

CSE 120

Principles of Operating Systems

Fall 2007

Lecture 15: Fast Metadata Updates: Rio, Journaling
and Soft Updates

Keith Marzullo

A metadata update problem

- Consider creating a file A, which is assigned an inode X.
 - (a) the link <A, X> is added to a directory.
 - (b) the inode X is marked as allocated.
- Since these writes are to different disk blocks, one will be written first. Which one is preferable to be first?

Metadata update rules

Metadata update rules that ensure consistency in the face of failures:

1. Never point to a structure before it has been initialized.
2. Never reuse a resource before nullifying all previous pointers to it.
3. Never reset the last pointer to a live resource before a new pointer has been set.

Traditional (FFS) approach

Reliability-induced synchronous writes

- (a) the link $\langle A, X \rangle$ is added to a directory.
- mark in-memory copy of inode X as allocated.
- force inode X to disk.
- add link $\langle A, X \rangle$ to buffered block of directory.

Rio approach

- Why is disk more stable than physical memory?
 - Battery backup removes chances of suddenly losing power.
 - Processors don't have to reset memory upon reboot, and so old value of memory can persist across crashes.
- ... so, why not recover from file buffer cache in memory?
 - Do modification of file buffers in the correct order.
 - Force contents of unflushed file buffers to disk upon restart.
- What do you think?

Protecting the buffer cache

- Write protect the buffer cache.
 - File cache procedures enable writing to buffer cache before writing a page and disable it afterwards.
 - The only time a file cache page is vulnerable is when it is being modified by a file cache procedure.
 - Existing systems already have such vulnerabilities.
 - Similar protection needed if kernel uses memory-mapped files.
 - User memory-mapped files pose no additional vulnerabilities.

How do you know?

- Use *fault injection*
 - Inject 20 faults.
 - If no crash within 10 minutes, discard run and reboot (50% of the time).
- Checked for corruption
 - *Direct*: a series of events eventually causes a procedure (usually not I/O procedure) to accidentally write to file data.
 - Find by using checksums of memory blocks in file cache.
 - *Indirect*: a series of events eventually causes a procedure to call an I/O procedure with the wrong parameters.
 - Find by creating deterministic *memTest* that generates a repeatable stream of file actions.

Fault injection results

Fault Type	Disk-Based	Rio without Protection	Rio with Protection
kernel text	2	1	
kernel heap			
kernel stack		1	1
destination reg.			
source reg.	2		
delete branch	1	1	1
delete random inst.	1		
initialization			1
pointer		1	
allocation			
copy overrun		4	
off-by-one	1	2	1
synchronization			
Total	7 of 650 (1.1%)	10 of 650 (1.5%)	4 of 650 (0.6%)

Performance results

	Data Permanent	cp+rm (seconds)	Sdet (5 scripts) (seconds)	Andrew (seconds)
Memory File System	never	21 (15+6)	43	13
UFS with delayed data and metadata	after 0-30 seconds, asynchronous	81 (76+5)	47	13
AdvFS (log metadata updates)	after 0-30 seconds, asynchronous	125 (110+15)	132	16
UFS	data after 64 KB, asynchronous metadata synchronous	332 (245+87)	401	23
UFS with write-through after each close	after close, synchronous	394 (274+120)	699	49
UFS with write-through after each write	after write, synchronous	539 (419+120)	910	178
Rio without protection	after write, synchronous	24 (18+6)	42	12
Rio with protection	after write, synchronous	25 (18+7)	42	13

Journaling approach

Maintain an *auxiliary log* that records all metadata operations.

- Ensure that log is written to disk before any pages containing data modified by the corresponding operations.
- When recovering from a crash, any unapplied metadata operations in the log are applied.

Variations

- Store log in pre-allocated file that is maintained as a circular buffer.
 - File cache buffers point to last entry in log that refers to them, so writing back buffer requires log to be forced out to that point.
 - Can also do synchronous updates.
- Store log in auxiliary file system that supports logs.
 - Locate on same drive as primary file system.
 - Locate on separate drive.

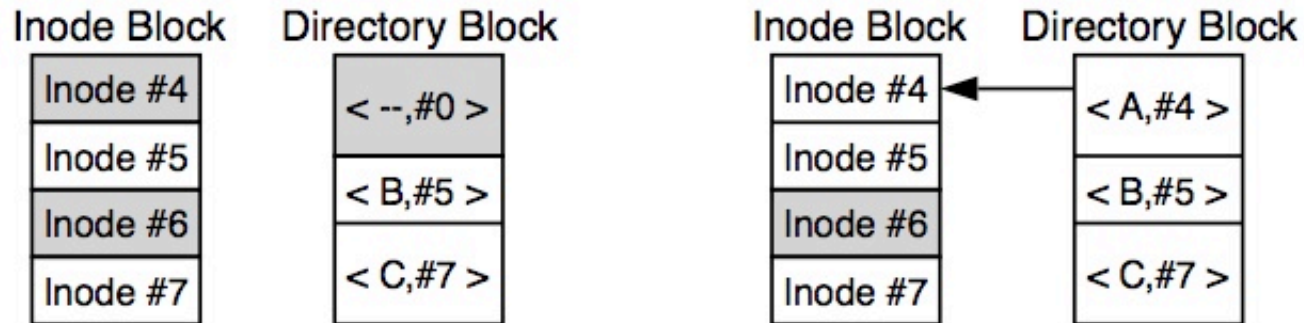
Soft Updates approach

Be lazy about writing back, but follow the ordering rules.

- Use delayed writes for metadata.
- Maintain dependency information that specifies the order in which data must be written back to disk.
- When forcing out a block, first force out all blocks that need to be written first (and haven't yet made it out).

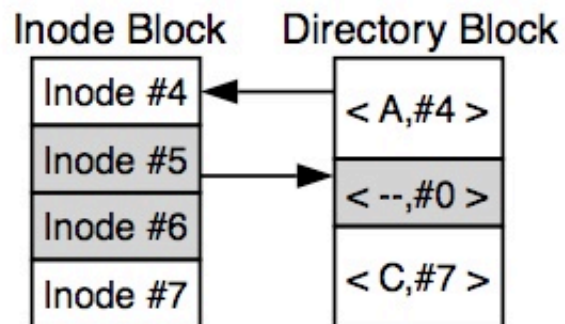
There's a hitch, though...

Circular dependencies



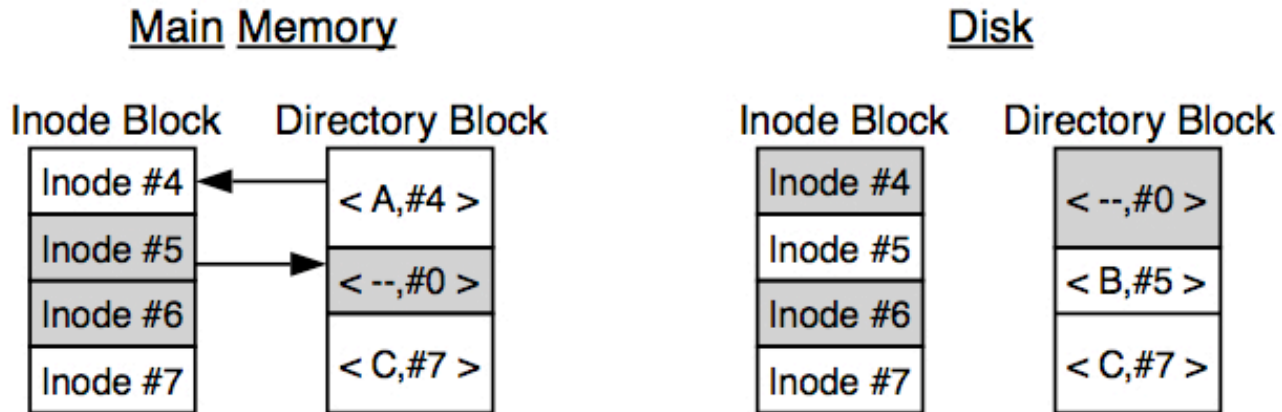
(a) Original Organization

(b) Create File A

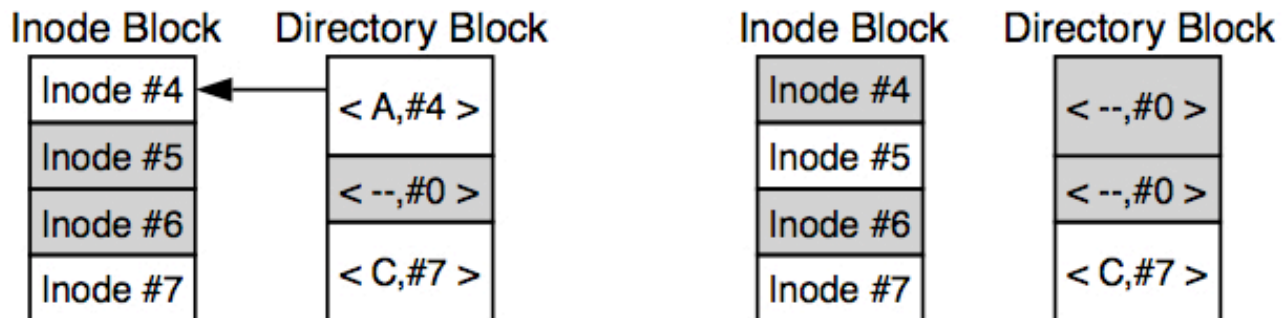


(c) Remove File B

Breaking circular dependencies: 1

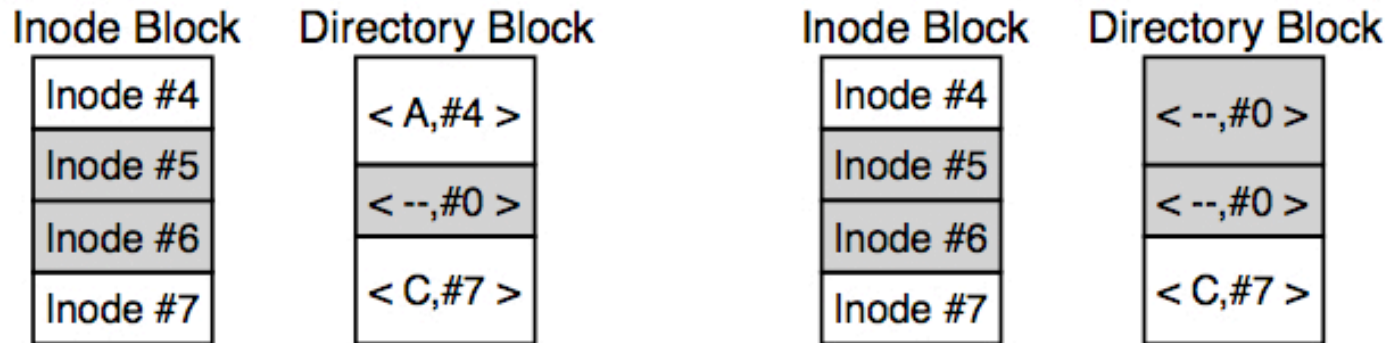


(a) After Metadata Updates

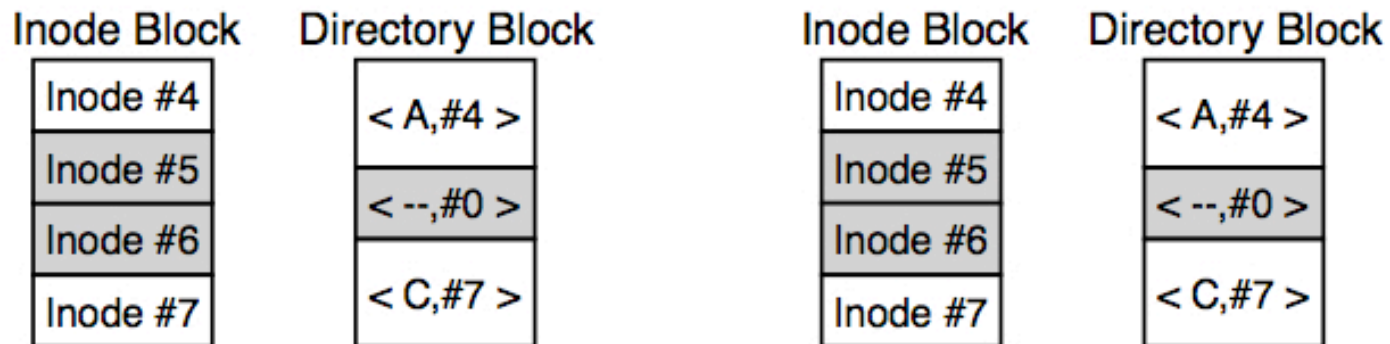


(b) Safe Version of Directory Block Written

Breaking circular dependencies: 2



(c) Inode Block Written



(d) Directory Block Written

Another Soft Updates feature

Soft Updates also defers work in some cases.

- When a file is deleted:
 - Remove the file name from the directory hierarchy.
 - Create a remove dependency associated with the buffer holding the corresponding directory entry.
 - When buffer is written, all delete dependencies are passed to a separate background syncer task.
 - When syncer finally runs (every minute or so) it walks inode and frees associated file data blocks.

Comparisons

	File System Configurations
FFS	Standard FFS
FFS-async	FFS mounted with the async option
Soft-Updates	FFS mounted with Soft Updates
LFFS-file	FFS augmented with a file log log writes are asynchronous
LFFS-wafs-1sync	FFS augmented with a WAFS log log writes are synchronous
LFFS-wafs-1async	FFS augmented with a WAFS log log writes are asynchronous
LFFS-wafs-2sync	FFS augmented with a WAFS log log is on separate disk log writes are synchronous
LFFS-wafs-2async	FFS augmented with a WAFS log log is on a separate disk log writes are asynchronous

Feature	File Systems
Meta-data updates are synchronous	FFS, LFFS-wafs-[12]sync
Meta-data updates are asynchronous	Soft Updates LFFS-file LFFS-wafs-[12]async
Meta-data updates are atomic.	LFFS-file LFFS-wafs-[12]*
File data blocks are freed in back-ground	Soft Updates
New data blocks are written before inodes	Soft Updates
Recovery requires full file system scan	FFS
Recovery requires log replay	LFFS-*
Recovery is non-deterministic and may be impossible	FFS-async

Microbenchmarks

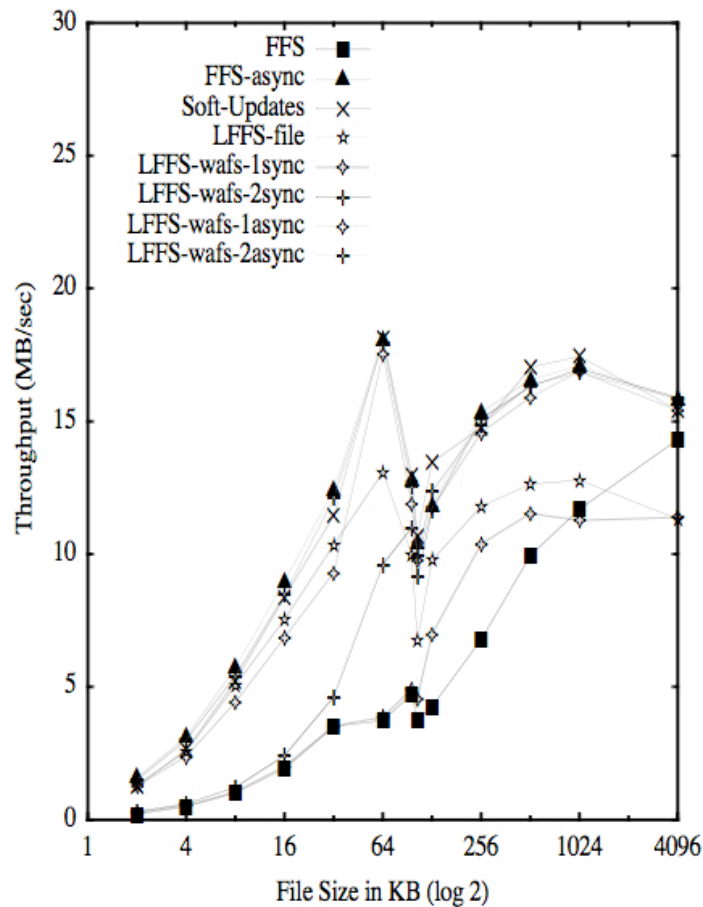


Figure 1. Create Performance as a Function of File Size.

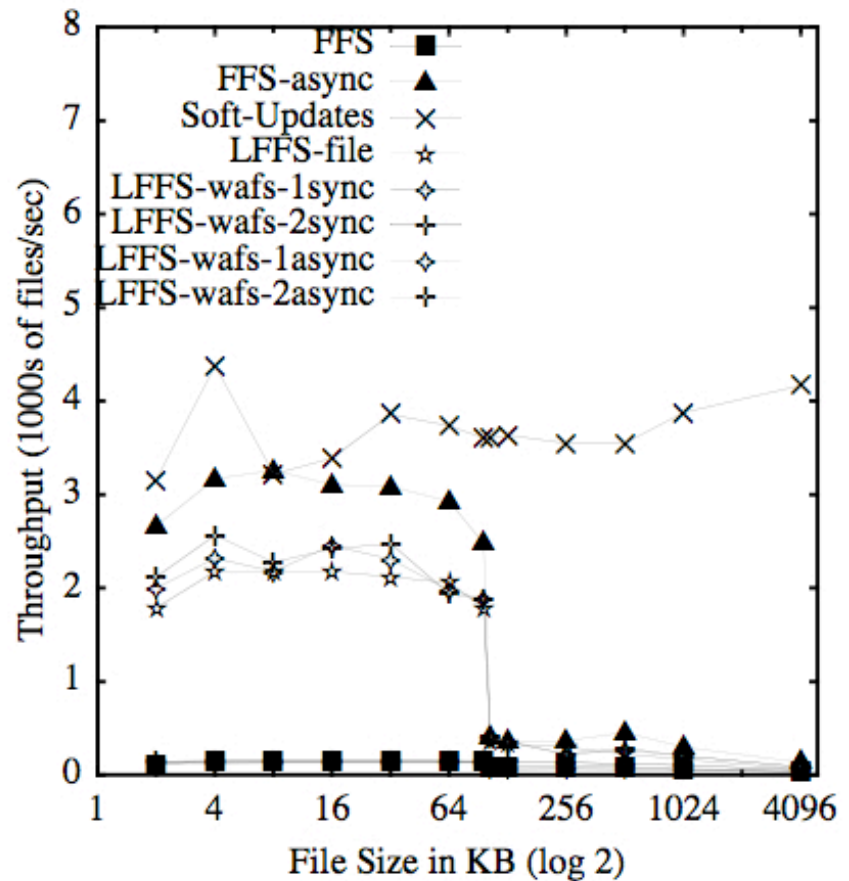


Figure 2. Delete Performance as a Function of File Size.

Macrobenchmark

	Unbatch	Expire	Total
	Absolute Time (in seconds)		
FFS-async	1282	640	1922
	Perf. Relative to FFS-async		
FFS	0.63	0.40	0.53
Soft-Updates	0.86	0.89	0.87
LFFS-file	0.95	0.95	0.95
LFFS-wafs-1sync	0.67	0.48	0.59
LFFS-wafs-1async	0.98	0.92	0.96
LFFS-wafs-2sync	0.91	0.67	0.81
LFFS-wafs-2async	0.97	0.95	0.96

Table 5. Netnews Results Normalized to FFS-async. These results are based on a single run, but we observed little variation between multiple runs of any configuration.

Summary

- Metadata consistency can be addressed in several ways.
 - Reliability-induced synchronous writes.
 - Ordered writes to resilient RAM.
 - Logging metadata operations.
 - Imposing an order on writes (which requires some rollback to avoid circular dependencies).

Next class...

- Virtual machines