CS202 (-002): Lecture 21: Journaling + file system permissions

Announcements

- Timeline
  - + Today: Finish crash recovery
    Some security
  - + Thursday: More security
  - + 12/09: Putting it all together
    → What happens when you power on a computer
  - + 12/11: Final exam review
  - + 12/18: Final Exam
    2 – 3:50 pm
    Here!!!

Where we were

- Crashes can leave filesystem in an inconsistent state
  - → Why?
    - Each function can require updating several disk blocks (which is where the authoritative version of FS state lives)
    - But disk only provides atomicity at the granularity of operation on a block
    - ⇒ Crash can result in only a part of a function being run.
  - → What we want?

'Functions that manipulate the file system should appear to be atomic

↳ All ops happen or none.

+ How?

→ Ad hoc — Clever ordering tricks

+ Efficient in absense of crashes

- Hard to write & maintain
- Expensive on recovery — Scan the disk

→ Copy-on-Write — All blocks (except super/uber block) are immutable

+ Efficient recovery — No recovery logic

+ Easy to write/maintain

- Write amplification — Inefficient common (crash-free) operations

- Space inefficient

For both — can ask what is the COMMIT POINT?

Journaling [NTFS, Ext 3, Ext 4, ...]

Historical note: Adopted before CoW (JFS ~1990; NTFS ~1993; HFS+ ~98)

But intellectually easier to derive from CoW

How does CoW ensure consistency

↳ Does not modify a committed copy

→ On crash-recovery — the last comitted copy is used

Want a way to reduce overheads (# of writes, space)

→ Record enough information to

Restore Last committed version (UNDO Logging)

OR

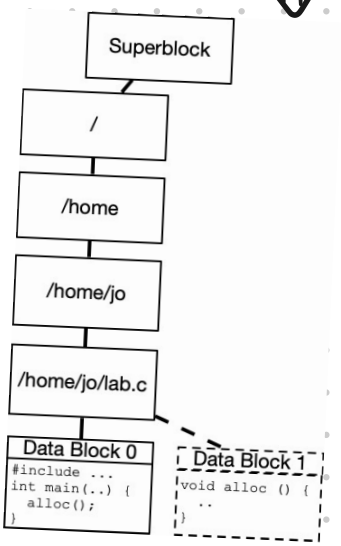
Finish uncommitted operation (REDO Logging)

↳ More common. Usually do not record changes to data blocks

[Lower overheads + less catastrophic]

Redo Logging

Want to achieve ——— How?



1. Log function
   - Allocate data block 1
   - Modify inode for /home/jo/lab.c

Commit Point →

2. Apply changes.

On recovery:

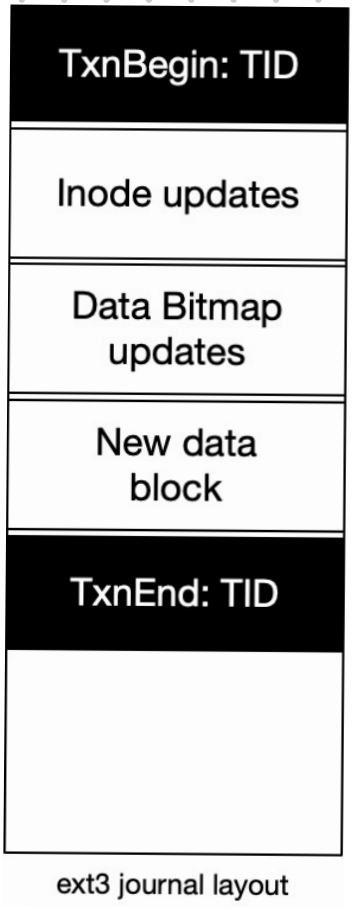
Apply any changes that were logged but not applied.

Problem: Logging operations might not be atomic

→ Why??

What we need: way to determine if all of a function's OPERATIONS were logged.

→ Use a common trick ↔ Order updates + record completion



ext3 journal layout

Log Procedure:

1. Write Txn Begin with an ID to log
   — Wait until write completes —
2. Log operations — 1 or more disk writes
   - Can be in any order
3. — Wait until ALL operations logged —
   Write Txn End with an ID to log
   — Wait until write completes —

Observe Txn End ⇒ Function logged completely.

Normal Case

→ (1) Log operations using Log Procedure
   ⟩— Commit Point —⟨
(2) Apply operations.

Recovery

(1) Read log to find COMMITTED CHANGES
   - How?

how?

② Apply them.

Undo logging

1. Write Txn Begin with TID

2. For each op ∈ operations
   - Write how to undo op
   - Once undo information is written, tell OS is safe to apply op

3. Write txn end

Recovery

① Find uncommitted log entries
   - how?

② Use log info to undo operations

Q. What is the commit point?

Redo vs Undo

+ Commit before updating blocks
  - ↳ Can delay block updates, allow better disk scheduling
- Cannot evict changed blocks from buffer cache until logging finishes
  - ↳ Memory pressure

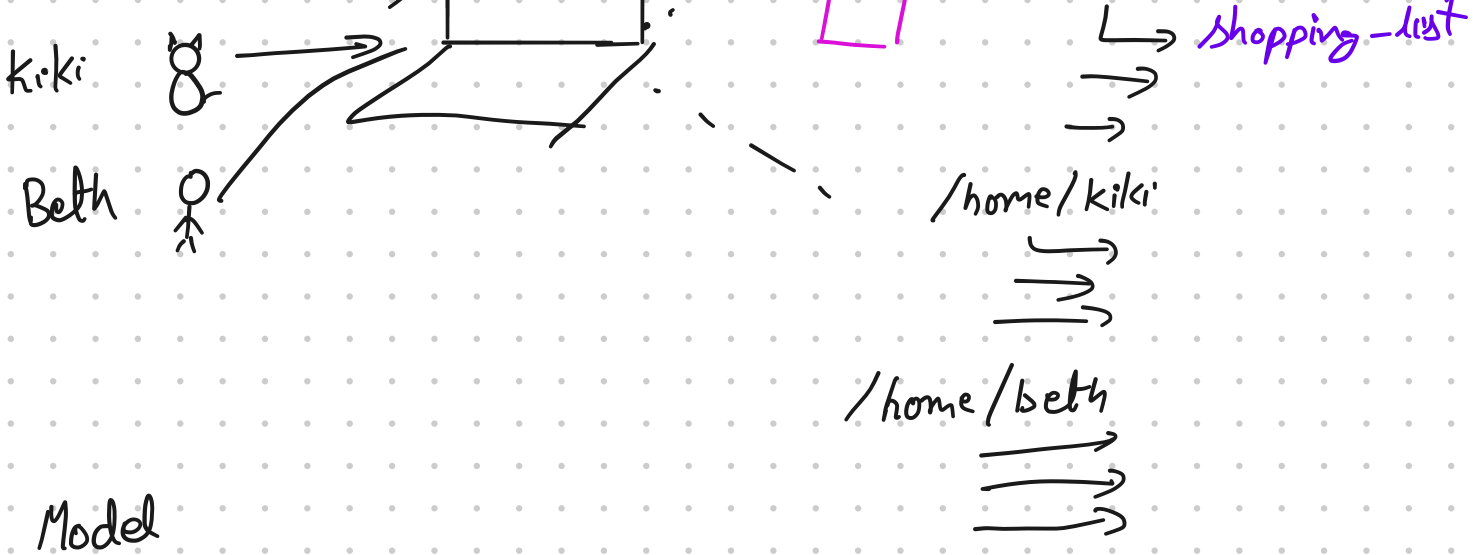
+ Flush & evict block from buffer cache as soon as operation logged
- Commit only after all ops complete.

Redo + Undo ≡ See notes.

OK, that is all for file systems
↳ Questions?

Security

Remember goal of multi-user OS (like Unix)



Model

- Users

username	user ID (uid)
root	0
alice	1000
kiki	1001
beth	1002

- Groups

group name	group ID (gid)
root	0
humans	25
cats	2000
apt dweller	2002
...	

Each user can be in 1 or more groups

alice	humans, apt dweller
kiki	cats, apt dweller
beth	humans, apt dweller

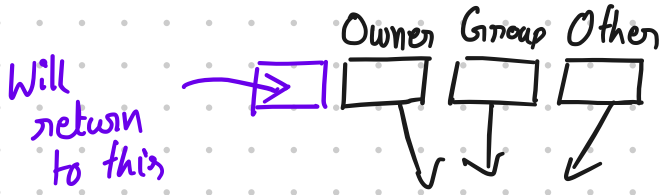
Each file has an owning user & group
↳ Where is this stored?

Eg, /home/alice/shopping-list
↳ Owner: alice

Group : humans

- Finally, each file has a set of permissions [lab 2]

  Often, expressed as a 3-4 digit octal (base 8) number



R	W	X	
0	0	1	⇒ 1
0	1	0	⇒ 2
1	0	0	⇒ 4
1	1	0	⇒ 6
...			

Decide who can access the file & how

a+rw

alice
kiki
beth

humans, apt-dweller
cats, apt-dweller
humans, apt-dweller

/home/alice/shopping-list
↳ Owner : alice
Group : humans

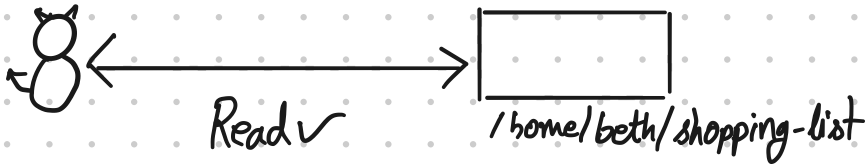
O G W
6 6 4

alice
beth
kiki

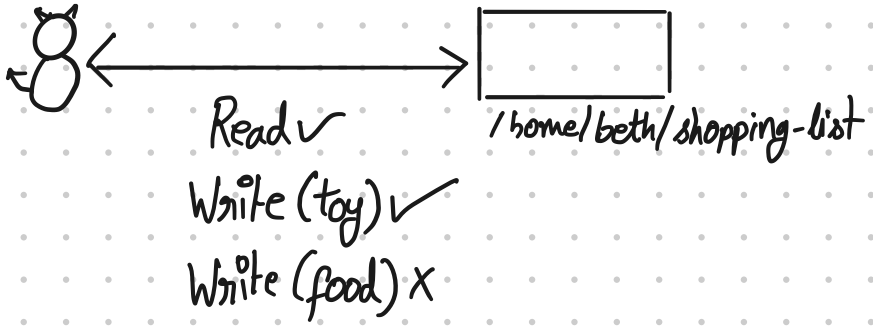
RWX
1 0
↳ Read
wr

100
↳ Rea

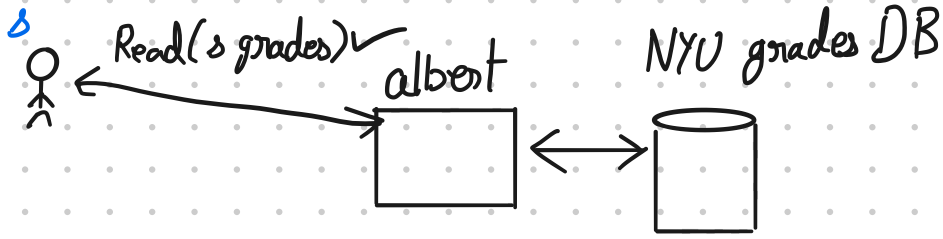
Somewhat of a blunt instrument — control whether or not a user can perform some op on a file



Often want something more complex

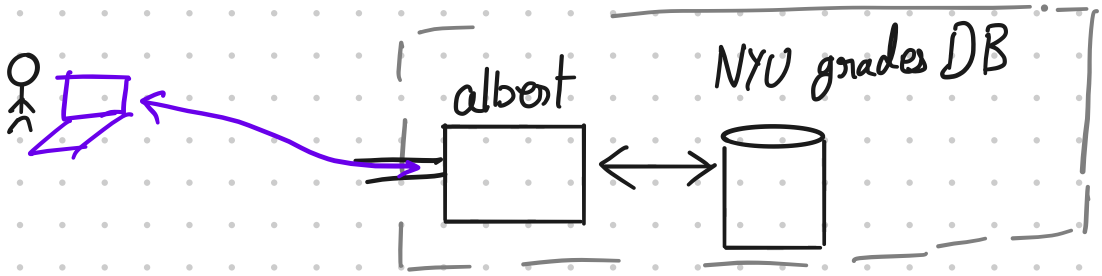


Pretty common in reality



To implement this — make sure shared file only accessible using a prog. that enforces some policy.

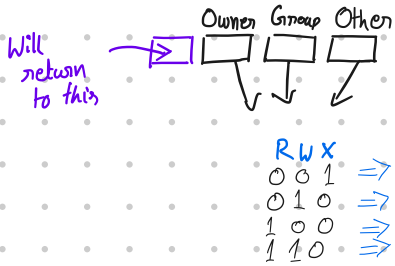
One way: make file, program inaccessible except over network

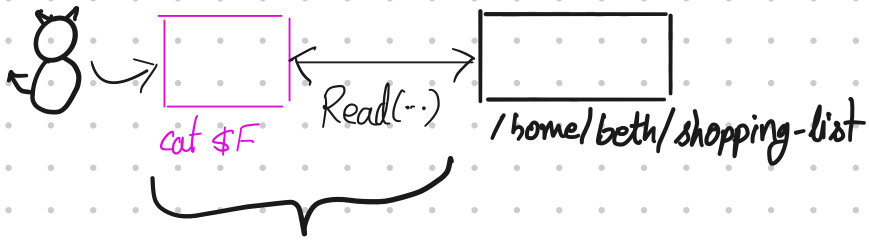


Mostly what we do today.

Has a couple of downsides —

Another way





↳ What kernel checks?

Similarly for group

- What user is running the process?
  getuid(2)
- Does the user have access?

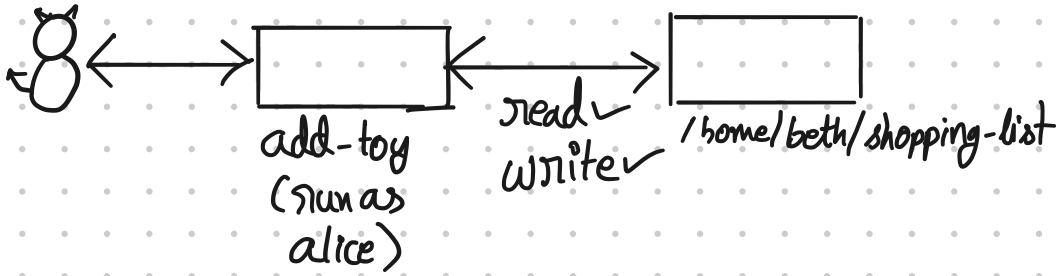
What is an executing proc's user ID: the uid of the user who started it

↳ Really: the uid of the process that called fork

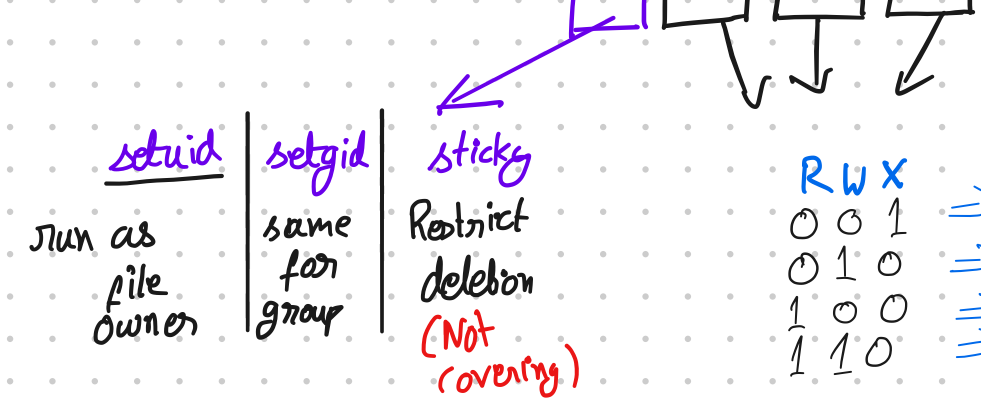
→ Or can be set using setuid(2)

There are rules - can lower (but not raise) privilege, etc.

Idea: What if an executable's owner determined the exec processes UID



Mech: setuid



setuid	setgid	sticky
Run as file owner	same for group	Restrict deletion (Not covering)

R	W	X	
0	0	1	⇒ 1
0	1	0	⇒ 2
1	0	0	⇒ 4
1	1	0	⇒ 6

Effective UID