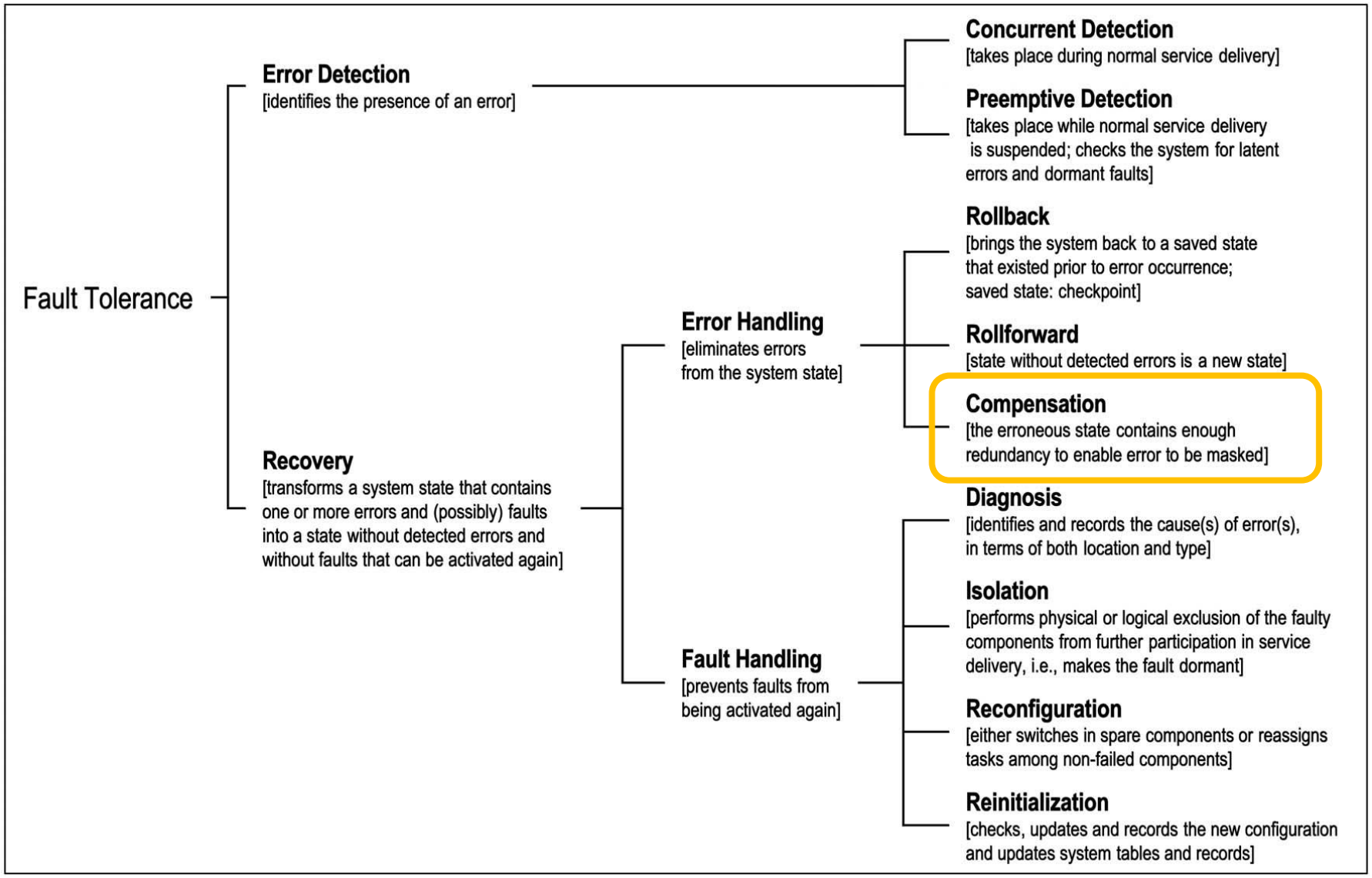
# Organisation of fault tolerance




From [Avizienis et al., 2004]

# Error compensation

fault masking

Hardware faults
- **Hardware components fail independently**
- replicas of the hw component

Software faults
- **Replicas of the same sw do not fail independently**
- Versions of the sw that implement the same function via separate designs and implementations(**design diversity**)

# Faults in the software

All software defects (faults)
are design faults


- Faults in the software *design phase*
  of the software lifecycles

- Faults in the implementation of the software

- Erroneous outputs from a test case

Design faults are not introduced maliciously.
Developers build the software using techniques aimed to produce the right product.

# Faults in the software

## Design faults:

hard to visualize, classify, detect, and correct

Consider the issue of determining the faults in a large software system

- Any phase of the software lifecycle (Requirements analysis, Requirements specification, Design, Implementation, Verification, Deployment), could have introduced faults

- The fault could remain dormant for extended periods, if the sw component affected by the fault is not on the execution path

- Complete elimination of design faults is not possible, some faults are expected to remain in the system

# Faults in the software

## Design faults:

closely related to human factors and the design process, of which we don't have a solid understanding

only some type of inputs will exercise that fault to cause failures. Number of failures depend on how often these inputs exercise the sw flaw

apparent reliability of a piece of software is correlated to how frequently design faults are exercised as opposed to number of design faults present

## Versions of the software

a simple duplication and comparison procedure or  a N-modular redundancy with voting (e.g., TMR) will not detect software faults if the replicated software modules are identical

Independent generation of N >= 2 functionally equivalent programs, called *versions*, **from the same initial requirements**.

# SW Fault tolerance techniques

## Design  diversity technique

if two or more software are built to provide the same functionality but with different designs, they might not all fail on the same input

Technique:

- use independent development teams which do not communicate with each other
- using different sw development tools  (like different programming languages (compilers), linkers and loaders …
- Using different sw design techniques: functional decomposition, object oriented ….

# SW Fault tolerance techniques

Due to the large cost of developing software, most of the software dependability effort has focused on

**fault prevention techniques and testing strategies**

## Multi-version approaches

mainly used in safety-critical systems (due to cost)

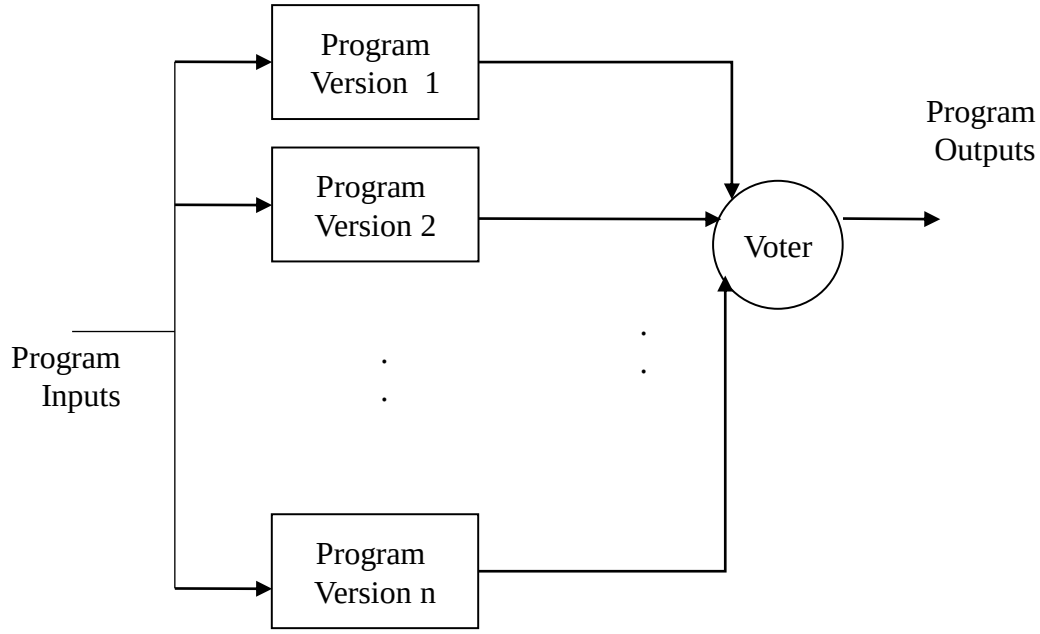
versions can be organised into different software systems architectures

# Passive SW Fault tolerance techniques: N-version systems

-independently developed versions of the sw

- the Voter votes on the results produced by the versions

- there is no delay or interruption of the service



# Passive SW Fault tolerance techniques: N-version programming

**BUT**

- all the versions need to be executed along with the Voter -> considerable hw resources needed

- a CLOCK is needed to synchronise the execution of the versions and the Voter, or the Voter must implement some sort of communication protocol to wait until all versions complete their processing or recognize the versions that do not complete (e.g., omission failure)

- NUMERIC ISSUES. the value supplied by the versions can be correct but differ because of rounding errors or similar numeric issues, e.g., Floating point output values will not be bit-for-bit the same

- the degree of differences is specific for each system

**Different values for numeric issues must be distinguished by erroneous values**

# Passive SW Fault tolerance techniques: N-version programming

ALGORITHMIC ISSUES. Two correct output could be different

Example: roots of a polinomial. Assume only one is requested

Two versions compute the same values, but they can be found in different ordering.

The two versions, retun different values

Example: navigation system (automotive). Heuristics are used to compute routes.

Routes computed by two versions can be different (both correct)

# Passive SW Fault tolerance techniques: N-version programming

## Voting

based on different techniques, specific for each system

**- Choose the value supplied by the majority (if possible)**

**- Choose the median value**

**- Choose the average of all values**

……..

→ how can be erroneous values ignored?

*value sufficiently different from the other values*

The result of previous voting changes if a version is considered faulty

# Active SW Fault tolerance techniques: N-self-checking programming

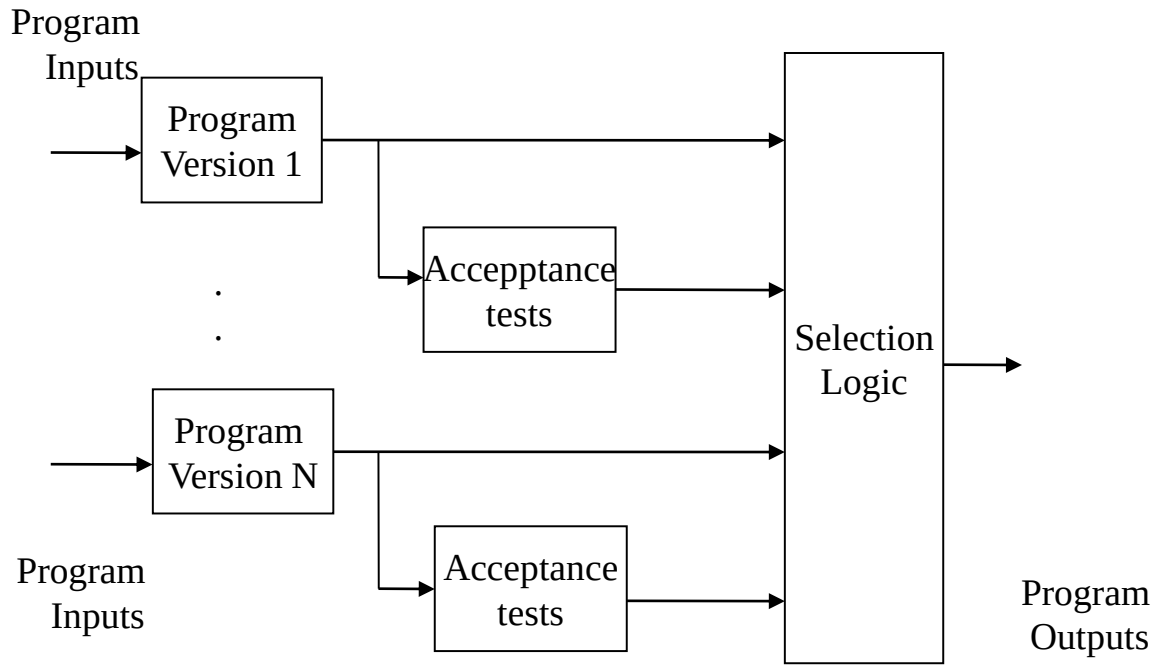
## based on acceptance tests rather than comparison with equivalent versions

N versions are written

- each version is running simultaneously and includes its acceptance tests

The selection logic chooses the results from one of the programs that passes the acceptance tests

Tolerates N-1 faults (independent faults)

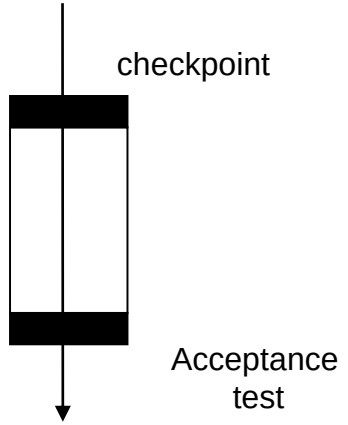


# Hybrid SW Fault tolerance techniques: Recovery-block

## based on an one acceptance test and a single alternate is run at a time

Basic structure:

**Ensure T**
**By P**
**else by Q**
**else error**




The versions are referred to as *alternates.*

Alternates are executed in series until one of them completes the computation successfully

Generally error detection mechanism built into the alternate and a final external error detection mechanism is present , the Acceptance test.

Accettability of the result is decided by an acceptance test **T.**

Primary alternate **P**, secondary alternates **Q**

# Hybrid SW Fault tolerance techniques: Recovery-block

Basic structure:

**Ensure T**
**By P**
**else by Q**
**else error**

**The recovery block-store the system state (checkpoint)**
(e.g., variables global to the block which are altered within the block)

If the primary alternate passes the acceptance test, the recovery-block is exited

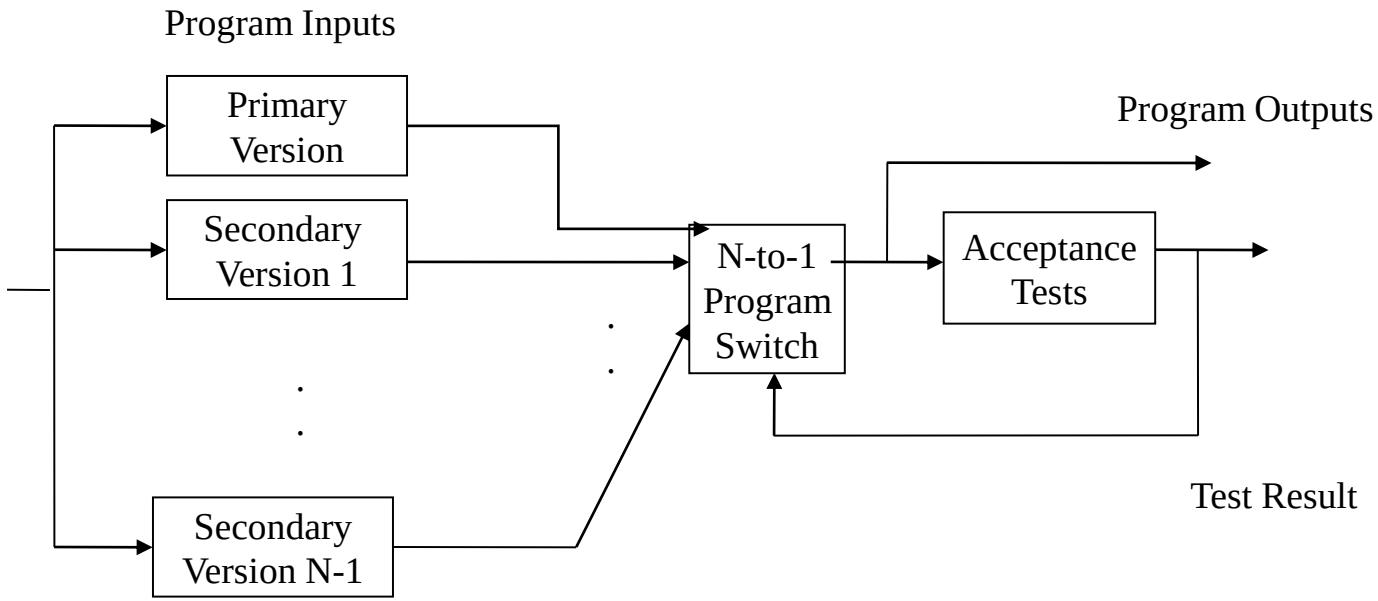
**If the primary alternate fails, the recovery-block restores the system state and executes the next alternate, which becomes the primary alternate.**
If no more alternatives exist, an error is reported.

# Hybrid SW Fault tolerance techniques: Recovery-block

The execution time of a recovery block is unpredictable.

- releases the programmer from determining which variables should be checkpointed and when

- linguistic structure for recovery blocks requires a suitable mechanism for providing automatic backward error recovery



# Example: Magnetic disk

**Disk controller** – interfaces between the computer system and the disk drive hardware

- accepts high-level commands to read or write a sector (block)
- Moves the disk arm to the right track and actually reads or writes the data

## Read failure

To deal with read failure, computes and attaches **checksums** to each sector to verify that data is read back correctly

If data is corrupted, with very high probability stored checksum won't match recomputed checksum

## Write failure

Ensure successful writing by reading back sector after writing it

# Checksumming

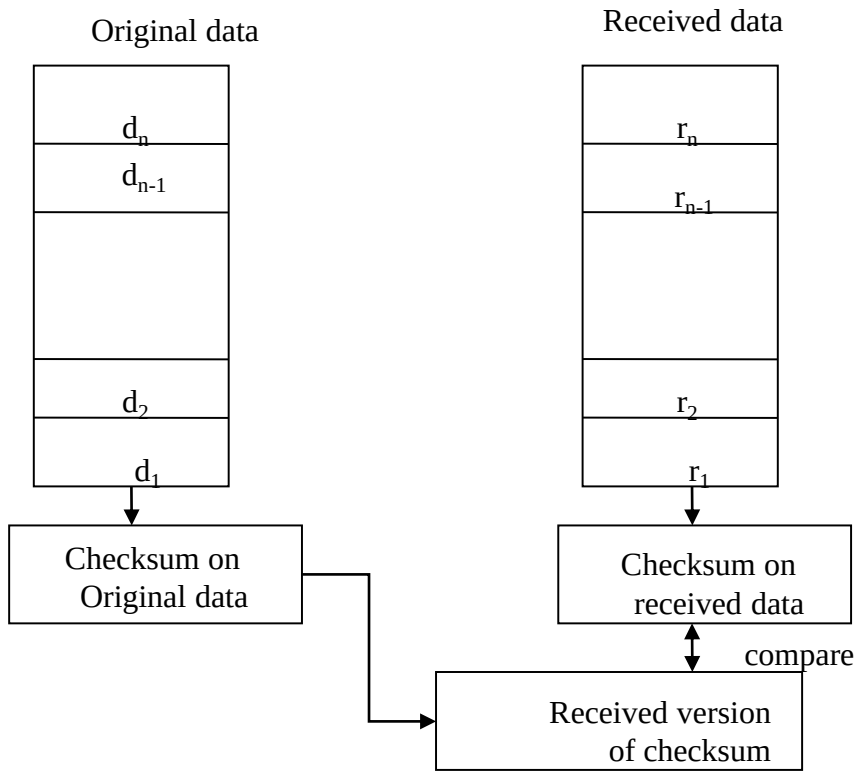
- applied to large block of data in memories
- coverage: single fault

checksum for a block of n words is formed by adding together all of the words in the block modulo-k, where k is arbitrary (one of the least expensive method)

- the checksum is stored with the data block

- when blocks of data are transferred (e.g. data transfer between mass-storage device) the sum is recalculated and compared with the checksum

- checksum is basically the sum of the original data




Code word = block + checksum

# Checksumming Code

- Disadvantages

  - if any word in the block is changed, the checksum must also be modified at the same time

  - allow error detection, no error location: the detected fault could be in the block of s words, the stored checksum or the checking circuitry

  - single point of failures for the comparison and encoder/detector element

- Different methods differ for how summation is executed

# RAID technology

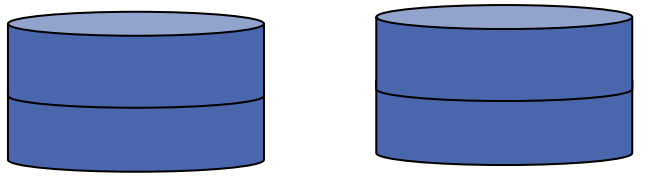
RAID (Redundant Arrays of Independent Disks) technology

disk organization techniques that manage a large numbers of disks, providing a view of a single disk of high capacity and high speed by using multiple disks in parallel, and high reliability by storing data redundantly

# RAID

Redundant information stored on multiple disks to recover from failures

- Mirroring




Every write is carried out on both disks

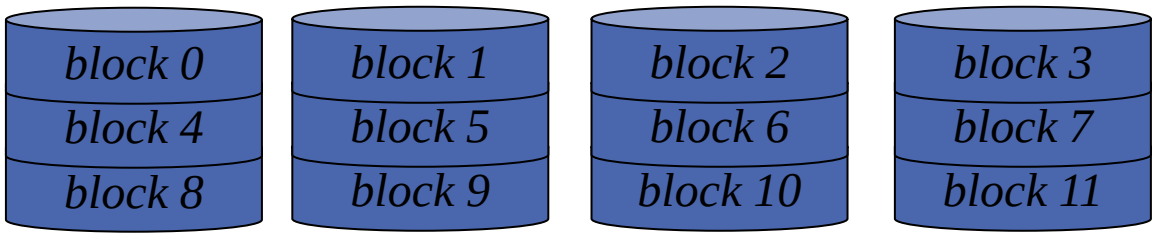
Reads can take place from either disk

If one disk in a pair fails, data still available in the other

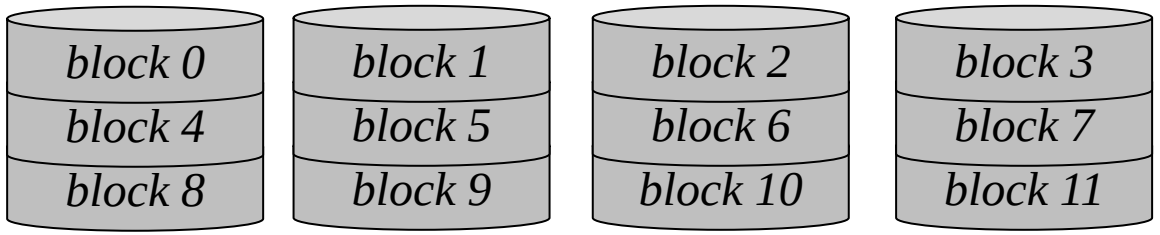
# RAID

- Block-level striping + mirrored disks

  Requests for different blocks can run in parallel if the blocks reside on different disks




*Disks*

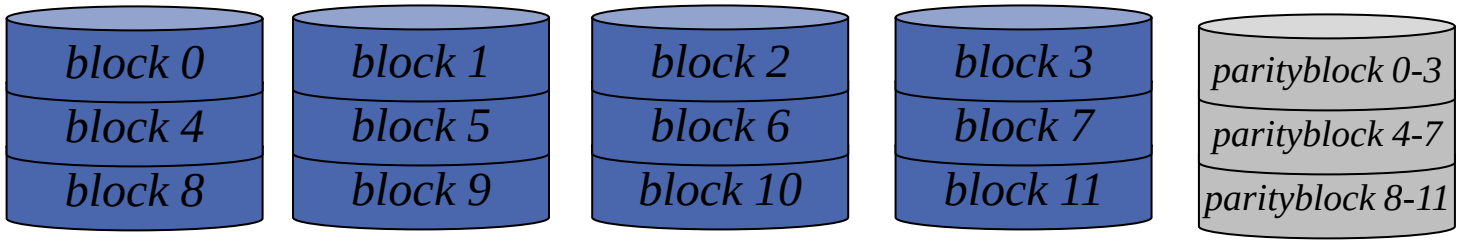



*Mirrored Disks*

# RAID

- Coding: Block-Interleaved Parity

  Block-level striping

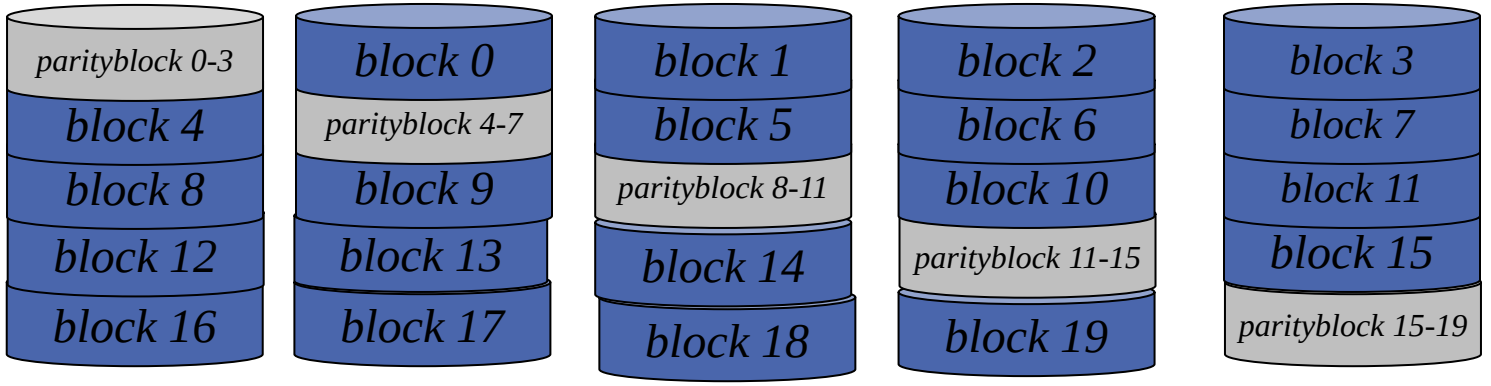



Parity block on a different disk for detection of bit errors
ALL 1-BIT ERRORS ARE DETECTED (Error Detection Code)

Before writing a block, parity data must be computed. Parity block becomes a bottleneck for independent block writes since every block write also writes to parity disk

# RAID

- Coding: Block-Interleaved Distributed Parity




Partition data and parity among all *N* + 1 disks, rather than storing data in *N* disks and parity in 1 disk

For each set of N logical blocks, one of the disks store the parity and the other N disks store the blocks

# Hamming Code (I)

## Parity bits spread through all the data word

| Bit position | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Encoded data bits | | p1 | p2 | d1 | p4 | d2 | d3 | d4 | p8 | d5 | d6 | d7 | d8 | d9 | d10 | d11 | p16 | d12 | d13 | d14 | d15 | |
| Parity bit coverage | p1 | X | | X | | X | | X | | X | | X | | X | | X | | X | | X | | |
| | p2 | | X | X | | | X | X | | | X | X | | | X | X | | | X | X | | ... |
| | p4 | | | | X | X | X | X | | | | | X | X | X | X | | | | | X | |
| | p8 | | | | | | | | X | X | X | X | X | X | X | X | | | | | | |
| | p16 | | | | | | | | | | | | | | | | X | X | X | X | X | |

Taken from: http://en.wikipedia.org/wiki/Hamming_code#Hamming_codes

**Parity bits**
*all bit positions that are powers of two : 1, 2, 4, 8, etc.*

**Data bits**
*all other bit positions*

(number the bit positions starting from 1: bit 1, 2, 3, etc..)

Parity bit pj covers all bits whose position has the j least significant bit equal to 1

Each data bit is included in a unique set of 2 or more parity bits, as determined by the binary form of its bit position

# Hamming code (II)

| Bit position | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Encoded data bits | | p1 | p2 | d1 | p4 | d2 | d3 | d4 | p8 | d5 | d6 | d7 | d8 | d9 | d10 | d11 | p16 | d12 | d13 | d14 | d15 | |
| Parity bit coverage | p1 | X | | X | | X | | X | | X | | X | | X | | X | | X | | X | | |
| | p2 | | X | X | | | X | X | | | X | X | | | X | X | | | X | X | | ... |
| | p4 | | | | X | X | X | X | | | | | X | X | X | X | | | | | X | |
| | p8 | | | | | | | | X | X | X | X | X | X | X | X | | | | | | |
| | p16 | | | | | | | | | | | | | | | | X | X | X | X | X | |

Taken from: http://en.wikipedia.org/wiki/Hamming_code#Hamming_codes

Parity bit p1 covers all bit positions which have the least significant bit set:

bit 1 (the parity bit itself), 3, 5, 7, 9, etc.

Parity bit p2 covers all bit positions which have the second least significant bit set:

bit 2 (the parity bit itself), 3, 6, 7, 10, 11, etc.

Parity bit 4 covers all bit positions which have the third least significant bit set:

bits 4–7, 12–15, 20–23, etc.

Parity bit 8 covers all bit positions which have the fourth least significant bit set:

bits 8–15, 24–31, 40–47, etc.

# Hamming code (III)

| Parity bits | Total bits | Data bits | Name | Rate |
|---|---|---|---|---|
| 2 | 3 | 1 | Hamming(3,1) (Triple repetition code) | 1/3 ≈ 0.333 |
| 3 | 7 | 4 | Hamming(7,4) | 4/7 ≈ 0.571 |
| 4 | 15 | 11 | Hamming(15,11) | 11/15 ≈ 0.733 |
| 5 | 31 | 26 | Hamming(31,26) | 26/31 ≈ 0.839 |
| ... | | | | |
| $m$ | $2^m - 1$ | $2^m - m - 1$ | Hamming$(2^m - 1, 2^m - m - 1)$ | $(2^m - m - 1)/(2^m - 1)$ |

Taken from: http://en.wikipedia.org/wiki/Hamming_code#Hamming_codes

Overlap of control bits: a data bit is controlled by more than one parity bit

Minimum Hamming distance: 3

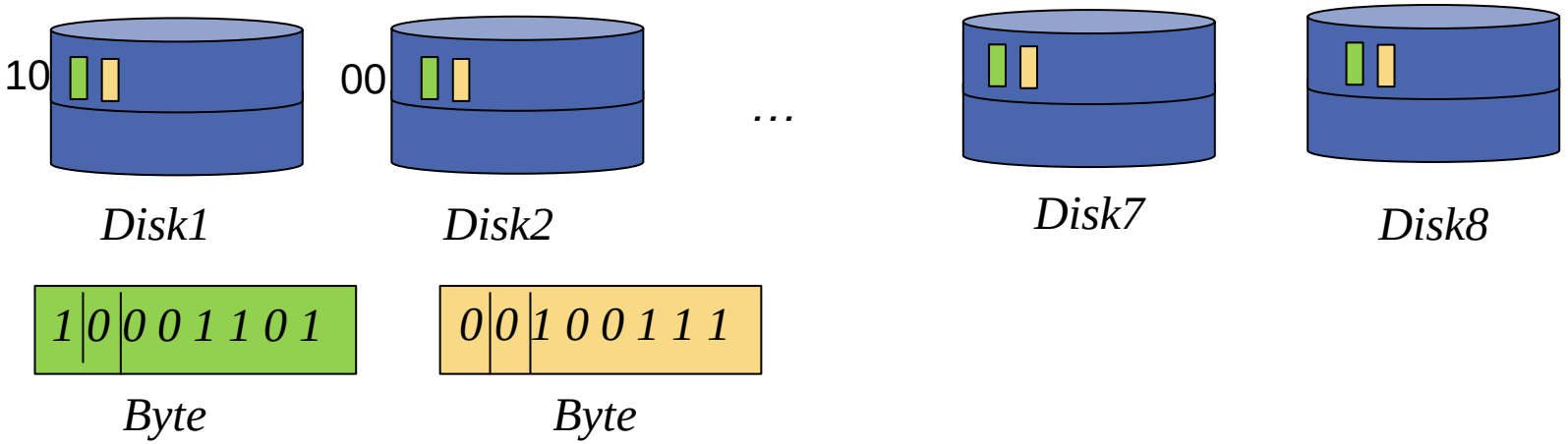
Double-error detection code
Single-error correction code

➡ SEC-DED code

# RAID

- Coding: Hamming code

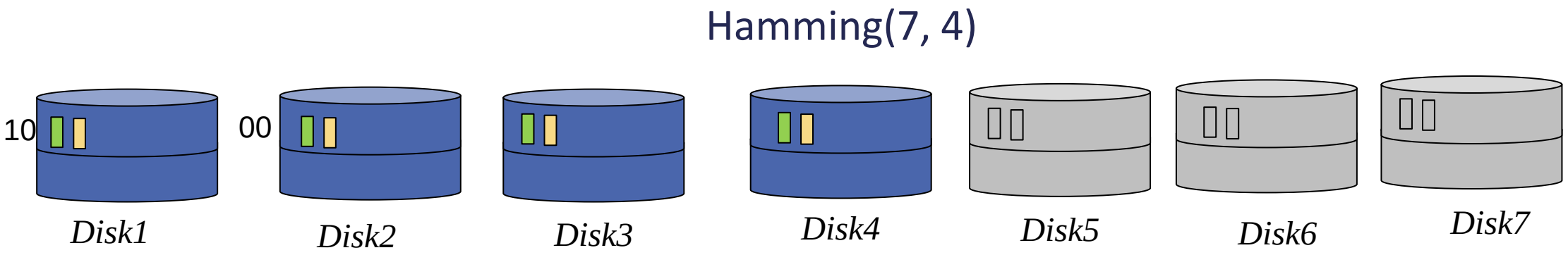
Bit-level striping




N bits written across n disks, one per disk.

# RAID

- Coding: Hamming code




N bits written across n disks, one per disk. The additional bit needed for error correcting codes are written across a set of additional disks  one bit at a time

A disk read failure can be masked and a complete disk replaced if necessary, without stopping the system

# Notes

- Fault tolerance  relies on the **independency of redundancies** with respect to the process of fault creation and activations

- **Fault masking** will conceal a possibly progressive and eventually fatal loss of protective redundancy

- Practical implementations of masking generally involve error detection (and possibly fault handling), leading to **masking and error detection and recovery**

# Notes

- When tolerance hw faults is foreseen, the replicas may be identical, based on the assumption that hardware components fail **independently**

- When tolerance to SW faults is foreseen, replicas have to provide identical service through separate designs  and implementation (through **design diversity**)

- Replicas al commonly named «*channels*»