Imperial College London

Operating Systems

Disk Management

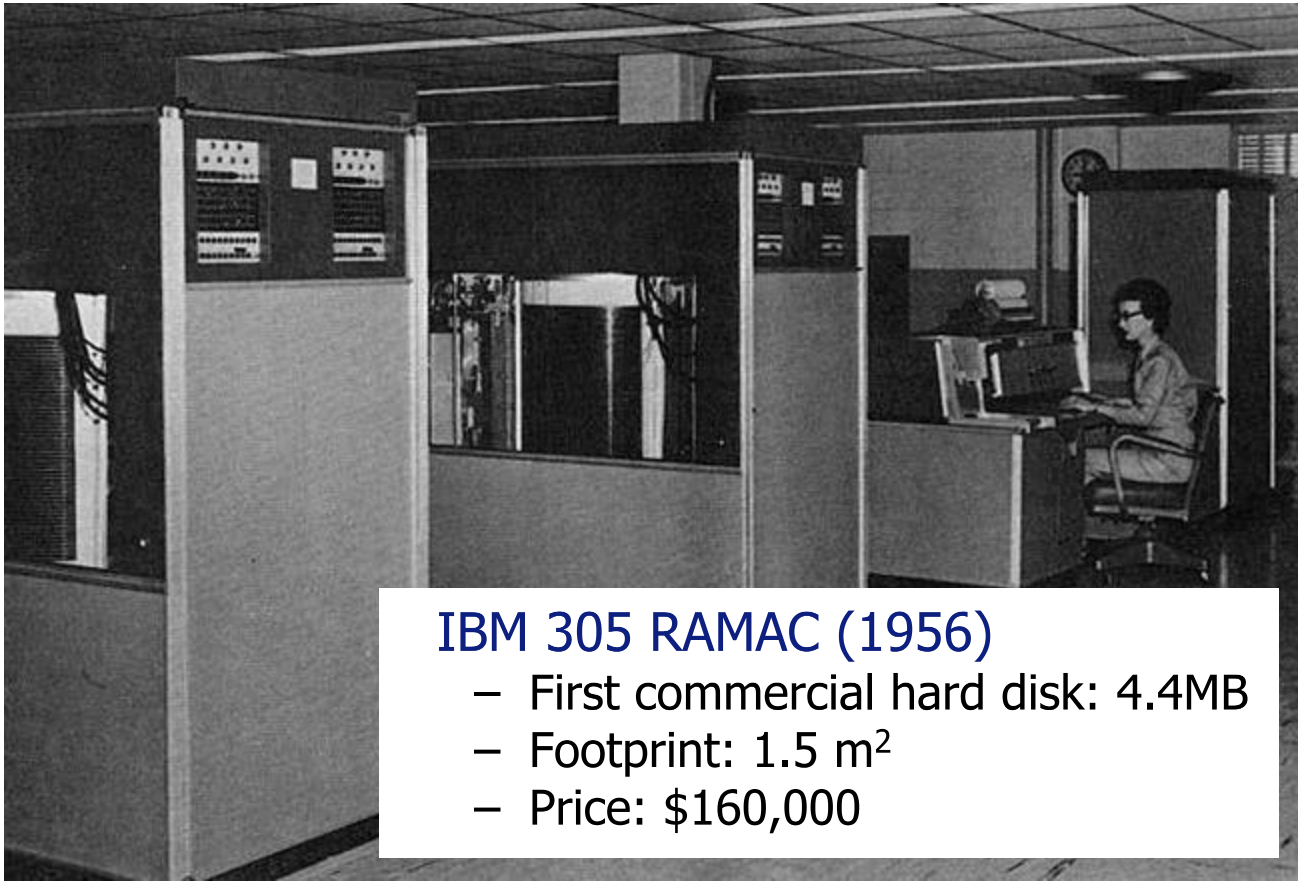
Course 502
Autumn Term 2014-2015

Based on slides by Daniel Rueckert, Peter Pietzuch, and Andrew Tanenbaum

Roman Kolcun

roman.kolcun08@imperial.ac.uk

Disks have come a long way...

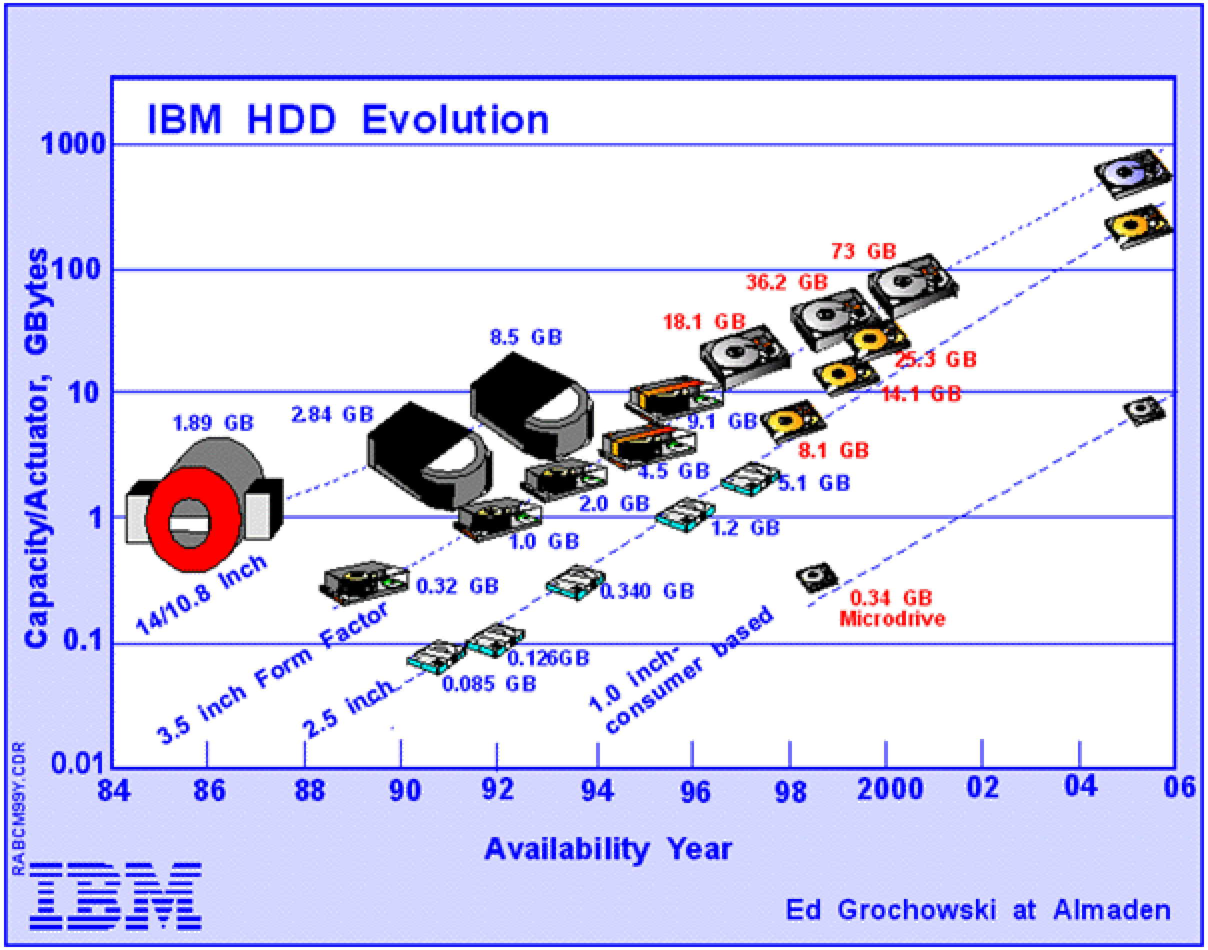


IBM 305 RAMAC (1956)
- First commercial hard disk: 4.4MB
- Footprint: 1.5 m^2
- Price: $160,000

From Computer Desktop Encyclopedia
Reproduced with permission.
© 2005 Toshiba Corporation

Toshiba 0.85” disk (2005)
- Capacity: 4GB
- Price: <$300

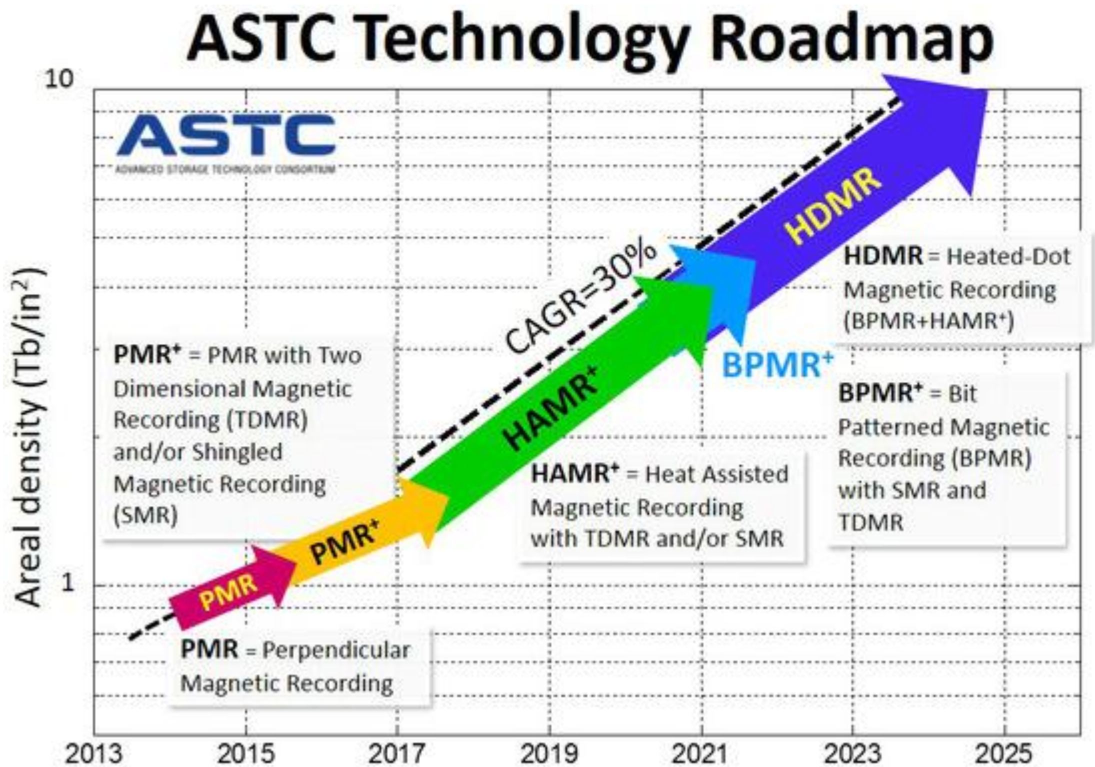
Disk Evolution



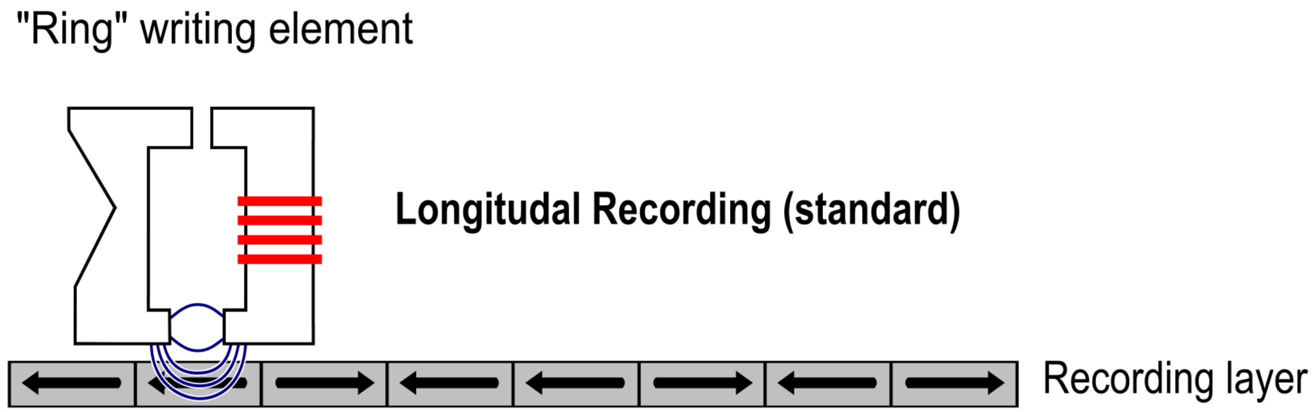
Capacity increases exponentially

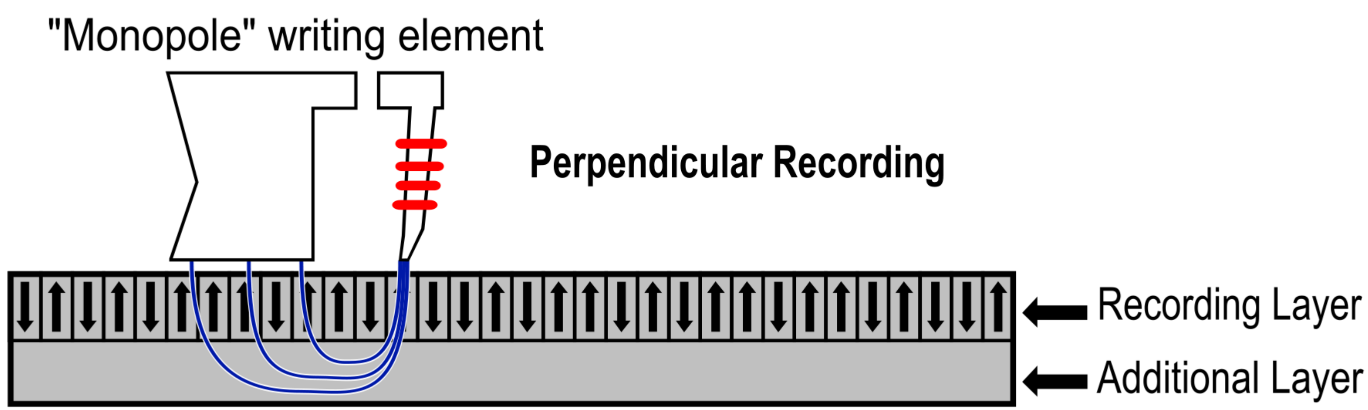
– Access speeds not so much... (why?)

Prediction of Disk Evolution

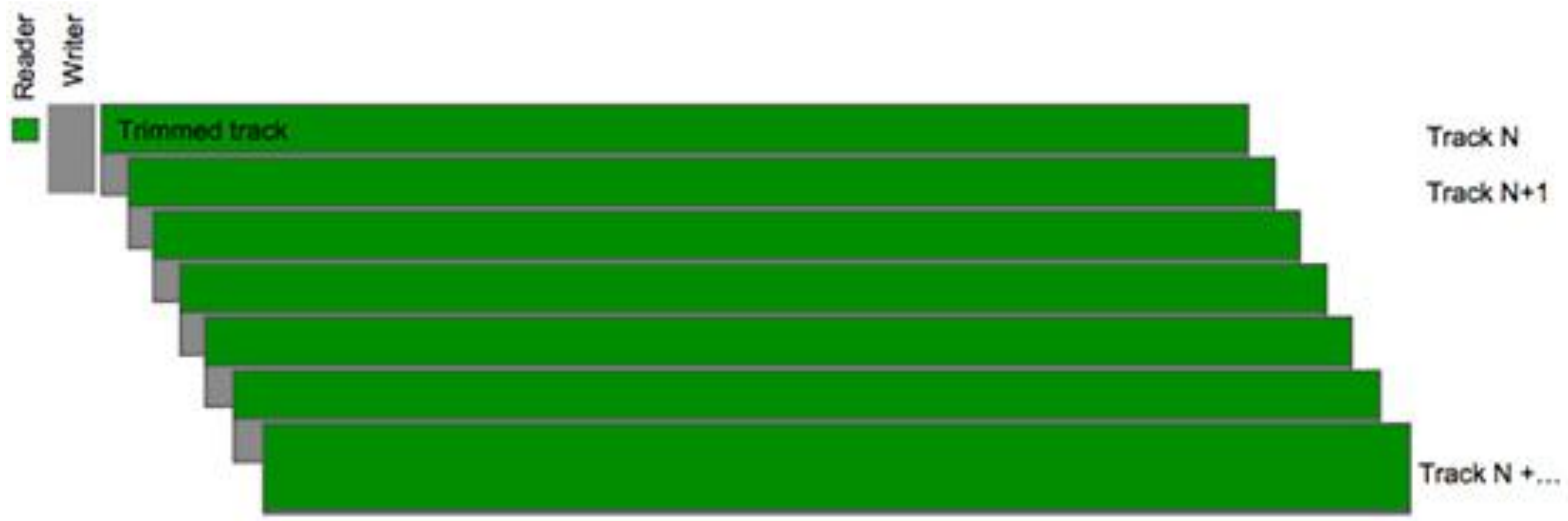


Longitudal vs. Perpendicular Recording

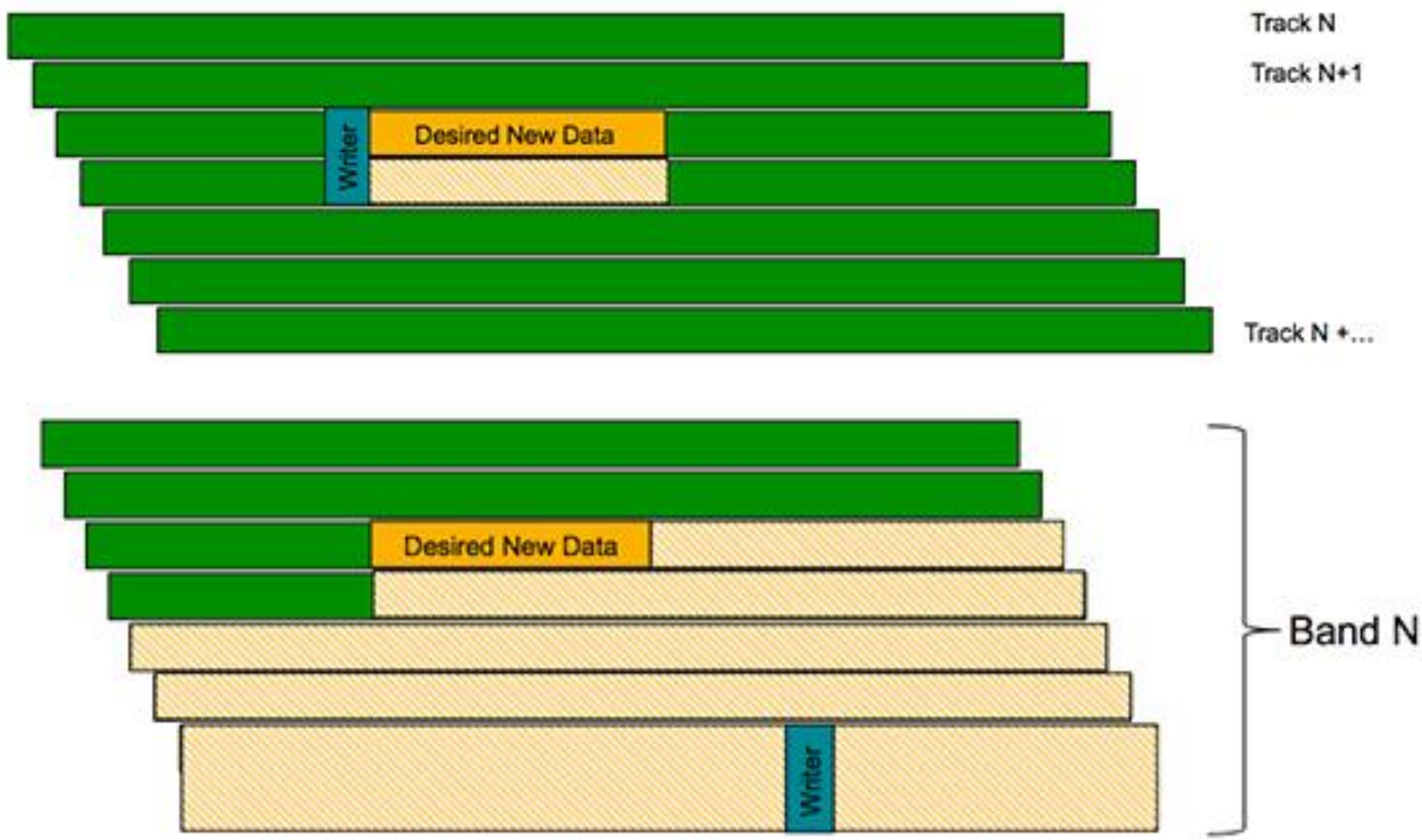




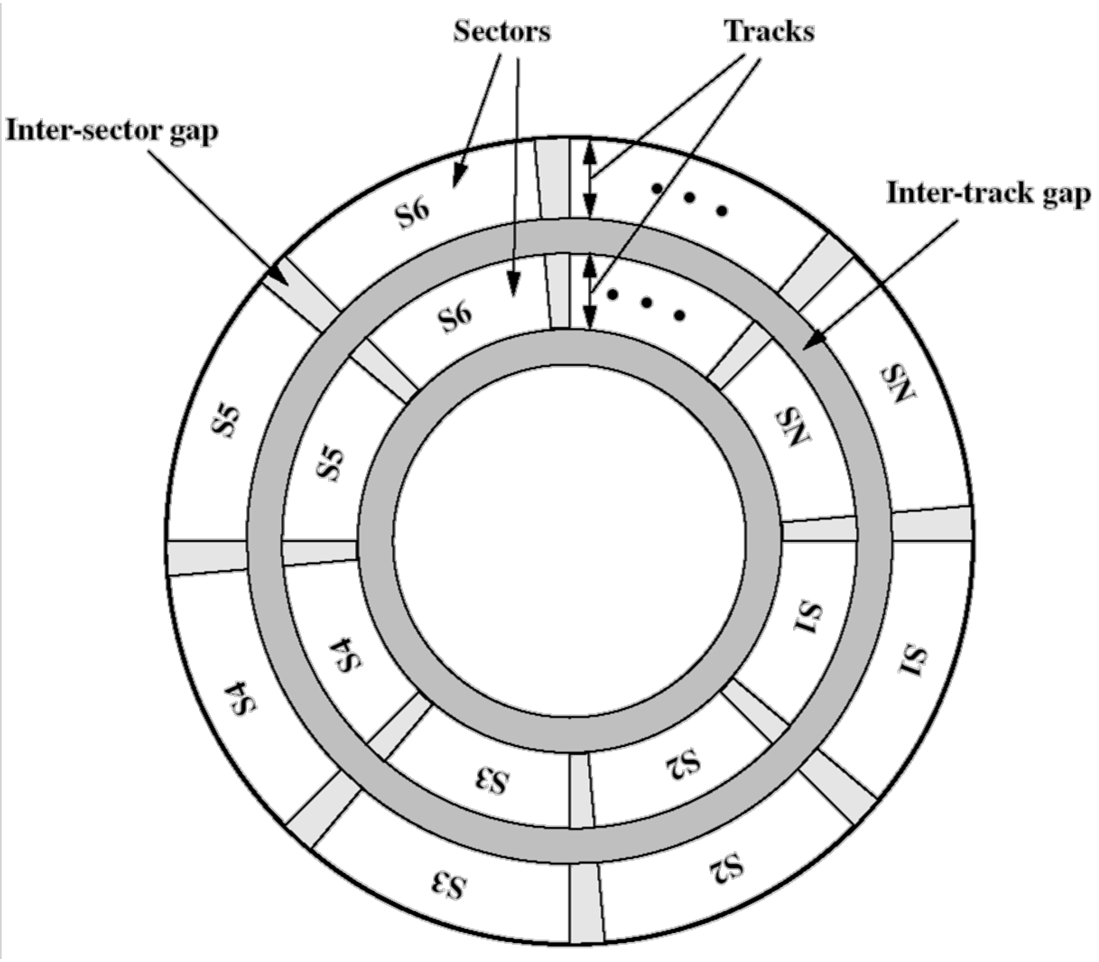
Shingled Magnetic Recording I

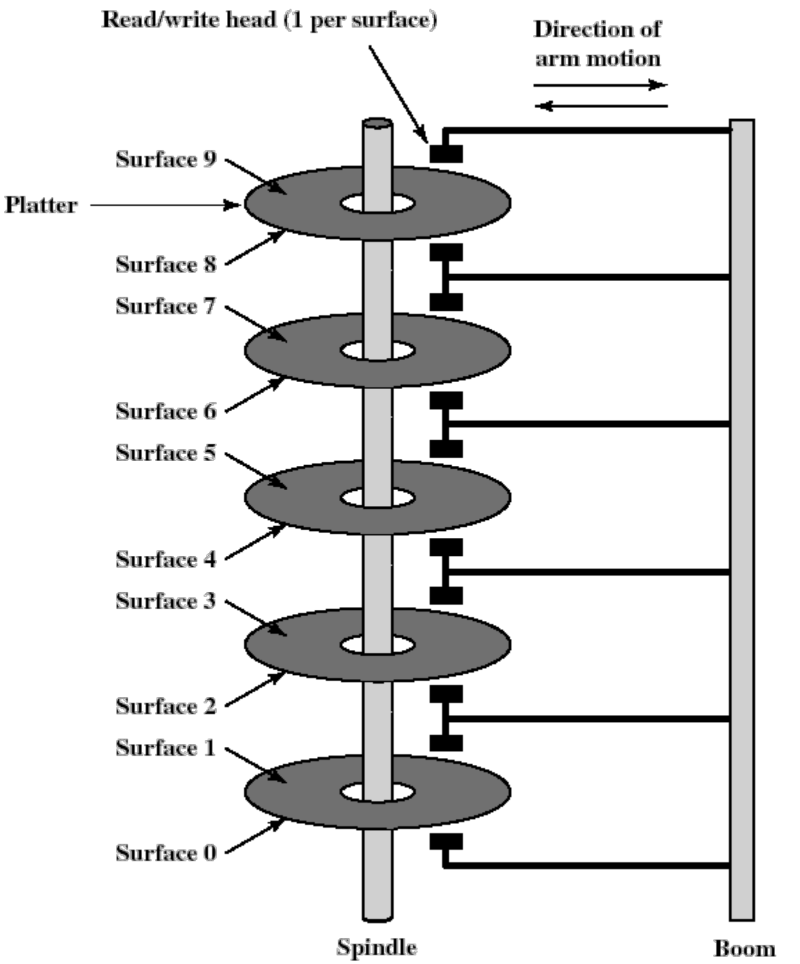


Shingled Magnetic Recording II

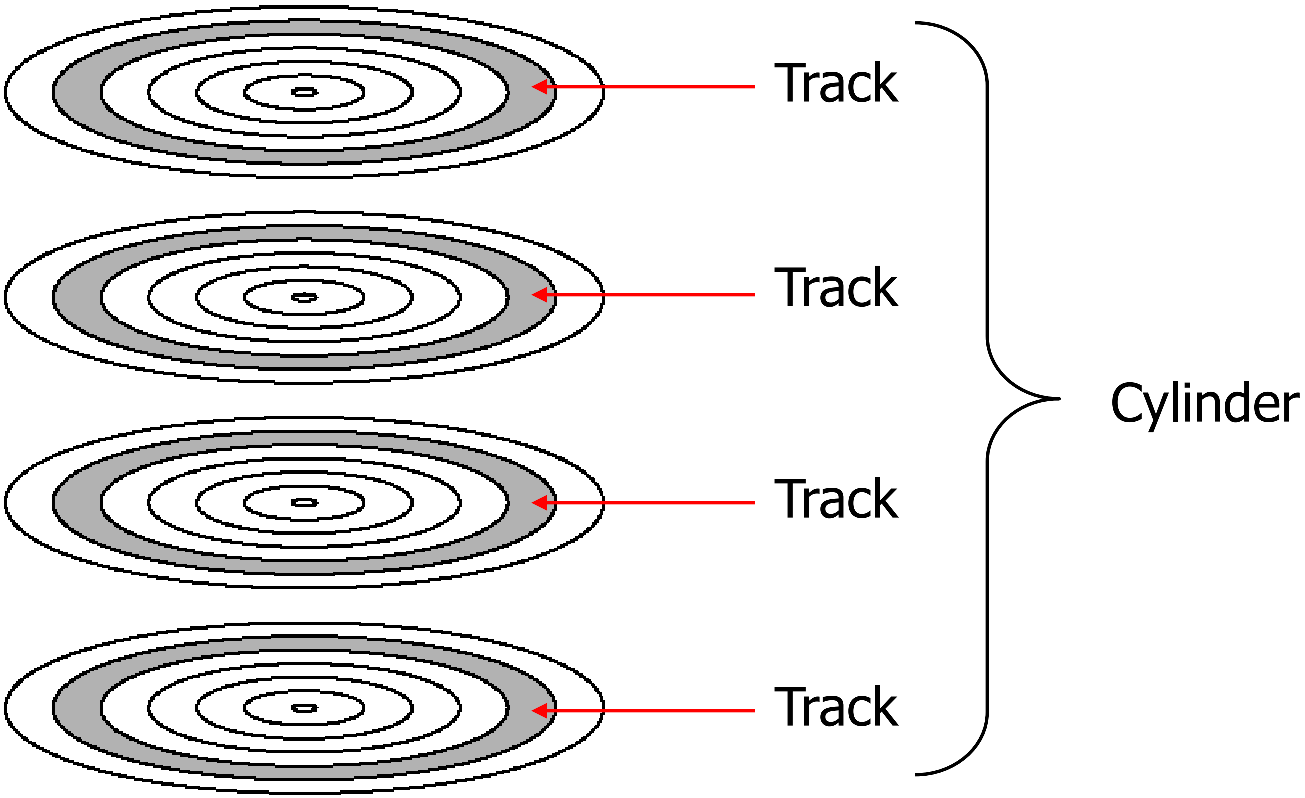


Disk Storage Devices





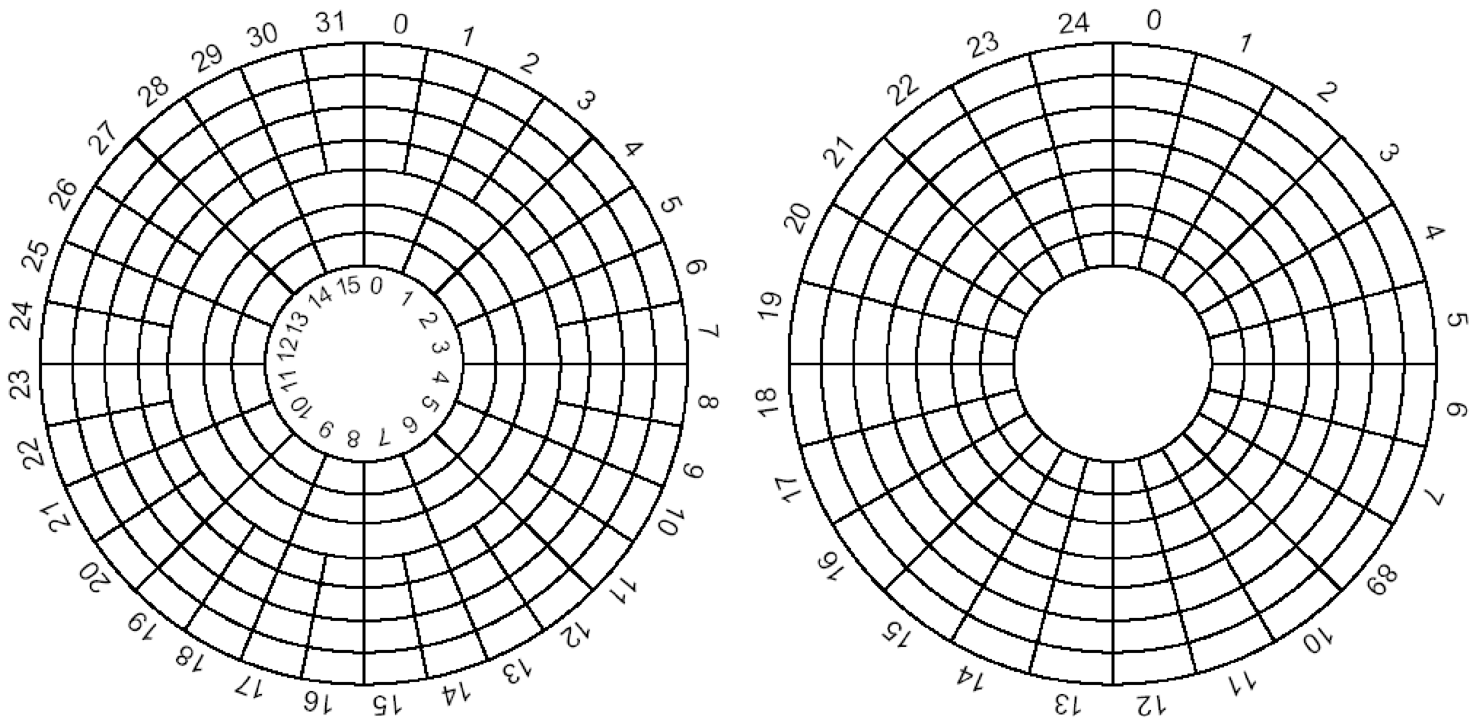
Tracks and Cylinders



Sample Disk Specification

Parameter	IBM 360KB floppy disk	Seagate Barracuda ST3400832AS
No. of cylinders	40	16,383
Tracks / cylinder	2	16
Sectors / track	9	63
Bytes / sector	512	512
Sectors / disk	720	781,422,768
Disk capacity	360KB	400GB

Sector Layout



(a) Physical geometry (b) Virtual geometry

Surface divided into 20 or more **zones**

- Outer zones have more sectors per track
- Ensures that sectors have same physical length
- Zones hidden using virtual geometry

Disk Addressing

Physical hardware address: (<u>cylinder</u>, <u>surface</u>, <u>sector</u>)

- But actual geometry complicated ➔ hide from OS

Modern disks use **logical sector addressing** (or logical block addresses LBA)

- Sectors numbered consecutively from 0..n
- Makes disk management much easier
- Helps work around BIOS limitations
  - Original IBM PC BIOS 8GB max
  - 6 bits for sector, 4 bits for head, 14 bits for cylinder

Disk Capacity

Disk capacity statements can be confusing!

1 KB = 2^{10} bytes = 1024 bytes **vs** 1 KB = 10^3 bytes = 1000 bytes

1 MB = 2^{20} bytes = 1024^2 bytes **vs** 1 MB = 10^6 bytes = 1000^2 bytes

1 GB = 2^{30} bytes = 1024^3 bytes **vs** 1 GB = 10^9 bytes = 1000^3 bytes

– For the exam: just make it consistent

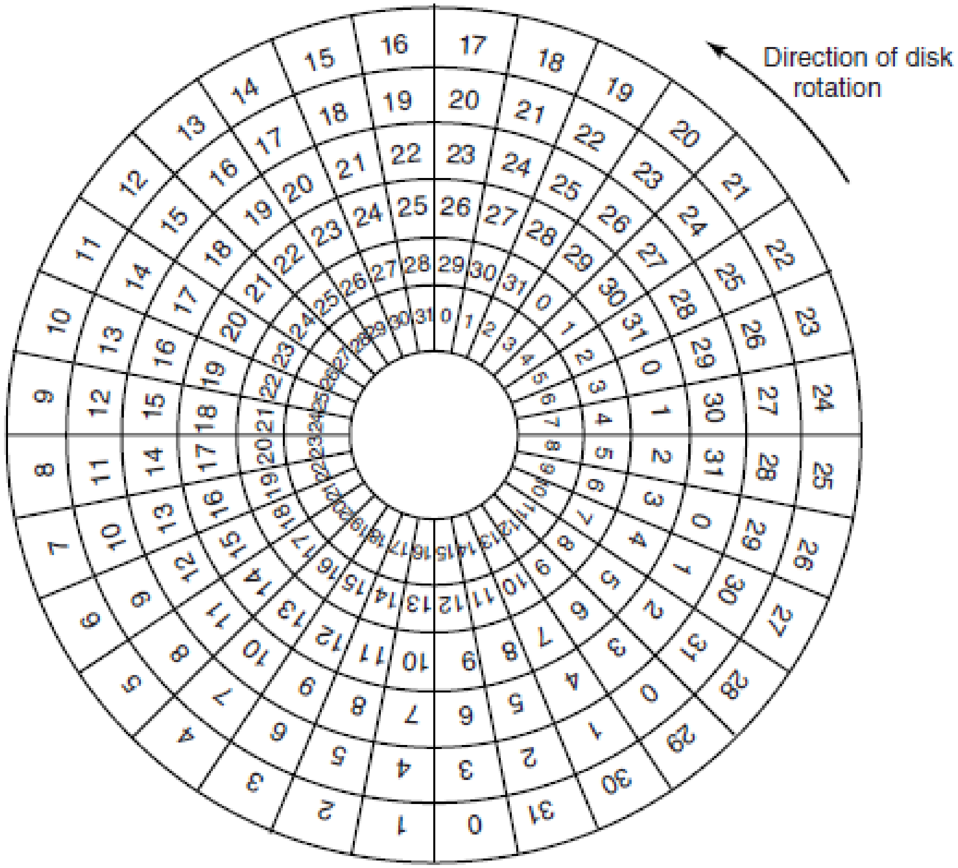
Disk Formatting I

Before disk can be used, it must be formatted:

- **Low level format**
  - Disk sector layout

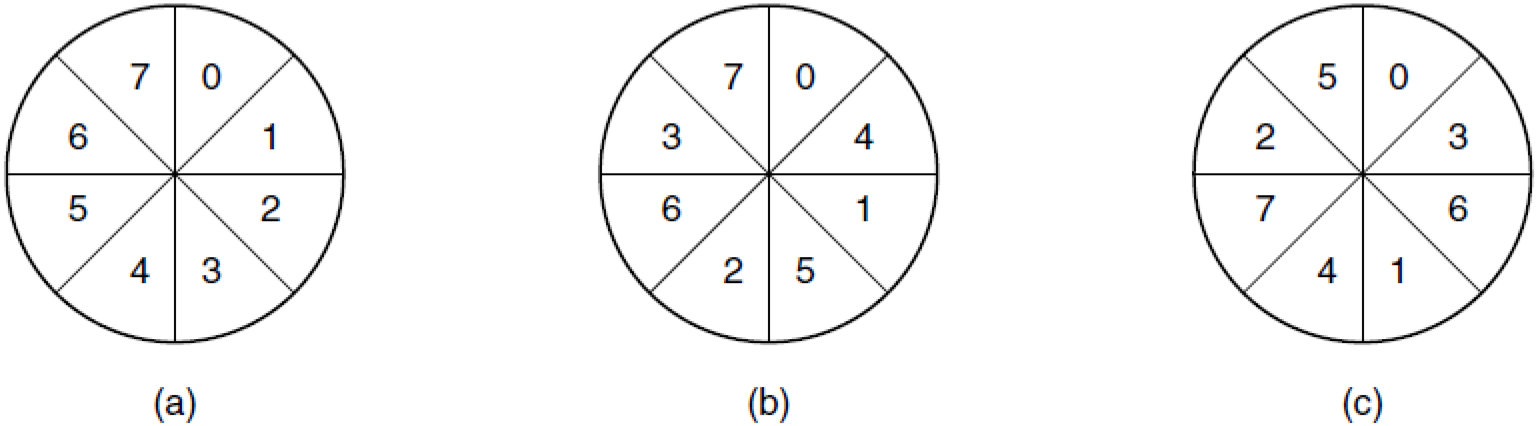
  - Cylinder skew
  - Interleaving
- **High level format**
  - Boot block
  - Free block list
  - Root directory
  - Empty file system

Disk Formatting II



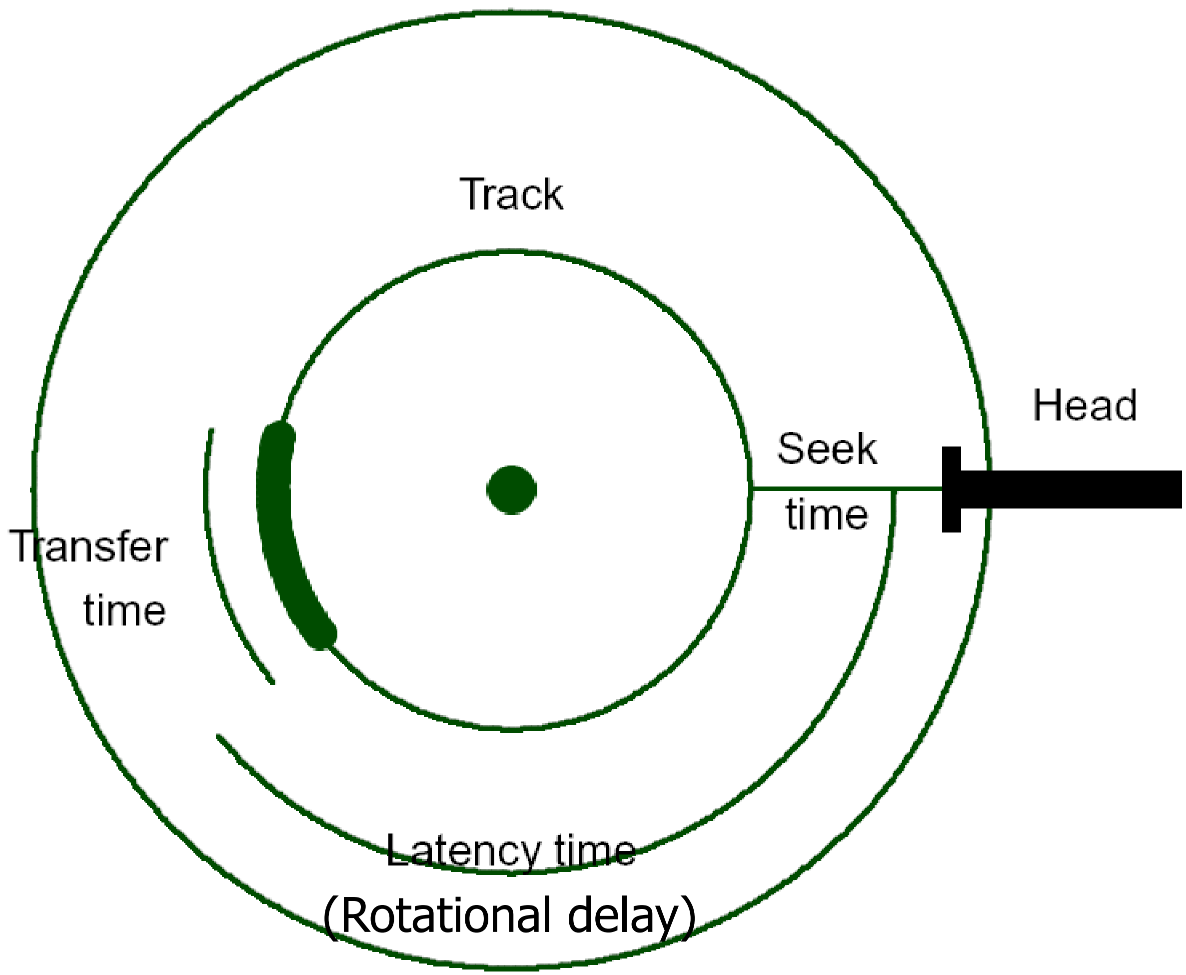
An illustration of cylinder skew.

Disk Formatting III



(a) No interleaving. (b) Single interleaving.
(c) Double interleaving.

Disk Delays I



Disk Delays II

Typical disk has:

Sector size:	512 bytes
Seek time (adjacent cylinder):	<1 ms
Seek time (average):	8 ms
Rotation time (average latency):	4 ms
Transfer rate:	up to 100MB per sec

Disk scheduling

- Minimise seek and/or latency times
- Order pending disk requests with respect to head position

Seek time approx. 2-3 larger than latency time

- More important to optimise

Disk Performance

Seek time: t_{seek}

Latency time (rotational delay): $t_{latency} = \frac{1}{2r}$

Transfer time: $t_{transfer} = \frac{b}{rN}$

where

b - number of bytes to be transferred

N - number of bytes per track

r - rotation speed in revolutions per second

Total access time: $t_{access} = t_{seek} + \frac{1}{2r} + \frac{b}{rN}$

Example: Disk Performance

Example:

Average seek time: 10ms
Rotation speed: 10,000 rpm
512 byte sectors
320 sectors per track
File size: 2560 sectors (1.3 MB)

Case A:

- Read file stored as compactly as possible on disk
  - i.e. file occupies all sectors on 8 adjacent tracks
  - 8 tracks x 320 sectors/track = 2560 sectors

Case B:

- Read file with all sectors randomly distributed across disk

Answer: Disk Performance

Case A:

- Time to read first track

Average seek =	10 ms	
Rotational delay =	3 ms	= 1 / [2 * (10,000 / 60)]
Read 320 sectors =	6 ms	= b / (N * (10,000 / 60)]
	19 ms	

- Time to read next track = 3 ms + 6 ms = 9 ms
- Total time = 19 ms + 7 x 9 ms = 82 ms = 0.082 seconds

Case B:

Average seek =	10 ms	
Rotational delay =	3 ms	
Read 1 sector =	0.01875 ms	= 512 / [512*320 * (10,000/60)]
	13.01875 ms	

- Total time = 2560 x 13.01875 ms = 33.328 seconds

Disk Scheduling

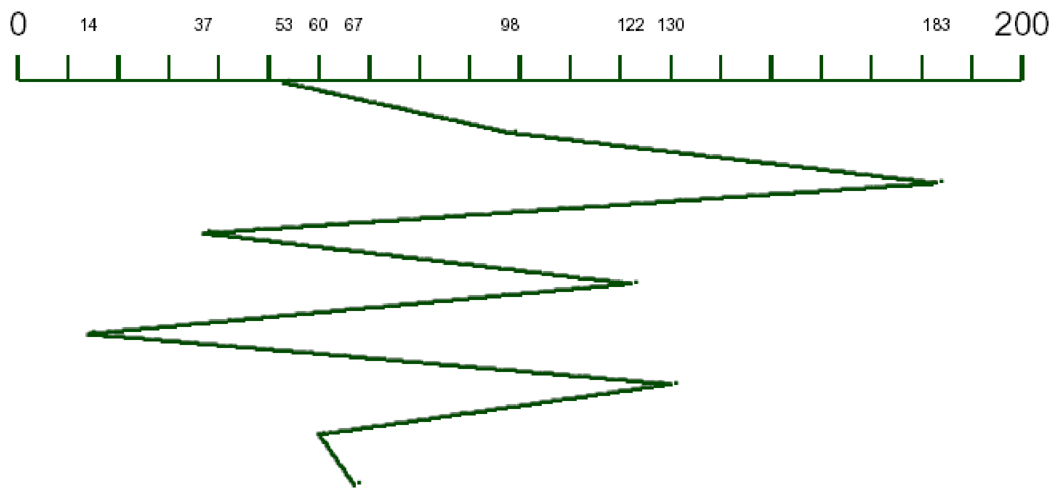
First Come First Served (FCFS)

No ordering of requests: random seek patterns

- OK for lightly-loaded disks
- But poor performance for heavy loads
- Fair scheduling

Queue: 98, 183, 37, 122, 14, 130, 60, 67

- Head starts at 53



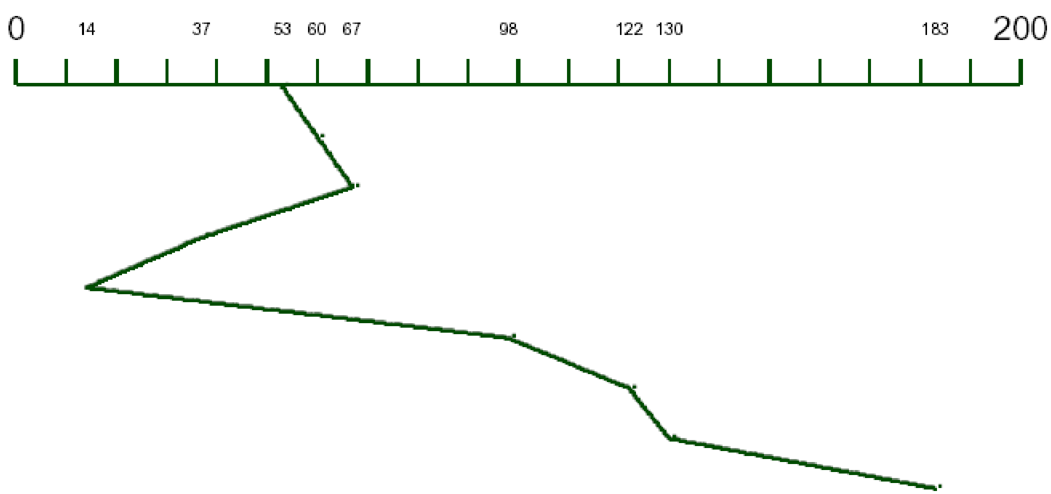
Shortest Seek Time First (SSTF)

Order requests according to shortest seek distance from current head position

- Discriminates against innermost/outermost tracks
- Unpredictable and unfair performance

Queue: 98, 183, 37, 122, 14, 130, 60, 67

- Head starts at 53
- If, when handling request at 14, new requests arrive for 50, 70, 100, ➔ long delay before 183 serviced



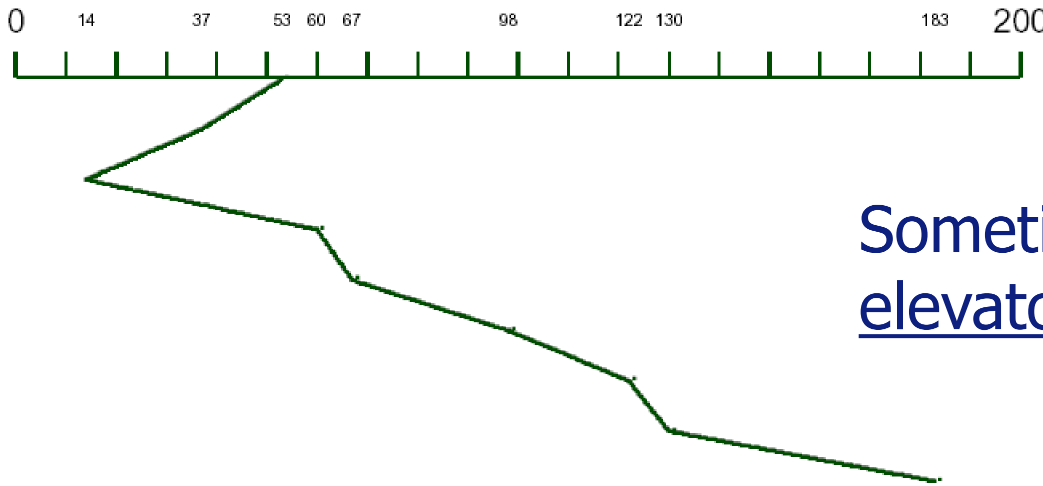
SCAN Scheduling

Choose requests which result in shortest seek time <u>in preferred direction</u>

- Only change direction when reaching outermost/innermost cylinder (or no further requests in preferred direction)
- Most common scheduling algorithm
- Long delays for requests at extreme locations

Queue: 98, 183, 37, 122, 14, 130, 60, 67

- Head starts at 53; direction: towards 0



Sometimes called <u>elevator scheduling</u>

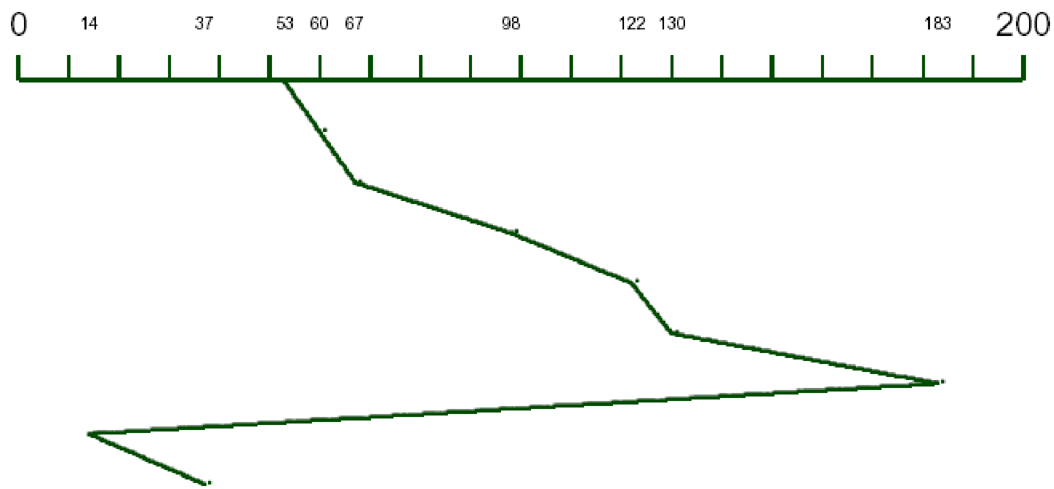
C-SCAN

Services requests in one direction only

- When head reaches innermost request, jump to outermost request
- Lower variance of requests on extreme tracks
- May delay requests indefinitely (though less likely)

Queue: 98, 183, 37, 122, 14, 130, 60, 67

- Head starts at 53



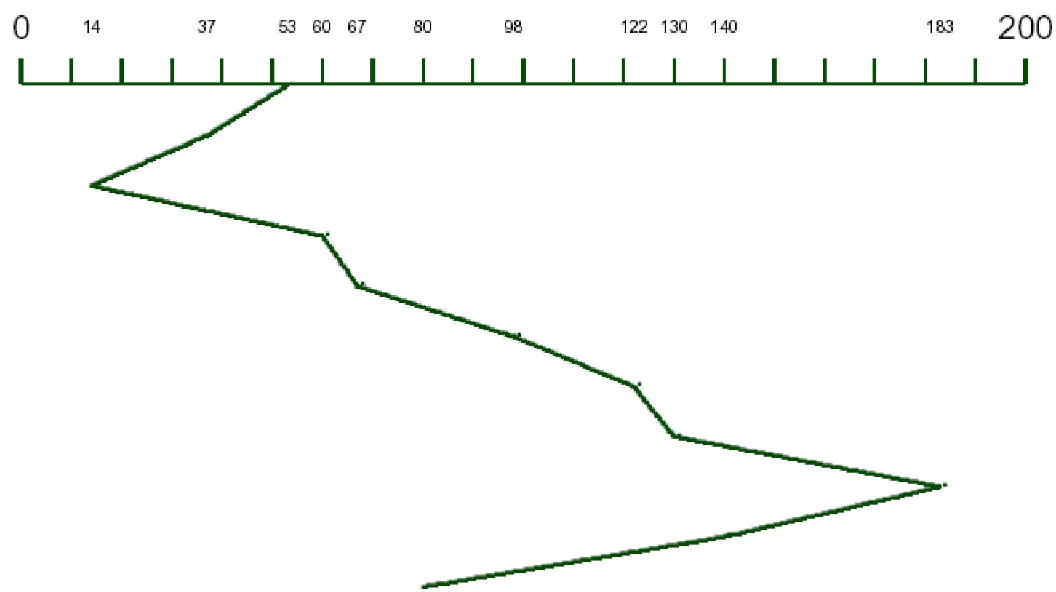
N-Step SCAN

As for SCAN, but services only requests waiting when sweep began

- Requests arriving during sweep serviced during return sweep
- Doesn't delay requests indefinitely

Queue: 98, 183, 37, 122, 14, 130, 60, 67

- Head starts at 53; direction: towards 0.
- Requests 80, 140 arrive when head moving outwards



Linux: Disk Scheduling

I/O requests placed in **request list**

- One request list for each device in system
- `bio` structure: associates memory pages with requests

Block device drivers define **request** operation called by kernel

- Kernel passes ordered request list
- Driver must perform all operations in list
- Device drivers do not define read/write operations

Some devices drivers (e.g. RAID) order their own requests

- Bypass kernel for request list ordering (why?)

Linux: Disk Scheduling Algorithms

Default: variation of SCAN algorithm

- Kernel attempts to merge requests to adjacent blocks
- But: synchronous read requests may starve during large writes

Deadline scheduler: ensures reads performed by deadline

- Eliminates read request starvation

Anticipatory scheduler: delay after read request completes

- Idea: process will issue another synchronous read operation before its quantum expires
- Reduces excessive seeking behaviour
- Can lead to reduced throughput if process does not issue another read request to nearby location
  - Anticipate process behaviour from past behaviour

RAID

RAID

Problem:

- CPU performance doubling every 18 months
- Disk performance has increased only 10 times since 1970

Solution: Use parallel disk I/O

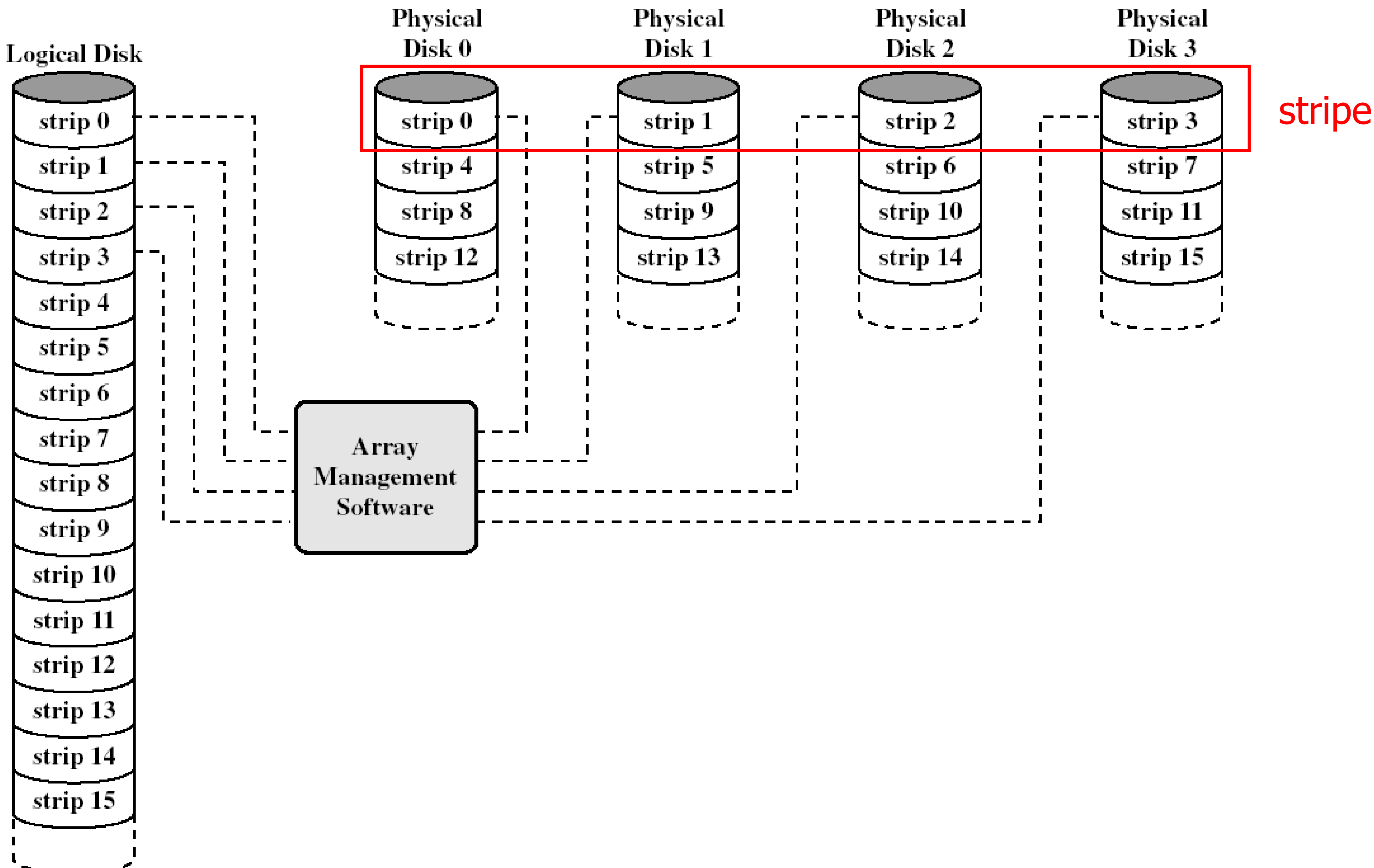
RAID (Redundant Array of Inexpensive Disks)

- Use array of physical drives appearing as single virtual drive
- Stores data distributed over array of physical disks to allow parallel operation (called **striping**)

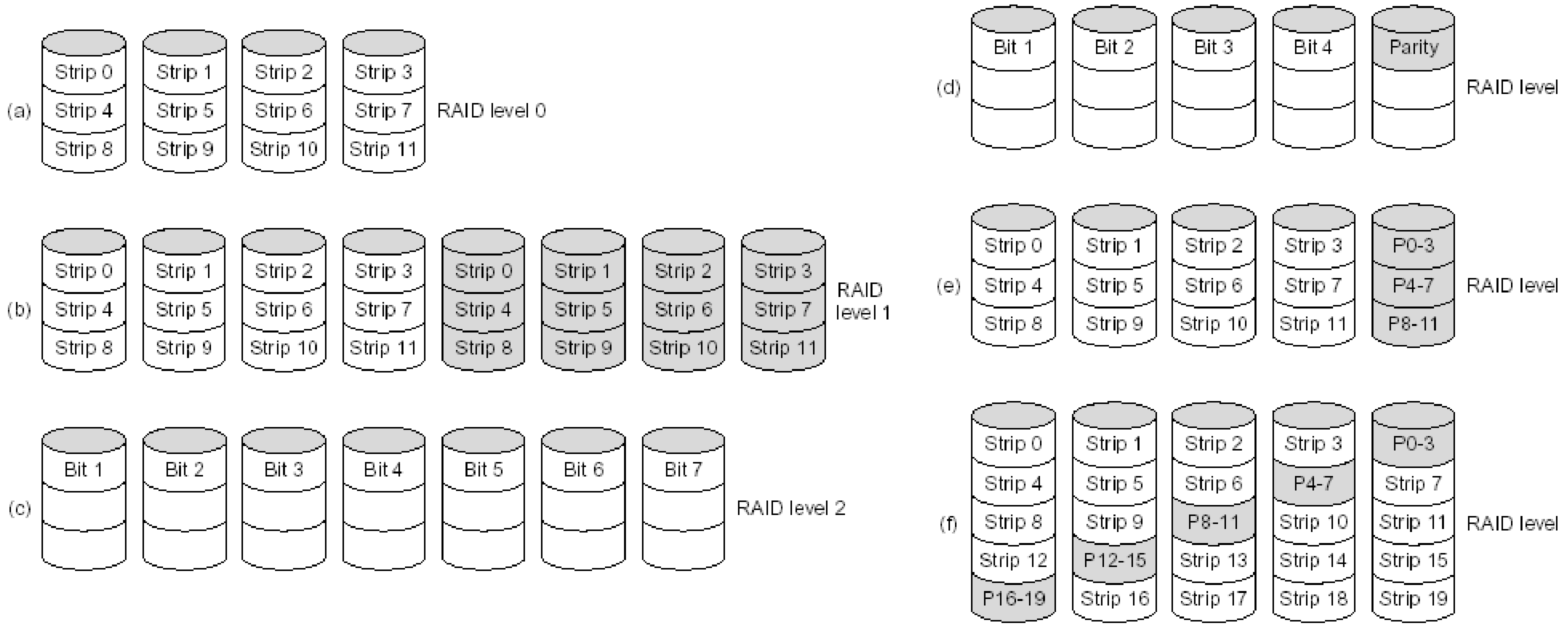
Use redundant disk capacity to respond to disk failure

- More disks $\Rightarrow$ lower mean-time-to-failure (**MTTF**)

RAID: Striping



RAID Levels



RAID levels with different properties in terms of

- performance characteristics
- level of redundancy
- degree of space efficiency (cost)

Some other these are of historic interest...

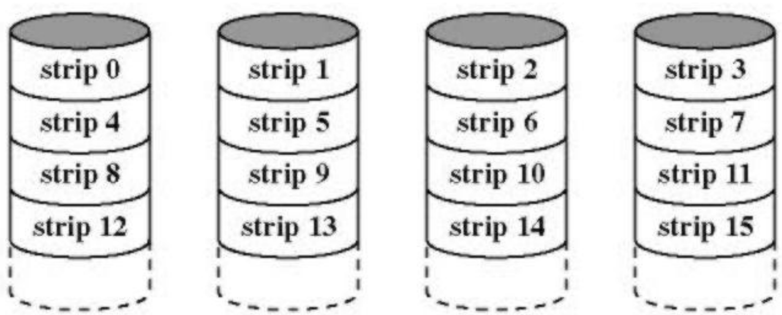
RAID Level 0 (Striping)

Use multiple disks and spread out data

Disks can seek/transfer data concurrently

- Also may balance load across disks

No redundancy ➔ no fault tolerance



RAID Level 1 (Mirroring)

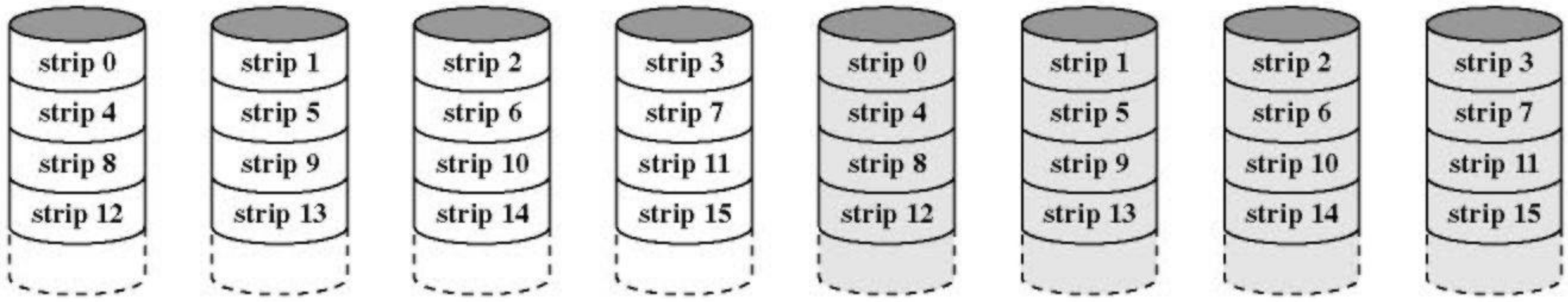
Mirror data across disks

Reads can be serviced by either disk (fast)

Writes update both disks in parallel (slower)

Failure recovery easy

- High storage overhead (high cost)



RAID Level 2 (Bit-Level Hamming)

Parallel access by striping at bit-level

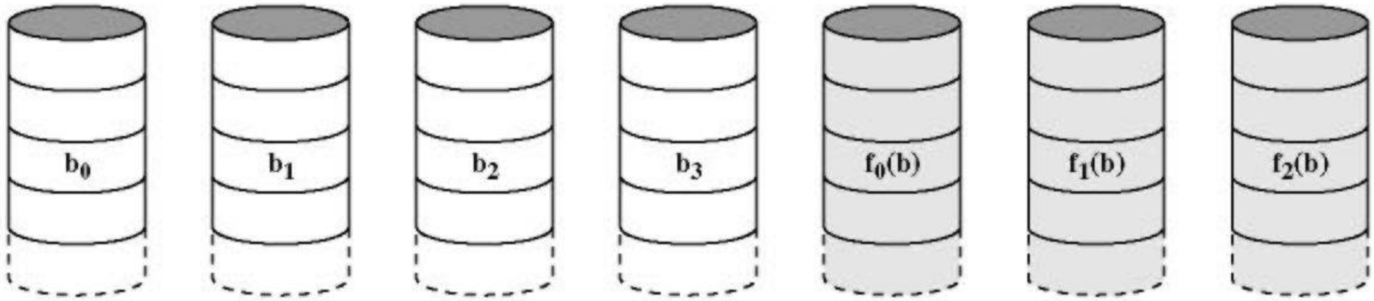
- Use Hamming error-correcting code (ECC)
- Corrects single-bit errors (and detect double-bit errors)

Very high throughput for reads/writes

- But all disks participate in I/O requests (no concurrency)
- **Read-modify-write cycle**

Only used if high error rates expected

- ECC disks become bottleneck
- High storage overhead



RAID Level 3 (Byte-Level XOR)

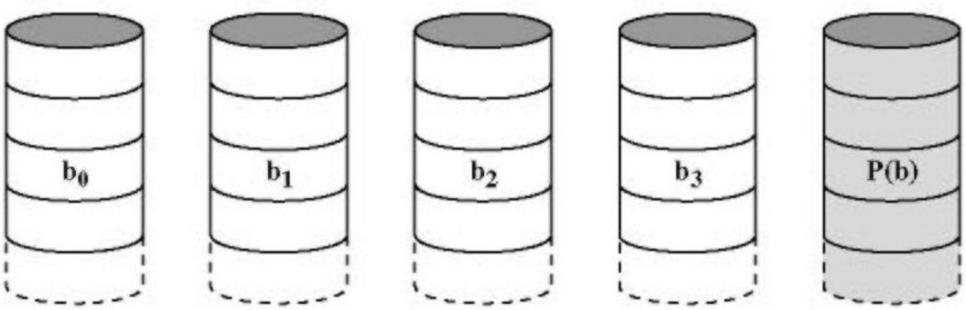
Only single parity strip used

Parity = data1 XOR data2 XOR data3 ...

- Reconstruct missing data from parity and remaining data

Lower storage overhead than RAID L2

- But still only one I/O request can take place at a time



RAID Level 4 (Block-Level XOR)

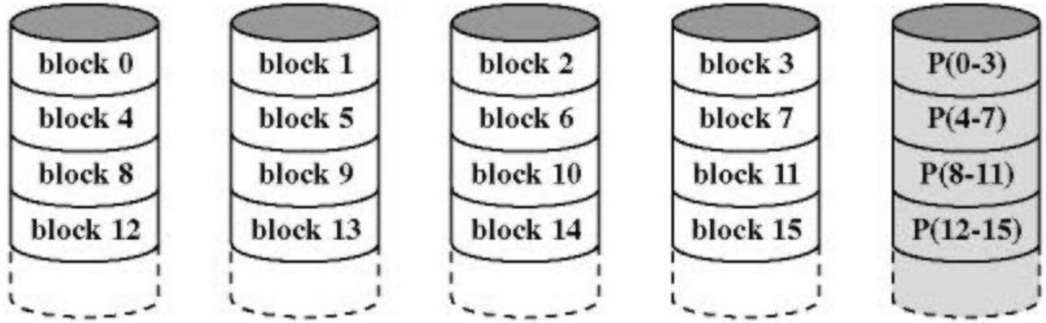
Parity strip handled on block basis

- Each disk operates independently

Potential to service multiple reads concurrently

Parity disk tends to become bottleneck

- Data and parity strips must be updated on each write



RAID Level 5 (Block-Level Distributed XOR)

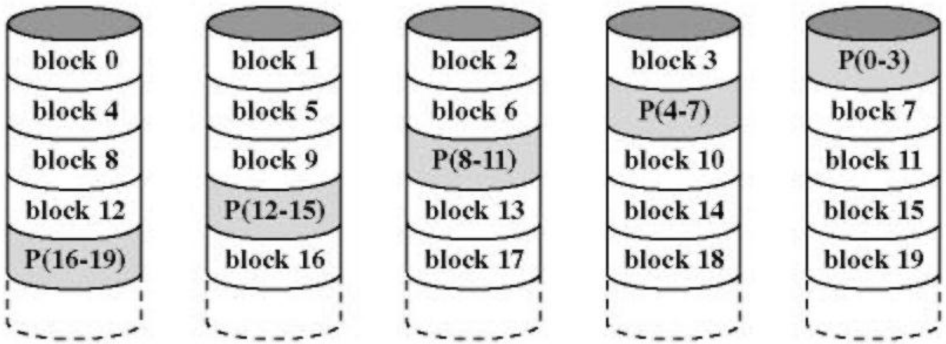
Like RAID 4, but distribute parity

- Most commonly used

Some potential for write concurrency

Good storage efficiency/redundancy trade-off

- Reconstruction of failed disk non-trivial (and slow)



RAID Summary

Category	Level	Description	I/O Data Transfer (read/write)	I/O Request Rate (reads/writes)
Striping	0	Non-redundant	+ / +	+ / +
Mirroring	1	Mirrored	+ / 0	+ / 0
Parallel access	2	Redundant via Hamming code	++ / ++	0 / 0
	3	Bit interleaved parity	++ / ++	0 / 0
Independent access	4	Block interleaved parity	+ / -	+ / -
	5	Block interleaved distributed parity	+ / -	+ / - or 0

better than single disk (+) / same (0) / worse (-)

Disk Caching

Disk Cache

Idea: Use main memory to improve disk access

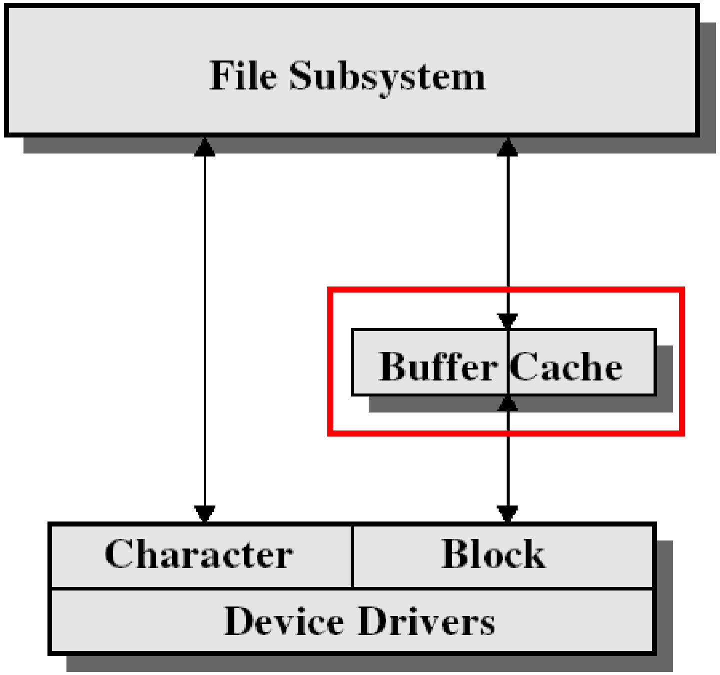
Buffer in main memory for disk sectors

- Contains copy of some sectors from disk
- OS manages disk in terms of **blocks**
  - Multiple sectors for efficiency
  - cf. Device Management (block devices)

Buffer uses finite space

- Need **replacement policy** when buffer full

Buffer Cache



Least Recently Used (LRU)

Replace block that was in cache longest with no references

Cache consists of stack of blocks

- Most recently referenced block on top of stack
- When block referenced (or brought into cache), place on top of stack
- Remove block at bottom of stack when new block brought in

Don't move blocks around in main memory

- Use stack of pointers instead

Problem: Doesn't keep track of block "popularity"

Least Frequently Used (LFU)

Replace block that has experienced fewest references

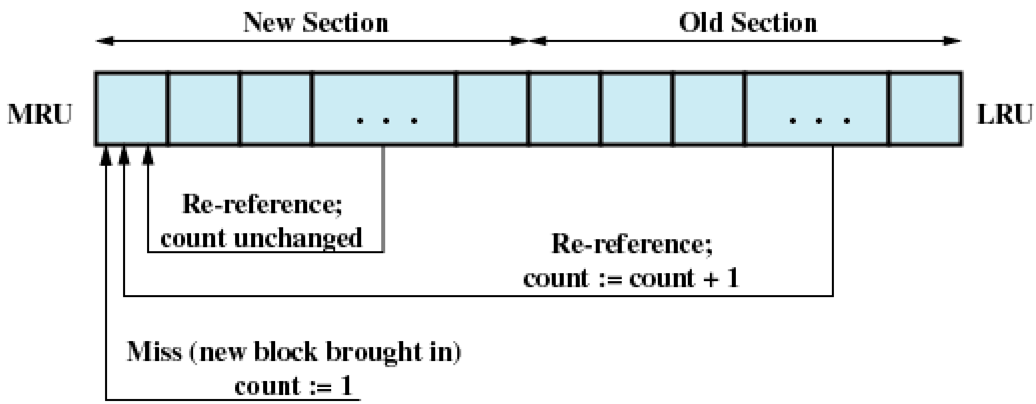
Counter associated with each block

- Counter incremented each time block accessed
- Block with smallest count selected for replacement

Some blocks may be referenced many times in short period of time

- Leads to misleading reference count
- Use **frequency-based replacement**

Frequency-Based Replacement



- Divide LRU stack into two sections: new and old
  - Block referenced ➔ move to top of stack
  - Only increment reference count if not already in new
- Problem: blocks "age out" too quickly (why?)
  - Use three sections and only replace blocks from old