Introduction

The file systems we've covered so far contain a lot of interesting features. This chapter focuses on reliability (having previously studied storage system reliability in the RAID chapter). How can a file system or storage system provide data security when modern storage technologies are unreliable?

This is known as data integrity or data protection. So, now we'll look into techniques for ensuring that data you store stays the same when it's returned to you.

CRUX: DATA INTEGRITY

How should systems protect data written to storage?
What techniques are necessary?
How can such techniques be made efficient in terms of both space and time?

Disk Failure Modes

As mentioned in the RAID section, disks can fail. Early RAID systems had a simple failure model: either the whole disk works or it doesn't, and that's easy to discover. Building RAID is easy with this fail-stop concept of disk failure.

You didn't learn about the additional failure modes contemporary disks have. A recent study found that current disks appear to be mostly operating but have problems accessing one or more blocks. Latent-sector errors (LSEs) and block corruption are two common single-block failures worth considering. We'll get into each one now.

LSEs occur when a disk sector (or set of sectors) is destroyed. For example, if the disk head contacts the surface (as in a head crash), the surface may be damaged, rendering the bits illegible. Cosmic rays can potentially flip bits, causing errors. If the on-disk bits in a block are not good, and the drive does not have enough information to remedy the fault, the disk will return an error when asked to read them.

In some circumstances, a disk block becomes corrupted in a way that the disk cannot detect. Like a faulty disk. When firmware writes a block to the erroneous location, the disk ECC indicates the block contents are acceptable, but the client receives the wrong block when accessed. A block may also be corrupted when transported from the host to the disk through a bad bus; the disk stores the corrupted data, but the client does not want it. These defects are particularly sneaky since they are silent; the disk returns the defective data with no warning.

This perspective allows disks to fail in parts, as in the classic fail-stop model, or to appear to work but have one or more blocks become inaccessible (LSEs) or contain incorrect data (i.e., corruption). So, while accessing a seemingly-working disk, it may occasionally deliver an error when reading or writing a block (a non-silent partial fault), or it may simply return the erroneous data (a silent partial fault).

Both of these flaws are rare, but how rare?

The figure shows the percent of drives that exhibited at least one LSE or block corruption over the course of the study (about 3 years, over 1.5 million disk drives). The graphic divides the results into "cheap" (SATA) and "costly" drives (usually SCSI or Fibre Channel). As you can see, while buying stronger drives reduces the incidence of these problems (by an order of magnitude), they still occur frequently enough that you need to plan ahead of time for how to deal with them.
Other LSE results include:

• Costly drives with more than one LSE are as likely to develop additional errors as cheaper drives
• For most drives, annual error rate increases in year two
• The number of LSEs increase with disk size
• Most disks with LSEs have less than 50
• Disks with LSEs are more likely to develop additional LSEs
• There exists a significant amount of spatial and temporal locality
• Disk scrubbing is useful (most LSEs were found this way)
Some findings about corruption:
• Chance of corruption varies greatly across different drive models within the same drive class
• Age effects are different across models
• Workload and disk size have little impact on corruption
• Most disks with corruption only have a few corruptions
• Corruption is not independent within a disk or across disks n RAID
• There exists spatial locality, and some temporal locality
• There is a weak correlation with LSEs

A dependable storage system must contain tools to detect and recover from LSEs and block damage.

Handling Latent Sector Errors

We should now investigate these two new partial disk failure modes. Let's start with the simplest one, latent sector faults.

CRUX: HOW TO HANDLE LATENT SECTOR ERRORS

What about hidden sector errors?
How much more equipment is required to handle partial failures?

Latent sector errors are easy to treat because they are (by definition) quickly discovered. Storage systems should use their redundancy mechanisms to return accurate data when a disk reports an error when trying to access a block. It should access the other copy in a mirrored RAID, and it should recreate the block from the other blocks in a parity RAID. Thus, easily discovered issues like LSEs may be recovered using typical redundancy methods.

LSEs have inspired RAID designs over time. When both full-disk faults and LSEs occur in RAID-4/5 systems, an intriguing dilemma develops. When a disk fails, the RAID tries to rebuild it (say, onto a hot spare) by reading the parity group's other disks and recalculating the missing values. If an LSE is found on any of the other drives during reconstruction, we have a problem.

Some systems incorporate extra redundancy to combat this. NetApp's RAID-DP, for example, features two parity disks instead of one. The increased parity helps recreate a missing block when an LSE is identified. The expense of keeping two parity blocks per stripe is higher, although the log-structured design of the NetApp WAFL file system mitigates this in many circumstances. The remaining cost is disk space for the second parity block.

Detecting Corruption

Attacking silent failures via data corruption is a more difficult task. How can we prevent users from receiving corrupted data when drives return corrupted data?

PRESERVING DATA INTEGRITY DESPITE CORRUPTION

What can a storage system do in order to detect corruption when it occurs, given the silent nature of such failures?
Which techniques are needed?
What are the best ways to implement them?

Unlike latent sector defects, finding corruption is difficult. How does a client know a block is bad? Once a block is identified as faulty, recovery is as simple as having another copy of it (hopefully not corrupt!) So we'll look at detection methods.

Modern storage systems use checksums to maintain data integrity. A checksum is just the output of a function that accepts a chunk of data (say, a 4KB block) and computes a function over it (say 4 or 8 bytes). The checksum is the summary. The purpose of such a computation is to allow a system to identify data corruption or alteration by storing the checksum with the data and comparing it to the original storage value.

Common Checksum Function

Checksums are generated using a variety of functions that vary in strength (data integrity protection) and speed (i.e., how quickly can they be computed). A common system trade-off occurs here: the more protection you get, the more it costs.

Some utilize a simple checksum technique based on exclusive or (XOR). When using XOR-based checksums, each chunk of the data block being checked is XORed together to get one value that reflects the complete block.

Imagine computing a 4-byte check-sum over a 16-byte block (this block is of course too small to really be a disk sector or block, but it will serve for the example). The 16 data bytes are hexadecimal:

365e c4cd ba14 8a92 ecef 2c3a 40be f666

If we look at them in binary, we get:

0011 0110 0101 1110
1011 1010 0001 0100
1110 1100 1110 1111
0100 0000 1011 1110

1100 0100 1100 1101
1000 1010 1001 0010
0010 1100 0011 1010
1111 0110 0110 0110

Because the data is arranged in 4 byte rows, the checksum is obvious: XOR each column to get the final checksum:

0010 0000 0001 1011 1001 0100 0000 0011

The result, in hex, is 0x201b9403.

XOR is a reasonable but limited checksum. The checksum will not detect a corruption if two bits in the same place change within a checksummed unit. So people looked for various checksum functions.

Addition is a fundamental checksum function. This method is fast since it only requires 2's complement addition over each block of data, ignoring overflow. It can identify many changes in data, but not shifted data.

The Fletcher checksum is a little more complicated algorithm named after its inventor, John G. Fletcher. It is computed by computing two check bytes, s1 and s2. Assume a block D contains bytes d1… dn; s1 = (s1 + di) mod 255

(computed across all di); s2 = (s2 + s1) mod 255 (again over all di). All single-bit, double-bit, and multiple burst mistakes are detected by the Fletcher checksum.

A cyclic redundancy check is another typical checksum (CRC). Assume you want to checksum a data block D. You just divide D by an agreed-upon value as though it were a huge binary number (k). The CRC value is the remainder of this division. The fact that this binary modulo operation can be implemented efficiently explains the CRC's usefulness in networking. Details elsewhere.

Whatever method is employed, there is no perfect checksum: two data blocks with different contents can have the same checksum, a situation known as a collision. After all, computing a checksum reduces a huge (e.g., 4KB) file to a considerably smaller one (e.g., 4 or 8 bytes). Choosing a suitable checksum function reduces the probability of collisions while being simple to compute.

Checksum Layout

Now that you know how to compute a checksum, let's look at how to use them in storage systems. The first question is how to save checksums on disk.

A checksum is simply stored with each disk sec- tor (or block). Let us name the checksum over a data block D C. (D). Without checksums, the disk configuration is as follows:

The layout adds one checksum per block with checksums:

Because checksums are often short (8 bytes) and drives can only write in sectors (512 bytes) or multiples of 512 bytes, achieving the aforementioned layout can be difficult. Manufacturers format drives with 520-byte sectors, with an extra 8 bytes each sector for the checksum.

On disks without this feature, the file system must store the checksums in 512-byte blocks. One option is as follows:

Then come n data blocks, followed by another checksum sector for the next n blocks, and so on. If the file system intends to overwrite block D1, it must first read in the checksum sector containing C(D1), update C(D1), and then write out the checksum sector and new data block D1 (thus, one read and two writes). Using one checksum per sector requires only one write.

Using Checksums

Having agreed on a checksum layout, we can now learn how to employ checksums. The client (file system or storage controller) receives the stored checksum from disk Cs(D) while reading a block D. (hence the subscript Cs). The client then computes the checksum over the obtained block D (Cc) (D). At this point, the client com- pares the stored and computed checksums; if they are equal (i.e., Cs(D) == Cc(D), the data has likely not been corrupted, and thus can be safely returned to the user. If they do not match (i.e., Cs(D) != Cc(D)), this implies the data has changed since the time it was stored (since the stored checksum reflects the value of the data at that time). In this example, the checksum helped us detect a corruption.

What should we do about a corruption? If the storage system contains a redundant copy, use it instead. If the storage system does not have such a copy, an error is likely. Realize that corruption detection isn't a magic bullet; if you can't acquire the non-corrupted data, you're out of luck.

Misdirected Writes

The aforementioned basic method works effectively for corrupted blocks. The failure modes of newer disks necessitate alternative solutions.

A misdirected write is the initial interest failure mode. Disk and RAID controllers write data to disk successfully, but in the wrong place. In a single-disk system, this means the disk wrote block Dx to address y instead of x (thus “corrupting” Dy); in a multi-disk system, the controller may write Di,x to address x of disk I instead of j. So, our query:

HOW TO RESPOND TO MISDIRECTED WRITINGS

Detection of misdirected writing by a storage system or disk controller
What features does the checksum need to have?

The answer is simple: give each checksum a little more information. In this instance, a physical identifier (ID) is quite useful. As an example, a client can easily detect whether the correct information is saved within a locale if the stored information comprises the checksum C(D) and both the disk and sector numbers. As illustrated in the example below, the stored information should include the client’s disk number and sector offset. If the data does not match, a corruption has occurred due to a misdirected write. On a two-disk system, this extra information looks like this. The checksums are little (e.g., 8 bytes) whereas the blocks are large (e.g., 4 KB or more):

For each block, the disk number is repeated within each block, and the offset of the block in question is also preserved next to it. Redundancy is the key to error detection (in this situation) and recovery (in others). While not required for pristine disks, a little extra information can go a long way in detecting potential issues.

Lost Writes

Sadly, misplaced writes are not the end of the road. So, a lost write occurs when the device alerts the up- per layer that a write has completed but it is never persisted, resulting in the old contents of the block being retained rather than the new ones.

So, do any of our checksumming mechanisms (basic checksums, physical identity) assist detect lost writes? A matching checksum and physical ID (disk number and block offset) are required for this to work. So, our final issue:

HOW TO HANDLE LOST WRITES

How should disk controllers or storage systems detect lost writes?
What features does the checksum need to have?

Performing a write verify or read-after-write ensures that data has reached the disk surface. This method, however, doubles the amount of I/Os required to accomplish a write.

Some systems add a checksum to detect lost writes. For example, Sun's Zettabyte File System (ZFS) includes a checksum in every inode and indirect block. Even if a write to a data block is lost, the inode's checksum will not match the old data. This approach will only fail if both the inode and data writes are lost simultaneously, which is unlikely (but conceivable!).

Scrubbing

After all of this, you may question when these checksums are truly tested. Of course, apps check data when they access it, but most data is rarely accessed and so inspected. The problem with unchecked data is that bit rot can eventually infect all copies of a data set.

Many systems use various sorts of disk scrubbing to solve this issue. The disk system can lessen the probability of all copies of a data item being corrupted by periodically reading over every block and checking checksums. Most systems run scans every night or week.

Overheads of Shecksumming

The overheads of utilizing check-sums for data protection are now discussed. In computer systems, overheads are divided into two categories: space and time.

A space overhead can be either. A stored checksum takes up space on the disk (or other storage device), which can no longer be used for user data. An 8-byte checksum per 4 KB data block typically consumes 0.19% of disk capacity.

The second type of system overhead is memory. When accessing data, both the checksums and the data must be stored in memory. However, if the system just checks the checksum and discards it, the overhead is minimal. This tiny overhead is only noticeable if checksums are retained in memory (to defend against memory corruption).

While space overheads are minor, time overheads from check-summing can be significant. The CPU must compute the checksum over each block both when storing and retrieving data (to compute the checksum again and compare it against the stored checksum). Many systems that use checksums (including network stacks) combine data copying and checksumming into one streamlined process because the copy is needed anyhow (e.g., to transfer data from the kernel page cache into a user buffer).

The extra I/O required for background scrubbing, as well as various checksumming techniques, might cause significant CPU and I/O overheads. In both cases, the impact can be lessened by controlling when the scrubbing occurs. Soaking up the scrubbing activity and strengthening the storage system may be best done late at night, when most (but not all!) productive workers have retired.

Summary

We have examined checksum implementation and usage in current storage systems. As storage devices evolve, new failure modes will inevitably emerge. Maybe the research community and industry will have to rethink some of these core approaches, or devise new ones. We'll see. Or not. Time is strange.