Reprinted from the

Proceedings of the Linux Symposium

July 23rd–26th, 2008
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
Dirk Hohndel, *Intel*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
Matthew Wilson, *rPath*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
Eugene Teo, *Red Hat, Inc.*
Kyle McMartin, *Red Hat, Inc.*
Jake Edge, *LWN.net*
Robyn Bergeron
Dave Boutcher, *IBM*
Mats Wichmann, *Intel*

Linux Data Integrity Extensions

Martin K. Petersen
Oracle
martin.petersen@oracle.com

Abstract

Many databases and filesystems feature checksums on their logical blocks, enabling detection of corrupted data. The scenario most people are familiar with involves bad sectors which develop while data is stored on disk. However, many corruptions are actually a result of errors that occurred when the data was originally written. While a database or filesystem can detect the corruption when data is eventually read back, the good data may have been lost forever.

A recent addition to SCSI allows extra protection information to be exchanged between controller and disk. We have extended this capability up into Linux, allowing filesystems (and eventually applications) to be able to attach integrity metadata to I/O requests. Controllers and disks can then verify the integrity of an I/O before committing it to stable storage. This paper will describe the changes needed to make Linux a data-integrity-aware OS.

1 Data Corruption

As witnessed by the prevalence of RAID deployment in the IT industry, there is a tendency to focus on data corruption caused by disk drive failures. While head misses and general bit corruptions on the platter *are* common problems, there are other possible corruption scenarios that occur frequently enough to warrant being remedied as well.

When corruption is experienced, hardware is often blamed. However, modern systems feature extensive protection on system buses, error checking and correcting memory, etc. In the Fibre Channel environments commonly used in enterprises, wire traffic is protected by a cyclic redundancy check. So in many ways the physical hardware level is becoming increasingly resilient against failures.

The software stack, however, is rapidly growing in complexity. This implies an increasing failure potential: Harddrive firmware, RAID controller firmware, host adapter firmware, operating system code, system libraries, and application errors. There are many things that can go wrong from the time data is generated in host memory until it is stored physically on disk.

Most storage devices feature extensive checking to prevent errors. However, these protective measures are almost exclusively being deployed internally to the device in a proprietary fashion. So far, there have been no means for collaboration between the layers in the I/O stack to ensure data integrity.

An extension to the SCSI family of protocols tries to remedy this by defining a way to check the integrity of an request as it traverses the I/O stack. This is done by appending extra information to the data. This extra information is known as *integrity metadata* or *protection information*.

The integrity metadata enables corrupted write requests to be detected and aborted, thereby preventing silent data corruption.

2 Data Integrity Field

A harddisk is generally divided into blocks of 512 bytes, called *sectors*. On the physical platter, the sector allocation is actually a bit bigger to allow for CRC and information internal to the drive. However, this extra information is not available outside of the disk drive firmware. Consumer-grade disks often trade capacity for reliability and use less space for the error checking information. Enterprise disks feature stronger protection schemes and as a result, expose less storage capacity to the user.

Unlike drives using parallel and serial ATA interfaces,

SCSI[1] disks allow for sectors bigger than 512 bytes to be exposed to the operating system. Sizes of 520 or 528 bytes are common. It is important to note that these 'fat' sectors really are bigger, and that the extra bytes are orthogonal to space used for the drive's internal error checking.

Traditionally these extra few bytes of information have been used by RAID controllers to store their own internal checksums. The drives connected to the RAID head are formatted using 520-byte sectors. When talking to the host operating system, the RAID controller only exposes 512-byte blocks; the remaining 8 bytes are used in a way proprietary to the RAID controller firmware.

A few years ago, an extension to the SCSI Block Commands specification was approved by the T10 technical committee that governs the SCSI family of protocols. The extension, known as *Data Integrity Field*, or *DIF*, standardizes the contents of the extra 8 bytes of information per 520-byte sector.

This allows the integrity metadata to be visible outside of the domain of disk or RAID controller firmware. And as a result, this opens up the possibility of doing true end-to-end data integrity protection.

2.1 The DIF Format

Each 8-byte DIF tuple (see Figure 1) contains 3 tags:

- *Guard tag:* a 16-bit CRC of the sector data.
- *Application tag:* a 16-bit value that can be used by the operating system.
- *Reference tag:* a 32-bit number that is used to ensure the individual sectors are written in the right order, and in some cases, to the right physical sector.

A DIF-capable host adapter will generate the 8 bytes of integrity metadata on a write and append it to the 512-byte sectors received from the host operating system. For read commands, the controller will receive 520-byte sectors from the disk, verify that the integrity metadata matches the data, and return 512-byte sectors to the operating system.

[1]We will use the term 'SCSI' to refer to any device using the SCSI protocol, regardless of whether the physical transport is SPI, Fibre Channel, or SAS.

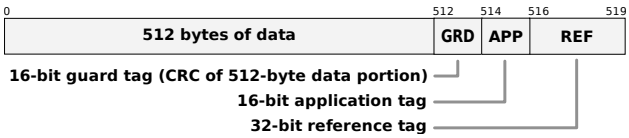

Figure 1: 520-byte sector containing 512 bytes of data followed by 8-byte DIF tuple

A DIF-capable disk drive can compare the data with the integrity metadata received from the host adapter and reject the I/O if there is a mismatch. How to interpret the DIF tuple content depends on the protection type the drive has been formatted with. The current specification allows three types, and they all mandate use of the guard tag to protect the contents of the 512-byte data portion of the sector.

- *DIF Type 1:* reference tag must match lower 32 bits of the target sector number.
- *DIF Type 2:* reference tag must match the seed value in the SCSI command + offset from beginning of I/O.
- *DIF Type 3:* this reference tag is undefined.

The drive must be low-level reformatted to switch between the three protection types or to turn off DIF and return to 512-byte sectors.

3 Data Integrity Extensions

The T10 standards committee only defines communication between SCSI controllers and storage devices. The DIF specification contains no means for sending/receiving integrity metadata to/from host memory, and traditionally host adapter programming interfaces have been proprietary and highly vendor-specific.

Oracle approached several fibre channel adapter vendors putting forth a set of requirements for controllers to allow exchanging integrity metadata with the host operating system. This resulted in a specification [2] for what is now known as the *Data Integrity Extensions*, or *DIX*.

DIX defines a set of interfaces that host adapters must provide in order to send and receive I/O requests with integrity metadata attached. This in turn enables us to extend the exchange of protection information all the way up to the application.

If the controller is DIX-capable and the storage device is DIF-capable, we can create a protection envelope that covers the entire I/O path, thus providing true end-to-end data integrity (see Figure 2).

3.1 Performance Impact

The 16-bit CRC mandated by the DIF specification is somewhat expensive to calculate in software. Benchmarks showed that for some workloads, calculating the CRC16 in software had a detrimental impact on performance. One of Oracle's partners had hardware capable of using the IP checksum instead of CRC16. The IP checksum is much cheaper to calculate and offers weaker protection than the CRC, so there is a trade-off between data integrity and performance. Consequently, the IP checksum feature is optional and can be enabled at will.

If the IP feature is enabled, Linux will put IP checksums in the guard tag instead of CRC16. The controller will verify the checksums and convert them to the T10-mandated CRC before passing the data to the drive. On reads, the opposite conversion takes place.

From a performance perspective, the cost is very low. It has less impact on system performance than software RAID5.

4 SCSI Layer

We have implemented support for both DIF and DIX in the Linux kernel. The work has been done from the bottom up, starting with the SCSI layer. The following sections will describe the changes required.

4.1 Discovery

For the exchange of integrity metadata to happen, it would seem reasonable to require that controller and storage device are DIX- and DIF-capable, respectively. However, even in a setup where the disk does not support DIF, there is still value in having the host adapter verify the data integrity before sending the command on to the drive.

Similarly, some controllers may support DIF while talking to the drive, but may not have the capability to exchange integrity metadata with Linux. In that situation it is still desirable to have communications between host adapter and disk protected.

Consequently two orthogonal negotiations are taking place at discovery time: One for DIX between Linux and the SCSI controller, and one for DIF between controller and storage device.

The controller driver indicates its DIF and DIX capabilities when it registers itself with the SCSI layer. The DIF type is probed when a drive is scanned. If both DIX and DIF are supported, integrity metadata can be exchanged end-to-end.

4.2 Scatter-Gather List Separation

A buffer in host memory that needs to be transferred to or from a storage device is virtually contiguous. This means that the application sees it as one linear blob of data. In reality the buffer is likely to be physically discontiguous, made up of several scattered portions of physical memory. Consequently, a more complex construct is needed to describe what to transfer.

Network and storage controllers use a scatter-gather list for this purpose. The list consists of one or more `<page address, offset, length>` tuples, each identifying a region in memory to transfer as part of the request.

Linux performs all block I/O in multiples of 512 bytes and it would be highly inconvenient to support 520-byte sectors and buffers throughout the kernel.

On the wire between controller and disk, however, integrity metadata must be interleaved with the data sectors; therefore, the buffer sent to the disk must be a multiple of 520 bytes long.

As a result, DIX requires separating the data and integrity metadata in host memory. The data buffer remains unchanged, while the integrity metadata is stored in a separate buffer. The two buffers are then mapped into separate scatter-gather lists which are handed to the I/O controller.

When writing, the controller will transfer the memory described by the two scatterlists from the host, check them, and interleave data and integrity metadata before the request goes out on the wire as 520-byte sectors.

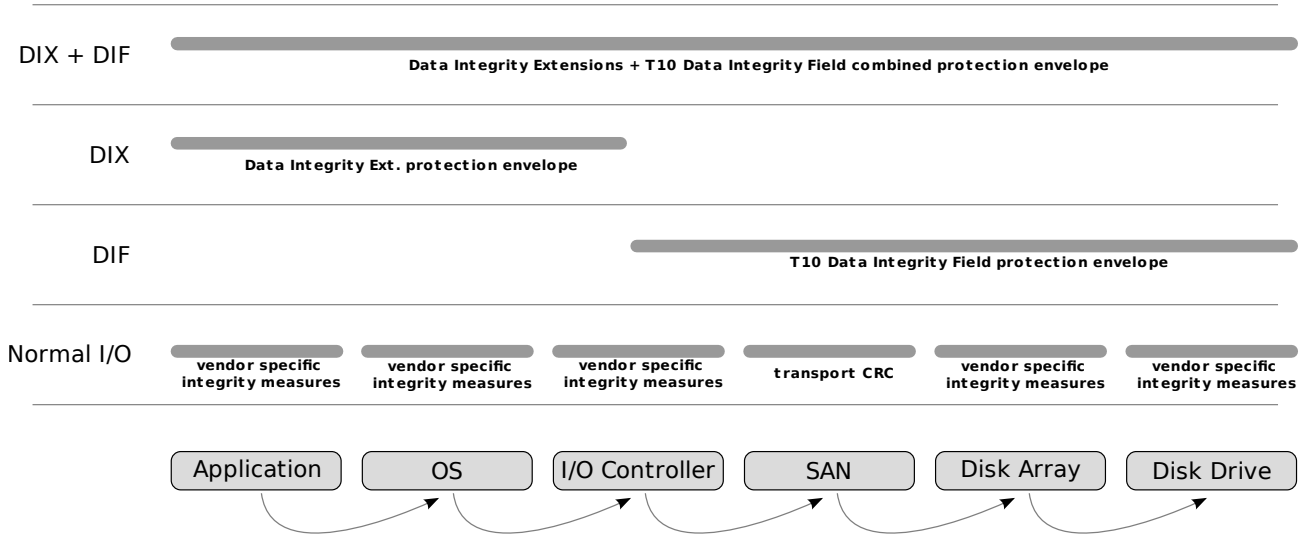

Figure 2: Protection Envelopes: The 'Normal I/O' line shows disjoint protection domains in a normal setup. Above that, the 'DIF' line illustrates the area covered by the T10 DIF standard. 'DIX' displays the coverage of the Data Integrity Extensions, and at the top, 'DIX+DIF' combined yields a full end-to-end protection envelope.

On read, the 520-byte sectors sent by the drive are verified, split up, and transferred into the host memory described by the two scatter-gather lists provided by the kernel.

This separation of data and integrity metadata makes it much less intrusive to support DIF in the kernel.

The integrity buffer is described by an extra `scsi_data_buffer` in `struct scsi_cmnd`, which is the container for SCSI requests in the kernel.

4.3 Reference Tag Remapping

When a drive is formatted with Type 1 protection, the reference tag must contain a value corresponding to the physical sector the data is being written to for the I/O to complete successfully. Thanks to partitioning and stackable devices such as MD or the Device Mapper, the physical sector LBA is often very different from what the filesystem requests when submitting the I/O. The reference tag needs to be remapped accordingly.

One solution would be to postpone filling out the reference tag until the physical sector number is actually known. However, we would like to leverage the protection offered by the reference tag's ability to tie the individual sectors of an I/O together.

Another option would be for the filesystem to recursively query the underlying block devices requesting the start LBA. Unfortunately, this will not work, as an I/O may straddle physical devices. The solution is to have a virtual reference tag filled out when the I/O is submitted by the filesystem. That virtual tag is then remapped to the physical value at the bottom of the I/O stack when writing. Similarly, when data is read, the physical reference tags received from the drive are remapped to the virtual numbers expected by the filesystem.

This approach also avoids multiple remapping steps as the request traverses a layered I/O stack.

5 Block Layer

Conceptually, DIF and DIX constitute *a blatant layering violation*. Applications do not know or care whether they are accessing a SATA or a SCSI disk, or whether the data is mounted over the network. On the other hand, for the end-to-end protection to work, applications or filesystems need to know how to prepare integrity metadata in a format understood by the actual physical device.

Thankfully, the provider of the integrity metadata does not have to be aware of the intricate details of what is inside the integrity buffer, and consequently the block layer treats the integrity metadata in an opaque fashion.

It has no idea what is stored inside the extra structure attached to the `bio`.[2]

5.1 Block Integrity Payload

The integrity metadata is stored in the block integrity payload, or `bip` struct which attached to the `bio`. The `bip` is essentially a trimmed-down version of the I/O vector portions of a `struct bio` with a few extra fields for housekeeping, including the virtual sector number used for remapping.

A series of `bio_integrity_*` calls allows interaction with the protection information, and these have been designed to closely mirror the calls for allocating `bio` structures, adding pages to them, etc.

5.2 Integrity Properties, Splitting and Merging

There are only a few things the block layer really needs to be aware of with respect to the attached protection information:

- Because a `bio` can be split and merged, the block layer needs to know how much integrity metadata goes with each chunk of data.
- The layer needs to know whether the device is capable of storing extra information in the application tag.
- It must be capable of generating and verifying the integrity metadata.

All this information is communicated to the block layer when a storage device registers itself using `blk_integrity_register()`. In the DIF case, this is done just after the SCSI disk makes its presence known to the kernel.

The 16 bits of space in the DIF application tag may or may not be used internally by the storage device. A bit in the device's SCSI `Control Mode Page` indicates whether it is available. If it is, the SCSI disk driver will signal to the block layer that the space is available for use by the filesystem.

As part of that registration process, the SCSI disk driver also provides two callback functions to the block layer: one for generating integrity metadata, and one for verifying integrity metadata. This way, the block layer can call the functions to opaquely generate and check protection information without knowing the intricate details of SCSI, DIF, or how the drive has been formatted.

5.3 Stacked Devices

Servers often use software RAID (MD) and/or the Logical Volume Manager. These are implemented as virtual block devices inside the kernel. If all the disks that constitute an MD disk or a logical volume support the same type of protection, the virtual block device is tagged as being integrity-capable.

A similar approach is taken for virtual block devices exposed to virtualized guests, allowing the protection envelope to reach all the way from the application running on the guest through the hypervisor to the storage device.

5.4 Automatic Generation/Verification

Filesystems that allow integrity metadata to be transferred to/from userland are expected to interact directly with the `bip` calls. However, legacy filesystems like `ext3` and `ext4` are not integrity-aware. There are also other I/O code paths that either originate inside the kernel or map user pages directly. For those cases, the integrity infrastructure allows protection information to be automatically generated by the block layer (writes) or verified before the `bio` is returned to the submitter (reads).

Normally, I/O completion is run in interrupt context, as it usually only involves marking the pages referenced by the request as being up-to-date. However, calculating a checksum for the entire I/O is a time-consuming process. If the request needs to be verified, completion is postponed using a `work_queue`.

The automatic generation/verification of integrity metadata enables integrity protection of all I/O from the block layer to the disk without any changes to the filesystem code.

6 Filesystem Interface

6.1 Protection Information Passthrough

Filesystems that wish to allow transfer of integrity metadata to and from userland applications will need to man-

[2] `struct bio` is the fundamental block I/O container in the Linux kernel.

ually attach it to the `bio`. This is done by attaching a `bip` to the `bio` and then adding the protection information pages using `bio_integrity_add_page()`.

6.2 Tagging

As mentioned above, the DIF tuple includes a 16-bit application tag that is stored by the block device as any other type of data; i.e., it is not used for integrity verification in any of the existing protection types.

These 16 bits can be used freely by the owner of the block device—in this case the filesystem—to tag the sectors. One possible use is to identify which inode a sector belongs to. This will significantly improve the `fsck` process' ability to recover a damaged filesystem.

Filesystems generally use blocks that are bigger than 512 bytes. Because two bytes per sector is a very limited space, the block integrity infrastructure allows tagging at the `bio` level instead. An opaque buffer containing the filesystem-internal information can be supplied at integrity-metadata-generation time. The data in the buffer is then interleaved between the application tags in the sectors targeted by the `bio`, enabling the filesystem to store 16 bytes of recovery information for each 4KB logical block.

The tag data can subsequently be read back by running `bio_integrity_get_tag()` upon completion of a read `bio`.

7 Future Work

Work is in progress to implement support for the data integrity extensions in btrfs [1], enabling the filesystem to use the application tag. The next step will be defining the interfaces that will allow applications to perform syscalls that include integrity metadata.

We are working on three different interfaces that expose integrity metadata to userspace applications:

1. *Transparent:* Integrity metadata is generated by the C library transparently to the application.

2. *Opaque:* This interface will allow the application to protect a buffer in memory prior to submitting the I/O to disk. Just like the block layer, the application will not know that the actual integrity metadata is in DIF format.

3. *Explicit:* Some applications will need direct access to the native integrity metadata, bypassing the filesystem. Examples are the `mkfs` and `fsck` programs that need to be able to read and write the application tag directly.

T13, the committee that governs the SATA specification, has proposed a feature called *External Path Protection* which is essentially the same as DIF. The Linux kernel data integrity infrastructure has been designed to accommodate DIF as well as EPP. A similar data integrity feature for SCSI tape drives is also in development.

Products supporting DIF and DIX are scheduled for general availability in 2008. The Linux Data Integrity Project can be found at `http://oss.oracle.com/projects/data-integrity/`.

Acknowledgements

Thanks to Randy Dunlap, Joel Becker, and Zach Brown for their feedback on this paper.

References

[1] Chris Mason. btrfs. `http://oss.oracle.com/projects/btrfs/`.

[2] Martin K. Petersen. I/O Controller Requirements for Data Integrity Aware Operating Systems. `http://oss.oracle.com/projects/data-integrity/dist/documentation/dif-dma.pdf`.