Data Integrity

1. Introduction

Modern filesystems feature checksumming of data and metadata to protect against data corruption. However, the detection of the corruption is done at read time which could potentially be months after the data was written. At that point the original data that the application tried to write is most likely lost.

The solution is to ensure that the disk is actually storing what the application meant it to. Recent additions to both the SCSI family protocols (SBC Data Integrity Field, SCC protection proposal) as well as SATA/T13 (External Path Protection) try to remedy this by adding support for appending integrity metadata to an I/O. The integrity metadata (or protection information in SCSI terminology) includes a checksum for each sector as well as an incrementing counter that ensures the individual sectors are written in the right order. And for some protection schemes also that the I/O is written to the right place on disk.

Current storage controllers and devices implement various protective measures, for instance checksumming and scrubbing. But these technologies are working in their own isolated domains or at best between adjacent nodes in the I/O path. The interesting thing about DIF and the other integrity extensions is that the protection format is well defined and every node in the I/O path can verify the integrity of the I/O and reject it if corruption is detected. This allows not only corruption prevention but also isolation of the point of failure.

2. The Data Integrity Extensions

As written, the protocol extensions only protect the path between controller and storage device. However, many controllers actually allow the operating system to interact with the integrity metadata (IMD). We have been working with several FC/SAS HBA vendors to enable the protection information to be transferred to and from their controllers.

The SCSI Data Integrity Field works by appending 8 bytes of protection information to each sector. The data + integrity metadata is stored in 520 byte sectors on disk. Data + IMD are interleaved when transferred between the controller and target. The T13 proposal is similar.

Because it is highly inconvenient for operating systems to deal with 520 (and 4104) byte sectors, we approached several HBA vendors and encouraged them to allow separation of the data and integrity metadata scatter-gather lists.

The controller will interleave the buffers on write and split them on read. This means that Linux can DMA the data buffers to and from host memory without changes to the page cache.

Also, the 16-bit CRC checksum mandated by both the SCSI and SATA specs is somewhat heavy to compute in software. Benchmarks found that calculating this checksum had a significant impact on system performance for a number of workloads. Some controllers allow a lighter-weight checksum to be used when interfacing with the operating system. Emulex, for instance, supports the TCP/IP checksum instead. The IP checksum received from the OS is converted to the 16-bit CRC when writing and vice versa. This allows the integrity metadata to be generated by Linux or the application at very low cost (comparable to software RAID5).

The IP checksum is weaker than the CRC in terms of detecting bit errors. However, the strength is really in the separation of the data buffers and the integrity metadata. These two distinct buffers must match up for an I/O to complete.

The separation of the data and integrity metadata buffers as well as the choice in checksums is referred to as the Data Integrity Extensions. As these extensions are outside the scope of the protocol bodies (T10, T13), Oracle and its partners are trying to standardize them within the Storage Networking Industry Association.

3. Kernel Changes

The data integrity framework in Linux enables protection information to be pinned to I/Os and sent to/received from controllers that support it.

The advantage to the integrity extensions in SCSI and SATA is that they enable us to protect the entire path from application to storage device. However, at the same time this is also the biggest disadvantage. It means that the protection information must be in a format that can be understood by the disk.

Generally Linux/POSIX applications are agnostic to the intricacies of the storage devices they are accessing. The virtual filesystem switch and the block layer make things like hardware sector size and transport protocols completely transparent to the application.

However, this level of detail is required when preparing the protection information to send to a disk. Consequently, the very concept of an end-to-end protection scheme is a layering violation. It is completely unreasonable for an application to be aware whether it is accessing a SCSI or SATA disk.

The data integrity support implemented in Linux attempts to hide this from the application. As far as the application (and to some extent the kernel) is concerned, the integrity metadata is opaque information that's attached to the I/O.

The current implementation allows the block layer to automatically generate the protection information for any I/O. Eventually the intent is to move the integrity metadata calculation to userspace for user data. Metadata and other I/O that originates within the kernel

will still use the automatic generation interface.

Some storage devices allow each hardware sector to be tagged with a 16-bit value. The owner of this tag space is the owner of the block device. I.e. the filesystem in most cases. The filesystem can use this extra space to tag sectors as they see fit. Because the tag space is limited, the block interface allows tagging bigger chunks by way of interleaving. This way, 8*16 bits of information can be attached to a typical 4KB filesystem block.

This also means that applications such as fsck and mkfs will need access to manipulate the tags from user space. A passthrough interface for this is being worked on.

4. Block Layer Implementation Details

4.1 Bio

The data integrity patches add a new field to struct bio when CONFIG_BLK_DEV_INTEGRITY is enabled. bio_integrity(bio) returns a pointer to a struct bip which contains the bio integrity payload. Essentially a bip is a trimmed down struct bio which holds a bio_vec containing the integrity metadata and the required housekeeping information (bvec pool, vector count, etc.)

A kernel subsystem can enable data integrity protection on a bio by calling bio_integrity_alloc(bio). This will allocate and attach the bip to the bio.

Individual pages containing integrity metadata can subsequently be attached using bio_integrity_add_page().

bio_free() will automatically free the bip.

4.2 Block Device

Because the format of the protection data is tied to the physical disk, each block device has been extended with a block integrity profile (struct blk_integrity). This optional profile is registered with the block layer using blk_integrity_register().

The profile contains callback functions for generating and verifying the protection data, as well as getting and setting application tags. The profile also contains a few constants to aid in completing, merging and splitting the integrity metadata.

Layered block devices will need to pick a profile that's appropriate for all subdevices. blk_integrity_compare() can help with that. DM and MD linear, RAID0 and RAID1 are currently supported. RAID4/5/6 will require extra work due to the application tag.

5.0 Block Layer Integrity API

5.1 Normal Filesystem

The normal filesystem is unaware that the underlying block device is capable of sending/receiving integrity metadata. The IMD will be automatically generated by the block layer at submit_bio() time in case of a WRITE. A READ request will cause the I/O integrity to be verified upon completion.

IMD generation and verification can be toggled using the:

```
/sys/block/<bdev>/integrity/write_generate
```

and:

```
/sys/block/<bdev>/integrity/read_verify
```

flags.

5.2 Integrity-Aware Filesystem

A filesystem that is integrity-aware can prepare I/Os with IMD attached. It can also use the application tag space if this is supported by the block device.

bool bio_integrity_prep(bio);

To generate IMD for WRITE and to set up buffers for READ, the filesystem must call bio_integrity_prep(bio).

Prior to calling this function, the bio data direction and start sector must be set, and the bio should have all data pages added. It is up to the caller to ensure that the bio does not change while I/O is in progress. Complete bio with error if prepare failed for some reson.

5.3 Passing Existing Integrity Metadata

Filesystems that either generate their own integrity metadata or are capable of transferring IMD from user space can use the following calls:

*struct bip * bio_integrity_alloc(bio, gfp_mask, nr_pages);*

Allocates the bio integrity payload and hangs it off of the bio. nr_pages indicate how many pages of protection data need to be stored in the integrity bio_vec list (similar to bio_alloc()).

The integrity payload will be freed at bio_free() time.

int bio_integrity_add_page(bio, page, len, offset);

Attaches a page containing integrity metadata to an existing bio. The bio must have an existing bip, i.e. bio_integrity_alloc() must have been called. For a WRITE, the integrity metadata in the pages must be in a format understood by the target device with the notable exception that the sector numbers will be remapped as the request traverses the I/O stack. This implies that the pages added using this call will be modified during I/O! The first reference tag in the integrity metadata must have a value of bip->bip_sector.

Pages can be added using bio_integrity_add_page() as long as there is room in the bip bio_vec array (nr_pages).

Upon completion of a READ operation, the attached pages will contain the integrity metadata received from the storage device. It is up to the receiver to process them and verify data integrity upon completion.

5.4 Registering A Block Device As Capable Of Exchanging Integrity Metadata

To enable integrity exchange on a block device the gendisk must be registered as capable:

int blk_integrity_register(gendisk, blk_integrity);

The blk_integrity struct is a template and should contain the following:

```
static struct blk_integrity my_profile = {
    .name                   = "STANDARDSBODY-TYPE-VARIANT-CSUM",
    .generate_fn            = my_generate_fn,
    .verify_fn              = my_verify_fn,
    .tuple_size             = sizeof(struct my_tuple_size),
    .tag_size               = <tag bytes per hw sector>,
};
```

'name' is a text string which will be visible in sysfs. This is part of the userland API so chose it carefully and never change it. The format is standards body-type-variant. E.g. T10-DIF-TYPE1-IP or T13-EPP-0-CRC.

'generate_fn' generates appropriate integrity metadata (for WRITE).

'verify_fn' verifies that the data buffer matches the integrity metadata.

'tuple_size' must be set to match the size of the integrity metadata per sector. I.e. 8 for DIF and EPP.

'tag_size' must be set to identify how many bytes of tag space are available per hardware sector. For DIF this is either 2 or 0 depending on the value of the Control Mode Page ATO bit.

2007-12-24 Martin K. Petersen <martin.petersen@oracle.com>