Instructor Notes

Rosenblum, M. and Ousterhout, J.
The Design and Implementation of a Log-Structured File System
ACM Transactions on Computer Systems, Vol. 10, No. 1, February 1992, pp. 26-52.

1. What is the technology motivation for a log-structured file system? What is the workload motivation? Why do existing file systems perform poorly under these circumstances?

Technology:

- Processors faster → more pressure on I/O
- Seek time is not improving in disks
- Main memory increasing → buffer cache
  - absorbs many reads from disk
  - write out writes in group

Workloads

- Many small files, I/Os - creating, deleting (meta-data) ↓ writing

Problems:

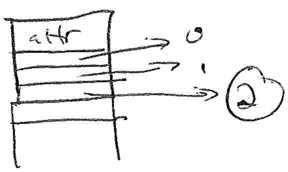
- Info allocated across disk (inodes vs. data)
  - many reads/writes w/ seeks
  - ~~updating file data...~~ creating file
- Synchronous writes to meta-data

2. How does one read a file in LFS? How does LFS know where the inode map is on disk? How does LFS avoid performing extra disk reads?

Basic LFS idea: Write everything sequentially to log - all metadata + data - that is only place info lives

Read: read block 2 of file

inode



read block from addr

How to find inode?

FFS: Fixed location depending on size of fs

LFS: Level of indirection

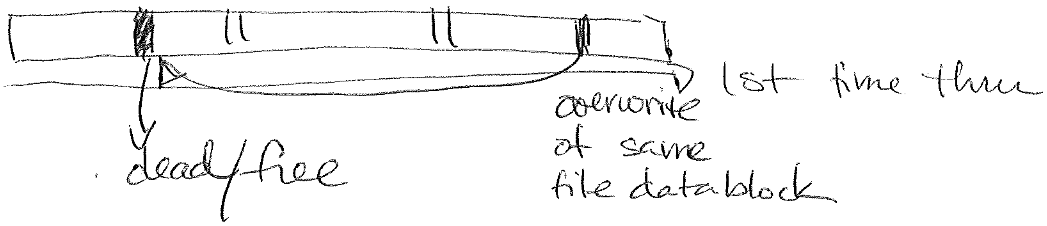
i-node map

Where is i-node map? (can be divided into chunks)

- Usually in memory (fast)
- Location given in <u>checkpoint region</u> (fixed loc)

3. A log-structured file system must have free space to write the log. How would a threaded log work? Why is it a bad idea? How would compaction with a single log work? Why is it a bad idea?

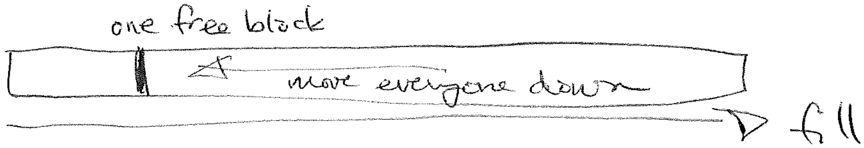
1) Threaded: (hole plugging)



2nd time: use dead/free space

Bad? Fragmentation

2) Pure Compaction



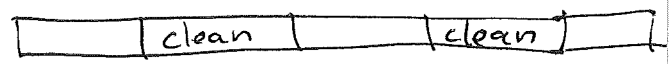
1)

2) clean - copy live data to beginning

Bad? Constantly re-reading/copying data

4. How can these two ideas for free space management be combined? How should segment size be chosen?

Hybrid



- Threaded segments
- Sequential within segment
  - try to put old data in same segment
  - clean individual segments

Size: Large enough to amortize seek cost

~1MB

5. What happens during segment cleaning? How can the cleaner know whether or not a particular block is live? (Why is it possible for there to be multiple segment summary blocks per segment?) Why is a version number useful? (What needs to be updated when a segment is cleaned?)

1) Read # segments into memory
2) Identify live data
3) Write live data back to smaller # of segments (change i-nodes + imap accordingly)
re-write!

Live?

Segment summary block: Part of each segment
- file #, block # : for each data block

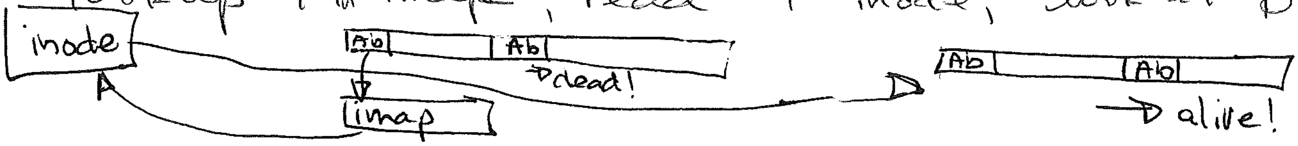
eg. F , B

Where does that currently point?

Dead → Somewhere Else

Live → Here

lookup F in imap, read F inode, look at B ptr...



Version #: Keep with imap + segment summary

Inc on every file delete/truncate

If don't match, dead data

Update: Imap

6. How is write cost defined for a log structured file system, if "u" is the utilization of segments being cleaned? What is write cost for FFS? What does Figure 3 show?

→ overhead imposed on writes

~~tota~~

$$wc = \frac{T(\overset{\text{cleaning}}{\text{overhead}}) + T(\text{data})}{T(\text{data})}$$ ideal is 1.0

sequential bw

$$\frac{\text{read segs} + \text{write live} + \text{write data}}{\text{write data}}$$

N: segments read u: utilization used (1-u : free space)

$$\frac{N + N \cdot u + N(1-u)}{N(1-u)} = \frac{2}{1-u}$$

FFS?

wc is slowdown vs. sequential bw of data

→ 10-20 (seeks of metadata)

→ 4 optimized

Point: Need to keep utilization < .5

Do a better job w/ cleaning policy

- get bimodal dist. of segments

- don't clean high util segments

7. There are a number of policy questions regarding segment cleaning. The first question we'll investigate is: which existing segments should be cleaned and how should the live data be grouped? Which segment does a simple greedy policy choose to clean? What was the idea behind the "Hot-and-Cold" policy? Why does "hot-and-cold" not work well with the greedy policy? Why is it less beneficial to clean a "hot" segment?

- Greedy: choose least-utilized segment (intuition: gives most free space)

- Hot-&-Cold: Segragate data by age
  - Sort live blocks by age, put hots w/ hot, cold w/ cold
  - Intuition: Act similarly

Why Bad?

- Cold segments take a long time to drop utilization - wasting space for long time
- See Fig. 5

Why Hot less beneficial? (to clean)

- likely to become even less utilized if wait longer

8. What is the idea of the cost-benefit policy? According to Figure 6, at what segment utilization are "hot" segments cleaned? At what utilization are "cold" segments cleaned?

- be lazy for ~~cold~~ hot
- aggressive for cold

- Judge benefit of cleaning too

high for cold

$$\frac{\text{benefit}}{\text{cost}} = \frac{\text{free space}^{(1-u)} \text{ generated} * \text{age of data}}{1 \text{ read} + u \text{ write}}$$

Hot: ~.15

Cold: ~.75

Fig. 6

9. When system crashes, updates to disk may be lost. How does FFS (or ext2) handle a system crash? (We'll talk more later about ext3 and other journaling file systems.) LFS uses checkpoints (consistent states of the fs) with a roll-forward (to update since last checkpoint). What are the two steps for creating a checkpoint? Why are two checkpoint regions needed? What are the pros/cons of having a longer time between checkpoints?

- FFS - doesn't know where writes were occuring
  → Must check entire fs hierarchy
  → fsck
- LFS - know writing at end of log

Checkpoint

1) Write everything out (meta, data, inode map, segment usage table)
2) Write to fixed checkpoint region saying where blocks for maps + tables are.

Two?

- Crash during checkpoint
- Add timestamp so use latest

Longer - good; less time
bad: lose more data

10. What needs to be updated during roll forward?

- Scan through segments since last checkpoint
- Recover new i-nodes (see in segment sum. block)
    - update i-map

11. Evaluation: How does LFS perform on small-file operations (create and delete)?

Fig. 8a

Very well

12. Evaluation: For large files, when does LFS perform better than SunOS? The same? Does it ever perform worse? What is logical versus temporal locality?

Fig. 9

LFS wins: write random
write seq

Tie: Read seq. or random

~~Re~~ Lose: Reread seq.
-Write random, read seq.

Access patterns don't match!

FFS: Logical locality - abstractions/entities determine accesses

LFS: Items created near each other in time have locality

13. Conclusions?

+ Perf. can be a win if all random

- Cleaning overheads very problematic

+ Motivation → matched design well

+ Log technology used in many other systems
  - useful for versioning
  - good match for distrib. FS