File System Layout

1. Notes from reviews:
    a. SSDs?
    b. Note: many details not explained
        i. Can't talk about everything
        ii. Some things aren't interesting – don't matter much, didn't evaluate
        iii. Need to select most interesting things to cover
    c. Are reads really not that important?
        i. Could reorganize data when clean segments (done in SSDs)
        ii.
2. Disk background
    a. Geometry
        i. Platters
        ii. Tracks
        iii. Sectors
    b. Classes
        i. SCSI: enterprise disks, higher speed (up to 15,000 rpm), smaller diameter (2.5 in), smaller capacity. Seek time ~ 4.3 ms, more platters
        ii. IDE/SATA: personal disks: lower speed (5,400 - 7,200 rpm), larger diameter (3.5 in). May not even use all surfaces available (e.g. 3 out of 4). Seek time ~9ms, fewer platters
    c. How used:
        i. ZBR: zoned bit recording. More zones on outer tracks, read more bits per revolution
        ii. Track ordering: serpentine. Tracks go in on top platter, on on bottom, in on next platter, then out. Cylinder idea not so good any more: platter-platter switching time may be higher than track-track seek.
        iii. Old model: 3 dimension address of a block: surface, cylinder, sector
            1. Why no cylinders?
            2. A: Disk geometries change a lot: Tracks per inch, sectors per track (varies over track). Complicated to expose all this to OS. Move to linear
            3. Flaws: disk can map around flaws – move a block somewhere else. Hard to do if raw geometry exposed.
            4. Scheduling done by OS, not controller
    d. New interfaces: SCSI

        i. provide a queue of commands, disk can do some ordering: e.g. balance both seek and rotation cost to optimize next block
        ii. can get average seek distance down to 1/10 maximum (vs. 1/3 for ata, non-queued)
    e. Key observation on disks: mechanical factors limit performance
        i. seek time (#1)
        ii. rotational latency (#2)

3. Big file system issues:
    a. Performance, based on type of workload
    b. Metadata performance: important when accessing many small files
    c. Recovery: ensure data doesn't get lost in crash

4. FFS recap

    a. Key ideas:
        i. Sub-block allocation:
            1. Want larger blocks for smaller metadata, more sequential reads
            2. Want smaller blocks for space efficiency
            3. Solution: only needs small blocks at the end; use a bitmap to track those
        ii. Physical locality:
            1. Track free blocks in a bitmap to be able to find adjacent blocks
            2. Locality via by cylinder groups
                a. Not necessarily sequential, but short seek distance
                b. Reason about locality at two levels – cylinder group and block level
            3. Contiguity policy
                a. Files in a directory go in same cylinder group
                b. Blocks in a file are sequential
        iii. Other things:
            1. Locating files: each inode number has a statically known location that can be calculated
            2. FSCK for reconstruction after failure

    b. Cylinder group: a group of close-together blocks.
        i. QUESTION: what does it map to logically? Why is it the size it is? For files, you don't want to do any seeks if possible
        ii. ANSWER: a directory; a group of related files

    c. QUESTION: what is benefit of breaking disk into smaller chunks?
        i. ANSWER: can have two-level policy; use global info to put a file in cylinder group and then use local info to find exact location of file.

d. **QUESTION: Why statically allocate inodes for each 2 kb of storage?**
   i. **Answer:** can find via calculation as compared to having to indirect if placed in a dynamic structure. 2kb means you can fill your disk with 2kb files – the average file size?

e. **QUESTION**: How do you quickly locate good blocks to use?
   i. FFS approach: break disk into rotational positions (8)
   ii. Rotational layout table: a vector, indexed by position, with pointers to block bitmap of of blocks at that position
   iii. Summary of # of free blocks at that position
   iv. When seeking block to allocate, use parameters to determine the position of the next block o allocate, then look for one in that position from vector.

f. **Layout policies: what can you accomplish?**
      1. ANSWER1: allow large sequential transfers of blocks in a file
      2. Answer 2: minimize seek latency between files/portions of files
   ii. need to decide where to allocate inodes, data blocks
   iii. need to decide when to predict locality, when to predict locality not needed.
   iv. Question: when is this for FFS?
      1. Answer: locality for files within a directory, but not between directories
      2. Answer: locality for contiguous chunks of a file up to some size, after that, if sequential access, can afford to do a seek (ammortize over long sequential reads).

g. **FFS approach: 2 level policy: global/local**
   i. Global policy: decided on cylinder group target for new files and directories
      1. Note: putting things nearby each other has an opportunity cost: cannot put other things nearby
      2. other examples: caching, prefetching
      3. put some related things together
      4. spread out other things to prevent hot spots, make space for related things to be nearby
   ii. **Question**: what should your policy be for creating new directory?
      1. A: FFS inode policy for directories: worst fit; assumes no locality between directories

2. choose a cylinder group with fewest directories, more than average # of available inodes
3. QUESTION: what about files in a directory?
   a. A: FFS wants locality. Put them all in a cylinder group
   b. note: takes a small # of blocks (many inodes/block), so can read directory contents without too many io's

iii. **QUESTION**: what about disk blocks?
   1. A: all in the same cylinder group, best at rotationally optimal on a single cylinder
   2. A: split at first indirect blocks (48 kb), as need a seek for that. Then every 1 mb to prevent cylinder group from overfilling
   3. 4 level policy:
      a. 1. Rotationally optimal block on same cylinder (allow fast transfer)
      b. 2. Another block in the cylinder group (minimize seeks)
      c. 3. Hash cylinder group # to find another group (leads all files to overflow to same place)
      d. 4. Search all CG.

iv. **QUESTION**: What would you do if you wanted to speed up write performance when disk is full?
   1. A: Maintain list of free block sorted by cylinder group
      a. NOTE: need to have 10% free space in the system for layout policies to work. Why?
      b. A: need to have enough free blocks to have a choice, need to have enough in the right places.

5.LFS

a. QUESTION: What were technology trends enabling LFS?

b. CPU speeds getting faster relative to disk
   i. QUESTION: What is implication? Can do more work per disk block to make good decisions

c. Memory sizes increasing with CPU speed
   i. More of read traffic satisfied in read
   ii. Writes still go to disk for reliability
   iii. QUESTION: Is this true?
      a. On dept Linux machines, I have 30x more writes/reads
      2. On my laptop
         a. 56% writes

d. Easy workloads: streaming large sequential reads/writes

      i. Easy to do layout here
   e. Hard workloads:
      i. Lots of random reads: nothing to be done, is fundamentally random
      ii. Lots of random writes, small files writes: **can be improved**
   f. Interesting workloads have lots of small files
   g. Good use of a metric:
      i. What is achieved write bandwidth relative to disk bandwidth?
         1. Tells you how much better you can do

6. **Big idea 1:** Asynchronous IO
   a. Why synchronous I/O?
      i. For metadata
      ii. Preserves consistency of directories, inodes, etc.
      iii. e.g. update free block bitmap before inode
   b. Decouple CPU from disk speed by removing need for programs to wait for disk
   c. Move burden of ensuring durability to application, which must request it via sync() and fsync() instead of being the default.
   d. QUESTION: What can be async?
      i. Hard for program to wait for read to complete; requires new programs
      ii. Can buffer write requests
   e. Write larger chunks of data – more sequentiality
   f. Higher likelihood files/blocks are overwritten and never need to be written
   g. **Reality**: apps care about losing data, some frequently call fsync()

7. **Big idea 2:** Sequential I/O even if data is random/different structures
   a. Much cheaper than random I/O – order of magnitude more efficient
   b. FFS spreads data around
      i. Inode separate from file
      ii. Directory separate from file
   c. Engineering Workloads
      i. Lots of small files (< 8kb)
      ii. Sequential, complete (in entirety) access
      iii. Average file lives < ½ day
      iv. Other workloads can be handled by other mechanisms
   d. FFS/UFS bad for metadata operations
   QUESTION: What does FFS do on these workloads?
   - A: no benefit of sequential layout for small files; shows up really for larger files
   - A: high cost to creating a small file: 5 writes
      -two access to file attributes, one for data, directory data, directory attributes

- A: synchronous writes: application must wait for these 5 writes to occur, slowing down the time to get to the next write. app can't use faster CPU because it is waiting for seeks on the slow disk.

Core ideas:

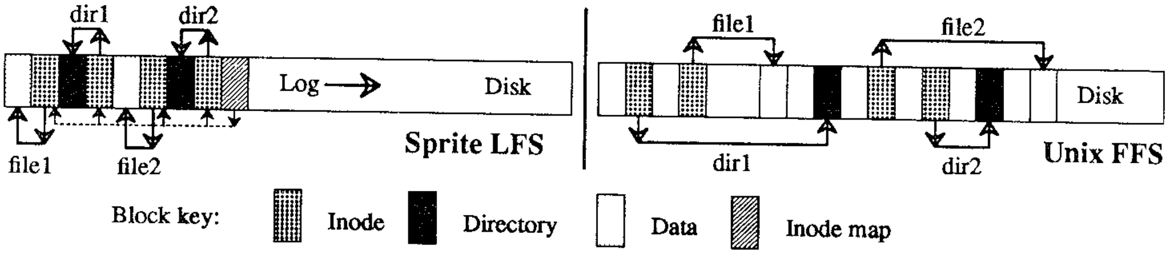
1. Usage-aware reorganization
    a. Hot/cold policy
2. Consistency via out-of-place updates
3. Table lookup to find metadata – in stead of direct looup
4. Use disk capacity for increased performance
    a. Reduced cleaning cost if delayed; more overwrites and less to copy
5. Redo logging/transactions for multi-variable updates

1. **Big Idea 3 No-overwrite/shadow updates**
    - All updates go to a different place
        - fast recovery (as we will see)
    - That other place is log structured
        - Favor sequential writes over sequential reads
        - **NOTE**: can do the previous two approaches without log structuring (see ZFS)

Problems that arise:

1. how do you find things?
    a. in FFS, how do you find an inode?
        i. A: statically mapped somewhere, or described by superblock
    b. in FFS, how do you find a block?
        i. From inode
    c. In LFS, need a layer of indirection: the INODE MAP
        i. for each inode, the current location.
        ii. how stored: dividing in chunks, stored in log. Checkpoint region (fixed) says where to find an inode map
    d. QUESTION: What kind of locality do you get?
        i. A: turn temporal locality into spatial locality.
        ii. A: if write a file in a large chunk (and nobody else does any writing in the middle), then it will be contiguous on disk

2. PICTURE: Cut and paste from paper



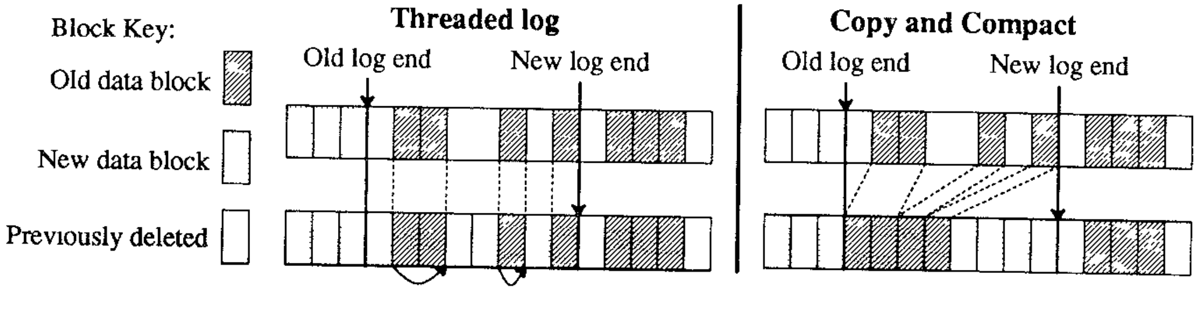
- How do you manage free space?

QUESTION: What is the equivalent of the free list/block bitmap?

A: list of clean segments. For other segments, need to check where it is part of a file.

a. Problem: as you overwrite a file, the old blocks become dead. How do you reclaim?
    i. at the block level: make the log weave through free blocks
    ii. space efficient but bandwidth inefficient (must skip over blocks still in use as a single contiguous chunk
    iii. when out of space, copy used blocks elsewhere & compact
b. with both: segments (contigous chunks) that are threaded together
    i. segments are only written when empty, always written sequentially
    ii. segments thread together to form the log.
c. QUESTION: How big should segment be?
    i. A: big enough so that you don't lose much bandwidth to seeking between segments
    ii. A: small enough that you can hold a number in memory to support cleaning

PICTURE: cut and paste from paper



QUESTION: what do you do when your log gets full?
A: segment cleaning.

PICTURE: show some partially full segments, read into memory, write back out fewer segments (possible over same segments or possibly someplace new).

BIG IDEA: do something fast at write time, do slow optimization later. Question: what are upsides/downsides?

QUESTION: How do you do cleaning?
A: you have to know which blocks are live/dead
A: you have to know which file owns which blocks, so you can update its inode

QUESTION: how do you know this?
A: you summarize the parts you know when writing (e.g. which files own which blocks)
QUESTION: with this, how do you know if a block is live or dead?
A: fine the file, then use normal lookup to see if its blocks are the same as the ones in the segment.
QUESTION: what if the file was deleted and the inode number reused?
A: then you know the blocks are all free. Optimize by storing version number of file with inode number.

Big questions:
1. When should cleaner run? background, night/weekends?
2. How many segments should be cleaned?
3. Which segments should be cleaned?
4. How do you organize blocks when writing them out? For locality or something else?

Answers:

1. When? A: threshold based; when # free drops

Discussion: how think about this?
A: compare to garbage collectors. Depends on your policy.
- Do you want low average latency? Then run periodically, but stop the world
- Do you want low variablility? Then run continously in background

2. How many?

Discussion: how think about this?
A: What is downside of cleaning more at once? takes longer (stop the world) and kicks more stuff out of memory (less cache)
A: calculate expected return (reduction in something) from cleaning more against expected cost.

3. Which segments to clean?

First question: how do you compare policies? What is the right metric?

A: Write cost = multiple of time to sequentially write blocks. Suppose have to move something 5 times - those 5 writes use disk bandwidth that could be used for writing new data.

QUESTION: Why is this the right metric? It ignores idle time in disk
A: it captures important differences.

QUESTION: how calculate?
In steady state, write cost = (total bytes read + written)/(new data written) ==

(read segs + write live + write new) / write new

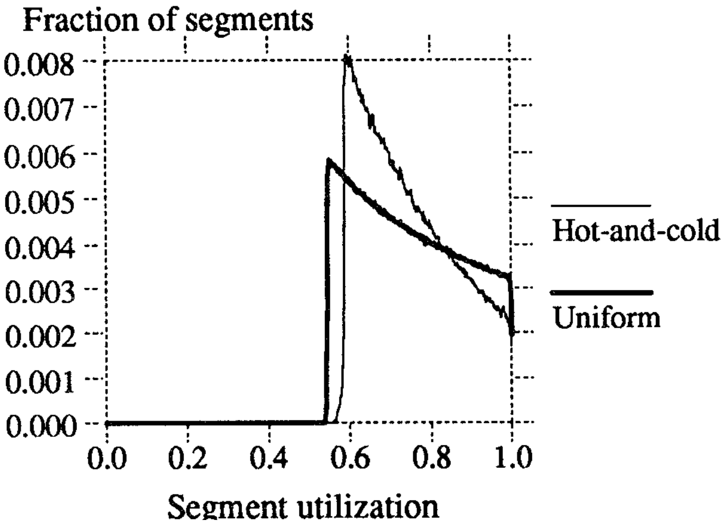
Note: when read a segment, always gets written, either as old data or as new data. So cost is 2: 1 read, 1 write

= 2/(1-u) where u is utilization

Big Idea: ideal case is highly-utilized cold segments (for capacity) + lightly utilized segment (for efficient GC)

Policies:

- Greedy: always clean least-utilized segments first, sort by age when writing out (to segregate cold data from hot data)



Problem: show figure 5
1. segment doesn' get cleaned until it is least utilized

2. Cold segments drop very slowly - not much change, tie up lots of segments

    1. Observation: free space in a cold segment worth more than free space in a hot segment
    2. Once cleaned, a cold segment stays full for a long time (system gets to keep free blocks)
    3. In a hot segment: data copied will be deleted soon; might as well delay and wait for more free space to accumulate.

IMPORTANT IDEA: value of a resource may vary based on its future behavior

2. Age-space policy: benefit/cost = (free space generated * age) / cost = (1-u)*age/(1+u)
3. Really: look at the "area" of free space – the size * the duration

- Result: clean cold segments at 7% usage, but cleans hot segments at 15%
- Result: reduce write cost by 50% at higher disk utilization

Implementation: segment usage table (yet another data structure)

4. record # of live bytes & most recently modified block for each segment
5. written to log, pointed to by checkpoint

QUESTION: What is the performance impact of cleaning?
Answer: depends on disk utilization and business

6. For full disk, must run cleaning all the time at write cost > 2, so write performance drops dramatically
7. For busy system where continuously writing (e.g. a database), write performance drops when write disks' worth of data
8. In flash drives (where used), typically see 60% degradation.

BIG IDEA: Trade space for performance

1. With more free space, cleaning is more efficient. **WHY?**
    a. If leave more space for cleaning, do less cleaning, more of blocks are empty
    b. With little free space have to clean often
2. Same applies to language GC
3. Can choose when buy a disk how much space you want to use

Reliability

FS crash recovery:

1. Updates written out in any order
    a. On recovery, metadata may be inconsistent:

  i. Inode is part of a directory, but free inode bitmap shows in use
  ii. File metadata extended to include a new block, but file data not yet written
 b. Can be solved by carefully ordering updates
  i. Basic rule: write data before linking to it
  ii. Write metadata removing use of a block before writing metadata marking it free

2. Still need to fix file system on recovery
   a. May have lost blocks, inodes that aren’t part of any file
   b. Not all file systems write things in order – bad for locality
   c. Disk may have cache that reorders requests (no in FFS era, but now)

LFS approach

1. All data written out sequentially
   a. Can easily control order of updates
2. Never overwrite data in place
   a. On crash, is like a transaction: old data is still there
3. Use garbage collection rather than free lists
   a. Don’t need to keep track of what blocks are in use/free; done at segment level
4. checkpoint all FS structures periodically
   a. Problem: reconstruct inode map requires replaying log to see all changes
   b. Solution: checkpoint inode map periodically (30 seconds)
      i. all dirty metadata in memory
      ii. write all maps location of inode map, segment usage table, current time, pointer to last segment) to fixed-location checkpoint region.
   c. During reboot, read checkpoint, then roll forward to generate up-to-date structures.
5. Can truncate a partial segment
   a. Lose some data, but keep consistency
   b. Reload checkpoint
   c. Read through log segments since then
   d. Update inode map, other in-memory tables based on what is in segments
6. If replay partial segments:
   a. Problem: may be halfway through multi-object operation
      i. Rename a file between directories
   b. Use an intention log for directory updates: otherwise may not get consistency between directories and inodes if one written but not the other “directory operation log” to allow atomically updating two or more data structures
   c. like a regular FS journal
7. Note: biggest win is from not overwriting

a. Old data is still available if update not made completely

QUESTION: Do you need fsck to repair?
Answer: log roll forward works for crash failures (on block boundaries) other failures, such as corrupt data, still need fsck

Notes:
- Not measure overhead of metadata-rewrites. Every write requries updating inode or indirect block (15% of written data to log).
- No live performance measurements

Big ideas

e. Use known locality, at write time, to drive layout, rather than predicted locality (within a file and within a directory) as FFS does
f. Separate writing to disk and long-term layout (e.g. cleaning)
g. Take advantage of idle cycles for cleaning so can handle large bursts
h. Summarize information (e.g. segment summaries) for performance
i. Take advantage of dynamic (run time) locality instead of static (file system layout) locality
j. Layout metadata contiguously with data (e.g. inode next to data and indirect blocks).

QUESTION: Which ideas should we take away from here?

- **asynchronous writes to a log**
- **focusing on metadata locality**
- **Log structure for Flash disks, where you can't overwrite.**
    - **Note: 13% of what is written is metadata, this may eat up flash write bandwidth**

Issues

k. LFS very good for metadata operations: create / delete (3-4x ffs)
l. Truly random access has bad performance
m. LFS has high CPU utilization due to extra data structures, cleaning
n. Depends on lots of memory, multiple users
    i. Must fill a segment to be efficient, single user may not fill it
    ii. Need lots of cache to avoid slow reads
o. Cleaner can cause problems for busy system at 80% utilization; cleaning is synchronous and blocks work
    i. 34% performance drop on TPC

What Happens Today
Use a mix: metadata writes go to a log, so not need to seek around, data writes made asynchronous and try to have high locality

Lazily write metadata to correct location, so often not have to wait for it

```
Discussion:  Is LFS a good idea?
  How have predicted technology trends played out?
    + More workloads now seekbound instead of CPU bound
    - Disk capacity grew faster than main memory
      So cache sizes aren't big enough to eliminate read traffic
  How have file systems dealt with problems LFS identified?
    Mostly by journaling metadata in write-ahead log
    Soft updates another technique
  But note LFS ideas particularly good when combined with RAID
    Will see a successful product using similar ideas in AutoRAID paper
  Big (at least perceived) limitation of LFS is cleaner
```