filesystem reliability

last time

inodes

(double-, triple-)indirect blocks

sparse files

hard and symbolic links

block groups for locality

extents and fragments

non-binary trees on disk

note on FAT assignment

you will need to use refernces

note: cluster 0 of FAT often not sector 0 of disk

references in assignment give actual correlation

also, see for format of FAT entries, etc.

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

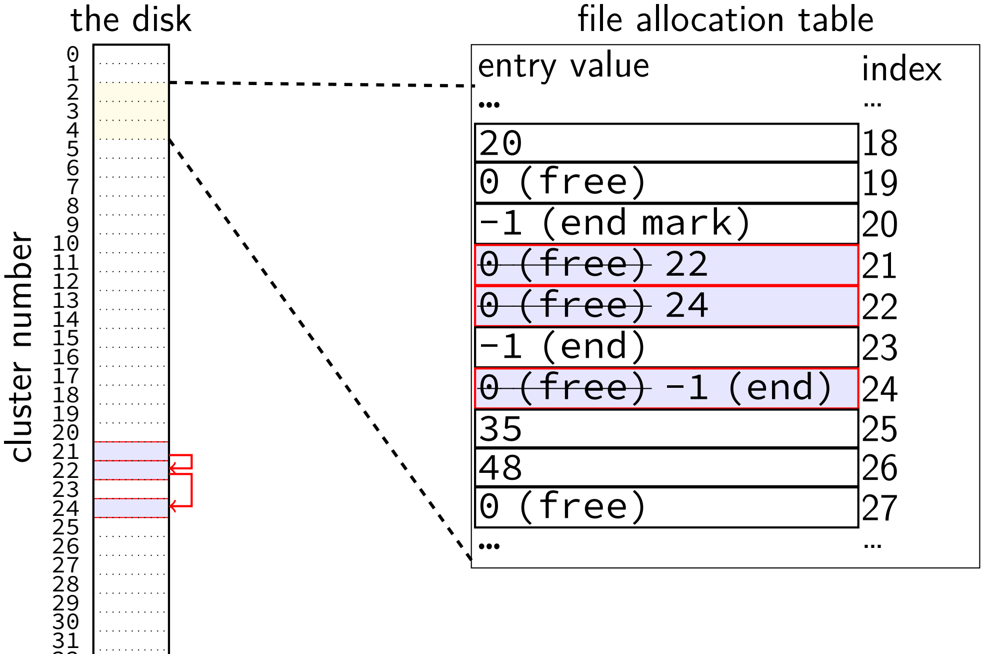
is the data there?

- can we find the file, etc.?

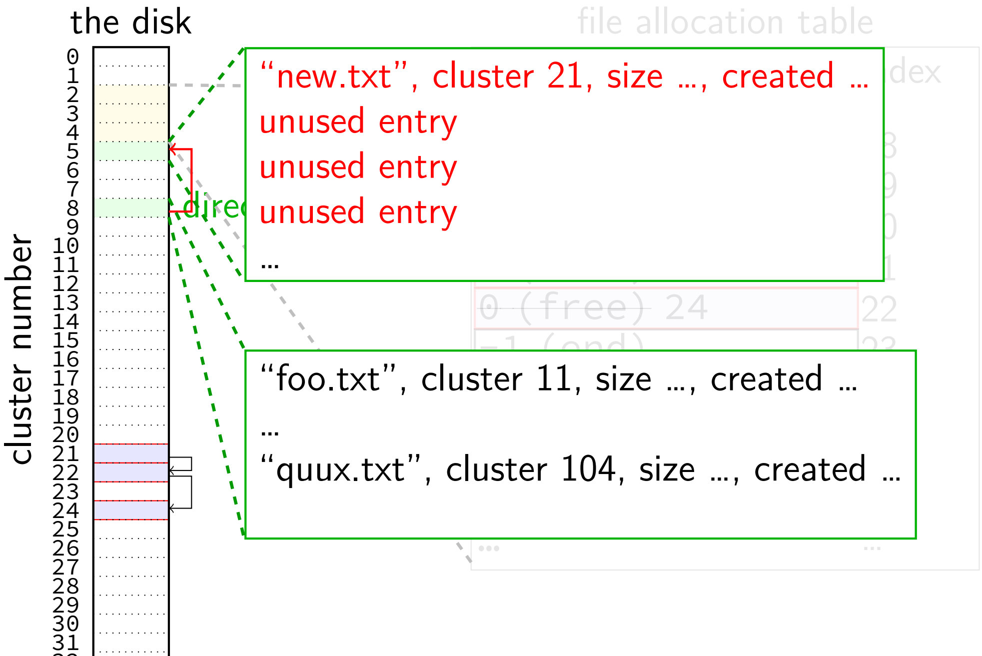
is the filesystem in a consistent state?

- do we know what blocks are free?

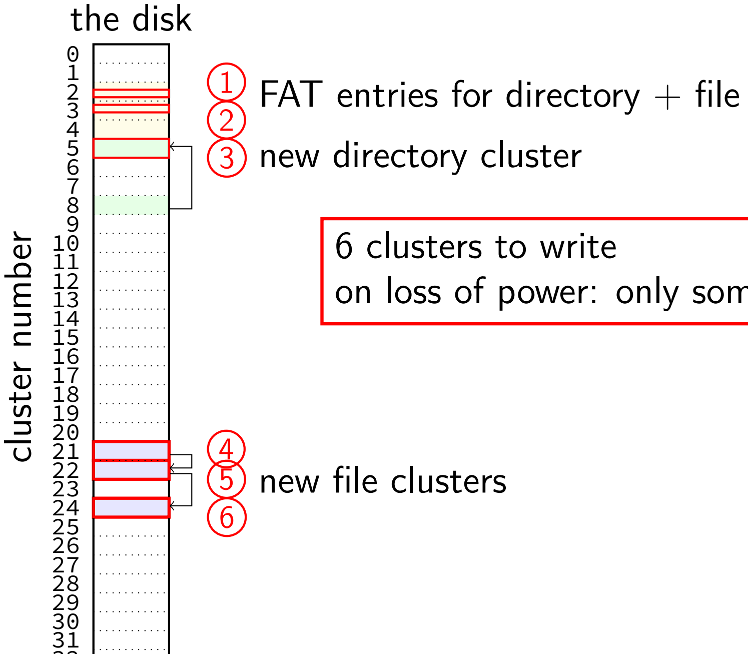
recall: FAT: file creation (1)



recall: FAT: file creation (2)



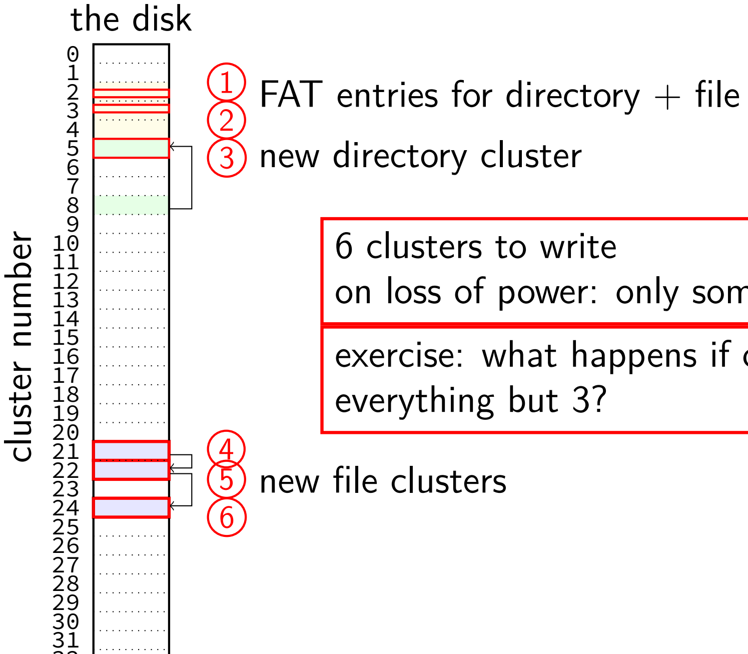
exercise: FAT file creation



6 clusters to write

on loss of power: only some completed

exercise: FAT file creation



6 clusters to write

on loss of power: only some completed

exercise: what happens if only 1, 2 complete?

everything but 3?

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

recovery with careful ordering

avoiding data loss → can 'fix' inconsistencies

programs like `fsck` (filesystem check), `chkdsk` (check disk)

- run manually or periodically or after abnormal shutdown

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

- Windows equivalent: `chkdsk`

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

maybe can fix other errors, too

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

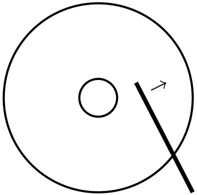
on boot: if clean bit clear, run fsck first

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

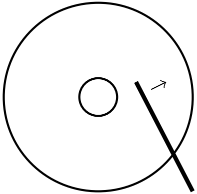


ordering and disk performance

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), …

beyond ordering

recall: updating a sector is atomic

happens entirely or doesn't

can we make filesystem updates work this way?

beyond ordering

recall: updating a sector is atomic

- happens entirely or doesn't

can we make filesystem updates work this way?

yes — 'just' make updating one sector do the update

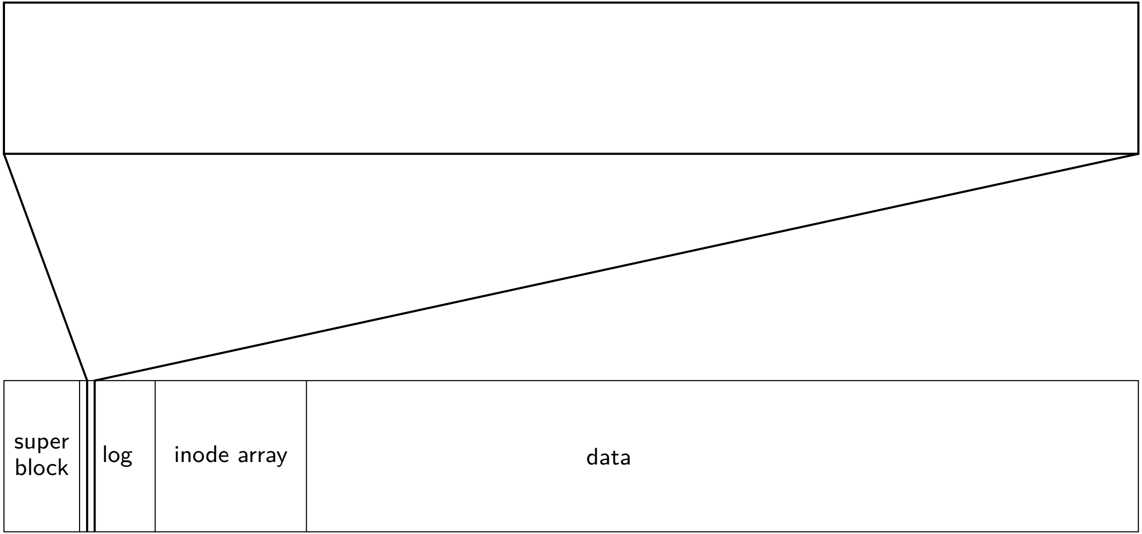
concept: transaction

transaction: bunch of updates that happen all at once

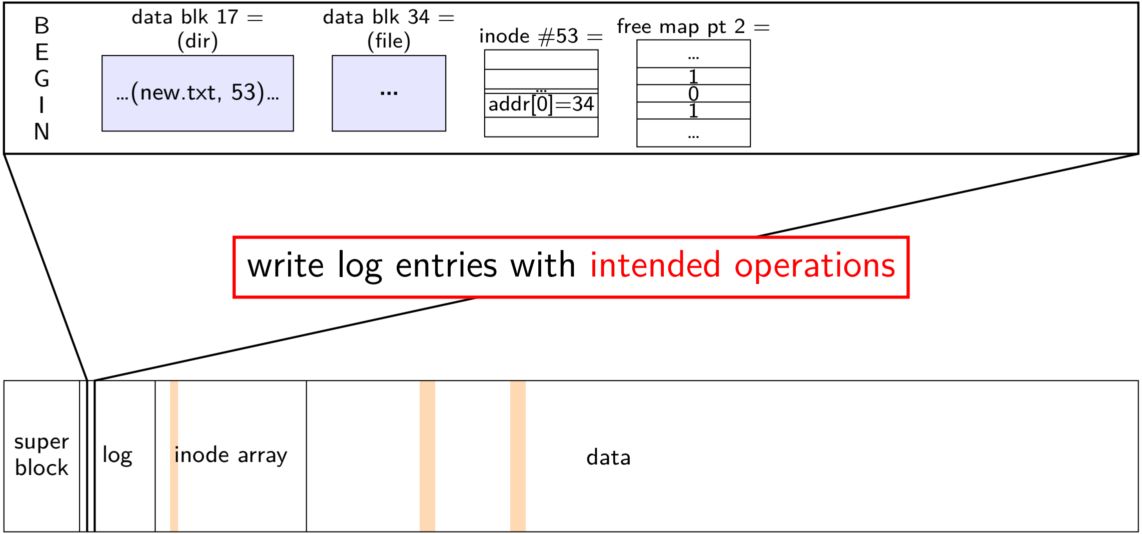
implementation trick: one update means transaction “commits”

- update done — whole transaction happened
- update not done — whole transaction did not happen

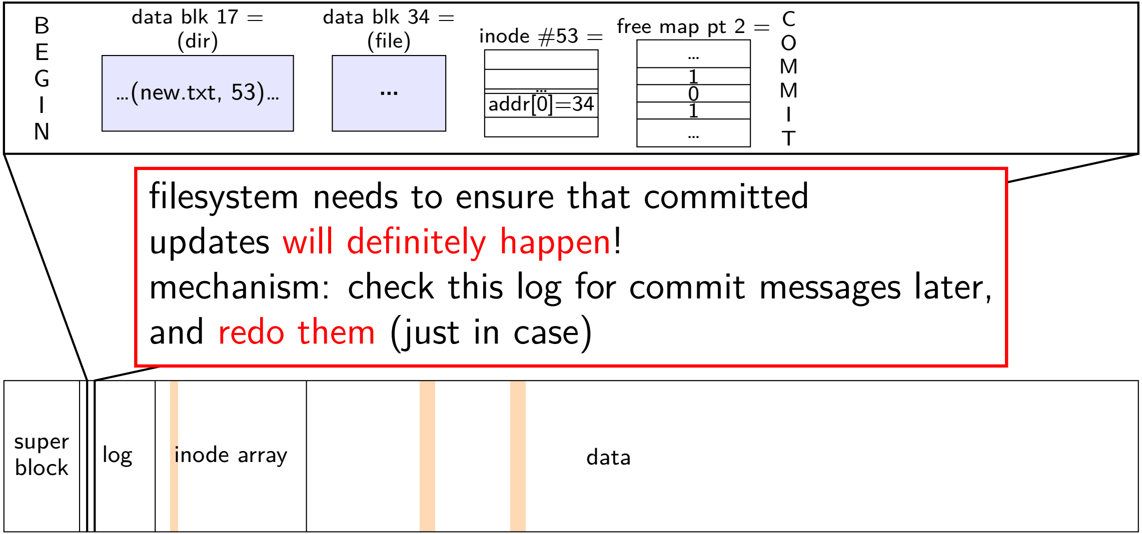
redo logging: file creation



redo logging: file creation



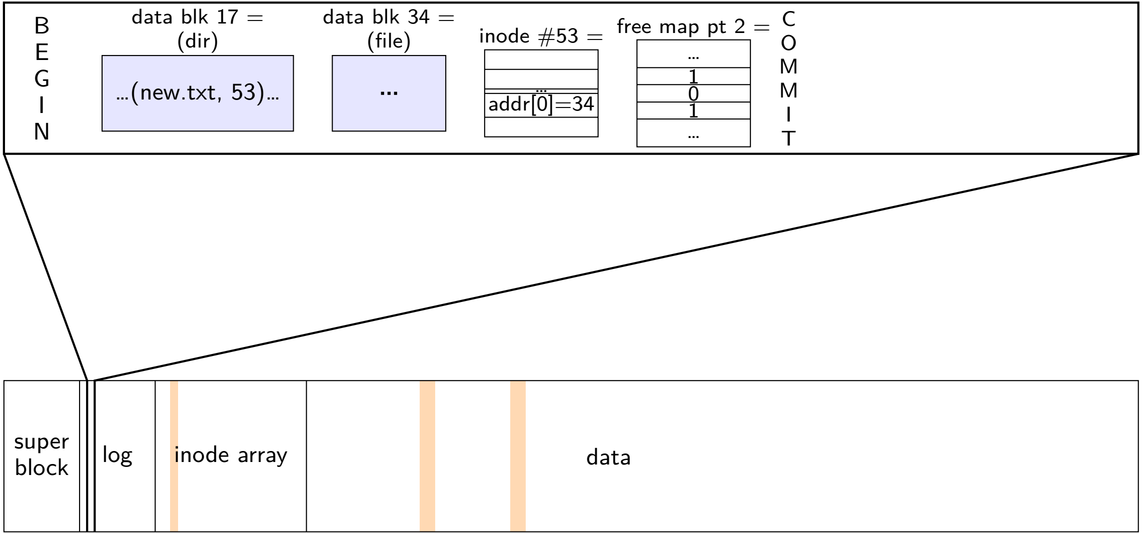
redo logging: file creation



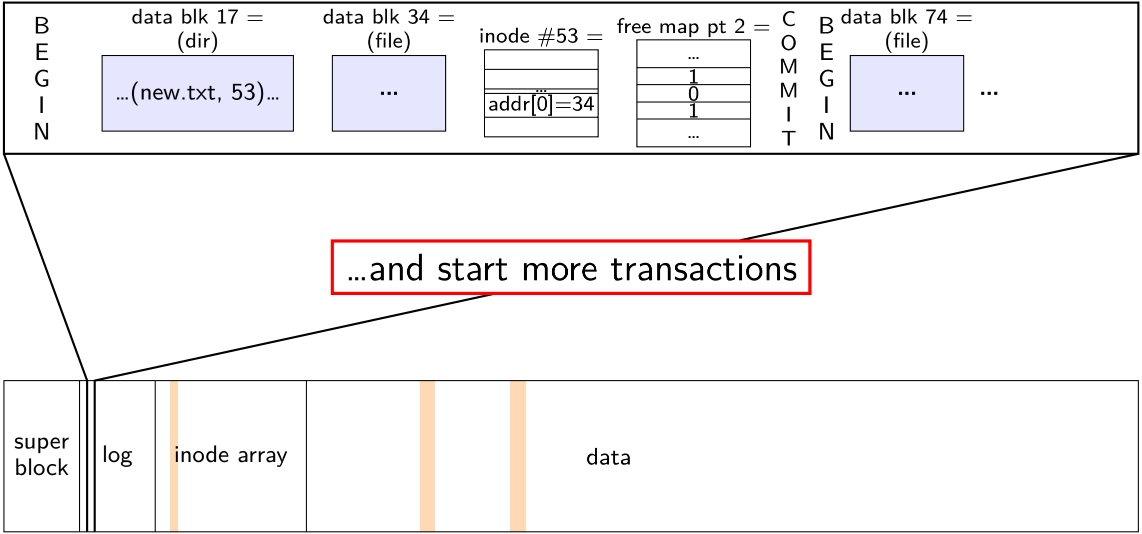
filesystem needs to ensure that committed updates will definitely happen!

mechanism: check this log for commit messages later, and redo them (just in case)

redo logging: file creation

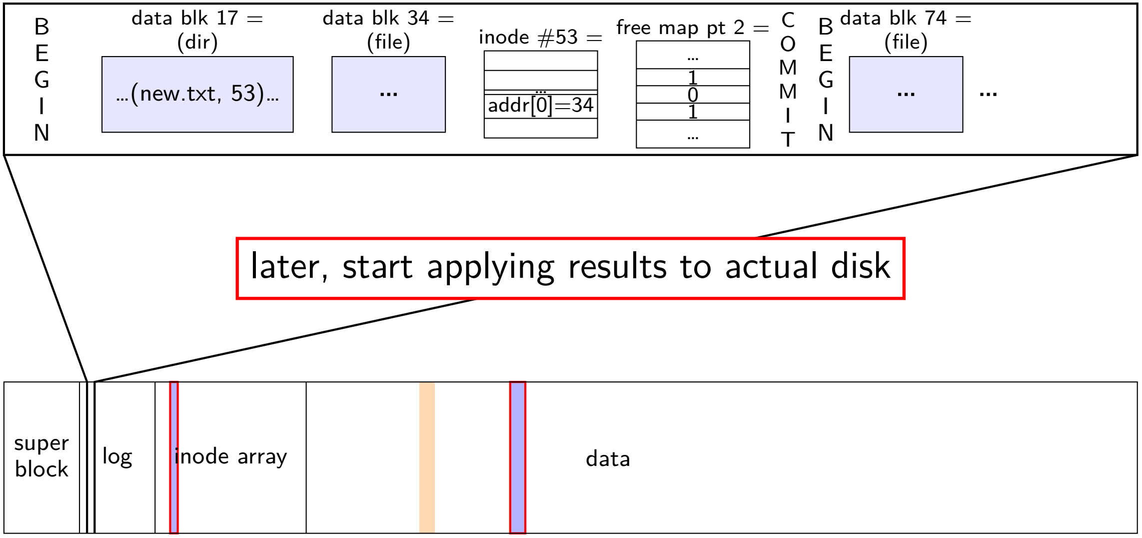


redo logging: file creation

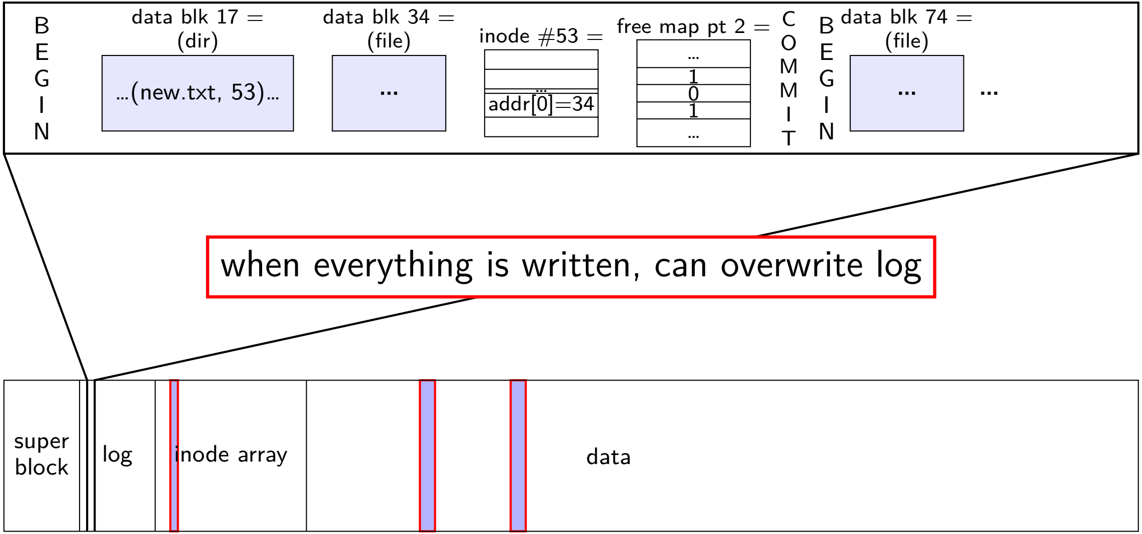


...and start more transactions

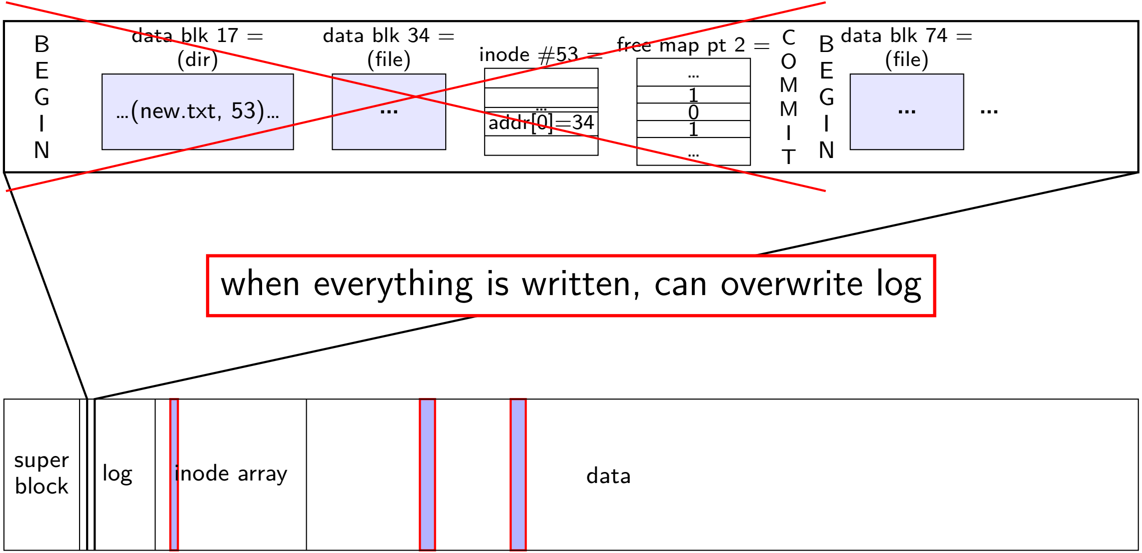
redo logging: file creation



redo logging: file creation



redo logging: file creation



redo logging: file creation

normal operation

write to log transaction steps:

- data blocks to create
- direcotry entry, inode to write
- directory inode (size, time) update

write to log "commit transaction"

in any order:

- update file data blocks
- update directory entry
- update file inode
- update directory inode

reclaim space in log

- "garbage collection"

redo logging: file creation

normal operation

write to log transaction steps:

- data blocks to create
- direcotry entry, inode to write
- directory inode (size, time) update

write to log "commit transaction"

in any order:

- update file data blocks
- update directory entry
- update file inode
- update directory inode

reclaim space in log

- "garbage collection"

crash before *commit*?

file not created

no partial operation to real data

redo logging: file creation

normal operation

write to log transaction steps:

- data blocks to create
- direcotry entry, inode to write
- directory inode (size, time) update

write to log "commit transaction"

in any order:

- update file data blocks
- update directory entry
- update file inode
- update directory inode

reclaim space in log

- "garbage collection"

crash after *commit*?

file created

promise: will perform logged updates

(after system reboots/recovers)

redo logging: file creation

normal operation

write to log transaction steps:

- data blocks to create
- direcotry entry, inode to write
- directory inode (size, time) update

write to log “commit transaction”

in any order:

- update file data blocks
- update directory entry
- update file inode
- update directory inode

reclaim space in log

- “garbage collection”

redo logging: file creation

normal operation

write to log transaction steps:

- data blocks to create
- direcotry entry, inode to write
- directory inode (size, time) update

write to log "commit transaction"

in any order:

- update file data blocks
- update directory entry
- update file inode
- update directory inode

reclaim space in log

- "garbage collection"

recovery

read log and...

ignore any operation with no "commit"

redo any operation with "commit"

- already done? — okay, setting inode twice

reclaim space in log

idempotency

logged operations should be *okay to do twice* $=$ *idempotent*

good example: set inode link count to 4

bad example: increment inode link count

good example: overwrite inode number X with new value

as long as last committed inode value in log is right...

bad example: allocate new inode with particular contents

good example: overwrite data block with new value

bad example: append data to last used block of file

redo logging summary

write intended operation to the log

- before ever touching ‘real’ data
- in format that’s safe to do twice

write marker to commit to the log

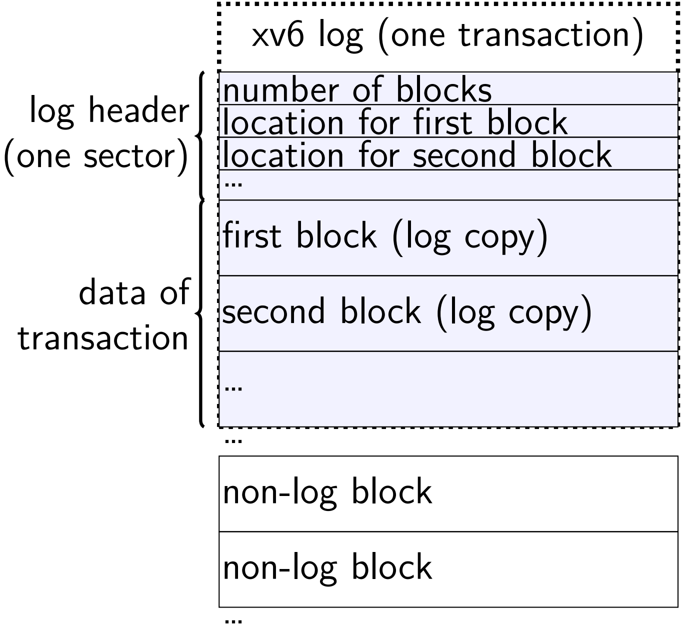
- if exists, the operation *will be done eventually*

actually update the real data

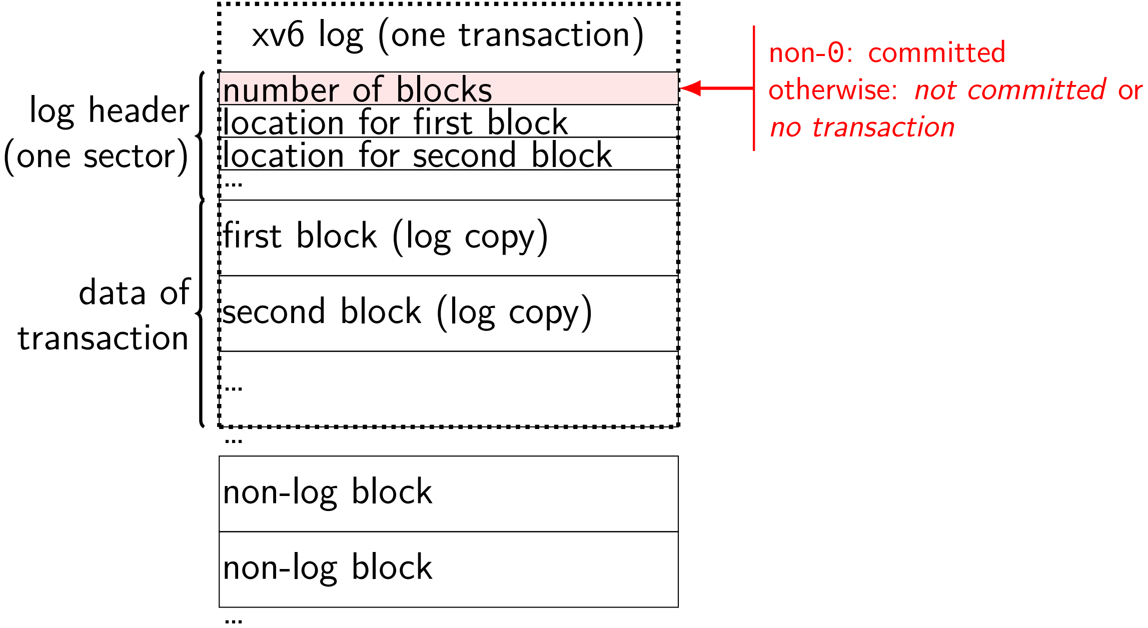
redo logging and filesystems

filesystems that do redo logging are called *journalling filesystems*

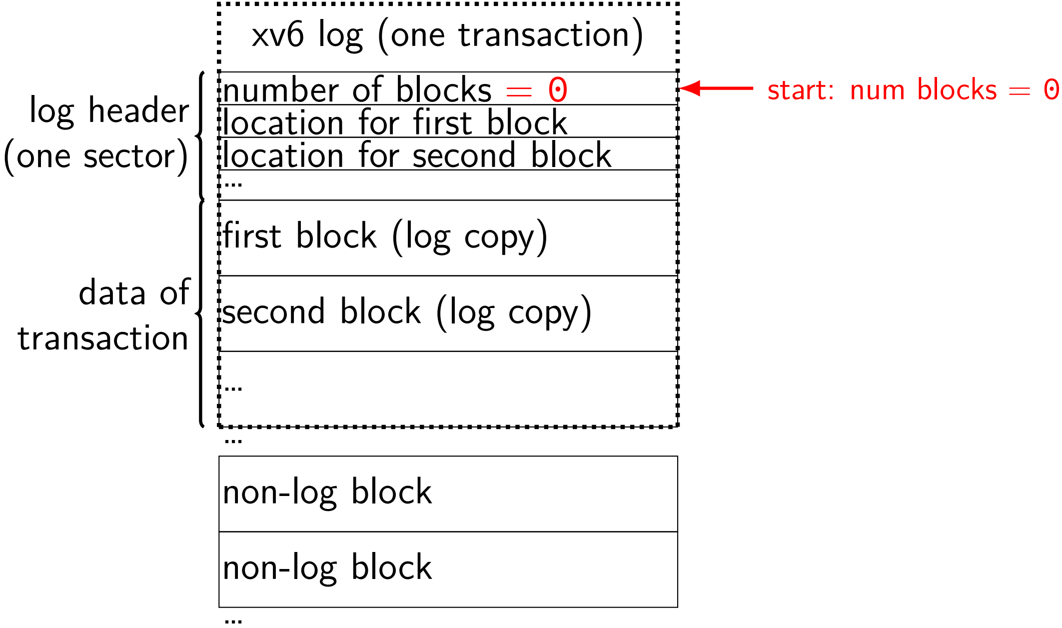
the xv6 journal



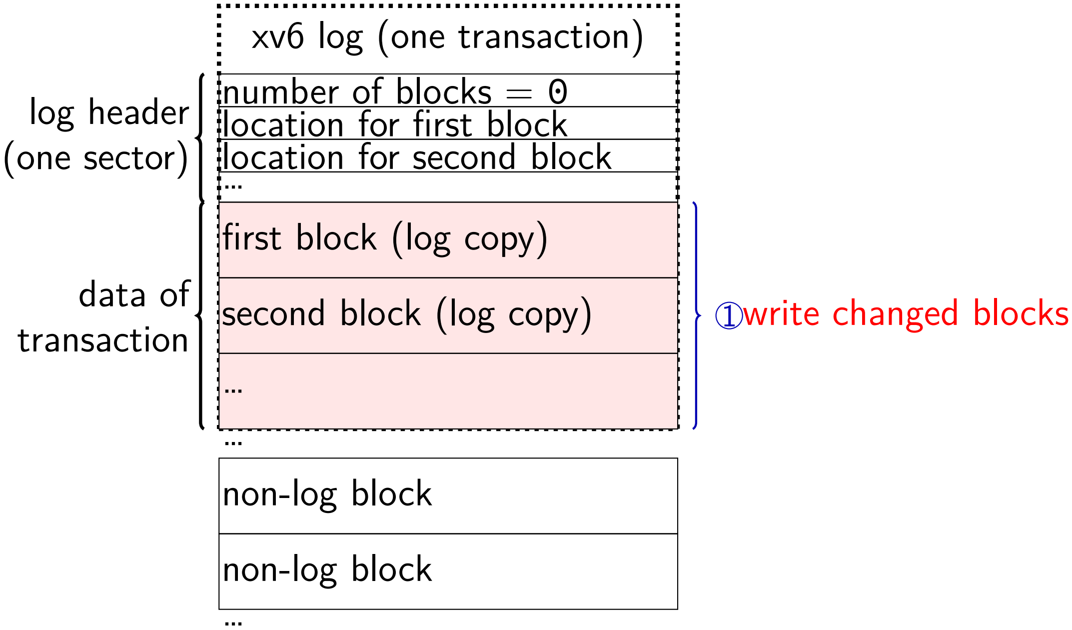
the xv6 journal



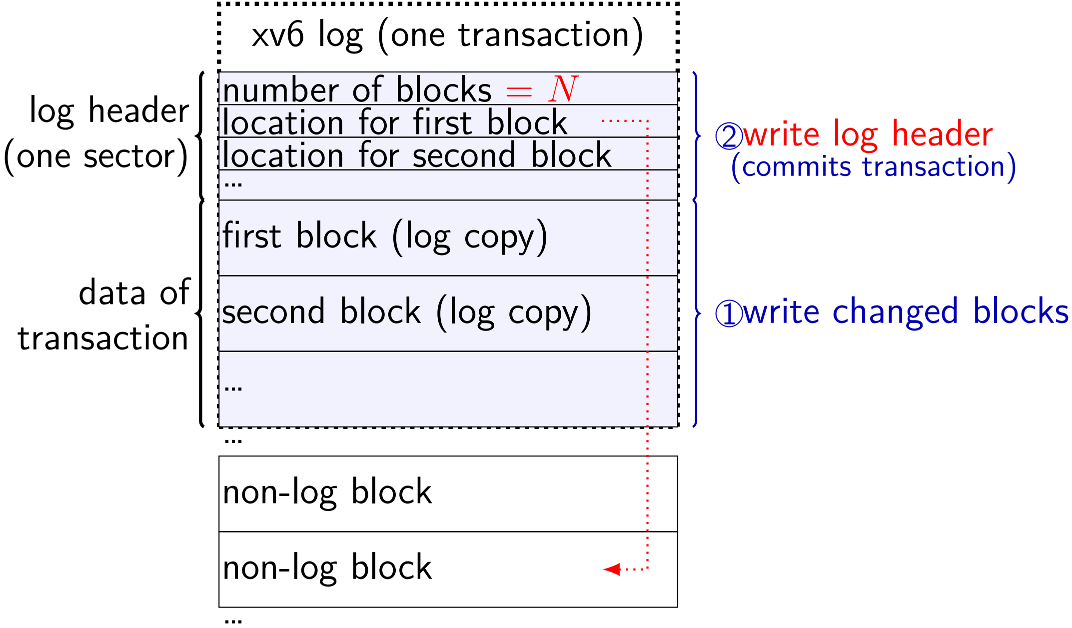
the xv6 journal



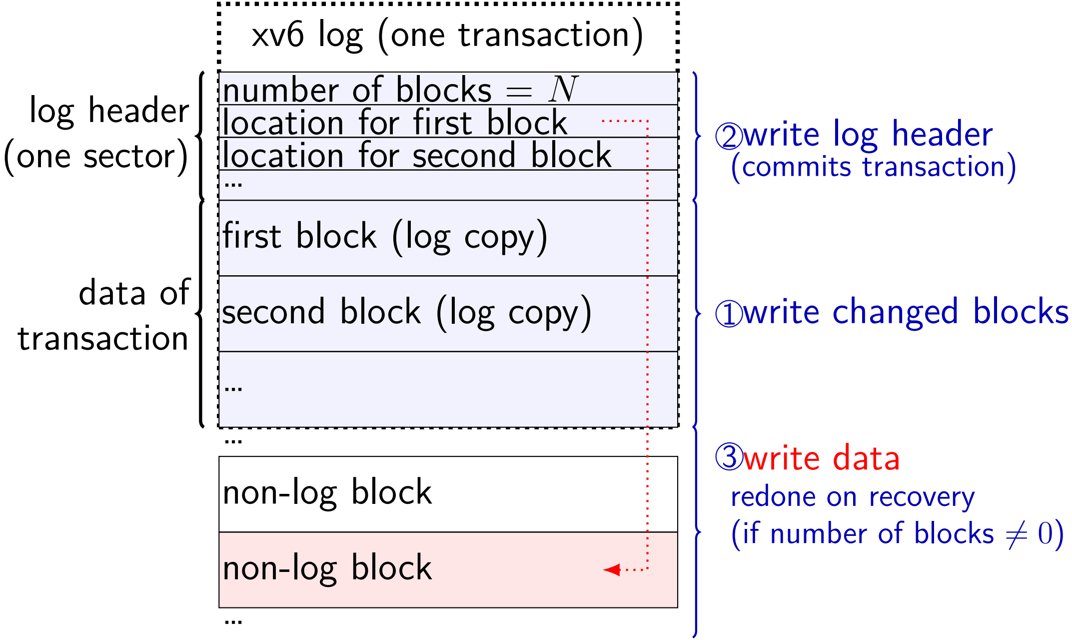
the xv6 journal



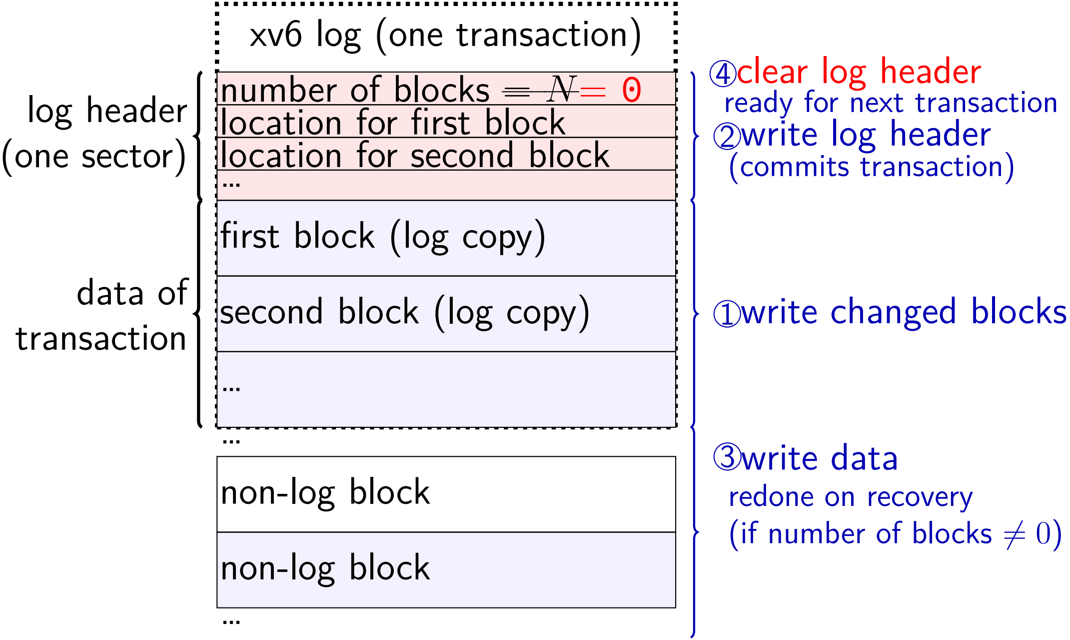
the xv6 journal



the xv6 journal



the xv6 journal



what is a transaction?

so far: each file update?

faster to do batch of updates together

- one log write finishes lots of things
- don't wait to write

xv6 solution: combine lots of updates into one transaction

only commit when...

- no active file operation, *or*
- not enough room left in log for more operations

what is a transaction?

so far: each file update?

faster to do batch of updates together

- one log write finishes lots of things
- don't wait to write

xv6 solution: combine lots of updates into one transaction

only commit when...

- no active file operation, *or*
- not enough room left in log for more operations

redo logging problems

doesn't the log get infinitely big?

writing everything twice?

redo logging problems

doesn't the log get infinitely big?

writing everything twice?

limiting log size

once transaction is written to real data, can discard

sometimes called "garbage collecting" the log

may sometimes need to block to free up log space

- perform logged updates before adding more to log

hope: usually log cleanup happens "in the background"

redo logging problems

doesn't the log get infinitely big?

writing everything twice?

lots of writing? (1)

entire log can be written sequentially

- ideal for hard disk performance
- also pretty good for SSDs

multiple updates can be done in any order

- can reorder to minimize seek time/rotational latency/etc.
- can interleave updates that make up multiple transactions

no waiting for 'real' updates

- application can proceed while updates are happening
- files will be updated even if system crashes

often better for performance!

lots of writing? (2)

updating 1000 files?

with redo logging — 2 big seeks
- write all updates to log in order
- write all updates to file/inode/directory data in order

lots of writing? (2)

updating 1000 files?

with redo logging — 2 big seeks

- write all updates to log in order
- write all updates to file/inode/directory data in order

careful ordering — lots of seeks?

- write to free block map
- seek + write to inode
- seek + write to directory entry
- repeat 1000x

maybe could also combine file updates with careful ordering??

- but sure starts to get complicated to track order requirements
- redo logging is probably simpler?

degrees of consistency

not all journalling filesystem use redo logging for everything

some use it *only for metadata operations*

some use it *for both metadata and user data*

only metadata: avoids lots of duplicate writing

metadata+user data: integrity of user data guaranteed

multiple copies

FAT: multiple copies of file allocation table and header

in inode-based filesystems: often multiple copies of superblocks

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

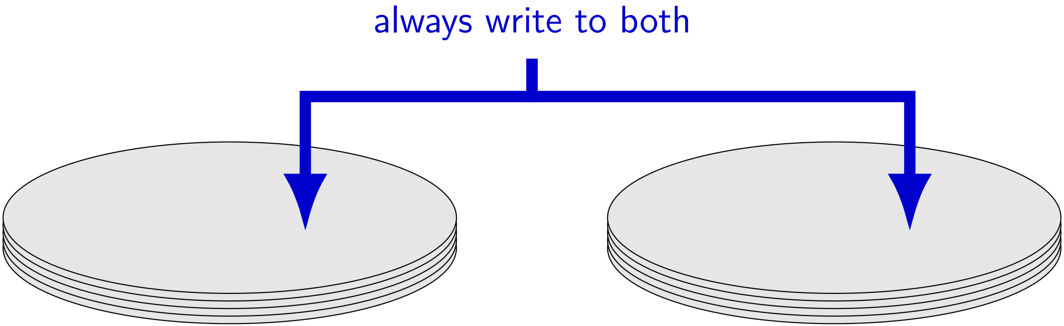
- always update both copies
- hope: disk failure to small group of sectors

hope: enough to recover most files on disk failure

- extra copy of metadata that is important for all files
- but won't recover specific files/directories whose data was lost

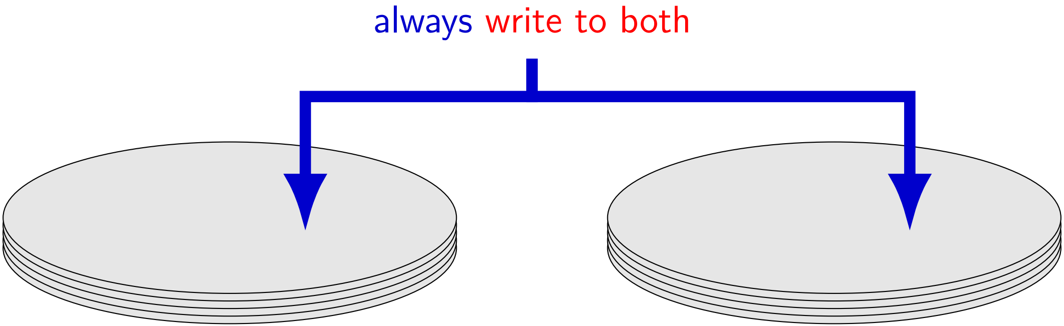
mirroring whole disks

alternate strategy: write everything to two disks



mirroring whole disks

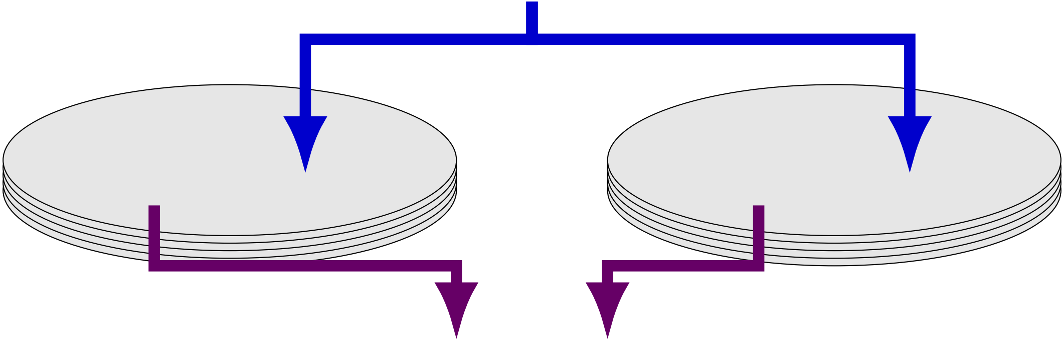
alternate strategy: write everything to two disks



mirroring whole disks

alternate strategy: write everything to two disks

always write to both



read from either
(or different parts of both – faster!)

beyond mirroring

mirroring seems to waste a lot of space

10 disks of data? mirroring $\rightarrow$ 20 disks

10 disks of data? how good can we do with 15 disks?

best possible: lose 5 disks, still okay

- can't do better or it wasn't really 10 disks of data

schemes that do this based on *erasure codes*

- erasure code: encode data in way that handles parts missing (being erased)

erasure code example

store 2 disks of data on 3 disks

recompute original 2 disks of data from any 2 of the 3 disks

extra disk of data: some formula based on the original disks

- common choice: bitwise XOR

common set of schemes like this: RAID

- Redundant Array of Independent Disks

snapshots

filesystem snapshots

idea: filesystem keeps old versions of files around

- accidental deletion? old version stil there
- eventually discard some old versions

can access *snapshot* of files at prior time

snapshots

filesystem snapshots

idea: filesystem keeps old versions of files around

- accidental deletion? old version stil there
- eventually discard some old versions

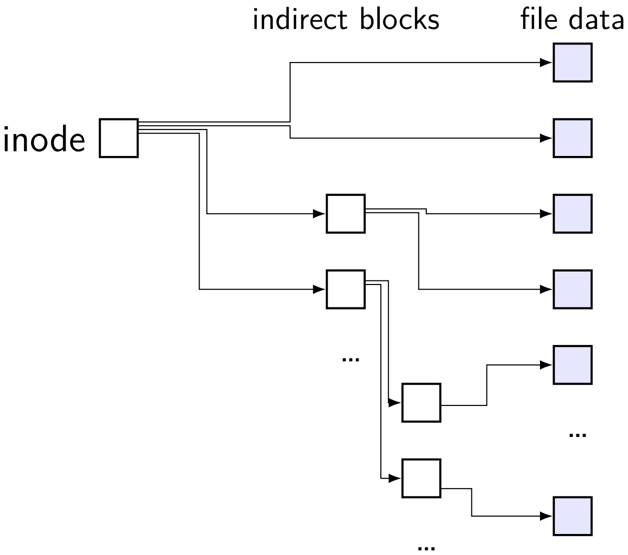
can access *snapshot* of files at prior time

mechanism: copy-on-write

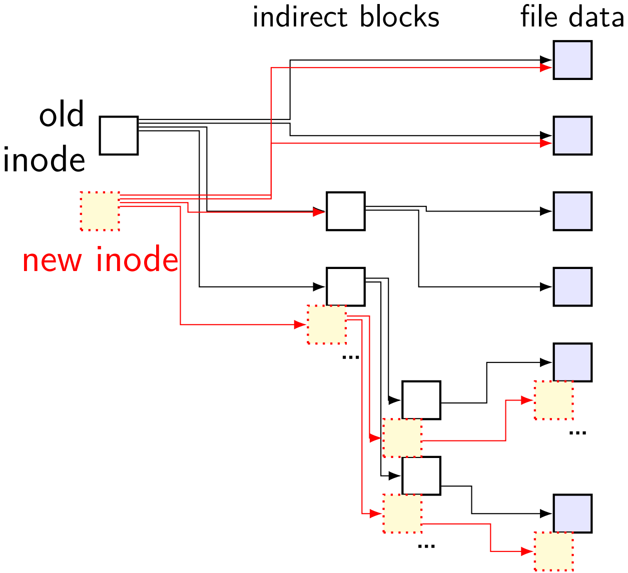
changing file makes new copy of filesystem

common parts shared between versions

inode and copy-on-write



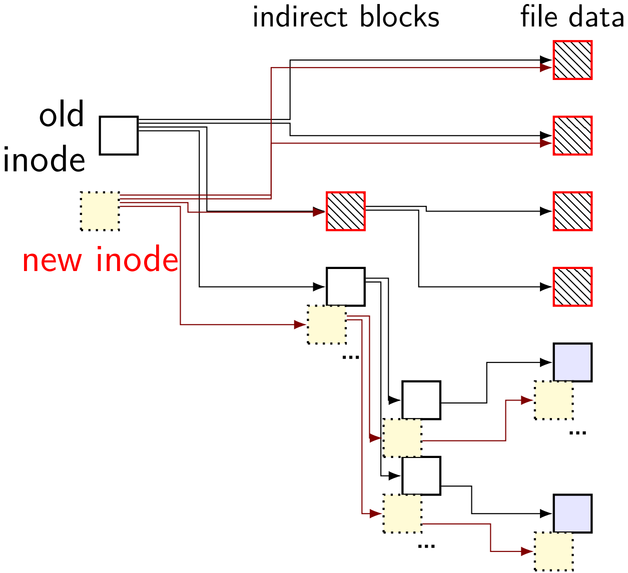
inode and copy-on-write



update: new data blocks
+ new indirect blocks
+ new inode

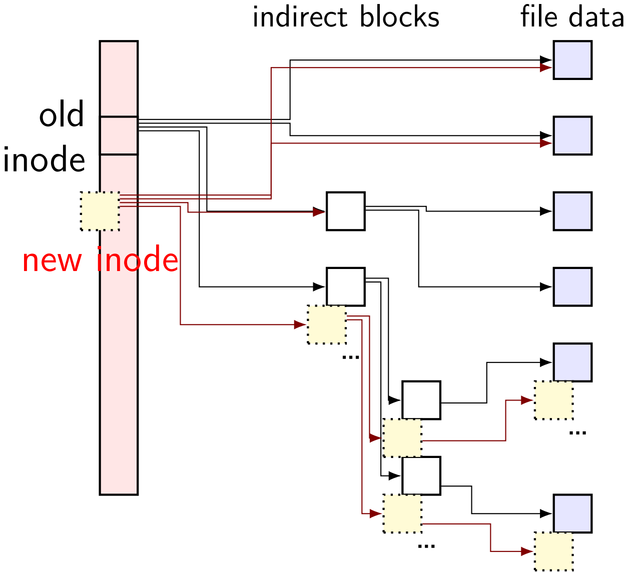
both old+new inode valid

inode and copy-on-write



unchanged parts of file shared

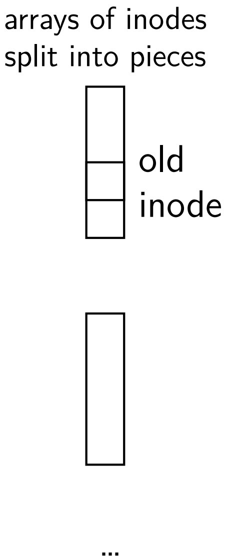
inode and copy-on-write



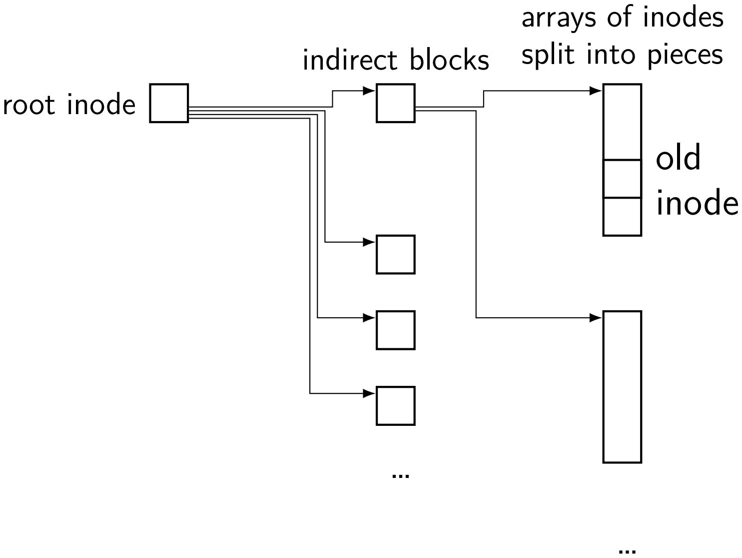
challenge: FFS/xv6/ext2 design has big array of inodes

don't want to write new copy of *entire inode array*

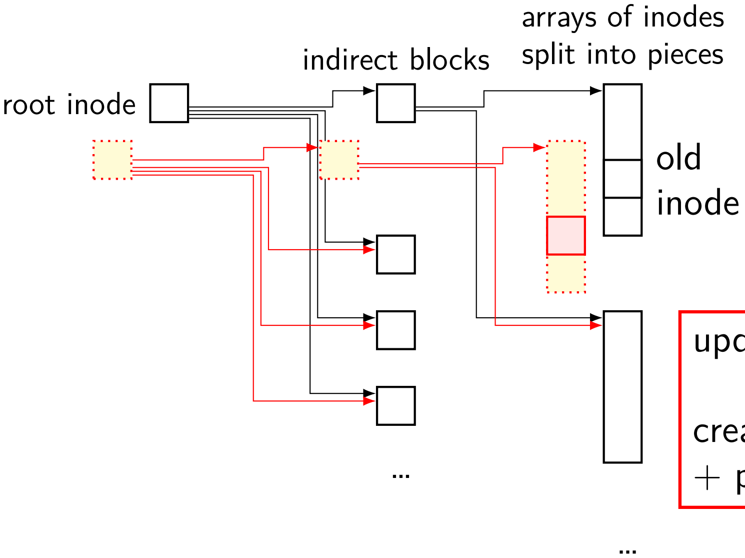
extra indirection for inode array



extra indirection for inode array



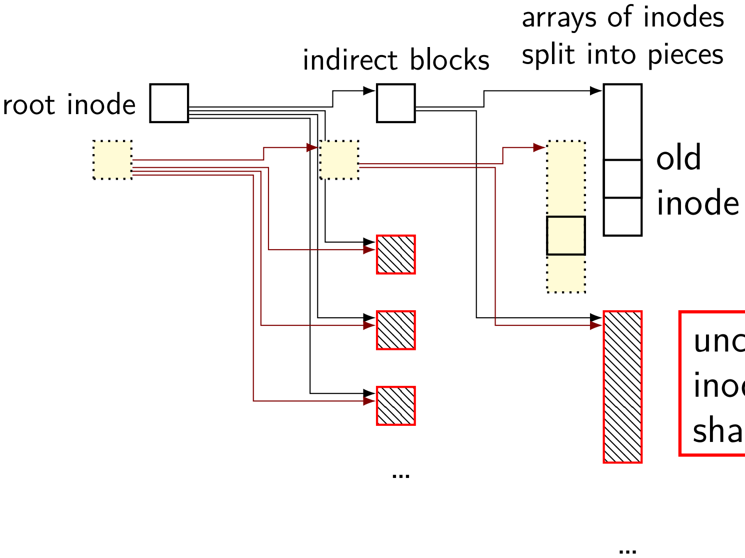
extra indirection for inode array



update one inode?

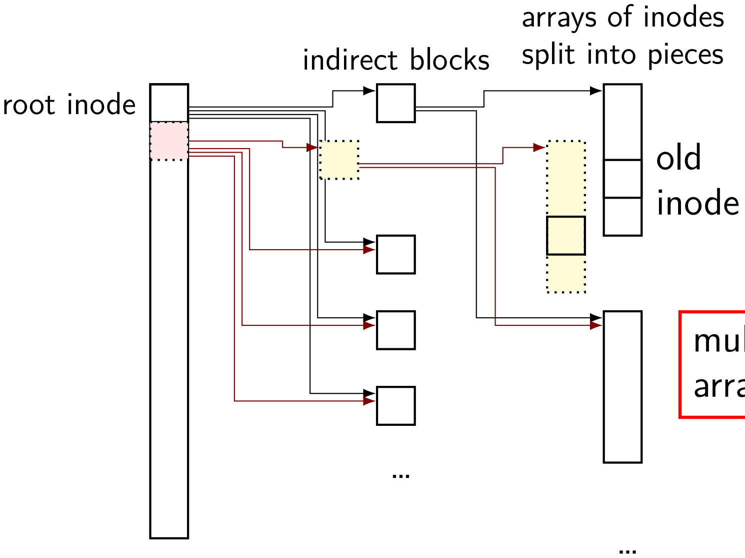
create new root inode
+ pointers

extra indirection for inode array



unchanged parts of
inode array
shared between versions

extra indirection for inode array



multiple snapshots?
array of root inodes

copy-on-write indirection

file update = replace with new version

array of versions of entire filesystem

only copy modified parts

- keep reference counts, like for paging assignment

lots of pointers — only change pointers where modifications happen

snapshots in practice

ZFS supports this (if turned on)

example: `.zfs/snapshots/11.11.18-06` pseudo-directory

contains contents of files at 11 November 2018 6AM

backup/if time slides

copy-on-write and logging

copy-on-write is a nice solution to duplicate writes

before (data journalling)

- write new data to journal
- copy new data to real location

after (copy-on-write)

- write new data to new location
- update pointer to point to new locatoin

useful even without snapshots

- but maybe not keeping file data in best place?

aside: fsync

filesystem can order things carefully

filesystem can make sure data on disk before proceeding

what if I, non-OS programmer want to do that?

POSIX mechanism: `fsync`

- "please actually write this file to disk now — I'll wait"

some stories of broken implementations of `fsync`

- nasty problem — how do you test it???

some varying interpretations

- some only send to disk, but *don't wait for disk to finish writing*
- does not gaurenteeing updating file's directory entry

changing file atomically?

often applications want to update a file all at once

changing file atomically?

often applications want to update a file all at once

on Unix, one way to do this:

create a new file with a hard-to-guess name in the same directory

rename the new file to replace the old file

- overwrites that directory entry

no one will ever read partially written file

log-structured filesystems

logging is a great access pattern for hard drives and SSDs

- sequential
- right for SSDs — write everything once before writing again

how about designing a filesystem around it!

idea: log-structured filesystems

log-structured filesystem

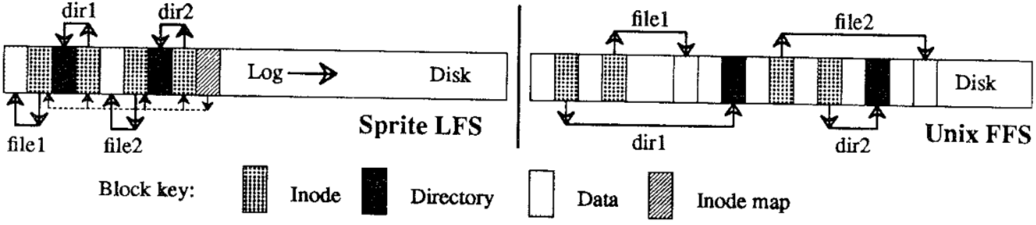


image: Rosenblum and Ousterhout, "The Design and Implementatoin of a Log Structures Filesystem"

log-structured filesystem ideas

write inodes + data + free map + etc. to log instead of disk

problem: scanning log to find latest version of inode?

periodically write *inode maps* to log

computed latest location of inodes

searching limited to last inode map

log-structured FS garbage collection

challenge: what happens when log gets to the end of the disk?

want to start from beginning of disk again...

either: copy data to free space or 'thread' log around used space:

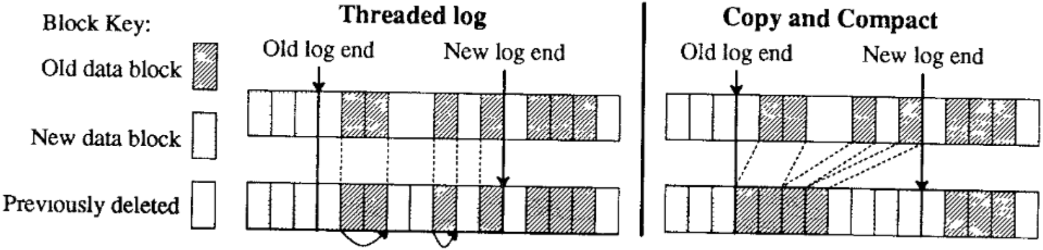


image: Rosenblum and Ousterhout, "The Design and Implementatoin of a Log Structures Filesystem"

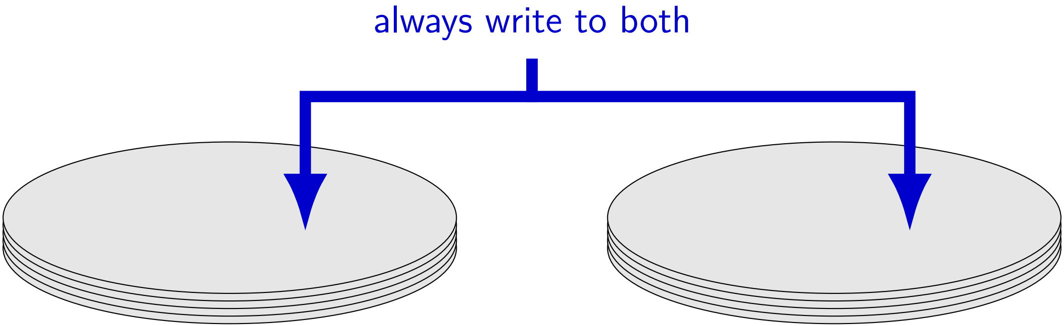
log-structured filesystems in practice

the kind of ideas you'd use to implement an SSD

used for some filesystems that work directly with Flash chips

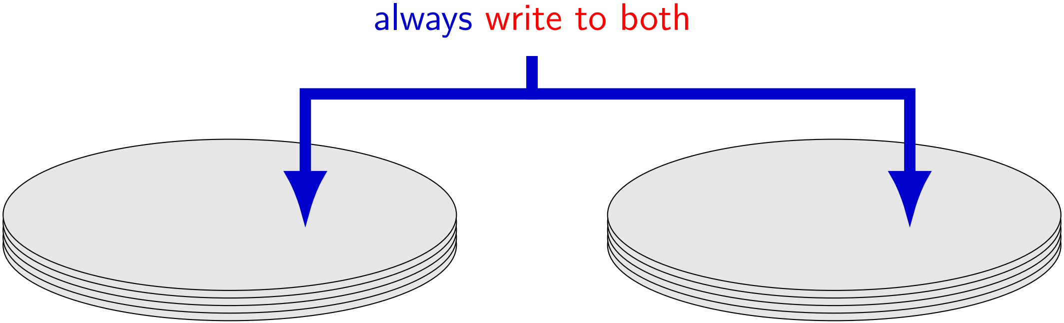
mirroring whole disks

alternate strategy: write everything to two disks



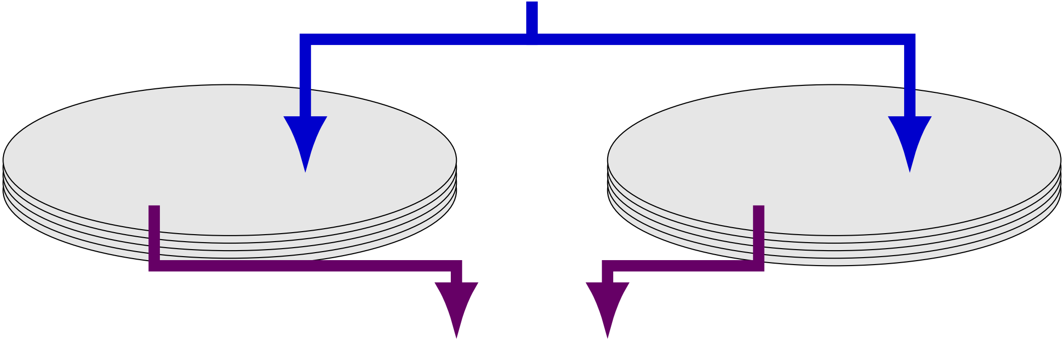
mirroring whole disks

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!

RAID: missing piece

what about losing data while blocks being updated

very tricky/failure-prone part of RAID implementations