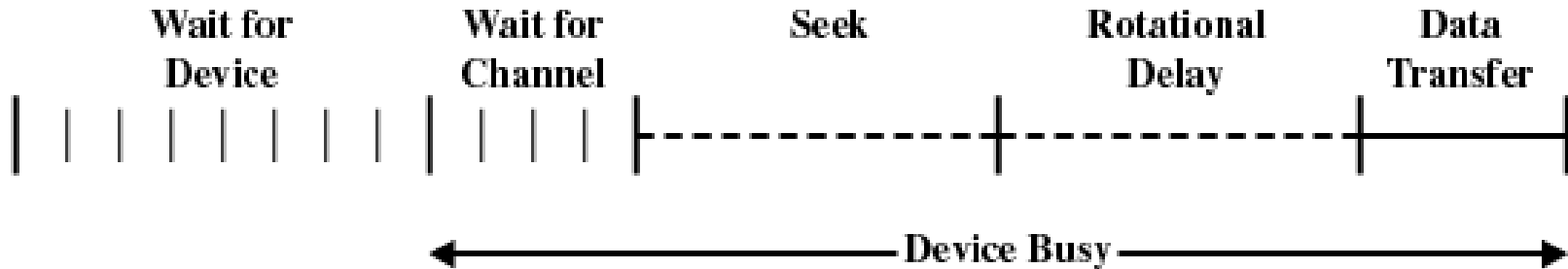


disk scheduling

Disk Performance Parameters

- To read or write, the disk head must be positioned at the desired track and at the beginning of the desired sector



Disk Performance Parameters

- **Seek time**
 - Time it takes to position the head at the desired track
 - 1 to 20 ms. (average 8 ms)
- **Rotational delay or rotational latency**
 - Time it takes for the beginning of the sector to reach the head
 - for 7200 rpm -> 8.3 ms for a full rotation and 4.2 for the average

Disk Performance Parameters

- access time
 - sum of seek time and rotational delay
 - the time it takes to get in position to read or write
- data transfer occurs as the sector moves under the head
 - transfer occurs in the disk buffer...
 - ...then in the controller buffer...
 - ...than in main memory
 - **data transfer is much faster than access**
 - several hundreds of sectors per track in current HD



disk scheduling: formal statement

- **input:**
 - a set of requests (tracks to seek)
 - current disk head location.
 - other algorithm state (e.g. current head direction)

- **output:** the next request to serve



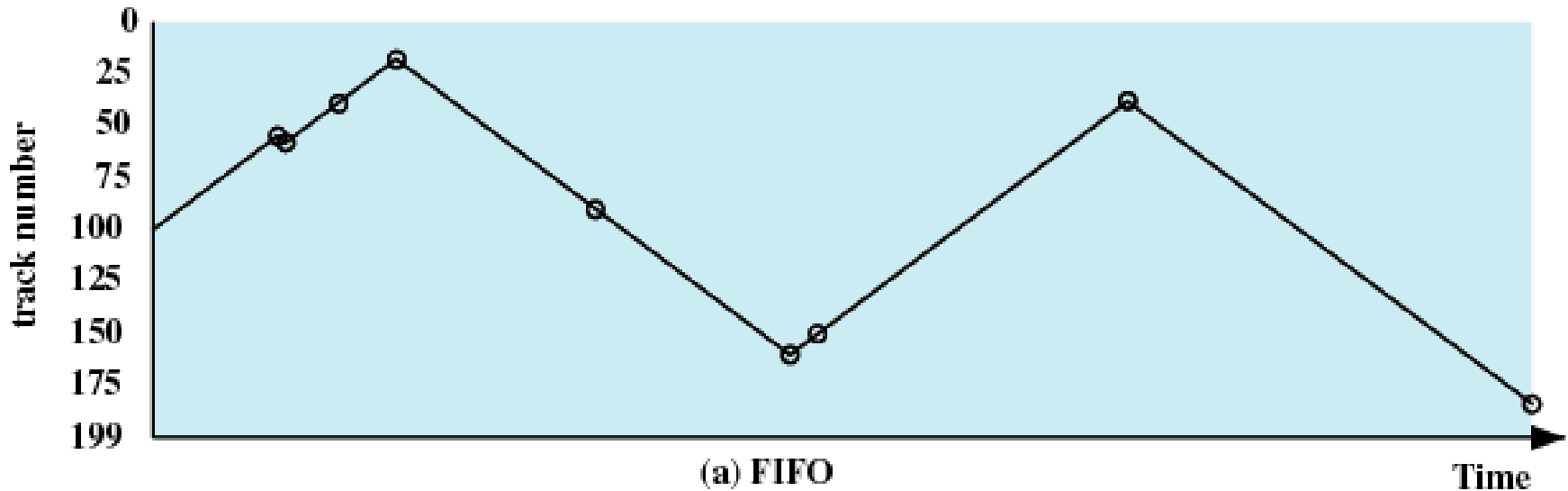
not on
the book

goals

- max throughput
- fairness
 - disk scheduling with priorities is rare
 - avoid starvation
 - avoid very long waits
- starvation vs. unfairness
 - starvation: a request never served
 - unfairness: certain requests wait longer
 - warning: often confused!

FIFO

- process request sequentially
- fair, no starvation
- if there are many processes it performs like random scheduling

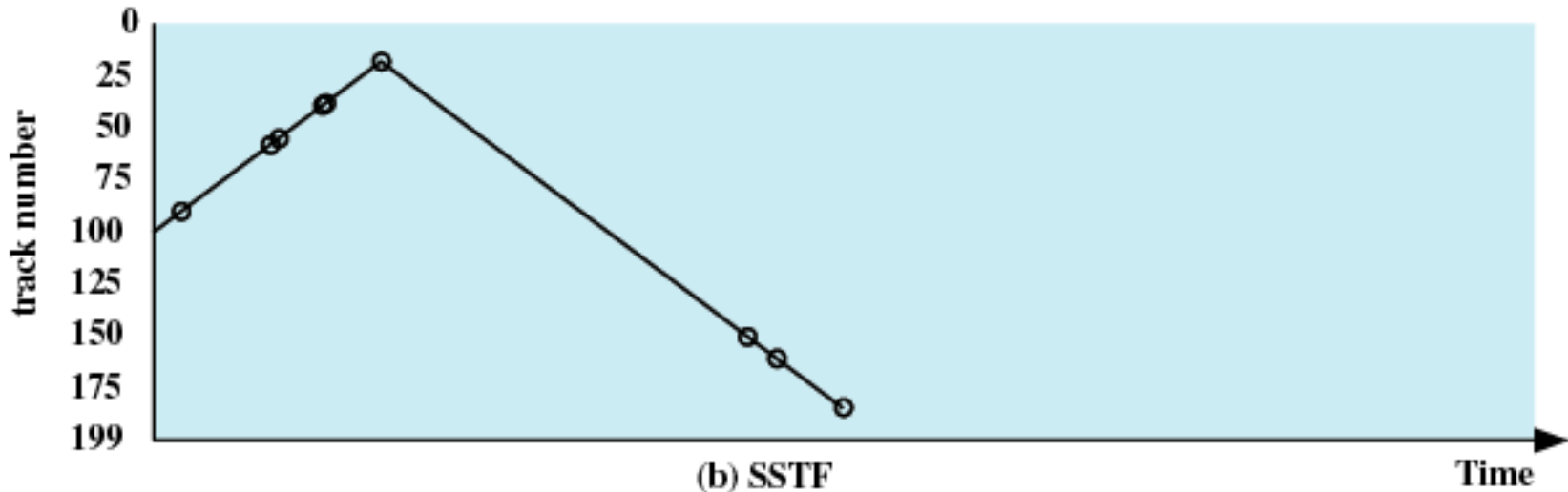


LIFO

- Last-in, first-out
- Good for transaction processing systems
 - The device is given to the most recent user so there should be little arm movement
- possibility of starvation since a job may never regain the head of the line
- only of theoretical interest

shortest service time first

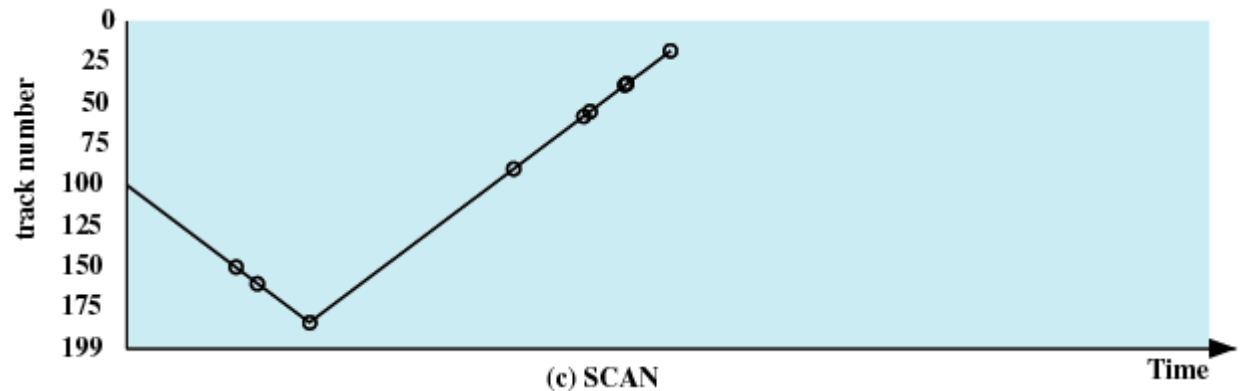
- Select the disk I/O request that requires the least movement of the disk arm from its current position
- optimal throughput!
- highly unfair!
- starvation when consecutive requests are for near tracks





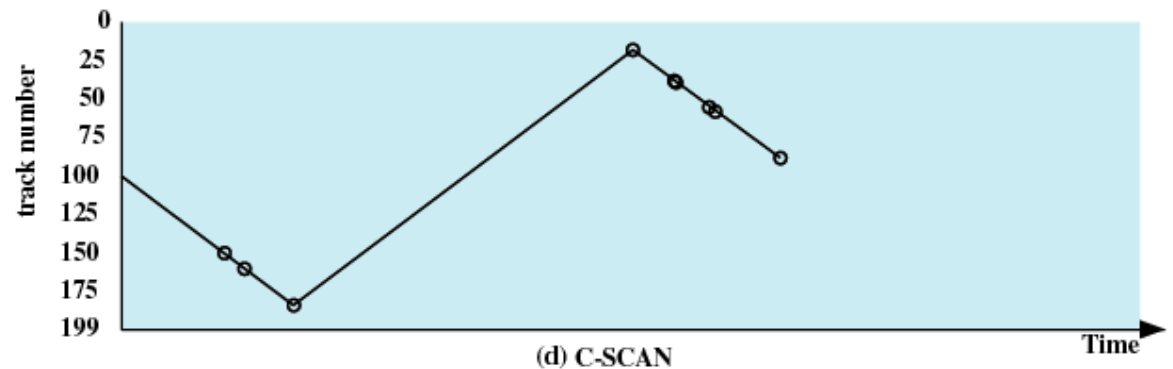
elevator

- famous, also called SCAN or LOOK
- arm moves in one direction only, satisfying all outstanding requests until it reaches the last track in that direction
- at the end direction is reversed
- good throughput
- unfair: average wait time for tracks on the edge is twice that for the tracks in the middle
- starvation only for continuous read on the same track
- used in practice



one-way (or cyclic) elevator

- anche detto C-SCAN
- like elevator but restricts scanning to one direction only
- when the last track is reached, the arm is returned to the opposite end and the scan begins again
 - performance penalty with respect to elevator
- good throughput
- fair
- starvation only for continuous read on the same track
- used in practice



request merging

- requests for adjacent blocks are treated as one
- avoid access time penalty
 - use the access of the first request
- may increase performance
 - e.g. irrelevant for SSTF, important for FIFO
- used in real systems



the “write-starving-reads” problem

- writes can be issued sequentially
 - processes usually do not depend on write request been actually fulfilled
- read are not usually issued sequentially
 - processes wait for the data before requesting the next read operation
- not really “starvation”, actually is “unfairness”

linux

- **noop**
 - FIFO + request merging
- **deadline**
 - one-way elevator
 - reads cannot wait more 500ms, and writes 5s, to avoid the write-starving-reads problem
 - but it seeks back and forth to meet deadlines!

- **anticipatory**
 - request merging
 - deadline approach
 - after a read, waits 6ms for another read
 - avoid the write-starving-reads problem
 - no seek back and forth
 - **waiting is not always performed**
 - heuristics to estimate the behavior of the running processes are implemented
 - **good for streaming, bad for dbms**

complete fair queueing (cfq)

- request merging
- one-way elevator
- fair with a round robin approach
 - schedule requests of each process for a few milliseconds
 - if no more requests for the scheduled process, wait a bit for further requests in the time slice
 - support process I/O priorities (command *ionice*)
- good for mutliuser environments
- latest versions performs as good as “anticipatory” for streaming
- usually set as default



linux: disk scheduling switching

- scheduling algorithm can be switched
 - at run time
 - per device

```
pisolo:~# cat /sys/block/hda/queue/scheduler
noop anticipatory deadline [cfq]

pisolo:~# echo anticipatory >
/sys/block/hda/queue/scheduler

pisolo:~# cat /sys/block/hda/queue/scheduler
noop [anticipatory] deadline cfq
```



not on
the book

RAID

RAID

- Redundant Array of Independent Disks
- Set of physical disk drives viewed by the operating system as a single logical drive
- improves...
 - ...performance
 - ...fault tollerance when one or more hard drives fail
 - availability: the service may still be available when a fault occur
 - data security: no data loss



degradation, rebuilding, and spare disks

- when a disk fails the array enters “**degraded**” (or **critical**) state
 - performance and redundancy is not as the full working array
- when the disk is substituted it must be updated with the data to fully work in the array: **rebuilding**
 - lasts hours, performance may be even worse
- substitution can be automatic in systems that have unused disks available (**hot spare disk**)

techniques

- **mirroring**
 - data are stored duplicated on (at least) two disks
 - duplexing: the two disks are controlled by a distinct controller
- **parity or humming error correction code**
 - redundant bits are stored with the data
- **striping**
 - “consecutive” data are distributed across the physical drives of an array
 - bit, byte, block level granularity

several kinds of RAIDs

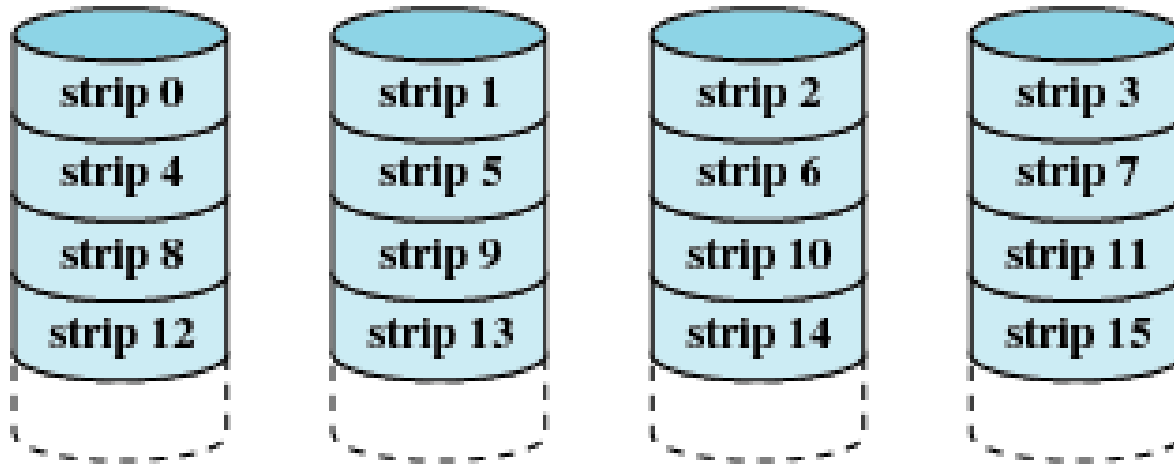
warning: this terminology is not followed by everybody

- **RAID0 used a lot, inexpensive**
- **RAID1 used a lot, inexpensive**
- RAID2 (not used any more)
- RAID3
- RAID4 similar to RAID3
- **RAID5 used a lot**
- RAID6 rare, expensive
- RAID7 proprietary
- nested (or multiple) RAID
 - 01/10, 03/30, 05/50, 15/51, ecc.

type of requests

- sequential I/O
 - big files (streaming, bulk)
 - blocks are stored contiguously
- random I/O
 - high frequency of requests for a very small amount of data (OLTP)
 - blocks are scattered through the disk
- read, write

RAID 0 (non-redundant)

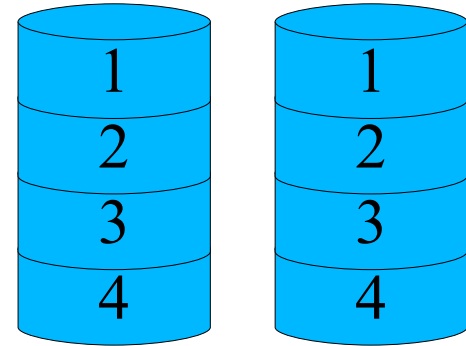


- just striping
- I/O: always very good
 - on average speedup xN (almost)
- failure of one disk makes all the array to fail

RAID0: mean time between failures

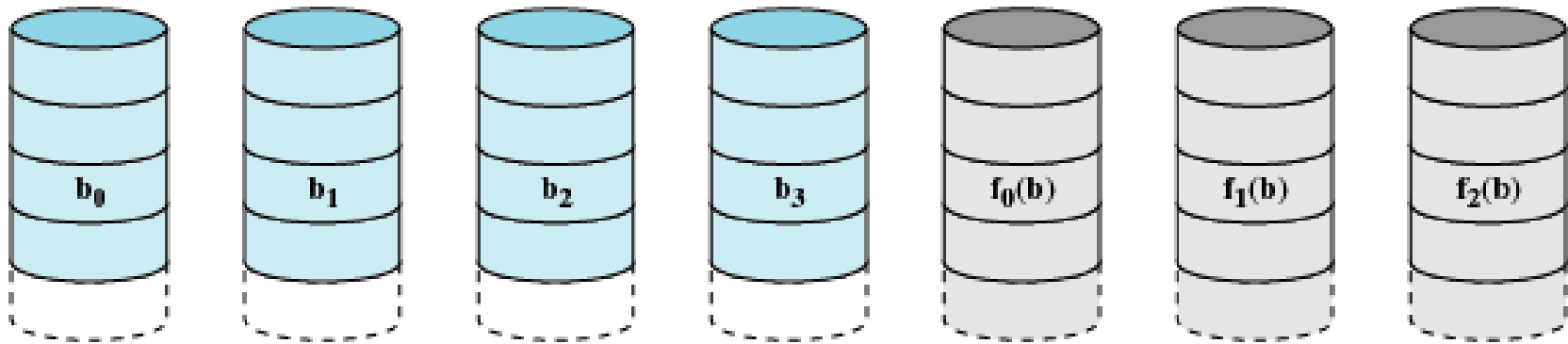
- $MTBF = 1/\lambda$
 - where λ is the fault frequency
- if a system A fail when one of its components $D1, \dots, DN$ fails
$$\lambda_A = \lambda_{D1} + \lambda_{D2} + \dots + \lambda_{DN}$$
 - frequencies can be summed up if faults are independent
- if disks are identical and in RAID0
- $MTBF_{Array} = MTBF_{Disk} / N$

RAID 1



- just mirroring (or duplexing)
 - rarely more than one mirror
- one disk may fail but the other still works
- I/O:
 - writes must be done on both disks
 - reads can be done in parallel on the two disk
- limited in size by the size of a single disk

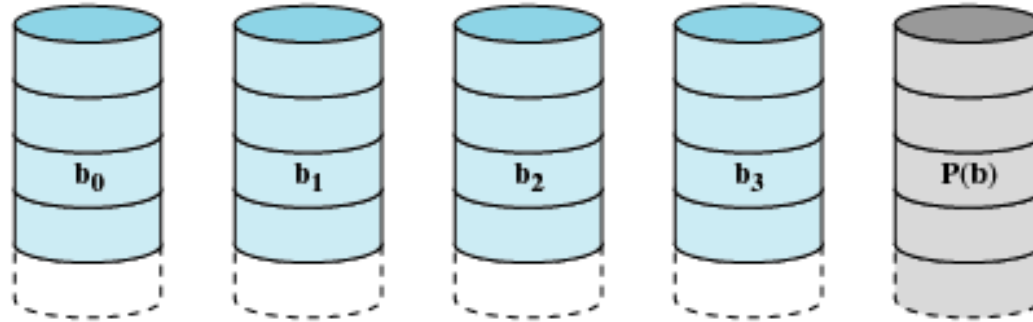
RAID 2



(c) RAID 2 (redundancy through Hamming code)

- bit-level striping, disks should be synchronized
- error correction by hamming code, useful for high bit error rate
 - but now all HD have error correction code built in!
- requires expensive proprietary hw, never used!

RAID 3

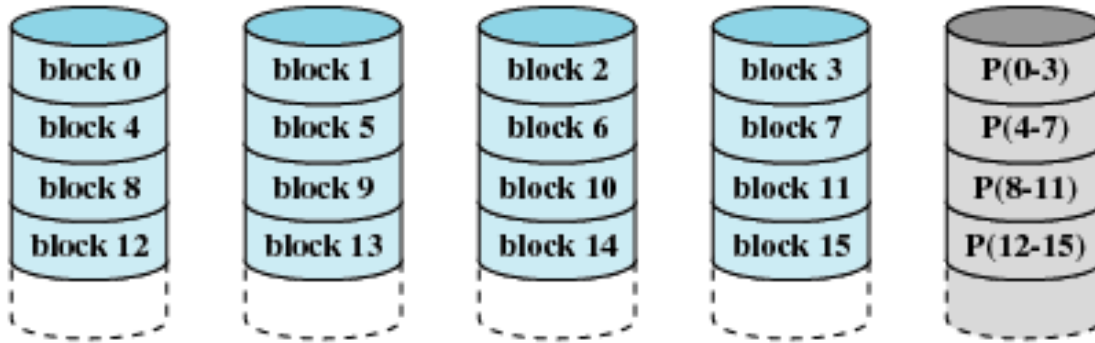


- byte-level striping with dedicated parity disk
 - a disk block contains more stripes
 - usually less than 1024 bytes in length
- read: very good, see RAID0
- write: poor, parity disk is a bottleneck, computation load on the CPU, hw implementation is preferred
- tolerant to 1 disk failure (rebuilding by XOR)

RAID3 write: computing parity

- in the same row: stripes A, B, C and parity stripe $P = A \text{ xor } B \text{ xor } C$
- A is written as A'
- new parity P' can be computed in two ways
 - $P' = A' \text{ xor } B \text{ xor } C$
 - read B and C? a cache may be very much useful
 - $P' = A' \text{ xor } A \text{ xor } P$
 - A may be read and in cache, P should be read.

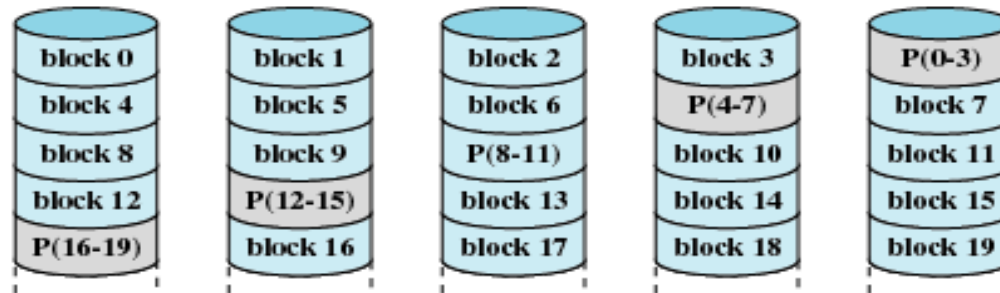
RAID 4



(e) RAID 4 (block-level parity)

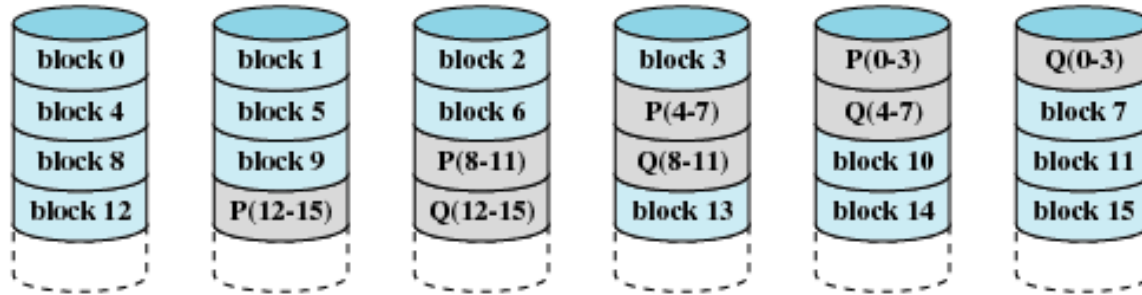
- block-level striping with dedicated parity disk
 - a stripe spans more disk blocks
- see RAID3

RAID 5



- block-level striping with distributed parity
- read: slightly better than RAID0 (1 disk more)
- write:
 - better than RAID4 (no parity disk bottleneck)
 - computation requires knowledge of the entire row or old parity (read? cache?)
 - cpu intensive, hw implementation is preferred
- tolerant to 1 disk failure (rebuilding by XOR)

RAID 6



(g) RAID 6 (dual redundancy)

- block-level striping with *dual* distributed parity
- two disks may fail without data loss
- read: slightly better than RAID5 (1 disk more)
- write: slightly worse than RAID5 (2 parities)
- costly

nested RAID arrays

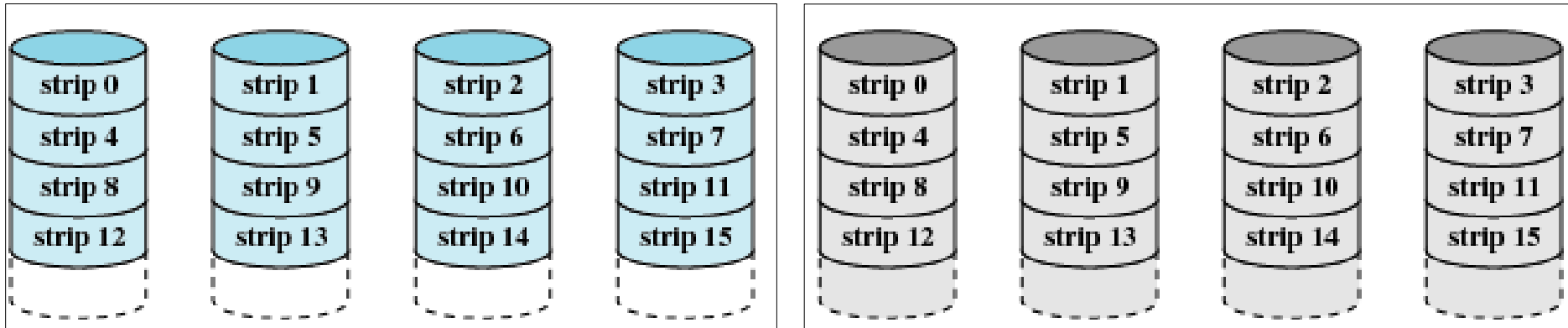
- the idea: treat several raid array as a disk and build a RAID array out of them
- notation: RAID XY
 - means that you first have several RAID-X array and you build a RAID-Y array out of them
 - warning: this notation is not followed by everybody

RAID 10



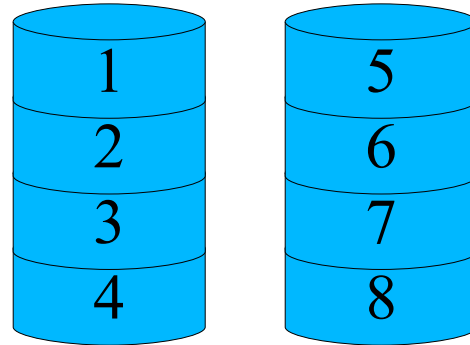
- striping on mirroring
- performances very very good (see RAID0 and 1)
- each RAID1 array can support a disk failure
- good performances also in rebuilding
 - only the affected array is slowed down

RAID 01



- mirroring on striping
- as RAID 10 but when one disk fails **the whole RAID0 array is considered down** by many controllers
- avoid it

just a bunch of disks (JBOD)



- a.k.a spanning, not really a RAID configuration
- same drawbacks as in RAID0
- sw/hw complexity as in RAID0
- usable with odd drives without wasting space
- easier disaster recovery than RAID0



comparison

- from www.pcguide.com

RAID Level	Number of Disks	Capacity	Storage Efficiency	Fault Tolerance	Availability	Random Read Perf	Random Write Perf	Sequential Read Perf	Sequential Write Perf	Cost
0	2,3,4,...	S*N	100%	none	★	★★★★★	★★★★★	★★★★★	★★★★★	\$
1	2	S*N/2	50%	★★★★★	★★★★★	★★★★	★★★★	★★	★★★	\$\$
2	many	varies, large	~ 70-80%	★★	★★★★★	★★	★	★★★★★	★★★	\$\$\$\$\$
3	3,4,5,...	S*(N-1)	(N-1)/N	★★★	★★★★★	★★★	★	★★★★★	★★★	\$\$
4	3,4,5,...	S*(N-1)	(N-1)/N	★★★	★★★★★	★★★★★	★★	★★★	★★	\$\$
5	3,4,5,...	S*(N-1)	(N-1)/N	★★★	★★★★★	★★★★★	★★	★★★★★	★★★	\$\$
6	4,5,6,...	S*(N-2)	(N-2)/N	★★★★★	★★★★★	★★★★★	★	★★★★★	★★	\$\$\$
7	varies	varies	varies	★★★	★★★★★	★★★★★	★★★★★	★★★★★	★★★★★	\$\$\$\$\$
01/10	4,6,8,...	S*N/2	50%	★★★★★	★★★★★	★★★★★	★★★★	★★★★★	★★★★	\$\$\$
03/30	6,8,9,10,...	S*N0*(N3-1)	(N3-1)/N3	★★★★	★★★★★	★★★★★	★★	★★★★★	★★★	\$\$\$\$
05/50	6,8,9,10,...	S*N0*(N5-1)	(N5-1)/N5	★★★★	★★★★★	★★★★★	★★★	★★★★★	★★★	\$\$\$\$
15/51	6,8,10,...	S*((N/2)-1)	((N/2)-1)/N	★★★★★	★★★★★	★★★★	★★★	★★★★★	★★★	\$\$\$\$\$

implementation

- software (in the OS)
 - no need for special drivers
 - may be inefficient for parity computation
 - Linux: (0, 1, 4, 5, and any kind of nesting)
 - Windows: XP (0,jbod), 2000 Server (0,1,5, jbod)
- hardware (in the controller)
 - need special drivers/software
 - efficient when parity should be computed
- hybrid (the bios drives the controller)
 - need special drivers
 - inefficient when parity computation is required