System Performance and Highly Concurrent Systems

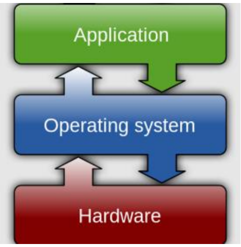
Lecture 12

Hartmut Kaiser

https://teaching.hkaiser.org/spring2025/csc4103/

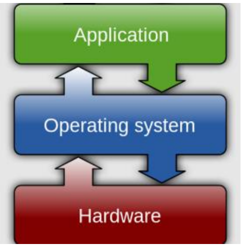
Recall: Deadlock

- Starvation vs. Deadlock
  - Starvation: Thread indefinitely unable to make progress
  - Deadlock: Thread(s) unable to make progress due to circular wait
- Four conditions for deadlock:
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular Wait
- Three different approaches to address deadlock:
  - Deadlock avoidance: dynamically delay resource requests so deadlock doesn't happen
  - Deadlock prevention: write your code in a way that it isn't prone to deadlock
  - Deadlock recovery: let deadlock happen, and then figure out how to recover from it
- Or deadlock denial: ignore the possibility of deadlock in applications



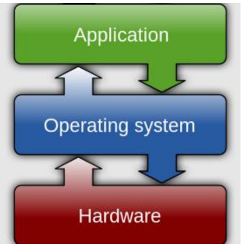
System Performance

- "Back of the Envelope" calculation and modeling
- Get the rough picture first… and don't lose sight of it

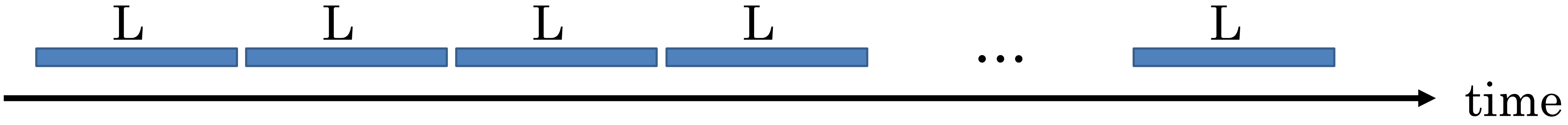


Times (s) and Rates (op/s)

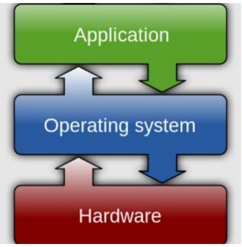
- Latency – time to complete a task
  - Measured in units of time (s, ms, us, …, hours, years)
- Response Time - time to initiate an operation and get its response
  - Able to issue an operation that depends on the result of another
  - Know that it is done (anti-dependence, resource usage)
- Throughput or Bandwidth – rate at which tasks are performed
  - Measured in units of things per unit of time (op/s, FLOP/s)
- Performance
  - Operation time (5 mins to run a mile…)
  - Rate (mph, mpg, …)



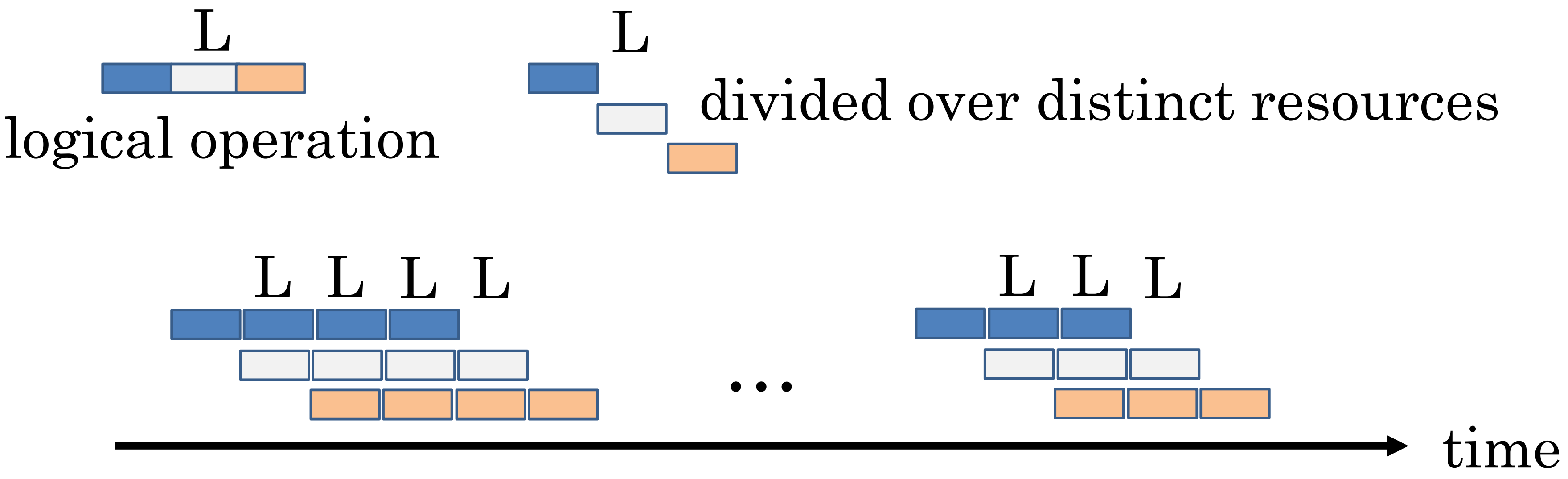
Sequential Server Performance



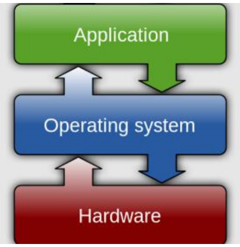
- Single sequential “server” that can deliver a task in time L operates at rate $\leq \frac{1}{L}$ (on average, in steady state, …)
  - $L = 10\text{ ms} \rightarrow B = 100\ ^{\text{op}}/_{\text{s}}$
  - $L = 2\text{ yr} \rightarrow B = 0.5\ ^{\text{op}}/_{\text{yr}}$
- Applies to a processor, a disk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



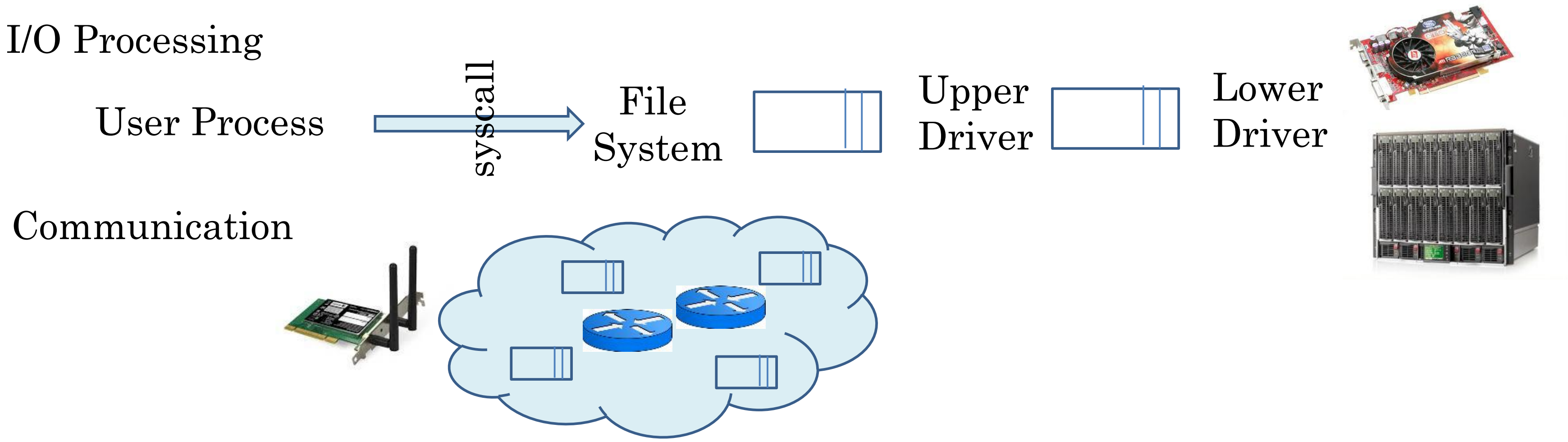
Single Pipelined Server



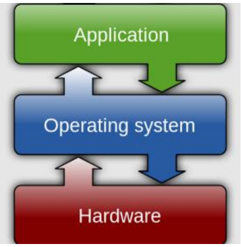
- Single pipelined server of k stages for tasks of length L (i.e., time ${}^{L}/_{k}$ per stage) delivers at rate $\leq {}^{k}/_{L}$.
  - $L = 10$ ms, $k = 4 \rightarrow B = 400\ {}^{\text{op}}/_{\text{s}}$
  - $L = 2$ yr, $k = 2 \rightarrow B = 1\ {}^{\text{op}}/_{\text{yr}}$



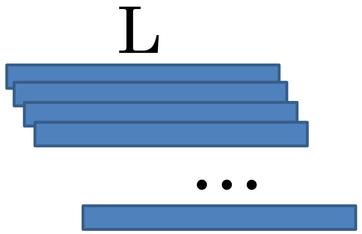
Example Systems "Pipelines"

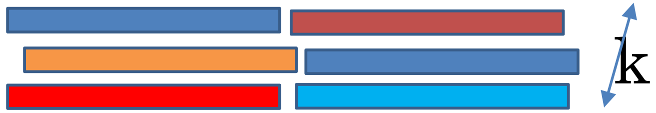


- Anything with queues between operational processes behaves roughly "pipeline like"
- Important difference is that "initiations" are decoupled from processing
  - May have to queue up a burst of operations
  - Not synchronous and deterministic

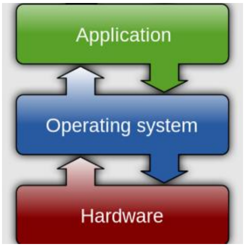


Multiple Servers



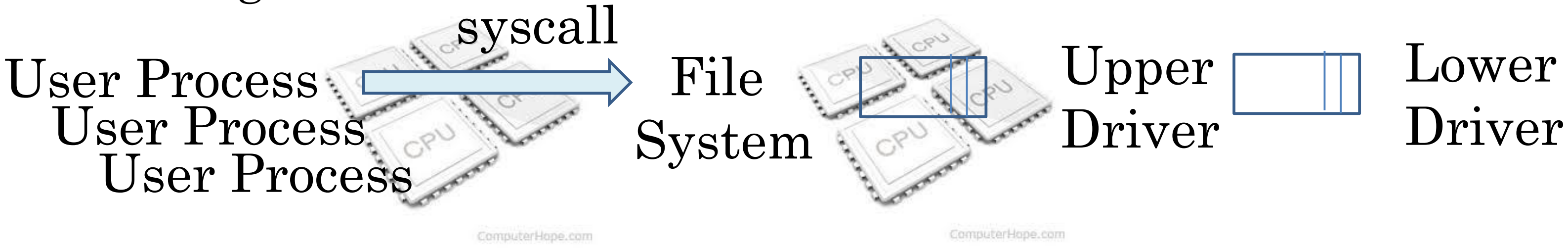


- k servers handling tasks of length L delivers at rate $\leq {}^{k}/_{L}$.
  - $L = 10\text{ ms}, k = 4 \rightarrow B = 400\ {}^{\text{op}}/_{\text{s}}$
  - $L = 2\text{ yr}, k = 2 \rightarrow B = 1\ {}^{\text{op}}/_{\text{yr}}$
- You have seen multiple processors (cores)
  - Systems present lots of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

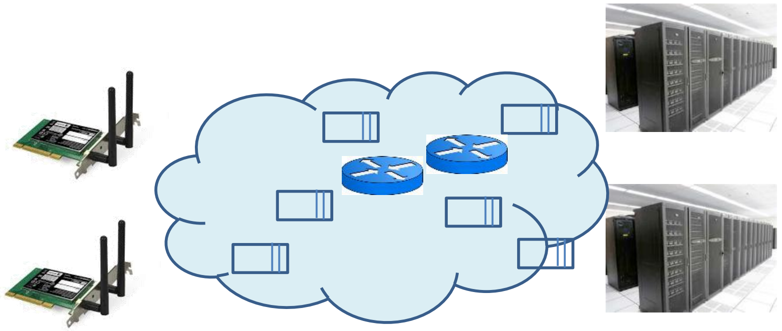


Example System "Parallelism"

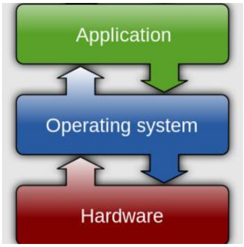
I/O Processing



Communication



Parallel Computation, Databases, …

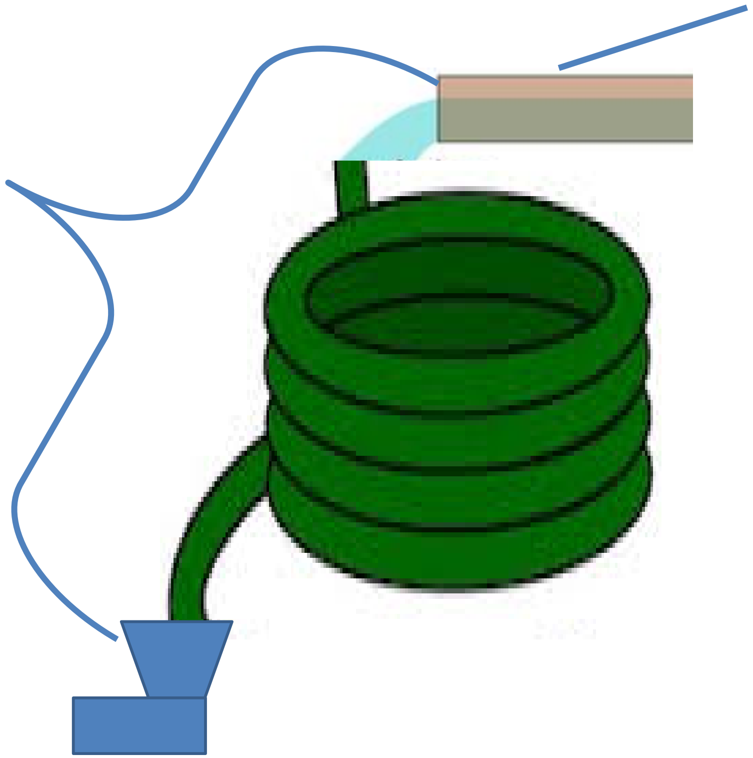


A Simple System Performance Model

Latency (L): time per op

- How long does it take to flow through the system

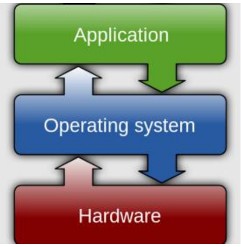
"Service Time"



Bandwidth (B): Rate, Op/s

e.g., flow: gal per min

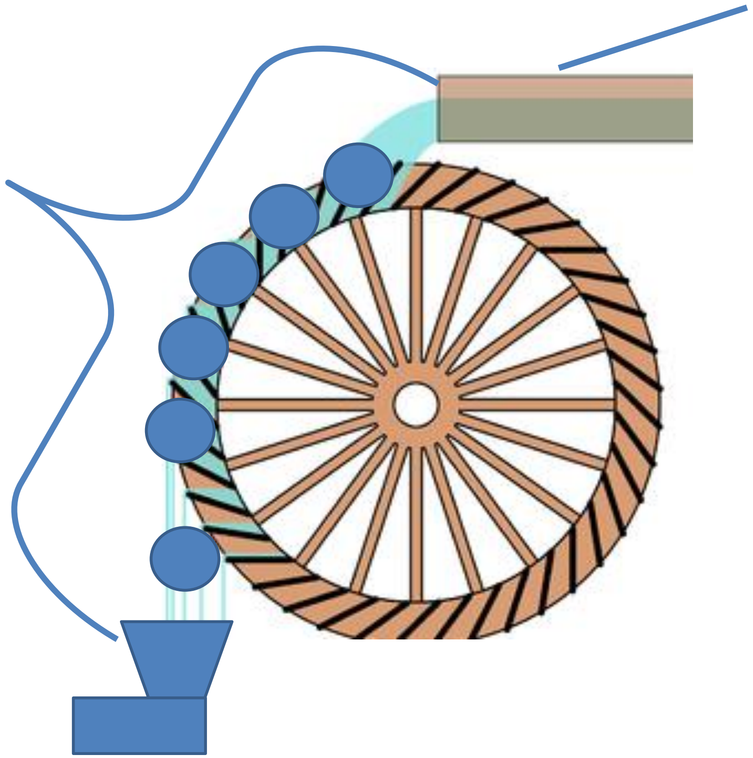
If $B = 2\ {}^{\text{gal}}/_{\text{s}}$ and L = 3 s
How much water is "in the system?"



A Simple System Performance Model

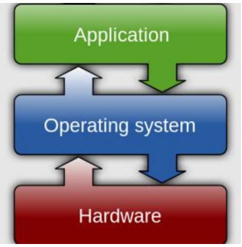
Latency (L): time per op
- How long does it take to flow through the system

"Service Time"



Bandwidth (B): Rate, Op/s
e.g., flow: gal per min

If $B = 2\ {}^{gal}/_{s}$ and L = 3 s
How much water is "in the system?"



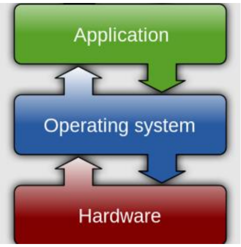
Little's Law

- The number of "things" in a system is equal to the bandwidth times the latency (on average)

$$n = L\ B$$

- Applies to any stable system (arrival rate = departure rate)

- Can be applied to an entire system:
  - Including the queues, the processing stages, parallelism, whatever
- Or to just one processing stage:
  - i.e., disk I/O subsystem, queue leading into a CPU or I/O stage, …

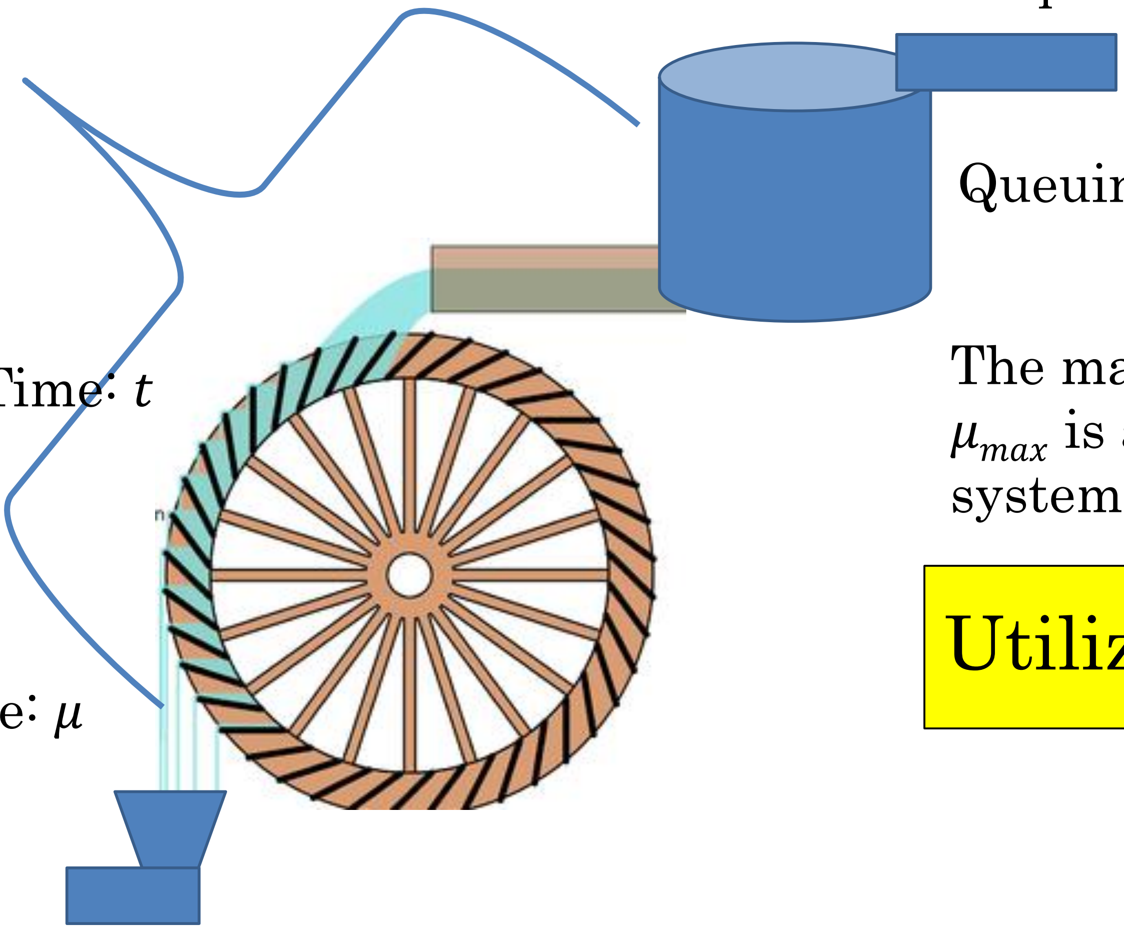


A Simple System Performance Model

Request Rate: λ

Latency (L)

Queuing delay: d

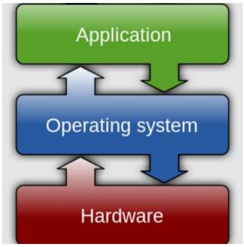


Operation Time: t

Service Rate: μ

The maximum service rate μ_{max} is a property of the system – the “bottleneck”

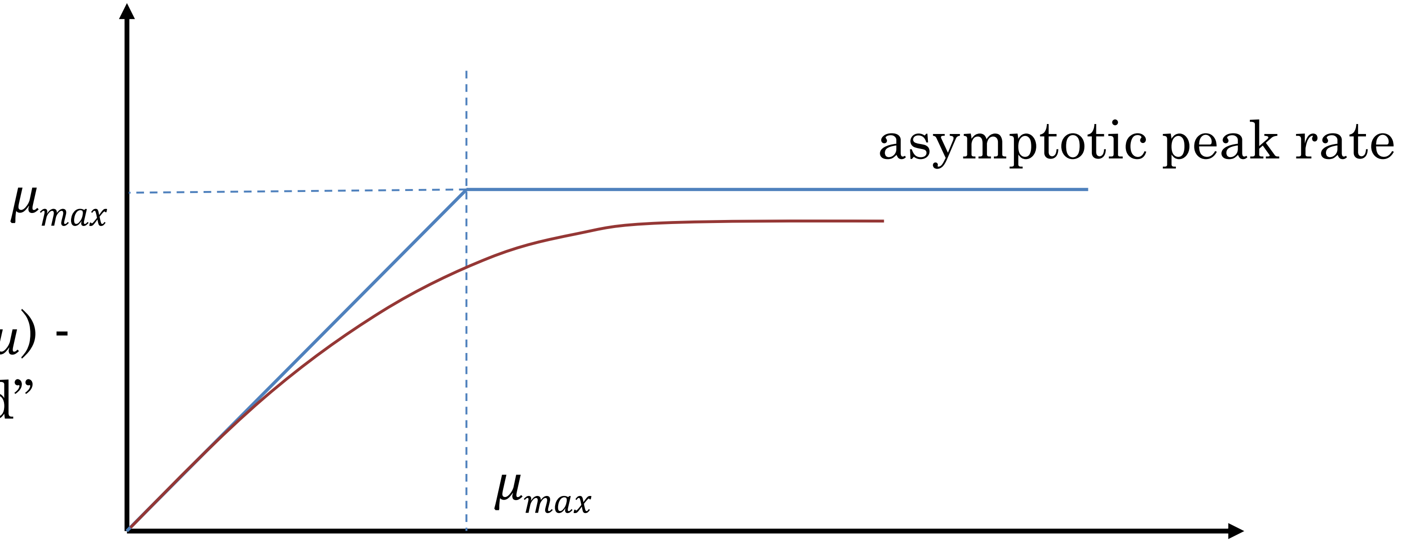
$$\text{Utilization: } \rho = \frac{\lambda}{\mu_{max}}$$



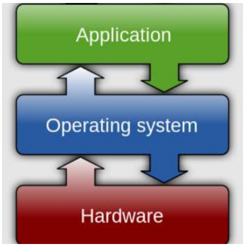
Ideal System Performance

- How does μ (service rate) vary with λ (request rate)?

Service Rate (μ) - "delivered load"



Request Rate (λ) - "offered load"

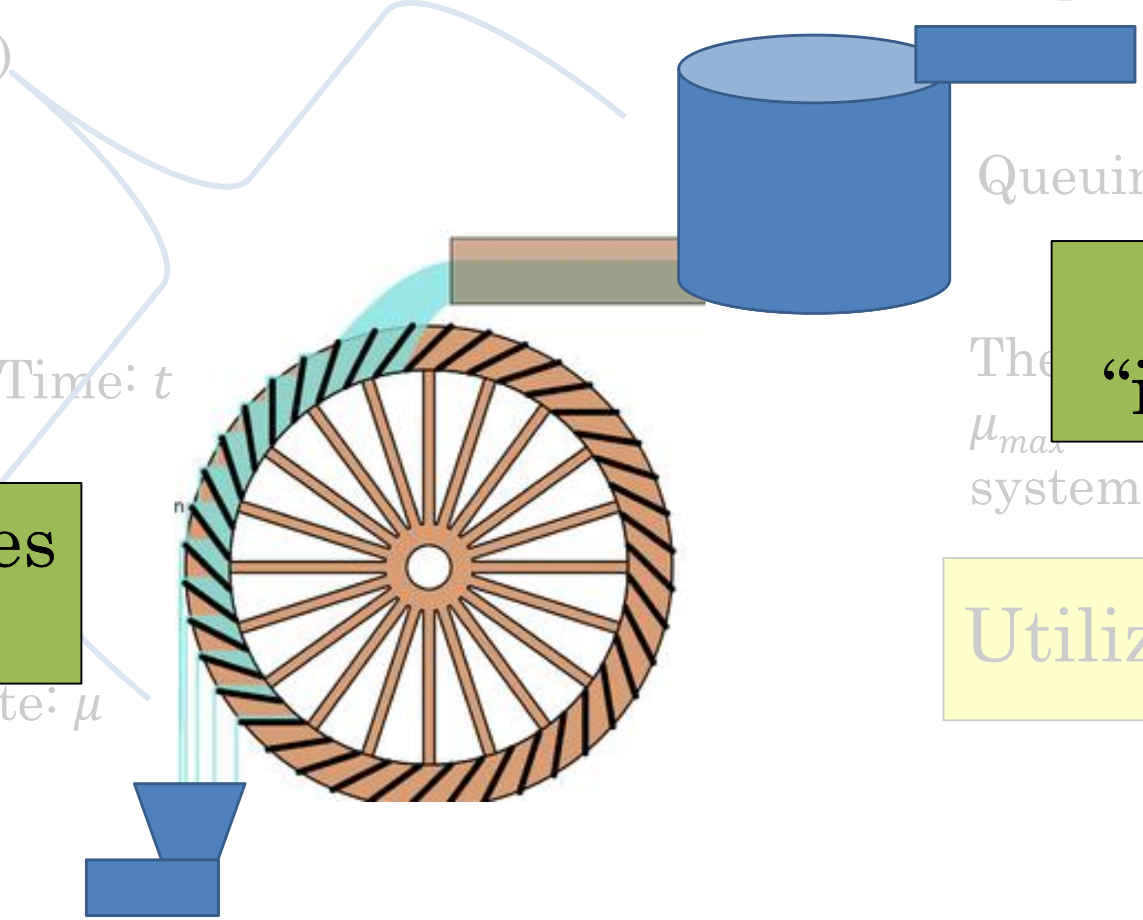


A Simple System Performance Model

Request Rate: λ

Latency (L)

Queuing delay: d



Operation Time: t

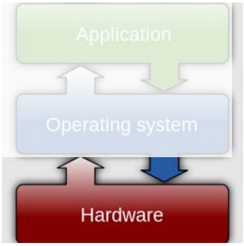
What about "internal" queues?

The ... μ_{max} ... system – the "bottleneck"

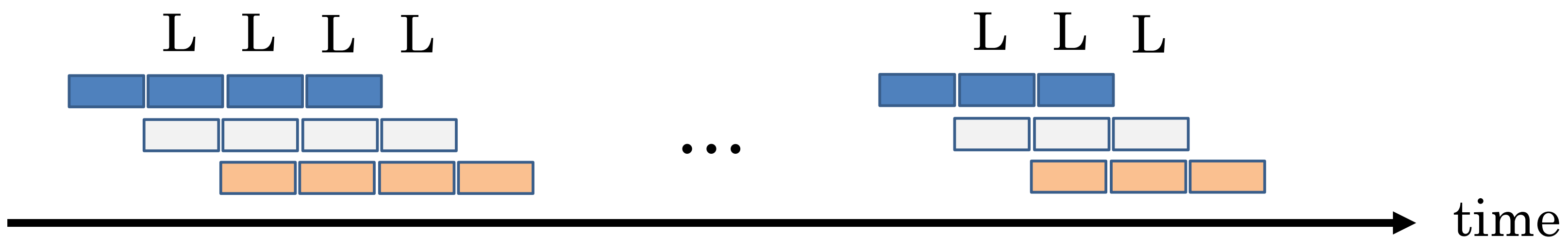
What determines μ_{max}?

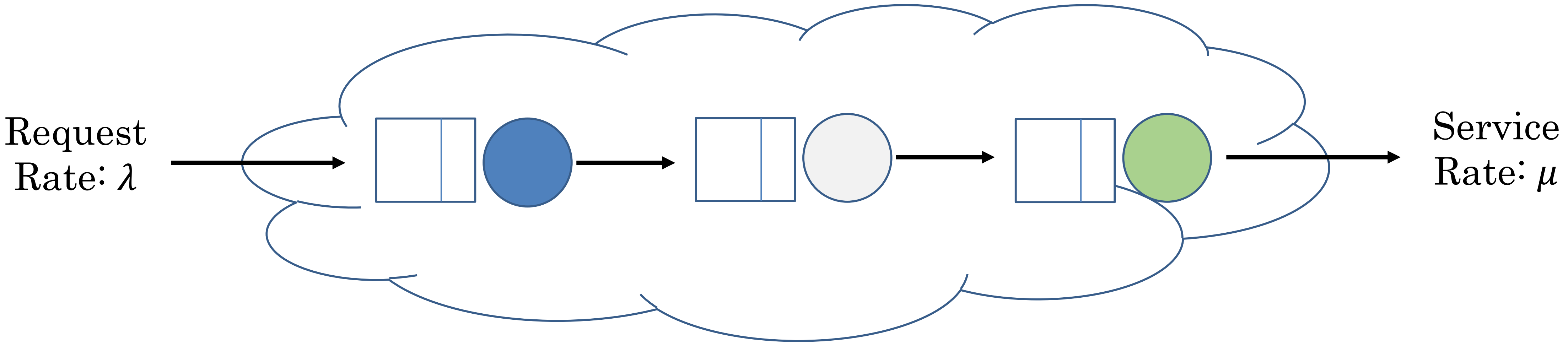
Service Rate: μ

Utilization: $\rho = \dfrac{\lambda}{\mu_{max}}$

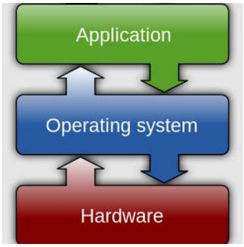


Bottleneck Analysis



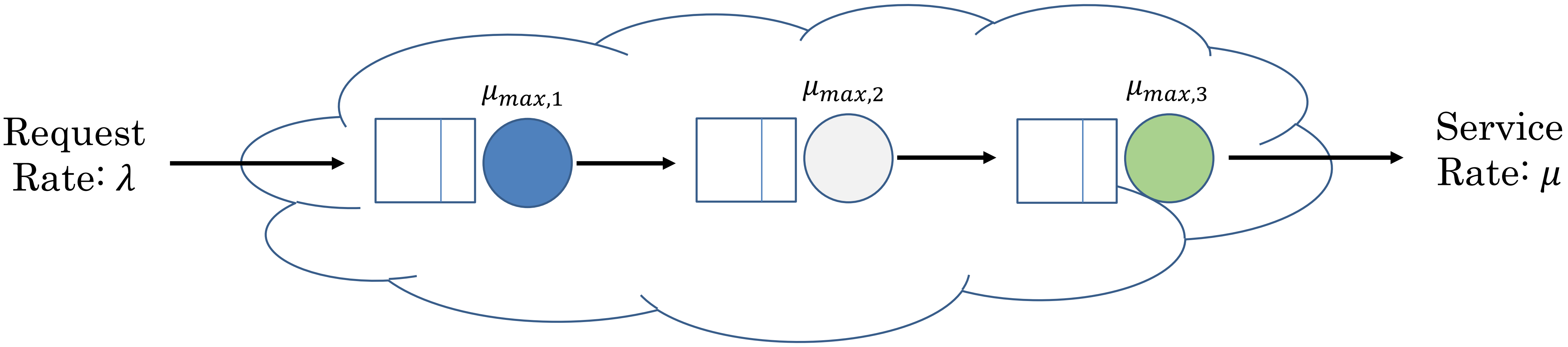


Overall System: Series of Stages

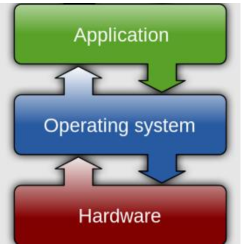


Bottleneck Analysis

- Each stage has its own queue and maximum service rate
- Suppose the green stage is the bottleneck

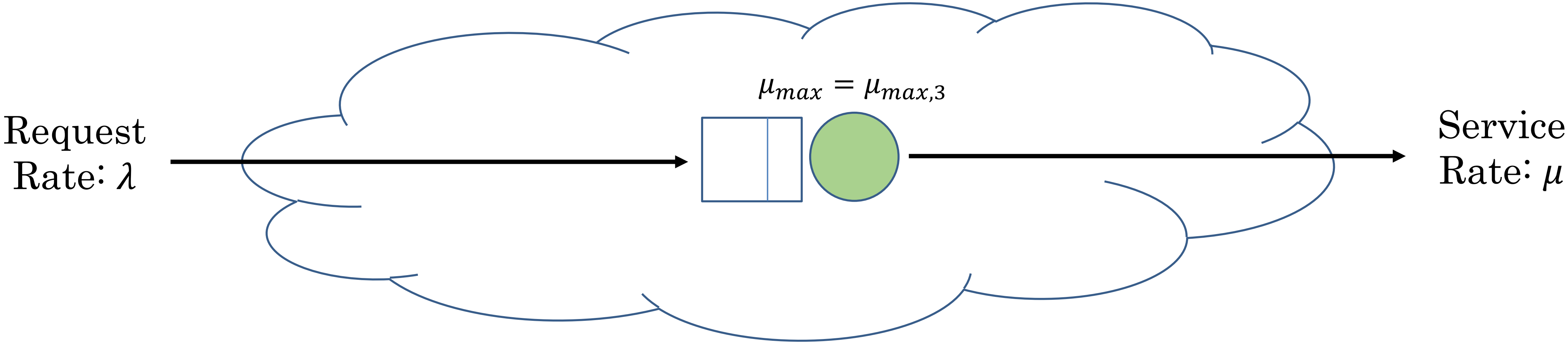


Overall System: Series of Stages

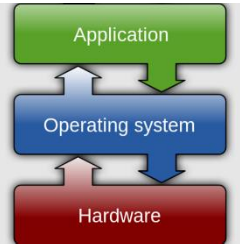


Bottleneck Analysis

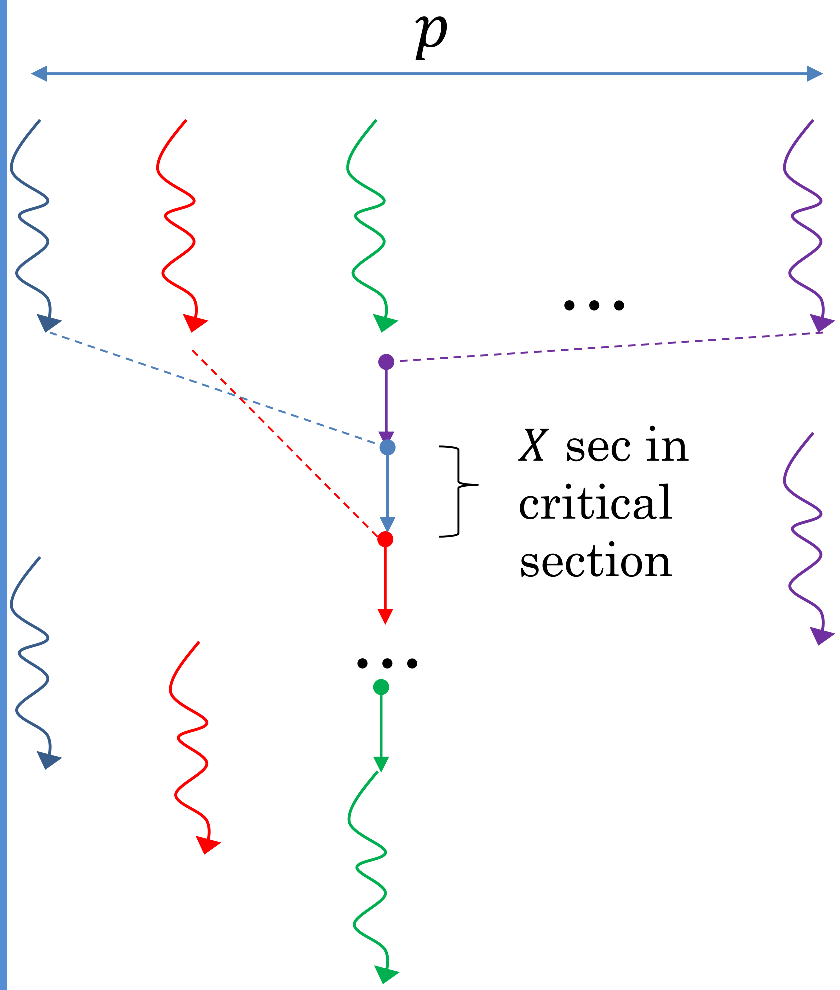
- Each stage has its own queue and maximum service rate
- Suppose the green stage is the bottleneck
- The bottleneck stage dictates the maximum service rate μ_{max}



System Model: Bottleneck Stage



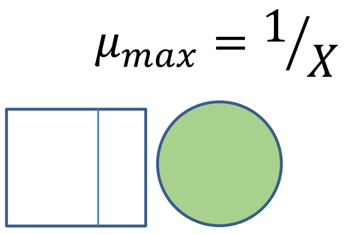
Example: Servicing a Highly Contended Lock



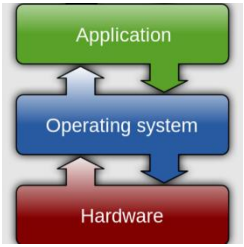
All try to grab lock

Time = $p \cdot X$ sec
Rate = $^1/_X$ ops/sec, regardless of # cores

Queue of waiting threads



Critical section guarded by lock

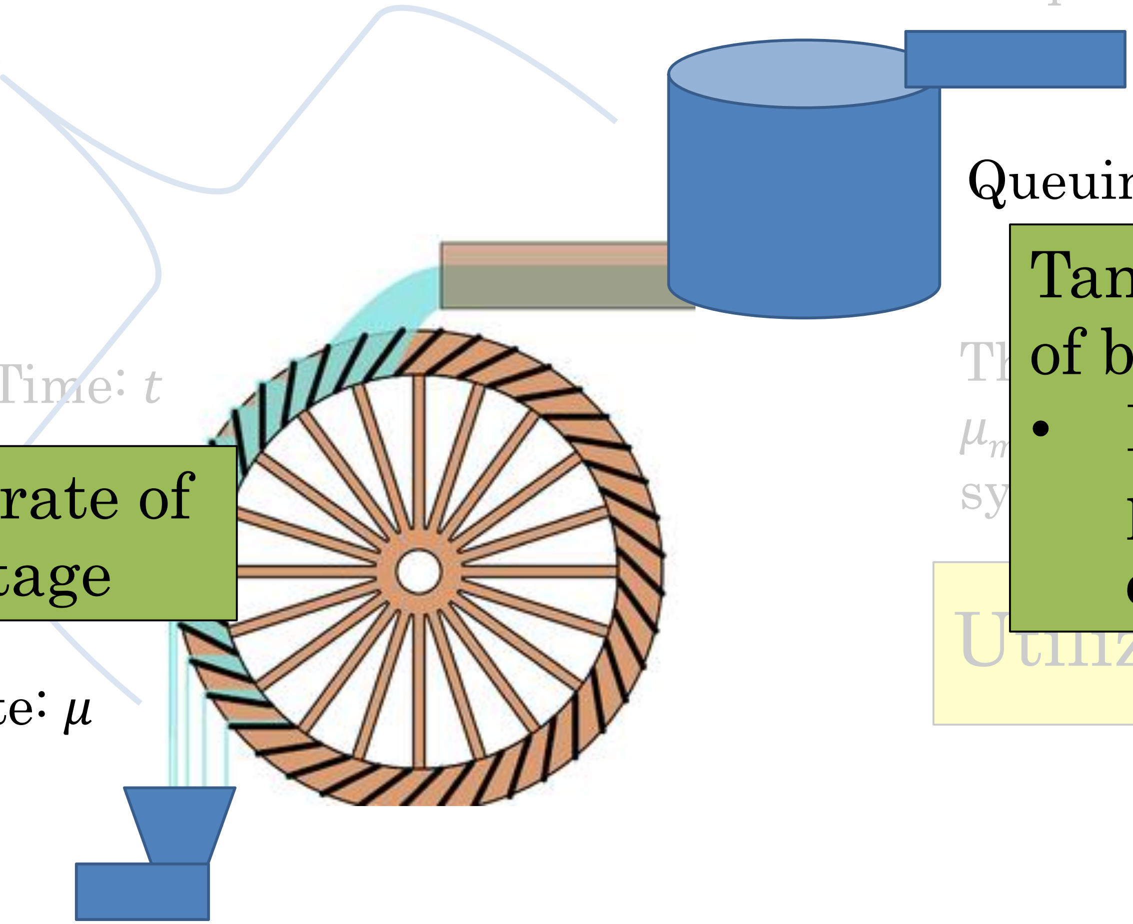


A Simple System Performance Model

Request Rate: λ

Latency (L)

Queuing delay: d



Tank represents queue of bottleneck stage
- Including queues of previous stages, in case of backpressure

Operation Time: t

μ_{max} is service rate of bottleneck stage

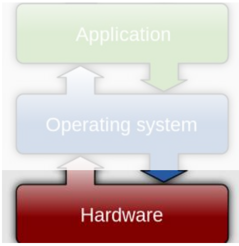
Service Rate: μ

Th

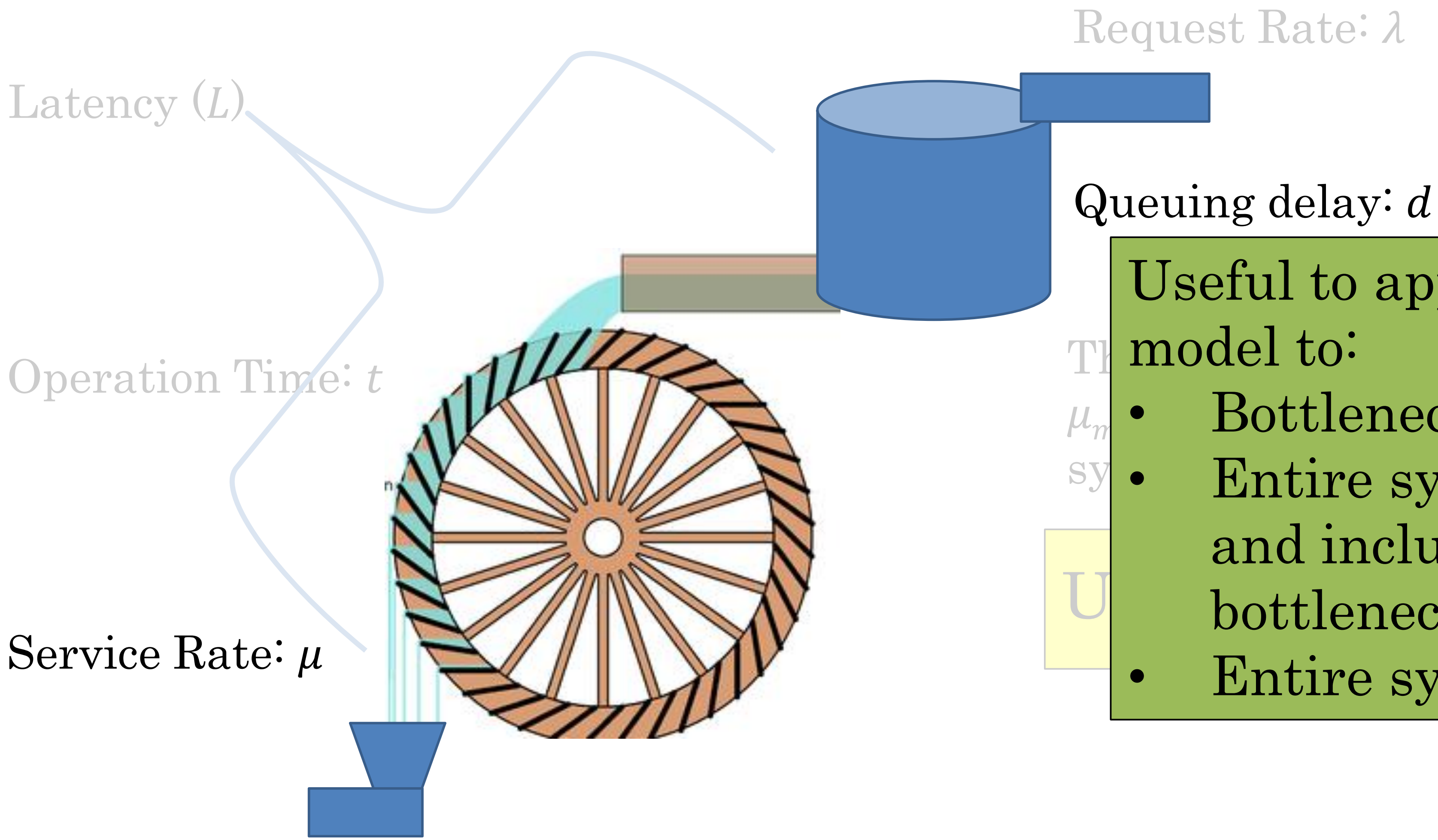
μ_m

sy

Utilization: $\rho = \frac{}{\mu_{max}}$

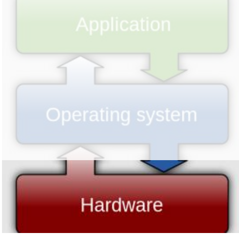


A Simple System Performance Model



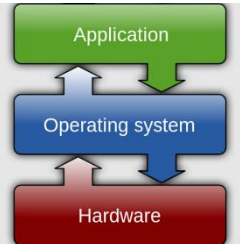
Useful to apply this model to:

- Bottleneck stage
- Entire system up to and including bottleneck stage
- Entire system



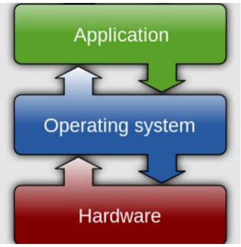
Announcements

- Assignment 2 due today
  - Let me know if you have trouble pushing to your repository
  - If auto-grader fails even if locally all is well – attach screenshot to submission
- Project 1 due today as well
  - Let me know if you need more time

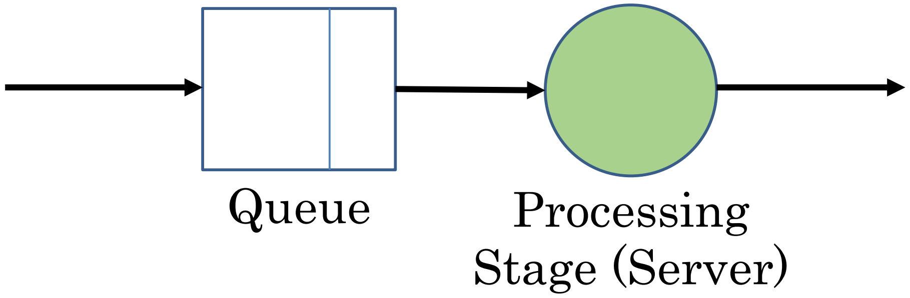


Rest of Today's Lecture

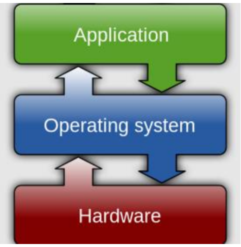
- Using this system model, we will:
  - Explore latency in more depth
  - Discuss how to build systems that perform well under load



Latency (Response Time)



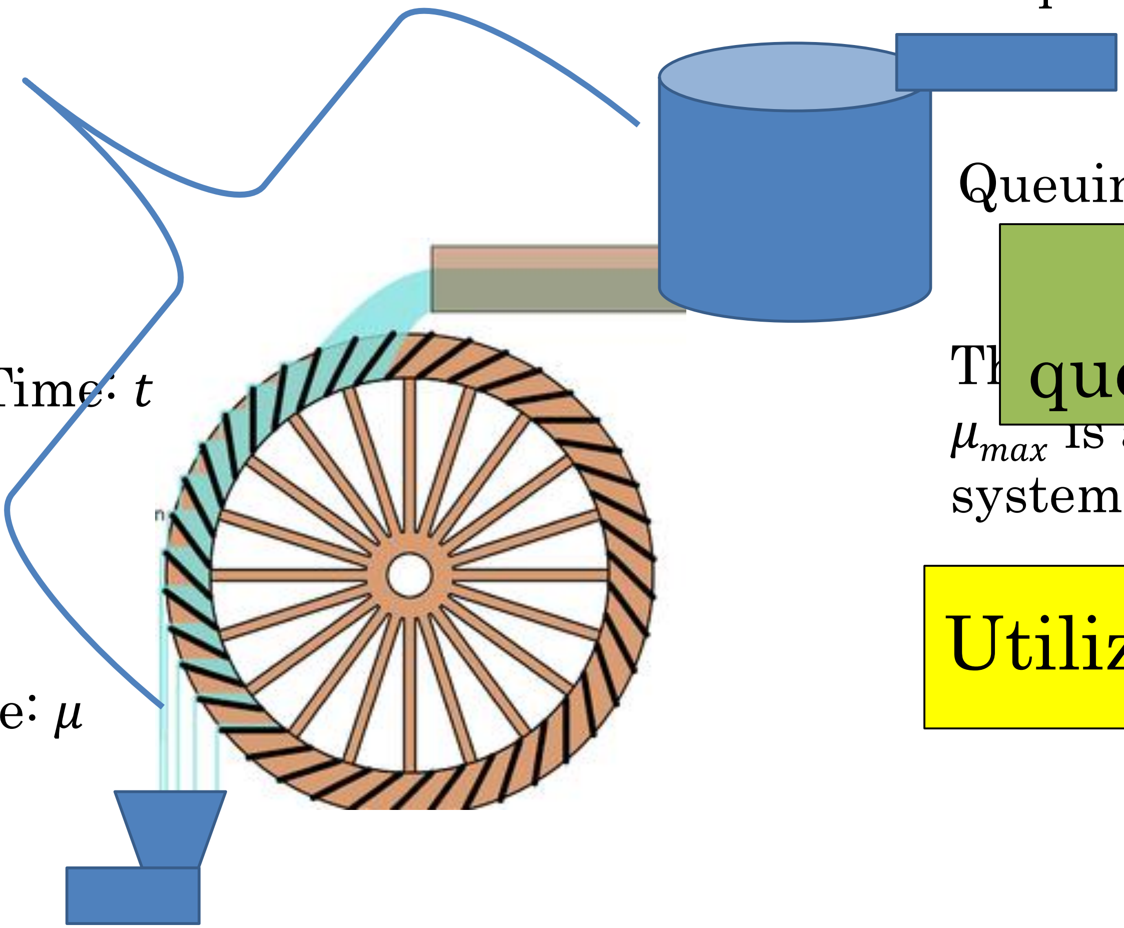
- Total latency (response time): queuing time + service time
- Service time depends on the underlying operation
  - For CPU stage, how much computation
  - For I/O stage, characteristics of the hardware
- What about the queuing time?



A Simple System Performance Model

Request Rate: λ

Latency (L)



Queuing delay: d

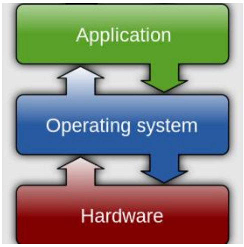
Operation Time: t

Service Rate: μ

When will the queue(s) start to fill?

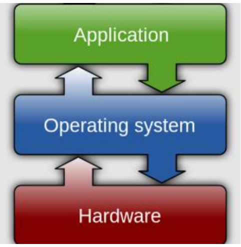
Th… μ_{max} is a property of the system – the “bottleneck”

Utilization: $\rho = \dfrac{\lambda}{\mu_{max}}$



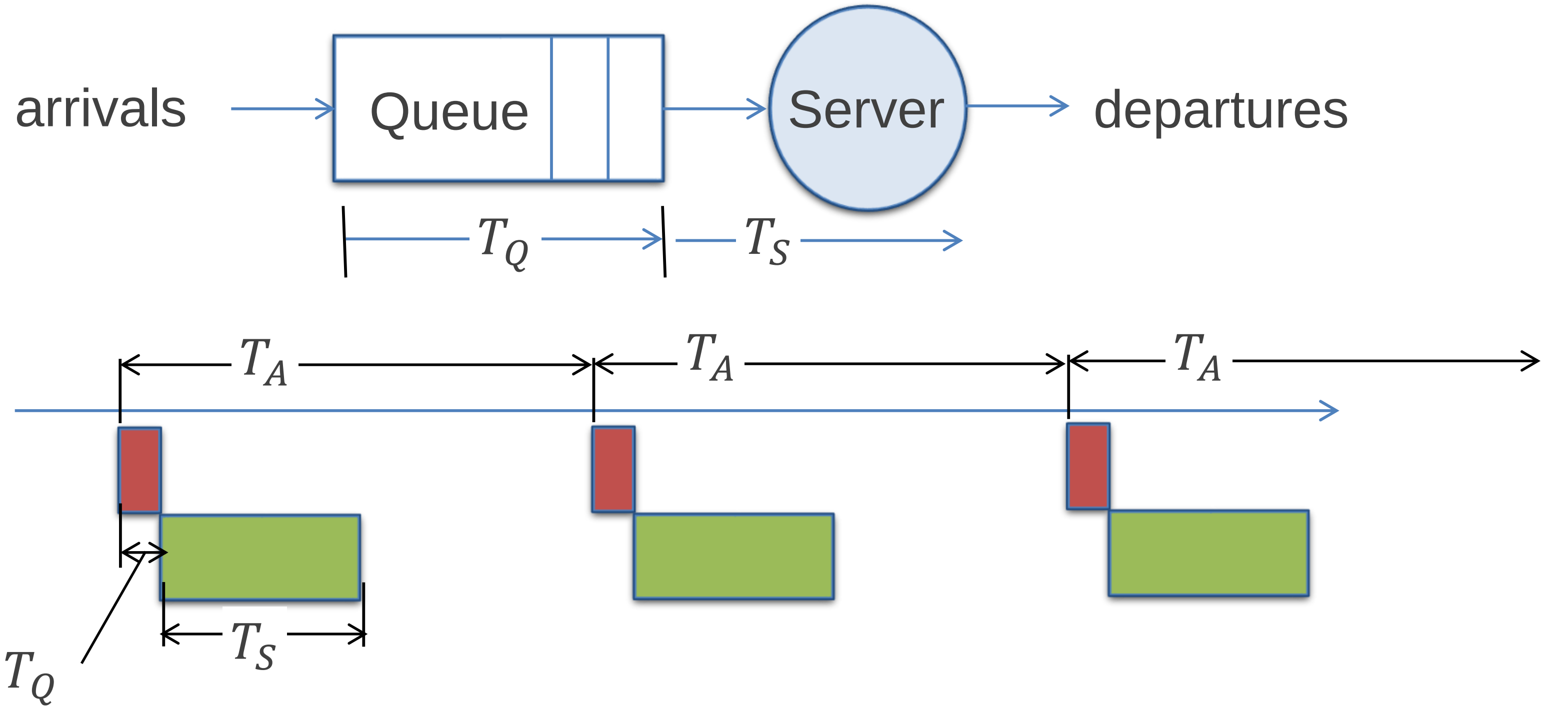
Queuing

- What happens when request rate (λ) exceeds max service rate (μ_{max})?
- Short bursts can be absorbed by the queue
  - If on average $\lambda < \mu$, it will drain eventually
- Prolonged $\lambda > \mu \rightarrow$ queue will grow without bound



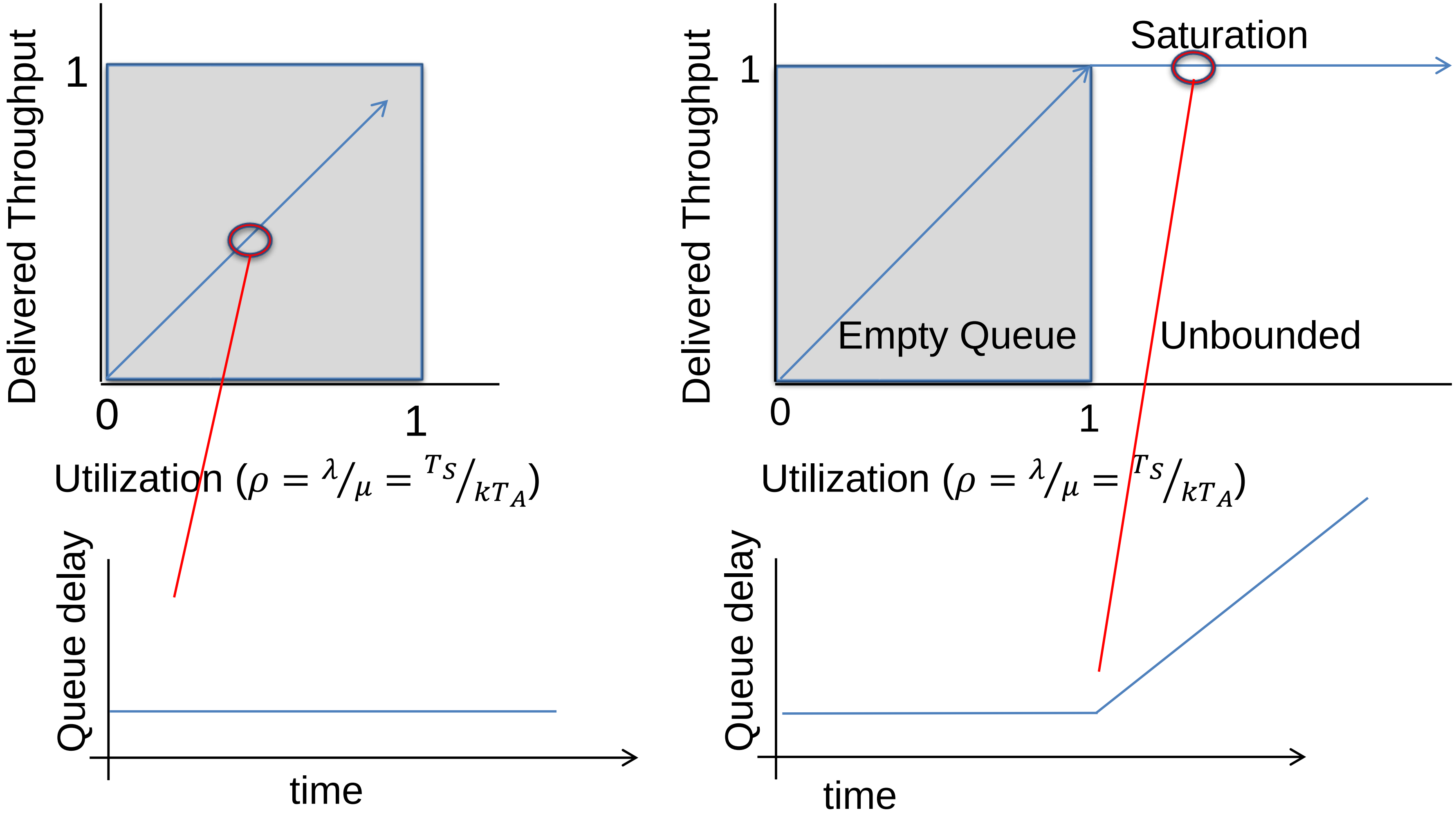
A Simple, Deterministic World

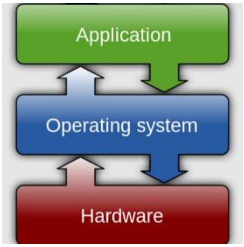
- T_A: time between arrivals
  - $\lambda = {}^{1}/_{T_A}$
- T_S: service time
  - $\mu = {}^{k}/_{T_S}$
- T_Q: queuing time
  - $L = T_Q + T_S$



- Assume requests arrive at regular intervals, take a fixed time to process, with plenty of time between …

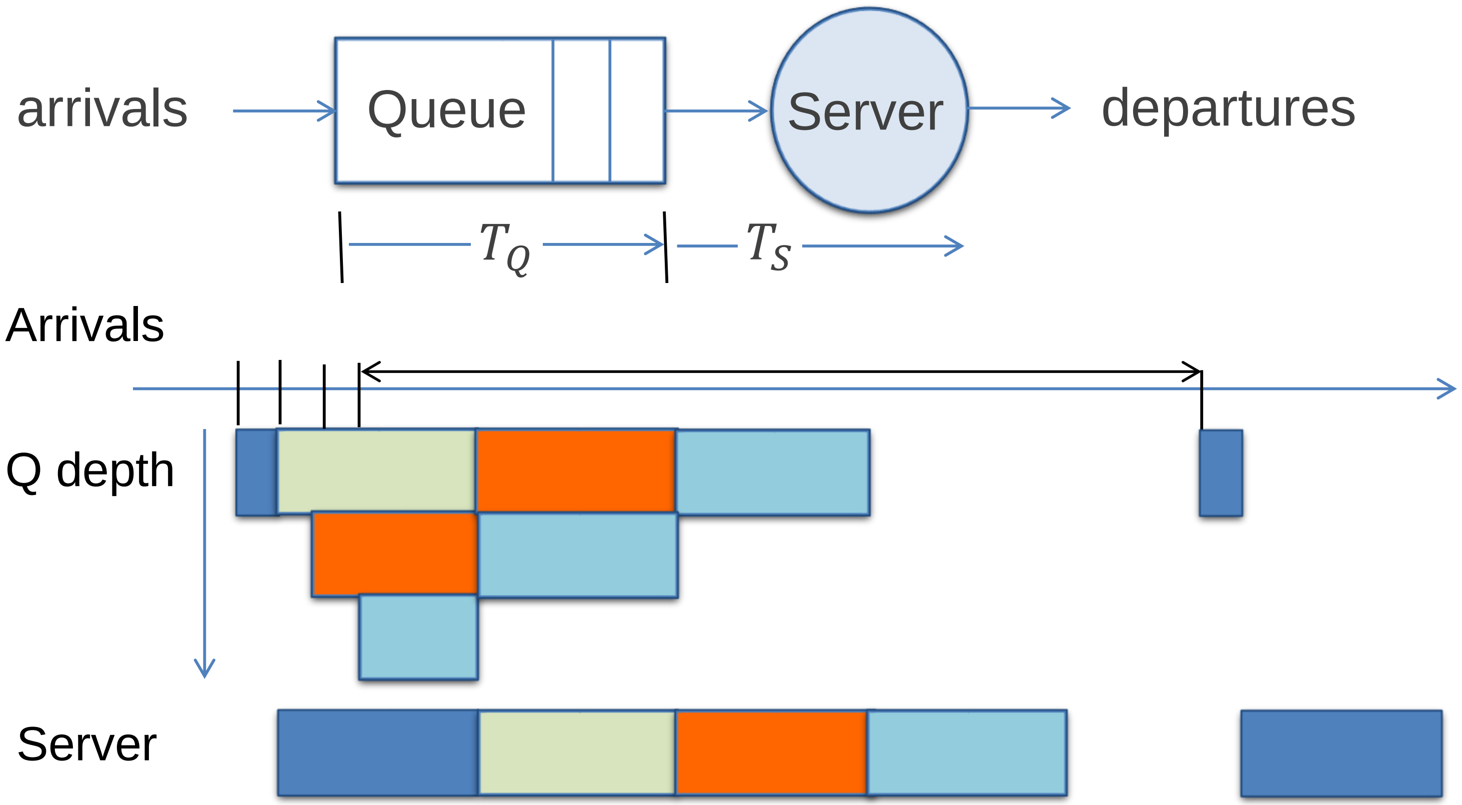
A Simple, Deterministic World



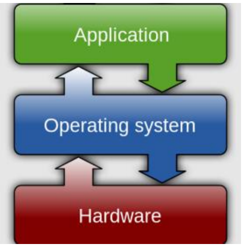


A Bursty World

- T_A: time between arrivals
  - Now, a random variable
- T_S: service time
  - $\mu = {}^{k}/_{T_S}$
- T_Q: queuing time
  - $L = T_Q + T_S$



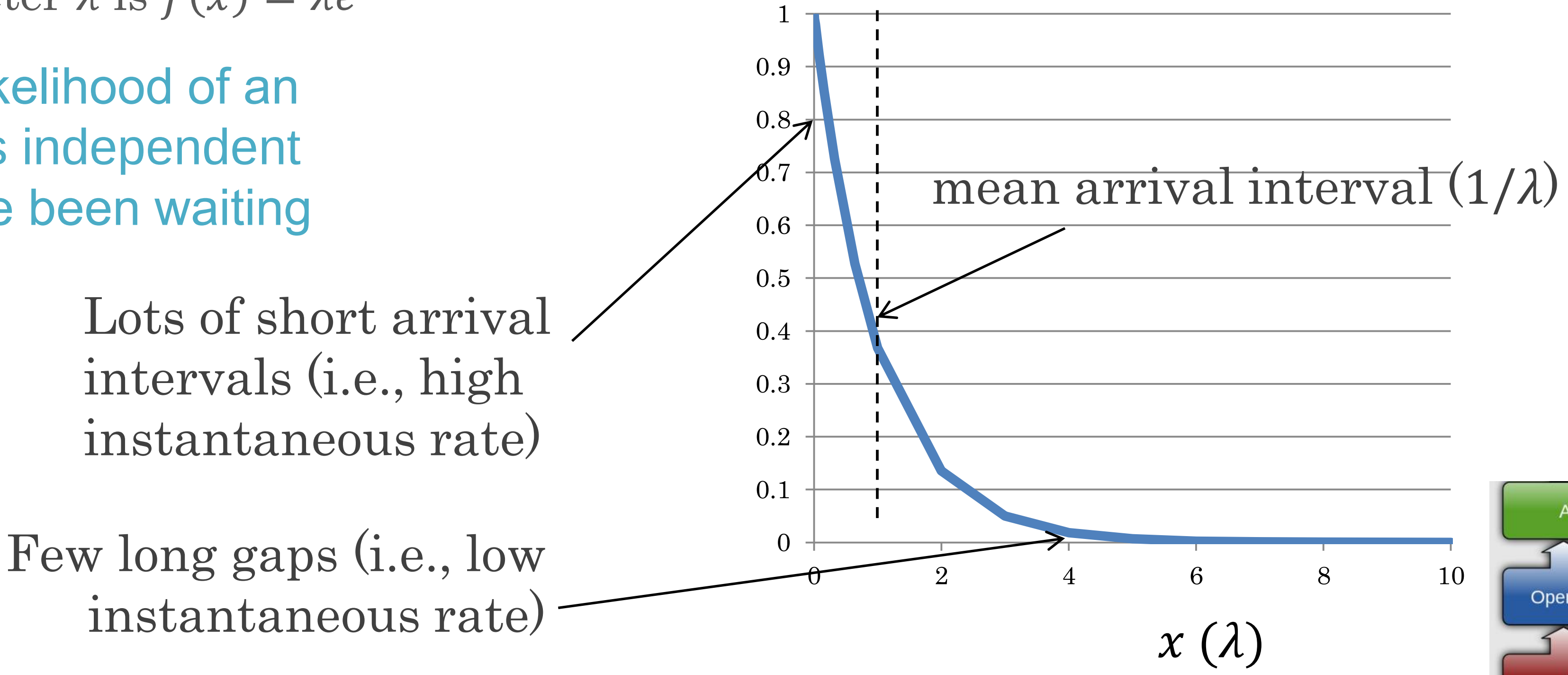
- Requests arrive in a burst, must queue up until served
- Same average arrival time, but almost all of the requests experience large queue delays (even though average utilization is low – $T_S/TA \ll 1$)

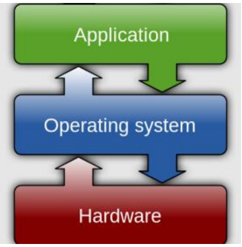


How to model Burstiness of Arrival?

- T_A, the time between arrivals, is now a random variable
  - Elegant mathematical framework if we model it as an exponential distribution
  - Probability distribution function of an exponential distribution with parameter λ is $f(x) = \lambda e^{-\lambda x}$

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



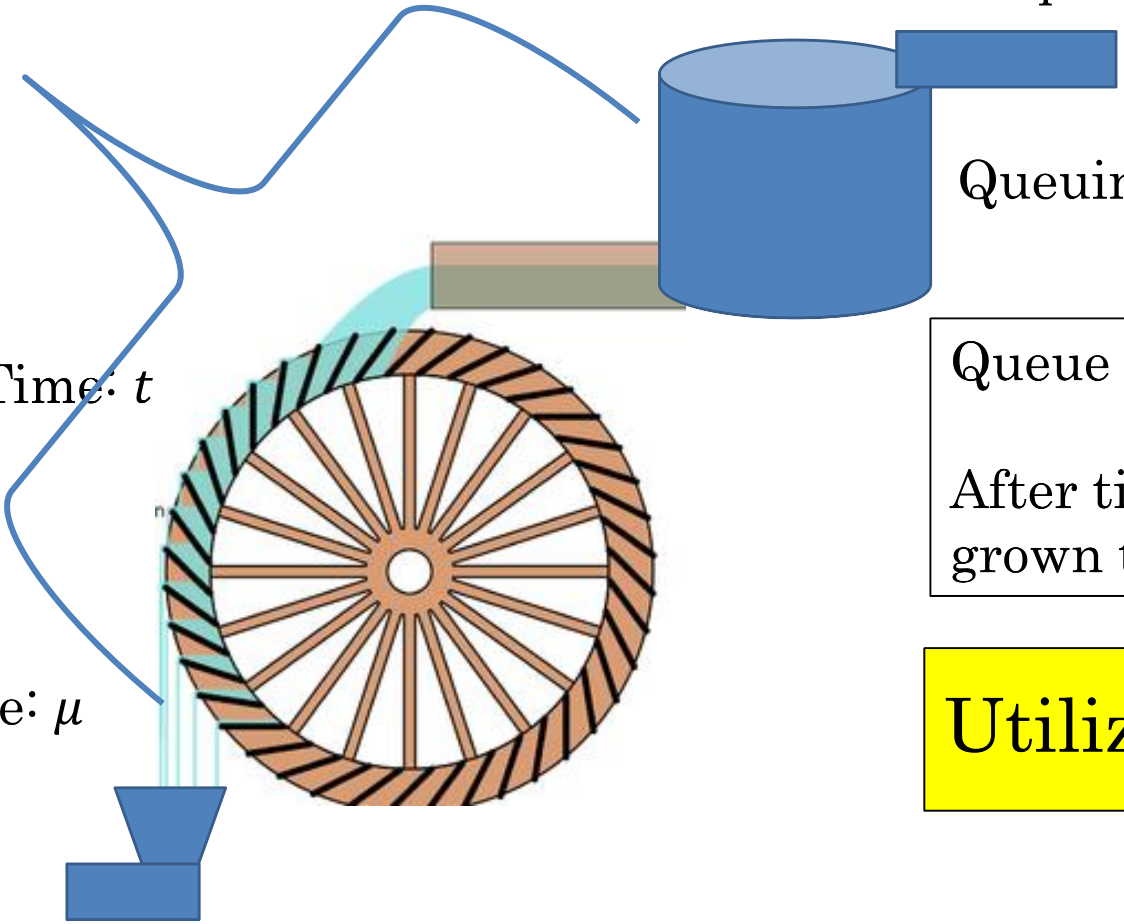


A Simple System Performance Model

Request Rate: λ

Latency (L)

Queuing delay: d



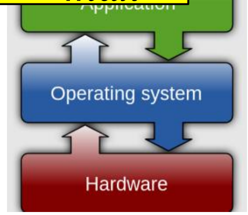
Operation Time: t

Service Rate: μ

Queue grows at rate $\mu - \lambda$

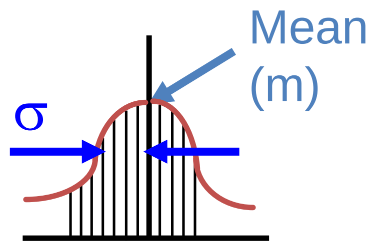
After time t, it will have grown to length $t(\mu - \lambda)$

Utilization: $\rho = \dfrac{\lambda}{\mu_{max}}$



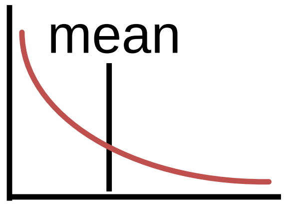
Background: Random Distributions

- Server spends variable time (T) with customers
  - Mean (Average): $m = \sum p(T) \cdot T$
  - Variance (stddev2): $\sigma^2 = \sum p(T) \cdot (T - m)^2$
  - Squared coefficient of variance: $C = \sigma^2 / m^2$

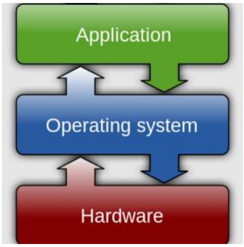


Distribution of service times

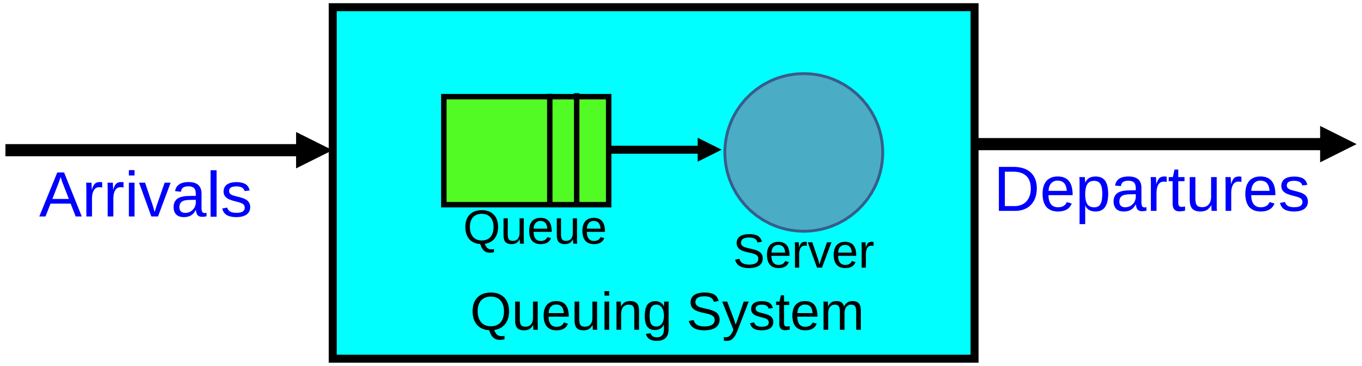
- Important values of C:
  - No variance or deterministic $\Rightarrow C = 0$
  - "Memoryless" or exponential $\Rightarrow C = 1$
    - Past tells nothing about future
    - Poisson process – 'purely' or 'completely' random process
    - Many complex systems (or aggregates) are well described as memoryless



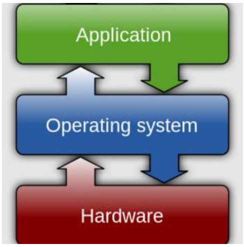
Memoryless



Introduction to Queuing Theory

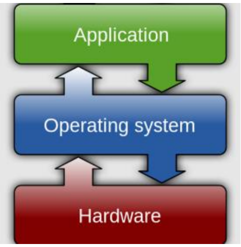


- Queuing Theory applies to long term, steady state behavior
  - Arrival rate = Departure rate ($\lambda = \mu$)
- Arrivals characterized by some probabilistic distribution
- Departures characterized by some probabilistic distribution



Our Goals with Queuing Theory

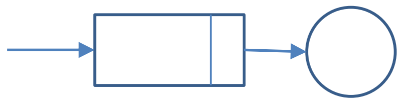
- We wish to compute:
  - T_Q: Time spent in queue
  - L_Q: Length of the queue



Little's Law Applied to a Queue

- Before, we had $n = LB$ (for a stable system):
  - B: bandwidth
  - L: latency
  - n: number of operations in the system

- When applied to a queue, we get:

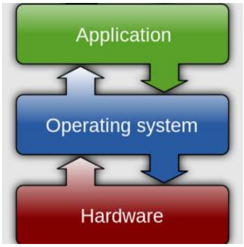


Average Arrival Rate

$$L_Q = \lambda T_Q$$

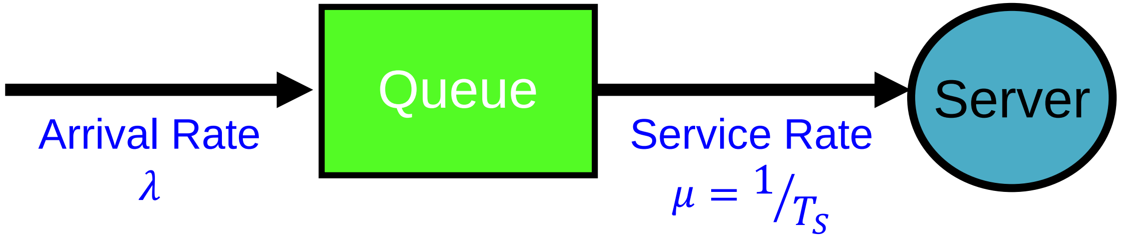
Average length of the queue

Average time "waiting"

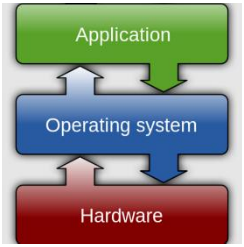


Some Results from Queuing Theory

- Assumptions: system in equilibrium, no limit to the queue, time between successive arrivals is random and memoryless



- λ: arrival rate
- T_S: mean time to service a customer
- C: squared coefficient of variance $\left({}^{\sigma^2}/_{T_S^2}\right)$
- μ: service rate $\left({}^{1}/_{T_S}\right)$
- ρ: utilization $\left({}^{\lambda}/_{\mu}\right)$



Some Results from Queuing Theory

- Memoryless service distribution ($C = 1$) - an "M/M/1 queue":

$$T_Q = \frac{\rho}{1-\rho} \cdot T_S$$

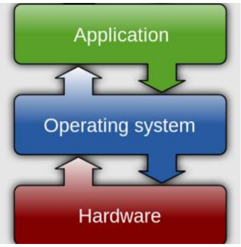
- General service distribution (no restrictions) - an "M/G/1 queue":

$$T_Q = \frac{1+C}{2} \cdot \frac{\rho}{1-\rho} \cdot T_S$$

- λ: arrival rate
- T_S: mean time to service a customer
- C: squared coefficient of variance (σ^2/T_S^2)
- μ: service rate ($1/T_S$)
- ρ: utilization (λ/μ)

M/M/1:
Input: Markovian (Poisson)
Output: Markovian (Poisson)
Number of servers: 1

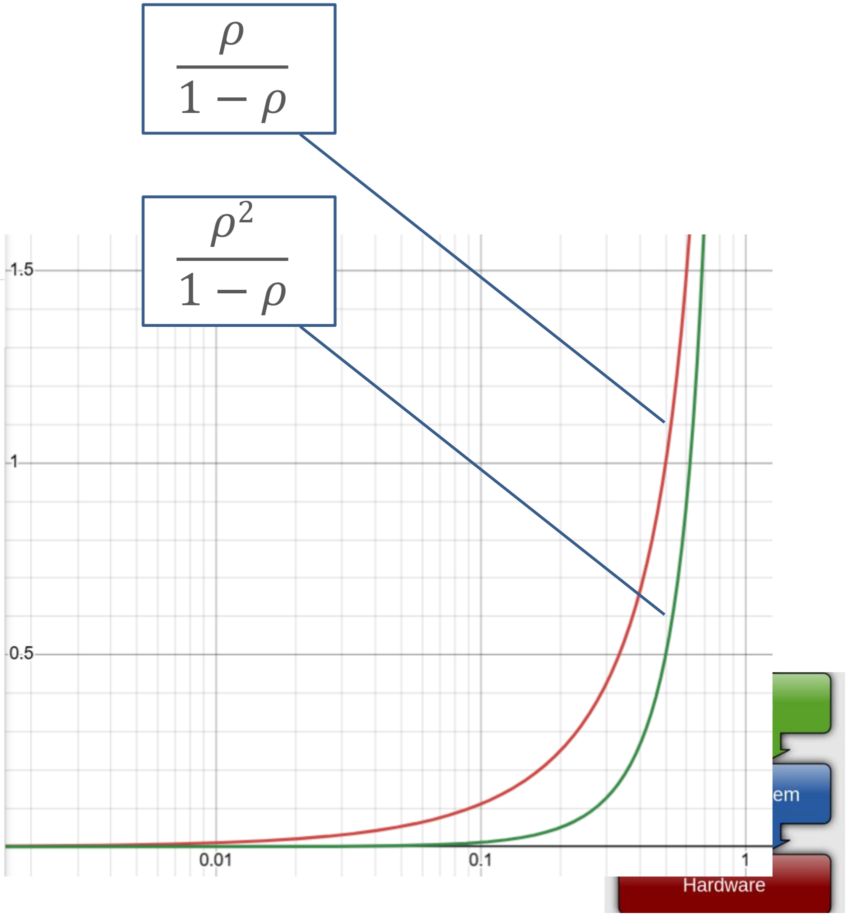
M/G/1:
Input: Markovian (Poisson)
Output: General distribution
Number of servers: 1



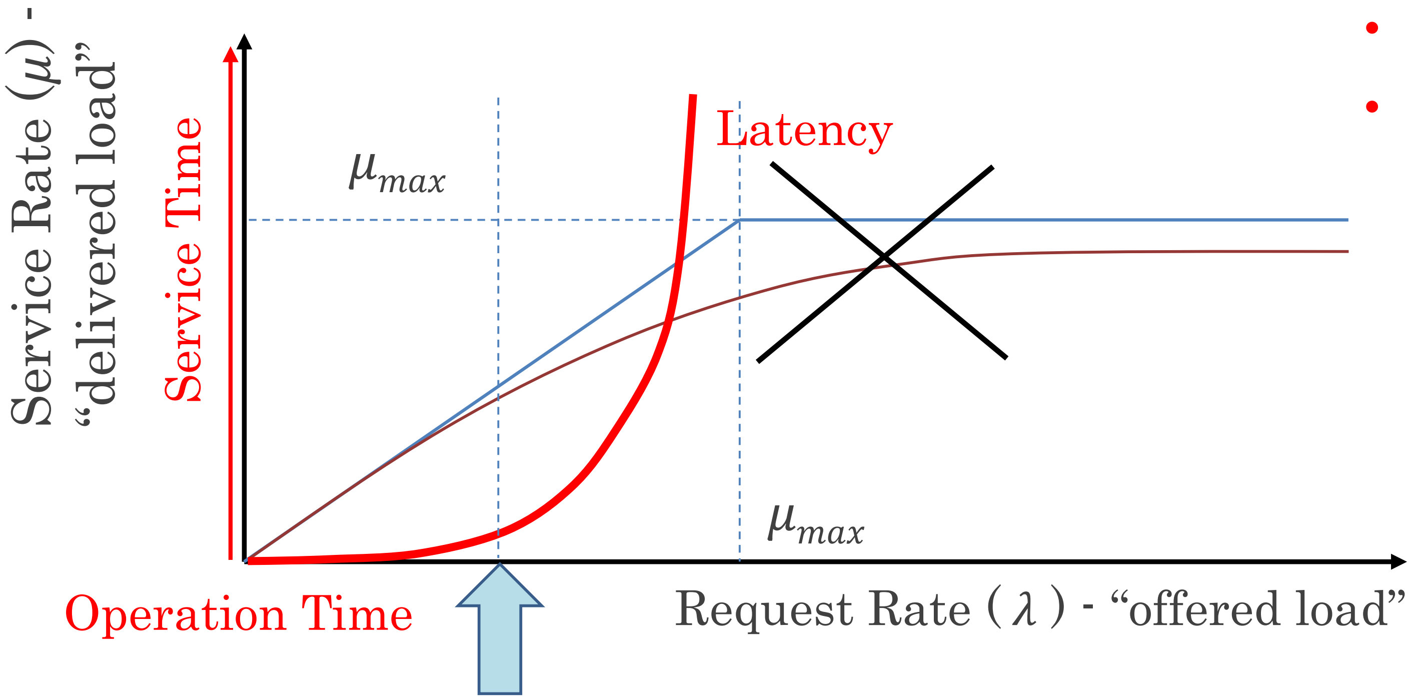
Key Results from Queuing Theory

- $T_Q = \frac{\rho}{1-\rho} \cdot T_S$ (memoryless service distribution)
- $L_Q = \lambda T_Q$ (by Little's Law)

- Utilization is $\rho = {}^{\lambda}/_{\mu_{max}} = \lambda T_S$, so
- $L_Q = \lambda T_Q = \frac{\rho}{T_S} \cdot T_Q = \frac{\rho^2}{1-\rho}$ (for a single server)



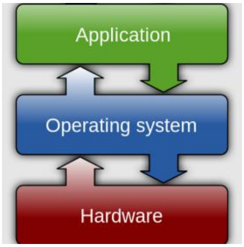
Ideal System Performance



- $T_Q \sim \frac{\rho}{1-\rho}$, $\rho = \lambda / \mu_{max}$
- 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

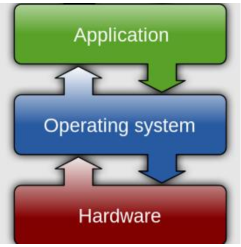
"Half-Power Point": load at which system delivers half of peak performance

- Design and provision systems to operate roughly in this regime
- Latency low and predictable, utilization good: ~50%

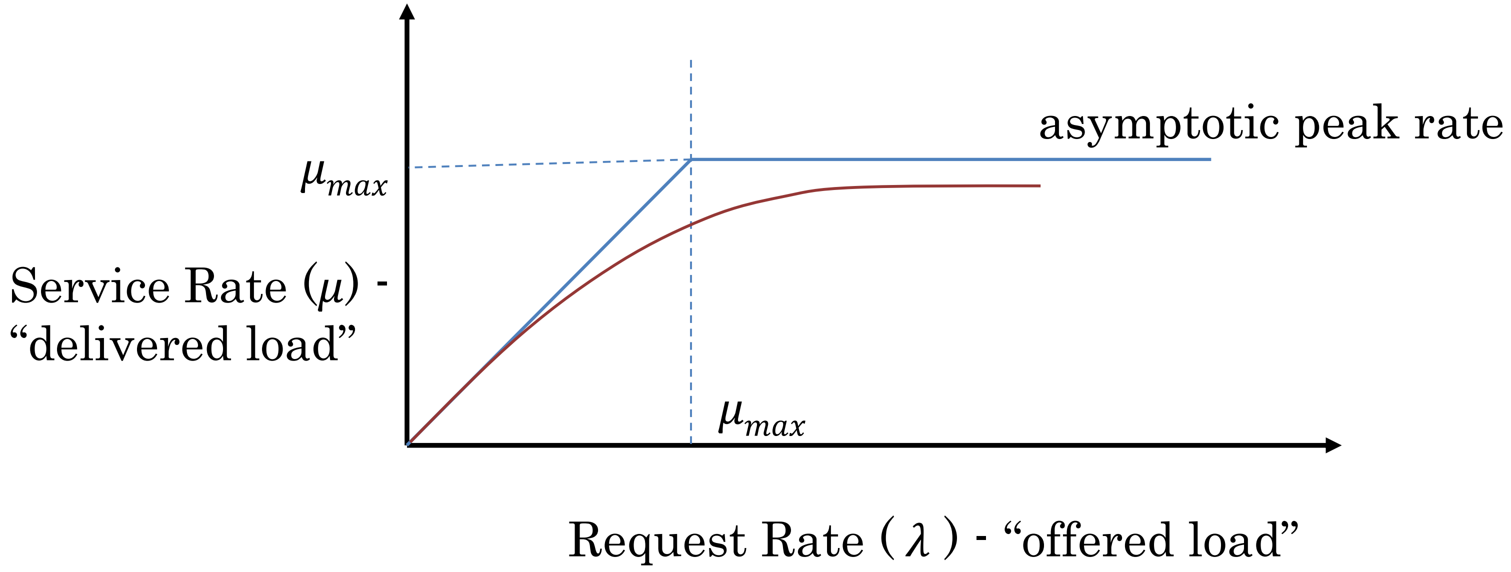


Rest of Today's Lecture

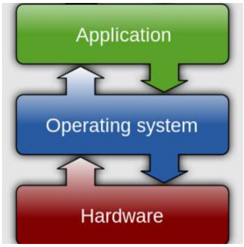
- Using this system model, we will:
  - Explore latency in more depth
  - Discuss how to build systems that perform well under load



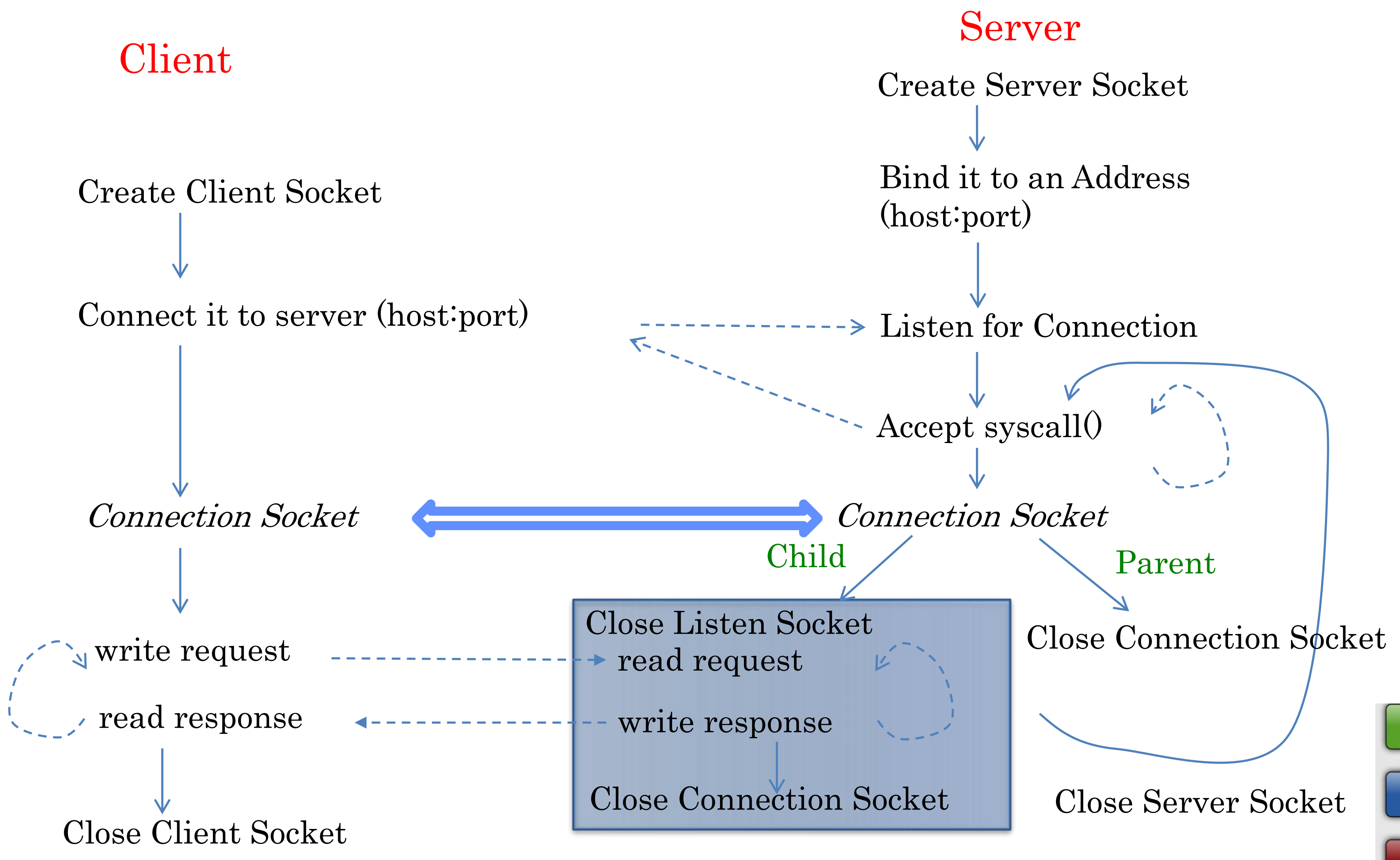
Ideal System Performance

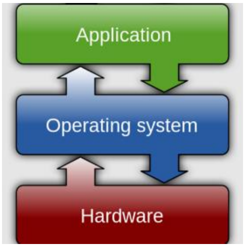


- A system that behaves this way is well-conditioned
  - Delivered load increases with offered load until pipeline saturates
  - As offered load increases further, throughput remains high



Sockets with Protection and Concurrency





Sockets with Concurrency, without Protection

Client

Create Client Socket

Connect it to server (host:port)

Connection Socket

write request

read response

Close Client Socket

Server

Create Server Socket

Bind it to an Address (host:port)

Listen for Connection

Accept syscall()

Connection Socket

Spawned Thread

pthread_create

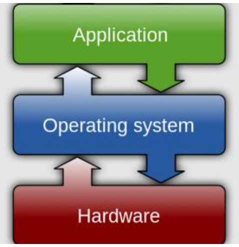
Main Thread

read request

write response

Close Connection Socket

Close Server Socket



Non-Well-Conditioned Systems

- A server that spawns a new pthread per request is not well-conditioned!

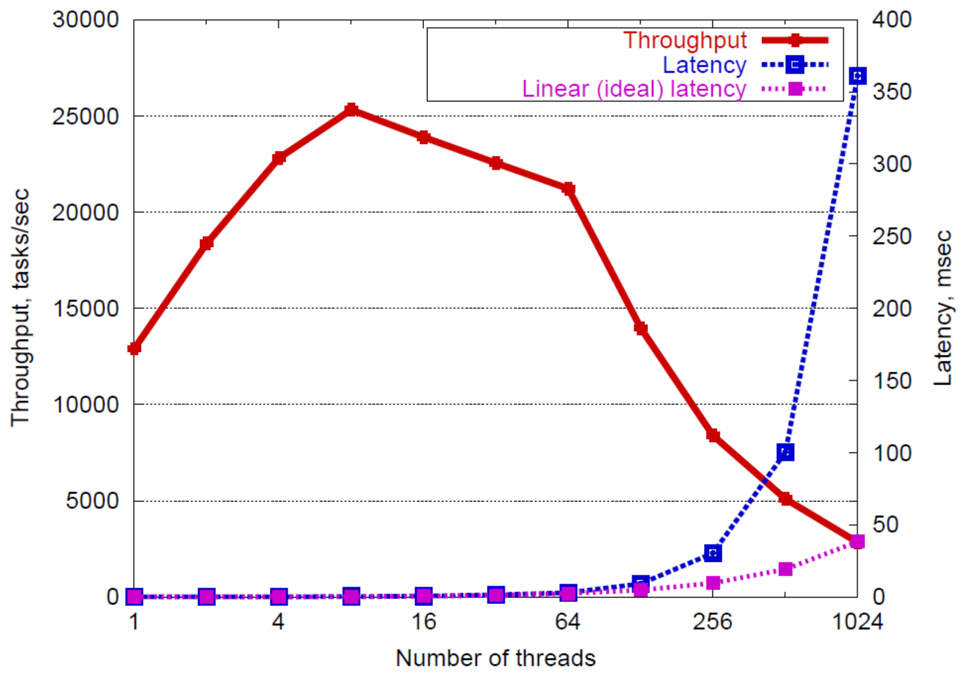
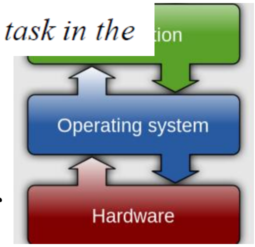


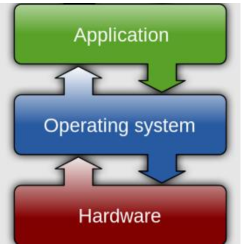
Figure 2: **Threaded server throughput degradation:** *This benchmark measures a simple threaded server which creates a single thread for each task in the*



https://people.eecs.berkeley.edu/~prabal/teaching/resources/eecs582/welsh01seda.pdf

Building Well-Conditioned Systems

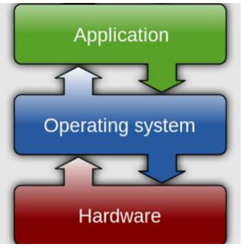
- Spawning a new thread or process for each request is not well-conditioned
- Too many threads is bad
  - Scheduling overhead becomes large
  - Context switch overhead becomes large
    - E.g., Poor cache performance
  - Synchronization overhead becomes large
    - E.g., Lock contention
- Was our original (v1) server well-conditioned?
  - The one that handles requests one at a time, with no concurrency?
  - Hint: yes!

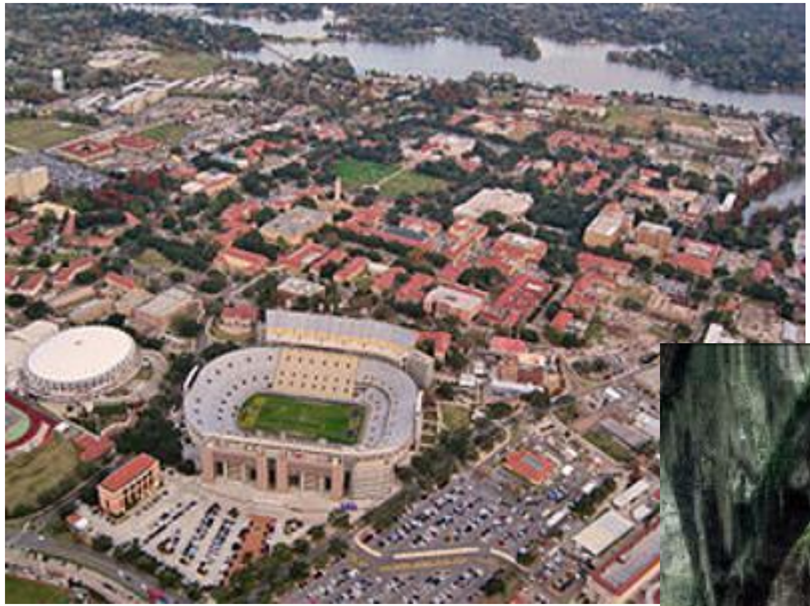


Building Well-Conditioned Systems

- Thread Pools
- User-Mode Threads
- Event-Driven Execution

- We'll discuss these next time…





LSU CENTER FOR COMPUTATION & TECHNOLOGY

STE||AR GROUP