IO Performance Oriented Drivers

David E. Culler

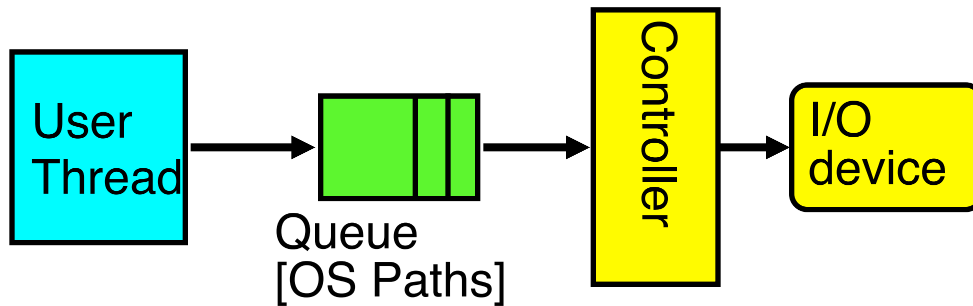
CS162 – Operating Systems and Systems Programming

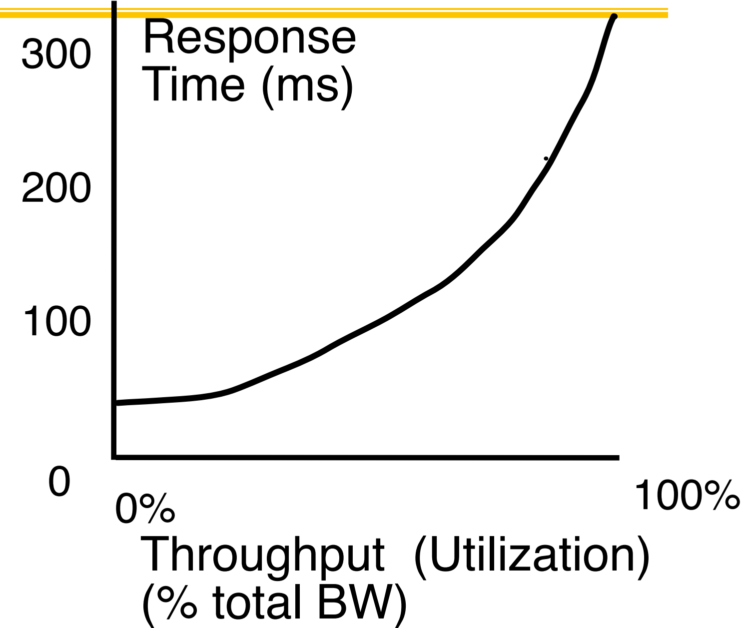
Lecture 22

October 20, 2014

Reading: A&D 7.5, 12.1c
HW 4 out
Proj 2 out

I/O Performance

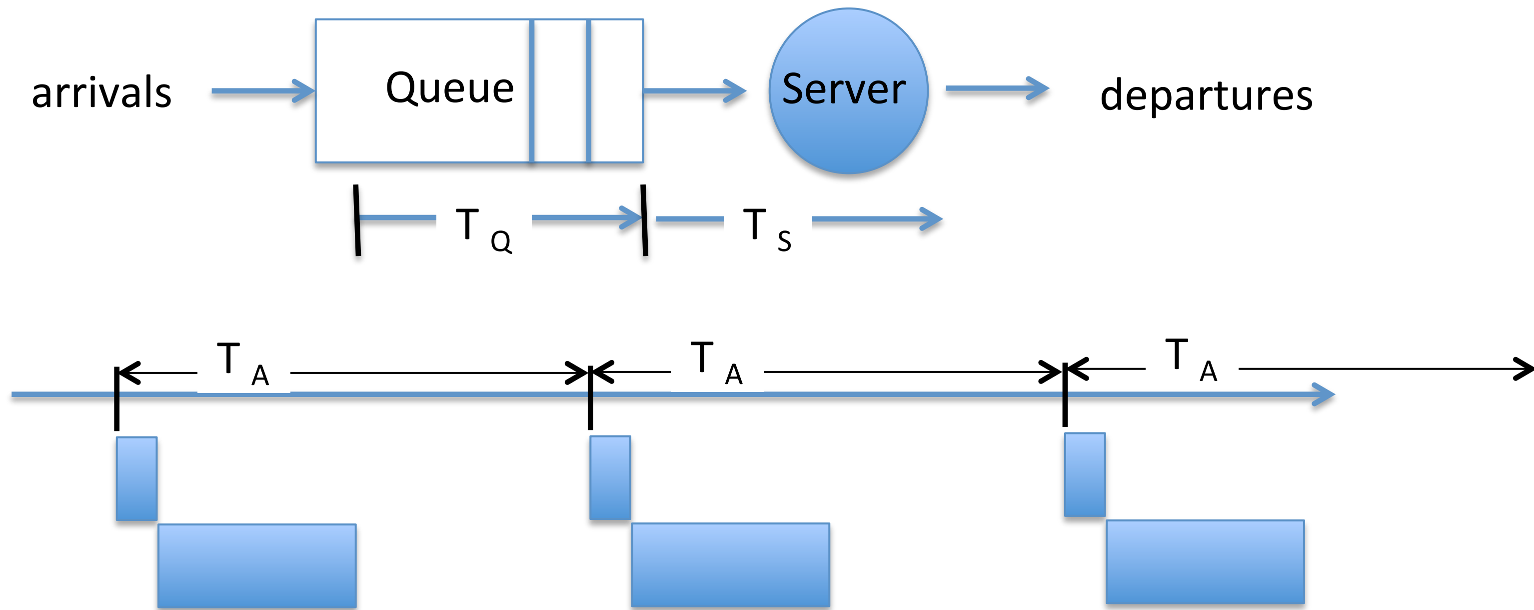




Response Time = Queue + I/O device service time

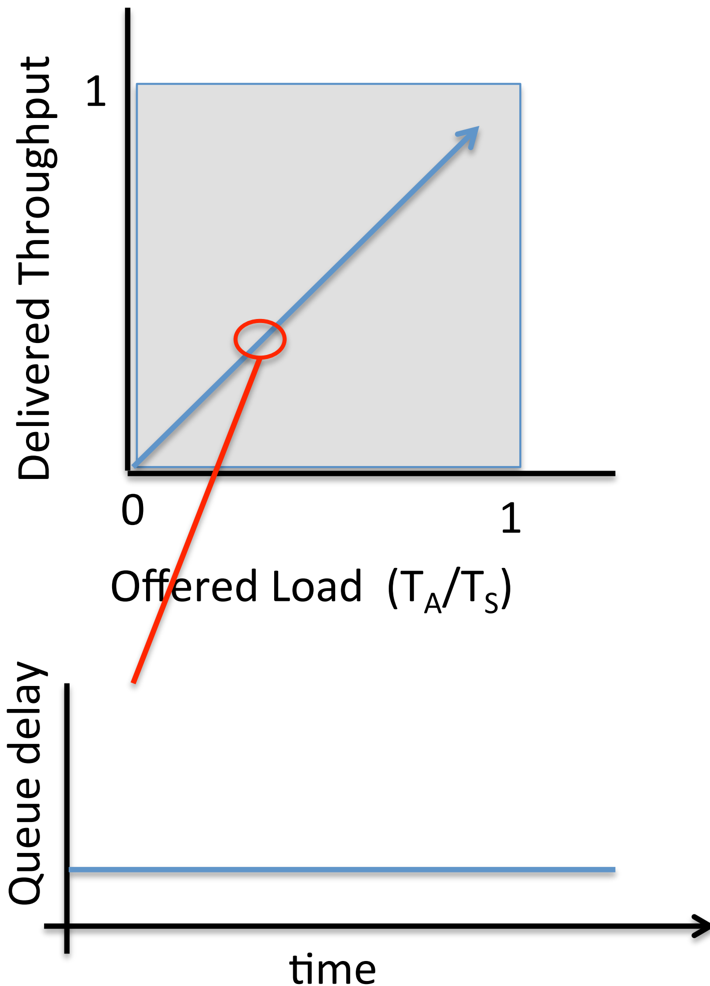
- Performance of I/O subsystem
  - Metrics: Response Time, Throughput
  - Effective BW per op = transfer size / response time
    - EffBW(n) = n / (S + n/B) = B / (1 + SB/n)
  - Contributing factors to latency:
    - Software paths (can be loosely modeled by a queue)
    - Hardware controller
    - I/O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

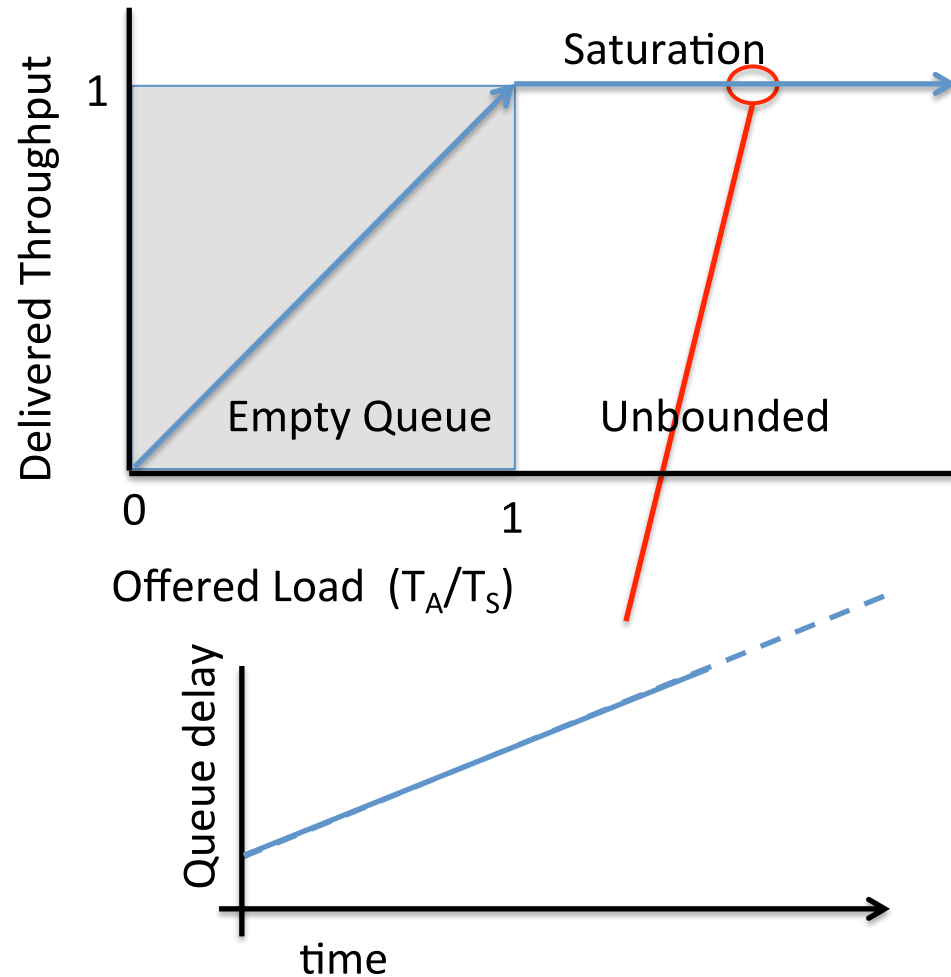
A Simple Deterministic World



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

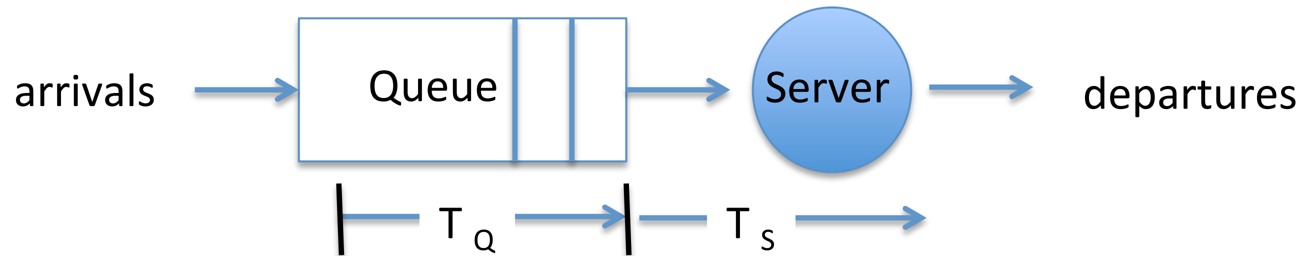
A Ideal Linear World

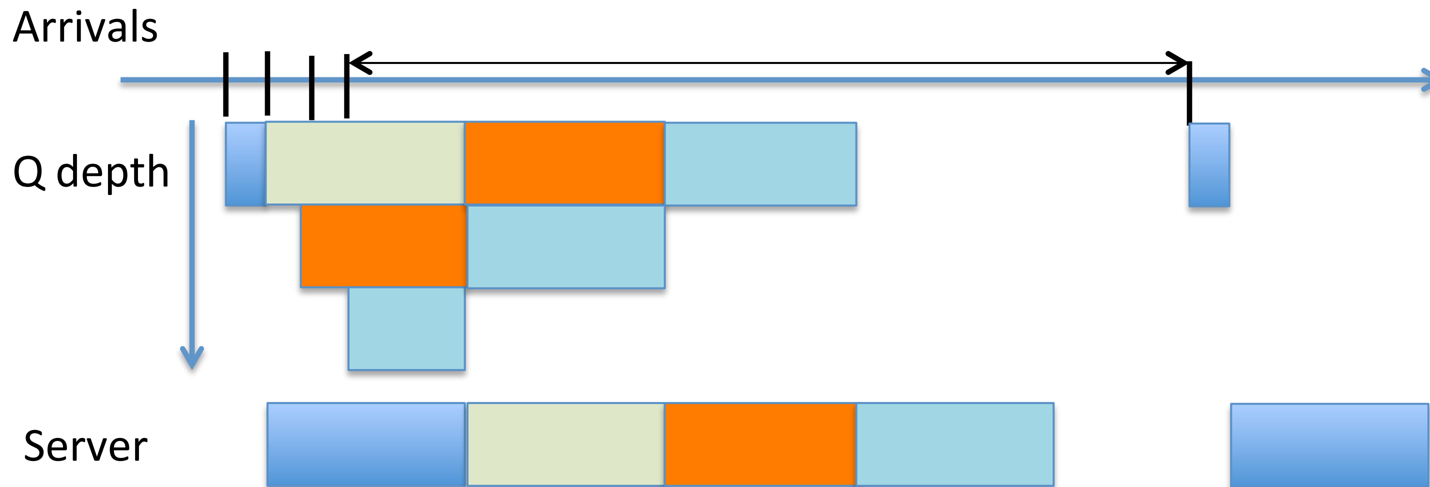




- What does the queue wait time look like?
  - Grows unbounded at a rate ~ (T_s/T_A) till request rate subsides

A Bursty World





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

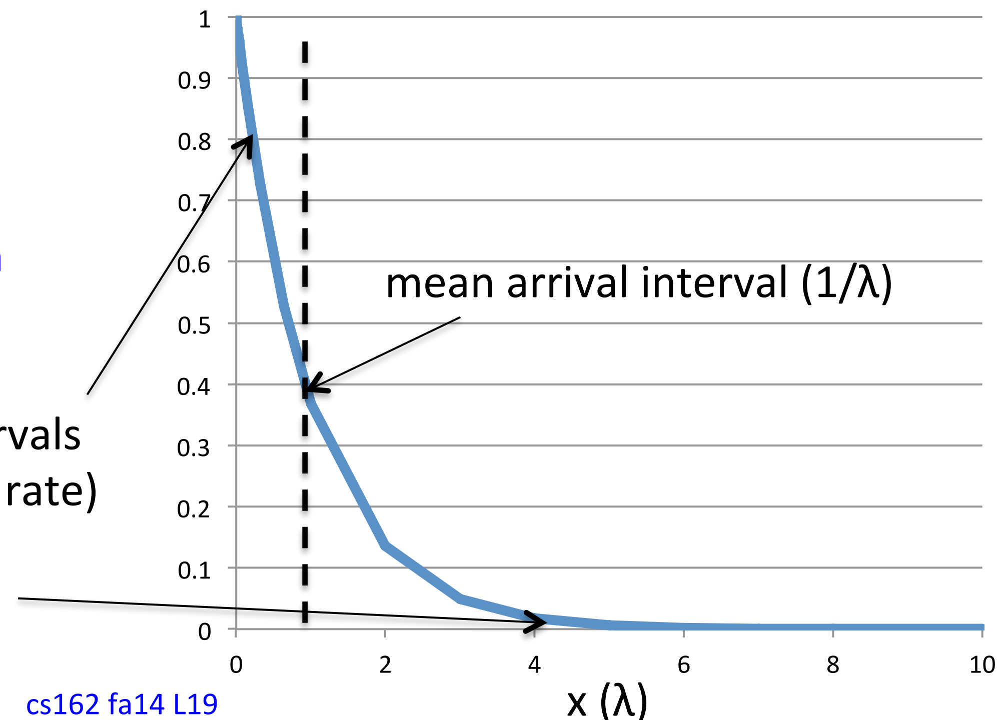
So how do we model the burstiness?

- Elegant mathematical framework if you start with *exponential distribution*
  - Probability density function of a continuous random variable with a mean of $1/\lambda$
  - $f(x) = \lambda e^{-\lambda x}$
  - *"Memoryless"*

Likelihood of an event occuring is independent of how long we've been waiting

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

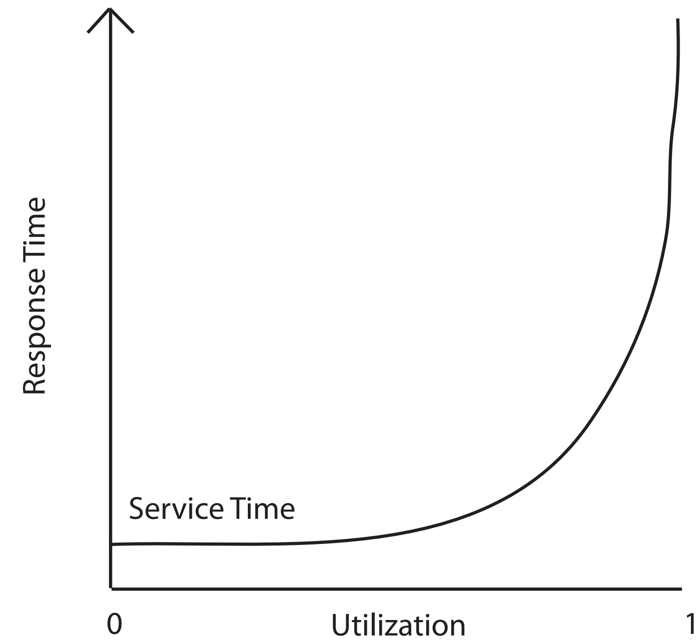
Few long gaps (i.e., low instantaneous rate)



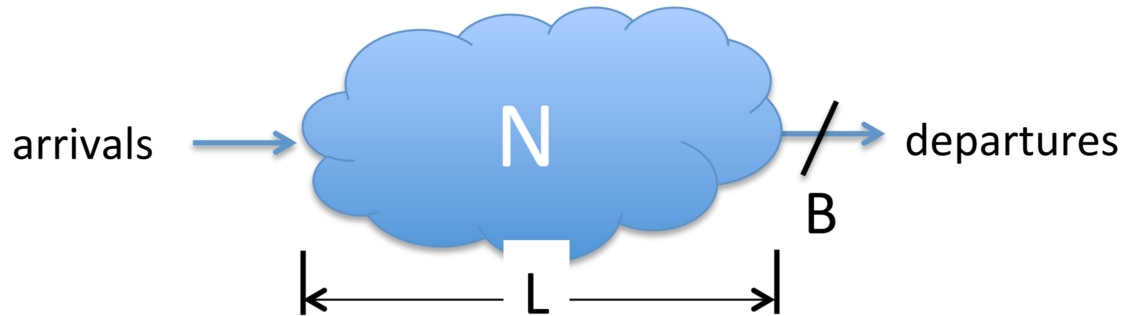
How Long should we expect to wait?

- RespTime = ServTime * 1/(1-U)
  - Better if gaussian (spread around the mean)
- Variance in R = $S/(1-U)^2$

Response Time vs. Utilization

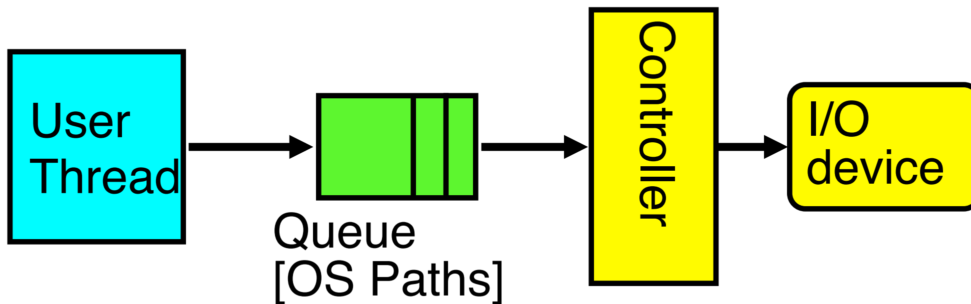


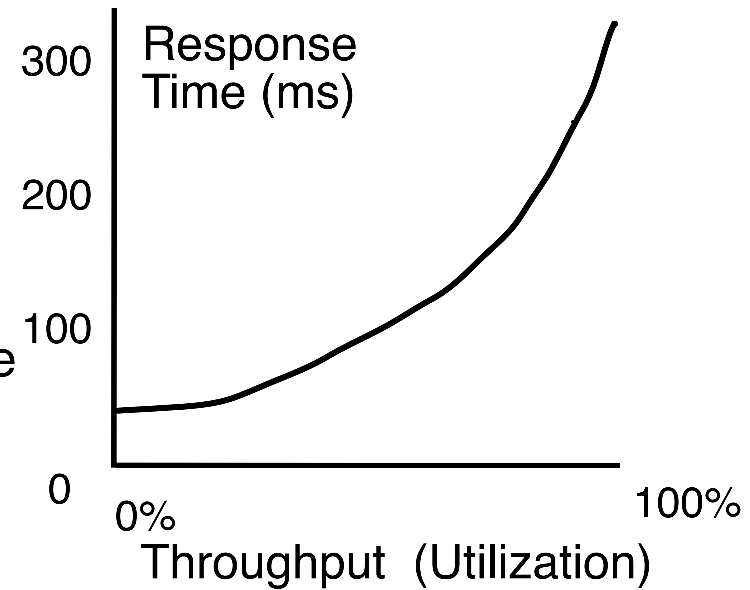
Little's Law



- In any *stable* system
  - Average arrival rate = Average departure rate
- the average number of tasks in the system (N) is equal to the throughput (B) times the response time (L)
- N (ops) = B (ops/s) x L (s)
- Regardless of structure, bursts of requests, variation in service
  - instantaneous variations, but it washes out in the average
  - Overall requests match departures

I/O Performance





Response Time = Queue + I/O device service time

- Solutions?
  - Make everything faster ☺
  - More Decoupled (Parallelism) systems
    - multiple independent buses or controllers
  - Optimize the bottleneck to increase service rate
    - Use the queue to optimize the service
  - Do other useful work while waiting
- Queues absorb bursts and smooth the flow
- Admissions control (finite queues)
  - Limits delays, but may introduce unfairness and livelock

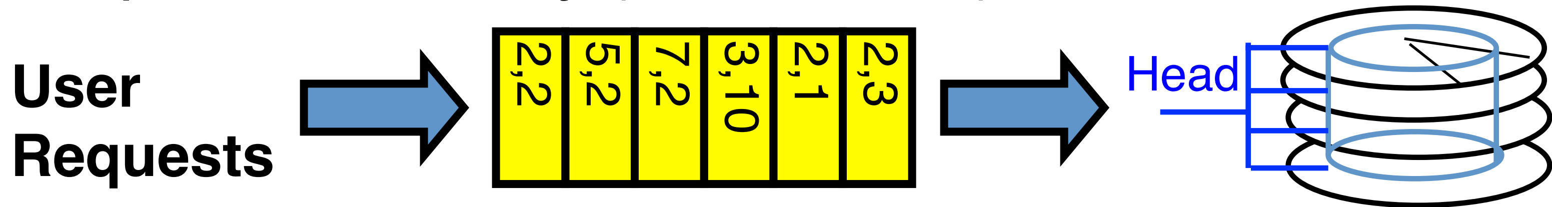
Ex: Disk Scheduling to Minimize Seek

Disk Performance Examples

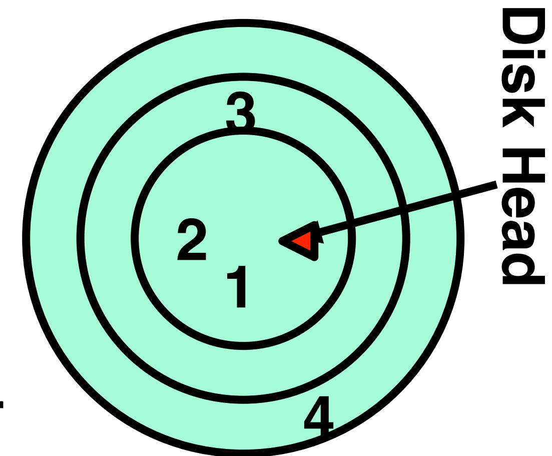
- Assumptions:
  – Ignoring queuing and controller times for now
  – Avg seek time of 5ms,
  – 7200RPM ⇒ Time for one rotation: 60000ms/7200 ~= 8ms
  – Transfer rate of 4MByte/s, sector size of 1 KByte
- Read sector from random place on disk:
  – Seek (5ms) + Rot. Delay (4ms) + Transfer (0.25ms)
  – Approx 10ms to fetch/put data: **100 KByte/sec**
- Read sector from random place in same cylinder:
  – Rot. Delay (4ms) + Transfer (0.25ms)
  – Approx 5ms to fetch/put data: **200 KByte/sec**
- Read next sector on same track:
  – Transfer (0.25ms): **4 MByte/sec**
- Key to using disk effectively (especially for file systems) is to *minimize seek and rotational delays*

Disk Scheduling

- Disk can do only one request at a time; What order do you choose to do queued requests?
  - Request denoted by (track, sector)

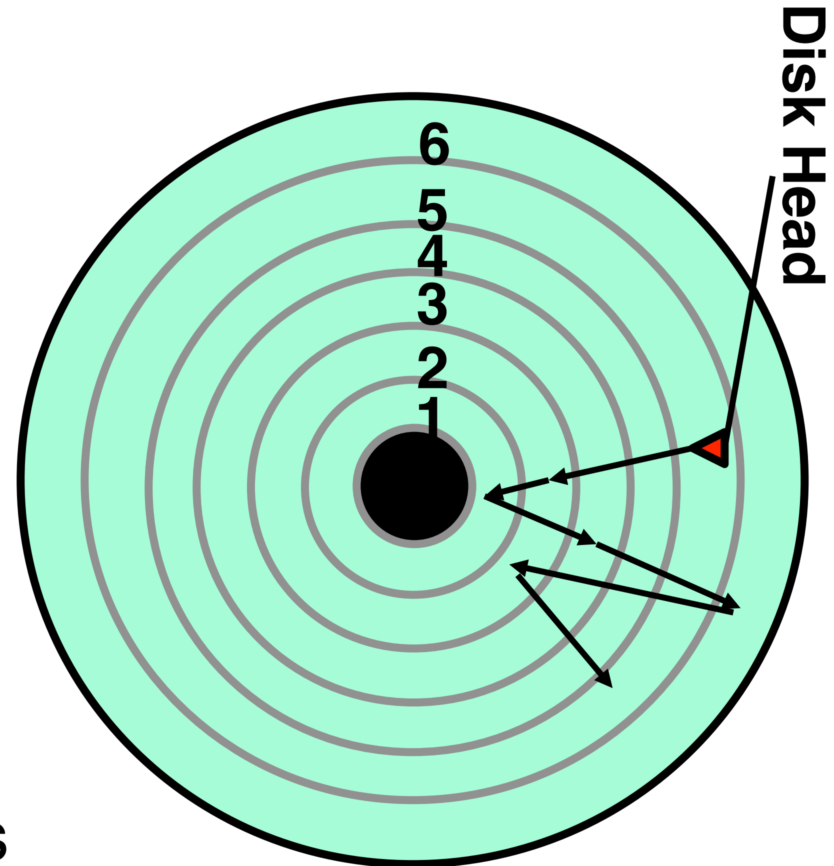


- Scheduling algorithms:
  - First In First Out (FIFO)
  - Shortest Seek Time First
  - SCAN
  - C-SCAN
- In our examples we ignore the sector
  - Consider only track #



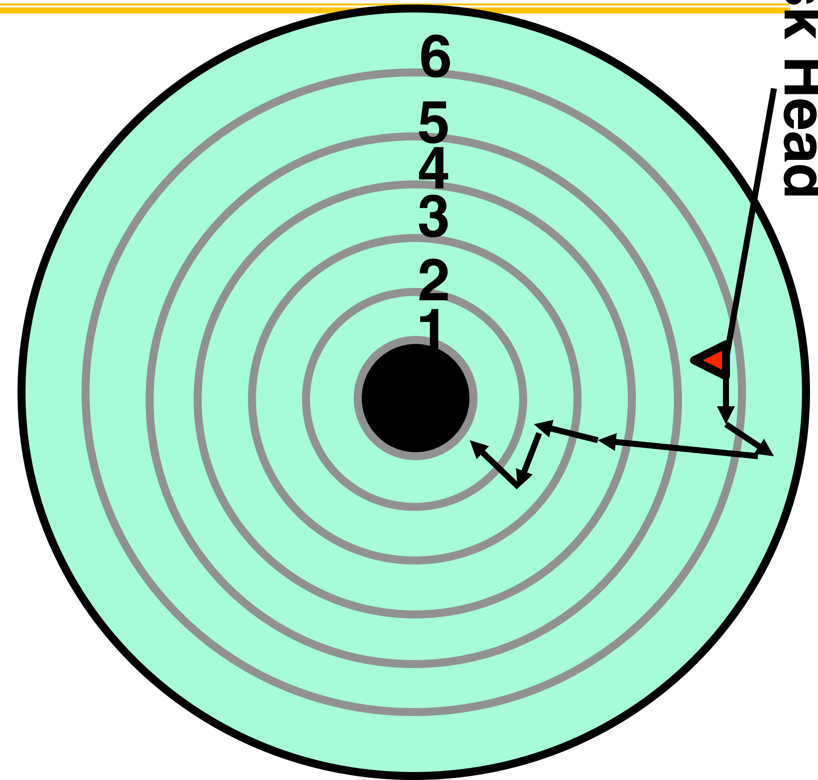
FIFO: First In First Out

- Schedule requests in the order they arrive in the queue
- Example:
  - Request queue: 2, 1, 3, 6, 2, 5
  - Scheduling order: 2, 1, 3, 6, 2, 5
  - 16 tracks, 6 seeks
  - Pros: Fair among requesters
  - Cons: Order of arrival may be to random spots on the disk ⇒ Very long seeks



SSTF: Shortest Seek Time First

- Pick the request that's closest to the head on the disk
  – Although called SSTF, include rotational delay in calculation, as rotation can be as long as seek

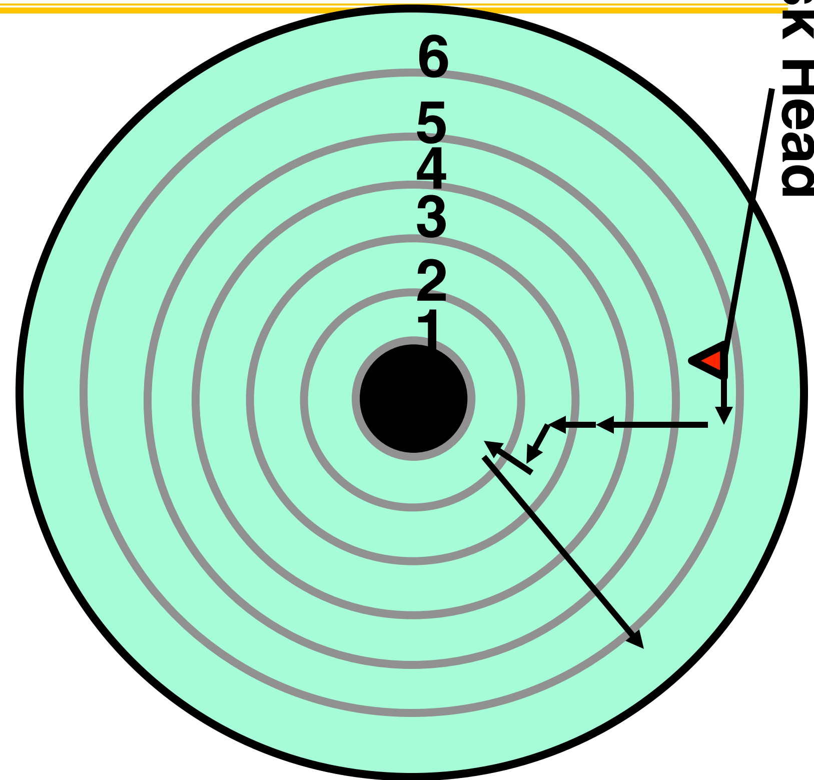


- Example:
  – Request queue: 2, 1, 3, 6, 2, 5
  – Scheduling order: 5, 6, 3, 2, 2, 1
  – 6 tracks, 4 seeks
- Pros: reduce seeks
- Cons: may lead to starvation
  – Greedy. Not optimal

SCAN

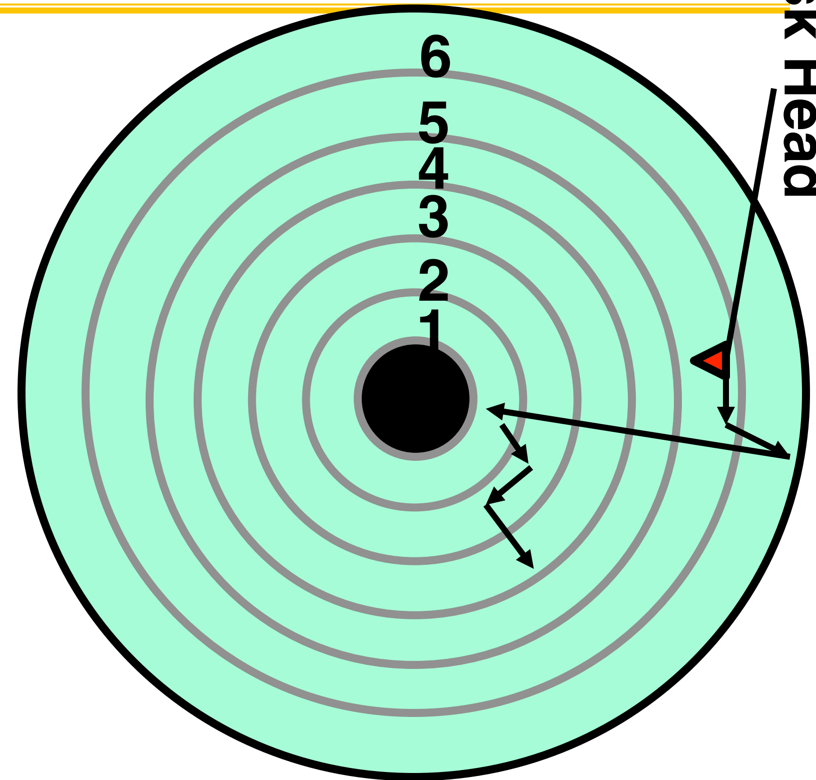
- Implements an Elevator Algorithm: take the closest request in the direction of travel
- Example:
  - Request queue: 2, 1, 3, 6, 2, 5
  - Head is moving towards center
  - Scheduling order: 5, 3, 2, 2, 1, 6

  - 8 tracks, 4 seeks
- Pros:
  - No starvation
  - Low seek
- Cons: favors middle tracks
  - May spend time on sparse tracks while dense requests elsewhere



C-SCAN

- Like SCAN but only serves request in only one direction
- Example:
  - Request queue: 2, 1, 3, 6, 2, 5
  - Head only serves request on its way from center towards edge
  - Scheduling order: 5, 6, 1, 2, 2, 3
  - 8 tracks, 5 seeks
- Pros:
  - Fairer than SCAN
  - Accumulate work in remote region then go get it
- Cons: longer seeks on the way back
- Optimization: dither to pickup nearby requests as you go



When is the disk performance highest

- When there are big sequential reads, or
- When there is so much work to do that they can be piggy backed (c-scan)

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

Ex: Concurrency to break the bottleneck

- Busses provide a way of connecting many (N) different things with a single set of wires and standard connections and protocols
  - N^2 relationships with 1 set of wires (!!!)
- But only one transaction can go on at a time
  - The rest have to wait
  - Queue up at "bus arbitration"

PCI Bus evolution

- PCI started life out as a bus
  - 32 physical bits double for address/data
- But a parallel bus has many limitations
  - Multiplexing address/data for many requests
  - Slowest device must be able to tell what's happening
  - ➔ Bus speed is set to that of the slowest device

PCI Express "Bus"

- No longer a parallel bus
- Really a **collection of fast serial channels** or "lanes"
- Devices can use as many as they need to achieve a desired bandwidth
- Slow devices don't have to share with fast ones
- Both motherboard slots and daughter cards are sized for the number of lanes, x4, x8, or x16
- Speeds (in an x16 configuration):
  - **v1.x**: 4 GB/s (40 GT/s)
  - **v2.x**: 8 GB/s (80 GT/s)
  - **v3.0**: 15.75 GB/s (128 GT/s)
  - **v4.0**: 31.51 GB/s (256 GT/s)

  3.0+ Speeds are competitive
  with **block memory-to-memory** operations on the CPU

PCI Express Interface (Linux)

- One of the successes of device abstraction in Linux was the ability to migrate from PCI to PCI-Express
- Although the physical interconnect changed completely, the old API still worked
- Drivers written for older PCI devices still worked, because of the standardized API for both models of the interface
- PCI register map:

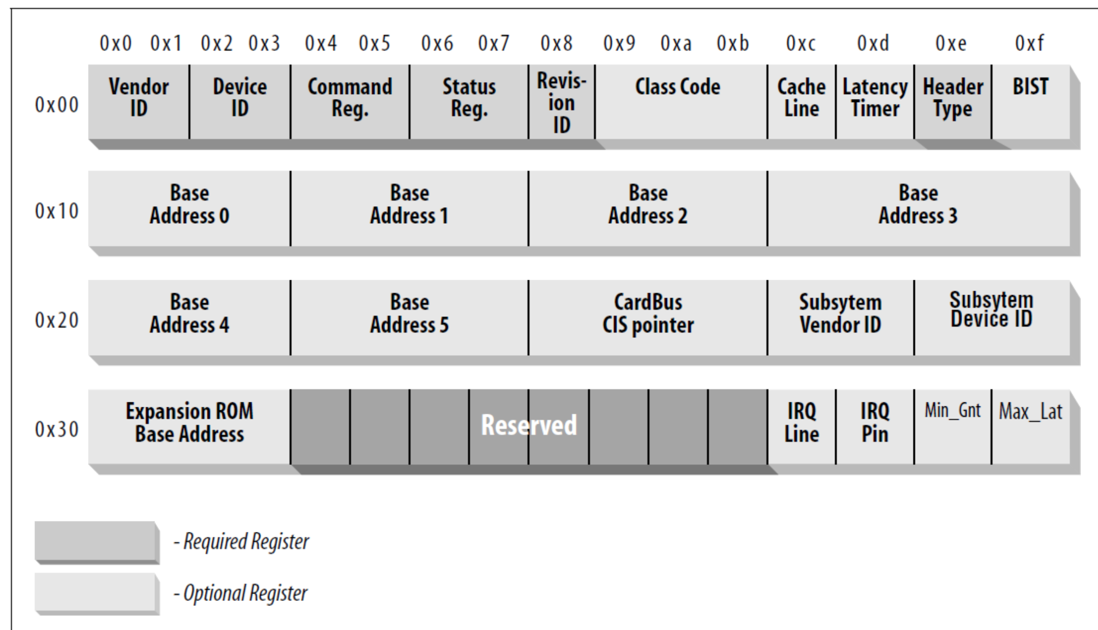


Figure 12-2. The standardized PCI configuration registers

Figure from "Linux Device Drivers," 3rd Ed, Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman

PCI Express Bus

In practice PCI is used as the interface to many other interconnects on a PC:

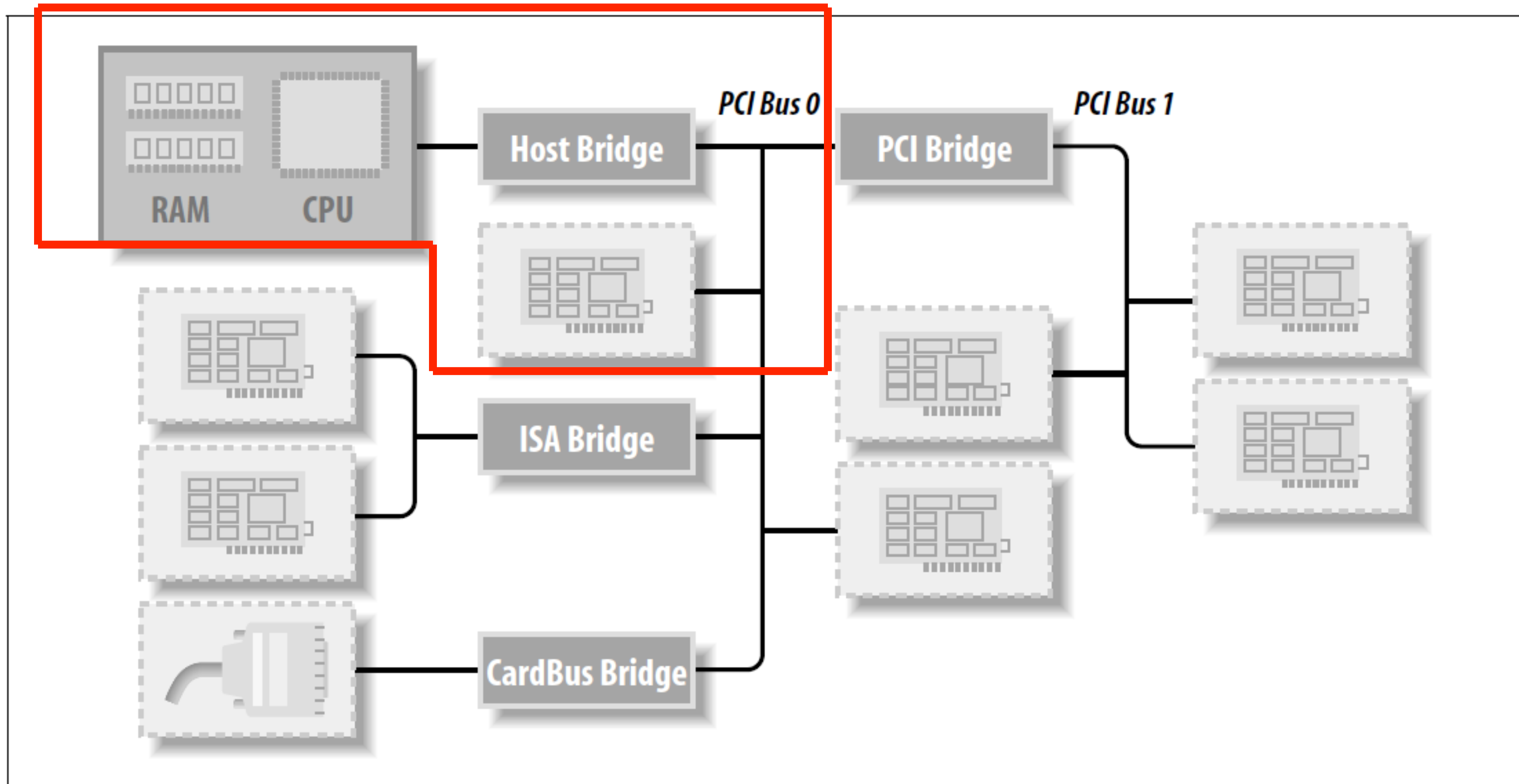


Figure 12-1. Layout of a typical PCI system

OS Solutions

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

How do we hide I/O latency?

- Blocking Interface: "Wait"
  - When request data (*e.g.*, `read()` system call), put process to sleep until data is ready
  - When write data (*e.g.*, `write()` system call), put process to sleep until device is ready for data
- Non-blocking Interface: "Don't Wait"
  - Returns quickly from read or write request with count of bytes successfully transferred to kernel
  - Read may return nothing, write may write nothing
- Asynchronous Interface: "Tell Me Later"
  - When requesting data, take pointer to user's buffer, return immediately; later kernel fills buffer and notifies user
  - When sending data, take pointer to user's buffer, return immediately; later kernel takes data and notifies user

Kernel vs User-level I/O

- Both are popular/practical for different reasons:
  - Kernel-level drivers for critical devices that must keep running, e.g. display drivers.
    - Programming is a major effort, correct operation of the rest of the kernel depends on correct driver operation.
  - User-level drivers for devices that are non-threatening, e.g USB devices in Linux (libusb).
    - Provide higher-level primitives to the programmer, avoid every driver doing low-level I/O register tweaking.
    - The multitude of USB devices can be supported by Less-Than-Wizard programmers.
    - New drivers don’t have to be compiled for each version of the OS, and loaded into the kernel.

Kernel vs User-level Programming Styles

- Kernel-level drivers
  - Have a much more limited set of resources available:
    - Only a fraction of libc routines typically available.
    - Memory allocation (e.g. Linux kmalloc) much more limited in capacity and required to be physically contiguous.
    - Should avoid blocking calls.
    - Can use asynchrony with other kernel functions but tricky with user code.
- User-level drivers
  - Similar to other application programs but:
    - Will be called often – should do its work fast, or postpone it – or do it in the background.
    - Can use threads, blocking operations (usually much simpler) or non-blocking or asynchronous.

Summary

- IO and data transfers often described by linear performance model and utilization model
  – $T(n) = S + n/B$
  – U = Service Rate / Request Rate
- But for shared resources burstiness in request rate can introduce substantial delays
- Subsystem design focuses on eliminating bottlenecks, e.g.,
  – disk scheduling to minimize seek and latency overhead
  – Multiple lanes to allow simultaneous transfers
  – Non-blocking requests (or threads) to overlap IO and compute
  – User level access to devices