CS152
Computer Architecture and Engineering
Lecture 24

I/O Systems II

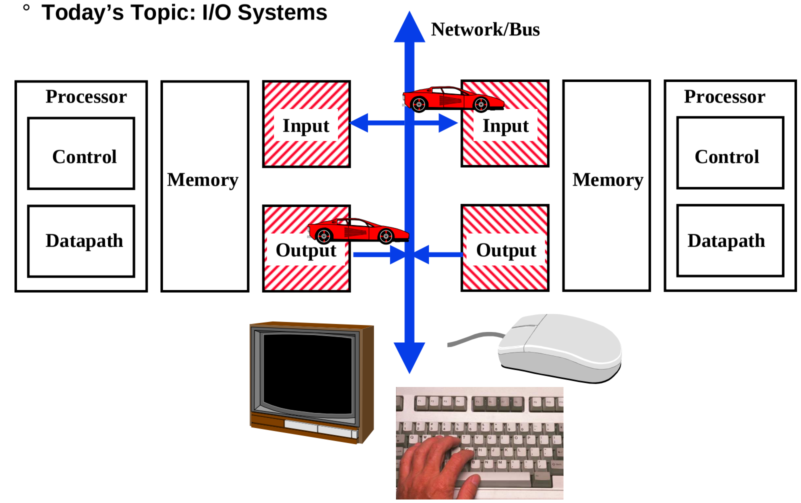
May 5, 1999

John Kubiatowicz (http.cs.berkeley.edu/~kubitron)

lecture slides: http://www-inst.eecs.berkeley.edu/~cs152/

The Big Picture: Where are We Now?

- Today's Topic: I/O Systems

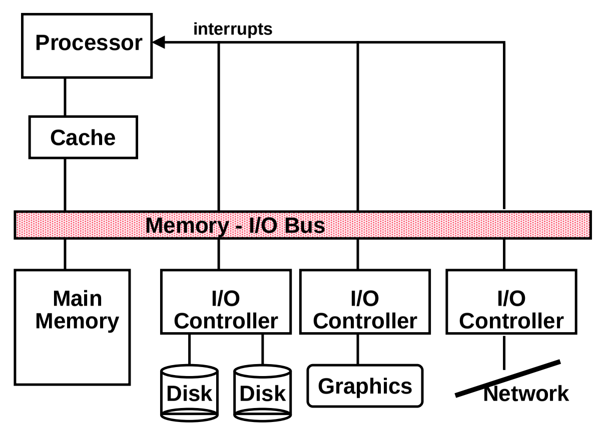


Outline of Today's Lecture

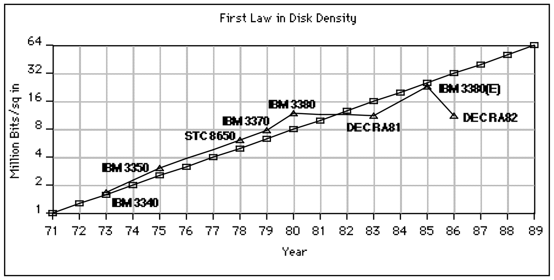
- Historical discussion of Disks
- A More queueing theory.
- Interfacing between processor and I/O devices
- RAID disk arrays
- Summary

Review: I/O System Design Issues

- Performance
- Expandability
- Resilience in the face of failure



Technology Trends



Disk Capacity now doubles every 18 months; before 1990 every 36 months

- Today: Processing Power Doubles Every 18 months
- Today: Memory Size Doubles Every 18 months(4X/3yr)
- Today: Disk Capacity Doubles Every 18 months
- *Disk Positioning Rate (Seek + Rotate) Doubles Every Ten Years!*

The I/O GAP

Storage Technology Drivers

° Driven by the prevailing computing paradigm
- 1950s: migration from batch to on-line processing
- 1990s: migration to ubiquitous computing
  - computers in phones, books, cars, video cameras, ...
  - nationwide fiber optical network with wireless tails

° Effects on storage industry:
- Embedded storage
  - smaller, cheaper, more reliable, lower power
- Data utilities
  - high capacity, hierarchically managed storage

Historical Perspective

° 1956 IBM Ramac — early 1970s Winchester
- Developed for mainframe computers, proprietary interfaces
- Steady shrink in form factor: 27 in. to 14 in.

° 1970s developments
- 5.25 inch floppy disk formfactor (microcode into mainframe)
- early emergence of industry standard disk interfaces
  - ST506, SASI, SMD, ESDI

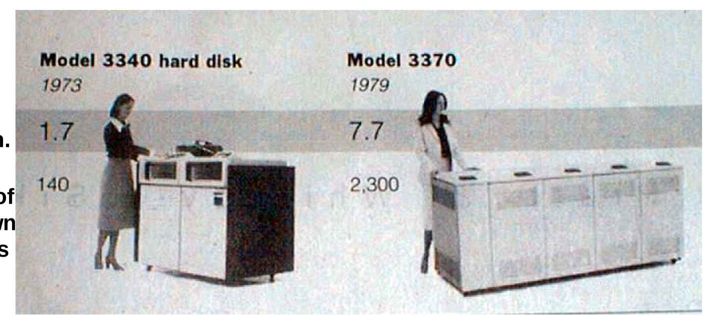
° Early 1980s
- PCs and first generation workstations

° Mid 1980s
- Client/server computing
- Centralized storage on file server
  - accelerates disk downsizing: 8 inch to 5.25 inch
- Mass market disk drives become a reality
  - industry standards: SCSI, IPI, IDE
  - 5.25 inch drives for standalone PCs, End of proprietary interfaces

Disk History

Data density Mbit/sq. in.

Capacity of Unit Shown Megabytes



1973: 1. 7 Mbit/sq. in 140 MBytes

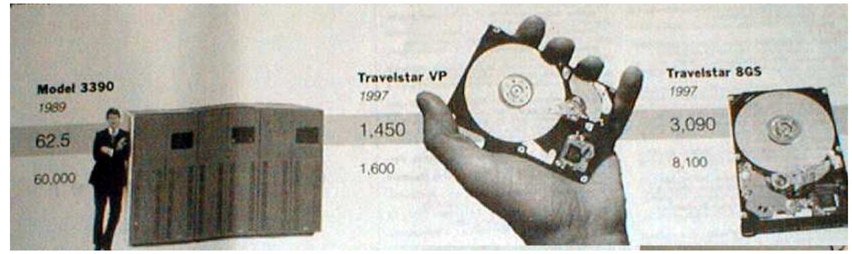
1979: 7. 7 Mbit/sq. in 2,300 MBytes

source: New York Times, 2/23/98, page C3,
"Makers of disk drives crowd even mroe data into even smaller spaces"

Historical Perspective

° **Late 1980s/Early 1990s:**
- Laptops, notebooks, (palmtops)
- 3.5 inch, 2.5 inch, (1.8 inch formfactors)
- Formfactor plus capacity drives market, not so much performance
  - Recently Bandwidth improving at 40%/ year
- Challenged by DRAM, flash RAM in PCMCIA cards
  - still expensive, Intel promises but doesn't deliver
  - unattractive MBytes per cubic inch
- Optical disk fails on performance (e.g., NEXT) but finds niche (CD ROM)

Disk History



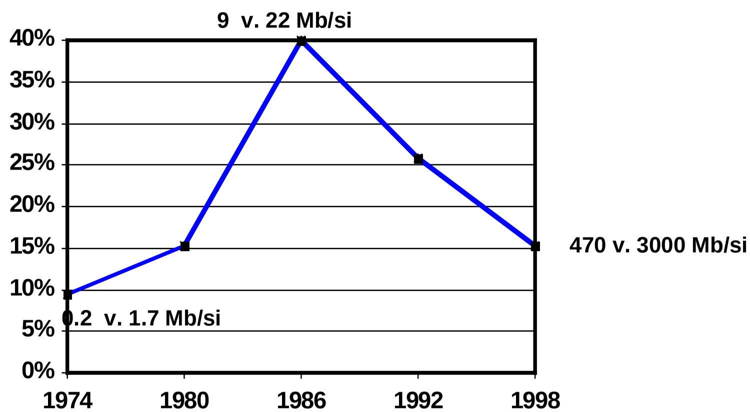
1989:
63 Mbit/sq. in
60,000 MBytes

1997:
1450 Mbit/sq. in
2300 MBytes

1997:
3090 Mbit/sq. in
8100 MBytes

source: New York Times, 2/23/98, page C3,
"Makers of disk drives crowd even more data into even smaller spaces"

MBits per square inch: DRAM as % of Disk over time

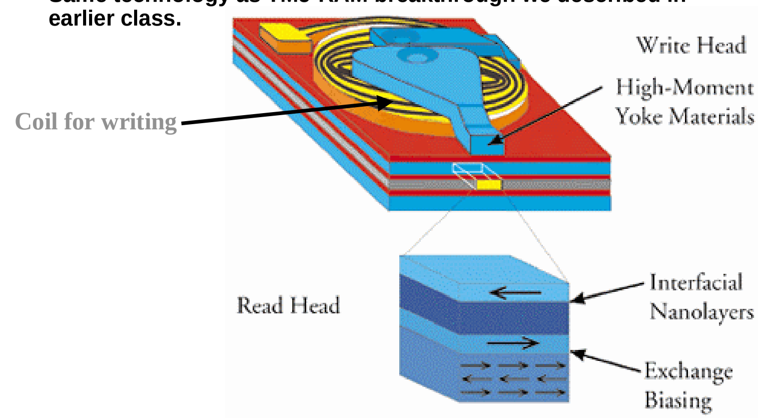


source: New York Times, 2/23/98, page C3,
"Makers of disk drives crowd even more data into even smaller spaces"

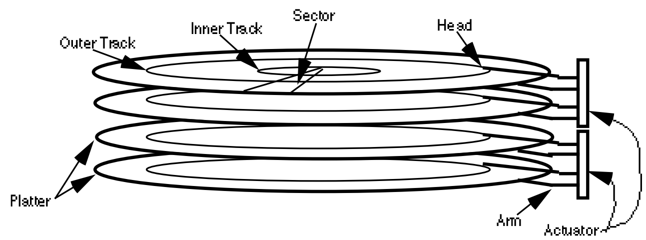
Nano-layered Disk Heads

° **Special sensitivity of Disk head comes from "Giant Magneto-Resistive effect" or (GMR)**

° **IBM is leader in this technology**
- Same technology as TMJ-RAM breakthrough we described in earlier class.



Review: Disk Device Terminology



Disk Latency = Queueing Time + Controller time + Seek Time + Rotation Time + Xfer Time

Order of magnitude times for 4K byte transfers:

Average Seek: 8 ms or less

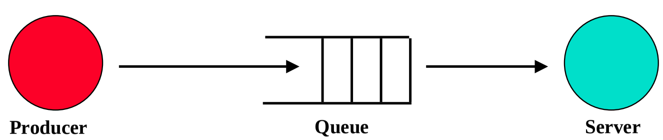
Rotate: 4.2 ms @ 7200 rpm

Xfer: 1 ms @ 7200 rpm

Review: Disk Time Example

- Disk Parameters:
  - Transfer size is 8K bytes
  - Advertised average seek is 12 ms
  - Disk spins at 7200 RPM
  - Transfer rate is 4 MB/sec
- Controller overhead is 2 ms
- Assume that disk is idle so no queuing delay
- What is Average Disk Access Time for a Sector?
  - Ave seek + ave rot delay + transfer time + controller overhead
  - 12 ms + 0.5/(7200 RPM/60) + 8 KB/4 MB/s + 2 ms
  - 12 + 4.15 + 2 + 2 = 20 ms
- Advertised seek time assumes no locality: typically 1/4 to 1/3 advertised seek time: 20 ms => 12 ms

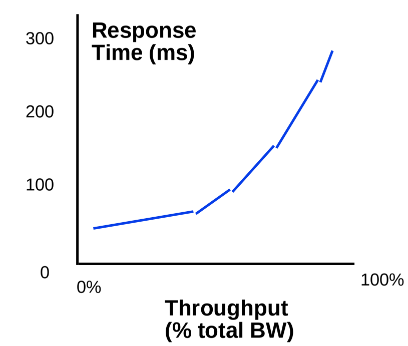
Review: Simple Producer-Server Model

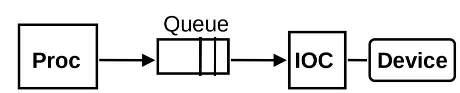


- Throughput:
  - The number of tasks completed by the server in unit time
  - In order to get the highest possible throughput:
    - The server should never be idle
    - The queue should never be empty
- Response time:
  - Begins when a task is placed in the queue
  - Ends when it is completed by the server
  - In order to minimize the response time:
    - The queue should be empty
    - The server will be idle

Review: Disk I/O Performance

Metrics:
Response Time
Throughput





Response time = Queue + Device Service time

Response Time vs. Productivity

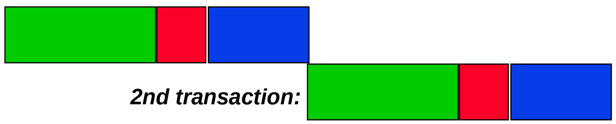
° **Interactive environments:**

Each interaction or *transaction* has 3 parts:

- *Entry Time*: time for user to enter command
- *System Response Time*: time between user entry & system replies
- *Think Time*: Time from response until user begins next command

1st transaction:

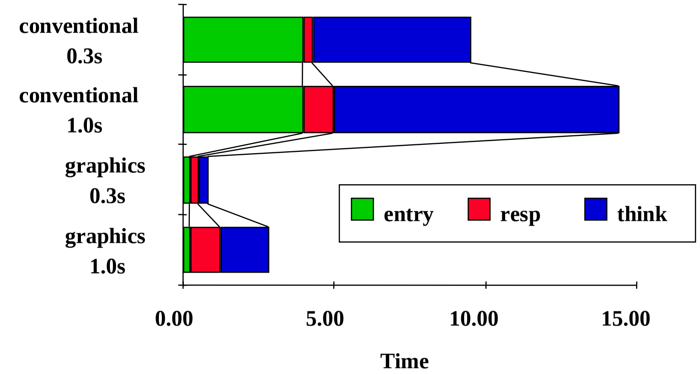
2nd transaction:



° What happens to transaction time as shrink system response time from 1.0 sec to 0.3 sec?

- With Keyboard: 4.0 sec entry, 9.4 sec think time
- With Graphics: 0.25 sec entry, 1.6 sec think time

Response Time & Productivity



° 0.7sec off response saves 4.9 sec (34%) and 2.0 sec (70%) total time per transaction => greater productivity

° Another study: everyone gets more done with faster response, but novice with fast response = expert with slow

Administrivia

° **Pending schedule:**

- Monday 5/10 Last class (wrap up, evaluations, etc)
- Tuesday 5/11 Oral reports: 10-12am and 2-4pmin 306 Soda
  - Signup sheet is on my office door
  - After oral reports (at 5pm), we will all meet in lab to run final mystery program.
- Tuesday 5/11 by 5pm: final project reports due.
- Friday 5/14 grades should be posted.

° **EMail to your TA an 8-digit "secret" number**

- Try to be somwhat creative with your number.
- We will combine this with the last few digits of your ID number to get a code for posting of final grades.

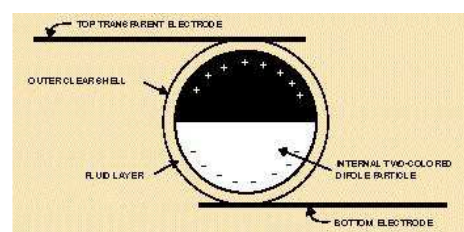
° **Solutions to Midterm II are finally up**

- Sorry about the delay; I have been traveling a lot

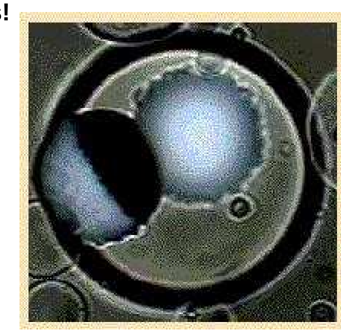
Computers in the News: Electronic Ink

° **Electronic Ink:**

- Little capsules with charged balls that are half black/half white
- Placing an electronic charge of one polarity makes dot black and the other polarity makes it white.
- Flexible, cheap, paper-like displays!



Schematic Diagram

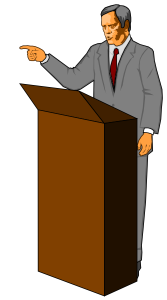


Electron Micrograph

Computers in the News: Logarithmic Computation

- Logarithmic Computation
  - New way to design ALUs to do add/sub/multiply/divide
  - Multiply/divide is easy => only add/subtract
  - Addition and subtraction is hard
    - Breakthrough claim is that way to do Addition and Subtraction easily on logarithms of numbers has been found
    - Combine table lookup + error calculation
  - Multiplication about 5 times faster than normal chip, division about 15 times faster.
  - 1/2 as complicated logic => lower power!
  - Researchers expect chips in 2001 time frame

7 Talk Commandments for a Bad Talk



I. Thou shalt not illustrate.

II. Thou shalt not covet brevity.

III. Thou shalt not print large.

IV. Thou shalt not use color.

V. Thou shalt not skip slides in a long talk.

VI. Thou shalt cover thy naked slides.

VII. Thou shalt not practice.

Following all the commandments

- We describe the philosophy and design of the control flow machine, and present the results of detailed simulations of the performance of a single processing element. Each factor is compared with the measured performance of an advanced von Neumann computer running equivalent code. It is shown that the control flow processor compares favorablylism in the program.
- We present a denotational semantics for a logic program to construct a control flow for the logic program. The control flow is defined as an algebraic manipulator of idempotent substitutions and it virtually reflects the resolution deductions. We also present a bottom-up compilation of medium grain clusters from a fine grain control flow graph. We compare the basic block and the dependence sets algorithms that partition control flow graphs into clusters.
- Our compiling strategy is to exploit coarse-grain parallelism at function application level: and the function application level parallelism is implemented by fork-join mechanism. The compiler translates source programs into control flow graphs based on analyzing flow of control, and then serializes instructions within graphs according to flow arcs such that function applications, which have no control dependency, are executed in parallel.
- A hierarchical macro-control-flow computation allows them to exploit the coarse grain parallelism inside a macrotask, such as a subroutine or a loop, hierarchically. We use a hierarchical definition of macrotasks, a parallelism extraction scheme among macrotasks defined inside an upper layer macrotask, and a scheduling scheme which assigns hierarchical macrotasks on hierarchical clusters.
- We apply a parallel simulation scheme to a real problem: the simulation of a control flow architecture, and we compare the performance of this simulator with that of a sequential one. Moreover, we investigate the effect of modelling the application on the performance of the simulator. Our study indicates that parallel simulation can reduce the execution time significantly if appropriate modelling is used.
- We have demonstrated that to achieve the best execution time for a control flow program, the number of nodes within the system and the type of mapping scheme used are particularly important. In addition, we observe that a large number of subsystem nodes allows more actors to be fired concurrently, but the communication overhead in passing control tokens to their destination nodes causes the overall execution time to increase substantially.
- The relationship between the mapping scheme employed and locality effect in a program are discussed. The mapping scheme employed has to exhibit a strong locality effect in order to allow efficient execution. We assess the average number of instructions in a cluster and the reduction in matching operations compared with fine grain control flow execution.
- Medium grain execution can benefit from a higher output bandwidth of a processor and finally, a simple superscalar processor with an issue rate of ten is sufficient to exploit the internal parallelism of a cluster. Although the technique does not exhaustively detect all possible errors, it detects nontrivial errors with a worst-case complexity quadratic to the system size. It can be automated and applied to systems with arbitrary loops and nondeterminism.

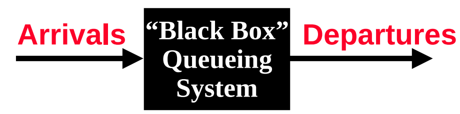
Alternatives to a Bad Talk

- Practice, Practice, Practice!
  - Use casette tape recorder to listen, practice
  - Try videotaping
  - Seek feedback from friends
- Use phrases, not sentences
  - Notes separate from slides (don't read slide)
- Pick appropriate font, size (~ 24 point to 32 point)
- Estimate talk length
  - - 2 minutes per slide
  - Use extras as backup slides (Question and Answer)
- Use color tastefully (graphs, emphasis)
- Don't cover slides
  - Use overlays or builds in powerpoint
- Go to room early to find out what is WRONG with setup
  - Beware: PC projection + dark rooms after meal!

Include in your final presentation

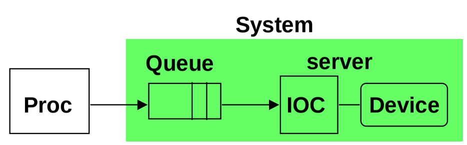
- Who is on team, and who did what
- High-level description of what you did and how you combined components together
  - Use block diagrams rather than detailed schematics
  - Assume audience knows Chapters 6 and 7 already
- Include novel aspects of design
  - Did you innovate? How?
  - Why did you choose to do things the way that you did?
- Give Critical Path and Clock cycle time
  - Bring paper copy of schematics in case there are detailed questions.
  - What could be done to improve clock cycle time?
- Mystery program statistics: instructions, clock cycles, CPI, why stalls occur (cache miss, load-use interlocks, branch mispredictions, ...)
- Lessons learned from project, what might do different next time

Introduction to Queueing Theory



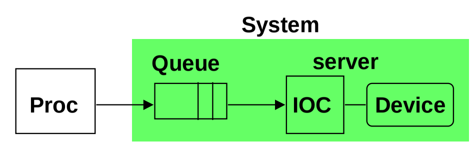
- Queueing Theory applies to long term, steady state behavior ⇒ Arrival rate = Departure rate
- Little's Law:
  Mean number tasks in system = arrival rate x mean reponse time
  - Observed by many, Little was first to prove
  - Simple interpretation: you should see the same number of tasks in queue when entering as when leaving.
- Applies to any system in equilibrium, as long as nothing in black box is creating or destroying tasks

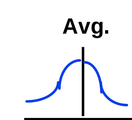
A Little Queuing Theory: Notation



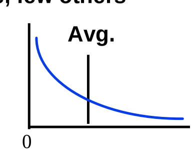
- Queuing models assume state of equilibrium: input rate = output rate
- Notation:
  - λ average number of arriving customers/second
  - T_{ser} average time to service a customer (tradtionally $\mu = 1/ T_{ser}$)
  - u server utilization (0..1): $u = \lambda \times T_{ser}$ (or $u = \lambda / \mu$)
  - T_q average time/customer in queue
  - T_{sys} average time/customer in system: $T_{sys} = T_q + T_{ser}$
  - L_q average length of queue: $L_q = \lambda \times T_q$
  - L_{sys} average length of system: $L_{sys} = \lambda \times T_{sys}$
- Little's Law: $L_{sys} = \lambda \times T_{sys}$
  (Mean number customers = arrival rate x mean service time)

A Little Queuing Theory: Use of random distributions

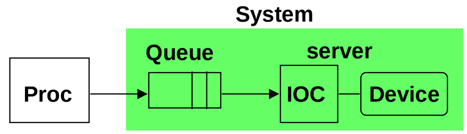




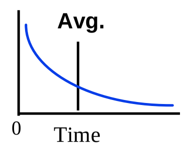
- Server spends a variable amount of time with customers
  - Weighted mean *m1* = (f1 x T1 + f2 x T2 +...+ fn x Tn)/F
  - *variance* = (f1 x T1^2 + f2 x T2^2 +...+ fn x Tn^2)/F – m1^2
    - Must keep track of unit of measure (100 ms^2 vs. 0.1 s^2)
  - *Squared coefficient of variance*: *C = variance/m1^2*
    - Unitless measure (100 ms^2 vs. 0.1 s^2)
- Exponential distribution *C = 1* : most short relative to average, few others long; 90% < 2.3 x average, 63% < average
  Hypoexponential distribution *C < 1* : most close to average, C=0.5 => 90% < 2.0 x average, only 57% < average
  Hyperexponential distribution *C > 1* : further from average C=2.0 => 90% < 2.8 x average, 69% < average



A Little Queuing Theory: Variable Service Time



- Disk response times $C \approx 1.5$ (majority seeks < average)
- Yet usually pick $C = 1.0$ for simplicity
  - Memoryless, exponential dist
  - Many complex systems well described by memoryless distribution!



- Another useful value is average time must wait for server to complete current task: $m1(z)$
  - Not just 1/2 x m1 because doesn't capture variance
  - Can derive $m1(z) = 1/2 \times m1 \times (1 + C)$
  - *No variance* $\Rightarrow C = 0 \Rightarrow m1(z) = 1/2 \times m1$
  - *Exponential* $\Rightarrow C = 1 \Rightarrow m1(z) = m1$

A Little Queuing Theory: Average Wait Time

- Calculating average wait time in queue T_q:
  - If something at server, it takes to complete on average $m1(z)$
  - Chance server is busy = u; average delay is $u \times m1(z)$
  - All customers in line must complete; each avg T_{ser}

$T_q = u \times \underline{m1(z)} + L_q \times T_{ser}$ Little's Law
$T_q = u \times m1(z) + \lambda \times T_q \times T_{ser}$ Defn of utilization (u)
$T_q = u \times m1(z) + u \times T_q$
$T_q \times (1 - u) = m1(z) \times u$
$T_q = m1(z) \times u/(1-U) = T_{ser} \times \{1/2 \times (1+C)\} \times u/(1 - u))$

Notation:

λ	average number of arriving customers/second
T_{ser}	average time to service a customer
u	server utilization (0..1): $u = r \times T_{ser}$
T_q	average time/customer in queue
L_q	average length of queue: $L_q = r \times T_q$

A Little Queuing Theory: M/G/1 and M/M/1

- Assumptions so far:
  - System in equilibrium
  - Time between two successive arrivals in line are random
  - Server can start on next customer immediately after prior finishes
  - No limit to the queue: works First-In-First-Out
  - Afterward, all customers in line must complete; each avg T_{ser}
- Described "memoryless" or Markovian request arrival (M for C=1 exponentially random), General service distribution (no restrictions), 1 server: *M/G/1 queue*
- When Service times have C = 1, *M/M/1 queue*

$$T_q = T_{ser} \times u / (1 - u)$$

T_{ser}	average time to service a customer
u	server utilization (0..1): $u = \lambda \times T_{ser}$
T_q	average time/customer in queue

A Little Queuing Theory: An Example

- Processor sends 10 x 8KB disk I/Os per second, requests & service exponentially distrib., avg. disk service = 20 ms
  - This number comes from disk equation: Service time = Ave seek + ave rot delay + transfer time + ctrl overhead
- On average, how utilized is the disk?
  - What is the number of requests in the queue?
  - What is the average time spent in the queue?
  - What is the average response time for a disk request?
- Notation:

λ	average number of arriving customers/second = 10
T_{ser}	average time to service a customer = 20 ms (0.02s)
u	server utilization (0..1): $u = \lambda \times T_{ser}$ = 10/s x .02s = 0.2
T_q	average time/customer in queue = $T_{ser} \times u / (1 - u)$ = 20 x 0.2/(1-0.2) = 20 x 0.25 = 5 ms (0 .005s)
T_{sys}	average time/customer in system: $T_{sys} = T_q + T_{ser}$ = 25 ms
L_q	average length of queue: $L_q = \lambda \times T_q$ = 10/s x .005s = 0.05 requests in queue
L_{sys}	average # tasks in system: $L_{sys} = \lambda \times T_{sys}$ = 10/s x .025s = 0.25

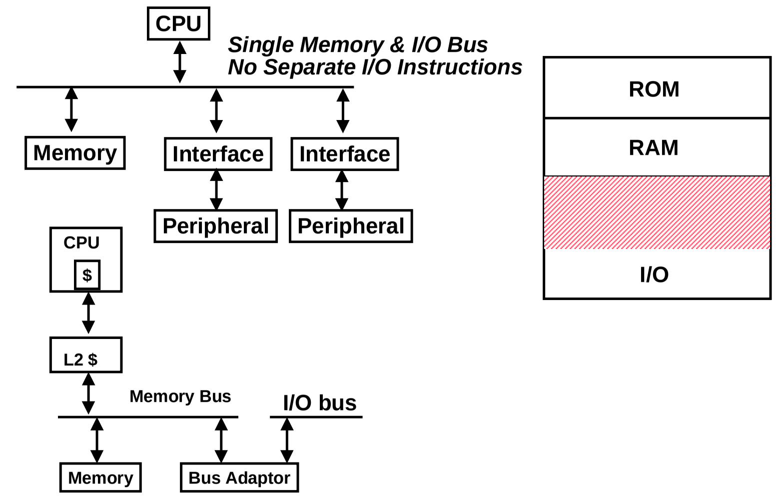
Giving Commands to I/O Devices

° Two methods are used to address the device:
  - Special I/O instructions
  - Memory-mapped I/O

° Special I/O instructions specify:
  - Both the device number and the command word
    - Device number: the processor communicates this via a set of wires normally included as part of the I/O bus
    - Command word: this is usually send on the bus's data lines

° Memory-mapped I/O:
  - Portions of the address space are assigned to I/O device
  - Read and writes to those addresses are interpreted as commands to the I/O devices
  - User programs are prevented from issuing I/O operations directly:
    - The I/O address space is protected by the address translation

Memory Mapped I/O



I/O Device Notifying the OS

° The OS needs to know when:
  - The I/O device has completed an operation
  - The I/O operation has encountered an error

° This can be accomplished in two different ways
  - I/O Interrupt:
    - Whenever an I/O device needs attention from the processor, it interrupts the processor from what it is currently doing.
  - Polling:
    - The I/O device put information in a status register
    - The OS periodically check the status register

I/O Interrupt

° An I/O interrupt is just like the exceptions except:
  - An I/O interrupt is asynchronous
  - Further information needs to be conveyed

° An I/O interrupt is asynchronous with respect to instruction execution:
  - I/O interrupt is not associated with any instruction
  - I/O interrupt does not prevent any instruction from completion
    - You can pick your own convenient point to take an interrupt

° I/O interrupt is more complicated than exception:
  - Needs to convey the identity of the device generating the interrupt
  - Interrupt requests can have different urgencies:
    - Interrupt request needs to be prioritized

Example: Device Interrupt

External Interrupt

```
        ...
add    $r1,$r2,$r3
subi   $r4,$r1,#4
slli   $r4,$r4,#2
    Hiccup(!)
lw     $r2,0($r4)
lw     $r3,4($r4)
add    $r2,$r2,$r3
sw     8($r4),$r2
        ...
```

PC saved
Disable All Ints
Supervisor Mode

Restore PC
User Mode

```
Raise priority
Reenable All Ints
Save registers
      ...
lw   $r1,20($r0)
lw   $r2,0($r1)
addi $r3,$r0,#5
sw   $r3,0($r1)
      ...
Restore registers
Clear current Int
Disable All Ints
Restore priority
RTI
```

"Interrupt Handler"

- Advantage:
  - User program progress is only halted during actual transfer
- Disadvantage, special hardware is needed to:
  - Cause an interrupt (I/O device)
  - Detect an interrupt (processor)
  - Save the proper states to resume after the interrupt (processor)

Alternative: Polling

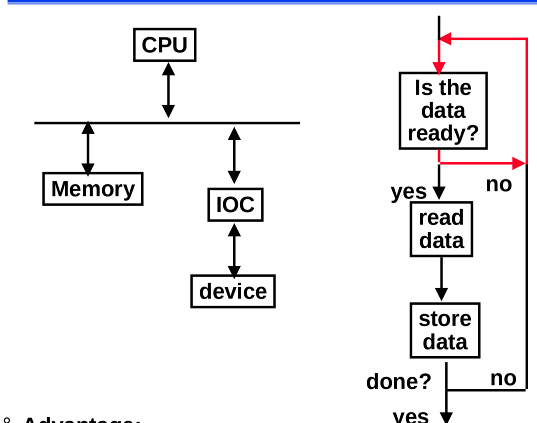
External Interrupt

```
      Disable Network Intr
         ...
      subi  $r4,$r1,#4
      slli  $r4,$r4,#2
      lw    $r2,0($r4)
      lw    $r3,4($r4)
      add   $r2,$r2,$r3
      sw    8($r4),$r2
      lw    $r1,12($zero)
      beq   $r1,no_mess
      lw    $r1,20($r0)
      lw    $r2,0($r1)
      addi  $r3,$r0,#5
      sw    0($r1),$r3
      Clear Network Intr
no_mess:   ...
```

Polling Point (check device register)

"Handler"

Polling: Programmed I/O



busy wait loop not an efficient way to use the CPU unless the device is very fast!

but checks for I/O completion can be dispersed among computation intensive code

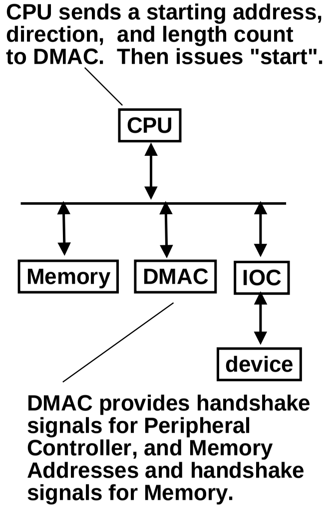
- Advantage:
  - Simple: the processor is totally in control and does all the work
- Disadvantage:
  - Polling overhead can consume a lot of CPU time

Polling is faster/slower than Interrupts

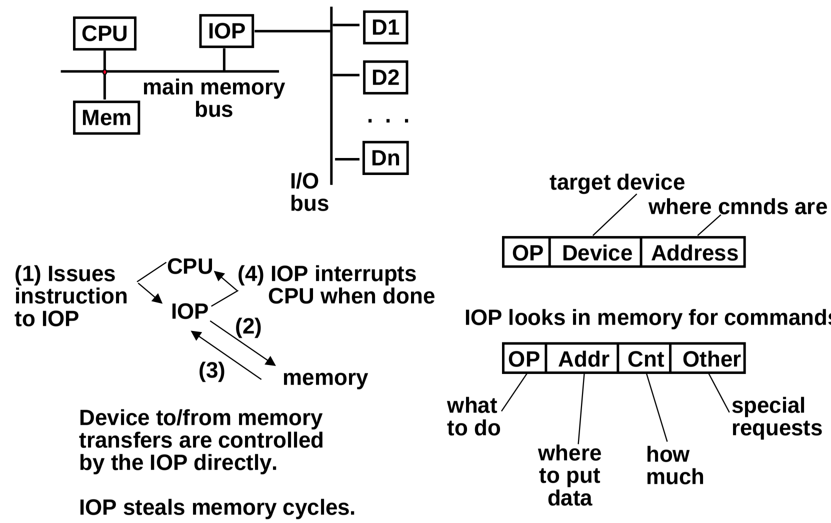
- Polling is faster than interrupts because
  - Compiler knows which registers in use at polling point. Hence, do not need to save and restore registers (or not as many).
  - Other interrupt overhead avoided (pipeline flush, trap priorities, etc).
- Polling is slower than interrupts because
  - Overhead of polling instructions is incurred regardless of whether or not handler is run. This could add to inner-loop delay.
  - Device may have to wait for service for a long time.
- When to use one or the other?
  - Multi-axis tradeoff
    - Frequent/regular events good for polling, *as long as device can be controlled at user level.*
    - Interrupts good for infrequent/irregular events
    - Interrupts good for ensuring regular/predictable service of events.

Delegating I/O Responsibility from the CPU: DMA

- Direct Memory Access (DMA):
  - External to the CPU
  - Act as a maser on the bus
  - Transfer blocks of data to or from memory without CPU intervention



Delegating I/O Responsibility from the CPU: IOP



Responsibilities of the Operating System

- The operating system acts as the interface between:
  - The I/O hardware and the program that requests I/O
- Three characteristics of the I/O systems:
  - The I/O system is shared by multiple program using the processor
  - I/O systems often use interrupts (external generated exceptions) to communicate information about I/O operations.
    - Interrupts must be handled by the OS because they cause a transfer to supervisor mode
  - The low-level control of an I/O device is complex:
    - Managing a set of concurrent events
    - The requirements for correct device control are very detailed

Operating System Requirements

- Provide protection to shared I/O resources
  - Guarantees that a user's program can only access the portions of an I/O device to which the user has rights
- Provides abstraction for accessing devices:
  - Supply routines that handle low-level device operation
- Handles the interrupts generated by I/O devices
- Provide equitable access to the shared I/O resources
  - All user programs must have equal access to the I/O resources
- Schedule accesses in order to enhance system throughput

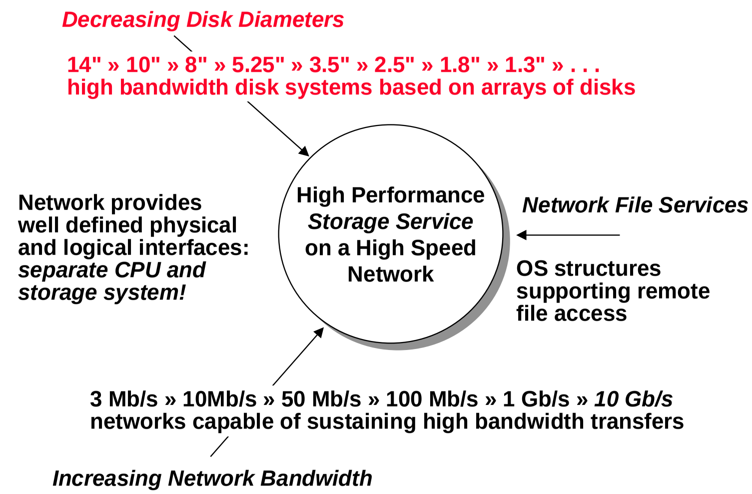
OS and I/O Systems Communication Requirements

° The Operating System must be able to prevent:
  • The user program from communicating with the I/O device directly

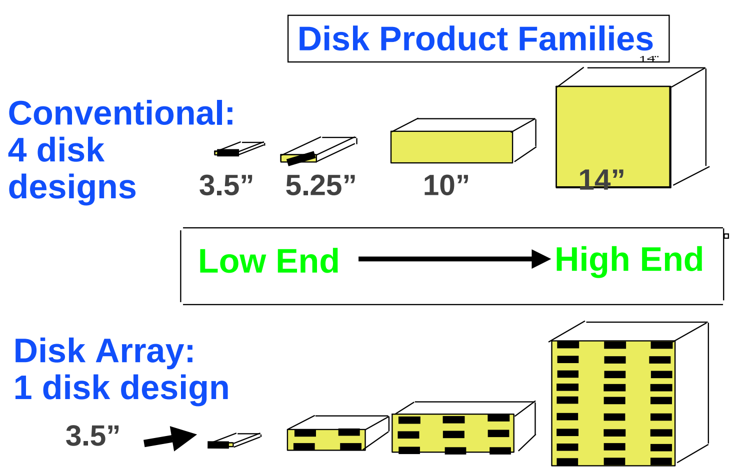
° If user programs could perform I/O directly:
  • Protection to the shared I/O resources could not be provided

° Three types of communication are required:
  • The OS must be able to give commands to the I/O devices
  • The I/O device must be able to notify the OS when the I/O device has completed an operation or has encountered an error
  • Data must be transferred between memory and an I/O device

Network Attached Storage



Manufacturing Advantages of Disk Arrays



Small # of Large Disks ⇒ Large # of Small Disks!

	IBM 3390 (K)	IBM 3.5" 0061	*x70*
Data Capacity	20 GBytes	320 MBytes	*23 GBytes*
Volume	97 cu. ft.	0.1 cu. ft.	*11 cu. ft.*
Power	3 KW	11 W	*1 KW*
Data Rate	15 MB/s	1.5 MB/s	*120 MB/s*
I/O Rate	600 I/Os/s	55 I/Os/s	*3900 IOs/s*
MTTF	250 KHrs	50 KHrs	*??? Hrs*
Cost	$250K	$2K	*$150K*

Disk Arrays have potential for
- large data and I/O rates
- high MB per cu. ft., high MB per KW
- reliability?

Array Reliability

- Reliability of N disks = Reliability of 1 Disk ÷ N

    50,000 Hours ÷ 70 disks = 700 hours

    Disk system MTTF: Drops from 6 years to 1 month!

- Arrays (without redundancy) too unreliable to be useful!

> **Hot spares support reconstruction in parallel with access: very high media availability can be achieved**

Redundant Arrays of Disks

- Files are "striped" across multiple spindles
- Redundancy yields high data availability

    Disks will fail

    Contents reconstructed from data redundantly stored in the array

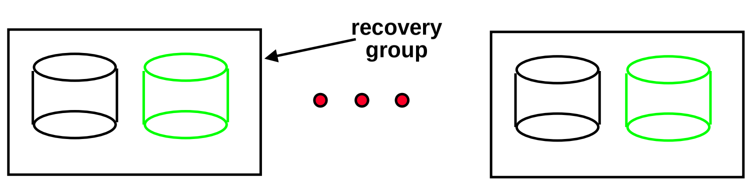
    → Capacity penalty to store it

    → Bandwidth penalty to update

Techniques:

- Mirroring/Shadowing (high capacity cost)
- Horizontal Hamming Codes (overkill)
- Parity & Reed-Solomon Codes
- Failure Prediction (no capacity overhead!)
  VaxSimPlus — Technique is controversial

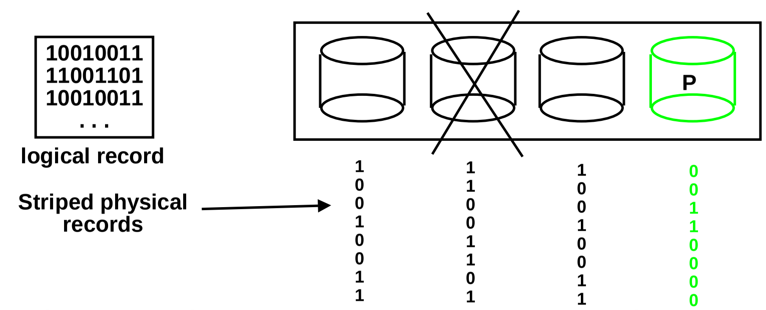
RAID 1: Disk Mirroring/Shadowing



- Each disk is fully duplicated onto its "shadow"
  Very high availability can be achieved
- Bandwidth sacrifice on write:
  Logical write = two physical writes
- Reads may be optimized
- Most expensive solution: 100% capacity overhead

Targeted for high I/O rate , high availability environments

RAID 3: Parity Disk



- Parity computed across recovery group to protect against hard disk failures
    - 33% capacity cost for parity in this configuration
    - wider arrays reduce capacity costs, decrease expected availability, increase reconstruction time
- Arms logically synchronized, spindles rotationally synchronized
  logically a single high capacity, high transfer rate disk

Targeted for high bandwidth applications: Scientific, Image Processing

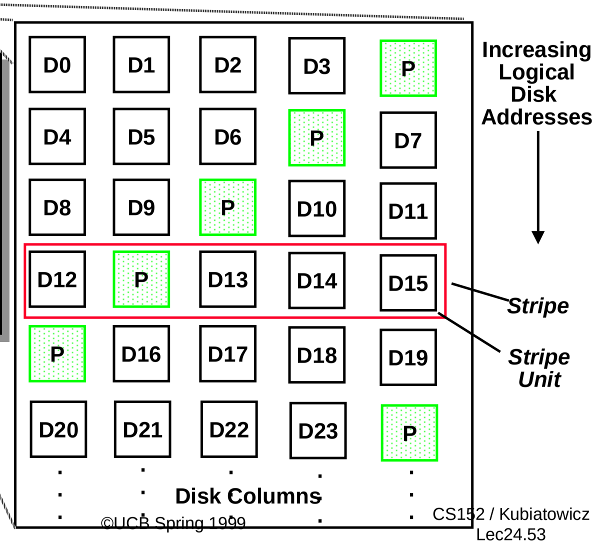
RAID 5+: High I/O Rate Parity

A logical write becomes four physical I/Os

Independent writes possible because of interleaved parity

Reed-Solomon Codes ("Q") for protection during reconstruction

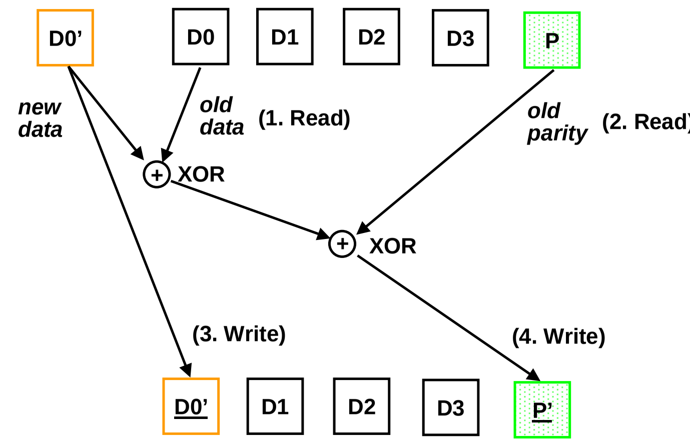
Targeted for mixed applications



Problems of Disk Arrays: Small Writes

RAID-5: Small Write Algorithm

1 Logical Write = 2 Physical Reads + 2 Physical Writes

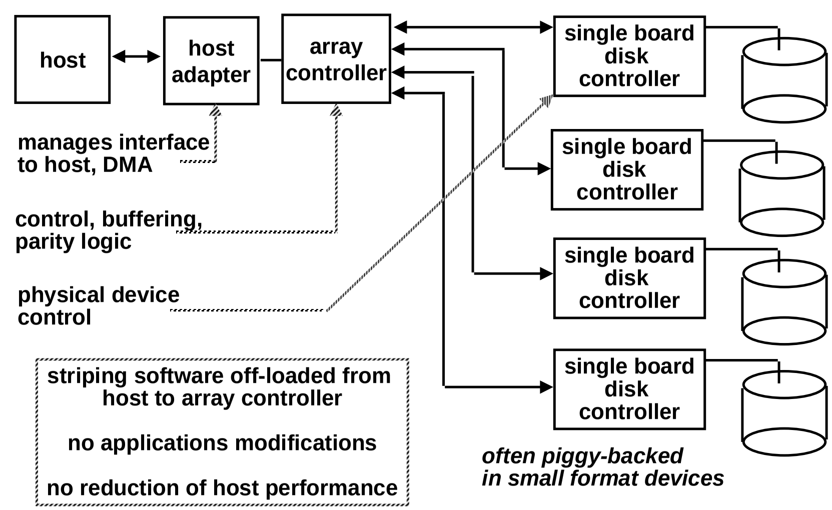


Hewlett-Packard (HP) AutoRAID

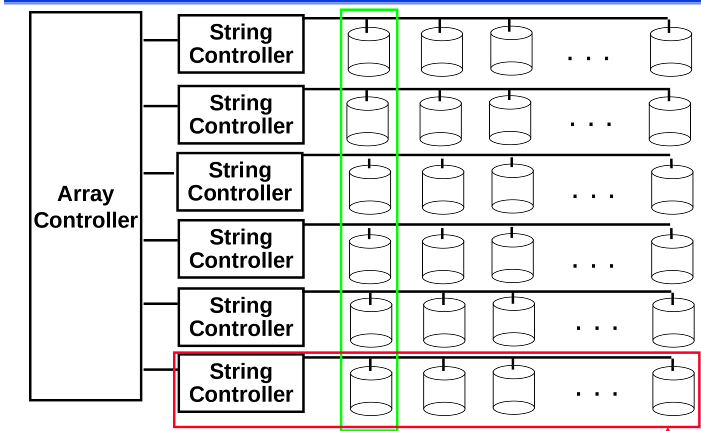
° HP has interesting solution which combines both mirroring and RAID level 5.

- Dynamically adapts disk storage
  - For recent or highly used data, uses mirroring
  - For less recently used data, uses RAID 5
- Gets speed of mirroring when it matters and density of RAID 5 on average

Subsystem Organization



System Availability: Orthogonal RAIDs

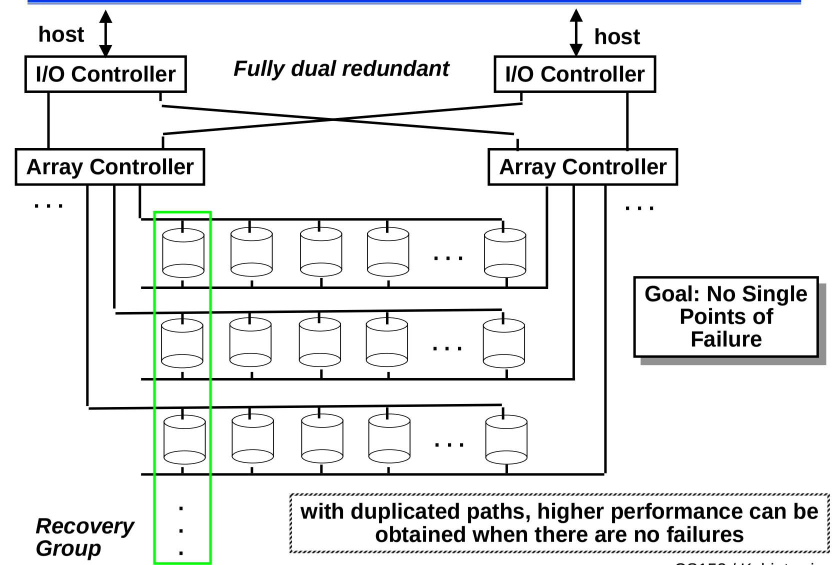


Data Recovery Group: **unit of data redundancy**

Redundant Support Components: **fans, power supplies, controller, cables**

End to End Data Integrity: **internal parity protected data paths**

System-Level Availability



Summary:

- **I/O performance limited by weakest link in chain between OS and device**
- **Queueing theory is important**
  - **100% utilization means very large latency**
  - **Remember, for M/M/1 queue (exponential source of requests/service)**
    - **queue size goes as u/(1-u)**
    - **latency goes as T_{ser}×u/(1-u)**
  - **For M/G/1 queue (more general server, exponential sources)**
    - **latency goes as m1(z) x u/(1-u) = T_{ser} x {1/2 x (1+C)} x u/(1-u)**
- **Three Components of Disk Access Time:**
  - **Seek Time: advertised to be 8 to 12 ms. May be lower in real life.**
  - **Rotational Latency: 4.1 ms at 7200 RPM and 8.3 ms at 3600 RPM**
  - **Transfer Time: 2 to 12 MB per second**
- **I/O device notifying the operating system:**
  - **Polling: it can waste a lot of processor time**
  - **I/O interrupt: similar to exception except it is asynchronous**
- **Delegating I/O responsibility from the CPU: DMA, or even IOP**