Filesystems: efficiency con't / reliability

Changelog

Changes made in this version not seen in first lecture:

8 November: correct several formatting errors on RAID slides
8 November: extra last time slide re: inode, block groups

last time

hard disks: adjacent accesses are fast

- sector numbers: closeby numbers = closeby sectors
- disk or OS: want to schedule accesses by location on disk

hard disks: error detection/correction

- redundancy to catch/correct some errors
- bad sector = tell OS usually (not give it bad data)
- relocate sectors to deal with broken parts of disk

SSDs: erasure blocks and wear leveling

- can only overwrite in big blocks
- can only overwrite so many times
- solution: controller moves blocks around “wear leveling”

last time (2)

inodes:
- store file information in one place
- creation/modification times, blocks in file, etc.
- directory entries point to inode

inodes and direct/indirect blocks
- direct block pointers: point to data
- indirect block pointers: point to pointers to data
- Nth pointer to data = pointer to block N

sparse files: represent strings of 0s via NULL block pointers

block groups:
- each group has set of inodes + data blocks
- typically (but not always) directory and its files contained without block group

correction re: symbolic links

I implied symbolic links: kept in directory entry

not true: usually they have their own inode

usually store string (name of referenced file) in inode

xv6 filesystem performance issues

inode, block map stored far away from file data

- long seek times for reading files

unintelligent choice of file/directory data blocks

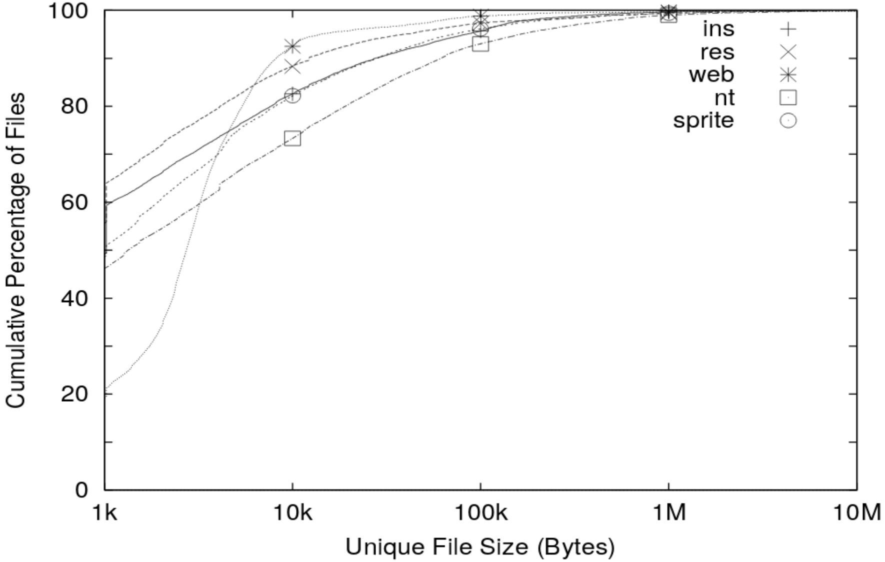
- xv6 finds *first free block/inode*
- result: files/directory entries scattered about

blocks are pretty small — needs lots of space for metadata

- could change size? but waste space for small files
- large files have giant lists of blocks

linear searches of directory entries to resolve paths

empirical file sizes



Roselli et al, "A Comparison of Filesystem Workloads", in FAST 2000

typical file sizes

most files are small

- sometimes 50+% less than 1kbyte
- often 5-20% less than 10kbyte

doens't mean large files are unimportant

- still take up most of the space
- biggest performance problems

fragments

FFS: a file's last block can be a *fragment* — only part of a block

each block split into approx. 4 fragments

- each fragment has its own index

extra field in inode indicates that last block is fragment

allows one block to store data for several small files

non-FFS changes

now some techniques beyond FFS

some of these supported by current filesystems, like

- Microsoft's NTFS
- Linux's ext4 (successor to ext2)

xv6 filesystem performance issues

inode, block map stored far away from file data
- long seek times for reading files

unintelligent choice of file/directory data blocks
- xv6 finds *first free block/inode*
- result: files/directory entries scattered about

blocks are pretty small — needs lots of space for metadata
- could change size? but waste space for small files
- large files have giant lists of blocks

linear searches of directory entries to resolve paths

extents

large file? lists of many thousands of blocks is awkward

solution: store extents: (start disk block, size)

- replaces or supplements block list

Linux's ext4 and NTFS both use this

allocating extents

challenge: finding contiguous set of free blocks

FFS's strategy "first in block group" doesn't work well

- first several blocks likely to be 'holes' from deleted files

NTFS: scan block map for "best fit"

- big enough chunk of free blocks
- smallest among all the candidates:

efficient seeking with extents

suppose a file has long list of extents

how to seek to byte X?

efficient seeking with extents

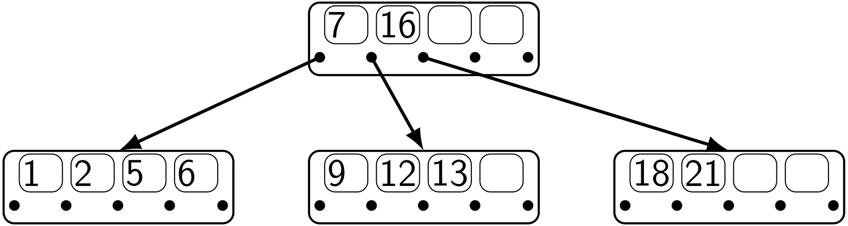
suppose a file has long list of extents

how to seek to byte X?

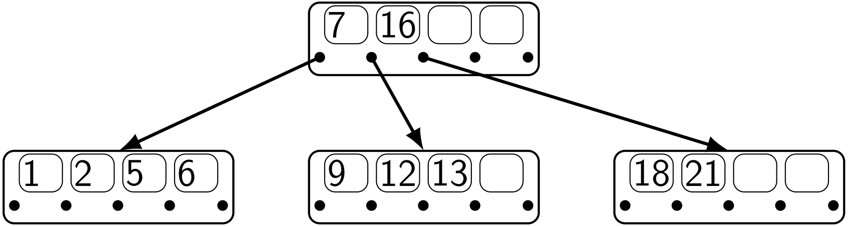
solution: store a tree

- ext4: each node stores minimum file index it covers
- ext4: each node has pointer (disk block) to its children

non-binary search trees



non-binary search trees



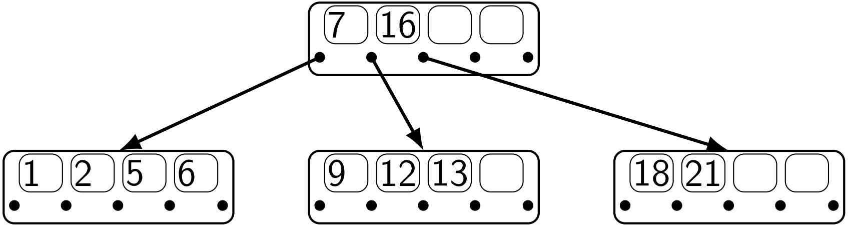
each node can be one block on disk

choose number of entries in node based on block size

avoid large or random accesses to disk and linear searches

can do binary search within a node

non-binary search trees



each node can be one block on disk

choose number of entries in node based on block size

avoid large or random accesses to disk and linear searches

can do binary search within a node

algorithms for adding to tree while keeping it balanced

similar idea to AVL trees

using trees on disk

linear search to find extent at offset X

store index by offset of extent within file

linear search to find file in directory?

index by filename

both problems — solved with non-binary tree on disk

filesystem reliability

a crash happens — what's the state of my filesystem?

hard disk atomicity

interrupt a hard drive write?

write whole disk sector or corrupt it

hard drive stores checksum for each sector

write interrupted? — checksum mismatch

- hard drive returns read error

reliability issues

is the data there?

- can we find the file, etc.?

is the filesystem in a consistent state?

- do we know what blocks are free?

multiple copies

FAT: multiple copies of file allocation table and header

in inode-based filesystems: often multiple superblocks

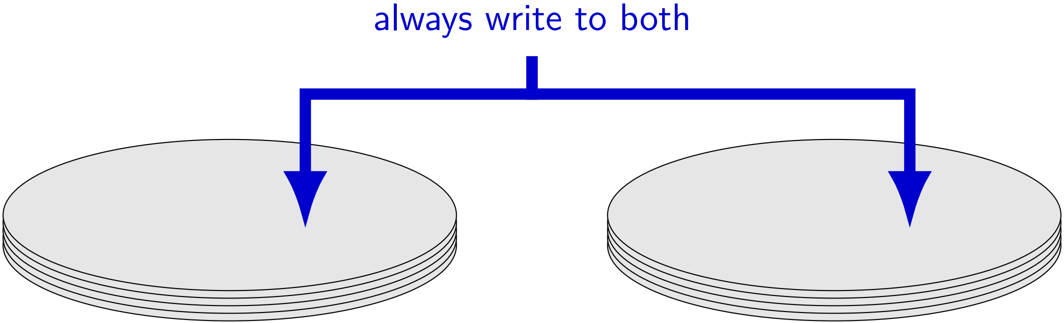
if part of disk's data is lost, have an extra copy

- always update both copies
- hope: disk failure to small group of sectors

hope: enough to recover most files on disk failure

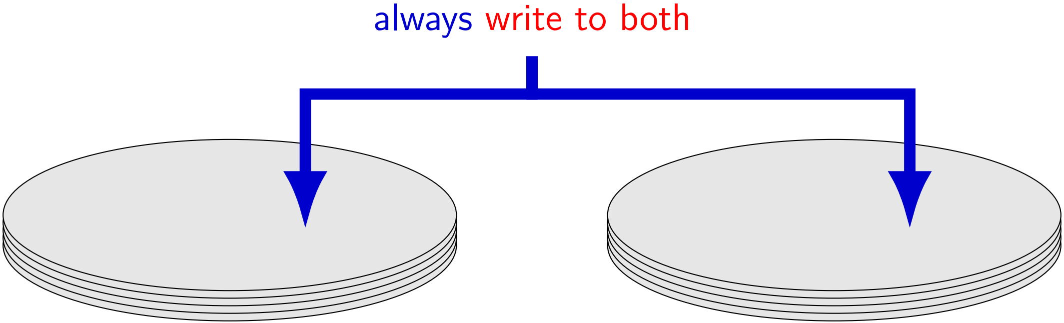
mirroring whole disks

alternate strategy: write everything to two disks



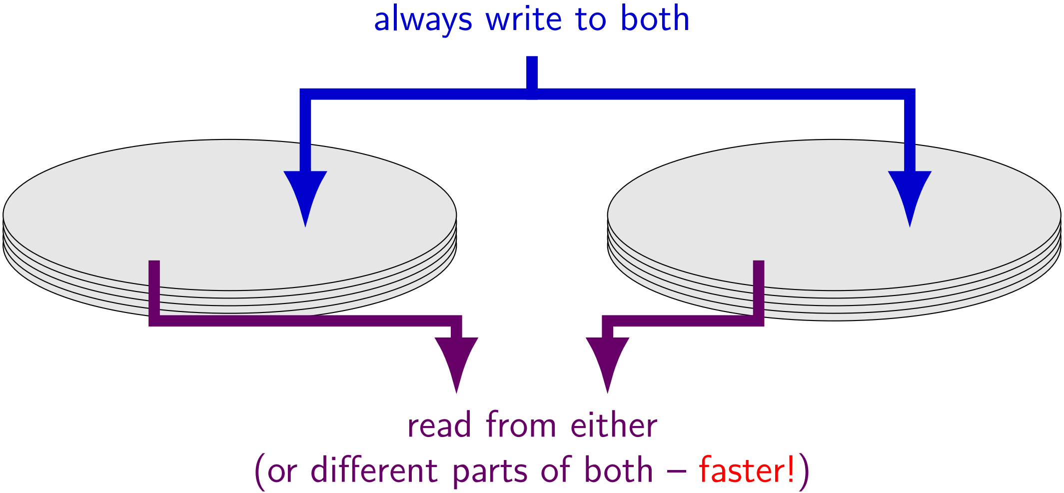
mirroring whole disks

alternate strategy: write everything to two disks



mirroring whole disks

alternate strategy: write everything to two disks



RAID 4 parity

$\oplus$ — bitwise xor

disk 1	disk 2	disk 3
A_1: sector 0	A_2: sector 1	A_p: $A_1 \oplus A_2$
B_1: sector 2	B_2: sector 3	B_p: $B_1 \oplus B_2$

... ...

RAID 4 parity

$\oplus$ — bitwise xor

disk 1	**disk 2**	**disk 3**
A_1: sector 0	A_2: sector 1	A_p: $A_1 \oplus A_2$
B_1: sector 2	B_2: sector 3	B_p: $B_1 \oplus B_2$

... ...

$$A_p = A_1 \oplus A_2$$
$$A_1 = A_p \oplus A_2$$
$$A_2 = A_1 \oplus A_p$$

can compute contents of any disk!

RAID 4 parity

$\oplus$ — bitwise xor

disk 1	disk 2	disk 3
A_1: sector 0	A_2: sector 1	A_p: $A_1 \oplus A_2$
B_1: sector 2	B_2: sector 3	B_p: $B_1 \oplus B_2$
...	...	...

exercise: how to replace sector 3 (B_2)with new value?
how many writes? how many reads?

RAID 4 parity (more disks)

disk 1	**disk 2**	**disk 3**	**disk 4**
A_1: sector 0	A_2: sector 1	A_3 sector 2	A_p: $A_1 \oplus A_2 \oplus A_3$
B_1: sector 3	B_2: sector 4	B_3: sector 5	B_p: $B_1 \oplus B_2 \oplus B_3$
...	...	...	

RAID 4 parity (more disks)

disk 1	**disk 2**	**disk 3**	**disk 4**
A_1: sector 0	A_2: sector 1	A_3 sector 2	A_p: $A_1 \oplus A_2 \oplus A_3$
B_1: sector 3	B_2: sector 4	B_3: sector 5	B_p: $B_1 \oplus B_2 \oplus B_3$
...	...	...	

$$A_p = A_1 \oplus A_2 \oplus A_3$$
$$A_1 = A_p \oplus A_2 \oplus A_3$$
$$A_2 = A_1 \oplus A_p \oplus A_3$$
$$A_3 = A_1 \oplus A_2 \oplus A_p$$

can still compute contents of any disk!

RAID 4 parity (more disks)

disk 1	**disk 2**	**disk 3**	**disk 4**
A_1: sector 0	A_2: sector 1	A_3 sector 2	A_p: $A_1 \oplus A_2 \oplus A_3$
B_1: sector 3	B_2: sector 4	B_3: sector 5	B_p: $B_1 \oplus B_2 \oplus B_3$
...	...	...	

exercise: how to replace sector 3 (B_1) with new value now?
how many writes? how many reads?

RAID 5 parity

disk 1	disk 2	disk 3	disk 4
A_1: sector 0	A_2: sector 1	A_3: sector 2	A_p: $A_1 \oplus A_2 \oplus A_3$
B_1: sector 3	B_2: sector 4	B_p: $B_1 \oplus B_2 \oplus B_3$	B_3:sector 5
C_1: sector 6	C_p: $C_1 \oplus C_2 \oplus C_3$	C_2: sector 7	C_3: sector 8
...	...	...	

RAID 5 parity

disk 1	disk 2	disk 3	disk 4
A_1: sector 0	A_2: sector 1	A_3: sector 2	A_p: $A_1 \oplus A_2 \oplus A_3$
B_1: sector 3	B_2: sector 4	B_p: $B_1 \oplus B_2 \oplus B_3$	B_3:sector 5
C_1: sector 6	C_p: $C_1 \oplus C_2 \oplus C_3$	C_2: sector 7	C_3: sector 8
…	…	…	

spread out parity updates across disks
so each disk has about same amount of work

more general schemes

RAID 6: tolerate loss of any two disks

can generalize to 3 or more failures

justification: takes days/weeks to replace data on missing disk

...giving time for more disks to fail

probably more in CS 4434?

but none of this addresses consistency

RAID-like redundancy

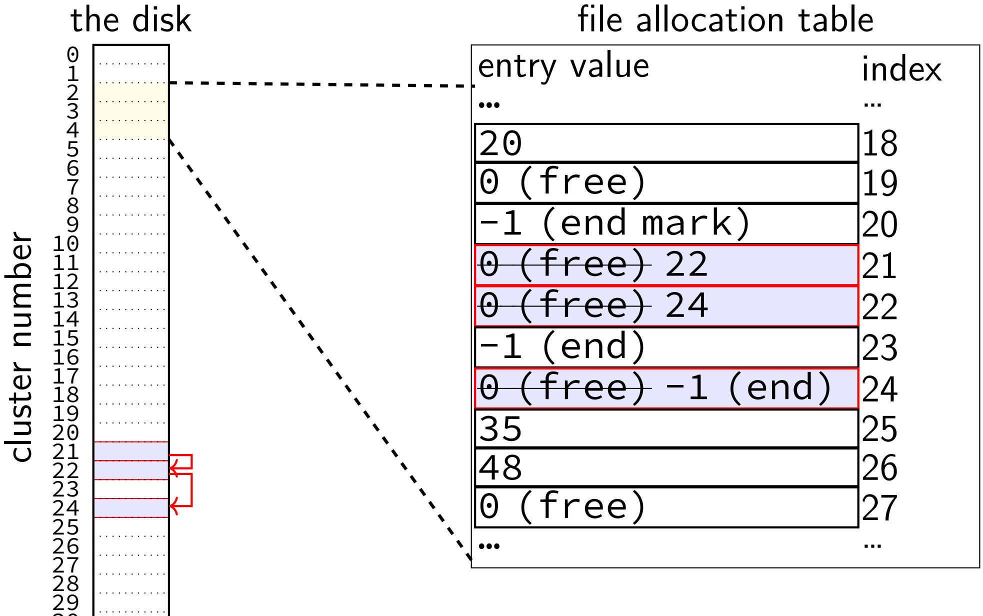
usually appears to filesystem as 'more reliable disk'

- hardware or software layers to implement extra copies/parity

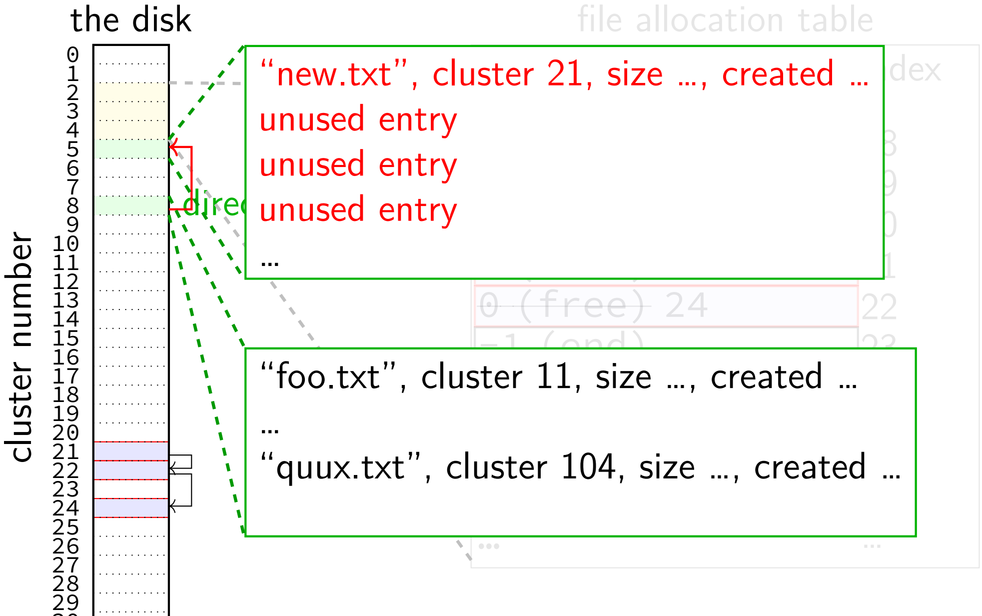
some filesystems (e.g. ZFS) implement this themselves

- more flexibility — e.g. change redundancy file-by-file
- ZFS combines with its own checksums — don't trust disks!

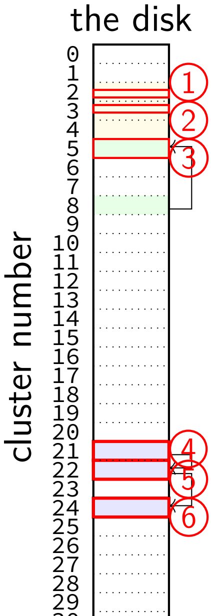
recall: FAT: file creation (1)



recall: FAT: file creation (2)



exercise: FAT file creation



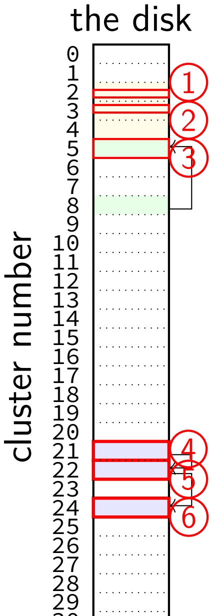
FAT entries for directory + file

new directory cluster

6 clusters to write
on loss of power: only some completed

new file clusters

exercise: FAT file creation



FAT entries for directory + file

new directory cluster

6 clusters to write
on loss of power: only some completed

exercise: what happens if only 1, 2 complete?
everything but 3?

new file clusters

exercise: FAT ordering

(creating a file that needs new cluster of direntries)

1. FAT entry for extra directory cluster
2. FAT entry for new file clusters
3. file clusters
4. file's directory entry (in new directory cluster)

what ordering is best if a crash happens in the middle?

A. 1, 2, 3, 4
B. 4, 3, 1, 2
C. 1, 3, 4, 2
D. 3, 4, 2, 1
E. 3, 1, 4, 2

exercise: xv6 FS ordering

(creating a file that neeeds new block of direntries)

1. free block map for new directory block
2. free block map for new file block
3. directory inode
4. new file inode
5. new directory entry for file (in new directory block)
6. file data blocks

what ordering is best if a crash happens in the middle?

A. 1, 2, 3, 4, 5, 6
B. 6, 5, 4, 3, 2, 1
C. 1, 2, 6, 5, 4, 3
D. 2, 6, 4, 1, 5, 3
E. 3, 4, 1, 2, 5, 6

r

inode-based FS: careful ordering

mark blocks as allocated before referring to them from directories

write data blocks before writing pointers to them from inodes

write inodes before directory entries pointing to it

remove inode from directory before marking inode as free

    or decreasing link count, if there's another hard link

idea: better to waste space than point to bad data

inode-based FS: creating a file

normal operation

allocate data block
write data block
update free block map
update file inode
update directory entry filename+inode number
update direcotry inode modification time

inode-based FS: creating a file

normal operation

allocate data block

write data block

update free block map

update file inode

update directory entry

filename+inode number

update direcotry inode

modification time

general rule:

better to waste space

than point to bad data

mark blocks/inodes used before writing

inode-based FS: creating a file

normal operation

- allocate data block
- write data block
- update free block map
- update file inode
- update directory entry
  - filename+inode number
- update direcotry inode
  - modification time

recovery (`fsck`)

- read all directory entries
- scan all inodes
- free unused inodes
  - unused = not in directory
- free unused data blocks
  - unused = not in inode lists
- scan directories for missing update/access times

inode-based FS: exercise: unlink

what order to remove a hard link (= directory entry) for file?

1. overwrite directroy entry for file
2. decrement link count in inode (but link count still > 1 so don't remove)

assume not the last hard link

inode-based FS: exercise: unlink

what order to remove a hard link (= directory entry) for file?

1. overwrite directroy entry for file
2. decrement link count in inode (but link count still > 1 so don't remove)

assume not the last hard link

what does recovery operation do?

inode-based FS: exercise: unlink last

what order to remove a hard link (= directory entry) for file?

1. overwrite last directroy entry for file
2. mark inode as free (link count = 0 now)
3. mark inode's data blocks as free

assume is the last hard link

inode-based FS: exercise: unlink last

what order to remove a hard link (= directory entry) for file?

1. overwrite last directroy entry for file
2. mark inode as free (link count = 0 now)
3. mark inode's data blocks as free

assume is the last hard link

what does recovery operation do?

fsck

Unix typically has an `fsck` utility

checks for *filesystem consistency*

- is a data block marked as used that no inodes uses?
- is a data block referred to by two different inodes?
- is a inode marked as used that no directory references?
- is the link count for each inode = number of directories referencing it?
- ...

assuming careful ordering, can fix errors after a crash without loss, probably

fsck costs

my desktop's filesystem: 2.4M used inodes; 379.9M of 472.4M used blocks

recall: check for data block marked as used that no inode uses:

- read blocks containing all of the 2.4M used inodes
- add each block pointer to a list of used blocks
- if they have indirect block pointers, read those blocks, too
- get list of all used blocks (via direct or indirect pointers)
- compare list of used blocks to actual free block bitmap

pretty expensive and slow

running fsck automatically

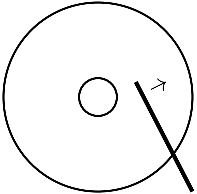
common to have "clean" bit in superblock

last thing written (to set) on shutdown

first thing written (to clear) on startup

on boot: if clean bit clear, run fsck first

ordering and disk performance



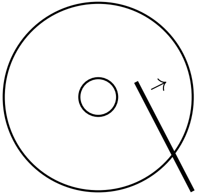
recall: seek times

would like to order writes based on locations on disk

- write many things in one pass of disk head
- write many things in cylinder in one rotation

ordering and disk performance

recall: seek times



would like to order writes based on locations on disk

- write many things in one pass of disk head
- write many things in cylinder in one rotation

ordering constraints make this hard:

free block map for file (start), then file blocks (middle), then...

file inode (start), then directory (middle), ...

modern windows: NTFS

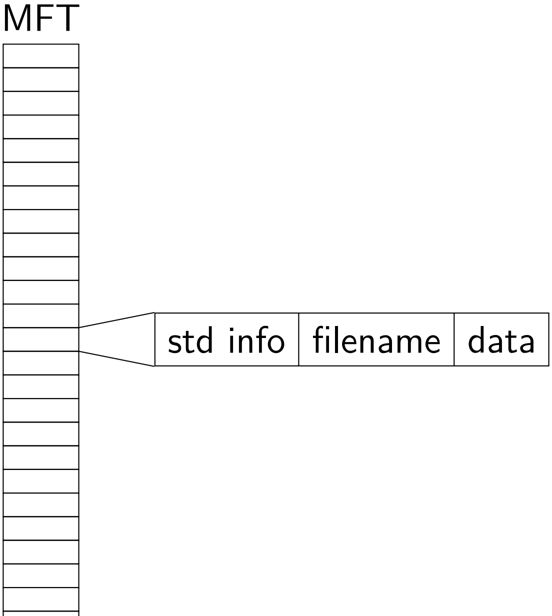
typical modern windows FS is NTFS or variants

uses extents, as mentioned

also has some neat tricks in high-level organization

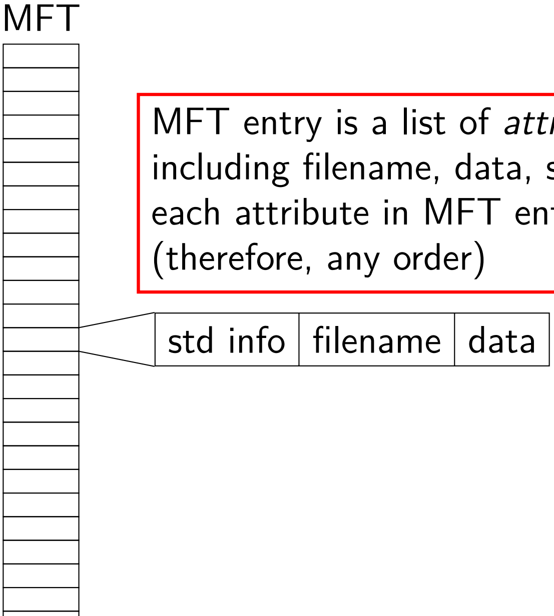
- it's not inodes

NTFS: Master File Table



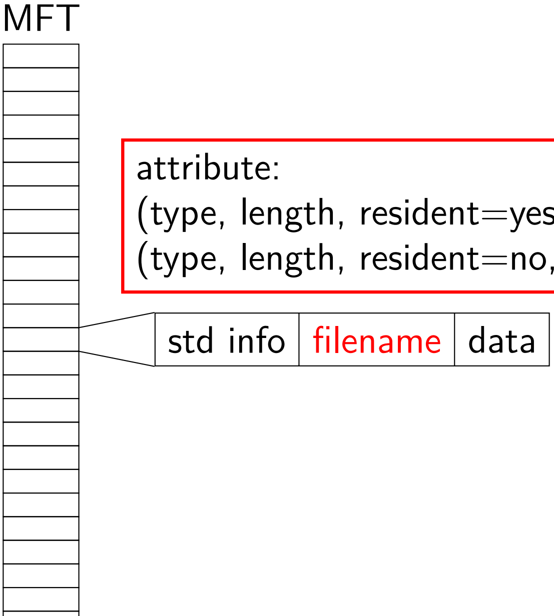
NTFS: Master File Table

MFT entry is a list of *attributes*
including filename, data, standard data
each attribute in MFT entry has type, length
(therefore, any order)

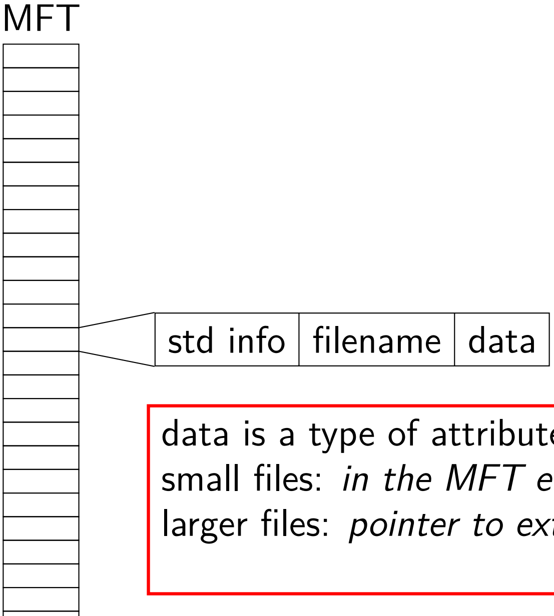


NTFS: Master File Table

attribute:
(type, length, resident=yes, data for attribute)
(type, length, resident=no, pointer to data for attribute)

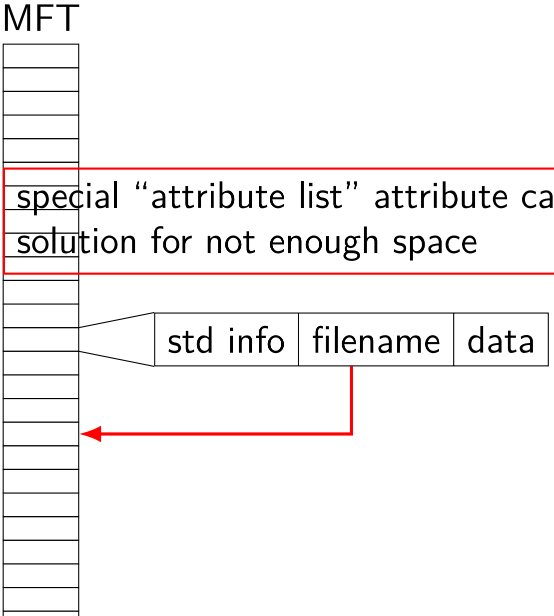


NTFS: Master File Table



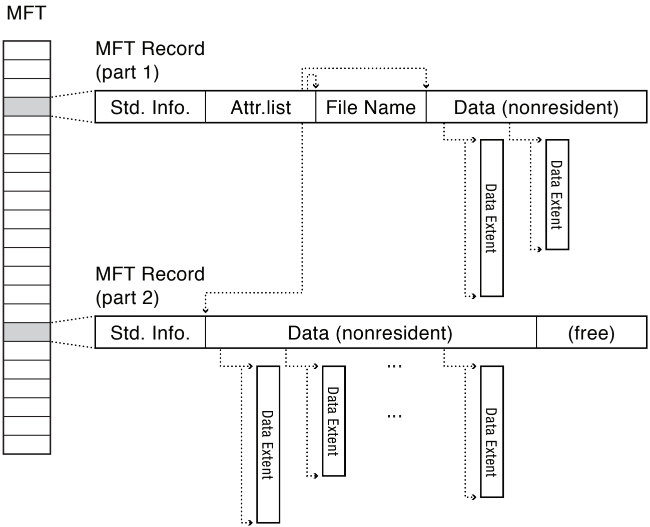
data is a type of attribute
small files: *in the MFT entry*
larger files: *pointer to extent of actual data*

NTFS: Master File Table



special "attribute list" attribute can point to extra MFT entry
solution for not enough space

NTFS file:



Anderson and Dahlin, *Operating Systems: Principles and Practice, Section Edition*, Figure 13.17

NTFS metadata

NTFS (current Windows FS) doesn't use inodes

has a Master File Table (MFT) containing file information

each 1KB entry: key-value pairs of info about file

too much info for 1KB — pointers to other entries

e.g. file stored as many, fragmented extents

NTFS metadata

NTFS (current Windows FS) doesn't use inodes

has a Master File Table (MFT) containing file information

each 1KB entry: key-value pairs of info about file

too much info for 1KB — pointers to other entries

e.g. file stored as many, fragmented extents

NTFS tricks

metadata stored in normal files

e.g. file for free block map

master file table is a file

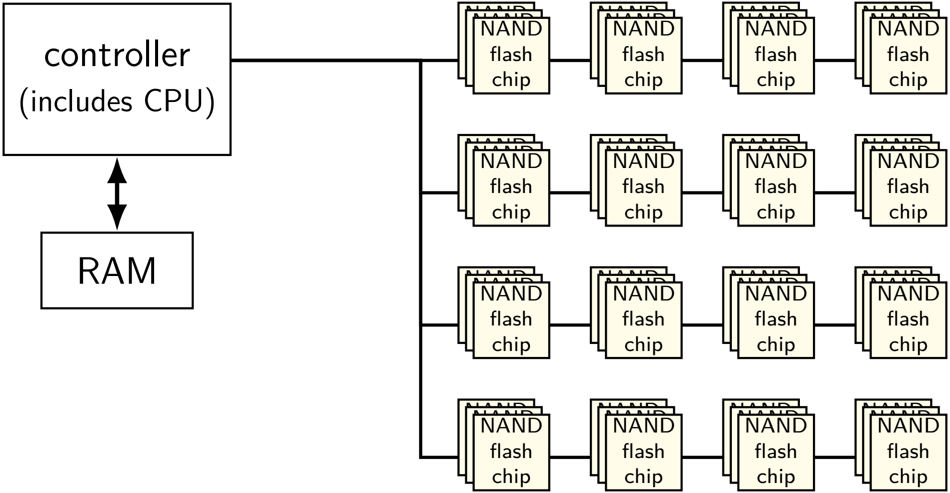
disk header has location of master file table

master file table itself is always first file

can change size of the master file table

small files — can store data in MFT entries

solid state disk architecture



flash

no moving parts

- no seek time, rotational latency

can read in sector-like sizes (“pages”) (e.g. 4KB or 16KB)

write once between erasures

erasure only in large *erasure blocks* (often 256KB to megabytes!)

can only rewrite blocks order tens of thousands of times

- afte that, flash fails

SSDs: flash as disk

SSDs: implement hard disk interface for NAND flash

- read/write sectors at a time
- read/write with use sector numbers, not addresses
- queue of read/writes

need to hide erasure blocks

- trick: block remapping — move where sectors are in flash

need to hide limit on number of erases

- trick: wear levening — spread writes out

block remapping

Flash Translation Layer

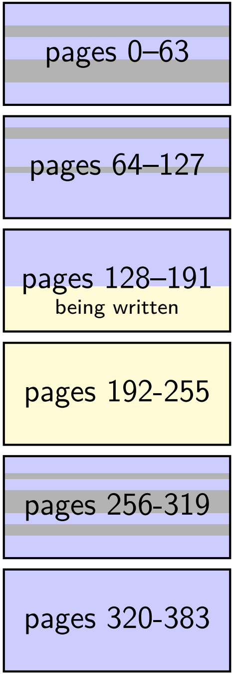
remapping table

logical	physical
0	93
1	260
...	...
31	74
32	75
...	...

active data

erased + ready-to-write

unused (rewritten elsewhere)



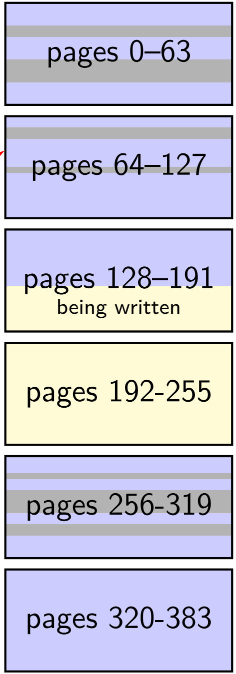
block remapping

read sector 31

Flash Translation Layer

remapping table

logical	physical
0	93
1	260
...	...
31	74
32	75
...	...

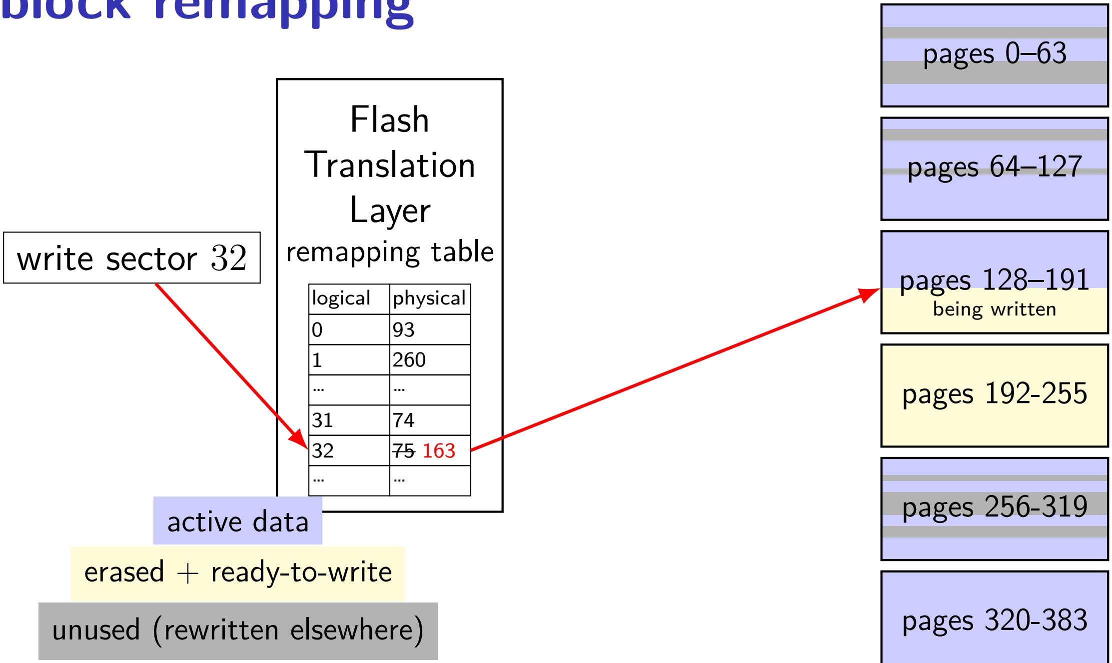


active data

erased + ready-to-write

unused (rewritten elsewhere)

block remapping



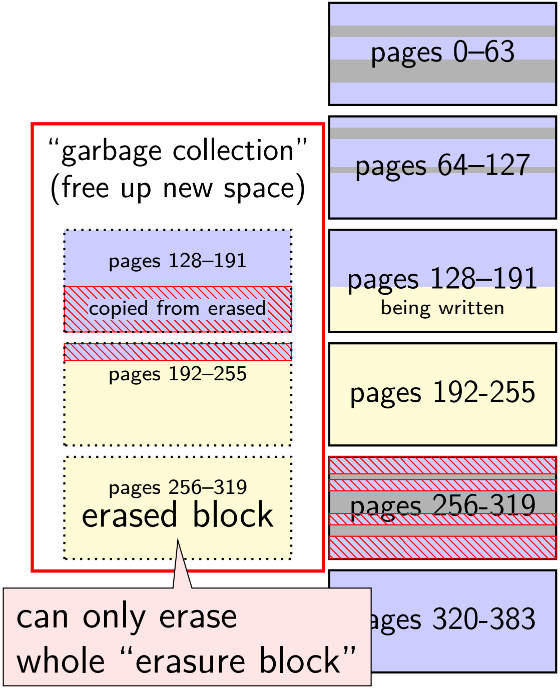
logical	physical
0	93
1	260
…	…
31	74
32	~~75~~ 163
…	…

block remapping

Flash Translation Layer

remapping table

logical	physical
0	93
1	~~260~~ 187
…	…
31	74
32	~~75~~ 163
…	…



active data

erased + ready-to-write

unused (rewritten elsewhere)

block remapping

controller contains mapping: sector $\rightarrow$ location in flash

on write: write sector to *new location*

eventually do *garbage collection* of sectors

- if erasure block contains some replaced sectors and some current sectors...
- copy current blocks to new locationt to reclaim space from replaced sectors

doing this efficiently is very complicated

SSDs sometimes have a 'real' processor for this purpose

SSD performance

reads/writes: sub-millisecond

contiguous blocks don't really matter

can depend a lot on the controller
- faster/slower ways to handle block remapping

writing can be slower, especially when almost full
- controller may need to move data around to free up erasure blocks
- erasing an erasure block is pretty slow (milliseconds?)