Advanced File Systems

- Metadata updates and crash recover
- Log-structured file system (LFS)
- Meta-data logging (or journaling)

Ordering on Metadata Updates

- Principles of ordering for consistency
  - never reset the old pointer to a resource before the new pointer has been set
  - never re-use a resource before nullifying all previous pointers to it
  - never point to a structure before it has been initialized

Examples of Update Ordering

- rename:
  - change the old directory entry, set up the new directory entry --- ordering?
- delete:
  - free the disk blocks, free the inode, remove the directory entry --- ordering?
- create:
  - create new inode, add a new directory entry

Crash Recovery via fsck

- **read and check all inodes and build a bitmap of used data blocks**
- **traverse directories from root to mark all connected directories**
- **validate the structure of directory tree, making sure all links are accounted for**
- **validate directory contents to account for all files**
- **scan inodes looking for disconnected directories, put them in the lost+found directory**
- **scan inode looking for disconnected files, put them in the lost+found directory**
- **check and repair the bitmaps and summary accounts for each cylinder group**

Log-structured File System

Motivation:

- Technology
  - disk seek/rotational time dropping very slowly
  - main memory cache absorbs read requests
- Workloads
  - office and engineering environment, dominated by access to small files and metadata updates

Problems with FFS

- too many small disk accesses
  - different small files not allocated sequentially
  - inodes separated from files
  - creating a small file:
    - read directory inode, read dir data (cached);
    - initialize file inode (sync), write dir data (sync), write dir inode (sync)
    - allocate blocks, write file data, write file inode
- synchronous update for consistency

Log-Structured File System

- buffer file system changes (including metadata and data) in the file cache
- write all changes to disk sequentially as a segment

=> treat the disk as a log

Issues:

- how to retrieve information from the log
- how to manager the free space on disk

Retrieving a File

- inode and indirect blocks: same as in FFS
- Location of the inode:
  - in FFS, index into the cylinder group's inode sectors using the inode number
  - in LFS, index into the *inode map* (an extra level of indirection)
  - inode maps: identified by a fixed checkpoint region on each disk
- Layout comparison with FFS

Writing a File

- allocate a new inode (in-memory operation)
- allocate data blocks (in-memory operation)
- when is on-disk structure changed?

=> at checkpoints (every 30 seconds)

LFS Checkpointing

- Checkpoints
  - write out (in the log) all modified file data blocks, indirect blocks, inodes, and blocks of inode maps, and segment usage table
- A checkpoint region contains:
  - addresses of all inode map blocks and segment usage table blocks
  - last segment written + time of checkpoint
  - two checkpoint regions, alternate between them

Free Space Management

- Motivation
- Approaches
  - threading
  - copying
- Segments in LFS
  - similar to generational garbage collection
  - read, copy live data, then write out

Segment Summary Blocks

- Identify each piece of information in the segment
  - file block: file inode number + block number
  - inode: inode number
  - inode map blocks: inode map block number
- Identifying live data:
  - for each file inode and block num, check the pointer in the file's new inode
  - add a version number to file id

Crash Recovery in LFS

- Sequential writes of data blocks, inodes, inode-maps, segment summary blocks
  - the last disk operations before crash are found at the end of the log
- Read checkpoints to rebuild in-memory data structure
- Use roll-forward to recover more information

Roll-Forward in Crash Recovery

- use segment summary block
  - if there is a new inode, then update inode map
  - adjust utilization data for segments written after checkpoints as well as old segments
- directory operation log
  - reason: no guarantee on the writing order of inode changes and directory entry updates
  - record contains op, dir inum, position in dir, new reference count for the inode involved
  - log appears before directory block and inode

Segment Cleaning Policies

- issues:
  - when should the segment cleaner execute
    - low water mark
  - how many segments to clean at a time
    - till reach high water mark
  - which segments to clean
  - how to group live blocks (locality sort or age sort)
- write-cost: amortized cost of writing new data
    - bimodal distribution is desired

Analysis of Cleaning Policies

- LFS greedy: choose the least utilized segment to clean
  - does not perform well in hot-cold model
  - cold segments tie up free space for long periods
- LFS cost-benefit (suppose u is utilization)
  - cost: 1+u
  - benefit: (1-u) * age (approximate how long the free space can be kept around)
  - segment usage table (u and age)

Looking at Benchmark Results

- small file accesses
  - LFS only kept disk busy 17% of the time
- large file accesses
  - comparable for writes
  - not as good as SunOS for random-writes-then-sequential-reads