CS 111 Fall 2013
Lecture 13: File System Robustness

By Justin Bass and Synthia Ling

File System Goals

Robustness: System will still work even if unusual data patterns are input
Can decompose into subgoals:

- Durability- survives hardware failures e.g. lose power, but not arbitrary failures like a computer exploding. Limited failures that we expect to survive: disk space full, during power loss should not lose data
- Atomicity- changes are either done or not done, not partially done (could help during power loss)
- Performance- thoughtput and latency maintained under heavy loads

What can go wrong with hardware?

- Lose power
  - lose contents of RAM, cache (CPU, controller), registers
  - data on disk is retained
- Run out of disk space
- Heavily loaded system
- Sudden disconnection (of a hard drive)
- Bad sectors/tracks (a partial failure)
- Lose disk (disk crash)
  - If there are 100 disks and 1 fails, we don't need to lose any data or even crash

Solutions:

- Backup power gives enough time to save RAM/cache- assume we're not doing this
- Suppose you're writing to disk - you still get a few microseconds after power is removed
  - CPU executes a few instructions
  - disk controller moves arms to a 'parking area' so data won't be incorrectly written somewhere else
- We'd like to assume sector writes are atomic, BUT they're not always

In Understanding the Robustness of SSDs under Power Fault by Zheng et al. at FAST '13

- Out of 15 SSDs, only 2 worked when they cut the power
- The 13 that failed to work suffered from
  - bit corruption- bad bits in same or adjacent sectors
  - flying writes- sectors got written perfectly but at wrong location, possibly due to wear-levelling that did not have time to store where the data was written
  - shorn writes (partially written sectors)
  - metadata corruption (where disk info is stored; this is serious because it affects the entire disk)
  - dead device (no read/writes at all upon reboot, metadata too scrambled: all data is lost, disk requires reformatting)
  - unserializability
    - Sequentially requests A,B,C,D,E
    - Writes A,C,E, but not B or D

Summary: Even though SSDs can do more operations after a power failure, they have more problems because their high performance involves parallel writes, wear-levelling, etc.

Loss of Power Model

Suppose we're trying to implement ^X^S in Emacs

```
write(fd, buf, sectorsize)
```

Assume no OS for now (no buffering, no dallying, etc.). If we lose power, we can lose data.

Assuming we have two devices, we must follow the...

> GOLDEN RULE OF ATOMICITY
>
> Never overwrite your only copy

Attempt 1: Have TWO copies of data

A old
B new
write(A)
write(B)
0: old version
1: new version

	1	2	3	4	5
A	0	?	1	1	1
B	0	0	0	?	1

Method: if they disagree, choose A
Problems: bloated data by 2x, STILL does not work for column 2 as A is undetermined, so choosing A will give a garbage value

Attempt 2: Have THREE copies of data

	1	2	3	4	5	6	7
A	0	?	1	1	1	1	1
B	0	0	0	?	1	1	1
C	0	0	0	0	0	?	1

Method: choose the majority of the three, and for a disagreement (column 4) choose A, which tends to be written sooner
Solves

- bit correction
- shorn writes

But not

- flying writes
- dead device
- unserialization
- metadata corruption

Problem: This solution is terrible for real-life purposes, and is rarely implemented.

This problem is only useful for helping us to learn how to model/formulate problems
Idealized model: writes are not atomic and they eventually finish

Lampson Sturgis Assumptions

- Storage writes may fail, or corrupt another piece of storage
- A later read can detect the bad sector
- Storage may spontaneously decay (e.g. bits may flip across bus)
- Errors are rare
- Repairs can be done in time (before another failure occurs)

Eggert's Assumptions (overly simple and optimistic)

- block writes are atomic

```
assert(lseek(fd, 0, SEEKSET) % 8192 == 0)
write(fd, buf, 8192) //assume block is allocated
```

Assume a BSD-style file system: superblock->bitmap->inodes->data
Thus the data is stored in the data blocks, and the timestamp is stored in the inodes
This could cause problems for time-sensitive programs (e.g. source control) if a power failure occurs when only one has written
now we do $ls -l file

Option 1: If we write a timestamp before we write data, the data could be wrong (a null change)
Option 2: If we write data before we write a timestamp, the timestamp could be wrong (a hidden change)

Option 1 is preferable, because we should err on the side of knowing our past actions rather than having actions that are not logged anywhere, but moving arm between timestamp (in inode) & data blocks requires a lot of time

Common optimization: do writes, daily timestamp
Problem: Breaks 'make' or other programs that depend on timestamp accuracy
Solution: Cleanup procedure after reboot

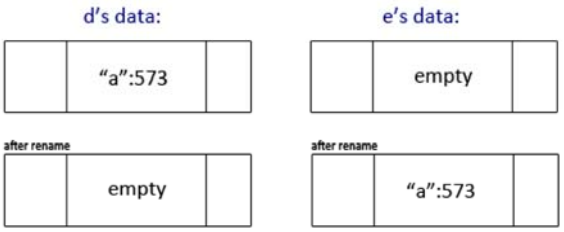
```
rename ("d/a", "d/b")
```

find d's inode
find d's data

```
rename ("d/a", "d/bb")
```

can't set hard block allocation

```
rename ("d/a", "e/b")
```



If we write before handling link count and power is lost, we will have the correct data but linkcount will still be 1
Then if linkcount--, the inode will be freed even though data still exists.

Thus write/handle linkcount in this order:
increment link count
write e's data
write d's data
decrement link count
Link count never underestimates the value, but can overestimate.

Models of Underlying Systems

Invariants for our system

1. Every block of the file system is used for exactly one purpose
2. All referenced blocks are initialized to data appropriate for their type
3. All referenced blocks are marked used in bitmap
4. All unreferenced blocks are marked free in bitmap

Consequences of violating rules

1. Disaster: If blocks have 2 purposes, you could think a block is free when it is not -> data loss
2. Disaster
3. Disaster
4. Storage leak

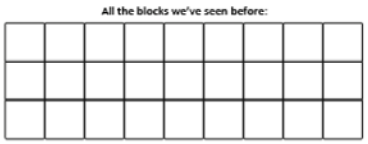
Number 4 has the least consequence. It is not critical during certain system calls, so it is not really an invariant

Performance Improvement Ideas

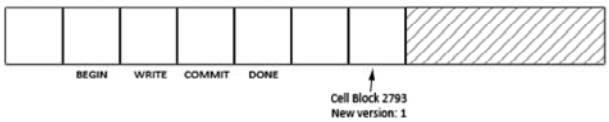
1. Commit Record
   - Write "data" outputs into a preliminary area, e.g. a temp file
   - Wait for all blocks to hit the disk (reordering is OK)
   - Write a commit record saying done
   - Copy temporary data back to original
   - Write "done" to a commit record

```
pattern (appears to be atomic at higher level)
     BEGIN
       precommit phase- work is being done but you can't back out
     COMMIT (atomic at lower level)
       postcommit phase- work, can't back out usually for performance reasons
     END
```

2. Divide disk into 2 parts:
   Cell Storage



   Journal



   Sequence of instruction listed in temporal order
   Grab unused space, description of change, do change
   Circularly linked: when journal reaches end, wrap to beginning (beginning entries should be obsolete by this point)
   Optimized for writes
   Reads are slow, but a commonly used subset could be cached in RAM
   Logging Protocols:
   - Write ahead log:
     - log planned writes
     - implement them

- Write behind logs:
  - log old data about to be rewritten
  - write new data into cell
  - mark as done
- When replaying logs after crash
  - write ahead log
    replayed L to R
  - write behind log
    replayed R to L