Classic File Systems: FFS and LFS

Presented by Hakim Weatherspoon

(Based on slides from Ben Atkin and Ken Birman)

A Fast File System for UNIX

Marshall K. McKusick, William N. Joy, Samuel J Leffler, and Robert S Fabry

- Bob Fabry
  - Professor at Berkeley. Started CSRG (Computer Science Research Group) developed the Berkeley SW Dist (BSD)
- Bill Joy
  - Key developer of BSD, sent 1BSD in 1977
  - Co-Founded Sun in 1982
- Marshall (Kirk) McKusick (Cornell Alum)
  - Key developer of the BSD FFS (magic number based on his birthday, soft updates, snapshot and fsck. USENIX
- Sam Leffler
  - Key developer of BSD, author of *Design and Implemention*

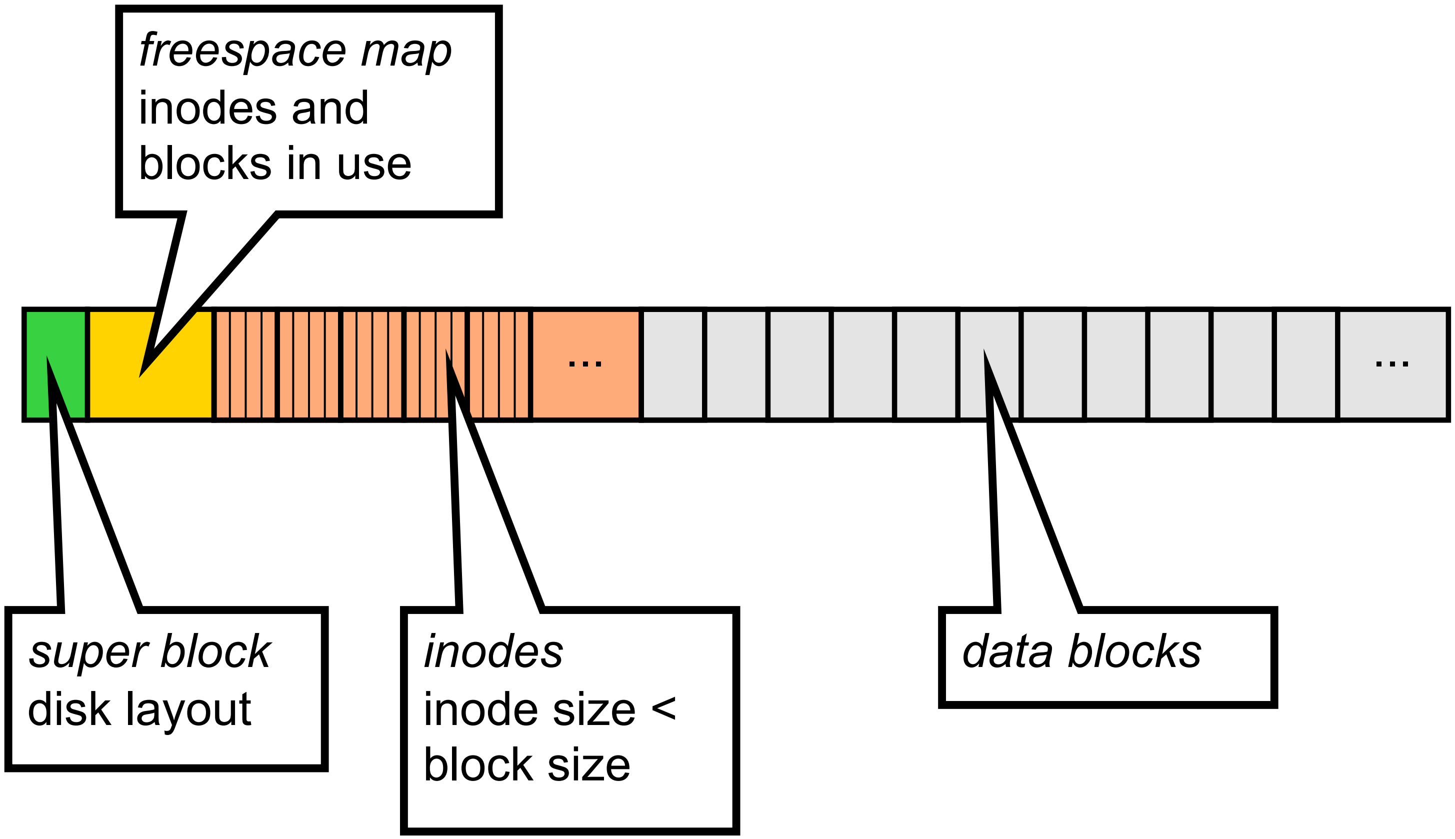
Background: Unix Fast File Sys

- Original UNIX File System (UFS)
  - Simple, elegant, but ***slow***
  - 20 KB/sec/arm; ~2% of 1982 disk bandwidth
- Problems
  - blocks too small
  - consecutive blocks of files not close together
    (random placement for mature file system)
  - i-nodes far from data
    (all i-nodes at the beginning of the disk, all data afterward)
  - i-nodes of directory not close together
  - no read-ahead

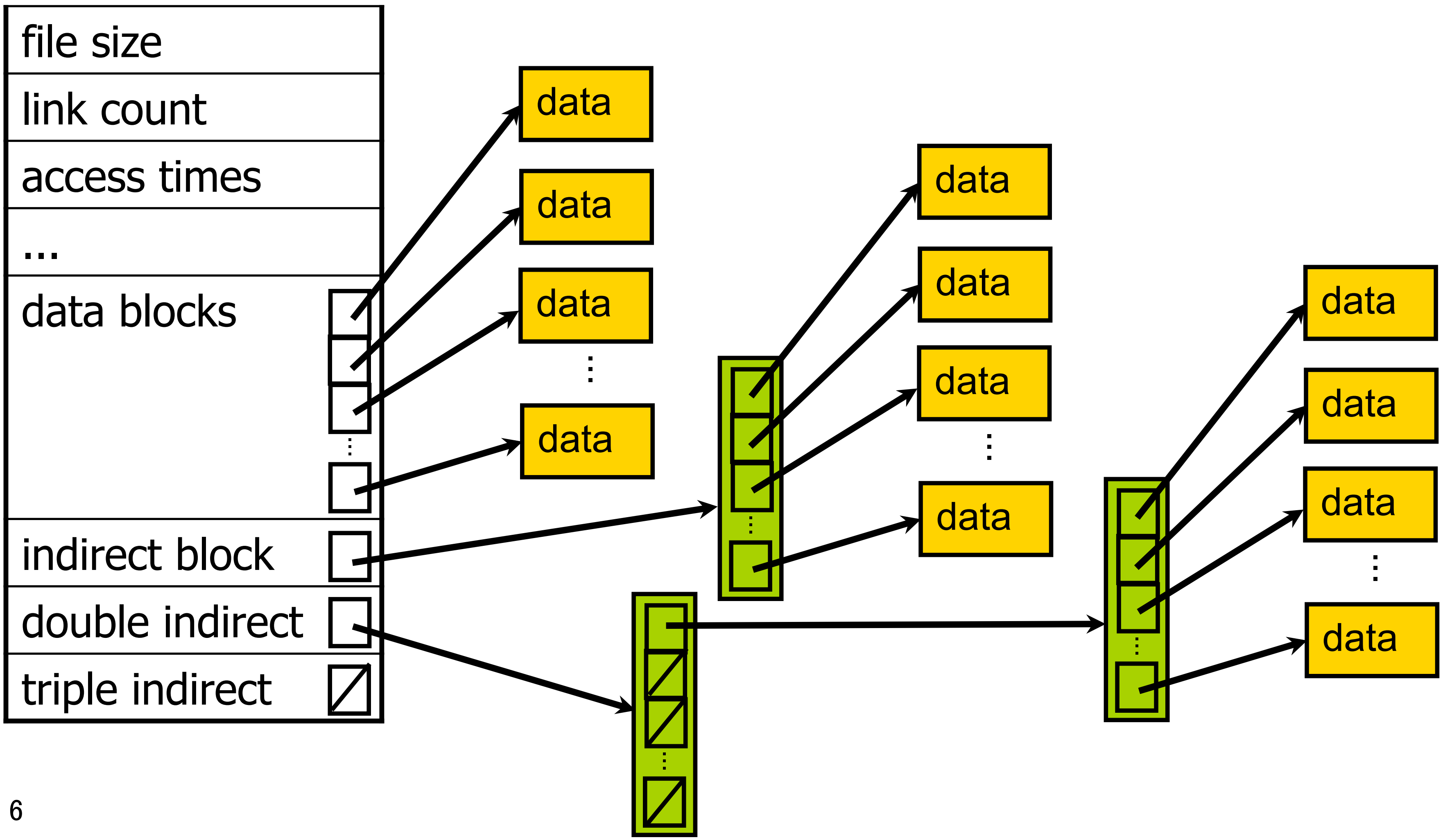
Inodes and directories

- Inode doesn't contain a file name
- Directories map files to inodes
  - Multiple directory entries can point to same Inode
  - Low-level file system doesn't distinguish files and directories
  - Separate system calls for directory operations

File system on disk



File representation



The Unix Berkeley Fast File System

- Berkeley Unix (4.2BSD)
  - Dsf
- 4kB and 8kB blocks
  - (why not larger?)
  - Large blocks and small fragments
- Reduces seek times by better placement of file blocks
  - i-nodes correspond to files
  - Disk divided into cylinders
    - contains superblock, i-nodes, bitmap of free blocks, summary info
  - Inodes and data blocks grouped together
  - Fragmentation can still affect performance

FFS implementation

- Most operations do multiple disk writes
  - File write: update block, inode modify time
  - Create: write freespace map, write inode, write directory entry
- Write-back cache improves performance
  - Benefits due to high write locality
  - Disk writes must be a whole block
  - Syncer process flushes writes every 30s

FFS Goals

- keep dir in cylinder group, spread out different dir's
- Allocate runs of blocks within a cylinder group, every once in a while switch to a new cylinder group (jump at 1MB).
- layout policy: global and local
  - global policy allocates files & directories to cylinder groups. Picks "optimal" next block for block allocation.
  - local allocation routines handle specific block requests. Select from a sequence of alternative if need to.

FFS locality

- don't let disk fill up in any one area
- paradox: for locality, spread unrelated things far apart
- note: FFS got 175KB/sec because free list contained sequential blocks

  (it did generate locality), but an old UFS had randomly ordered blocks and only got 30 KB/sec

FFS Results

- 20-40% of disk bandwidth for large reads/writes
- 10-20x original UNIX speeds
- Size: 3800 lines of code vs. 2700 in old system
- 10% of total disk space unusable

FFS Enhancements

- long file names (14 -> 255)
- advisory file locks (shared or exclusive)
  - process id of holder stored with lock => can reclaim the lock if process is no longer around
- symbolic links (contrast to hard links)
- atomic rename capability
  - (the only atomic read-modify-write operation, before this there was none)
- Disk Quotas
- Overallocation
  - More likely to get sequential blocks; use later if not

FFS crash recovery

- Asynchronous writes are lost in a crash
  - `Fsync` system call flushes dirty data
  - Incomplete metadata operations can cause disk corruption (order is important)
- FFS metadata writes are synchronous
  - Large potential decrease in performance
  - Some OSes cut corners

After the crash

- **`Fsck`** file system consistency check
  - Reconstructs freespace maps
  - Checks inode link counts, file sizes
- Very time consuming
  - Has to scan all directories and inodes

Perspective

- Features
  - parameterize FS implementation for the HW in use
  - measurement-driven design decisions
  - locality “wins”
- Flaws
  - measuremenets derived from a single installation.
  - ignored technology trends
- Lessons
  - Do not ignore underlying HW characteristics
- Contrasting research approach
  - Improve status quo vs design something new

The Design and Impl of a Log-structured File System

Mendel Rosenblum and John K. Ousterhout

- Mendel Rosenblum
  - Designed LFS, PhD from Berkeley
  - Professor at Stanford, designed SimOS
  - Founder of VM Ware

- John Ousterhout
  - Professor at Berkeley 1980-1994
  - Created Tcl scripting language and TK platform
  - Research group designed Sprite OS and LFS
  - Now professor at stanford after 14 years in industry

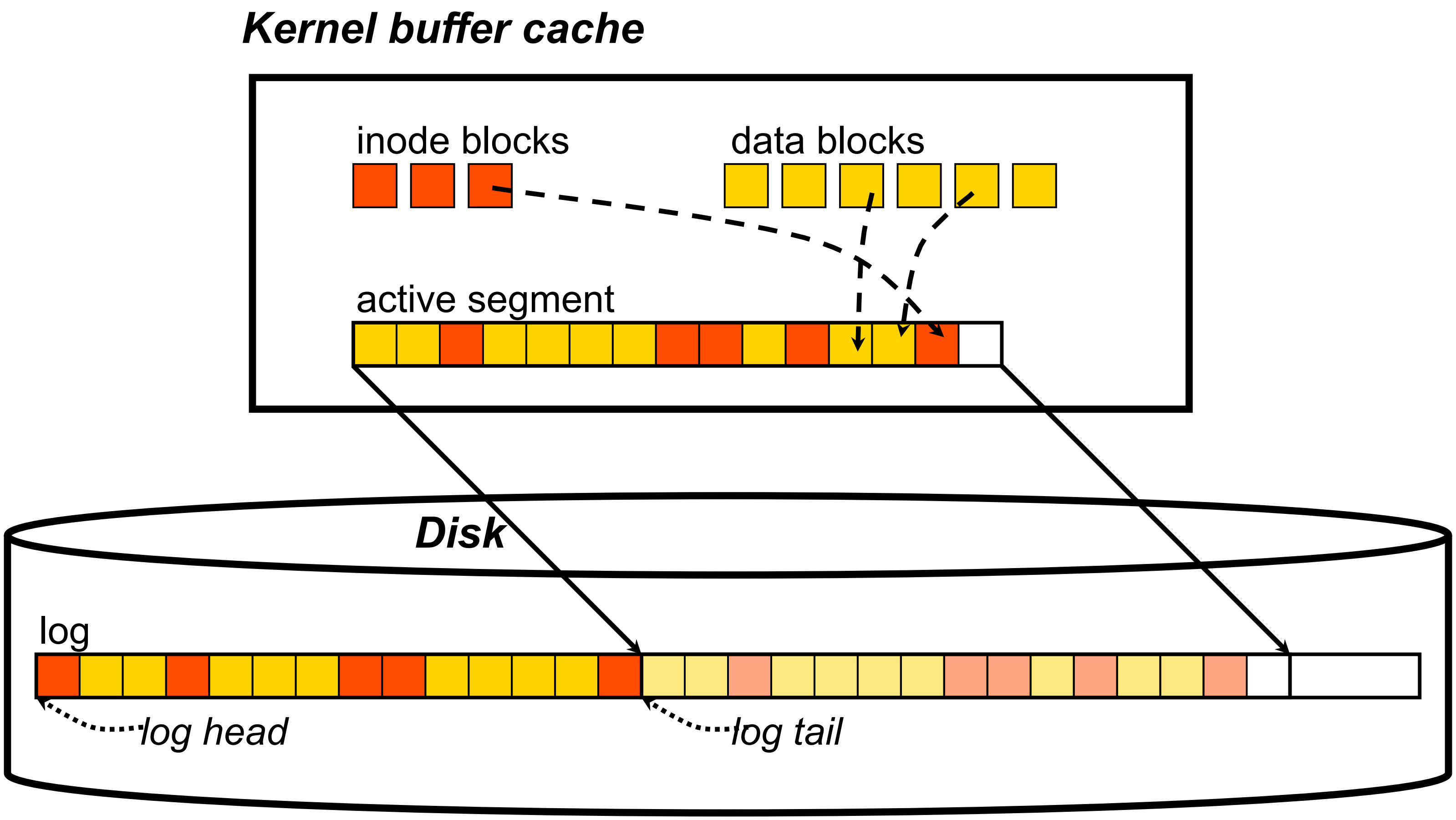
The Log-Structured File System

- Technology Trends
  - I/O becoming more and more of a bottleneck
  - CPU speed increases faster than disk speed
  - Big Memories: Caching improves read performance
  - Most disk traffic are writes
- Little improvement in write performance
  - Synchronous writes to metadata
  - Metadata access dominates for small files
  - e.g. Five seeks and I/Os to create a file
    - file i-node (create), file data, directory entry, file i-node (finalize), directory i-node (modification time).

LFS in a nutshell

- Boost write throughput by writing all changes to disk contiguously
  - Disk as an array of blocks, append at end
  - Write data, indirect blocks, inodes together
  - No need for a free block map
- Writes are written in *segments*
  - ~1MB of continuous disk blocks
  - Accumulated in cache and flushed at once
- Data layout on disk
  - "temporal locality" (good for writing) rather than "logical locality" (good for reading).
  - Why is this a better?
    - Because caching helps reads but not writes!

Log operation



LFS design

- Increases write throughput from 5-10% of disk to 70%
  - Removes synchronous writes
  - Reduces long seeks
- Improves over FFS
  - "Not more complicated"
  - Outperforms FFS except for one case

LFS challenges

- Log retrieval on cache misses
  - Locating inodes
- What happens when end of disk is reached?

Locating inodes

- Positions of data blocks and inodes change on each write
  - Write out inode, indirect blocks too!
- Maintain an inode map
  - Compact enough to fit in main memory
  - Written to disk periodically at *checkpoints*
    - Checkpoints (map of inode map) have special location on disk
    - Used during crash recovery

Cleaning the log: “Achilles Heel”

- Log is infinite, but disk is finite
  - Reuse the old parts of the log
- Clean old segments to recover space
  - Writes to disk create holes
  - Segments ranked by "liveness", age
  - Segment cleaner "runs in background"
- Group slowly-changing blocks together
  - Copy to new segment or "thread" into old

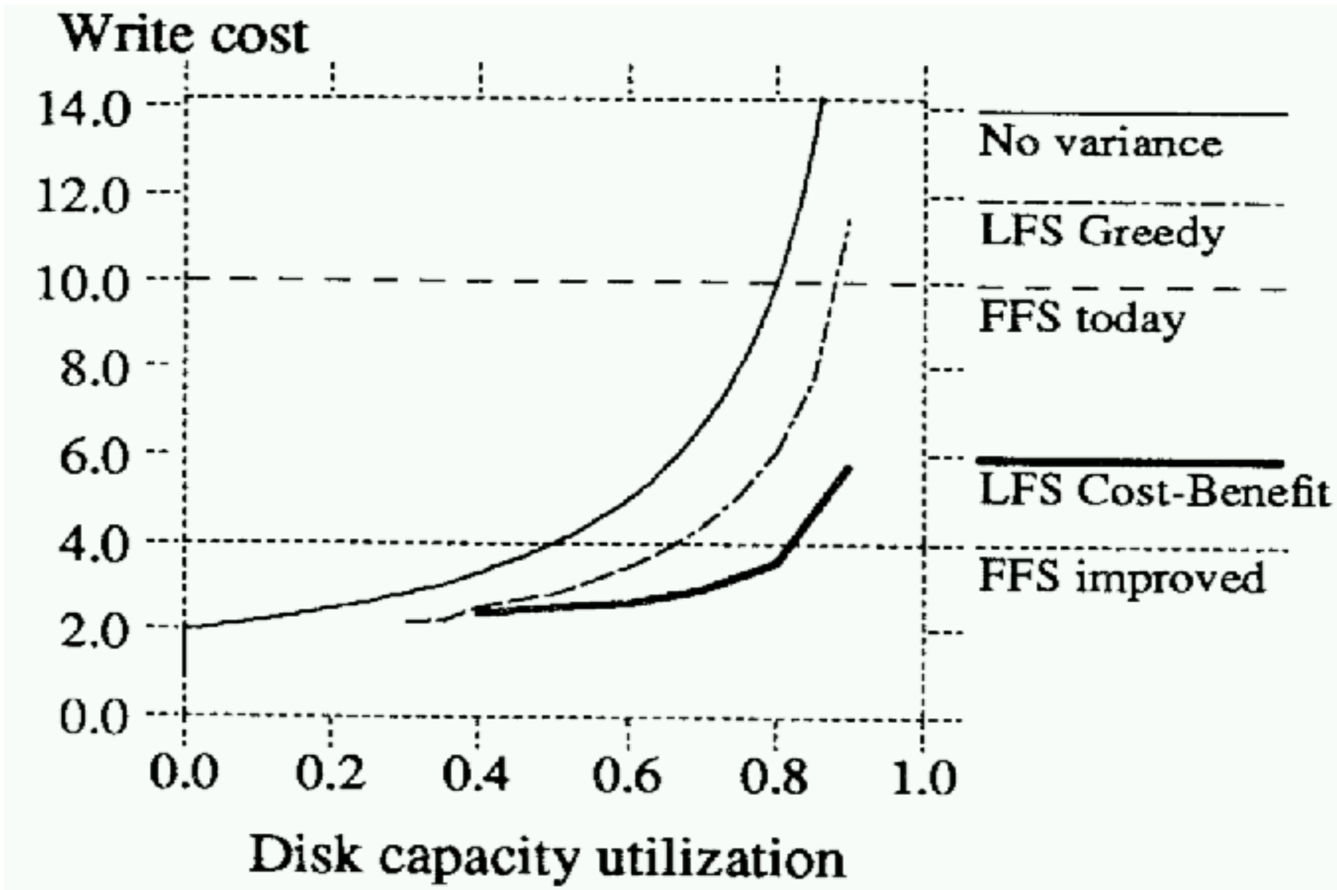
Cleaning policies

- Simulations to determine best policy
  - Greedy: clean based on low utilization
  - Cost-benefit: use age (time of last write)

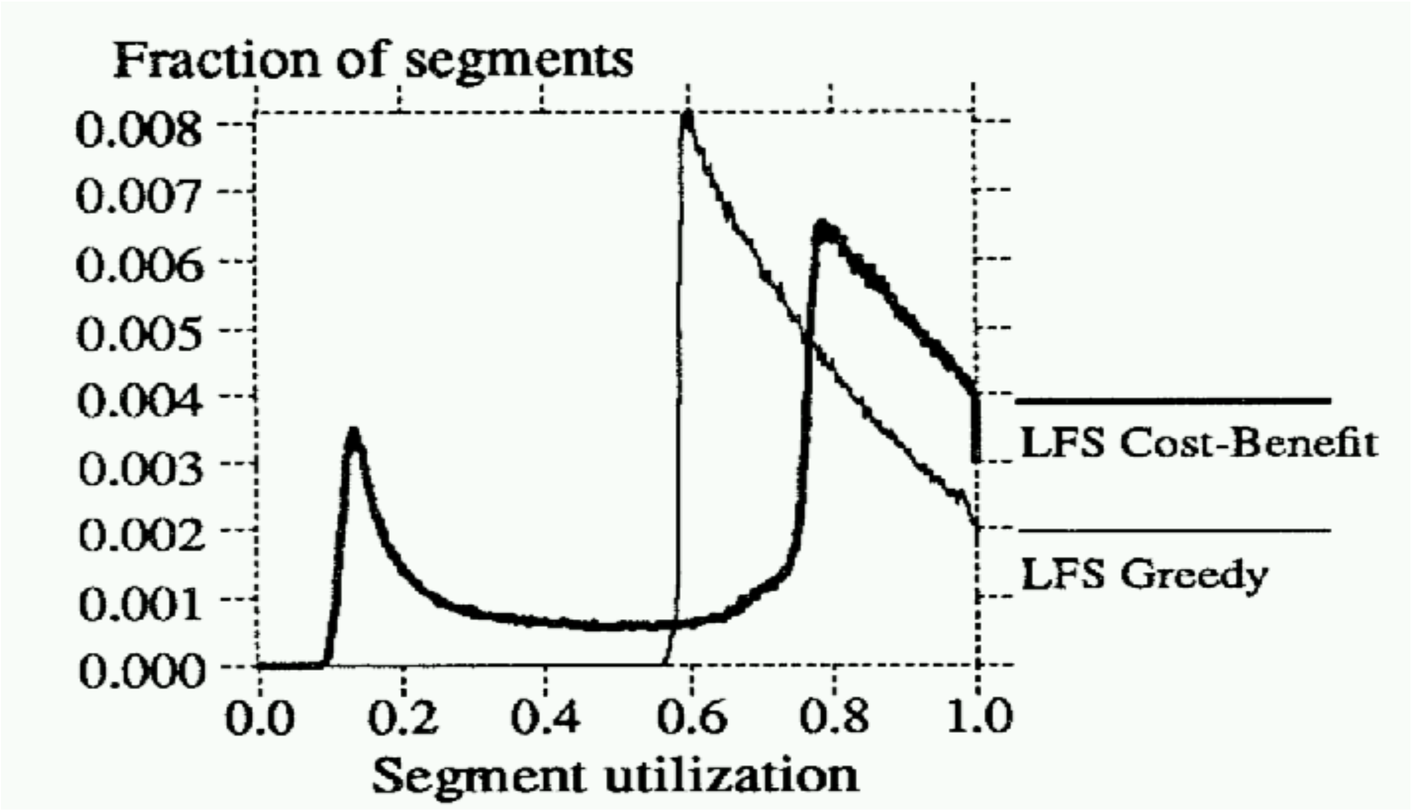
$$\frac{\texttt{benefit}}{\texttt{cost}} = \frac{\texttt{(free space generated)*(age of segment)}}{\texttt{cost}}$$

- Measure *write cost*
  - Time disk is busy for each byte written
  - Write cost 1.0 = no cleaning

Greedy versus Cost-benefit



Cost-benefit segment utilisation



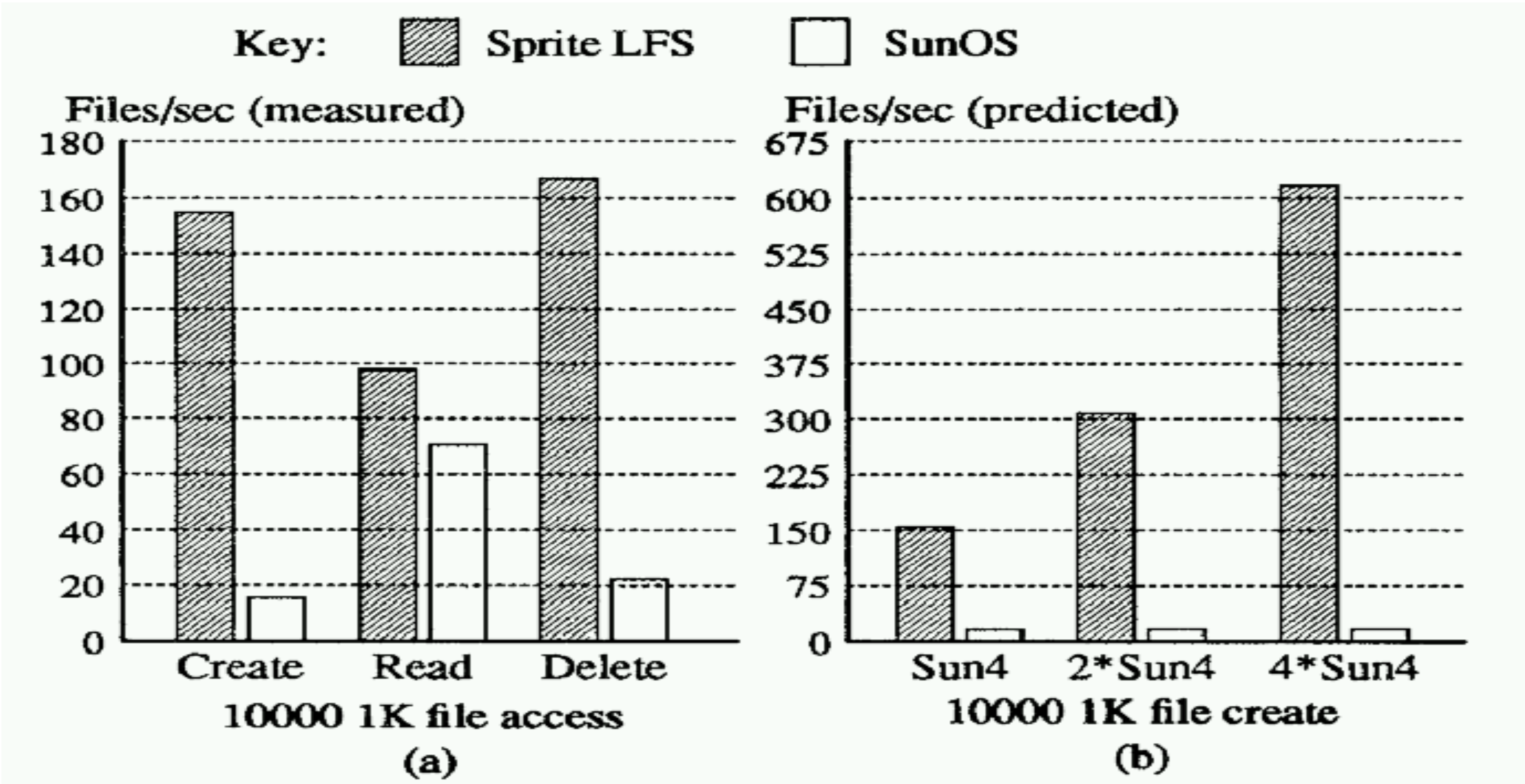
LFS crash recovery

- Log and checkpointing
  - Limited crash vulnerability
  - At checkpoint flush active segment, inode map
- No **`fsck`** required

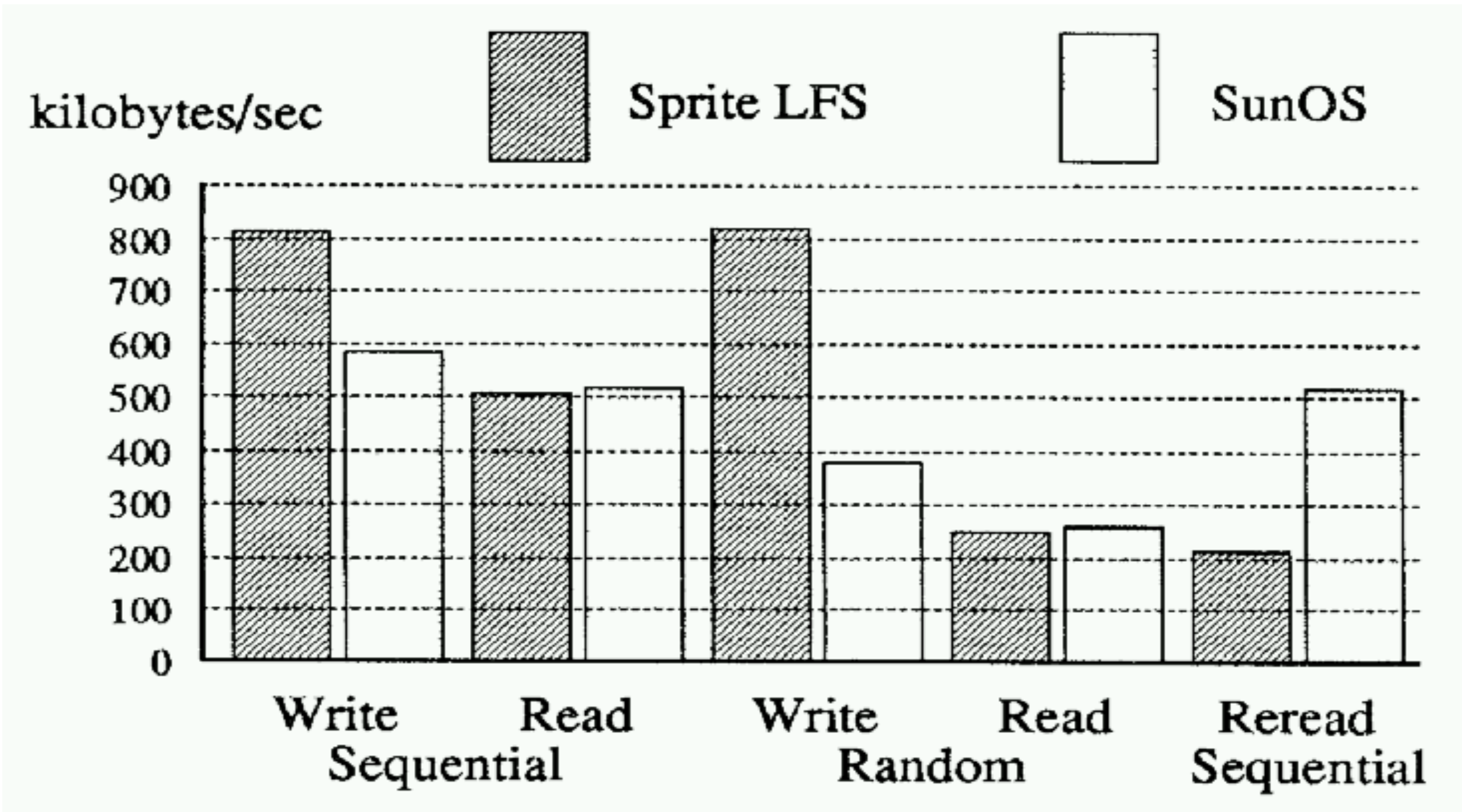
LFS performance

- Cleaning behaviour better than simulated predictions
- Performance compared to SunOS FFS
  - Create-read-delete 10000 1k files
  - Write 100-MB file sequentially, read back sequentially and randomly

Small-file performance



Large-file performance



Perspective

- Features
  - CPU speed increasing faster than disk => I/O is bottleneck
  - Write FS to log and treat log as truth; use cache for speed
  - Problem
    - Find/create long runs of (contiguous) disk space to write log
  - Solution
    - clean live data from segments,
    - picking segments to clean based on a cost/benefit function
- Flaws
  - Intra-file Fragmentation: LFS assumes entire files get written
  - If small files “get bigger”, how would LFS compare to UNIX?
- Lesson
  - Assumptions about primary and secondary in a design
  - LFS made log the truth instead of just a recovery aid

Conclusions

- Papers were separated by 8 years
  - Much controversy regarding LFS-FFS comparison
- Both systems have been influential
  - IBM Journalling file system
  - Ext3 filesystem in Linux
  - Soft updates come enabled in FreeBSD

Next Time

- Read and write review:
  - *Lightweight Recoverable Virtual Memory*, M. Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. Proceedings of the fourteenth ACM symposium on Operating systems principles, 1994, pages 146--160.
  - *The evolution of Coda*, M. Satyanarayanan. ACM Transactions on Computer Systems, Volume 20, Issue 2 (May 2002), pages 85--124

Next Time

- Read and write review:
- Lab 1 – available later today and due next Friday
- Project Proposal due next week, next Thursday
  - Possible projects presentations yesterday, slides online
  - Also, talk to faculty and email and talk to me
- Check website for updated schedule

Overview of talk

- Unix Fast File System
- Log-Structured System
- Soft Updates
- Conclusions

Soft updates

- Alternative mechanism for improving performance of writes
  - All metadata updates can be asynchronous
  - Improved crash recovery
  - Same on-disk structure as FFS

The metadata update problem

- Disk state must be consistent enough to permit recovery after a crash
  - No dangling pointers
  - No object pointed to by multiple pointers
  - No live object with no pointers to it
- FFS achieves this by synchronous writes
  - Relaxing sync. writes requires update sequencing or atomic writes

Design constraints

- Do not block applications unless **`fsync`**
- Minimise writes and memory usage
- Retain 30-second flush delay
- Do not over-constrain disk scheduler
  - It is already capable of some reordering

Dependency tracking

- Asynchronous metadata updates need ordering information
  - For each write, pending writes which precede it
- Block-based ordering is insufficient
  - Cycles must be broken with sync. writes
  - Some blocks stay dirty for a long time
  - False sharing due to high granularity

Circular dependency example

directory

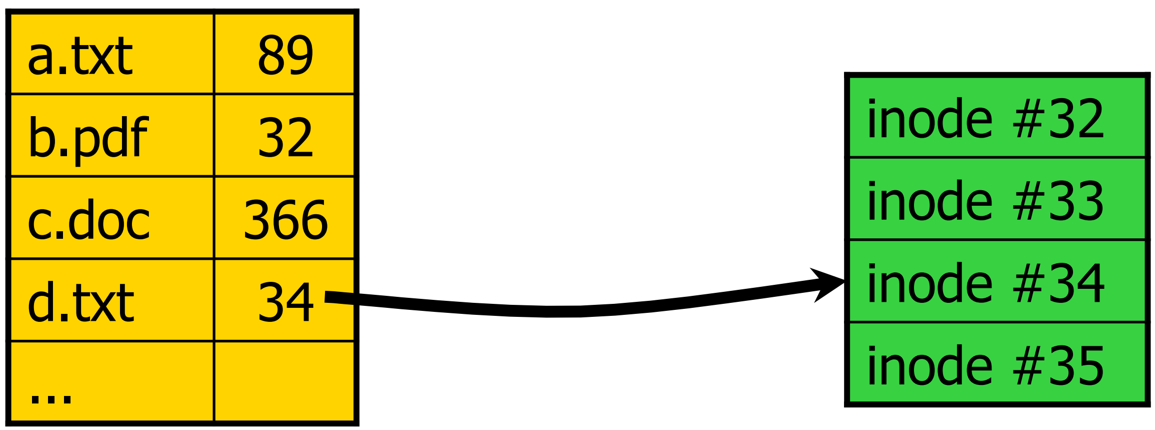
a.txt	89
b.pdf	32
c.doc	366
...	

inode block

inode #32
inode #33
inode #34
inode #35

Circular dependency example

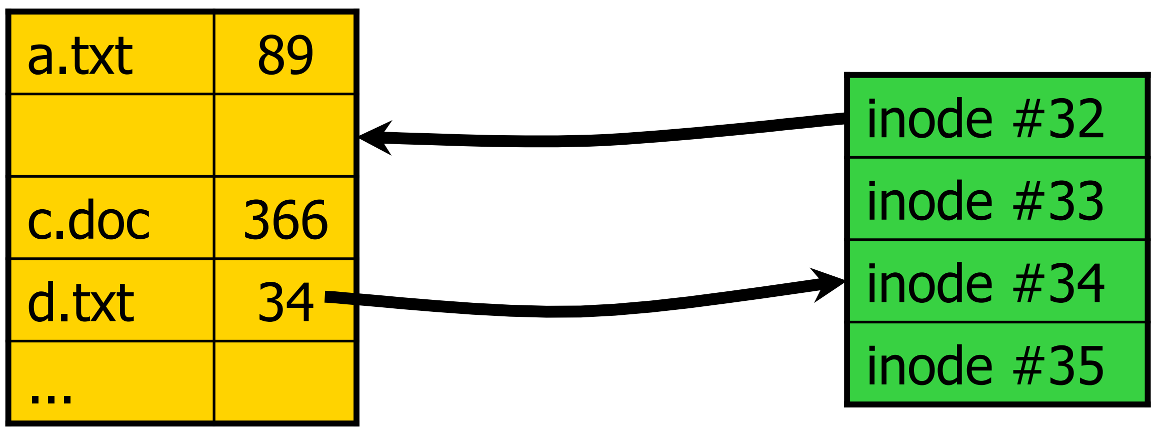
create file d.txt



Inode must be initialised before directory entry is added

Circular dependency example

remove file b.pdf

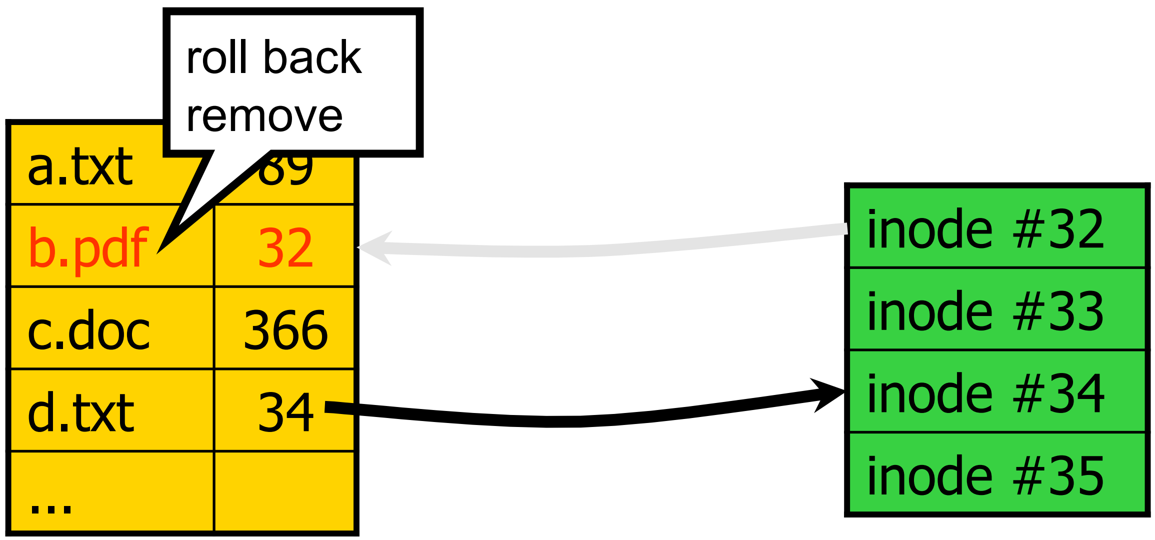


Directory entry must be removed before inode is deallocated

Update implementation

- Update list for each pointer in cache
  - FS operation adds update to each affected pointer
  - Update incorporates dependencies
- Updates have "before", "after" values for pointers
  - Roll-back, roll-forward to break cycles

Circular dependency example



Rollback allows dependency to be suppressed

Soft updates details

- Blocks are locked during roll-back
  - Prevents processes from seeing stale cache
- Existing updates never get new dependencies
  - No indefinite aging
- Memory usage is acceptable
  - Updates block if usage becomes too high

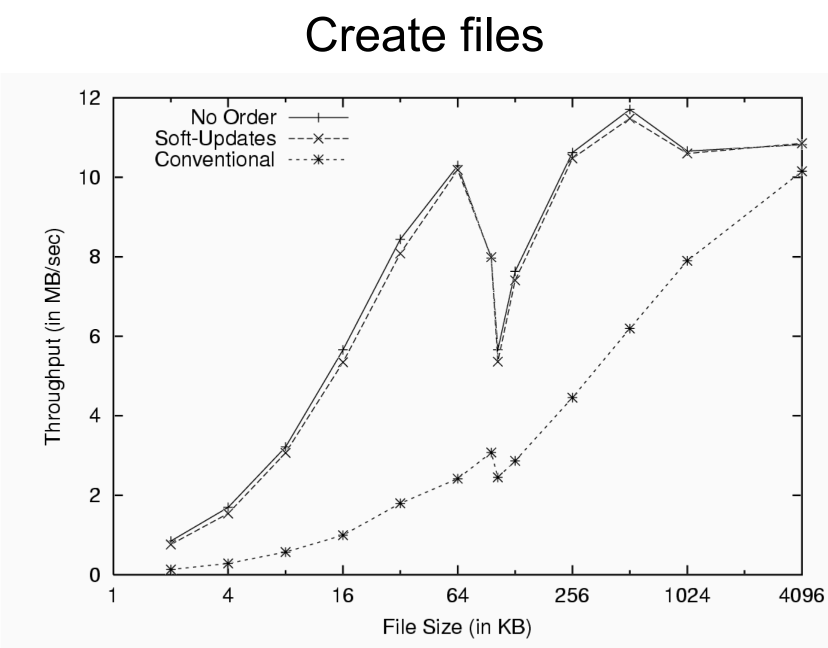
Recovery with soft updates

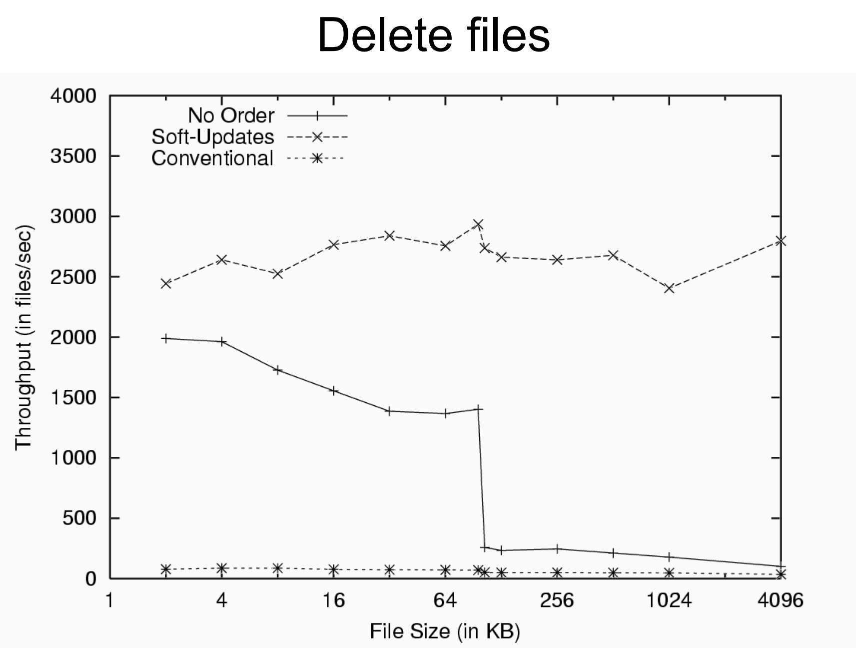
- "Benign" inconsistencies after crashes
  - Freespace maps may miss free entries
  - Link counts may be too high
- **`Fsck`** is still required
  - Need not run immediately
  - Only has to check in-use inodes
  - Can run in the background

Soft updates performance

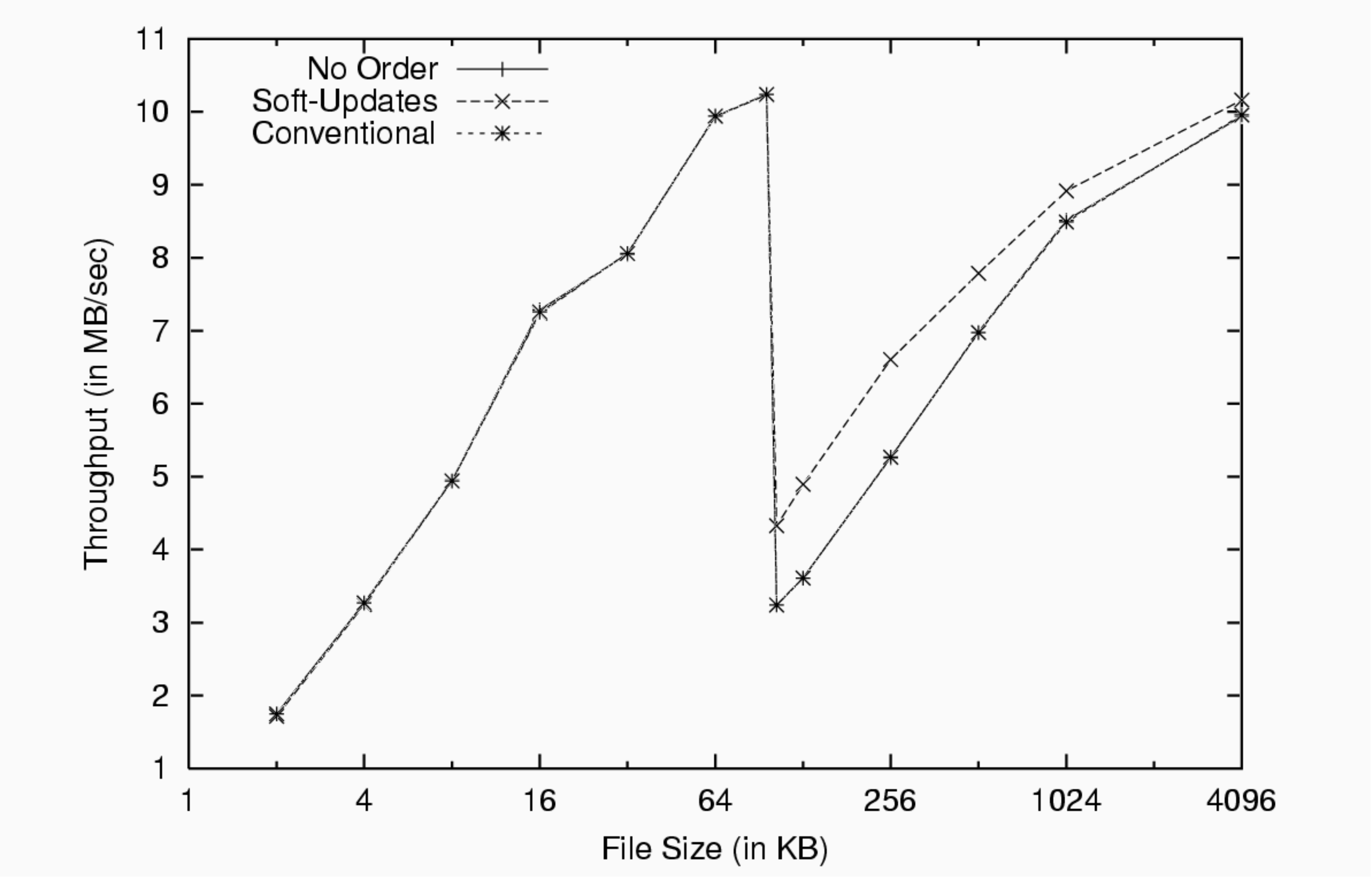
- Recovery time on 76% full 4.5GB disk
  - 150s for FFS **`fsck`** versus 0.35s ...
- Microbenchmarks
  - Compared soft updates, async writes, FFS
  - Create, delete, read for 32MB of files
- Soft updates versus update logging
  - **`Sdet`** benchmark of "user scripts"
  - Various degrees of concurrency

Create and delete performance

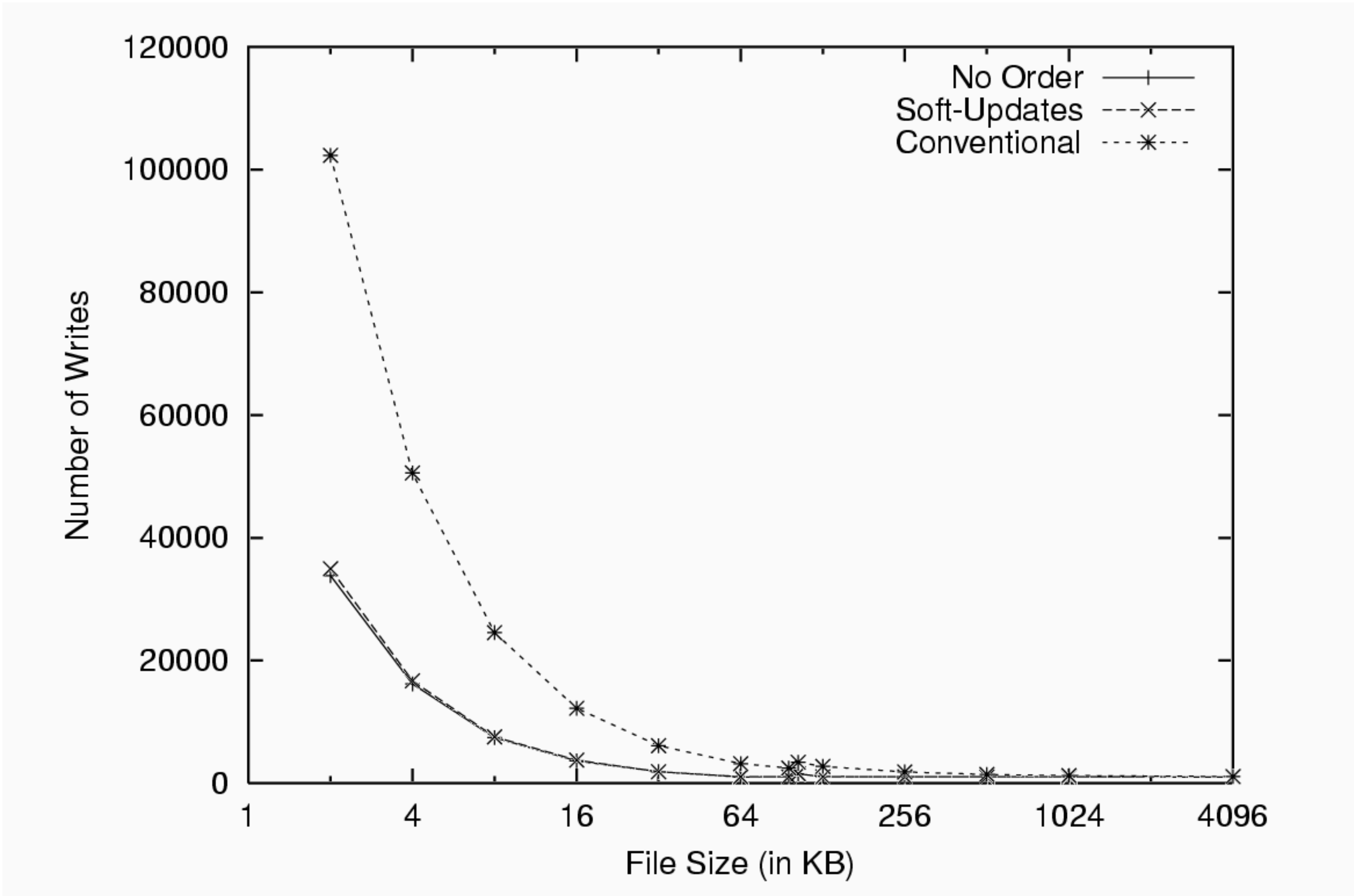




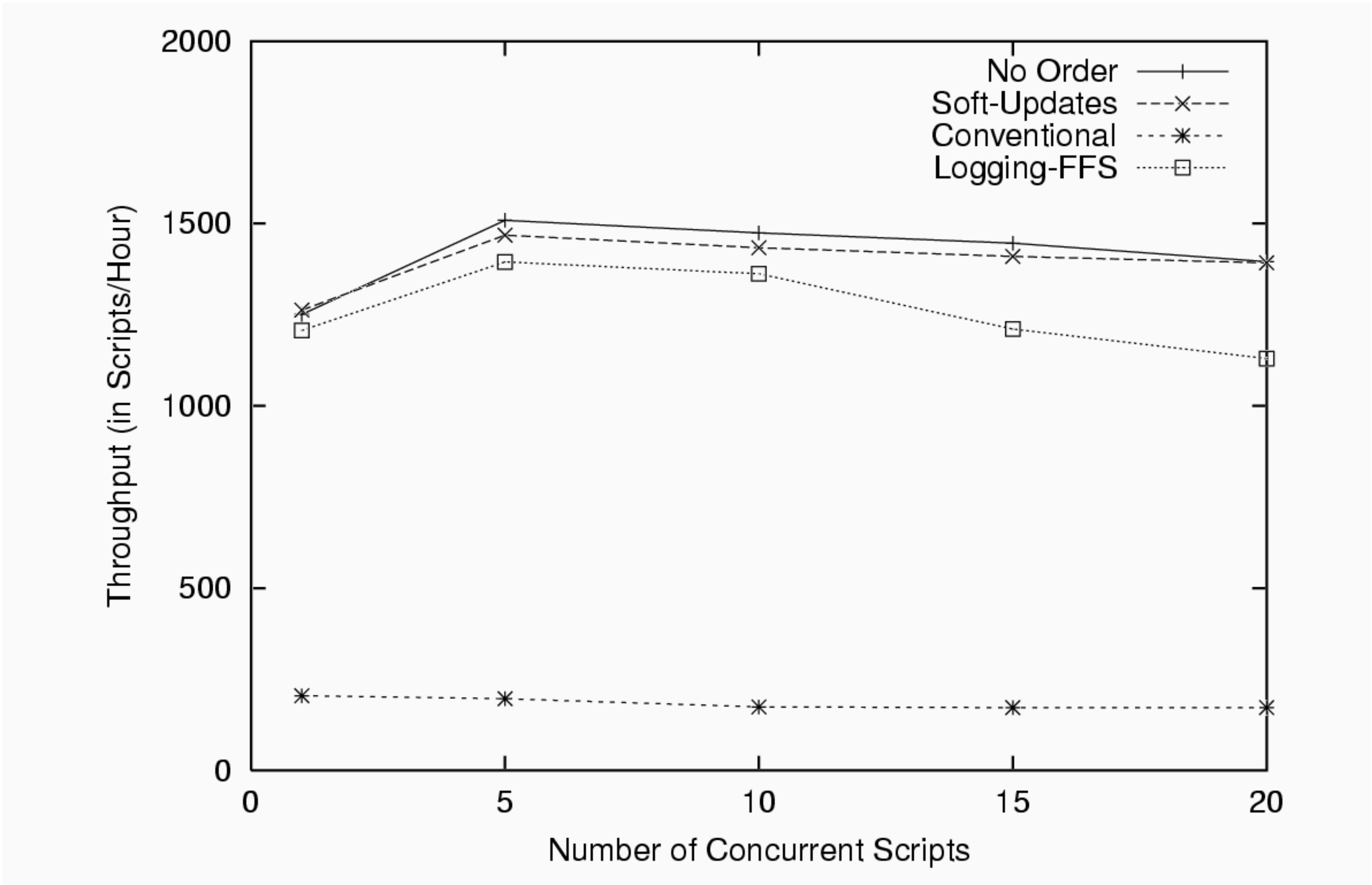
Read performance



Overall create traffic



Soft updates versus logging



Conclusions

- Papers were separated by 8 years
  - Much controversy regarding LFS-FFS comparison
- Both systems have been influential
  - IBM Journalling file system
  - Ext3 filesystem in Linux
  - Soft updates come enabled in FreeBSD

Next Time

- Read and write review:
  - *SEDA: An Architecture for Well Conditioned, Scalable Internet Services*, Matt Welsch, David Culler, and Eric Brewer. Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (Banff, Alberta, Canada, 2001), pages 230--243
  - *On the duality of operating system structures*, H. C. Lauer and R. M. Needham. ACM SIGOPS Operating Systems Review Volume 12, Issue 2 (April 1979), pages 3--19.

Next Time

- Read and write review:
- Lab 1 – available later today and due next Friday
- Project Proposal due next week, next Thursday
  - Possible projects presentations yesterday, slides online
  - Also, talk to faculty and email and talk to me
- Check website for updated schedule