Foundations: Operational Analysis and Queuing Models

Software Performance Engineering: Theory & Practice

Outline

- Operational Analysis
  - Utilization Law
  - Little's Law
  - Bottleneck analysis

- Queuing models
  - Open models
    - M/M/1; M/M/n; M/D/1; M/G/1; G/G/1
  - Closed Network models

Analytic modeling of computer system performance

- Prediction ⇨ Capacity planning
  - Out-of-Capacity conditions can be catastrophic
  - Performance is usually cited as the 2^{nd} most important factor related to user satisfaction
- Finite limits on computing resources
  - Understand queueing behavior when resources saturate
- Analytic methods:
  - Bottleneck analysis for current systems
  - Compare design alternatives for new application development

Historical Development

- The 1st generation of computers (~1960) that used semiconductor technology led to rapid expansion of the field of Computer Science
  - IBM 360
  - timesharing – required cost accounting based on resource usage

- Similarity between computerized task scheduling algorithms and optimizations familiar from Operations Research
  - e.g., see Donald Knuth, "The Art of Computer Programming: Fundamental Algorithms," first published in 1968.
  - Instrumentation added to assist with fine-tuning these algorithms

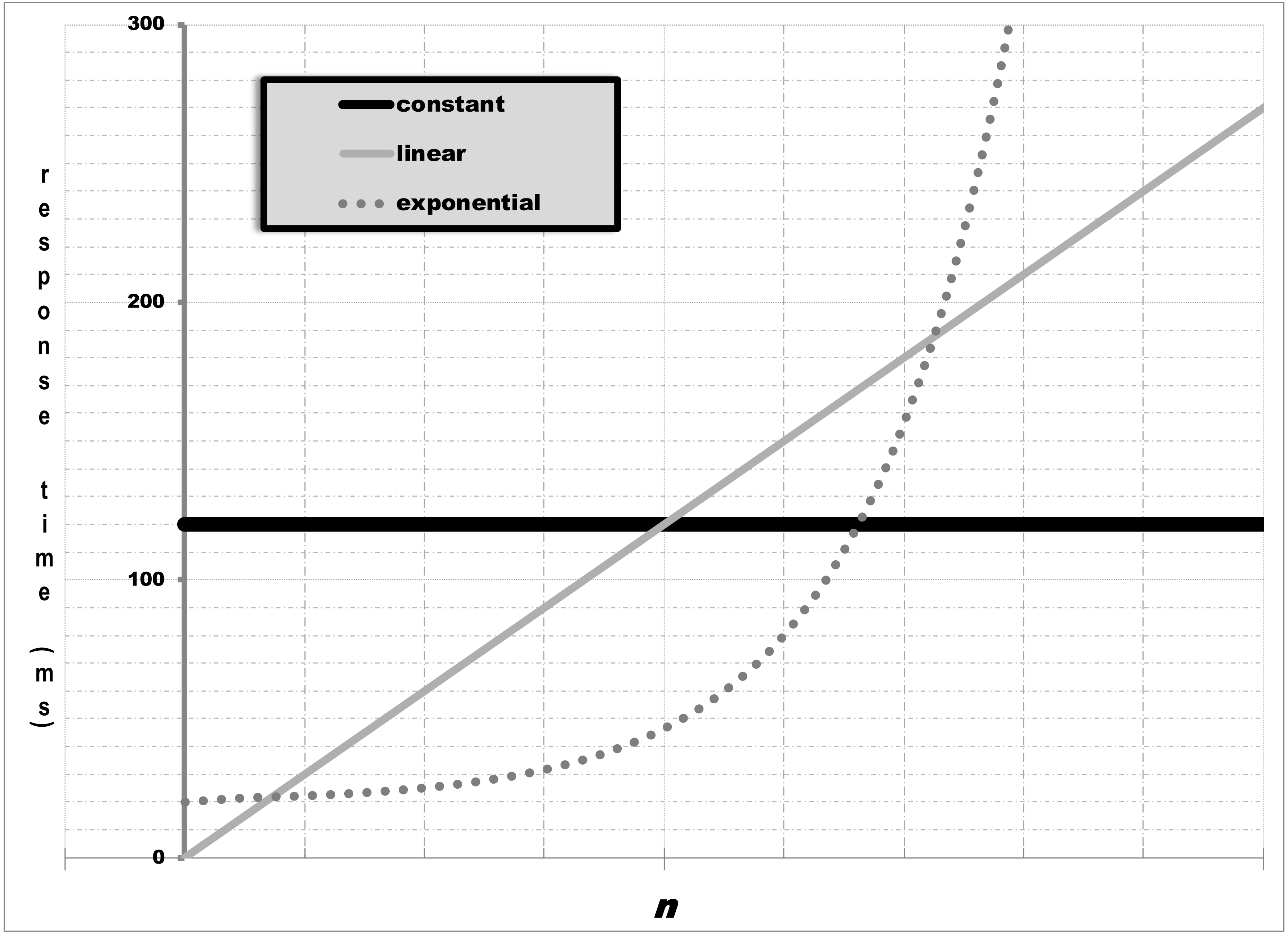
Historical Development

- Early computers were not only expensive, but they were slow (by today's standards)
- these limitations inspired intense interest in ***performance***
  - e.g., Sort algorithms
- Cost accounting in early ***time-shared*** systems that ran **batch jobs** required instrumentation:
  - execution time + queueing = **turnaround time**
  - resource consumption
    - CPU time
    - IOs to peripherals (disk, tape)
    - lines printed
    - etc.

Historical Development: References

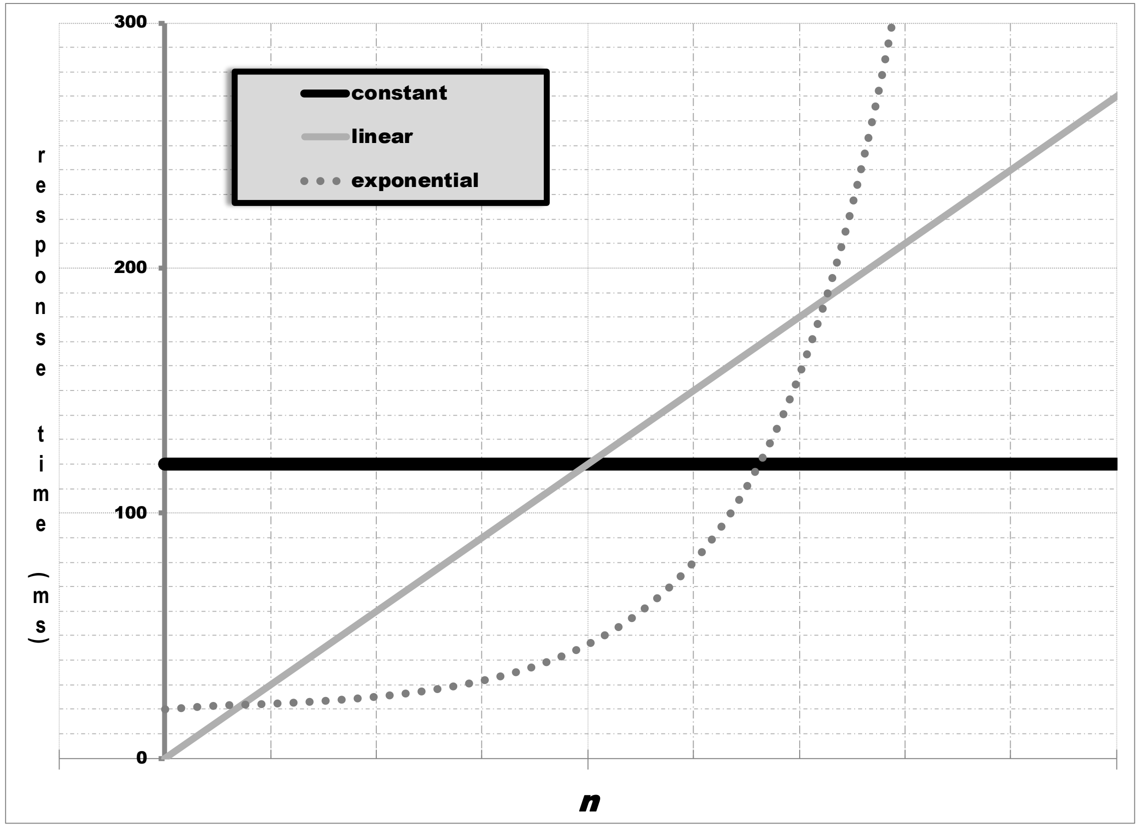
- Leonard Kleinrock, *Queuing Systems: Volume II – Computer Applications*, 1976.
- Peter Denning and Jeff Buzen, "The Operational analysis of queuing network models," *Computing Surveys*, 1978.
- Ed Lazowska, *et. al.*, *Quantitative System Performance*, 1984.
- Connie Smith, *Performance Engineering of Software Systems*, 1990.

Scalability

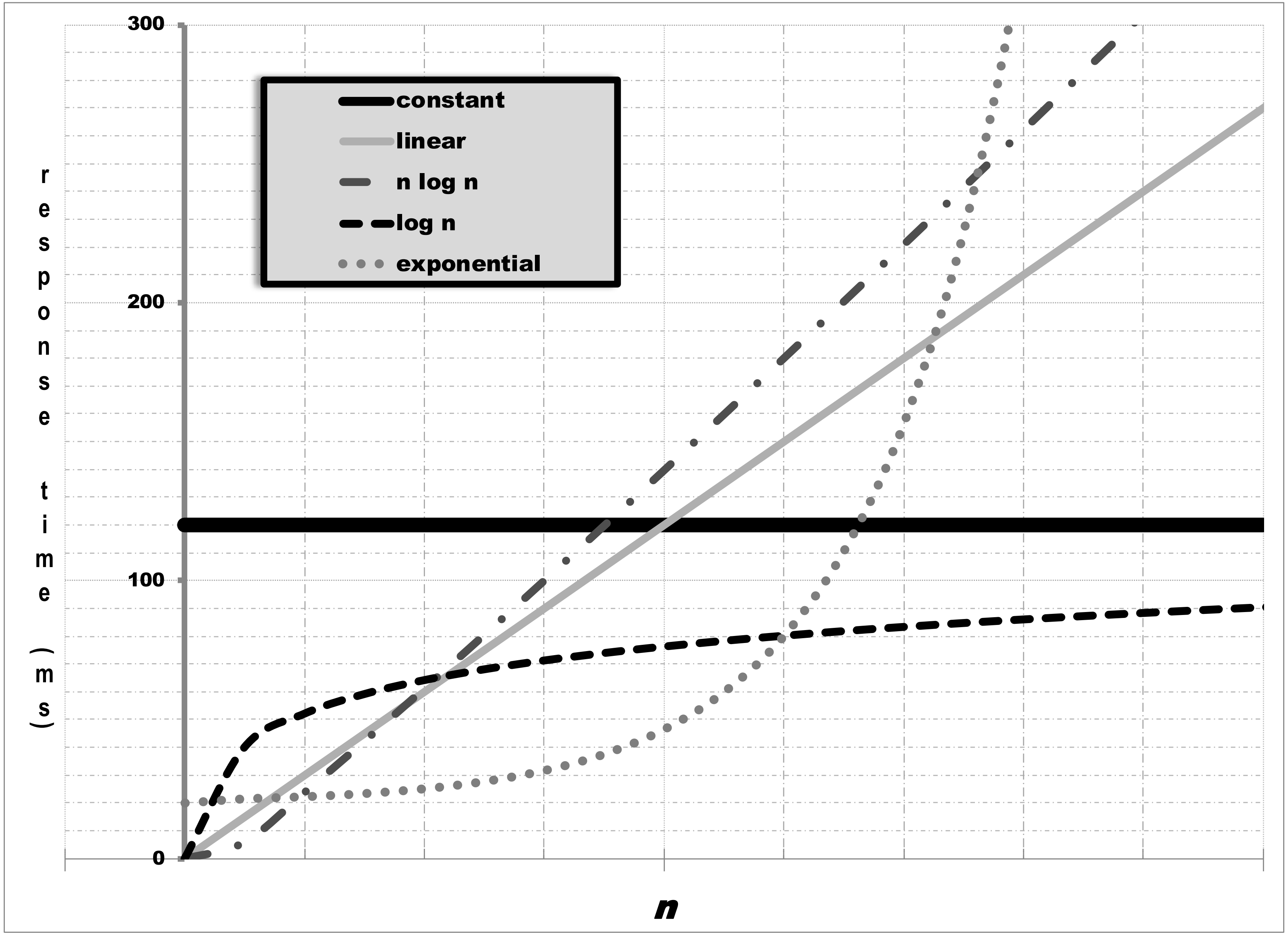


Scalability

- Why does actual performance diverge from the ideal?
  - Computer resources have finite capacity limits
  - As the workload grows, these limits eventually become manifest
  - Concurrent requests for shared resources generates contention
    - e.g., processor sharing:
      - time-slicing
      - priority



Algorithm complexity (scalability)



Operational analysis

- Notation
  - T : the length of time or ***duration*** of the observation period
  - K : the set of computer resources: CPUs, disks, etc.
  - B_i : total ***busy*** time of resource K_i during observation period *T*
  - A_i : service requests to resource K_i during period *T*
  - A_0 : Total requests (***arrivals***) during period *T*
  - C_i : service requests completed at resource K_i during period *T*
  - C_0 : Total requests completed (***completions***) during period *T*

Operational analysis

- **Notation**
  - T : the length of time or *duration* of the observation period
  - K : the set of computer resources: CPUs, disks, etc.
  - B_i : total *busy* time of resource K_i during observation period T
  - A_i : service requests to resource K_i during period T
  - A_0 : Total requests (*arrivals*) during period T
  - C_i : service requests completed at resource K_i during period T
  - C_0 : Total requests completed (*completions*) during period T

- **Basic Equations**
  - mean service time at resource K_i
    - $S_i = B_i / C_i$
  - utilization of resource K_i
    - $U_i = B_i / T$
  - throughput (completions) at resource K_i during T
    - $X_i = C_i / T$
  - λ_i , the arrival rate at resource K_i during T
    - $\lambda_i = A_i / T$
  - system thruput
    - $X_0 = C_0 / T$
  - visits per request at resource K_i
    - $V_i = C_i / C_0$

Operational analysis

- Example:
  - $T = 60$ seconds
  - $K = 1$ resource
  - $B_1 = 36$ seconds
  - $A_1 = A_0 = 1800$ requests
  - $C_1 = C_0 = 1800$ requests

- Basic Equations
  - mean service time at resource K_i
    - $S_i = B_i / C_i$
  - utilization of resource K_i
    - $U_i = B_i / T$
  - throughput (completions) at resource K_i during T
    - $X_i = C_i / T$
  - λ_i , the arrival rate at resource K_i during T
    - $\lambda_i = A_i / T$
  - system thruput
    - $X_0 = C_0 / T$
  - visits per request at resource K_i
    - $V_i = C_i / C_0$

Operational analysis

- Example:
  - T = 60 seconds
  - K = 1 resource
  - B_1 = 36 seconds
  - $A_1 = A_0$ = 1800 requests
  - $C_1 = C_0$ = 1800 requests

$S_1 = B_1 / C_1 = 36 / 1800 = 20$ ms.

$U_1 = B_1 / T = 36 / 60 = 60\%$

$\lambda_1 = A_1 / T = 1800 / 60 = 30/\text{sec}$

$C_1 = C_1 / T = 1800 / 60 = 30/\text{sec}$

- Basic Equations
  - mean service time at resource K_i
    - $S_i = B_i / C_i$
  - utilization of resource K_i
    - $U_i = B_i / T$
  - throughput (completions) at resource K_i during T
    - $X_i = C_i / T$
  - λ_i , the arrival rate at resource K_i during T
    - $\lambda_i = A_i / T$
  - system thruput
    - $X_0 = C_0 / T$
  - visits per request at resource K_i
    - $V_i = C_i / C_0$

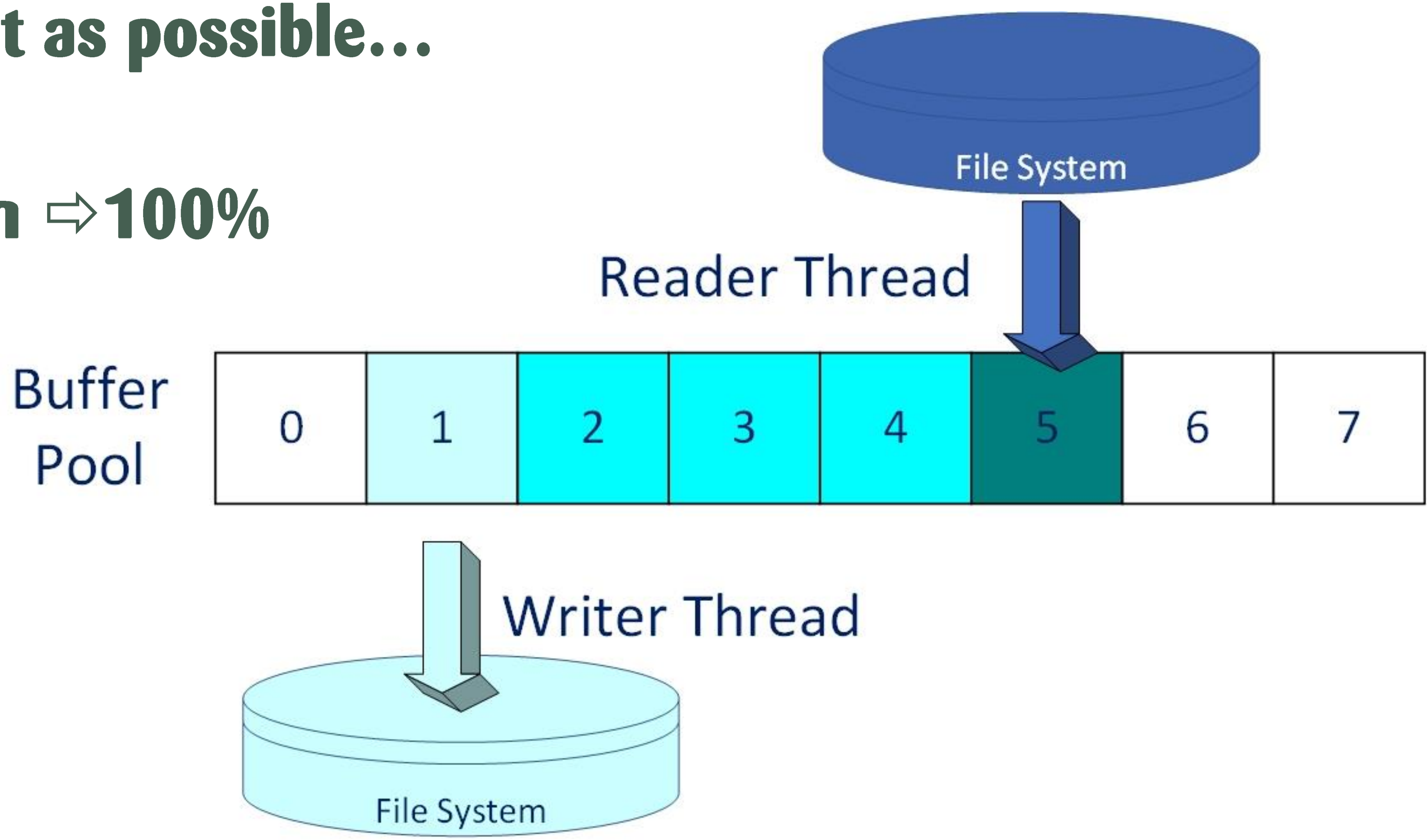
Utilization Law:

$$u = \lambda * \bar{s}$$

- Service time is also frequently called the average *latency*
- Utilization (% busy) is a value between 0 and 1.
  - no device can be utilized more than 100%
  - a device can be driven to 100% utilization if it is (carefully) *scheduled*...

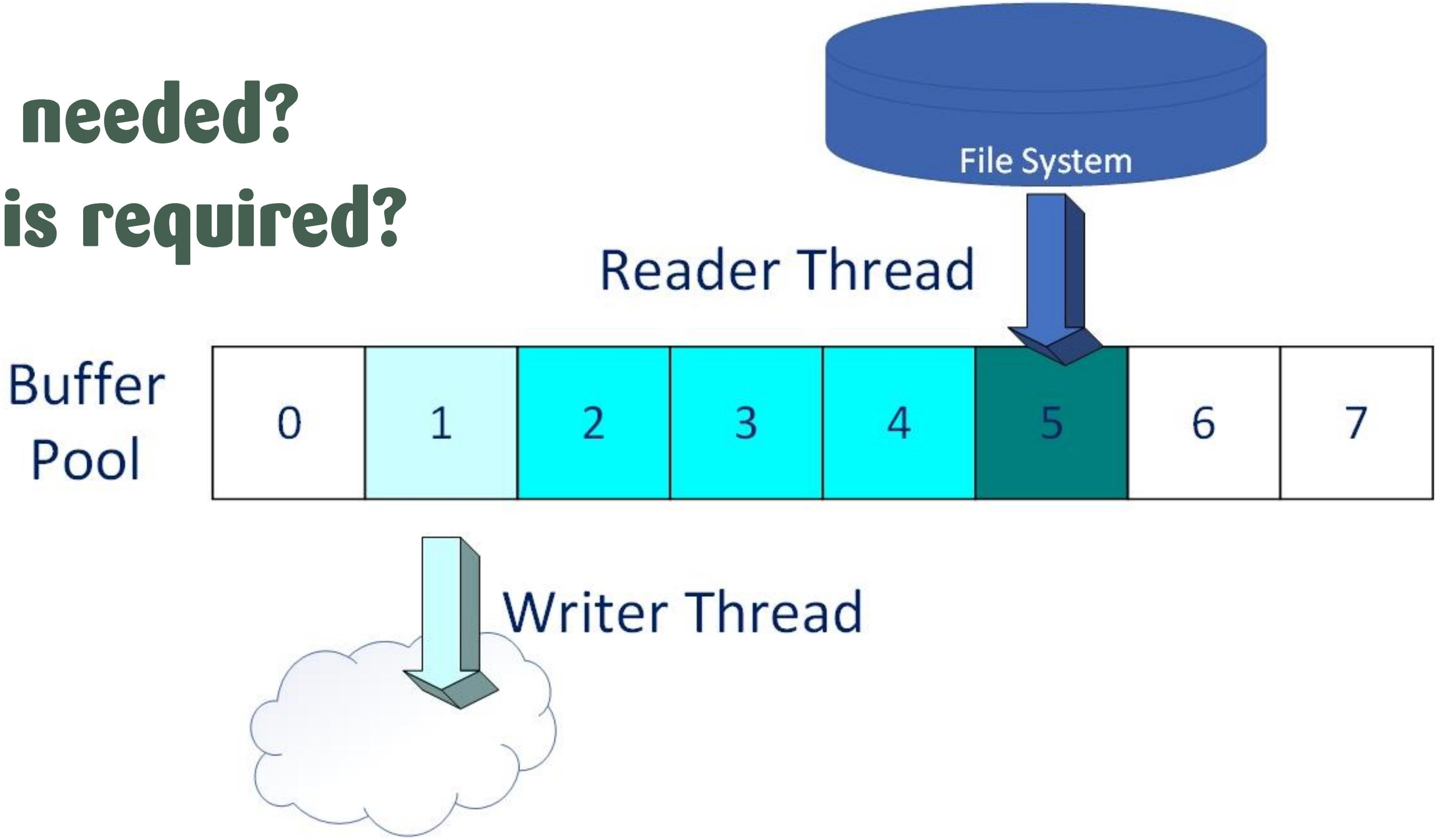
Utilization Law

- Consider the problem of copying a file from one disk to another as fast as possible…
- Goal: Drive disk utilization ⇨100%



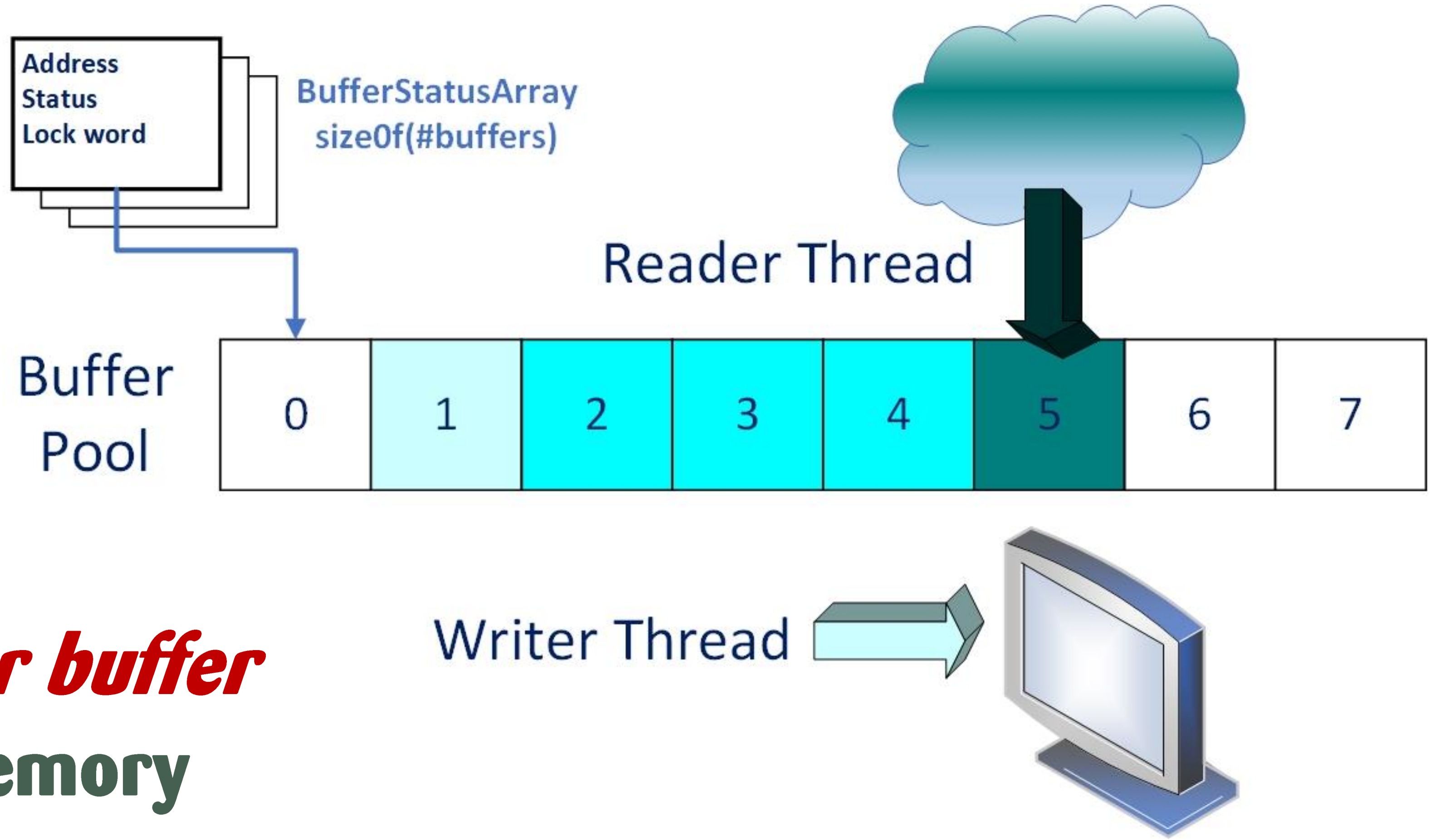
Utilization Law

- What if you are copying a file from one disk to a location in the **cloud**...
  - How many buffers are needed?
  - What synchronization is required?



Utilization Law

- In general, what if the Reader and Writer speeds are mis-matched?
  - e.g.,
  - streaming video



  - plus, use a *circular buffer* to save space in memory

Capacity

- Since no device can be utilized more than 100%, at (or near) 100% utilization, a resource has reached its ***capacity*** limit.
  - Throughput
  - Bandwidth

- Consider some common types of computer hardware and their ***finite*** capacity limits:
  - Processor (CPU)
  - Memory
  - Disk
  - Network adapter/endpoint

Capacity

- **Computer resources have finite *capacity* limits:**

Component	Performance, Capacity, or Bandwidth
CPU	Clock speed; Instructions executed/clock
Memory	Access time (nanoseconds); bus bandwidth
Rotating Disk	Access time (milliseconds)
Solid State Disk	Access time (microseconds)
Network adapter	Bandwidth; Latency

Bounds on performance

- Consider a computer servicing requests at a rate *= 13,680 /hour*

Disk	Reads/sec	Writes/sec	IOPS	Utilization
1	24	8	32	0.30
2	28	8	36	0.41
3	40	10	50	0.54

Bounds on performance

- Consider a computer servicing requests at a rate *= 13,680 /hour*

Disk	Reads/sec	Writes/sec	IOPS	Utilization
1	24	8	32	0.30
2	28	8	36	0.41
3	40	10	50	0.54

- Calculate the average service time at each disk…

Bounds on performance

- Consider a computer servicing requests at a rate = *13,680 /hour*

Disk	Reads/sec	Writes/sec	IOPS	Utilization
1	24	8	32	0.30
2	28	8	36	0.41
3	40	10	50	0.54

- average service time = utilization / thruput

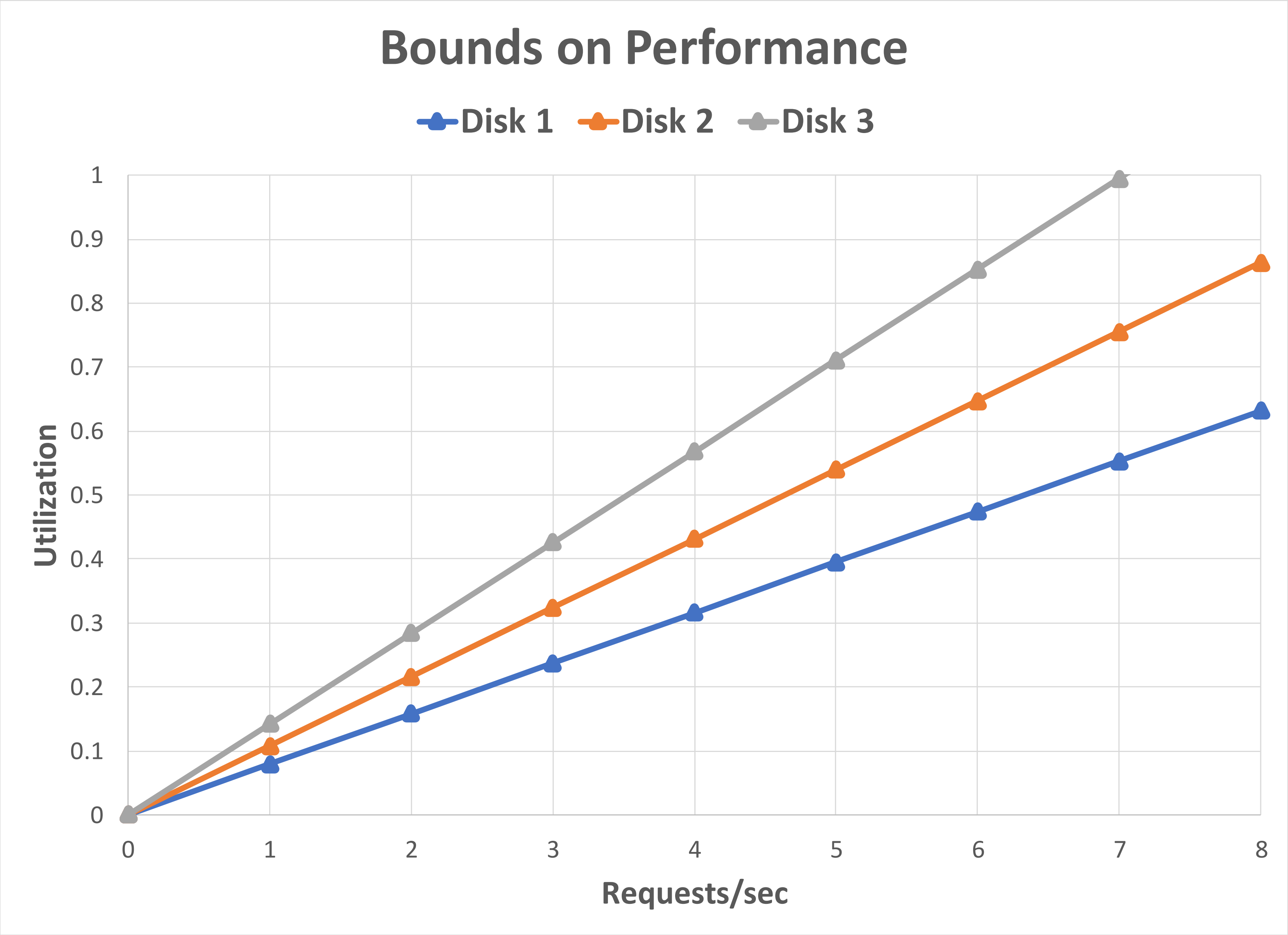
Bounds on performance

- Consider a computer servicing requests at a rate *= 13,680 /hour*

Disk	Reads/sec	Writes/sec	IOPS	Utilization	Ave Service Time (ms)
1	24	8	32	0.30	9.4
2	28	8	36	0.41	11.4
3	40	10	50	0.54	10.8

- average service time = utilization / thruput

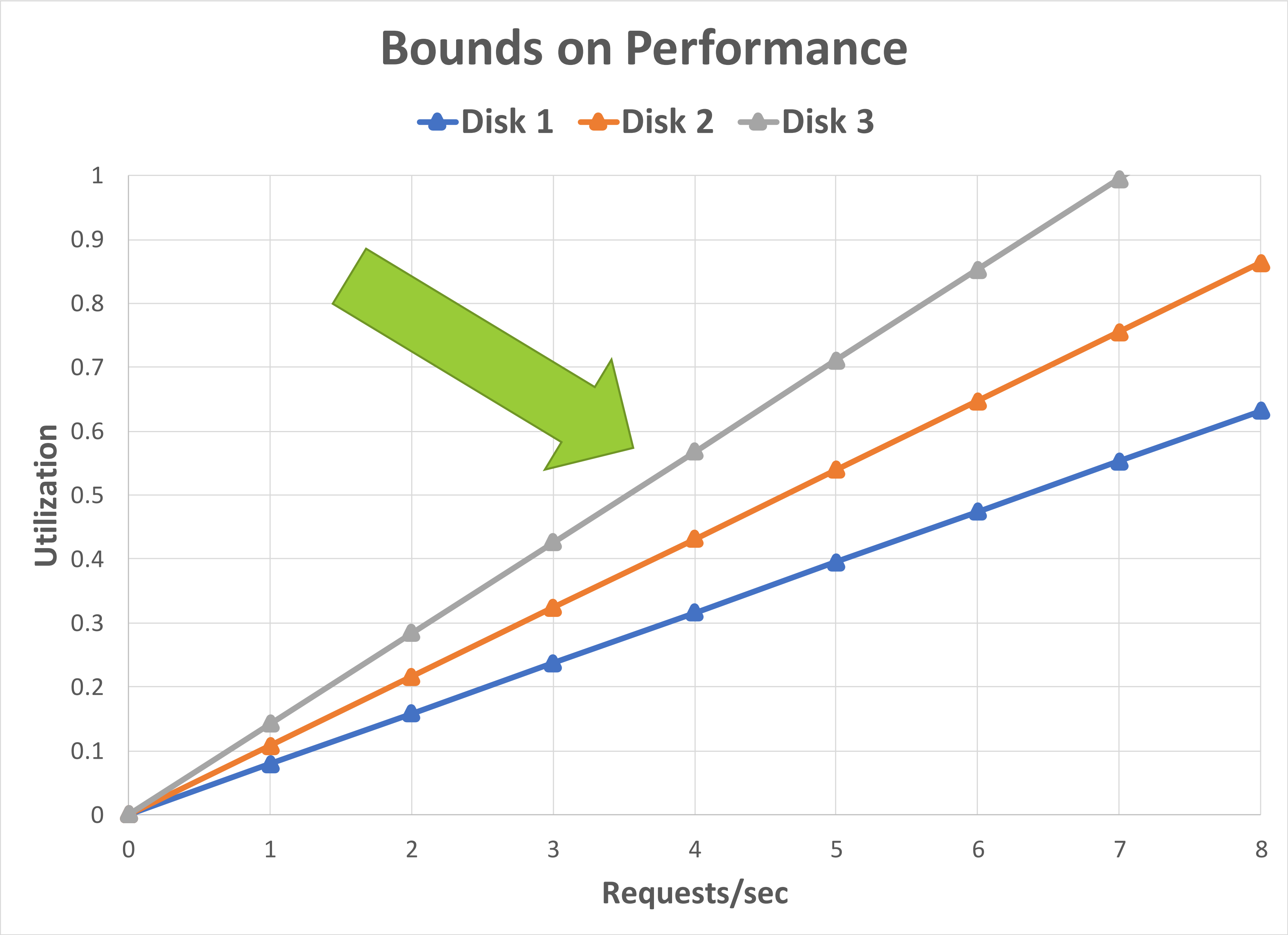
Assuming that the load on each device grows as a *linear function* of the Request rate:



When Disk 3, saturates, the system is at its maximum capacity

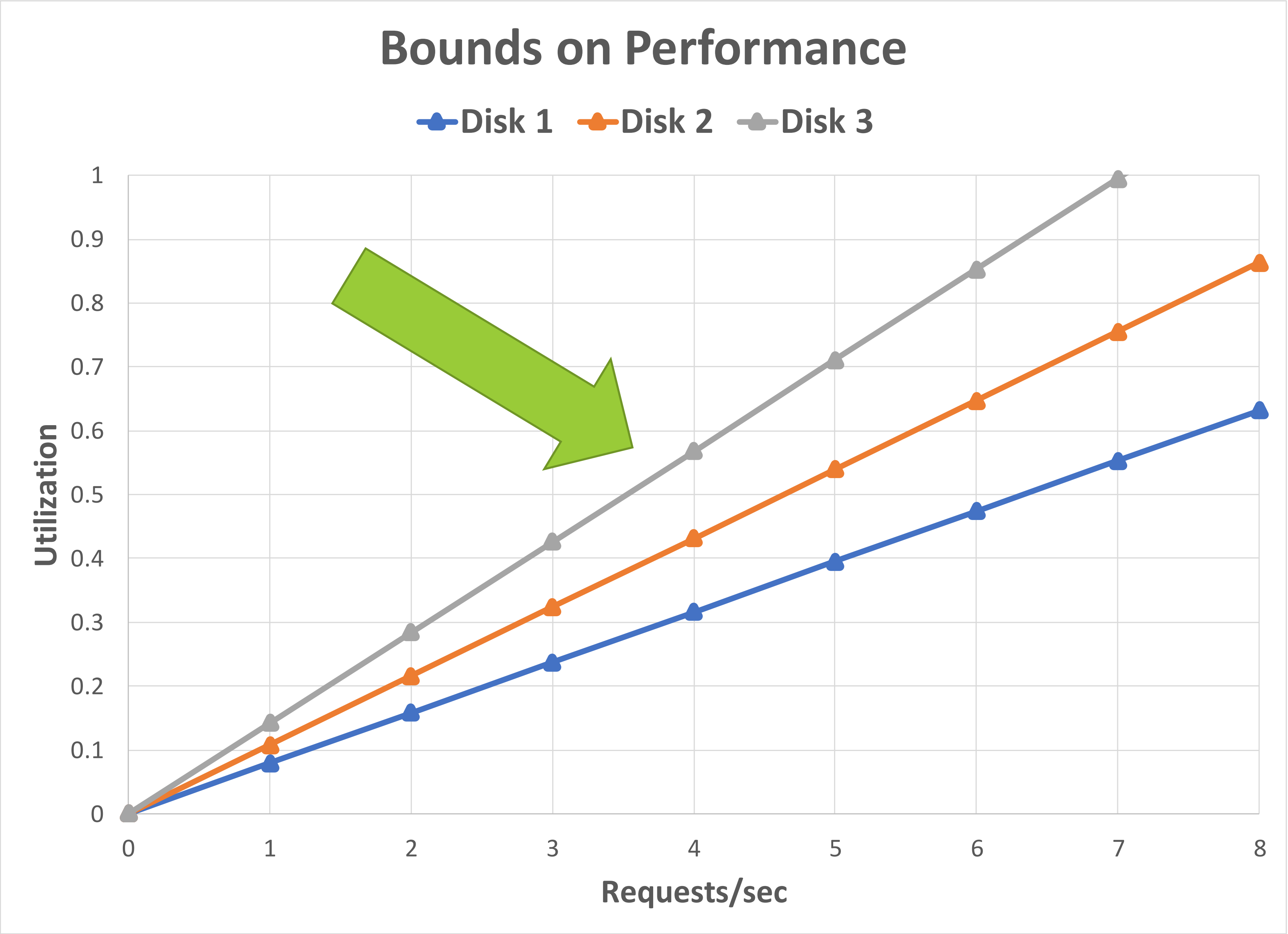
Upper bound **on throughput under heavy load**

Disk 3 is the *bottleneck* device



What happens when you replace Disk 3 with a faster SSD?

Disk 2 becomes the next *bottleneck* device



Bottleneck analysis

1. Find the bottleneck device and fix, improve or remove it.
2. Increase the workload until another bottleneck emerges
3. Repeat Step 1

- **Decomposition**: break down Request processing into smaller sub-components whose performance you can also measure
  - Bottlenecks are not always hardware components
  - Not all subcomponents are instrumented
  - Linear scaling is seldom achievable

Response Time

- Whenever there are multiple customers issuing independent Requests for service to the *same* server (or resource), there is the possibility of *contention*.
-
- When a Request encounters a busy server, the Request is (usually) queued for service.

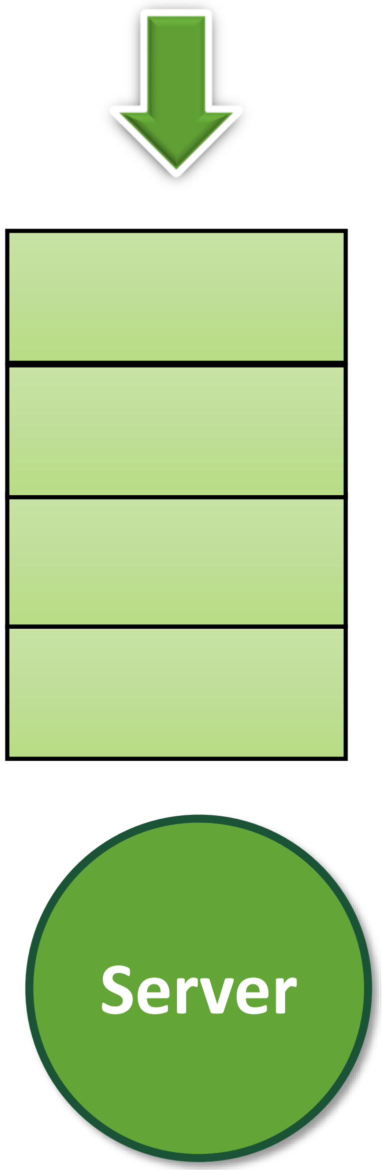
$$R = \overline{W}_s + \overline{W}_q$$

Response Time =

mean Service Time + mean Queue Time

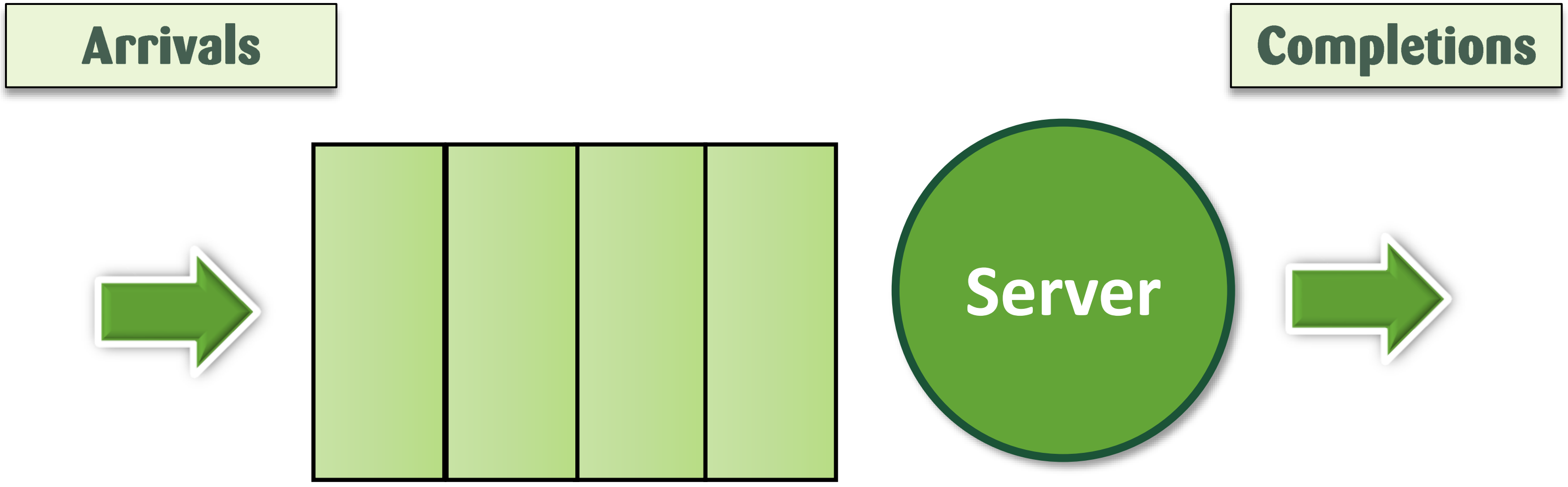
Queue Time

- Independent requests for service from a shared resource lead to ***contention***
- The ***amount*** of contention is a function of
  - how busy the server is
  - variability in the arrival rate of requests
  - variability in the service time
- A Request that encounters a busy server is queued



Queue Time

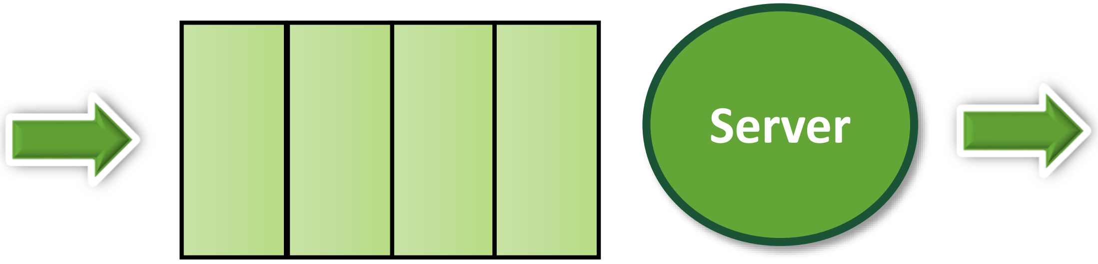
- Elements of a queueing system
  - Arrivals
  - Completions
  - Server
  - Queue



Response Time = Queue Time + Service Time

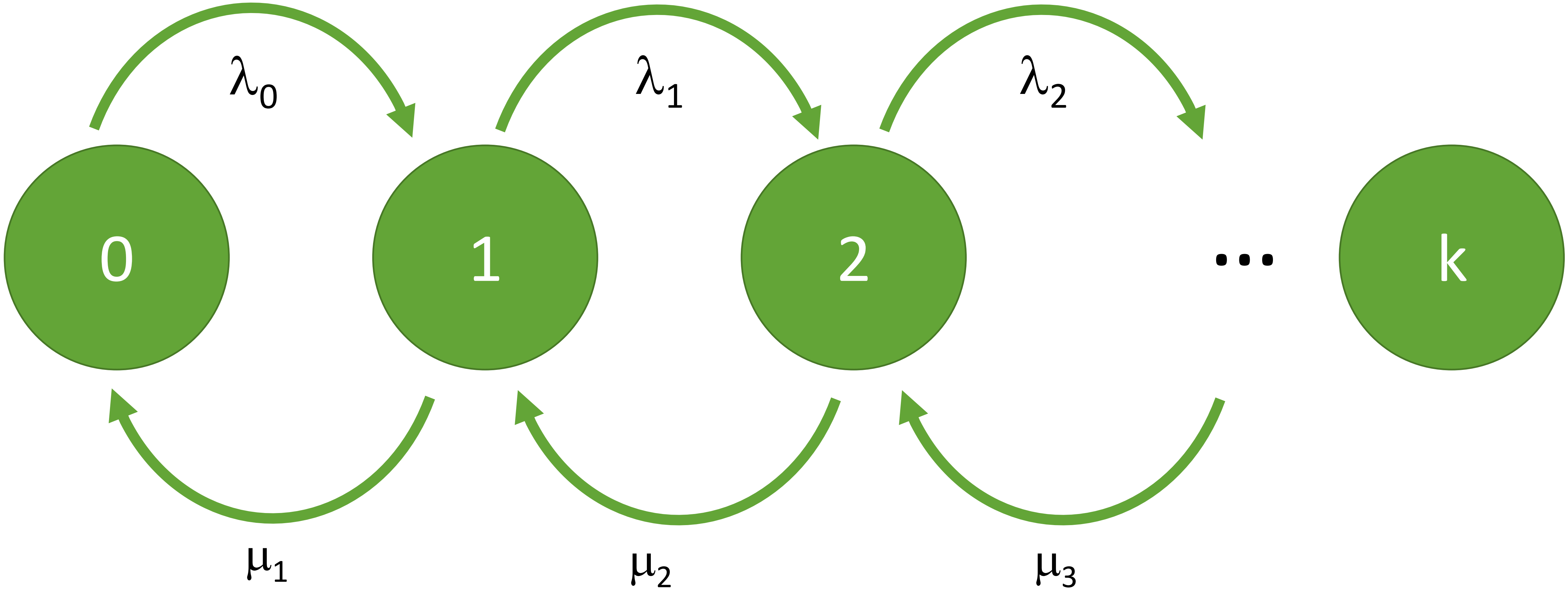
Queue Time

- How long a Request that is queued waits is a function of
  - # of Requests already waiting &
  - the service times of those Requests



- Familiar examples of queueing systems
  - Fast Food restaurant
  - Customs check at a border crossing
  - Company cafeteria at lunch time
  - Waiting for a bus or a ferry ride
  - Checking in at an airport
  - Checking out of a supermarket

Generalized Birth-Death Markov Models

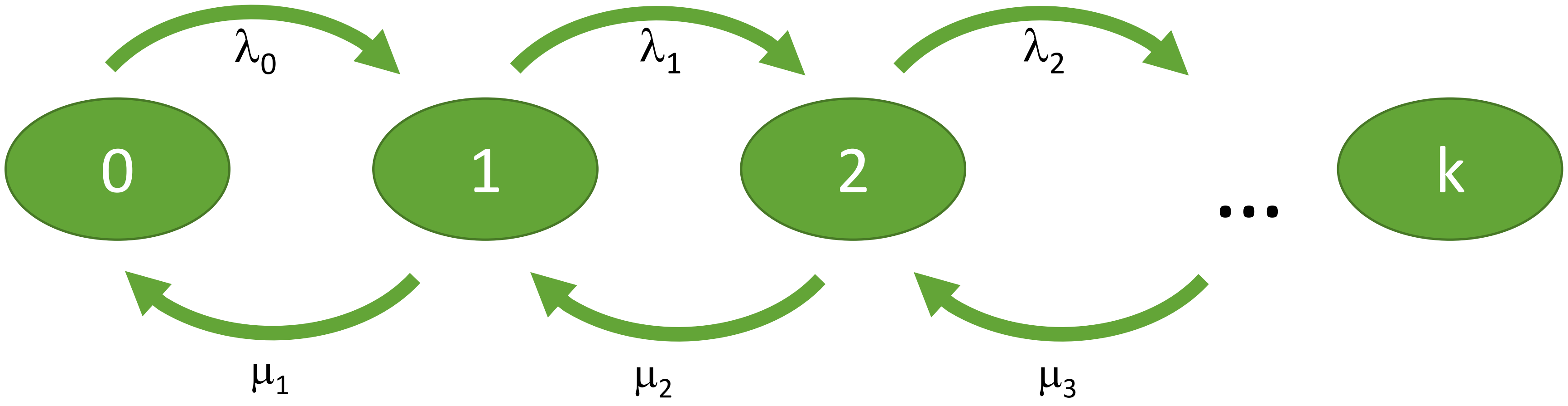


Generalized Birth-Death Markov Models (Erlang)

utilization = $1 - P_0$

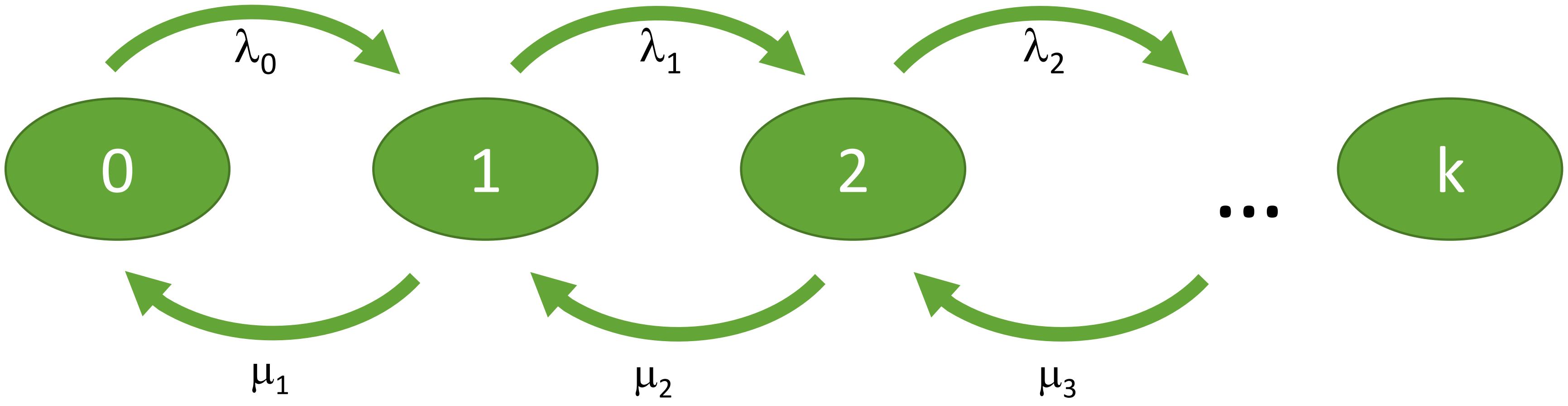
throughput = $\sum_{k=1}^{\infty} \mu_k P_k$

queue length = $\sum_{k=1}^{\infty} k P_k$



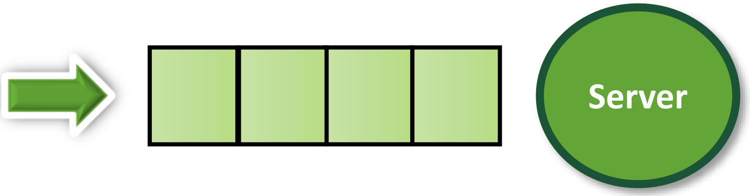
Generalized Birth-Death Markov Models (Erlang)

- Intuitively,
  - How long a customer waits for a service in a Queue is a function of:
    - the customer's position in the queue
    - service times for the Requests of customers ahead of you in the

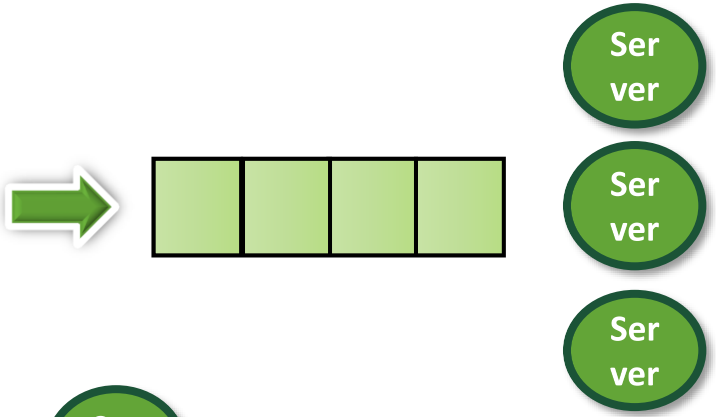


Types of Queues

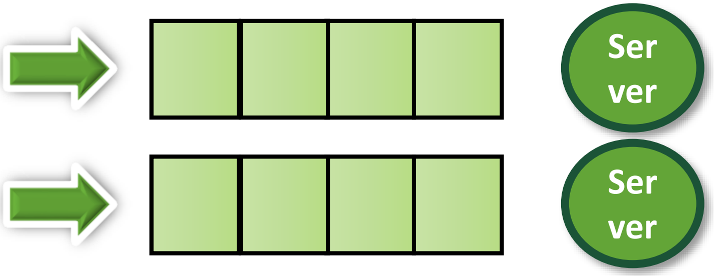
- Single Server



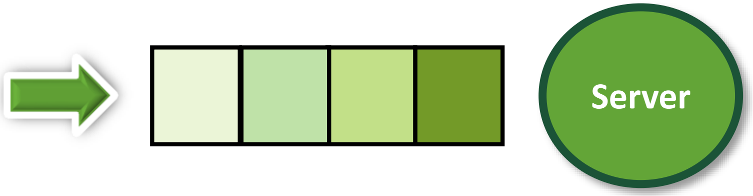
- Multiple Servers



- Multiple classes of service



- Queueing discipline:
  FCFS, round-robin or priority



Little's Law

- Equivalence relationship involving
  - L, the average number of customers waiting in a queueing system
    - i.e., the **Queue length**
  - λ, the rate customers arrive to request service
    - assume λ = C, the completion rate (the ***equilibrium*** assumption)
  - W, the average amount of time customers wait in the system
    - i.e., the **Response Time**

$$L = \lambda * W$$

Little's Law

$$L = \lambda * W$$

- N, the number of customers in the system
  = Throughput * Response Time

- Common applications of Little's Law include measuring two of the variables and calculating the 3rd term

Little's Law

$$L = \lambda * W$$

- Example
  - A Fast Food restaurant takes orders from 720 customers/hour during lunch. Processing an order takes an average of 90 seconds. How big does the waiting room need to be?

  - N = (720 / 3600) * 90 = 0.2 * 90 = 15 customers

Assignment

- Prove Little's Law
  - Due prior to class next week.

Class exercise

- Navigate to the PDQ (Pretty Damn Quick) info page
- PDQ Software Distribution page
- and follow the instructions to install the PDQ library for use with Perl, Python, C or R
  - http://www.perfdynamics.com/Tools/PDQcode.html
  - open source: see https://sourceforge.net/projects/pdq-qnm-pkg/
- Test your install by executing the sample script at section 4.2 (PDQ Model in Perl)

Queueing Models

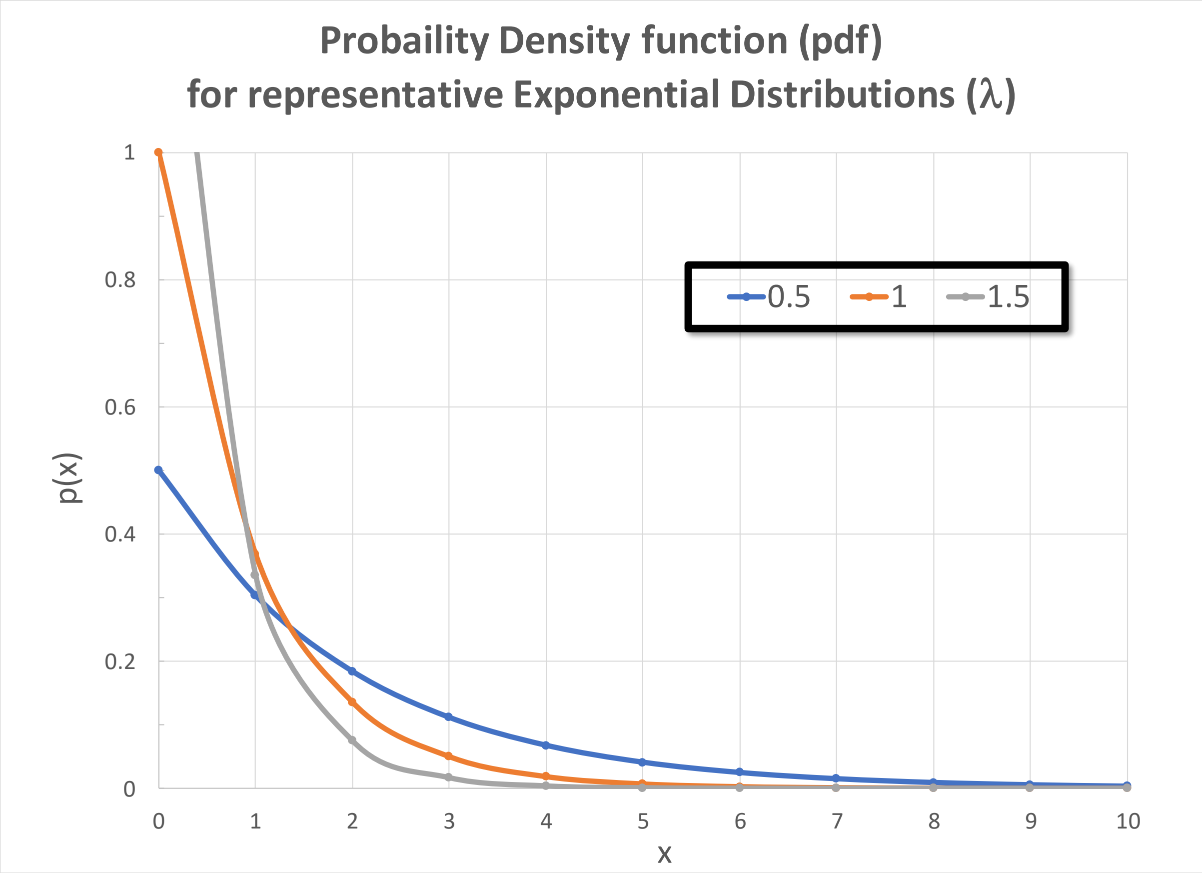
- In general, use Markov chains to characterize a queueing system based on
  - the Arrival rate distribution
  - the Service time distribution
  - the number of servers
  - Notation:

$$G/G/n$$

M/M/1 Queue

- The arrival rate is exponential
- The service time is exponential
- 1 server

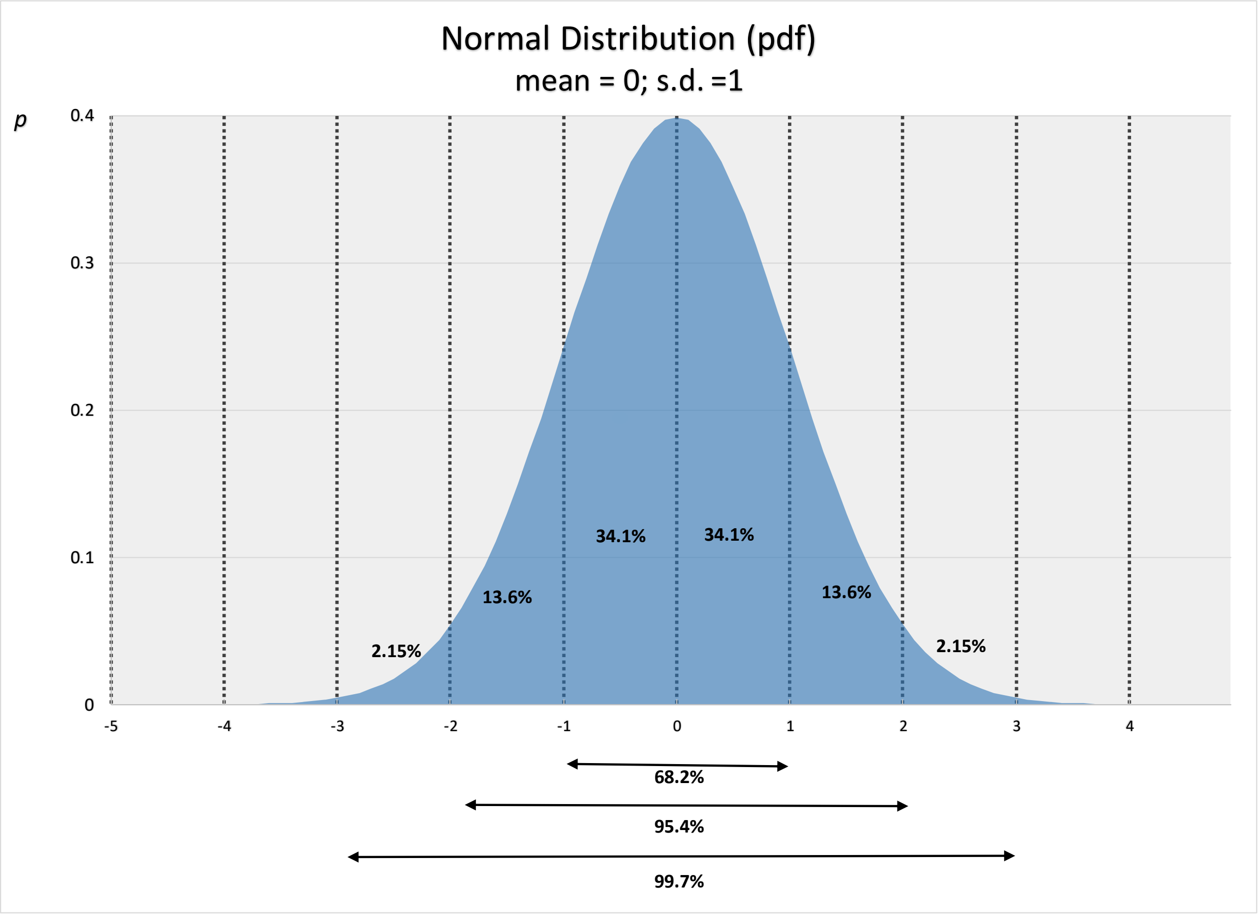
- What is an exponential distribution?
  - probability density function = $\lambda e^{-\lambda x}$
  - mean = standard deviation = $1/\lambda$
  - memoryless
  - models the time between arrivals in a Poisson process
    (i.e., independent, randomly-occurring discrete events)

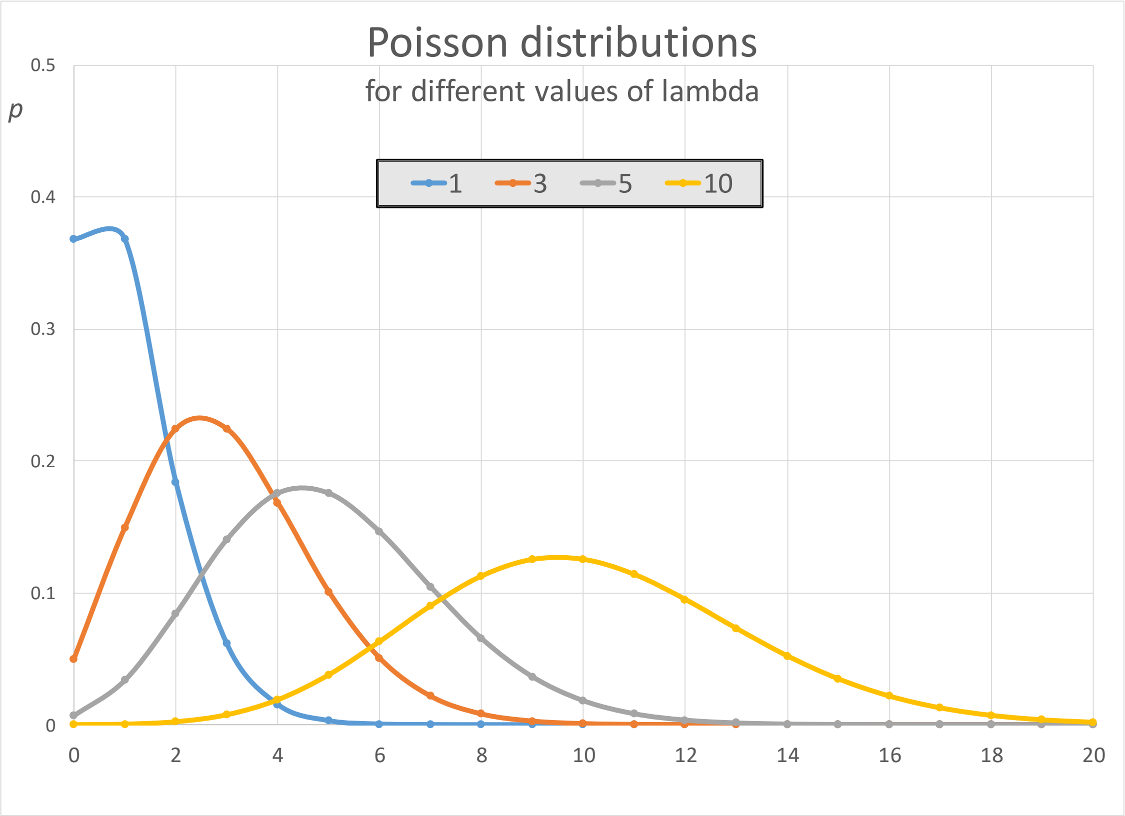


right-hand exponential

Normal vs. Poisson distributions

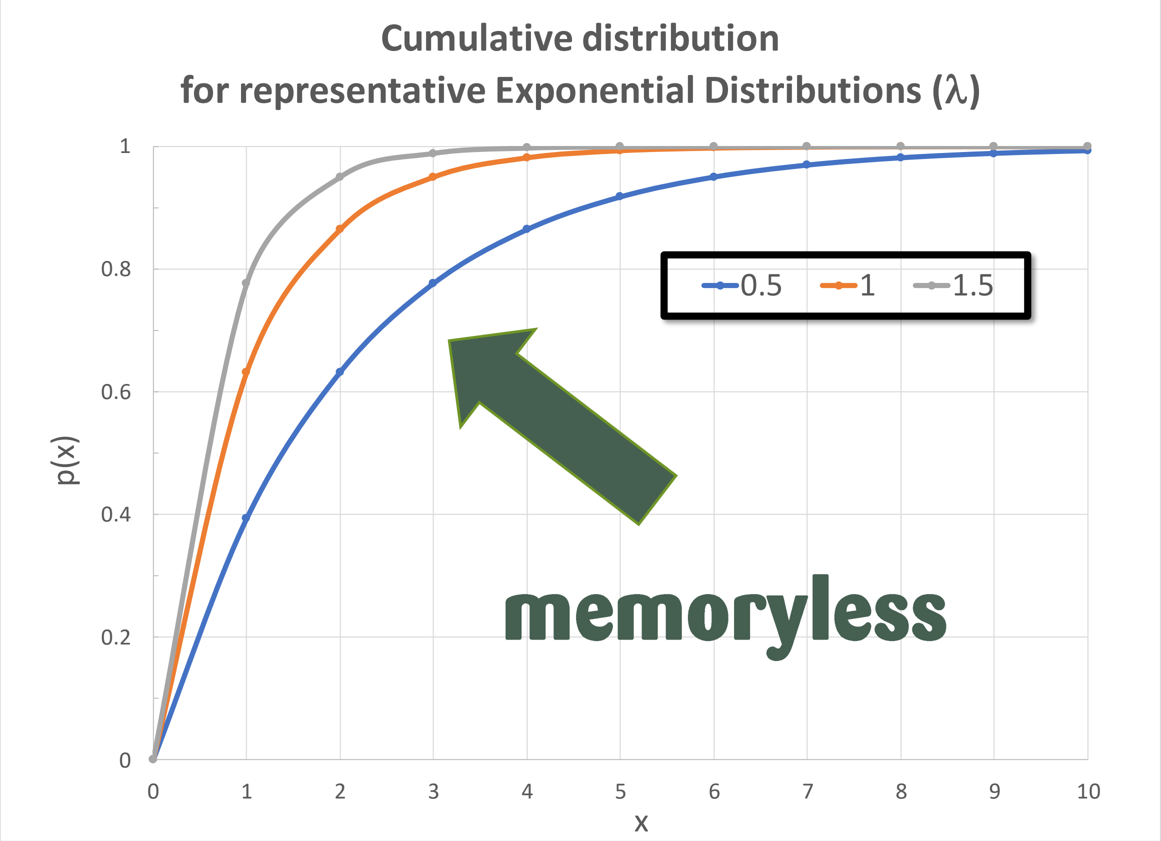
Normal	Poisson





M/M/1 Queue

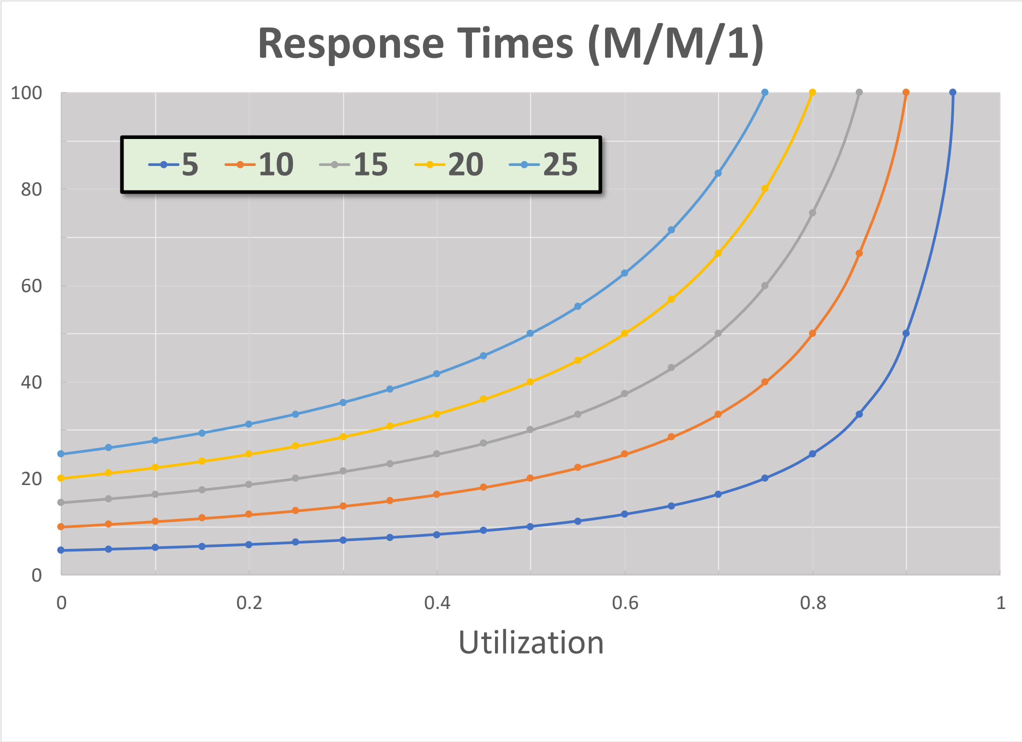
- **Formulas**
  - $N = {}^{p}/_{1-p}$
  - $RT = {}^{S}/_{1-p}$
  - where ρ is the probability the server is busy
  - (Note: ρ = utilization)



M/M/1 Queue

$$RT = {}^{S}/_{1-p}$$

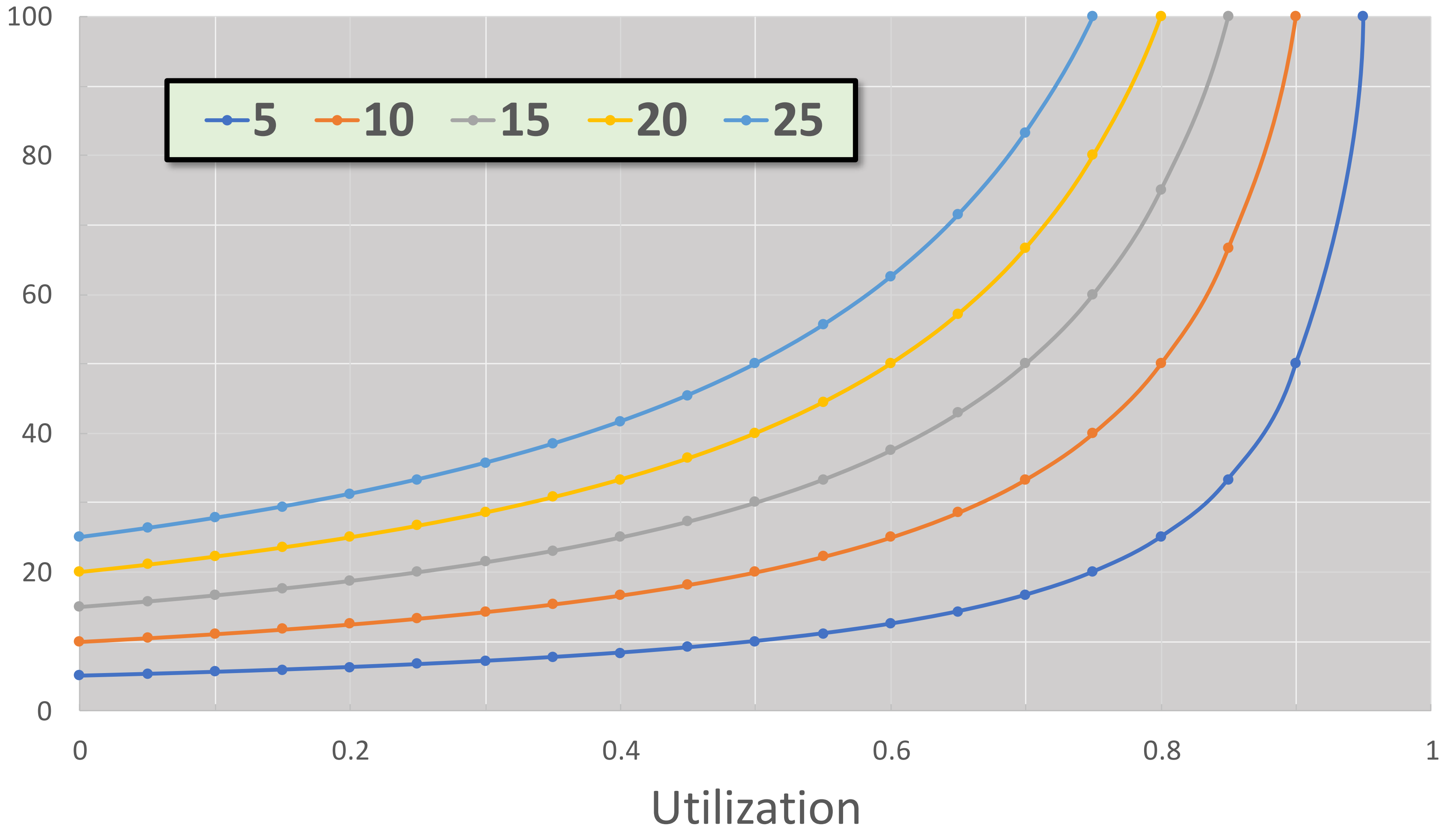
- **Calculate RT, if the average service time and the server utilization are known**
  - Note: **S = u / λ**, from the Utilization Law
- **How realistic are the assumptions?**
  - exponential arrivals:
    - are arrivals independent?
    - is the $mean \cong standard\ deviation$?
  - exponential service time



e.g.,

Calculate RT for disks with mean service time = 5-25 ms.

Response Times (M/M/1)

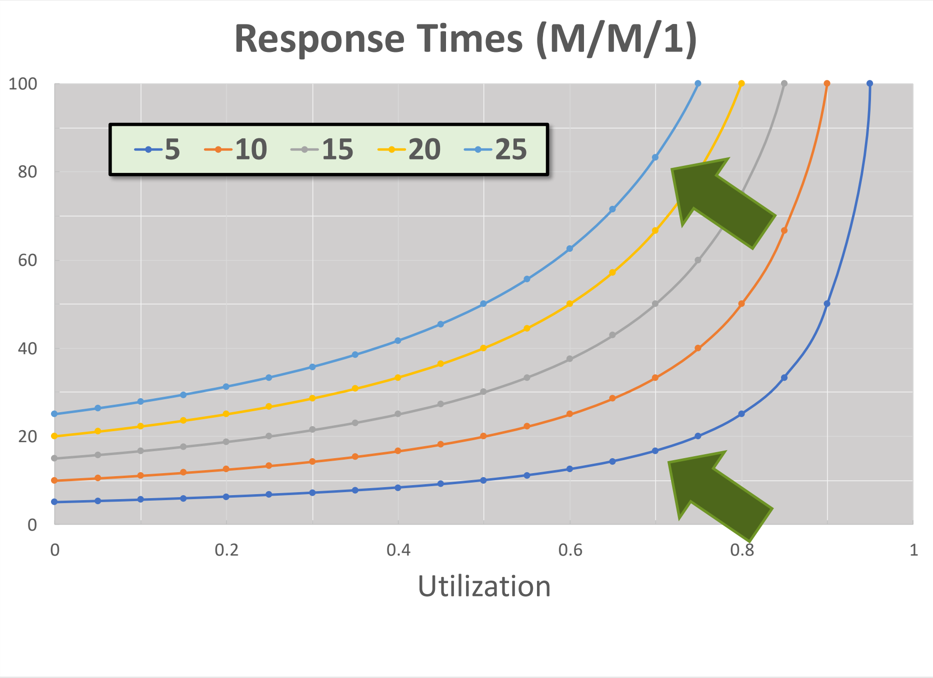


Discussion

1. What is the **shape** of the m/m/1 response time distributions?
2. When does a gradual **quantitative** change manifest a **qualitative** change?
3. What happens when u = 1?
4. When is RT = 2 * S
5. Is there a "knee" of the RT curves?

M/M/1 Queue

- Queueing theory is useful because it models actual system behavior!
  - e.g., Erlang
  - When a bottleneck device nears its saturation point, **small** changes in λ cause **large** changes in performance.
  - Answer "What if…?" questions
    - λ increases by a factor of x
    - substitute a faster server for bottleneck y
    - model the performance of several proposed solutions without having to build them

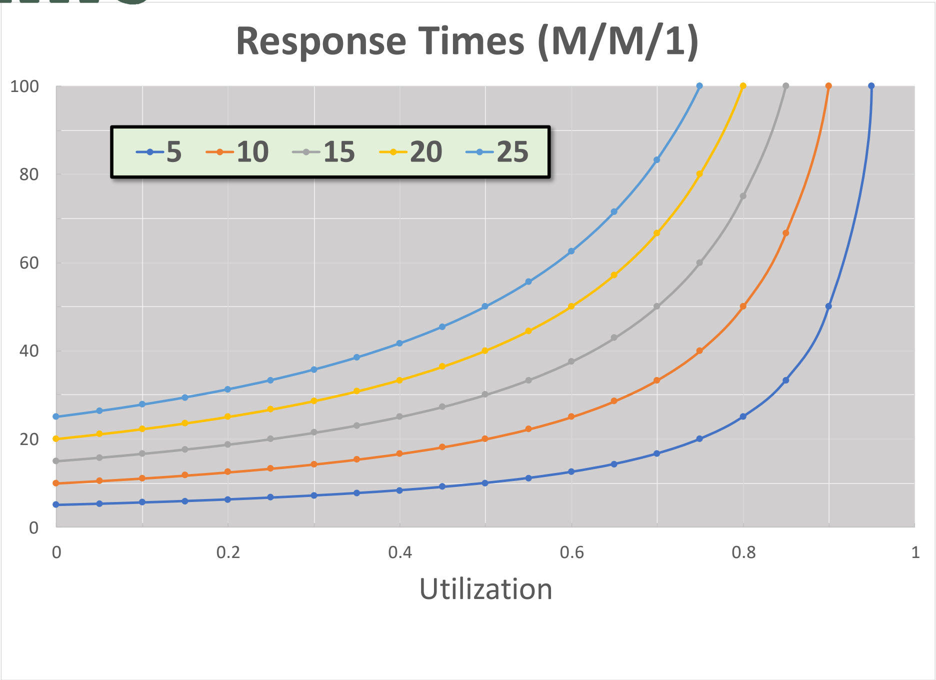


Note: the mathematics breaks down as u ⇨∞

Heavy traffic approximations

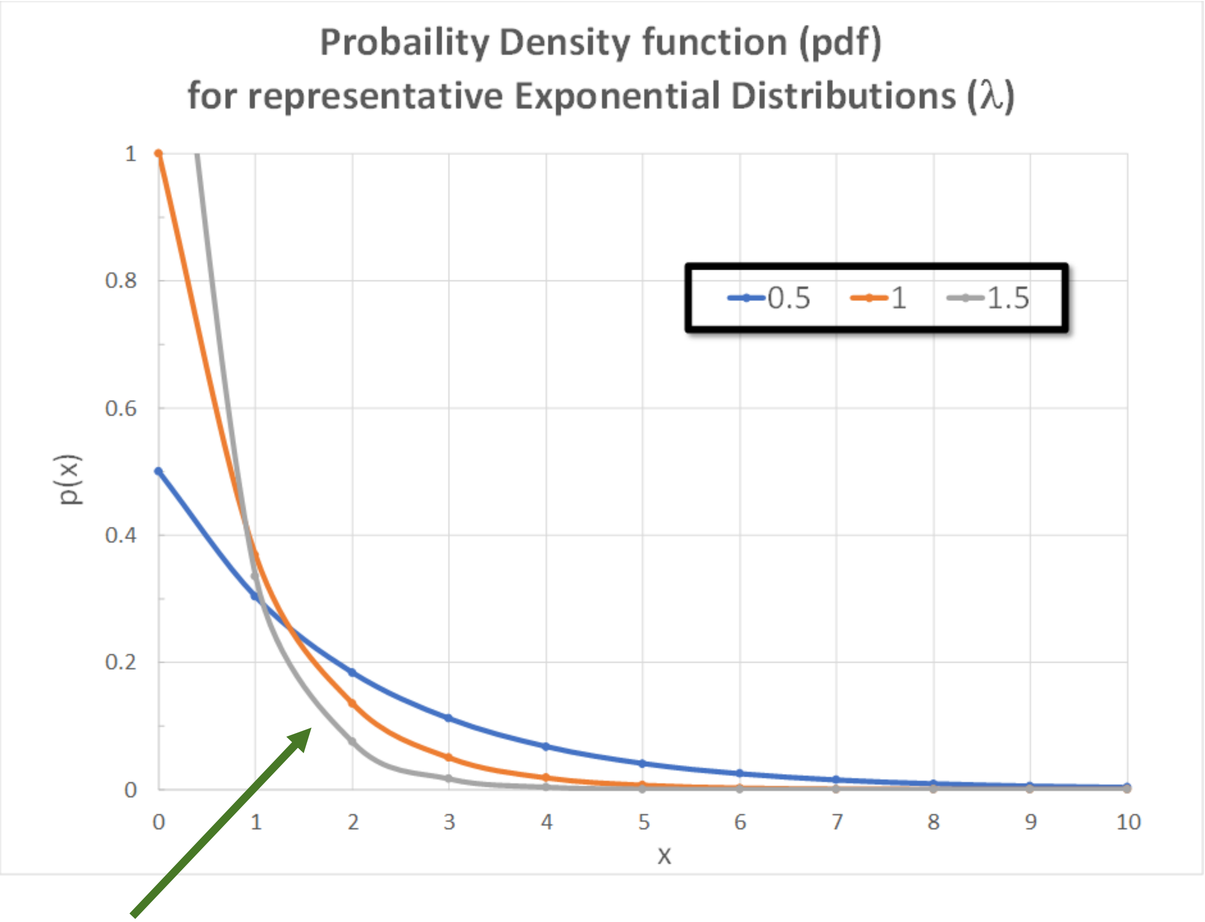
Strategies for reducing Queue Time

- Reduce the variability in the arrival rate
  - Improved scheduling
  - Independent arrivals?
- Improve the service time
  - faster devices; leaner code
- Reduce the variability in the service time
  - M/D/1 compared to M/M/1 has 50% less queueing
  - "D" stands for a deterministic distribution; i.e., sd << mean



Reducing Queue Time

- Reduced variability in the service time distribution
  - M/D/1
    - sd << mean
  - e.g.,
    - time-slicing for sharing processors
    - packet-switching in networks

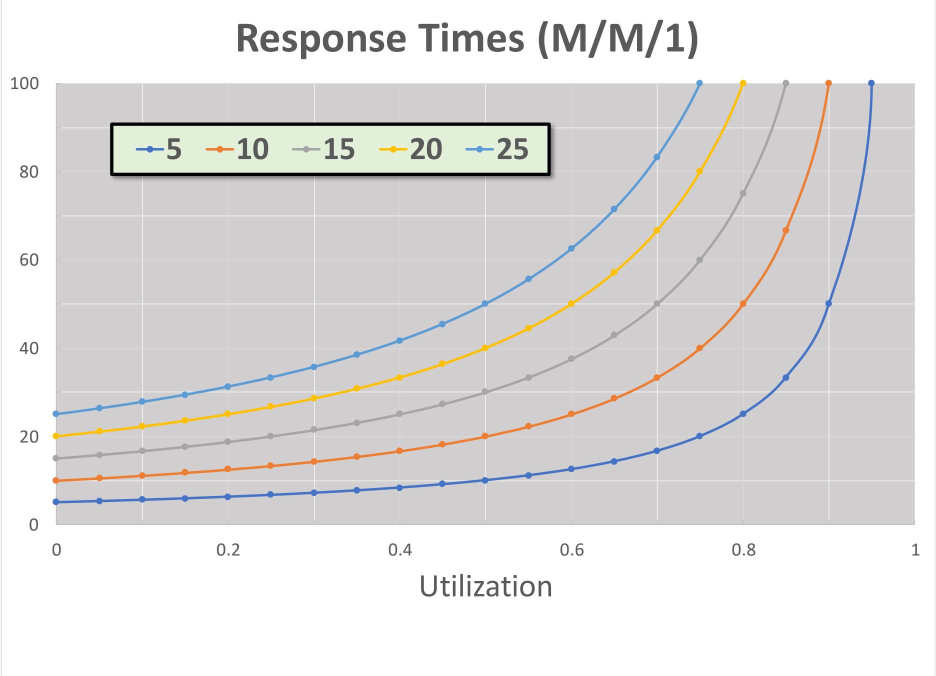


Break large requests into a sequence of smaller, uniformed- size Request packets

Strategies for reducing Queue Time

- Multiple servers
  - **M/M/*n***
  - If all service requests can be processed at any available server
  - ***p***, the probability that the Request will encounter a busy server is the joint probability that all *n* servers are busy

$$p = u^n$$

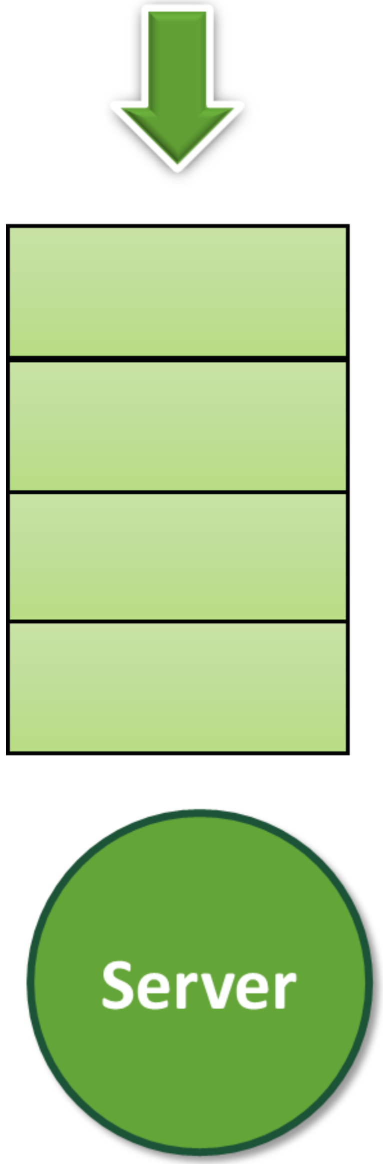


approximately:

$$RT = {}^{S}/_{1-p^{n}}$$

Queuing disciplines

- First Come, First Serve or First In, First Out
  - (FCFS or FIFO)
- Last In, First Out (LIFO)
  - stack
- Time-slicing (Fair)
  - reduces variability in the service time distribution
- Priority (unfair)
  - priority queuing with preemptive scheduling
  - introduces the possibility of starvation, deadlocks



M/G/1

- Service time distributions are less likely to be exponential
  - e.g., Memory access time is constant (M/D/1)
  - e.g., access time of a memory hierarchy (with cache) is bi-modal

  - "G" = general (in effect, any service time distribution)

- Fortunately, there is the PK (Pollaczek-Khinchine) mean value equation:

$$RT = S + \frac{pS(1 + C_s^2)}{2(1-p)}$$

where C_s is the Coefficient of Variation (CoV) of the service time

M/G/1

PK (Pollaczek-Khinchine) mean value equation:

$$RT = S + \frac{pS(1 + C_s^2)}{2(1-p)}$$

where C_s is the Coefficient of Variation (CoV) of the service time

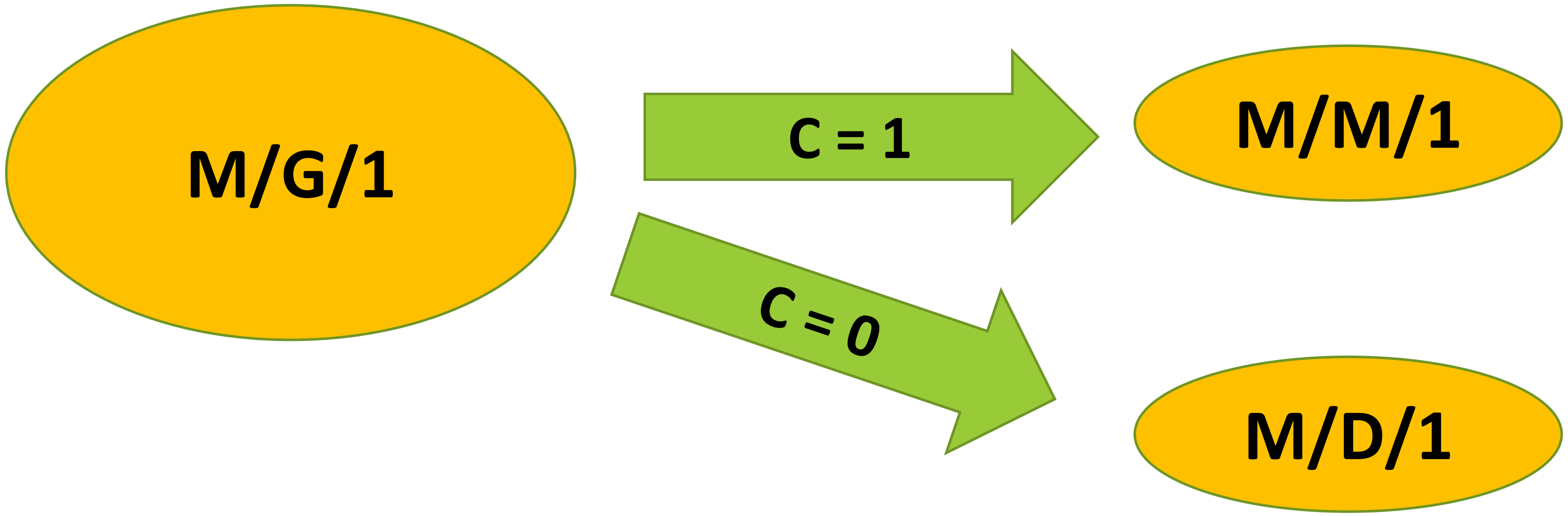
- CoV = σ_s / S
- Deriving the PK mean value equation requires a more accurate assumption about queue time than we have been using so far
  - namely, that a Request that finds a server is busy on average waits only S/2 for the active Request to complete

PK (Pollaczek-Khinchine) mean value equation:

$$RT = S + \frac{pS(1 + C_s^2)}{2(1 - p)}$$

where C_s is the Coefficient of Variation (CoV) of the service time

- Useful whenever C >> 1 (e.g., **bi-modal**, due to cache)

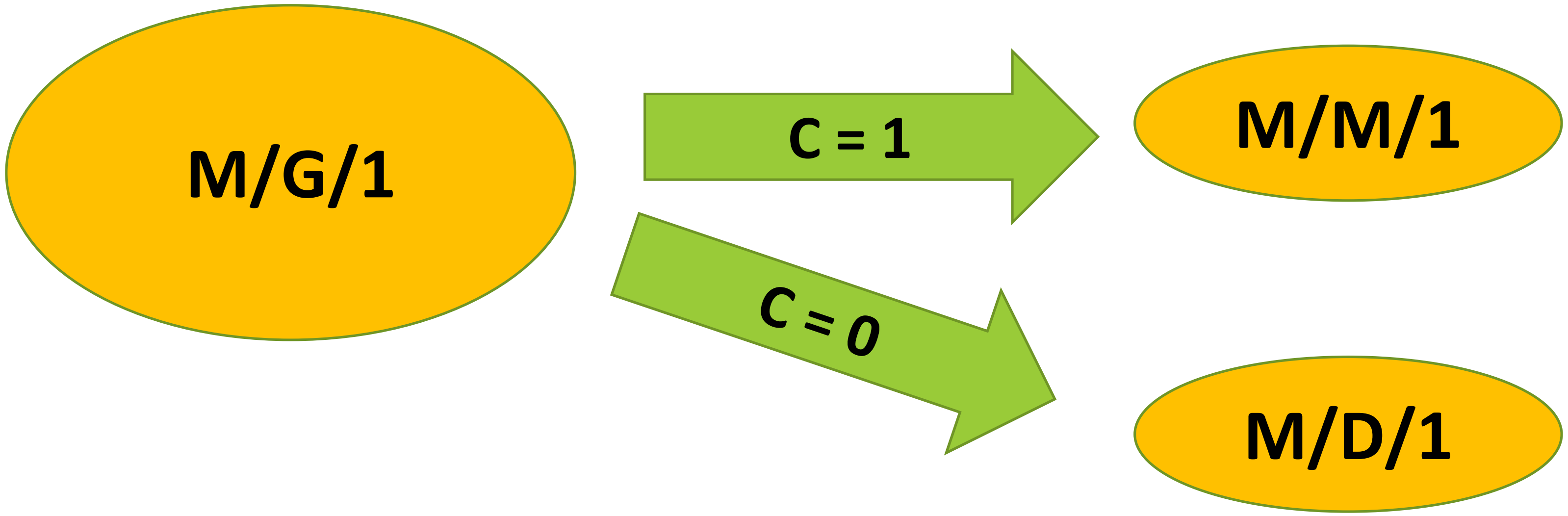


PK (Pollaczek-Khinchine) mean value equation:

$$RT = S + \frac{pS(1 + C_s^{\,2})}{2(1-p)}$$

where C_s is the Coefficient of Variation (CoV) of the service time

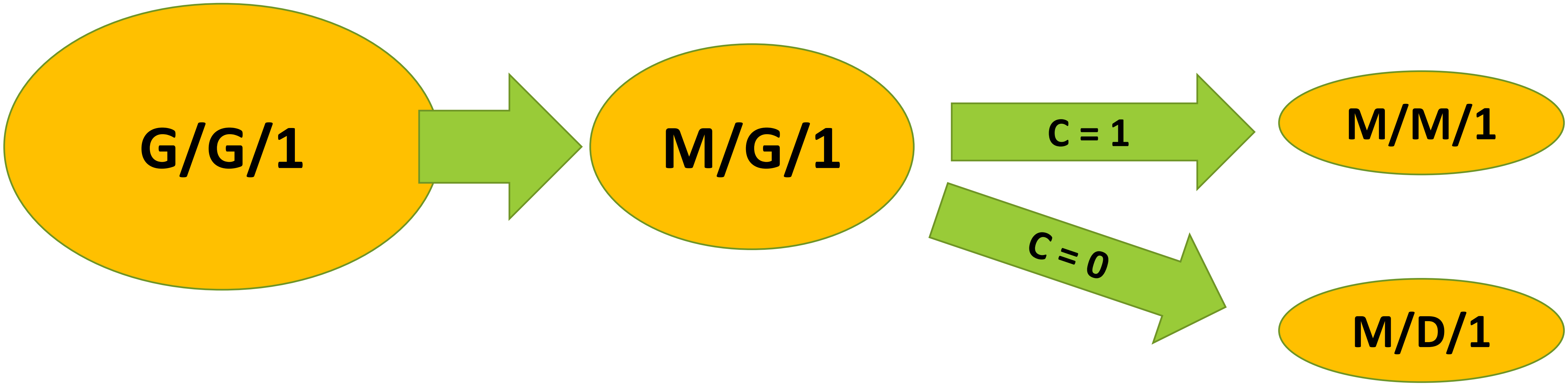
- When C >> 1, Queue time increases more rapidly than M/M/1



G/G/1

- any arrival rate distribution
- any service time distribution

- no practical formulas exist to solve the G/G/1 case!



PK (Pollaczek-Khinchine) mean value equation:

$$RT = S + \frac{pS(1 + C_s^{\,2})}{2(1 - p)}$$

- To enable your component so that Queue times can be calculated, what measurements should you gather?
  - count the arrivals
  - accumulate (i.e., sum) the service time, **S**
  - accumulate the service time squared, **S^2**

- Report λ, **Sum[S]**, and **Sum[S^2]** each measurement Δ to calculate the service time **mean** and **sd** for that corresponding interval
- or measure Queue time (or Response time, since Q = R - S) directly

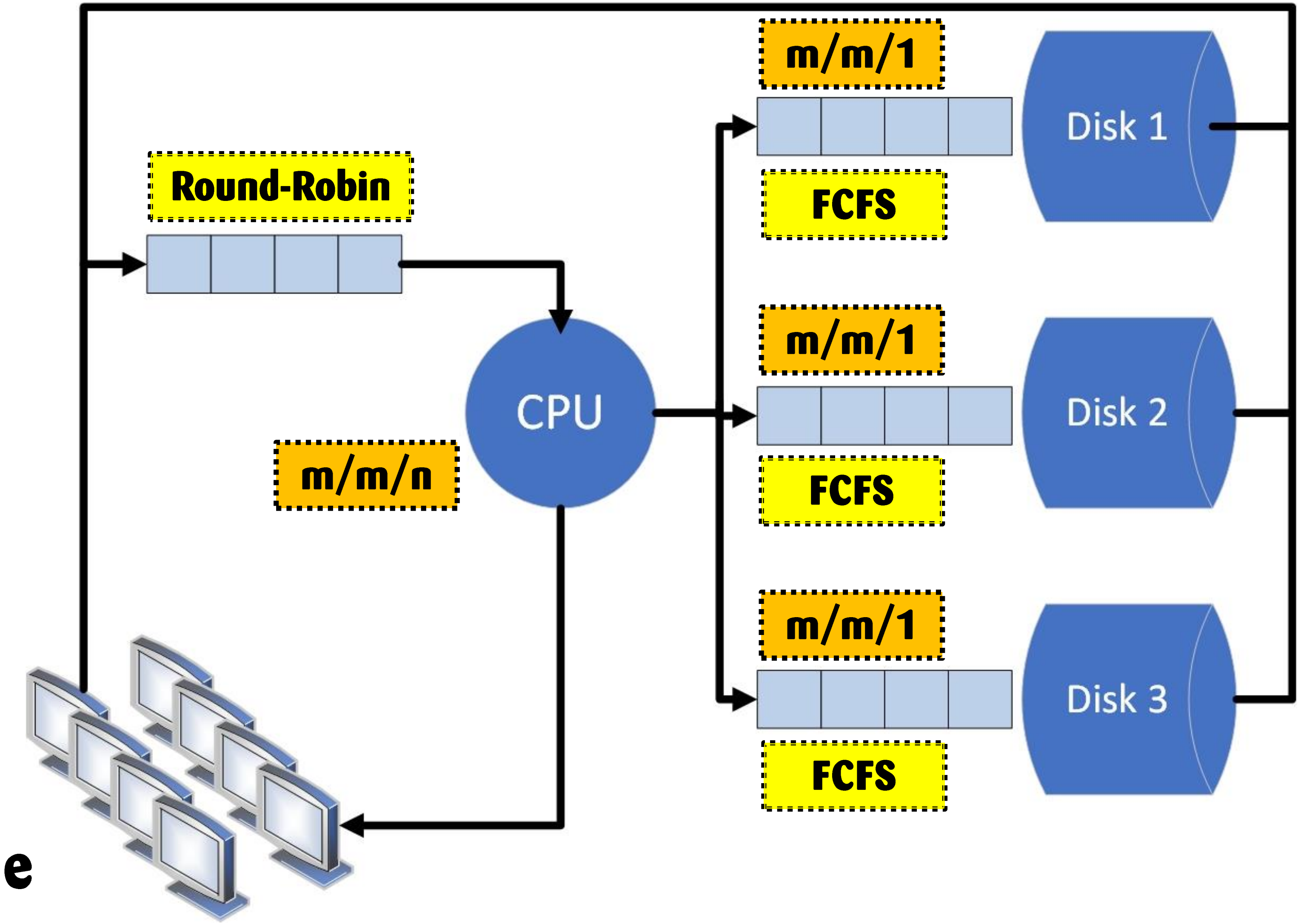
Open and Closed network models

- Applications requiring more than 1 resource can be modeled as a **network** (or **circuit**) of resources and their queues:
  - system resources: CPU, disks, network interface, etc.
  - arrival rates, service times: visits
  - multiple classes of workloads (different arrival rates, service rates, priorities)
  - multiples of systems

- Closed models impose a limit *n*, on the number of concurrent customers
  - Closed network queueing models were used to model interactive workloads on large scale mainframe computers with a fixed number of attached terminals
    - e.g., an internal computer system serving a bank and its workers

Closed network queueing model

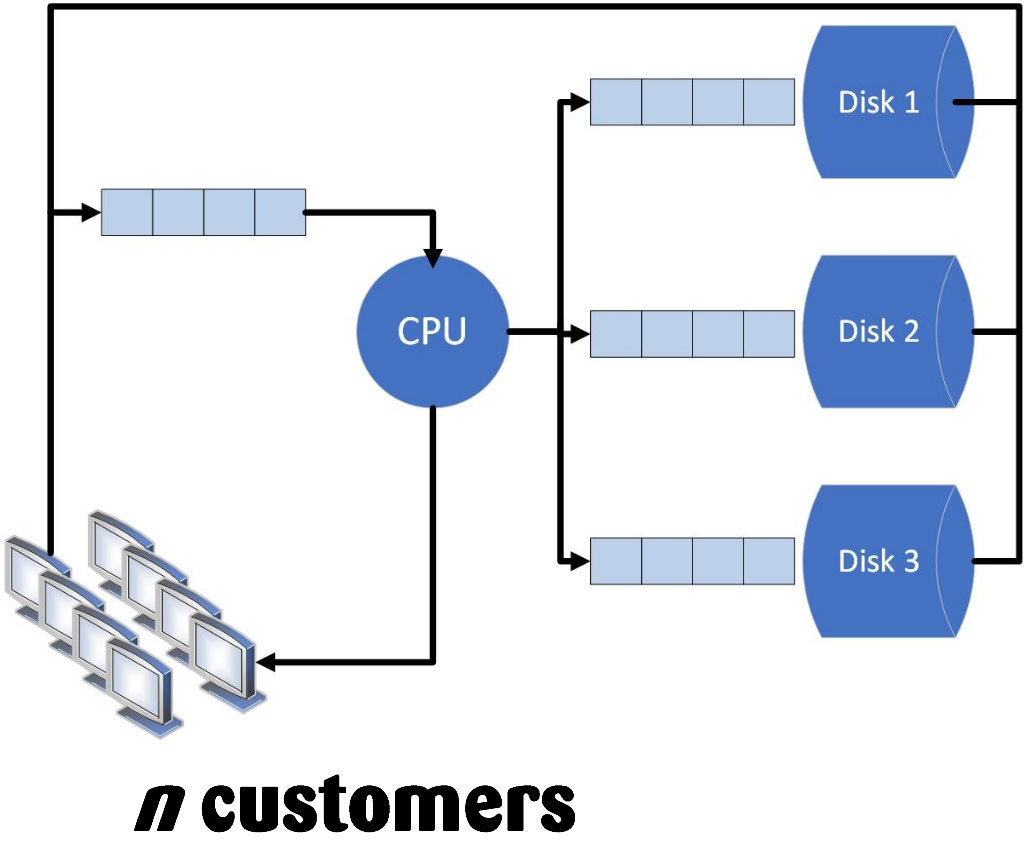
e.g., a Web Server



λ = RT + Think Time

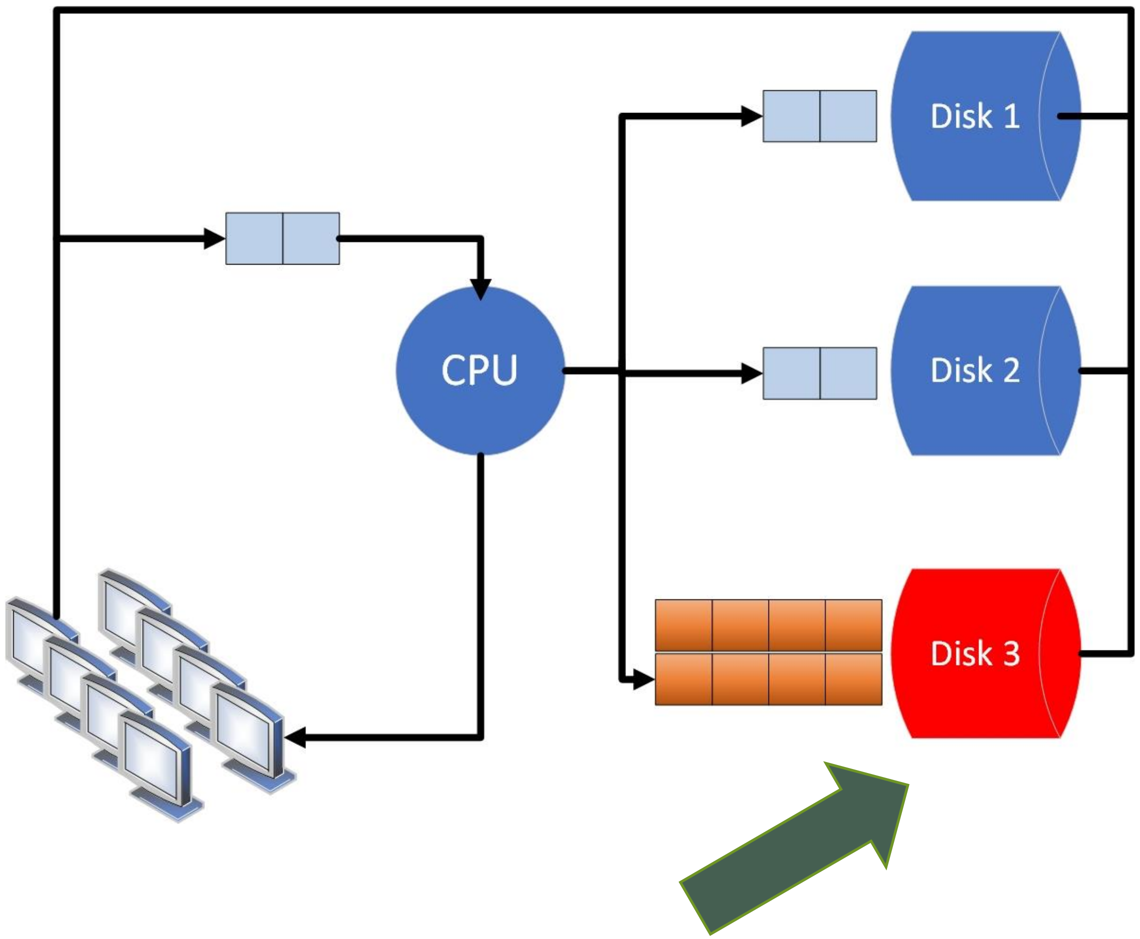
Open and Closed network models

- Closed models impose a limit ***n***, on the number of concurrent customers
  - When a bottlenecked resource in a closed model saturates, the maximum Q_{len} that can be observed is limited to ***n-1***
- In contrast, Open models draw customers from an infinite source, λ remains constant, so the maximum Q_{len} is ∞



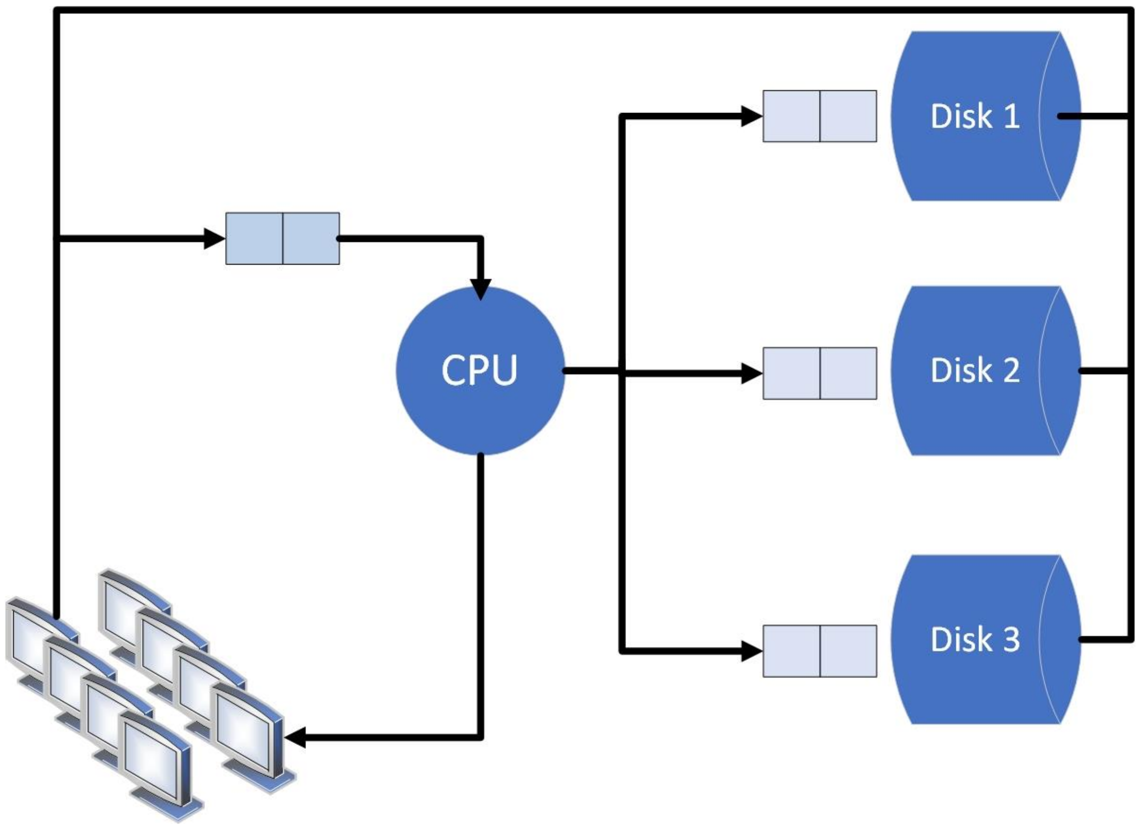
Closed network queueing model

- When there is a bottlenecked resource, the model shows the Q_{len} elongates and customers are "stuck" in the system
- This dampens the arrival rate for new service Requests, since the number of customers is fixed
- Corollary: RT is optimal when resources are lightly loaded and queueing delays are minimal



Closed network queueing model

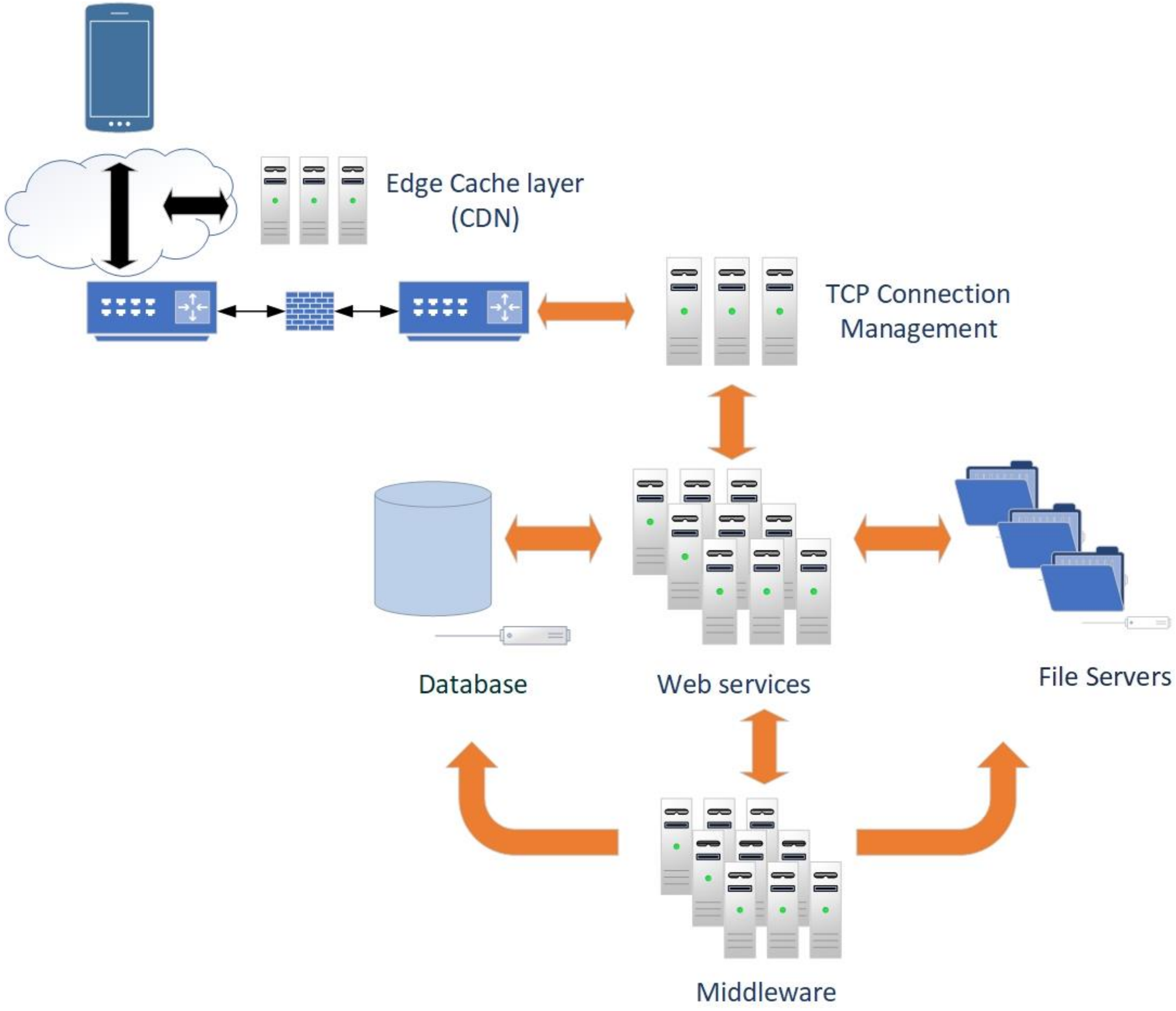
- A **balanced** system where all resource Queue Times are approximately equal is the optimal configuration
- No bottleneck!
- Corollary: **load balancing** is an optimal solution to most queueing circuits



Modern connected applications

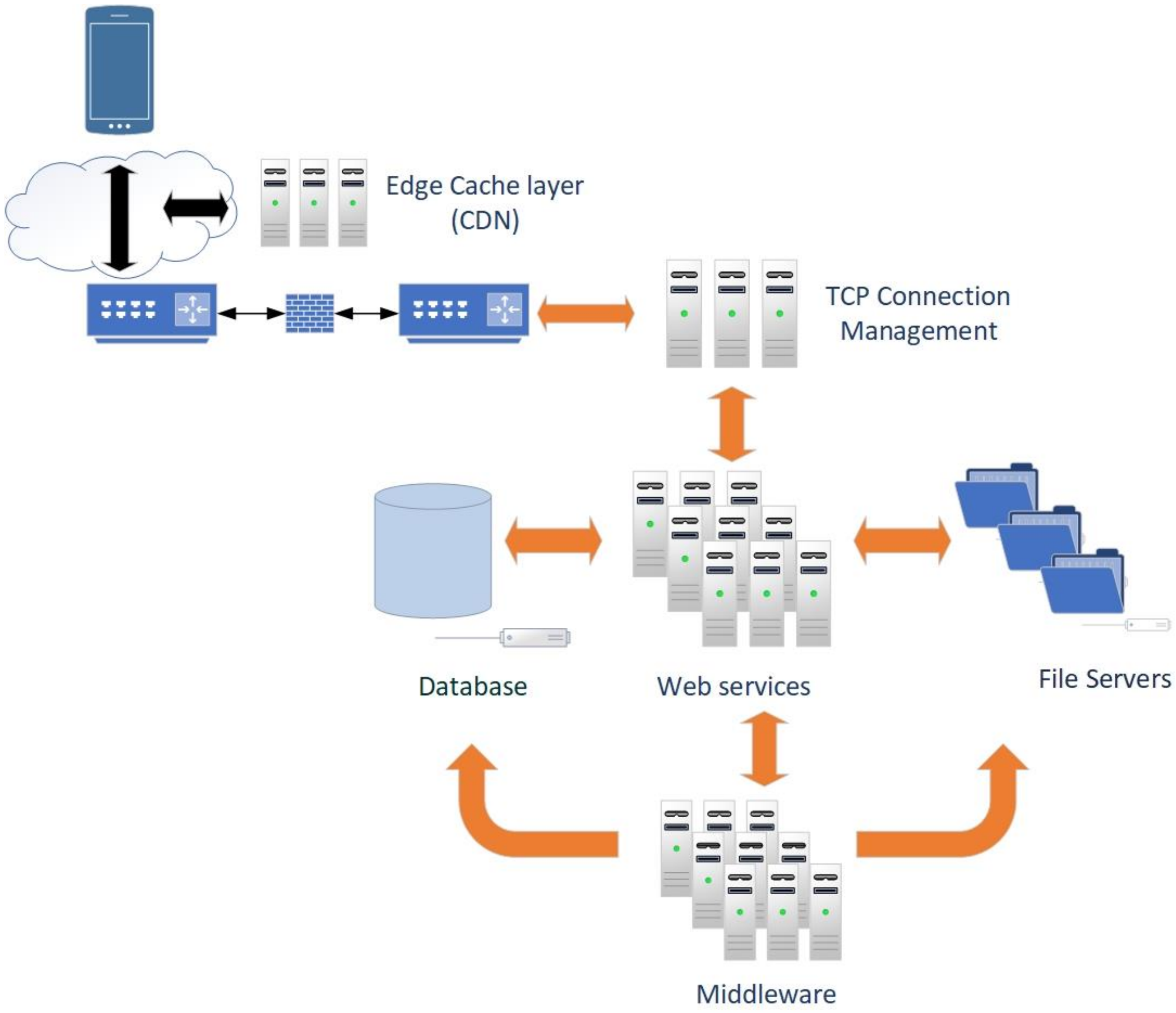
- Multiple tiers
  - Cloud-based
  - TCP Connection management
  - Web servers/services
  - Middleware
  - Database back-end(s)
  - File servers
  - Storage Area Networks
  - Virtualization

  - Edge networks
    - e.g., Content Delivery Network (CDN)



Modern connected applications

- Complications
  - Individual tiers/components have **incomplete** and/or **inconsistent** instrumentation
  - Synchronous vs. asynchronous calls (apparent Response times vs. actual Response times)
  - Are measurements taken across Callers & Providers correlated?
    - i.e., web service ⇨ DBMS
  - Caches ⇨ bi-modal service time distributions
    - report Hit ratios
    - break out service times for Hits/Misses separately

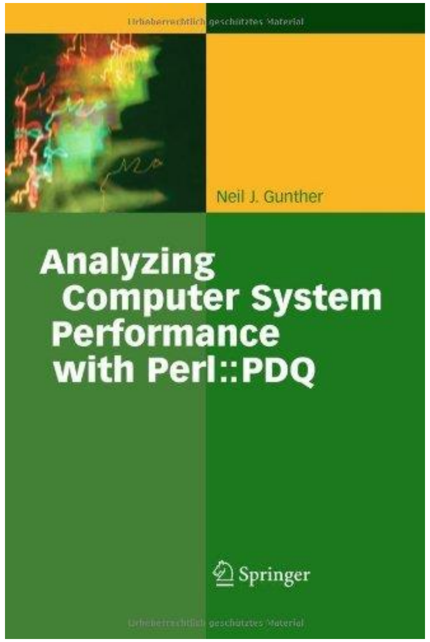


Perl & PDQ sample

```
use pdq;

# Globals
$arrivRate = 0.75;
$servTime  = 1.0;

pdq::Init("Open Network with M/M/1");
pdq::CreateOpen("Work", $arrivRate);
pdq::CreateNode("Server", $pdq::CEN, $pdq::FCFS);
pdq::SetDemand("Server", "Work", $servTime);

# Solve the model
pdq::Solve($pdq::CANON);

pdq::Report();
```

Perl & PDQ sample

- Extend the simple sample:
  - add a loop in Perl that increases the arrival rate variable until the “Server” resource saturates
  - add additional secondary resources: disk, DBMS, network, etc.
  - add additional servers
    - model the network latency between servers as a delay server (no queueing)



Open and Closed models

- Closed models assume a limit ***n***, on the number of concurrent customers
  - requires the equilibrium assumption
  - When a bottlenecked resource in a closed model saturates, the maximum Q_{len} that can be observed is limited to ***n-1***
- Open models draw customers from an infinite source, so the maximum Q_{len} is ∞
  - the potential number of customers for some connected web-based applications is so large that Open models can apply
    - when arrival rates remain steady, even where there is contention!
- Heavy-traffic approximations: $u \Rightarrow \infty$

Limitations of closed network models

- Separability*
  - must be able to be solve models for individual nodes separately, which are then combined (Product-Form solution)
  - Service policies:
    - FIFO or FCFS
    - Round robin
    - Delay (no queueing behavior)
    - Priority queuing with preemptive scheduling (approximations)
  - Exponential service times
  - Flow balance ($\lambda = C$)
  - * BCMP (Baskett, Chandy, Muntz & Palacios, 1975)

Limitations of closed network models

- see Gunther, ch. 3.
  - Bulk arrivals (in general, anytime $\lambda \neq C$)
  - non-exponential service times
  - Blocking, Mutual exclusion (locking)
  - Mutual exclusion
  - Queuing defections
  - Fork/Join

- There are clever ways around most of these limitations
  - Load-dependent servers
  - Priority queueing (with preemptive scheduling)

Perl & PDQ sample

```
use pdq;

$model    = "Middleware";
$work     = "eBiz-tx";

$node1    = "WebServer";
$node2    = "AppServer";
$node3    = "DBMServer";
$node4    = "DummySvr";

$think    = 0.0 * 1e-3;   # treat as free param
$users    = 10;


pdq::Init($model);
pdq::CreateNode($node1, $pdq::CEN, $pdq::FCFS);pdq::CreateNode($node2, $pdq::CEN, $pdq::FCFS);
pdq::CreateNode($node3, $pdq::CEN, $pdq::FCFS);
pdq::CreateNode($node4, $pdq::CEN, $pdq::FCFS);

pdq::CreateClosed($work, $pdq::TERM, $users, $think);

#  NOTE: timebase is seconds
pdq::SetDemand($node1, $work, 9.8 * 1e-3);
pdq::SetDemand($node2, $work, 2.5 * 1e-3);
pdq::SetDemand($node3, $work, 0.72 * 1e-3);
pdq::SetDemand($node4, $work, 9.8 * 1e-3);

pdq::Solve($pdq::EXACT);
pdq::Report();
```

Analytic queuing models: an Assessment

- Because they mimic the actual behavior of computer applications, queuing models inform much of computer performance analysis
  - relationship between response time & unitization is **nonlinear**

$$QT > ST, \text{ if } u > .50 \qquad (m/m/1)$$

- Scheduling algorithms that reduce **variability** in the service time distribution help
  - multiple service classes (minimally: foreground : background)
  - time-slicing; avoiding starvation
  - packet-switching in networks
    - should routers queue requests when a server along the route is busy?

Analytic queuing models: an Assessment

- Because they mimic the actual behavior of computer applications, queuing models inform much of computer performance analysis

- Model building & validation
  - Train them on available measurement data – can the model accurately predict observed performance?
    - validation step often reveals the need for missing data or uncovers hidden sources of resource contention
  - Exact solution vs. more tractable approximation methods
  - What if? predictive scenarios
    - impact of new equipment that runs faster
    - impact of adding load to model customer growth

Analytic queueing models: an Assessment

- Practical "guerilla" approach to using analytic models
  - emphasize results; de-emphasize time-consuming model validation
  - e.g., Model the application before you build it
  - PDQ library is programmable

- Bottleneck analysis
  - Required for intelligent alerting, automatic provisioning

- Alternatives to analytic models
  - discrete-event simulation (see SPE*ED : UML ⇨ model)
  - trace-driven simulation

Additional References

- Ed Lazowska, *et. al.*, *Quantitative System Performance*, 1984.
- Neil Gunther, *The Practical Performance Analyst*, 1998.
- Daniel Menascé, *et. al.*, *Performance By Design*, 2004.