Storage Devices

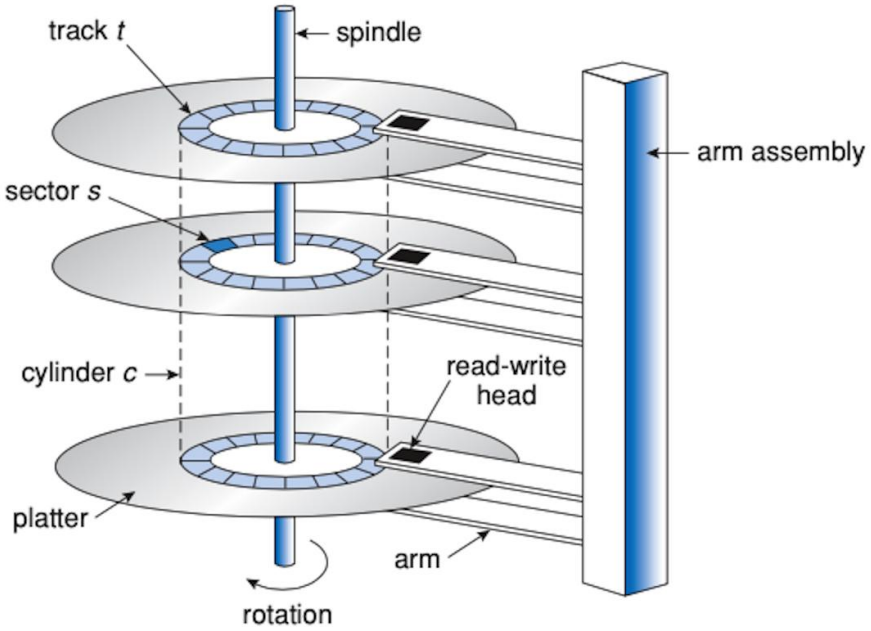
W4118 Operating Systems I

columbia-os.github.io

Credits to Jae and David Mazières

Hard Disk Drive (HDD)

- 2-5 aluminum **platters** attached to a rotating **spindle**
- The surfaces have a thin magnetic coating that stores bits
- The platters are divided into **cylinders/tracks** and **sectors** (512 bytes each)
- The platters rotate fast and the **arm** moves back and forth to cover the surface.
- The **read-write** head is used to access the data.



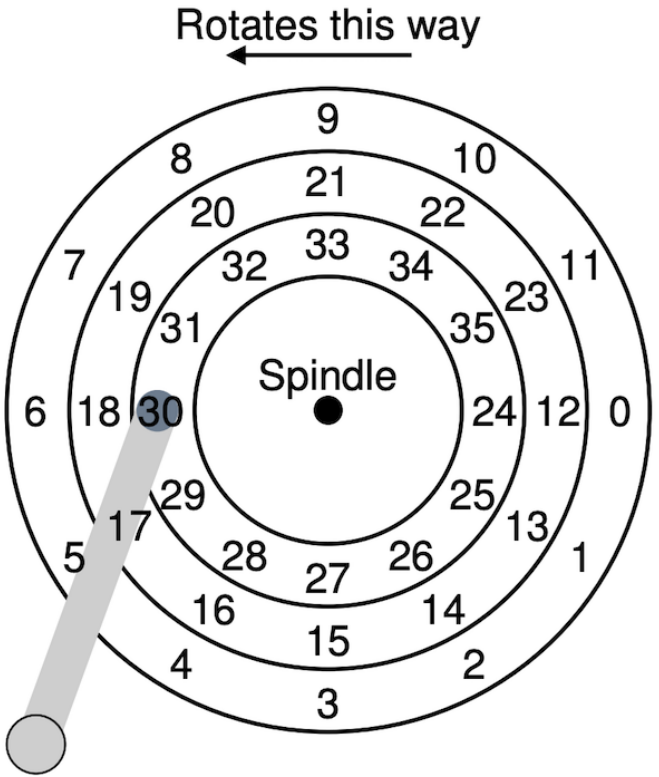
Accessing Data

Older HDDs required the OS to provide cylinder #, head #, sector # (**CHS**)

Newer devices expose a uniform **LBA** logical block address

The disk controller translates the LBA to CHS

OS views storage device as 1D-array of groups of sectors known as **blocks**



Performance Considerations

Reading a sector:

- **Seek time:** move disk arm to the correct track; the most expensive
  - requires acceleration, coasting, decelerating, and settling
- **Rotational delay:** wait for sectors to pass under the head
- **Transfer time:** read bits off the disk

Avoid expensive seek time by accessing disk sequentially (e.g. by interacting with adjacent sectors). Random access is 200-300x more expensive than sequential access. Data structures and algorithms are designed around this trade-off.

Seek Performance Considerations

Move head to specific track and keep it there

- Resist physical shocks, imperfect tracks, etc.

A seek consists of up to four phases:

- *speedup* – accelerate arm to max speed or half way point
- *coast* – at max speed (for long seeks)
- *slowdown* – stops arm near destination
- *settle* – adjusts head to actual desired track

Very short seeks dominated by settle time (~1 ms)

Short (200-400 cyl.) seeks dominated by speedup

- Accelerations of 40g (upto 500g)

Seek Performance Considerations

Head switches comparable to short seeks

- May also require head adjustment
- Settles take longer for writes than for reads – Why?

Disk keeps table of pivot motor power

- Maps seek distance to power and time
- Disk interpolates over entries in table
- Table set by periodic “thermal recalibration”
- But, e.g., ~500 ms recalibration every ~25 min bad

Seek Performance Considerations

Head switches comparable to short seeks

- May also require head adjustment
- Settles take longer for writes than for reads – Why?
  - If read strays from track, catch error with checksum, retry
  - If write strays, you've just clobbered some other track

Disk keeps table of pivot motor power

- Maps seek distance to power and time
- Disk interpolates over entries in table
- Table set by periodic "thermal recalibration"
- But, e.g., ~500 ms recalibration every ~25 min bad

Example: Hard Disk Performance (1)

- Seek
  - Position heads over cylinder, typically we assume 10msec avg:
    - 4 msecs for highend disk drives to
    - 15 msecs for mobile devices
- Rotational delay
  - Wait for a sector to rotate underneath the heads
  - Typically 7.14 – 2.0 ms (4,200 – 15,000 rpm)
    [or ½ rotation]
- Transfer bytes
  - Average transfer bandwidth
    (50-80 Mbytes/sec @4KB block sz)

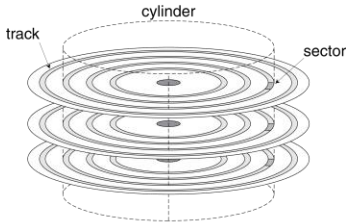
HDD spindle speed [rpm]	Average rotational latency [ms]
4,200	7.14
5,400	5.56
7,200	4.17
10,000	3.00
15,000	2.00

Example: Hard Disk Performance (2)

- Performance of transfer 1 Kbytes
  - Seek (5.3 ms) + half rotational delay (3ms) + transfer (0.04 ms)
  - Total time is 8.34ms or 120 Kbytes/sec!
- What block size can get 90% of the disk transfer bandwidth?

Disk Behaviors (example)

- There are more sectors on outer tracks than inner tracks
  - Read outer tracks: 37.4MB/sec
  - Read inner tracks: 22MB/sec
- Seek time and rotational latency dominate the cost of small reads
  - A lot of disk transfer bandwidth is wasted
  - Need algorithms to reduce seek time



Block Size (Kbytes)	% of Disk Transfer Bandwidth
1Kbytes	0.5%
8Kbytes	3.7%
256Kbytes	55%
1Mbytes	83%
2Mbytes	90%

Observations

- Getting first byte from disk read is slow
  - high latency
- Peak bandwidth high, but rarely achieved
- Need to mitigate disk performance impact
  - Do extra calculations to speed up disk access
    - Schedule requests to shorten seeks
  - Move some disk data into main memory – file system caching

File Access Patterns

- **Sequential access**
  - Data read/written in order (e.g., copy files)
  - Can be made very fast, if file data is collocated on disk

- **Random access**
  - Randomly address any block (e.g. update database record)
  - Difficult to make fast because of seek time and rotational delay

Disk Management (1)

Placement & ordering of requests a huge issue

- Sequential I/O much, much faster than random
- Long seeks much slower than short ones
- Power might fail any time, leaving inconsistent state

Must be careful about order for crashes

- More on this in next lectures

Disk Management (2)

Try to achieve contiguous accesses where possible

- E.g., make big chunks of individual files contiguous

Try to order requests to minimize seek times

- OS can only do this if it has multiple requests to order
- Requires disk I/O concurrency
- High-performance apps try to maximize I/O concurrency

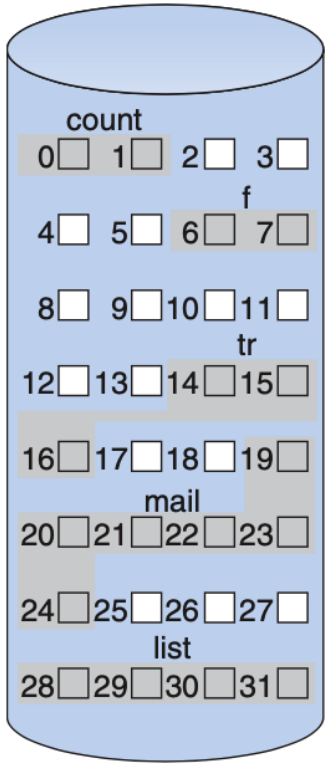
Next: How to place blocks and how to schedule concurrent requests

Allocation Strategies Considerations

- Internal/external fragmentation
- Easy to grow file over time?
- Fast to find data for sequential/random access?
- Easy to implement?
- Storage overhead of metadata and data structures?

Contiguous Allocation

- user specifies length, FS allocates space all at once
- find disk space by examining bitmap
- simple metadata: “base & limit” of file – starting location and size of file



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Contiguous Allocation

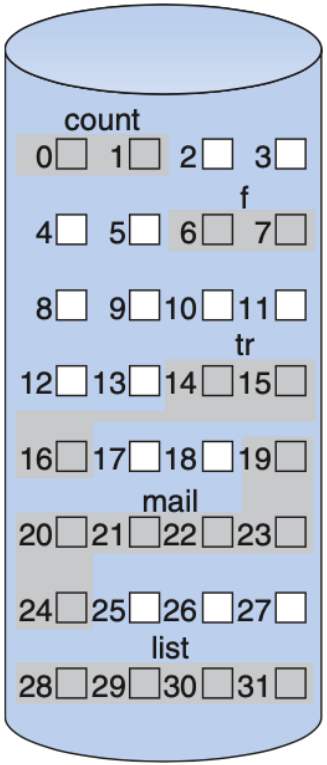
- user specifies length, FS allocates space all at once
- find disk space by examining bitmap
- simple metadata: “base & limit” of file – starting location and size of file

Pros

- Easy to implement
- Low storage overhead (start & length for each file)
- Fast sequential access because of contiguous blocks

Cons

- Suffers from external fragmentation as files are created and removed overtime
- Difficult to grow file because of contiguous requirement



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Extent-based Allocation

- Multiple contiguous regions per file (like segmentation for memory).
- Metadata is an array of extents, where each extent has a start and size..
- Easier to grow files compared to contiguous allocation, but still suffers from external fragmentation.
- You can check the extents used for a file

```
frankeh@lnx4:~$ sudo filefrag -v /boot/vmlinuz-6.8.0-106-generic
Filesystem type is: ef53
File size of /boot/vmlinuz-6.8.0-106-generic is 15034760 (3671 blocks of 4096 by
tes)
 ext:     logical_offset:        physical_offset: length:   expected: flags:
   0:        0..    1844:    7562955..   7564799:   1845:
   1:     1845..    2047:    7599630..   7599832:    203:    7564800:
   2:     2048..    3423:    7725056..   7726431:   1376:    7599833:
   3:     3424..    3670:    5798400..   5798646:    247:    7726432: las
/boot/vmlinuz-6.8.0-106-generic: 4 extents found
```

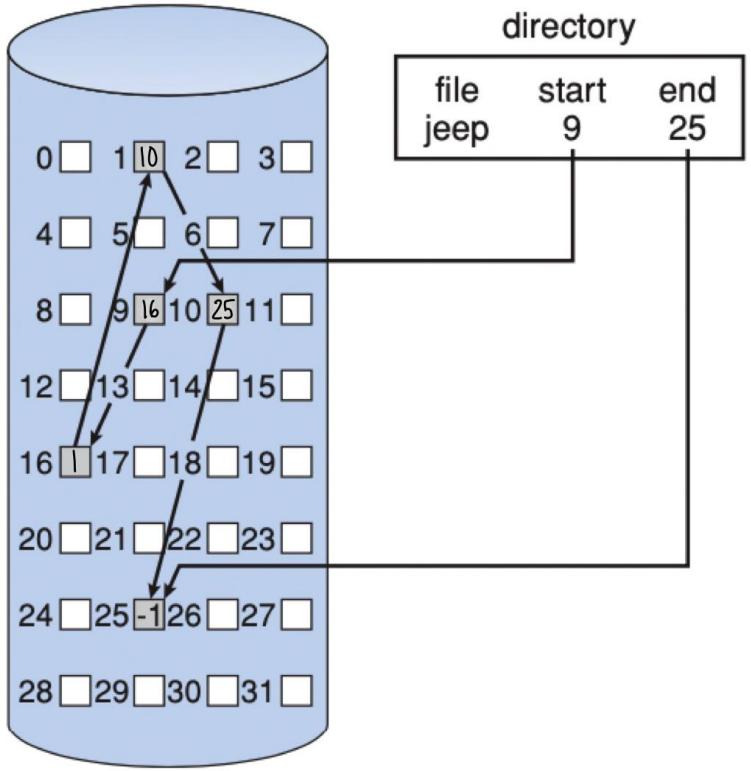
```
root@lnx4:/home# uniq -c /tmp/myfilestats
  39275 1
     10 2
      4 3
      1 4
      1 25
```

Linked Allocation

All data blocks are part of a linked list

Reserve some metadata bytes at the beginning of the data block for next pointer

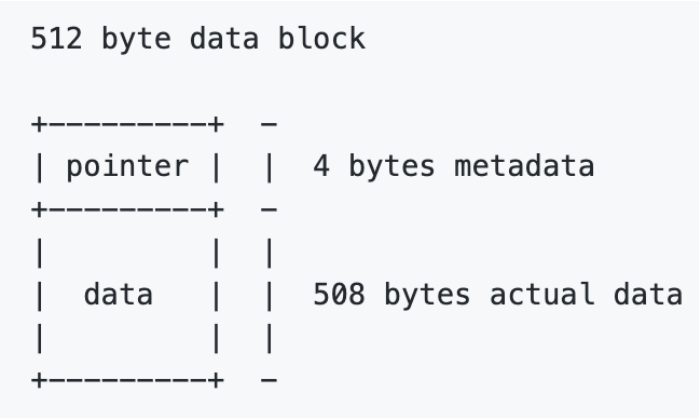
```
512 byte data block

+---------+  -
| pointer |  |  4 bytes metadata
+---------+  -
|         |  |
|  data   |  |  508 bytes actual data
|         |  |
+---------+  -
```



Linked Allocation

All data blocks are part of a linked list

Reserve some metadata bytes at the beginning of the data block for next pointer



Pros

- No external fragmentation – similar to paging
- File can easily grow without limits
- Easy to implement

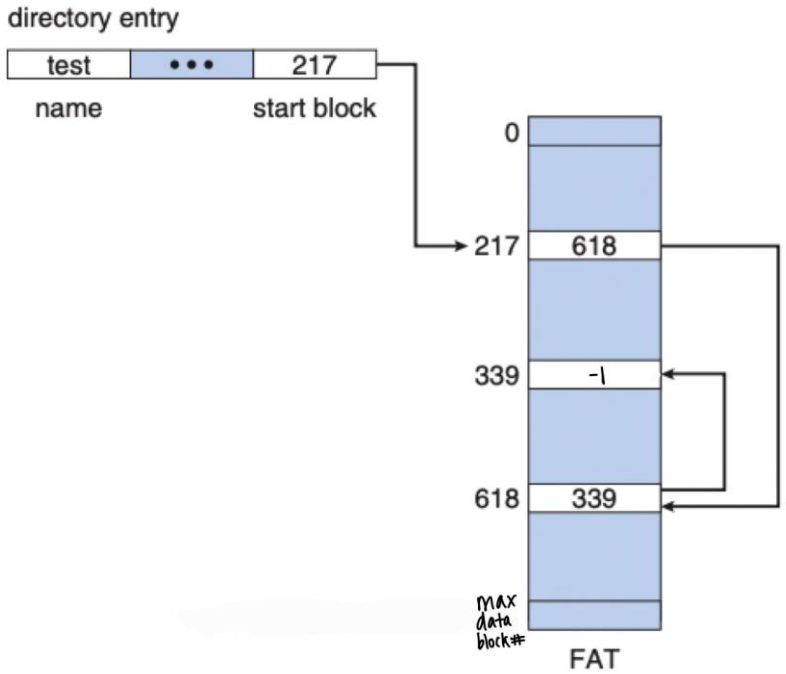
Cons

- Potentially slow sequential access: lots of seeking since blocks need not be contiguous
- Difficult to compute random access, need to walk the linked list
- Some storage overhead (one pointer per block)

File Allocation Table (FAT)

Store linked-list pointers outside of data block in a dedicated pointer table.

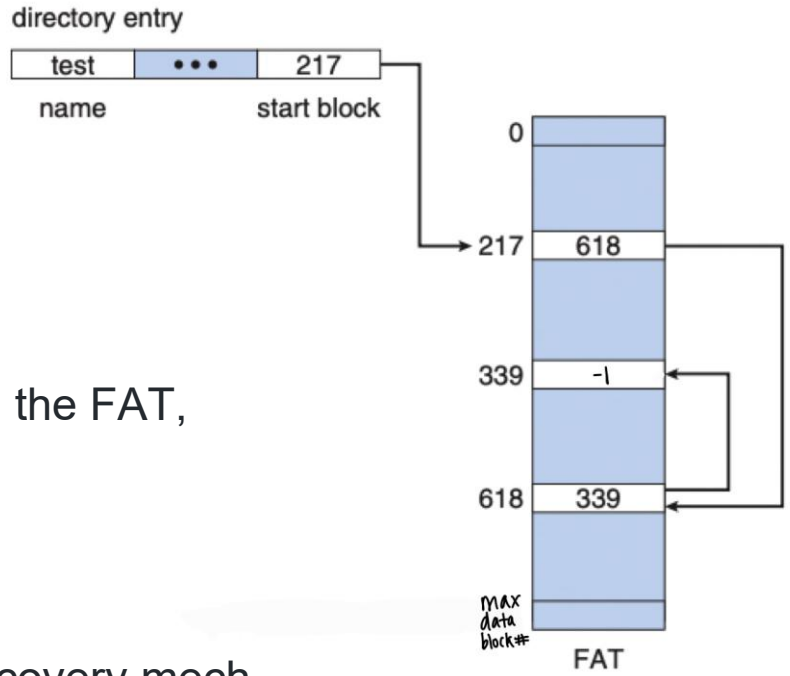
Used in MSDOS and Windows



File Allocation Table (FAT)

Store linked-list pointers outside of data block in a dedicated pointer table.

Used in MSDOS and Windows



Pros

- Better random access: only need to search the FAT, which can be cached in memory

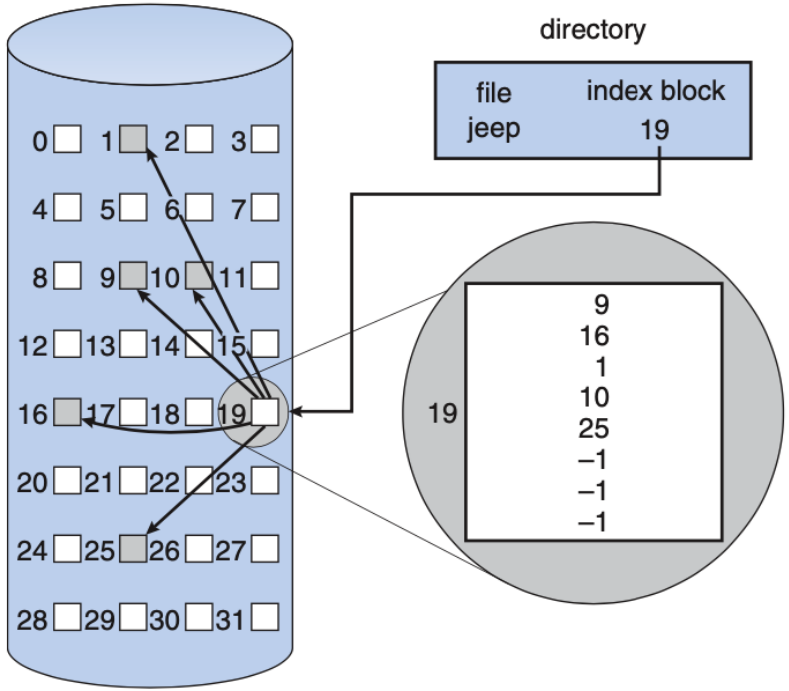
Cons

- Sequential access still slow
- What happens if system crashes? Need recovery mech.

Indexed Allocation

File metadata: array of pointers (index) to data blocks

- file size is limited by number of pointers
- allocate blocks on demand (pointers are marked as invalid in the meantime)



Indexed Allocation

File metadata: array of pointers (index) to data blocks

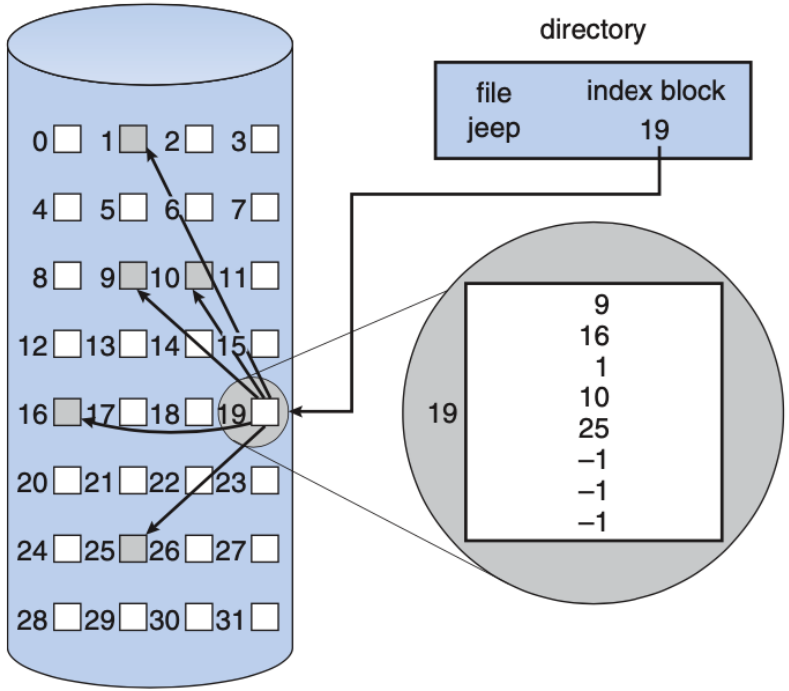
- file size is limited by number of pointers
- allocate blocks on demand (pointers are marked as invalid in the meantime)

Pros

- No external fragmentation
- File can easily grow within the limit
- Fast random access: consult index

Cons

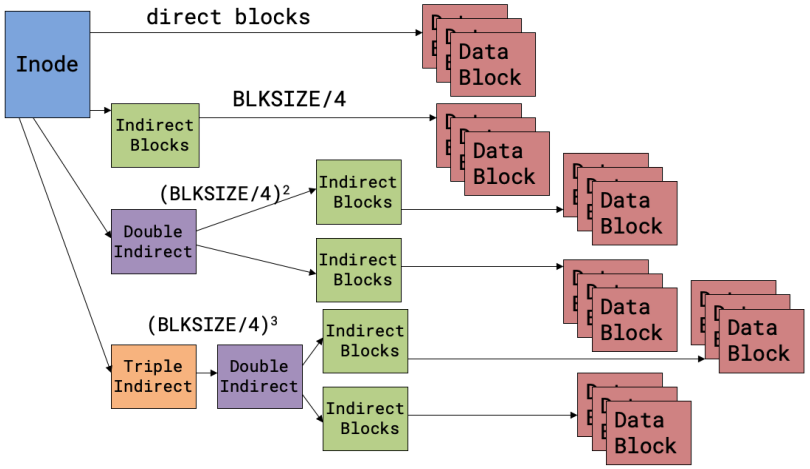
- Large storage overhead for index
- Sequential access can still be slow because of disk seeks



Multi-level indexed Allocation

- Instead of having a single index block of having direct data block pointers, maintain indirect pointers to have larger file size. Used in UNIX FFS, Linux ext2/ext3

- In addition to direct blocks, like we saw in indexed allocation, inodes (index nodes) can also refer to:
  - **indirect block:** contains pointers to other data blocks
  - **double indirect block:** contains pointers to other indirect blocks
  - **triple indirect block:** contains pointers to other double indirect blocks



Max file size: `(NDIRECT + BLKSIZE/4 + (BLKSIZE/4)^2 + (BLKSIZE/4)^3) * BLKSIZE`

\+ **Huge file size limit**

Observations

- Getting first byte from disk read is slow
  - high latency
- Peak bandwidth high, but rarely achieved
- Need to mitigate disk performance impact
  - Do extra calculations to speed up disk access
    - Schedule requests to shorten seeks
    - We are assuming that multiple requests are pending
  - Move some disk data into main memory – file system caching

Linux Block I/O Layer

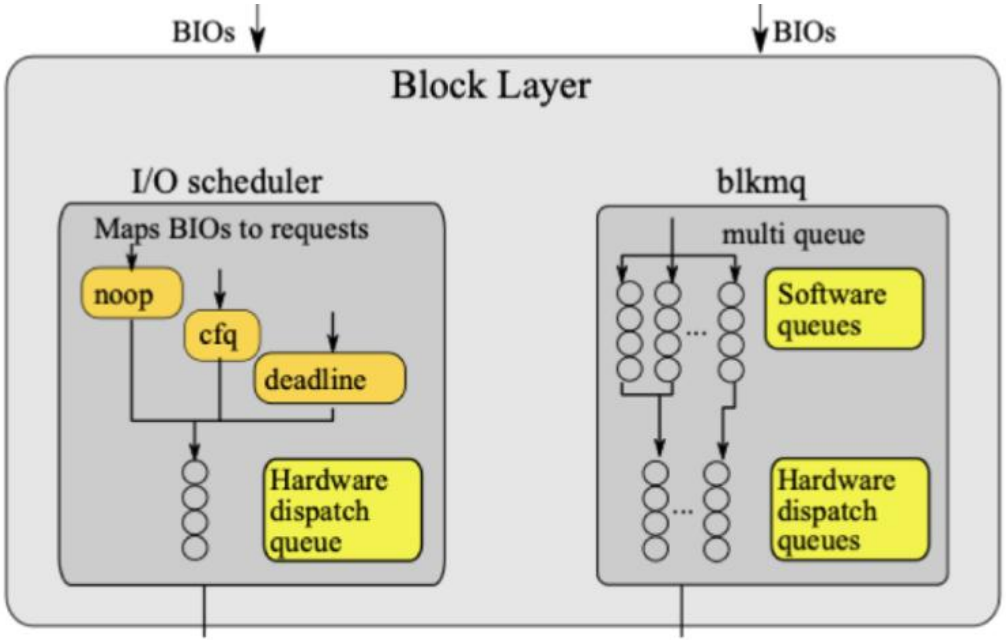


Image Credit: Thomas-Krenn AG, *Linux Storage Stack Diagram*

Scheduling: FCFS

“First Come First Served”

- Process disk requests in the order they are received

Pros

Cons

Scheduling: FCFS

“First Come First Served”

- Process disk requests in the order they are received

Pros

- Easy to implement
- Good fairness

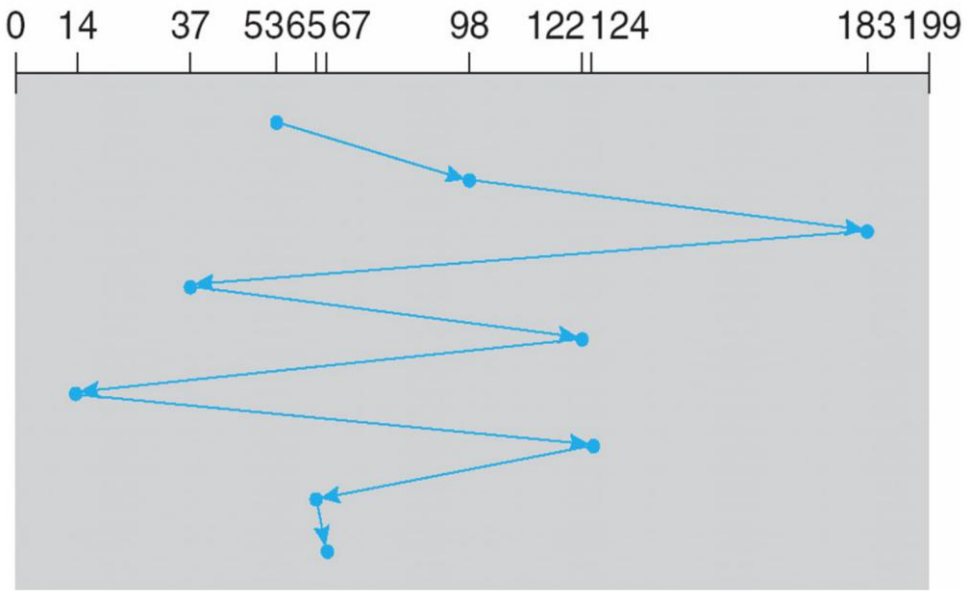
Cons

- Cannot exploit request locality
- Increases average latency, decreasing throughput

FCFS Example

queue = 98, 183, 37, 122, 14 , 124, 65, 67

head starts at 53



Shortest Positioning Time First (SPTF)

Shortest positioning time first (SPTF)

- Always pick request with shortest seek time

Also called Shortest Seek Time First (SSTF)

Pros

Cons

Shortest Positioning Time First (SPTF)

Shortest positioning time first (SPTF)

- Always pick request with shortest seek time

Also called Shortest Seek Time First (SSTF)

Improvement?

Pros

- Exploits locality of disk requests
- Higher throughput

Cons

- Starvation
- Don’t always know what request will be fastest

Shortest Positioning Time First (SPTF)

Shortest positioning time first (SPTF)

- Always pick request with shortest seek time

Also called Shortest Seek Time First (SSTF)

Pros

- Exploits locality of disk requests
- Higher throughput

Cons

- Starvation
- Don’t always know what request will be fastest

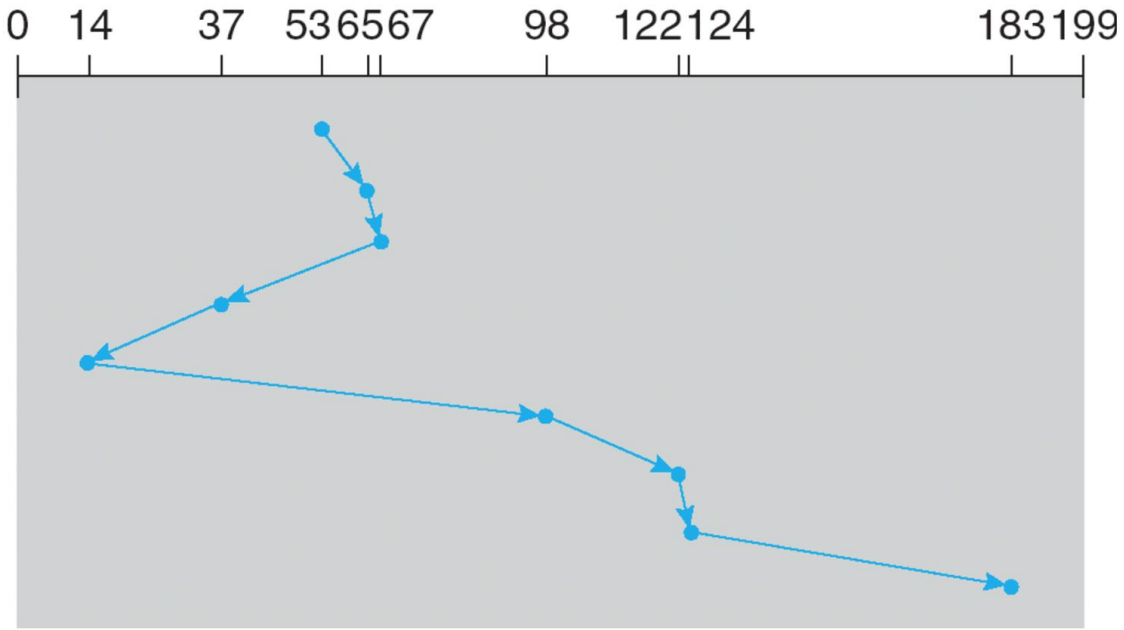
Improvement?

- Give older requests higher priority
- Adjust effective seek time with weighing factor

SPTF Example

queue = 98, 183, 37, 122, 14 , 124, 65, 67

head starts at 53



Elevator Scheduling

Sweep across disk, servicing all requests passed

- Like SPTF, but next seek must be in same direction
- Switch directions only if no further requests

Pros

Cons

Elevator Scheduling

Sweep across disk, servicing all requests passed

- Like SPTF, but next seek must be in same direction
- Switch directions only if no further requests

Pros

- Takes advantage of locality
- Bounded waiting

Cons

- Might miss locality SPTF could exploit

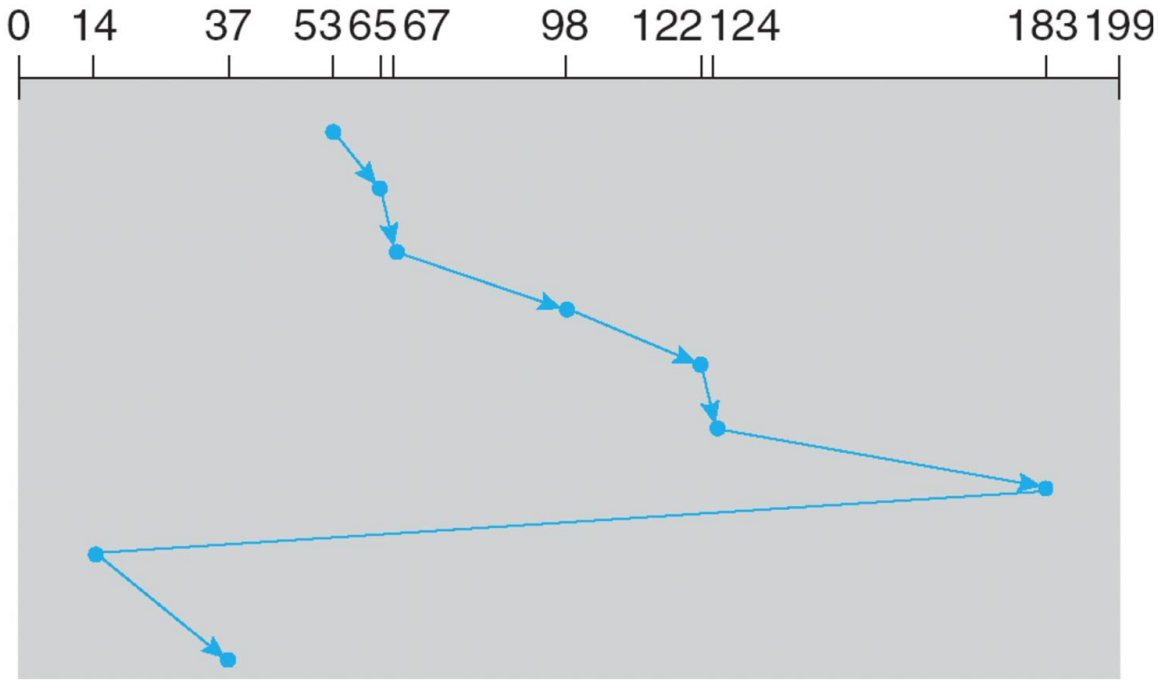
CSCAN: Only sweep in one direction – Commonly used in Linux

Also called LOOK/CLOOK

Elevator Scheduling Example

queue = 98, 183, 37, 122, 14 , 124, 65, 67

head starts at 53

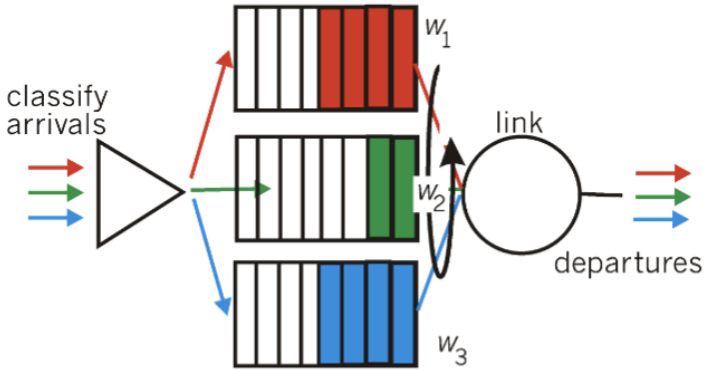


Other modern schedulers (1)

- Deadline Scheduler
  - Guarantees a start time for each I/O requests → deadline
  - Read requests have an expiration time of 500 ms,
    write requests expire in 5 seconds.
  - Maintains read + write deadline queues + sorted (by sector) queue
  - If a deadline expired then it is served otherwise, a batch is served from sorted queue.

Other modern schedulers (2)

- CFQ (completely fair queueing)
  - places requests submitted into a number of per-process queues
  - allocates timeslices for each queue to access the disk.
  - The length of the time slice and the number of requests a queue is allowed to submit depends on the I/O priority of the given process.
  - Implicitly provides “idle” time at end of I/O request to allow subsequent requests to be issued and bundled



Solid State Drive (SSD)

- Based on flash-NAND technology; similar to RAM but non-volatile

Composed of a grid of cells. Each cell stores a bit of data by charging the cell with low (0) or high (1) voltage. More complicated cells can store several bits by adding more granularity to the voltage charge

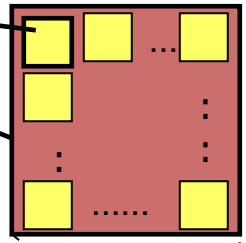
- Cells are grouped into pages (1KB-8KB)
- Pages are grouped into blocks (100-1000 pages).

Don’t confuse SSD terminology with memory management terminology, they are referring to different things!

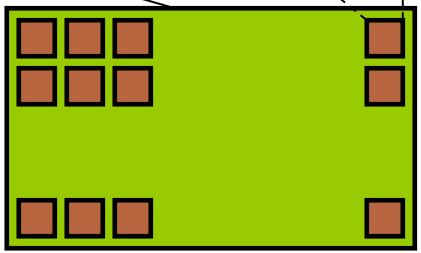
Organization of a Typical FLASH Chip

- 1 Page = 8KB-16KB

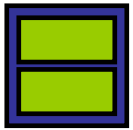
- 1 Block = 1MB-16MB
  (128-256pages)



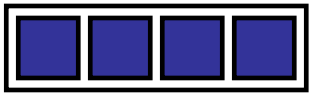
- **1 Plane = O(1k) Blocks**
  ~= 128GB



- **1 Die = 2 Planes**
  = 512GB



- **1 Flash Chip = 2TB**
  (4 Dies)



- **Functions supported**
  - Read
  - Erase
  - Program

Accessing an SSD – Reads :)

Reads: simply read the voltage levels of cells.

- No moving parts reduces access time and results in a less fragile and power-consumptive device
- Faster sequential read/writes, where random I/O is magnitudes faster

Takes ~25 µsec + time to get data off chip

SSD read accesses are at page level (~8-16 KB)

Accessing an SSD – Writes :(

Writes: much much harder

- Back-up a block (not page 1MB)
- Erase the block (on the device)
- Change some pages within the block in memory
- Write-out the entire block (reprogram the block on the disk)

Very expensive (2 msec)

Programming pre-erased block requires moving data to internal buffer, then 200 – 800 µsec

Repeated writes on the same cells will wear them out by leaving behind residual charge over time

Flash Translation Layer (FTL)

Translates LBA according to device geometry and helps prevent device wearout by “spreading out” the writes over the entire device by implementing wear-leveling.

Let’s say you have a “hot” file, how can you do this?

Flash Translation Layer (FTL)

Translates LBA according to device geometry and helps prevent device wearout by “spreading out” the writes over the entire device by implementing wear-leveling.

Let’s say you have a “hot” file, how can you do this?

- **log-structured I/O:** Instead of treating files as fixed regions of the device, store “snapshots” of the file across the device. The latest snapshot reflects the latest version of the file.
- periodically move data around (even read-only data) to help with wear-leveling.

Write amplification: writes issued to the device end up causing more internal writes.

Log-structured I/O

- This concept is not unique to SSDs – file systems were also designed in this way to maximize cheap sequential access on HDDs.
- For SSDs, this means don’t update blocks in-place, we copy them and write the new version somewhere else so that no one block is overused.
- Requires garbage collection – clearing out older snapshots so the blocks can be reused.

FTL straw man: in-memory map

Keep in-memory map of logical → physical page #

- On write, pick unused page, mark previous physical page free
- Repeated writes of a logical page will hit different physical pages

Idea: Put header on each page, scan all headers on power-up:

⟨logical page #, Allocated bit, ,Written bit, Obsolete bit⟩

- A-W-O = 0-0-0: free page
- A-W-O = 1-0-0: about to write page
- A-W-O = 1-1-0: successfully written page
- A-W-O = 1-1-1: obsolete page (can erase block without copying)

Why the 1-0-0 state?

What's wrong still?

FTL straw man: in-memory map

Keep in-memory map of logical → physical page #

- On write, pick unused page, mark previous physical page free
- Repeated writes of a logical page will hit different physical pages

Idea: Put header on each page, scan all headers on power-up:

⟨logical page #, Allocated bit, ,Written bit, Obsolete bit⟩

- A-W-O = 0-0-0: free page
- A-W-O = 1-0-0: about to write page
- A-W-O = 1-1-0: successfully written page
- A-W-O = 1-1-1: obsolete page (can erase block without copying)

Why the 1-0-0 state? After power failure partly written ≠ free

What's wrong still?

FTL straw man: in-memory map

Keep in-memory map of logical → physical page #

- On write, pick unused page, mark previous physical page free
- Repeated writes of a logical page will hit different physical pages

Idea: Put header on each page, scan all headers on power-up:

⟨logical page #, Allocated bit, ,Written bit, Obsolete bit⟩

- A-W-O = 0-0-0: free page
- A-W-O = 1-0-0: about to write page
- A-W-O = 1-1-0: successfully written page
- A-W-O = 1-1-1: obsolete page (can erase block without copying)

Why the 1-0-0 state? After power failure partly written ≠ free

What's wrong still?

- FTL requires a lot of RAM on device, plus time to scan all headers
- Some blocks still get erased more than others (w. long-lived data)
- Blocks with obsolete pages may also contain live pages

More realistic FTL

Store the FTL map in the flash device itself

- Add one header bit to distinguish map page from data page
- Logical read may miss map cache, require 2 flash reads
- Keep smaller “map-map” in memory, cache some map pages

Must garbage-collect blocks with obsolete pages

- Copy live pages to a new block, erase old block
- Always need free blocks, can’t use 100% physical storage

Problem: write amplification

- Small random writes punch holes in many blocks
- If small writes require garbage-collecting a 90%-full block. . . means you are writing 10× more physical than logical data!

Must also periodically re-write even blocks w/o holes

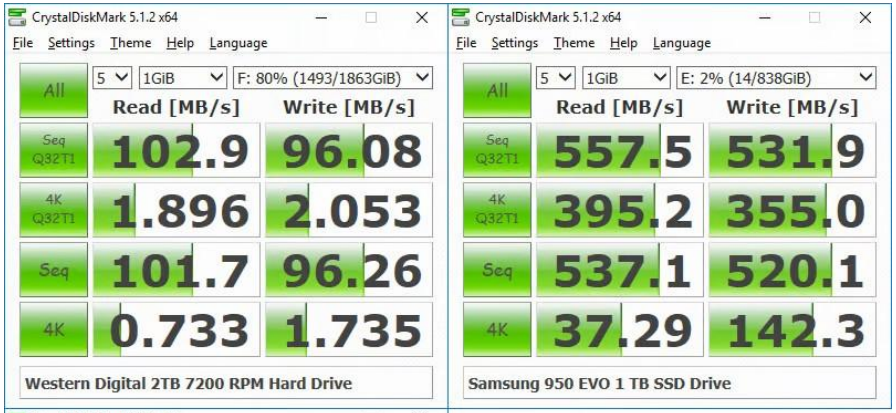
- *Wear leveling* ensures active blocks don’t wear out first

Disk Scheduling with SSDs

- Disk scheduling discussed earlier are largely not useful with SSD:
  - No mechanical based seek times -> reorders not required

- Schedulers used:
  - Noop → pass request directly to the device
  - MQ-Deadline → see before
  - Wear Level Focus → write coalescing

Performance comparison (2020 ☹)



Performance Summary:

Hard Drive: 103 MB/sec read, 96 MB/sec write speed
SSD Drive: 558 MB/sec read, 532 MB/sec write speed
M.2 Drive: 2591 MB/sec read, 1544 MB/sec write speed

In Read Performance:

SSD is 5x Faster than HDD
M.2 is 5x Faster than SSD
M.2 is 25x Faster than HDD

Summary: SSDs vs. HDDs

- **Pros**
  - No moving parts – less fragile, less power hungry
  - Faster sequential read/write
  - Orders of magnitude faster random read/write
- **Cons**
  - Expensive
  - Slow erase
  - Limited life span (though with wear leveling often exceeds HDD life)

Dealing with Hard Drive Errors

What are common hard drive errors:

- Transient checksum error : caused by dust on the head
- Seek error : the arm sent to cylinder 6 but it went to 7
- Permanent checksum error : disk block physically damaged
- Programming error : request for nonexistent sector
- Controller error : controller refuses to accept commands

Persistent Hard Drive Errors can lead to catastrophic data loss !

Recovery from data failures → Redundancy (RAID)

- *Redundant Array of Inexpensive Disks* OR

  Redundant Array of Independent Disks

- Use parallel processing to speed up CPU performance
- Use parallel I/O to improve disk performance, reliability (1988, Patterson)
- Design new class of I/O devices called RAID – Redundant Array of Inexpensive Disks (also Redundant Array of Independent Disks)
- Use the RAID in OS as a SLED (Single Large Expensive Disk), but with better performance and reliability

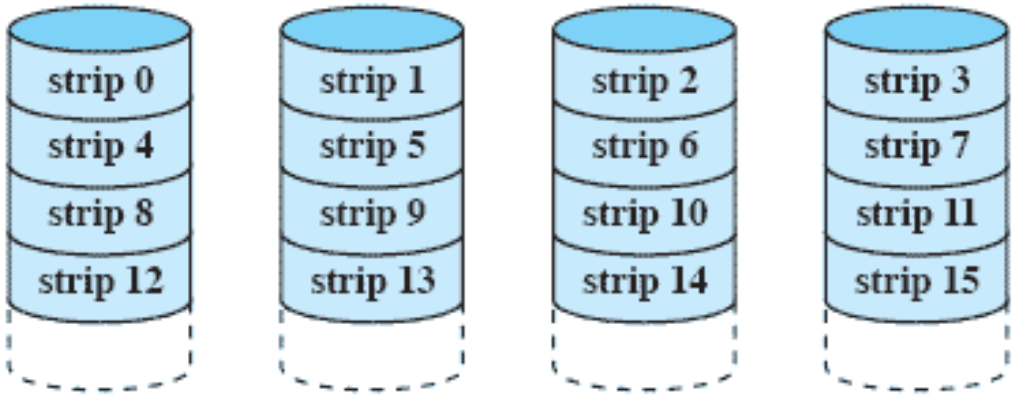
RAID (2)

- RAID consists of RAID SCSI controller plus a box of SCSI disks
- Data are divided into strips and distributed over disks for parallel operation
- RAID 0 … RAID 6 levels
- RAID 0 organization writes consecutive strips over the drives in round-robin fashion – operation is called striping
- RAID 1 organization uses striping and duplicates all disks
- RAID 2 uses words, even bytes and stripes across multiple disks; uses error codes, hence very robust scheme
- RAID 3, 4, 5, 6 alterations of the previous ones
- RAID 10 (aka Raid 1+0) combines disk mirroring (RAID-1) with striping (RAID-0)

Small Computer System Interface (**SCSI**, /ˈskʌzi/ ***SKUZ**-ee*) is a set of standards for physically connecting and transferring data between computers and peripheral devices. The SCSI standards define commands, protocols, electrical and optical interfaces. (source Wikipedia)

RAID Level 0

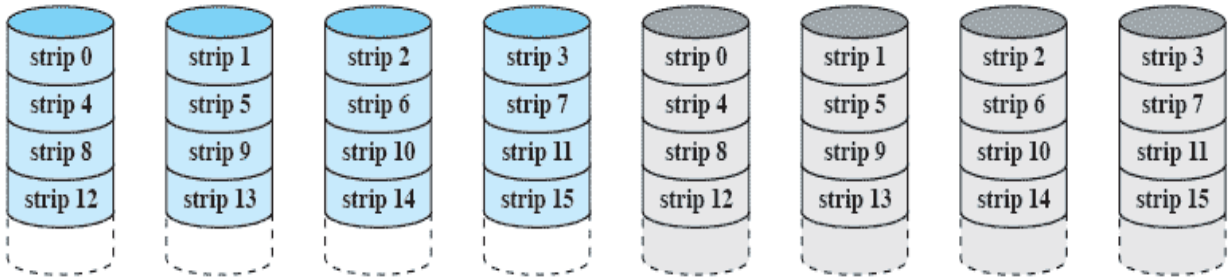
- Not a true RAID because it does not include redundancy to improve performance or provide data protection
- User and system data are distributed across all of the disks in the array
- Logical disk is divided into strips



(a) RAID 0 (non-redundant)

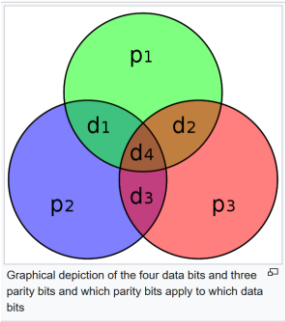
RAID Level 1

- Redundancy is achieved by the simple expedient of duplicating all the data
- There is no “write penalty”
- When a drive fails the data may still be accessed from the second drive
- Principal disadvantage is the cost



(b) RAID 1 (mirrored)

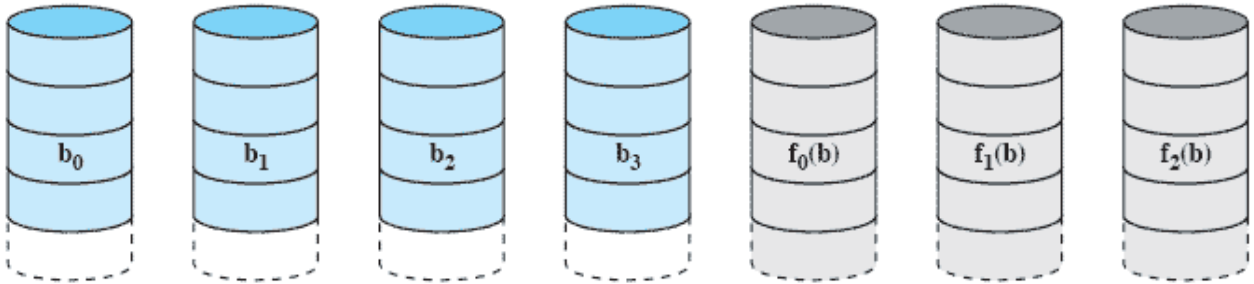
RAID Level 2



Graphical depiction of the four data bits and three parity bits and which parity bits apply to which data bits

Requires > log(N) redundant disks

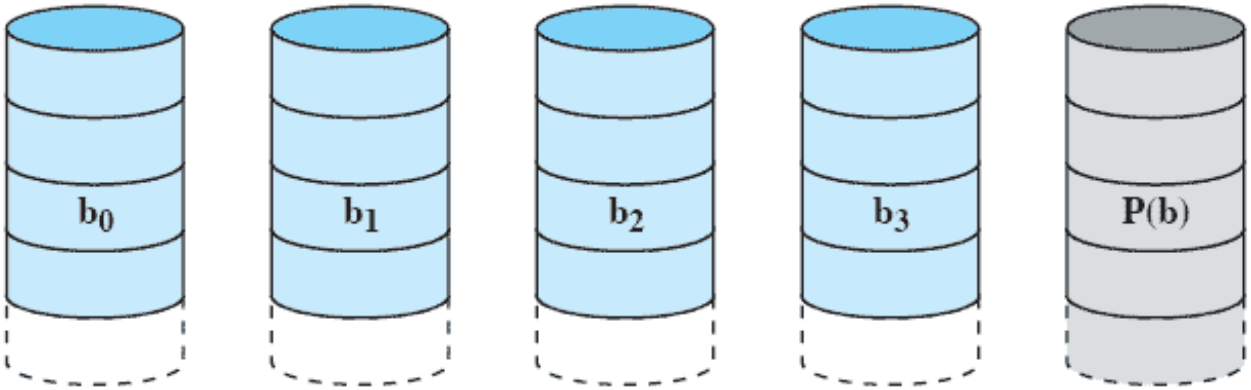
- Makes use of a parallel access technique, all disks participate
- **Data striping is used, strips are very small, sometimes byte/words**.
- Typically a Hamming code is used
- Effective choice in an environment in which many disk errors occur
- Can correct-single bit, detect 2-bit



(c) RAID 2 (redundancy through Hamming code)

RAID Level 3

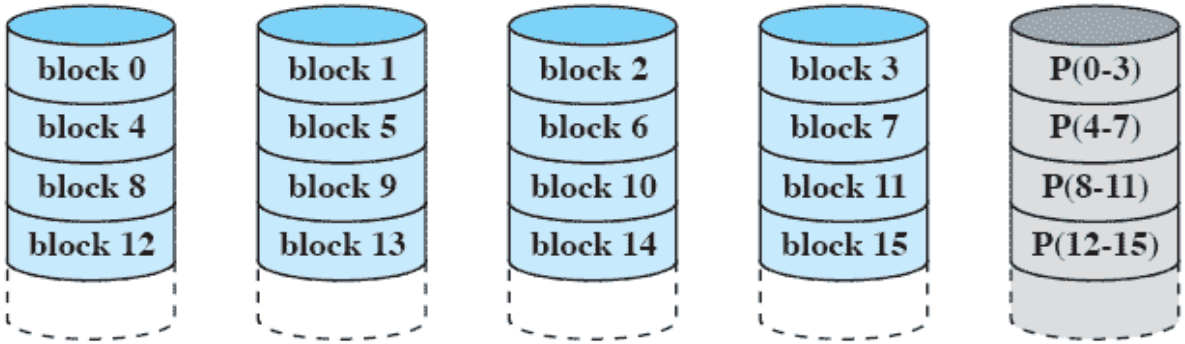
- Similar to Level2, but requires only a single redundant disk, no matter how large the disk array
- Employs parallel access, with data distributed in small strips
- Can achieve very high data transfer rates, but only one I/O at a time



(d) RAID 3 (bit-interleaved parity)

RAID Level 4

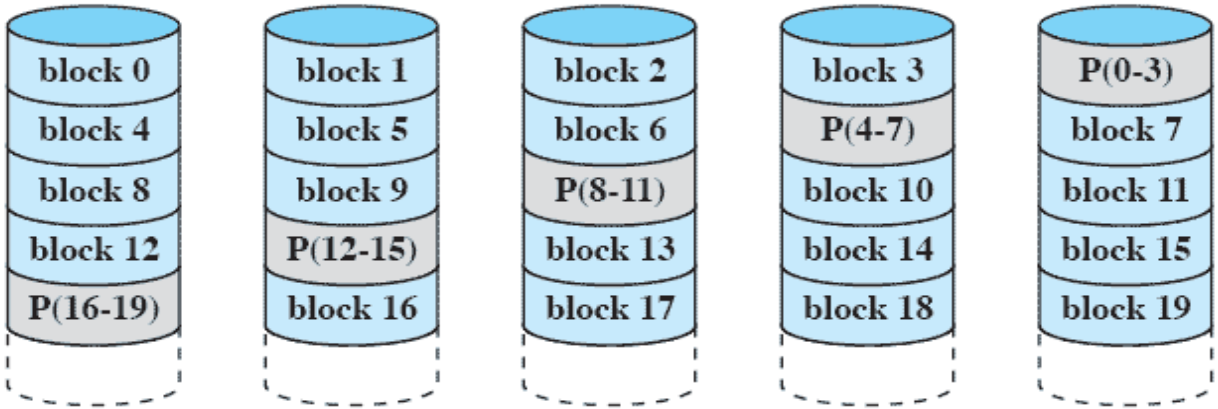
- Makes use of an independent access technique and strips are large.
- A bit-by-bit parity strip is calculated across corresponding strips on each data disk, and the parity bits are stored in the corresponding strip on the parity disk
- Involves a write penalty when an I/O



(e) RAID 4 (block-level parity)

RAID Level 5

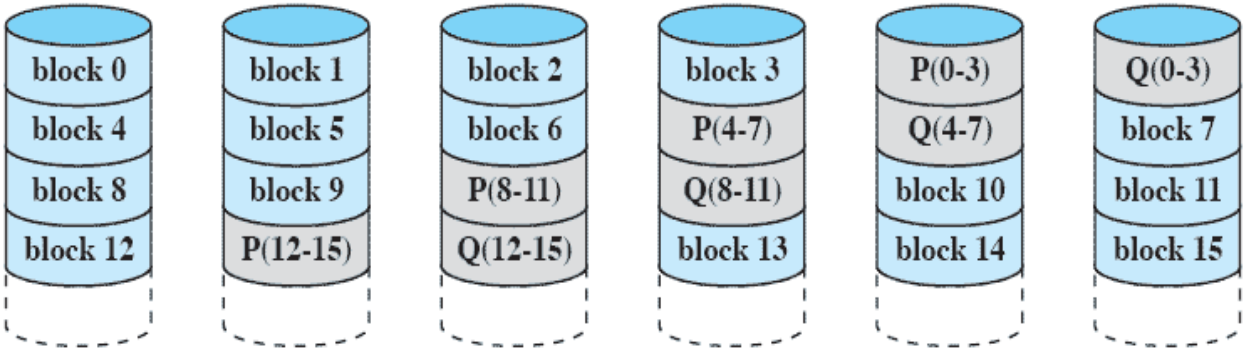
- Similar to RAID-4 but distributes the parity bits across all disks
- Typical allocation is a round-robin scheme
- Has the characteristic that the loss of any one disk does not result in data loss



(f) RAID 5 (block-level distributed parity)

RAID Level 6

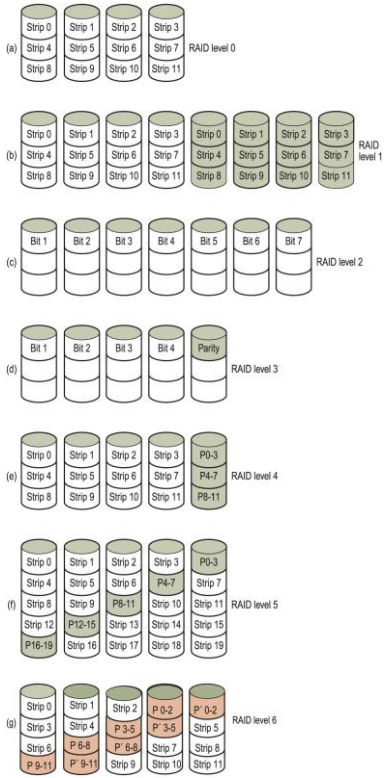
- Two different parity calculations are carried out and stored in separate blocks on different disks
- Provides extremely high data availability
- Incurs a substantial write penalty



(g) RAID 6 (dual redundancy)

RAID Levels Summary

Category	Level	Description	Disks required	Data availability	Large I/O data transfer capacity	Small I/O request rate
Striping	0	Nonredundant	N	Lower than single disk	Very high	Very high for both read and write
Mirroring	1	Mirrored	$2N$	Higher than RAID 2, 3, 4, or 5; lower than RAID 6	Higher than single disk for read; similar to single disk for write	Up to twice that of a single disk for read; similar to single disk for write
Parallel access	2	Redundant via Hamming code	$N + m$	Much higher than single disk; comparable to RAID 3, 4, or 5	Highest of all listed alternatives	Approximately twice that of a single disk
	3	Bit-interleaved parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 4, or 5	Highest of all listed alternatives	Approximately twice that of a single disk
Independent access	4	Block-interleaved parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 3, or 5	Similar to RAID 0 for read; significantly lower than single disk for write	Similar to RAID 0 for read; significantly lower than single disk for write
	5	Block-interleaved distributed parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 3, or 4	Similar to RAID 0 for read; lower than single disk for write	Similar to RAID 0 for read; generally lower than single disk for write
	6	Block-interleaved dual distributed parity	$N + 2$	Highest of all listed alternatives	Similar to RAID 0 for read; lower than RAID 5 for write	Similar to RAID 0 for read; significantly lower than RAID 5 for write



N = number of data disks; m proportional to log N

Logical Volume Manager (LVM)

- Physical Volume ~= HDD or SSD or partition
- LVM allows to combine a set of physical volumes to be treated as a volume group (via software)
- LVM allows flexible creation of logical volumes (think virtual disks) out of the volume group
- I/O requests are directed to correct physical disk