RAID

Redundant Array of Inexpensive (Independent) Disks
- Use multiple smaller disks (c.f. one large disk)
- Parallelism improves performance
- Plus extra disk(s) for redundant data storage

Provides fault tolerant storage system
- Especially if failed disks can be “hot swapped”

RAID 0
- No redundancy (“AID”?)
  - Just stripe data over multiple disks
  - But it does improve performance

"allocation of logically sequential data blocks to separate disks to allow higher performance than a single disk can deliver"
RAID 1 & 2

RAID 1: Mirroring
- N + N disks, replicate data
  - Write data to both data disk and mirror disk
  - On disk failure, read from mirror

RAID 2: Error correcting code (ECC)
- N + E disks (e.g., 10 + 4)
- Split data at bit level across N disks
- Generate E-bit ECC
- Too complex, not used in practice (for disks, but…)
RAID 3: Bit-Interleaved Parity

N + 1 disks
- Data striped across N disks at byte level
- Redundant disk stores parity
- Read access
  - Read all disks
- Write access
  - Generate new parity and update all disks
- On failure
  - Use parity to reconstruct missing data

Not widely used
RAID 4: Block-Interleaved Parity

N + 1 disks
- Data striped across N disks at block level
- Redundant disk stores parity for a group of blocks
- Read access
  - Read only the disk holding the required block
- Write access
  - Just read disk containing modified block, and parity disk
  - Calculate new parity, update data disk and parity disk
- On failure
  - Use parity to reconstruct missing data

Not widely used
RAID 3 vs RAID 4

New Data 1. Read 2. Read 3. Read

- D0' - D0 - D1 - D2 - D3 - P
- XOR

4. Write 5. Write

New Data 1. Read

- D0' - D0 - D1 - D2 - D3 - P
- XOR

2. Read

3. Write 4. Write
RAID 5: Distributed Parity

N + 1 disks
- Like RAID 4, but parity blocks distributed across disks
  - Avoids parity disk being a bottleneck

Widely used
RAID 6: P + Q Redundancy

N + 2 disks
- Like RAID 5, but two lots of parity
- Greater fault tolerance through more redundancy
RAID Summary

RAID can improve performance and availability
- High availability requires hot swapping

Assumes independent disk failures
- Too bad if the building burns down!

See “Hard Disk Performance, Quality and Reliability”
**Error Detection**

*Error detecting codes* enable the detection of errors in data, but do not determine the precise location of the error.

- store a few extra state bits per data word to indicate a necessary condition for the data to be correct
- if data state does not conform to the state bits, then something is wrong
- e.g., represent the correct *parity* (# of 1’s) of the data word
- 1-bit parity codes fail if 2 bits are wrong…

```
1011 1101 0001 0000 1101 0000 1111 0010 1
```

A 1-bit parity code is a *distance-2 code*, in the sense that at least 2 bits must be changed (among the data and parity bits) produce an incorrect but legal pattern. In other words, any two legal patterns are separated by a distance of at least 2.
Two common schemes (for single parity bits):
- even parity 0 parity bit if data contains an even number of 1's
- odd parity 0 parity bit if data contains an odd number of 1's

We will apply an even-parity scheme.

```
1011 1101 0001 0000 1101 0000 1111 0010 1
```

The parity bit could be stored at any fixed location with respect to the corresponding data bits.

Upon receipt of data and parity bit(s), a check is made to see whether or not they correspond.

Cannot detect errors involving two bit-flips (or any even number of bit-flips).
Error correcting codes provide sufficient information to locate and correct some data errors.

- must use more bits for state representation, e.g. 6 bits for every 32-bit data word
- may indicate the existence of errors if up to $k$ bits are wrong
- may indicate how to correct the error if up to $l$ bits are wrong, where $l < k$
- $c$ code bits and $n$ data bits $\Rightarrow 2^c \geq n + c + 1$

We must have at least a distance-3 code to accomplish this.

Given such a code, if we have a data word + error code sequence $X$ that has 1 incorrect bit, then there will be a unique valid data word + error code sequence $Y$ that is a distance of 1 from $X$, and we can correct the error by replacing $X$ with $Y$. 
Error Correction

A distance-3 code is also known as a single-error correcting, double-error detecting or SECDED code.

If X has 2 incorrect bits, then we will replace X with an incorrect (but valid) sequence.

We cannot both detect 2-bit errors and correct 1-bit errors with a distance-3 code.

But, hopefully flipped bits will be a rare occurrence and so sequences with two or more flipped bits will have a negligible probability.
Hamming Codes

Richard Hamming described a method for generating minimum-length error-correcting codes. Here is the (7,4) Hamming code for 4-bit words:

<table>
<thead>
<tr>
<th>Data bits</th>
<th>Check bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>000</td>
</tr>
<tr>
<td>0001</td>
<td>011</td>
</tr>
<tr>
<td>0010</td>
<td>101</td>
</tr>
<tr>
<td>0011</td>
<td>110</td>
</tr>
<tr>
<td>0100</td>
<td>110</td>
</tr>
<tr>
<td>0101</td>
<td>101</td>
</tr>
<tr>
<td>0110</td>
<td>011</td>
</tr>
<tr>
<td>0111</td>
<td>000</td>
</tr>
<tr>
<td>1000</td>
<td>111</td>
</tr>
<tr>
<td>1001</td>
<td>100</td>
</tr>
<tr>
<td>1010</td>
<td>010</td>
</tr>
<tr>
<td>1011</td>
<td>001</td>
</tr>
<tr>
<td>1100</td>
<td>001</td>
</tr>
<tr>
<td>1101</td>
<td>010</td>
</tr>
<tr>
<td>1110</td>
<td>100</td>
</tr>
<tr>
<td>1111</td>
<td>111</td>
</tr>
</tbody>
</table>

Say we had the data word 0100 and check bits 011.

The two valid data words that match that check bit pattern would be 0001 and 0110.

The latter would correspond to a single-bit error in the data word, so we would choose that as the correction.

Note that if the error was in the check bits, we'd have to assume the data word was correct (or else we have an uncorrectable 2-bit error or worse). In that case, the check bits would have to be 1 bit distance from 110, which they are not.
Hamming Code Details

Hamming codes use extra parity bits, each reflecting the correct parity for a different subset of the bits of the code word. Parity bits are stored in positions corresponding to powers of 2 (positions 1, 2, 4, 8, etc.). The encoded data bits are stored in the remaining positions.

The parity bits are defined as follows:
- position 1: check 1 bit, skip 1 bit, check 1 bit, skip 1 bit, …
- position 2: check 2 bits, skip 2 bits, …
  ...
- position $2^k$: check $2^k$ bits, skip $2^k$ bits, …

Consider the data byte: 10011010

Expand to allow room for the parity bits: _ _ 1 _ 001_1010

Now compute the parity bits as defined above…
We have the expanded sequence: _ _ 1 _ 0 0 1 _ 1 0 1 0

The parity bit in position 1 (first bit) would depend on the parity of the bits in positions 1, 3, 5, 7, etc:

_ _ 1 _ 0 0 1 _ 1 0 1 0

Those bits have even parity, so we have: 0 _ 1 _ 0 0 1 _ 1 0 1 0

The parity bit in position 2 would depend on bits in positions 2, 3, 6, 7, etc:

0 _ 1 _ 0 0 1 _ 1 0 1 0

Those bits have odd parity, so we have: 0 1 1 _ 0 0 1 _ 1 0 1 0

Continuing, we obtain the encoded string: 0 1 1 0 0 1 0 1 0 1 0
Hamming Code Correction

Suppose we receive the string: \(0 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0\)

How can we determine whether it's correct? Check the parity bits and see which, if any, are incorrect. If they are all correct, we must assume the string is correct. Of course, it might contain so many errors that we can't even detect their occurrence, but in that case we have a communication channel that's so noisy that we cannot use it reliably.

Checking the parity bits above:

```
0  1  1  1  0  0  1  0  1  1  1  0
```

OK  
WRONG
WRONG
OK
WRONG

So, what does that tell us, aside from that the string is incorrect? Well, if we assume there's no more than one incorrect bit, we can say that because the incorrect parity bits are in positions 2 and 8, the incorrect bit must be in position 10.