# CS 423
# Operating System Design: Persistence: LFS
# 04/30

Xuhao Luo/Ram Alagappan

# LFS PERFORMANCE GOAL

Motivation:

- Growing gap between sequential and random I/O performance
- RAID-5 especially bad with small random writes

Idea: use disk purely sequentially

Design for writes to use disk sequentially – how?

# LFS STRATEGY

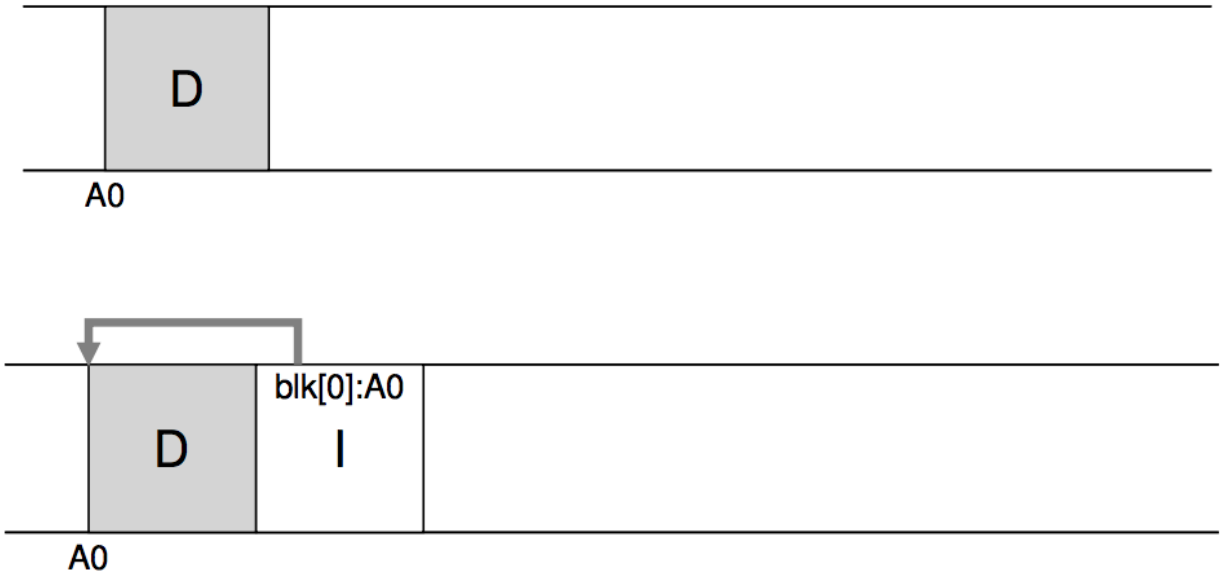
File system buffers writes in main memory until "enough" data

- How much is enough?
- Enough to get good sequential bandwidth from disk (MB)

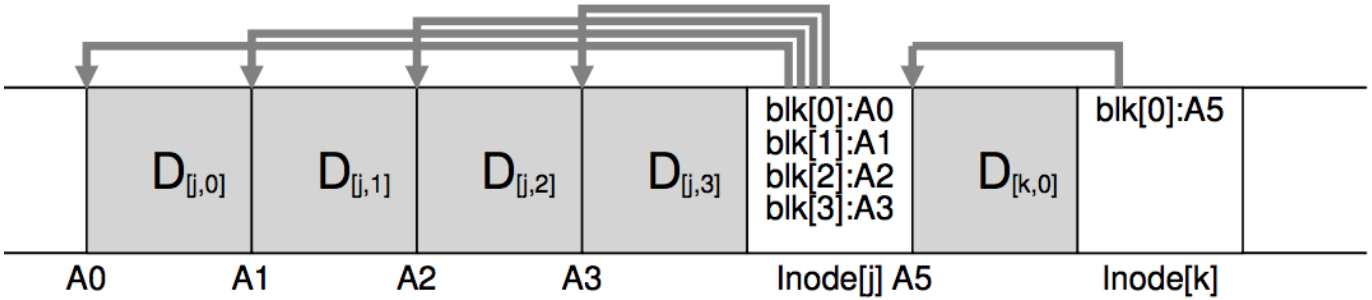
Write buffered data sequentially to new **segment** on disk

Never overwrite old info: old copies left behind

# WHERE DO INODES GO?



# BUFFERED WRITES



# What else is diff from VSFS?

What data structures has LFS removed?

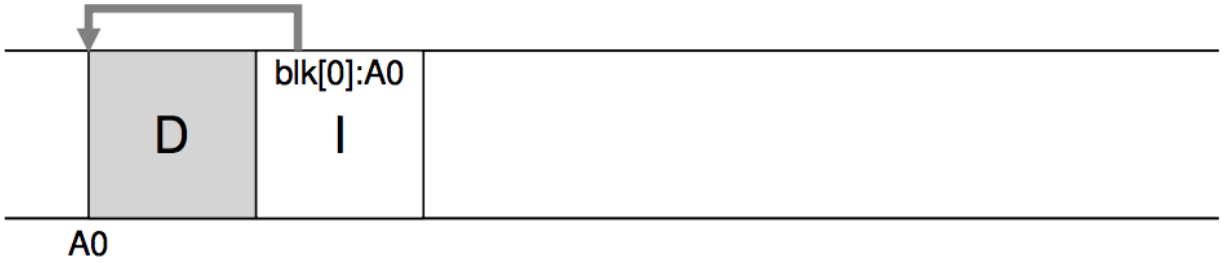
allocation structs: data + inode bitmaps

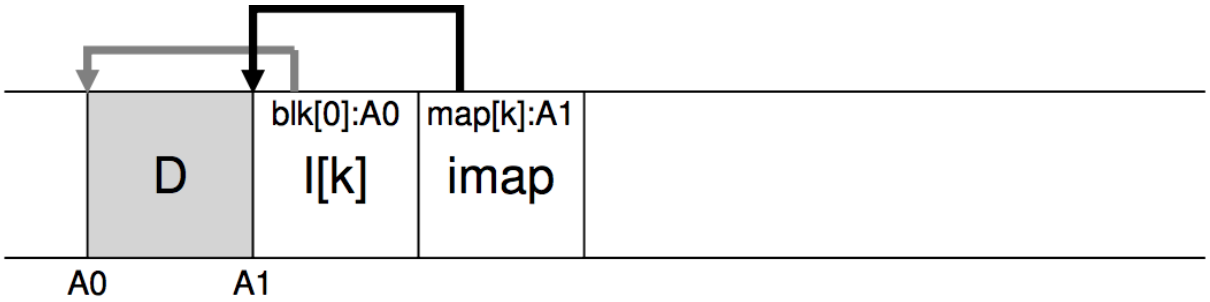
How to do reads?

Inodes are no longer at fixed offset

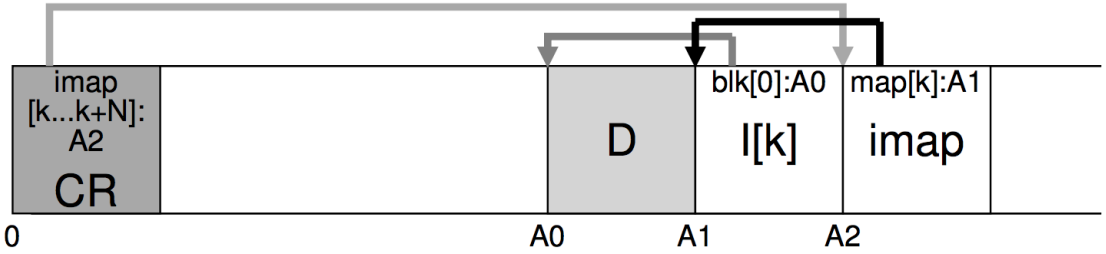
Use imap structure to map:
inode number => inode location on disk

# IMAP EXPLAINED



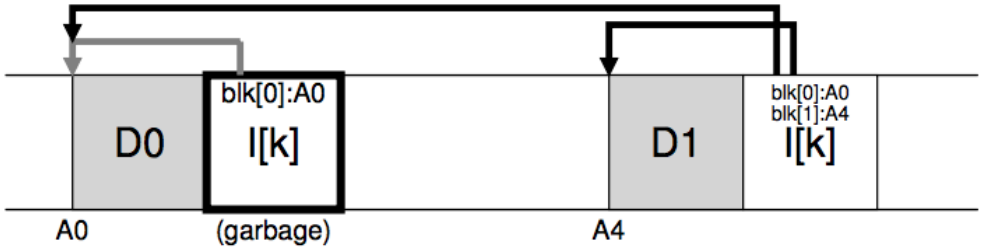



# READING IN LFS




1. Read the Checkpoint region
2. Read all imap parts, cache in mem
3. To read a file:
    1. Lookup inode location in imap
    2. Read inode
    3. Read the file block

# GARBAGE COLLECTION



# WHAT TO DO WITH OLD DATA?

Old versions of files ☐ garbage

Approach 1: garbage is a feature!

- Keep old versions in case user wants to revert files later
- Versioning file systems
- Example: Dropbox

Approach 2: garbage collection

# GARBAGE COLLECTION

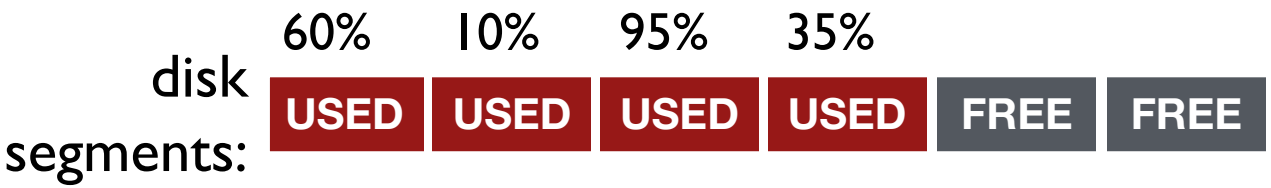
Need to reclaim space:

1. When no more references (any file system)
2. After newer copy is created (COW file system)

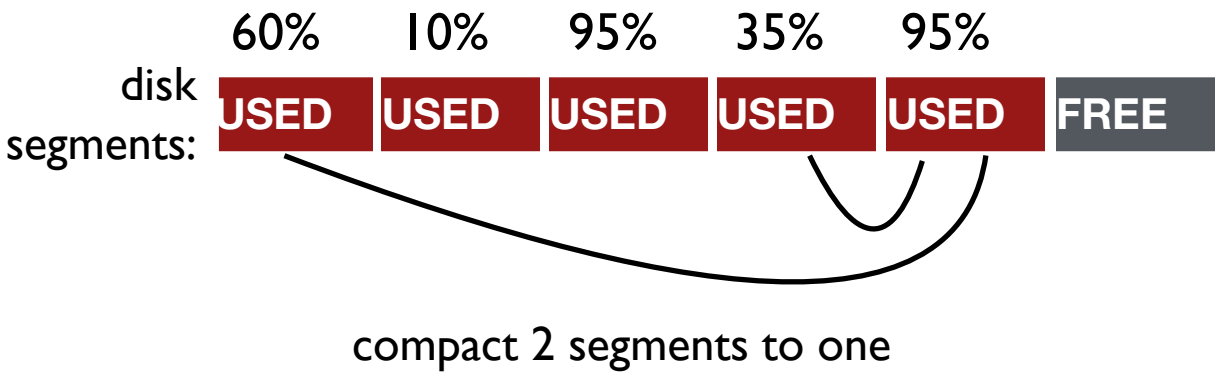
LFS reclaims **segments** (not individual inodes and data blocks)

- Want future overwites to be to sequential areas
- Tricky, since segments are usually partly valid

# GARBAGE COLLECTION



# GARBAGE COLLECTION




When moving data blocks, copy new inode to point to it
When move inode, update imap to point to it

# GARBAGE COLLECTION

General operation:

Pick M segments, compact into N (where N < M).

Mechanism:

How does LFS know whether data in segments is valid?

Policy:

Which segments to compact?

# GARBAGE COLLECTION MECHANISM

Is an inode the latest version?

- Check imap to see if this inode is pointed to
- Fast!

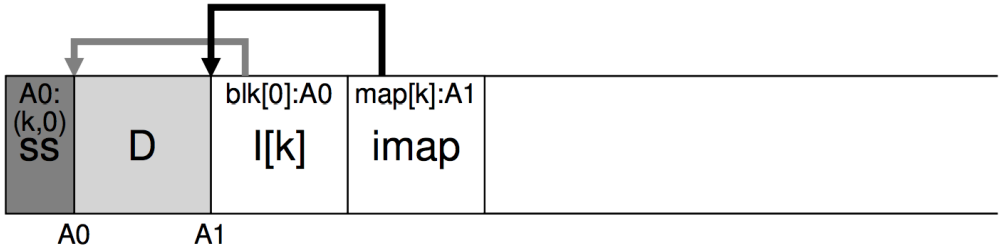
Is a data block the latest version?

- Scan ALL inodes to see if any point to this data
- Very slow!

How to track information more efficiently?

- **Segment summary** lists inode and data offset corresponding to each data block in segment (reverse pointers)

# SEGMENT SUMMARY




```
(N, T) = SegmentSummary[A];

inode = Read(imap[N]);

if (inode[T] == A)
        // block D is alive
else
        // block D is garbage
```

# GARBAGE COLLECTION

General operation:

Pick M segments, compact into N (where N < M).

Mechanism:

Use segment summary, imap to determine liveness

Policy:

Which segments to compact?

- clean most empty first
- clean coldest (ones undergoing least change)
- more complex heuristics…

# CRASH RECOVERY

What data needs to be recovered after a crash?

- Need imap (lost in volatile memory)

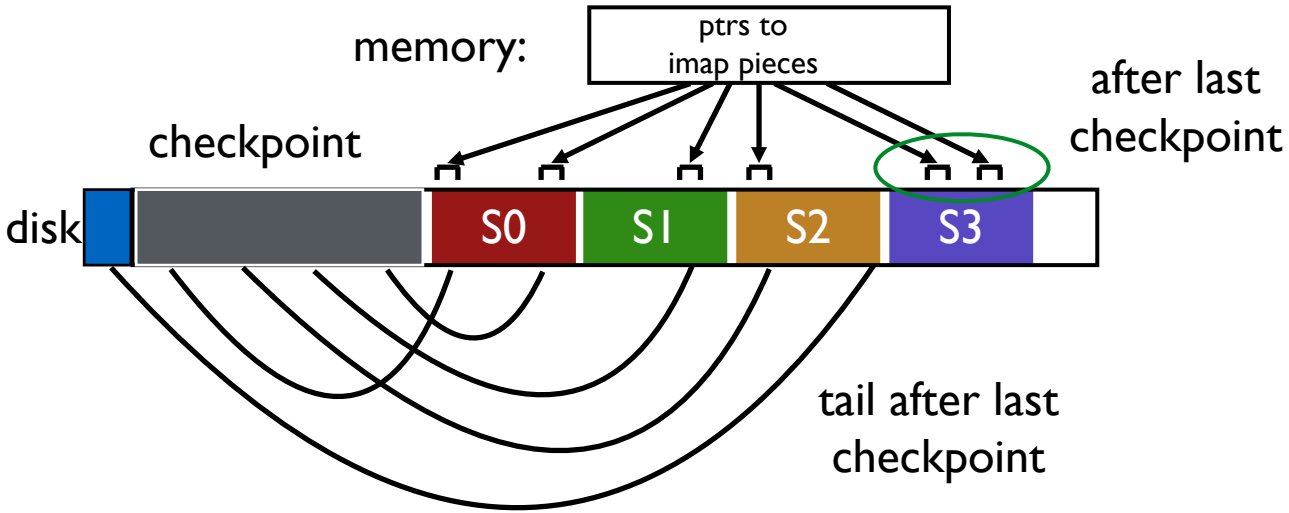
Better approach?

- Occasionally save to checkpoint region the pointers to imap pieces

How often to checkpoint?

- Checkpoint often: random I/O
- Checkpoint rarely: lose more data, recovery takes longer
- Example: checkpoint every 30 secs

# CRASH RECOVERY



# CHECKPOINT SUMMARY

Checkpoint occasionally (e.g., every 30s)

Upon recovery:

- read checkpoint to find most imap pointers and segment tail
- find rest of imap pointers by reading past tail

What if crash during checkpoint?

# CHECKPOINT STRATEGY

Have two checkpoint regions
Only overwrite one checkpoint at a time
Use checksum/timestamps to identify newest checkpoint

# LFS SUMMARY

Journaling:

Put final location of data wherever file system chooses (usually in a place optimized for future reads)

LFS:

Puts data where it's fastest to write, assume future reads cached in memory

Other COW file systems: WAFL, ZFS, btrfs