Writing a GEOM Class

Abstract

This text documents some starting points in developing GEOM classes, and kernel modules in general. It is assumed that the reader is familiar with C userland programming.

Table of Contents

1. Introduction 1
2. Preliminaries 2
3. On FreeBSD Kernel Programming 4
4. On GEOM Programming 5

1. Introduction

1.1. Documentation

Documentation on kernel programming is scarce - it is one of few areas where there is nearly nothing in the way of friendly tutorials, and the phrase "use the source!" really holds true. However, there are some bits and pieces (some of them seriously outdated) floating around that should be studied before beginning to code:

- The FreeBSD Developer's Handbook - part of the documentation project, it does not contain anything specific to kernel programming, but rather some general useful information.
- The FreeBSD Architecture Handbook - also from the documentation project, contains descriptions of several low-level facilities and procedures. The most important chapter is 13, Writing FreeBSD device drivers.
- The Blueprints section of FreeBSD Diary web site - contains several interesting articles on kernel facilities.
- The man pages in section 9 - for important documentation on kernel functions.
- The geom(4) man page and PHK's GEOM slides - for general introduction of the GEOM subsystem.
- Man pages g_bio(9), g_event(9), g_data(9), g_geom(9), g_provider(9), g_consumer(9), g_access(9) & others linked from those, for documentation on specific functionalities.
- The style(9) man page - for documentation on the coding-style conventions which must be followed for any code which is to be committed to the FreeBSD tree.

2. Preliminaries

The best way to do kernel development is to have (at least) two separate computers. One of these would contain the development environment and sources, and the other would be used to test the newly written code by network-booting and network-mounting filesystems from the first one. This way if the new code contains bugs and crashes the machine, it will not mess up the sources (and other "live" data). The second system does not even require a proper display. Instead, it could be connected with a serial cable or KVM to the first one.

But, since not everybody has two or more computers handy, there are a few things that can be done to prepare an otherwise "live" system for developing kernel code. This setup is also applicable for developing in a VMWare or QEmu virtual machine (the next best thing after a dedicated development machine).

2.1. Modifying a System for Development

For any kernel programming a kernel with `INVARIANTS` enabled is a must-have. So enter these in your kernel configuration file:

```
options INVARIANT_SUPPORT
options INVARIANTS
```

For more debugging you should also include WITNESS support, which will alert you of mistakes in locking:

```
options WITNESS_SUPPORT
options WITNESS
```

For debugging crash dumps, a kernel with debug symbols is needed:

```
  makeoptions    DEBUG=-g
```

With the usual way of installing the kernel (`make installkernel`) the debug kernel will not be automatically installed. It is called kernel.debug and located in /usr/obj/usr/src/sys/KERNELNAME/. For convenience it should be copied to /boot/kernel/.

Another convenience is enabling the kernel debugger so you can examine a kernel panic when it happens. For this, enter the following lines in your kernel configuration file:

```
options KDB
options DDB
options KDB_TRACE
```

For this to work you might need to set a sysctl (if it is not on by default):

```
debug.debugger_on_panic=1
```

Kernel panics will happen, so care should be taken with the filesystem cache. In particular, having softupdates might mean the latest file version could be lost if a panic occurs before it is committed to storage. Disabling softupdates yields a great performance hit, and still does not guarantee data consistency. Mounting filesystem with the "sync" option is needed for that. For a compromise, the softupdates cache delays can be shortened. There are three sysctl's that are useful for this (best to be set in /etc/sysctl.conf):

```
kern.filedelay=5
kern.dirdelay=4
kern.metadelay=3
```

The numbers represent seconds.

For debugging kernel panics, kernel core dumps are required. Since a kernel panic might make filesystems unusable, this crash dump is first written to a raw partition. Usually, this is the swap partition. This partition must be at least as large as the physical RAM in the machine. On the next boot, the dump is copied to a regular file. This happens after filesystems are checked and mounted, and before swap is enabled. This is controlled with two /etc/rc.conf variables:

```
dumpdev="/dev/ad0s4b"
dumpdir="/usr/core
```

The `dumpdev` variable specifies the swap partition and `dumpdir` tells the system where in the filesystem to relocate the core dump on reboot.

Writing kernel core dumps is slow and takes a long time so if you have lots of memory (>256M) and lots of panics it could be frustrating to sit and wait while it is done (twice - first to write it to swap, then to relocate it to filesystem). It is convenient then to limit the amount of RAM the system will use via a /boot/loader.conf tunable:

```
hw.physmem="256M"
```

If the panics are frequent and filesystems large (or you simply do not trust softupdates+background fsck) it is advisable to turn background fsck off via /etc/rc.conf variable:

```
background_fsck="NO"
```

This way, the filesystems will always get checked when needed. Note that with background fsck, a new panic could happen while it is checking the disks. Again, the safest way is not to have many local filesystems by using another computer as an NFS server.

2.2. Starting the Project

For the purpose of creating a new GEOM class, an empty subdirectory has to be created under an arbitrary user-accessible directory. You do not have to create the module directory under /usr/src.

2.3. The Makefile

It is good practice to create Makefiles for every nontrivial coding project, which of course includes kernel modules.

Creating the Makefile is simple thanks to an extensive set of helper routines provided by the system. In short, here is how a minimal Makefile looks for a kernel module:

```
SRCS=g_journal.c
KMOD=geom_journal

.include <bsd.kmod.mk>
```

This Makefile (with changed filenames) will do for any kernel module, and a GEOM class can reside in just one kernel module. If more than one file is required, list it in the `SRCS` variable, separated with whitespace from other filenames.

3. On FreeBSD Kernel Programming

3.1. Memory Allocation

See malloc(9). Basic memory allocation is only slightly different than its userland equivalent. Most notably, `malloc`() and `free`() accept additional parameters as is described in the man page.

A "malloc type" must be declared in the declaration section of a source file, like this:

```
static MALLOC_DEFINE(M_GJOURNAL, "gjournal data", "GEOM_JOURNAL Data");
```

To use this macro, sys/param.h, sys/kernel.h and sys/malloc.h headers must be included.

There is another mechanism for allocating memory, the UMA (Universal Memory Allocator). See uma(9) for details, but it is a special type of allocator mainly used for speedy allocation of lists comprised of same-sized items (for example, dynamic arrays of structs).

3.2. Lists and Queues

See queue(3). There are a LOT of cases when a list of things needs to be maintained. Fortunately, this data structure is implemented (in several ways) by C macros included in the system. The most used list type is TAILQ because it is the most flexible. It is also the one with largest memory requirements (its elements are doubly-linked) and also the slowest (although the speed variation is

on the order of several CPU instructions more, so it should not be taken seriously).

If data retrieval speed is very important, see tree(3) and hashinit(9).

3.3. BIOs

Structure `bio` is used for any and all Input/Output operations concerning GEOM. It basically contains information about what device ('provider') should satisfy the request, request type, offset, length, pointer to a buffer, and a bunch of "user-specific" flags and fields that can help implement various hacks.

The important thing here is that `bio`s are handled asynchronously. That means that, in most parts of the code, there is no analogue to userland's read(2) and write(2) calls that do not return until a request is done. Rather, a developer-supplied function is called as a notification when the request gets completed (or results in error).

The asynchronous programming model (also called "event-driven") is somewhat harder than the much more used imperative one used in userland (at least it takes a while to get used to it). In some cases the helper routines `g_write_data`() and `g_read_data`() can be used, but *not always*. In particular, they cannot be used when a mutex is held; for example, the GEOM topology mutex or the internal mutex held during the `.start`() and `.stop`() functions.

4. On GEOM Programming

4.1. Ggate

If maximum performance is not needed, a much simpler way of making a data transformation is to implement it in userland via the ggate (GEOM gate) facility. Unfortunately, there is no easy way to convert between, or even share code between the two approaches.

4.2. GEOM Class

GEOM classes are transformations on the data. These transformations can be combined in a tree-like fashion. Instances of GEOM classes are called *geoms*.

Each GEOM class has several "class methods" that get called when there is no geom instance available (or they are simply not bound to a single instance):

- `.init` is called when GEOM becomes aware of a GEOM class (when the kernel module gets loaded.)
- `.fini` gets called when GEOM abandons the class (when the module gets unloaded)
- `.taste` is called next, once for each provider the system has available. If applicable, this function will usually create and start a geom instance.
- `.destroy_geom` is called when the geom should be disbanded
- `.ctlconf` is called when user requests reconfiguration of existing geom

Also defined are the GEOM event functions, which will get copied to the geom instance.

Field `.geom` in the `g_class` structure is a LIST of geoms instantiated from the class.

These functions are called from the g_event kernel thread.

4.3. Softc

The name "softc" is a legacy term for "driver private data". The name most probably comes from the archaic term "software control block". In GEOM, it is a structure (more precise: pointer to a structure) that can be attached to a geom instance to hold whatever data is private to the geom instance. Most GEOM classes have the following members:

- `struct g_provider *provider` : The "provider" this geom instantiates
- `uint16_t n_disks` : Number of consumer this geom consumes
- `struct g_consumer **disks` : Array of `struct g_consumer*`. (It is not possible to use just single indirection because struct g_consumer* are created on our behalf by GEOM).

The `softc` structure contains all the state of geom instance. Every geom instance has its own softc.

4.4. Metadata

Format of metadata is more-or-less class-dependent, but MUST start with:

- 16 byte buffer for null-terminated signature (usually the class name)
- uint32 version ID

It is assumed that geom classes know how to handle metadata with version ID's lower than theirs.

Metadata is located in the last sector of the provider (and thus must fit in it).

(All this is implementation-dependent but all existing code works like that, and it is supported by libraries.)

4.5. Labeling/creating a GEOM

The sequence of events is:

- user calls geom(8) utility (or one of its hardlinked friends)
- the utility figures out which geom class it is supposed to handle and searches for geom_CLASSNAME.so library (usually in /lib/geom).
- it dlopen(3)-s the library, extracts the definitions of command-line parameters and helper functions.

In the case of creating/labeling a new geom, this is what happens:

- geom(8) looks in the command-line argument for the command (usually `label`), and calls a

helper function.

- The helper function checks parameters and gathers metadata, which it proceeds to write to all concerned providers.
- This "spoils" existing geoms (if any) and initializes a new round of "tasting" of the providers. The intended geom class recognizes the metadata and brings the geom up.

(The above sequence of events is implementation-dependent but all existing code works like that, and it is supported by libraries.)

4.6. GEOM Command Structure

The helper geom_CLASSNAME.so library exports `class_commands` structure, which is an array of `struct g_command` elements. Commands are of uniform format and look like:

```
verb [-options] geomname [other]
```

Common verbs are:

- label - to write metadata to devices so they can be recognized at tasting and brought up in geoms
- destroy - to destroy metadata, so the geoms get destroyed

Common options are:

- `-v` : be verbose
- `-f` : force

Many actions, such as labeling and destroying metadata can be performed in userland. For this, `struct g_command` provides field `gc_func` that can be set to a function (in the same .so) that will be called to process a verb. If `gc_func` is NULL, the command will be passed to kernel module, to `.ctlreq` function of the geom class.

4.7. Geoms

Geoms are instances of GEOM classes. They have internal data (a softc structure) and some functions with which they respond to external events.

The event functions are:

- `.access` : calculates permissions (read/write/exclusive)
- `.dumpconf` : returns XML-formatted information about the geom
- `.orphan` : called when some underlying provider gets disconnected
- `.spoiled` : called when some underlying provider gets written to
- `.start` : handles I/O

These functions are called from the `g_down` kernel thread and there can be no sleeping in this context, (see definition of sleeping elsewhere) which limits what can be done quite a bit, but forces the handling to be fast.

Of these, the most important function for doing actual useful work is the `.start`() function, which is called when a BIO request arrives for a provider managed by a instance of geom class.

4.8. GEOM Threads

There are three kernel threads created and run by the GEOM framework:

- `g_down` : Handles requests coming from high-level entities (such as a userland request) on the way to physical devices
- `g_up` : Handles responses from device drivers to requests made by higher-level entities
- `g_event` : Handles all other cases: creation of geom instances, access counting, "spoil" events, etc.

When a user process issues "read data X at offset Y of a file" request, this is what happens:

- The filesystem converts the request into a struct bio instance and passes it to the GEOM subsystem. It knows what geom instance should handle it because filesystems are hosted directly on a geom instance.
- The request ends up as a call to the `.start`() function made on the g_down thread and reaches the top-level geom instance.
- This top-level geom instance (for example the partition slicer) determines that the request should be routed to a lower-level instance (for example the disk driver). It makes a copy of the bio request (bio requests *ALWAYS* need to be copied between instances, with `g_clone_bio`()!), modifies the data offset and target provider fields and executes the copy with `g_io_request`()
- The disk driver gets the bio request also as a call to `.start`() on the `g_down` thread. It talks to hardware, gets the data back, and calls `g_io_deliver`() on the bio.
- Now, the notification of bio completion "bubbles up" in the `g_up` thread. First the partition slicer gets `.done`() called in the `g_up` thread, it uses information stored in the bio to free the cloned `bio` structure (with `g_destroy_bio`()) and calls `g_io_deliver`() on the original request.
- The filesystem gets the data and transfers it to userland.

See g_bio(9) man page for information how the data is passed back and forth in the `bio` structure (note in particular the `bio_parent` and `bio_children` fields and how they are handled).

One important feature is: *THERE CAN BE NO SLEEPING IN G_UP AND G_DOWN THREADS*. This means that none of the following things can be done in those threads (the list is of course not complete, but only informative):

- Calls to `msleep`() and `tsleep`(), obviously.
- Calls to `g_write_data`() and `g_read_data`(), because these sleep between passing the data to consumers and returning.
- Waiting for I/O.

- Calls to malloc(9) and `uma_zalloc`() with `M_WAITOK` flag set
- sx and other sleepable locks

This restriction is here to stop GEOM code clogging the I/O request path, since sleeping is usually not time-bound and there can be no guarantees on how long will it take (there are some other, more technical reasons also). It also means that there is not much that can be done in those threads; for example, almost any complex thing requires memory allocation. Fortunately, there is a way out: creating additional kernel threads.

4.9. Kernel Threads for Use in GEOM Code

Kernel threads are created with kthread_create(9) function, and they are sort of similar to userland threads in behavior, only they cannot return to caller to signify termination, but must call kthread_exit(9).

In GEOM code, the usual use of threads is to offload processing of requests from `g_down` thread (the `.start`() function). These threads look like "event handlers": they have a linked list of event associated with them (on which events can be posted by various functions in various threads so it must be protected by a mutex), take the events from the list one by one and process them in a big `switch`() statement.

The main benefit of using a thread to handle I/O requests is that it can sleep when needed. Now, this sounds good, but should be carefully thought out. Sleeping is well and very convenient but can very effectively destroy performance of the geom transformation. Extremely performance-sensitive classes probably should do all the work in `.start`() function call, taking great care to handle out-of-memory and similar errors.

The other benefit of having a event-handler thread like that is to serialize all the requests and responses coming from different geom threads into one thread. This is also very convenient but can be slow. In most cases, handling of `.done`() requests can be left to the `g_up` thread.

Mutexes in FreeBSD kernel (see mutex(9)) have one distinction from their more common userland cousins - the code cannot sleep while holding a mutex). If the code needs to sleep a lot, sx(9) locks may be more appropriate. On the other hand, if you do almost everything in a single thread, you may get away with no mutexes at all.