LFS motivation

- As caches get big, most reads will be satisfied from the cache
- No matter how you cache write operations, though, they are eventually going to have to get back to disk
- Thus, most disk traffic will be write traffic
- If you eventually put blocks (i-nodes, file content blocks) back where they came from, then even if you schedule disk writes cleverly, there's still going to be a lot of head movement (which dominates disk performance)

LFS approach

- Suppose, instead, what you wrote to disk was a log of changes made to files
  - log includes modified data blocks and modified metadata blocks
  - buffer a huge block ("segment") in memory – 512K or 1M
  - when full, write it to disk in one efficient contiguous transfer
    - right away, you've decreased seeks by a factor of 1M/4K = 250
- So the disk is just one big long log, consisting of threaded segments

Questions

- What happens when a crash occurs?
  - you lose some work
  - but the log that's on disk represents a consistent view of the file system at some instant in time
- Suppose you have to read a file?
  - once you find its current i-node, you're fine
  - i-node maps provide a level of indirection that makes this possible
    - details aren't that important

- How do you prevent overflowing the disk (because the log just keeps on growing)?
  - segment cleaner coalesces the active blocks from multiple old log segments into a new log segment, freeing the old log segments for re-use
    - Again, the details aren't that important

Tradeoffs

- LFS wins, relative to FFS
  - metadata-heavy workloads
    - small file writes
    - deletes

  (metadata requires an additional write, and FFS does this synchronously)
- LFS loses, relative to FFS
  - many files are partially over-written in random order
    - file gets splayed throughout the log