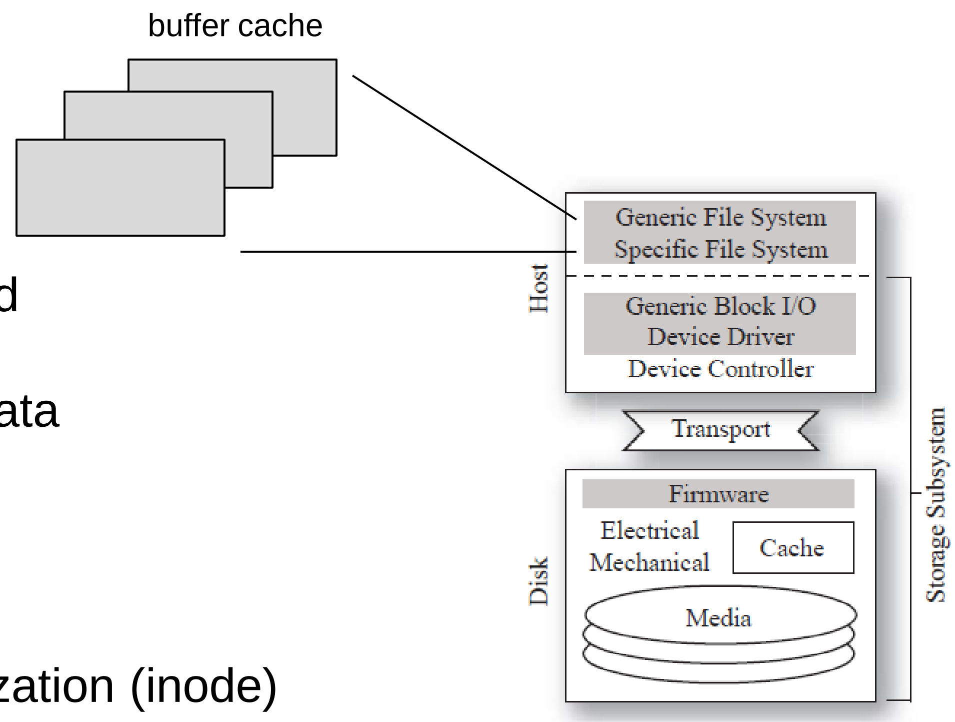
File Systems

Structure of a File System



buffer cache

- improves efficiency
  - delayed writes, read ahead
  - improved scheduling
- contains both data and metadata

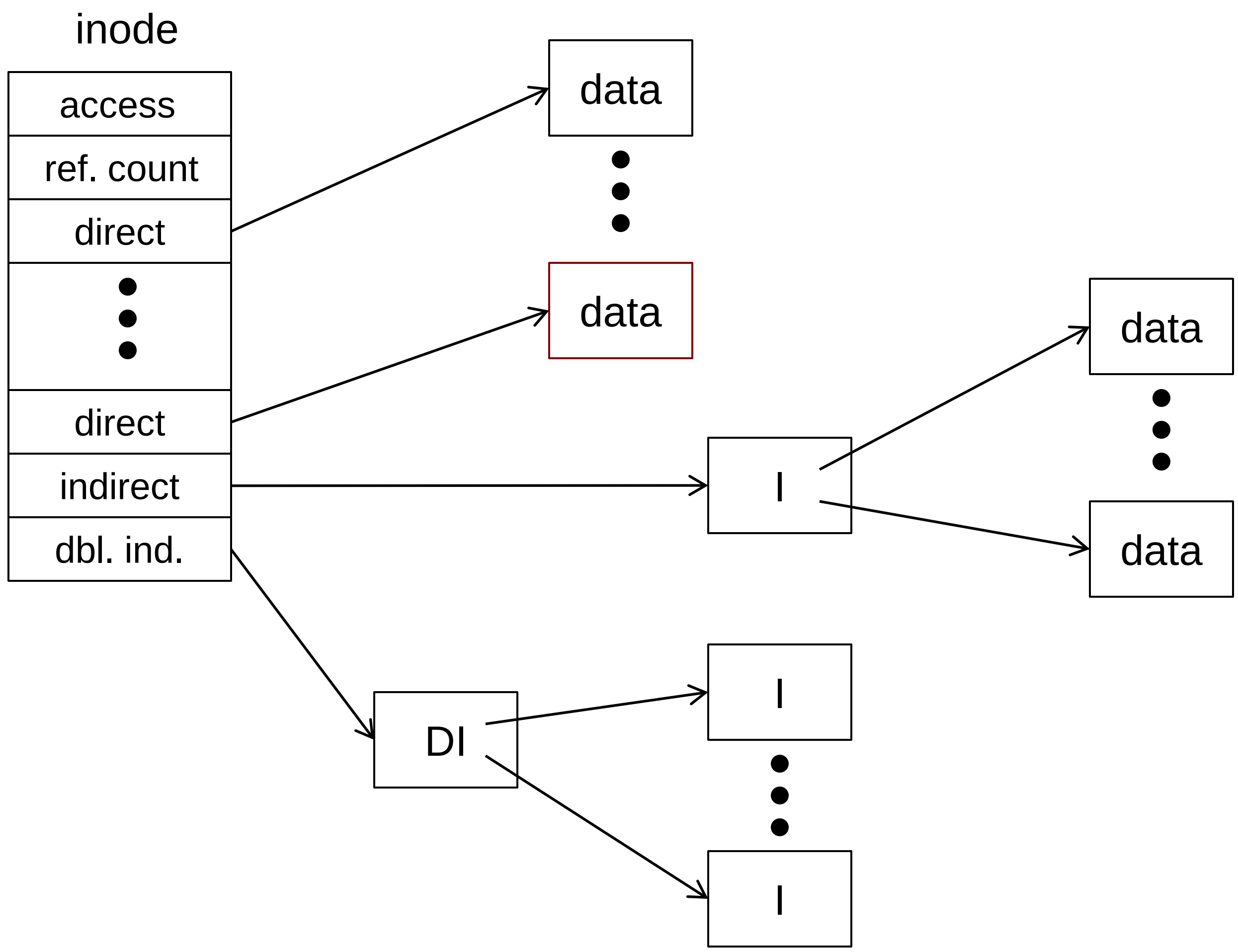
metadata

- file organization (inode)
- naming (directories)
- management (bitmaps)

data

- application defined

File Metadata



Directory Metadata

(root)

name	*inode*
usr	97

(97)

name	*inode*
staff	27

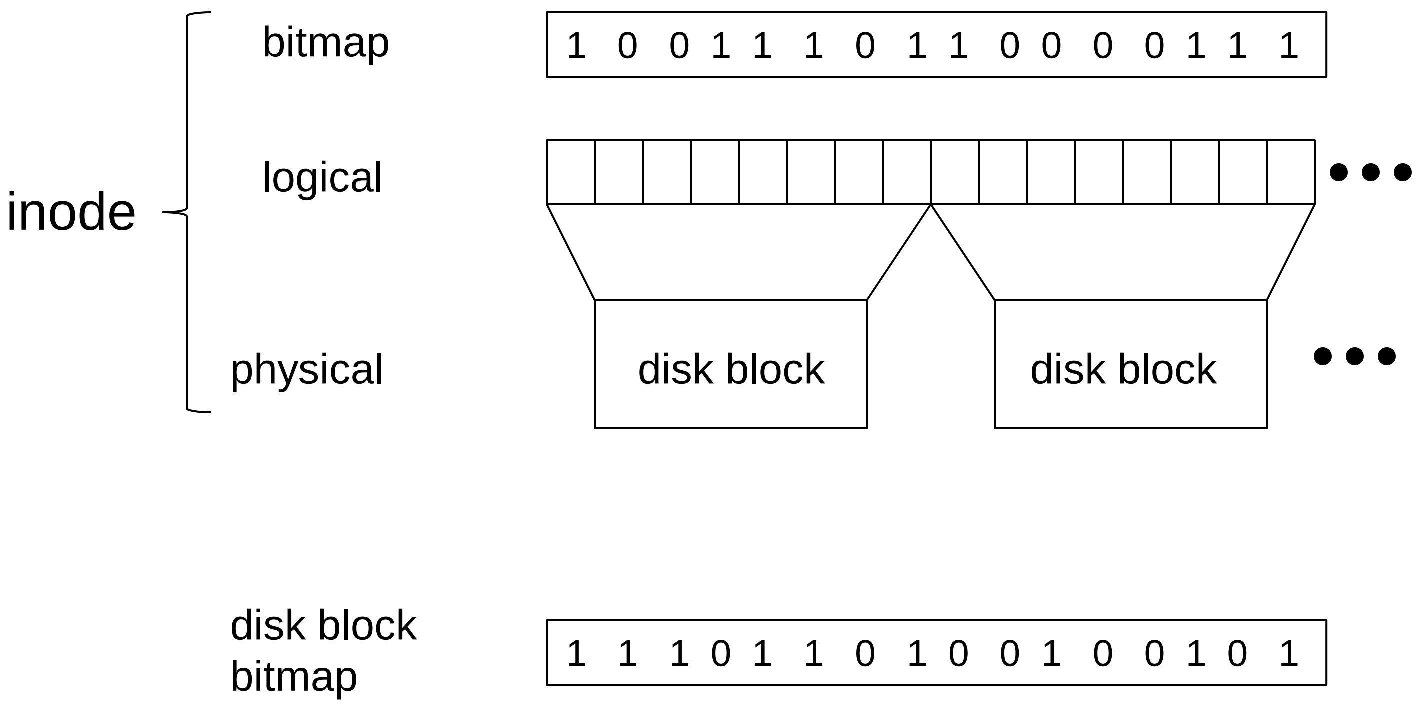
(27)

name	*inode*
mgr	152

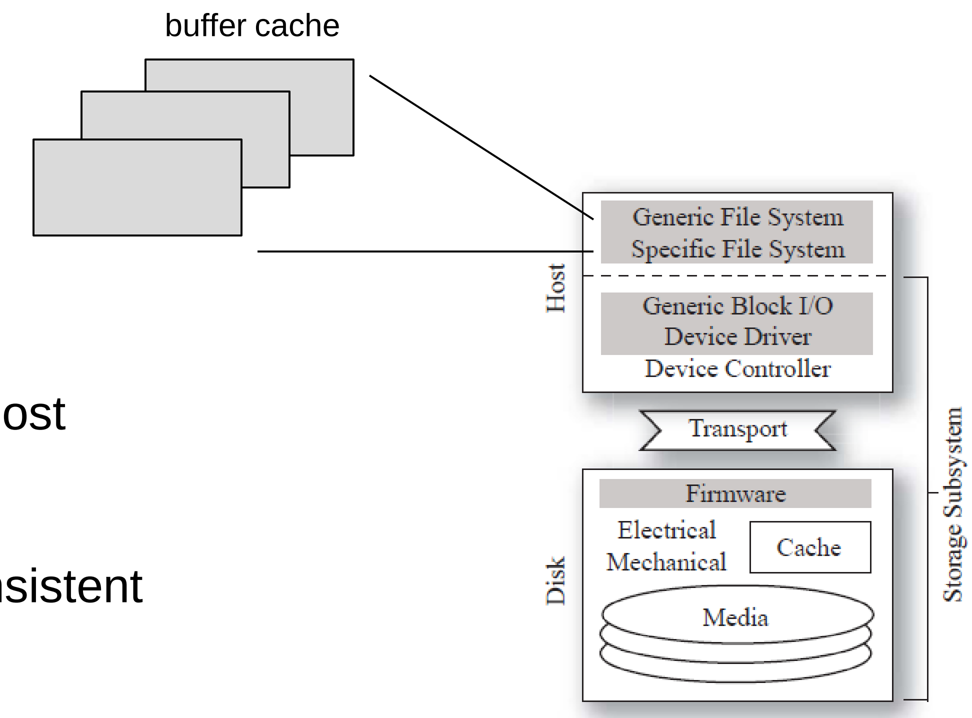
- directory
  - file of directory entries
  - root directory at a known location
- directory entry
  - name component
  - inode of sub-directory file
- example

  /usr/staff/mgr

Management Metadata



Failure Modes



host (system) failure

- cached data and metadata lost
- disk contents
    - stable (survives)
    - metadata may be inconsistent

disk (media) failure

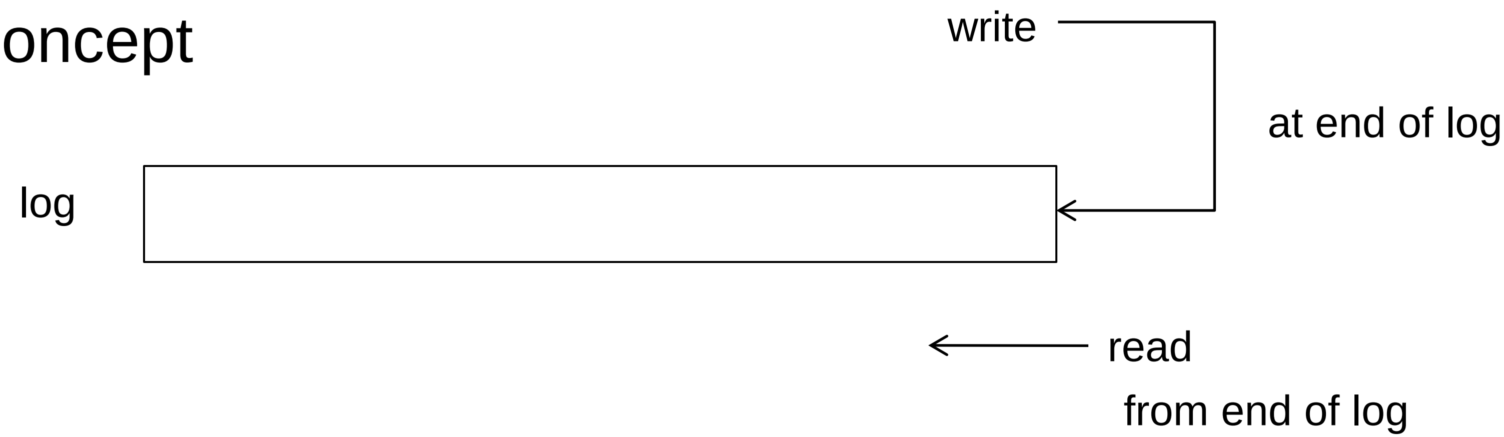
•potential corruption of arbitrary data/metadata

Goals & Approaches

- Improving performance
  - Creating a different structure
    - Log-structured file systems
    - Google file system
- Improving resilience to crashes
  - Changing structure to reduce/eliminate consistency problems
    - Log-structured file system
    - Google file system
  - Maintaining consistency on disk
    - Journaling (a logging technique)
    - Soft updates (enforcing update dependencies)

Log-structured file system

Concept



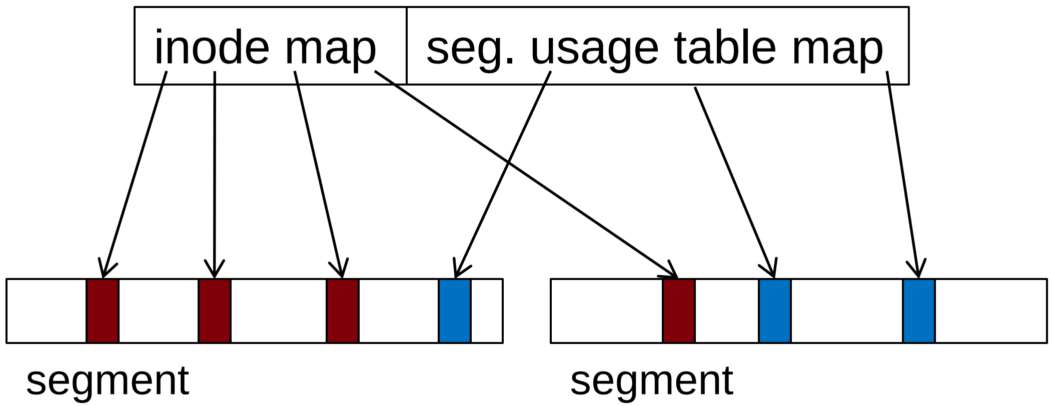
more recently written block renders obsolete a version of that block written earlier.

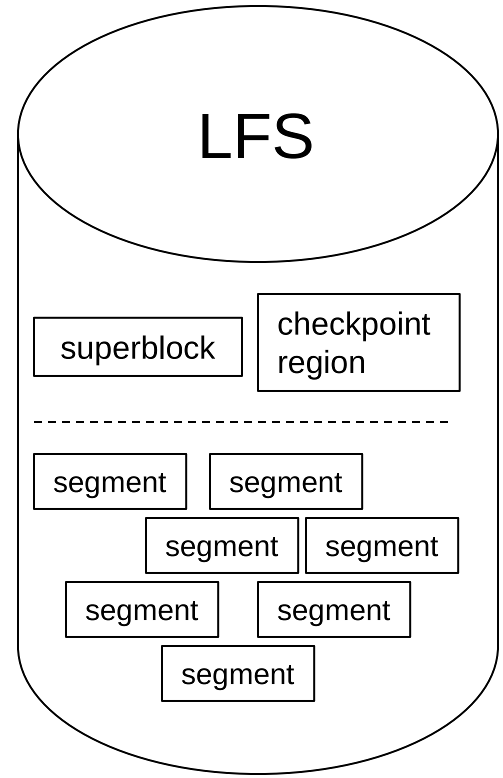
Issue	Approach
How to structure data/metadata	segments
How to manage disk space	segment cleaning

LFS structure

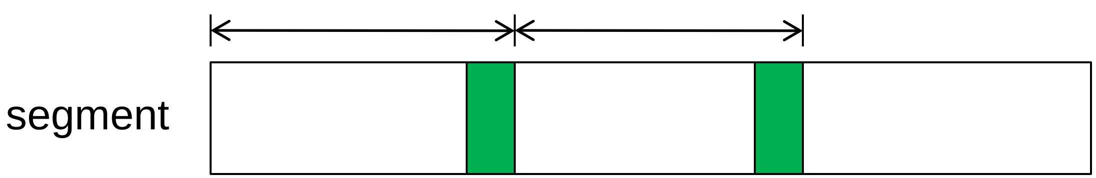
Superblock - list: (segment, size)

Checkpoint region:





inode map – list: (inode location, version#)
segment usage table – list: (live bytes, modified time)



Segment summary block – list: (inode, version, block)

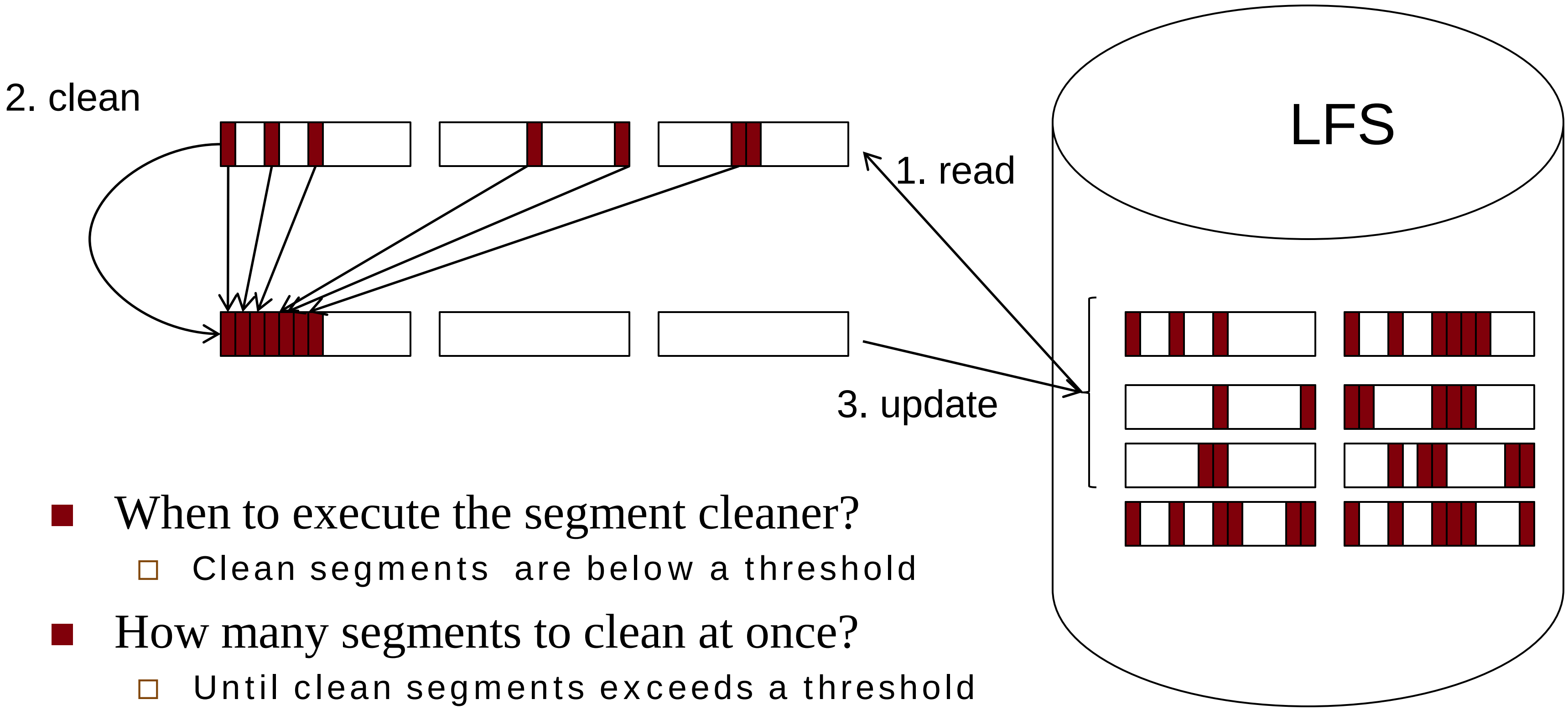
Checkpoint

- Creation
  - Flush to disk
    - data
    - I-nodes
    - I-node map blocks
    - segment usage table
  - In fixed checkpoint region, write addresses of I-node map blocks and segment usage table blocks.
  - Mark checkpoint region with "current" timestamp.
  - Use two checkpoints for reliability

Recovery

- Read latest checkpoint
- Roll-forward
  - Scan segment usage blocks
    - New inodes are incorporated into inode map (data blocks automatically included)
    - Data blocks for new versions are ignored
  - Adjust segment utilizations in segment table map
  - Insure consistency of directories and inodes
    - Directory operations log
    - Records entry for each directory change
    - Written to log before directory/inode blocks

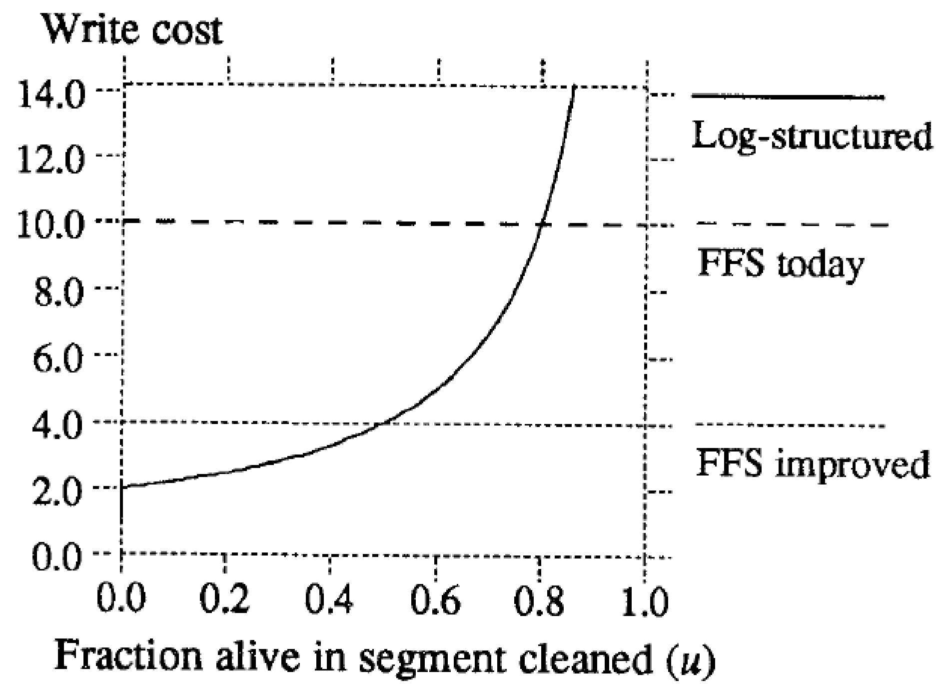
Segment cleaning



- When to execute the segment cleaner?
  - Clean segments are below a threshold
- How many segments to clean at once?
  - Until clean segments exceeds a threshold
- Which segments to clean?
- How should live blocks be grouped?

Cleaning Policies

$$\text{write cost} = \frac{\text{total bytes read and written}}{\text{new data written}}$$
$$= \frac{\text{read segs} + \text{write live} + \text{write new}}{\text{new data written}}$$
$$= \frac{N + N*u + N*(1-u)}{N*(1-u)} = \frac{2}{1-u}$$

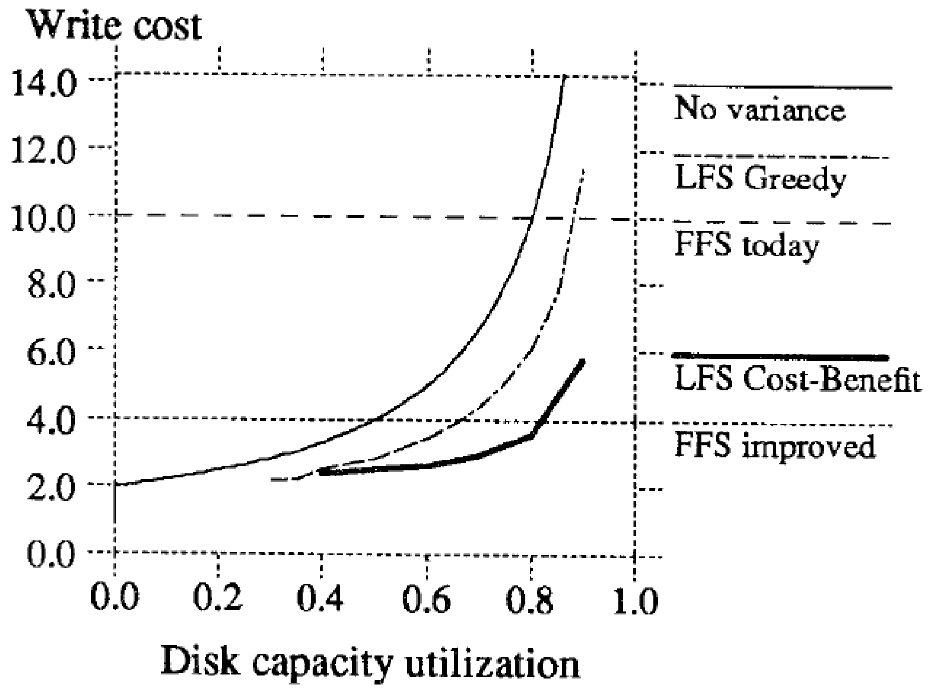


“The key to achieving high performance at low cost in a log-structured file system is to force the disk into a bimodal segment distribution where most of the segments are nearly full, a few are empty or nearly empty, and the cleaner can almost always work with the empty segments.” (Rosenblum/Ousterhout)

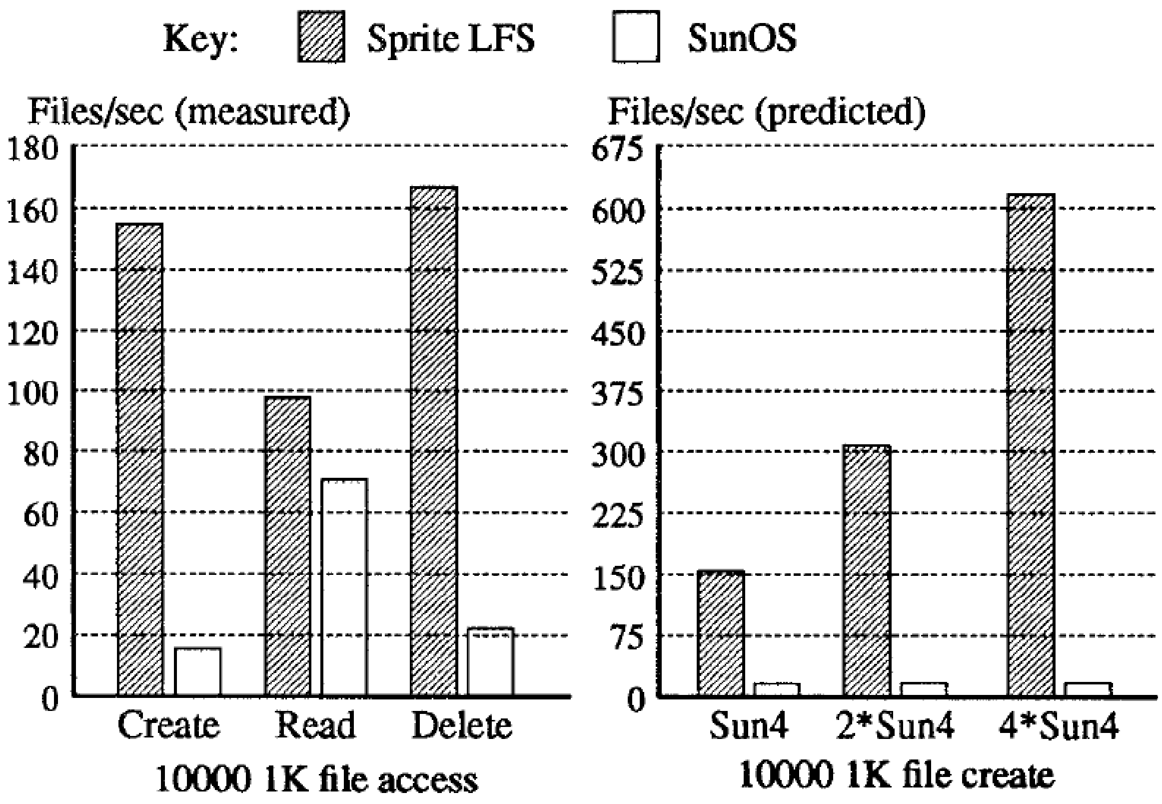
Cost benefit policy

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

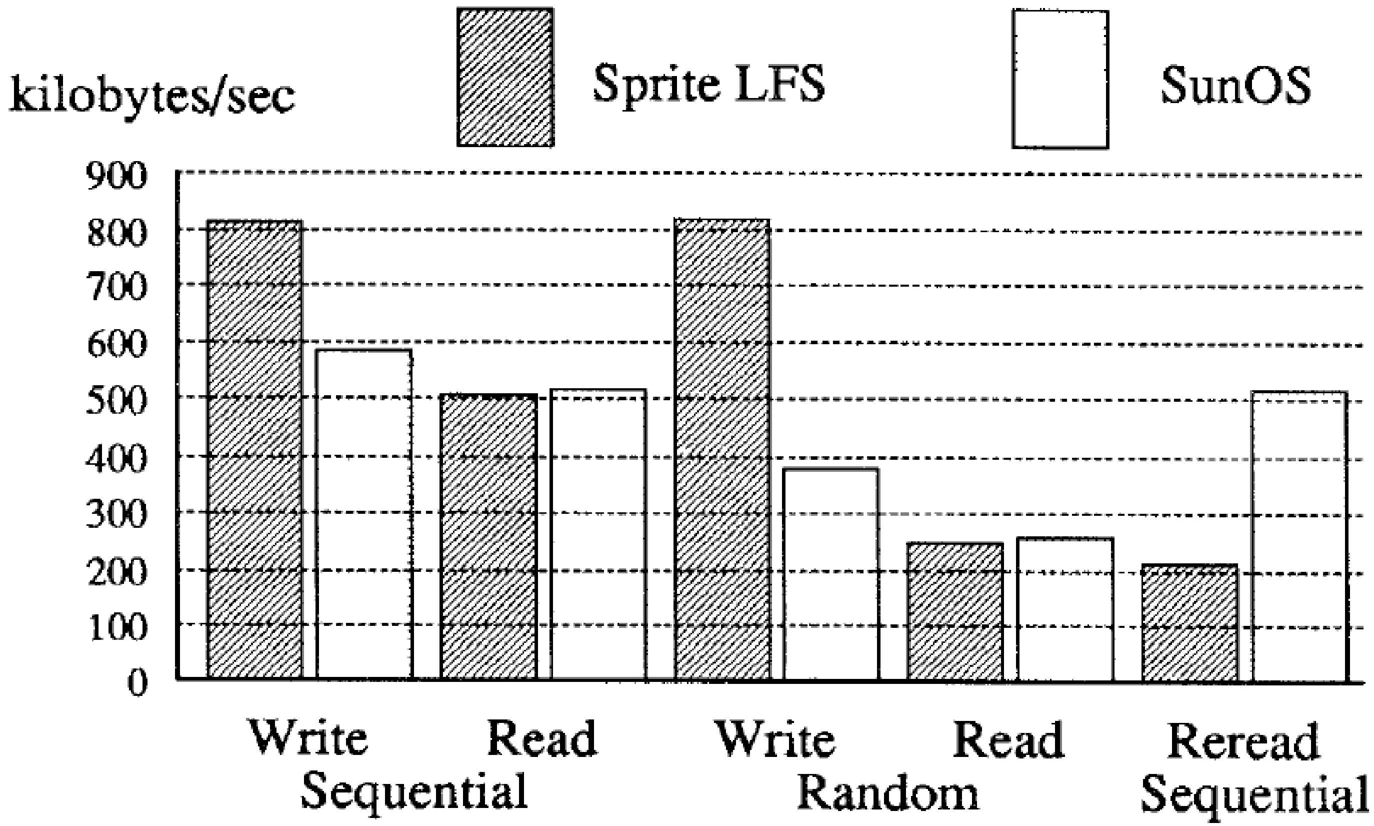
- Select for cleaning the segment with the highest ratio of benefit to cost
- Use age to approximate the stability of the data in a segment



LFS Performace



LFS Performance



LFS Overhead

Sprite LFS recovery time in seconds			
File Size	File Data Recovered		
	1 MB	10 MB	50 MB
1 KB	1	21	132
10 KB	< 1	3	17
100 KB	< 1	1	8

Recovery time is dominated by the number of files.

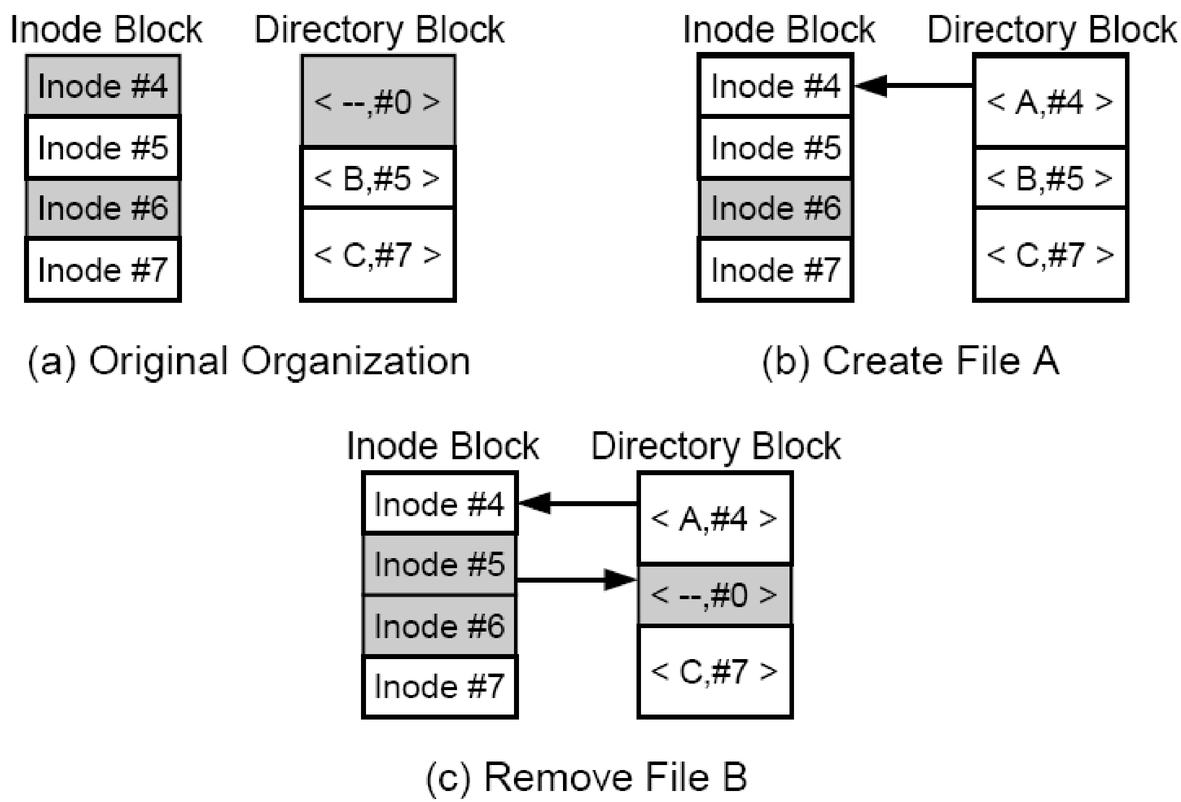
Resources are highly focused on user data.

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%

Soft Update Concept

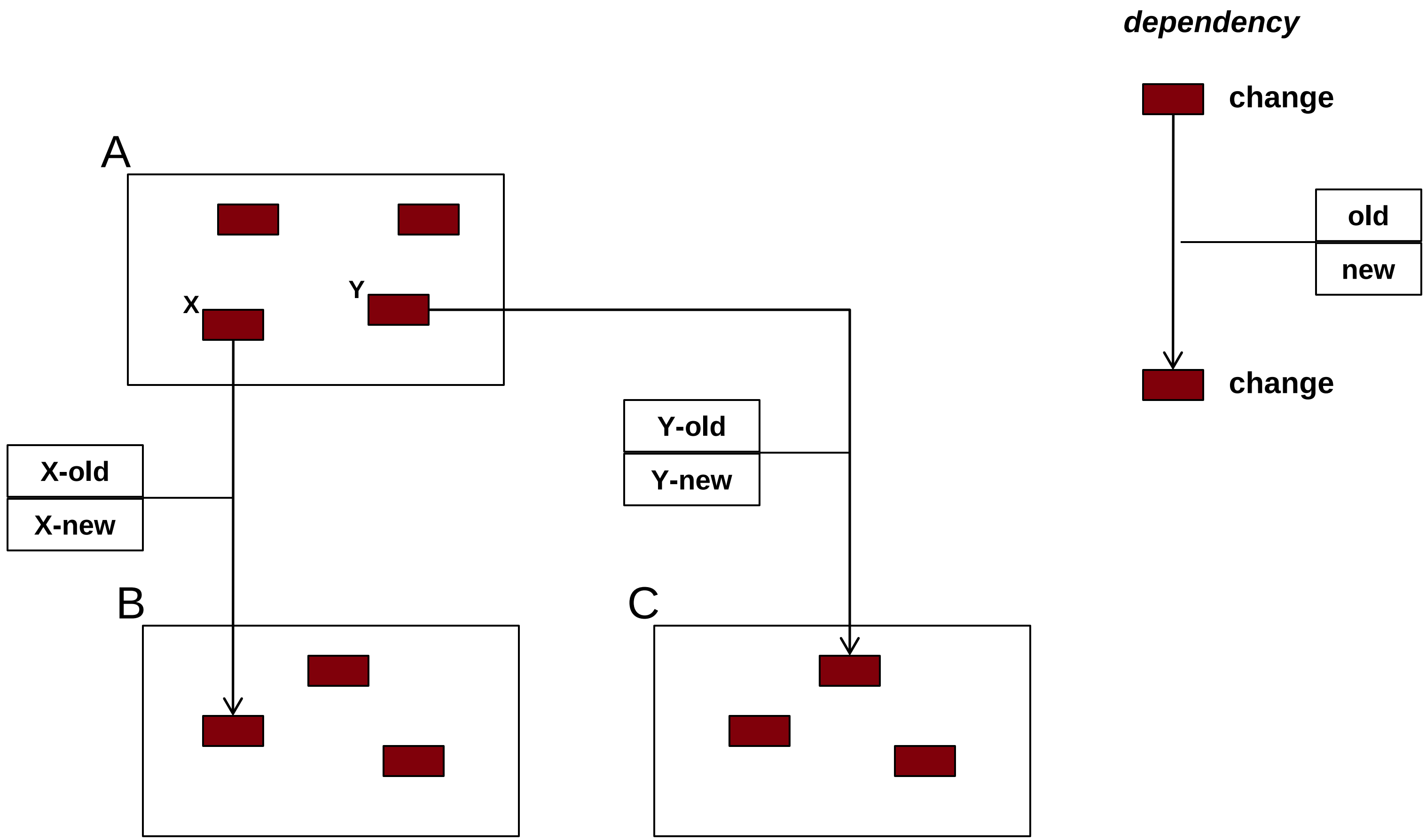
- Idea: maintain dependencies among in-cache metadata blocks so that writes to disk will preserve the consistency of the on-disk metadata.
- Ensures that the only metadata inconsistencies are unclaimed blocks or inodes that can be reclaimed by a background process examining the active file system
- Reduces by 40% to 70% the number of disk writes in file intensive environments

Metadata Dependencies



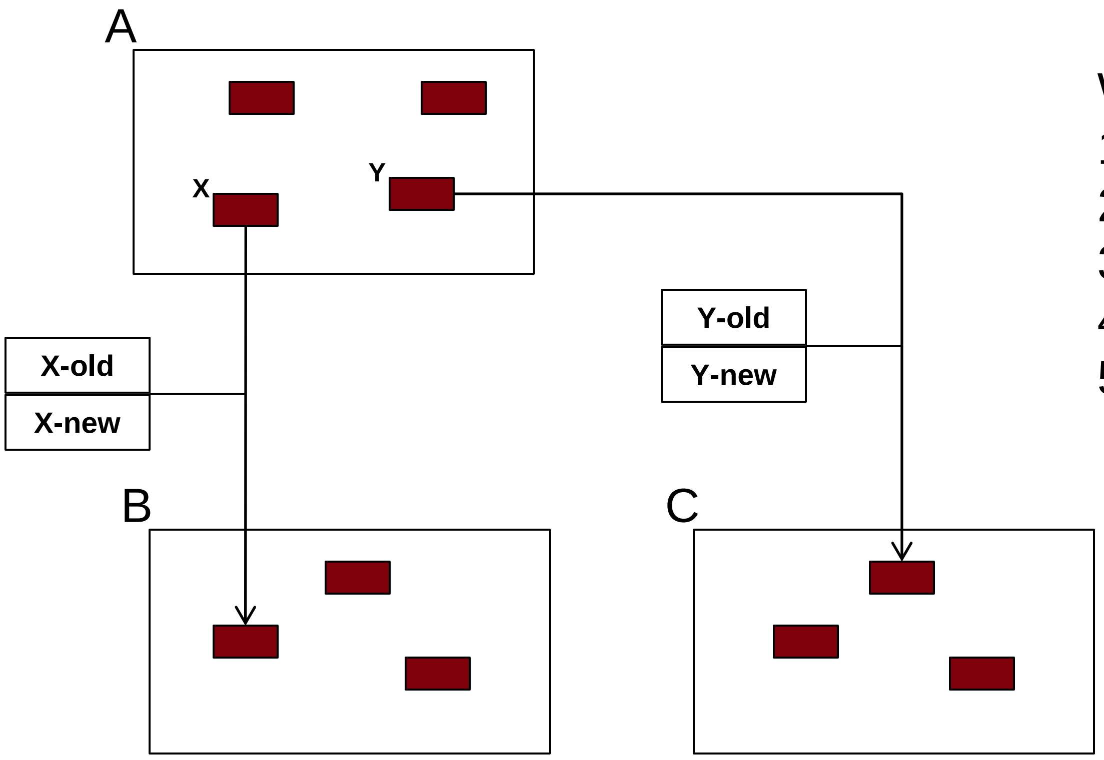
- File operations create dependencies between related metadata changes
- Cyclic dependencies can arise between metadata blocks

Soft Updates Example



- Maintaining old/new values allows undo-redo operations
- Cyclic dependencies can arise between metadata blocks

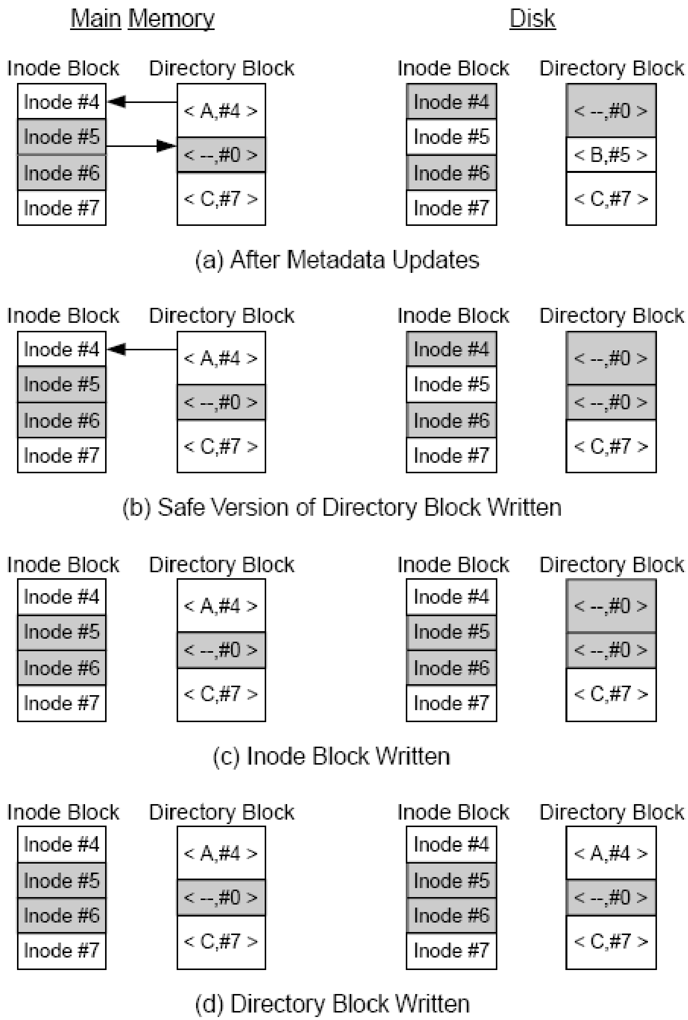
Soft Updates Example



Write block A:
1. Rollback X in A using X-old
2. Rollback Y in A using Y-old
3. Write A to disk
4. Restore X in A using X-new
5. Restore Y in A using Y-new

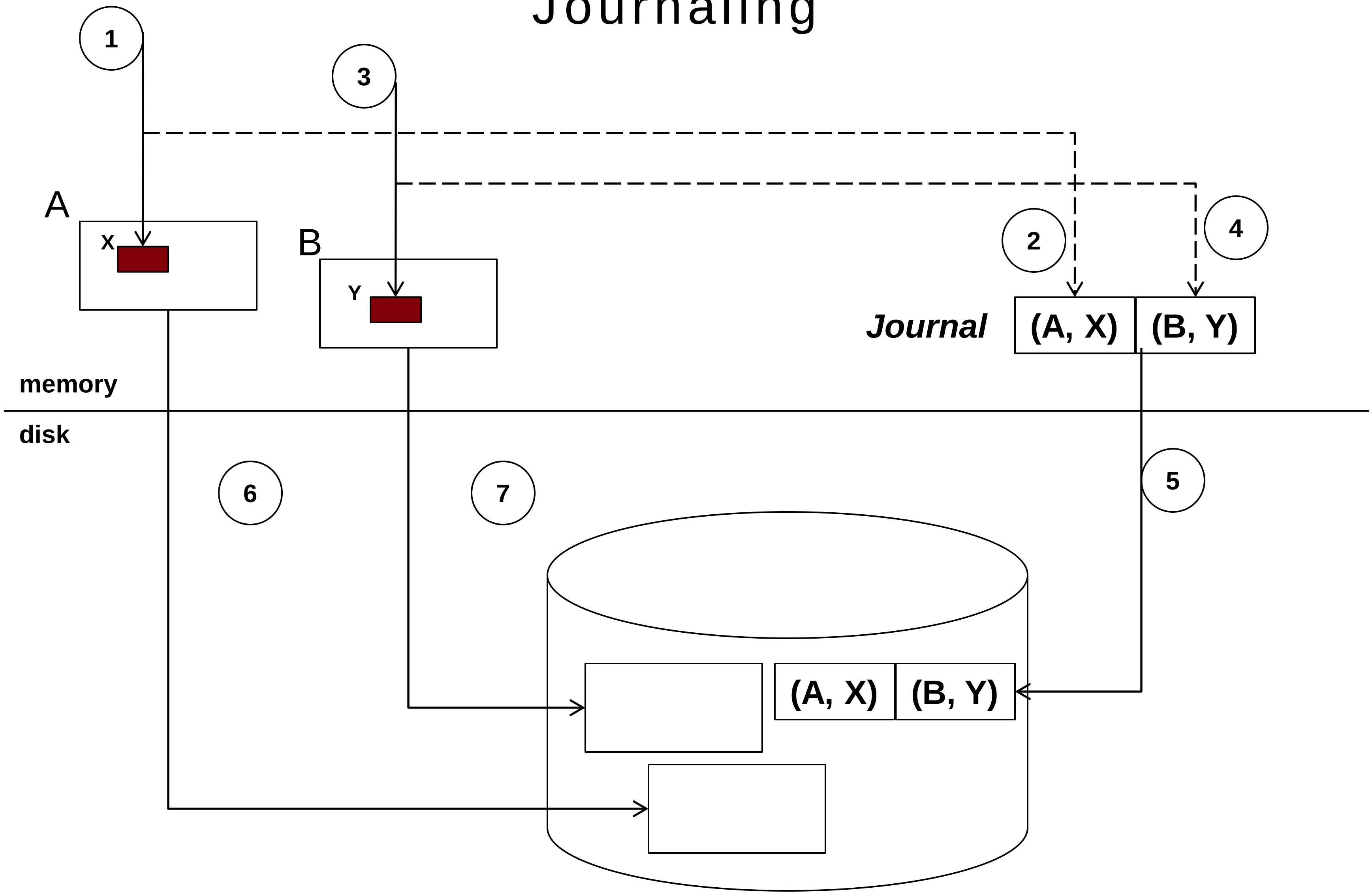
Write block B:
1. Write B to disk
2. Remove dependency from X in A

Example



- A metadata block may be written more than once to insure consistency

Journaling



Journaling

- Process:
  - record changes to cached metadata blocks in journal
  - periodically write the journal to disk
  - on-disk journal records changes in metadata blocks that have not yet themselves been written to disk
- Recovery:
  - apply to disk changes recorded in on-disk journal
  - resume use of file system
- On-disk journal
  - maintained on same file system as metadata
  - stored on separate, stand-alone file system

Journaling Transaction Structure

- A journal transaction
  - consists of all metadata updates related to a single operation
  - transaction order must obey constraints implied by operations
  - the memory journal is a single, merged transaction
- Examples
  - Creating a file
    - creating a directory entry (modifying a directory block),
    - allocating an inode (modifying the inode bitmap),
    - initializing the inode (modifying an inode block)
  - Writing to a file
    - updating the file's write timestamp (modifying an inode block)
    - may also cause changes to inode mapping information and block bitmap if new data blocks are allocated

Journaling in Linux (ext2fs)

- Close the (merged) transaction
- Start flushing the transaction to disk
  - Full metadata block is written to journal
  - Descriptor blocks are written that give the home disk location for each metadata block
- Wait for all outstanding filesystem operations in this transaction to complete
- Wait for all outstanding transaction updates to be completely
- Update the journal header blocks to record the new head/tail
- When all metadata blocks have been written to their home disk location, write a new set of journal header blocks to free the journal space occupied by the (now completed) transaction

Configurations & Features

	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

Benchmark study

	Unpack	Config	Build	Total
	Absolute Time (in seconds)			
FFS-async	1.02	10.38	42.60	53.99
	Performance Relative to FFS-async			
FFS	0.14	0.66	0.85	0.73
Soft-Updates	0.99	0.98	1.01	1.01
LFFS-file	0.72	1.08	0.95	0.96
LFFS-wafs-1sync	0.15	1.01	0.88	0.82
LFFS-wafs-1async	0.90	0.94	1.00	0.99
LFFS-wafs-2sync	0.20	0.85	0.93	0.86
LFFS-wafs-2async	0.90	1.05	0.98	0.99

***SSH Benchma*rk**