Filesystem Reliability + Sockets Intro

last time

extents

non-binary trees on disk

extra copies of data

- two or more FATs, two or more superblocks
- mirroring
- *erasure coding*: redundancy without full copies
- examples of RAID 4/5

careful ordering of operations

- key idea: don't store pointers to bad data
- file system checking (fsck) — scan disk for inconsistencies

anonymous feedback

(paraphrased) the TAs don't know about using mmap

while I recommend mmap, you are welcome to /will succeed using seek/read

have given a little tutorial/info for TAs

inode-based FS: careful ordering

mark blocks as allocated before referring to them from directories

write data blocks before writing pointers to them from inodes

write inodes before directory entries pointing to it

remove inode from directory before marking inode as free

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

idea: better to waste space than point to bad data

inode-based FS: creating a file

normal operation

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

inode-based FS: creating a file

normal operation

allocate data block

write data block

update free block map

update file inode

update directory entry

filename+inode number

update direcotry inode

modification time

general rule:

better to waste space

than point to bad data

mark blocks/inodes used before writing

inode-based FS: creating a file

normal operation	recovery (`fsck`)
allocate data block	read all directory entries
write data block	scan all inodes
update free block map	free unused inodes unused = not in directory
update file inode	free unused data blocks unused = not in inode lists
update directory entry filename+inode number	scan directories for missing update/access times
update direcotry inode modification time	

inode-based FS: exercise: unlink

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

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

assume not the last hard link

inode-based FS: exercise: unlink

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

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

assume not the last hard link

what does recovery operation do?

inode-based FS: exercise: unlink last

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

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

assume is the last hard link

inode-based FS: exercise: unlink last

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

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

assume is the last hard link

what does recovery operation do?

fsck

Unix typically has an `fsck` utility

checks for *filesystem consistency*

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

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

fsck costs

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

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

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

pretty expensive and slow

running fsck automatically

common to have “clean” bit in superblock

last thing written (to set) on shutdown

first thing written (to clear) on startup

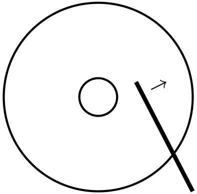
on boot: if clean bit clear, run fsck first

ordering and disk performance

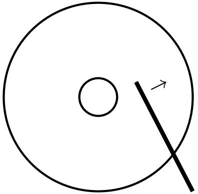
recall: seek times

would like to order writes based on locations on disk

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



ordering and disk performance



recall: seek times

would like to order writes based on locations on disk

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

ordering constraints make this hard:

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

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

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

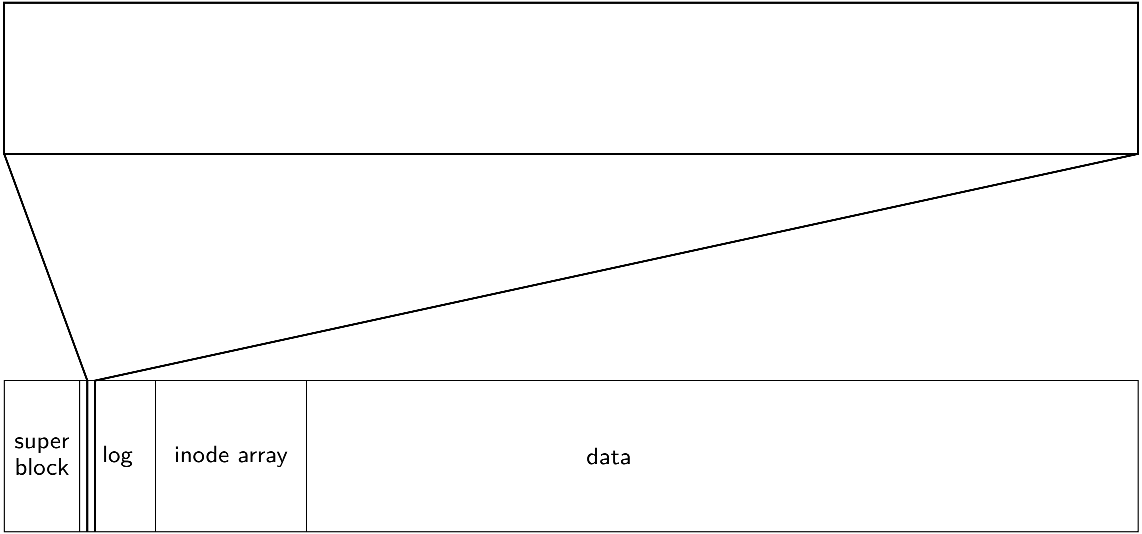
concept: transaction

transaction: bunch of updates that happen all at once

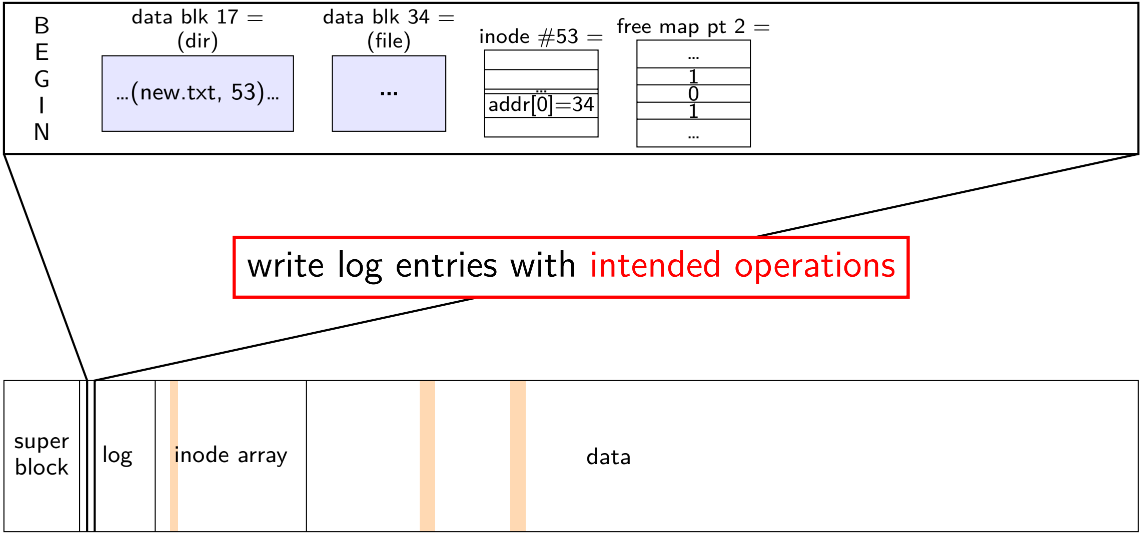
implementation trick: one update means transaction “commits”

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

redo logging: file creation

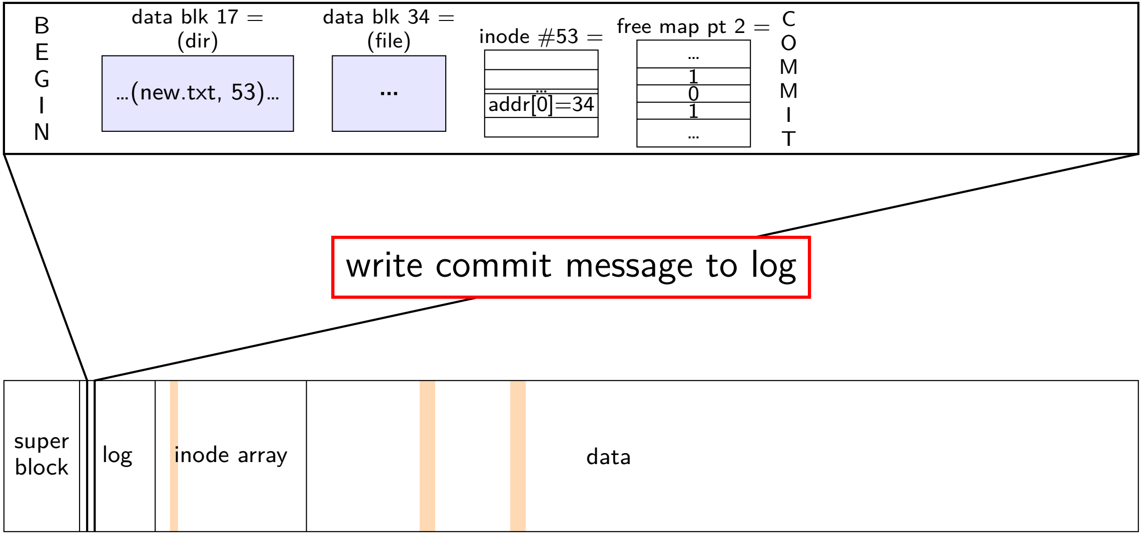


redo logging: file creation

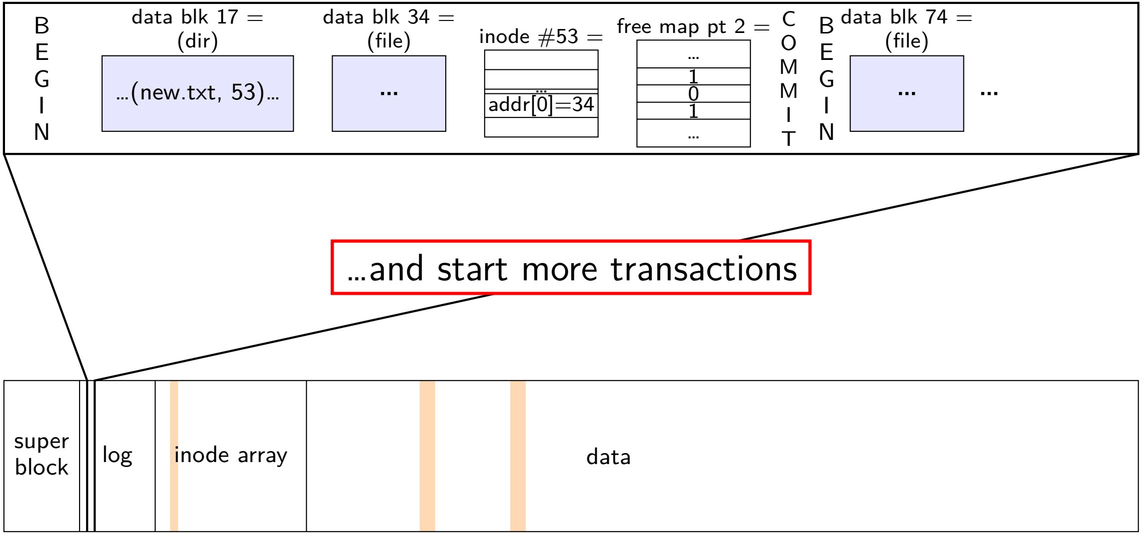


write log entries with intended operations

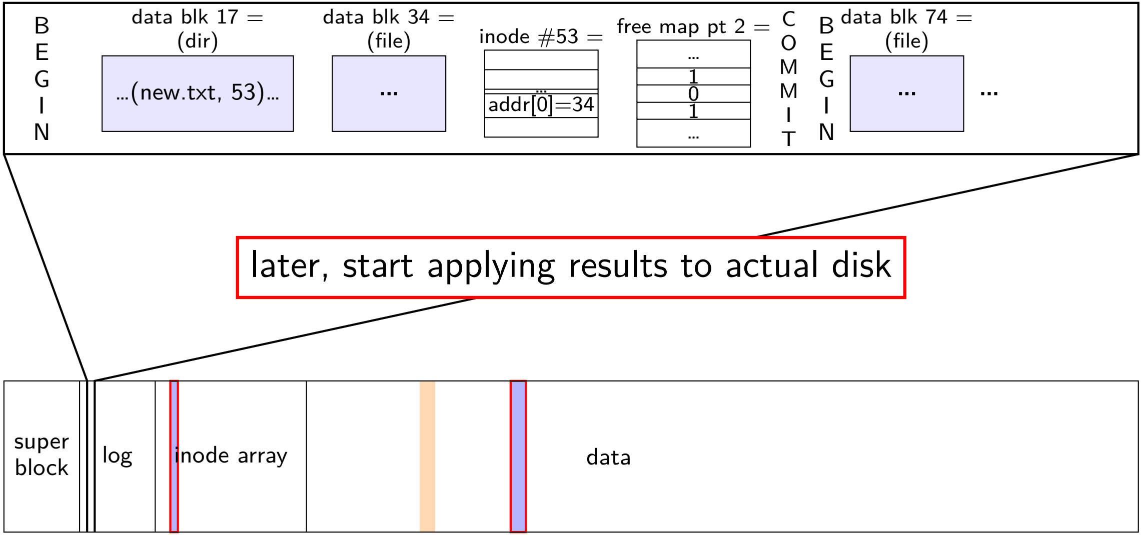
redo logging: file creation



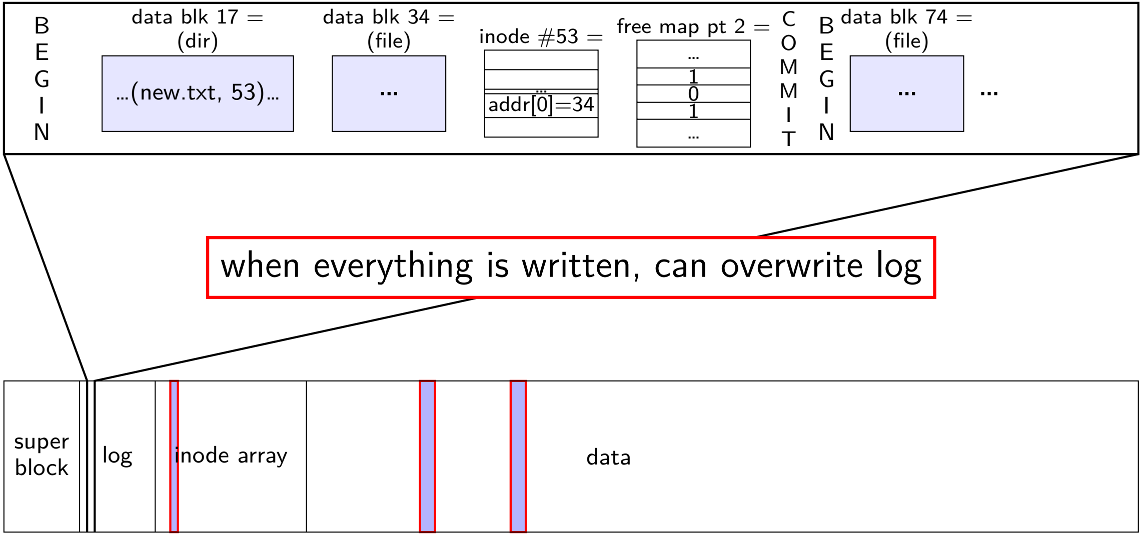
redo logging: file creation



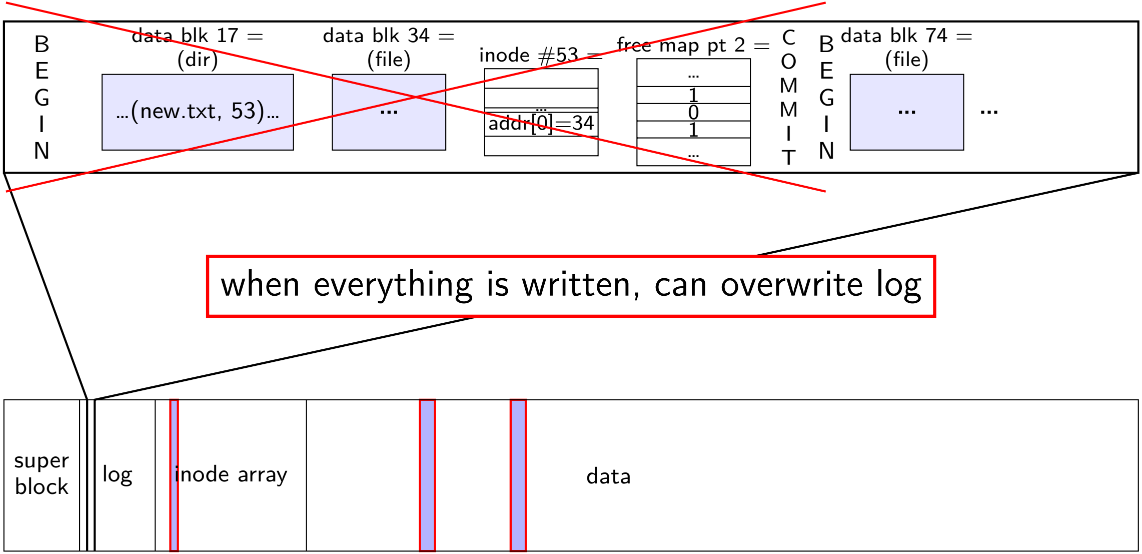
redo logging: file creation



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

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

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

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

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

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 with new inode value

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

good example: overwrite data block with new value

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

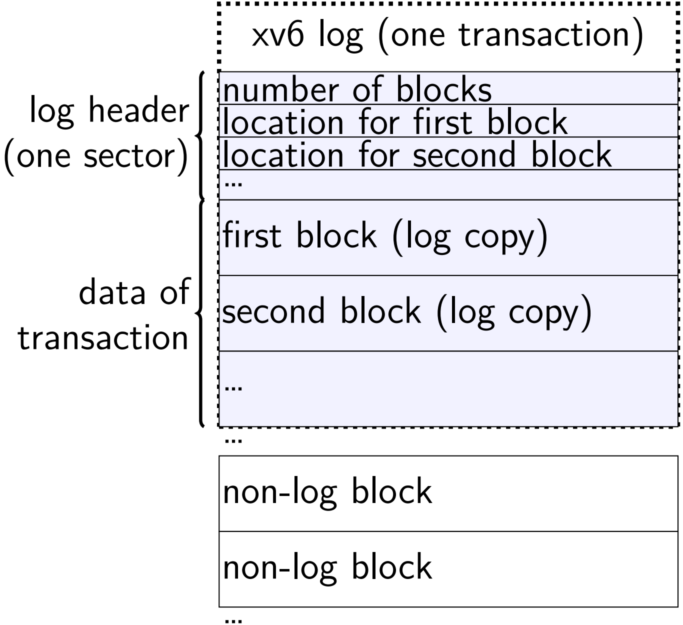
if exists, the operation *will be done eventually*

actually update the real data

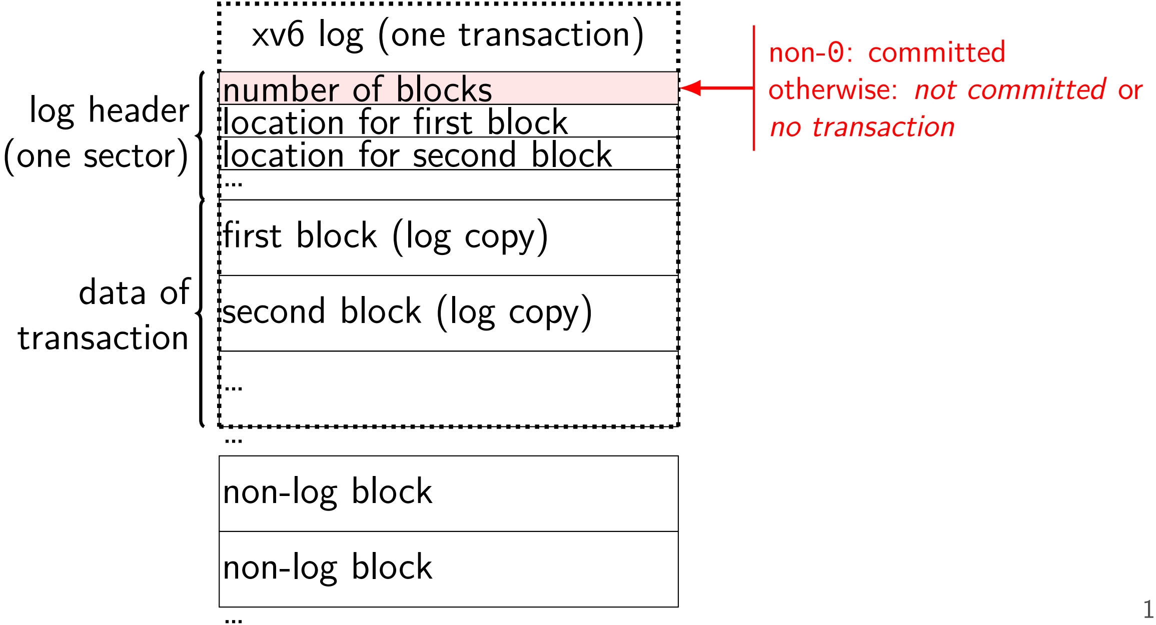
redo logging and filesystems

filesystems that do redo logging are called *journalling filesystems*

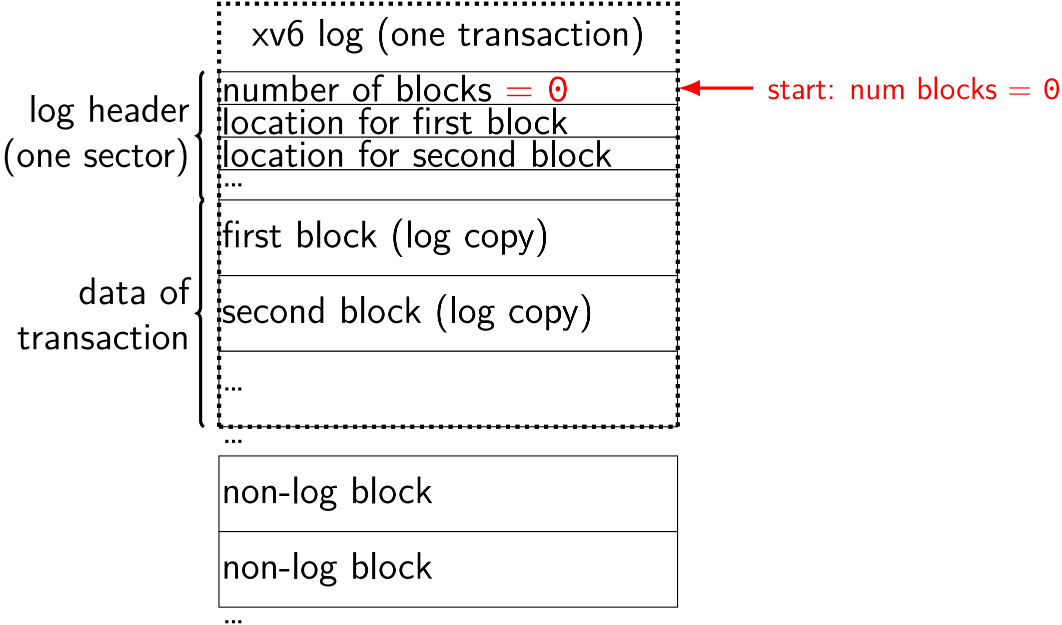
the xv6 journal



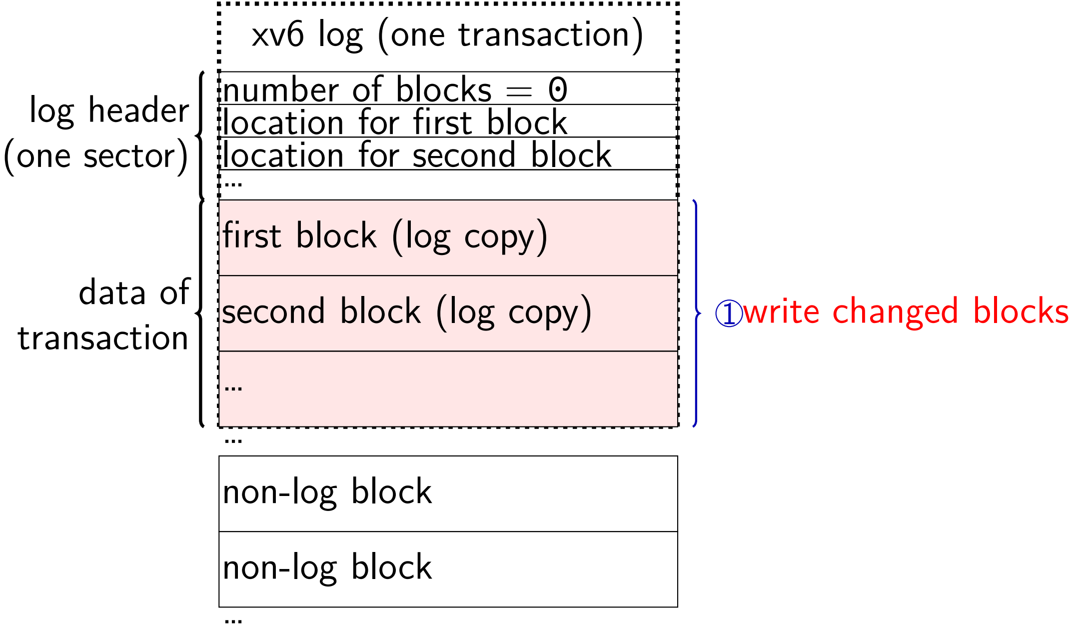
the xv6 journal



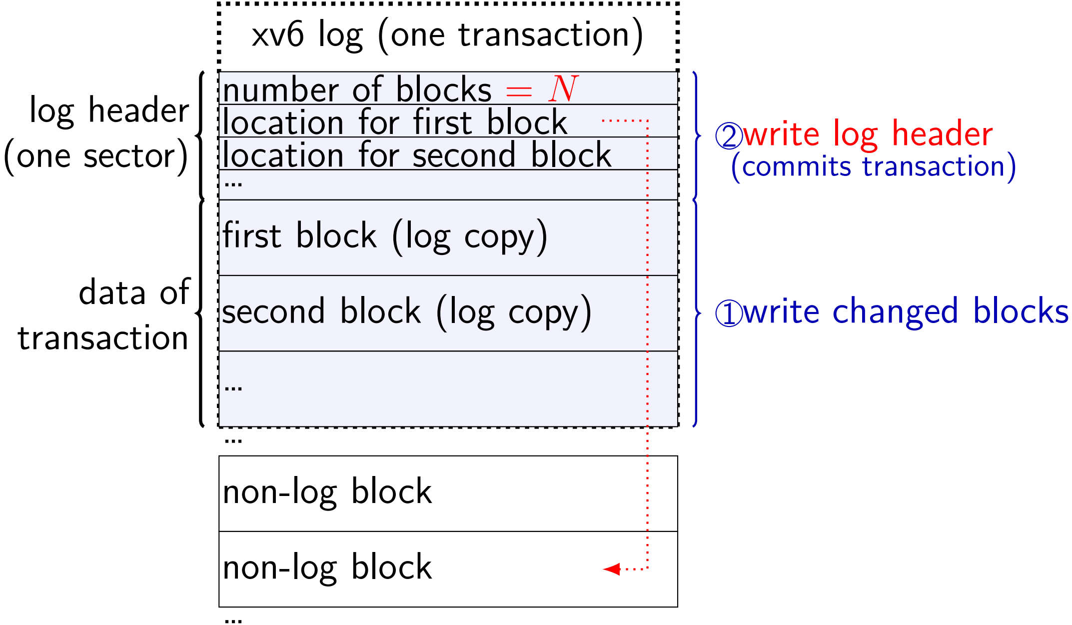
the xv6 journal



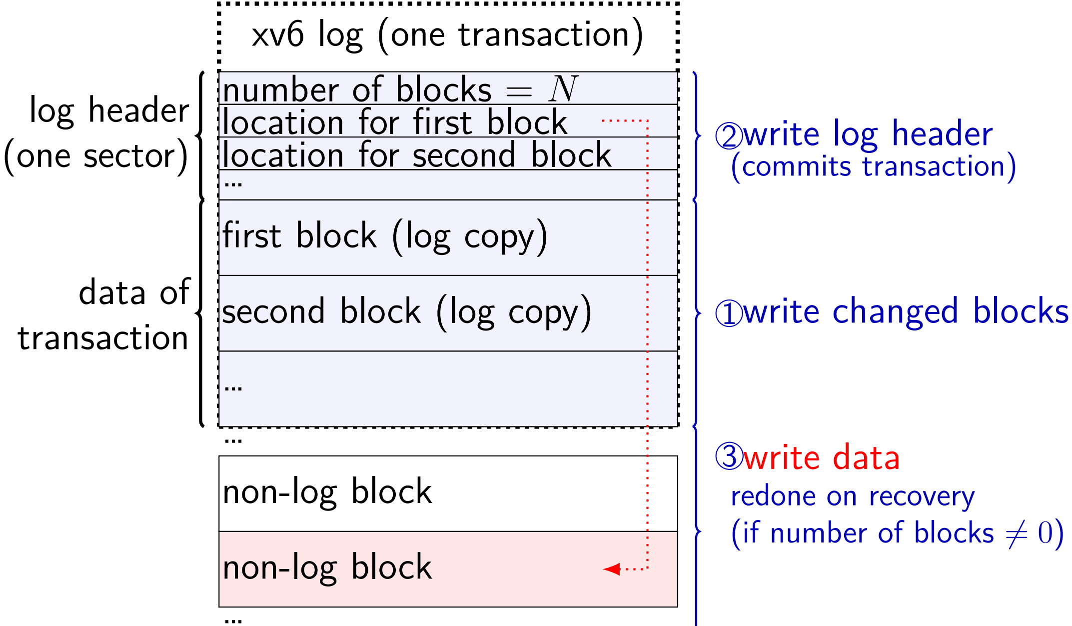
the xv6 journal



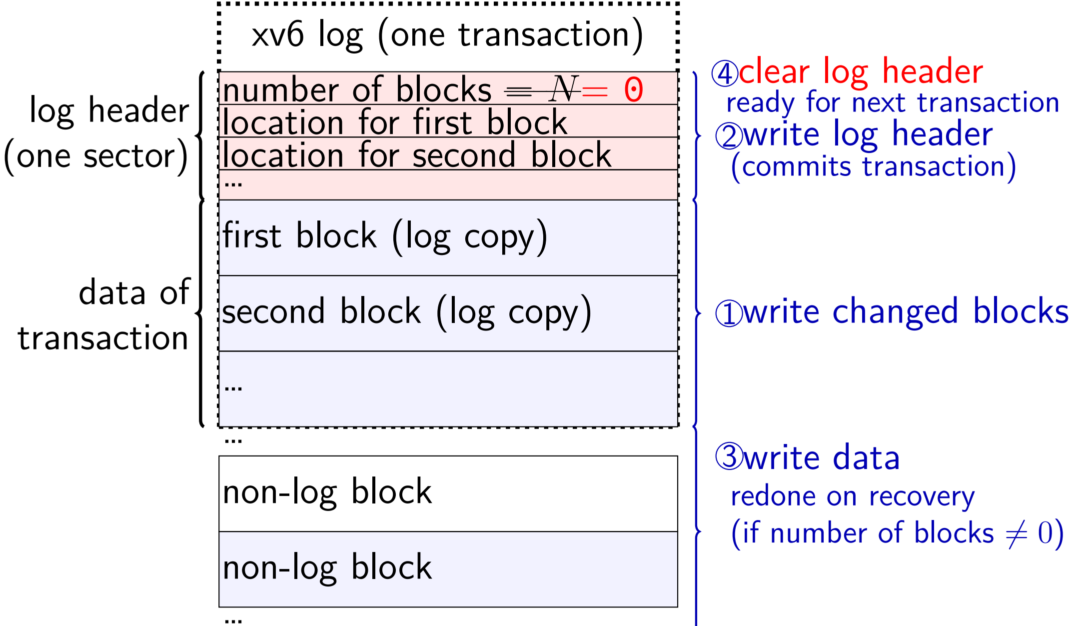
the xv6 journal



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?

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?

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?

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 combine file updates with careful ordering??

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

degrees of durability

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

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

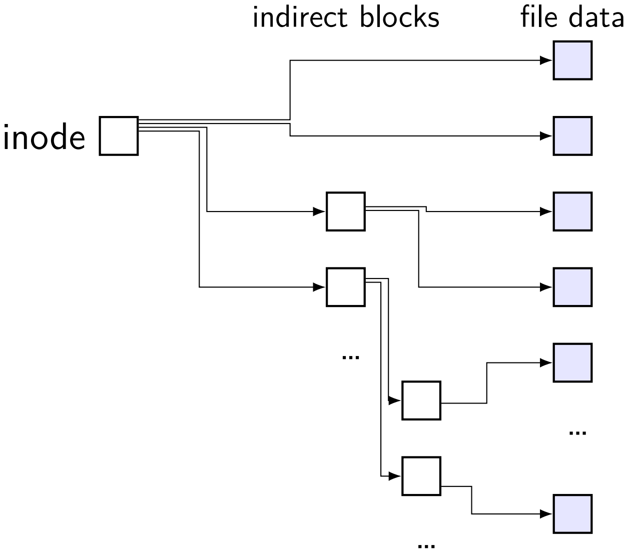
can access *snapshot* of files at prior time

mechanism: copy-on-write

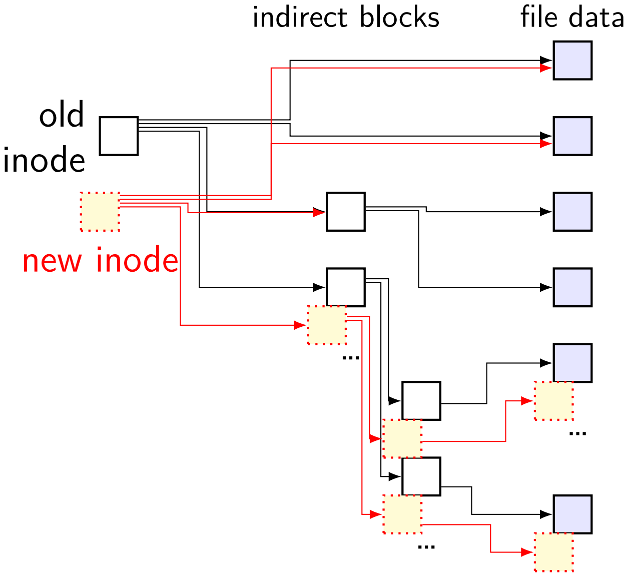
changing file makes new copy of filesystem

common parts shared between versions

inode and copy-on-write



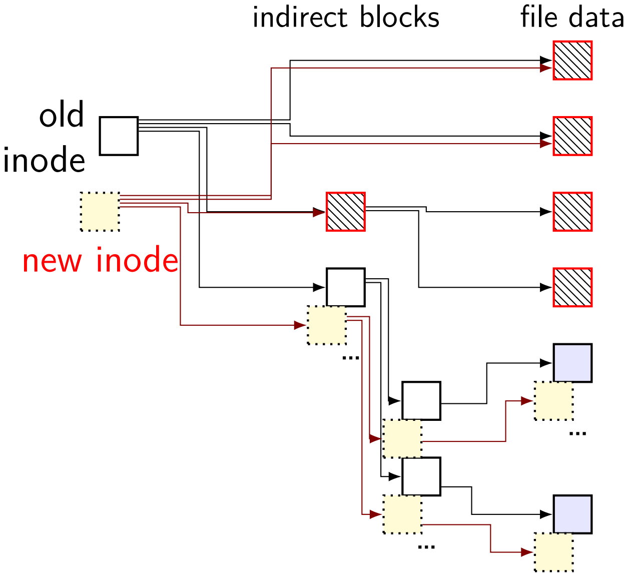
inode and copy-on-write



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

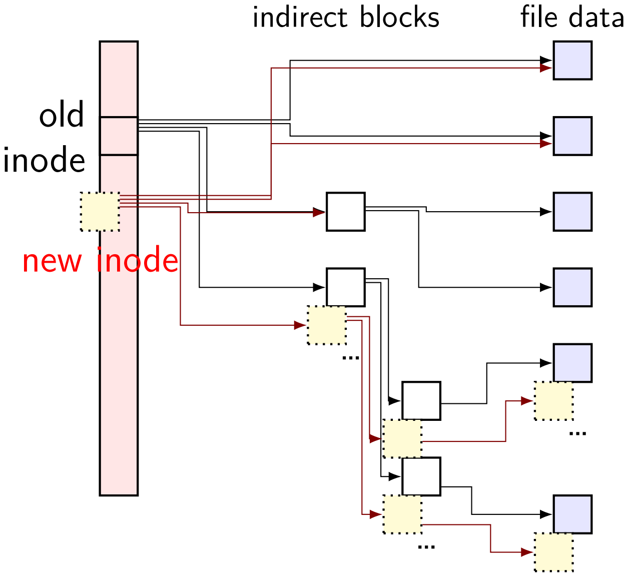
both old+new inode valid

inode and copy-on-write



unchanged parts of file shared

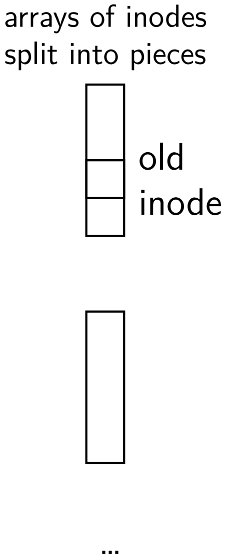
inode and copy-on-write



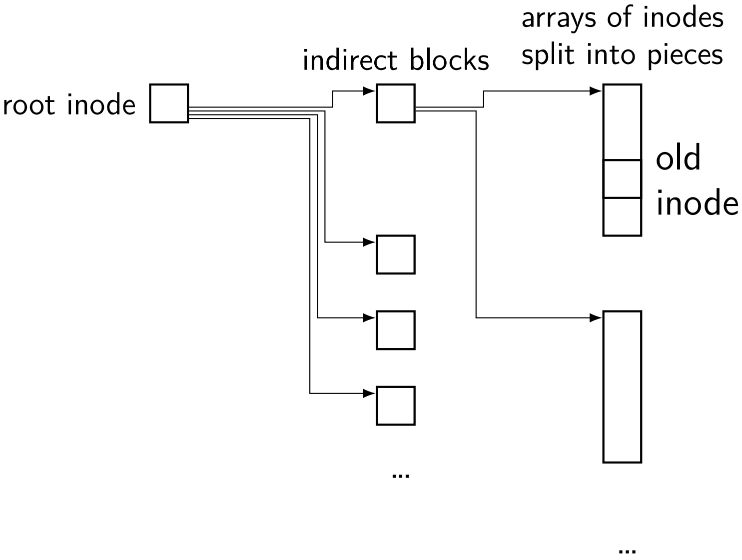
challenge: FFS/xv6/ext2 design has big array of inodes

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

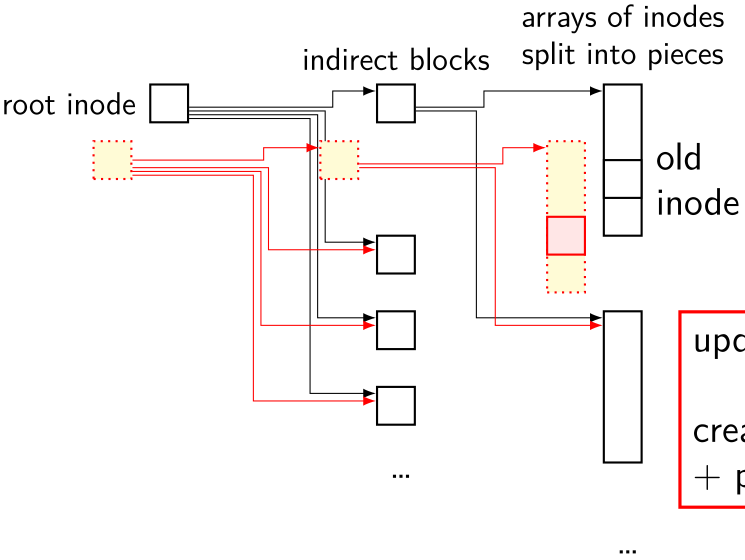
extra indirection for inode array



extra indirection for inode array



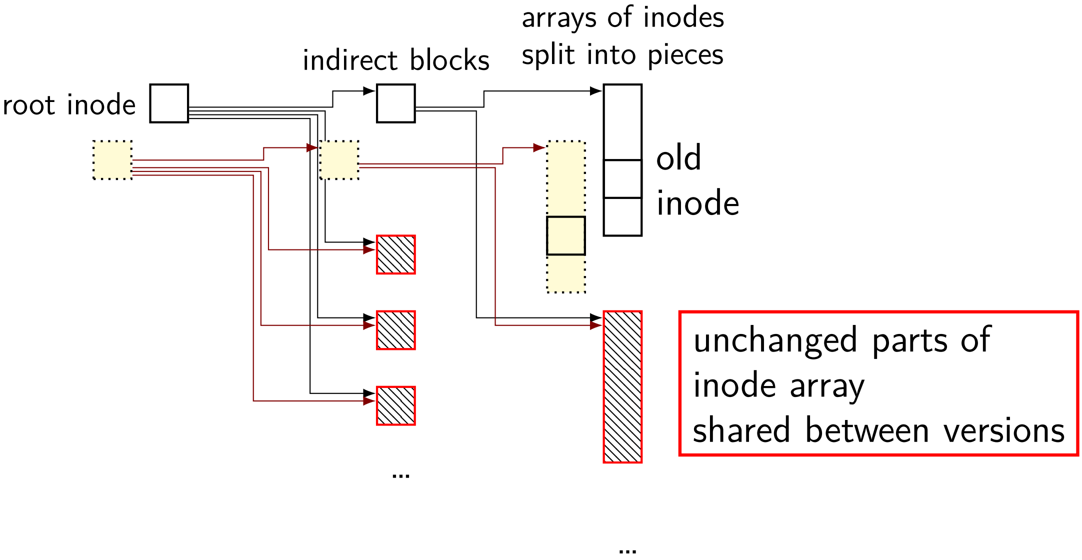
extra indirection for inode array



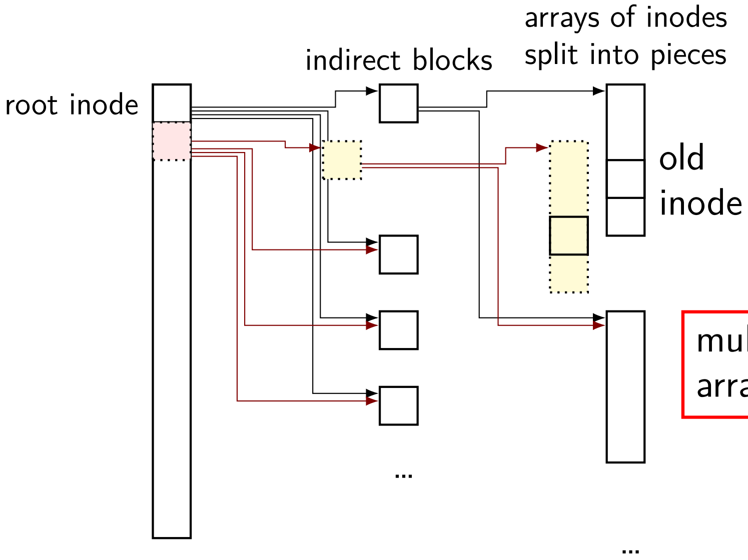
update one inode?

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 (used on department machines) implements this

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

contains contents of files at 11 November 2018 6AM

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.

linux VFS operations

superblock: write_inodez, sync_fs, …

inode: create, link, unlink, mkdir, open …

- most just for inodes which are directories

dentry: compare, delete …

- more commonly argument to inode operation
- can be created for non-yet-existing files

file: read, write, …

linux VFS operations example

```
struct inode_operations {
    struct dentry * (*lookup) (struct inode *,struct dentry *, unsig
    ...
    int (*create) (struct inode *,struct dentry *, umode_t, bool);
    int (*link) (struct dentry *,struct inode *,struct dentry *);
    int (*unlink) (struct inode *,struct dentry *);
    int (*symlink) (struct inode *,struct dentry *,const char *);
    int (*mkdir) (struct inode *,struct dentry *,umode_t);
    int (*rmdir) (struct inode *,struct dentry *);
    int (*mknod) (struct inode *,struct dentry *,umode_t,dev_t);
    int (*rename) (struct inode *, struct dentry *,
                    struct inode *, struct dentry *, unsigned int);
    ...
    int (*update_time)(struct inode *, struct timespec64 *, int);
    int (*atomic_open)(struct inode *, struct dentry *,
                       struct file *, unsigned open_flag,
                       umode_t create_mode);
    ..
}
```

FS abstractions and awkward FSes

example: inode object for FAT?

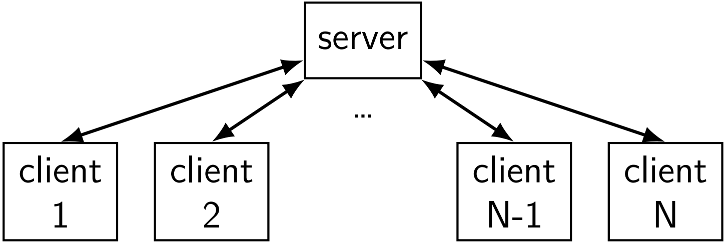
- fake it: point to directory entry?

distributed systems

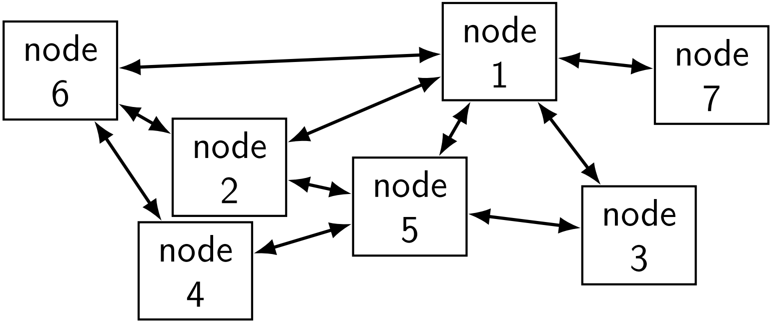
multiple machines working together to perform a single task

called a *distributed system*

some distibuted systems models

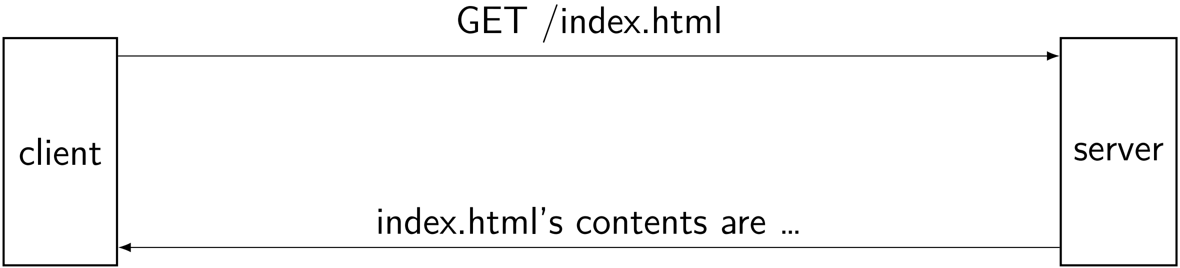


client/server

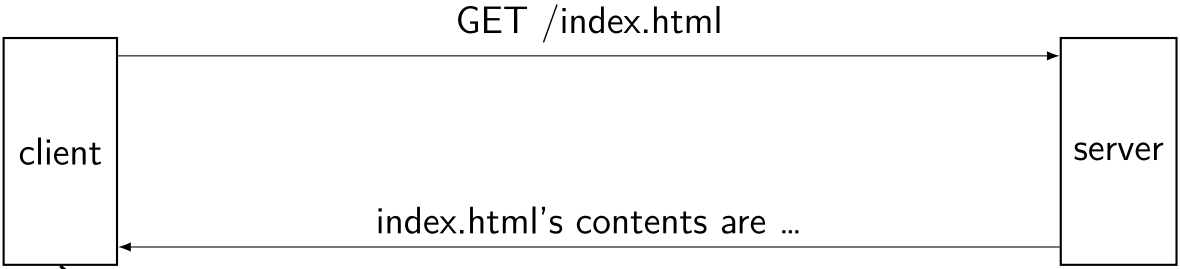


peer-to-peer

client/server model



client/server model

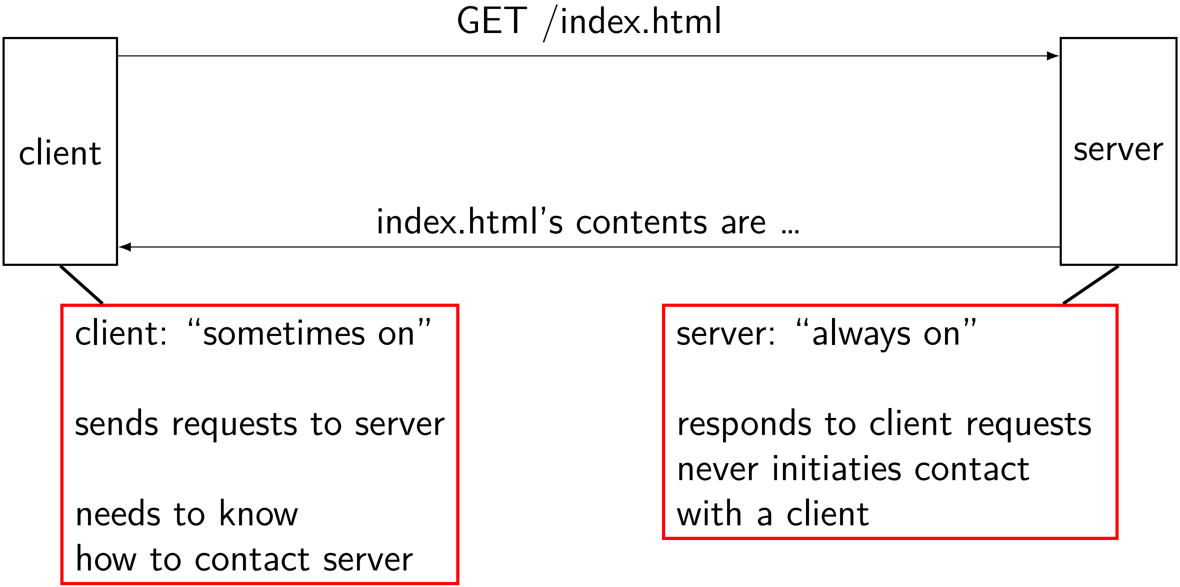


client: "sometimes on"

sends requests to server

needs to know
how to contact server

client/server model



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

distributed system reasons

functional reasons:

multiple people collaborating

delegating responsiblities to another person/company

"the cloud"

distributed system reasons

functional reasons:

multiple people collaborating

delegating responsiblities to another person/company

"the cloud"

performance/reliability/cost reasons:

combine many cheap machines to replace expensive machine

easier to add incrementally

redundancy — one machine can fail and others still work?

transparency goal

common goal of distributed systems is *transparency*

normal user doesn't notice that it's distributed

except because of the extra features that provides

hopefully acts like better single-node system

transparency goal

common goal of distributed systems is *transparency*

normal user doesn't notice that it's distributed

- except because of the extra features that provides

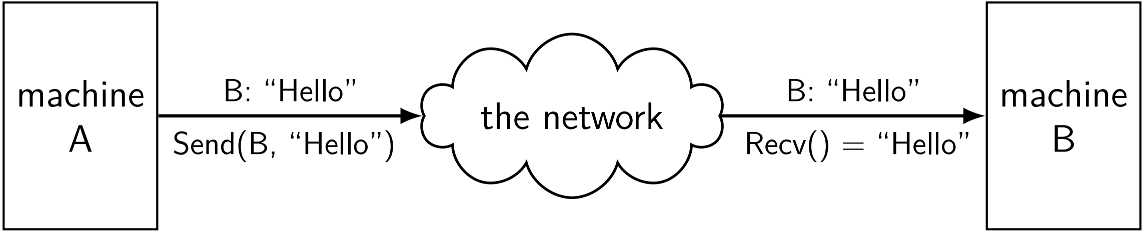
hopefully acts like better single-node system

hope: user can rely on system to

- figure out which machines to use
- handle failures
- ...

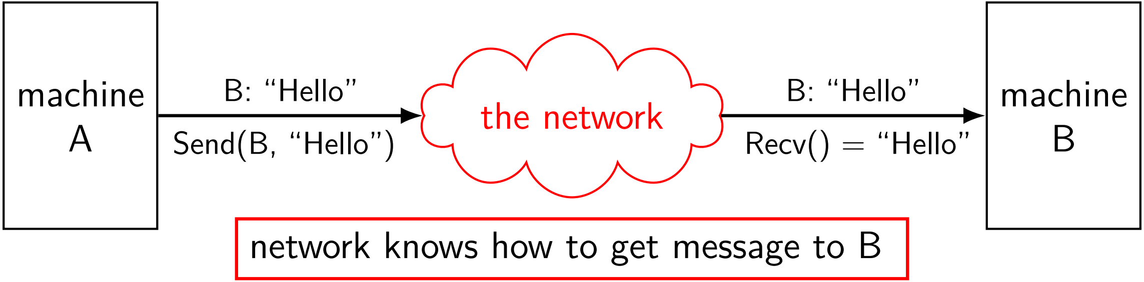
mailbox model

mailbox abstraction: send/receive messages



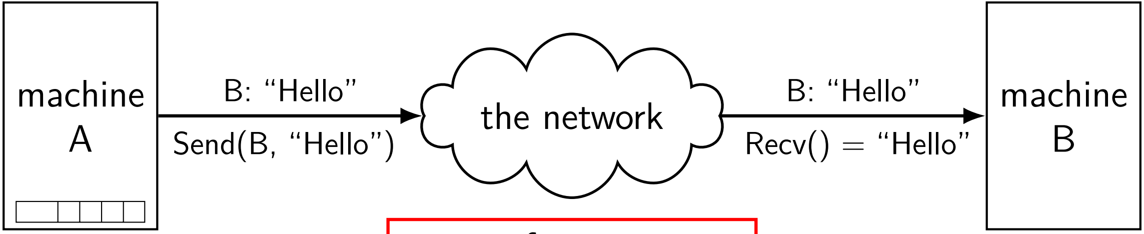
mailbox model

mailbox abstraction: send/receive messages



mailbox model

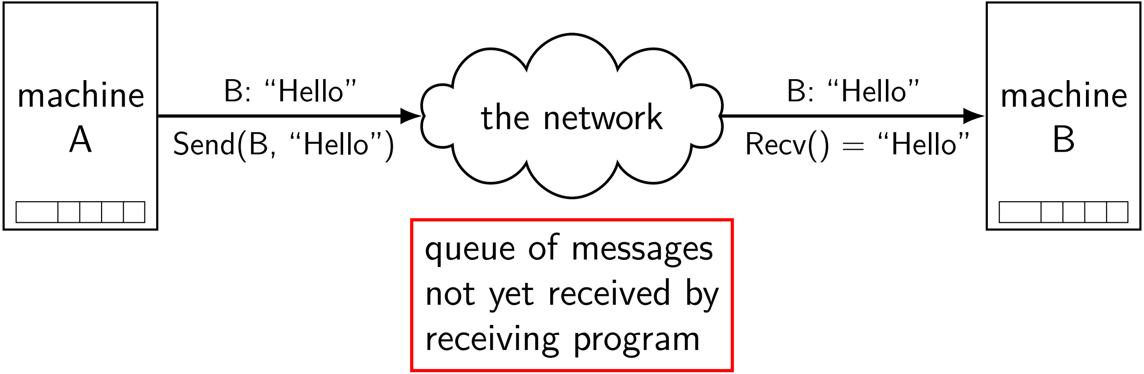
mailbox abstraction: send/receive messages



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

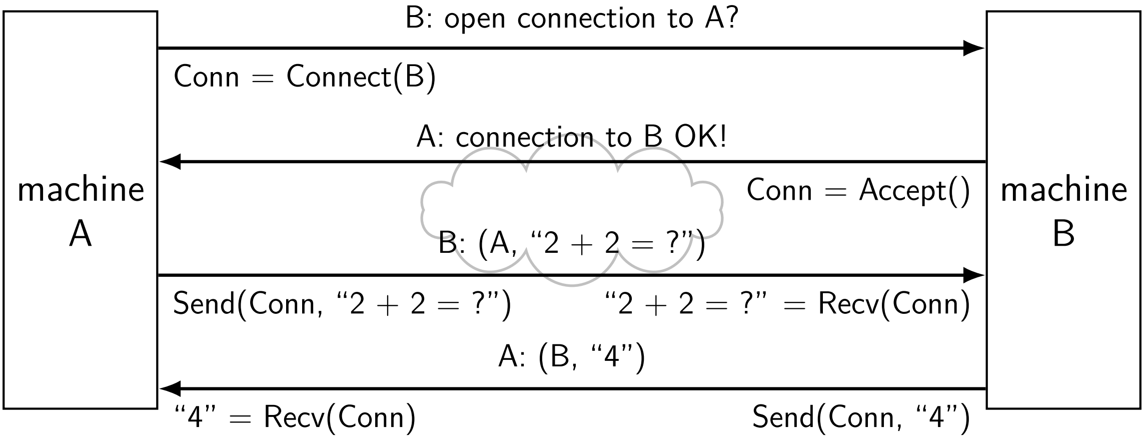
- send a 'return address'
- need to track related messages

common abstraction that does this: the connection

extension: conections

connections: two-way channel for messages

extra operations: connect, accept



connections over mailboxes

real Internet: mailbox-style communication

connections implemented on top this

including handling errors, transmitting more data than fits in message, ...

full details: take networking

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()

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

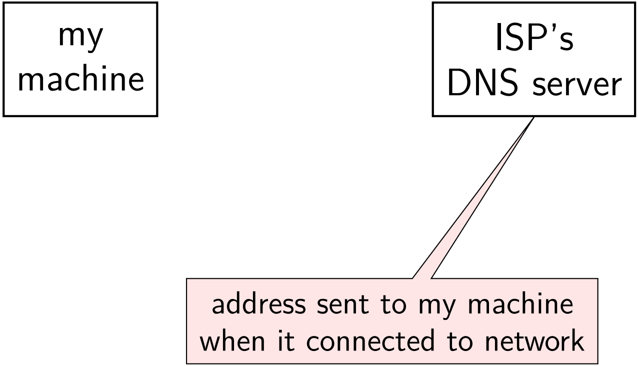
maps logical names like `www.virginia.edu`

- chosen for humans
- hierarchy of names

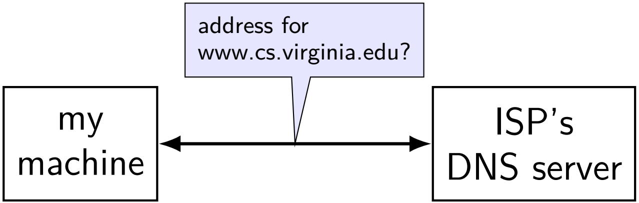
...to *addresses* the network can use to move messages

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

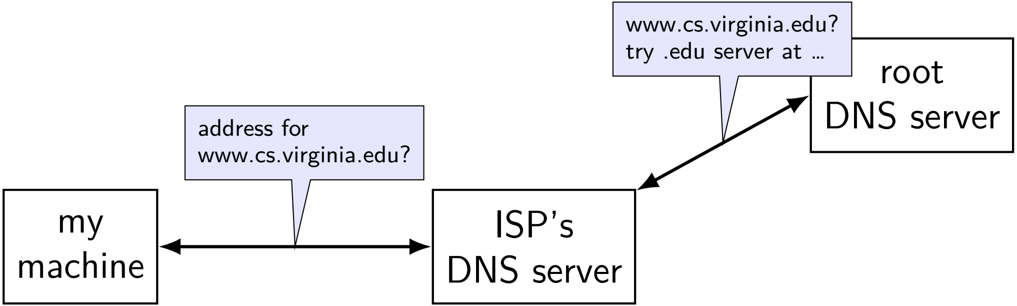
DNS: distributed database



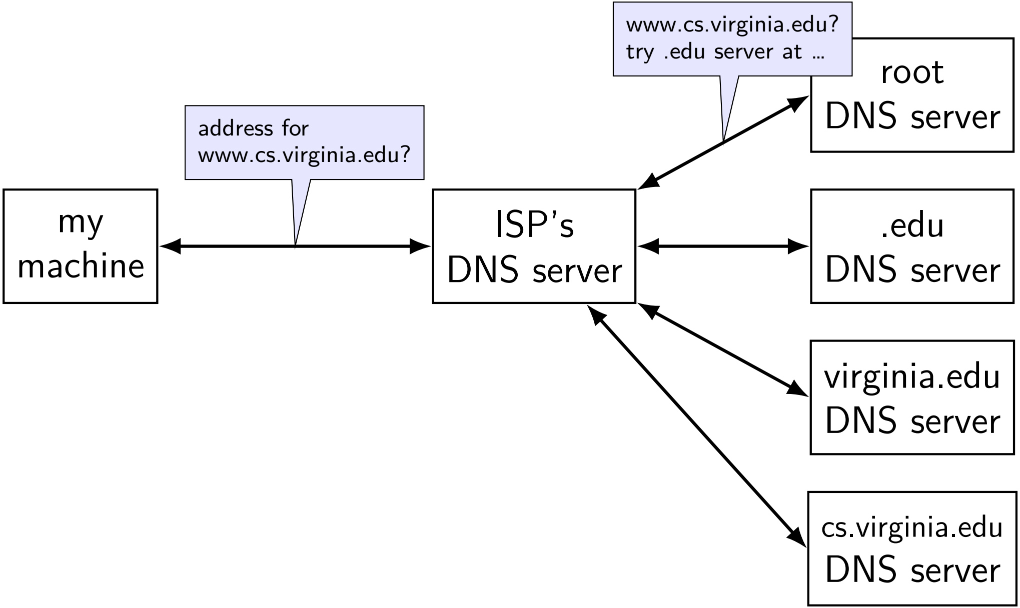
DNS: distributed database



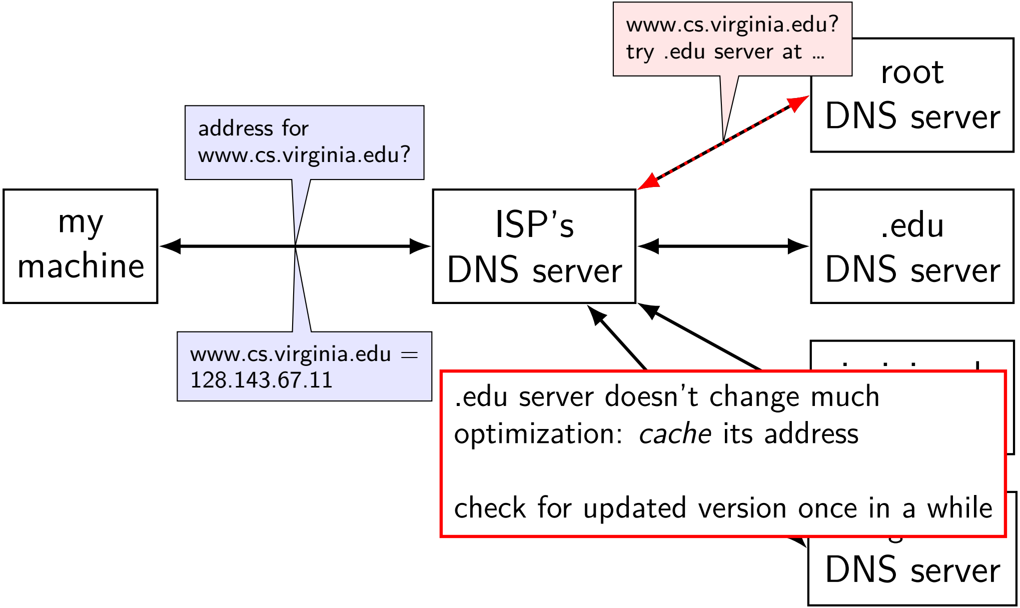
DNS: distributed database



DNS: distributed database



DNS: distributed database



IPv4 addresses

32-bit numbers

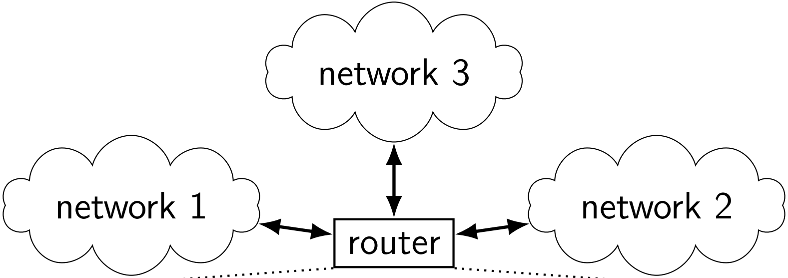
typically written like 128.143.67.11

- 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

IPv4 addresses and routing tables



if I receive data for...	send it to...
128.143.0.0—128.143.255.255	network 1
192.107.102.0–192.107.102.255	network 1
...	...
4.0.0.0–7.255.255.255	network 2
64.8.0.0–64.15.255.255	network 2
...	...
anything else	network 3

selected special IPv4 addresses

127.0.0.0 — 127.255.255.255 — localhost
- AKA loopback
- the machine we're on
- typically only 127.0.0.1 is used

192.168.0.0–192.168.255.255 and
10.0.0.0–10.255.255.255 and
172.16.0.0–172.31.255.255
- "private" IP addresses
- not used on the Internet
- commonly connected to Internet with network address translation
- also 100.64.0.0–100.127.255.255 (but with restrictions)

169.254.0.0-169.254.255.255
- link-local addresses — 'never' forwarded by routers

network address translation

IPv4 addresses are kinda scarce

solution: *convert* many private addrs. to one public addr.

locally: use private IP addresses for machines

outside: private IP addresses become a single public one

commonly how home networks work (and some ISPs)

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`

$\texttt{2607f8b0400d0c0000000000000006a}_{\text{SIXTEEN}}$

selected special IPv6 addresses

`::1` = localhost

anything starting with `fe80` = link-local addresses

never forwarded by routers

port numbers

we run multiple programs on a machine

- IP addresses identifying machine — not enough

port numbers

we run multiple programs on a machine

- IP addresses identifying machine — not enough

so, add 16-bit *port numbers*

- think: multiple PO boxes at address

port numbers

we run multiple programs on a machine

- IP addresses identifying machine — not enough

so, add 16-bit *port numbers*

- think: multiple PO boxes at address

0–49151: typically assigned for particular services

- 80 = http, 443 = https, 22 = ssh, ...

49152–65535: allocated on demand

- default "return address" for client connecting to server

protocols

protocol = agreement on how to comunicate

sytnax (format of messages, etc.)

semantics (meaning of messages — actions to take, etc.)

human protocol: telephone

caller: pick up phone caller: check for service caller: dial caller: wait for ringing	
	callee: “Hello?”
caller: “Hi, it’s Casey...”	
	callee: “Hi, so how about ...”
caller: “Sure, ...”	
...	...
	callee: “Bye!”
caller: “Bye!”	
hang up	hang up

layered protocols

IP: protocol for sending data by IP addresses

- mailbox model
- limited message size

UDP: send datagrams built on IP

- still mailbox model, but *with port numbers*

TCP: reliable connections built on IP

- adds port numbers
- adds resending data if error occurs
- splits big amounts of data into many messages

HTTP: protocol for sending files, etc. built on TCP

other notable protocols (transport layer)

TLS: Transport Layer Security — built on TCP

like TCP, but adds encryption + authentication

SSH: secure shell (remote login) — built on TCP

SCP/SFTP: secure copy/secure file transfer — built on SSH

HTTPS: HTTP, but over TLS instead of TCP

FTP: file transfer protocol

…

other notable protocols (transport layer)

TLS: Transport Layer Security — built on TCP

like TCP, but adds encryption + authentication

SSH: secure shell (remote login) — built on TCP

SCP/SFTP: secure copy/secure file transfer — built on SSH

HTTPS: HTTP, but over TLS instead of TCP

FTP: file transfer protocol

…

sockets

socket: POSIX abstraction of network I/O queue

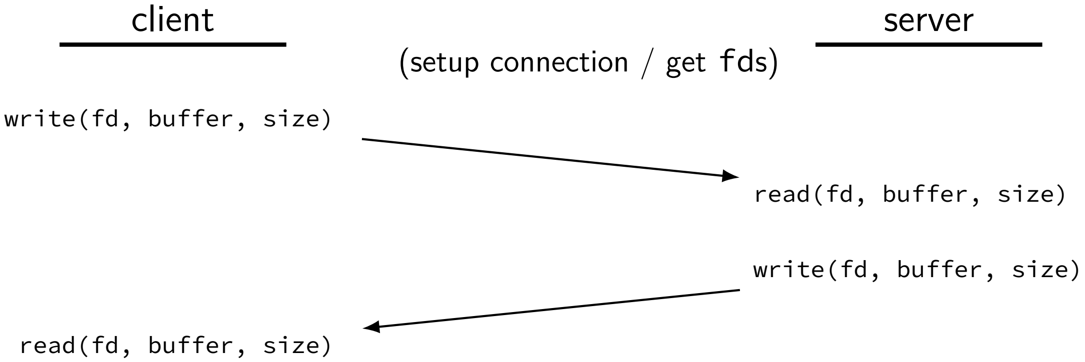
- any kind of network
- can also be used between processes on same machine

a kind of file descriptor

connected sockets

sockets can represent a connection

act like bidirectional pipe



echo client/server

```
void client_for_connection(int socket_fd) {
    int n; char send_buf[MAX_SIZE]; char recv_buf[MAX_SIZE];
    while (prompt_for_input(send_buf, MAX_SIZE)) {
        n = write(socket_fd, send_buf, strlen(send_buf));
        if (n != strlen(send_buf)) {...error?...}
        n = read(socket_fd, recv_buf, MAX_SIZE);
        if (n <= 0) return; // error or EOF
        write(STDOUT_FILENO, recv_buf, n);
    }
}
```

```
void server_for_connection(int socket_fd) {
    int read_count, write_count; char request_buf[MAX_SIZE];
    while (1) {
        read_count = read(socket_fd, request_buf, MAXSIZE);
        if (read_count <= 0) return; // error or EOF
        write_count = write(socket_fd, request_buf, read_count);
        if (read_count != write_count) {...error?...}
    }
}
```

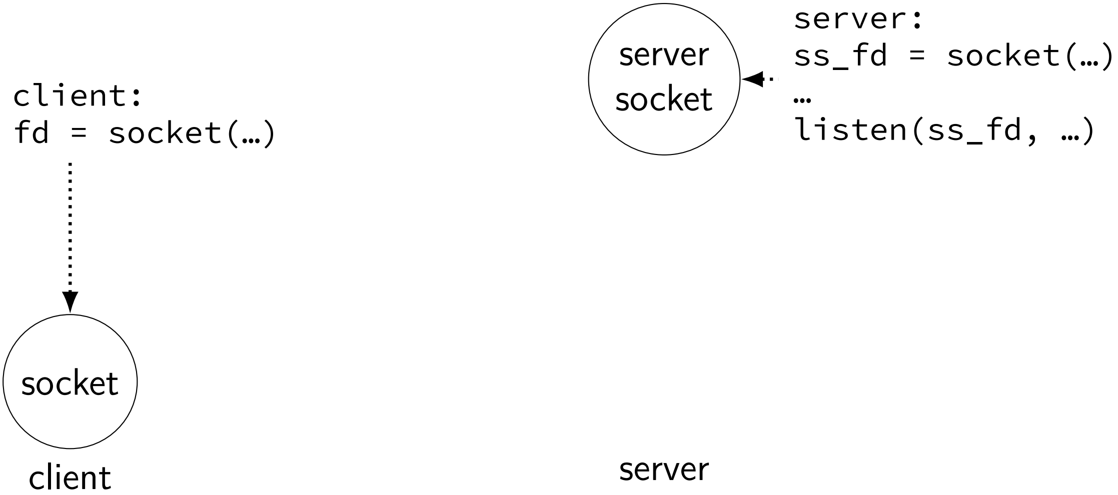
aside: send/recv

sockets have some alternate read/write-like functions:

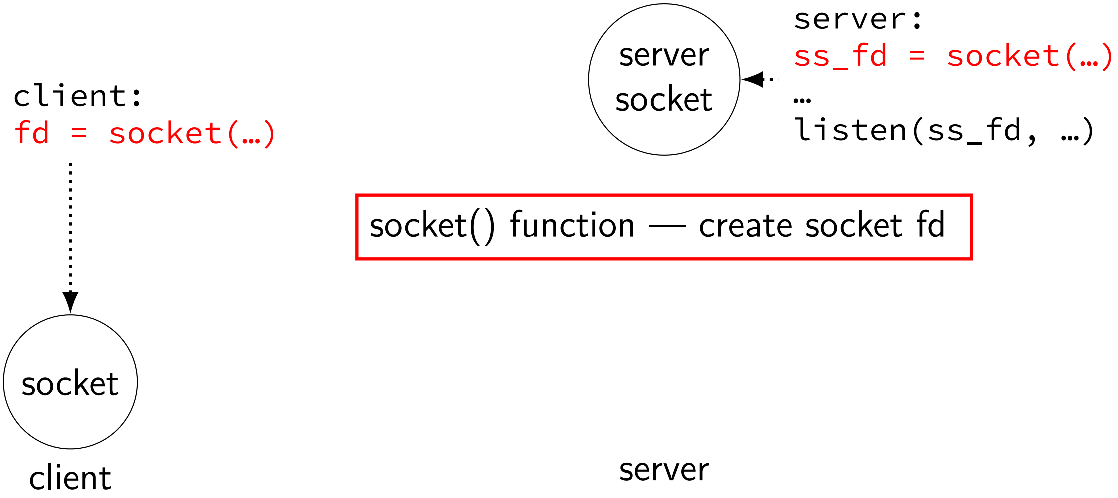
- recv, recvfrom, recvmsg
- send, sendmsg

have some additional options we won't need in this class

sockets and server sockets

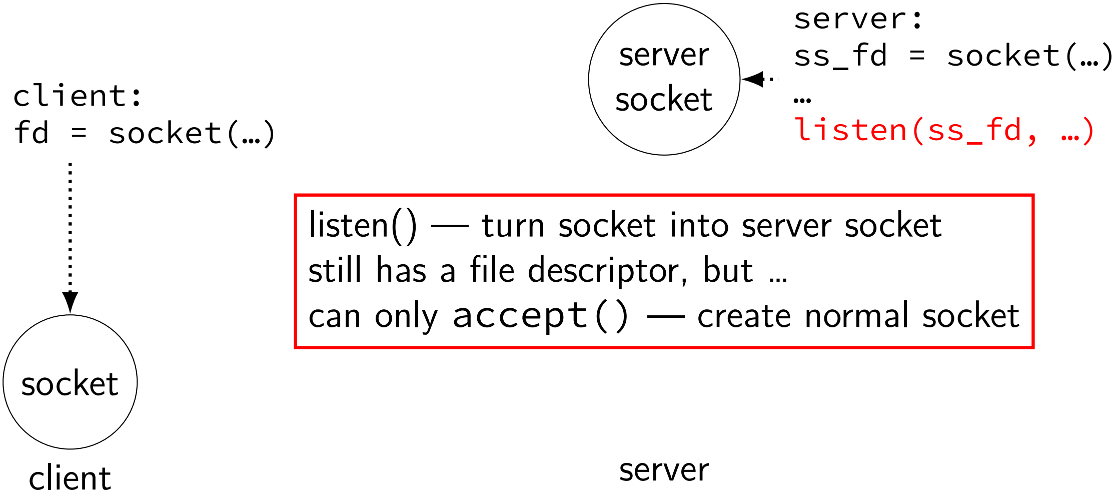


sockets and server sockets



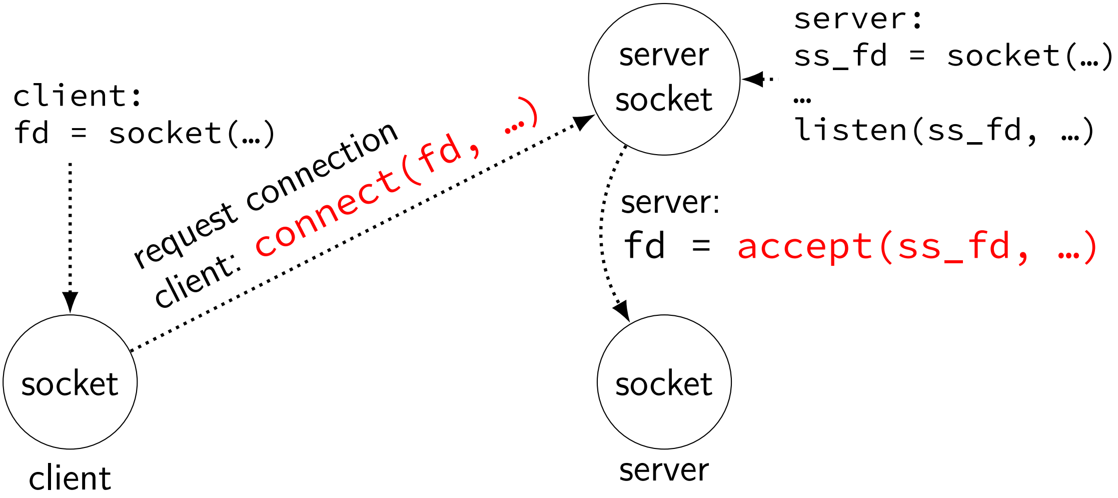
socket() function — create socket fd

sockets and server sockets

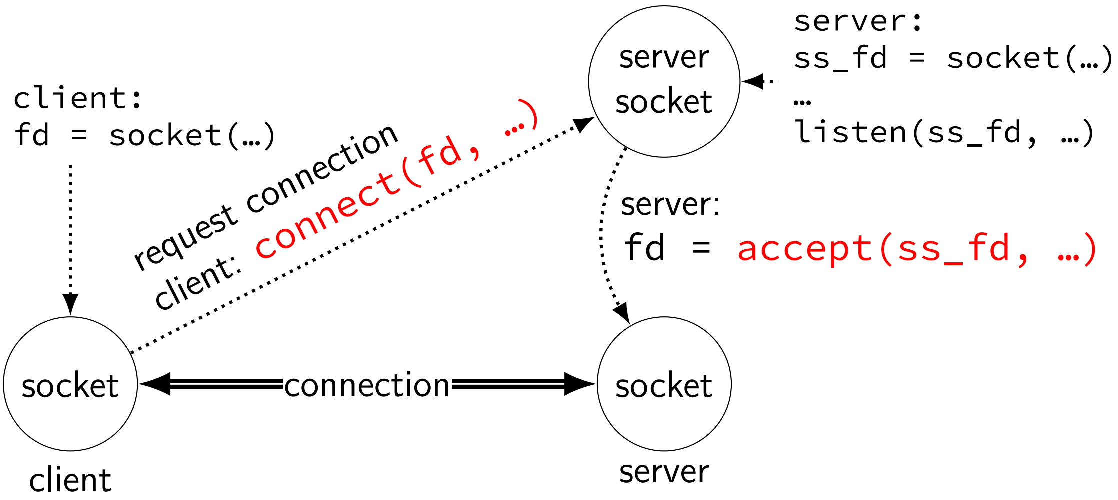


listen() — turn socket into server socket
still has a file descriptor, but …
can only `accept()` — create normal socket

sockets and server sockets



sockets and server sockets



connections in TCP/IP

connection identified by *5-tuple*

(protocol=TCP, local IP addr., local port, remote IP addr., remote port)

- how messages are tagged on the network
- (other notable protocol value: UDP)

both ends always have an address+port

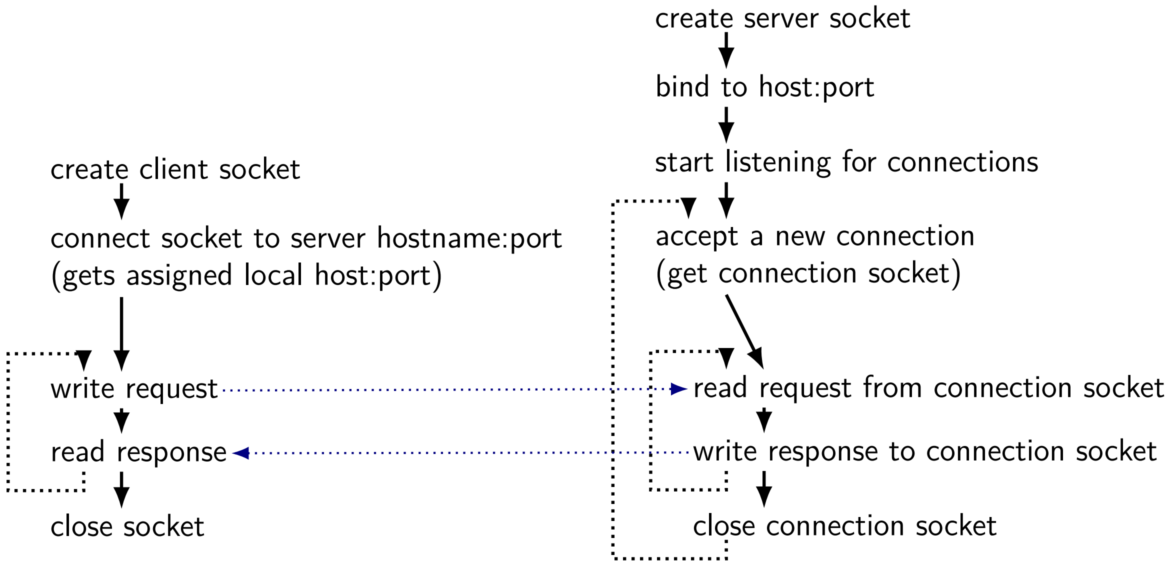
what is the IP address, port number? set with `bind()` function

- *typically* always done for servers, not done for clients
- system will choose default if you don't

connections on my desktop

```
cr4bd@reiss-t3620
: /zf14/cr4bd ; netstat --inet --inet6 --numeric
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address
State
tcp        0      0 128.143.67.91:49202     128.143.63.34:22
ESTABLISHED
tcp        0      0 128.143.67.91:803       128.143.67.236:2049
ESTABLISHED
tcp        0      0 128.143.67.91:50292     128.143.67.226:22
TIME_WAIT
tcp        0      0 128.143.67.91:54722     128.143.67.236:2049
TIME_WAIT
tcp        0      0 128.143.67.91:52002     128.143.67.236:111
TIME_WAIT
tcp        0      0 128.143.67.91:732       128.143.67.236:63439
TIME_WAIT
tcp        0      0 128.143.67.91:40664     128.143.67.236:2049
TIME_WAIT
```

client/server flow (one connection at a time)



connection setup: client — manual addresses

```
int sock_fd;

server = /* code on later slide */;
sock_fd = socket(AF_INET, /* IPv4 */
                 SOCK_STREAM, /* byte-oriented */
                 IPPROTO_TCP);

if (sock_fd < 0) { /* handle error */ }

struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = htonl(2156872459); /* 128.143.67.11 */
addr.sin_port = htons(80); /* port 80 */
if (connect(sock_fd, (struct sockaddr*) &addr, sizeof(addr)) {
    /* handle error */
}
DoClientStuff(sock_fd); /* read and write from sock_fd */
close(sock_fd);
```

connection setup: client — manual addresses

```
int sock_fd;

server = /* code on later slide */;
sock_fd = socket(AF_INET, /* IPv4 */
                 SOCK_STREAM, /* byte-oriented */
                 IPPROTO_TCP);

if

str
add
addr.sin_addr.s_addr = htonl(2156872459); /* 128.143.67.11 */
addr.sin_port = htons(80); /* port 80 */
if (connect(sock_fd, (struct sockaddr*) &addr, sizeof(addr)) {
    /* handle error */
}
DoClientStuff(sock_fd); /* read and write from sock_fd */
close(sock_fd);
```

specify IPv4 instead of IPv6 or local-only sockets

specify TCP (byte-oriented) instead of UDP (‘datagram’ oriented)

connection setup: client — manual addresses

htonl/s = host-to-network long/short
network byte order = big endian

```
int sock_fd;

server = /* code
sock_fd = socket(AF_INET, /* IPv4 */
                 SOCK_STREAM, /* byte-oriented */
                 IPPROTO_TCP);

if (sock_fd < 0) { /* handle error */ }

struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = htonl(2156872459); /* 128.143.67.11 */
addr.sin_port = htons(80); /* port 80 */
if (connect(sock_fd, (struct sockaddr*) &addr, sizeof(addr)) {
    /* handle error */
}
DoClientStuff(sock_fd); /* read and write from sock_fd */
close(sock_fd);
```

connection setup: client — manual addresses

```
int sock_fd

server = /*
sock_fd = s

              IPPROTO_TCP);

if (sock_fd < 0) { /* handle error */ }

struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = htonl(2156872459); /* 128.143.67.11 */
addr.sin_port = htons(80); /* port 80 */
if (connect(sock_fd, (struct sockaddr*) &addr, sizeof(addr)) {
    /* handle error */
}
DoClientStuff(sock_fd); /* read and write from sock_fd */
close(sock_fd);
```

struct representing IPv4 address + port number
declared in `<netinet/in.h>`
see `man 7 ip` on Linux for docs

sockaddr_in

```
/* from 'man 7 ip' */
struct sockaddr_in {
   sa_family_t    sin_family; /* address family: AF_INET */
   in_port_t      sin_port;   /* port in network byte order */
   struct in_addr sin_addr;   /* internet address */
};

/* Internet address. */
struct in_addr {
   uint32_t       s_addr;     /* address in network byte order */
};
```

sockaddr_in6

```
/* from 'man 7 ipv6' */
struct sockaddr_in6 {
    sa_family_t     sin6_family;   /* AF_INET6 */
    in_port_t       sin6_port;     /* port number */
    uint32_t        sin6_flowinfo; /* IPv6 flow information */
    struct in6_addr sin6_addr;     /* IPv6 address */
    uint32_t        sin6_scope_id; /* Scope ID (new in 2.4) */
};

struct in6_addr {
    unsigned char   s6_addr[16];   /* IPv6 address */
};
```

connection setup: client, using addrinfo

```
int sock_fd;
struct addrinfo *server = /* code on next slide */;

sock_fd = socket(server->ai_family,
      // ai_family = AF_INET (IPv4) or AF_INET6 (IPv6) or ...
                 server->ai_socktype,
      // ai_socktype = SOCK_STREAM (bytes) or ...
                 server->ai_prototcol
      // ai_protocol = IPPROTO_TCP or ...
                );

if (sock_fd < 0) { /* handle error */ }
if (connect(sock_fd, server->ai_addr, server->ai_addrlen) < 0) {
     /* handle error */
}
freeaddrinfo(server);
DoClientStuff(sock_fd); /* read and write from sock_fd */
close(sock_fd);
```

connection setup: client, using addrinfo

```
int sock_fd;
struct addrinfo *server = /* code on next slide */;

sock_fd = socket(server->ai_family,
      // ai_family = AF_INET (IPv4) or AF_INET6 (IPv6) or ...
                 server->ai_socktype,
      // ai_socktype = SOCK_STREAM (bytes) or ...
      //

if (soc
if (con                                                        0) {
    /* handle error */
}
freeaddrinfo(server);
DoClientStuff(sock_fd); /* read and write from sock_fd */
close(sock_fd);
```

addrinfo contains all information needed to setup socket

set by `getaddrinfo` function (next slide)

handles IPv4 and IPv6

handles DNS names, service names

connection setup: client, using addrinfo

```
int sock_fd;
struct addrinfo *server = /* code on next slide */;

sock_fd = socket(server->ai_family,
     // ai_family = AF_INET (IPv4) or AF_INET6 (IPv6) or ...
                 server->ai_socktype,
     // ai_socktype = SOCK_STREAM (bytes) or ...
                 server->ai_prototcol
     // ai_protocol = IPPROTO_TCP or ...
                );

if (sock_fd < 0) { /* handle error */ }
if (connect(sock_fd, server->ai_addr, server->ai_addrlen) < 0) {
    /* handle error */
}
freeaddrinfo(server);
DoClientStuff(sock_fd); /* read and write from sock_fd */
close(sock_fd);
```

connection setup: client, using addrinfo

```
int sock_f
struct add

sock_fd =
     // ai                                                    ...
                 server->ai_socktype,
     // ai_socktype = SOCK_STREAM (bytes) or ...
                 server->ai_prototcol
     // ai_protocol = IPPROTO_TCP or ...
                );

if (sock_fd < 0) { /* handle error */ }
if (connect(sock_fd, server->ai_addr, server->ai_addrlen) < 0) {
    /* handle error */
}
freeaddrinfo(server);
DoClientStuff(sock_fd); /* read and write from sock_fd */
close(sock_fd);
```

ai_addr points to a `struct sockaddr_in*` or a `struct sockaddr_in6*` (cast to a `struct sockaddr*`)

connection setup: client, using addrinfo

since addrinfo contains pointers to dynamically allocated memory, call this function to free everything

```
int
str

sock_fd = socket(server->ai_family,
     // ai_family = AF_INET (IPv4) or AF_INET6 (IPv6) or ...
                  server->ai_socktype,
     // ai_socktype = SOCK_STREAM (bytes) or ...
                  server->ai_prototcol
     // ai_protocol = IPPROTO_TCP or ...
                 );

if (sock_fd < 0) { /* handle error */ }
if (connect(sock_fd, server->ai_addr, server->ai_addrlen) < 0) {
    /* handle error */
}
freeaddrinfo(server);
DoClientStuff(sock_fd); /* read and write from sock_fd */
close(sock_fd);
```

connection setup: lookup address

```
/* example hostname, portname = "www.cs.virginia.edu", "443" */
const char *hostname; const char *portname;
...
struct addrinfo *server;
struct addrinfo hints;
int rv;
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_UNSPEC; /* for IPv4 OR IPv6 */
// hints.ai_family = AF_INET4; /* for IPv4 only */

hints.ai_socktype = SOCK_STREAM; /* byte-oriented --- TCP */
rv = getaddrinfo(hostname, portname, &hints, &server);
if (rv != 0) { /* handle error */ }

/* eventually freeaddrinfo(result) */
```

connection setup: lookup address

```
/* example hostname, portname = "www.cs.virginia.edu", "443" */
const char *hostname; const char *portname;
...
struct addrinfo *server;
struct addrinfo hints;
int rv;
memset(&hints, 0, sizeof(hints));
hints.ai_family
// hints.ai_fam

hints.ai_socktype = SOCK_STREAM; /* byte-oriented --- TCP */
rv = getaddrinfo(hostname, portname, &hints, &server);
if (rv != 0) { /* handle error */ }

/* eventually freeaddrinfo(result) */
```

NB: pass pointer to addrinfo to fill in

connection setup: lookup address

AF_UNSPEC: choose between IPv4 and IPv6 for me

AF_INET, AF_INET6: choose IPv4 or IPV6 respectively

```
/* exam                                                   " */
const c
...
struct addrinfo *server;
struct addrinfo hints;
int rv;
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_UNSPEC; /* for IPv4 OR IPv6 */
// hints.ai_family = AF_INET4; /* for IPv4 only */

hints.ai_socktype = SOCK_STREAM; /* byte-oriented --- TCP */
rv = getaddrinfo(hostname, portname, &hints, &server);
if (rv != 0) { /* handle error */ }

/* eventually freeaddrinfo(result) */
```

connection setup: multiple server addresses

```
struct addrinfo *server;
...
rv = getaddrinfo(hostname, portname, &hints, &server);
if (rv != 0) { /* handle error */ }

for (struct addrinfo *current = server; current != NULL;
      current = current->ai_next) {
    sock_fd = socket(current->ai_family, current->ai_socktype, curre
    if (sock_fd < 0) continue;
    if (connect(sock_fd, current->ai_addr, current->ai_addrlen) == 0
        break;
    }
    close(sock_fd); // connect failed
}
freeaddrinfo(server);
DoClientStuff(sock_fd);
close(sock_fd);
```

connection setup: multiple server addresses

```
struct addrinfo *server;
...
rv = getaddrinfo(hostname, portname, &hints, &server);
if (rv != 0) { /* handle error */ }

for (struct addrinfo *current = server; current != NULL;
      current = current->ai_next) {
    sock_fd = socket(current->ai_family, current->ai_socktype, curre
    if (sock_fd < 0) continue;
    if (connect(sock_fd, current->ai_addr, current->ai_addrlen) == 0
        break;
    }
    clos
}
freeaddr
DoClient
close(so
```

addrinfo is a linked list

name can correspond to multiple addresses

example: redundant copies of web server

example: an IPv4 address and IPv6 address

example: wired + wireless connection on one machine

connection setup: old lookup function

```
/* example hostname, portnum= "www.cs.virginia.edu", 443*/
const char *hostname; int portnum;
...
struct hostent *server_ip;
server_ip = gethostbyname(hostname);

if (server_ip == NULL) { /* handle error */ }

struct sockaddr_in addr;
addr.s_addr = *(struct in_addr*) server_ip->h_addr_list[0];
addr.sin_port = htons(portnum);
sock_fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
connect(sock_fd, &addr, sizeof(addr));
...
```

connection setup: server, manual

```
int server_socket_fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = INADDR_ANY; /* "any address I can use" */
    /* or: addr.s_addr.in_addr = INADDR_LOOPBACK (127.0.0.1) */
    /* or: addr.s_addr.in_addr = htonl(...); */
addr.sin_port = htons(9999); /* port number 9999 */

if (bind(server_socket_fd, &addr, sizeof(addr)) < 0) {
    /* handle error */
}
listen(server_socket_fd, MAX_NUM_WAITING);
...
int socket_fd = accept(server_socket_fd, NULL);
```

connection setup: server, manual

```
int server_socket_fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = INADDR_ANY; /* "any address I can use" */
    /* or: addr.s_addr.in_addr = INADDR_LOOPBACK (127.0.0.1) */
    /* or: addr.s_addr.in_addr = htonl(...); */
addr.sin_port = htons(9999); /* port number 9999 */

if (bind(server_socket_fd, &addr, sizeof(addr)) < 0) {
    /* handle error */
}
listen
...
int socket_fd = accept(server_socket_fd, NULL);
```

INADDR_ANY: accept connections for any address I can!

alternative: specify specific address

connection setup: server, manual

```
int server_socket_fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = INADDR_ANY; /* "any address I can use" */
    /* or: addr.s_addr.in_addr = INADDR_LOOPBACK (127.0.0.1) */
    /* or: addr.s_addr.in_addr = htonl(...); */
addr.sin_port = htons(9999); /* port number 9999 */

if (bind(server_socket_fd, &addr, sizeof(addr)) < 0) {
    /* handle error */
}
list
...
int socket_fd = accept(server_socket_fd, NULL);
```

bind to 127.0.0.1? only accept connections from same machine
what we recommend for FTP server assignment

connection setup: server, manual

```
int server_socket_fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = INADDR_ANY; /* "any address I can use" */
    /* or: addr.s_addr.in_addr = INADDR_LOOPBACK (127.0.0.1) */
    /* or: addr.s_addr.in_addr = htonl(...); */
addr.sin_port = htons(9999); /* port number 9999 */

if (bind(server_socket_fd, &addr, sizeof(addr)) < 0) {
    /* handle error */
}
listen(server_socket_fd, MAX_NUM_WAITING);
...
int socket_fd = accept(server_socket_fd, NULL);
```

choose the number of unaccepted connections

aside: on server port numbers

Unix convention: must be `root` to use ports 0–1023

root = superuser = 'adminstrator user' = what sudo does

so, for testing: probably ports > 1023

connection setup: server, address setup

```
/* example (hostname, portname) = ("127.0.0.1", "443") */
const char *hostname; const char *portname;
...
struct addrinfo *server;
struct addrinfo hints;
int rv;
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET; /* for IPv4 */
/* or: */ hints.ai_family = AF_INET6; /* for IPv6 */
/* or: */ hints.ai_family = AF_UNSPEC; /* I don't care */
hints.ai_flags = AI_PASSIVE;
rv = getaddrinfo(hostname, portname, &hints, &server);
if (rv != 0) { /* handle error */ }
```

connection setup: server, address setup

```
/* example (hostname, portname) = ("127.0.0.1", "443") */
const char *hostname; const char *portname;
...
struct addrinfo *server;
struct addrinfo hints;
int rv;
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET; /* for IPv4 */
/* or: */ hints.ai_family = AF_INET6; /* for IPv6 */
/* or: */ hints.ai                                           care */
hints.ai_flags =
rv = getaddrinfo(                                     ver);
if (rv != 0) { /*
```

hostname can be NULL

means “use all possible addresses”

only makes sense for servers

connection setup: server, address setup

```
/* example (hostname, portname) = ("127.0.0.1", "443") */
const char *hostname; const char *portname;
...
struct addrinfo *server;
struct addrinfo hints;
int rv;
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET; /* for IPv4 */
/* or: */ hints.ai_family = AF_INET6; /* for IPv6 */
/* or: */ hints.ai_family = AF_UNSPEC; /* I don't care */
hints.ai_flags
rv = getaddrinf                                          r);
if (rv != 0) {
```

portname can also be NULL
means "choose a port number for me"
only makes sense for servers

connection setup: server, address setup

AI_PASSIVE: “I’m going to use bind”

```
/* example (hos                                   3") */
const char *hos
...
struct addrinfo *server;
struct addrinfo hints;
int rv;
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET; /* for IPv4 */
/* or: */ hints.ai_family = AF_INET6; /* for IPv6 */
/* or: */ hints.ai_family = AF_UNSPEC; /* I don't care */
hints.ai_flags = AI_PASSIVE;
rv = getaddrinfo(hostname, portname, &hints, &server);
if (rv != 0) { /* handle error */ }
```

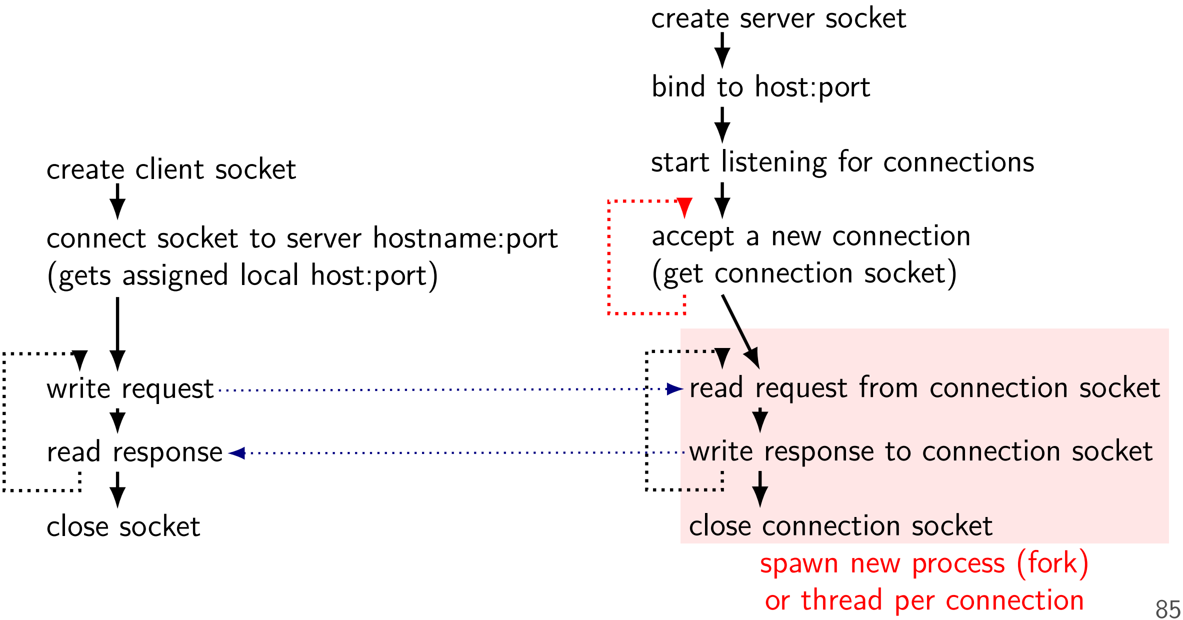
connection setup: server, addrinfo

```
struct addrinfo *server;
... getaddrinfo(...) ...

int server_socket_fd = socket(
    server->ai_family,
    server->ai_sockttype,
    server->ai_protocol);

if (bind(server_socket_fd, ai->ai_addr, ai->ai_addr_len)) < 0) {
    /* handle error */
}
listen(server_socket_fd, MAX_NUM_WAITING);
...
int socket_fd = accept(server_socket_fd, NULL);
```

client/server flow (multiple connections)



incomplete writes

`write` might write less than requested

error, or buffer full

`read` might read less than requested

error, or didn't get there in time

handling incomplete writes

```
bool write_fully(int fd, const char *buffer, ssize_t count) {
    const char *ptr = buffer;
    const char *end = buffer + count;
    while (ptr != end) {
        ssize_t written = write(fd, (void*) ptr, end - ptr);
        if (written == -1) {
            return false;
        }
        ptr += written;
    }
    return true;
}
```

on filling buffers

```
char buffer[SIZE];
ssize_t buffer_end;

int fill_buffer(int fd) {
    ssize_t amount = read(
        fd, buffer + buffer_end, SIZE - buffer_end
    );
    if (amount == 0) {
        /* handle EOF */ ???
    } else if (amount == -1) {
        return -1;
    } else {
        buffer_end += amount;
    }
}
```

reading lines

```
int read_line(int fd, const char *p_line, size_t *p_size) {
    const char *newline;
    while (1) {
        newline = memchr(buffer, '\n', buffer_end);
        if (newline != NULL || buffer_end == SIZE) break;
        fill_buffer();
    }
    memcpy(p_line, buffer, newline - buffer);
    *p_size = newline - buffer;
    memmove(newline, buffer, buffer + SIZE - newline);
    buffer_end -= (newline - buffer);
}
```

aside: getting addresses

on a socket fd: `getsockname` = local addresss

- `sockaddr_in` or `sockaddr_in6`
- IPv4/6 address + port

on a socket fd: `getpeername` = remote address

addresses to string

can access numbers/arrays in `sockaddr_in/in6` directly

another option: `getnameinfo`

- supports getting W.X.Y.Z form or looking up a hostname

example echo client/server

handle reporting errors from incomplete writes

handle avoiding SIGPIPE

- OS kills program trying to write to closed socket/pipe

set the SO_REUSEADDR "socket option"

- default: OS reserves port number for a while after server exits
- this allows keeps it unreserved
- allows us to bind() immediately after closing server

client handles reading until a newline

- but doesn't check for reading multiple lines at once

example echo client/server

handle reporting errors from incomplete writes

handle avoiding SIGPIPE

- OS kills program trying to write to closed socket/pipe

set the SO_REUSEADDR "socket option"

- default: OS reserves port number for a while after server exits
- this allows keeps it unreserved
- allows us to bind() immediately after closing server

client handles reading until a newline

- but doesn't check for reading multiple lines at once

reading and writing at once

so far assumption: alternate between reading+writing

- sufficient for FTP assignment
- how many protocols work

“half-duplex”

don’t have to use sockets this way, but tricky

threads: one reading thread, one writing thread *OR*

event-loop: use *non-blocking I/O* and select()/poll()/etc. functions

- non-blocking I/O setup with fcntl() function
- non-blocking write() fills up buffer as much as possible, then returns
- non-blocking read() returns what’s in buffer, never waits for more

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

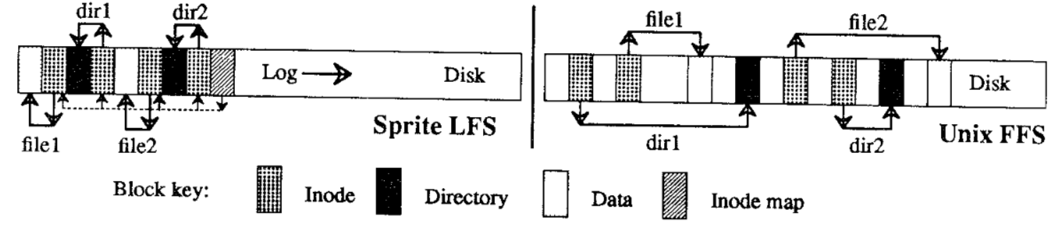


image: Rosenblum and Ousterhout, “The Design and Implementatoin of a Log Structures Filesystem”

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:

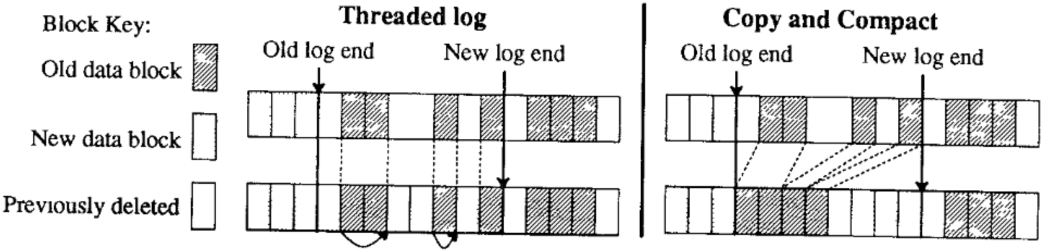


image: Rosenblum and Ousterhout, "The Design and Implementatoin of a Log Structures Filesystem"

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

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

aside: fsync

so, filesystem can order things carefully

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???

beyond threads: event based programming

writing server that servers multiple clients?

- e.g. multiple web browsers at a time

maybe don't really need multiple processors/cores

- one network, not that fast

idea: one thread handles multiple connections

beyond threads: event based programming

writing server that servers multiple clients?

- e.g. multiple web browsers at a time

maybe don't really need multiple processors/cores

- one network, not that fast

idea: one thread handles multiple connections

issue: read from/write to multiple streams at once?

event loops

```
while (true) {
    event = WaitForNextEvent();
    switch (event.type) {
    case NEW_CONNECTION:
        handleNewConnection(event); break;
    case CAN_READ_DATA_WITHOUT_WAITING:
        connection = LookupConnection(event.fd);
        handleRead(connection);
        break;
    case CAN_WRITE_DATA_WITHOUT_WAITING:
        connection = LookupConnection(event.fd);
        handleWrite(connection);
        break;
        ...
    }
}
```

some single-threaded processing code

```
void ProcessRequest(int fd) {
    while (true) {
        char command[1024] = {};
        size_t comamnd_length = 0;
        do {
            ssize_t read_result =
                read(fd, command + com
                     sizeof(command) -
            if (read_result <= 0) handle_error();
            command_length += read_result;
        } while (command[command_length - 1] != '\n');
        if (IsExitCommand(command)) { return; }
        char response[1024];
        computeResponse(response, commmand);
        size_t total_written = 0;
        while (total_written < sizeof(response)) {
            ...
        }
    }
```

```
class Connection {
  int fd;
  char command[1024];
  size_t command_length;
  char response[1024];
  size_t total_written;
  ...
};
```

some single-threaded processing code

```
void ProcessRequest(int fd) {
    while (true) {
        char command[1024] = {};
        size_t comamnd_length = 0;
        do {
            ssize_t read_result =
                read(fd, command + com
                     sizeof(command) -
            if (read_result <= 0) handle_error();
            command_length += read_result;
        } while (command[command_length - 1] != '\n');
        if (IsExitCommand(command)) { return; }
        char response[1024];
        computeResponse(response, commmand);
        size_t total_written = 0;
        while (total_written < sizeof(response)) {
            ...
        }
    }
```

```
class Connection {
  int fd;
  char command[1024];
  size_t command_length;
  char response[1024];
  size_t total_written;
  ...
};
```

as event code

```
handleRead(Connection *c) {
    ssize_t read_result =
        read(fd, c->command + command_length,
             sizeof(command) - c->command_length);
    if (read_result <= 0) handle_error();
    c->command_length += read_result;

    if (c->command[c->command_length - 1] == '\n') {
        computeResponse(c->response, c->command);
        if (IsExitCommand(command)) {
          FinishConnection(c);
        }
        StopWaitingToRead(c->fd);
        StartWaitingToWrite(c->fd);
    }
}
```

as event code

```
handleRead(Connection *c) {
    ssize_t read_result =
        read(fd, c->command + command_length,
             sizeof(command) - c->command_length);
    if (read_result <= 0) handle_error();
    c->command_length += read_result;

    if (c->command[c->command_length - 1] == '\n') {
        computeResponse(c->response, c->command);
        if (IsExitCommand(command)) {
          FinishConnection(c);
        }
        StopWaitingToRead(c->fd);
        StartWaitingToWrite(c->fd);
    }
}
```

POSIX support for event loops

`select` and `poll` functions

- take list(s) of file descriptors to read and to write
- wait for them to be read/writeable without waiting
- (or for new connections associated with them, etc.)

many OS-specific extensions/improvements/alternatives:

- examples: Linux epoll, Windows IO completion ports
- better ways of managing list of file descriptors
- do read/write when ready instead of just returning when reading/writing is okay