last time

FAT file system

- beginning of disk: 'header': sizes, location of FAT, data clusters
- linked lists of data clusters
  - next pointers in FAT (near beginning of disk)
- directory entries: file info incl. starting data cluster

inode-based filesystems

- header (called *superblock*): location/size of inode array, free block map, data blocks
- inodes (in inode array):
  - file type, size, other metadata
  - block pointers (some direct, then less direct for larger files)
- directory entries: name + inode number (index in inode array)
- indirect pointer: points to block of more pointers to data blocks
- double-indirect: pointers to blocks of indirect pointers

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 */
    ...
    __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 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 */
    ...
};
```

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 */
    ...
};
```

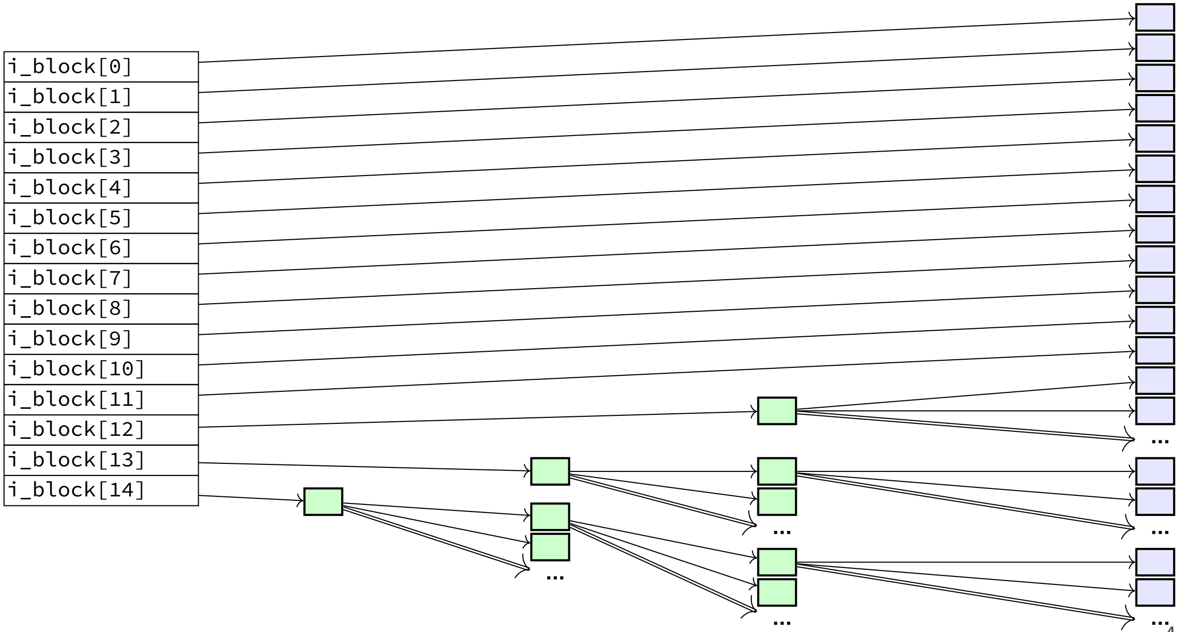
whole bunch of times

Linux ext2 inode

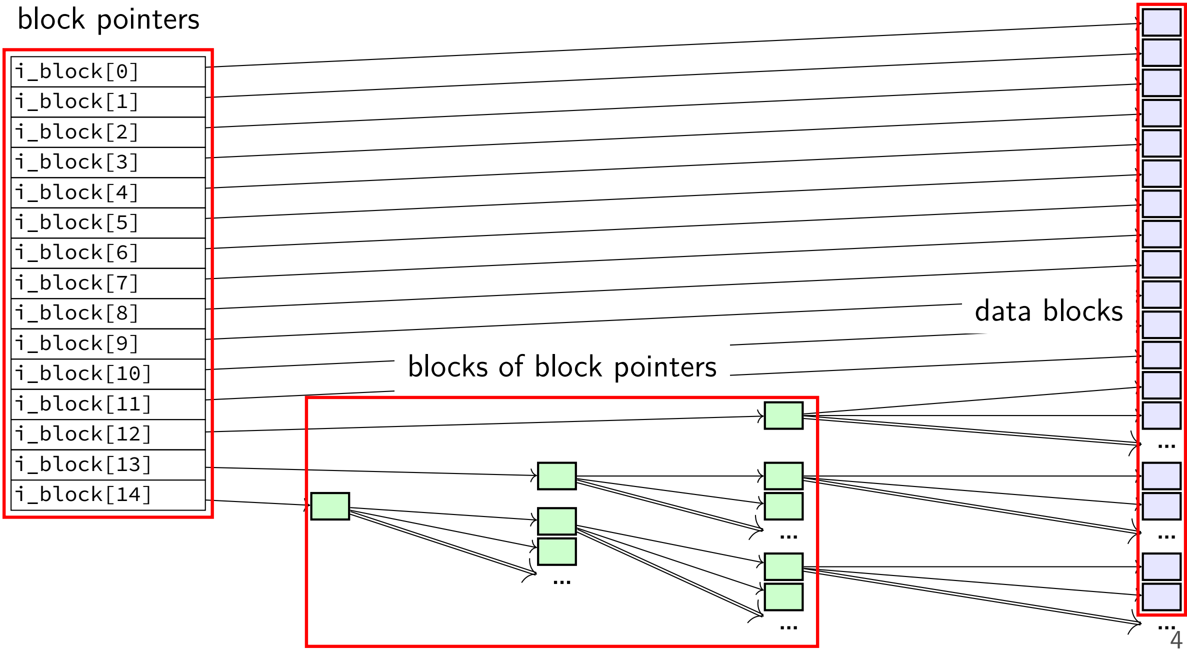
```
struct ext2_inode {
    __le16 i_mod
    __le16 i_uid
    __le32 i_size;                /* Size in bytes */
    __le32 i_atime;      /* Access time */
    __le32 i_ctime;      /* Creation time */
    __le32 i_mtime;      /* Modification time */
    __le32 i_dtime;      /* Deletion Time */
    __le16 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

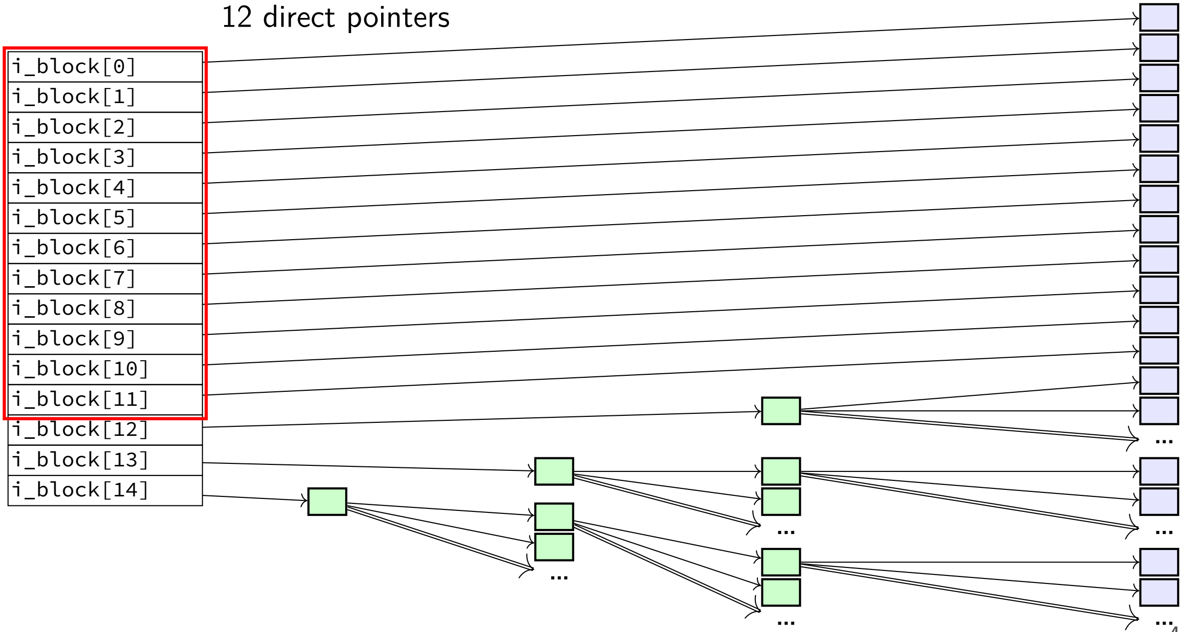
double/triple indirect



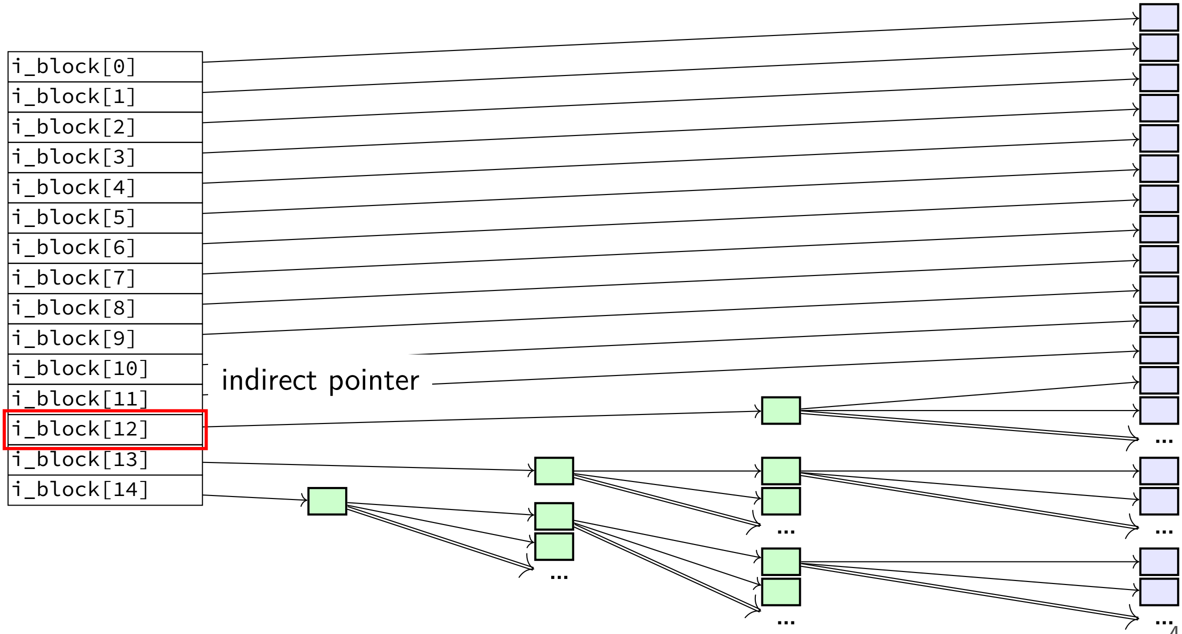
double/triple indirect



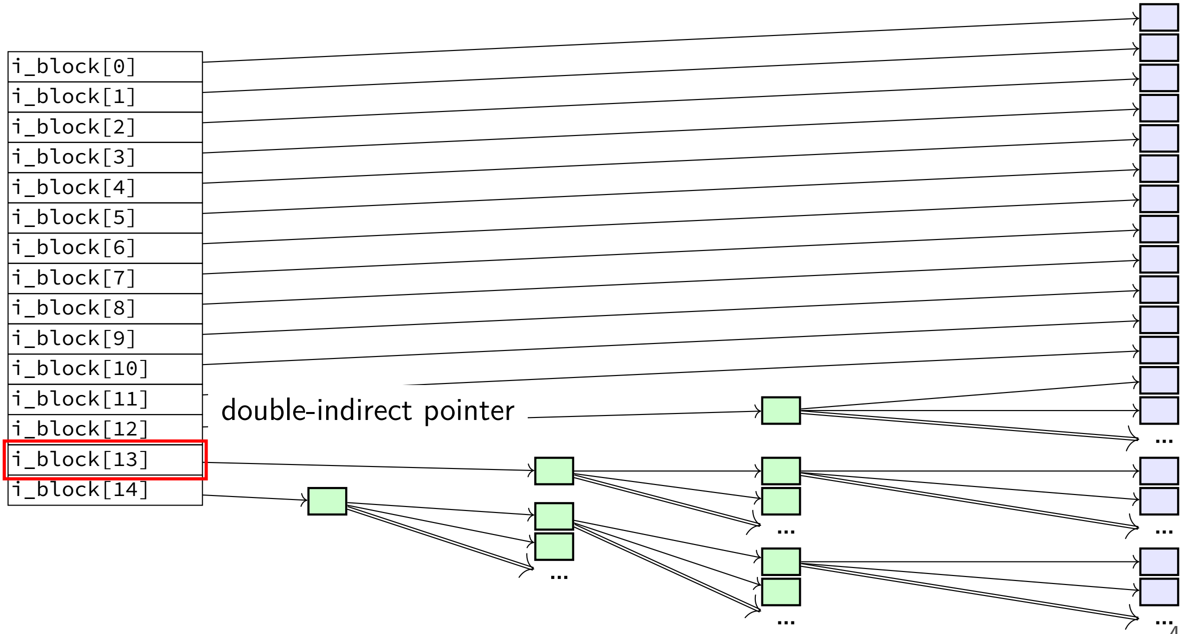
double/triple indirect



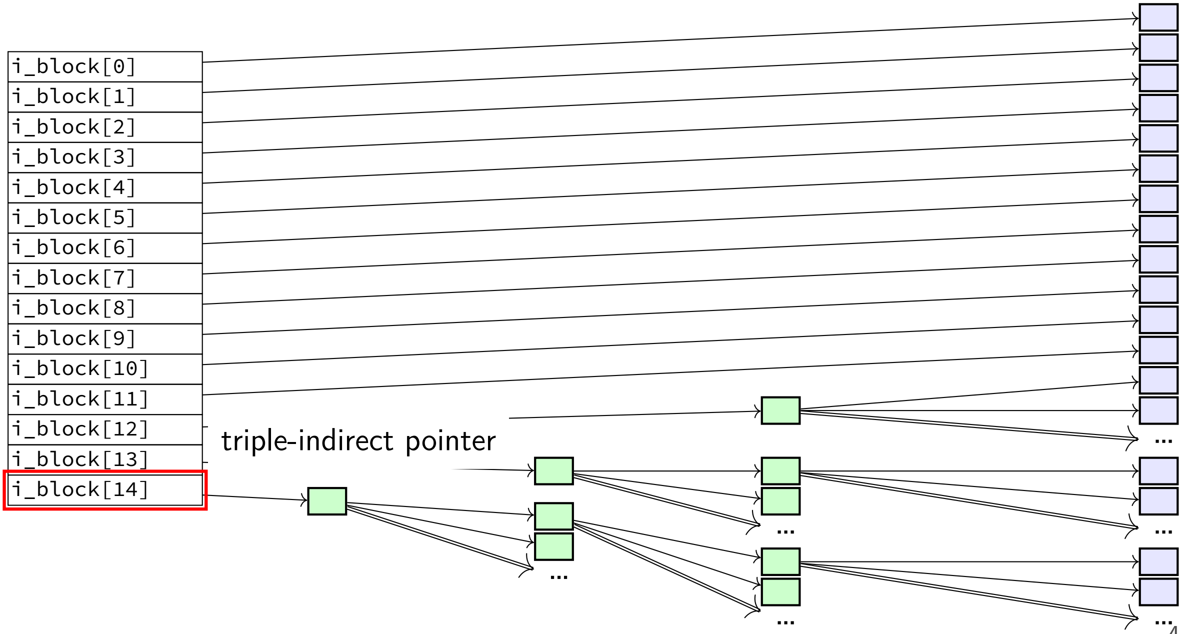
double/triple indirect



double/triple indirect



double/triple indirect



ext2 indirect blocks (1)

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 (1)

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 (solution)

12 direct pointers: first 1K (block size) $\times$ 12 bytes of data

1 indirect pointer:

- points to block with 1K (block size)/4 byte (pointer size) $=$ 256 pointers
- 256 pointers point to 1K blocks
- next 256KB of data

1 double indirect pointer

- points to block with 1K (block size)/4 byte (pointer size) $=$ 256 pointers
- 256 pointers point to pointers that each are like an indirect pointer
- 256KB per indirect pointer $\rightarrow$ next $256 \cdot 256$ KB of data

1 triple indiret

- next $256 \cdot 256 \cdot 256$ KB of data

total size: $12 + 256 + 256^2 + 256^3$ KB $=$ 16843020 KB $\approx$ 16GB

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?

ext2 indirect blocks (2) (solution)

byte $2^{15} = 32$KB into file

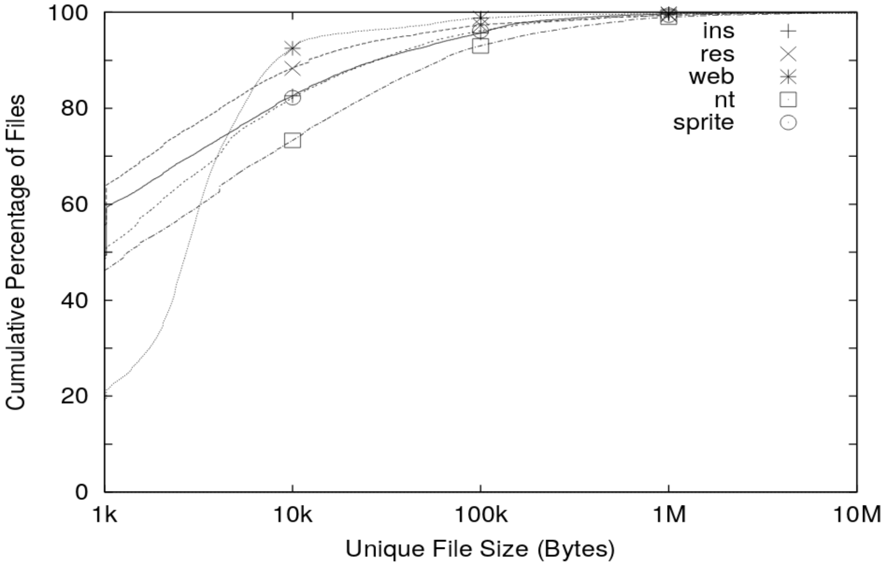
12 direct pointers: first 1K (block size) $\times$ 12 bytes of data

1 indirect pointer:

- points to block with 1K (block size)/4 byte (pointer size) $=$ 256 pointers
- 256 pointers point to 1K blocks
- next 256KB of data

going to be (32 - 12)th element

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 80-95% less than 10kbyte

doens't mean large files are unimportant

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

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

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 list?

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/SSD stores checksum for each sector

write interrupted? — checksum mismatch

- hard drive/SSD 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!

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

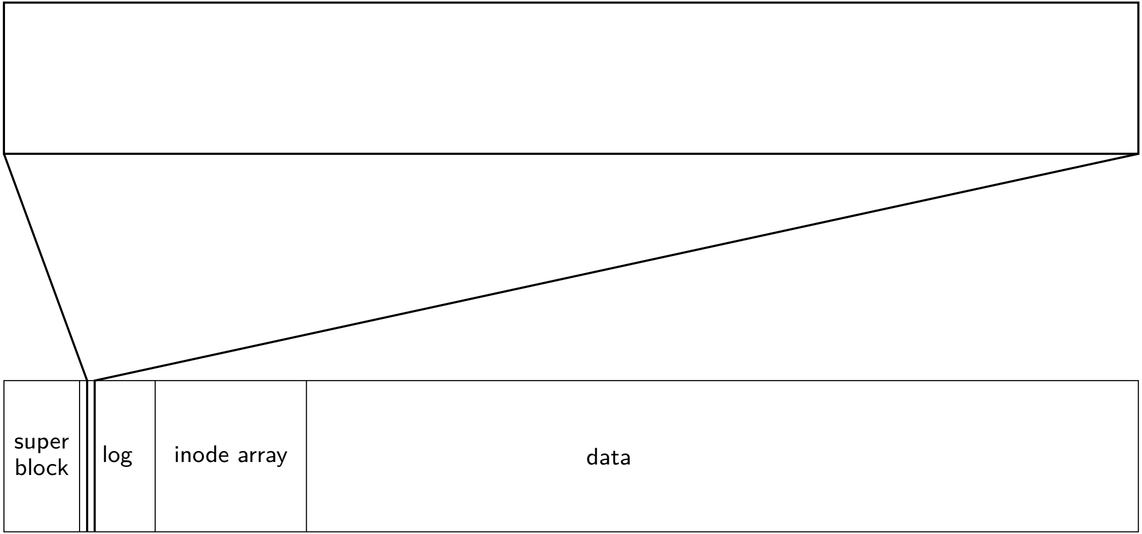
concept: transaction

transaction: bunch of updates that happen all at once

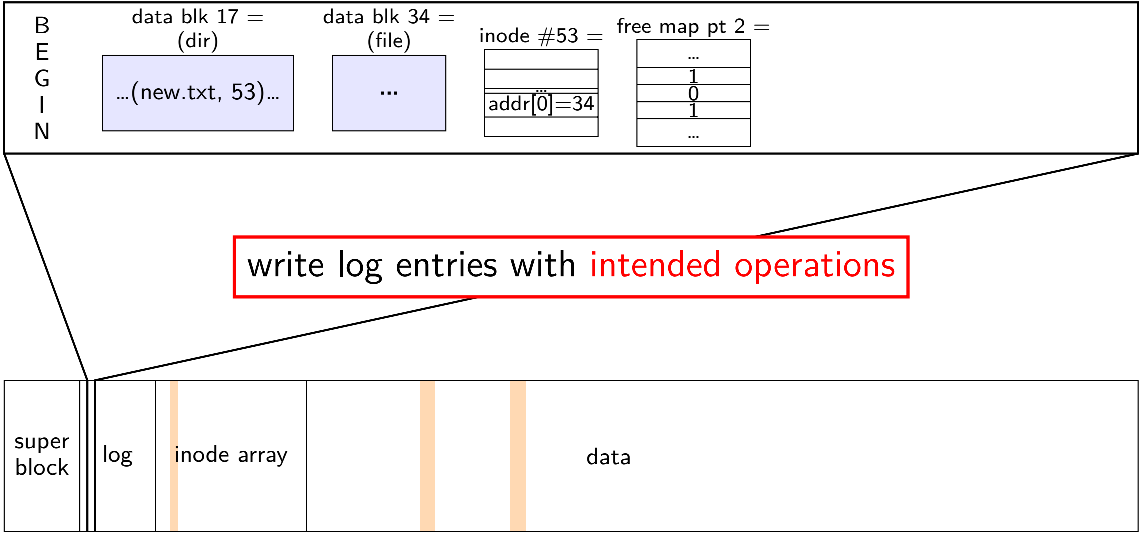
implementation trick: one update means transaction "commits"

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

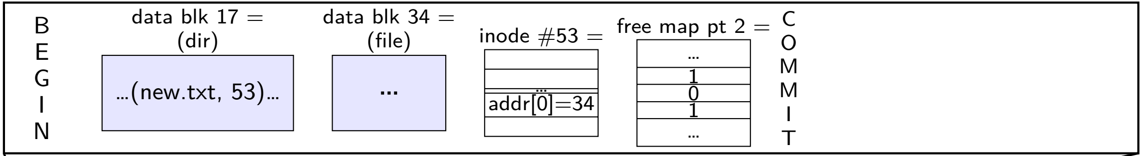
redo logging: file creation



redo logging: file creation

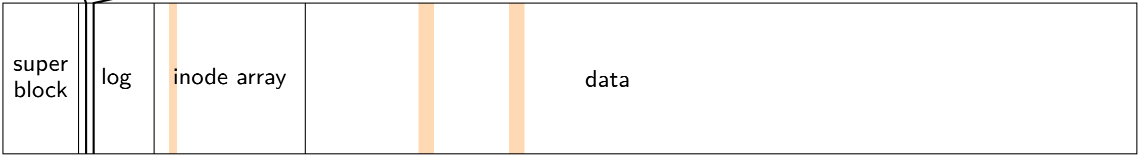


redo logging: file creation

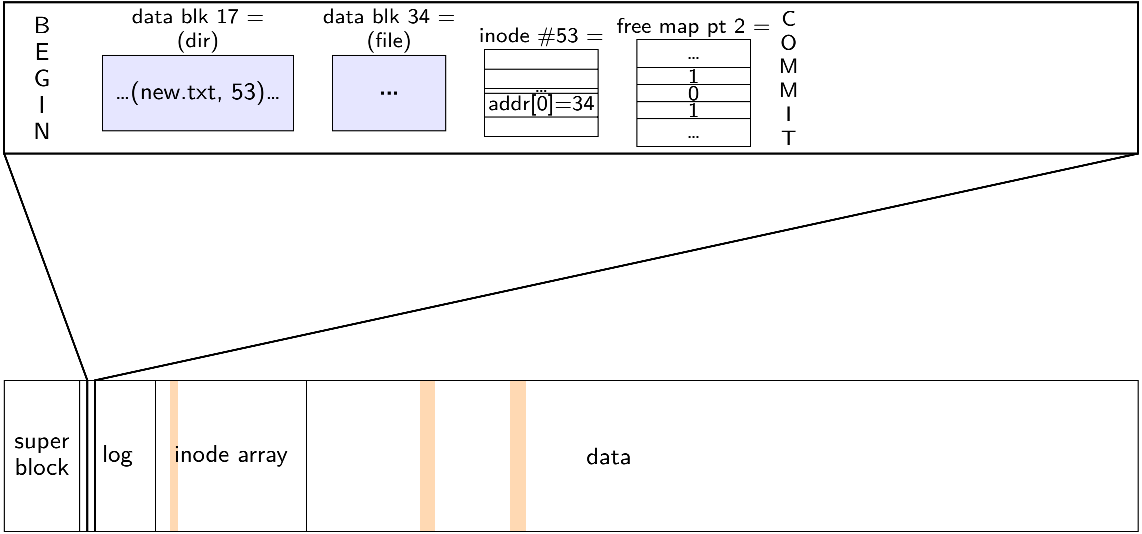


filesystem needs to ensure that committed updates will definitely happen!

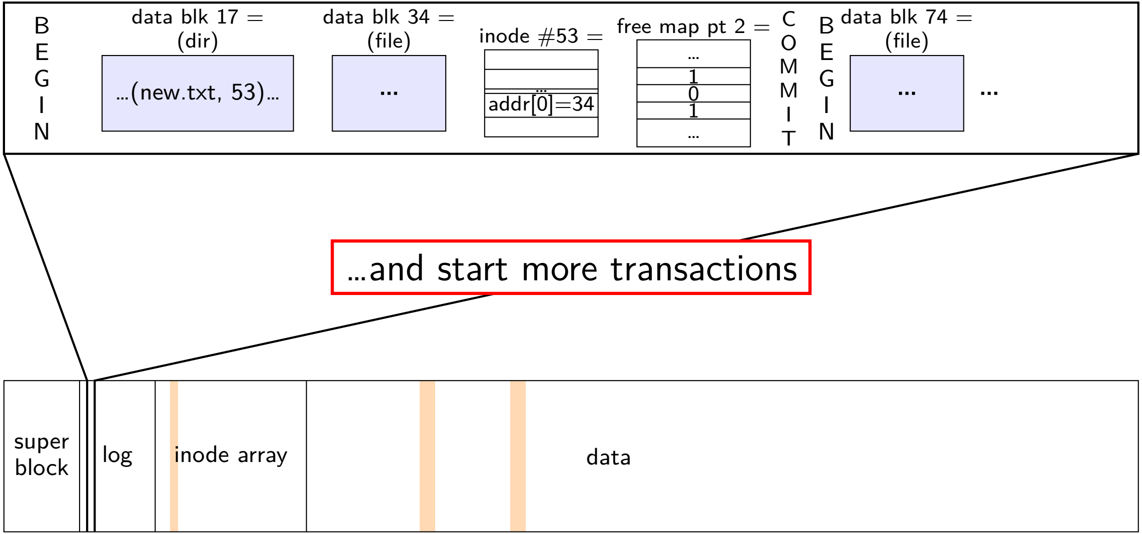
mechanism: check this log for commit messages later, and redo them (just in case)



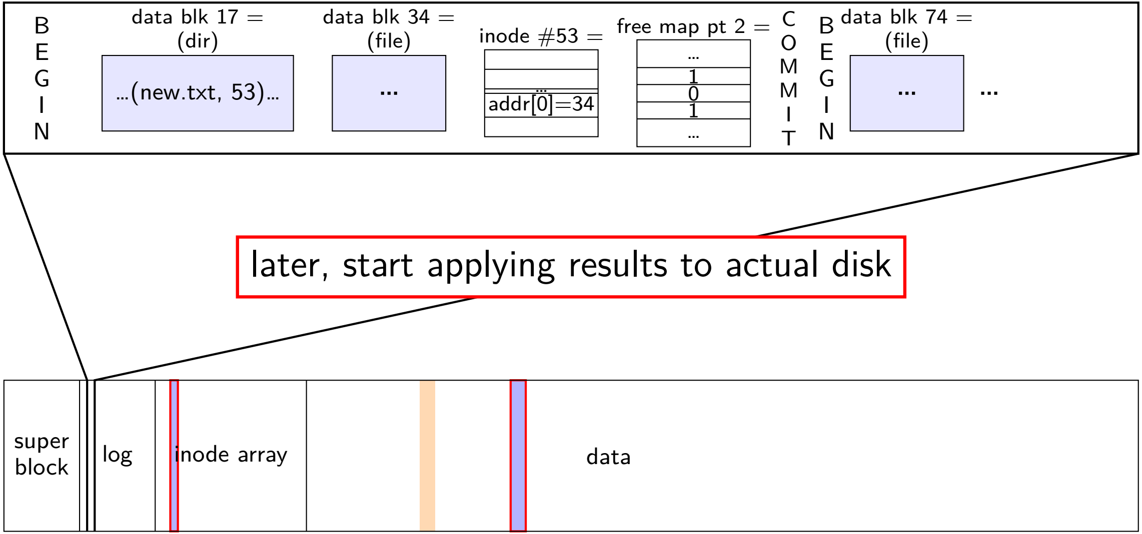
redo logging: file creation



redo logging: file creation

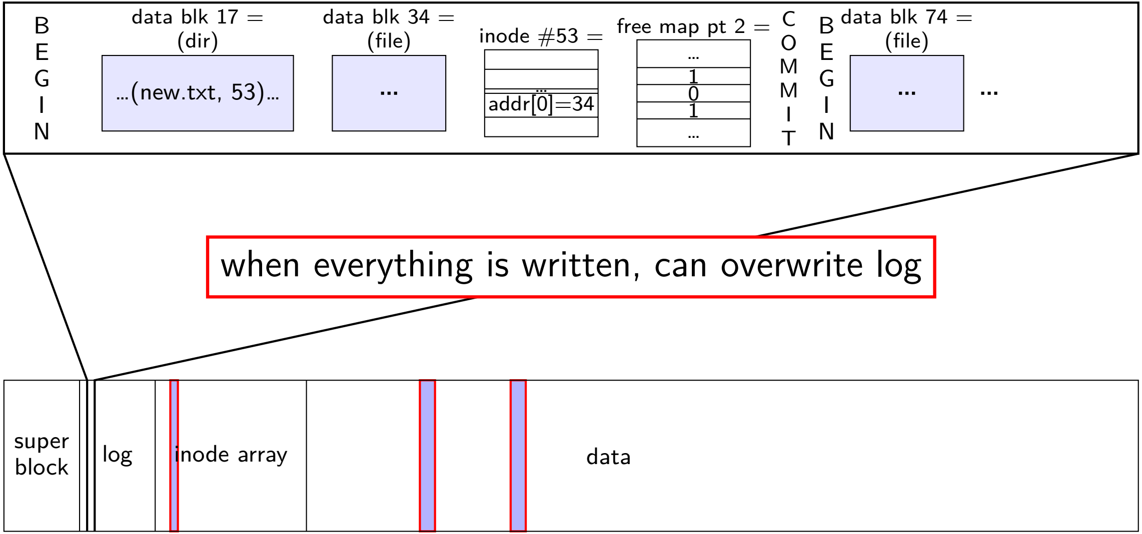


redo logging: file creation

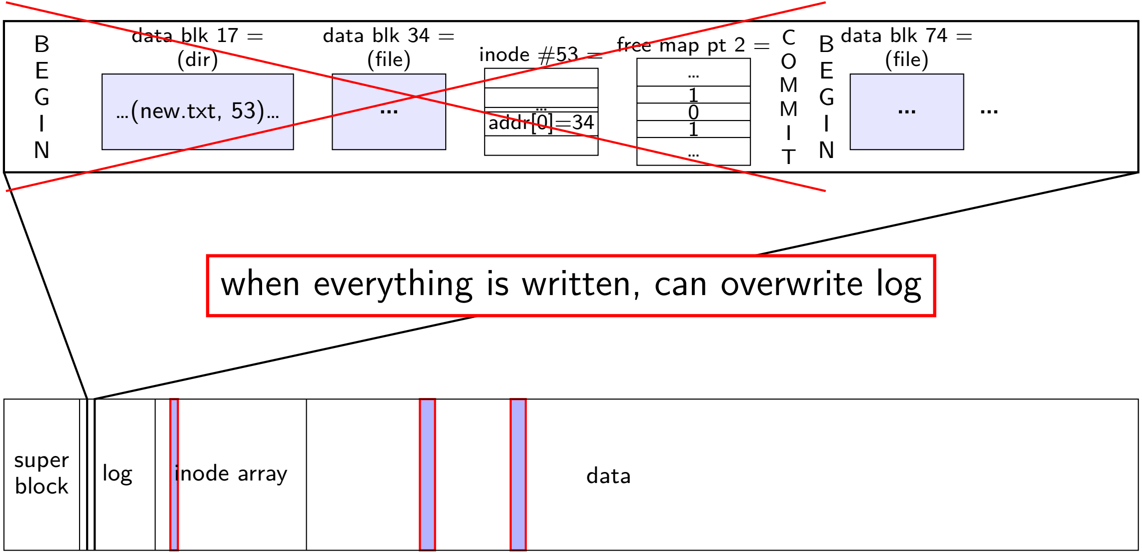


later, start applying results to actual disk

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*

exercise (1)

suppose OS performing operation of appending 100KB to a 100KB file X in directory Y and uses redo logging, ext2-like filesystem with 1KB blocks, 4B block pointers

part 1: what's modified?

[A] free block map
[B] data blocks for file
[C] indirect blocks for file
[D] data blocks for directory
[E] inode for file
[F] inode for directory
[G] the log

exercise (2)

suppose OS performing operation of appending 100KB to a 100KB file X in directory Y and uses redo logging

part 2: crash happens after writing:

- log entries for entire operation
- free block map changes
- indirect blocks for file

...what is written after restart as part of this operation?

- [A] free block map
- [B] data blocks for file
- [C] indirect blocks for file
- [D] data blocks for directory
- [E] inode for file
- [F] inode for directory
- [G] the log

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

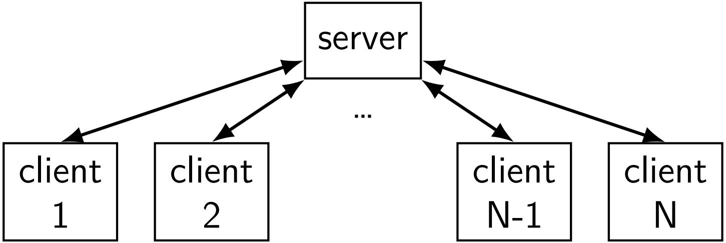
metadata+user data: integrity of user data guaranteed

distributed systems

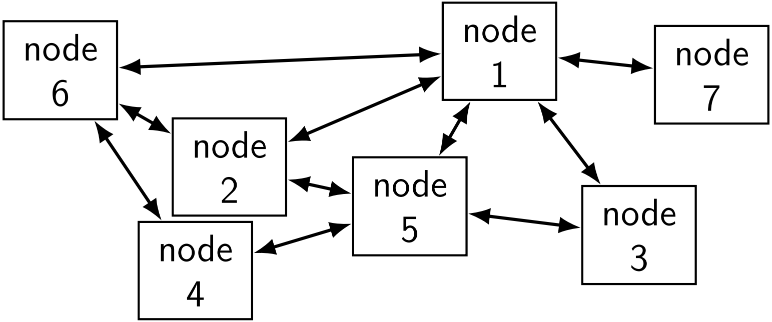
multiple machines working together to perform a single task

called a *distributed system*

some distibuted systems models

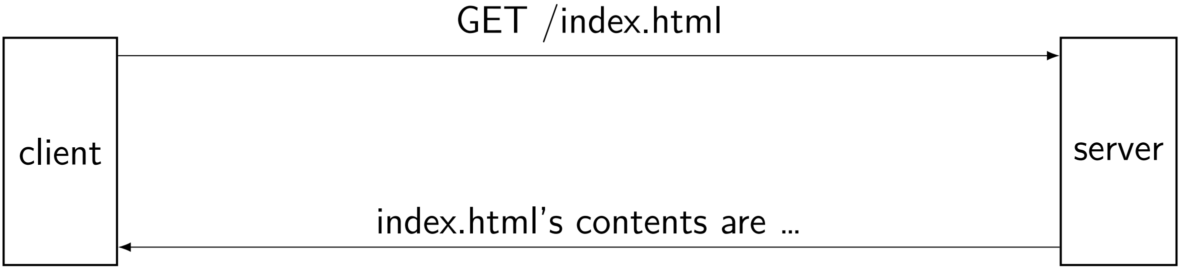


client/server

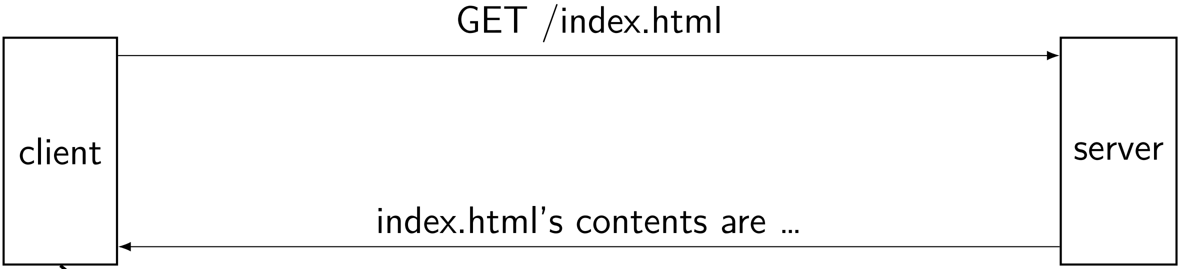


peer-to-peer

client/server model



client/server model

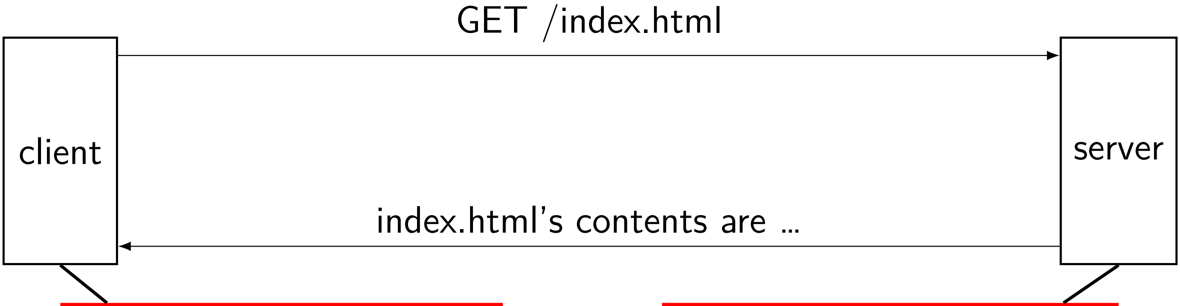


client(s): "sometimes on"

sends requests to server(s)

needs to know
how to contact server

client/server model



client(s): "sometimes on"

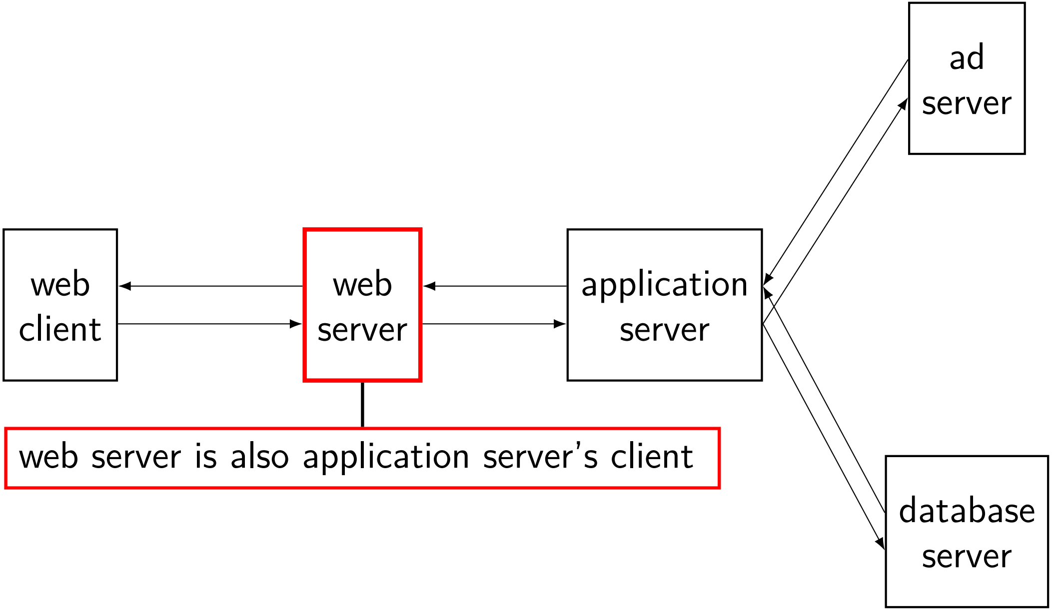
sends requests to server(s)

needs to know
how to contact server

server(s): "always on"

responds to client requests
never initiaties contact
with a client

layers of servers?



example: Wikipedia architecture

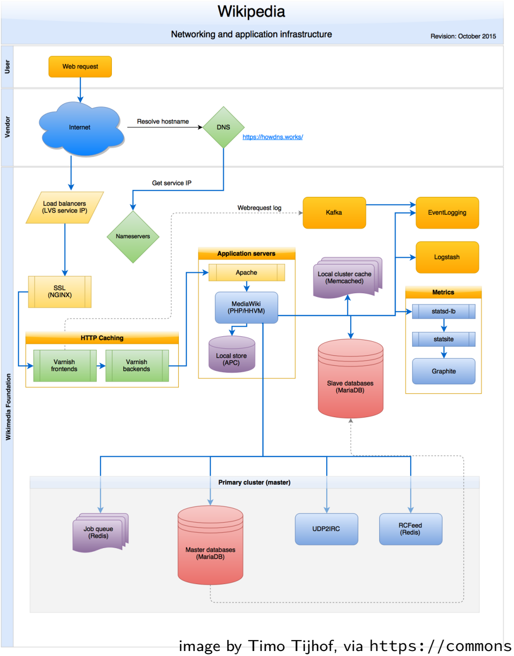


image by Timo Tijhof, via `https://commons.wikimedia.org/wiki/File:Wikipedia_webrequest_flow_2015-10.png`

example: Wikipedia architecture (zoom)

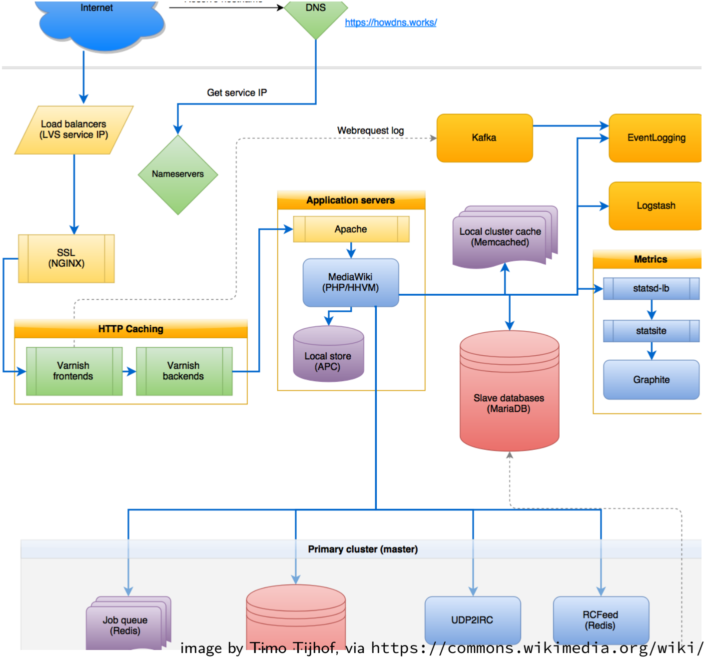


image by Timo Tijhof, via https://commons.wikimedia.org/wiki/File:Wikipedia_webrequest_flow_2015-10.png

peer-to-peer

no always-on server everyone knows about

    hopefully, no one bottleneck — “scalability”

any machine can contact any other machine

    every machine plays an approx. equal role?

set of machines may change over time

why distributed?

multiple machine owners collaborating

delegation of responsiblity to other entity

put (part of) service "in the cloud"

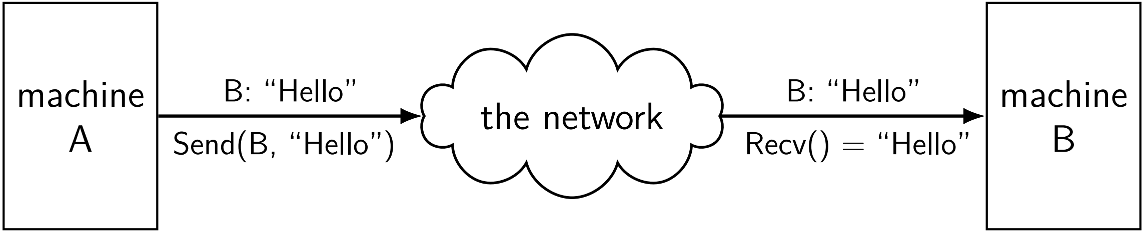
combine many cheap machines to replace expensive machine

easier to add incrementally

redundancy — one machine can fail and *system* still works?

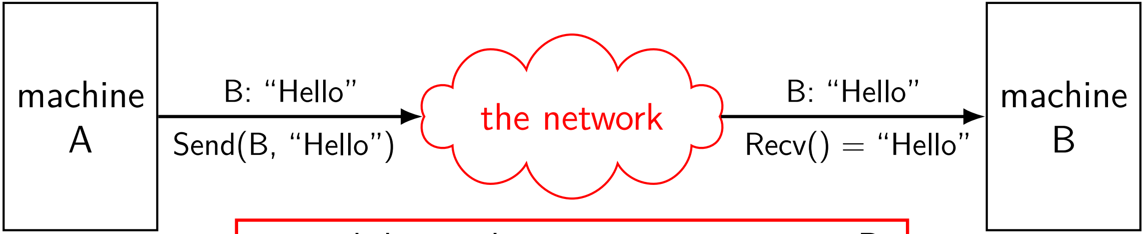
mailbox model

mailbox abstraction: send/receive messages



mailbox model

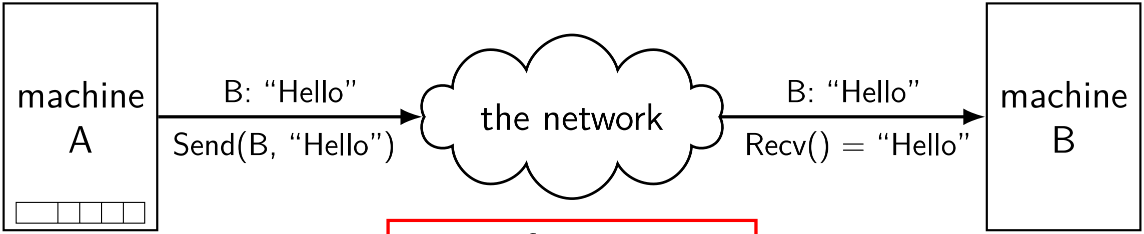
mailbox abstraction: send/receive messages



network knows how to get message to B

mailbox model

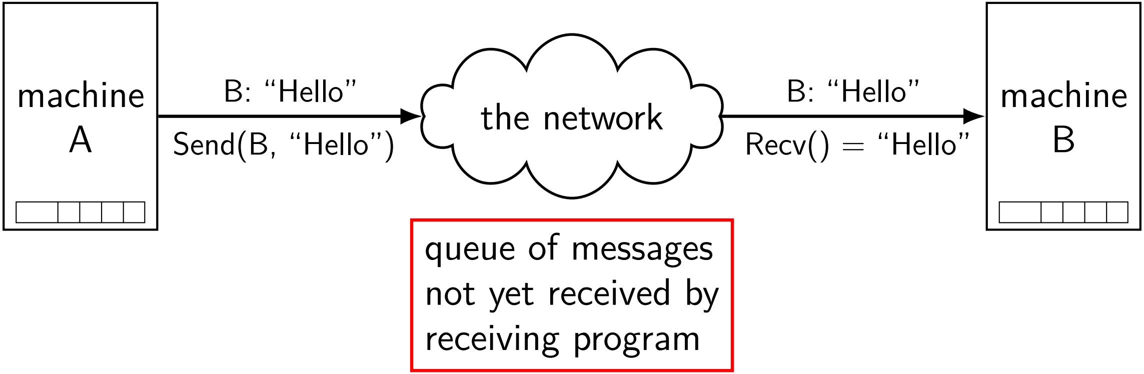
mailbox abstraction: send/receive messages



queue of messages
from sending program
waiting to be sent

mailbox model

mailbox abstraction: send/receive messages



what about servers?

client/server model: server wants to reply to clients

might want to send/receive multiple messages

what about servers?

client/server model: server wants to reply to clients

might want to send/receive multiple messages

can build this with mailbox idea

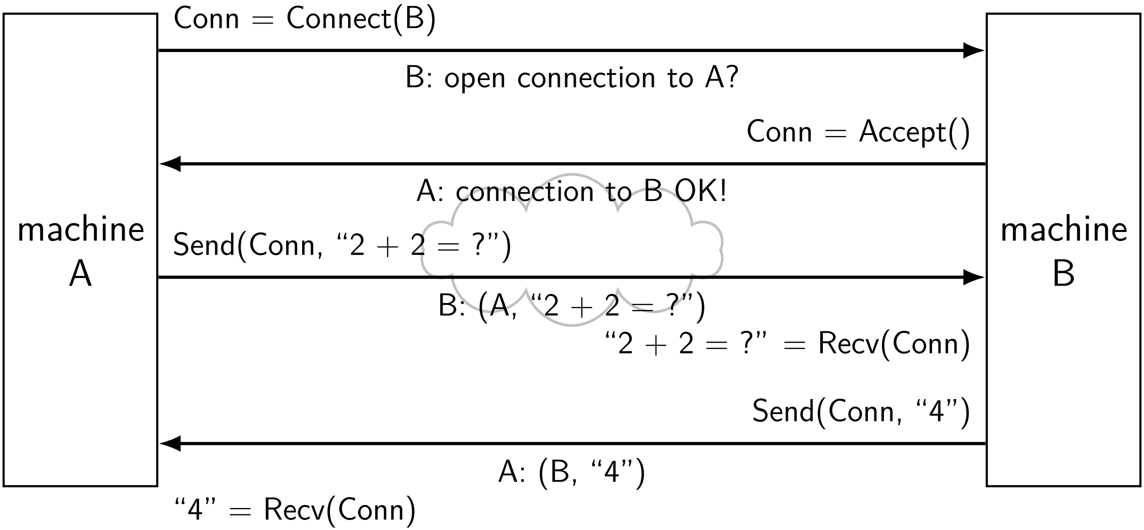
- send a 'return address'
- need to track related messages

common abstraction that does this: the connection

extension: conections

connections: two-way channel for messages

extra operations: connect, accept



connections versus pipes

connections look kinda like two-direction pipes

in fact, in POSIX will have the same API:

each end gets file descriptor representing connection

can use read() and write()

connections over mailboxes

real Internet: mailbox-style communication

- send packets to particular mailboxes
- no gaurentee on order, when received
- no relationship between

connections implemented on top of this

full details: take networking (CS/ECE 4457)

connection missing pieces?

how to specify the machine?

multiple programs on one machine? who gets the message?

names and addresses

name	**address**
logical identifier	location/how to locate
hostname `www.virginia.edu`	IPv4 address `128.143.22.36`
hostname `mail.google.com`	IPv4 address `216.58.217.69`
hostname `mail.google.com`	IPv6 address `2607:f8b0:4004:80b::2005`
filename `/home/cr4bd/NOTES.txt`	inode# `120800873` and device `0x2eh/0x46d`
variable `counter`	memory address `0x7FFF9430`
service name `https`	port number `443`

hostnames

typically use *domain name system* (DNS) to find machine names

maps logical names like `www.virginia.edu`

- chosen for humans
- hierarchy of names

…to *addresses* the network can use to move messages

- numbers
- ranges of numbers assigned to different parts of the network
- network *routers* knows "send this range of numbers goes this way"

connection missing pieces?

how to specify the machine?

multiple programs on one machine? who gets the message?

IPv4 addresses

32-bit numbers

typically written like 128.143.67.11

- four 8-bit decimal values separated by dots
- first part is most significant
- same as $128 \cdot 256^3 + 143 \cdot 256^2 + 67 \cdot 256 + 11 = 2\,156\,782\,459$

organizations get blocks of IPs

- e.g. UVa has 128.143.0.0–128.143.255.255
- e.g. Google has 216.58.192.0–216.58.223.255 and 74.125.0.0–74.125.255.255 and 35.192.0.0–35.207.255.255

some IPs reserved for non-Internet use (127.*, 10.*, 192.168.*)

IPv6 addresses

IPv6 like IPv4, but with 128-bit numbers

written in hex, 16-bit parts, seperated by colons (`:`)

strings of 0s represented by double-colons (`::`)

typically given to users in blocks of 2^{80} or 2^{64} addresses

no need for address translation?

`2607:f8b0:400d:c00::6a` =

`2607:f8b0:400d:0c00:0000:0000:0000:006a`

$\texttt{2607f8b0400d0c00000000000000006a}_{\text{SIXTEEN}}$

selected special IPv6 addresses

`::1` = localhost

anything starting with `fe80` = link-local addresses

never forwarded by routers

backup slides

exercise

which are likely advantages of client/server model over peer-to-peer?

[A] easier to make whole system work despite failure of any machine

[B] easier to handle most machines being offline a majority of the time

[C] better suited to a mix of a few very big/high-performance and many small/low-performance machines

fragments

Linux FS: 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

mounting filesystems

Unix-like system

root filesystem appears as /

other filesystems *appear as directory*

- e.g. lab machines: my home dir is in filesystem at /net/zf15

directories that are filesystems look like normal directories

- `/net/zf15/..` is `/net` (even though in different filesystems)

mounts on a dept. machine

```
/dev/sda1 on / type ext4 (rw,errors=remount-ro)
proc on /proc type proc (rw,noexec,nosuid,nodev)
...
udev on /dev type devtmpfs (rw,mode=0755)
devpts on /dev/pts type devpts (rw,noexec,nosuid,gid=5,mode=0620)
tmpfs on /run type tmpfs (rw,noexec,nosuid,size=10%,mode=0755)
...
/dev/sda3 on /localtmp type ext4 (rw)
...
zfs1:/zf2 on /net/zf2 type nfs (rw,hard,intr,proto=udp,nfsvers=3,
                                noacl,sloppy,addr=128.143.136.9)
zfs3:/zf19 on /net/zf19 type nfs (rw,hard,intr,proto=udp,nfsvers=3,
                                  noacl,sloppy,addr=128.143.67.236)
zfs4:/sw on /net/sw type nfs (rw,hard,intr,proto=udp,nfsvers=3,
                              noacl,sloppy,addr=128.143.136.9)
zfs3:/zf14 on /net/zf14 type nfs (rw,hard,intr,proto=udp,nfsvers=3,
                                  noacl,sloppy,addr=128.143.67.236)
...
```

kernel FS abstractions

Linux: *virtual file system* API

object-oriented, based on FFS-style filesystem

to implement a filesystem, create object types for:

- superblock (represents “header”)
- inode (represents file)
- dentry (represents cached directory entry)
- file (represents *open file*)

common code handles directory traversal

- and caches directory traversals

common code handles file descriptors, etc.

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

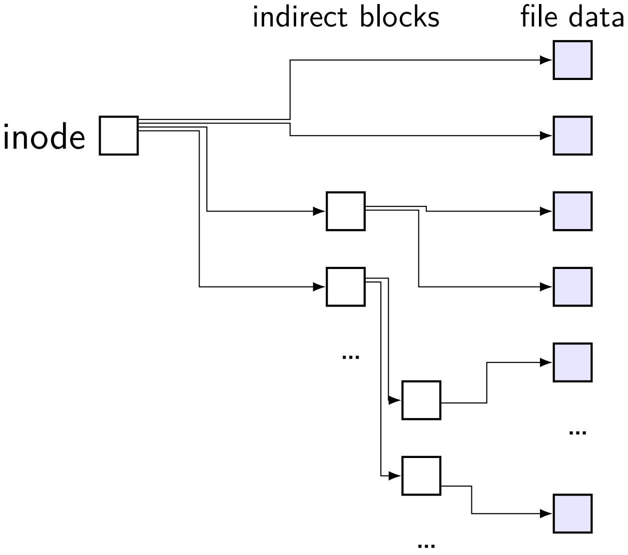
can access *snapshot* of files at prior time

mechanism: copy-on-write

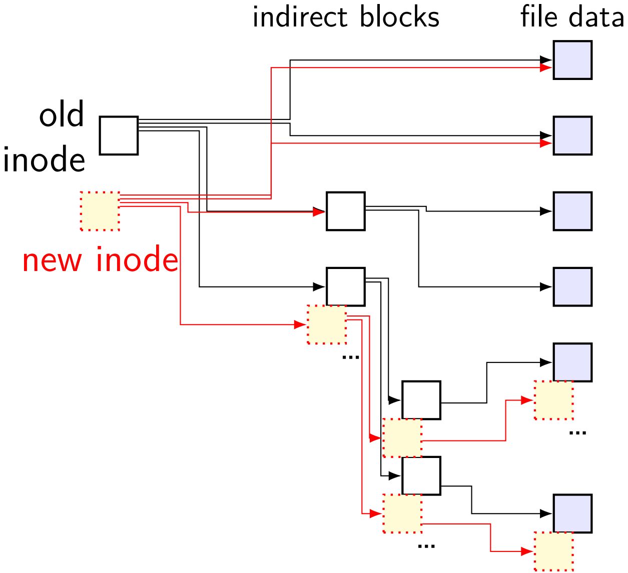
changing file makes new copy of filesystem

common parts shared between versions

inode and copy-on-write



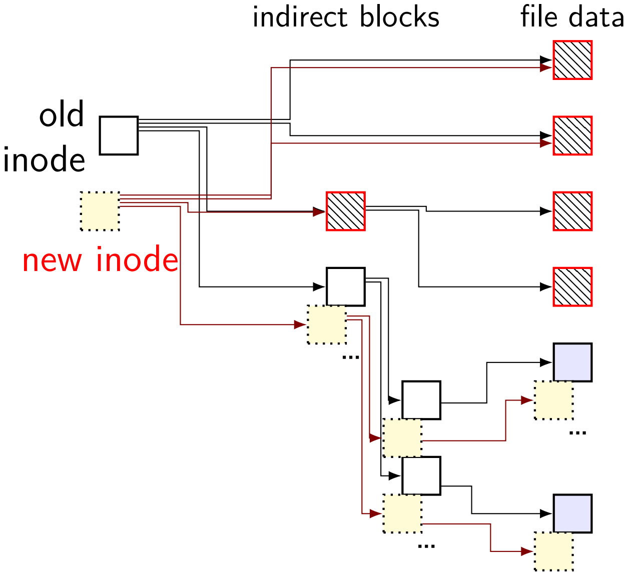
inode and copy-on-write



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

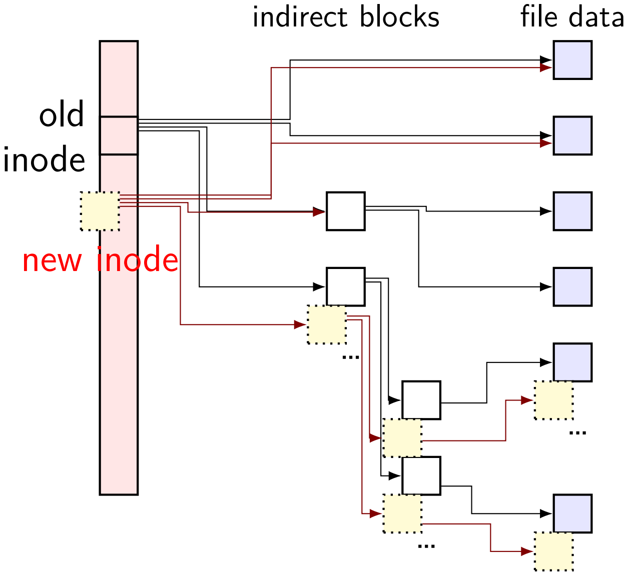
both old+new inode valid

inode and copy-on-write



unchanged parts of file shared

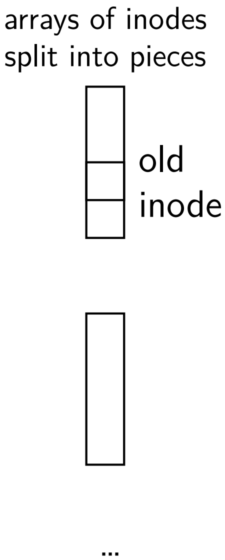
inode and copy-on-write



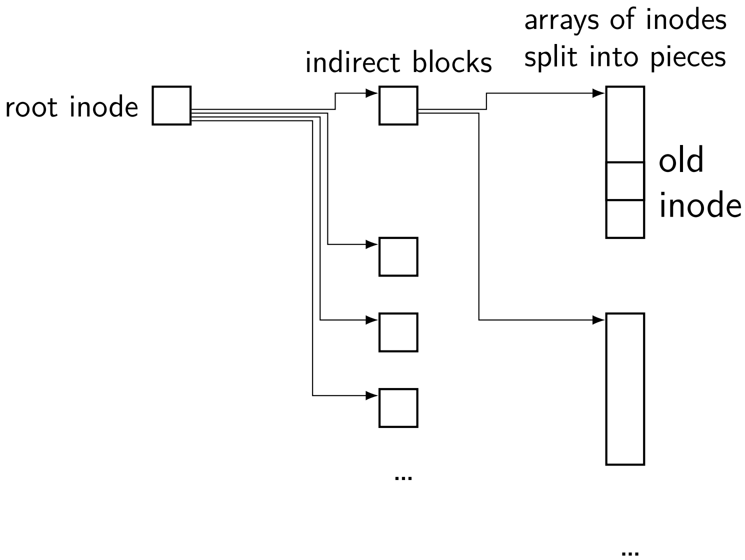
challenge: FFS/xv6/ext2 design has big array of inodes

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

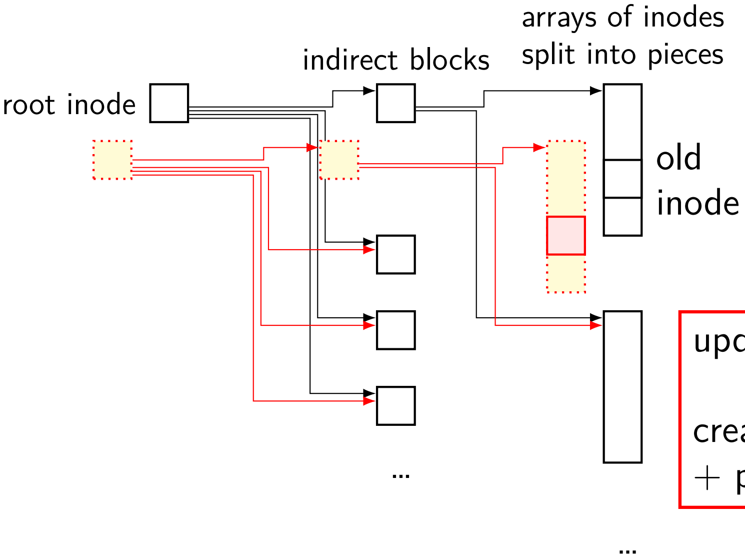
extra indirection for inode array



extra indirection for inode array



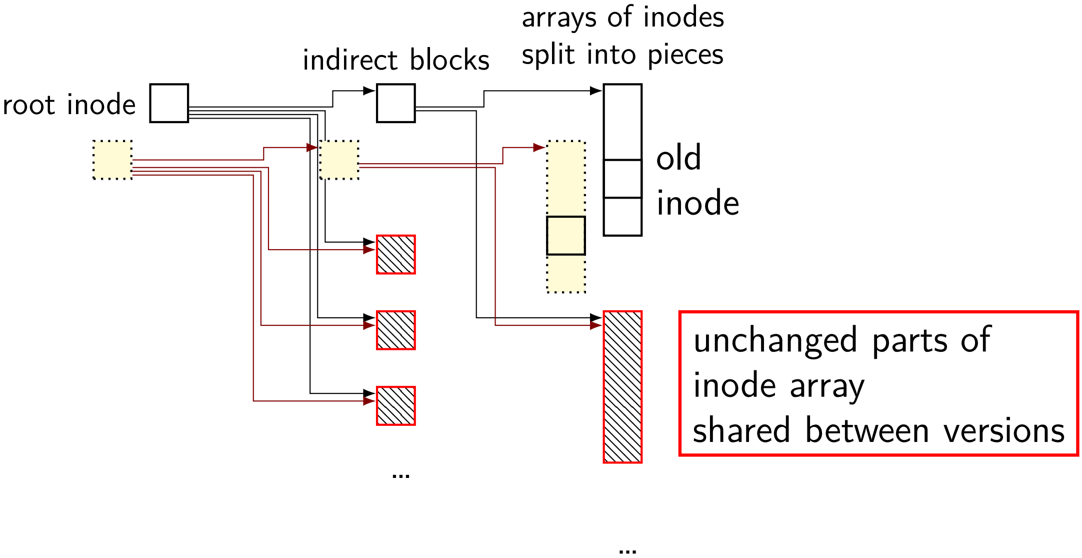
extra indirection for inode array



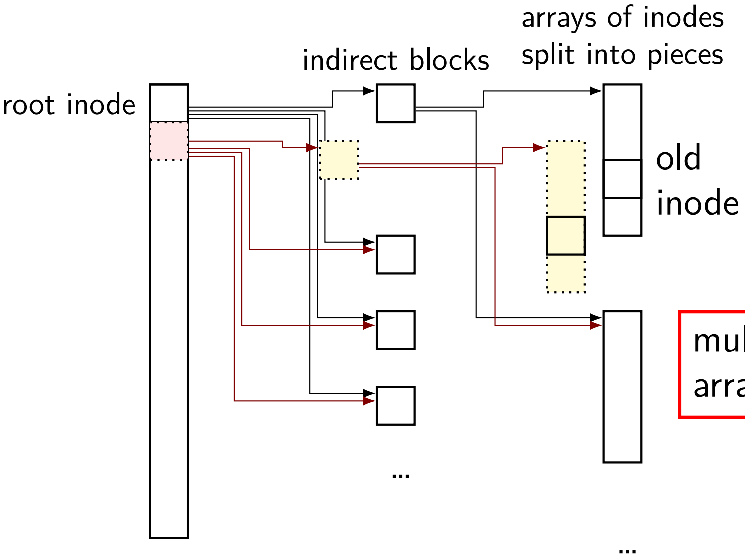
update one inode?

create new root inode
+ pointers

extra indirection for inode array



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

aside: FAT date encoding

seperate date and time fields (16 bits, little-endian integers)

bits 0-4: seconds (divided by 2), 5-10: minute, 11-15: hour

bits 0-4: day, 5-8: month, 9-15: year (minus 1980)

sometimes extra field for 100s(?) of a second

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

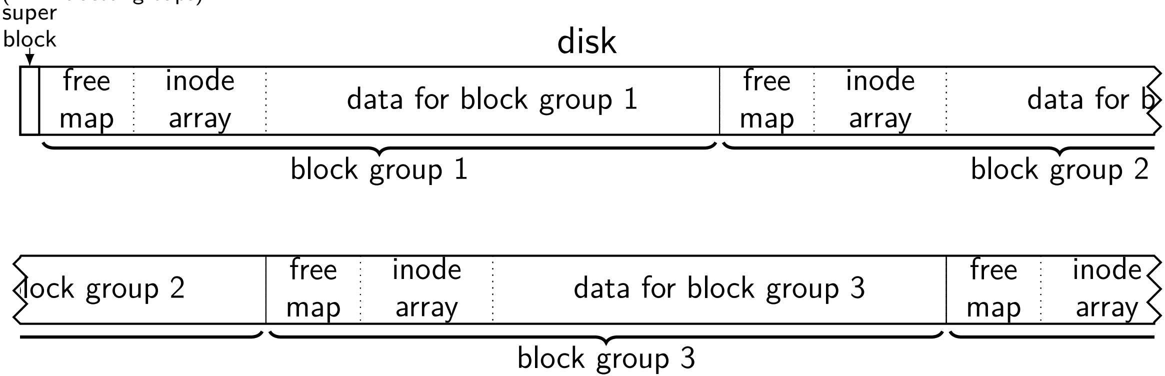
Linux's ext2 filesystem based on FFS

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

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

block groups

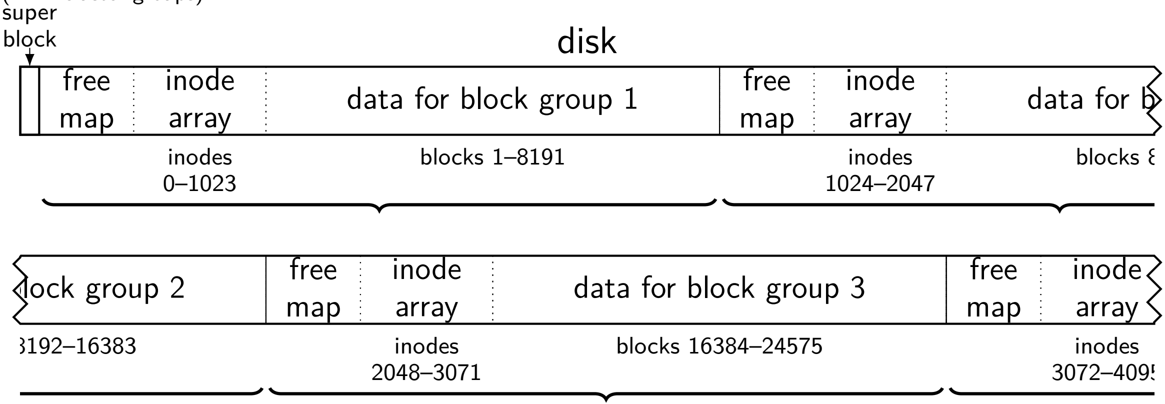
(AKA cluster groups)



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

block groups

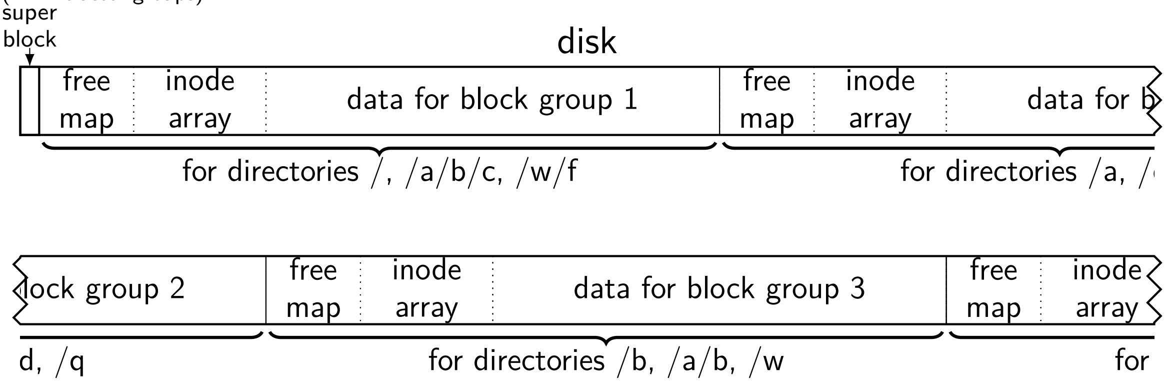
(AKA cluster groups)



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

block groups

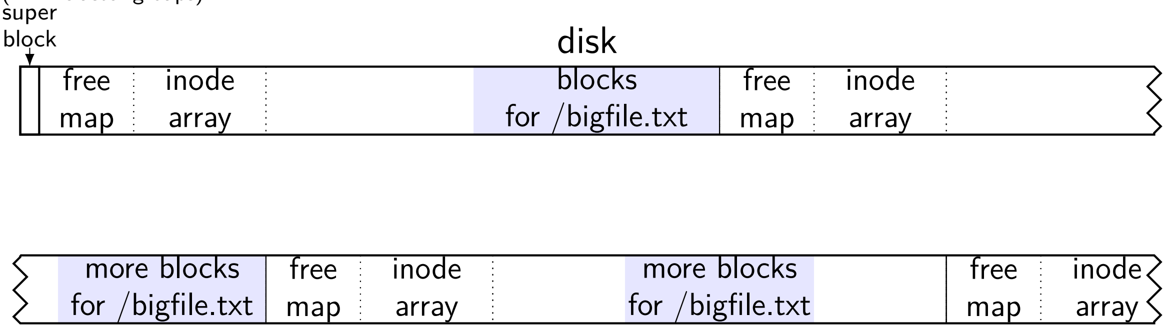
(AKA cluster groups)



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

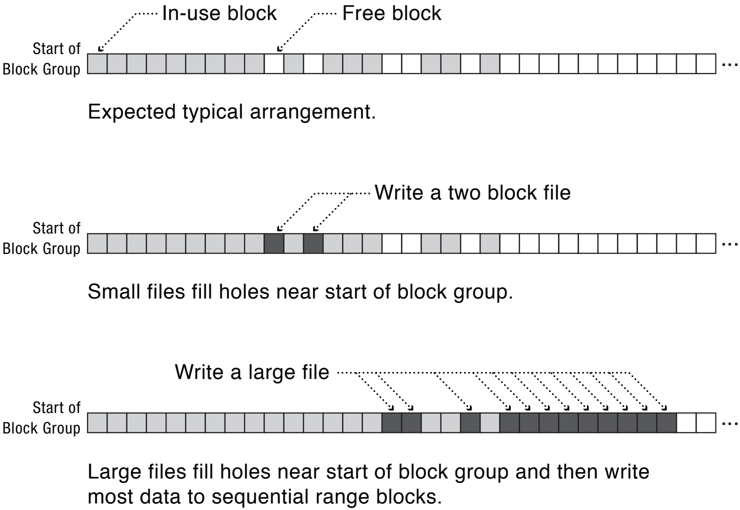
block groups

(AKA cluster groups)



large files might need to be split across block groups

allocation within block groups



Anderson and Dahlin, *Operating Systems: Principles and Practice* 2nd edition, Figure 13.14

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

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

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

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

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

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

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?

ext2 indirect blocks (2) (solution)

byte 2^{15} = 32KB into file

12 direct pointers: first 1K (block size) $\times$ 12 bytes of data

1 indirect pointer:

- points to block with 1K (block size)/4 byte (pointer size) = 256 pointers
- 256 pointers point to 1K blocks
- next 256KB of data

going to be (32 - 12)th element

exercise

say xv6 filesystem with:

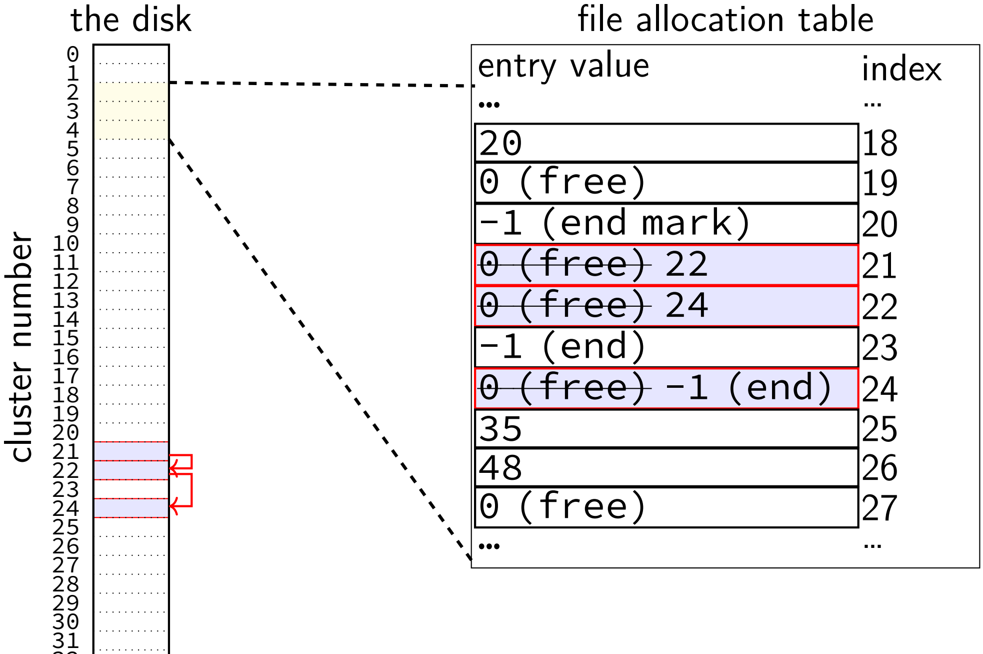
- 64-byte inodes (12 direct + 1 indirect pointer)
- 16-byte directory entries
- 512 byte blocks
- 2-byte block pointers

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

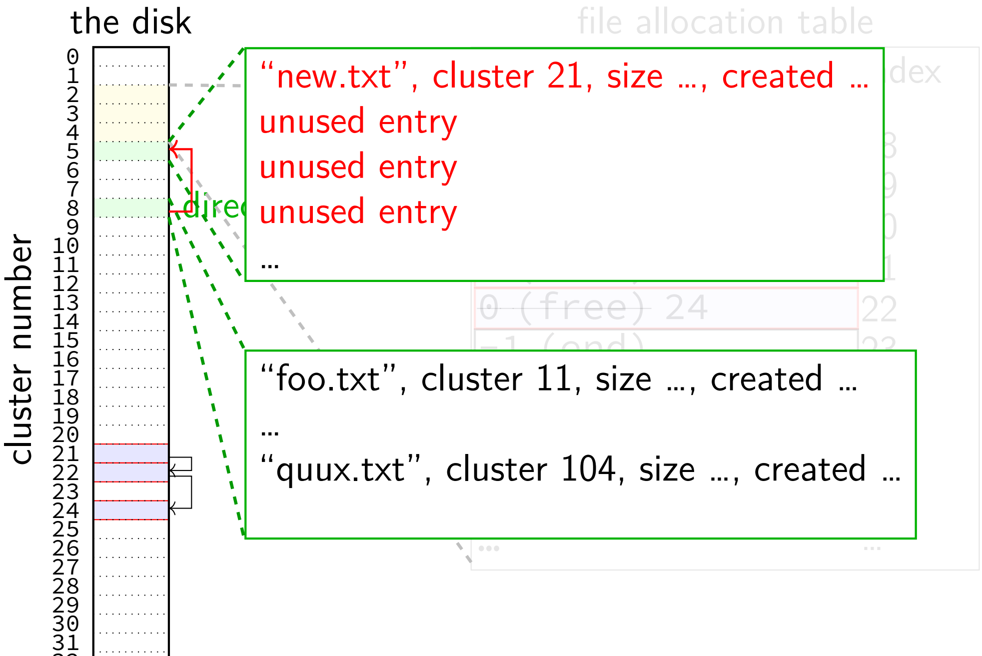
- remember: blocks could include blocks storing data or block pointers or directory enties

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

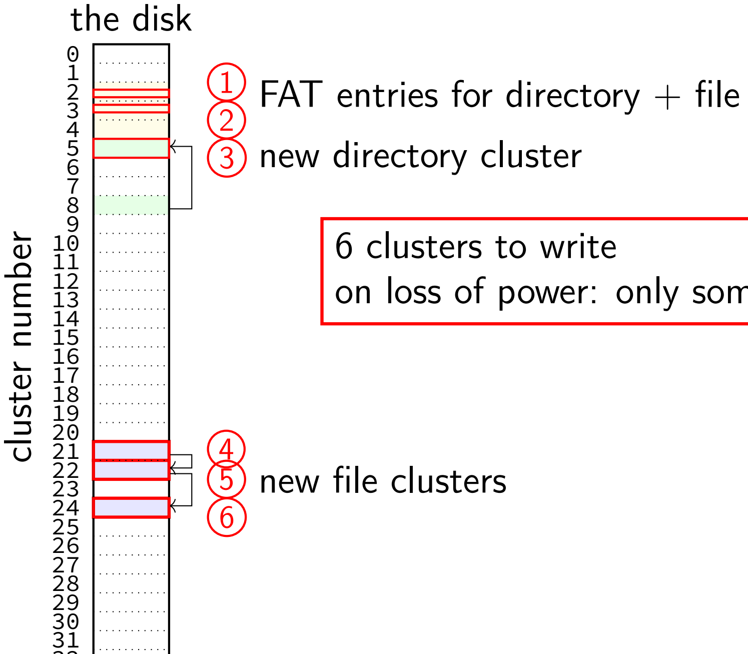
recall: FAT: file creation (1)



recall: FAT: file creation (2)



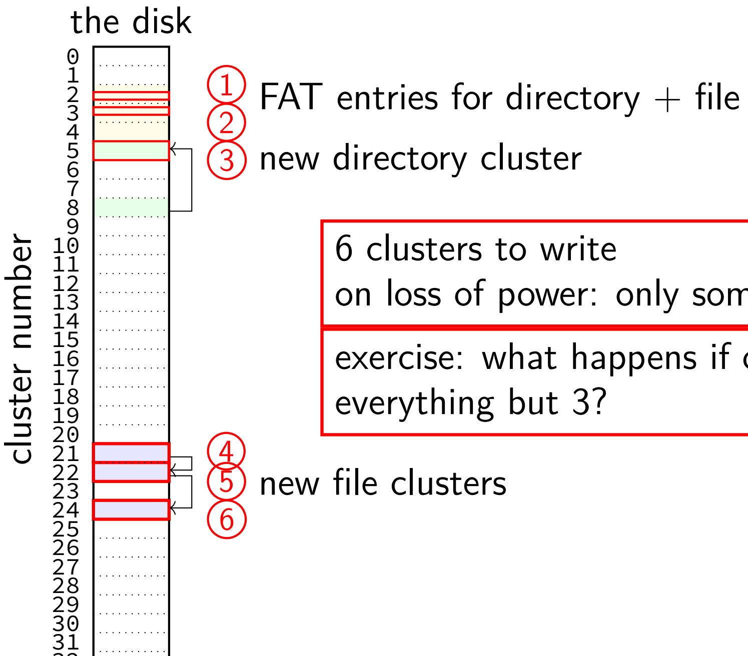
exercise: FAT file creation



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

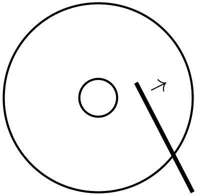
on boot: if clean bit clear, run fsck first

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

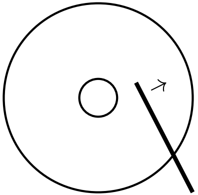


ordering and disk performance

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



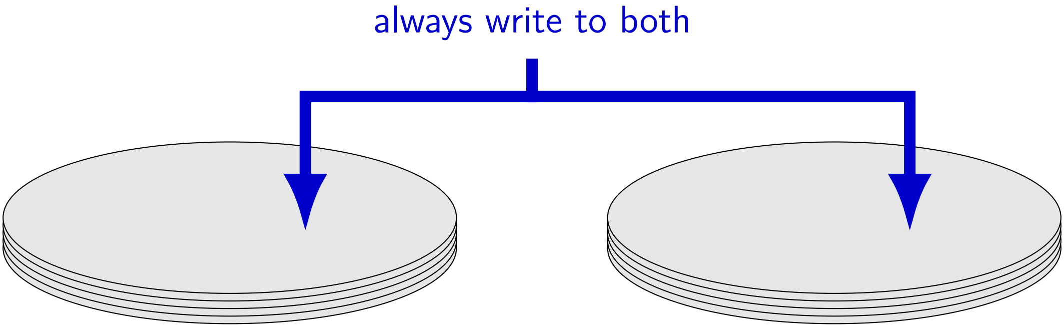
ordering constraints make this hard:

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

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

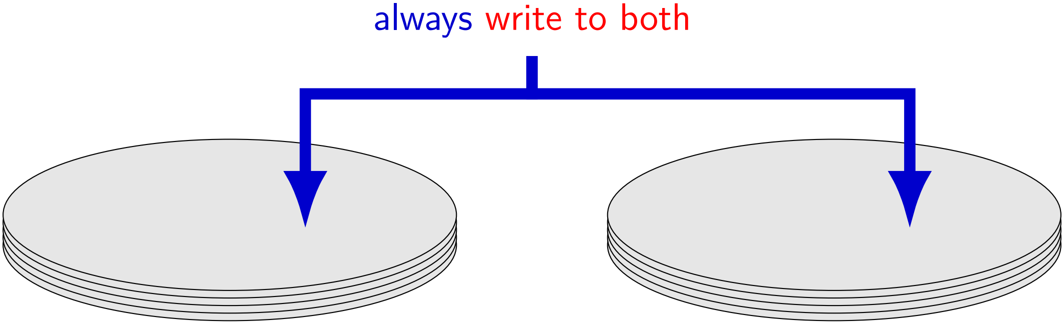
mirroring whole disks

alternate strategy: write everything to two disks



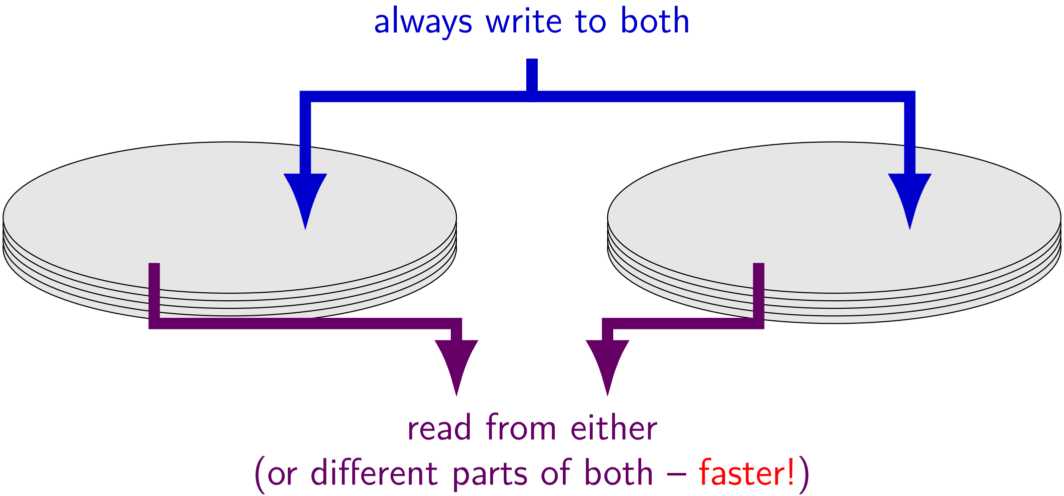
mirroring whole disks

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

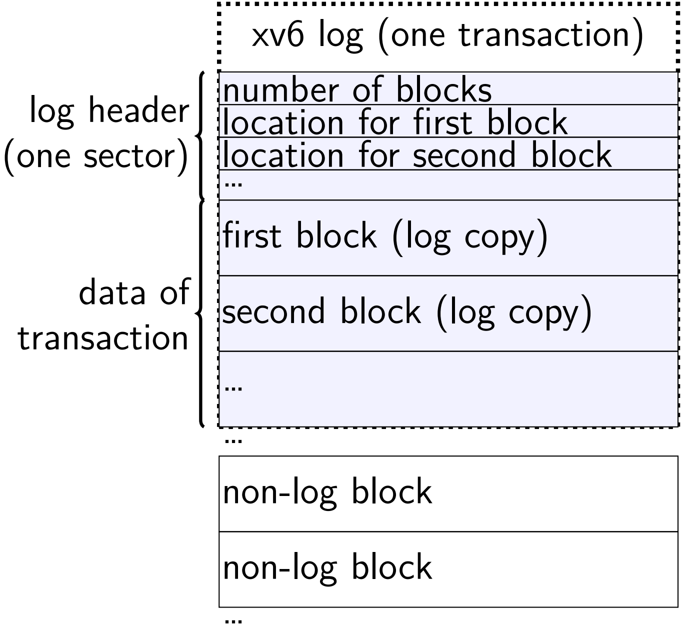
extra disk of data: some formula based on the original disks

- common choice: bitwise XOR

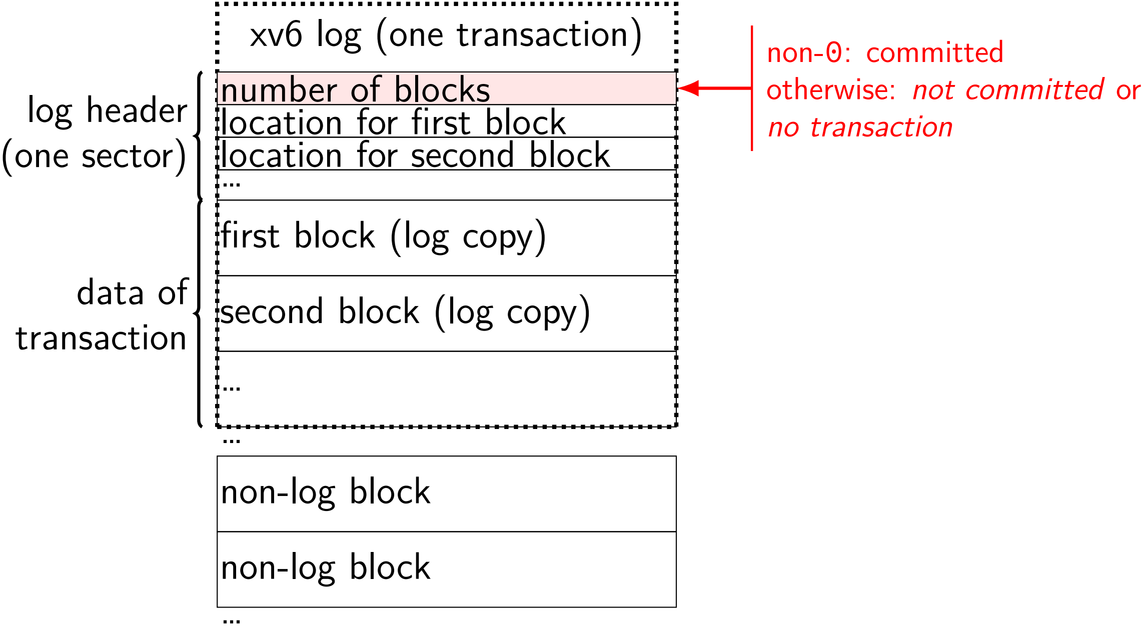
common set of schemes like this: RAID

- Redundant Array of Independent Disks

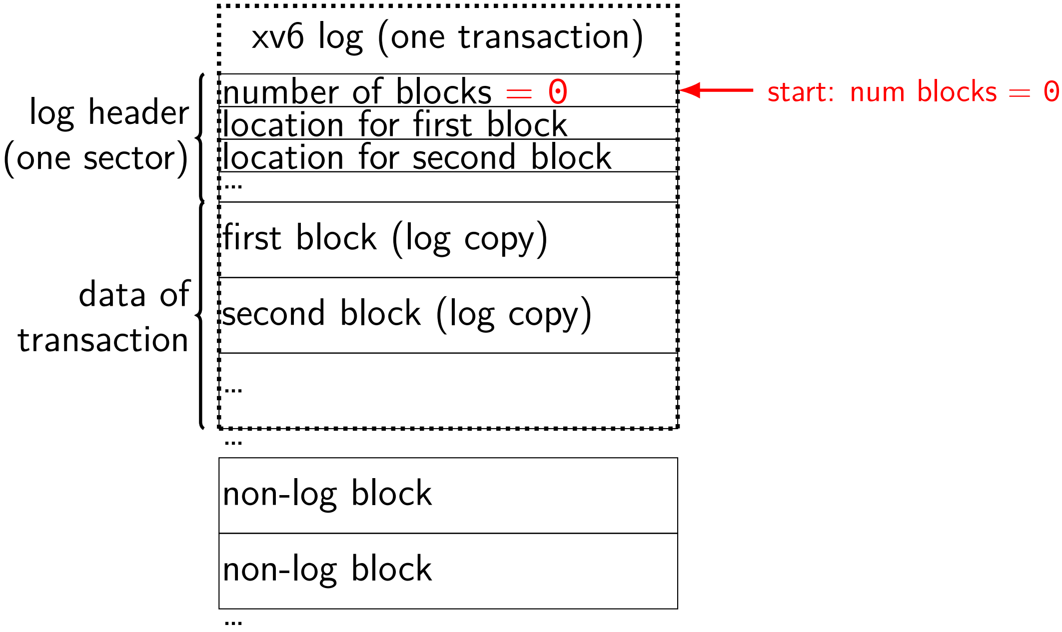
the xv6 journal



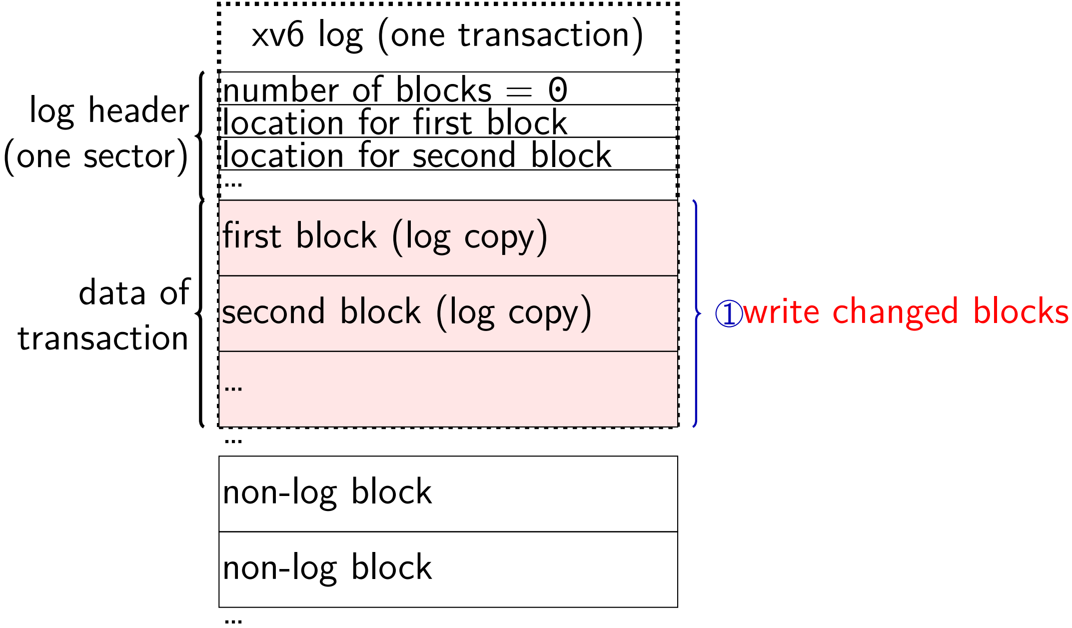
the xv6 journal



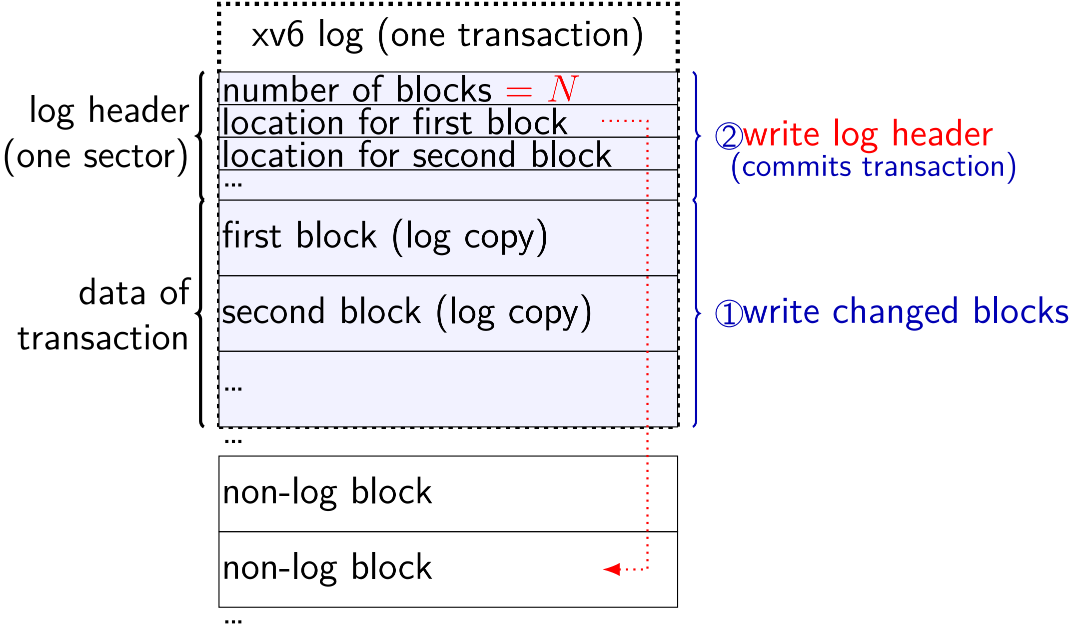
the xv6 journal



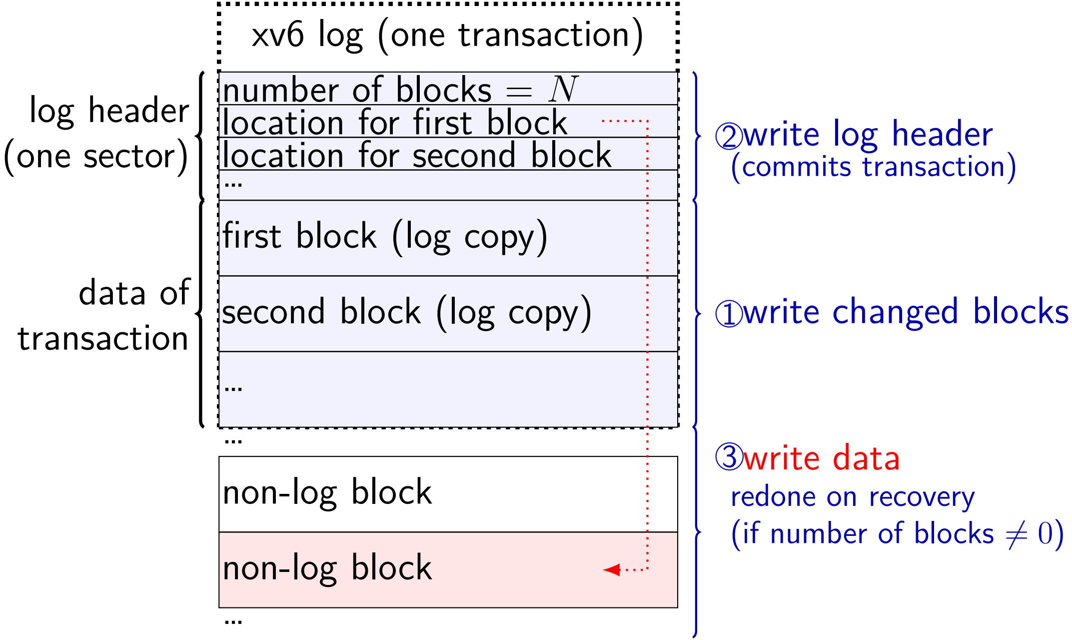
the xv6 journal



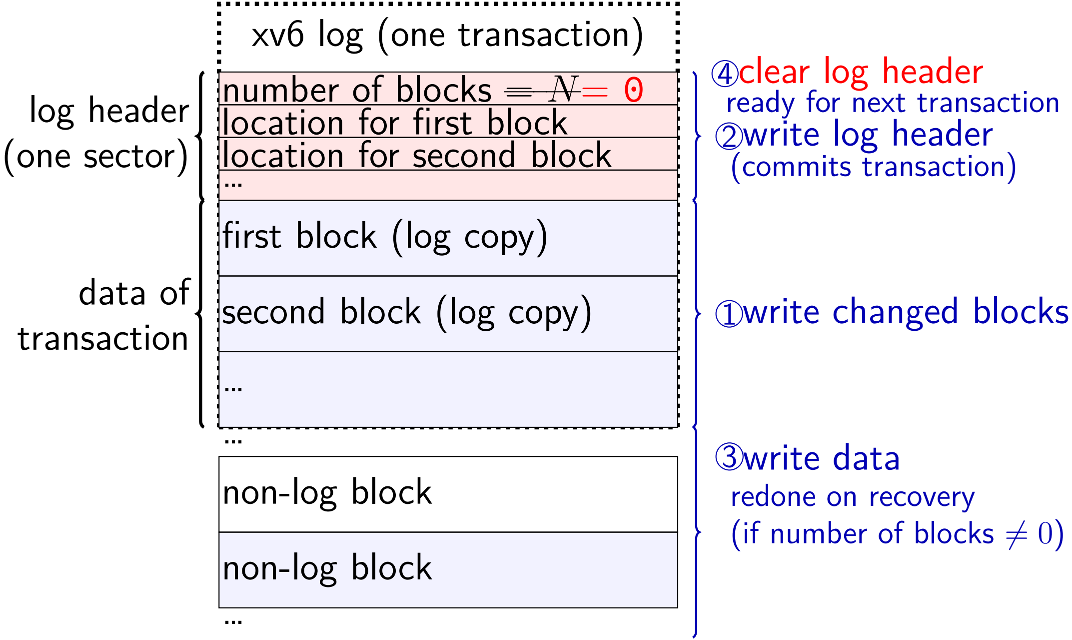
the xv6 journal



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?