CS 162: Operating Systems and Systems Programming

Lecture 10: Understanding System Performance & Fair Scheduling

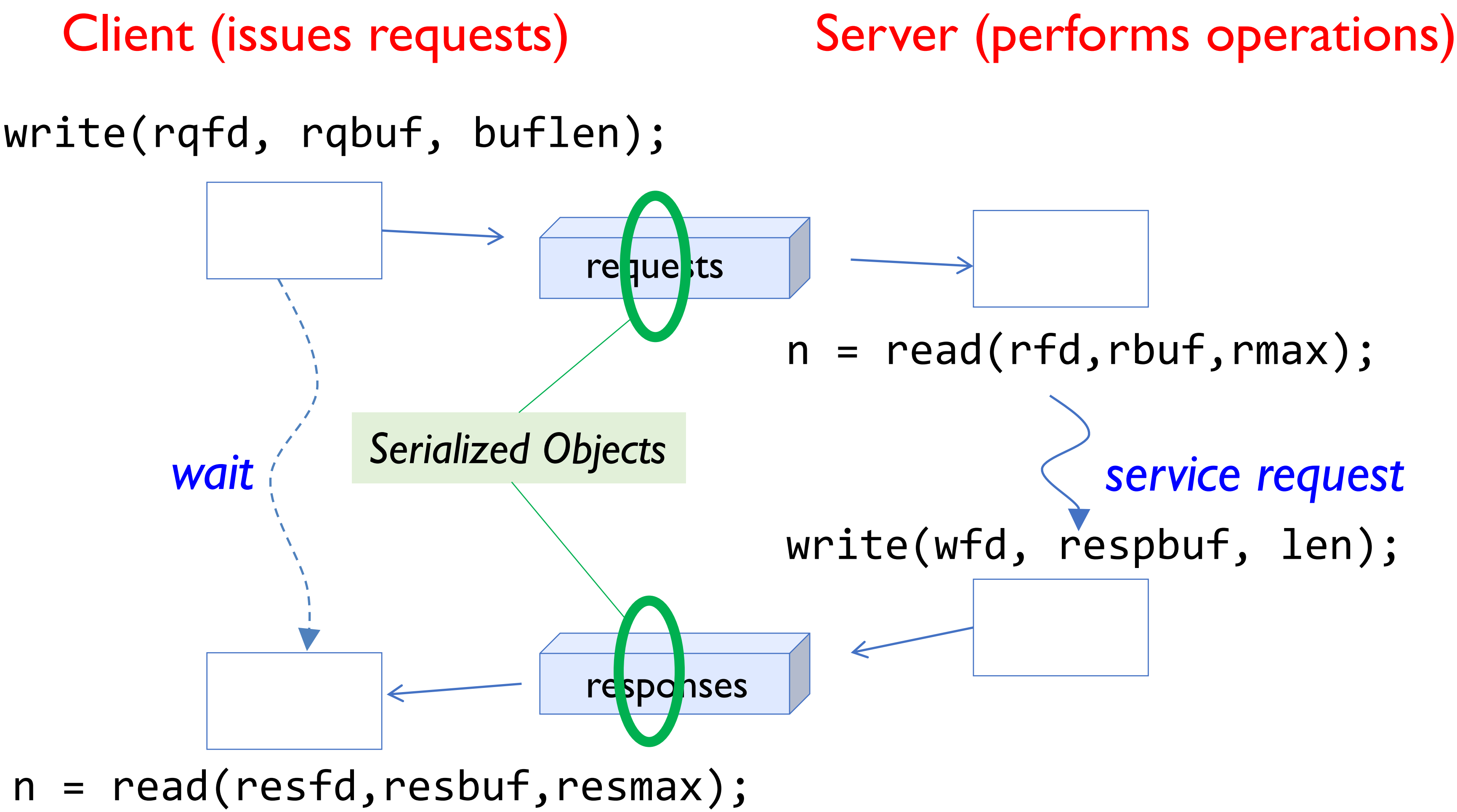
October 1, 2019
Instructor: David E. Culler
https://cs162.eecs.berkeley.edu

Read: 3easy ch 9

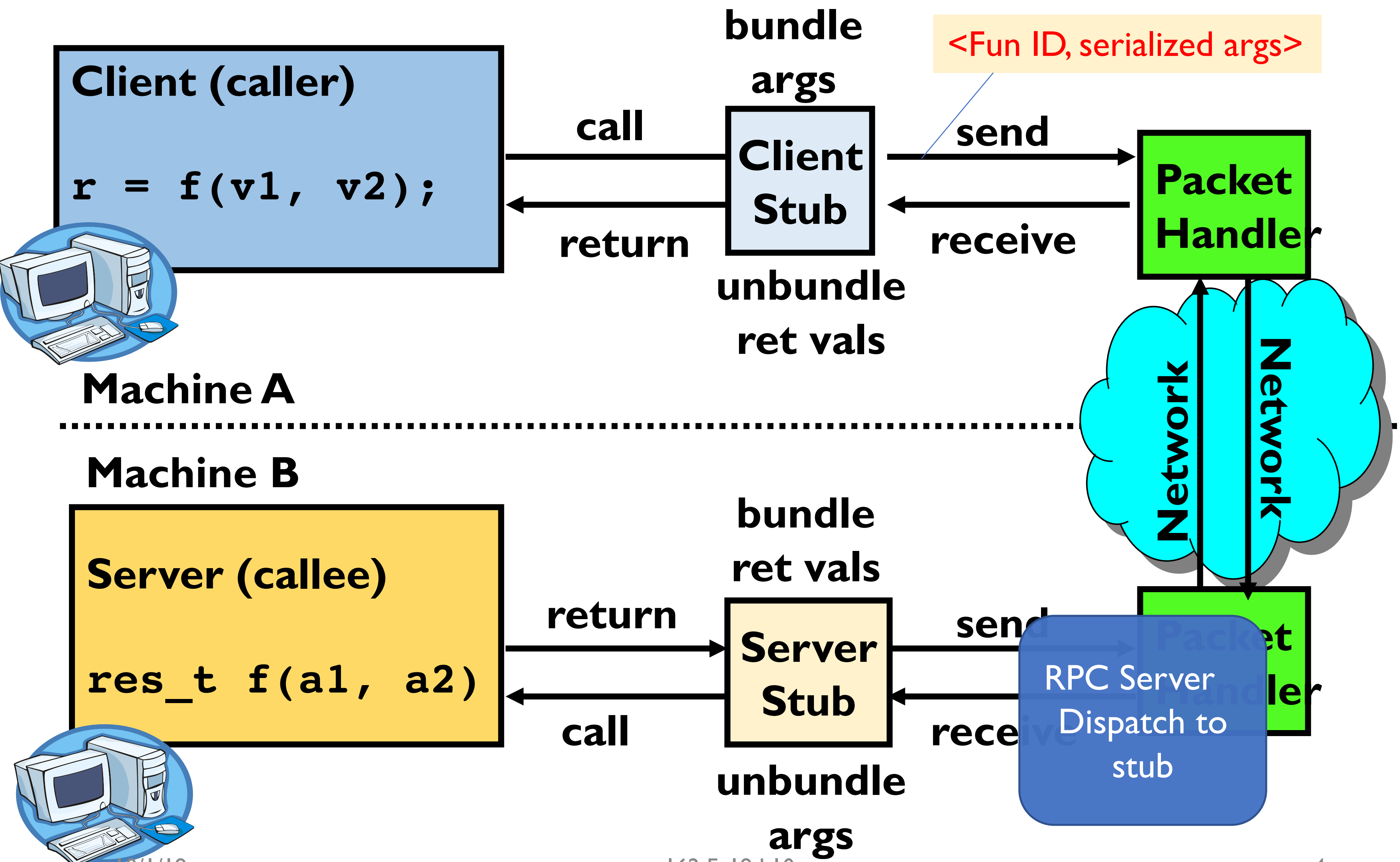
Recap

- Process: instance of an executing program
  - one or more threads, address space, systems resources (file descr., locks, …)
  - Isolated from other processes (its own address space)
  - Threads communicate primarily through shared variables, including locks, but could send messages
- Inter-Process Communication (IPC)
  - External means of communication information between processes, i.e., across distinct address spaces
    - File write/read, Socket write/read, message channels
    - Enables "inter-machine" communication among processes
- Serialization
  - Conversion of an "in-memory" data structure into an external canonical form that can be recovered by anpther process
    - Independent of binary representation in the machine
- Remote Procedure Call (RPC)
  - Methodology used to make interactions across processes familiar and convenient
  - Involves serialization and IPC
  - Also dispatch to the proper remote method by the RPC server
  - Stubs provide the glue on both side

Recall: Request/Response Protocol



RPC Information Flow



Recall: RPC Six steps

1. The client calls the client stub. The call is a local procedure call, with parameters pushed on to the stack in the normal way.
2. The client stub packs the parameters into a message and makes a system call to send the message. Packing the parameters is called marshaling.
3. The client's local operating system sends the message from the client machine to the server machine.
4. The local operating system on the server machine passes the incoming packets to the RPC server which *dispatches* the call to the server stub.
5. The server stub unpacks the parameters from the message. Unpacking the parameters is called unmarshaling.
6. Finally, the server stub ***calls*** the server procedure. The reply traces the same steps in the reverse direction

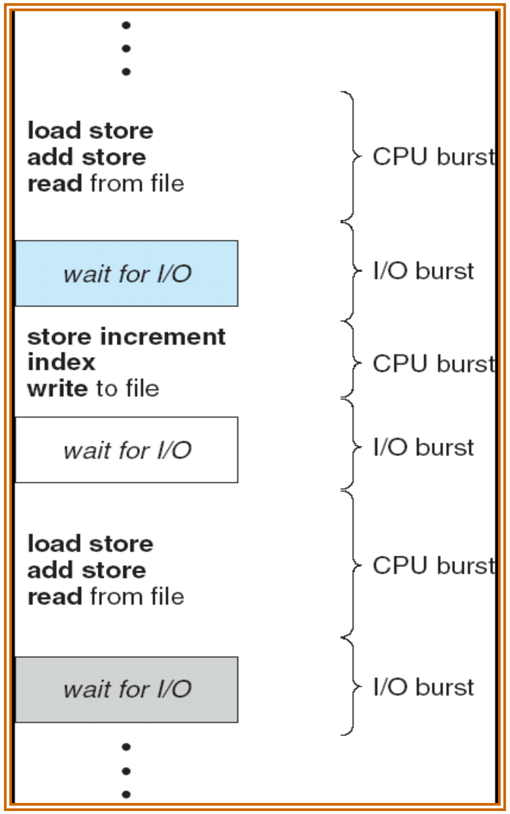
Systems Design:

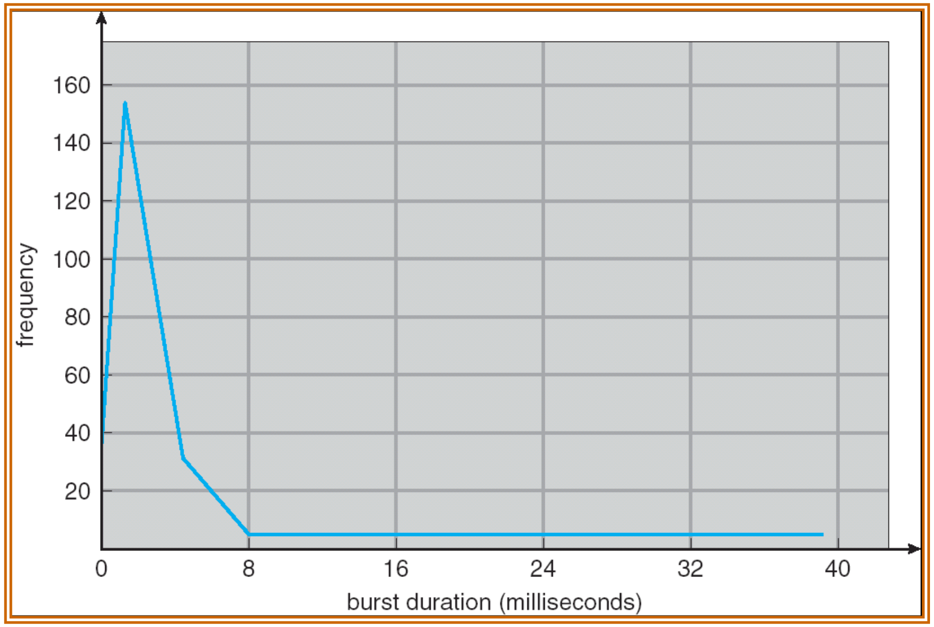
- Translate important high-level goals down to simple, robust mechanisms – and implement them
- Example: maintain "performance" in the presence of diverse, perhaps increasing demand
- Schedulers are a critical algorithmic component
  - Boil down to specific operations on certain (fast) data structures

Recall: Evaluating Schedulers

- **Response Time** (ideally *low*)
  - What user sees: from keypress to character on screen
- **Throughput** (ideally *high*)
  - Total operations per second
  - Problem: Overhead (e.g. from context switching)
- **Fairness**
  - Conflicts with best avg. throughput/resp. time

CPU & I/O Bursts





Support interactive programs: prefer I/O-bound tasks

Basics of System Performance

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

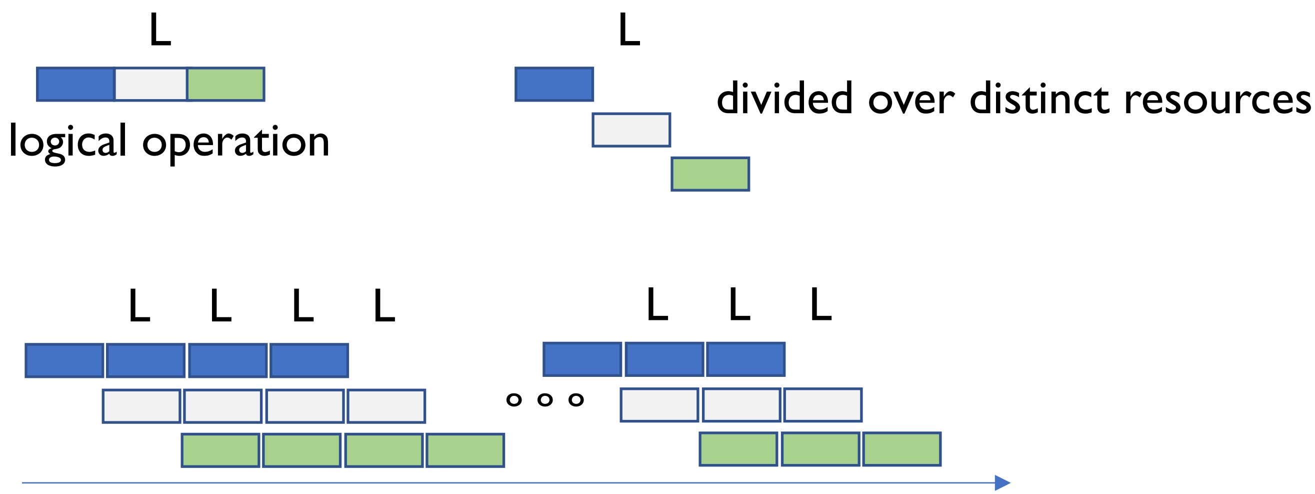
Times (s) and Rates (ops/s)

- ***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

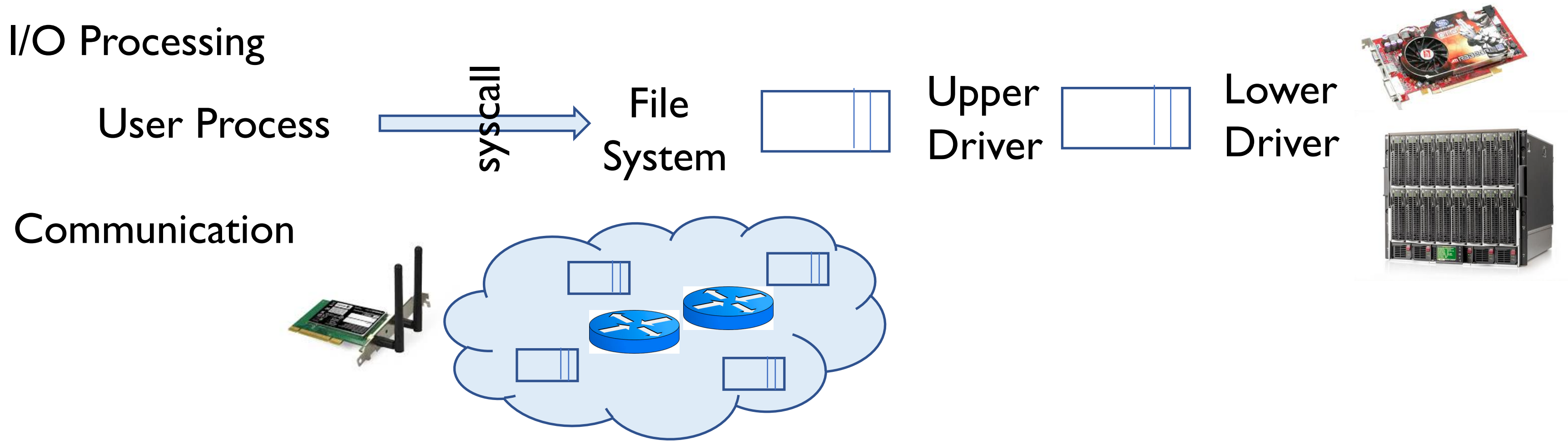
- Single sequential "server" that can deliver a task in time L operates at rate $\leq \frac{1}{L}$ (on average, in steady state, …).
  - L = 10 ms => B = 100 ops/sec
  - L = 2 years => B = 0.5 ops/year
- a processor, a disk drive, a Person, a TA, …

Single Pipelined Server



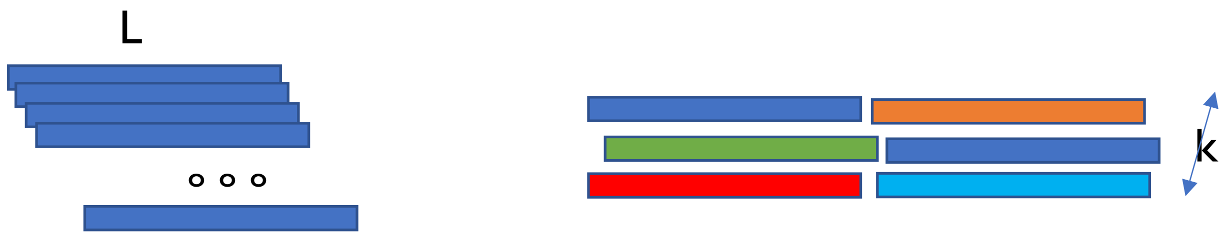
- Single pipelined server of k stages for tasks of length `L` (i.e., time `L/k` per stage) delivers at rate $\leq \frac{k}{L}$. (asymptote)
  - L = 10 ms, k = 4 => B = 400 ops/sec
  - L = 2 years, k = 2 => B = 1 ops/year
- In 61C you saw "synchronous" pipelines in processors
  - Systems present lots of asynchronous pipeline-like settings

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

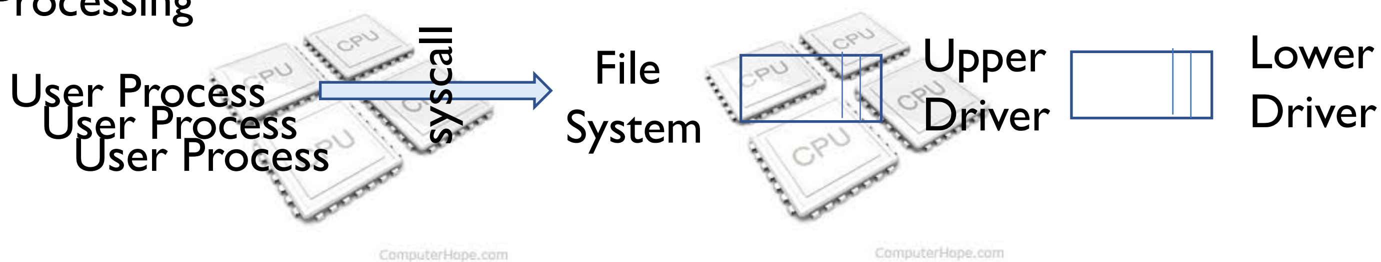
Multiple Servers

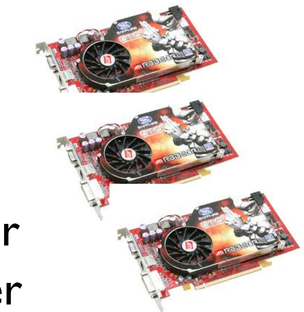


- k servers handling tasks of length `L` (i.e., time `L/k` per stage) delivers at rate $\leq \frac{k}{L}$.
  - L = 10 ms, k = 4 => B = 400 ops/sec
  - L = 2 years, k = 2 => B = 1 ops/year
- In 61C you saw multiple processors (cores)
  - Systems present lots of multiple parallel servers
  - Often with lots of queues

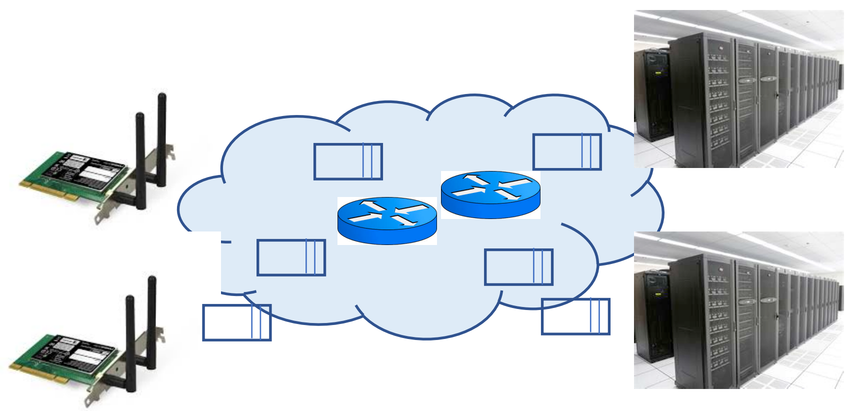
Example Systems "Parallelism"

I/O Processing





Communication



Parallel Computation, Databases, …

A Simple Systems Performance Model

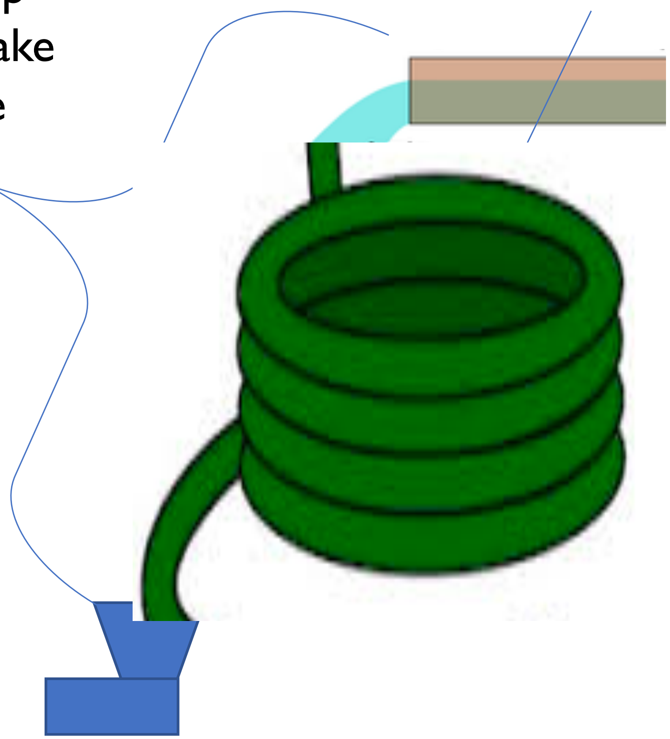
Latency (L) time per op

- How long does it take to flow through the system

"Service Time"

Bandwidth (B): Rate, Op/s

e.g., flow: gal per min



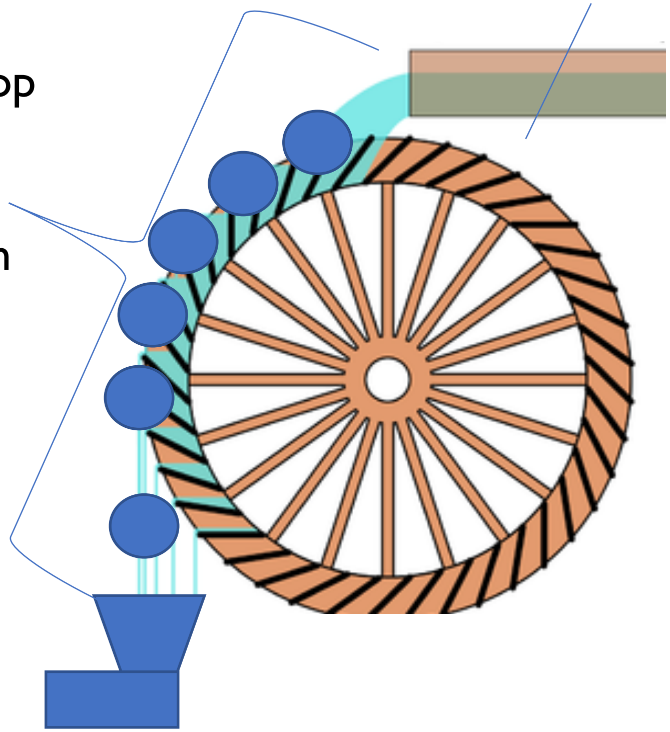
If $B = 2\frac{gal}{s}$ and L $= 3s$

How much water is "in the system?"

A Simple Systems Performance Model

Latency (L) time per op
- How long does an operation to flow through the system

Service Time



Bandwidth (B): Rate, Op/s

If $B = 2\frac{ops}{s}$ and L $= 3s$
How many ops are "in the system?"

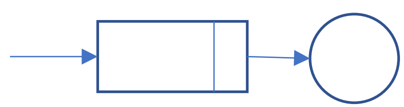
Little's Law

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

$$n = L\,B$$

- In networks, the bandwidth-delay product
- Include the queues, the processing stages, parallelism, whatever

"in other words"



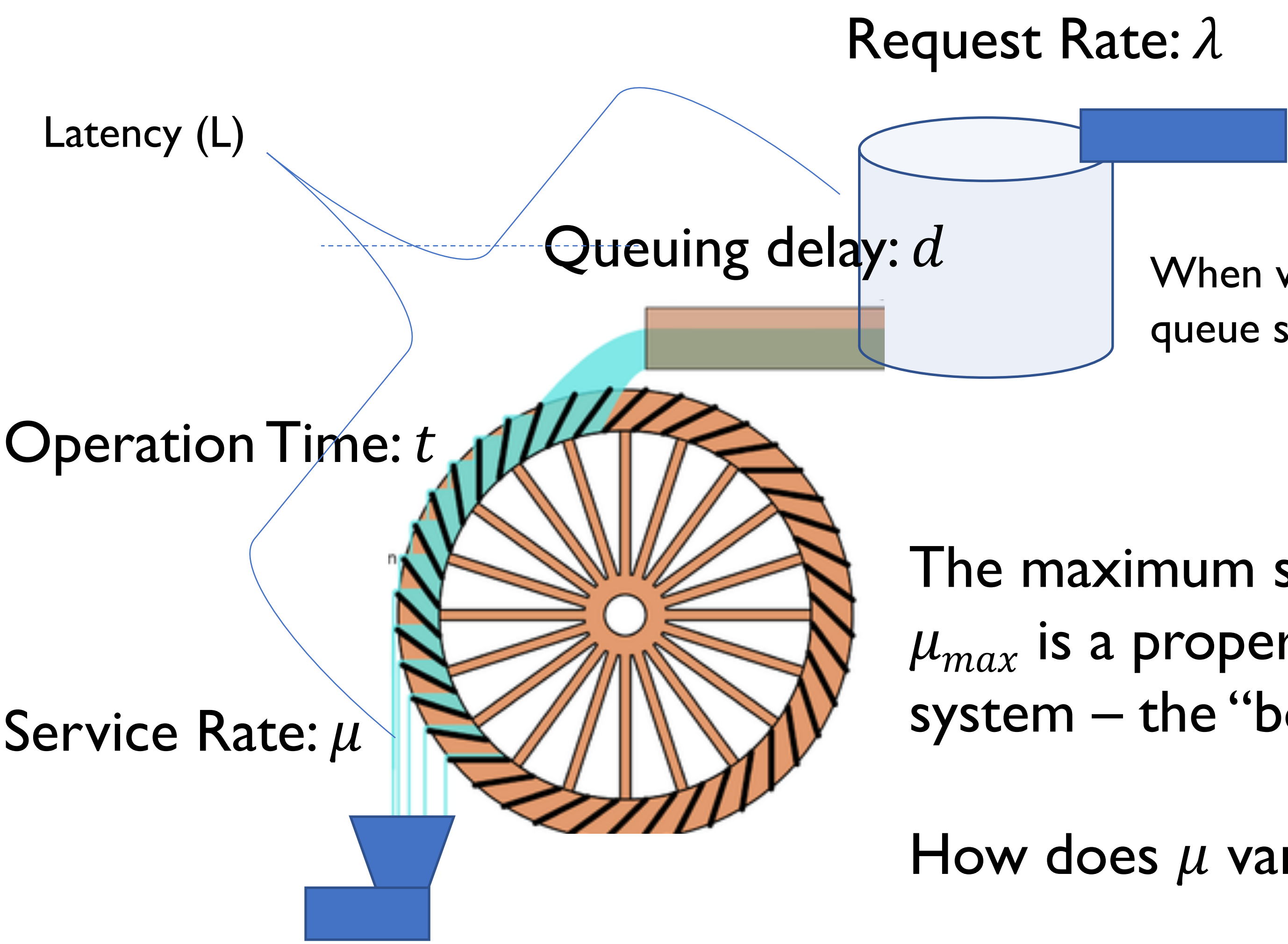
Average Arrival Rate

$$\text{“}L = \lambda\, W\text{”}$$

Average length of the queue

Average time "waiting"

A Simple Systems Performance Model

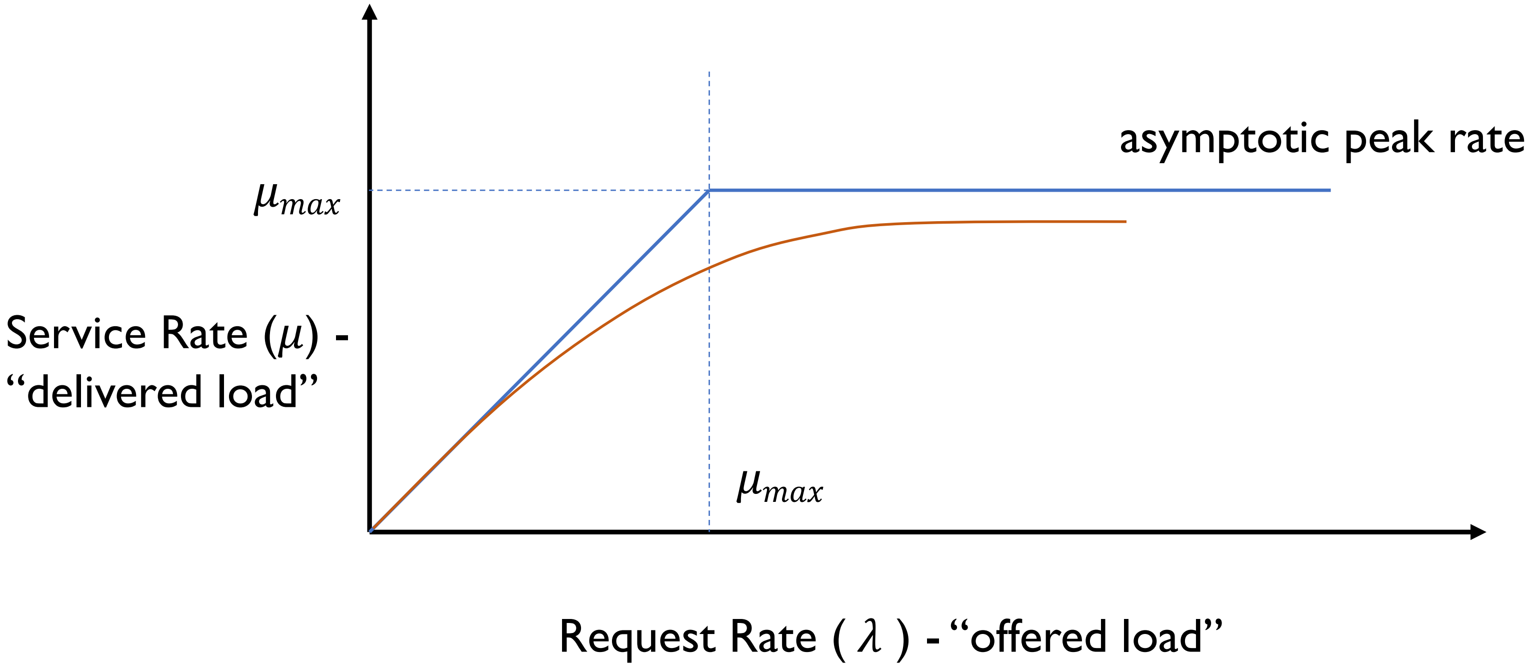


When will the queue start to fill?

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

How does μ vary with λ ?

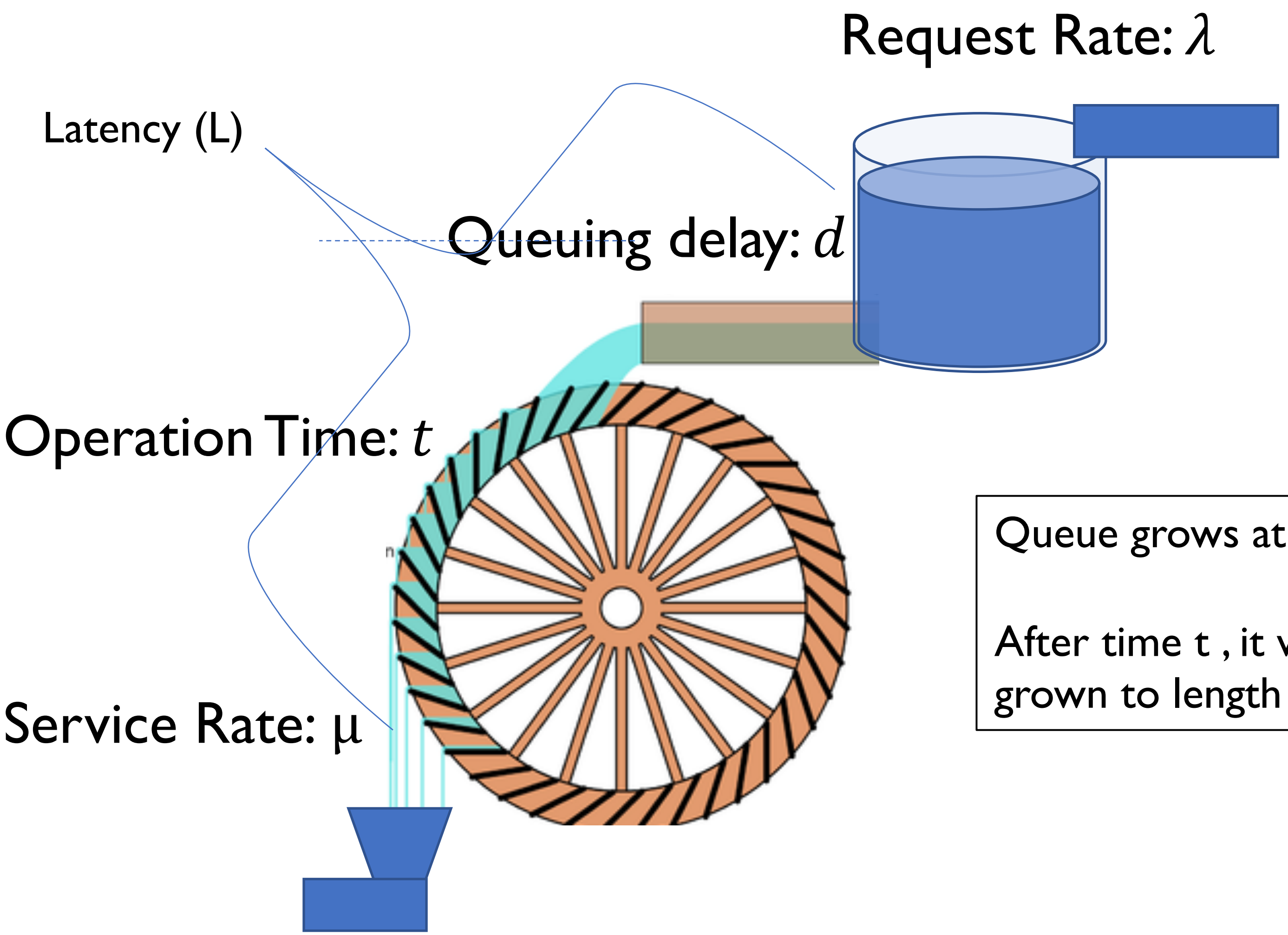
Idealized System Performance



Queuing

- What happens when request rate exceeds max service rate?
- Short bursts can be absorbed by the queue
  - If on average $\lambda < \mu$, it will drain eventually
- Prolonged $\lambda > \mu$ – queue will grow arbitrarily

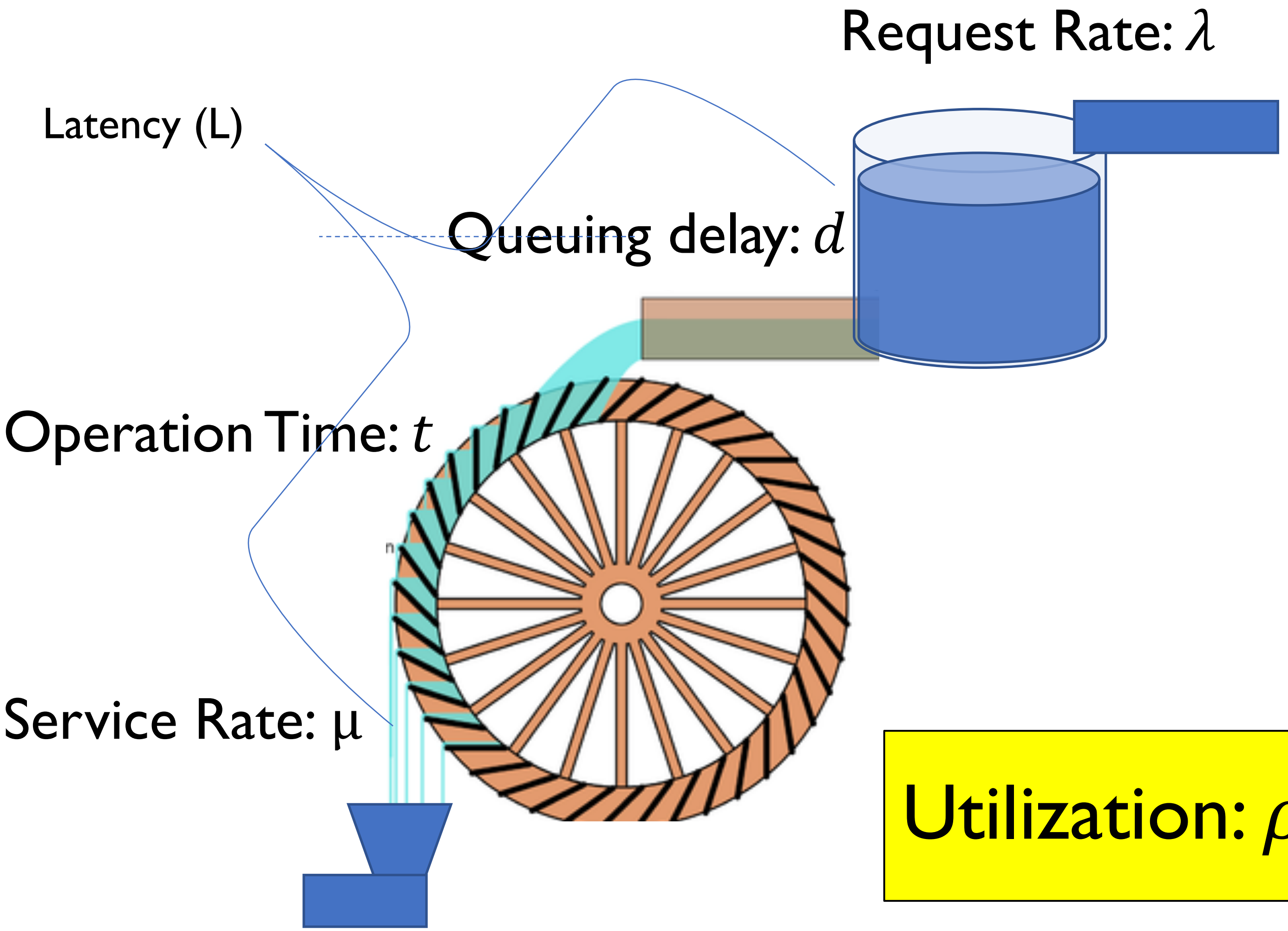
A Simple Systems Performance Model



Queue grows at rate $\mu - \lambda$

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

A Simple Systems Performance Model



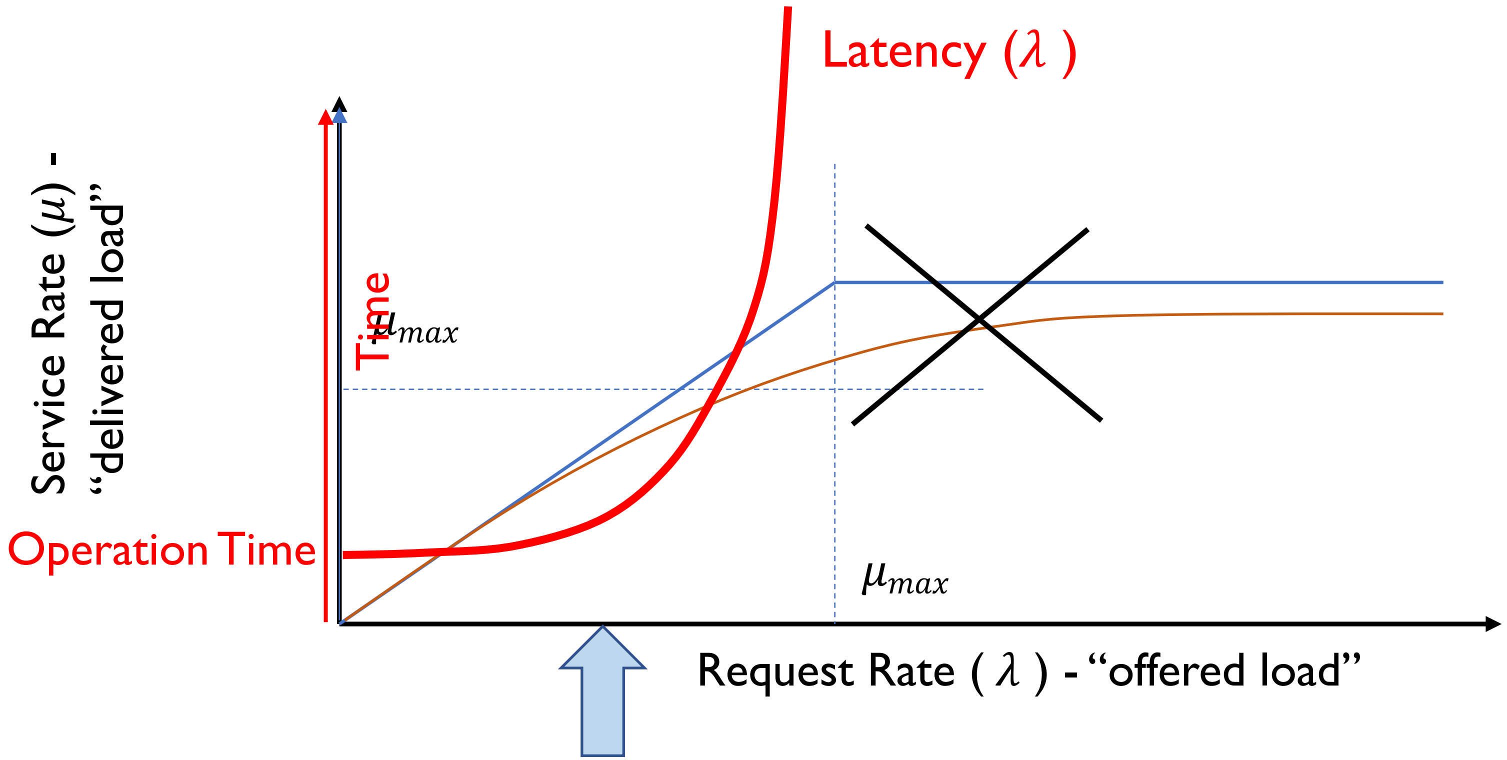
Utilization: $\rho = \frac{\lambda}{\mu_{max}}$

Length of the queue …

- Queuing Theory describes steady-state (stationary, not transient) behavior in the presence of stochastic variations – primarily the request rate distribution, i.e., bursts and lulls.

- Ave Number of Operations in the system = $\frac{\rho}{1-\rho}$

Idealized System Performance



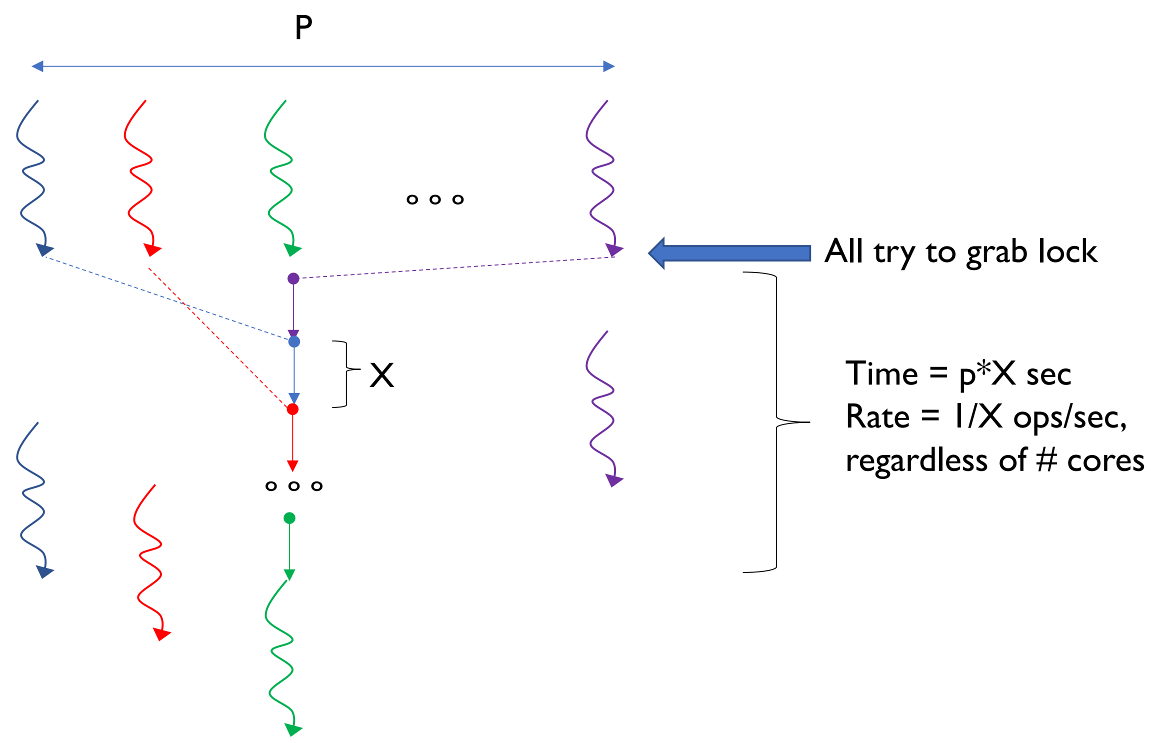
"Half-Power Point" : load at which system delivers half of peak performance

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

What's the bottleneck that determines μ_{max} ?

Time to service a highly contended lock?

Highly Contended Case – in a picture



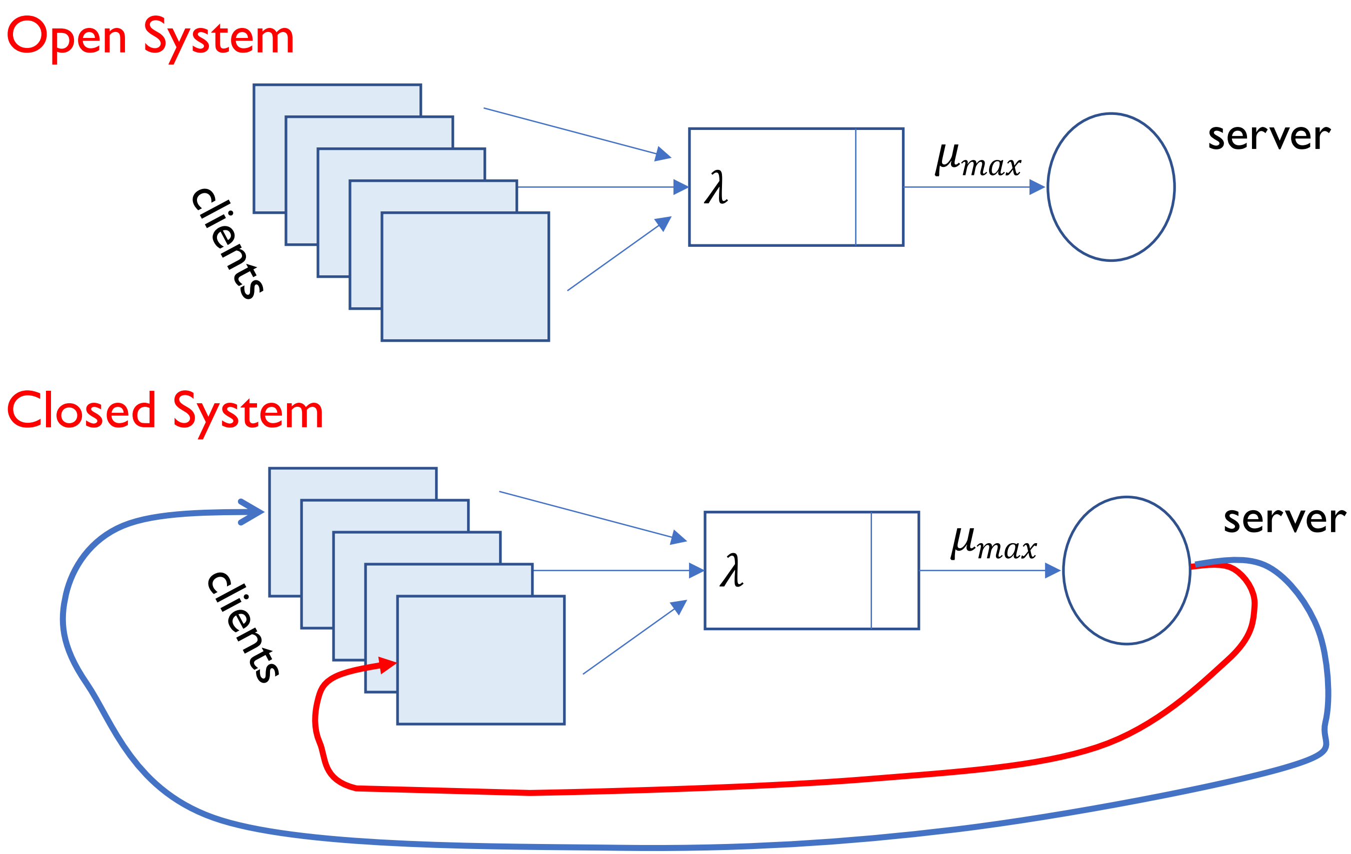
Potential Bottlenecks …

- Actual operation time servicing request – apply parallelism
- System calls – if the operating system is only able to process one or few calls at a time
- System call overhead to grab even uncontended locks
- Socket accept/bind/fork (thread create)
- Time for scheduler to select the next thread to run

- Time to write to a log file
- … <your idea / concern goes here >

Do real systems really hit a wall as utilization approaches 100% ?

- And how do we measure and test it ?

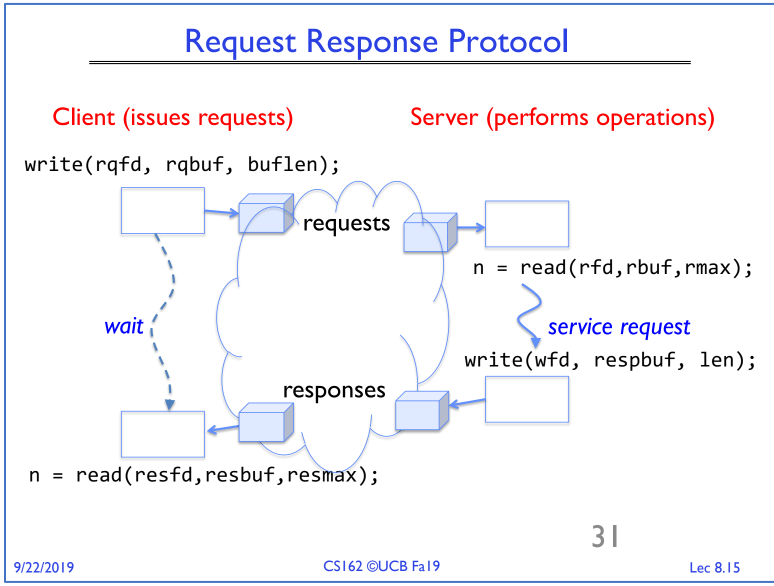


Closed System

- Clients generating the load depend on completion of previous requests
  - Request-response protocols
  - Humans in-the-loop waiting for results
- Model of client: {request, wait}+ repeat
  - Request rate determined by length of wait
- In closed system, wait time depends on response time (latency = operation time + queuing delay)
- As system saturates (utilization $\rightarrow 100\%$) delay increases, request rate is limited by service rate
  - Queueing smooths bursts, but does not grow unbounded due to rate mismatch

What causes systems to "close"?

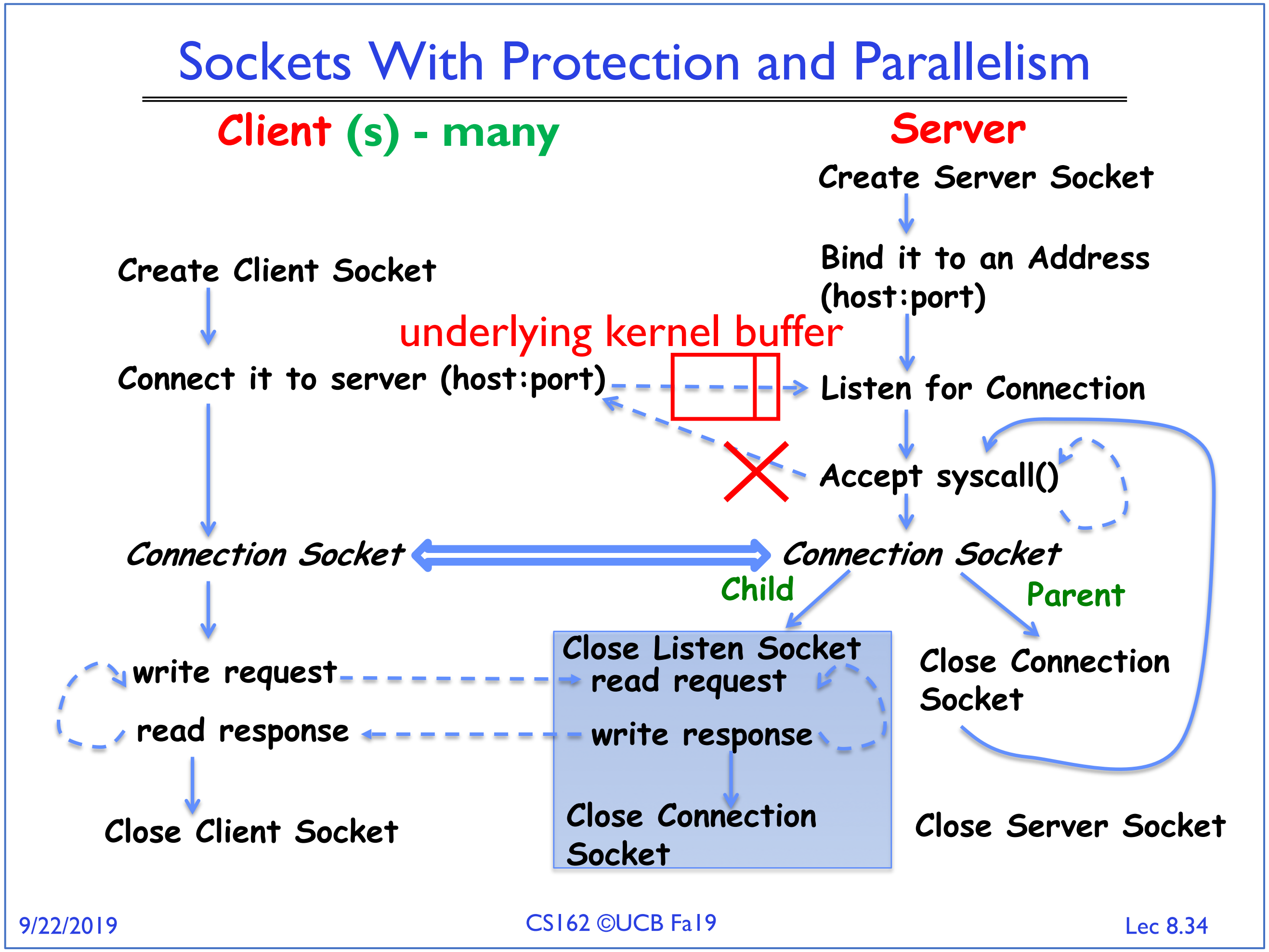
- Protocols are designed to have self-limited behavior
  - Request-response, bounded number of outstanding requests per client
- Underlying system induces "back pressure" even if higher level services and applications don't
  - Bounded size queues (not just because of memory size)
  - What happens when it fills up?



What do you do when a queue fills up?

- Within a machine?
  - Block the thread (or process) – slowing it down
  - Lot's of cheap ways to set things aside
  - Scheduling – take on only what you can make progress on
  - Trying to do too much causes thrashing
  - Design as network of bounded queues with finite servicing
- Across machines
  - Drop the request
  - Don't respond, keep the client hanging
  - But, all those millions are clients are independent !!! Doesn't that make it behave "open" ?

Limiting excessive load



- By not accepting connections, server slows ALL clients
  - But some still make progress

Break

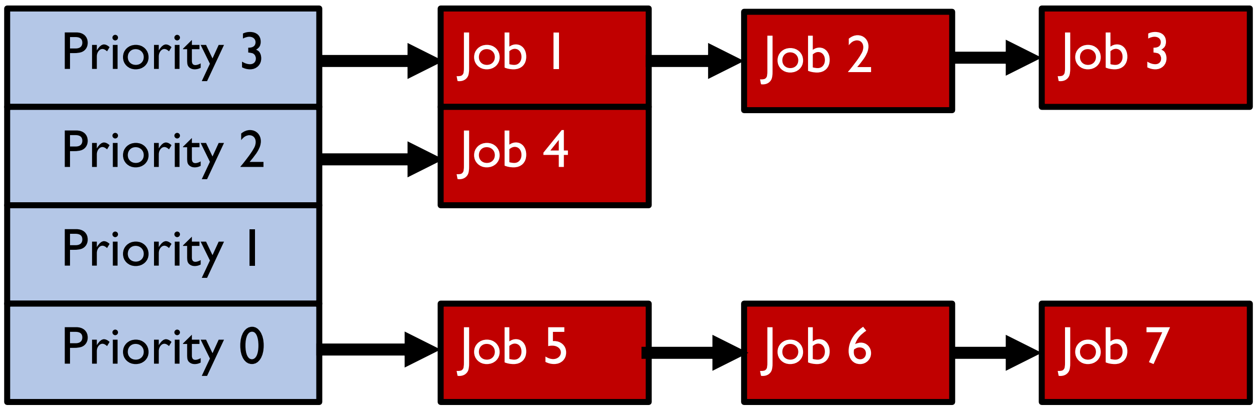
Recall: First-Come First-Served

- Just run processes in order of arrival
- **Convoy Effect: Short processes stuck behind long processes**

Recall: Round Robin

- Give out *small* units of CPU time ("time quantum")
- Preempt a thread when its quantum expires
- Cycle through ready threads

Recall: Priority Scheduling



- Something gives jobs (processes) priority
  - User sets it explicitly
  - System manipulates priorities in pursuit of some policy

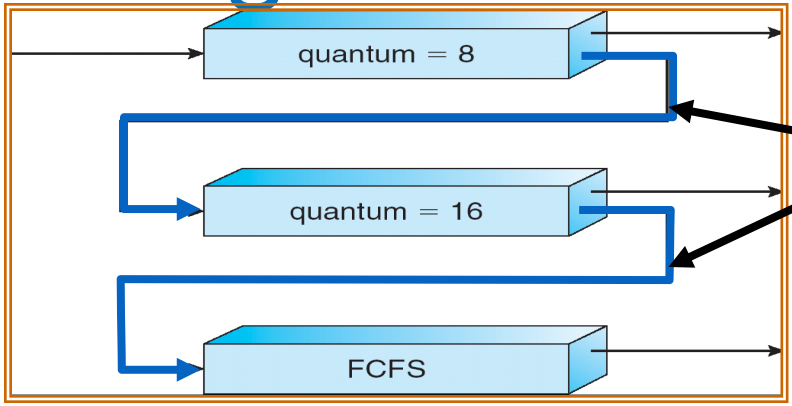
- Always run the **ready** thread with *highest priority*

Recall: Shortest Job First & Shortest Remaining Time First

- **Provably Optimal** with respect to *Response Time*

- Key Idea: remove convoy effect
  - Short jobs always stay ahead of long ones

Recall: Multi-Level Feedback Scheduling



Long-Running Compute Tasks Demoted to Low Priority

- Observe process behavior to approximate SRTF
- Starvation still possible without a workaround
- Basis for real OS schedulers, e.g. Linux O(1)

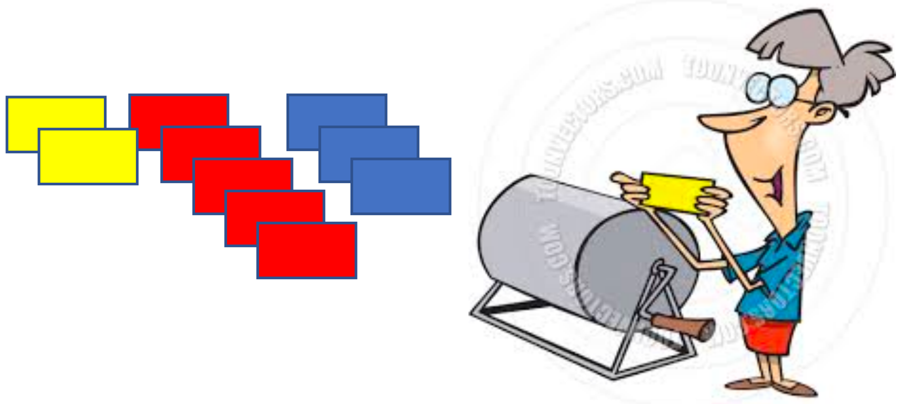
Goals, Ends and Means

- Priorities were a means, not an end
- Our end goal was to serve a mix of CPU-bound, I/O bound, and Interactive jobs effectively on common hardware
  - Give the I/O bound ones enough CPU to issue their next file operation and wait (on those slow discs)
  - Give the interactive ones enough CPU to respond to an input and wait (on those slow humans)
  - Let the CPU bound ones grind away without too much disturbance

Changing landscape of scheduling

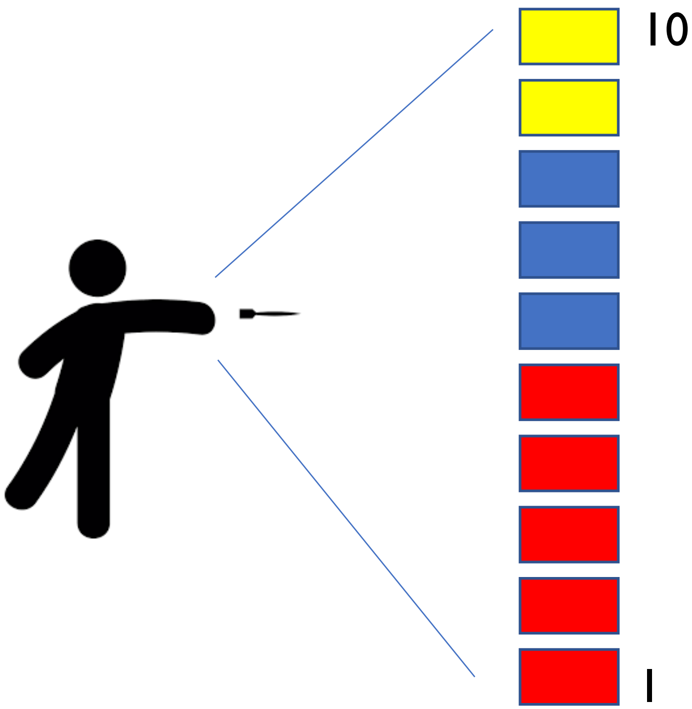
- Priority-based scheduling rooted in "time-sharing"
  - Allocating precious, limited resources across a diverse workload
    - CPU bound, vs interactive, vs I/O bound
- 80's brought about personal computers, workstations, and servers on networks
  - Different machines of different types for different purposes
  - Shift to fairness and avoiding extremes (starvation)
- 90's emergence of the web, rise of internet-based services, the data-center-is-the-computer
  - Server consolidation, massive clustered services, huge flashcrowds
  - It's about predictability, 95^{th} percentile performance guarantees

Proportion Share Scheduling: lottery



- Given a set of jobs (the mix), provide each with a proportional share of a resource
  - e.g., 50% of the CPU for Job A, 30% for **Job B**, and 20% for Job C
- Idea: Give out tickets according to the portion each should receive,
- Every quanta (tick) draw one at random, schedule that job (thread) to run

Simpler Mechanism



- $N_{ticket} = \sum N_i$
- Pick a number *d* *in* $1 .. N_{ticket}$ as the random "dart"
- Jobs record their N_i of allocated tickets
- Order them by N_i
- Select the first j such that $\sum N_i$ up to j exceeds *d*.

Various additional measures to allow a job to subdivide its tickets to divide up its share.

UnFairness

- E.g., Given two jobs A and B of same run time (# Qs) that are each supposed to receive 50%,

  U = finish time of first / finish time of last

- As a function of run time

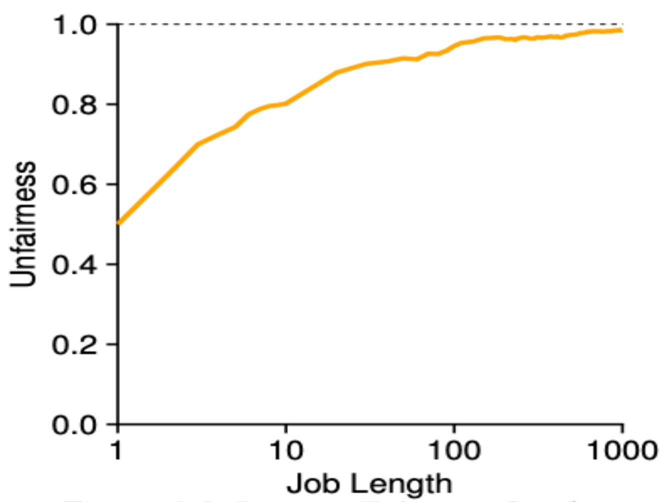


Figure 9.2: **Lottery Fairness Study**

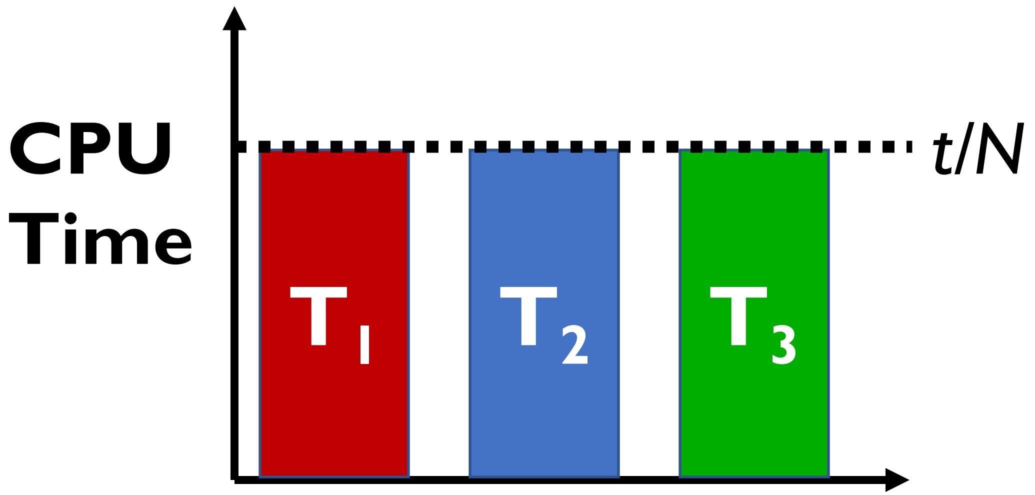
Stride Scheduling

- Achieve proportional share scheduling without resorting to randomness, and overcome the "law of small numbers" problem.
- "Stride" of each job is $\frac{big\#W}{N_i}$
  - The larger your share of tickets, the smaller your stride
  - Ex: W = 10,000, A=100 tickets, B=50, C=250
  - A stride: 100, B: 200, C: 40
- Each job as a "pass" counter
- Scheduler: pick job with lowest *pass*, runs it, add its *stride* to its *pass*
- Low-stride jobs (lots of tickets) run more often
  - Job with twice the tickets gets to run twice as often
- Some messiness of counter wrap-around, new jobs, …

Linux Completely Fair Scheduler

- Goal: Each process gets an equal share of CPU
- *N* threads "simultaneously" execute on 1/*N*th of CPU

At *any* time *t* we would observe:

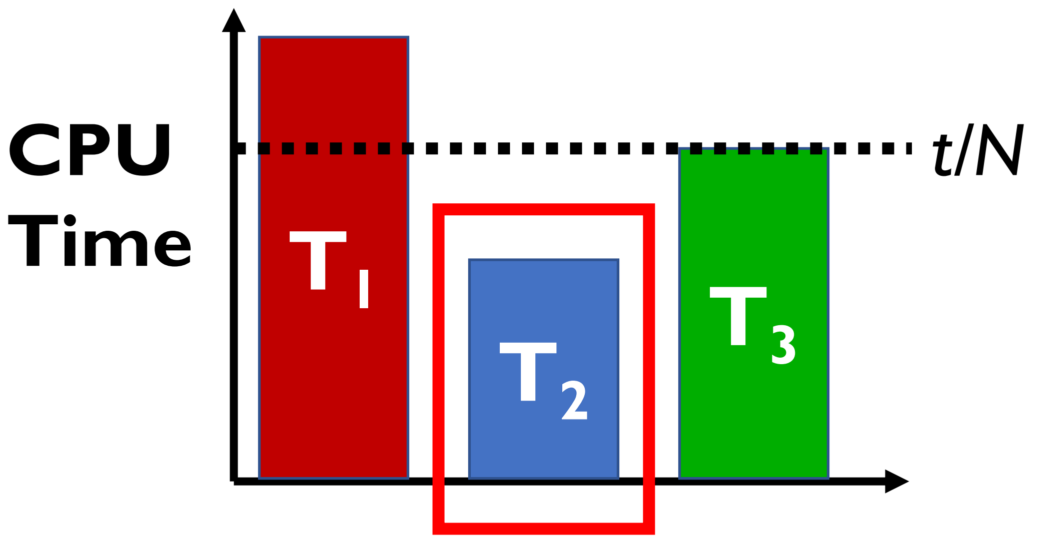


Linux Completely Fair Scheduler

- Can't do this with real hardware
  - Still need to give out full CPU in time slices
- Instead: track CPU time given to a thread so far

Scheduling Decision:

- "Repair" illusion of complete fairness
- Choose thread with minimum CPU time



Linux CFS: Responsiveness

- Goal: Preserve response time
- Constraint 1: *Target Latency*
  - Period of time over which every process gets service
  - Quanta = Target_Latency / n
- Target Latency: 20ms, 4 Processes
  - Each process gets 5ms time slice
- Target Latency: 20 ms, 200 Processes
  - Each process gets 0.1ms time slice (!!!)
  - Recall Round-Robin: large context switching overhead if slice gets to small

Linux CFS: Overhead

- Goal: throughput
  - avoid excessive overhead,
- Constraint 2: *Minimum Granularity*
  - Minimum length of any time slice

- Target Latency 20ms, Minimum Granularity 1ms, 200 processes
  - Each process gets 1ms time slice

Aside: Priority in Unix – Being Nice

- The industrial operating systems of the 60s and 70's provided priority to enforced desired usage policies.
- When it was being developed at Berkeley, instead it provided ways to "be nice".
- `nice` values range from -20 to 19
  - Negative values are "not nice"
  - If you wanted to let you friends get more time, you would nice up you job
- Schedule puts higher nice (lower priority) to sleep more …

Linux CFS: beyond equal share

- What if we want to give more to some and less to others (proportional share) ?
- Reuse `nice` value to reflect share, rather than priority
- Key Idea: Assign a weight w_i to each process *i*

- Basic equal share: $Q = \text{Target Latency} * \frac{1}{N}$
- Weighted Share:

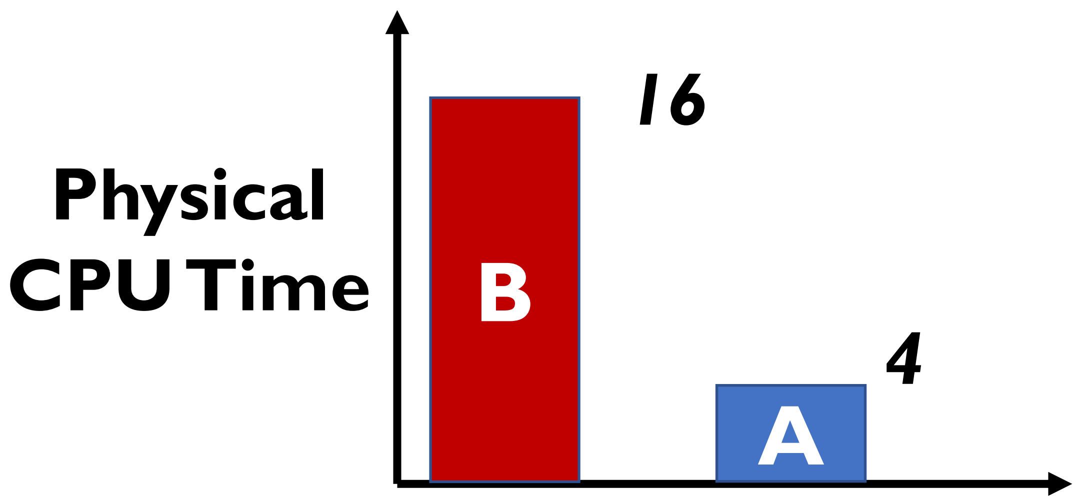
$$Q_i = (w_i / \sum_p w_p) * \text{Target Latency}$$

Linux CFS: weighted share

- Target Latency = 20ms,
- Minimum Granularity = 1ms
- Two CPU-Bound Threads
  - Thread *A* has weight 1
  - Thread *B* has weight 4
- Time slice for *A*? 4 ms
- Time slice for *B*? 16 ms

Linux CFS: equal weighted shares

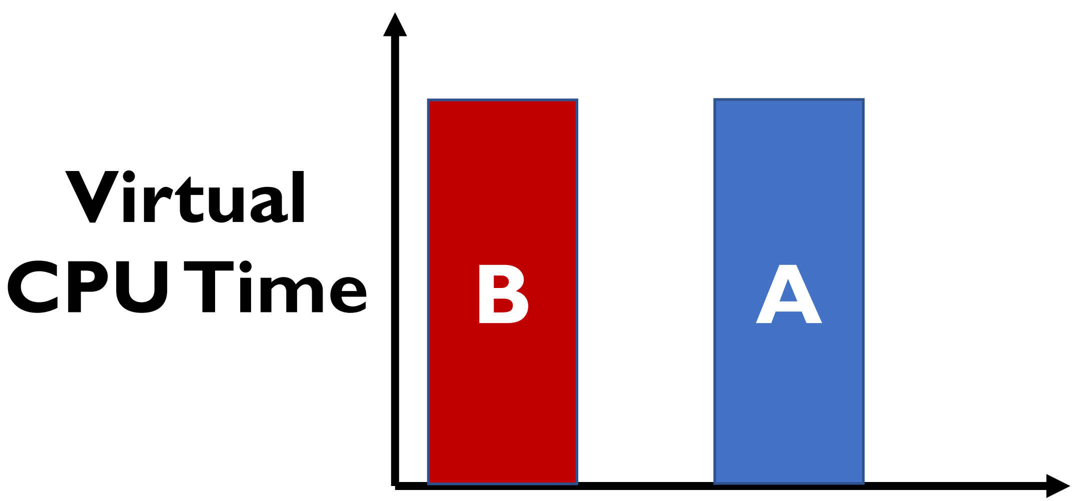
- Track a thread's *virtual* runtime rather than its true physical runtime
- Higher weight: Virtual runtime increases more slowly
- Lower weight: Virtual runtime increases more quickly



Linux CFS: equal weighted shares

- Track a thread's *virtual* runtime rather than its true physical runtime
- Higher weight: Virtual runtime increases more slowly
- Lower weight: Virtual runtime increases more quickly

Actually Used for Decisions



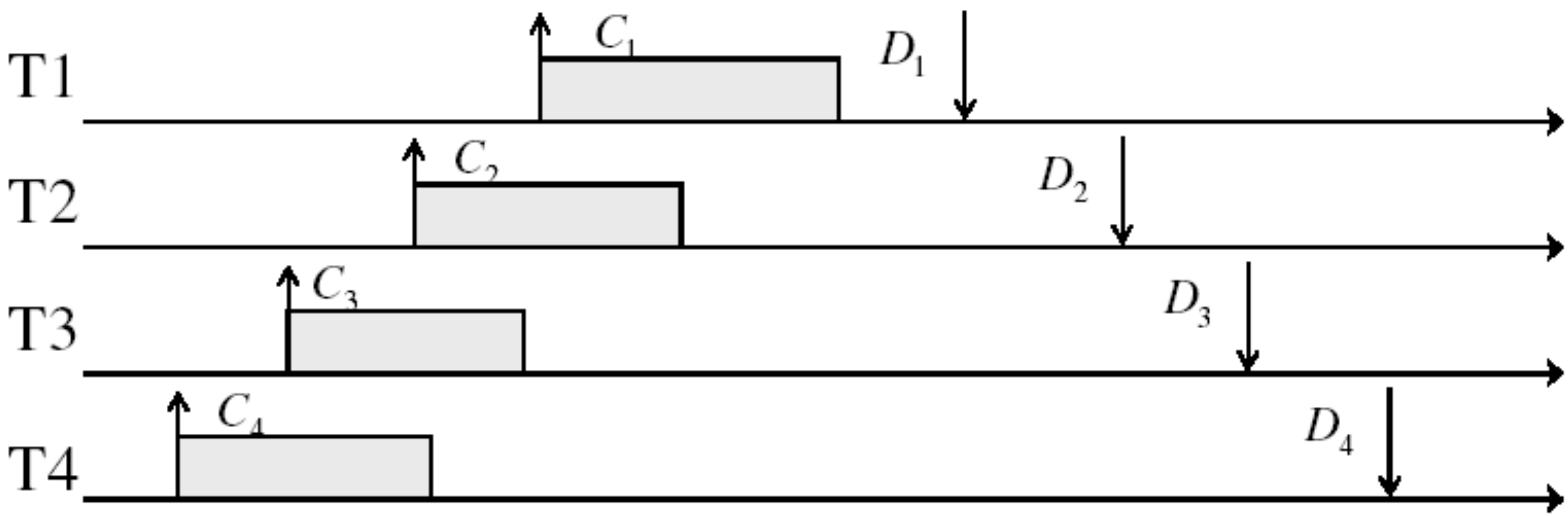
Real-Time Scheduling

- Goal: **Guaranteed Performance**
  - Meet **deadlines** even if it means being unfair or slow
  - Limit how bad the **worst case** is

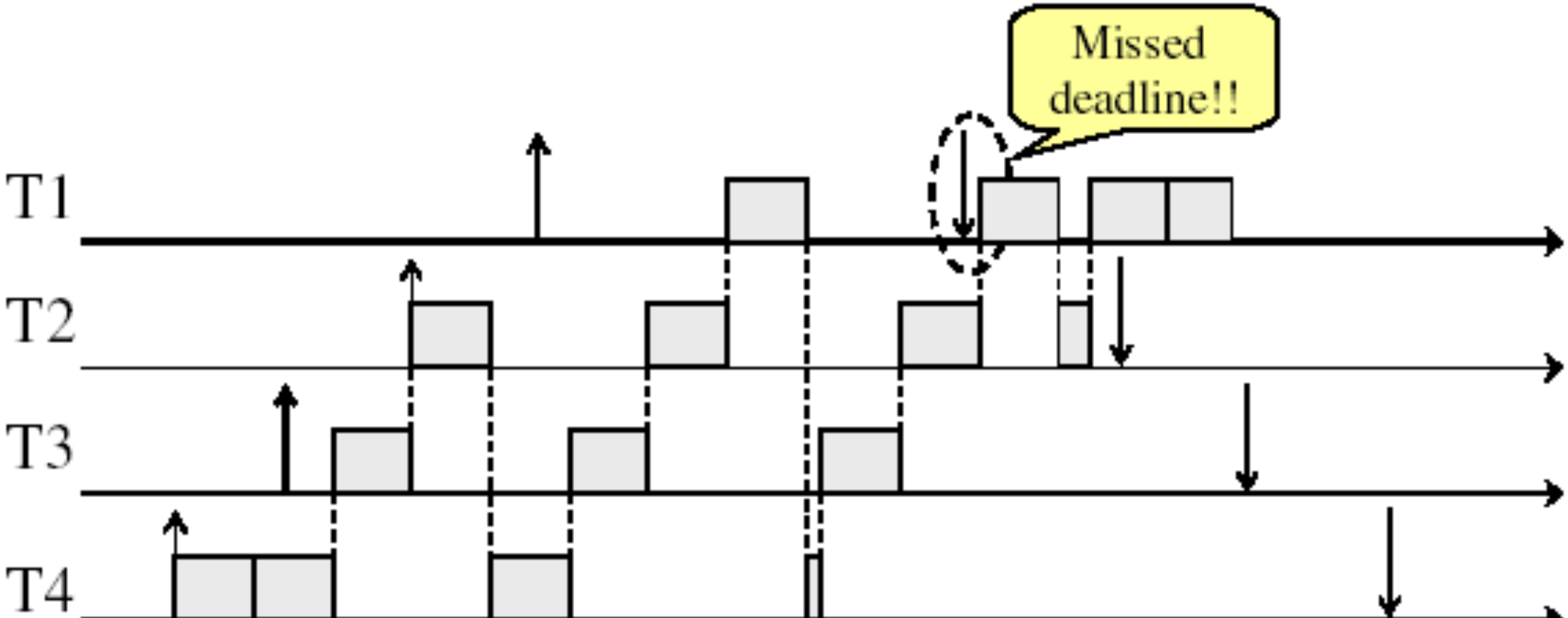
- Hard real-time:
  - Meet **all deadlines** (if possible)
  - Ideally: determine in advance if this is possible

Real-Time Example

Preemptible tasks with known deadlines (*D*) and known burst times (*C*)

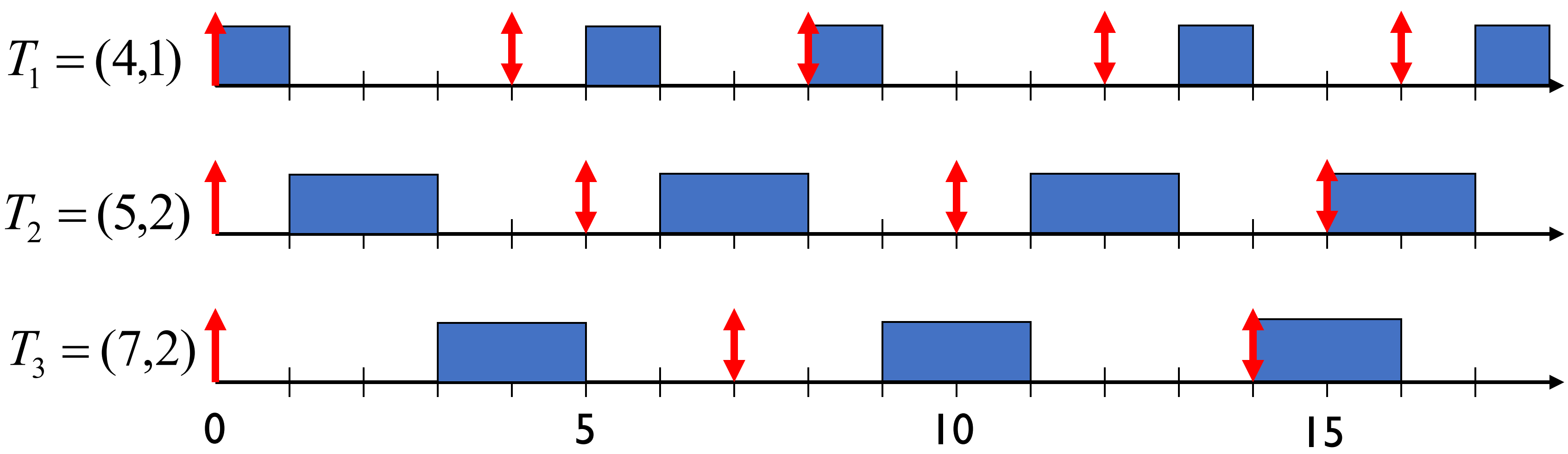


What if we try Round-Robin?



Earliest Deadline First

- Priority scheduling with preemption
- Priority proportional to time until deadline
- Example with **periodic tasks:**



EDF: Feasibility Testing

- Even EDF won't work if you have too many tasks
- For n tasks with computation time C and deadline D, a feasible schedule exists if:

$$\sum_{i=1}^{n} \left(\frac{C_i}{D_i}\right) \leq 1$$

Choosing the Right Scheduler

I Care About:	Then Choose:
CPU Throughput	FCFS
Avg. Response Time	SRTF Approximation
I/O Throughput	SRTF Approximation
Fairness (CPU Time)	Linux CFS
Fairness – Wait Time to Get CPU	Round Robin
Meeting Deadlines	EDF
Favoring Important Tasks	Priority

Further reading

- **Lottery scheduling: flexible proportional-share resource management, Waldspurger and Weihl, OSDI 1994**