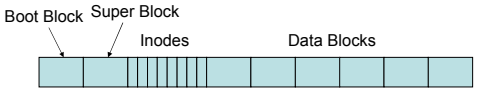
Storage & File Systems

Ravikant Dintyala

Unix File System 4.2 BSD

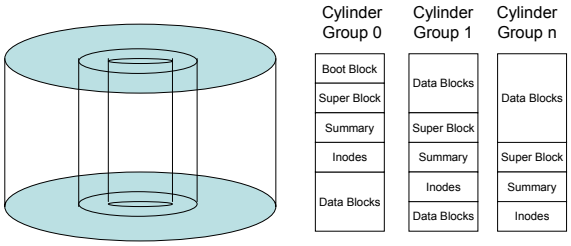


- Boot block – disk layout information
- Super block – fs size & type, free inode list, free data block bitmap
- Inodes
- Data Blocks

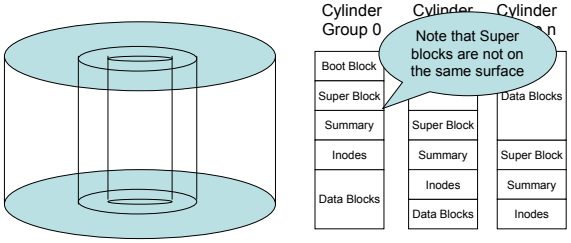
KJL84 - Idea

- Divide disk into cylinder groups (4 MB), each cylindrical group has sufficient information to handle free space.
- Increase block size, address fragmentation.
- Writes are always in full blocks, except for a partial block at the end.
- Blocks are allocated in the same group whenever available otherwise they are allocated in a “rotationally optimal manner”.

KJL84 - Continued



KJL84 - Continued



Layout Policies

- Global – place new directory in a cylinder group that has maximum free inodes and minimum no of directories, force long seeks to new cylinder groups.
- Local – keep file allocation rotationally optimal within a cylinder group, spread big files across the disk in chunks of 1MB.

Performance

Table IIa. Reading Rates of the Old and New UNIX File Systems

Type of file system	Processor and bus measured	Speed (Kbytes/s)	Read bandwidth %	% CPU
Old 1024	750/UNIBUS	29	29/983 3	11
New 4096/1024	750/UNIBUS	221	221/983 22	43
New 8192/1024	750/UNIBUS	233	233/983 24	29
New 4096/1024	750/MASSBUS	466	466/983 47	73
New 8192/1024	750/MASSBUS	466	466/983 47	54

Table IIb. Writing Rates of the Old and New UNIX File Systems

Type of file system	Processor and bus measured	Speed (Kbytes/s)	Write bandwidth %	% CPU
Old 1024	750/UNIBUS	48	48/983 5	29
New 4096/1024	750/UNIBUS	142	142/983 14	43
New 8192/1024	750/UNIBUS	215	215/983 22	46
New 4096/1024	750/MASSBUS	323	323/983 33	94
New 8192/1024	750/MASSBUS	466	466/983 47	95

Metadata

- Metadata (directories, inodes, free block maps, ..) give structure to raw storage capacity.
- File system must maintain integrity of metadata in face of unpredictable failures.
- Disk image must be consistent.

Update Dependencies

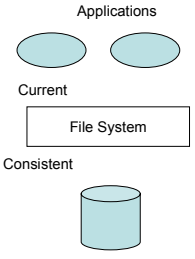
- Never point to a structure before it has been initialized.
  - An inode must be initialized before a directory entry references it.
- Never reuse a resource before nullifying all previous pointers to it.
  - An inode's pointer to a data block must be nullified before that disk block may be reallocated for a new inode.
- Never reset the last pointer to a live resource before a new pointer has been set.
  - When renaming a file, do not remove the old name for an inode until after the new name has been written.

Options

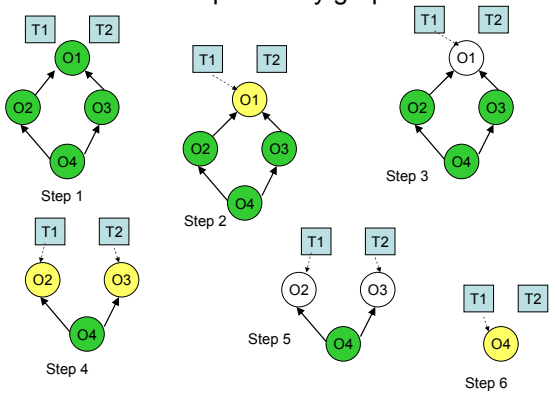
- Synchronous writes – compromise performance.
- Asynchronous writes – compromise integrity.
- Special purpose hardware (NVRAM) – costly.
- Atomic updates (write ahead logging).

Soft Updates - Idea

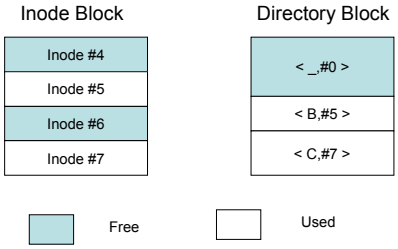
- Write back caching.
- Track dependencies among updates.
- Sequence updates respecting these dependencies.
- Break cycles by roll-back before block is written and roll-forward afterward.



DAG dependency graphs

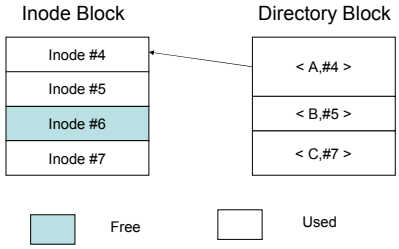


Cyclic Dependency



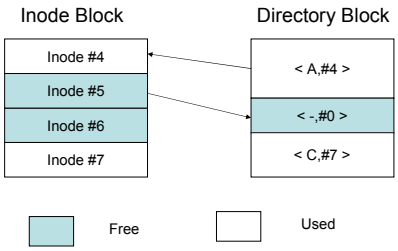
Original Organization

Cyclic Dependency



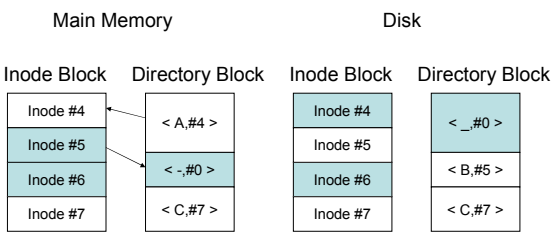
Create File A

Cyclic Dependency



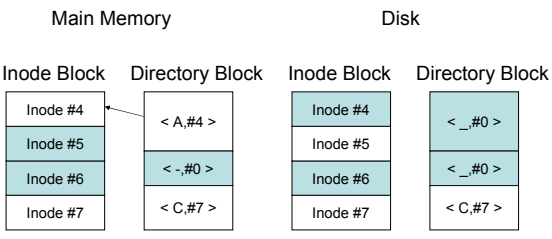
Delete File B

Undo/Redo



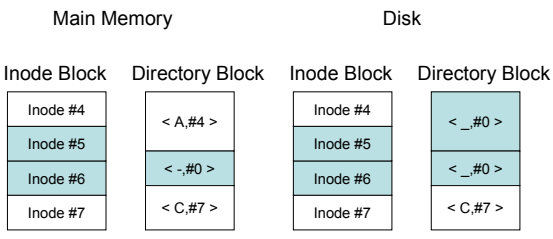
After Metadata Updates

Undo/Redo



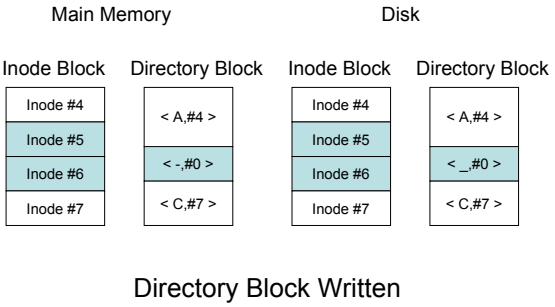
Safe version of Directory Block Written

Undo/Redo



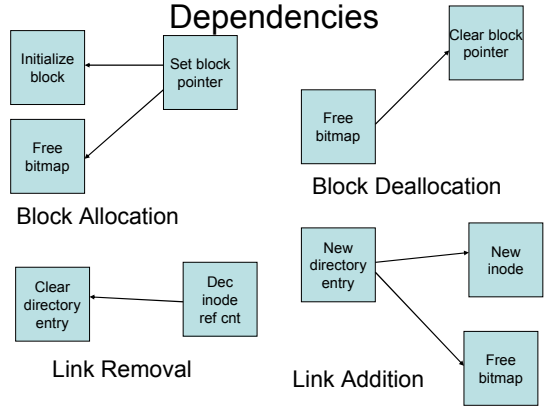
Inode Block Written

Undo/Redo



Directory Block Written

Dependencies



Recovery

- Possible inconsistencies:
  - Unused blocks may not be in free space maps.
  - Unreferenced nodes may not appear in the free inode maps.
  - Inode link counts may exceed the actual number of directory entries.
- Disk Image is safe to use.
- Run **fsck** (background/during downtime) to reclaim resources.

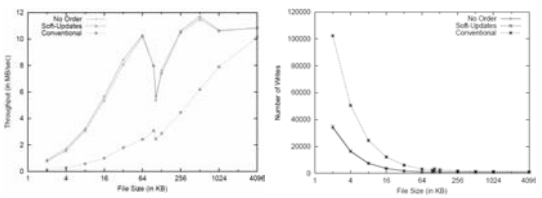
Other Issues addressed

- Memory used for dependency structures – hack to handle deletes of large directory trees.
- Useless write-backs – upgraded flush routines and cache replacement routines based on dependency information (final overhead – 2.5 – 5%).

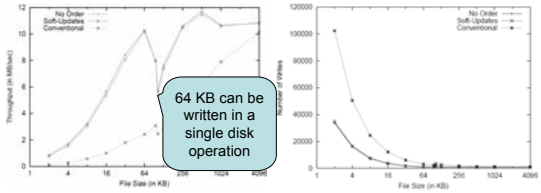
Performance

- Measure the speed with which a system can create, read, delete 32MB of data for files that range in size from 2KB to 4MB.
- Scenarios compared:
  - No Order (no write order enforced)
  - Soft Updates
  - Conventional (BSD FFS)

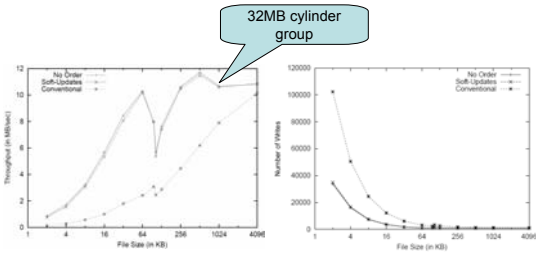
Create



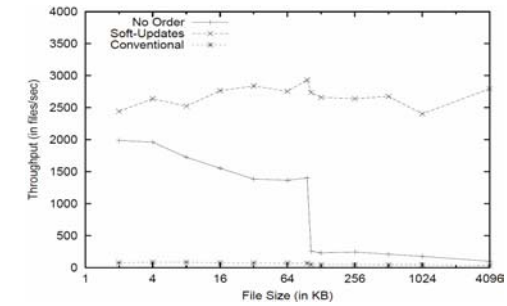
Create



Create

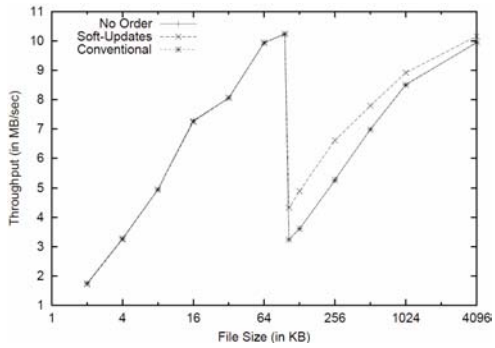


Delete



Soft-Updates outperforms No Order since the latter actually removes the files.

Read



Log-Structured File System - Idea

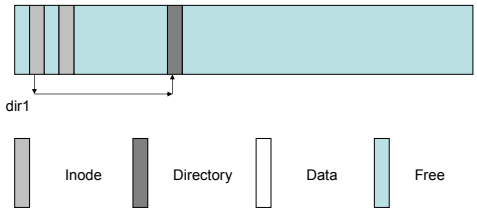
- Log is the only structure on the disk.
- Buffer sequence of changes, write it all at once sequentially to the end of the log.
- Write includes almost everything: file data, indexes, inodes.
- Maintain indexes for efficient read.
- Clean ‘segments’ to maintain large free areas.

Data Structures - Location

Table I. Summary of the Major Data Structures Stored on Disk by Sprite LFS.

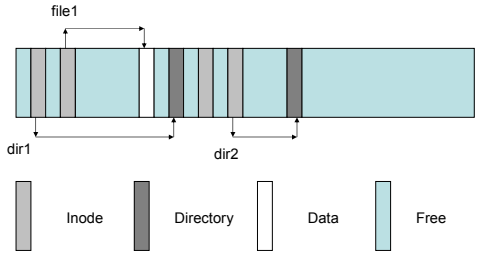
Data structure	Purpose	Location
Inode	Locates blocks of file, holds protection bits, modify time, etc.	Log
Inode map	Locates position of inode in log, holds time of last access plus version number.	Log
Indirect block	Locates blocks of large files.	Log
Segment summary	Identifies contents of segment (file number and offset for each block).	Log
Segment usage table	Counts live bytes still left in segments, stores last write time for data in segments.	Log
Superblock	Holds static configuration information such as number of segments and segment size.	Fixed
Checkpoint region	Locates blocks of inode map and segment usage table, identifies last checkpoint in log.	Fixed
Directory change log	Records directory operations to maintain consistency of reference counts in inodes.	Log

Disk Status - FFS



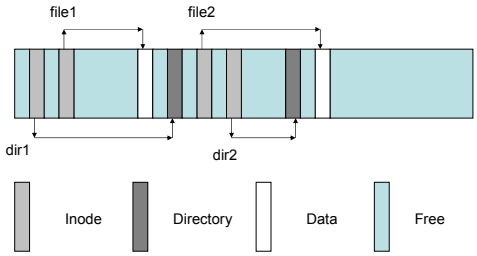
After create dir1/file1

Disk Status - FFS



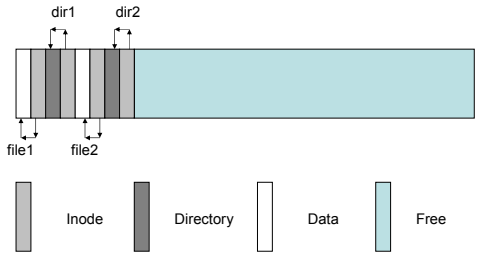
After create dir2/file2

Disk Status - FFS

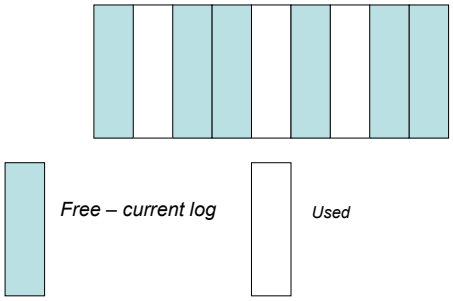


Delayed write-back

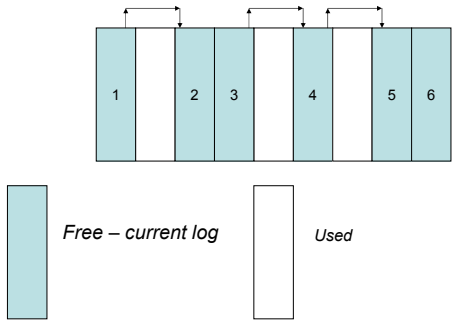
Disk Status - LFS



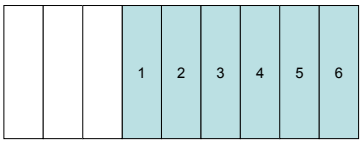
Writing Log?

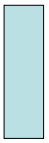


Writing Log - Threading

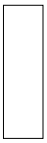


Writing Log - Compaction



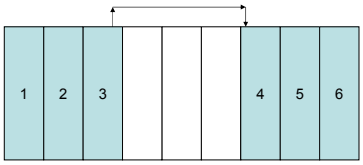


Free – current log



Used

Segments



- Sprite LFS uses a hybrid scheme.
  - Disk divided into fixed size segments.
    - Threaded between segments.
    - Compaction within a segment.
  - Segment size chosen so that transfer time is much greater than access time: 512 KB or 1 MB.

Segment Summary

- Each segment maintains a summary that identifies each piece of information in a segment - for each data block, it has the file number, file version, inode number and the block number of that block in the file.
- Cross checking the file's version, inode/indirect block and the segment summary helps identify dead data.

Segment Cleaning

- Four policy issues:
  - When should segment cleaner execute?
    - Continuously, only when needed, etc.
  - How many segments to clean at a time?
    - Must find enough free space to result in one clean segment.
  - Which segments to clean?
    - Lowest utilized, oldest, etc.
  - What re-orderings to perform when rewriting a segment?
    - Group files in same directory, group temporally, group by age, etc.
- First two ignored, do not seem to be important.
  - Starts cleaning when #segments drop below a watermark. Cleans a few tens of segments at a time.
- Third and fourth are important.

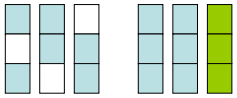
Selecting segments - Write Cost

- Write cost is the ratio of total work done to useful work done.

$$\text{write cost} = \frac{\text{total bytes read and written}}{\text{new data written}}$$

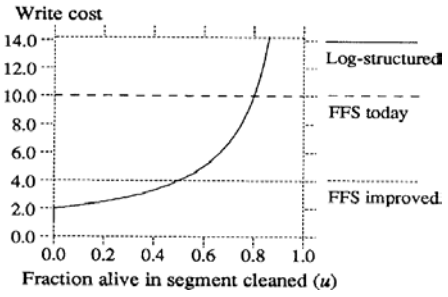
$$= \frac{\text{read segs + write live + write new}}{\text{new data written}}$$

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



(3 + 2 + 1)/1

Write Cost

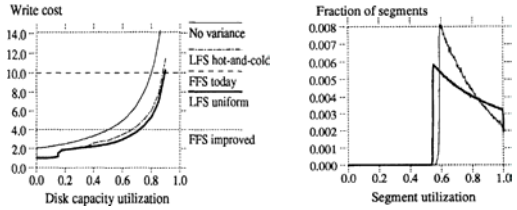


Clean Segments with least u

Simulation

- Fixed number of 4 KB files. No reading, just rewriting.
- Two access patterns:
  - Uniform (no cleaner reordering)
  - Hot-and-cold (cleaner reordering based on age)
    - One group: contains 10% of the files with 90% chance of being selected.
    - Other group: contains 90% of the files with 10% chance of being selected.
- Simulator runs till all clean segments exhausted, then runs cleaner until a threshold of clean segments reached.
- Cleaner always chooses least-utilized segments, reorders blocks by age

Results, Introspection



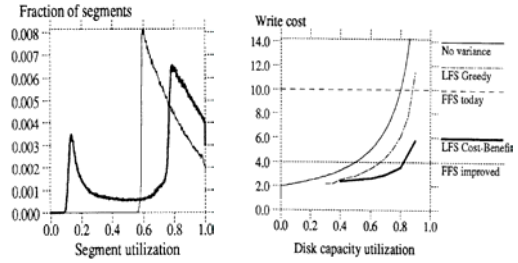
- How to clean such that reordering by age is efficient?
- How to get a bimodal distribution for u so that average u is low?

Idea

- Want to have greater amount of free space in circulation so that segment cleaner finds free space whenever it wants
- Free space in “old” segments is getting locked for a longer time under ‘least u’ policy
- ‘Running Average Oldness’ of data easy to maintain in the segment summary (actually the paper uses current time - the time of most recent modification)

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

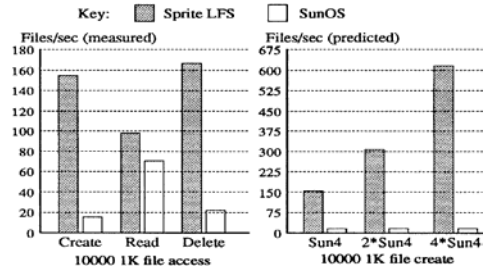
Results



Crash Recovery

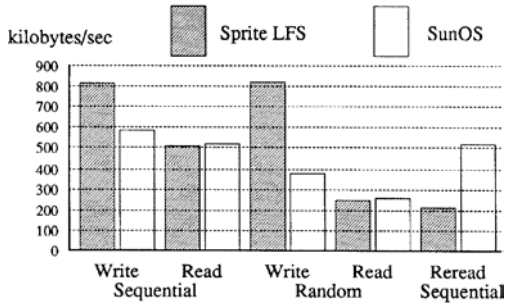
- Two-pronged approach:
  - Checkpoint: a complete, self-contained record of a consistent state of the file system.
  - Roll-forward: recover operations performed after the checkpoint by re-doing the operations after the checkpoint (only the recent log needs to be accessed).

Performance – small files



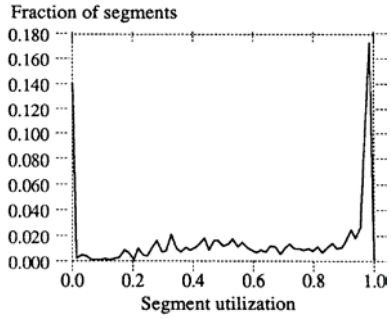
- Small-file performance under Sprite LFS and SunOS: create 10,000 1K files, then read back in same order, then delete.
  - The logging approach provides an order of magnitude speedup for creation and deletion.
  - In SunOS, disk 85% saturated, so faster processors will not help much. In Sprite LFS, disk only 17% saturated, while the CPU was 100% utilized.

Performance - large files



- Large-file performance under Sprite LFS and SunOS: create a 100-MB file with sequential writes, read back sequentially, write 100 MB randomly, read 100 MB randomly, finally read sequentially.

Segment Utilization (in practice)



Disk Usage

Table IV. Disk Space and Log Bandwidth Usage of /user6.

Sprite LFS /user6 file system contents		
Block type	Live data	Log bandwidth
Data blocks*	98.0%	85.2%
Indirect blocks*	1.0%	1.6%
Inode blocks*	0.2%	2.7%
Inode map	0.2%	7.8%
Seg Usage map*	0.0%	2.1%
Summary blocks	0.6%	0.5%
Dir Op Log	0.0%	0.1%

- Block types marked with '*' have equivalent data structures in Unix FFS.

Disk Usage

Table IV. Disk Space and Log Bandwidth Usage of /user6.

Sprite LFS /user6 file system contents		
Block type	Live data	Log bandwi
Data blocks*	98.0%	85.2%
Indirect blocks*	1.0%	1.6%
Inode blocks*	0.2%	2.7%
Inode map	0.2%	7.8%
Seg Usage map*	0.0%	2.1%
Summary blocks	0.6%	0.5%
Dir Op Log	0.0%	0.1%

Increased usage of Inode map due to the regular cleaning of segments

- Block types marked with '*' have equivalent data structures in Unix FFS.

Conclusion

- Soft Updates allow writes to go out of order, cash on disk drivers' write scheduling. Nice way of breaking dependencies by book keeping.
- Soft Updates – disk image consistent, immediate recovery.
- LFS uses disk as a log, disk write bandwidth utilized efficiently. Nice cost benefit based segment cleaning.
- LFS – a very good alternative if the workload is small file intensive.
- LFS – recovery by checkpoint + roll forward of the recent log.

Unaddressed Issues

- LFS - how to maintain file locality while cleaning segments?