SCSI overview

- **SCSI domain** consists of devices and an SDS
  - Devices: host adapters & SCSI controllers
  - *Service Delivery Subsystem* connects devices—e.g., SCSI bus

- **SCSI-2 bus (SDS) connects up to 8 devices**
  - Controllers can have > 1 “logical units” (LUNs)
  - Typically, controller built into disk and 1 LUN/target, but “bridge controllers” can manage multiple physical devices

- **Each device can assume role of *initiator* or *target***
  - Traditionally, host adapter was initiator, controller target
  - Now controllers act as initiators (e.g., COPY command)
  - Typical domain has 1 initiator, ≥ 1 targets
SCSI requests

- A request is a command from initiator to target
  - Once transmitted, target has control of bus
  - Target may disconnect from bus and later reconnect (very important for multiple targets or even multitasking)

- Commands contain the following:
  - Task identifier—initiator ID, target ID, LUN, tag
  - Command descriptor block—e.g., read 10 blocks at pos. \( N \)
  - Optional task attribute—SIMPLE, ORDERD, HEAD OF QUEUE
  - Optional: output/input buffer, sense data
  - Status byte—GOOD, CHECK CONDITION, INTERMEDIATE, ...
Executing SCSI commands

- Each LUN maintains a queue of *tasks*
  - Each task is DORMANT, BLOCKED, ENABLED, or ENDED
  - SIMPLE tasks are dormant until no ordered/head of queue
  - ORDERED tasks dormant until no HoQ/more recent ordered
  - HOQ tasks begin in enabled state

- Task management commands available to initiator
  - Abort/terminate task, Reset target, etc.

- Linked commands
  - Initiator can link commands, so no intervening tasks
  - E.g., could use to implement atomic read-modify-write
  - Intermediate commands return status byte INTERMEDIATE
SCSI exceptions and errors

- After error stop executing most SCSI commands
  - Target returns with CHECK CONDITION status
  - Initiator will eventually notice error
  - Must read specifics w. REQUEST SENSE

- Prevents unwanted commands from executing
  - E.g., initiator may not want to execute 2nd write if 1st fails

- Simplifies device implementation
  - Don’t need to remember more than one error condition

- Same mechanism used to notify of media changes
  - I.e., ejected tape, changed CD-ROM
Why disk arrays?

- CPUs improving faster than disks
  - disks will increasingly be bottleneck

- New applications (audio/video) require big files
  (motivation for XFS)

- Disk arrays - make one logical disk out of many physical disks

- More disks results in two benefits
  - Higher data transfer rates on large data accesses
  - Higher I/O rates on small data accesses
Reliability implications of arrays

- **JBOD array with 100 disks is 100 times more likely to fail!**
  - Each disk 200,000 hours MTBF $\rightarrow$ array 2,000 hours (3 months)
    [approximately – double counts two disks failing]

- **Use redundancy to improve reliability**
  - But makes writes slower, since redundant info must be updated
  - Raises issues of consistency in the face of power failures
Disk array basics

- Data striping - balances load across disks
- Redundancy - improve reliability
- Many different schemes, depending on
  - granularity of data interleaving
    - fine grained $\rightarrow$ high transfer rates for all requests
    - course grained $\rightarrow$ allow parallel small requests to different disks
  - method in which redundant information computed & distributed
RAID 0

- Nonredundant storage (JBOD - just a bunch of disks)
  - E.g., Stripe sequential 128K logical regions to different disk
- Offers best possible write performance (only one write per write)
- Offers best possible storage efficiency (no redundancy)
- Offers good read performance
- Use if speed more important that reliability (scientific computing)
RAID 1

- Mirrored storage – Each block stored on two disks
- Writing slower (twice as many operations)
- Storage efficiency 1/2 optimal
- Small reads better than other scheme
  - can read on disk with shortest seek
RAID 2

• Use Hamming codes, like ECC memory
  - Multiply data by generator matrix $G = (I \ A)$

$$
G = 
\begin{pmatrix}
1 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 1 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 1 & 1 & 0 \\
\end{pmatrix}
$$

$$
D = (d_1 \ d_2 \ d_3 \ d_4)
$$

$$
E = G \times D = 
\begin{pmatrix}
d_1 \\
d_2 \\
d_3 \\
d_4 \\
\end{pmatrix}
$$

$$
\begin{pmatrix}
d_1 + d_3 + d_4 \\
d_1 + d_2 + d_4 \\
d_1 + d_2 + d_3 \\
\end{pmatrix}
$$
hamming codes (continued)

- Decode by multiplying by $H = (A^T \ I)$

$$
H = \begin{pmatrix}
1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 \\
1 & 1 & 1 & 0 & 0 & 0 & 1
\end{pmatrix}
$$

$$
H \times E = (d_1 \ d_2 \ d_3 \ d_4 \ 0 \ 0 \ 0)
$$

- Can recover any two missing bits
- Can even recover from one incorrect bit!
  - If one extra bit is 1, it is wrong
  - If two extra bits are 1, $d_2$, $d_3$, or $d_4$ is wrong
  - If all 3 extra bits are 1, $d_1$ is wrong
Properties of RAID 2

- Small reads about like RAID 0
  - Though more contention for data disks
- Writes must update multiple parity disks
- Storage more efficient than RAID 1
  (uses more than half of disks)
- Recovers from errors (RAID 1 assumes fail-stop)
  - Is this overkill?
  - Most disks are mostly fail-stop
RAID 3

- Bit interleaved parity
- Assume fail stop disks, add one parity disk

\[ D = \begin{pmatrix} d_1 & d_2 & d_3 & d_4 \end{pmatrix} \]

\[ E = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_1 + d_2 + d_3 + d_4 \end{pmatrix} \]

- Any read touches all data disks
- Any write touches all data disks plus parity disk
RAID 4

- Block-interleaved parity
- Interleave data in blocks instead of bits
- Reads smaller than striping unit can access only one disk
- Writes must update data and compute and update parity block
  - Small writes require two reads plus two writes
  - Heavy contention for parity disk (all writes touch it)
RAID 5

- Block-interleaved *distributed* parity
  - Distribute parity uniformly over all the disks
  - Want to access disks sequentially when sequentially accessing logical blocks:
    
    \[
    \begin{array}{cccccc}
    0 & 1 & 2 & 3 & p_0 \\
    5 & 6 & 7 & p_1 & 4 \\
    10 & 11 & p_2 & 8 & 9 \\
    15 & p_3 & 12 & 13 & 14 \\
    p_4 & 16 & 17 & 18 & 19 \\
    \end{array}
    \]

- Better load balancing than RAID 4
- Small writes still require read-modify-write
RAID 6

- P+Q Redundancy (rarely implemented)
  - Have two parity blocks instead of one
  - With Reed-Solomon codes, can lose any two blocks

- Now must read-modify-write two parity blocks plus data
### RAID Summary

<table>
<thead>
<tr>
<th>RAID Level</th>
<th>SmRd</th>
<th>SmWr</th>
<th>BigRd</th>
<th>BigWr</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1/2</td>
<td>1/2</td>
<td>1</td>
<td>1</td>
<td>1/2</td>
</tr>
<tr>
<td>3</td>
<td>1/G</td>
<td>1/G</td>
<td>(G-1)/G</td>
<td>(G-1)/G</td>
<td>(G-1)/G</td>
</tr>
<tr>
<td>5</td>
<td>max(1/G, 1/4)</td>
<td>1</td>
<td>(G-1)/G</td>
<td>(G-1)/G</td>
<td>(G-1)/G</td>
</tr>
<tr>
<td>6</td>
<td>max(1/G, 1/6)</td>
<td>1</td>
<td>(G-2)/G</td>
<td>(G-2)/G</td>
<td>(G-2)/G</td>
</tr>
</tbody>
</table>

- RAID 4 is strictly inferior to RAID 5
- RAID 2 inferior to RAID 5 for fail-stop disks
- RAID 1 is just RAID 5 with parity group size G=2
- RAID 3 is just like RAID 5 with a very small stripe unit
When/how do disks fail?

- Disks can fail very early in their lifetimes (manufacturing errors)
- Also tend to fail late in their lifetimes (when disk wears out)
- Systematic manufacturing defect can make entire batch of disks fail early
  - Beware disks with consecutive serial numbers!
- Environmental factors can kill a bunch of disks (air conditioning failure)
- Disks can fail when a bad block is read Bad block may exist for a while before being detected
Dealing with failures

• Basic idea:
  - Add new disk
  - Reconstruct failed disk’s state on new disk

• Must store metadata information during recovery
  - Which disks are failed?
  - How much of failed disk has been reconstructed?

• System crashes become very serious in conjunction with disk failure
  - Parity may be inconsistent (particularly bad for P+Q)
  - You could lose a block other than the one you were writing
  - MUST log in NVRAM enough info to recover parity
  - Makes software-only implementation of RAID risky
Maximizing availability

- **Want to keep operating after failure**

- **Demand reconstruction**
  - Assumes spare disk immediately (or already) installed
  - Reconstruct blocks as accessed
  - Background thread reconstructs all blocks

- **Parity sparing**
  - Replaces parity block with reconstructed data block
  - Need extra metadata to keep track of this
Unrecoverable RAID failures

- Double disk failures (or triple, if P+Q redundancy)
- System crash followed by disk failure
- Disk failure, then read and discover bad block during reconstruction
RAID improvements

• Parity logging
  - Log difference of old and new parity blocks
  - Delay updating actual parity
  - Further writes may save you from a read

• Declustered parity
  - Many parity groups, spread over many disks

• Parity sparing
  - Use spare disks to improve performance by spreading load
Tuning RAID

- **What is optimal size of data stripe in RAID 0 disk array?**
  \[ \sqrt{\frac{PX(L - 1)Z}{N}} \]
  - \( P \) - average positioning time
  - \( X \) - disk transfer rate
  - \( L \) - concurrency of workload
  - \( Z \) - request size
  - \( N \) - size of array in disks

- **What about in RAID 5?**
  - Reads - similar to RAID 0
  - Writes - optimal is a factor of 4 smaller than for reads (for 16 disks)
  - Seems to vary WITH #disks, while reads vary inversely!

- **Conclusion:** Very workload dependent!