NAME

GEOM - modular disk I/O request transformation framework

SYNOPSIS

options GEOM_BDE
options GEOM_CACHE
options GEOM_CONCAT
options GEOM_ELI
options GEOM_GATE
options GEOM_JOURNAL
options GEOM_LABEL
options GEOM_LINUX_LVM
options GEOM_MAP
options GEOM_MIRROR
options GEOM_MOUNTVER
options GEOM_MULTIPATH
options GEOM_NOP
options GEOM_PART_APM
options GEOM_PART_BSD
options GEOM_PART_BSD64
options GEOM_PART_EBR
options GEOM_PART_EBR_COMPAT
options GEOM_PART_GPT
options GEOM_PART_LDM
options GEOM_PART_MBR
options GEOM_RAID
options GEOM_RAID3
options GEOM_SHSEC
options GEOM_STRIPE
options GEOM_UZIP
options GEOM_VIRSTOR
options GEOM_ZERO

DESCRIPTION

The **GEOM** framework provides an infrastructure in which "classes" can perform transformations on disk I/O requests on their path from the upper kernel to the device drivers and back.

Transformations in a **GEOM** context range from the simple geometric displacement performed in typical disk partitioning modules over RAID algorithms and device multipath resolution to full blown cryptographic protection of the stored data.

Compared to traditional "volume management", **GEOM** differs from most and in some cases all previous implementations in the following ways:

- **GEOM** is extensible. It is trivially simple to write a new class of transformation and it will not be given stepchild treatment. If someone for some reason wanted to mount IBM MVS diskpacks, a class recognizing and configuring their VTOC information would be a trivial matter.

- **GEOM** is topologically agnostic. Most volume management implementations have very strict notions of how classes can fit together, very often one fixed hierarchy is provided, for instance, subdisk - plex - volume.

Being extensible means that new transformations are treated no differently than existing transformations.

Fixed hierarchies are bad because they make it impossible to express the intent efficiently. In the fixed hierarchy above, it is not possible to mirror two physical disks and then partition the mirror into subdisks, instead one is forced to make subdisks on the physical volumes and to mirror these two and two, resulting in a much more complex configuration. **GEOM** on the other hand does not care in which order things are done, the only restriction is that cycles in the graph will not be allowed.

TERMINOLOGY AND TOPOLOGY

GEOM is quite object oriented and consequently the terminology borrows a lot of context and semantics from the OO vocabulary:

A "class", represented by the data structure *g_class* implements one particular kind of transformation. Typical examples are MBR disk partition, BSD disklabel, and RAID5 classes.

An instance of a class is called a "geom" and represented by the data structure *g_geom*. In a typical i386 FreeBSD system, there will be one geom of class MBR for each disk.

A "provider", represented by the data structure *g_provider*, is the front gate at which a geom offers service. A provider is "a disk-like thing which appears in */dev*" - a logical disk in other words. All providers have three main properties: "name", "sectorsize" and "size".

A "consumer" is the backdoor through which a geom connects to another geom provider and through which I/O requests are sent.

The topological relationship between these entities are as follows:

- A class has zero or more geom instances.

- A geom has exactly one class it is derived from.
- A geom has zero or more consumers.
- A geom has zero or more providers.
- A consumer can be attached to zero or one providers.
- A provider can have zero or more consumers attached.

All geoms have a rank-number assigned, which is used to detect and prevent loops in the acyclic directed graph. This rank number is assigned as follows:

1. A geom with no attached consumers has rank=1.
2. A geom with attached consumers has a rank one higher than the highest rank of the geoms of the providers its consumers are attached to.

SPECIAL TOPOLOGICAL MANEUVERS

In addition to the straightforward attach, which attaches a consumer to a provider, and detach, which breaks the bond, a number of special topological maneuvers exists to facilitate configuration and to improve the overall flexibility.

TASTING is a process that happens whenever a new class or new provider is created, and it provides the class a chance to automatically configure an instance on providers which it recognizes as its own. A typical example is the MBR disk-partition class which will look for the MBR table in the first sector and, if found and validated, will instantiate a geom to multiplex according to the contents of the MBR.

A new class will be offered to all existing providers in turn and a new provider will be offered to all classes in turn.

Exactly what a class does to recognize if it should accept the offered provider is not defined by **GEOM**, but the sensible set of options are:

- Examine specific data structures on the disk.
- Examine properties like "sectorsize" or "mediasize" for the provider.
- Examine the rank number of the provider's geom.

- Examine the method name of the provider's geom.

ORPHANIZATION is the process by which a provider is removed while it potentially is still being used.

When a geom orphans a provider, all future I/O requests will "bounce" on the provider with an error code set by the geom. Any consumers attached to the provider will receive notification about the orphanization when the event loop gets around to it, and they can take appropriate action at that time.

A geom which came into being as a result of a normal taste operation should self-destruct unless it has a way to keep functioning whilst lacking the orphaned provider. Geoms like disk slicers should therefore self-destruct whereas RAID5 or mirror geoms will be able to continue as long as they do not lose quorum.

When a provider is orphaned, this does not necessarily result in any immediate change in the topology: any attached consumers are still attached, any opened paths are still open, any outstanding I/O requests are still outstanding.

The typical scenario is:

- A device driver detects a disk has departed and orphans the provider for it.
- The geoms on top of the disk receive the orphanization event and orphan all their providers in turn. Providers which are not attached to will typically self-destruct right away. This process continues in a quasi-recursive fashion until all relevant pieces of the tree have heard the bad news.
- Eventually the buck stops when it reaches geom_dev at the top of the stack.
- Geom_dev will call destroy_dev(9) to stop any more requests from coming in. It will sleep until any and all outstanding I/O requests have been returned. It will explicitly close (i.e.: zero the access counts), a change which will propagate all the way down through the mesh. It will then detach and destroy its geom.
- The geom whose provider is now detached will destroy the provider, detach and destroy its consumer and destroy its geom.
- This process percolates all the way down through the mesh, until the cleanup is complete.

While this approach seems byzantine, it does provide the maximum flexibility and robustness in handling disappearing devices.

The one absolutely crucial detail to be aware of is that if the device driver does not return all I/O requests, the tree will not unravel.

SPOILING is a special case of orphanization used to protect against stale metadata. It is probably easiest to understand spoiling by going through an example.

Imagine a disk, *da0*, on top of which an MBR geom provides *da0s1* and *da0s2*, and on top of *da0s1* a BSD geom provides *da0s1a* through *da0s1e*, and that both the MBR and BSD geoms have autoconfigured based on data structures on the disk media. Now imagine the case where *da0* is opened for writing and those data structures are modified or overwritten: now the geoms would be operating on stale metadata unless some notification system can inform them otherwise.

To avoid this situation, when the open of *da0* for write happens, all attached consumers are told about this and geoms like MBR and BSD will self-destruct as a result. When *da0* is closed, it will be offered for tasting again and, if the data structures for MBR and BSD are still there, new geoms will instantiate themselves anew.

Now for the fine print:

If any of the paths through the MBR or BSD module were open, they would have opened downwards with an exclusive bit thus rendering it impossible to open *da0* for writing in that case. Conversely, the requested exclusive bit would render it impossible to open a path through the MBR geom while *da0* is open for writing.

From this it also follows that changing the size of open geoms can only be done with their cooperation.

Finally: the spoiling only happens when the write count goes from zero to non-zero and the retasting happens only when the write count goes from non-zero to zero.

CONFIGURE is the process where the administrator issues instructions for a particular class to instantiate itself. There are multiple ways to express intent in this case - a particular provider may be specified with a level of override forcing, for instance, a BSD disklabel module to attach to a provider which was not found palatable during the TASTE operation.

Finally, I/O is the reason we even do this: it concerns itself with sending I/O requests through the graph.

I/O REQUESTS, represented by *struct bio*, originate at a consumer, are scheduled on its attached provider and, when processed, are returned to the consumer. It is important to realize that the *struct bio* which enters through the provider of a particular geom does not "come out on the other side". Even simple transformations like MBR and BSD will clone the *struct bio*, modify the clone, and schedule the clone on their own consumer. Note that cloning the *struct bio* does not involve cloning the actual data area specified in the I/O request.

In total, four different I/O requests exist in **GEOM**: read, write, delete, and "get attribute".

Read and write are self explanatory.

Delete indicates that a certain range of data is no longer used and that it can be erased or freed as the underlying technology supports. Technologies like flash adaptation layers can arrange to erase the relevant blocks before they will become reassigned and cryptographic devices may want to fill random bits into the range to reduce the amount of data available for attack.

It is important to recognize that a delete indication is not a request and consequently there is no guarantee that the data actually will be erased or made unavailable unless guaranteed by specific geoms in the graph. If "secure delete" semantics are required, a geom should be pushed which converts delete indications into (a sequence of) write requests.

"Get attribute" supports inspection and manipulation of out-of-band attributes on a particular provider or path. Attributes are named by ASCII strings and they will be discussed in a separate section below.

(Stay tuned while the author rests his brain and fingers: more to come.)

DIAGNOSTICS

Several flags are provided for tracing **GEOM** operations and unlocking protection mechanisms via the *kern.geom.debugflags* sysctl. All of these flags are off by default, and great care should be taken in turning them on.

0x01 (G_T_TOPOLOGY)
: Provide tracing of topology change events.

0x02 (G_T_BIO)
: Provide tracing of buffer I/O requests.

0x04 (G_T_ACCESS)
: Provide tracing of access check controls.

0x08 (unused)

0x10 (allow foot shooting)
: Allow writing to Rank 1 providers. This would, for example, allow the super-user to overwrite the MBR on the root disk or write random sectors elsewhere to a mounted disk. The implications are obvious.

0x40 (G_F_DISKIOCTL)
: This is unused at this time.

0x80 (G_F_CTLDUMP)
: Dump contents of gctl requests.

SEE ALSO

libgeom(3), geom(8), DECLARE_GEOM_CLASS(9), disk(9), g_access(9), g_attach(9), g_bio(9), g_consumer(9), g_data(9), g_event(9), g_geom(9), g_provider(9), g_provider_by_name(9)

HISTORY

This software was initially developed for the FreeBSD Project by Poul-Henning Kamp and NAI Labs, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

The following obsolete **GEOM** components were removed in FreeBSD 13.0:

- **GEOM_BSD**,
- **GEOM_FOX**,
- **GEOM_MBR**,
- **GEOM_SUNLABEL**, and
- **GEOM_VOL**.

Use

- **GEOM_PART_BSD**,
- **GEOM_MULTIPATH**,
- **GEOM_PART_MBR**, and
- **GEOM_LABEL**

options, respectively, instead.

AUTHORS

Poul-Henning Kamp <*phk@FreeBSD.org*>