

Lecture 4

File System Integrity

2227, Spring 2006 1

Plan for today

- What FS integrity is all about
 - internal consistency, especially in the face of problems
- Importance and Challenges
- Tools
 - “atomicity” of writes
 - update ordering
 - real atomicity
- Applying the tools
 - examples

2227, Spring 2006 2

Software Interface Layers

Program		
File System (Database)	<File Name, Offset>	
Device Driver	<Partition, Block #>	
I/O Controller	<Disk #, Sector #>	
Disk Media	<Cylinder, Track, Sector>	

Each layer uses **metadata** to translate higher-level names to lower-level names

2227, Spring 2006 3

Metadata contains lots of pointers

```

graph TD
    Inode3[Inode #3] --> B20[Block #20]
    Inode3 --> B42[Block #42]
    Inode3 --> B44[Block #44]
    Inode5[Inode #5] --> B42
    Inode5 --> B44
    B20 --> B20
    B20 --> B2[Block #2]
    B20 --> B8[Block #8]
  
```

2227, Spring 2006 4

...and other convoluted inter-relationships

- Counts
 - e.g., inodes usually contain a “link count” (# names for file)
- Refinements
 - e.g., file length refines the block list
- Auxiliary information
 - e.g., last modification time for a file
- Aggregates
 - e.g., maximum or average values in a file
- Redundant listings
 - e.g., an extra hash index for fast directory lookups
- Note: all have consistency issues
 ... though not of equal importance

2227, Spring 2006

5

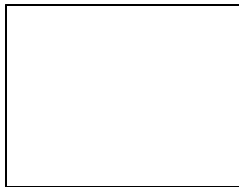
Importance and Problems

- Why is metadata integrity so important?
- What challenges are faced?

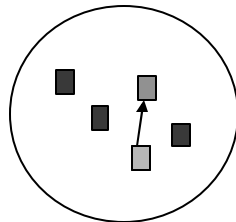
2227, Spring 2006

6

Volatile main memory and caching



Cache (in main memory)



Disk contents

- Notes:
 - Why not just write everything to disk?
 - Why not just make memory non-volatile?

2227, Spring 2006

7

Specific examples

- Among fields of an inode or directory entry
 - problem: what if the structure is only partially written?
- File creation
 - problem: new inode must be initialized before directory entry
- Rename
 - problem: new directory entry added and old entry removed
 - not necessarily both in same directory
- Account balances when doing money transfer
 - problem: need to ensure that sum of balances stays same

2227, Spring 2006

8

Tools for protecting internal consistency

- static mappings
 - if they don't change, they don't cause problems
 - most people don't think of this one most of the time...
- "atomicity" of writes
 - ala the tri-state postwrite guarantee of per-sector ECC
- update ordering
 - simply ensuring that one update propagates before another
- real atomicity
 - ensuring that a set of updates all occur or none do

2227, Spring 2006 9

"Atomicity" of writes as a tool

- Unwritten guarantee provided by per-sector ECC
 - because the ECC check will fail if only partially written
- Same trick can be used by FS or applications
- Good for grouping inter-related updates
 - but increases likelihood of data loss due to the third state
 - data is lost when write is only partially completed
 - while uncommon, such loss is more likely than a grown defect
 - especially if not physically co-located
 - as a result, this mechanism is used for limited cases
 - e.g., internal consistency of directory chunks and inodes

2227, Spring 2006 10

Update ordering as a tool

- Just what it sounds like...
- Good for single-direction dependencies
 - just do one before the other
- Problem: doesn't work for bidirectional dependencies
 - which, unfortunately, is most of them
- Solution: some can be converted to single-direction
 - because some directions are more important than others ;)
 - clean-up must be done after system failures

2227, Spring 2006 11

Basic Update Ordering Rules

- Purpose: integrity of metadata pointers
 - in face of unpredictable system failures
 - similar to rules of programming with pointers
- Resource Allocation
 - initialize resource before setting pointer
- Resource De-allocation
 - nullify previous pointer before reuse
- Resource "Movement"
 - set new pointer before nullifying old one
- Notice that something always left dangling
 - ... assuming a badly-timed crash

2227, Spring 2006 12

FS crash recovery given update ordering

- Need to deal with all of the dangling stuff
 - worse: need to find it all first
- Traditional recovery examines entire contents
 - walk entire directory hierarchy and each file's block list
 - identify unclaimed resources and incorrect counts
 - rebuild free space/inode bitmaps
- Post-crash time to mount
 - Traditional: 5 to 7 minutes (several years ago)
 - grows with FS size and #files

2227, Spring 2006

13

Implementing update ordering

- Synchronous writes
 - wait for one write to complete before proceeding
 - this was once the common technique in FSs
 - but, the performance overhead can be dramatic
 - Note: what could be done about it?
- Soft updates
 - use write-back caching for all (non-fsync) updates
 - make sure updates propagate to disk in the correct order
 - works great, but only goes as far as update ordering can go

2227, Spring 2006

14

Real multi-write atomicity as a tool

- Ensure that multiple updates are all reflected or not
 - by a combination of extra writes and post-failure clean-up
- Good at all of the internal consistency issues
 - may be over-kill (in space+code) for simple requirements
 - as characterizes many FS issues
 - but, this is the standard mechanism in databases
 - and also common in many modern file systems
- Two main approaches
 - write-ahead logging
 - shadow paging

2227, Spring 2006

15

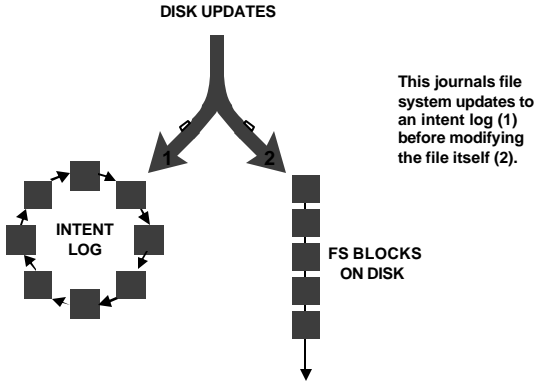
Write-ahead logging (or “journaling”)

- Pre-post intended changes to log (a.k.a. journal)
 - make sure that it is written before corresponding blocks
 - during crash recovery, can roll forward intended changes
 - recovery time proportional to uncheckpointed log size
 - log space reclaimed via periodic “checkpointing”
- Key concept: recording changes to alternate location first so that a safe version is always present
 - can even deal with third state of ECC thing
- Write-ahead logging can be nearly zero overhead
 - requires delayed log/grouped writes and enforced ordering

2227, Spring 2006

16

Updates go to log first, then to real destination



Shadow Paging

- New versions of data blocks written to new locations
 - never overwrite live data in place; instead remap identity
 - during crash recovery, non-remapped writes discarded
 - recovery time can be immediate or like write-ahead logging
 - deprecated versions of data blocks must be garbage collected
- Key concept: write changes to alternate location and replace previous so that safe version is always present
 - Note: shadow paging can support transactions too...

Shadow Paging illustrated

