Network FS / Access Control

last time

two-phase commit

- consensus: workers + coordinator agree to commit
- no take-backs via redo-logging at each node
- blocking on failure

idea of majority-vote consensus

logistics note

last week's post-quiz due Thursday

twophase due next Monday

network filesystems

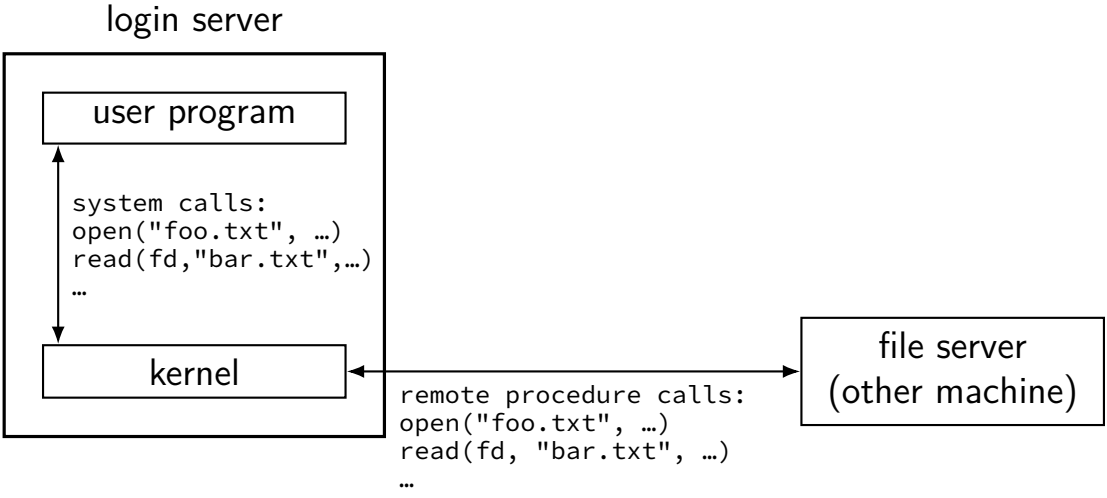
department machines — your files always there

even though several machines to log into

how? there's a *network file server*

filesystem is backed by a remote machine

simple network filesystem



system calls to RPC calls?

just turn system calls into RPC calls?

> (or calls to the kernel's internal fileystem abstraction, e.g. Linux's Virtual File System layer)

has some problems:

what state does the server need to store?

what if a client machine crashes?

what if the server crashes?

how fast is this?

state for server to store?

open file descriptors for each client process?

- what file
- offset in file

current working directory?

- what if we want some local and non-local files?

kinda expensive with many clients, running many processes

...but that's not the biggest issue

if a client crashes?

suppose a client hasn't sent anything to the server in 1 hour

can the server delete its open file information yet?

exercise: reasons why not?

take a minute to come up with your most plausible reason

if the server crashes?

well, first we restart the server/start a new one...

then, what do clients do?

probably need to restart to?

can we do better?

exercise: what information could clients keep to help

NFSv2

NFS (Network File System) version 2

standardized in RFC 1094 (1989)

based on RPC calls

NFSv2 RPC calls (subset)

LOOKUP(dir file ID, filename) $\rightarrow$ file ID

GETATTR(file ID) $\rightarrow$ (file size, owner, ...)

READ(file ID, offset, length) $\rightarrow$ data

WRITE(file ID, data, offset) $\rightarrow$ success/failure

CREATE(dir file ID, filename, metadata) $\rightarrow$ file ID

REMOVE(dir file ID, filename) $\rightarrow$ success/failure

SETATTR(file ID, size, owner, ...) $\rightarrow$ success/failure

NFSv2 RPC calls (subset)

LOOKUP(dir file ID, filename) → file ID

GETATTR(file ID) → (file size, owner, ...)

READ(file ID, offset, length) → data

WRITE(file ID, data, offset) → success/failure

CREATE(dir file ID, filename, metadata) → file ID

REMOVE(dir file ID, filename) → success/failure

SETATTR(file ID, size, owner, ...) → success/failure

file ID: opaque data (support multiple implementations)
example implementation: device+inode number+"generation number"

NFSv2 client versus server

clients: file descriptor $\rightarrow$server name, file ID, offset

client machine crashes? mapping automatically deleted

- "fate sharing"

server: convert file IDs to files on disk

- typically find unique number for each file
- usually by inode number

server doesn't get notified unless client is using the file

file IDs

device + inode + "*generation number*"?

generation number: incremented every time inode reused

file IDs

device + inode + "*generation number*"?

generation number: incremented every time inode reused

problem: file removed while client has it open

later client tries to access the file

- maybe inode number is valid *but for different file*
- inode was deallocated, then reused for new file

file IDs

device + inode + "*generation number*"?

generation number: incremented every time inode reused

problem: file removed while client has it open

later client tries to access the file

- maybe inode number is valid *but for different file*
- inode was deallocated, then reused for new file

Linux filesystems store a "generation number" in the inode

- basically just to help implement things like NFS

NFSv2 RPC calls (subset)

LOOKUP(dir file ID, filename) → file ID

GETATTR(file ID) → (file size, owner, ...)

READ(file ID, offset, length) → data

WRITE(file ID, data, offset) → success/failure

CREATE(dir file ID, filename, metadata) → file ID

REMOVE(dir file ID, filename) → success/failure

SETATTR(file ID, size, owner, ...) → success/failure

"stateless protocol" — no open/close/etc.
each operation stands alone

NFSv2 RPC (more operations)

READDIR(dir file ID, count, optional offset "cookie") $\rightarrow$
(names and file IDs, next offset "cookie")

NFSv2 RPC (more operations)

READDIR(dir file ID, count, optional offset "cookie") $\rightarrow$
(names and file IDs, next offset "cookie")

pattern: client storing opaque tokens

- for client: remember this, don't worry about what it means

tokens represent something the server can easily lookup

- file IDs: inode, etc.
- directory offset cookies: byte offset in directory, etc.

strategy for making stateful service stateless

statefulness

stateful protocol (example: FTP, two-phase commit)

- previous things in connection matter
- e.g. logged in user
- e.g. current working directory
- e.g. where to send data connection

stateless protocol (example: HTTP, NFSv2)

- each request stands alone
- servers remember nothing about clients between messages
- e.g. file IDs for each operation instead of file descriptor

stateful versus stateless

in client/server protocols:

stateless: more work for client, less for server

- client needs to remember/forward any information
- can run multiple copies of server without syncing them
- can reboot server without restoring any client state

stateful: more work for server, less for client

- client sets things at server, doesn't change anymore
- hard to scale server to many clients (store info for each client
- rebooting server likely to break active connections

exercise

Suppose we want to make a *stateless* file server. Which of the following features are possible?

[A] allowing clients to retrieve or write a whole file at a time, no matter its size

[B] a client that detects updates to a file within 60 seconds (assuming no failures or network slowdowns)

[C] a client completing an *open* operation without contacting the server, without risking inconsistency if another client is also modifying the directories containing that file

[D] a client completing an *open* operation without contacting the server, without risking inconsistency provided that another client has not/will not access the file or the directories containing it for at least 5 minutes

performance

before: reading/writing files/directories goes to local memory

- lots of work to use memory to cache, read-ahead

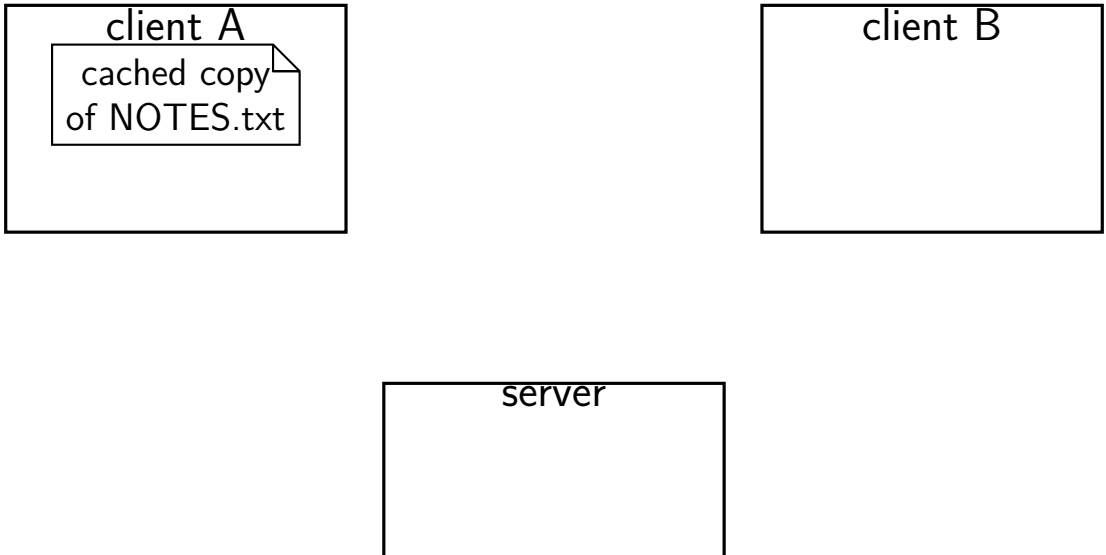
so open/read/write/close/rename/readdir/etc. take microseconds

- open that file? yes, I have the direntry cached
- read from that file? already in my memory

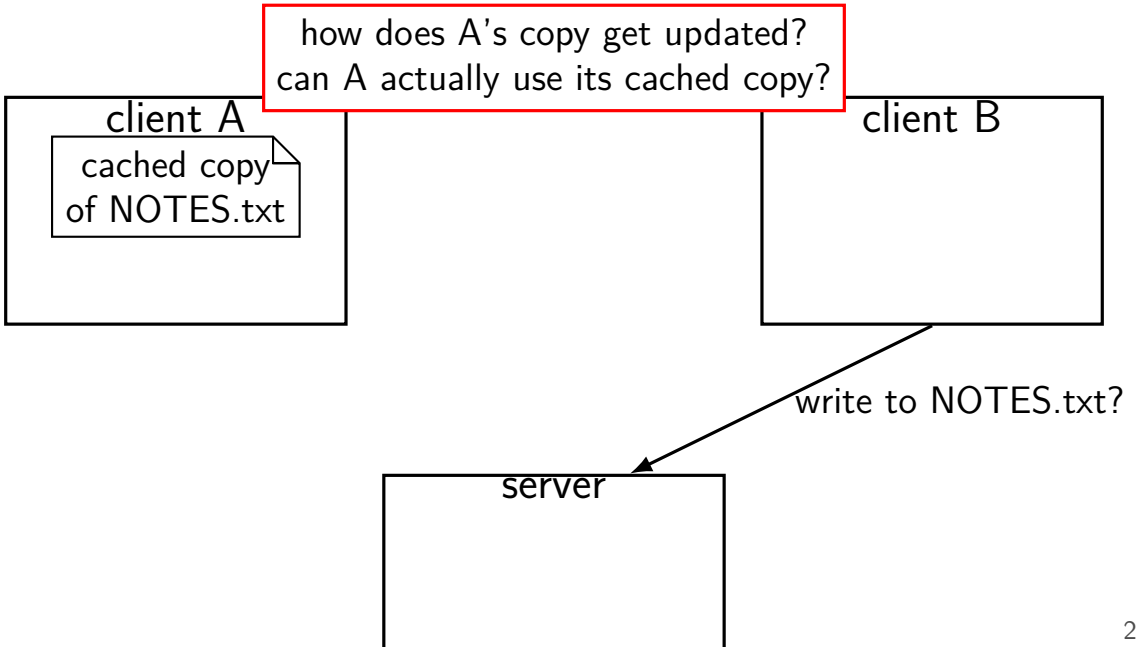
now: take milliseconds+

- open that file? let's ask the server if that's okay
- read from that file? let's copy it from the server
- etc.

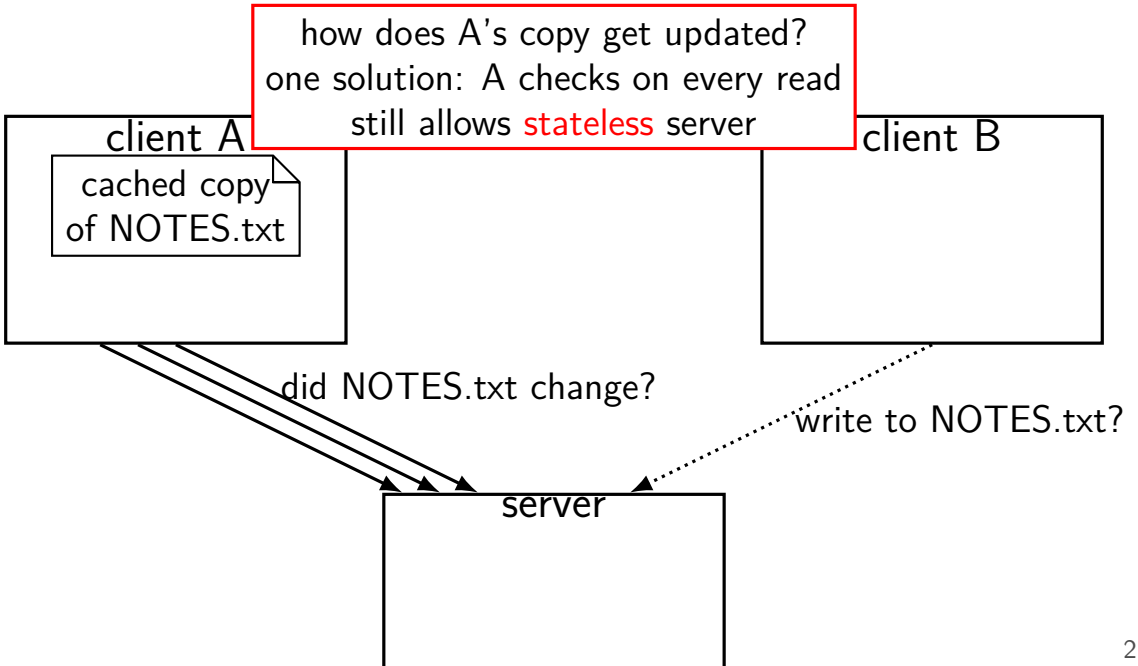
updating cached copies?



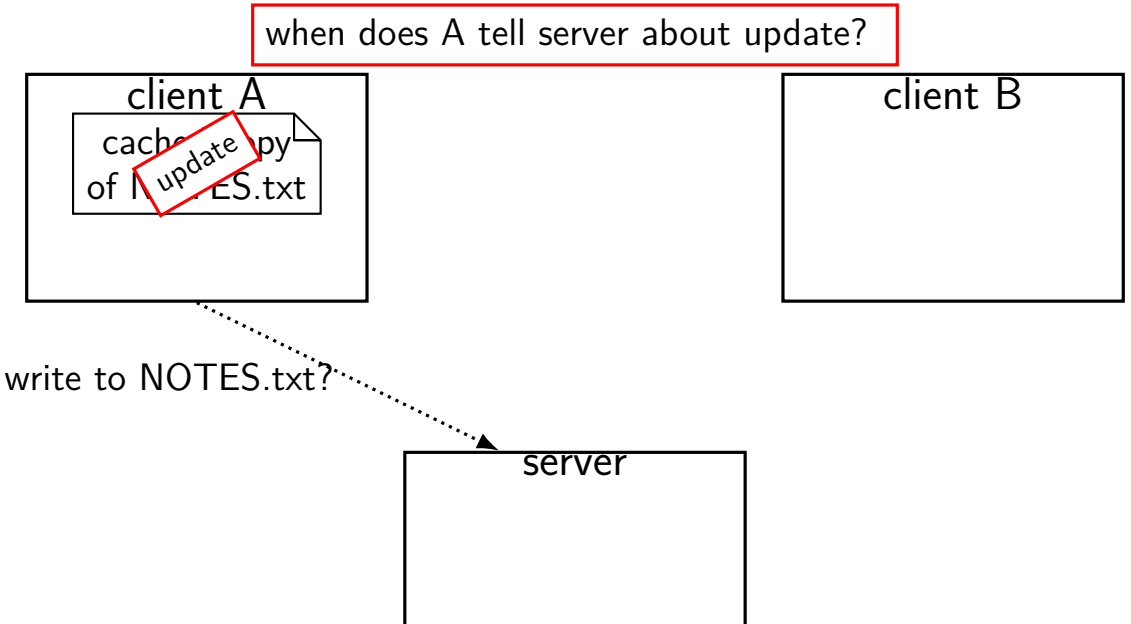
updating cached copies?



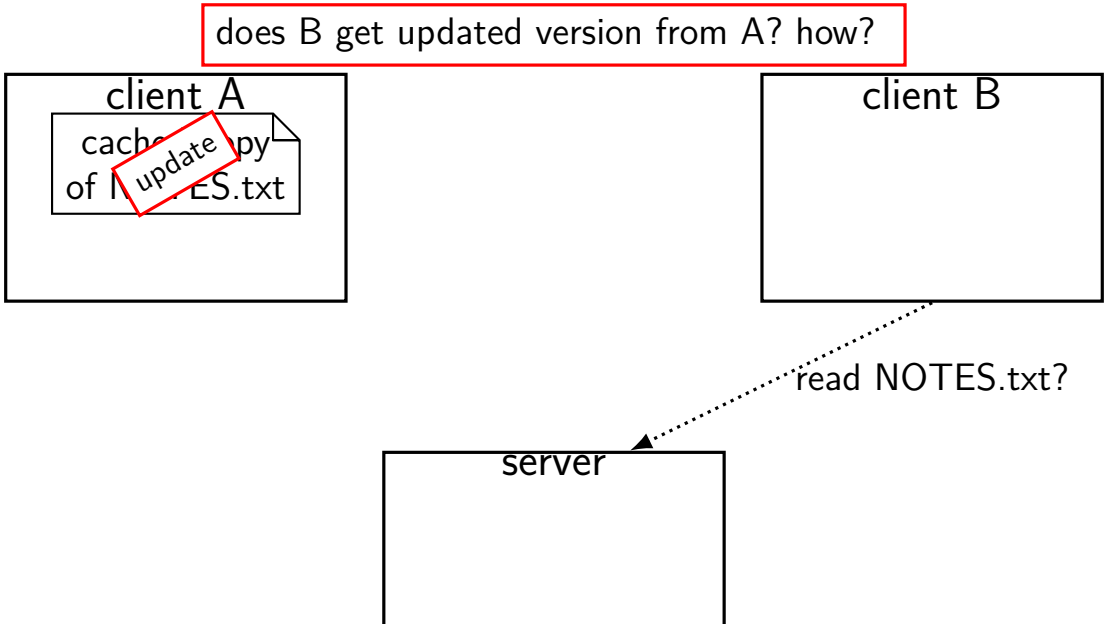
updating cached copies?



updating cached copies?



updating cached copies?



consistency with stateless server

always check server before using cached version

write through *all* updates to server

consistency with stateless server

always check server before using cached version

write through *all* updates to server

allows server to not remember clients

- no extra code for server/client failures, etc.

consistency with stateless server

always check server before using cached version

write through *all* updates to server

allows server to not remember clients

- no extra code for server/client failures, etc.

...but kinda destroys benefit of caching

- many milliseconds to contact server, even if not transferring data

consistency with stateless server

always check server before using cached version

write through *all* updates to server

allows server to not remember clients

- no extra code for server/client failures, etc.

...but kinda destroys benefit of caching

- many milliseconds to contact server, even if not transferring data

NFSv3's solution: allow some inconsistency

- update server on close; check cache on open

alternate solution: give up on stateless server

typical text editor/word processor

typical word processor:

opening a file:

- open file, read it, load into memory, close it

saving a file:

- open file, write it from memory, close it

two people saving a file?

have a word processor document on shared filesystem

Q: if you open the file while someone else is saving, what do you expect?

Q: if you save the file while someone else is saving, what do you expect?

two people saving a file?

have a word processor document on shared filesystem

Q: if you open the file while someone else is saving, what do you expect?

Q: if you save the file while someone else is saving, what do you expect?

observation: not things we really expect to work anyways

most applications don't care about accessing file while someone has it open

open to close consistency

a compromise:

opening a file checks for updated version

otherwise, use latest cache version

closing a file writes updates from the cache

otherwise, may not be immediately written

open to close consistency

a compromise:

opening a file checks for updated version

- otherwise, use latest cache version

closing a file writes updates from the cache

- otherwise, may not be immediately written

idea: as long as one user loads/saves file at a time, great!

protection/security

protection: mechanisms for controlling access to resources

- page tables, preemptive scheduling, encryption, ...

security: *using protection* to prevent misuse

- misuse represented by **policy**
- e.g. "don't expose sensitive info to bad people"

this class: about mechanisms more than policies

goal: provide enough flexibility for many policies

adversaries

security is about **adversaries**

do the worst possible thing

challenge: adversary can be clever...

authorization v authentication

authentication — who is who

authorization v authentication

authentication — who is who

authorization — who can do what

probably need authentication first...

authentication

password

hardware token

...

authentication

password

hardware token

...

this class: mostly won't deal with how

just tracking afterwards

access control matrix: who does what?

	file 1	file 2	process 1
domain 1	read/write		
domain 2	read	write	wakeup
domain 3	read	write	kill

access control matrix: who does what?

	file 1	file 2	process 1
domain 1	read/write		
domain 2	read	write	wakeup
domain 3	read	write	kill

each process belongs to 1+ *protection domains*:

"user cr4bd"

"group csfaculty"

...

access control matrix: who does what?

objects (whatever type) with restrictions

	file 1	file 2	process 1
domain 1	read/write		
domain 2	read	write	wakeup
domain 3	read	write	kill

each process belongs to 1+ *protection domains*:

"user cr4bd"

"group csfaculty"

...

access control matrix: who does what?

objects (whatever type) with restrictions

	file 1	file 2	process 1
domain 1	read/write		
domain 2	read	write	wakeup
domain 3	read	write	kill

each process belongs to 1+ *protection domains*:

"user cr4bd"

"group csfaculty"

...

access control matrix: who does what?

objects (whatever type) with restrictions

	file 1	file 2	process 1
domain 1	read/write		
domain 2	read	write	wakeup
domain 3	read	write	kill

each process belongs to 1+ *protection domains*:
"user cr4bd"
"group csfaculty"
...

user IDs

most common way OSes identify what *domain* process belongs to:

(unspecified for now) procedure sets user IDs

every process has a user ID

user ID used to decide what process is authorized to do

POSIX user IDs

```
uid_t geteuid(); // get current process's "effective" user ID
```

process's user identified with unique number

kernel typically only knows about number

effective user ID is used for all permission checks

also some other user IDs — we'll talk later

POSIX user IDs

```
uid_t geteuid(); // get current process's "effective" user ID
```

process's user identified with unique number

kernel typically only knows about number

effective user ID is used for all permission checks

also some other user IDs — we'll talk later

standard programs/library maintain number to name mapping

- `/etc/passwd` on typical single-user systems
- network database on department machines

POSIX groups

```
gid_t getegid(void);
    // process's"effective" group ID

int getgroups(int size, gid_t list[]);
    // process's extra group IDs
```

POSIX also has *group IDs*

like user IDs: kernel only knows numbers

- standard library+databases for mapping to names

also process has some other group IDs — we'll talk later

id

```
cr4bd@power4
: /net/zf14/cr4bd ; id
uid=858182(cr4bd) gid=21(csfaculty)
        groups=21(csfaculty),325(instructors),90027(cs4414)
```

id command displays uid, gid, group list

names looked up in database

- kernel doesn't know about this database
- code in the C standard library

groups that don't correspond to users

example: video group for access to monitor

put process in video group when logged in directly

don't do it when SSH'd in

groups that don't correspond to users

example: video group for access to monitor

put process in video group when logged in directly

don't do it when SSH'd in

...but: user can keep program running with video group in the background after logout?

access control matrix: who does what?

objects (whatever type) with restrictions

	file 1	file 2	process 1
domain 1	read/write		
domain 2	read	write	wakeup
domain 3	read	write	kill

each process belongs to 1+ *protection domains*:

"user cr4bd"

"group csfaculty"

...

representing access control matrix

with objects (files, etc.): *access control list*

- list of protection domains (users, groups, processes, etc.) allowed to use each item

list of (domain, object, permissions) stored "on the side"

- example: AppArmor on Linux
- configuration file with list of program + what it is allowed to access
- prevent, e.g., print server from writing files it shouldn't

POSIX file permissions

POSIX files have a very restricted access control list

one user ID + read/write/execute bits for user
    "owner" — also can change permissions

one group ID + read/write/execute bits for group

default setting — read/write/execute

(see docs for chmod command)

POSIX/NTFS ACLs

more flexible access control lists

list of (user or group, read or write or execute or ...)

supported by NTFS (Windows)

a version standardized by POSIX, but usually not supported

POSIX ACL syntax

```
# group students have read+execute permissions
group:students:r-x
# group faculty has read/write/execute permissions
group:faculty:rwx
# user mst3k has read/write/execute permissions
user:mst3k:rwx
# user tj1a has no permissions
user:tj1a:---

# POSIX acl rule:
    # user take precedence over group entries
```

authorization checking on Unix

checked on system call entry

- no relying on libraries, etc. to do checks

files (open, rename, ...) — file/directory permissions

processes (kill, ...) — process UID = user UID

...

superuser

user ID `0` is special

superuser or *root*

some system calls: only work for uid `0`

shutdown, mount new file systems, etc.

automatically passes all (or almost all) permission checks

how does login work?

```
somemachine login: jo
password: ********

jo@somemachine$ ls
...
```

this is a program which...

checks if the password is correct, and

changes user IDs, and

runs a shell

how does login work?

```
somemachine login: jo
password: ********

jo@somemachine$ ls
...
```

this is a program which...

checks if the password is correct, and

changes user IDs, and

runs a shell

Unix password storage

typical single-user system: `/etc/shadow`

- only readable by root/superuser

department machines: network service

- Kerberos / Active Directory:
- server takes (encrypted) passwords
- server gives tokens: "yes, really this user"
- can cryptographically verify tokens come from server

aside: beyond passwords

/bin/login entirely user-space code

only thing special about it: when it's run

could use any criteria to decide, not just passwords

- physical tokens
- biometrics
- ...

how does login work?

```
somemachine login: jo
password: ********

jo@somemachine$ ls
...
```

this is a program which...

checks if the password is correct, and

changes user IDs, and

runs a shell

changing user IDs

```
int setuid(uid_t uid);
```

if superuser: sets effective user ID to arbitrary value

- and a "real user ID" and a "saved set-user-ID" (we'll talk later)

system starts in/login programs run as superuser

- voluntarily restrict own access before running shell, etc.

sudo

```
tj1a@somemachine$ sudo restart
Password: *********
```

sudo: run command with superuser permissions

started by non-superuser

recall: inherits non-superuser UID

can't just call `setuid(0)`

set-user-ID sudo

extra metadata bit on *executables*: set-user-ID

if set: `exec()` syscall changes effective user ID to owner's ID

`sudo` program: owned by root, marked set-user-ID

marking setuid: `chmod u+s`

set-user ID gates

set-user ID program: gate to higher privilege

controlled access to extra functionality

make authorization/authentication decisions *outside the kernel*

way to allow normal users to do *one thing that needs privileges*

- write program that does that one thing — nothing else!
- make it owned by user that can do it (e.g. root)
- mark it set-user-ID

want to allow only some user to do the thing

- make program check which user ran it

uses for setuid programs

mount USB stick

- setuid program controls option to kernel mount syscall
- make sure user can't replace sensitive directories
- make sure user can't mess up filesystems on normal hard disks
- make sure user can't mount new setuid root files

control access to device — printer, monitor, etc.

- setuid program talks to device + decides who can

write to secure log file

- setuid program ensures that log is append-only for normal users

bind to a particular port number < 1024

- setuid program creates socket, then becomes not root

set-user-ID program v syscalls

hardware decision: some things only for kernel

system calls: *controlled* access to things kernel can do

decision about how can do it: in the kernel

kernel decision: some things only for root (or other user)

set-user-ID programs: controlled access to things root/... can do

decision about how can do it: made by root/...

a broken setuid program: setup

suppose I have a directory all-grades on shared server

in it I have a folder for each assignment

and within that a text file for each user's grade + other info

say I don't have flexible ACLs and want to give each user access

a broken setuid program: setup

suppose I have a directory all-grades on shared server

in it I have a folder for each assignment

and within that a text file for each user's grade + other info

say I don't have flexible ACLs and want to give each user access

one (bad?) idea: setuid program to read grade for assignment

`./print_grade assignment`

outputs grade from `all-grades/assignment/USER.txt`

a very broken setuid program

print_grade.c:

```
int main(int argc, char **argv) {
    char filename[500];
    sprintf(filename, "all-grades/%s/%s.txt",
            argv[1], getenv("USER"));
    int fd = open(filename, O_RDWR);
    char buffer[1024];
    read(fd, buffer, 1024);
    printf("%s: %s\n", argv[1], buffer);
}
```

HUGE amount of stuff can go wrong

examples?

set-user ID programs are very hard to write

what if stdin, stdout, stderr start closed?

what if the PATH env. var. set to directory of malicious programs?

what if `argc == 0`?

what if dynamic linker env. vars are set?

what if some bug allows memory corruption?

...

a delegation problem

consider printing program marked setuid to access printer

> decision: no accessing printer directly
> printing program enforces page limits, etc.

command line: file to print

can printing program just call open()?

a broken solution

```
if (original user can read file from argument) {
    open(file from argument);
    read contents of file;
    write contents of file to printer
    close(file from argument);
}
```

hope: this prevents users from printing files than can't read

problem: race condition!

a broken solution / why

setuid program	other user program
	create normal file `toprint.txt`
check: can user access? (yes)	—
	`unlink("toprint.txt")`
	`link("/secret", "toprint.txt")`
`open("toprint.txt")`	—
read ...	—

link: create new directory entry for file

another option: rename, symlink ("symbolic link" — alias for file/directory)

another possibility: run a program that creates secret file (e.g. temporary file used by password-changing program)

time-to-check-to-time-of-use vulnerability

TOCTTOU solution

temporarily ‘become’ original user

then open

then turn back into set-uid user

this is why POSIX processes have multiple user IDs

can swap out effective user ID temporarily

practical TOCTTOU races?

can use symlinks *maze* to make check slower

symlink `toprint.txt` → `a/b/c/d/e/f/g/normal.txt`
symlink `a/b` → `../a`
symlink `a/c` → `../a`
...

lots of time spent following symbolic links when program opening toprint.txt

gives more time to sneak in unlink/link or (more likely) rename

exercise

which (if any) of the following would fix for a TOCTTOU vulnerability in our setuid printing application? (assume the Unix-permissions without ACLs are in use)

[A] **both before and after** opening the path passed in for reading, check that the path is accessible to the user who ran our application

[B] after opening the path passed in for reading, using `fstat` with the file descriptor opened to check the permissions on the file

[C] before opening the path, verify that the user controls the file referred to by the path **and** the directory containing it

some security tasks (1)

helping students collaborate in ad-hoc small groups on shared server?

Q1: what to allow/prevent?

Q2: how to use POSIX mechanisms to do this?

some security tasks (2)

letting students assignment files to faculty on shared server?

Q1: what to allow/prevent?

Q2: how to use POSIX mechanisms to do this?

some security tasks (3)

running untrusted game program from Internet?

Q1: what to allow/prevent?

Q2: how to use POSIX mechanisms to do this?

backup slides

sandboxing

sandbox — restricted environment for program

idea: dangerous code can play in the sandbox as much as it wants

can't do anything harmful

sandbox use cases

buggy video parsing code that has buffer overflows

browser running scripts in webpage

autograder running student submissions

...

sandbox use cases

buggy video parsing code that has buffer overflows

browser running scripts in webpage

autograder running student submissions

...

(parts of) program that don't need to have user's full permissions

no reason video parsing code should be able open() my taxes

can we have a way to ask OS for this?

Google Chrome architecture

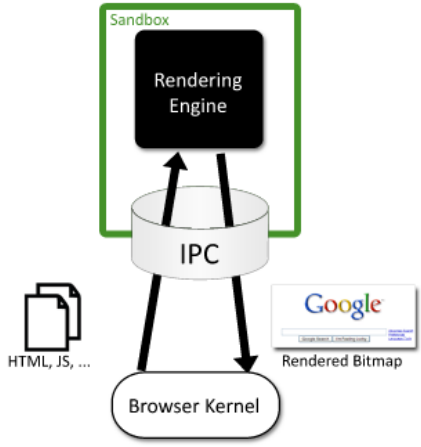


Figure 1: The browser kernel treats the rendering engine as a black box that parses web content and emits bitmaps of the rendered document.

sandboxing mechanisms

create a new user with few privileged, switch to user

- problem: creating new users usually requires sysadmin access
- problem: every user can do too much
- e.g. everyone can open network connection?

with capabilities, just discard most capabilities

- just close capabilities you don't need
- run rendering engine with only pipes to talk to browser kernel

otherwise: system call filtering

- disallow all 'dangerous' system calls

Linux system call filtering

`seccomp()` system call

"strict mode": only allow read/write/_exit/sigreturn

- current thread gives up all other privileges
- usage: setup pipes, then communicate with rest of process via pipes

alternately: setting a whitelist of allowed system calls + arguments

- little programming language (!) for supported operations

browsers use this to protect from bugs in their scripting implementations

- hope: find a way to execute arbitrary code? — not actually useful

sandbox browser setup

create pipe

spawn subprocess (“rendering engine”)

put subproces in strict system call filter mode

send subprocesses webpages + events

subprocess sends images to render back on pipe

aside: real/effective/saved

POSIX processes have *three* user IDs

effective — determines permission — `geteuid()`

- jo running sudo: geteuid = superuser's ID

real — the user who started the program — `getuid()`

- jo running sudo: getuid = jo's ID

saved set-user-ID — user ID from *before* last exec

- effective user ID saved when a set-user-ID program starts
- jo running sudo: = jo's ID
- no standard get function, but see Linux's `getresuid`

process can swap or set effective UID with real/saved UID

aside: real/effective/saved

POSIX processes have *three* user IDs

effective — determines permission — `geteuid()`

- jo running sudo: geteuid = superuser's ID

real — the user who started the program — `getuid()`

- jo running sudo: getuid = jo's ID

saved set-user-ID — user ID from *before* last exec

- effective user ID saved when a set-user-ID program starts
- jo running sudo: = jo's ID
- no standard get function, but see Linux's `getresuid`

process can swap or set effective UID with real/saved UID

- idea: become other user for one operation, then switch back

why so many?

two versions of Unix:

System V — used effective user ID + saved set-user-ID

BSD — used effective user ID + real user ID

POSIX commitee solution: keep both

aside: confusing setuid functions

setuid — if root, change all uids; otherwise, only effective uid

seteuid — change effective uid

> if not root, only to real or saved-set-user ID

setreuid — change real+effective; sometimes saved, too

> if not root, only to real or effective or saved-set-user ID

...

more info: Chen et al, "Setuid Demystified"

> `https://www.usenix.org/conference/11th-usenix-security-symposium/setuid-demystified`

also group-IDs

processes also have a real/effective/saved-set group-ID

can also have set-group-ID executables

same as set-user-ID, but only changes group

sandboxing use case: buggy video decoder

```
/* dangerous video decoder to isolate */
int main() {
    EnterSandbox();
    while (fread(videoData, sizeof(videoData), 1, stdin) > 0) {
        doDangerousVideoDecoding(videoData, imageData);
        fwrite(imageData, sizeof(imageData), 1, stdout);
    }
}

/* code that uses it */
    FILE *fh = RunProgramAndGetFileHandle("./video-decoder");
    for (;;) {
        fwrite(getNextVideoData(), SIZE, 1, fh);
        fread(image, sizeof(image), 1, fh);
        displayImage(image);
    }
```

typical text editor/word processor

typical word processor:

opening a file:

- open file, read it, load into memory, close it

saving a file:

- open file, write it from memory, close it

two people saving a file?

have a word processor document on shared filesystem

Q: if you open the file while someone else is saving, what do you expect?

Q: if you save the file while someone else is saving, what do you expect?

two people saving a file?

have a word processor document on shared filesystem

Q: if you open the file while someone else is saving, what do you expect?

Q: if you save the file while someone else is saving, what do you expect?

observation: not things we really expect to work anyways

most applications don't care about accessing file while someone has it open

open to close consistency

a compromise:

opening a file checks for updated version

- otherwise, use latest cache version

closing a file writes updates from the cache

- otherwise, may not be immediately written

open to close consistency

a compromise:

opening a file checks for updated version

- otherwise, use latest cache version

closing a file writes updates from the cache

- otherwise, may not be immediately written

idea: as long as one user loads/saves file at a time, great!

an alternate compromise

application opens a file, read it a day later, result?

- day-old version of file

modification 1: check server/write to server after an amount of time

doesn't need to be much time to be useful

- word processor: typically load/save file in $<$ second

AFSv2

Andrew File System version 2

uses a stateful server

also works file at a time — not parts of file

- i.e. read/write entire files

but still chooses consistency compromise

- still won't support simulatenous read+write from diff. machines well

stateful: avoids repeated 'is my file okay?' queries

NFS versus AFS reading/writing

NFS reading: read/write block at a time

AFS reading: always read/write *entire file*

exercise: pros/cons?

- efficient use of network?
- what kinds of inconsistency happen?
- does it depend on workload?

AFS: last writer wins

on client A	on client B
open NOTES.txt	
	open NOTES.txt
write to cached NOTES.txt	
	write to cached NOTES.txt
close NOTES.txt	
AFS: write whole file	
	close NOTES.txt
	AFS: write whole file

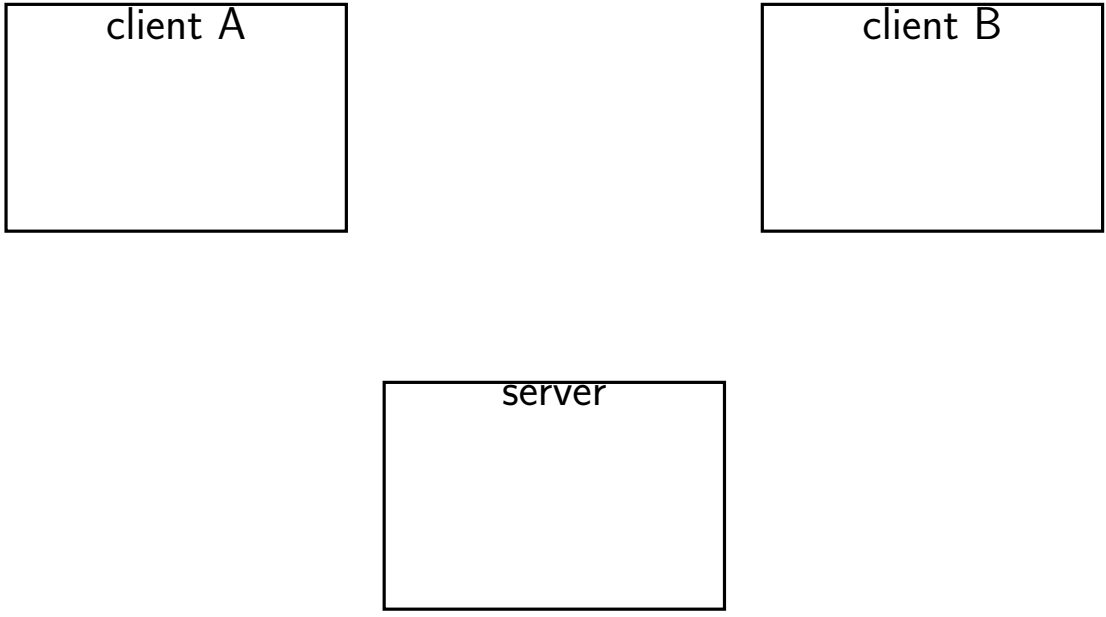
last writer wins

NFS: last writer wins per block

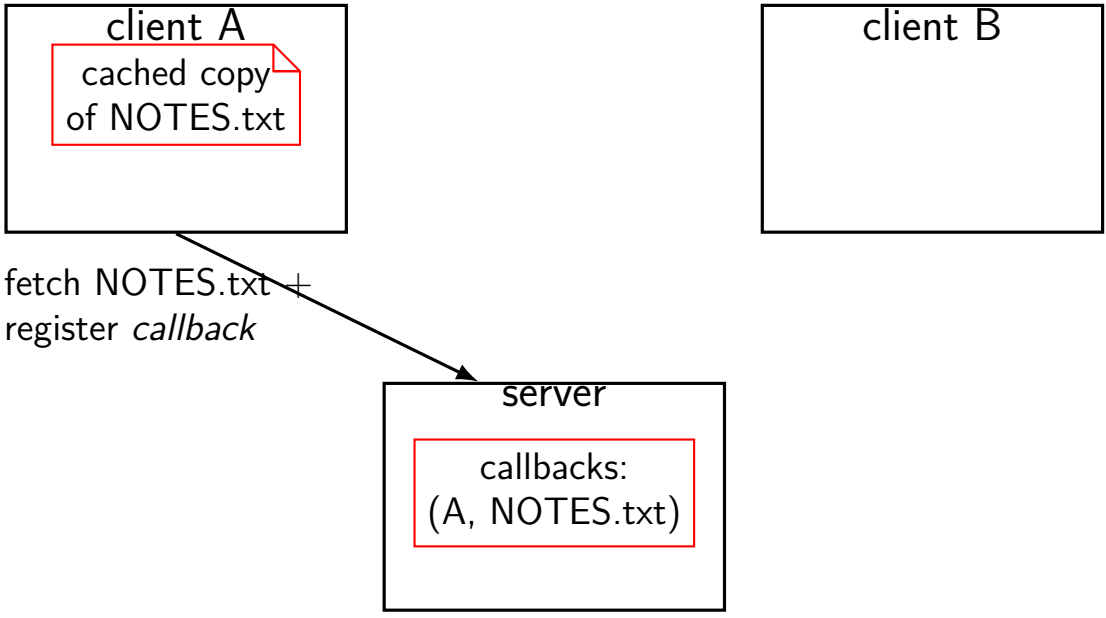
on client A	on client B
open NOTES.txt	
	open NOTES.txt
write to cached NOTES.txt	
	write to cached NOTES.txt
close NOTES.txt	
NFS: write NOTES.txt block 0	
	close NOTES.txt
	NFS: write NOTES.txt block 0
	NFS: write NOTES.txt block 1
NFS: write NOTES.txt block 1	
NFS: write NOTES.txt block 2	
	NFS: write NOTES.txt block 2

NOTES.txt: 0 from B, 1 from A, 2 from B

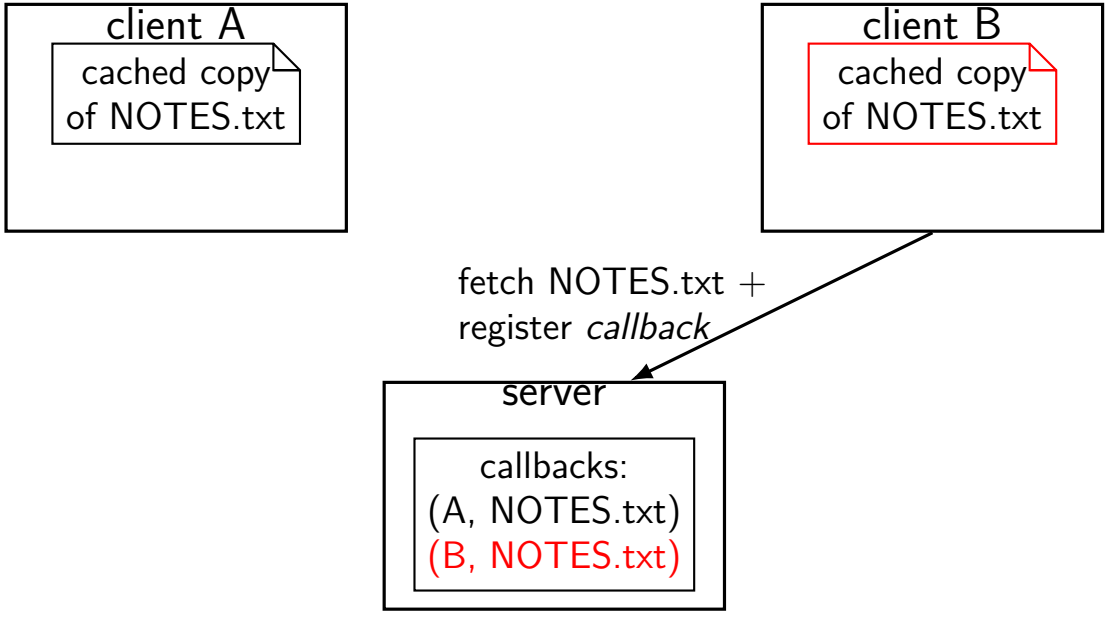
AFS caching



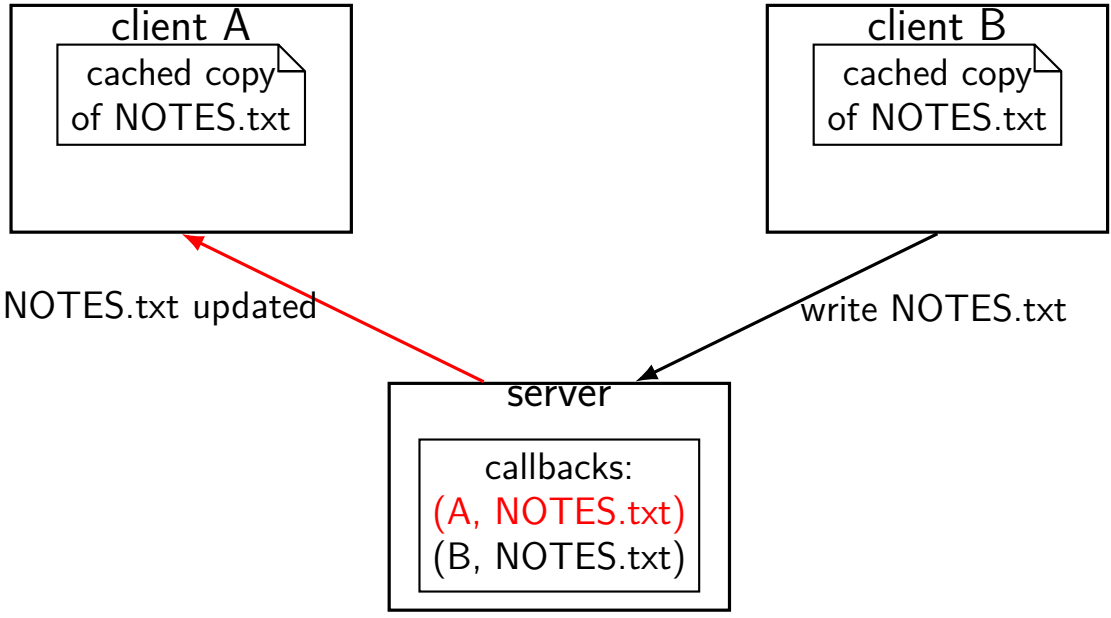
AFS caching


client A
cached copy
of NOTES.txt
client B
fetch NOTES.txt +
register *callback*
server
callbacks:
(A, NOTES.txt)

AFS caching


client A
cached copy of NOTES.txt
client B
cached copy of NOTES.txt
fetch NOTES.txt +
register *callback*
server
callbacks:
(A, NOTES.txt)
(B, NOTES.txt)

AFS caching


client A
cached copy of NOTES.txt
client B
cached copy of NOTES.txt
NOTES.txt updated
write NOTES.txt
server
callbacks:
(A, NOTES.txt)
(B, NOTES.txt)

callback inconsistency (1)

on client A	on client B
open NOTES.txt	
(AFS: NOTES.txt fetched)	
read from cached NOTES.txt	
	open NOTES.txt
	(NOTES.txt fetched)
	read from NOTES.txt
write to cached NOTES.txt	
	read from NOTES.txt
write to cached NOTES.txt	
close NOTES.txt	
(write to server)	
	(AFS: callback: NOTES.txt changed)

callback inconsistency (1)

on client A	on client B
open NOTES.txt	
(AFS: NOTE	
read from ca	
	read from NOTES.txt
write to cached NOTES.txt	
	read from NOTES.txt
write to cached NOTES.txt	
close NOTES.txt	
(write to server)	
	(AFS: callback: NOTES.txt changed)

problem with close-to-open consistency
same issue w/NFS: B can't know about write
because server doesn't
(could fix by notifying server earlier)

callback inconsistency (1)

on client A	on client B
open NOTES.txt	
(AFS: NOTES.txt fetched)	
read from cached NOTES.txt	
	open NOTES.txt
	(NOTES.txt fetched)
	read from NOTES.txt
write to cached NOTES.txt	
	read from NOTES.txt
write to cached NOTES.txt	
close NOTES.txt	
(write to server)	
	(AFS: callback: NOTES.txt changed)

close-to-open consistency assumption:
are not accessing file from two places at once

supporting offline operation

so far: assuming constant contact with server

someone else writes file: we find out

we finish editing file: can tell server right away

good for an office

- my work desktop can almost always talk to server

not so great for mobile cases

- spotty airport/café wifi, no cell reception, ...

basic offline operation idea

when offline: work on cached data only

writeback whole file only

problem: more opportunity for overlapping accesses to same file

recall: AFS: last writer wins

on client A	on client B
open NOTES.txt	
	open NOTES.txt
write to cached NOTES.txt	
	write to cached NOTES.txt
close NOTES.txt	
AFS: write whole file	
	close NOTES.txt
	AFS: (over)write whole file

probably losing data!

usually wanted to merge two versions

recall: AFS: last writer wins

on client A	on client B
open NOTES.txt	
	open NOTES.txt
write to cached NOTES.txt	
	write to cached NOTES.txt
close NOTES.txt	
AFS: write whole file	
	close NOTES.txt
	AFS: (over)write whole file

probably losing data!

usually wanted to merge two versions

worse problem with delayed writes for disconnected operation

Coda FS: conflict resolution

Coda: distributed FS based on AFSv2 (c. 1987)

supports offline operation with conflict resolution

while offline: clients remember *previous version ID of file*

clients include version ID info with file updates

allows detection of conflicting updates

- avoid problem of last writer wins

Coda FS: conflict resolution

Coda: distributed FS based on AFSv2 (c. 1987)

supports offline operation with conflict resolution

while offline: clients remember *previous version ID of file*

clients include version ID info with file updates

allows detection of conflicting updates

- avoid problem of last writer wins

and then...ask user? regenerate file? ...?

Coda FS: what to cache

idea: user specifies list of files to keep loaded

when online: client synchronizes with server

uses version IDs to decide what to update

Coda FS: what to cache

idea: user specifies list of files to keep loaded

when online: client synchronizes with server

uses version IDs to decide what to update

DropBox, etc. probably similar idea?

version ID?

not a version number?

actually a *version vector*

version number for each machine that modified file

- number for each server, client

allows use of multiple servers

- if servers get desync'd, use version vector to detect
- then do, uh, something to fix any conflicting writes

file locking

so, your program doesn't like conflicting writes

what can you do?

if offline operation, probably not much...

otherwise file locking

except it often doesn't work on NFS, etc.

advisory file locking with fcntl

```
int fd = open(...);
struct flock lock_info = {
    .l_type = F_WRLCK,  // write lock; RDLOCK also available
    // range of bytes to lock:
    .l_whence = SEEK_SET, l_start = 0, l_len = ...
};
/* set lock, waiting if needed */
int rv = fcntl(fd, F_SETLKW, &lock_info);
if (rv == -1) { /* handle error */ }
/* now have a lock on the file */

/* unlock --- could also close() */
lock_info.l_type = F_UNLCK;
fcntl(fd, F_SETLK, &lock_info);
```

advisory locks

fcntl is an *advisory* lock

doesn't stop others from accessing the file...

unless they always try to get a lock first

POSIX file locks are horrible

actually two locking APIs: fcntl() and flock()

fcntl: *not* inherited by fork

fcntl: closing any fd for file release lock

even if you dup2'd it!

fcntl: maybe sometimes works over NFS?

flock: less likely to work over NFS, etc.

fcntl and NFS

seems to require extra state at the server

typical implementation: separate *lock server*

not a stateless protocol

lockfiles

use a separate *lockfile* instead of "real" locks

- e.g. convention: use `NOTES.txt.lock` as lock file

lock: create a *lockfile* with `link()` or `open()` with `O_EXCL`

- can't lock: link()/open() will fail "file already exists"
- for current NFSv3: should be single RPC calls that always contact server
- some (old, I hope?) systems: link() atomic, open() O_EXCL not

unlock: remove the lockfile

- annoyance: what if program crashes, file not removed?