CS 202 (-002) Lecture 20: More file systems

Announcements

1. No class next week – Thanksgiving
2. No office hours next week – also Thanksgiving
3. But: Lab 5 Review Session with Hao
   – Tuesday May 25. This room @ 3:30pm

Things we have covered

- Disk geometry
- File system – why & how
  - Files
  - Directories
  - Bootstrap
- Performance
  - Allocate for locality
- Buffer cache.

PLAN FOR TODAY

- CRASH CONSISTENCY.

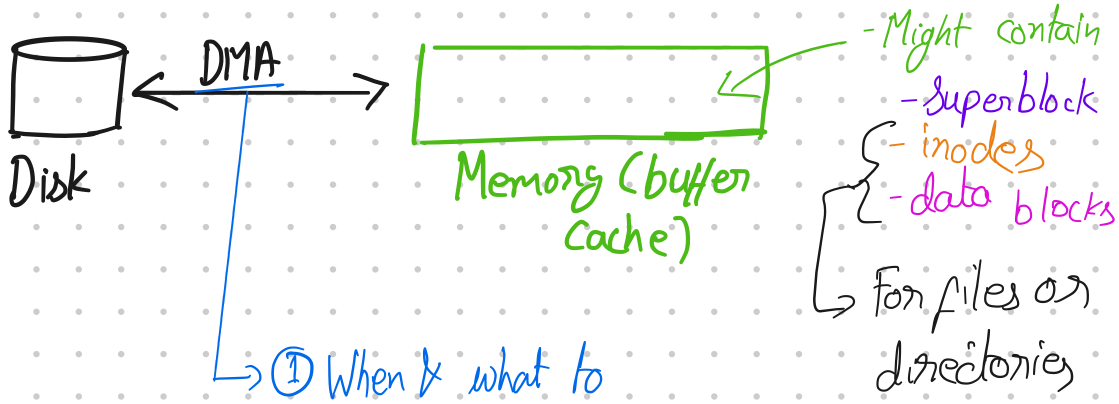
Previously on...

Need to block if:

@ Requested content is not

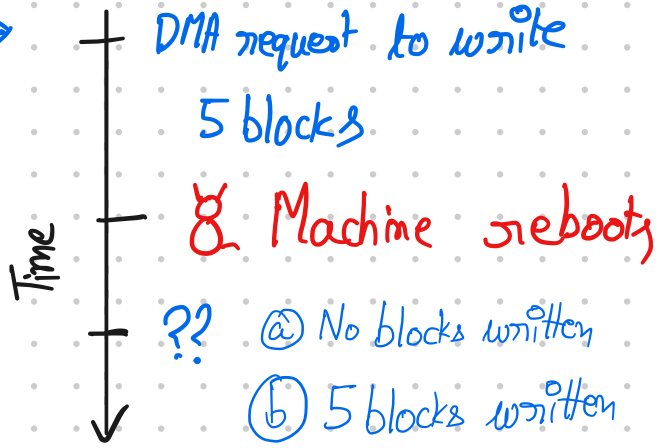
already in buffer cache

(b) Required for safe concurrency.



① When & what to read/write determined by OS

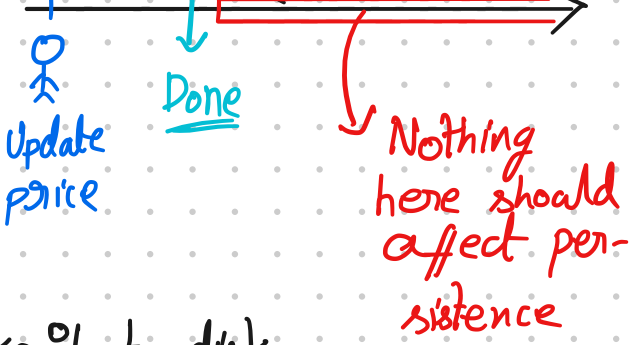
② Reads & writes are not ATOMIC



Two concerns

(1)

- Want to ensure user knows their changes will persist



Need to ensure changes make it to disk BEFORE DB says it is done

- Provide a syscall for this: fflush
  - ↳ Lots of complexity, won't cover it now

② A single ~~operation~~ function involves changes to many blocks

```
(demo) >>> ls
(demo) >>> echo "This is a test" > nfile
(demo) >>> ls
nfile
(demo) >>>
```

How many blocks are changed?

1. Inode for nfile
2. Data block for nfile
3. Data block for demo
   - ↳ Link nfile
4. Bitmap/free list
   - ↳
5. (Maybe) Inode for demo

...

Remember, a crash might mean some of these changes

are not persisted. What can go wrong?

1. Inode for nfile
2. Data block for nfile
3. Data block for demo
   ↳
4. Bitmap/free list
   ↳
5. (Maybe) Inode for demo

...

- Allocator might incorrectly allocate nfile's inode or data block.
  * Data corruption
- No dirent might point to nfile
  * Leak disk space
- ...

Overall: Inopportune crash can lead to inconsistent file system state.

↳ Need a way to ensure file system remains CONSISTENT despite crashes.

How? Make each function ATOMIC

For example

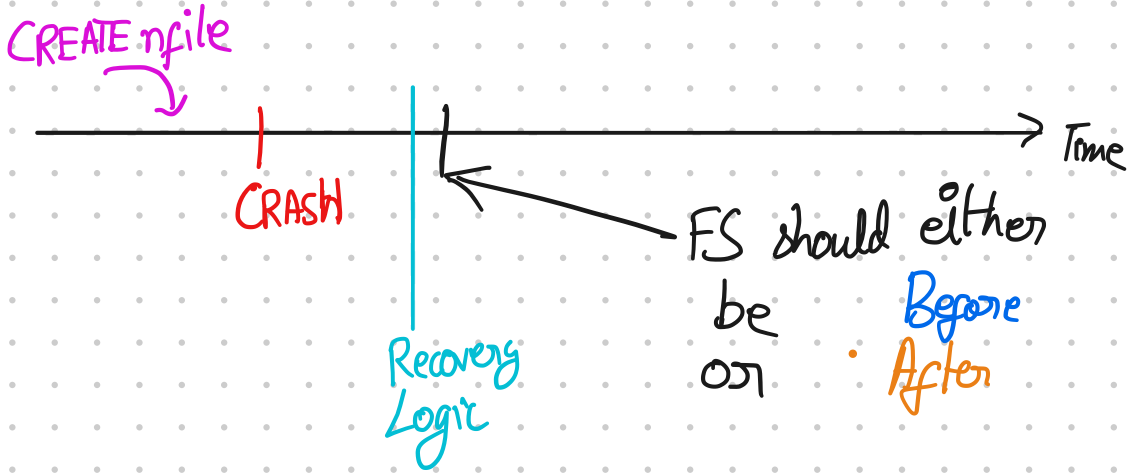
```
(demo) >>> ls
(demo) >>> echo "This is a test" > nfile
(demo) >>> ls
nfile
(demo) >>>
```

Before func
No dirent for nfile
No inode allocated for nfile

↳ CREATE nfile | OP

Inode I allocated for nfile
Dirent with nfile → I Link

after func



Tools we have

1. Each disk block written atomically
2. Operating system decides order of writes
3. Disk capacities are large - can use extra space.

Using the tools

Three approaches

1. Ad hoc
   - ↳ Carefully order operations to enable recovery
2. CoW
3. Journal

Ad Hoc

```
(demo) >>> ls
(demo) >>> echo "This is a test" > nfile
(demo) >>> ls
nfile
(demo) >>>
```

Before op
No dirent for nfile
No inode allocated for nfile

CREATE nfile op

Inode I allocated for nfile
Dirent with nfile → I link
after op

CREATE nfile

① Write inode to disk
② Update directory
③ Update bitmap (to mark inode allocated)

Recovery

① Check directories
↳ If linked inode is not allocated, fix bitmap

Problems

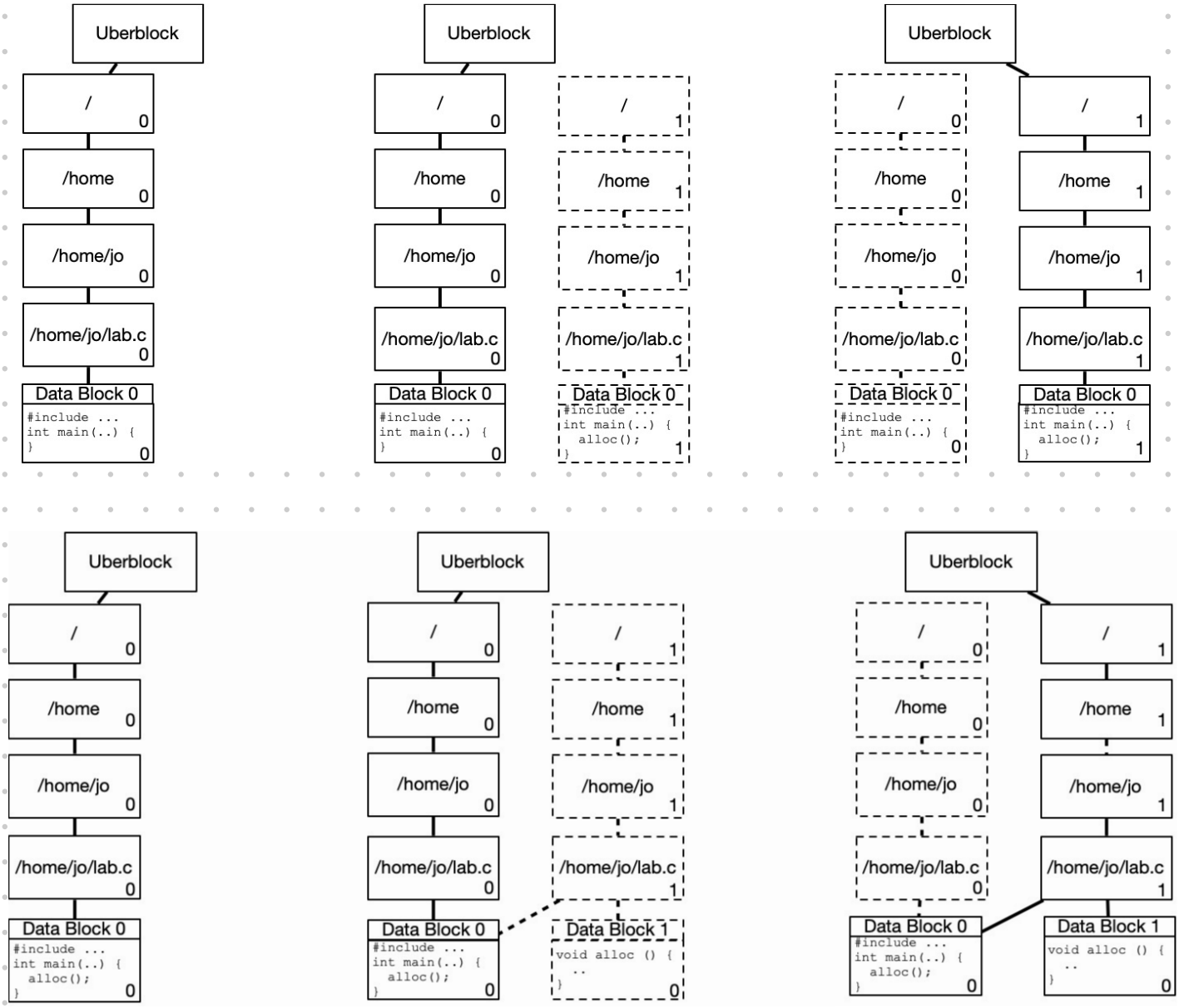
① Hard to get right & maintain correctness
② Recovery expensive —— scan all directories, etc.

Before CoW, Journaling

- Commit point: Time/point in logic after which function's effects are guaranteed to Persist

Copy-on-Write - ZFS, APFS, btrfs, ...

→ Other than superblock (uberblock)
Do not modify any block.
→ Instead → make a new copy when modifications are necessary
→ Commit by updating uberblock to point to new copy.



Benefits
- No recovery logic - always consistent
- No ordering requirements for most changes
- New feature : history

Disadvantage
- Write amplification
- Space overhead
- Need garbage collection

JOURNALING - Ext4, Ext3, NTFS, ...

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)

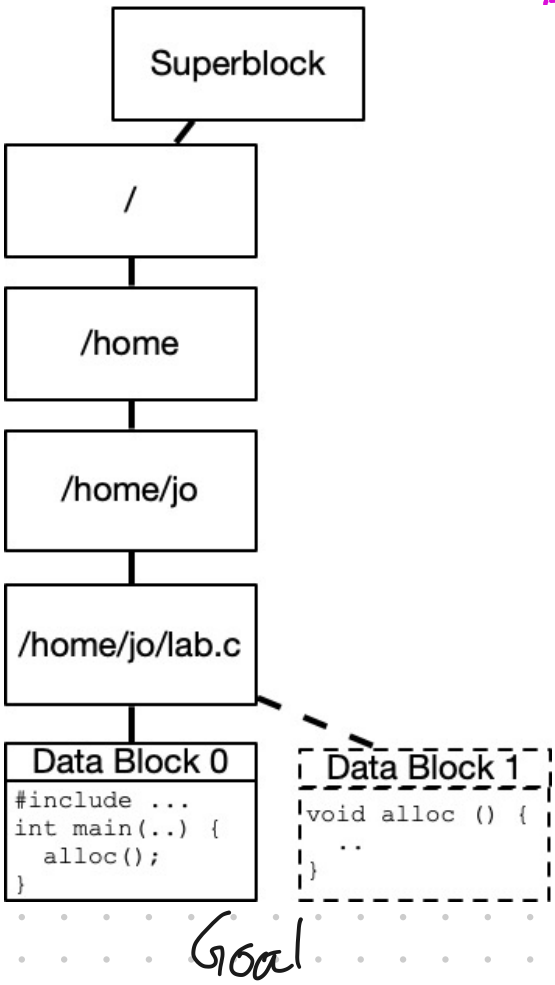
→ Record enough information to

Restore Last committed version (UNDO Logging)

OR

Finish uncommitted operation (REDO Logging)

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



Goal

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

Commit Point →

2. Apply changes.

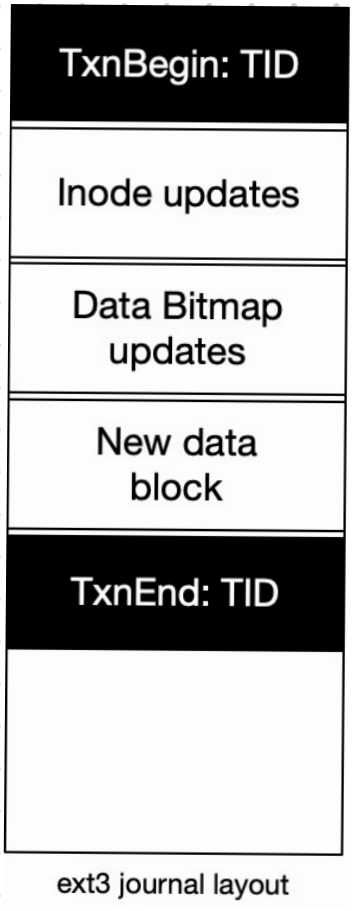
On recovery:

Apply any changes that were logged but not applied.

Problem: Logging changes can require writing to more than

one block — Depends on operation, etc.

Solution: Order log updates so that recovery logic can distinguish b/w operations that were completely logged & ones which were not.



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 —

Redo Logging

Normal Case →
① Log operations using Log Procedure
— Commit Point —
② Apply operations.

Recovery
① Read log to find COMMITTED CHANGES

- How?

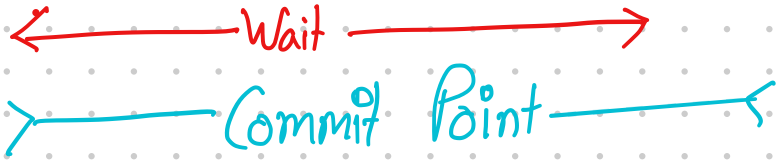
(2) Apply COMMITTED CHANGES.

UNDO Logging - The logging procedure is slightly different

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

Undo vs Redo

Redo + Change committed once logged

- Operations cannot be applied ⇒ written to disk until all ops in func. logged
  ↳ Instead effects in buffer cache
  ⇒ Can need larger buffer cache/ more memory

UNDO + Can apply each op as soon as its undo info is logged
  ↳ Evict from buffer cache earlier

- Func only committed after log + ops complete.

Most file systems just use REDO logging

(Buffer Cache Req. is not a major ...

No one uses just UNDO logging

NTFS: REDO + UNDO — see notes for details.