CS202 : File Systems III : Crash Recovery

- Logistics Reminder:
  - No class on Thu (11/23) or Next Tuesday (11/28)
  - Lab 5 Has New Due Date: 12/8
    → Review Session: 11/27 (Monday) 7—8 PM
- Last Class
  - Directories
  - FFS
- This Class
  - Dealing With Crashes

Crash Recovery

Core Problem

- Disk Writes Persist
  ↳ By design
- Disk Operation Only Atomic At Sector Or Block Granularity

• WHY BAD?

> ls > /tmp/out

(1) Allocate inode for out

↳ Read inode bitmap

→ Update bitmap

C

(2) Modify /tmp to point to allocated inode

- Read /tmp inode to find datablock

- Read datablock to find empty index

- Add link to out

C

(3) Allocate datablock for /tmp/out

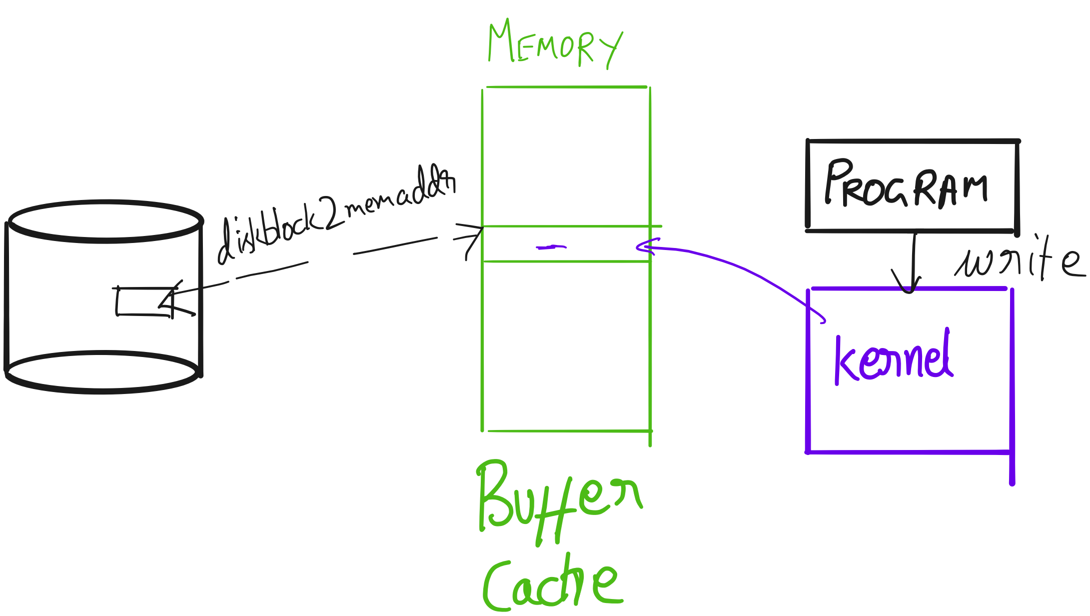
(4) Write to datablock

OBSERVATION: ILL-TIMED CRASHES CAN LEAD TO

- RESOURCE LEAKS
- INCONSISTENT FILE METADATA
- DATA LOSS
- ...

GOAL: COME UP WITH MECHANISMS TO ENSURE DISK CONSISTENCY AFTER CRASH

BUT FIRST, THINGS GET A BIT WORSE



MODIFICATIONS ARE NOT IMMEDIATELY WRITTEN TO DISK.

Can Force Them (fsync(2)) But Expensive.

Why Is This A Problem For Crash Recovery?

Recovery Mechanisms

① Ad Hoc / Fsck

② Copy On Write

③ Journaling

Ad Hoc / FSCK

Assumption: Recovery Logic Will Execute When A Computer Restarts After A Crash

Usually Implemented In A Program Called Fsck

Goal: Ensure Metadata Remains Consistent
→ No Leaked Data Blocks Or Inodes

Non-Goal: Data Consistency
↳ A File's Contents Might

NOT MAKE SENSE

APPROACH: DESIGN FS LOGIC SO THAT fsck CAN REPAIR METADATA AFTER CRASH

> ls > /tmp/out

1. WRITE DATA TO DATA BLOCK

2. WRITE DATA BLOCK REF TO INODE FOR /tmp/out

3. UPDATE INODE BITMAP

4. UPDATE DATA BLOCK BITMAP

5. UPDATE /tmp TO LINK TO /tmp/out

... THIS APPROACH?

PROBLEMS WITH THIS APPROACH?

COPY-ON-WRITE

- CORE IDEA: DO NOT MODIFY* BLOCKS
  COPY INSTEAD

  (ONE EXCEPTION: UBERBLOCK / SUPERBLOCK)

- CAN ACHIEVE BOTH METADATA &
  DATA CONSISTENCY

- How? [Switch To Handout]

- Why This Works
  - Uberblock Updated Atomically
    (One Block)
  - Operation
    - Copies

COMMIT POINT → Uber Block Update

Other Questions

(a) Problems With This Approach?

(b) Benefits Beyond Crash Recovery

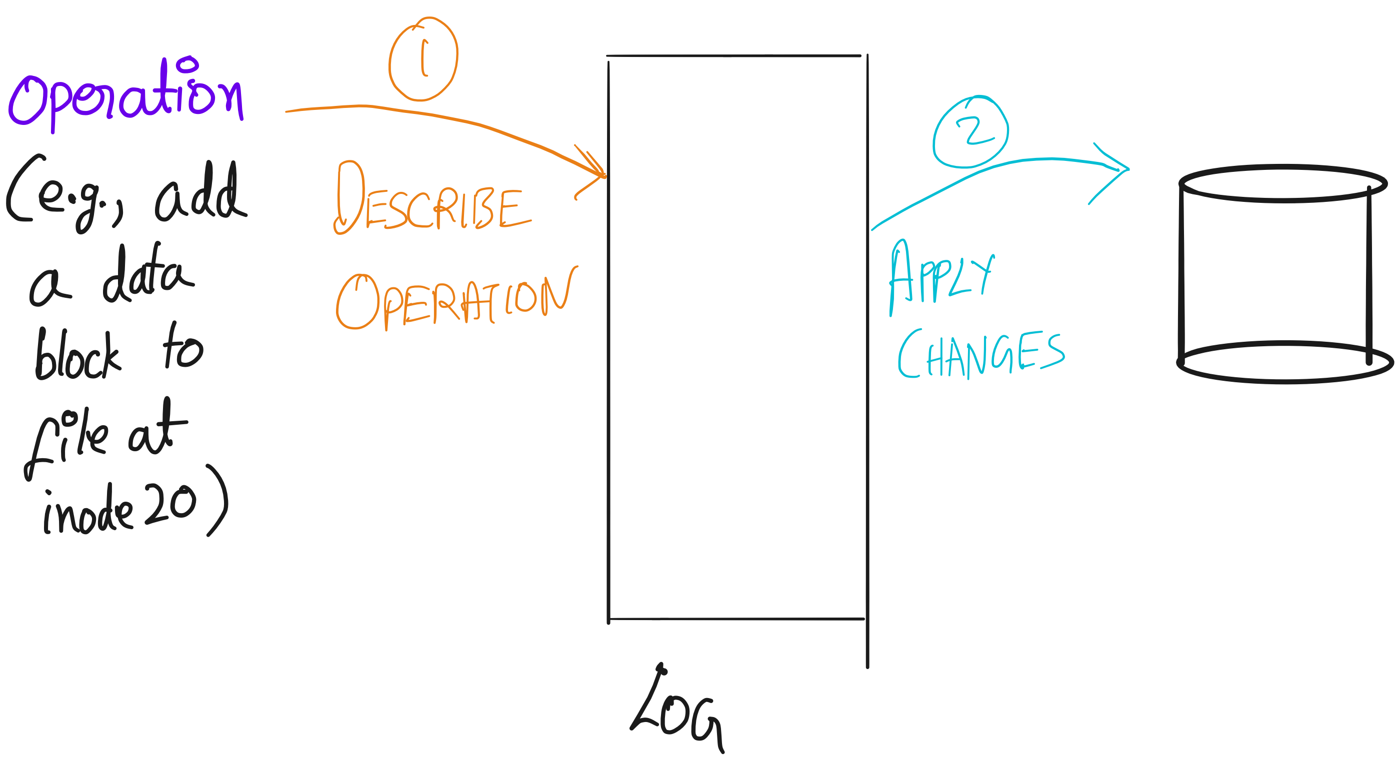
Journaling

- Builds On Ideas That Are Similar To Copy On Write

  Do Not Modify The Only Copy Of The Filesystem

  ⇒ Need A Commit Point After Which Operation Is Guaranteed To Apper

- Difference: Reduce Overheads



Log Requirements

Need To Be Able To Distinguish B/W Ops That Have Been Completely Logged & Those Who Have Not.

Why Might Log For An Op Be Incomplete?

How? [Switch To]

Generally, A Completely Logged Op Is **Commited**

Log Contents

Depends On Type Of Log

ⓐ Redo Logging

Log Records How Op Changes Disk

Ex → [illegible]

(i) Allocate inode for out

(i) Allocate inode ...

↳ Read inode bitmap
→ Update bitmap

(ii) Modify /tmp to point to allocated inode

(iii) Allocate data block

...

RECOVERY LOGIC APPLIES CHANGES FROM ANY COMPLETELY LOGGED OP

Q: WHEN IS IT SAFE TO APPLY CHANGES TO DISK (OTHER THAN DURING RECOVERY)?

(b) UNDO LOGGING

LOG RECORDS HOW TO UNDO EACH STEP IN AN OP

(i) Allocate inode for out

↳ VALUE OF BITMAP BEFORE ALLOCATION

ALLOCATION

(ii) Modify /tmp to point to allocated inode

↳ OLD VALUE OF /tmp

...

RECOVERY LOGIC: UNDO STEPS FROM ANY OP THAT IS NOT COMPLETELY LOGGED

Q: WHEN IS IT SAFE TO MARK THAT AN OP IS COMPLETELY LOGGED?

Q: WHEN IS IT SAFE TO APPLY CHANGES TO THE DISK?

REDO VS UNDO LOGGING

REDO	UNDO

PRO

CON

REDO + UNDO LOGGING

- SEE NOTES

WHAT TO LOG?

- METADATA: INODES, BITMAPS, ETC.
- DATA:
- ...

CS202 Handout 11

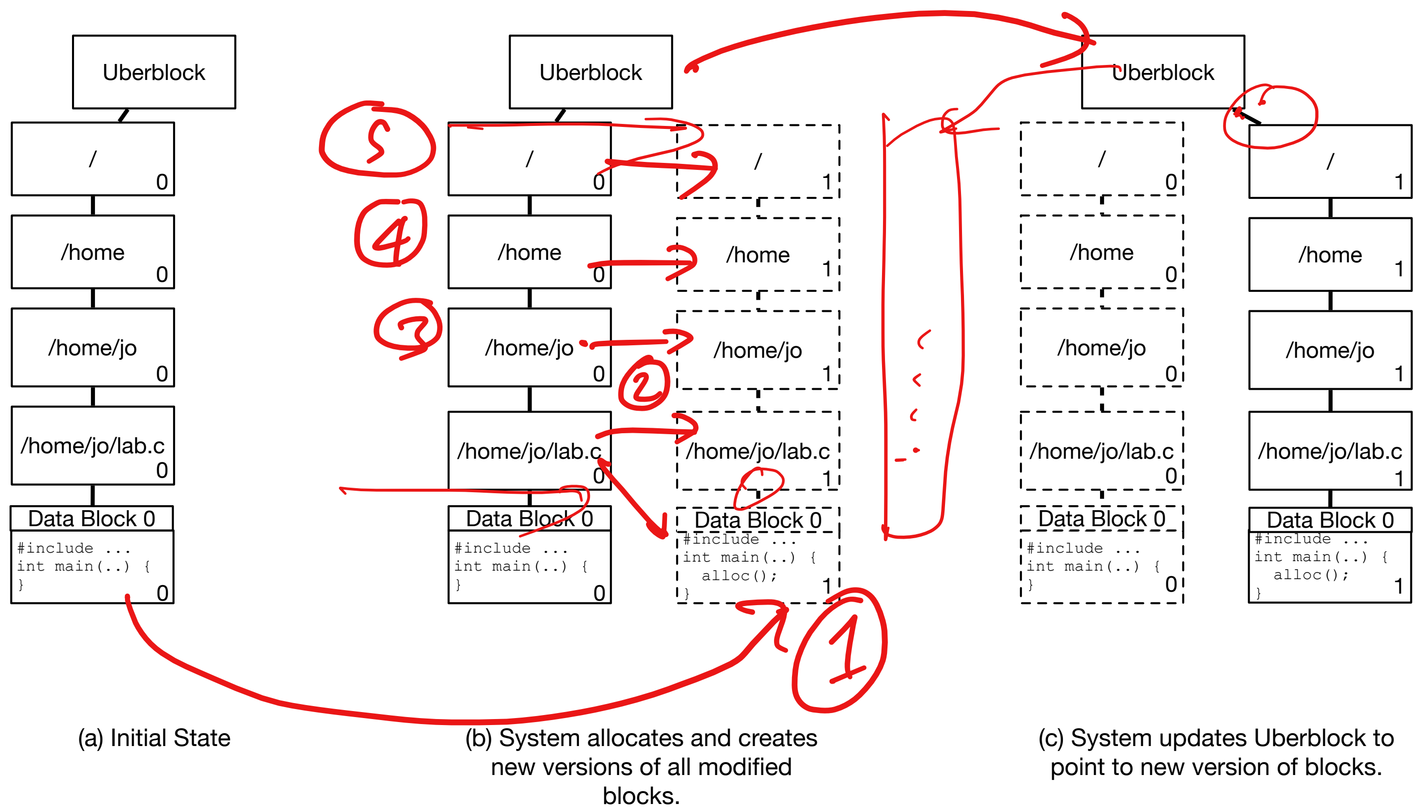


Figure 1: Copy-on-write filesystem: modifying a data block

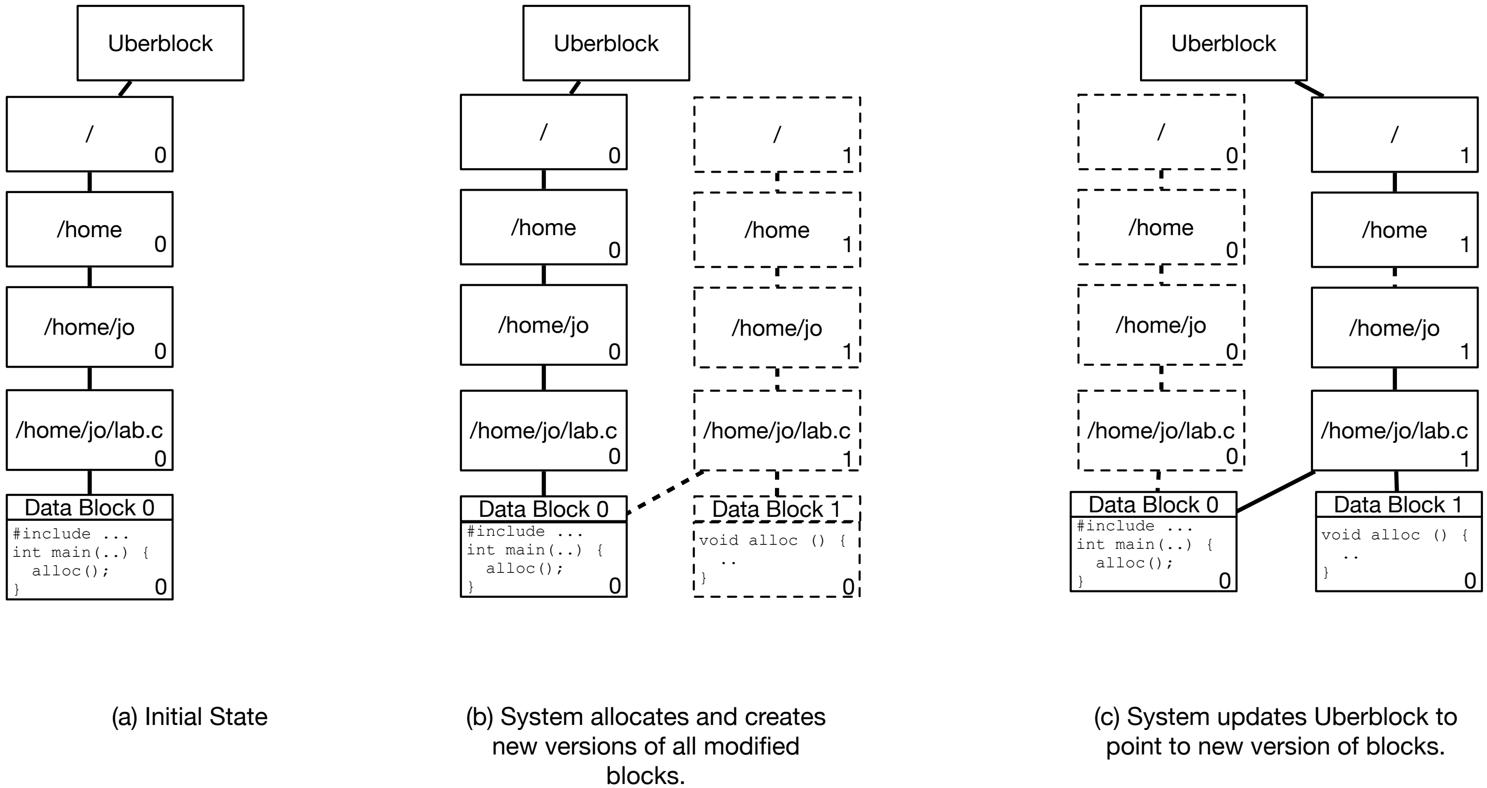


Figure 2: Copy-on-write filesystem: adding a data block

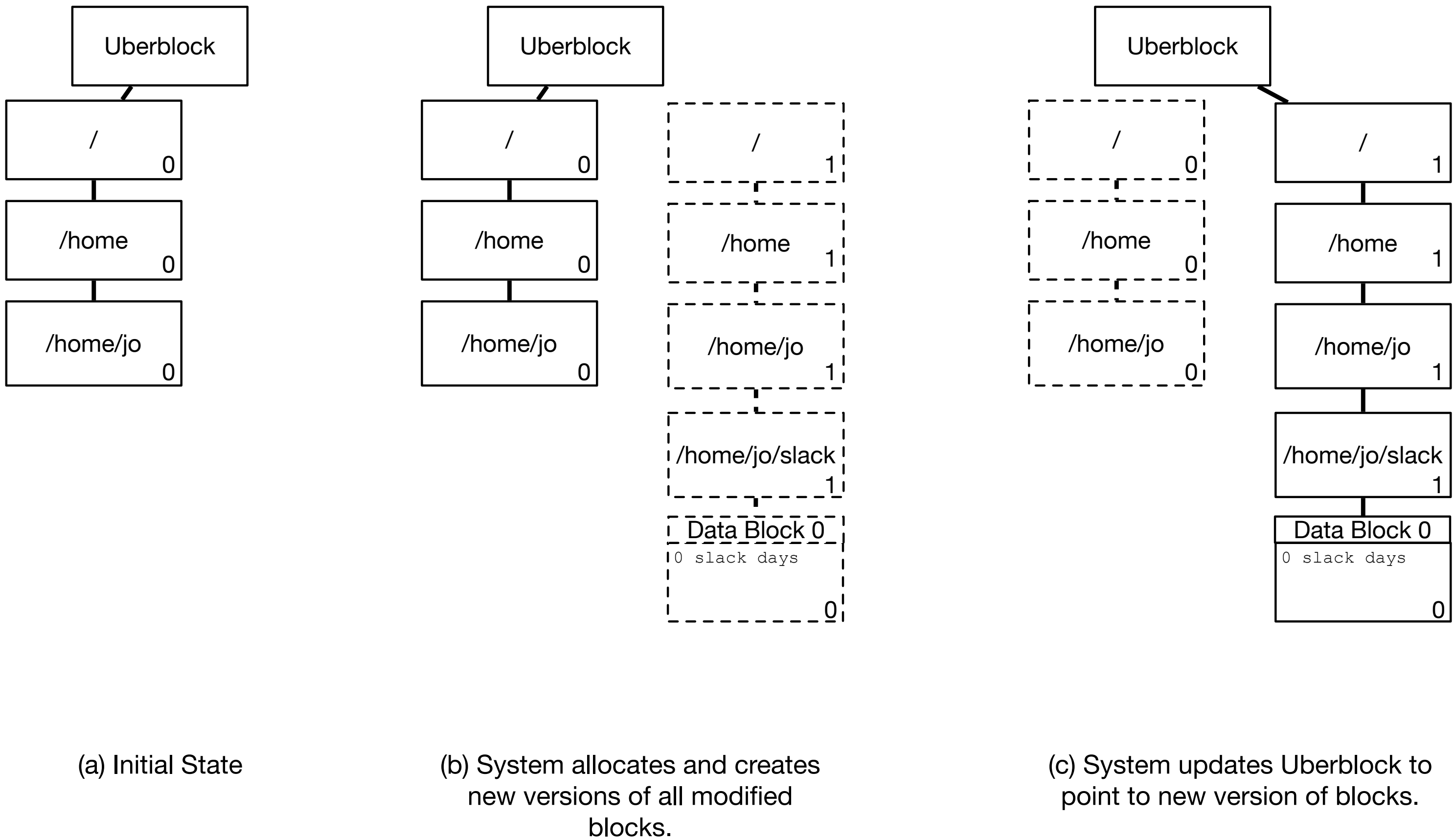


Figure 3: Copy-on-write filesystem: creating a file

Figure 4: Redo logging in a filesystem