# CS 111

## Lecture 13 Scribe Notes (Fall 2013)

*by David Kang*

## File System Robustness

### Goals

We can decompose FS robustness into subgoals:

Durability - Survives (limited) hardware failures. E.g. can survive power loss, but cannot survive computer melting.

Atomicity - Changes are done or not done.

Performance - Throughput and latency.

### What can go wrong with hardware?

- lose power
- run out of disk space
- heavily loaded system
- sudden disconnect
- bad sectors/tracks
- lose disk (disk crashes)

We focus on the "lose power" aspect since it is a simple problem. We need to deal with losing contents of RAM, CPU/controller caches, and registers. We only keep what was written to the disk.

Suppose you're writing to a disk, you only get a few microseconds to:
have the CPU execute a few instructions and
make the disk controller park its head.
We'd like to assume that sectors writes are atomic for the sake of this example.

### Exceptions to atomicity

However, it is important to remember that sector writes are not always atomic. For example, according to Zheng et al. FAST '13 "Understanding the robustness of SSD's under power fault," out of 15 SSD's, only 2 worked after power loss during operation. The other 13 suffered from:

- bit corruption - random bit errors
- flying writes - bits get written to the wrong location when the SSD is powered back on due to the controller's wear leveling becoming confused upon power up
- shorn writes - write operations are only partially done

- metadata corruption - metadata in FTL is corrupted
- dead device - devices is dead or mostly non-functional
- unserializability - some requests were written, but not in an understandable order

## Implementing save in emacs

Suppose we're trying to implement ^X^S (save) in emacs.

```
write(fd, buf, sector size)
```

We don't have an operating system for now. If we lose power here, we can lose data. To try circumvent this issue, suppose we have 2 devices.

**Golden Rule of Atomicity**: Never overwrite your only copy.

We have two blocks, A and B. We write A first with B's data, then overwrite B's data.

old to new picture

One problem with this is that we bloat data by 2x. The second, more significant problem is that if we lose power right before we overwrite B's data, we do not know whether A or B's data is correct:

Don't know which data is right.

The solution to this is to do three writes. Then if we have a power loss, we take the 2/3 majority that are the same. If A, B, C are all different, we take A as the correct bit.

Doing three writes solves bit corruption. Writes are not atomic, but they eventually finish since we know which bit is correct.

Don't know which data is right.

## Lampson-Sturg's assumptions

- storage writes may fail or corrupt another piece of storage
- a late read can detect the bad sector
- storage may spontaneously decay
- errors are rare
- repairs can be done in time (before another failure occurs

## Eggert Assumption (overly rosy)

- block writes are atomic

We use Eggert's Assumption to simply things for us.

## An example

```
if (lseek(fd, 0, SEEKSET) % 8192 == 0)
        write(fd, buf, 8192);  // assume already allocated
        // assume BSD style file system
```

Picture of drive layout.

We update the time stamp first, then the data. This will cause 'make' to do extra work, but it will still succeed.

A common optimization is to do the write, dally, and then timestamp the file. This, however, has the issue of breaking 'make' since it depends on updated timestamps. The solution to this is to use a cleanup procedure following a reboot, i.e. 'make clean'.

## Another example

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

a being crossed out to b

For the first operation, we find d's inode, then we find d's data and change the "a" to "b".

The second operation can be more difficult because it requires dealing with block allocation, as the file names are of different sizes.

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

showing directory change.

A problem that arises here is that when we write first, the link count is 1, although there are now two links pointing to the inode.

To fix this:

1. increment link count
2. write e's data
3. write d's data
4. decrement link count

This way, the link count never under underestimates the true value. It can, however, overestimate, but this is better. This way we may lose disk space but cannot lose data.

## Invariants

It is important to have a model for invariants for your file system. An invariant is a condition that should always be true. Otherwise there can be serious consequences, like data loss.

Example of invariants for a BSD-style file system include:

1. Every block of f.s. is used for exactly one purpose. Otherwise, there is "disaster" (data loss).
2. All referenced blocks are initialized to data appropriate for that type. Otherwise, disaster.
3. All referenced blocks are marked used in the bitmap. Otherwise, disaster.
4. All unreferenced blocks are mareked free in the bitmap. Otherwise, there can be storage loss.

The fourth invariant is the least important and not really an invariant since its consequence isn't too serious. Other invariants exist as well and vary for different file systems.

## Performance improvement ideas

### 1) Commit Record

1. write "data" outputs into a preliminary area, like a temp file
2. wait for all blocks to hit disk (reordering is okay)
3. copy temporary data back to the original, wait for it to hit the disk
4. write "done" to the commit record

```
BEGIN
        pre-commit phase        // work is being done, but you can back out
COMMIT or ABORT                 // atomic at lower level
        post-commit phase       // work is being done, but cannot back out
END
```

This process appears to be atomic to the user.

### 2) Journal

Showing what journal looks like.

It is possible to have only a journal and no cell blocks. This allows very fast writes and slow reads. Some cell storage can be cached to RAM

When the journal runs out, it can start over at the beginning of the jornal. This requires garbage collecting the journal as only some of the early parts of the journal can be used.

ext4 is an example of a journaled file system.

## Logging protocols

### Writeahead logs

- log planned writes
- implement them later

### Writebehind logs

- log old data about to be written
- write new data into cell
- mark as done

When replaying logs after a crash, for writeahead logs replay left to right. For writebehind logs, replay right to left.

# File System Robustness

# CS111 Lecture 13

Prepared by Sean Zarringhalam

## Robustness Defined

Still works even if you throw unusual data, usage patterns, and/or a lot of data at it.
You can try to decompose file system robustness into subgoals. We want to meet all these goals even when something goes wrong.

1. **Durability :** A file system is durable if it survives failures in the hardware. We are not talking about arbitrary hardware failures (ie: computer melts). We are talking about limited hardware failures. Some examples include space filling up, pulling the plug, or any other ordinary hardware failure that causes loss of data.
2. **Atomicity :** It is important for a file system's changes to be atomic. Changes are either done, or not done. If you override your file to include an updated statistic (ie: "Bill Gates currently has 50 billion dollars"), you do not want it to be scrambled. You should only either have the old version, or new version.
3. **Performance :** We want the file system to perform well, even when given big requests (ie: a file to use up all the disk space available). This boils down to throughput and latency. When somebody makes a request, we want that request to return.

## What can go wrong in the hardware?

- We can lose power.
- We can run out of disk spcae.
- We could have a heavily loaded system.
- We could have a sudden disconnection. The hard drive and computer may still have power, and we don't want to lose data.
- Partial failure could occur (bad sectors/tracks). This means that a failure occurs on part of the disk, but not the whole disk. In this case, we may have to settle to lose a tiny bit of data.
- The disk could be lost. If you have a computer with a hundred disks, you should be able to recover. However, if you only have one disk, then that's all she wrote.

These are the sort of things one must consider when building robust systems (This is why designers charge so much!).

## Loss of Power Scenario

### This lecture will primarily focus on loss of power, because it is the simplest problem to consider.

In effect, when you lose power, you lose the contents of RAM, cache, registers, file descriptors, and data structures (ie: process table). *But*, you don't want to lose data. We can lose all this, but we get to keep what is on disk (or if it's flash memory, what's in flash).

One common approach is that we assume we have backup power. In those five minutes of backup power, we can carefully save everything. *But*, let's assume that we are not doing this. For example, a system without backup power describes Professor Eggert's desktop in his office. He has no battery backup becase he wanted to save money. Even if we assume no backup power, there is still an issue we have to address. We still haven't precisely said what happens when we lose power.

Suppose you are the CPU, and you're writing to the disk. Then, you lose power. What happens is there will be enough reserve electrons in the power supply because of the built-in capacitance in the power supply. You will get a few microseconds of power. When this situation arises, disk controllers will normally lift the disk heads, and move them to a "parking" area. That way, when the disk stops spinning, the heads are not over something that would cause them to crash.

The CPU can also execute a few instructions during this time. If everything is engineered correctly, you can actually get a bit of work done before the power goes out. Ideally, what we would like to say is "if we were in the middle of writing a sector before the power goes out, the sector is only a few bytes. The amount of time it takes to finish writing the sector is so small that we can rely on reserve power." We would like to assume that, from our point of view, sector writes are **atomic**. That is, when we issue a write sector, and we lose power at that time, the sector either does not start, or has finished.

**There is still a bug here!** Sector writes are not *always* atomic.

## Zheng's Paper

Take for instance a paper by Zheng et al. entitled FAST '13: Understanding the robustness of SSD's under Power Fault (https://www.usenix.org/system/files/conference/fast13/fast13-final80.pdf). In the paper, they investigated 15 SSD's, all different devices commercially available, and all respected vendors. They discovered that out of these, only two of them worked when power was cut. The other devices suffered from the following errors:

- **Bit corruption:** That is, you cut power while writing to the flash drive, turn power back on, and find that the sector being written had some bad bits in it.
- **Flying writes:** When you write to a sector and cut the power in the middle, you find that the sector got written perfectly to the wrong location.
- **Shorn writes:** This happens when only part of the sector has been written, but not the whole thing. Remaining bytes don't get written because we lost power.
- **Metadata corruption:** When we have corruption over the location of where everything is. This is a more serious error.
- **Dead device:** This happens when power to a device has been cut, the power is turned back on, and nothing works anymore.
- **Unserializability:** This occurs when a series of commands are issued to a device. Power is cut in the middle, and when it resumes, some of the blocks have not been written, but some blocks later had been written. For example, a request to write A, write B, write C results with A and C getting written. The order of writes does not match.

When a disk drive loses power, it's either in the middle of a write, or isn't. If it *is* writing, it finishes the sector, if it isn't writing, it stops. In ssd's, the situation is better described as three blocks written in parallel. To increase performance of SSD's, robustness can suffer.

What we need is to come up with a model that describes all the bad things that can happen.

Suppose we are trying to do something simple: implement `ctrl x ctrl s` in emacs (this saves a file). In emacs, there is a copy of the file in main memory. One way to implement this is to do the following:

```
write(fd, buf, sector size);
```

Let's assume we have a file that is one sector large. For now, let's also assume no operating system, so that when we issue a write, it is actually going to disk. If we were making this assumption, and were worried about the issues of ssd, our method is not quite right. One problem is that we could lose power in the middle of a write, in which case we would be dead on a bad ssd. We would like to think of a solution...

**GOLDEN RULE OF ATOMICITY:** If you want to get atomic behavior, you should never overwrite your only copy. If you overwrite your only copy of the data, and it gets perturbed by one of these events, you're dead.

Instead, we will have another copy (two devices A and B). A has the old version, so write the new version to B. When write(B) is finished, and the result is good, we will do a write(A). If it breaks in the middle of writing to A, then A is garbage. *But*, we saved the data in B.

One problem with this approach is that it's inefficient. We've bloated the data by a factor of 2x. We need two copies of everything. And, a problem worse than that, is that we can still lose the data. Imagine the timeline below, where 0 represents old data, and 1 represents the new data.

| **Time Index** | [0] | [1] | [2] | [3] | [4] |
|---|---|---|---|---|---|
| **Device A** | 0 | ? | 1 | 1 | 1 |
| **Device B** | 0 | 0 | 0 | ? | 1 |

Suppose we lose power in the middle of this, if we recover at [0], then it's easy. If we recover at [4], we're also fine, since A and B are the same. Anywhere else, then A and B disagree. So who's right? We have NO way of knowing. Here is a standard way to solve the problem.

Instead of having two copies of the data, have three copies of the data. A B and C. They all start with the old version...

| **Time Index** | [0] | [1] | [2] | [3] | [4] | [5] | [6] |
|---|---|---|---|---|---|---|---|

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Device A** | 0 | ? | 1 | 1 | 1 | 1 | 1 |
| **Device B** | 0 | 0 | 0 | ? | 1 | 1 | 1 |
| **Device C** | 0 | 0 | 0 | 0 | 0 | ? | 1 |

You take the majority, and if all three disagree, choose A. This definitely solves the problem of dropping a write, bit corruption, and shorn writes. But it will not solve the unserializability problem, the flying writes problem, dead device problem, or metadata corruption. But, still, we've attacked two major problems, so this is worth doing.

## The Lampson-Styrgis Assumption

Think of how to build models, how things break, and how to come up with solutions. If you can do this with realistics devices, you're doing well. The Lampson-Styrgis Assumption is a combination of the following assumptions.

1. Storage writes may fail, and when they fail, they might corrupt another piece of storage.
2. A late read can detect the bad sector. That is, if you've written bit corruption, a later read will say there's a bad checksum on this.
3. Repairs can be done in time. If you have a disk failure, someone can go and change that disk with the new disk in time, before another part of the system fails.

If you play your cards right, the SSD's from Zheng's paper will fit the Lampson-Sturgis Assumptions.

## Eggert Assumptions

What assumptions should we assume? Lampson are realistic, but too hard for this lecture. Here are a set of nicer assumptions provided by the Professor.

1. If you write out a block, it either works, or does not. Now, we don't have to worry about the ABC schema.
2. Suppose we do a `write(fd, buf, 8192);` when we write our file, we're writing to an aligned location. Assume `lseek(fd, 0, SEEKSET) % 8192 == 0`
3. Also assume a BSD style file system. Since it's a BSD style file system, it will have [boot block] [super block] [bitmap] [inodes] [data].

When we write a block out, we're replacing it in data. Even when blocks are considered atomic, they are still problematic. We are assuming they are already allocated in the bitmap. If a file grows, we're going to have to add blocks, and write in two areas other than one. But let's assume it's already allocated... Suppose we do a `ls -l` on that file later. Associated with that file is a whole bunch of other information, including the last modifier time: Nov 18 12:53. This time will be stored in the inode.

If we write the data first, and then the time stamp, but theres a crash: the data will be written, the timestamp will be wrong. In the vice versa situation, the timestamp will be right, and data will be wrong. This matters, especially in the case of version control. In this case, update the timestamp first. For make's correctness, for example, it relies on timestamps. It may require make to do extra work, but that's okay. This leads to moving the disk arm between inodes and data, back and forth.

We are going to want to take advantage of a common optimization: do writes, dally timestamp.
The problem: this breaks make, or breaks any application that depends on timestamps being right.
The solution: cleanup procedure ( `make clean` ) after reboot. If you dally the data instead, it requires caching a lot of data in ram, whereas in the vice versa situation, not as much.

## The rename Problem

Consider the following system call:

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

This involves moving the file *a* to *b*, which is located in a different directory. To accomplish this, we need to write e's data, then d's data. However, this approach presents a problem. Say we crash after writing e's data. The link count will still be one, resulting in a possible trashing of data.
Here is a better approach. Assuming the file is at inode 573, we take the following steps to complete this request:

1. Increment the link count of inode 573
2. Write e's data
3. Write d's data
4. Decrement the link count of inode 573

> The link count **never** underestimates the true value of links. This would result in data being trashed. But, it can overestimate. In this event, it can lose disk space, but will not lose the data.

Suppose you have 5 different processes. We can sometimes interleave the processes of the steps above, as long as they have no dependencies with each other.

You need to have a model for the invariance in the file system. Some of these invariances are more important than others. For this purpose, the following statements can be assumed to be true:

| | Consequence of Violation | Condition |
|---|---|---|
| 1 | Distaster | Every block of the file system is used for exactly one purpose: boot, super block, bitmap, inode, data |
| 2 | Disaster | All reference blocks are initialized to data that is appropriate for their type |
| 3 | Disaster | All referenced data blocks are marked as used in the bitmap |
| 4 | Storage | All unreferenced blocks are marked free in the bitmap |

I/O Operations: (4 seek) Each one will be in different area on disk, which takes about 40ms to rename a file. This means that it will take a whopping 40 seconds to rename 1000 files (this sucks). Our goal is to make this faster.

## Performance: Improvement Ideas

### Commit Record

A commit record maintains whether an atomic actiona happened. A commit record is implemented by following the following steps when completing a write.

1. Write "data" outputs into a preliminary area on disk (ie: temporary file). You want the write to act like it is atomic.
2. Wait for all blocks to hit the disk (reordering of disk)
3. Write a commit record saying done 1 block
4. Copy temporary data back to original
5. And if we are worried about two copies existing, write done to commit record

Following the steps above leads to an atomic process:

```
BEGIN
  precommit phase (work is being done, but you can back out)
COMMIT (atomic at lower level) OR ABORT (pretend nothing happened)
  postcommit phase (work, can't back out)
END
```

### Journal

Journaling involves describing the changes to the file system.

In its most extreme form, a journal-based file system is only the journal. It has no self-storage. Only the diffs are stored, like a version-control system. An example of a system like this is ext4 (http://en.wikipedia.org/wiki/Ext4).

Another way to improve performance (assuming hard disk) is to put the journal on a seperate disk drive. In this case, the arms won't interfere with each other.

## Logging Protocol

Write ahead logs:

1. Log planned writes
2. (later) implement them

Write behind logs

1. Log the old data (about to be rewritten)
2. Write new data into the cell
3. Mark as done

Replaying logs

1. Write ahead means the order will be left to right
2. Write behind means the order will be right to left

# File System Robustness

By Ryosuke Takeuchi

File system robustness is the most difficult part of file system implementation; it is where most of the problems tend to occur.

## What can go wrong?

-There may be bugs in the system code
-What is an inexpensive way to find these bugs?
-We can implement invariants in our file system code, and make sure they are true.

**Invariants** are boolean expressions over variables which must always stay true.

## FFS Invariants

1. Every block in the file system is used for exactly one purpose.

  - For example: bool block, super block, bitmap block, inode block, data block
  - Invariant: A block should not be used for more than one purpose
  - This is an obvious one, but we want to check in case someone violates this rule.

2. All referenced blocks are initialized with data that is apprpriate for their block type.

  - For the bitmap block, all blocks should be initialized at 1 since they must all be free.
  - For the inode block, only the blocks being used by the super block should be correctly initialized.
  - For data blocks, all blocks can have any data except the root directory, which should be empty.
  - We make sure these invariants stay true as the file system grows and shrinks.

3. All referenced data blocks are marked as used in the bitmap.
4. All unreferenced blocks are marked as free.

### What happens if any of these are violated?

1. Bitmap may overwrite important data -> data loss.
2. Data is incorrectly initialized.
3. Data may be incorrectly allocated, and may cause security problems.
4. Memory leak (not a critical problem).

## Cache and Performance, Coherence Issues

Because reading from RAM is faster than reading from disk, we want to keep commonly used items cached in RAM.
However, this means we must deal with the problem of **cache coherence**: What do we do if the cache

is not the same as the original data?
This may occur due to a performance technique known as **dallying**.
Dallying is a process in which some tasks are prioritized before working on the task which was originally instructed.

## Caches and System Crashes

When we call write() with a buffer, we may write the buffer to cache instead of disk and return immediately.
If we need to read this data later, we may look at cache instead of disk, leading to speed up.
However, if the system crashes we do not have the data saved to disk!

## What can cause crashes?

Assuming we do not have system bugs, crashes may be cause by power shortages or disk disconnections.
We may prevent this by having an uninterruptable power supply, or by saving the contents of the cache before the power goes out.
Alternatively, we may change our invariants to address the power outage problem.

sync()

- Schedule so that all of the cache is saved to disk, preventing the system from dallying until the entire cache is written.
- This means that everything written before sync is called is saved.
- However, this writes out the entire cache when we may only need part of it.
- This function returns immediately.

fsync(fd)

- Write all of cached data in fd to disk, and wait until finished.
- This takes more real time than sync because it waits.
- This function is slow because even for a write of 1 byte, we must work on at least 1 data block and 1 inode block.

fdatasync(fd)

- Only guaranteed to save the data blocks; not guaranteed for the metadata in inode blocks.
- This is twice as fast because it only uses one block.

## Using Cache for rename

1. rename("d/a", "e/b")
2. read d inode
3. read d data
4. read e inode
5. write inode for files (+1 link count)
6. read e data
7. write e data
8. write d data

9. write d inode
10. write e inode

1-4 can be done in any order as long as they are done before the others.
9 and 10 can be done in any order.

## Commit Records

A drive may contain 2 copies of a file - the commit record tells us which copy of the file we should use.

```
Keep 3 copies of the commit record
1. A ? B B B B B
2. A A A ? B B B
3. A A A A A ? B
Use the majority
---A A A B B B B
```

If all 3 records disagree, use the first record.
This method is usually used for small data such as metadata. (2 copies of data and 3 copies of record)

## Flash Memory

Blocks are used in flash - However, eventually the blocks wear out from usage.
The FTL is used as an image of the drive which does not match what is stored on the drive.
The FTL is part of the controller of the flash drive, spreading out the usage of blocks.
If power is lost, the contents of the FTL is lost - this is similar to a file system.

Problems of Flash Drives:

- Bit corruption
- Shorn writes: Loss of trailing data when writing a block
- Unserializable writes: If power is lost while writing multiple blocks, blocks written in a different order due to the FTL may not be saved
- Metadata corruption
- Total drive failure

To build a reliable file system, we must make assumptions about underlying drives (Lampson/Sturgis assumptions)

- Storage writes can fail, or corrupt another piece of storage. A later read can detect these failures.
- Storage my spontaneously decay.
- Assume errors are rare enough so that repair is feasible.
- Make similar assumptions about process failures.

If we write a block and lose power, we do not know what is written.
With these assumptions we can detect the bad block.
We can also narrow down the file system design - e.g. only need 2 records.

## Using these assumptions to design a robust file system

1. Commit Record

- If we want to do a large action atomically, commit to the record.
- Tells the system if the large object was written or not.

2. Journal
Two Areas: Cell data and Journal

- Cell Data: Data blocks
- Journal: Record of actions and changes made to the data.
- Commit after the changes are made.
- If the system crashes, redo the actions in the journal which are not written to data.
- Can work with a complicated set of instructions such as rename.
- Written sequentially

# CS 111

## Scribe Notes for 11/19/14

*by Christian Thorenfeldt*

# File System Robustness

## Motivation & Invariants

When a file system is built, there are many possible mistakes that can occur. Simply put, you may have bugs in your system code, and we need a way to find those bugs really fast. To alleviate this issue, we introduce the **invariants**, which are boolean expressions that are always true. We have to check our code to keep these invariants true, and as long as they stay true, we know our code passes our basic rules that we've set up.

**Berkeley Fast File System Invariants:**

1. Every block in the file system is used for exactly one purpose. For example, we would look at every block in the system, check that it's a boot block, a superblock, a bitmap block, inode block, or a data block. We want to make sure that the system isn't using a single block for two different purposes.
2. All referenced blocks are initialized with data that is appropriate for their type. For example, a bit map block should be initialized to all 1's (not random values), referenced inode blocks should have a size field set up, and also set up to indicate whether or not the inode is in use.
3. All referenced data blocks are marked used in the bitmap.
4. All unreferenced blocks are marked free.

## Increasing Performance

**What happens when we want to increase performance of our system?** In order to get increase our performance, the system will have to play around with the cache. We can think of the problem as a **cache coherence problem**. We will have to cache a lot of the file system in RAM, maybe 1% of the system, and we hope that the system will talk to the cache.

For example, let's look at the following: /usr/bin/sh

The cursor will first look at the root directory, look at the inodes, and maybe accumulate up to 10 disk accesses for the entire file. That's quite a lot of accesses and will take a significant amount of time if each access takes 20 milliseconds.
**Solution:** We can cache things that we need often such as inode information and commonly used data. Because /usr/bin is so commonly used, most UNIX systems keep this easily accessible in RAM.

However, we now run into the aforementioned problem: cache coherence. If we look at the contents of the cache, it says that, for example, we have a certain copy of a block on disk, but in reality, it's not.

Moreover, to increase performance even more, we use the technique called **Dallying** . The idea is that we have some work that we need to do, but we postpone it a bit to do some work on the way. For example, let's look at the system call: `write (fd, buffer, buf_size)` . The system will place a copy of the buffer into the system cache and immediately return without actually writing the data. If there is a subsequent read in another process, the system will look into the cache to check if the buffer is there and read it. However, this is incredibly vulnerable to crashes since the data isn't saved to disk, which leads us to our next section.

**System Crashes**

There are many reasons why our system might crash. Maybe our flash drive that we are using is abruptly unplugged (a storage disconnect), or perhaps our power runs out and the system shuts down.

At the end of the day, we want to make sure that the disk represents what people write. We have several options to address this problem:

1. `sync()` , which schedules all of the cache to get written to the disk (in effect, stop dallying).
2. `fsync (fd)` , which writes all of fd's cache data to disk and waits for it to finish (but can be quite slow and use extra blocks)
3. `fdatasync (fd)` , which is only guaranteed to save data blocks (not metadata and costs only 1 block).

We are telling the users about cache coherence and complicating user programs by giving the user the responsibility to issue these calls. It would be best if this was invisible to the user.

## Implementing Rename System Call

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

The rename is changing the two directories "d" and "e" (file doesn't change). First directory will have directory entry removed and the second will have a directory entry added to it.

1. We can try to update the directory name, and then update the file name. If we encounter a power failure here, we could lose the file and consequently lose all of our data.
2. If we try to update the file name, and then the directory name, we may end up having two hard links to the same file.

## Example Application… Emacs

Suppose emacs operates on a flash drive. Let's say we have a file that takes 10 blocks in the flash drive. We edit the file, and save the file. However, we want to save the file in a reliable way. When we hit save, we want to know that losing power in the middle of a save means we don't lose any of our data. It is okay though to lose the new data, as long as we still have the old version of the file.

A solution to this problem would be to keep a set of blocks for the old file, and a set of blocks for the new file. That way if we lose power when editing the new file, we still can access the old file's data. However, it might be difficult to differentiate between the two. This is where the **commit record** comes in. We could keep a sort of "directory" block and it tells us which set of block is our old file and which set of blocks is our new file that we are editing. Any changes only occur when we commit a change in the “directory” block.

One more problem though until we can be at peace. What if we lose power during a commit? There could be garbage values. We have to worry about the atomicity of the commit record. To solve this, we could have three copies of the commit record. If two records disagree with one another, choose the majority. This is important if one of the three records has undefined data due to power loss.

## Problems Concerning Flash Drives

Writing to a flash drive over time can wear out a block. A solution to this is FTL which is a flash file system that in a way distributes block usage evenly so as to not wear out commonly used ones, which is completely unbeknownst to the user. For example, if the user wants to use block 3, FTL may assign it to block 7 or some other block.

Moreover, we also have a couple more real world problems concerning flash drives to consider…

1. Bit corruption - maybe we lose power and our bit that was A turned into B
2. Shorn writes - write out a block, lose power, and read the block back in.. and we only have part of the write
3. Unserializable writes
4. Meta data corruption - if we lose metadata, how will the FTL map block 3 to block 7. Everything can be scrambled and this is very bad.
5. Total drive failure - cut the power, turn it on, and nothing works.

We could write a file system that survives these kind of issues, but it’s close to impossible.

We want a file system that is reliable, so we have to make some assumptions about our underlying storage…

## Lampson-Sturgis Assumptions (developed for hard disk)

1. storage writes can fail and it can corrupt another piece of storage
2. storage may spontaneously decay - assume if a block decays, can get a read error.
3. assume errors are rare enough so that repair is feasible.
4. similar assumptions about process failures

With these assumptions, we know there is no such thing as undefined data in a block. If we read a bad block, we will know that it’s undefined as per the assumptions. This will help us narrow down file system design, and perhaps allow us to have two copies of commit records rather than three.

# 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

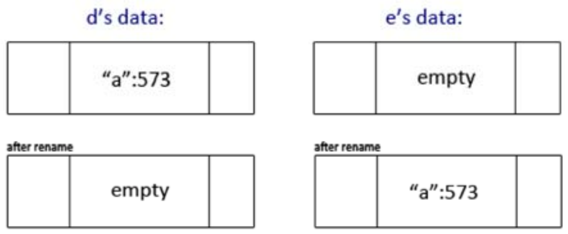
```
rename ("d/a", "d/b")
```

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

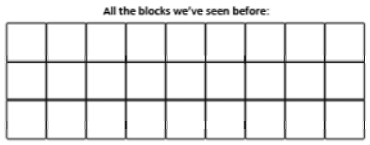
Number 4 has the least consequence. It is not critical during certain system calls, so it is not really an invariant

## Performance Improvement Ideas

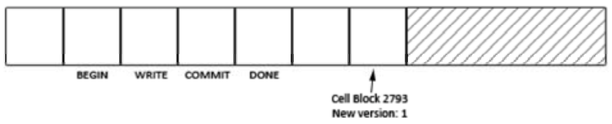
1. Commit Record
   - Write "data" outputs into a preliminary area, e.g. a temp file
   - Wait for all blocks to hit the disk (reordering is OK)
   - Write a commit record saying done
   - Copy temporary data back to original
   - Write "done" to a commit record

```
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

# CS 111 Spring 2014

## Lecture 13: File System Robustness

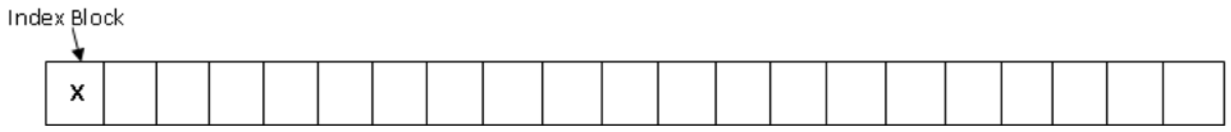
*by Weichen Huang, Yuanzhi Gao, Pengcheng Pan, Yifang Liu*

### Problems in real SSD

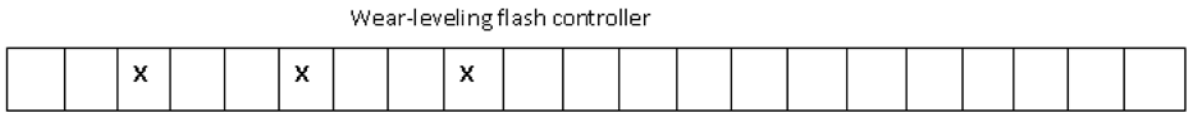
In Understanding the Robustness of SSDs under Power Fault by Zheng et al. at FAST '13, out of 15 SSDs, only 2 worked when they cut the power. The 13 SSDs that failed to work suffered from :

1. **bit corruption :** random bit errors on subsequent reads
2. **flying writes :** Block is written in wrong place when SSD restore.

   Further explanation of flying writes : Flash drive is not as reliable as hard drive. Flash may wear out after writing to the flash drive 10 billion times.

   What flash controller does to overcome this problem is they use wear-leveling.

   You think you are reading from block zero, but you are actually reading wherever block zero happens to be today because the blocks are moving around. So when you lose power when writing to flash, you may lose the table that maps the virtual and physical location, which will give inconsistancy.
3. **shorn writes :** Writes truncated and there is garbage sitting there in the rest of the block when you go back and read it.
4. **metadata corruption :** Drive loses track of much of data location (more serious problem). You can lose large trunk of your original data because it's out there in the physical disk, you just don't know where it is.
5. **dead device :** Fried data. After restore the power, absolutely nothing works. The entire device is dead.
6. **unserializability :** Actual writes are out of order compared to what you and this matters.

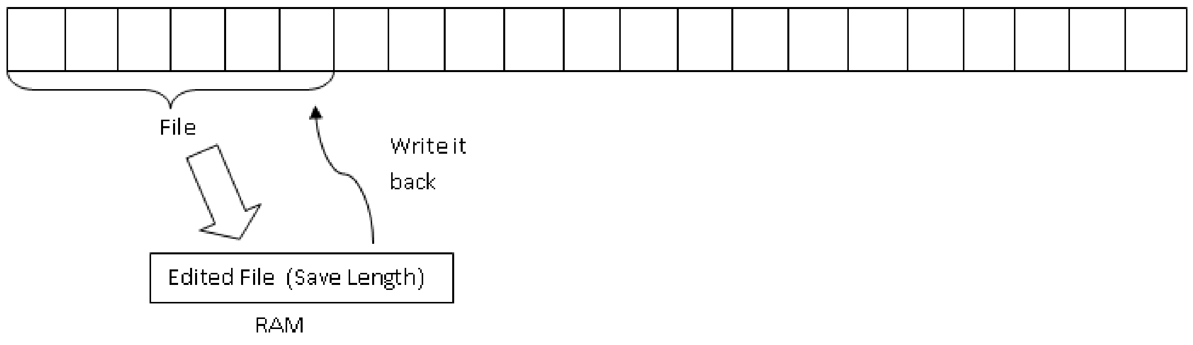
### File System Robustness Goals

- **Durability** - survive limited failure in underlying hardware
- **Atomicity** - Changes are either done or not done
- **Performance** - throughput + latency

### Implementing save in Emacs

**Question :** How should ^x^s (SAVE FILE) work on a raw device?

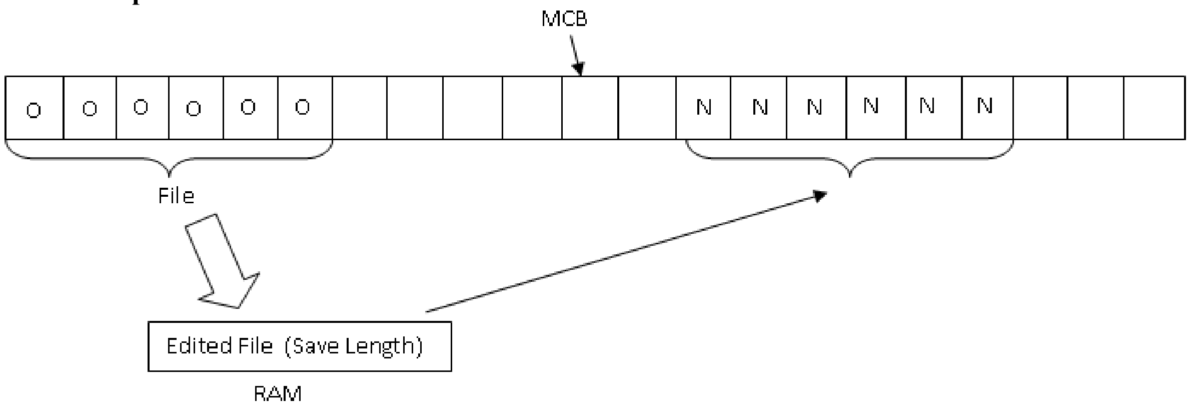
**First Attempt :**



**Problem :** What if unplugged at the middle of the write back?

**GOLDEN RULE OF ATOMICITY :** Never overwrite your only copy, i.e. write into a different location.

**Second Attempt :**



Instead of overwriting your original data, we will overwrite this area so that if the system crashes in the middle, you can continue to refer to your old copy it was not changed and once you finished writing it, you will have the new copy saved on disk as well. So in this way, we obey the golden rule of atomicity.

**Problem :** How do you know which copy you are using after rebooting?

**Timestamp Approach :**

What we can do at application level is that we can put a timestamp in each file. In operating system level is to put timestamp into every block that you write on the disk, which might help you to achieve the reliability here. But that can waste space in each block. Timestamp approach will work but it got its own problem(occupy too much space in block).

**Master Control Block :**

MCB is a block that we keep separate from files. We just store one bit into the master control block and if this bit is zero that means this is the old version of the file. if it's one that means this is the new version of the file.

But this approach does not survive all of the problems. For example, it won't survive the flying write, bit corruption: It actually can't survive any of the problems! But the master control block approach is better if we assume we are one of those 2 of 15 SSD's that actually work without having those problems (bit corruption, flying write).

**Problem with using only one MCB :**

Block size in file system is large. Our problem is that a write we have at file system level can't be implemented atomically at the lower level because the device can't simply write 8 KB block atomically a time. Let's suppose writes can be partial at the lower level. Let's say we pick an 8 KB file system block and the device is 512 bytes sectors. The way you can model it is: if you issue a high level write of 8KB, at the low level, you can issue 8 writes. The way to do it is to issue a bunch of writes to the disk controller and we are going to wait to get 16 responses back.

When you have a block A and you issue a write so that the new contents in the block is B instead of A, in between, the block has indeterminate content (some mixture of A and B that we don't care). Our model is in the middle of a write, if you lose power, the contents are indeterminate. We need an algorithm that helps us reliably tell whether we store a zero or one with its data.

**One idea : we have two copy of MCB instead of just having one.**

They both start up with the same content

| | | |
|---|---|---|
| MCB1 | A | ? |
| MCB2 | A | A |

Then we start to write in MCB1

| | | | |
|---|---|---|---|
| MCB1 | A | ? | B |
| MCB2 | A | A | A |

As time goes on

| | | | | | |
|---|---|---|---|---|---|
| MCB1 | A | ? | B | B | B |
| MCB2 | A | A | A | ? | B |

Notice that at all time during execution, we have either A or B stored, so we will always have reliable data.

**Problem :** we can't tell which one is garbage and which one is correct after you reboot.

**Another idea : use three MCBs**

They start up having the same value

| | |
|---|---|
| MCB1 | A |
| MCB2 | A |
| MCB3 | A |

Then we start to write in MCB1

| | | | |
|---|---|---|---|
| MCB1 | A | ? | B |
| MCB2 | A | A | A |
| MCB3 | A | A | A |

As time goes on

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| MCB1 | A | ? | B | B | B | B | B |
| MCB2 | A | A | A | ? | B | B | B |
| MCB3 | A | A | A | A | A | ? | B |

After reboot, you look at all 3 :

1. If they all agree or If two out of three agree, then use that.
2. If all three disagree, the use MCB1.

## Lampson-Sturgis Assumptions

- Storage writes may fail
  - a later read can detect the bad sector

- Storage may spontaneously decay
- Errors are rare
- Repairs can be done in time

## Professor Eggert's Assumptions

- The only failure is a power failure
- Writes are atomic

## Rename Problem

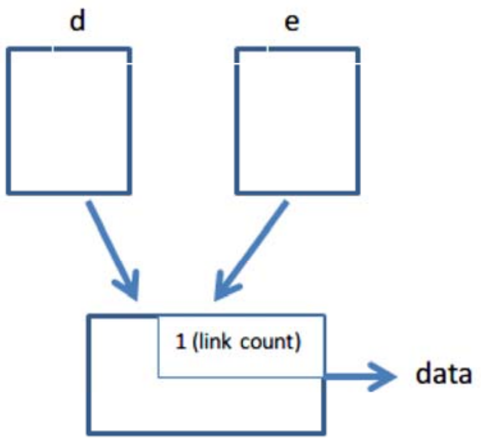
**rename("a","aa");**

**Procedure to complete this request:**

1. allocate a new block
2. write two blocks

Because directory entry varies in length, it can be longer. Rename remains atomic as long as the entry fits into block.

**rename("d/a", "e/b"); - (assume e has space and has no "b")**



Two hard links link to the same file and its link count is 1. When you unlink ("d/a"), link count will go to 0, and thus you will get a dangling link. So what is a safe procedure?

1. increment file's link count (write inode)
2. write e's new block
3. write d's new block
4. decrement file's link count (write inode)
5. write d's new timestamp into its inode
6. write e's new timestamp into its inode

## File system correctness invariants

1. Every block in file system's partition is used for exactly 1 purpose: superblock, bitmap, inode, data.
2. All referenced blocks are properly initialized for their type
3. All referenced data blocks are marked as used in bitmap
4. All unreferenced data blocks are marked as free in bitmap

### Consequence of :

- Violating property 1: multiple purposes at the same time -> you lose data -> disaster
- Violating property 2: allocating data twice -> you lose data -> disaster
- Violating property 3: you lose data -> disaster
- Violating property 4: disk space leak, though fsck would probably fix this

### fsck

- inspects file system and look for storage leaks (while file system is idle)
- updates bitmap accordingly

If you lose power, assume fsck is idempotent.

## File system Robustness Ideas: surviving power failures

#### Commit Record:

> A commit is the single, low-level write operation that matters. It is the operation that actually determines whether or not a file has been written.

#### A general outline for Commit Record process:

```
BEGIN
   Pre-commit phase (can back out, leaving no trace)
Commit
   Post-commit phase (does not affect doneness, may be used for performance reasons such as clean-up)
END
```