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/

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

### 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.
  - Model 3340 hard disk 1973 1.7 140 2,300
  - Model 3370 1979 7.7 140 2,300 MBytes

**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**

<table>
<thead>
<tr>
<th>Year</th>
<th>63 Mbit/sq. in</th>
<th>1450 Mbit/sq. in</th>
<th>3090 Mbit/sq. in</th>
</tr>
</thead>
<tbody>
<tr>
<td>1989</td>
<td>60,000 MBytes</td>
<td>2300 MBytes</td>
<td>8100 MBytes</td>
</tr>
<tr>
<td>1997</td>
<td>1450 Mbit/sq. in</td>
<td>2300 MBytes</td>
<td>8100 MBytes</td>
</tr>
</tbody>
</table>


---

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

- 9 v. 22 Mb/si
- 470 v. 3000 Mb/si


---

**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.

Coil for writing

Write Head

Read Head

High-Moment Yoke Materials

Interfacial Nanolayers

Exchange Biasing
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

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

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

Response time = Queue + Device Service time
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

° 0.7 sec 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

Response Time vs. Productivity

<table>
<thead>
<tr>
<th>Time</th>
<th>0.00</th>
<th>5.00</th>
<th>10.00</th>
<th>15.00</th>
</tr>
</thead>
<tbody>
<tr>
<td>graphics</td>
<td>1.0s</td>
<td>graphics</td>
<td>0.3s</td>
<td>conventional</td>
</tr>
</tbody>
</table>

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

Administrivia

° Pending schedule:
  - Monday 5/10 Last class (wrap up, evaluations, etc)
  - Tuesday 5/11 Oral reports: 10-12am and 2-4pm in 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 somewhat 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

° Scheduling:
  - Monday 5/10 Last class (wrap up, evaluations, etc)
  - Tuesday 5/11 Oral reports: 10-12am and 2-4pm in 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 somewhat 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: 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 favorably 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 interpretation substitutions and 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 dependency 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 both parallelism. The compiler translates source programs into control flow graphs based on analyzing flow of control, and then schedules the instructions within graphs according to flow arcs such that function applications, which have no control dependencies, are executed in parallel.
- A hierarchical coarse-control flow computation allows them to exploit coarse-grain parallelism in 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. However, we investigate the effect of modeling the application on the performance of the simulator. Our study indicates that the mapping scheme employed has to exhibit a strong locality effect in order to obtain high parallelism.
- 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 employed are particularly important. In addition, we observe that a large number of subsystem nodes allows more actions to be held simultaneously, but the communication overhead between a pair of nodes causes the overall execution time to increase.
- 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 interpretation substitutions and 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 dependency sets algorithms that partition control flow graphs into clusters.

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.
° 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 response 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:
  \( \lambda \) average number of arriving customers/second
  \( T_{ser} \) average time to service a customer (traditionally \( \mu = 1 / T_{ser} \))
  \( u \) server utilization (0..1): \( u = \lambda x T_{ser} \) (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: \( L_q = \lambda x T_q \)
  \( L_{sys} \) average length of system: \( L_{sys} = \lambda x T_{sys} \)
° Little’s Law: \( L_{sys} = \lambda x T_{sys} \)
  (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: \( C = \text{variance}/m1^2 \)
    - 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 $C = 1.5$ (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: $m_1(z)$
  - Not just $1/2 \times m_1$ because doesn’t capture variance
  - Can derive $m_1(z) = 1/2 \times m_1 \times (1 + C)$
  - Exponential $\Rightarrow C = 1 \Rightarrow m_1(z) = m_1$

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 $m_1(z)$
  - Chance server is busy $\Rightarrow u$; average delay is $u \times m_1(z)$
  - All customers in line must complete; each avg $T_{ser}$:
    - $T_q = u \times m_1(z) + L_q \times T_{ser}$
    - $T_q = u \times m_1(z) + \lambda \times T_q \times T_{ser}$
    - $T_q = u \times m_1(z) + u \times T_q$
    - $T_q = (1-u) \times m_1(z) \times u$
    - $T_q = m_1(z) \times u/(1-U) = T_{ser} \times (1/2 \times (1+C)) \times u/(1-u)$

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

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:
- $\lambda$ average number of arriving customers/second $= 10$
- $T_{ser}$ average time to service a customer $= 20$ ms (0.02s)
- $u$ server utilization (0..1): $u = \lambda \times T_{ser} = 10/0.02 = 0.2$
- $T_q$ average time/customer in queue $= T_{ser} \times u / (1-u)$
  - $= 20 \times 0.2/(1-0.2) = 20 \times 0.25 = 5$ ms (0.005s)
- $T_{sys}$ average time/customer in system: $T_{sys} = T_q + T_{ser} = 25$ ms
- $L_q$ average length of queue: $L_q = \lambda \times T_q$
  - $= 10/0.02 \times .005s = 0.05$ requests in queue
- $L_{sys}$ average # tasks in system: $L_{sys} = \lambda \times T_{sys} = 10/0.02 \times .025s = 0.25$
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

CPU

Single Memory & I/O Bus
No Separate I/O Instructions

ROM

RAM

I/O

Memory

Interface

Peripheral

Interface

Peripheral

CPU

$L$

L2 $

Memory Bus

I/O bus

Memory Bus Adaptor

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
  - \textit{Hiccup (!)}
  - lw $r2,0($r4)
  - lw $r3,4($r4)
  - add $r2,$r2,$r3
  - \textit{sw} 8($r4),$r2

- **Raise priority**
- **Reenable All Ints**
- **Save registers**

\textit{“Interrupt Handler”}

- **Restore registers**
- **Clear current Int**
- **Disable All Ints**
- **Restore priority**
- **RTI**

° **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**

- **Disable Network Intr**
  - subi $r4,$r1,#4
  - slli $r4,$r4,#2
  - lw $r2,0($r4)
  - addi $r3,$r0,#5
  - sw $r3,0($r1)
  - lw $r1,12($zero)
  - beq $r1,no_mess

- **Clear Network Intr**

° **Polling Point** (check device register)

- **Handler**

- **no_mess:**
  - ...
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

  CPU sends a starting address, direction, and length count to DMAC. Then issues “start”.

  CPU
  \[\text{Memory} \quad \text{DMAC} \quad \text{IOC} \quad \text{device}\]

  DMAC provides handshake signals for Peripheral Controller, and Memory Addresses and handshake signals for Memory.

Delegating I/O Responsibility from the CPU: IOP

1. Issues instruction to IOP
2. Device to/from memory transfers are controlled by the IOP directly.
3. IOP steals memory cycles.
4. IOP interrupts CPU when done

CPU
\[\text{IOP} \quad \text{Mem} \quad \text{I/O bus} \quad \text{D1} \quad \text{D2} \quad \ldots \quad \text{Dn}\]

CPU
\[\text{IOP} \quad \text{Device} \quad \text{Address}\]

I/O looks in memory for commands

OP
\[\text{Addr} \quad \text{Cnt} \quad \text{Other}\]

what to do
where to put data
how much
special requests

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 programs 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

Decreasing Disk Diameters

14” » 10” » 8” » 5.25” » 3.5” » 2.5” » 1.8” » 1.3” » . . .

High bandwidth disk systems based on arrays of disks

Network provides well defined physical and logical interfaces: separate CPU and storage system!

Increasing Network Bandwidth

3 Mb/s » 10Mb/s » 50 Mb/s » 100 Mb/s » 1 Gb/s » 10 Gb/s

networks capable of sustaining high bandwidth transfers

Manufacturing Advantages of Disk Arrays

Disk Product Families

Conventional:
- 4 disk designs
- 3.5”, 5.25”, 10”, 14”

Low End

High End

Disk Array:
- 1 disk design
- 3.5”

IBM 3390 (K)
- Data Capacity: 20 GBytes
- Volume: 97 cu. ft.
- Power: 3 KW
- Data Rate: 15 MB/s
- I/O Rate: 600 I/Os/s
- MTTF: 250 Khrs
- Cost: $250K

IBM 3.5” 0061 x70
- Data Capacity: 320 MBytes
- Volume: 0.1 cu. ft.
- Power: 11 W
- Data Rate: 1.5 MB/s
- I/O Rate: 55 I/Os/s
- MTTF: 50 Khrs
- Cost: $2K

x70
- Data Capacity: 23 GBytes
- Volume: 11 cu. ft.
- Power: 1 KW
- Data Rate: 120 MB/s
- I/O Rate: 3900 I/Os/s
- MTTF: ??? Hrs
- Cost: $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 ÷ $N$
  
  \[
  \frac{50,000 \text{ Hours}}{70 \text{ disks}} = 700 \text{ hours}
  \]

  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!)

  \[VaxSimPlus — \text{Technique is controversial}\]

### 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

  \[\text{Targeted for high I/O rate, high availability environments}\]

### 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

  \[\text{Targeted for high bandwidth applications: Scientific, Image Processing}\]
**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.

**Problems of Disk Arrays: Small Writes**

**RAID-5: Small Write Algorithm**

1 Logical Write = 2 Physical Reads + 2 Physical Writes

- (1. Read) old data
- (2. Read) old parity
- (3. Write) new data
- (4. Write) old parity

**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**

- Host manages interface to host, DMA.
- Array controller control, buffering, parity logic.
- Single board disk controller.
- Striping software off-loaded from host to array controller.
- No applications modifications.
- No reduction of host performance.

- Often piggy-backed in small format devices.
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

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 $T_{serv} \times u/(1-u)$
  - For M/G/1 queue (more general server, exponential sources)
    - latency goes as $m_1(z) \times u/(1-u) = T_{serv} \times (1/2 \times (1+C)) \times u/(1-u)$
* 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