

CS161: Operating Systems

Matt Welsh
mdw@eecs.harvard.edu



Lecture 15: The Berkeley Fast File System
April 10, 2007

Berkeley FFS

Motivated by performance problems with older UNIX filesystems:

- Older UNIX FS had small blocks (512 bytes)
- Free list was unordered; no notion of allocating chunks of space at a time
- inodes and data blocks may be located far from each other (long seek time)
- Related files (in same directory) might be very far apart
- No symbolic links, file locking, limited filenames (14 chars), no quotas

Main goal of FFS was to improve performance:

- Use a larger block size – *why does this help??*
- Allocate blocks of a file (and files in same directory) near each other on the disk

Entire filesystem described by a *superblock*

- Contains free block bitmap, location of root directory inode, etc.
- Copies of superblock stored at multiple locations on disk (for safety)

FFS Cylinder Groups

Store related blocks on nearby tracks but on different platters

- That is, a whole *group of cylinders*:

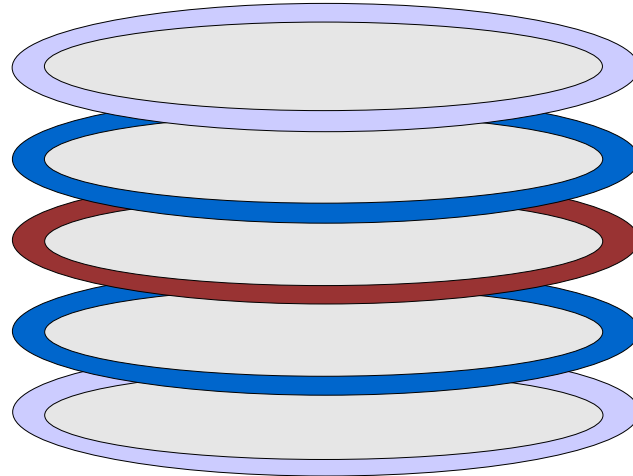
data blocks

inode blocks

superblock

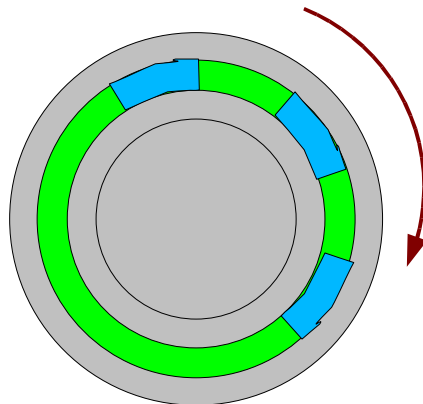
inode blocks

data blocks



Allocate blocks in a rotationally optimal fashion:

- Try to estimate rotation speed of disk and allocate next block where the disk head will happen to be when the next read will be ready!



Does this stuff matter anymore?

Modern disks have a lot of internal buffering and logic

- Batch multiple write requests into a single write
- Internally reorder multiple outstanding requests
- Caching on disk (8 MB typical)
- Internal remapping of bad blocks to different places on the physical disk
- OS has little information on physical disk geometry anyway!
 - *Blocks with similar block #'s are usually close to each other, but that's about it...*

So, how useful are this fancy OS-driven block layout techniques?

- Clearly used to have significant impact on disk performance
- These days, not clear that they are so useful

Still, lots of debate in the FS community about this

- Modern filesystems still use notion of block and cylinder grouping
- Argument that OS can know more about the workload, multiple users, different request priorities, and tradeoffs in terms of bandwidth vs. latency

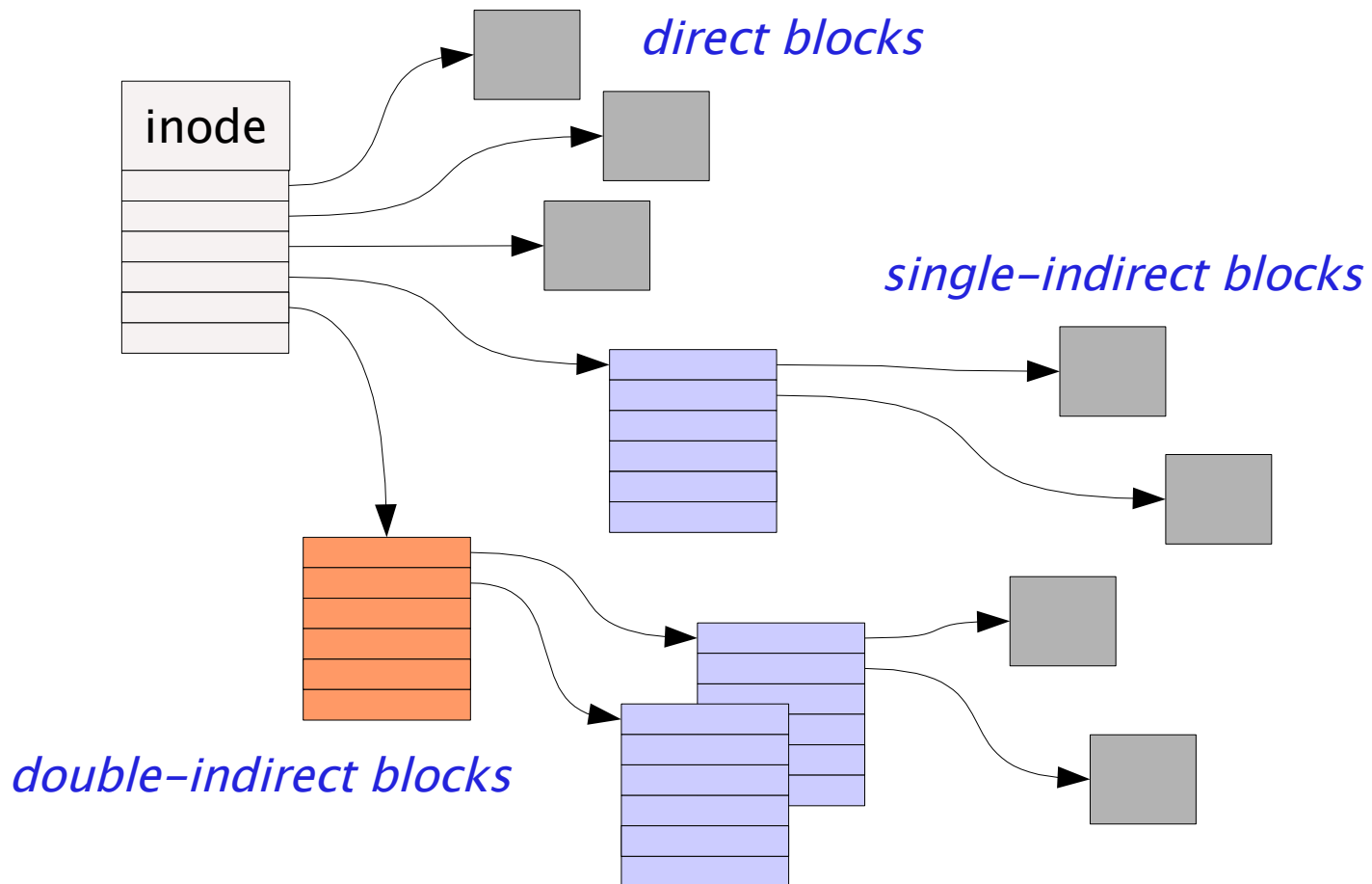
Recall: Multilevel Indexed Files

Inode contains a list of 10-15 *direct blocks*

- First few blocks of file

Also contains a pointer to a *single indirect*, *double indirect*, and *triple indirect* blocks

- Allows file to grow to be incredibly large!!!



Maximum File Size

Assume 1 KB blocks. How large can a file be with...

Single-level indirect table?

- How many block pointers can be stored in one block?
- Assume 4 bytes per block pointer, so $(1\text{KB} / 4) = 256$ blocks
- So ... $256 * 1\text{KB} = 256 \text{ KB}$

Double-level indirect table?

- $256 * 256 * 1\text{KB} = 65536 \text{ KB} = 64 \text{ MB}$

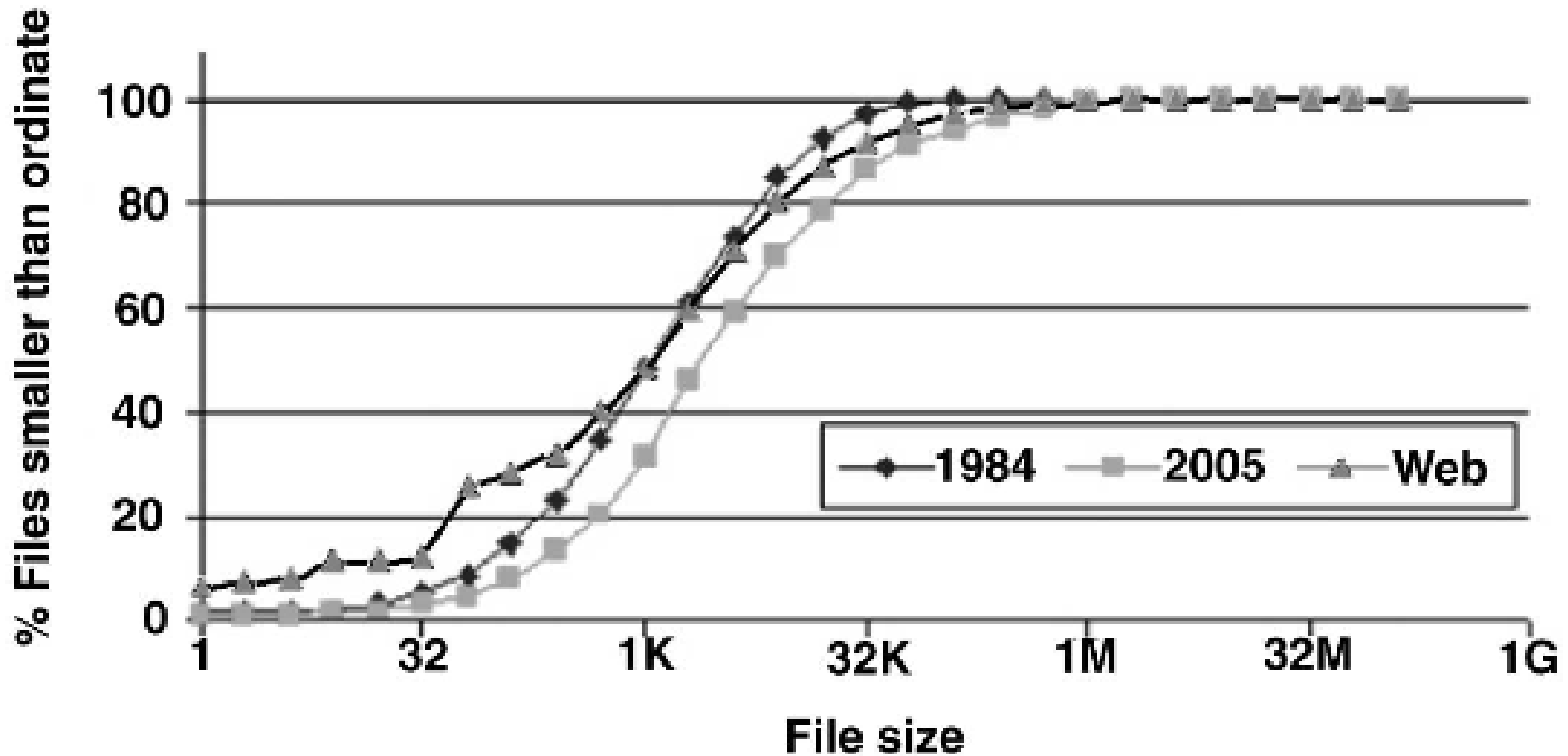
Triple-level indirect table?

- $256 * 256 * 256 * 1\text{KB} = 16 \text{ GB}$

FFS-style: 13 direct blocks, 1 single indirect, 1 double indirect, 1 triple indirect?

- $(13 * 1\text{KB}) + (256 * 1\text{KB}) + (256 * 256 * 1\text{KB}) + (256 * 256 * 256 * 1\text{KB}) = 16.06 \text{ GB}$
 - *So why use this wacko multi-level indirection scheme rather than just a triple-level indirect table???*

File size distribution



Data from Tanenbaum et al, "File Size Distribution on UNIX Systems – Then and Now", OSR Jan'06

65% of files smaller than 4KB

FFS Block Sizes

Older UNIX filesystems used small blocks (512B or 1KB)

- Low disk bandwidth utilization
- Maximum file size is limited (how many blocks each inode could keep track of)

FFS introduced larger block sizes (4KB)

- Allows multiple sectors to be read/written at once
- Introduces *internal fragmentation*: a whole block may not be used

Fix: Block “fragments” (1KB)

- The last block in a file may consist of 1, 2, or 3 fragments
- Fragments from different files stored on the same block
 - *inode needed to store both block ID and “fragment index” of fragment within block*

Evolution of FFS

The last 15+ years of filesystems literature has many improvements to the basic FFS design.

What are some of the FFS limitations?

- Inability to “stream” multiple contiguous blocks in a track
- Seeks between inodes and data blocks
- Synchronous updates to metadata: force a sync when files created, deleted, or renamed

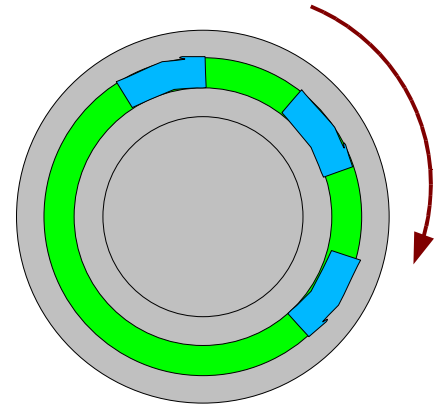
Some improvements (discussed in detail next...)

- Extent-based transfers
- Block remapping on writes
- Colocating directories and small files
- Soft metadata updates

Extent-based transfers

Recall: FFS adds a gap between sectors on a track

- Try to take advantage of rotational latency for performing next read or write operation
- Problem: Hurts performance for multi-sector I/O!
 - *FFS cannot achieve the full transfer rate of the disk for large, contiguous reads or writes.*



Possible fix: Just get rid of the gap between sectors

- Problem: “Dropped rotation” between consecutive reads or writes: have to wait for next sector to come around under the heads.

Hybrid approach - “extents” [McVoy, USENIX'91]

- Group blocks into “extents” or clusters of contiguous blocks
- Try to do all I/O on extents rather than individual blocks
- To avoid wasting I/O bandwidth, only do this when FS detects sequential access
 - *Kind of like just increasing the block size...*

Block remapping

Problem: Block numbers are allocated when they are first written

- FS maintains a free list of blocks and simply picks the first block off the list
 - *No guarantee that these blocks will be contiguous for a large write!*
- A single file may end up with blocks scattered across the disk

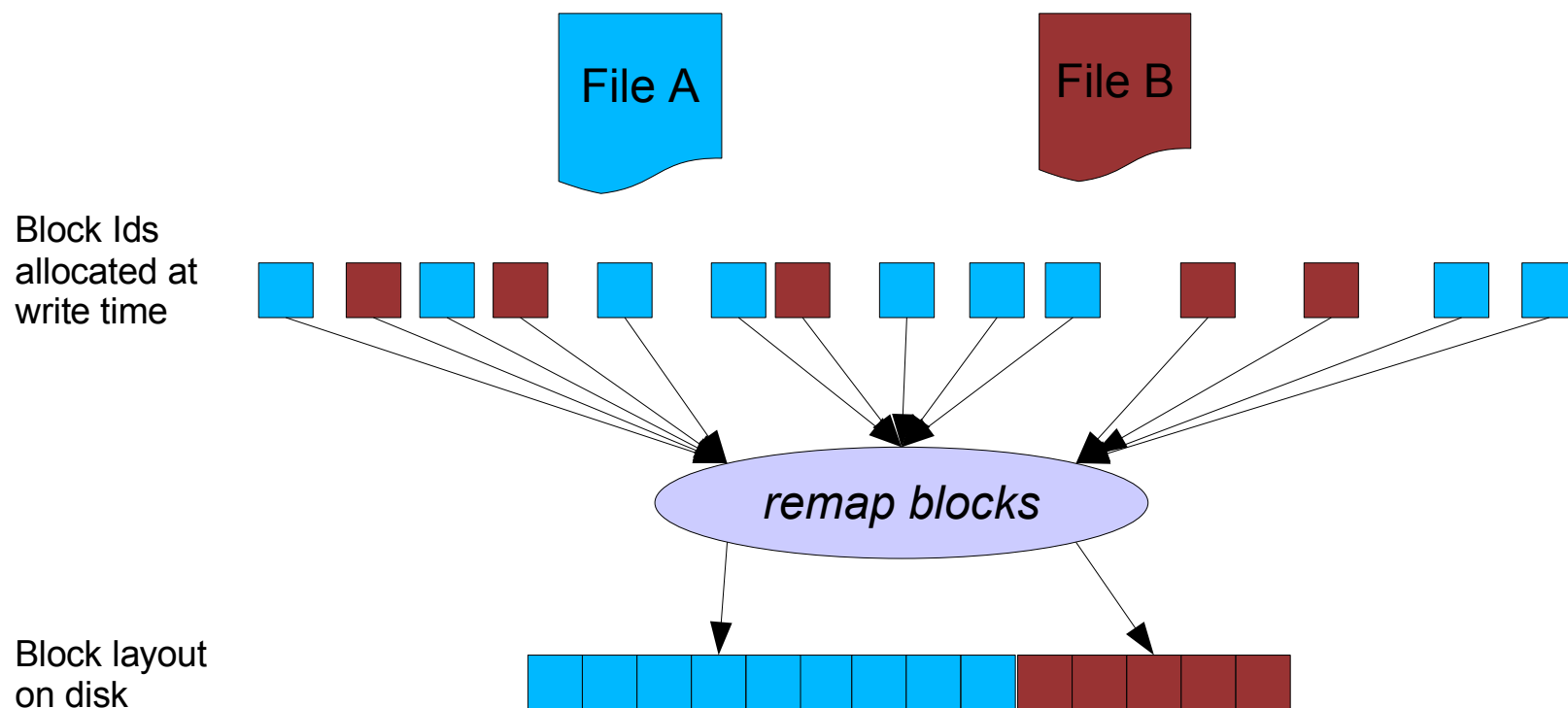
Why can't we maintain the free list in some sorted order?

- Problem: Interleaved writes to multiple files may end up causing each file to be discontinuous.

Block remapping

Idea: Delay determination of block address until cache is flushed

- Hope that multiple block writes will accumulate in the cache
- Can *remap* the block addresses for each file's writes to a contiguous set
 - *This is kind of a hack, introduced “underneath” the FFS block allocation layer.*
 - *Meant fewer changes to the rest of the FFS code.*
 - *Sometimes building real systems means making these kinds of tradeoffs!*



Colocating inodes and directories

Problem: Reading small files is slow. Why?

- What happens when you try to read all files in a directory (e.g., “ls -l” or “grep foo *”) ?

Colocating inodes and directories

Problem: Reading small files is slow. Why?

- What happens when you try to read all files in a directory (e.g., “ls -l” or “grep foo *”) ?
- Must first read directory.
- Then read inode for each file.
- Then read data pointed to by inode.

Solution: Embed the inodes in the directory itself!

- Recall: Directory just a set of <name, inode #> values
- Why not stuff inode contents in the directory file itself?
 - *What filesystem feature do we possibly give up when doing this?*

Problem #2: Must still seek to read *contents* of each file in the directory.

- Solution: Pack all files in a directory in a contiguous set of blocks.

Synchronous metadata updates

Problem: Some updates to metadata require synchronous writes

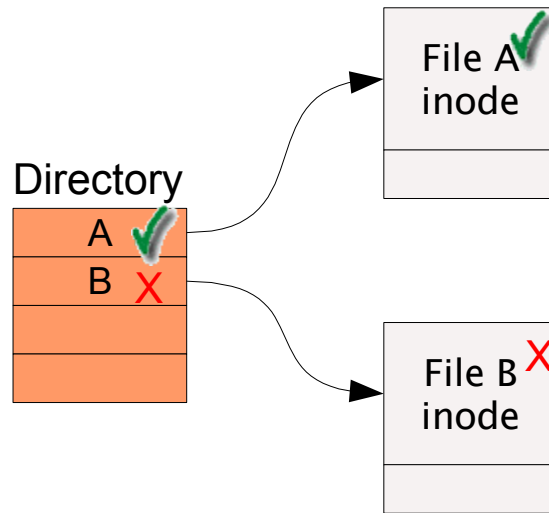
- Means the data has to “hit the disk” before anything else can be done.

Example #1: Creating a file

- Must write the new file's inode to disk before the corresponding directory entry.
 - *Why???*

Example #2: Deleting a file

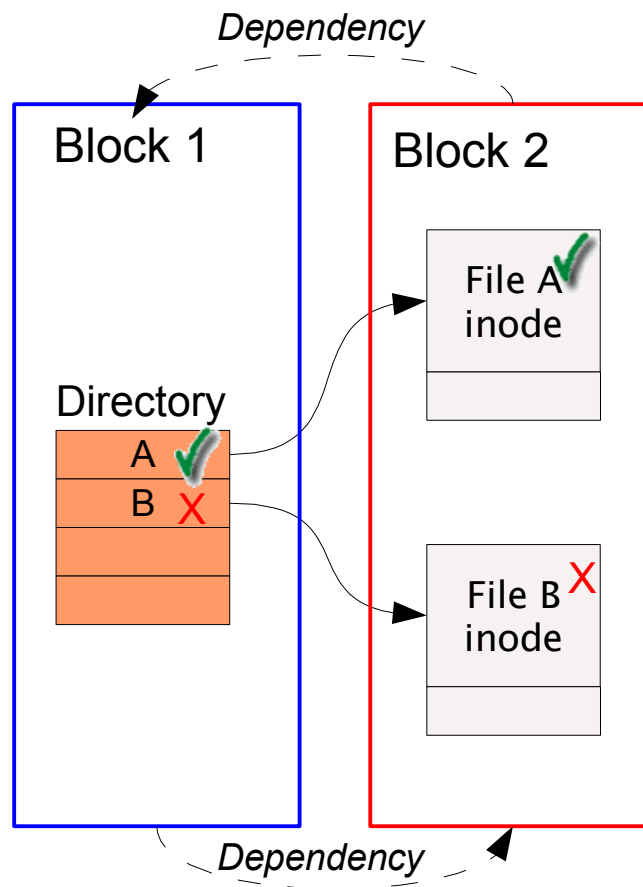
- Must clear out the directory entry before marking the inode as “free”
 - *Why???*



Synchronous metadata updates

Say that ...

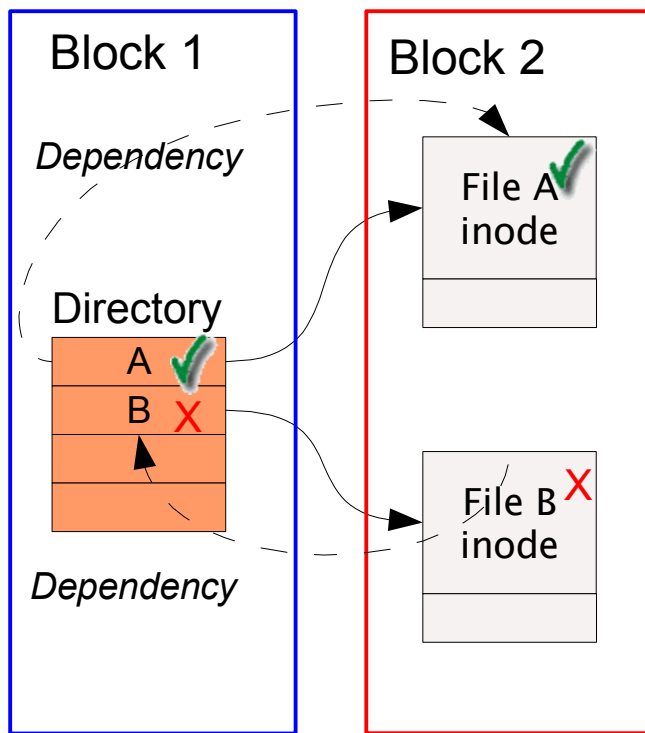
- 1) Both inodes are in the same disk block.
- 2) Both the file create and file delete have happened in the cache, but neither has hit the disk yet.
- Given this, what order are we allowed to write the disk blocks out?
 - *We have a cyclic dependency here!!! Arggghhh*



Soft Updates

Idea: Keep track of dependencies on a finer granularity

- Rather than at a block level, do this at a “data structure level”
- Example: Track dependencies on individual inodes or directory entries.



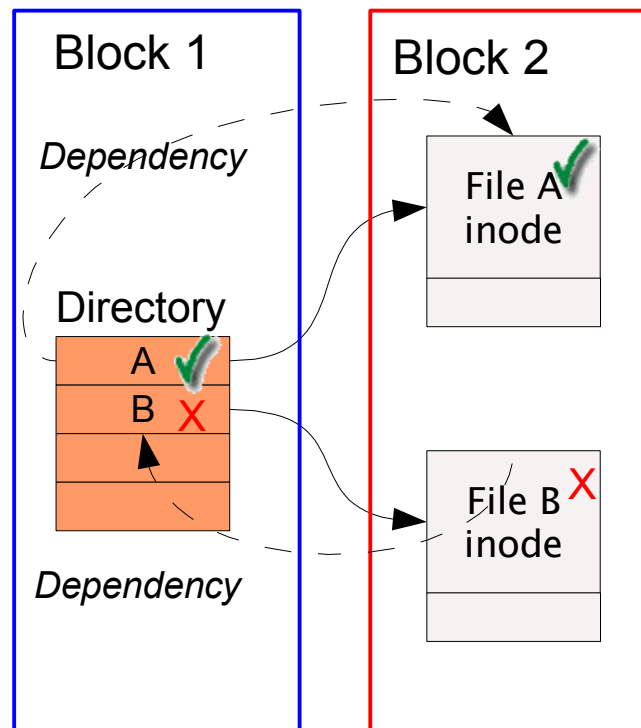
Soft Updates - Example

How to break the cyclic dependency?

- “Roll back” one of the changes before writing the data out to disk!

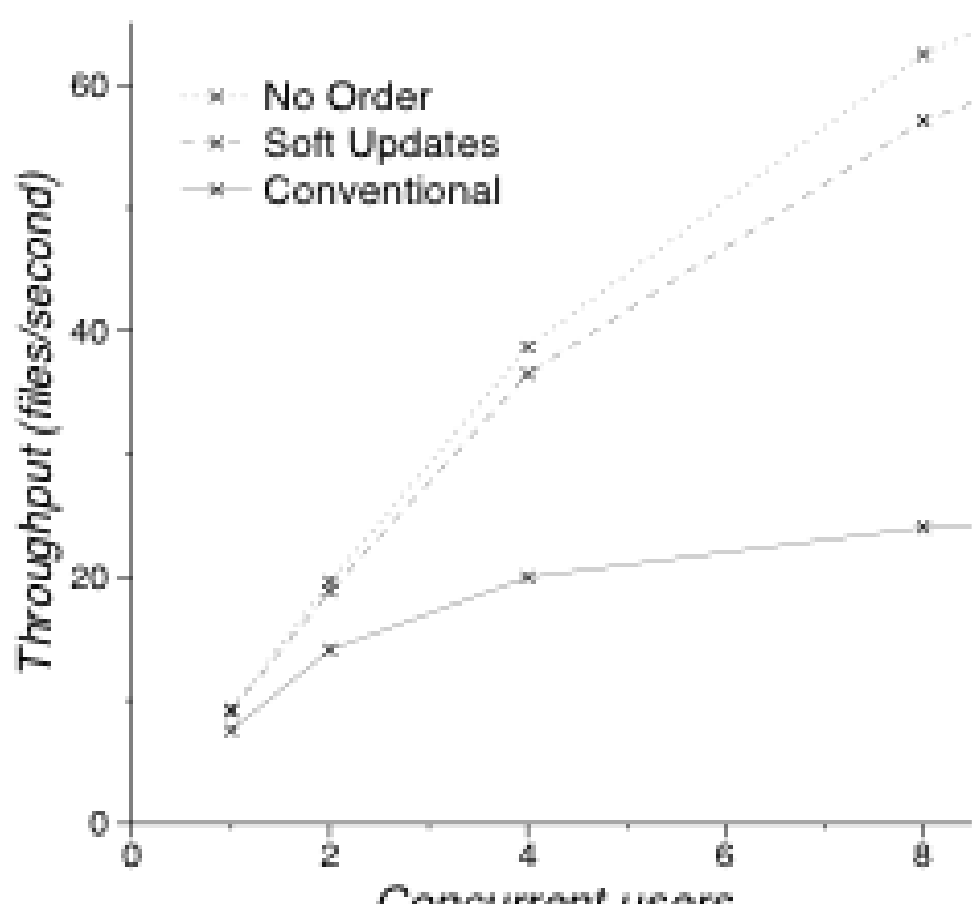
When flushing inode block (Block 2) to disk...

- **Undo** the file delete operation (as if it never happened!)
- Write out the inode block (Block 2) – *still contains B!*
- Then write out the directory block (Block 1) – *still contains entry for B!*
- Then **redo** the file delete operation ... can now proceed.

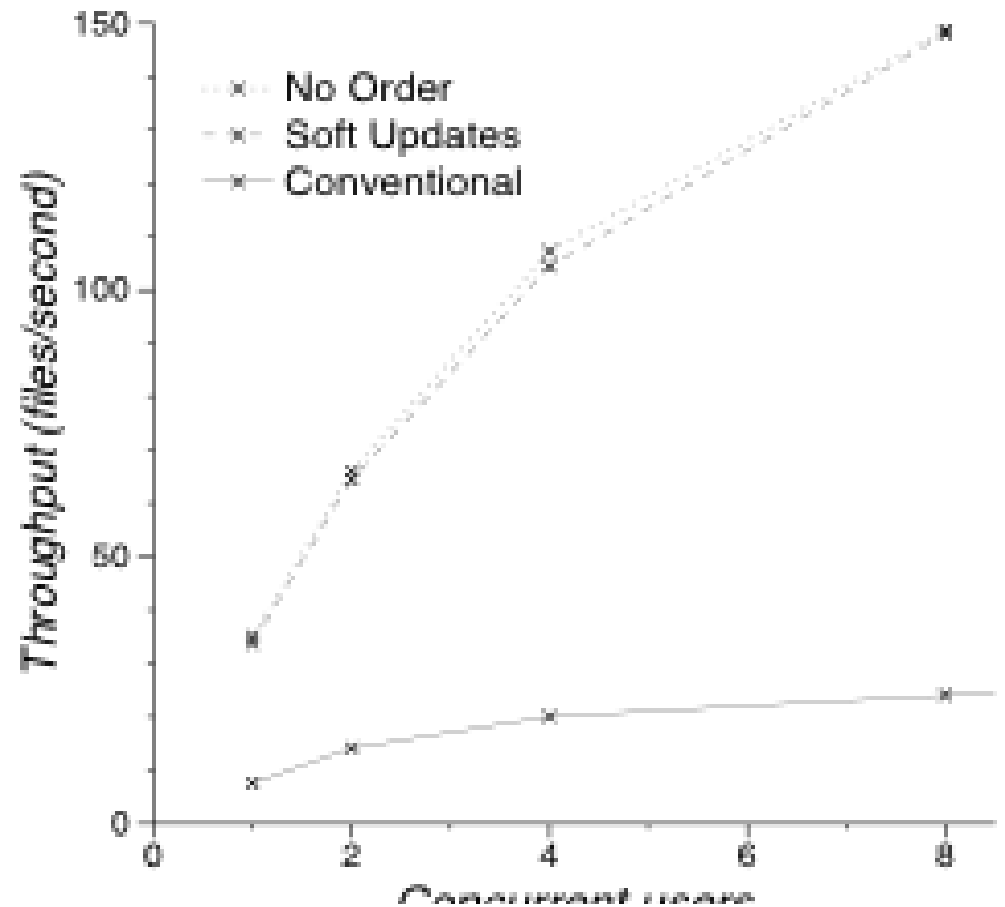


Soft Updates - Performance

Close to performance of “unordered” metadata updates:



1 KB file creates



1 KB file removes