CSCC 69H3

Operating Systems

Winter 2017

Professor Sina Meraji

U of T

Announcement

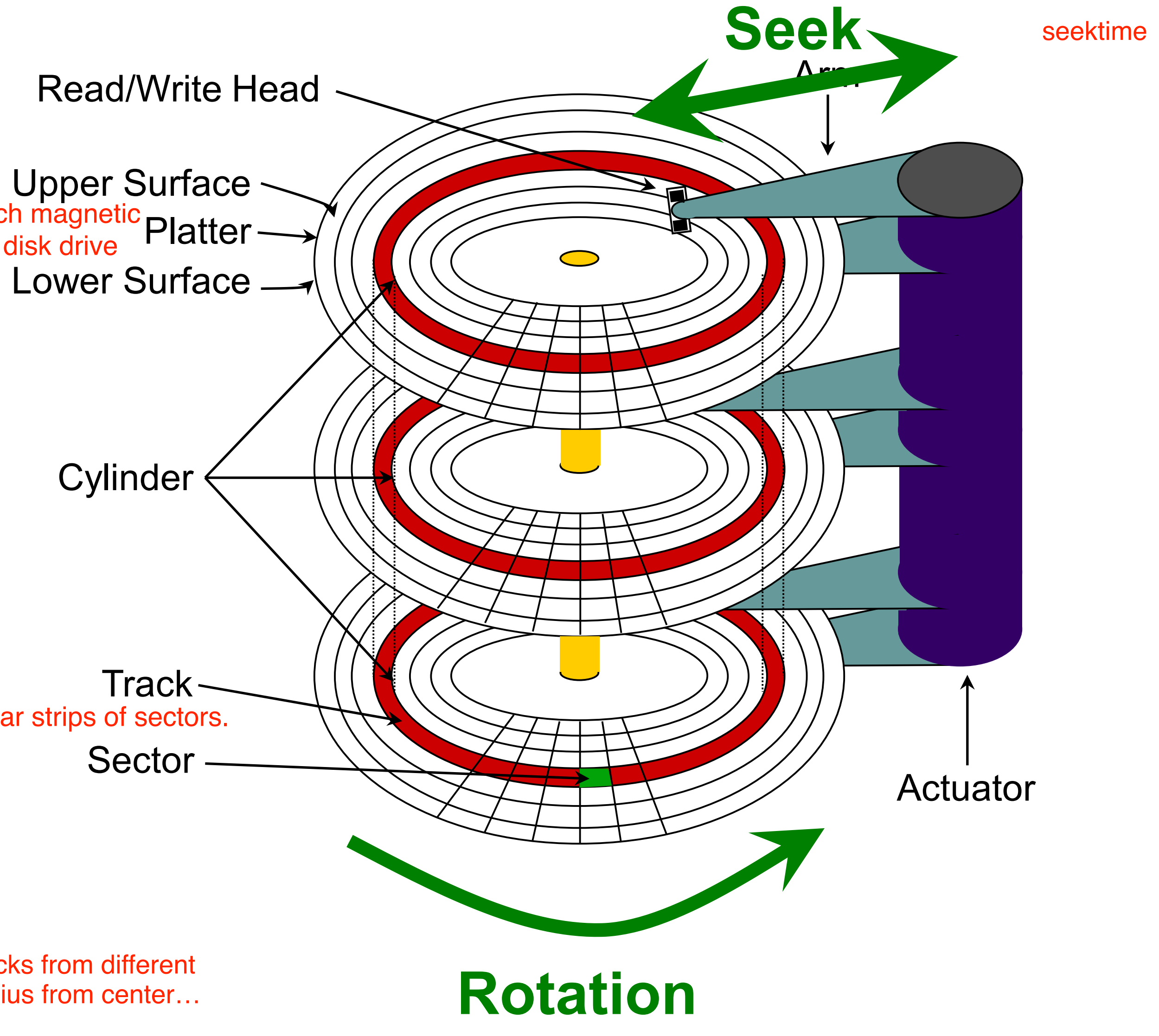
- A3 is posted
- Do not copy code from prevous semesters, github, etc

- How are file systems implemented?

File system implementation

- Files and directories live on **secondary storage**
  - Anything outside of “primary memory”
  - Anything that does not permit direct instruction execution or data fetch via machine load/store instructions
  - Is persistent: data survives loss of power
- We are focusing on the use of fixed hard magnetic disks for implementing secondary storage

Disk Components

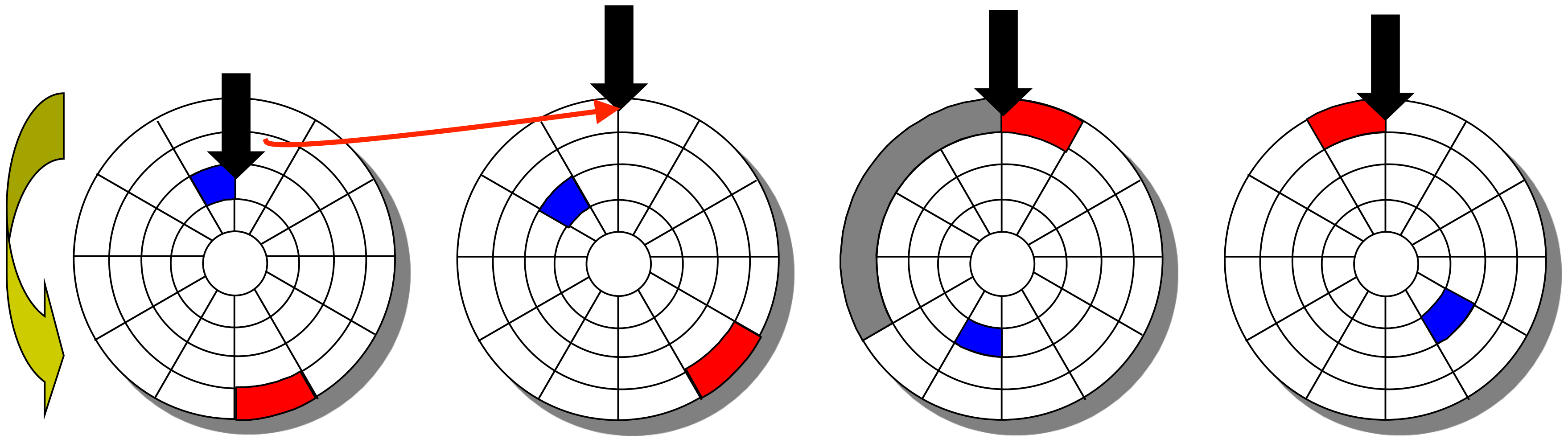


the circular disk on which magnetic data is stored in a hard disk drive

thin concentric circular strips of sectors.

cylinder: different tracks from different platter with same radius from center…

Disk service time components


After BLUE read

Seek for RED

Rotational latency

After RED read

- Sequential access is a lot faster than random access

Mixing workloads can be tricky

- Example scenario: Suppose there are two processes
  - Each run in isolation (by itself) gets **20 MB/s** disk throughput
  - If you run the two processes simultaneously each gets **2 MB/s**
  - **What happened?**

2 process in potentially different track/sector…

+ context switch usually faster than IO, so disk head switches its physical location quite frequently

Components of disk access time

- Disk request performance depends on three steps
  - Seek – moving the disk arm to the correct cylinder
    - Depends on how fast disk arm can move (increasing very slowly)
  - Rotation – waiting for the sector to rotate under the head
    - Depends on rotation rate of disk (increasing, but slowly)
  - Transfer – transferring data from surface into disk controller electronics, sending it back to the host
    - Depends on density (increasing quickly)

physical movements

- How long does this typically take?

Disks are slow

- Seek times:
  - 1-15ms, depending on distance
  - average 5-6ms
  - improving at 7-10% per year
- Rotation speeds:
  - ~7200 RPMs for cheap SATA disks
  - 10,000-15,000 RPMs for high-end SCSI disks
  - average latency of 3ms
  - improving at 7-10% per year
- This is slow!!!

- Sequential access much faster than random!

OS design principles

- Since disk I/O is slow:

can cache data in memory

  - Minimize number of disk accesses
    - E.g. by caching

but may be wasteful… since no one is asking for it

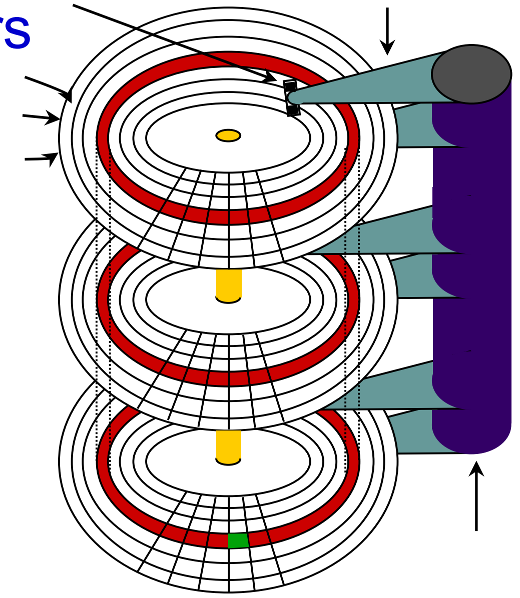
  - When using the disk try to minimize access cost
    - Particularly seeks and rotation
    - Use smart disk request scheduling
    - Arrange data for sequential access over random access

Disks are messy

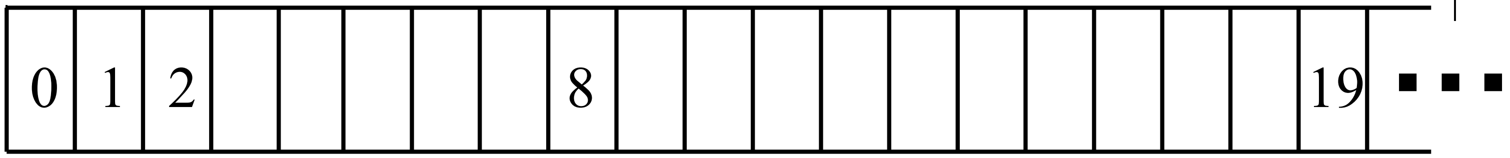
- Disks are messy physical devices:
  - Errors, bad blocks, missed seeks, etc.
- The job of the OS is to hide this mess from higher level software
  - Low-level device control (initiate a disk read, etc.)
  - Higher-level abstractions (files, databases, etc.)

OS ⇔ disk interaction

- Specifying disk requests requires a lot of info:
  - Cylinder #, surface #, track #, sector #, transfer size…
- Modern disks are even more complicated
  - Not all tracks have the same number of sectors, sectors are remapped, etc.
- Older disks required the OS to specify all of this
  - The OS needed to know all disk parameters
- Fortunately modern drives provide a more high-level interface:
  logical block addressing



Logical block address~~s~~ing



OS's view of storage device

Storage exposed as linear array of blocks

Common block size: 512 bytes

- The disk exports its data as logical array of blocks [0…N]
  - Disk maps logical blocks to cylinder/surface/track/sector
- Only need to specify the logical block # to read/write
- But now the disk parameters are hidden from the OS

disk controller does that translation

Disk Scheduling

- Because seeks are so expensive, the OS tries to schedule disk requests that are queued waiting for the disk
  - FCFS (do nothing)

    I/O low
    - Reasonable when load is low
    - Long waiting times for long request queues
  - SSTF (shortest seek time first)
    - Minimize arm movement (seek time), maximize request rate
    - Favors middle blocks
  - SCAN (elevator)

    track = floors. Service one track until all task finish in the current track, then go to next…
    - Service requests in one direction until done, then reverse
  - C-SCAN

    0 -> 1 -> …. -> n -> 0 -> 1 -> ….
    - Like SCAN, but only go in one direction (typewriter)
  - LOOK / C-LOOK
    - Like SCAN/C-SCAN but only go as far as last request in each direction (not full width of the disk)

Disk Scheduling (2)

- In general, unless there are request queues, disk scheduling does not have much impact
  - Important for servers, less so for PCs
- Modern disks often do the disk scheduling themselves
  - Disks know their layout better than OS, can optimize better
  - Ignores, undoes any scheduling done by OS

Back to files and directories …

- How does the OS implement the abstraction of files and directories on top of this logical array of disk blocks?

Disk Layout Strategies

- Files span multiple disk blocks
- How do you find all of the blocks for a file?

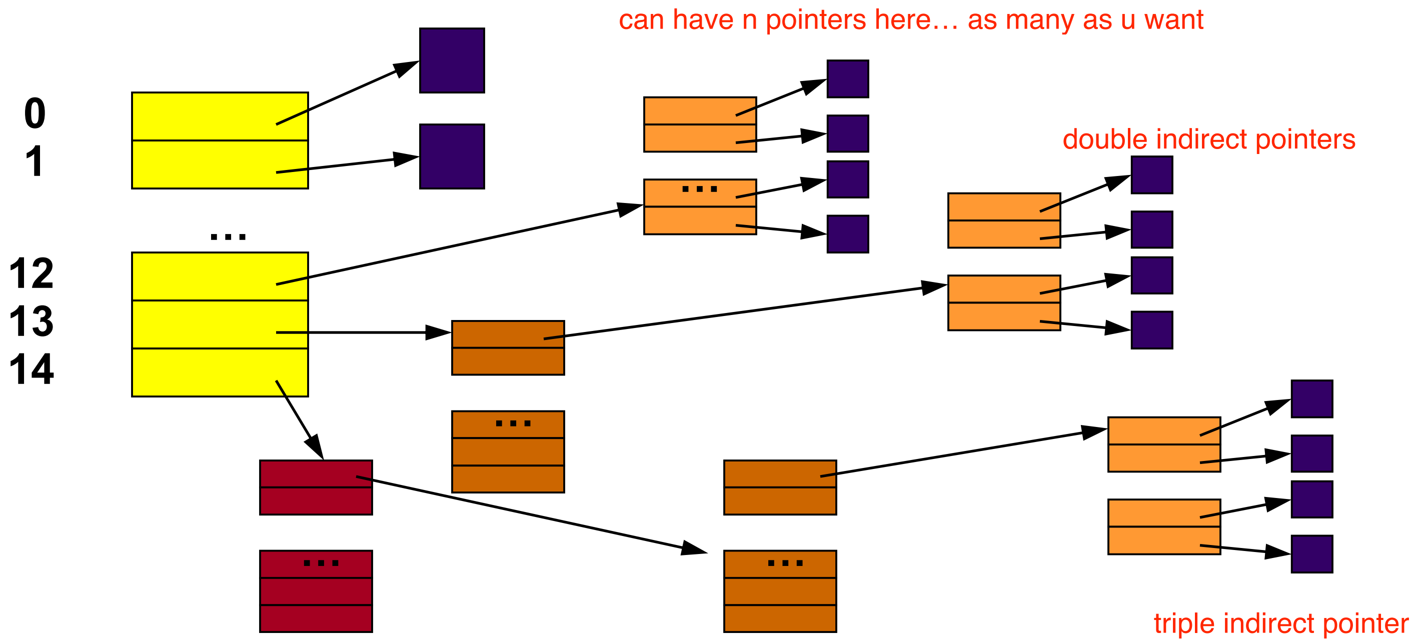
1. Contiguous allocation
   - Like memory
   - Fast, simplifies directory access
   - Inflexible, causes fragmentation, needs compaction
2. Linked, or chained, structure

   bad for random access

   - Each block points to the next, directory points to the first
   - Good for sequential access, bad for all others
3. Indexed structure (indirection, hierarchy)
   - An “index block” contains pointers to many other blocks
   - Handles random better, still good for sequential
   - May need multiple index blocks (linked together)

Indexed Allocation: Unix Inodes

- Unix inodes implement an indexed structure for files
- Each inode contains 15 block pointers
  - First 12 are direct block pointers (e.g., 4 KB data blocks)
  - Then single, double, and triple indirect



Unix Inodes and Path Search

- Example: We need to find the first data block for the file /one.txt

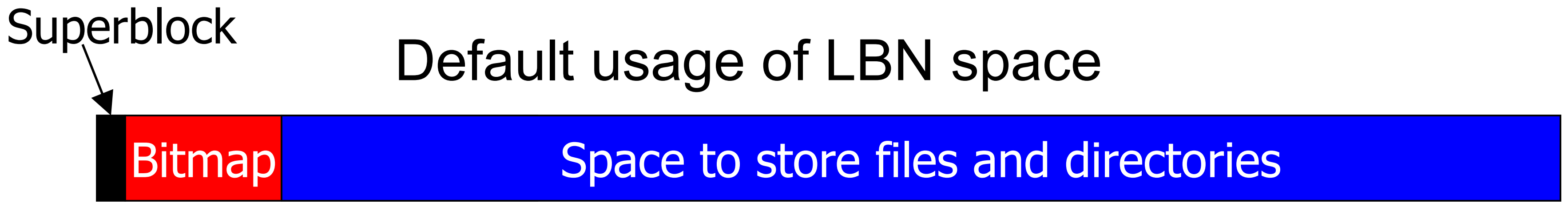
- Remember:
  - Unix Inodes are not directories
  - They describe where on the disk the blocks for a file are placed
    - Directories are files, so inodes also describe where the blocks for directories are placed on the disk

Unix Inodes and Path Search

- Directory entries map file names to inodes
  - To open “/one.txt”, use Master Block to find inode for “/” on disk and read inode into memory
  - inode allows us to find data block for directory “/”
  - Read “/”, look for entry for “one.txt”
  - This entry gives locates the inode for “one.txt”
  - Read the inode for “one” into memory
  - The inode says where first data block is on disk
  - Read that block into memory to access the data in the file

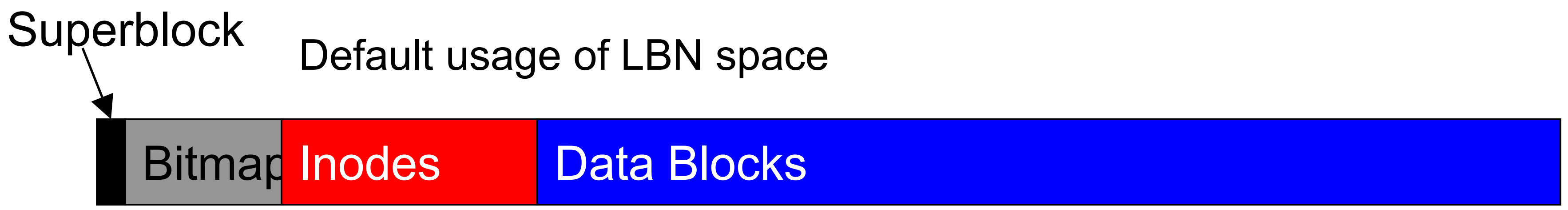
File System Implementation

- A "Master Block" determines location of root directory (aka *partition control block, superblock)*
- A free map determines which blocks are free, allocated
- Remaining disk blocks used to store files (and dirs)
  - There are many ways to do this


Superblock
Default usage of LBN space
Bitmap
Space to store files and directories

Original Unix File System

- Recall FS sees storage as linear array of blocks
  - Each block has a *logical block number (LBN)*



- Simple, straightforward implementation
  - Easy to implement and understand
- Problems:
  - Poor utilization of disk bandwidth (lots of seeking). Why???

jump between inodes and data blocks is frequent and is inefficient…

Data and Inode Placement

Aging -> consequence:
+ cannot allocate contiguous allocation of disk for the next files
+ so cant allocate every data block for one file in the same place
+ later read/write have to go to different disk locations…

Original Unix FS had two placement problems:

1. Data blocks allocated randomly in aging file systems
   - Blocks for the same file allocated sequentially when FS is new
   - As FS "ages" and fills, need to allocate into blocks freed up when other files are deleted

     allocation will be in random places
   - Problem: Deleted files essentially randomly placed
   - So, blocks for new files become scattered across the disk
2. Inodes allocated far from blocks
   - All inodes at beginning of disk, far from data
   - Traversing file name paths, manipulating files, directories requires going back and forth from inodes to data blocks

Both of these problems generate many long seeks

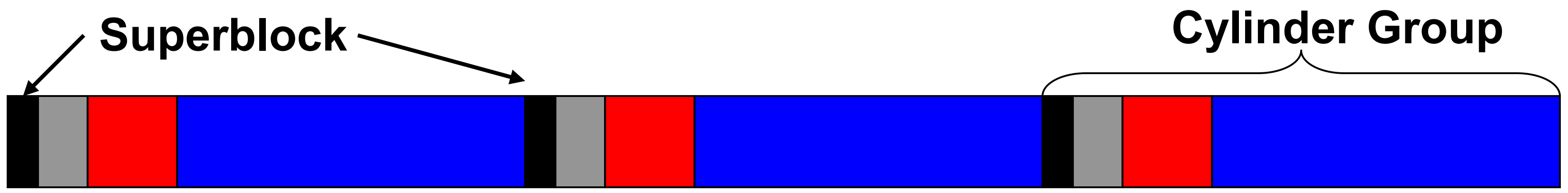
FFS

- BSD Unix folks did a redesign (early-mid 80s) that they called the **Fast File System (FFS)**
  - Improved disk utilization, decreased response time
- Now the FS from which all other Unix FS's have been compared
- Good example of being device-aware for performance

Cylinder Groups

- BSD FFS addressed placement problems using the notion of a cylinder group (aka *allocation groups* in lots of modern FS's)
  - Disk partitioned into groups of cylinders
  - Data blocks in same file allocated in same cylinder group
  - Files in same directory allocated in same cylinder group
  - Inodes for files allocated in same cylinder group as file data blocks

disk = cylinder groups
+ file: consisting data blocks in same cylinder group
+ directory: files within dir in same cylinder group
+ inodes + data blocks are in same cylinder group



Cylinder group organization

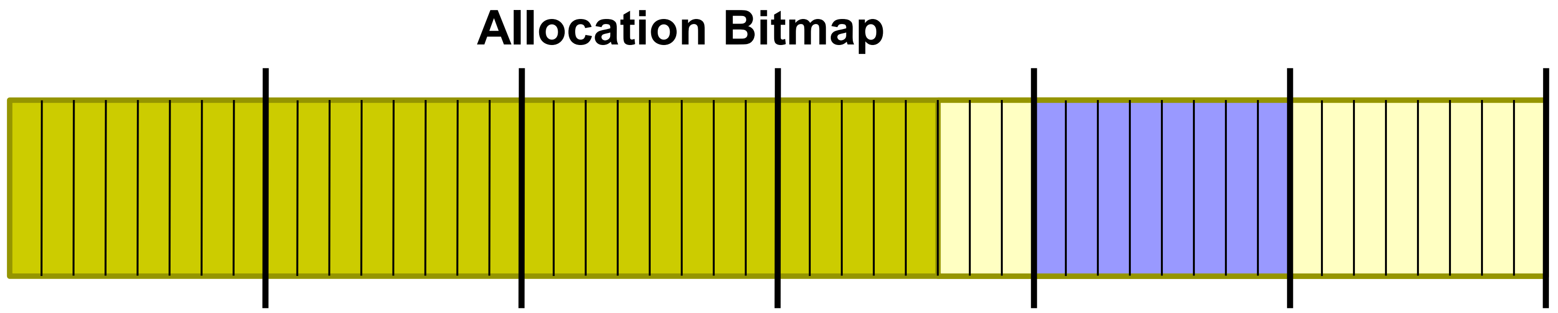
closeness between inodes and data
blocks is more efficient

Cylinder Groups (continued)

- Allocation in cylinder groups provides *closeness*
  - Reduces number of long seeks

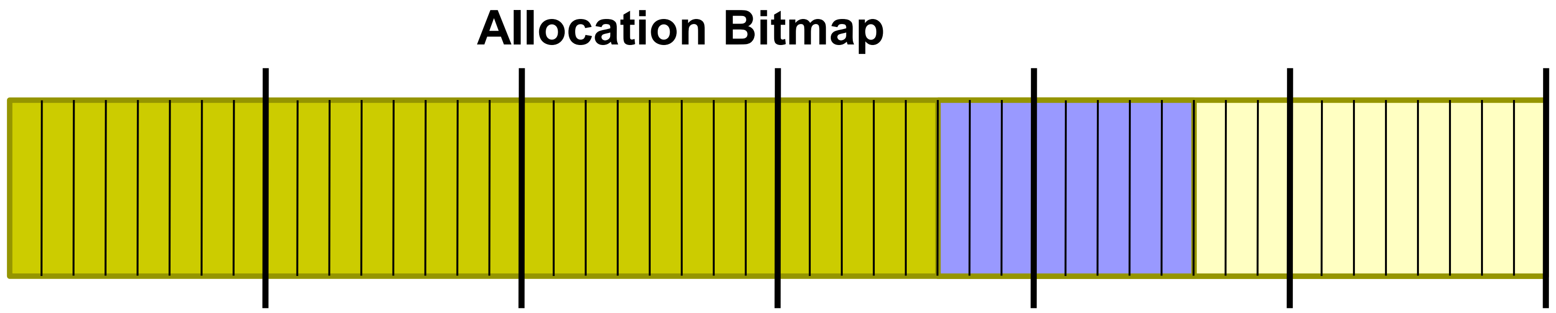
- Free space requirement
  - To be able to allocate according to cylinder groups, the disk must have free space scattered across cylinders
  - 10% of the disk is reserved just for this purpose
  - If preferred cylinder group is full, allocate from a “nearby” group

Space Allocation in Cylinder Groups



- If possible do allocation in groups of 8 blocks (bytes in bitmap)
  - Find first byte that's all zero
  - Backtrack the bits before to check for zero

Space Allocation in Cylinder Groups

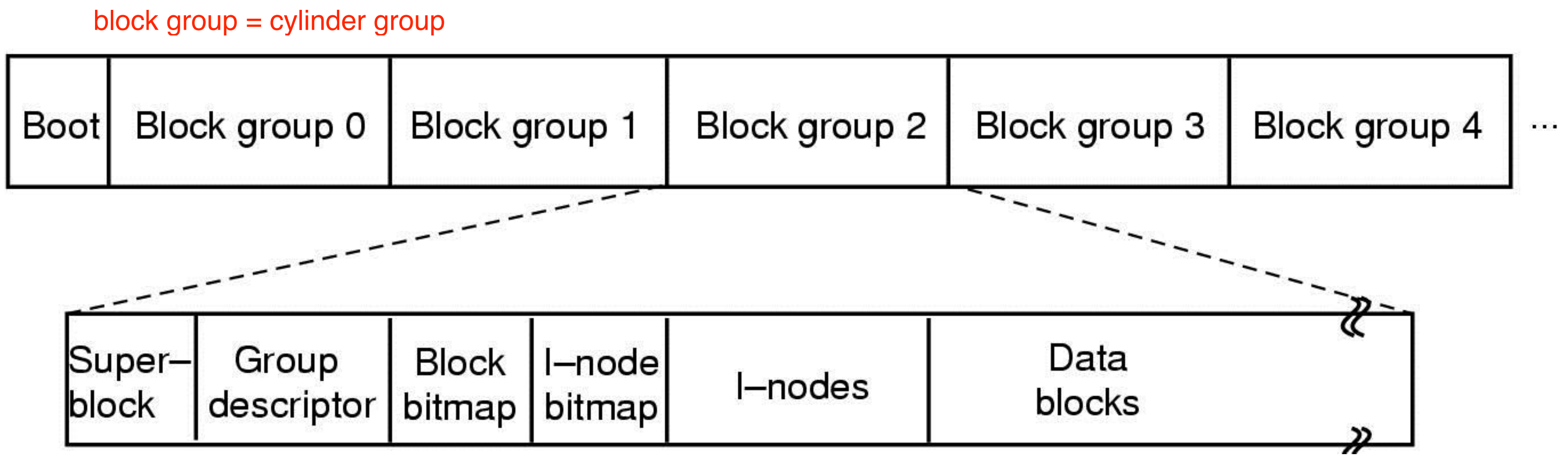


- If possible do allocation in groups of 8 blocks (bytes in bitmap)
  - Find first byte that’s all zero
  - Backtrack the bits before to check for zero

More FFS solutions

- Small blocks (1K) in orig. Unix FS caused 2 problems:
  - Low bandwidth utilization
  - Small max file size (function of block size)
- Fix using a larger block (4K)
  - New Problem: internal fragmentation

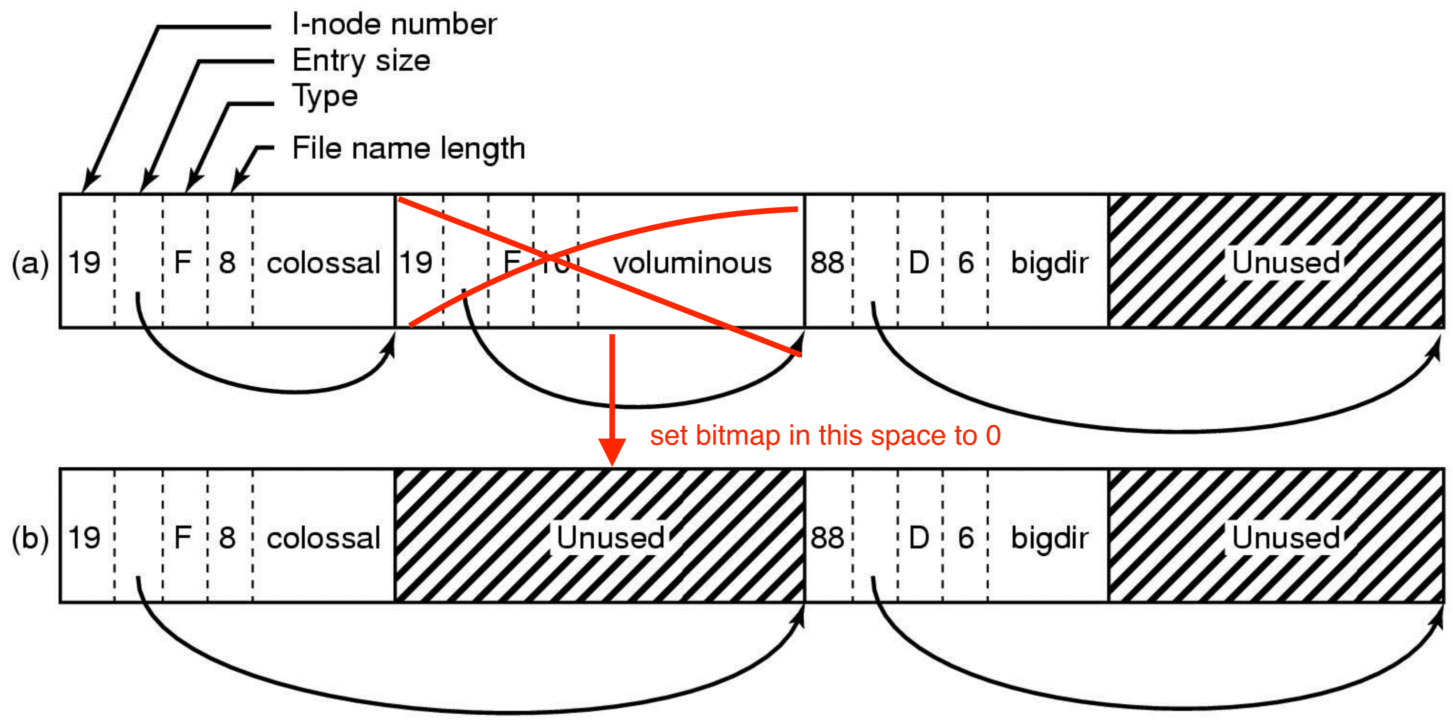
The Linux Second Extended File System (EXT2)



Disk layout of the Linux ext2 file system.

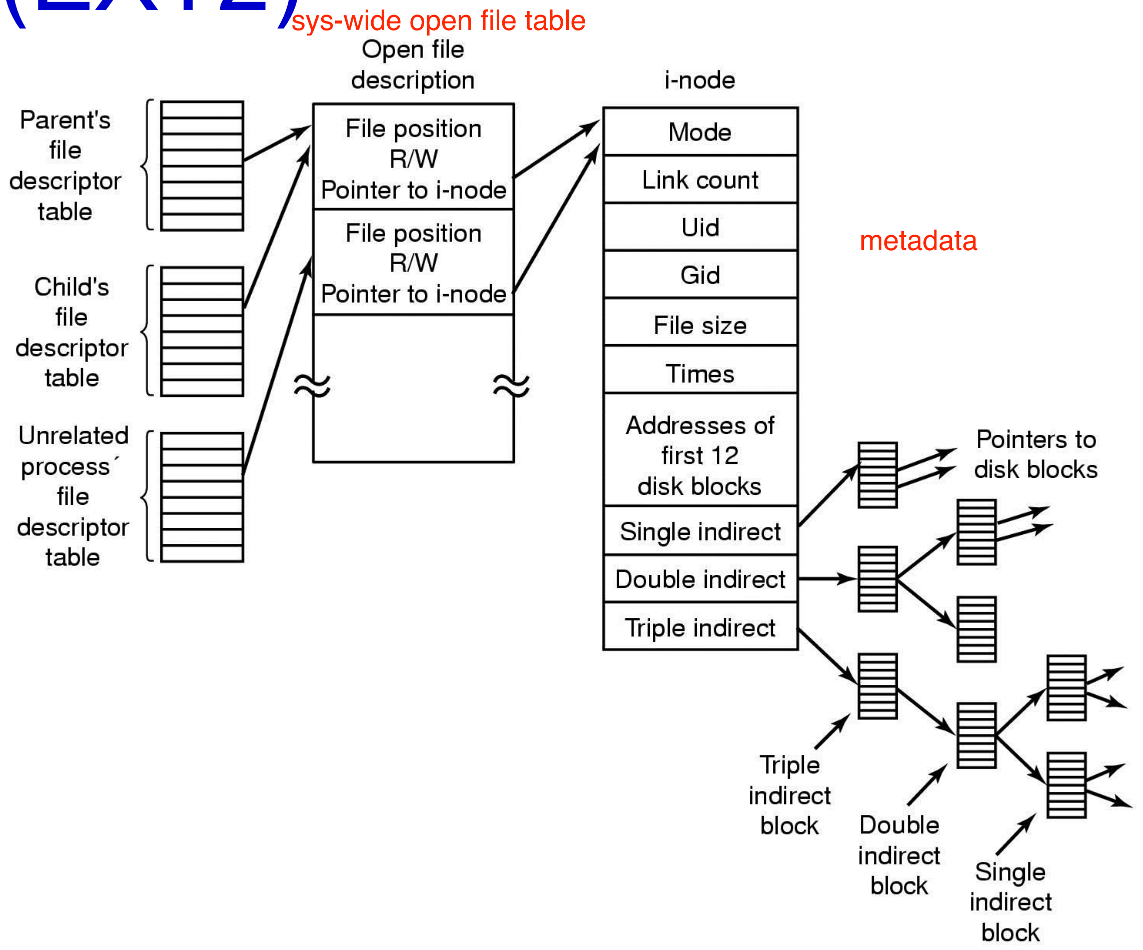
The Linux Second Extended File System (EXT2)

always have fragmentation



(a) A Linux directory with three files. (b) The same directory after the file voluminous has been removed.

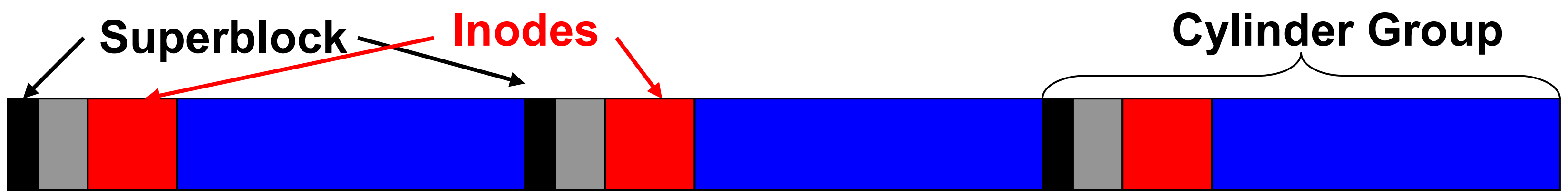
The Linux Second Extended File System (EXT2)



The relation between the file descriptor table, the open file description table, and the i-node table.

FFS: Consistency Issues

- Inodes: fixed size structure stored in cylinder groups



- Metadata updates are synchronous operations:
  - Write newly allocated inode to disk before its name is entered in a directory.

    allocate inode -> store its name in directory
    remove name -> deallocate inode
  - Remove a directory name before the inode is deallocated
  - Write a deallocated inode to disk before its blocks are placed into the cylinder group free list.
- Some updates create cyclical dependences

FFS Observation 1

- If the server crashes in between any of these synchronous operations, then the file system is in an inconsistent state.
- Solutions:
  - fsck – post-crash recovery process to scan file system structure and restore consistency
  - Log updates to enable roll-back or roll-forward.

FFS Observation 2

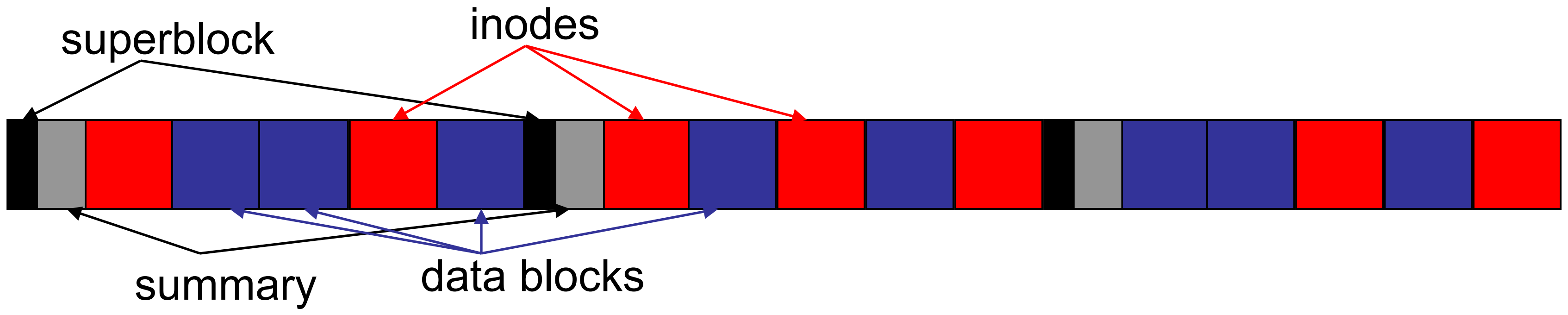
- The performance of FFS is optimized for disk block clustering, using properties of the disk to inform file system layout
- Observation: Memory is now large enough that most of the reads that go to the disk are the first read of a file. Subsequent reads are satisfied in memory by file buffer cache.
- I.e., there is no performance problem with reads. But write calls could be made faster.
- Writes are not well-clustered, they include inodes and data blocks.

Log Structured File System (LSF)

writing may be slower since have to update inode…

-

- Ousterhout 1989
- Write ***all*** file system data in a continuous log.
- Uses inodes and directories from FFS
- Needs an inode map to find the inodes
  - An inode number is no longer a simple index.
- Cleaner reclaims space from overwritten or deleted blocks.



LFS Reads

- If the writes are easy, what happens to the reads?
- To read a file from disk:
  1. Read the superblock to find the index file
  2. Read the index file (linear search on block of inodes)
  3. Use the disk address in inode to read the block of index file containing the inode-map
  4. Get the file's inode
  5. Use the inode as usual to find the file's data blocks
- But remember, we expect reads to hit in memory most of the time.

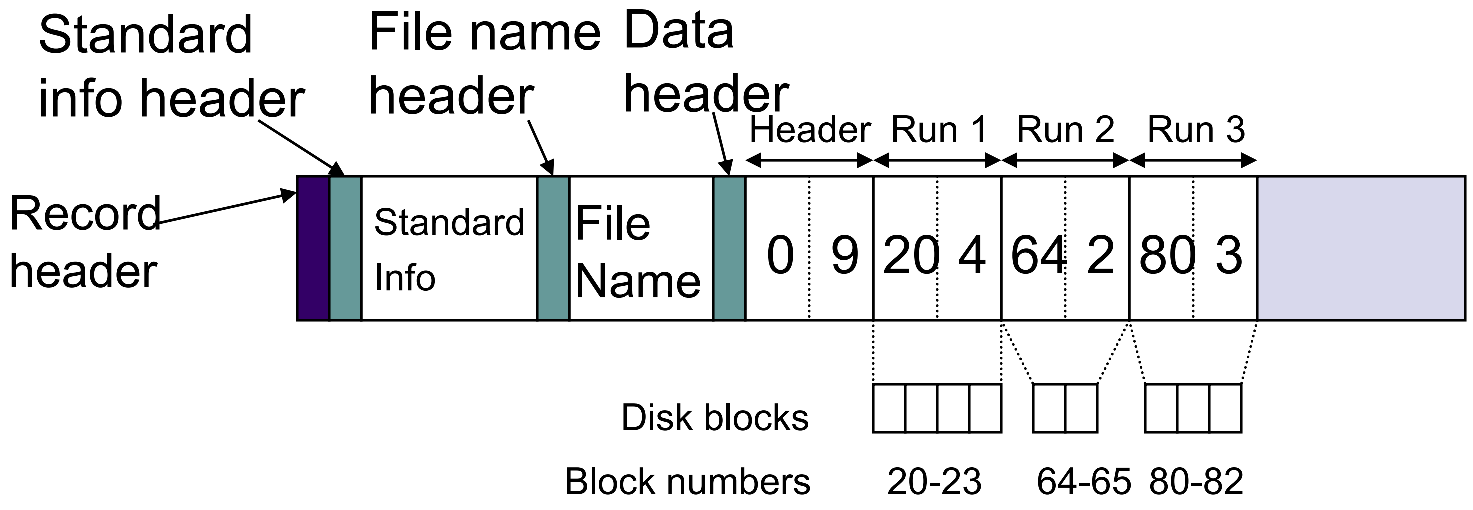
NTFS (Windows)

- The New Technology File System (NTFS) from Microsoft replaced the old FAT file system.
- The designers had the following goals:
  1. Eliminate fixed-size short names
  2. Implement a more thorough permissions scheme
  3. Provide good performance
  4. Support large files
  5. Provide extra functionality:
     - Compression
     - Encryption
     - Types
- In other words, they wanted a file system flexible enough to support future needs.

NTFS

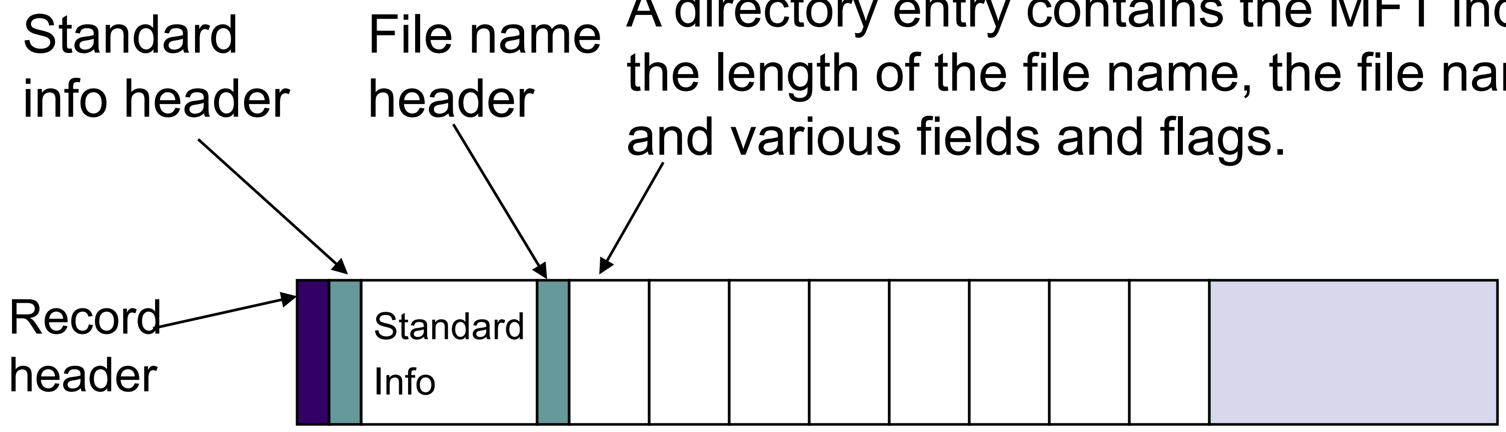
- Each volume (partition) is a linear sequence of blocks (usually 4 Kb block size).
- Each volume has a Master File Table (MFT).
  - Sequence of 1 KB records.
  - One or more record per file or directory
    - Similar to inodes, but more flexible
  - Each MFT record is a sequence of variable length (attribute, value) pairs.
  - Long attributes can be stored externally, and a pointer kept in the MFT record.
- NTFS tries to allocate files in runs of consecutive blocks.

MFT Record



- An MFT record for a 3-run 9-block file.
- Each “data” attribute indicates the starting block and the number of blocks in a “run” (or extent)
- If all the records don’t fit into one MFT record, extension records can be used to hold more.

MFT Record for a Small Directory

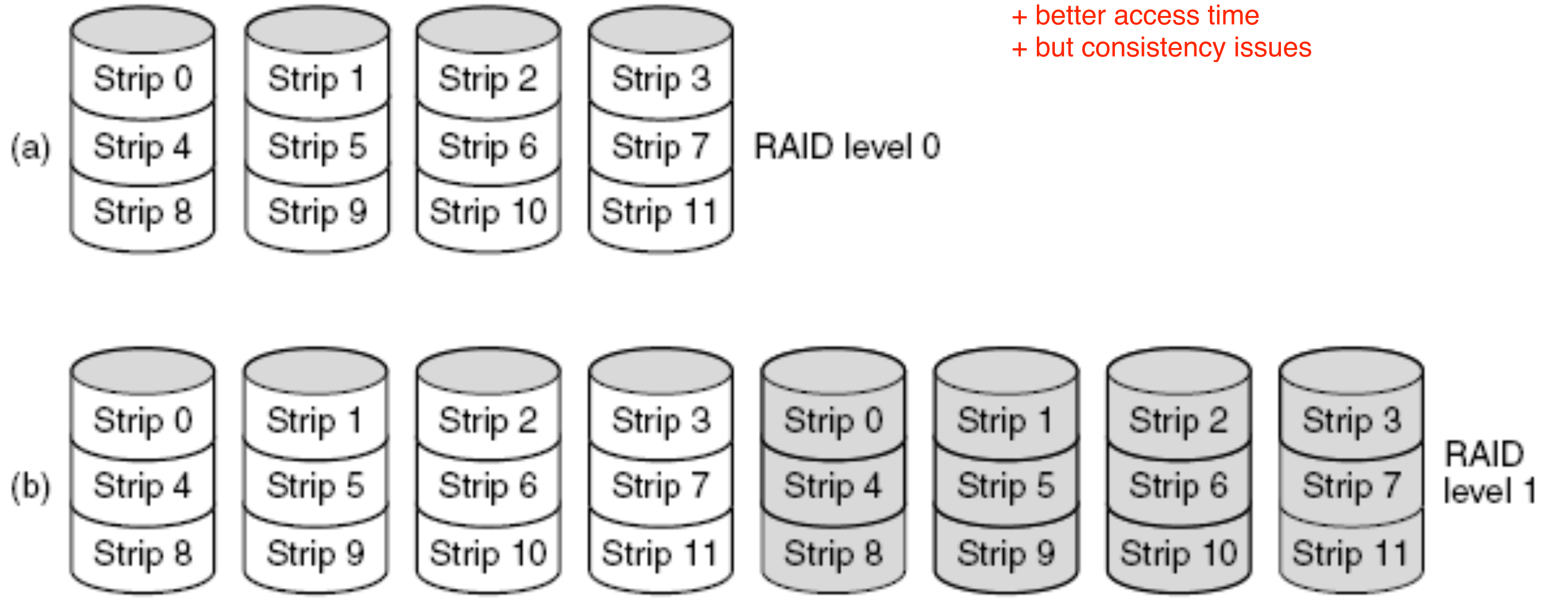

Standard info header
File name header
A directory entry contains the MFT index for the file, the length of the file name, the file name itself, and various fields and flags.
Record header
Standard Info

- Directory entries are stored as a simple list
- Large directories use B+ trees instead.

Better I/O performance through parallelism

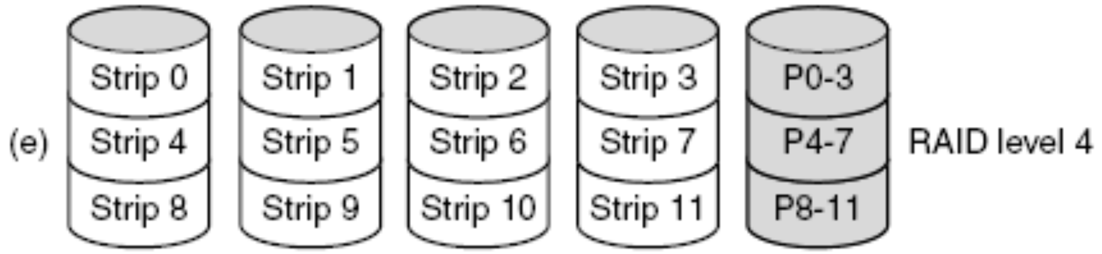
- Idea: Spread the work across several disks
- RAID: Redundant Arrays of Inexpensive Disks
- While we're at it, why not also increase reliability ….

duplicating data
+ better access time
+ but consistency issues



Better I/O performance through parallelism

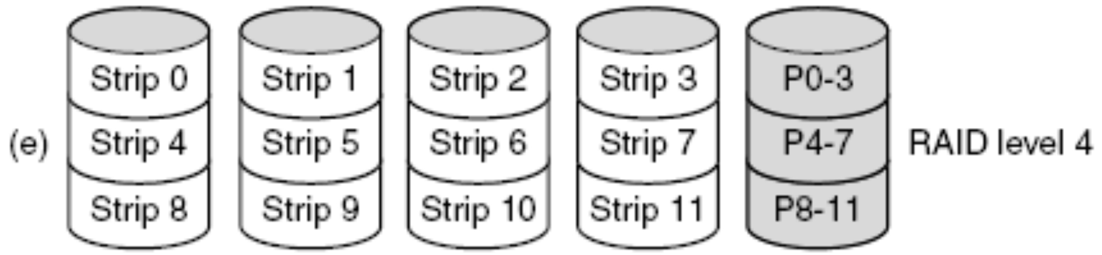
- Idea: Spread the work across several disks
- RAID: Redundant Arrays of Inexpensive Disks
- While we're at it, why not also increase reliability ….



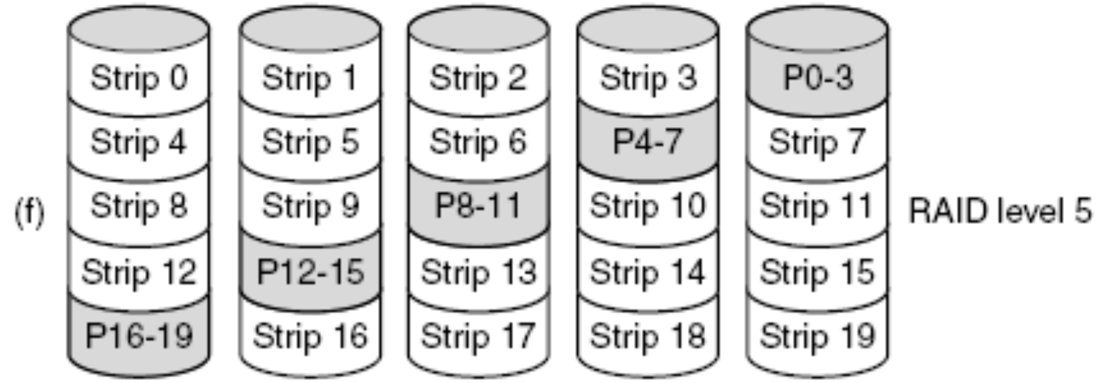
parity bit:
+ checksum..

Better I/O performance through parallelism

- Idea: Spread the work across several disks
- RAID: Redundant Arrays of Inexpensive Disks
- While we're at it, why not also increase reliability ….



(e)
Strip 0
Strip 1
Strip 2
Strip 3
P0-3
Strip 4
Strip 5
Strip 6
Strip 7
P4-7
Strip 8
Strip 9
Strip 10
Strip 11
P8-11
RAID level 4



(f)
Strip 0
Strip 1
Strip 2
Strip 3
P0-3
Strip 4
Strip 5
Strip 6
P4-7
Strip 7
Strip 8
Strip 9
P8-11
Strip 10
Strip 11
Strip 12
P12-15
Strip 13
Strip 14
Strip 15
P16-19
Strip 16
Strip 17
Strip 18
Strip 19
RAID level 5