CS 423
Operating System Design: LFS & COW File Systems
Apr 23

Ram Kesavan

LOGISTICS

MP3 grades + breakdown released

- Resubmission deadline Apr 27, 11:59 CT
- Only if you did not resubmit MP2

Midterm papers: pickup from my office during OH

MP4 due by May 6, 11:59 CT

Last lecture: 5/5 will be a review

Finals: 5/13 at 7-10pm, DCL 1320 (next door)

AGENDA / LEARNING OUTCOMES

Log-structured File System (LFS)

- Journal is the file system!

Enterprise-grade COW File Systems

- Eg. WAFL, ZFS, BtrFS

RECAP

Journaling or WAL

WAL: write ahead logging

Foundational database technique for “modify” ops

Basic idea:

- write a “note” to a well-known location about the op
  - while modifying in-memory file system
- once “note” is persisted
  - ack the op to the client
  - then, modify the persistent file system (aka checkpointing)
- if crash, replay yet-to-be-checkpointed notes on restart
  - avoid an fsck-style full file-system walk

Tradeoff: additional work during op→reduce work at restart & **full** recovery

Note: fsck avoided on restart, but still needed when file system corruption occurs!

Journaling

Journal/Log: circular append-only structure

- 3 pointers: checkpointed, committed, current

The “modify” op

1. executes the op to generate in-memory dirty state + writes entry to journal
2. acks the op
3. checkpoints the dirty state, then reclaims journal entry

Typically hold off on #3

1. checkpoint many ops together
2. always more efficient to write to storage in a larger batch
3. but, need to hold down resources for a longer time:
   a. more in-memory dirty state
   b. more journal entries that cannot be reclaimed

Replay

Physical logging: log resultant “dirty” content and the location of those blocks

Replay: needs to know where to write the logged blocks

Op/Logical logging: minimal info about the op and any allocated inode-num or block-number

Extra info needed for replay idempotency

Else, eg.

op: Give me an example

END RECAP

Large Checkpoint

Most file systems: delay checkpoint to persistent file system

- Avoid writing multiple copies of the same metadata blocks
- Checkpoint many transactions together
- **Trade-off**: longer the delay, better the amortization but larger the footprint (dirty state in memory & journal entries)

Journal: write a *compound transaction*

- Several ops as one large compound transaction
- TxB followed by
  - metadata about all ops in that compound transaction
  - modified metadata and data blocks
- terminated by TxE

Log-structed file systems (LFS): the journal is the file system!

LFS Motivation

More efficient to write sequentially to storage than randomly

- Sequential writes work well with RAID 4 and 5 too

Let's avoid/minimize random writes

Log-appends are inherently sequential

- Let's make the log the file system!
- Entire storage space used as a circular log

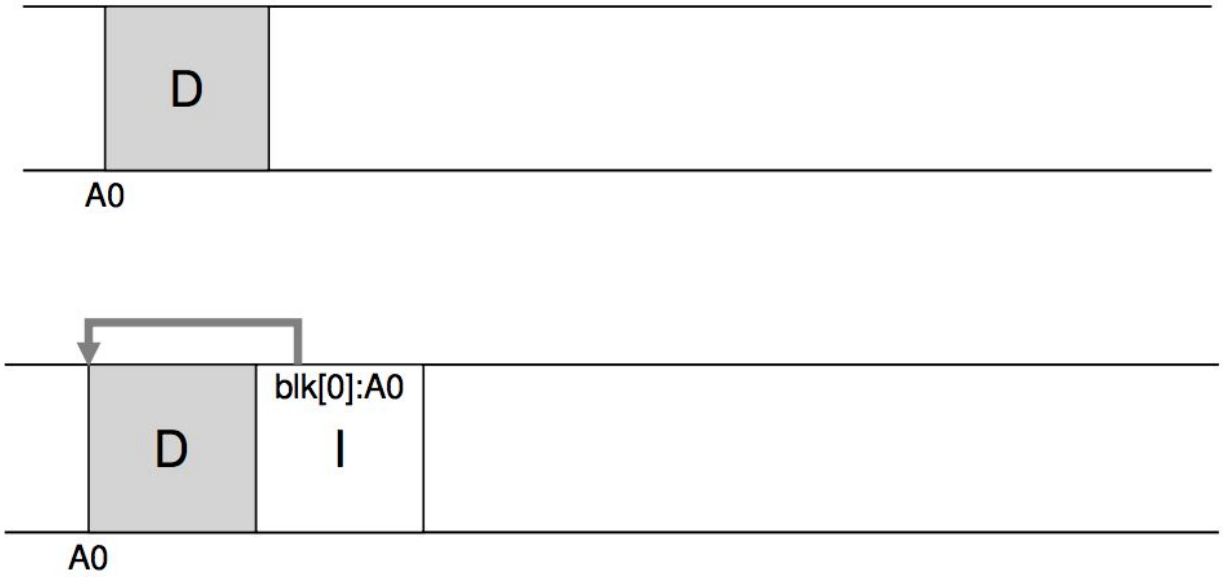
Idea: persist modifications to the file system as large compound transactions

- Called **segments** (sized for efficient writes)
- Buffer segment-worth of dirty state in memory; then write it

Never overwrite the old data; COW (copy-on-write) everything

- Except the superblock; aka Checkpoint Region (CR)

WHERE DO INODES GO?



SEGMENT

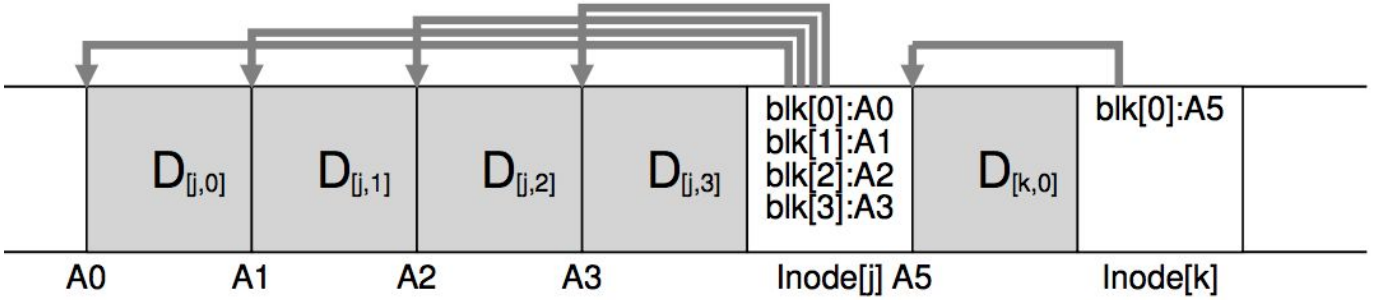
Example segment with 2 write ops

- A segment is much larger (few MB)
- Amortization within segment: only 1 version of an inode

Segment header with metadata:

- length of segment
- portion of imap* that points to inodes in this segment

Superblock points to linked list of segments



*imap: next slide

LFS: Other Differences

No allocation structs: data or inode bitmaps

- Free blks? file system comprises free and used segments
- Free inum? imap structure that maps inum → latest inode location

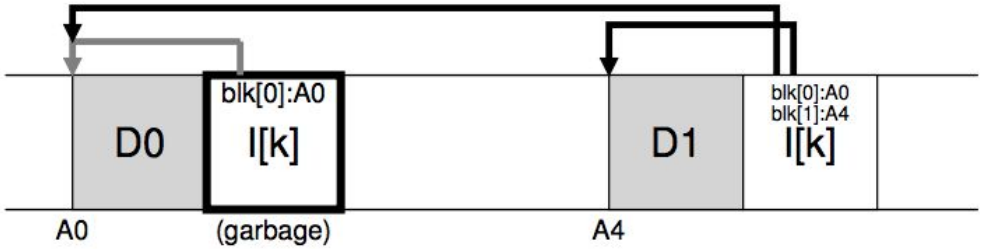
So, how do reads work?

- Inodes are scattered all over the place!
- Uses imap to find latest location of inode; inum→block# with latest version of inode
- The superblock (CR) points to inode-map

Claims (no longer true):

- Sufficient memory in the system that imap can be fully cached
- Random reads from LFS require same #IOs as from other file systems

COW & GARBAGE



COW file system: writes result in stale/unneeded blocks
Old versions of inodes, data blocks, etc.

Garbage is getting created constantly as file system gets modified!

Garbage Collection

COW file system: writes result in stale/unneeded blocks

Old versions of inodes, data blocks, etc.

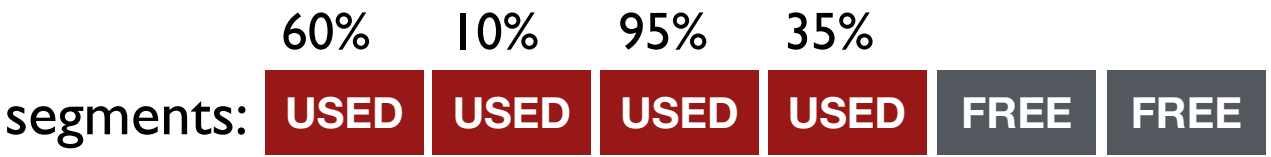
In traditional file system

bitmap value for blk is set to 0

LFS reclaims **one to few segments at a time**

- So future sequential (segment) writes can be efficient
- But, a segment may be partially valid

Garbage Collection



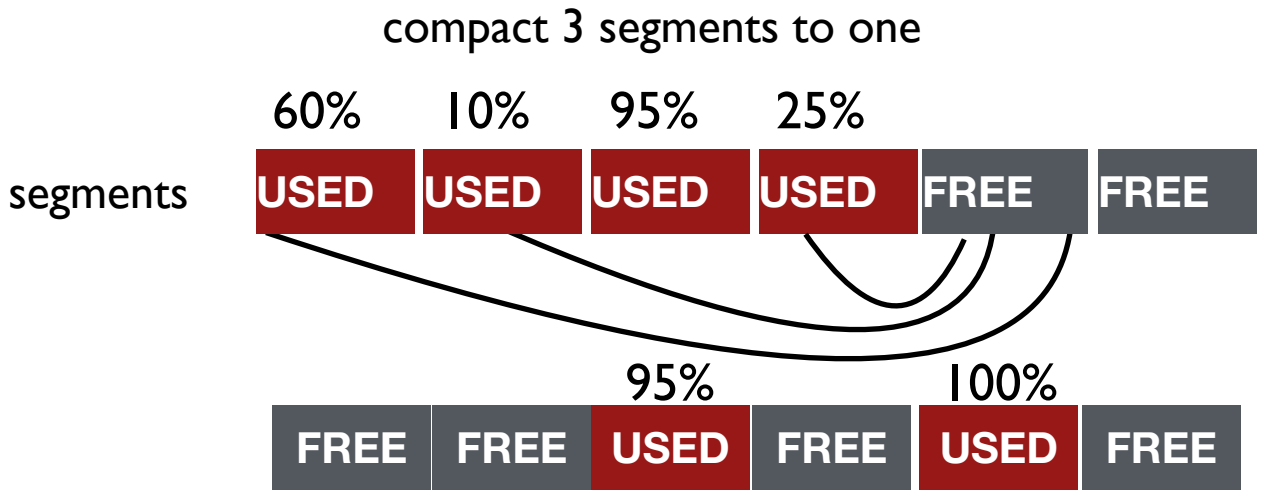
Segment Cleaner

Job: ensure sufficient empty segments

When moving data blocks, copy new inode to point to them

When moving inode, update imap to point to it

GARBAGE COLLECTION



Segment Cleaner: Read Ch 43

Mechanism: Pick M segments to compact into N segments; M > N

How to detect usage in a segment?

Policy: How to pick those M segments (cold vs hot, fullness, etc.)

CRASH RECOVERY

LFS does not prevent data loss

How frequently to write out segments? Trade-off

- Less frequently: lose more data, but better write performance

Need to recover (and read into memory) the imap on restart

LFS periodically (every ~30s) writes the checkpoint region (CR), which contains

- Pointers to imap sections
- Linked list of segments

Restart:

- Start with the CR's state (a bit stale): imap & linked-list of segments
- Walk **fully committed** segments past the tail, i.e., the most recently written segments
- Freshen the imap using imap data from that walk

Two cases of crashes: Read CH43

- while writing to a segment or the CR: Read CH43

LFS SUMMARY

Typical journaling file systems:

- Persistent file system layout is optimized for future reads
- Write to journal, and then write to file system

LFS

- Place data where it's fastest to write (in journal)
- Segment clean to generate free space
- Assume: future reads hit in memory cache

LFS is a Copy-On-Write (COW) file system

- Next: A more sophisticated COW file system
  - Eg. WAFL, ZFS, btrfs

ENTERPRISE-GRADE FILE SYSTEMS

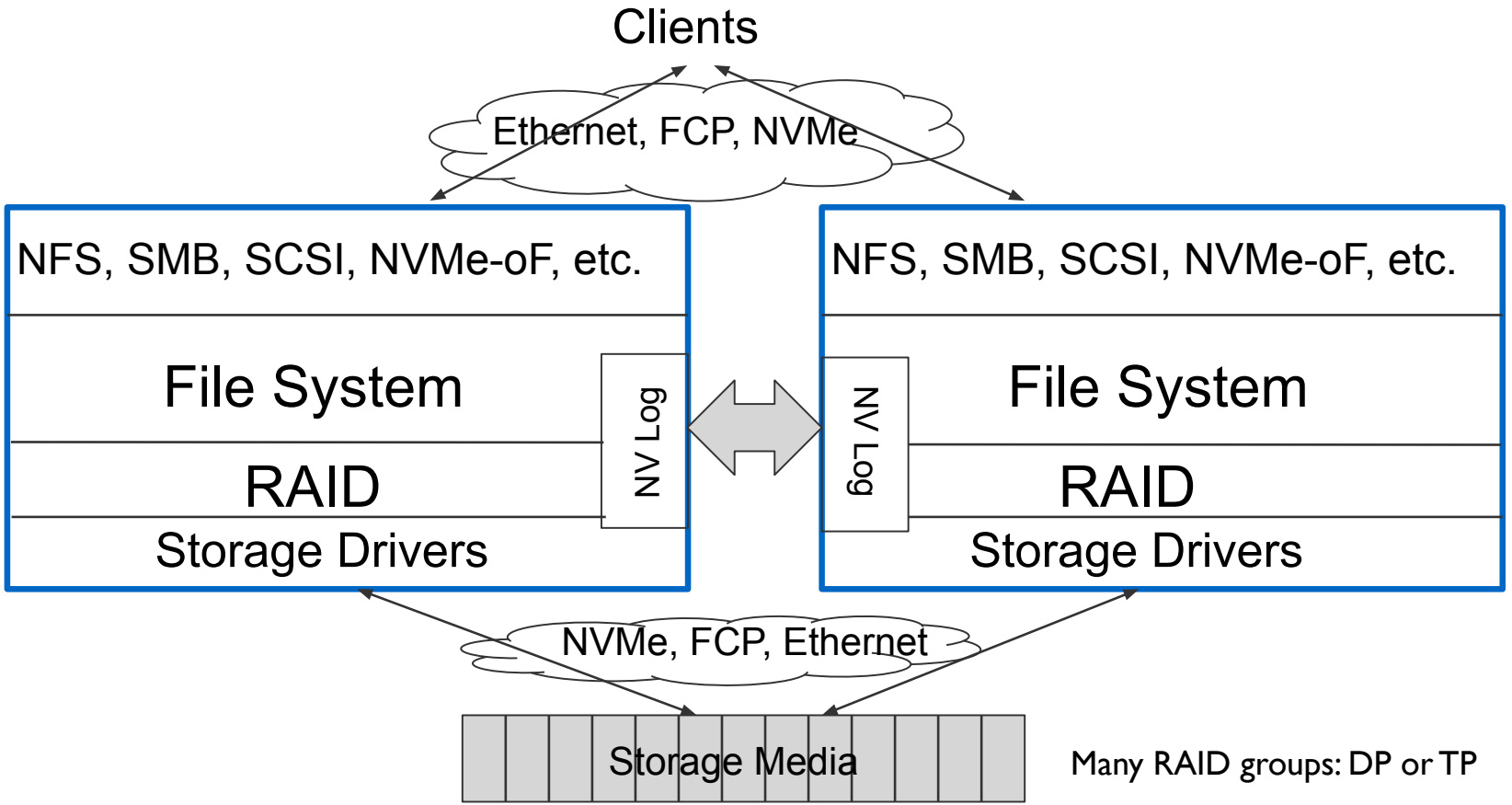
Deployed as HA (high availability) systems

- 2-node or 4-node
- Expensive ($million), low-latency, fast sub-section takeover
- On-prem or public cloud

WAFL (NetApp); but ZFS or BtrFS would look similar

- Exports 100s of file systems, 100s of TB, billions of files
- Many large workloads running at the same time
- NFS/SMB file sharing
- LUNs with FC/iSCSI/NVMeoF access

2-NODE HA SYSTEM



FILE SYSTEM LAYOUT

WAFL file system is a tree of blocks

- everything is a file, including all metadata
- metadata files have well-known inums
- each dirty buffer written to new (free) block in storage, except superblock

Modify op:

- dirties in-memory state
- creates log entry in NVRAM, is mirrored to partner's NVRAM
- then ack'ed

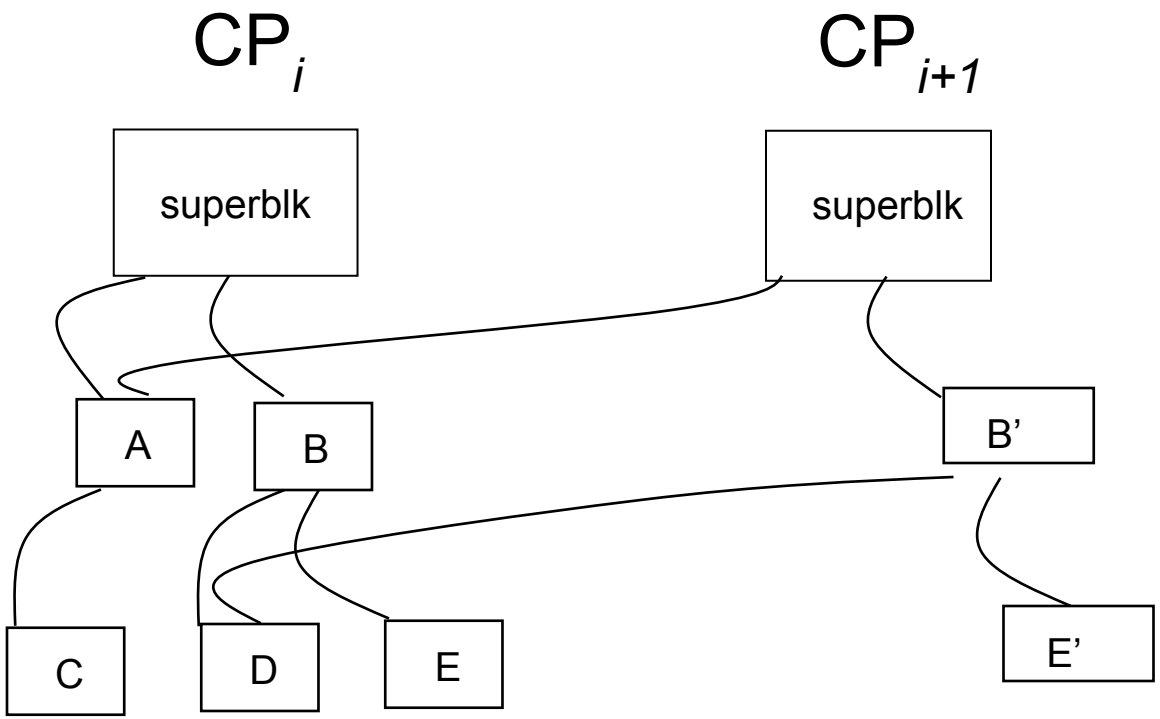
Large checkpoints: ~GB worth of dirty buffers

- ~2s to 5s long
- all dirty buffers can be persisted in any order; superblock written last
- COW: so persistent file system is always self-consistent

PICTURE OF FILE SYSTEM TREE

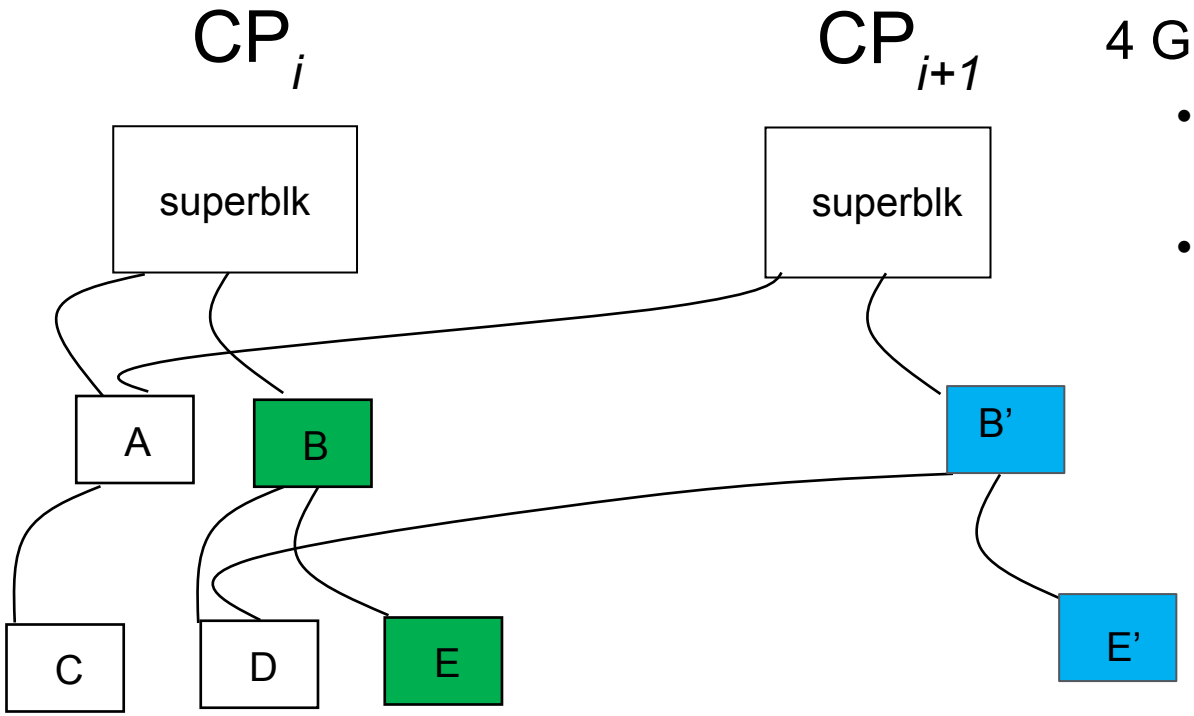
Original WAFL paper: https://rcs.uwaterloo.ca/papers/wafl.pdf

COW: Atomic Checkpoint



CHECKPOINTS, JOURNAL, & DOUBLE BUFFERING

Block Bitmap Updates



4 GB/s writes need

- 1 M/s allocations of new blocks
- 1 M/s frees of unused blocks

ACM paper

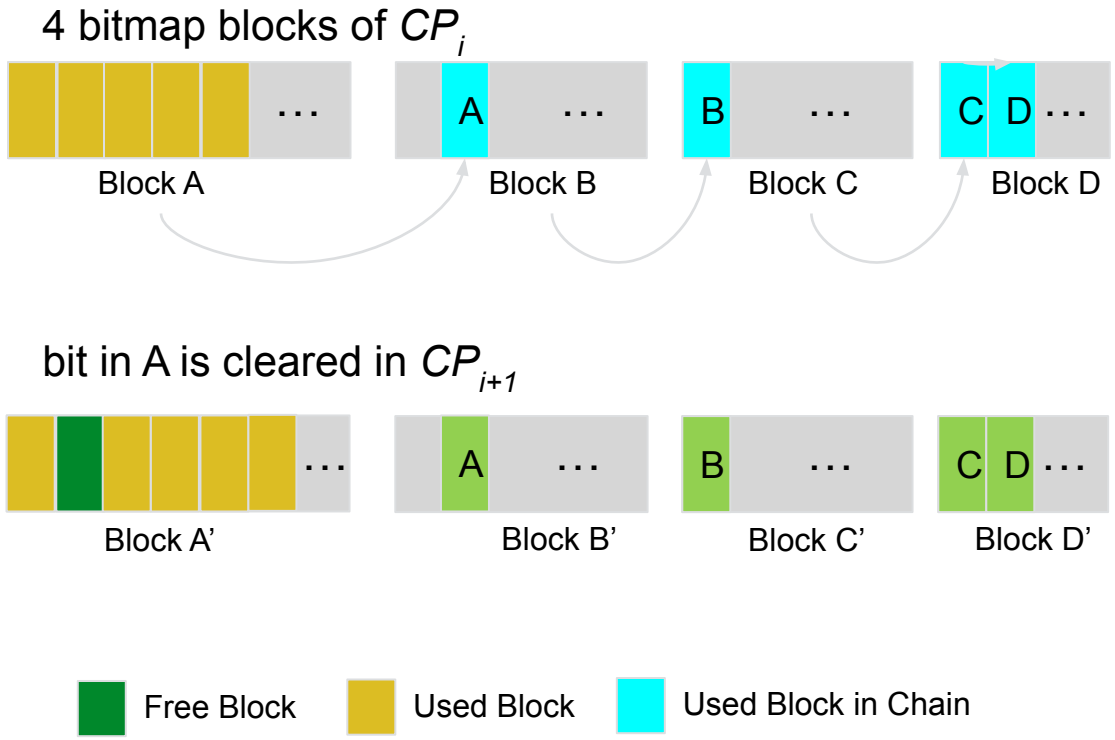
Block Bitmap: used(1)→free(0)

Long CPs due to:

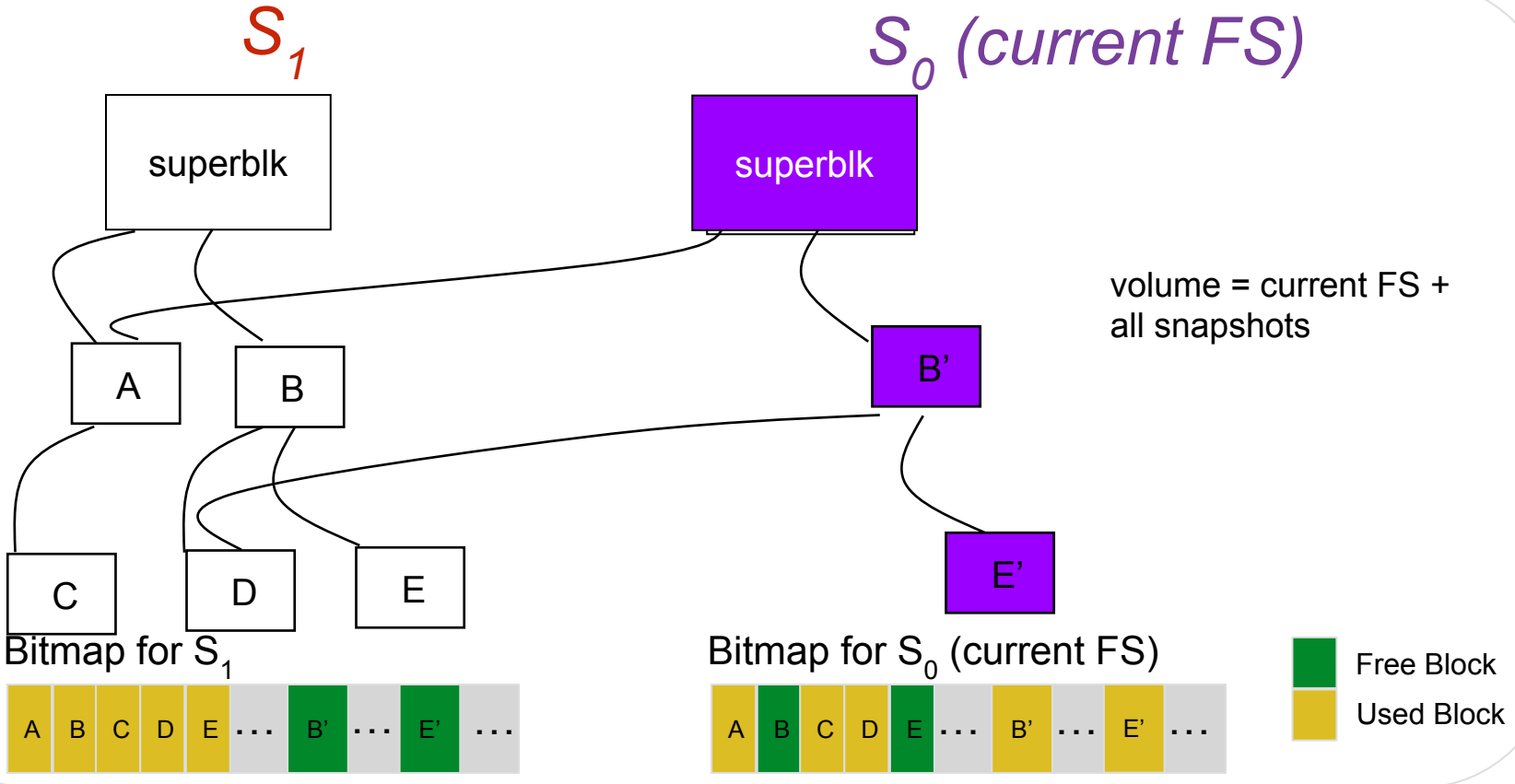
- random frees → more dirty block bitmap blocks
- long activemap chains

Long CPs hurt file system write throughput

- too many dirty bufs in memory
- journal gets filled up



SNAPSHOTS


S_1
S_0 (current FS)
superblk
superblk
A
B
B'
C
D
E
E'
volume = current FS + all snapshots
Bitmap for S_1
A B C D E ... B' ... E' ...
Bitmap for S_0 (current FS)
A B C D E ... B' ... E' ...
Free Block
Used Block

SUMMARY

Modern COW file systems (WAFL, ZFS, BtrFS)

- Are log-structured like LFS
- But keep a separate journal to guarantee zero data loss

Enterprise-grade storage systems

- Can be built using COW file systems
- Large checkpoints→efficient sequential writes to RAID4-ish storage

Many features:

High performance, snapshots, clones, compression, deduplication, worm, encryption, tiering, replication, etc.

NEXT

Remote File Access

Client-Server architecture
NFS/SMB

Distributed File Systems

GFS + Colossus