filesystems 3

last time

hard drives
- seek time
- rotational latency
- block remapping by disk controller

solid state disks
- wear leveling — move data to new block when written
- work around limitations of large erasure blocks (that can't be rewritten too much)

inode-based filesystems
- inode: all data about a file
- fixed-sized array of inodes on disk
- seperate tracking of free blocks
- directory contains only (name, index into inode array)

indirect block advantages

small files: all direct blocks $+$ no extra space beyond inode

larger files — more indirection

    file should be large enough to hide extra indirection cost

(log N)-like time to find block for particular offset

    no linear search like FAT

xv6 FS pros versus FAT

support for reliability — log
- more on this later

possibly easier to scan for free blocks
- more compact free block map

easier to find location of kth block of file
- element of addrs array

file type/size information held with block locations
- inode number $=$ everything about open file
- easier to read/modify file info all at once?

missing pieces

what's the log? (more on that later)

other file metadata?

- creation times, etc. — xv6 doesn't have it

not good at taking advantage of HDD architecture

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

Fast File System

the Berkeley Fast File System (FFS) ‘solved’ some of these problems

- McKusick et al, “A Fast File System for UNIX” `https://people.eecs.berkeley.edu/~brewer/cs262/FFS.pdf`
- avoids long seek times, wasting space for tiny files

Linux’s ext2 filesystem based on FFS

some other notable newer solutions (beyond what FFS/ext2 do)

- better handling of very large files
- avoiding linear directory searches

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

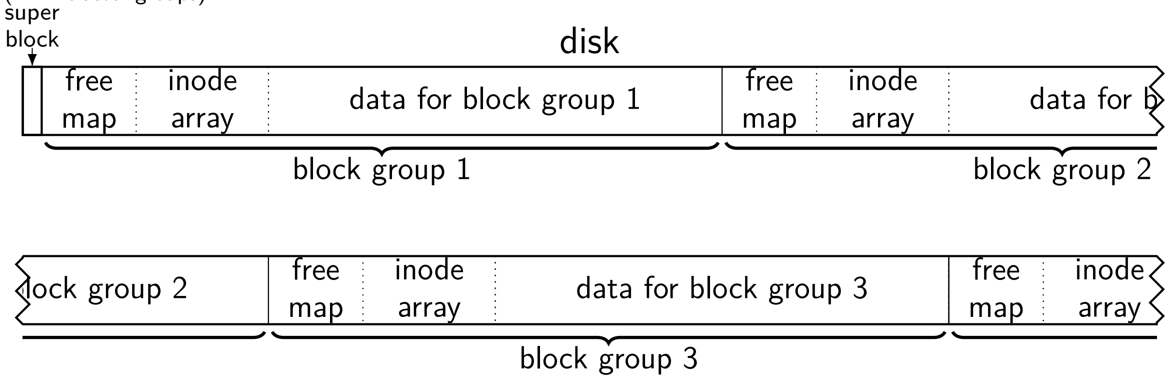
blocks are pretty small — needs lots of space for metadata

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

linear searches of directory entries to resolve paths

block groups

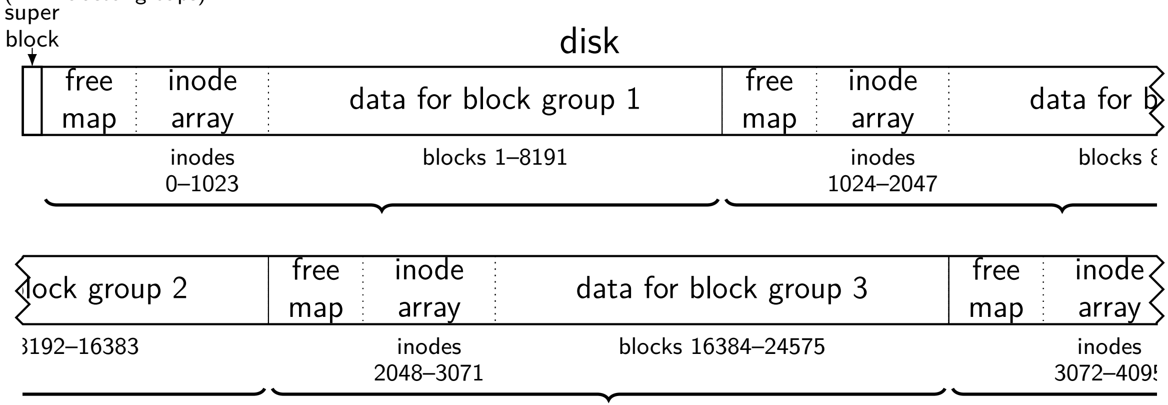
(AKA cluster groups)



split disk into block groups
each block group like a mini-filesystem

block groups

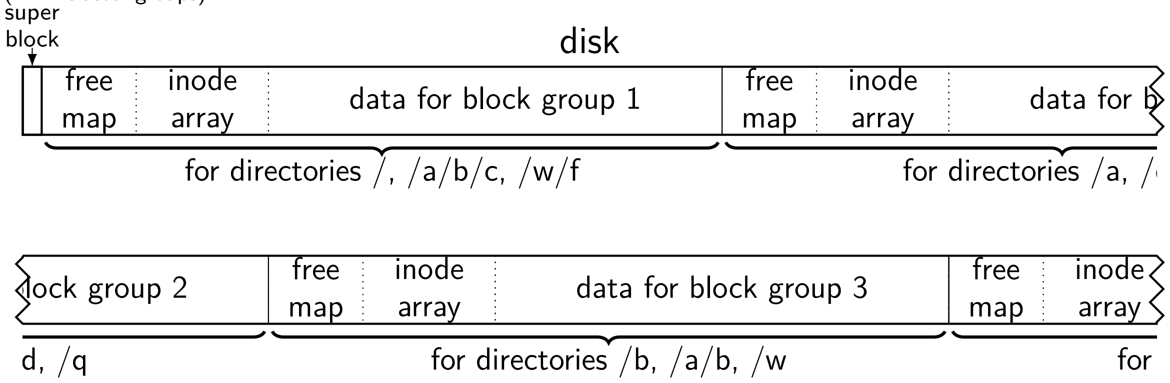
(AKA cluster groups)



split block + inode numbers across the groups
inode in one block group can reference blocks in another
(but would rather not)

block groups

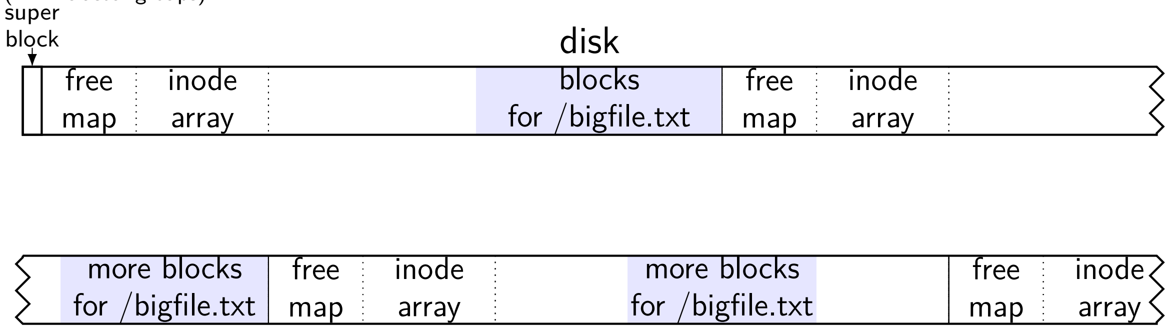
(AKA cluster groups)



goal: *most data* for each directory within a block group
directory entries + inodes + file data close on disk
lower seek times!

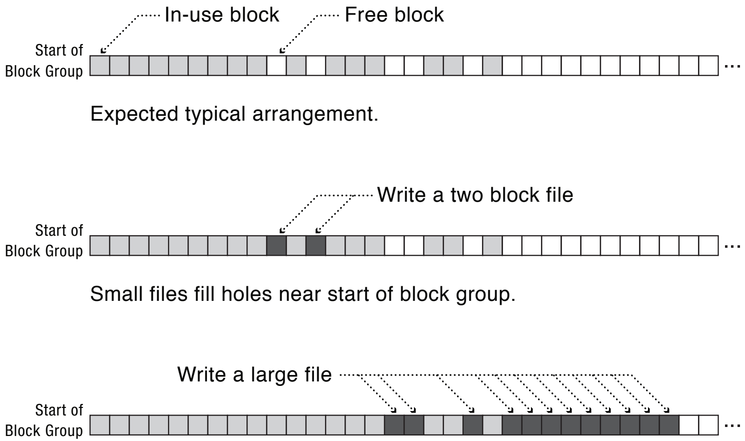
block groups

(AKA cluster groups)



large files might need to be split across block groups

allocation within block groups



Expected typical arrangement.

Small files fill holes near start of block group.

Large files fill holes near start of block group and then write most data to sequential range blocks.

FFS block groups

making a subdirectory: new block group
- for inode + data (entries) in different

writing a file: same block group as directory, first free block
- intuition: non-small files get contiguous groups at end of block
- FFS keeps disk deliberately underutilized (e.g. 10% free) to ensure this

can wait until dirty file data flushed from cache to allocate blocks
- makes it easier to allocate contiguous ranges of blocks

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

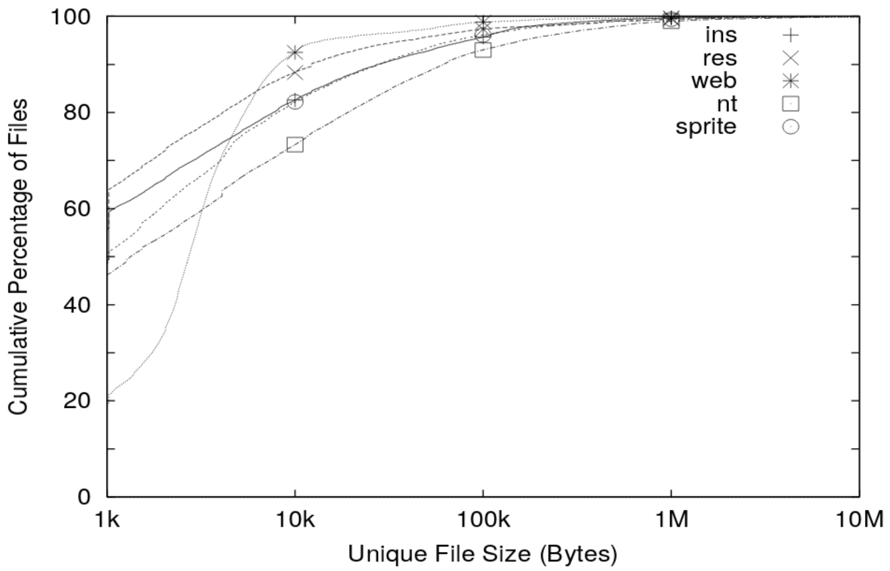
- xv6 finds *first free block/inode*
- result: files/directory entries scattered about

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 14

typical file sizes

most files are small

- sometimes 50+% less than 1kbyte
- often 80-95% 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

- …and requires multiple reads from disk to get

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

- replaces or supplements block list

Linux's ext4 and Windows's NTFS both use this

allocating extents

challenge: finding contiguous sets 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"
- look for big enough chunk of free blocks
- choose smallest among all the candidates

don't find any? okay: use more than one extent

seeking with extents

challenge: finding byte X of the file

with block pointers: can compute index

with extents: need to scan ist?

exericse

filesystem has:

- root directory with 2 subdirectories
- each subdirectory contains 3 512B files, 2 4MB files
- (1MB = 1024KB; 1KB = 1024B)
- 32B directory entries
- 4B block pointers
- 4KB blocks
- inode: 12 direct pointers, 1 indirect pointer, 1 double-indirect, 1 triple-indirect

(a) how many inodes used?

(b) how many blocks (outside of inodes) with 1KB fragments? [minimum w/partial blocks]

(c) how many blocks (outside of inodes) with block pointers replaced by 8B extents (no fragments)? [compute minimum]

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 filesystem in a consistent state?

- do we know what blocks are free?
- do we know what files exist?
- is the data for files actually what was written?

also important topics, but won't spend much time on these:

what data will I lose if storage fails?

- mirroring, erasure coding (e.g. RAID) — using multiple storage devices
- idea: if one storage device fails, other(s) still have data

what data will I lose if I make a mistake?

- filesystem can store *multiple versions*
- "snapshots" of what was previously there

several bad options (1)

suppose we're moving a file from one directory to another on xv6

steps:

A: write new directory entry

B: overwrite (remove) old directory entry

several bad options (1)

suppose we're moving a file from one directory to another on xv6

steps:

A: write new directory entry

B: overwrite (remove) old directory entry

if we do A before B and crash happens after A:

- can have extra pointer of file
- problem: if old directory entry removed later, will get confused and free the file!

several bad options (1)

suppose we're moving a file from one directory to another on xv6

steps:

A: write new directory entry

B: overwrite (remove) old directory entry

if we do A before B and crash happens after A:

- can have extra pointer of file
- problem: if old directory entry removed later, will get confused and free the file!

if we do B before A and crash happens after B:

- the file disappeared entirely!

several bad options (2)

suppose we're creating a new file

A: mark blocks as used in free block map

B: write inode for file

C: write directory entry for file

several bad options (2)

suppose we're creating a new file

A: mark blocks as used in free block map

B: write inode for file

C: write directory entry for file

if we do A before B+C and crash happens after A:

  have blocks we can't use (not free), but which are unused

several bad options (2)

suppose we're creating a new file

A: mark blocks as used in free block map

B: write inode for file

C: write directory entry for file

if we do A before B+C and crash happens after A:
- have blocks we can't use (not free), but which are unused

if we do B before A+C and crash happens after B:
- have inode we can't use (not free), but which is not really used

several bad options (2)

suppose we're creating a new file

A: mark blocks as used in free block map

B: write inode for file

C: write directory entry for file

if we do A before B+C and crash happens after A:
- have blocks we can't use (not free), but which are unused

if we do B before A+C and crash happens after B:
- have inode we can't use (not free), but which is not really used

if we do C before A+B and crash happens after C:
- have directory entry that points to junk — will behave weirdly

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

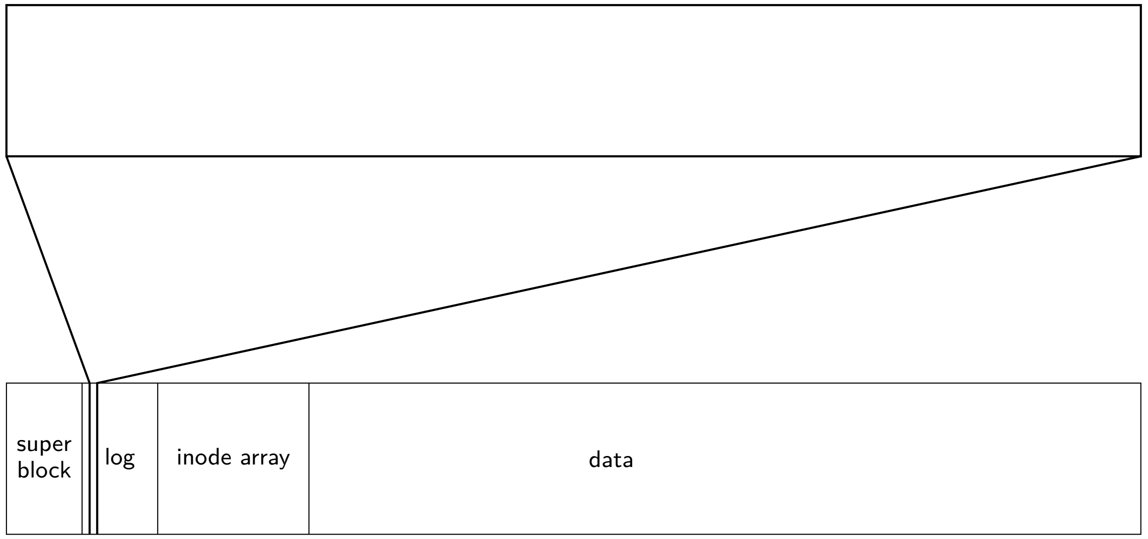
concept: transaction

transaction: bunch of updates that happen all at once

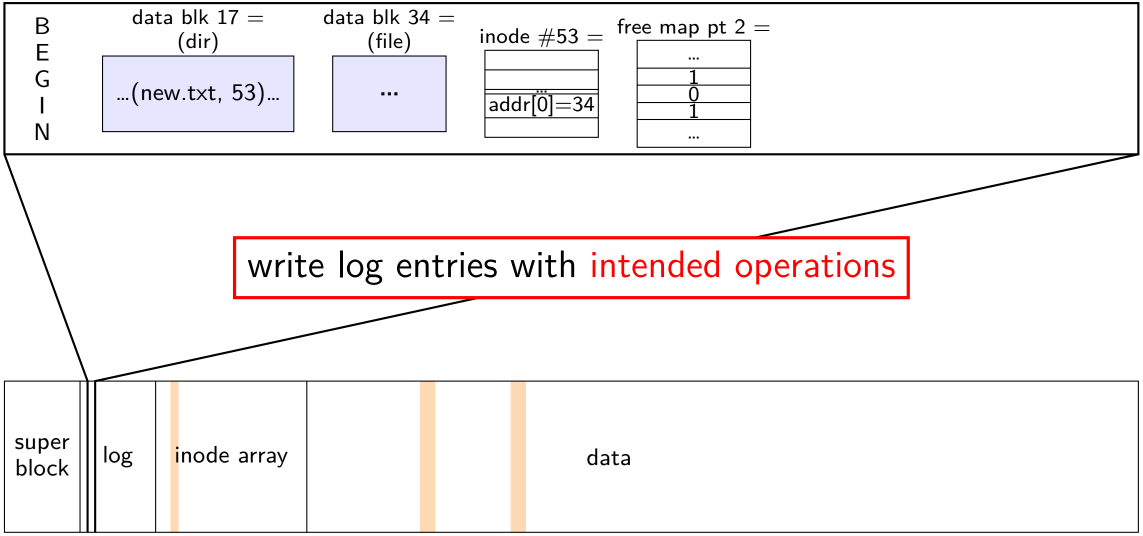
implementation trick: one update means transaction “commits”

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

redo logging: file creation

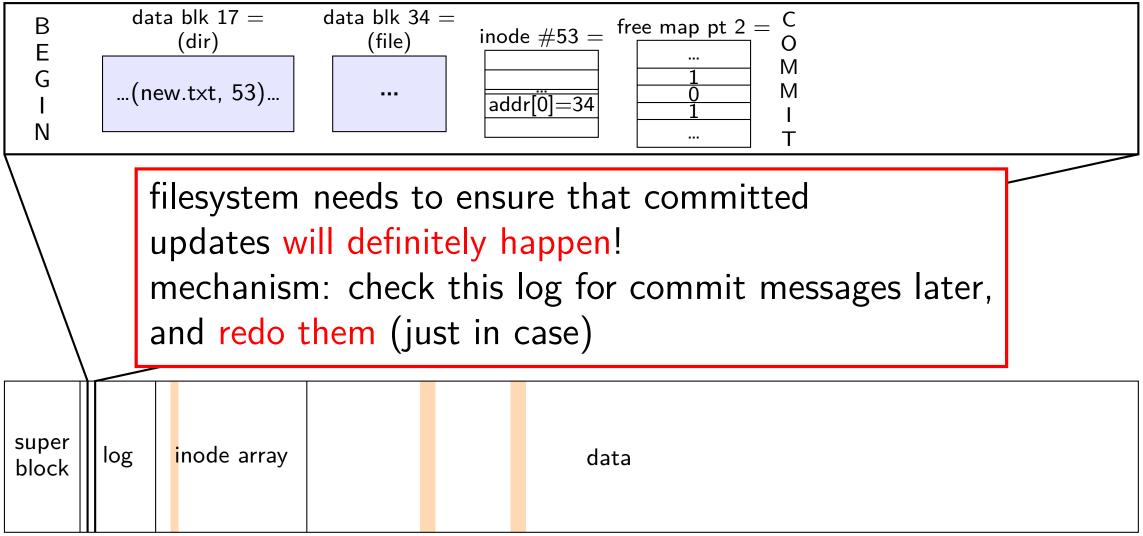


redo logging: file creation



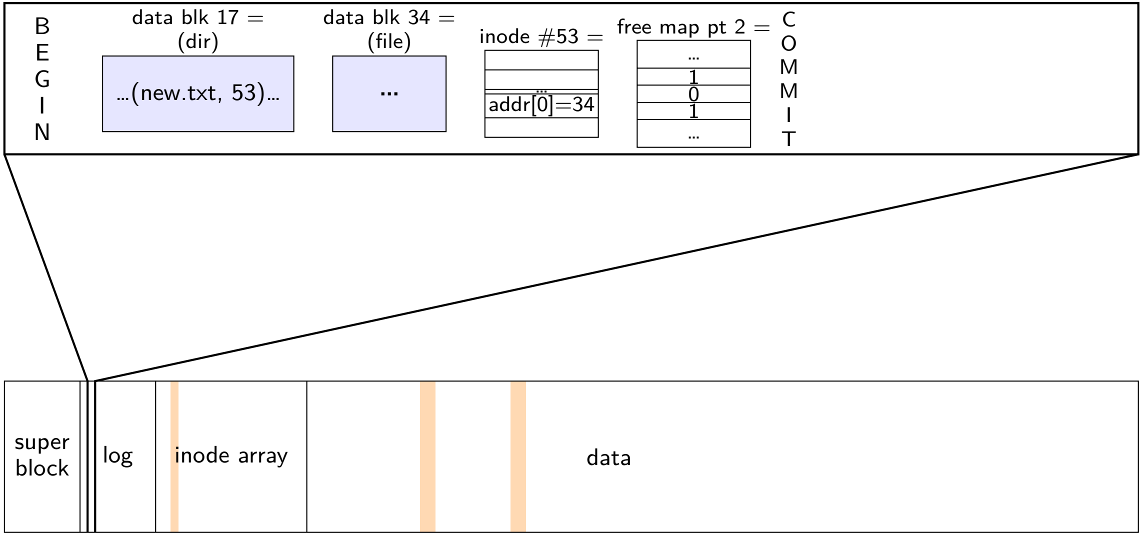
write log entries with intended operations

redo logging: file creation

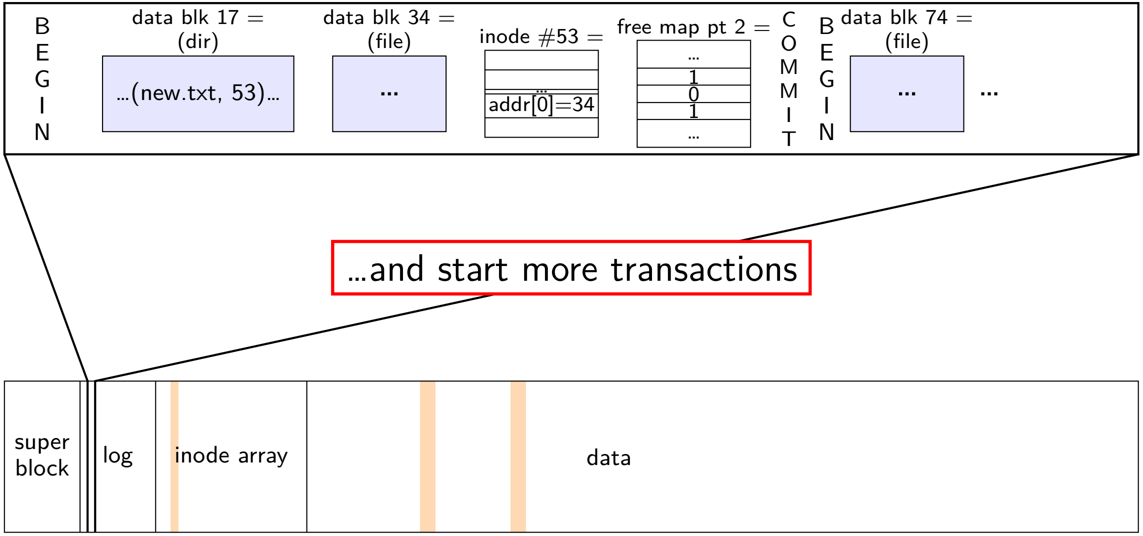


filesystem needs to ensure that committed updates will definitely happen!
mechanism: check this log for commit messages later, and redo them (just in case)

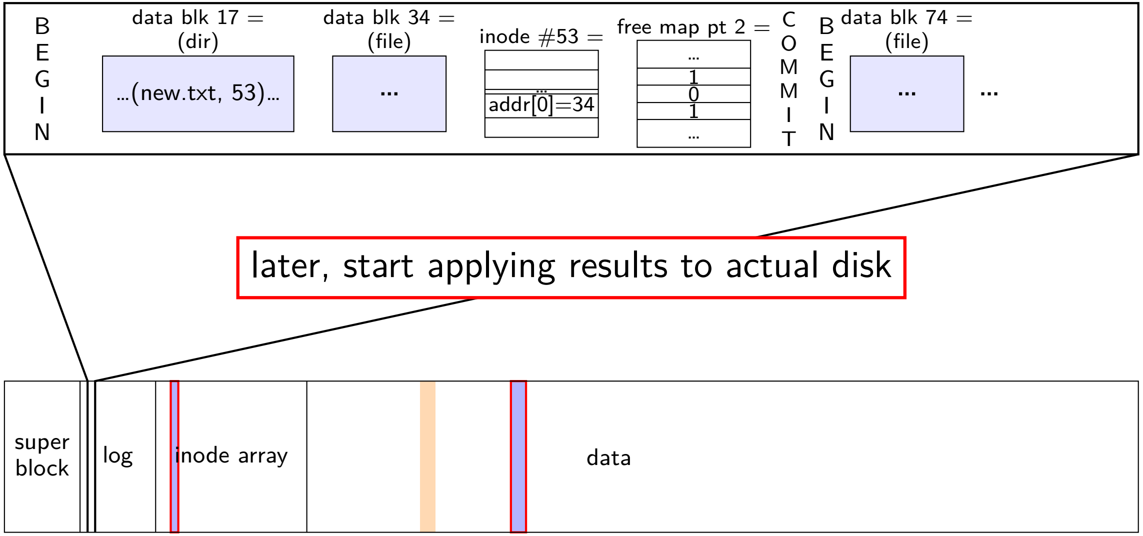
redo logging: file creation



redo logging: file creation

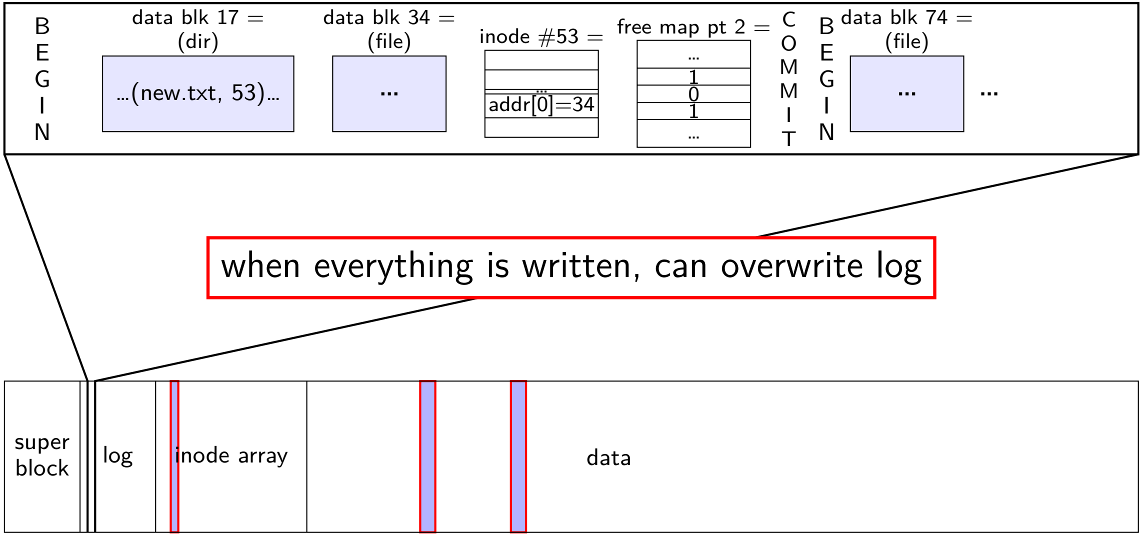


redo logging: file creation



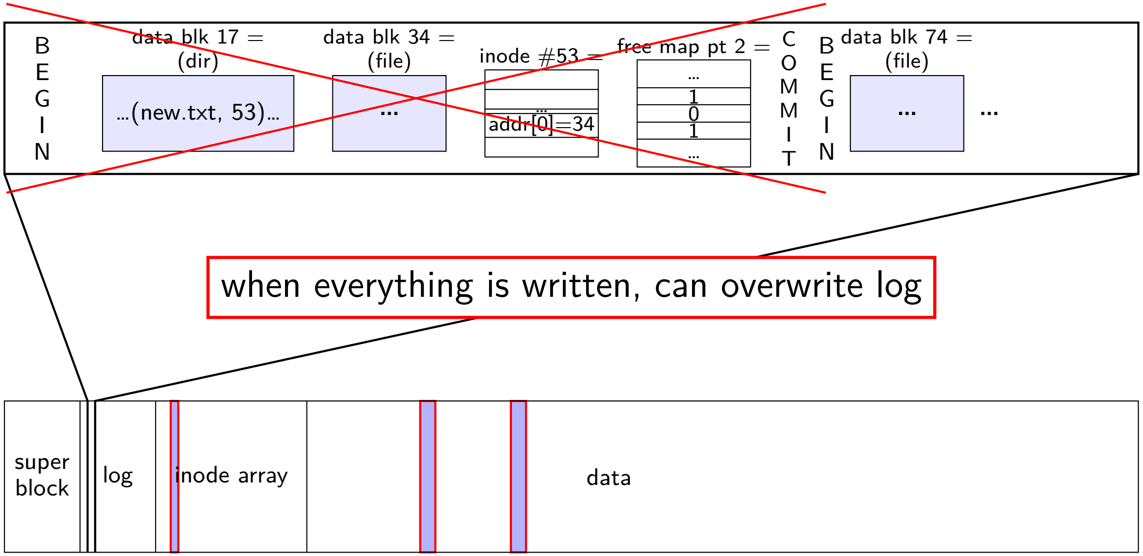
later, start applying results to actual disk

redo logging: file creation



when everything is written, can overwrite log

redo logging: file creation



when everything is written, can overwrite log

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*

backup slides

Linux ext2 inode

```
struct ext2_inode {
    __le16 i_mode;                  /* File mode */
    __le16 i_uid;                   /* Low 16 bits of Owner Uid */
    __le32 i_size;                  /* Size in bytes */
    __le32 i_atime;         /* Access time */
    __le32 i_ctime;         /* Creation time */
    __le32 i_mtime;         /* Modification time */
    __le32 i_dtime;         /* Deletion Time */
    __le16 i_gid;                   /* Low 16 bits of Group Id */
    __le16 i_links_count;           /* Links count */
    __le32 i_blocks;        /* Blocks count */
    __le32 i_flags;         /* File flags */
    ...
    __le32 i_block[EXT2_N_BLOCKS]; /* Pointers to blocks */
    ...
};
```

Linux ext2 inode

```
struct ext2_inode {
    __le16 i_mode;                  /* File mode */
    __le16 i_uid;                   /* Low 16 bits of Owner Uid */
    __le32 i_size;                  /* Size in bytes */
    __le32 i_atime;         /* Access time */
    __le32 i_ctime;         /* Creation time */
    __l
    __l
    __l
    __l
    __le16 i_links_count;           /* Links count */
    __le32 i_blocks;        /* Blocks count */
    __le32 i_flags;         /* File flags */
    ...
    __le32 i_block[EXT2_N_BLOCKS]; /* Pointers to blocks */
    ...
};
```

type (regular, directory, device)
and permissions (read/write/execute for owner/group/others)

Linux ext2 inode

```
struct ext2_inode {
    __le16 i_mode;              /* File mode */
    __le16 i_uid;               /* Low 16 bits
    __le32 i_size;              /* Size in bytes */
    __le32 i_atime;     /* Access time */
    __le32 i_ctime;     /* Creation time */
    __le32 i_mtime;     /* Modification time */
    __le32 i_dtime;     /* Deletion Time */
    __le16 i_gid;               /* Low 16 bits of Group Id */
    __le16 i_links_count;       /* Links count */
    __le32 i_blocks;    /* Blocks count */
    __le32 i_flags;     /* File flags */
    ...
    __le32 i_block[EXT2_N_BLOCKS]; /* Pointers to blocks */
    ...
};
```

owner and group

Linux ext2 inode

```
struct ext2_inode {
    __le16 i_mode;              /* File mod
    __le16 i_uid;               /* Low 16 b
    __le32 i_size;              /* Size in bytes */
    __le32 i_atime;     /* Access time */
    __le32 i_ctime;     /* Creation time */
    __le32 i_mtime;     /* Modification time */
    __le32 i_dtime;     /* Deletion Time */
    __le16 i_gid;               /* Low 16 bits of Group Id */
    __le16 i_links_count;       /* Links count */
    __le32 i_blocks;    /* Blocks count */
    __le32 i_flags;     /* File flags */
    ...
    __le32 i_block[EXT2_N_BLOCKS]; /* Pointers to blocks */
    ...
};
```

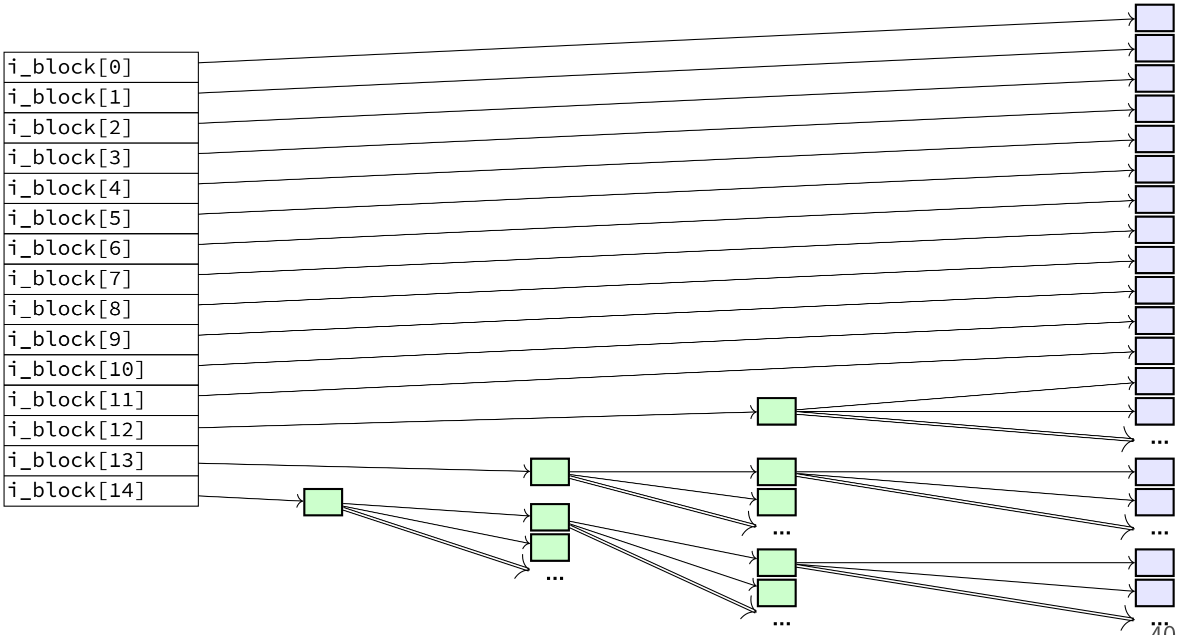
whole bunch of times

Linux ext2 inode

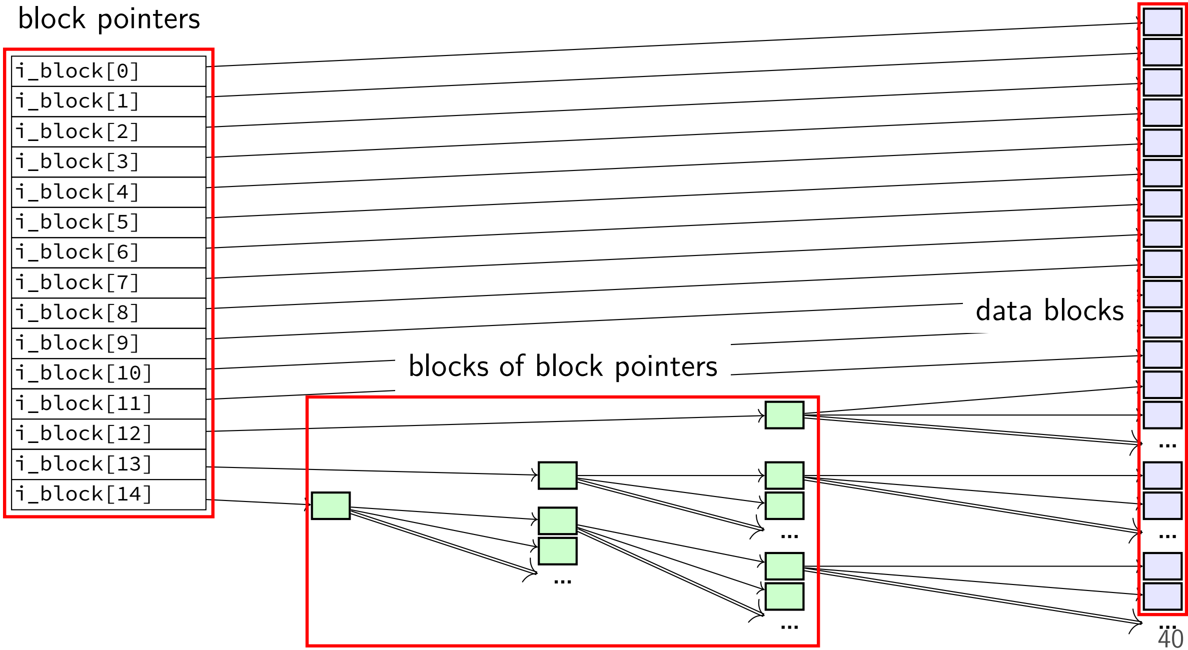
```
struct ext2_inode {
    __le16 i_mod
    __le16 i_uid
    __le32 i_size;                  /* Size in bytes */
    __le32 i_atime;        /* Access time */
    __le32 i_ctime;        /* Creation time */
    __le32 i_mtime;        /* Modification time */
    __le32 i_dtime;        /* Deletion Time */
    __le16 i_gid;                   /* Low 16 bits of Group Id */
    __le16 i_links_count;           /* Links count */
    __le32 i_blocks;       /* Blocks count */
    __le32 i_flags;        /* File flags */
    ...
    __le32 i_block[EXT2_N_BLOCKS]; /* Pointers to blocks */
    ...
};
```

similar pointers like xv6 FS — but more indirection

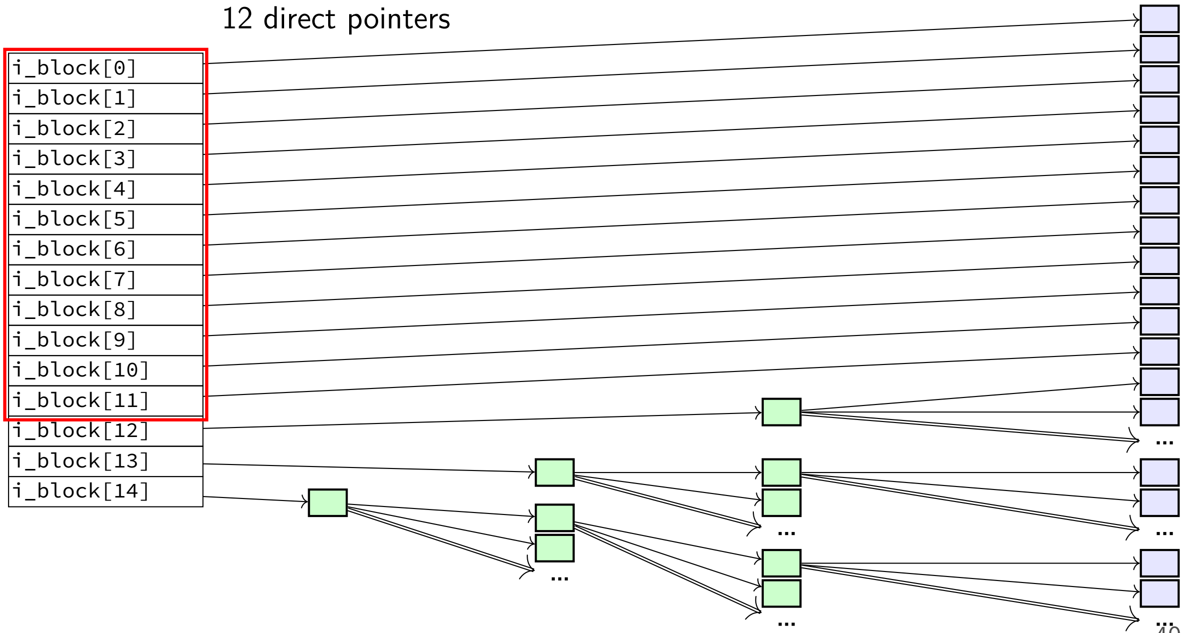
double/triple indirect



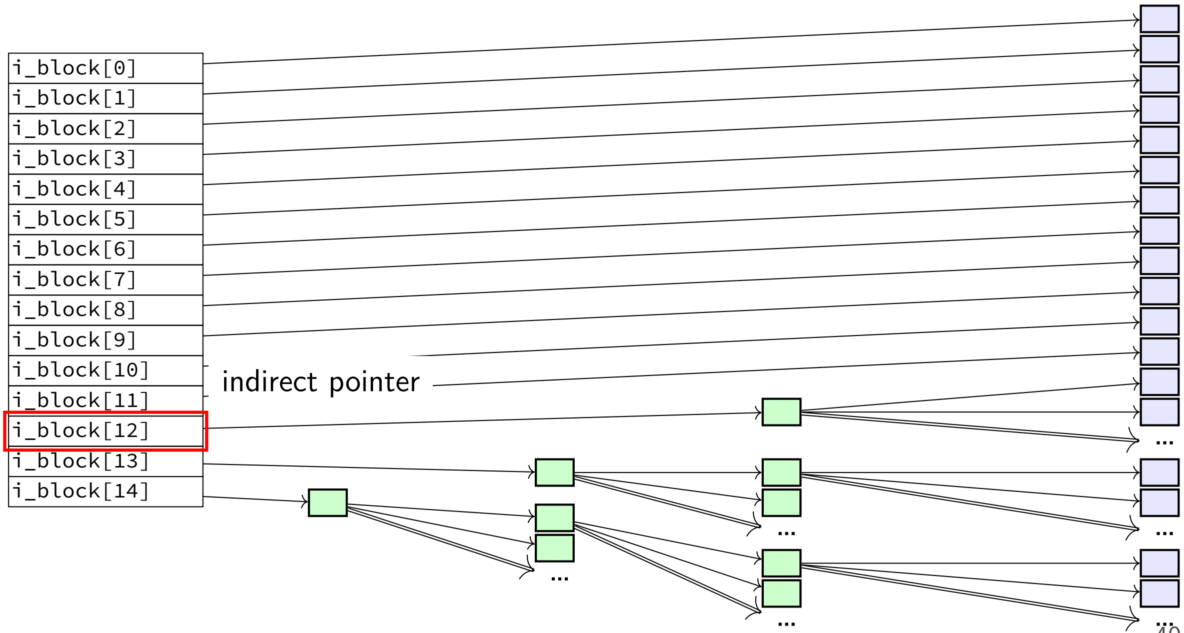
double/triple indirect



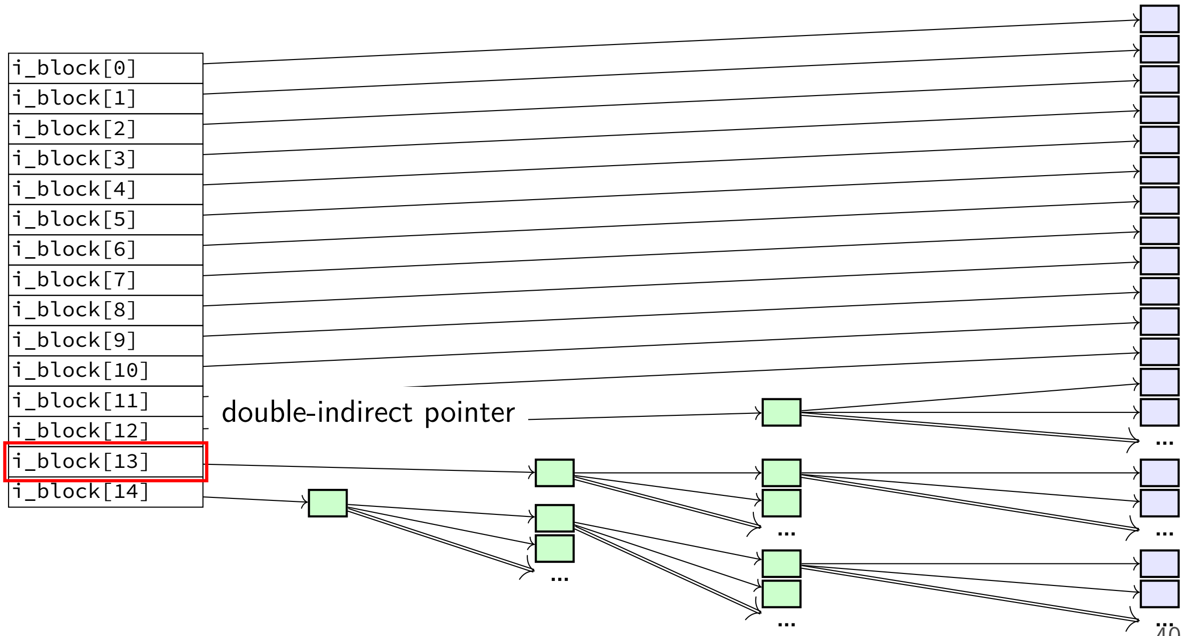
double/triple indirect



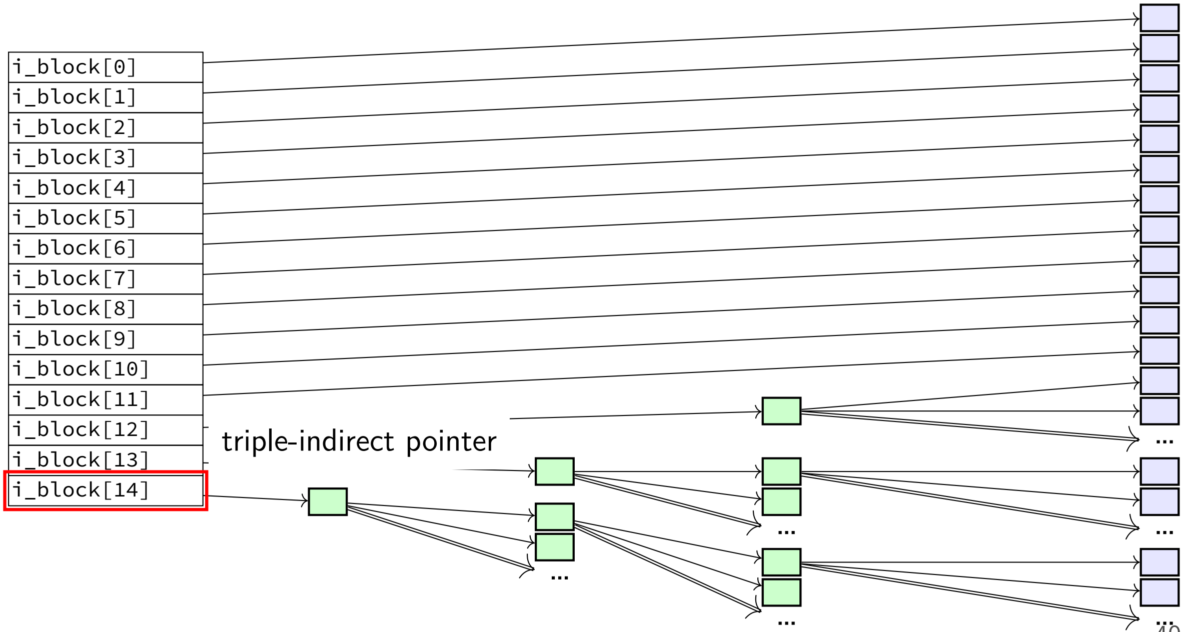
double/triple indirect



double/triple indirect



double/triple indirect



ext2 indirect blocks

12 direct block pointers

1 indirect block pointer

  pointer to block containing more direct block pointers

1 double indirect block pointer

  pointer to block containing more indirect block pointers

1 triple indirect block pointer

  pointer to block containing more double indirect block pointers

ext2 indirect blocks

12 direct block pointers

1 indirect block pointer
- pointer to block containing more direct block pointers

1 double indirect block pointer
- pointer to block containing more indirect block pointers

1 triple indirect block pointer
- pointer to block containing more double indirect block pointers

exercise: if 1K blocks, 4 byte block pointers, how big can a file be?

ext2 indirect blocks (2)

12 direct block pointers

1 indirect block pointer

1 double indirect block pointer

1 triple indirect block pointer

exercise: if 1K (2^{10} byte) blocks, 4 byte block pointers,
how does OS find byte 2^{15} of the file?

(1) using indirect pointer or double-indirect pointer in inode?
(2) what index of block pointer array pointed to by pointer in inode?

exercise

say xv6 filesystem with:

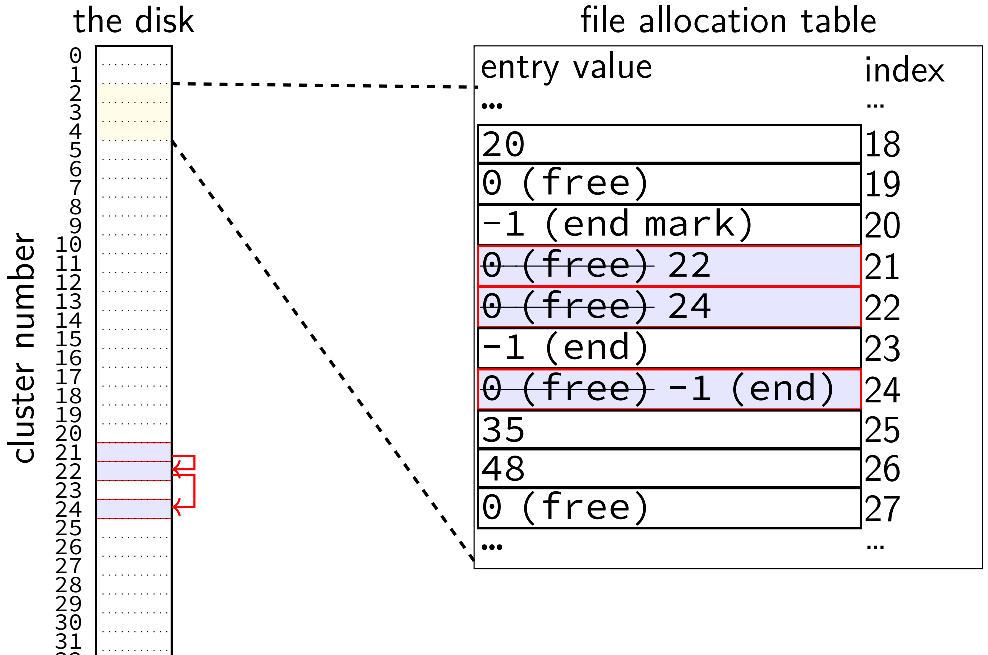
- 64-byte inodes (12 direct $+$ 1 indirect pointer)
- 16-byte directory entries
- 512 byte blocks
- 2-byte block pointers

how many blocks (not storing inodes) is used to store a directory of 200 30464B ($29 \cdot 1024 + 256$ byte) files?

- remember: blocks could include blocks storing data or block pointers or directory enties

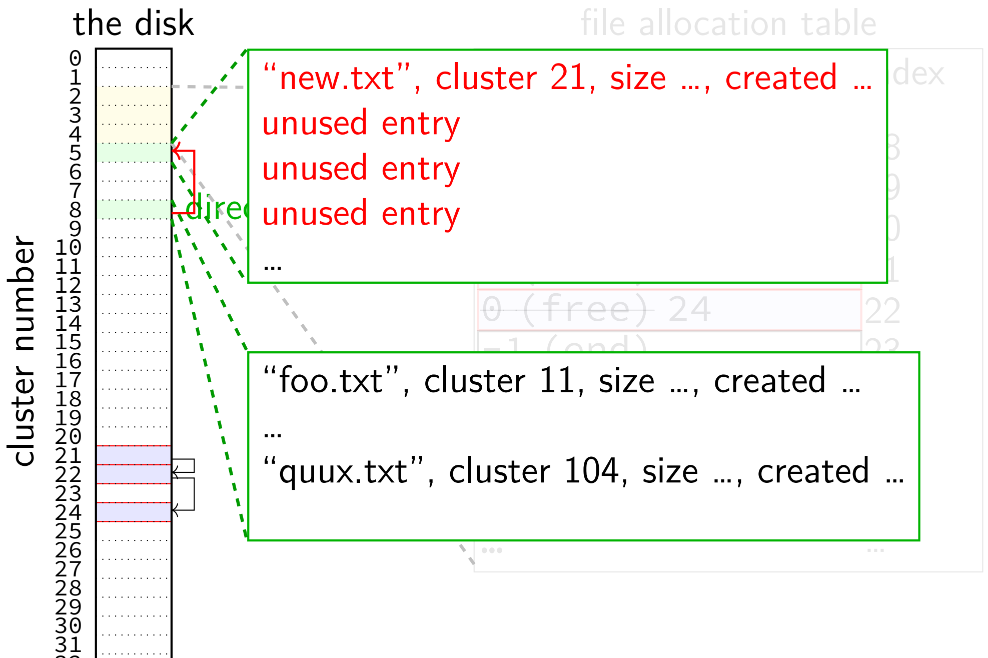
how many blocks is used to store a directory of 2000 3KB files?

recall: FAT: file creation (1)



entry value	index
…	…
20	18
0 (free)	19
-1 (end mark)	20
~~0 (free)~~ 22	21
~~0 (free)~~ 24	22
-1 (end)	23
~~0 (free)~~ -1 (end)	24
35	25
48	26
0 (free)	27
…	…

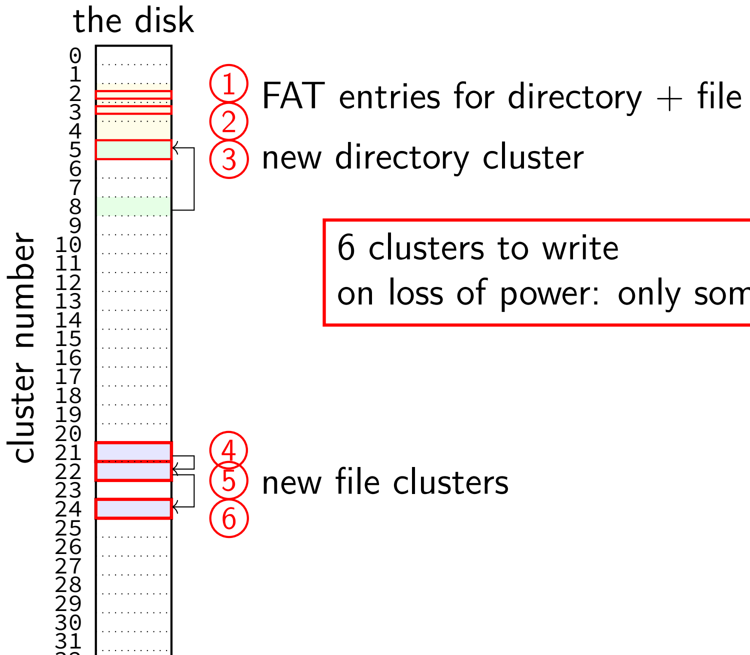
recall: FAT: file creation (2)



"new.txt", cluster 21, size ..., created ...
unused entry
unused entry
unused entry
...

"foo.txt", cluster 11, size ..., created ...
...
"quux.txt", cluster 104, size ..., created ...

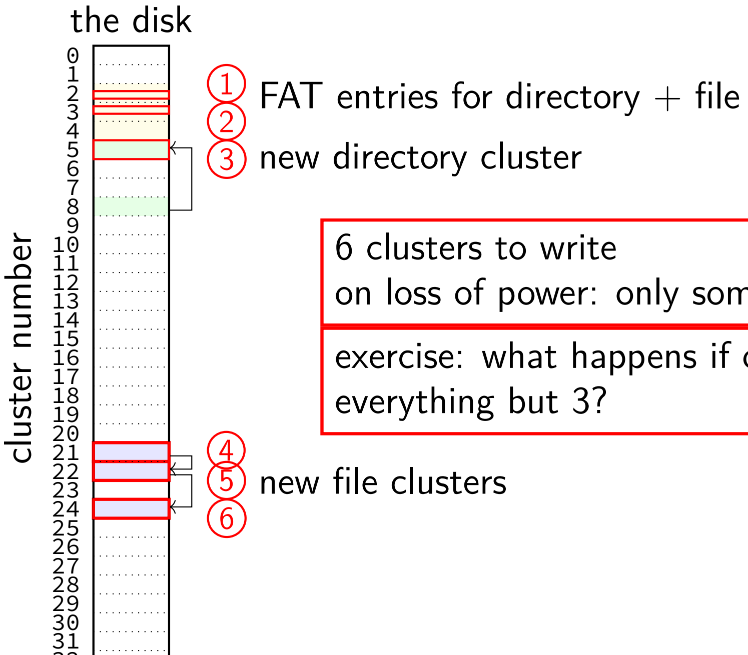
exercise: FAT file creation



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

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

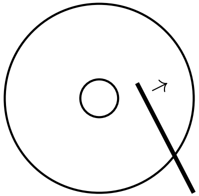
on boot: if clean bit clear, run fsck first

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

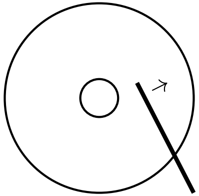


ordering and disk performance

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 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

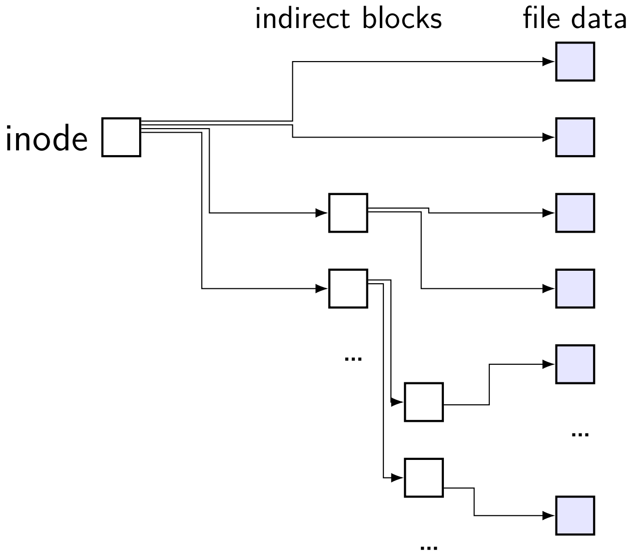
can access *snapshot* of files at prior time

mechanism: copy-on-write

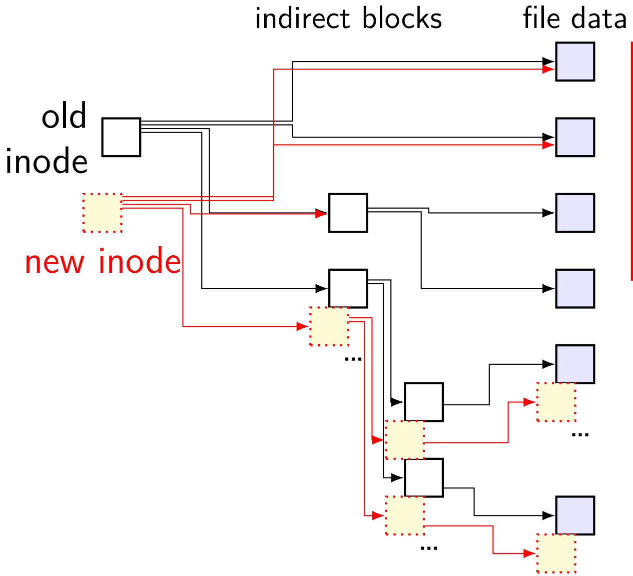
changing file makes new copy of filesystem

common parts shared between versions

inode and copy-on-write



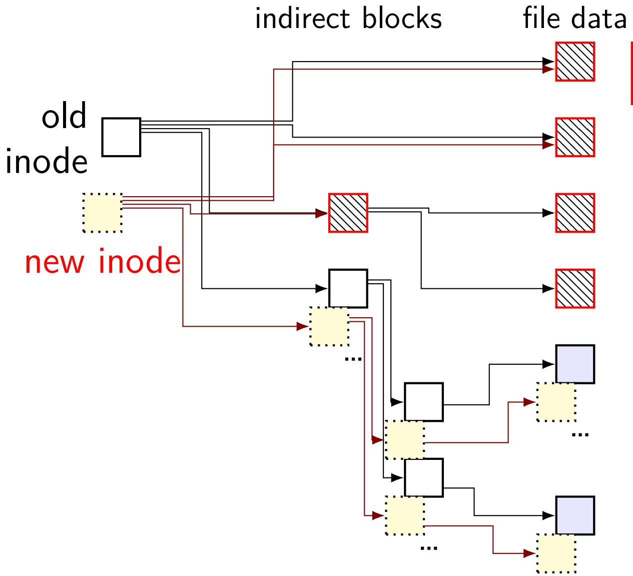
inode and copy-on-write



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

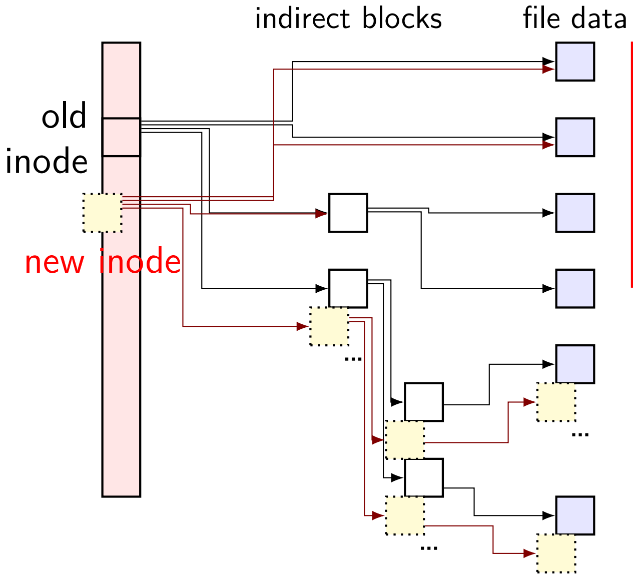
both old+new inode valid

inode and copy-on-write



unchanged parts of file shared

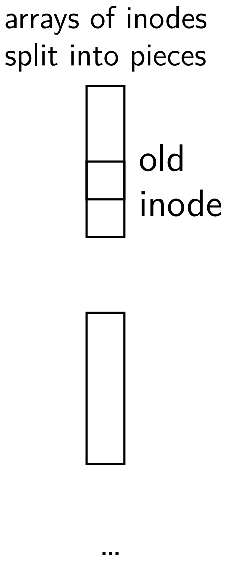
inode and copy-on-write



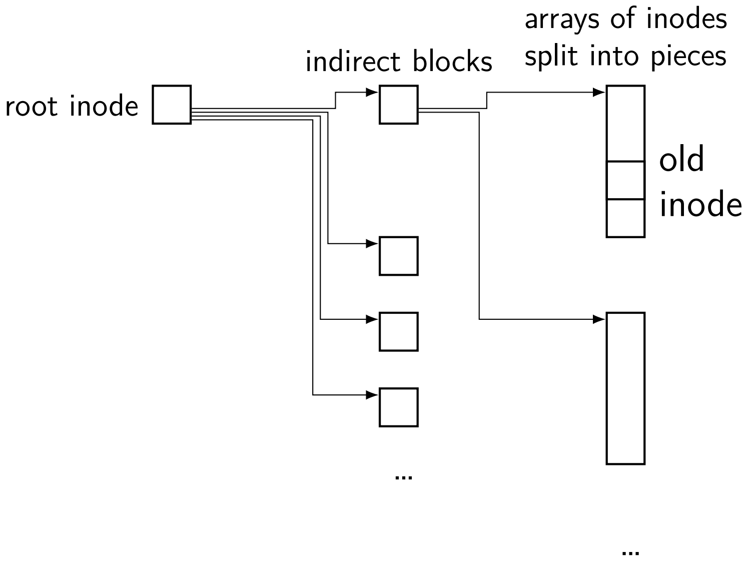
challenge: FFS/xv6/ext2 design has big array of inodes

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

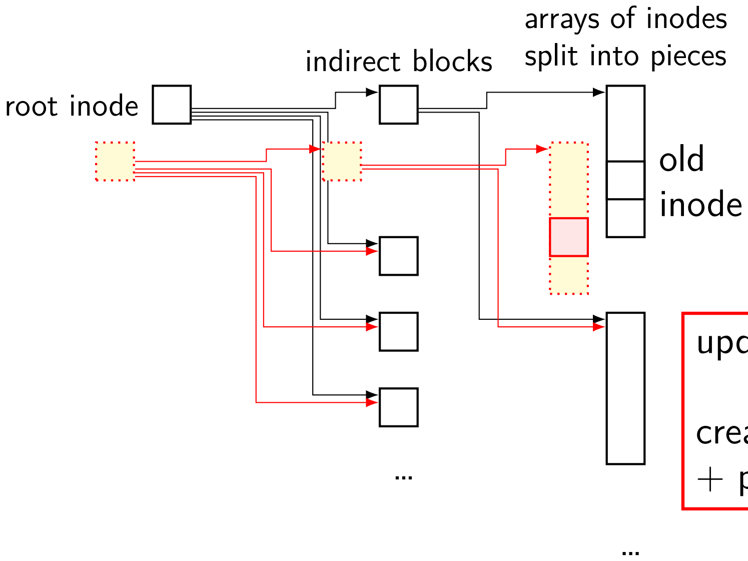
extra indirection for inode array



extra indirection for inode array



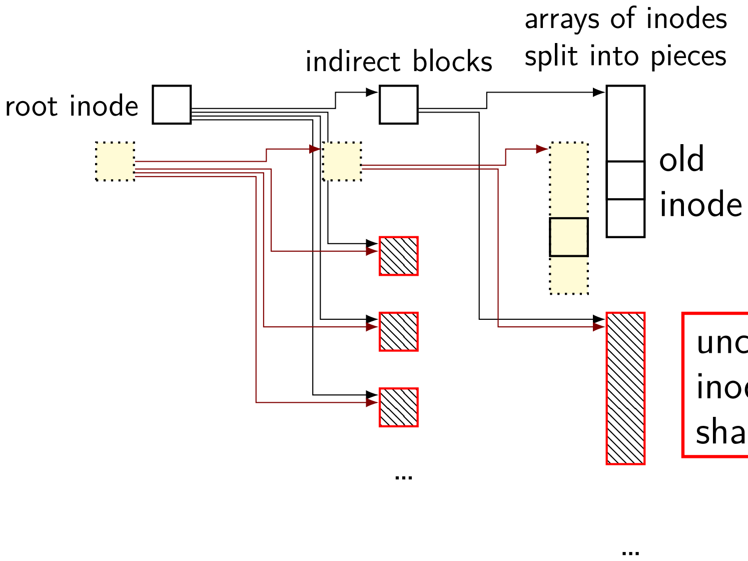
extra indirection for inode array



update one inode?

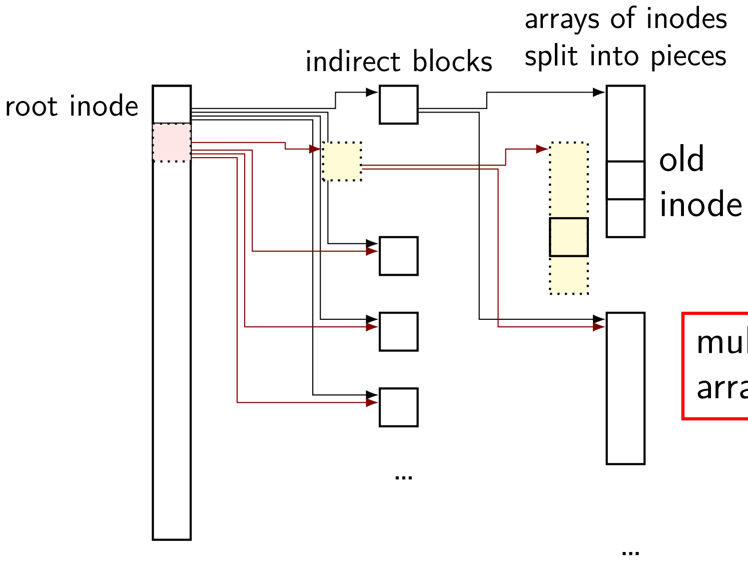
create new root inode
+ pointers

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

multiple copies

FAT: multiple copies of file allocation table and header

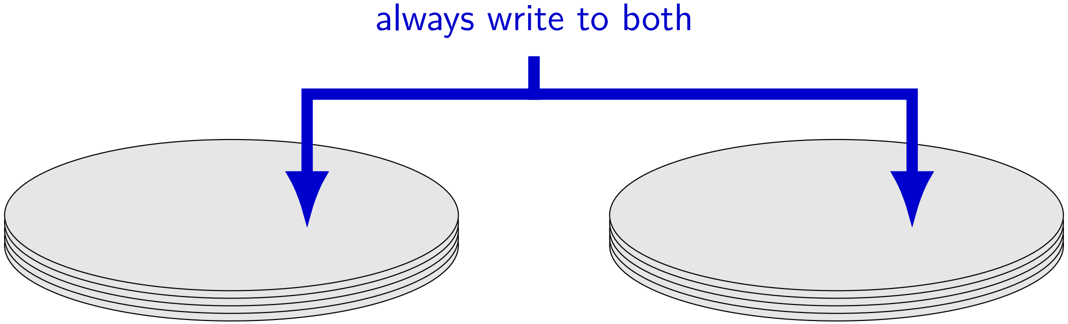
in inode-based filesystems: often multiple copies of superblocks

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

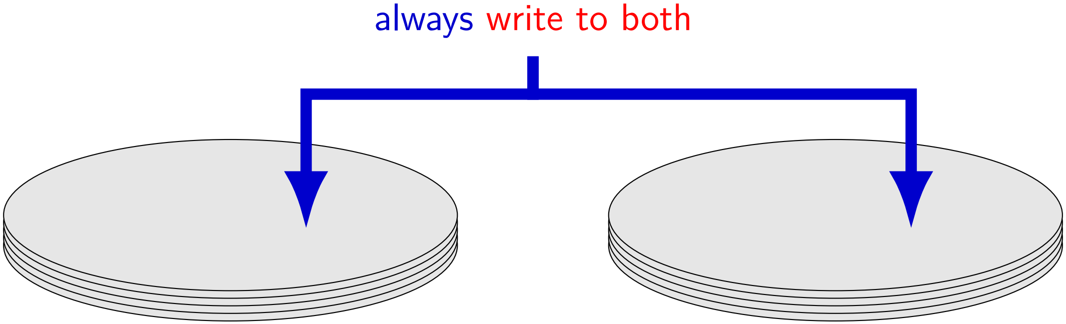
mirroring whole disks

alternate strategy: write everything to two disks



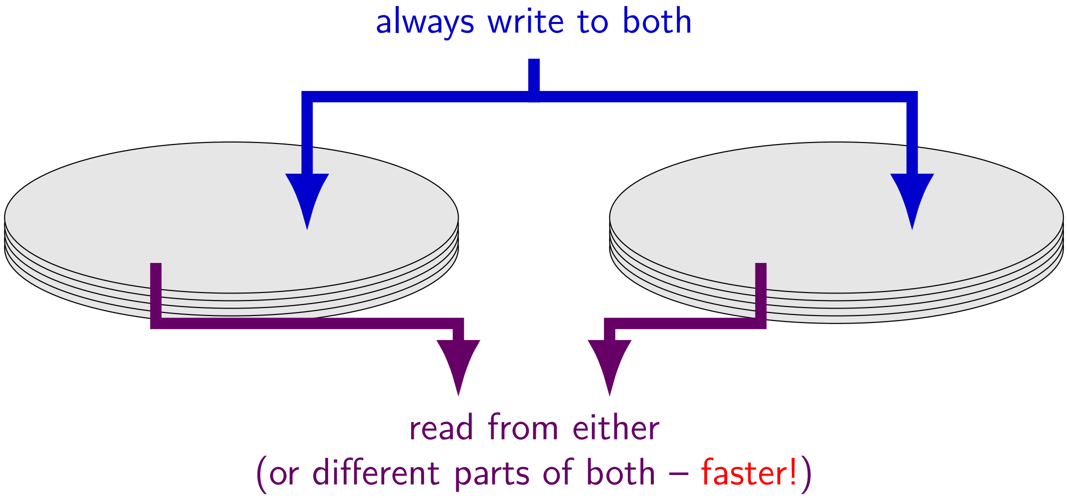
mirroring whole disks

alternate strategy: write everything to two disks



mirroring whole disks

alternate strategy: write everything to two disks



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