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