System Performance and Highly Concurrent Systems

Sam Kumar

CS 162: Operating Systems and System Programming

Lecture 14

https://inst.eecs.berkeley.edu/~cs162/su20

Read: A&D 7.5,
SEDA Section 2

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:

1. Deadlock avoidance: dynamically delay resource requests so deadlock doesn't happen
2. Deadlock prevention: write your code in a way that it isn't prone to deadlock
3. Deadlock recovery: let deadlock happen, and then figure out how to recover from it
4. 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)

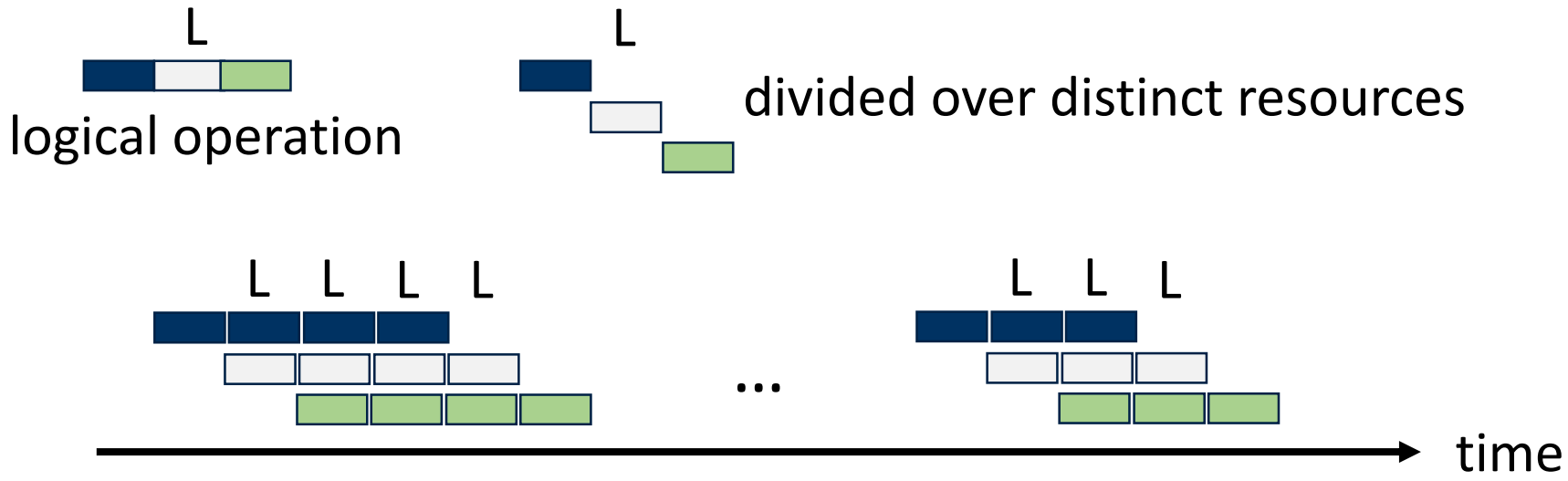
- ***Latency*** – time to complete a task
  - Measured in units of time (s, ms, us, …, hours, years)
- ***Response Time*** - time to initiate and operation and get its response
  - Able to issue one that *depends* on the result
  - Know that it is done (anti-dependence, resource usage)
- ***Throughput*** or ***Bandwidth*** – rate at which tasks are performed
  - Measured in units of things per unit time (ops/s, GLOP/s)
- Performance???
  - Operation time (4 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, …)
  - $L = 10\ \text{ms} \rightarrow B = 100\ {}^{\text{op}}/_{\text{s}}$
  - $L = 2\ \text{yr} \rightarrow B = 0.5\ {}^{\text{op}}/_{\text{yr}}$

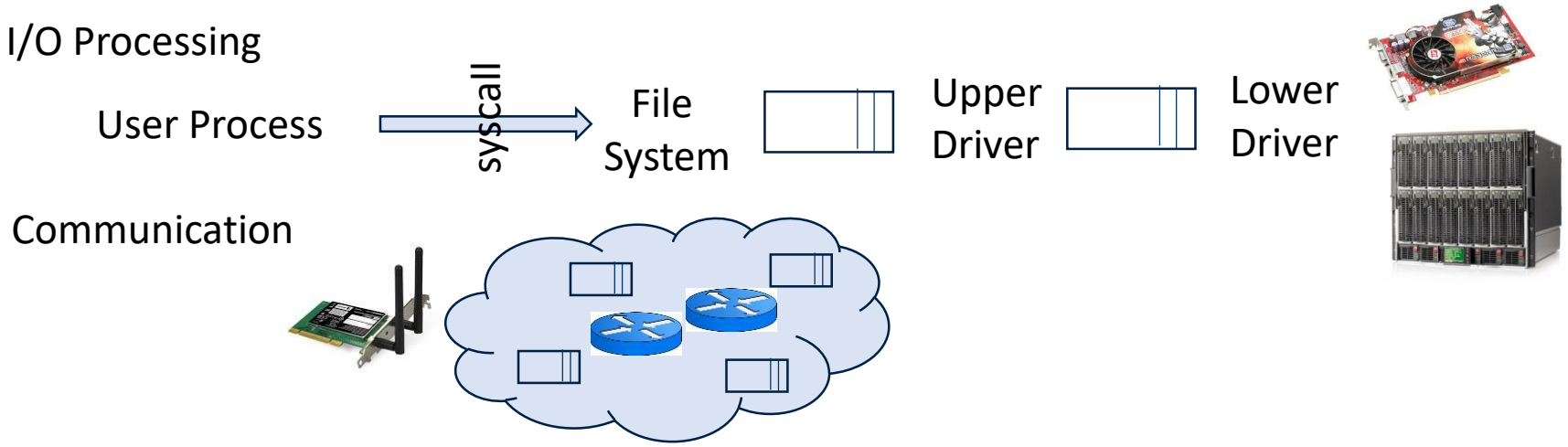
- Applies to a processor, a disk drive, a person, a TA, …

Single Pipelined Server



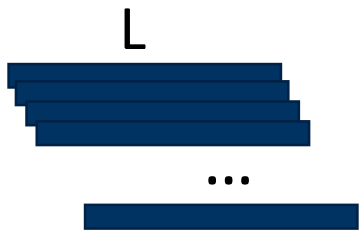
- Single pipelined server of k stages for tasks of length L (i.e., time L/k per stage) delivers at rate $\leq k/L$.
  - $L = 10$ ms, $k = 4 \rightarrow B = 400$ op/s
  - $L = 2$ yr, $k = 2 \rightarrow B = 1$ op/yr

Example Systems "Pipelines"



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

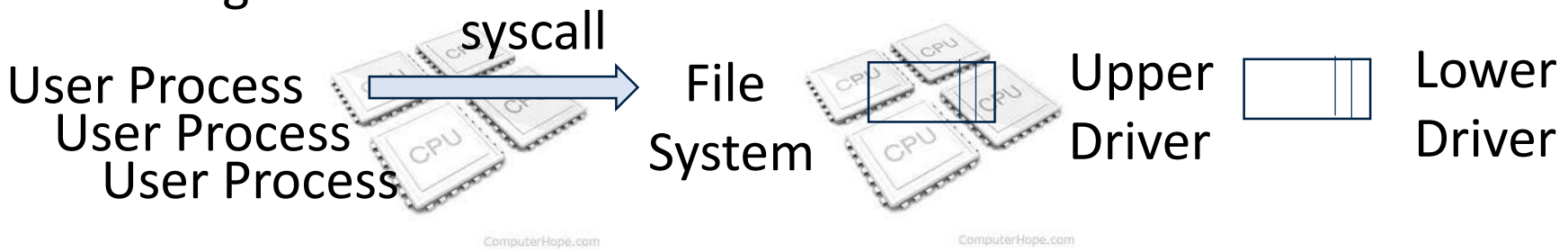
Multiple Servers



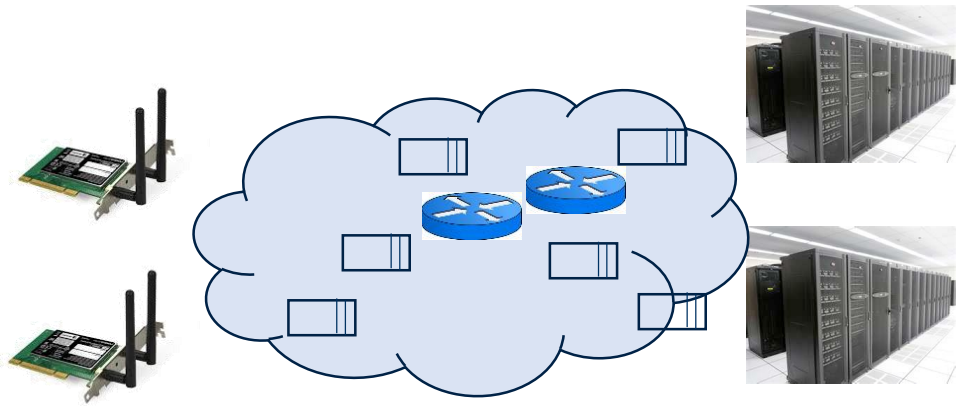
- k servers handling tasks of length L delivers at rate $\leq {}^{k}/_{L}$.
  - $L = 10$ ms, $k = 4 \rightarrow B = 400\ {}^{\text{op}}/_{\text{s}}$
  - $L = 2$ yr, $k = 2 \rightarrow B = 1\ {}^{\text{op}}/_{\text{yr}}$
- In 61C you saw multiple processors (cores)
  - Systems present lots of multiple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

Example Systems "Parallelism"

I/O Processing



Communication

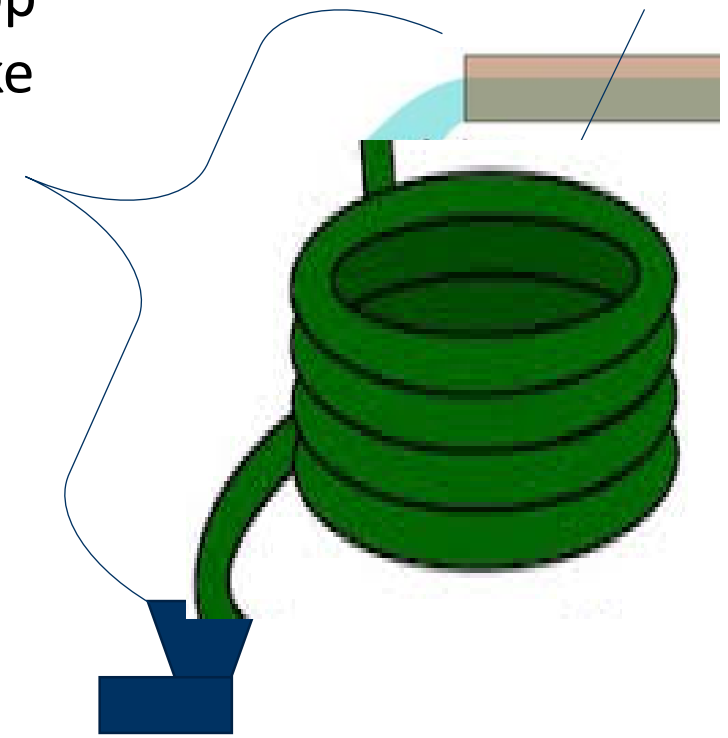


Parallel Computation, Databases, …

A Simple Systems Performance Model

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

"Service Time"



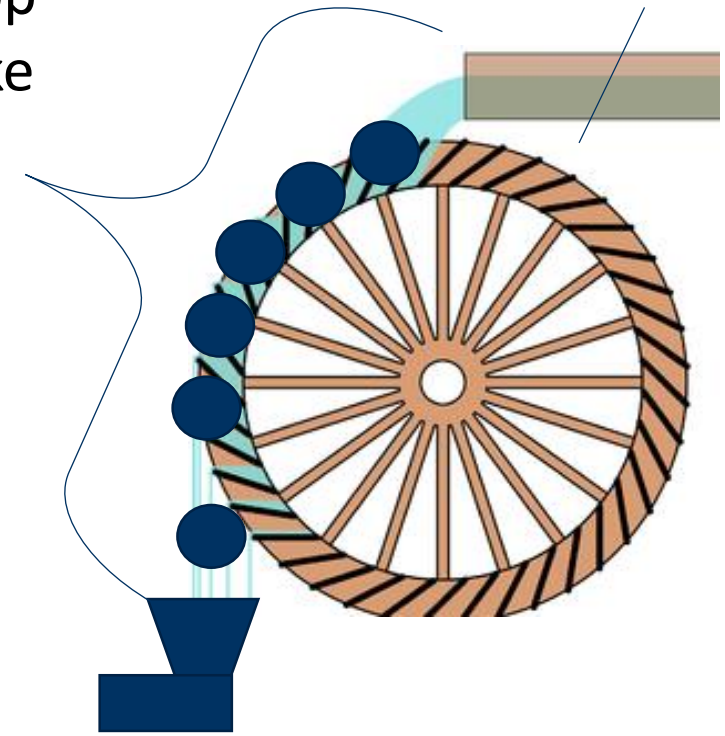
Bandwidth (B): Rate, Op/s
e.g., flow: gal per min

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

A Simple Systems 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 $B = 2\ {}^{\text{op}}/_{\text{s}}$ and $\text{L} = 3\ \text{s}$
How many ops are "in the system?"

Little's Law

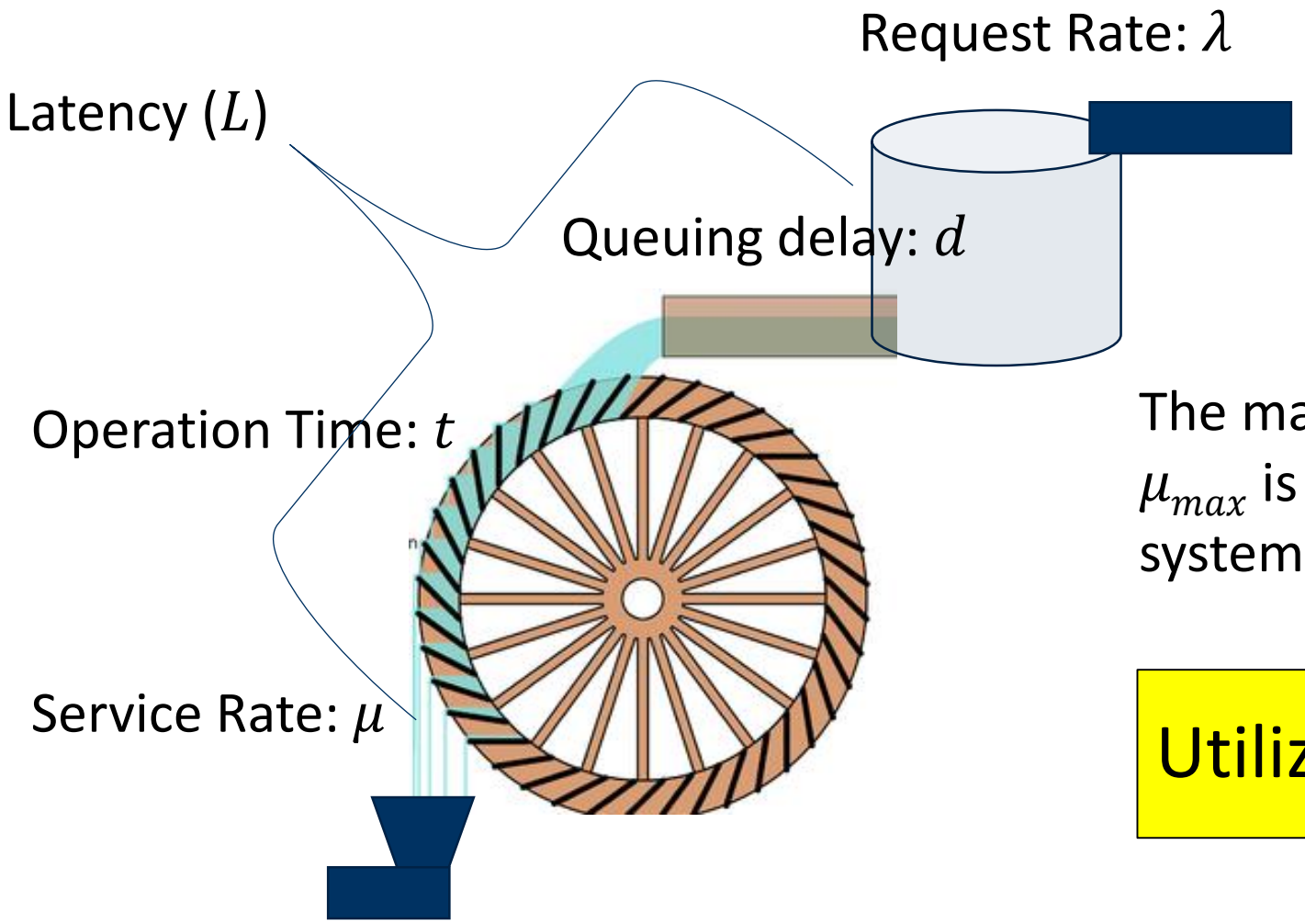
- The number of "things" in a system is equal to the bandwidth times the latency (on average)

$$n = L\,B$$

- 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 Systems Performance Model

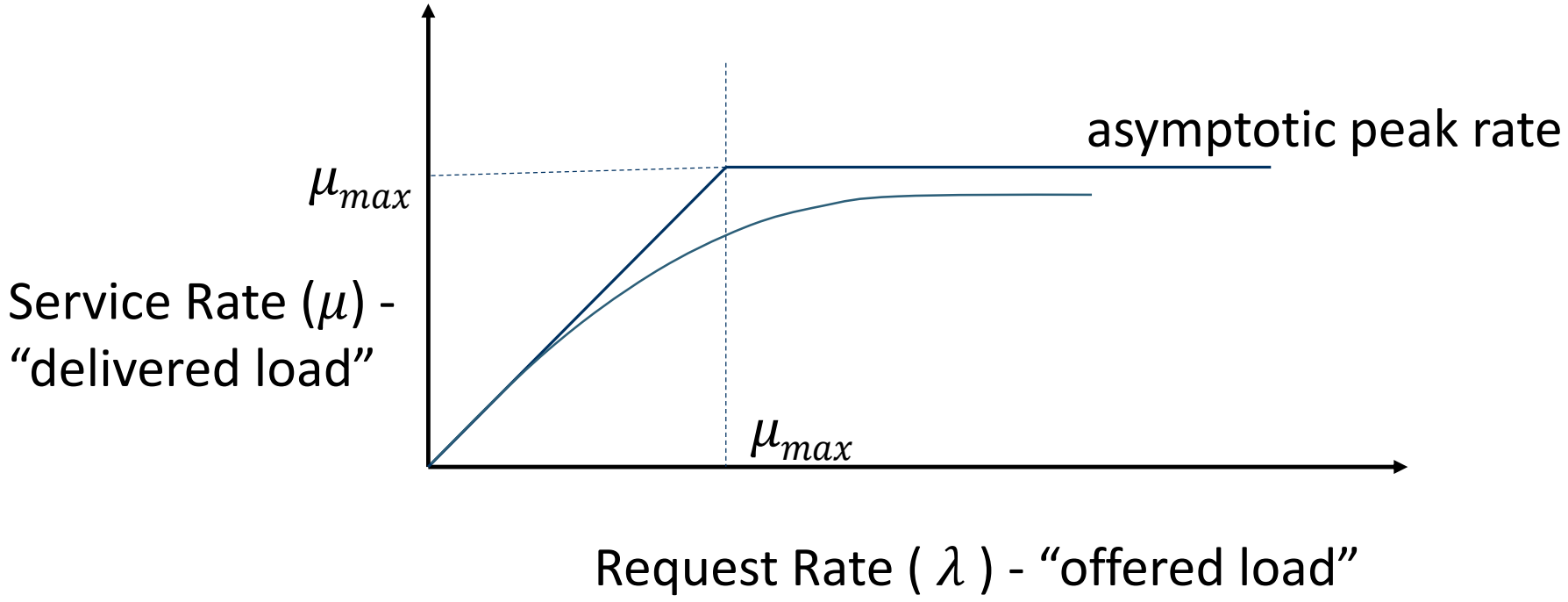


The maximum service rate μ_{max} is a property of the system – the "bottleneck"

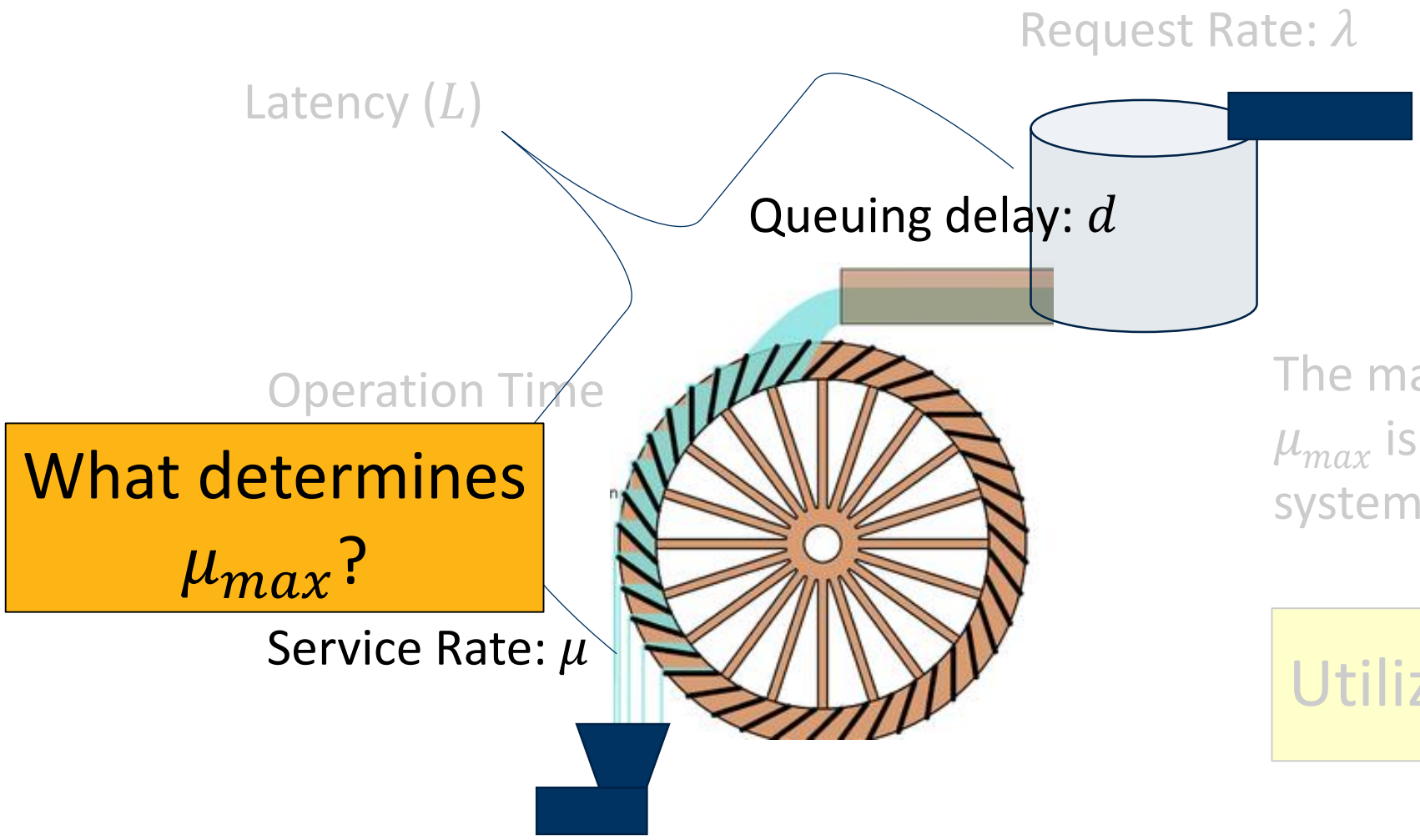
Utilization: $\rho = \frac{\lambda}{\mu_{max}}$

Ideal System Performance

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



Two Related Questions



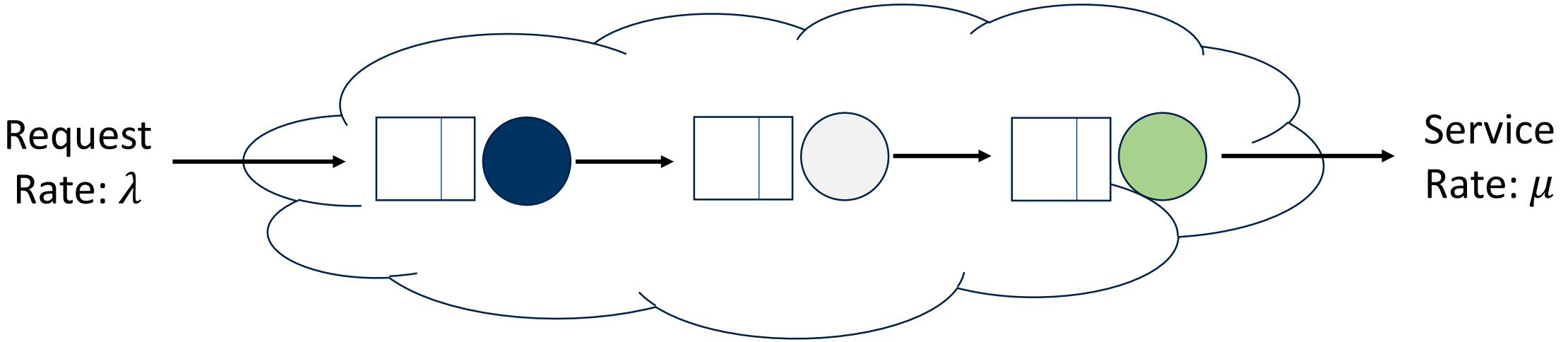
What about "internal" queues?

What determines μ_{max}?

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

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

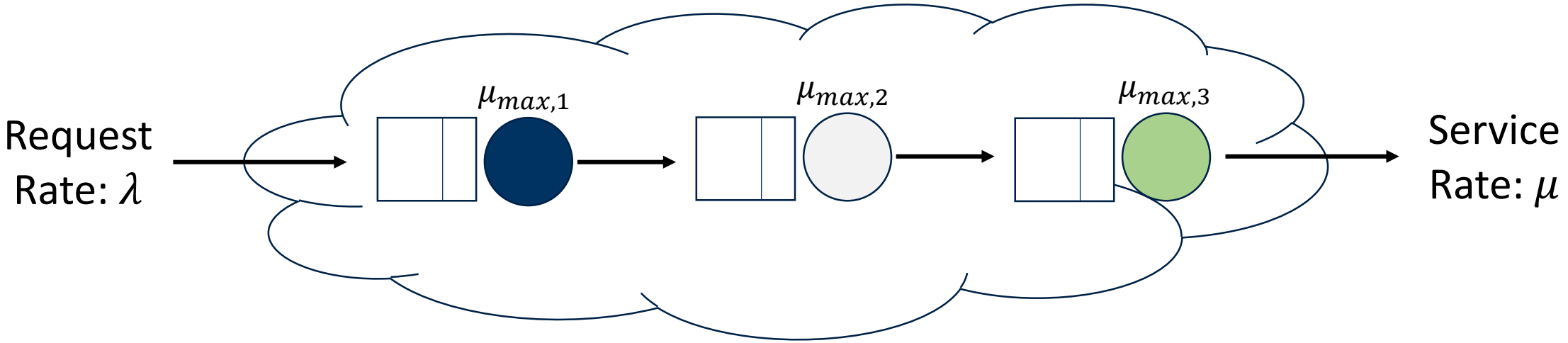
Bottleneck Analysis



Overall System: Series of Stages

Bottleneck Analysis

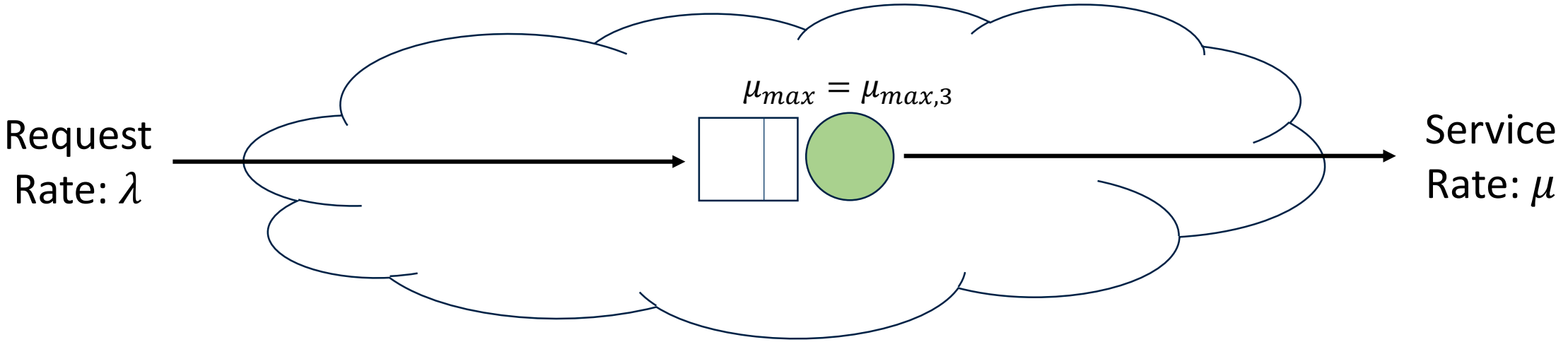
- Each stage has its own queue and maximum service rate
- Suppose the green stage is the bottleneck



Overall System: Series of Stages

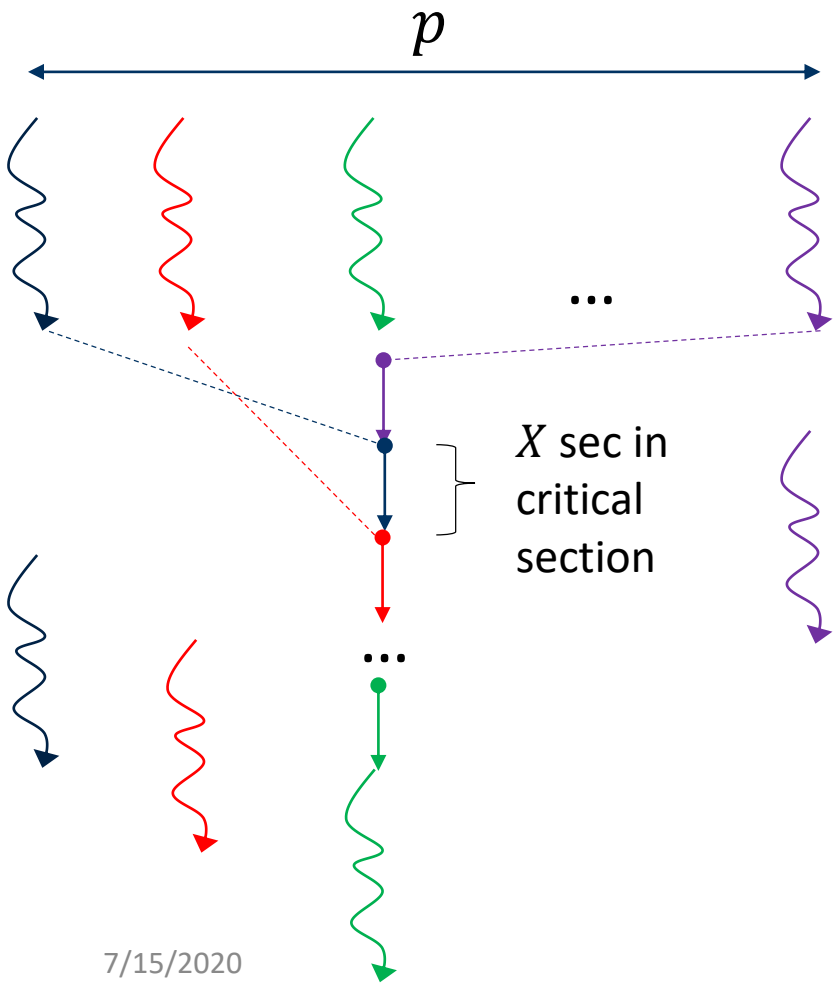
Bottleneck Analysis

- 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

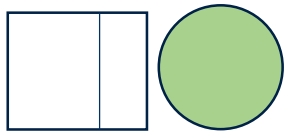


All try to grab lock

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

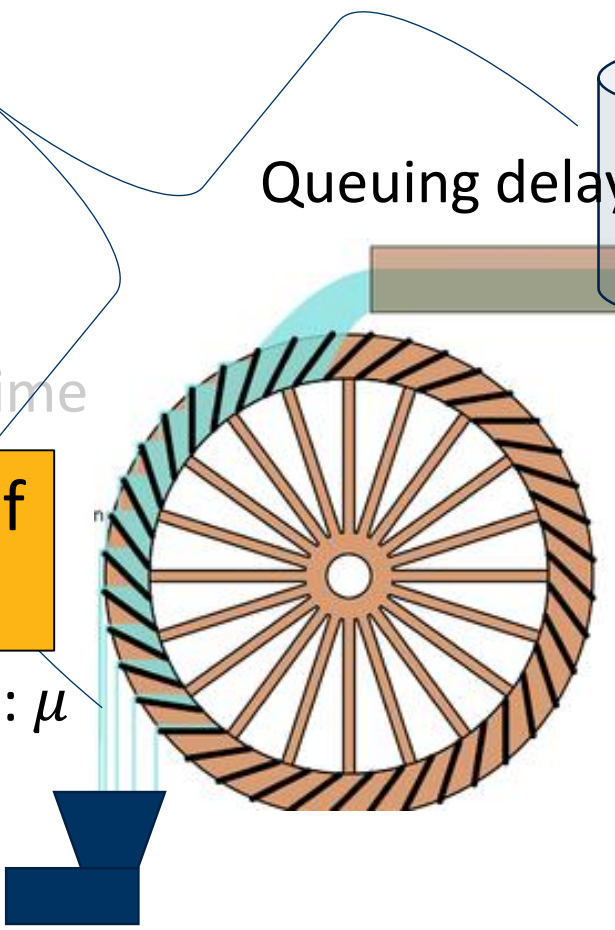
Queue of waiting threads

$\mu_{max} = {}^1/_X$



Critical section guarded by lock

Two Related Questions



Tank represents queue of bottleneck stage

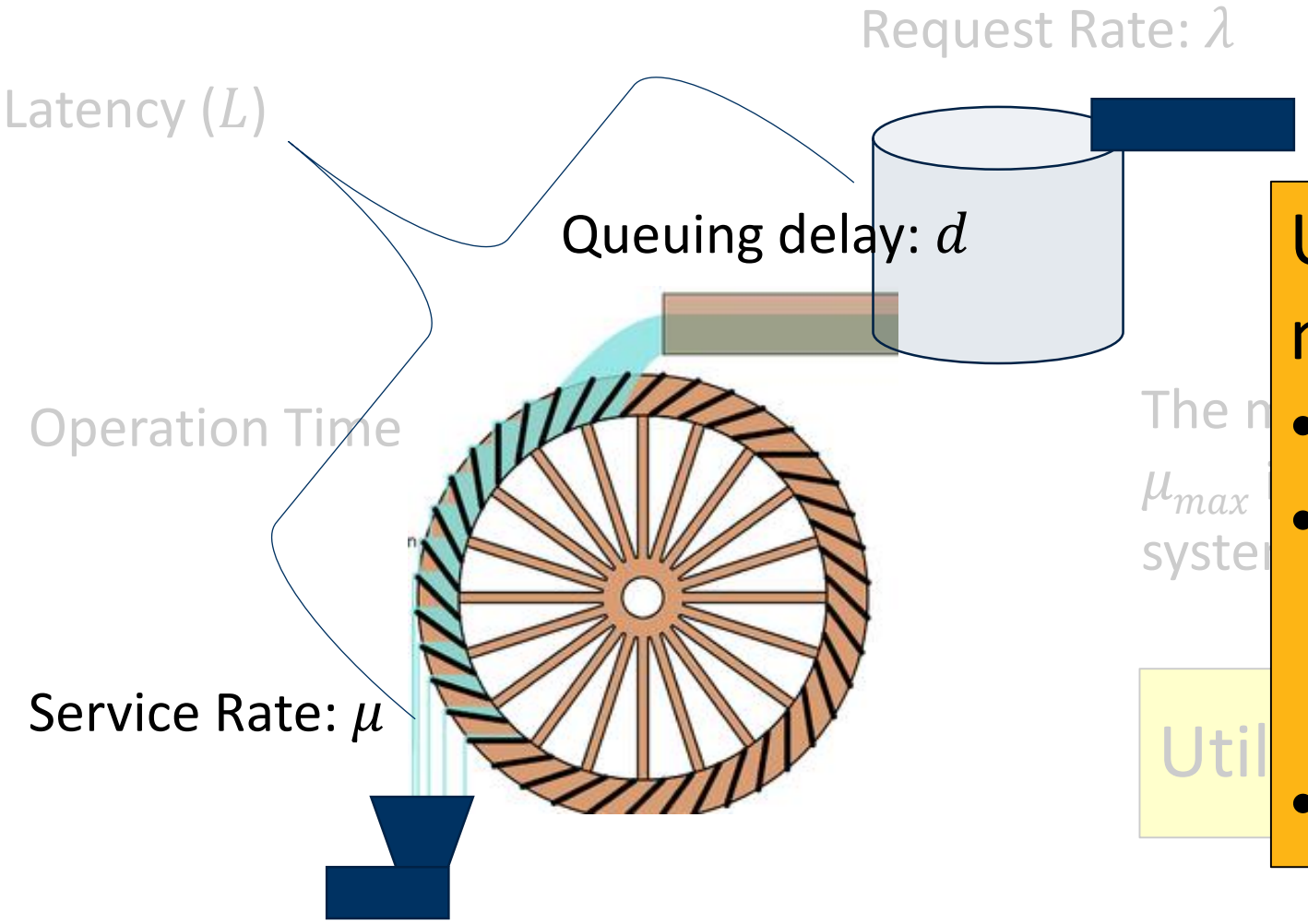
- Including queues of previous stages, in case of backpressure

μ_{max} is service rate of bottleneck stage

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

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

A Simple Systems Performance Model



Useful to apply this model to:

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

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

Announcements

- Homework 3 is due on Friday

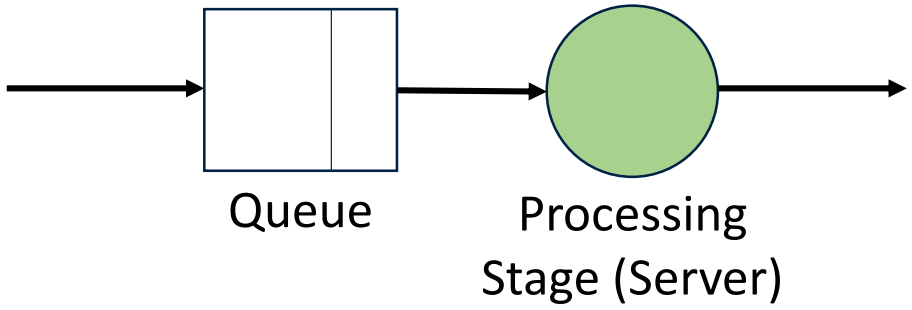
- Change in policy: if you use a slip day on project, your final report deadline also gets pushed back a day

Rest of Today's Lecture

Using this system model, we will:

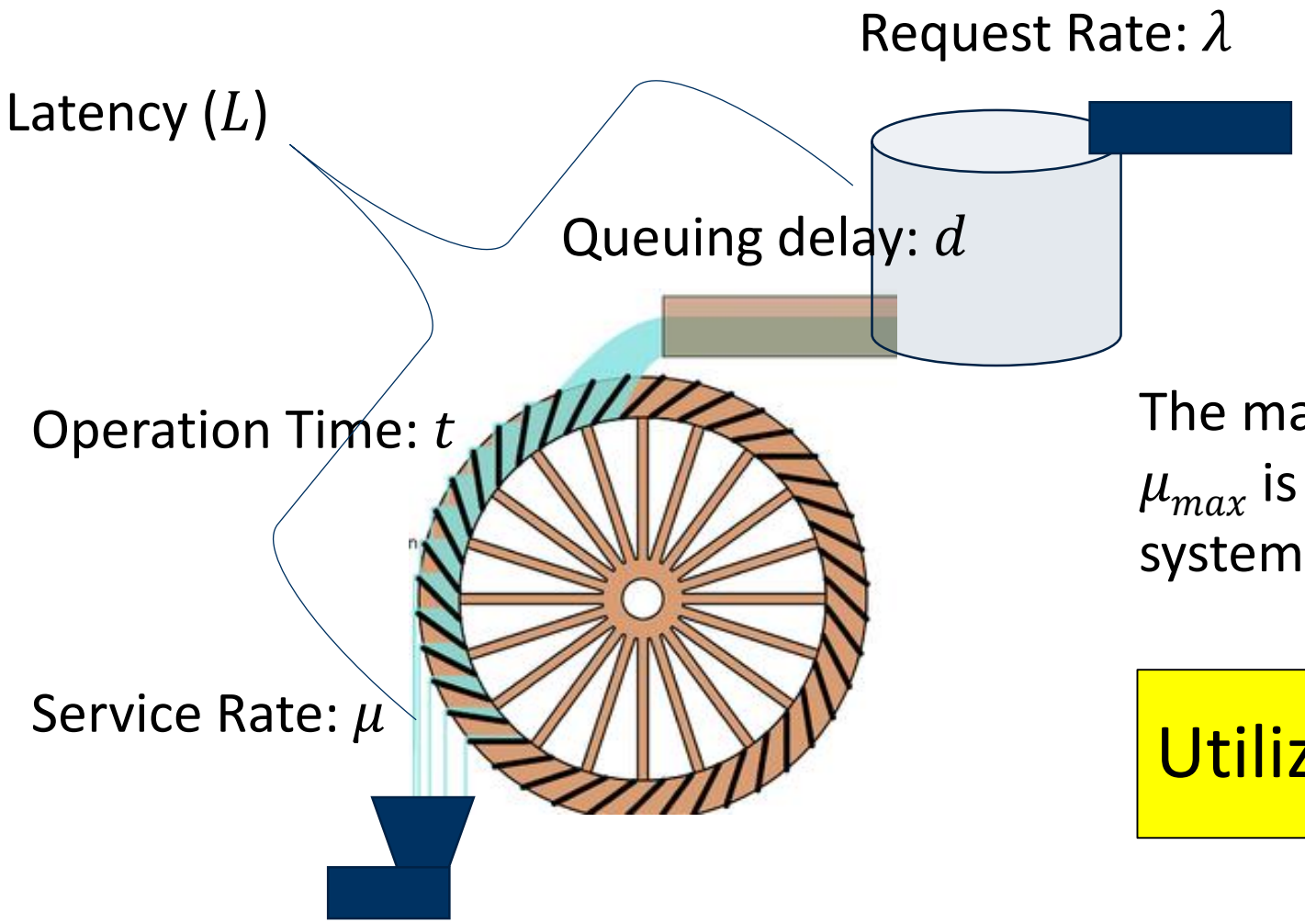
- **Explore latency in more depth**
- Discuss how to build systems that perform well under load

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 Systems Performance Model



When will the queue(s) start to fill?

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

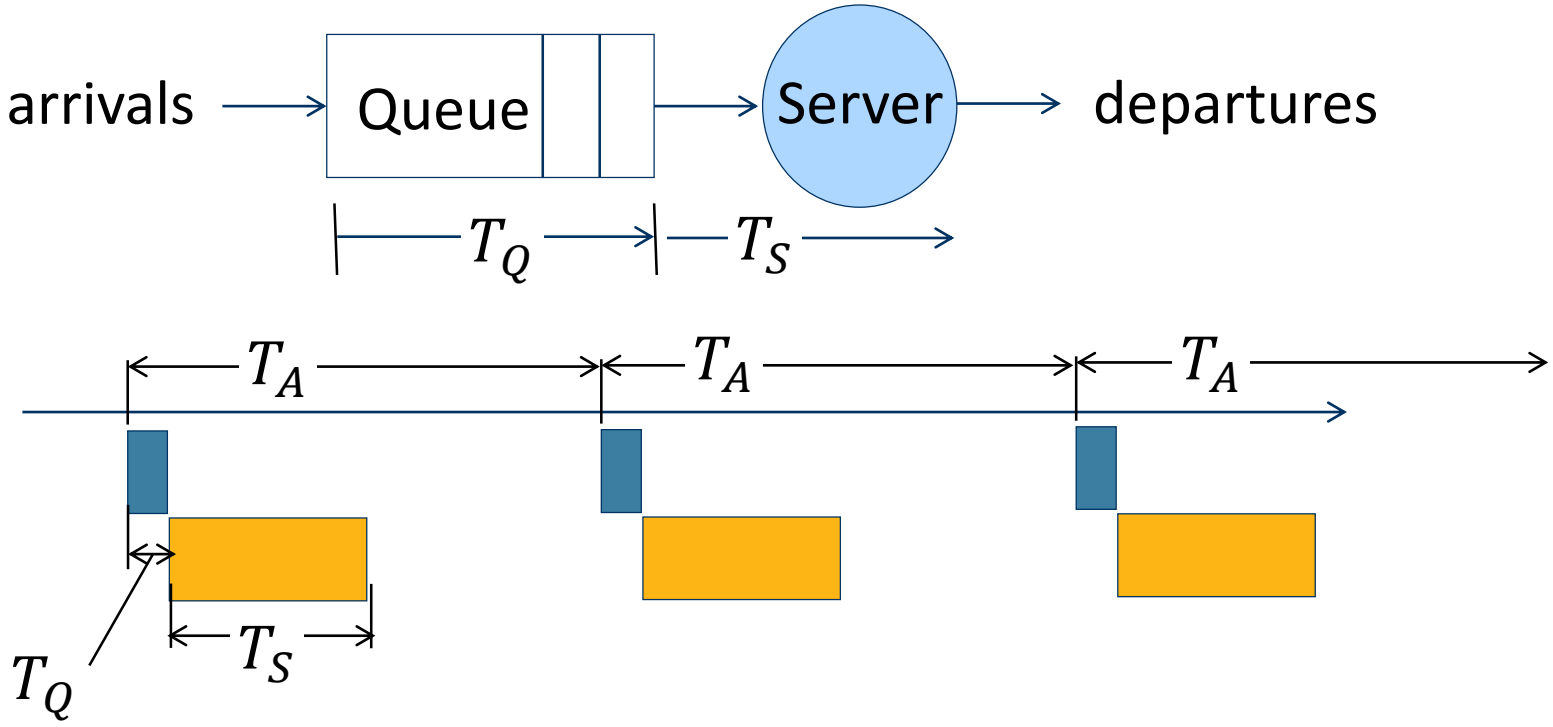
Utilization: $\rho = \frac{\lambda}{\mu_{max}}$

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

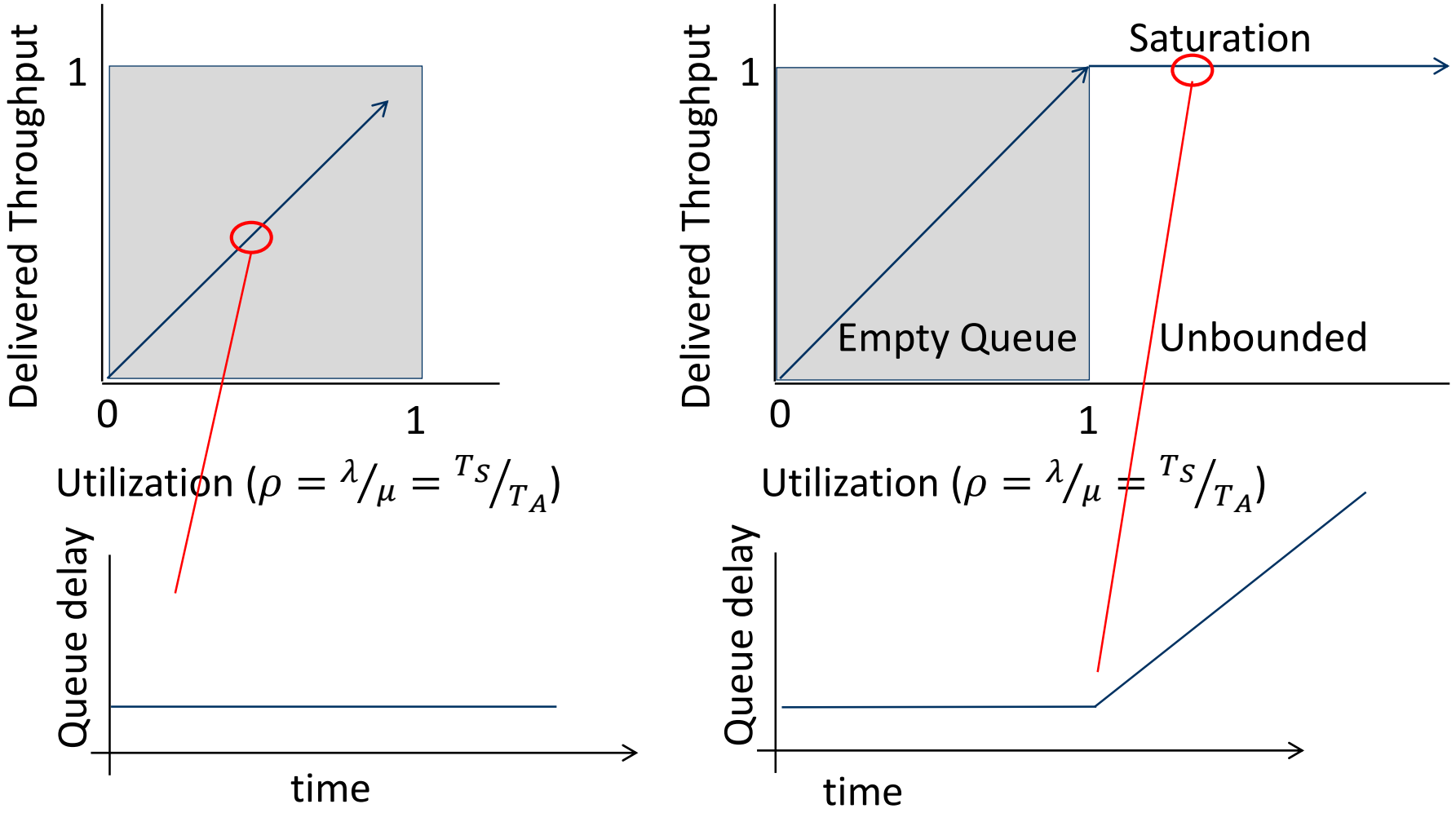
A Simple, Deterministic World

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



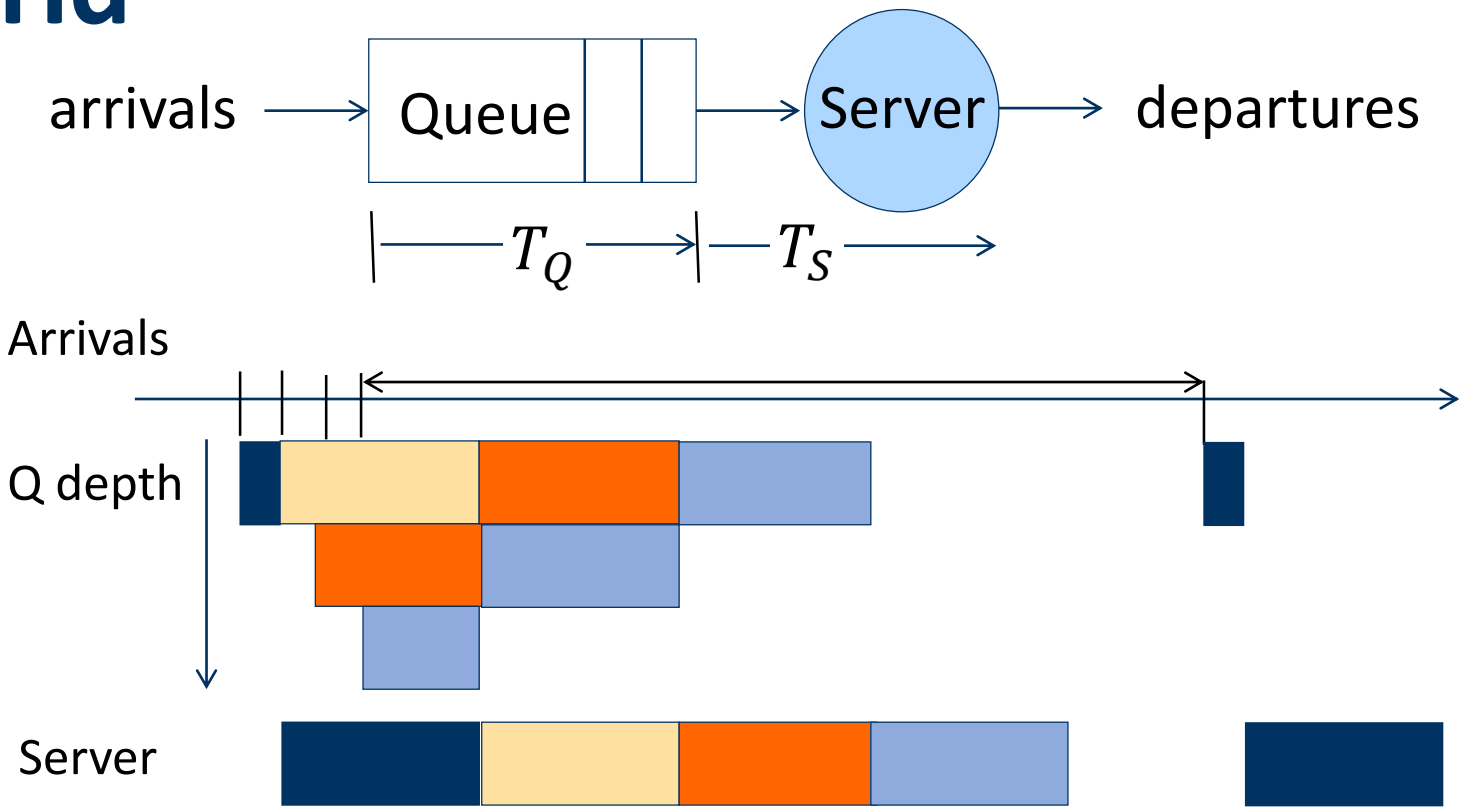
- Assume requests arrive at regular intervals, take a fixed time to process, with plenty of time between …

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
  - $L = T_Q + T_S$

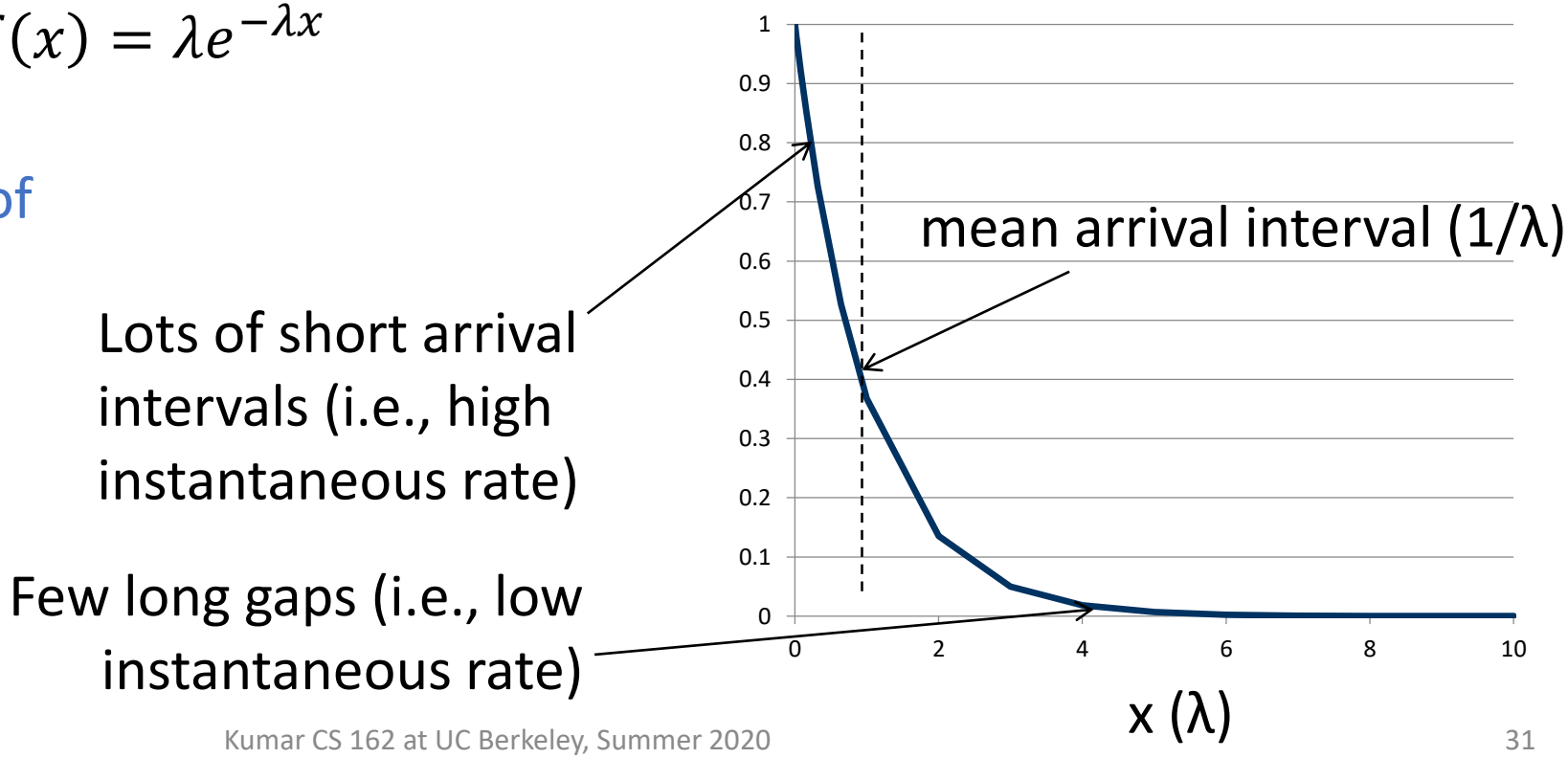


- Requests arrive in a burst, must queue up until served
- Same average arrival time, but almost all of the requests experience large queue delays (even though average utilization is low)

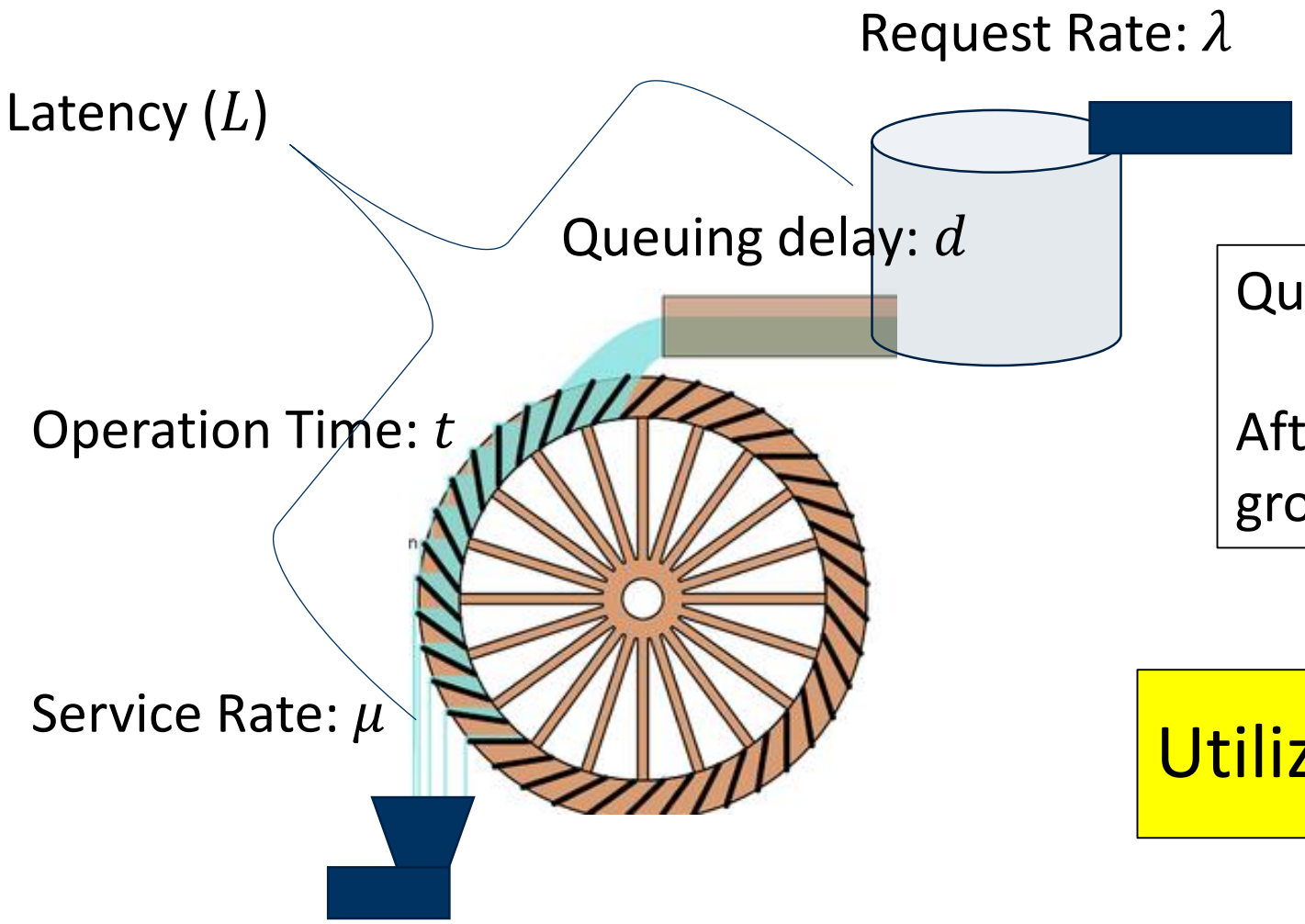
How to model burstiness of arrival?

- 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 Systems Performance Model



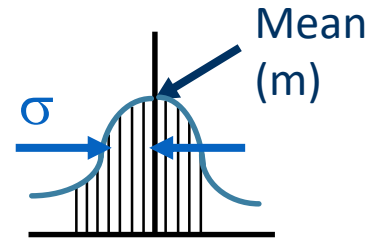
Queue grows at rate $\mu - \lambda$

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

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

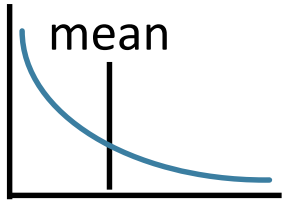
Background: Random Distributions

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



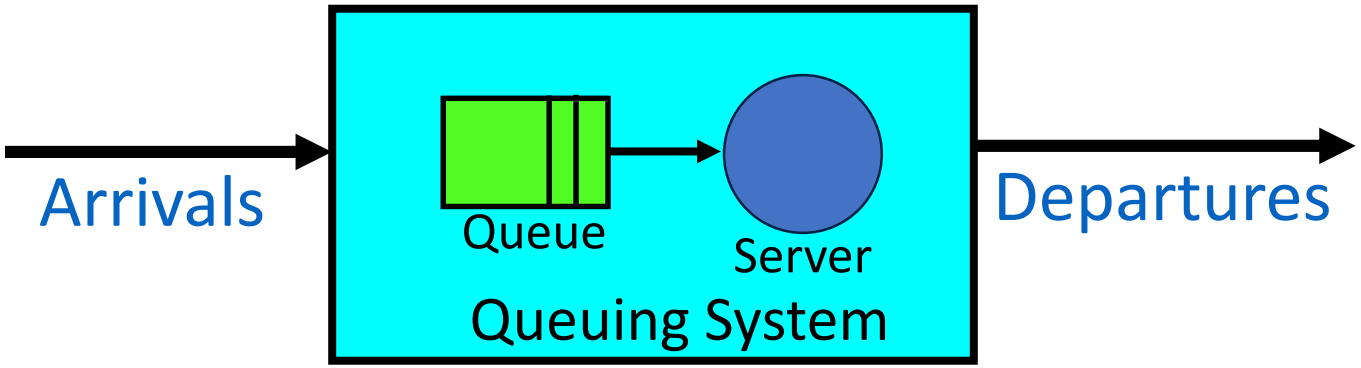
Distribution
of service times

- 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
- Arrivals characterized by some probabilistic distribution
- Departures characterized by some probabilistic distribution

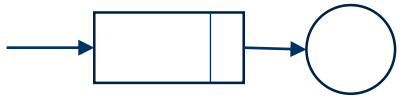
Our Goals with Queuing Theory

- 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

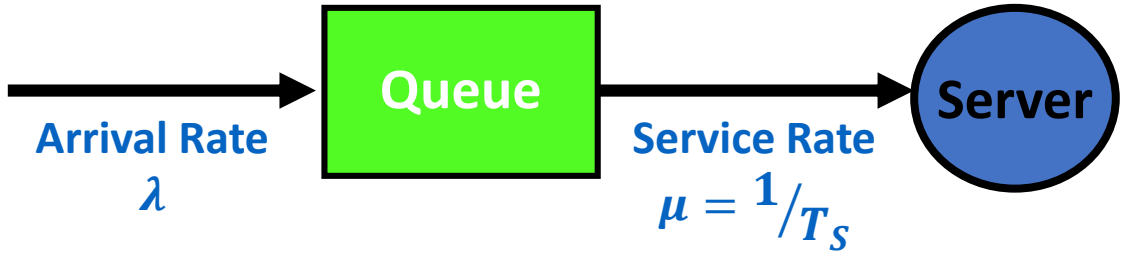
Average time "waiting"

Our Goals with Queuing Theory

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

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 (${}^{\sigma^2}/_{T_S^2}$)
- μ: service rate (${}^{1}/_{T_S}$)
- ρ: utilization (${}^{\lambda}/_{\mu}$)

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)

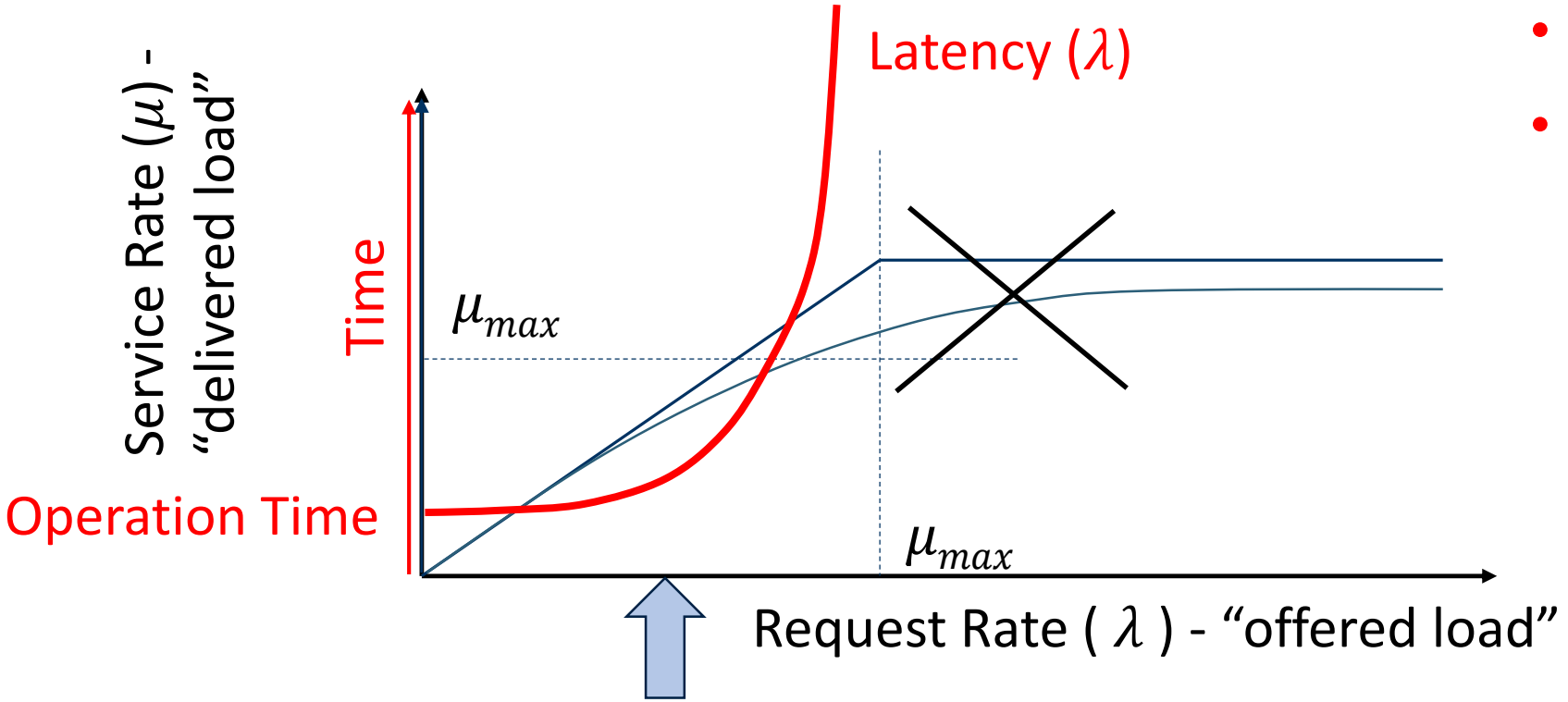
Key Results from Queuing Theory

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

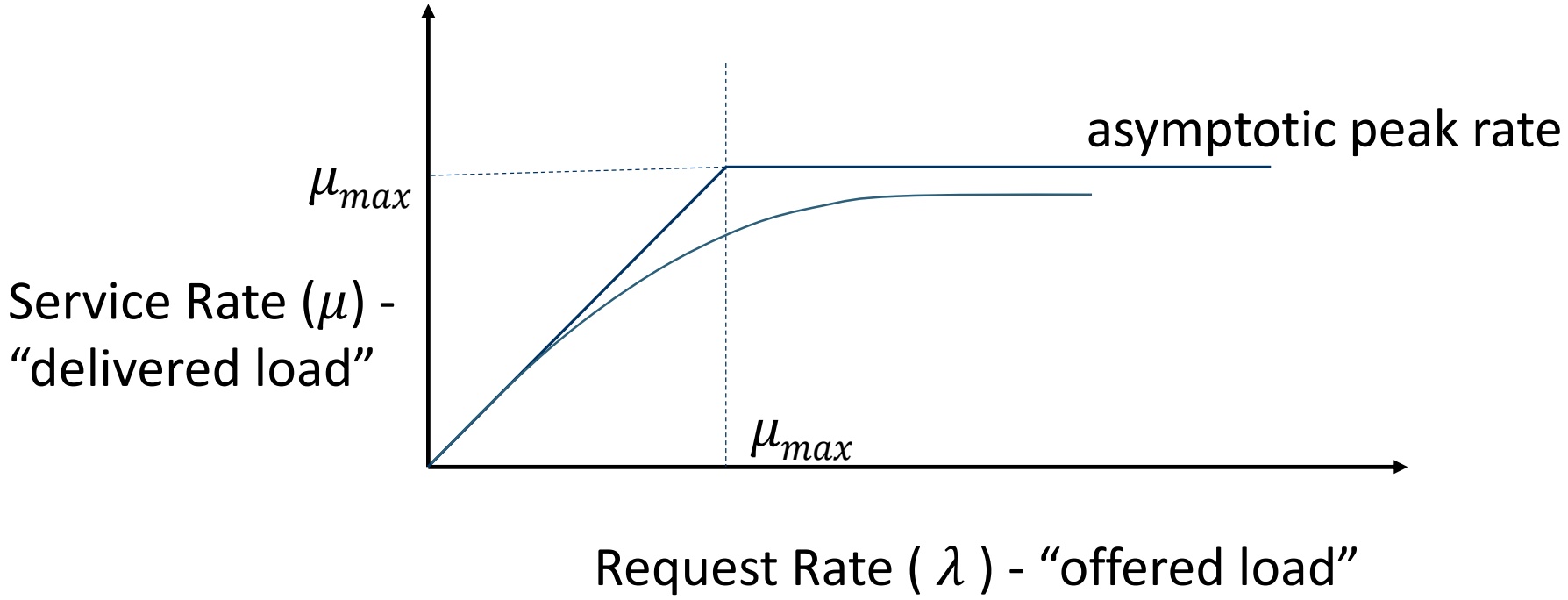
Break (If Time)

Rest of Today's Lecture

Using this system model, we will:

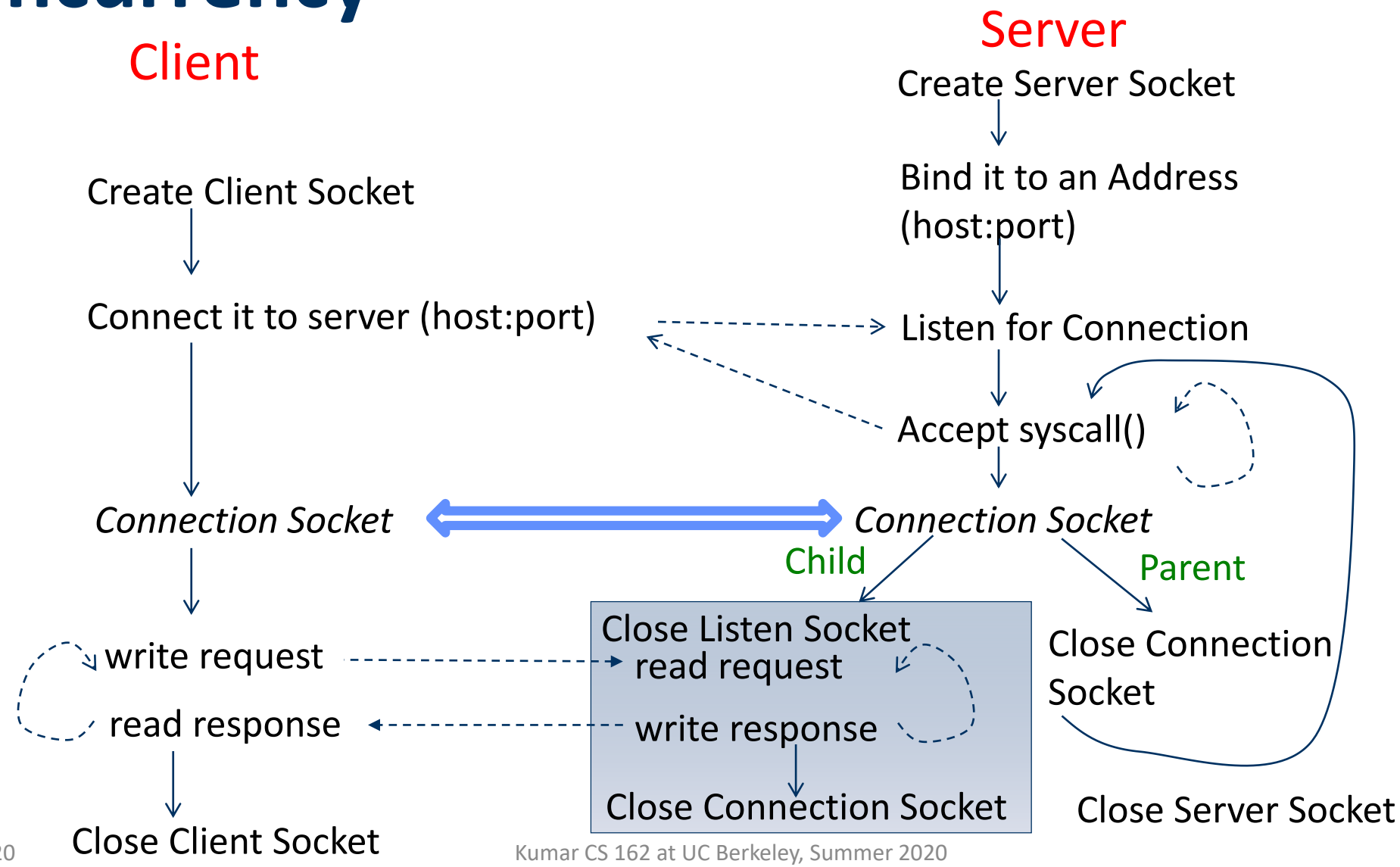
- Explore latency in more depth
- **Discuss how to build systems that perform well under load**

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

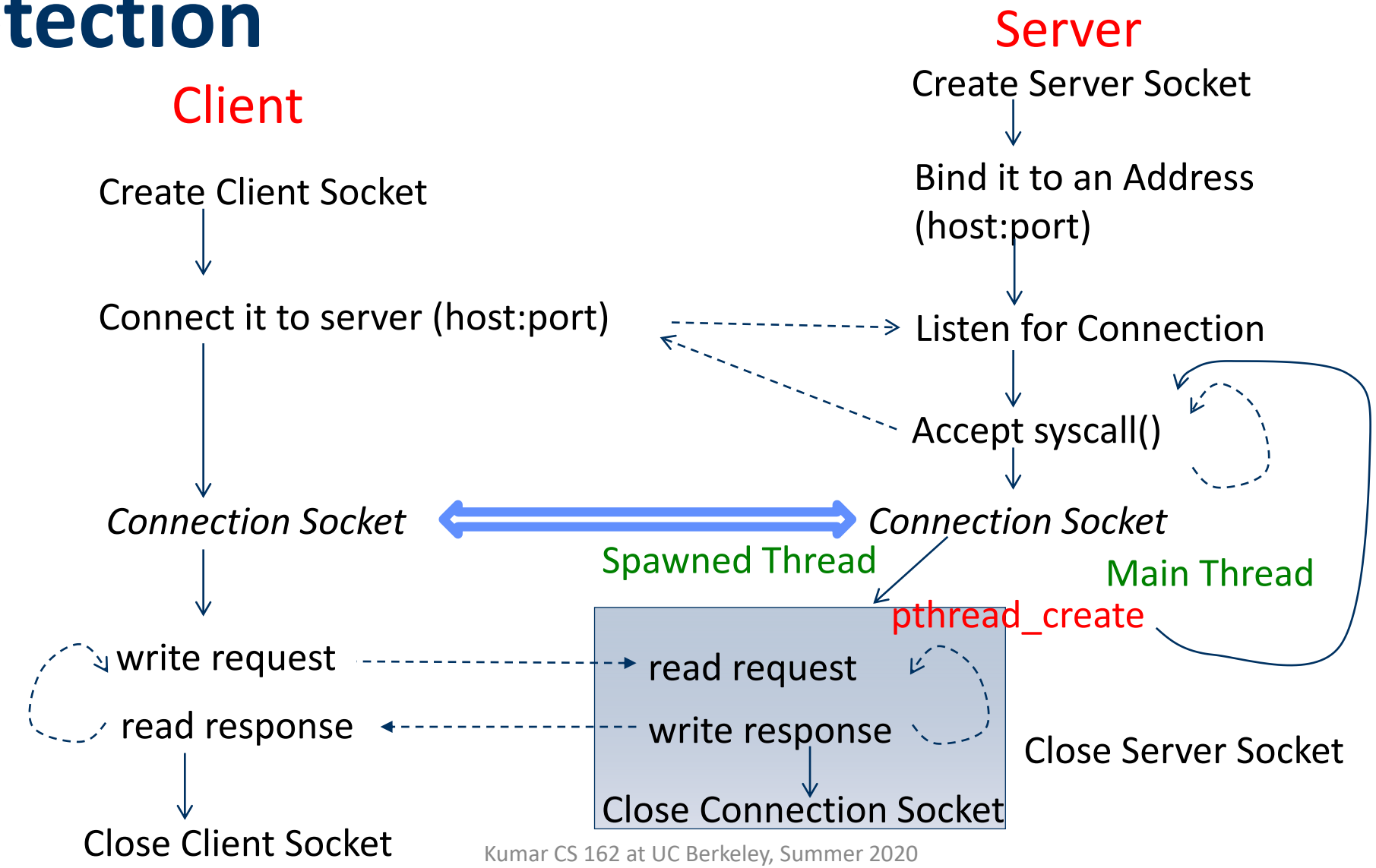
Recall: Sockets with Protection and Concurrency



Recall: Server Protocol (v3)

```
// Socket setup code elided…
while (1) {
  // Accept a new client connection, obtaining a new socket
  int conn_socket = accept(server_socket, NULL, NULL);
  pid_t pid = fork();
  if (pid == 0) {
    close(server_socket);
    serve_client(conn_socket);
    close(conn_socket);
    exit(0);
  } else {
    close(conn_socket);
    //wait(NULL);
  }
}
close(server_socket);
```

Recall: Sockets with Concurrency, without Protection



Non-Well-Conditioned Systems

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

- Figure from SEDA Section 2 reading (Welsh 2001)

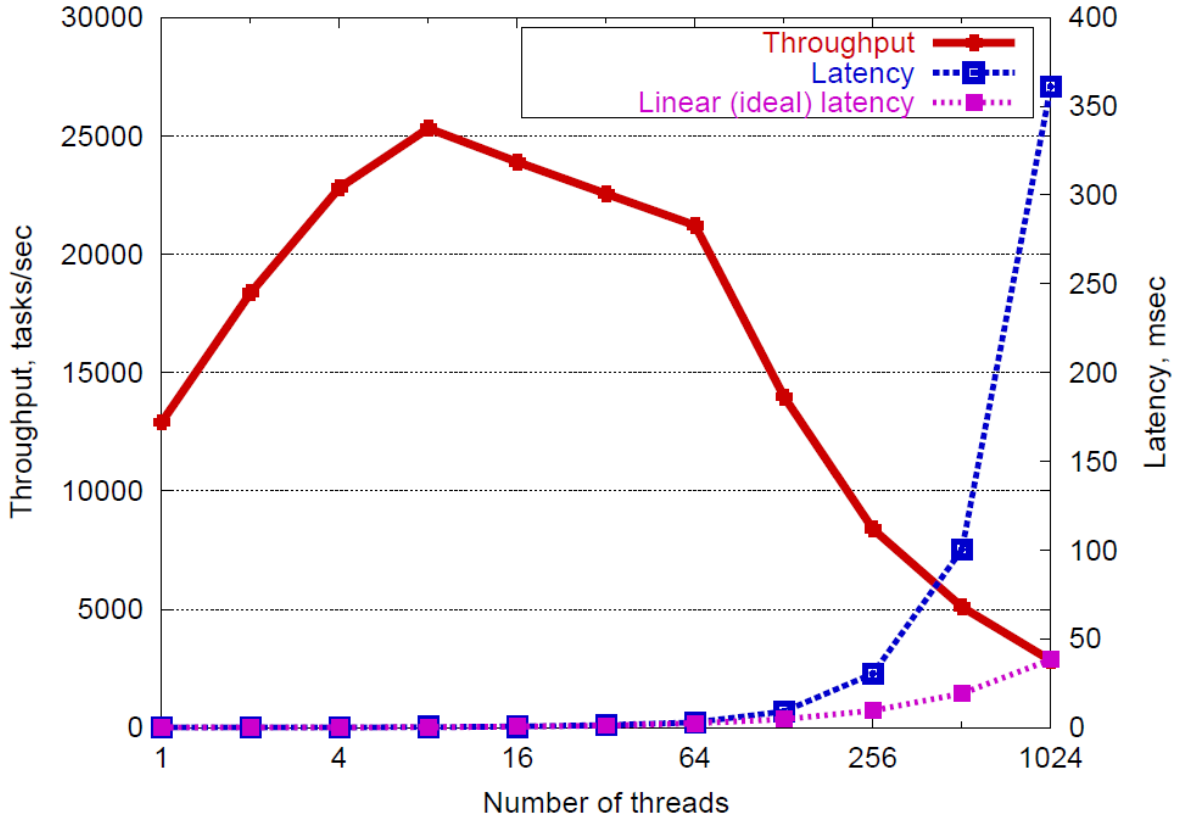


Figure 2: **Threaded server throughput degradation:** *This benchmark measures a simple threaded server which creates a single thread for each task in the*

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?

Building Well-Conditioned Systems

1. Thread Pools
2. User-Mode Threads
3. Event-Driven Execution

We'll discuss these next time…