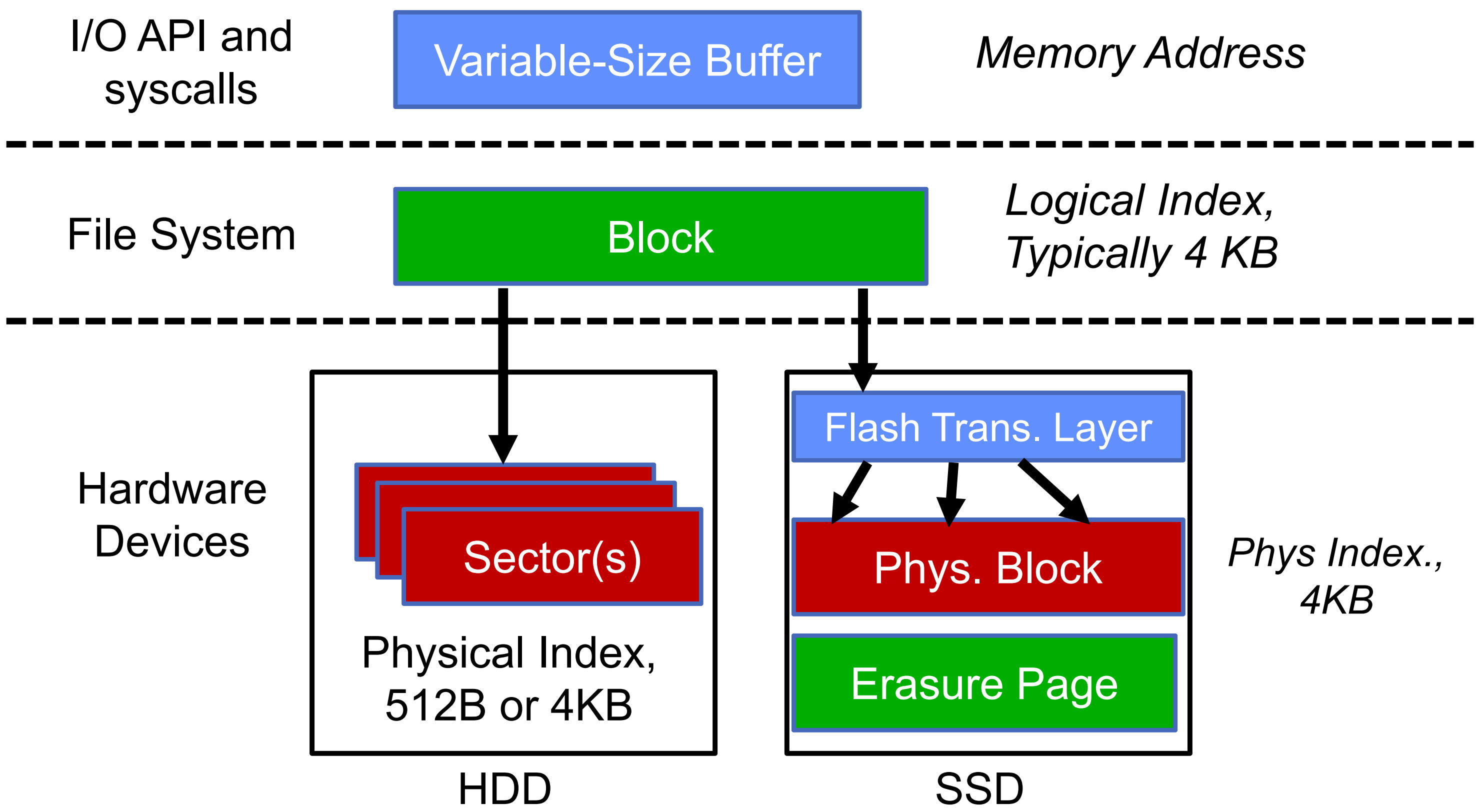
What we just covered…







# From Storage to File Systems



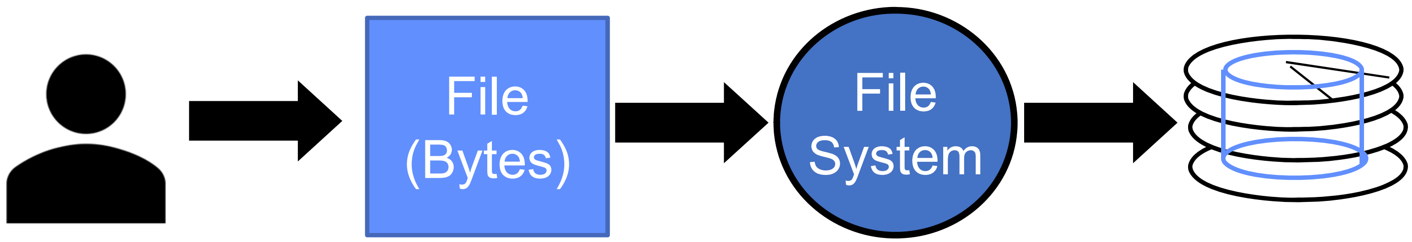
# Building a File System

- 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 $\geq$ 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 anymore
    - » OS/BIOS must deal with bad sectors
  - Logical Block Addressing (LBA)
    - » Every sector has integer address
    - » Controller translates from address ⇒ 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**

# Conclusion

- Disk Performance:
  – Queuing time + Controller + Seek + Rotational + Transfer
  – Rotational latency: on average ½ rotation
  – Transfer time: spec of disk depends on rotation speed and bit storage density
- Devices have complex interaction and performance characteristics
  – Response time (Latency) = Queue + Overhead + Transfer
    » Effective BW = BW * T/(S+T)
  – HDD: Queuing time + controller + seek + rotation + transfer
  – SDD: Queuing time + controller + transfer (erasure & wear)
- 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 $\rightarrow \infty$

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