Escribiendo una clase GEOM

Resumen

Este texto documenta algunos puntos de partida en el desarrollo de clases GEOM y módulos del kernel en general. Se asume que el lector está familiarizado con la programación en C en modo usuario.

Tabla de contenidos

1. Introducción 1
2. Preliminares 2
3. Programación del kernel de FreeBSD 4
4. Programación GEOM 5

1. Introducción

1.1. Documentación

La documentación sobre la programación del kernel es escasa - es una de las pocas áreas donde casi no hay tutoriales amigables, y la frase, "¡usa el código fuente!", realmente es cierta. Sin embargo, hay algunos trozos (algunos de ellos muy desactualizados) flotando alrededor que deben estudiarse antes de comenzar a programar:

- El FreeBSD Developer's Handbook - parte del proyecto de documentación, no contiene nada específico a la programación del kernel sino más bien algo de información útil en general.
- El FreeBSD Architecture Handbook - también parte del proyecto de documentación, contiene descripciones de varios servicios y procedimientos de bajo nivel. El capítulo más importante es el 13, Writing FreeBSD device drivers.
- La sección Blueprints del sitio web FreeBSD Diary - contiene varios artículos interesantes sobre servicios del kernel.
- Las páginas del manual en la sección 9 — para documentación importante sobre las funciones del kernel.
- La página de manual geom(4) y PHK's GEOM slides - para una introducción general al subsistema GEOM.
- Las páginas del manual g_bio(9), g_event(9), g_data(9), g_geom(9), g_provider(9), g_consumer(9), g_access(9) y otras enlazadas desde estas, para documentación sobre funcionalidades específicas.
- La página del manual style(9) - para documentación sobre las convenciones que debe seguir el estilo del código para todo aquel que se va a añadir al árbol de FreeBSD.

2. Preliminares

La mejor forma de hacer desarrollo del kernel es tener (al menos) dos ordenadores separados. Uno de ellos debería de tener el entorno de desarrollo y el código fuente, y el otro sería usado para probar el código recién escrito, inicializando y montando su sistema de archivos a través de la red desde el primer ordenador. De esta forma, si el nuevo código contiene errores y bloquea el ordenador, esto no dañará el código fuente (ni ningún otro dato que este ejecutándose en "caliente"). El segundo sistema ni siquiera necesita un monitor adecuado. En su lugar, podría estar conectado con un cable serie o KVM al primer ordenador.

Pero como no todo el mundo tiene dos o más ordenadores a mano, hay unas pocas cosas que se pueden hacer para preparar un entorno "en caliente" para desarrollar código del kernel. Esta configuración también se puede aplicar para desarrollar en una máquina virtual VMWare o QEmu (lo siguiente mejor después de tener una máquina como entorno dedicado).

2.1. Modificar un sistema para el desarrollo

Para cualquier programación del kernel, es obligatoria tener activada la opción `INVARIANTS`. Así que añade estas líneas a tu archivo de configuración del kernel:

```
options INVARIANT_SUPPORT
options INVARIANTS
```

Para tener un mayor nivel de depuración, también debes incluir el soporte de WITNESS, que te advertirá sobre errores relacionados con los bloqueos:

```
options WITNESS_SUPPORT
options WITNESS
```

Para depurar los volcados de memoria, se necesita un kernel con símbolos de depuración:

```
  makeoptions    DEBUG=-g
```

Con la forma habitual de instalar el kernel (`make installkernel`) no se instala el kernel de depuración de forma automática. Se llama kernel.debug y se encuentra en /usr/obj/usr/src/sys/KERNELNAME/. Por comodidad se debería copiar a /boot/kernel/.

Otra cosa útil es habilitar la depuración del kernel para que puedas examinar el kernel panic cuando suceda. Para esto, introduce las siguientes líneas en su archivo de configuración del kernel:

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

Para que esto funcione, es posible que necesites establecer un sysctl (si no está activado de forma predeterminada):

```
debug.debugger_on_panic=1
```

Ocurrirán kernel panics, por lo que se debe tener cuidado con la cache del sistema de archivos. En particular, tener habilitadas las softupdates puede significar que la última versión del archivo podría perderse si se produce un kernel panic antes de que se haya hecho commit al almacenamiento. Deshabilitar las softupdates produce un gran impacto en el rendimiento, y no garantiza la consistencia de los datos. Para eso es necesario montar el sistema de archivos con la opción "sync". Como solución de compromiso, se pueden acortar los tiempos de espera de la cache de softupdates. Hay tres sysctls que son útiles para esto (lo mejor es establecerlas en /etc/sysctl.conf):

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

Los números representan segundos.

Para depurar los kernel panics, los volcados del kernel (core dumps) son necesarios. Dado que un kernel panic podría inutilizar los sistemas de archivos, este volcado de memoria se graba primero en una partición sin formato (raw). Por lo general, ésta es la partición swap. Esta partición debe ser al menos tan grande como la RAM física del ordenador. En el siguiente arranque, el volcado se copia en un archivo normal. Esto sucede después de que los sistemas de archivos se verifiquen y monten, y antes de habilitar la swap. Este proceso se controla con dos variables en /etc/rc.conf:

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

La variable `dumpdev` especifica la partición swap y `dumpdir` le indica al sistema dónde copiar el dore dump al reiniciar.

Escribir volcados del kernel es lento y lleva mucho tiempo, por lo que si tienes mucha memoria (>256M) y muchos kernel panics, puede ser frustrante sentarse y esperar mientras se hace (dos veces — primero escribirlo en el swap, luego reubicarlo al sistema de archivos). Es conveniente limitar la cantidad de RAM que utilizará el sistema a través de una variable en /boot/loader.conf:

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

Si los kernel panics son frecuentes y los sistemas de archivos son grandes (o simplemente no confías en softupdates + fsck en segundo plano), es aconsejable desactivar fsck en segundo plano mediante la siguiente variable en /etc/rc.conf:

```
background_fsck="NO"
```

De esta forma, los sistemas de archivos siempre serán verificados cuando sea necesario. Ten en cuenta que con fsck en segundo plano, podría producirse un nuevo kernel panic mientras comprueba los discos. Una vez más, la forma más segura es no tener muchos sistemas de archivos sino utilizar otro ordenador como servidor NFS.

2.2. Empezando el proyecto

Con el fin de crear una nueva clase GEOM, se debe crear un subdirectorio vacío bajo un directorio arbitrario que sea accesible por el usuario. No es necesario crear el directorio del módulo en /usr/src.

2.3. El Makefile

Es una buena práctica crear Makefiles para cada proyecto de programación que no sea trivial, lo que incluye obviamente módulos del kernel.

Crear el archivo Makefile es sencillo gracias a un extenso conjunto de rutinas de ayuda proporcionadas por el sistema. En resumen, aquí hay un ejemplo de cómo es un Makefile mínimo para un módulo del kernel:

```
SRCS=g_journal.c
KMOD=geom_journal

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

Este Makefile (con nombres de archivo modificados) funcionará para cualquier módulo del kernel, y una clase GEOM puede residir en un solo módulo del kernel. Si se necesita más de un archivo, añadelo a la variable `SRCS`, separado con espacios en blanco de los otros nombres de archivos.

3. Programación del kernel de FreeBSD

3.1. Asignación de memoria

Lee malloc(9). La asignación básica de memoria sólo es un poco diferente de su equivalente en espacio de usuario. Lo más llamativo es que `malloc`() y `free`() aceptan parámetros adicionales como se describe en la página del manual.

Se tiene que declarar un "malloc type" en la sección de declaraciones de un fichero de código de este modo:

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

Para usar esta macro se tienen que incluir los ficheros de cabecera sys/param.h, sys/kernel.h y sys/malloc.h.

Hay otro mecanismo para reservar memoria, el UMA (Universal Memory Allocator). Lee uma(9) para obtener detalles, pero es un tipo especial de gestor que se utiliza principalmente para acelerar la reserva de listas compuestas de elementos del mismo tamaño (por ejemplo, arrays dinámicos de estructuras).

3.2. Listas y colas

Lee queue(3). Hay MUCHOS casos en los que se necesita mantener una lista de cosas. Afortunadamente esta estructura de datos se implementa (de muchas formas) mediante macros de C incluidas en el sistema. El tipo de lista más utilizado es TAILQ porque es el más flexible. También es el que requiere más memoria (sus elementos están doblemente enlazados) y también el más lento (aunque la variación de velocidad está en el orden de varias instrucciones de CPU, así que esto no se debería tomar en serio).

Si la velocidad de recuperación de los datos es importante, consulta tree(3) y hashinit(9).

3.3. BIOs

La estructura `bio` se utiliza para todas las operaciones de Entrada/Salida que tengan que ver con GEOM. Básicamente contiene información acerca de qué dispositivo ('provider') debería satisfacer la petición, el tipo de petición, el desplazamiento, la longitud, el puntero al buffer, y un montón de flags y campos "específicos de usuario" que pueden ayudar a implementar varios hacks.

Lo importante aquí es que los `bio` se manejan de forma asíncrona. Esto quiere decir que en la mayor parte del código no hay un análogo a las llamadas read(2) y write(2) de espacio de usuario que no retornan hasta que la petición ha terminado. En su lugar se utiliza una función proporcionada por el desarrollador que es invocada como una notificación cuando la solicitud se ha completado (o ha terminado en error).

El modelo asíncrono de programación (también llamado orientado a eventos, o "event-driven") es algo más difícil que el modo imperativo que se usa mucho más en espacio de usuario (al menos lleva un tiempo acostumbrarse). En algunos casos se pueden usar las rutinas de soporte `g_write_data`() y `g_read_data`() pero *no siempre*. En concreto, no se pueden usar cuando se está bloqueando en un mutex; por ejemplo, el mutex para la topología de GEOM o el mutex interno que se adquiere en las funciones `.start`() y `.stop`().

4. Programación GEOM

4.1. Ggate

Si no se necesita el máximo rendimiento, una forma mucho más sencilla de realizar una transformación de datos es implementarla en el espacio de usuario a través del servicio ggate (GEOM gate). Desafortunadamente, no hay una manera fácil de convertir o incluso compartir código entre las dos aproximaciones.

4.2. Clase GEOM

Las clases GEOM son transformaciones sobre los datos. Estas transformaciones se pueden combinar en forma de árbol. Las instancias de las clases GEOM se llaman *geoms*.

Cada clase GEOM tiene varios "métodos de clase" que son invocados cuando no se dispone de una instancia geom (o simplemente no están ligadas a una única instancia):

- `.init` se llama cuando GEOM se entera de una nueva clase GEOM (cuando se carga el módulo del kernel.)
- `.fini` se llama cuando GEOM abandona la clase (cuando se descarga el módulo)
- A continuación se llama a `.taste`, una vez por cada proveedor que el sistema tenga disponible. Si corresponde, esta función generalmente creará e iniciará una instancia geom.
- `.destroy_geom` se llama cuando se debe desmantelar el geom
- `.ctlconf` se invoca cuando el usuario solicita la reconfiguración de un geom existente

También se definen las funciones de eventos GEOM, que se copiarán en la instancia de geom.

El campo `.geom` en la estructura `g_class` es una lista (LIST) de los geoms instanciados a partir de la clase.

Estas funciones son llamadas desde el hilo del kernel g_event.

4.3. Softc

El nombre "softc" es un término heredado para"driver private data" (datos privados del controlador). El nombre probablemente proviene del término arcaico "software control block" (bloque de control software). En GEOM, es una estructura (para ser más precisos: un puntero a una estructura) que se puede adjuntar a una instancia de geom para mantener cualquier información que sea privada para dicha instancia. La mayoría de las clases de GEOM tienen los siguientes elementos:

- `struct g_provider *provider` : El "provider" que instancia este geom
- `uint16_t n_disks` : Número de consumidores que consume este geom
- `struct g_consumer **disks` : Array de `struct g_consumer*`. (No es posible utilizar un sólo nivel de indirección porque los struct g_consumer* los crea GEOM en nuestro nombre).

La estructura `softc` contiene el estado completo de la instancia geom. Cada instancia geom tiene su propio softc.

4.4. Metadatos

El formato de los metadatos es más o menos dependiente de la clase, pero DEBE comenzar por:

- Un buffer de 16 bytes para la firma terminada en null (normalmente el nombre de la clase)
- ID de la versión del tipo uint32

Se supone que las clases de geom saben cómo manejar los metadatos con ID de versión menores que los suyos.

Los metadatos se encuentran en el último sector del proveedor (y, por lo tanto, deben encajar en él).

(Todo depende de la implementación, pero todo el código existente funciona así, y es compatible con las bibliotecas.)

4.5. Etiquetar/crear un GEOM

La secuencia de eventos es:

- El usuario invoca la utilidad geom(8) (o alguno de sus amigos que están enlazados)
- la utilidad averigua qué clase geom se supone que tiene que manejar y busca la librería geom_CLASSNAME.so (que está normalmente en /lib/geom).
- usa dlopen(3) para cargar la librería y extrae las definiciones de los parámetros de línea de comandos y de las funciones de apoyo.

En el caso de crear/etiquetar un nuevo geom, esto es lo que sucede:

- geom(8) busca el comando (normalmente `label`) en los argumentos de línea de comando y llama a la función de apoyo.
- La función auxiliar comprueba los parámetros y recopila los metadatos, que procede a escribir a todos los proveedores interesados.
- Esto "echa a perder" los geoms existentes (si los hubiera) e inicializa una nueva ronda de "pruebas" de los proveedores. La clase geom reconoce los metadatos y levanta el geom.

(La secuencia de eventos anterior depende de la implementación, pero todo el código existente funciona así, y es compatible con las bibliotecas.)

4.6. Estructura del comando GEOM

La librería de apoyo geom_CLASSNAME.so exporta la estructura `class_commands` que es un array de elementos de tipo `struct g_command`. Los comandos tienen un formato uniforme que se parece a:

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

Los verbos comunes son:

- label — para escribir metadatos en los dispositivos para que puedan ser reconocidos en la prueba y creados en geoms
- destroy — para destruir los metadatos, de forma que se destruyen los geoms

Las opciones comunes son:

- `-v` : sé verboso

- `-f` : forzar

Muchas acciones, como etiquetar y destruir los metadatos, se pueden hacer en espacio de usuario. Para esto, `struct g_command` proporciona el campo `gc_func` al que se puede asignar una función (en el mismo .so) que será llamada para procesar un verbo. Si `gc_func` es NULL, el comando se pasará al módulo del kernel, a la función `.ctlreq` de la clase geom.

4.7. Geoms

Los geoms son instancias de clases GEOM. Tienen datos internos (una estructura softc) y algunas funciones con las que responden a eventos externos.

Las funciones del evento son:

- `.access` : calcula permisos (read/write/exclusive)
- `.dumpconf` : devuelve información sobre el geom en formato XML
- `.orphan` : llamada cuando algún proveedor subyacente se desconecta
- `.spoiled` : llamada cuando se escribe en algún proveedor subyacente
- `.start` : maneja E/S (I/O)

Estas funciones se llaman desde el hilo `g_down` del kernel y no puede haber inactividad en este contexto, (consulta la definición de inactividad en otra parte) lo que limita un poco lo que se puede hacer, pero obliga a que el manejo sea rápido.

De estas funciones, la más importante para realizar un trabajo útil real es la función `.start`() , que se llama cuando una solicitud BIO llega a un proveedor administrado por una instancia de la clase geom.

4.8. Hilos de GEOM

Hay tres hilos del kernel creados y ejecutados por el framework GEOM:

- `g_down` : Maneja las peticiones que vienen de entidades de nivel superior (como una petición de espacio de usuario) hacia los dispositivos físicos
- `g_up` : Maneja respuestas de los controladores de dispositivos a las peticiones hechas por entidades de nivel superior
- `g_event` : Maneja los demás casos: creación de instancias geom, contadores de acceso, eventos "spoil", etc.

Cuando un proceso de usuario realiza una petición de tipo "lee el dato X en el offset Y de un fichero", esto es lo que sucede:

- El sistema de archivos convierte la solicitud en una instancia de struct bio y lo transmite al subsistema GEOM. Sabe qué instancia geom debería encargarse porque los sistemas de archivos se alojan directamente en una instancia geom.
- La solicitud termina como una llamada a la función `.start`() realizada en el hilo g_down y llega

a la instancia de geom de nivel superior.

- Esta instancia de nivel superior (por ejemplo el particionador) determina que la petición se debería dirigir a una instancia de nivel inferior (por ejemplo el controlador del disco). Hace una copia de la petición bio (¡las peticiones bio *SIEMPRE* se tienen que copiar entre instancias, con `g_clone_bio`()!), modifica los campos para el offset de los datos y el proveedor objetivo y ejecuta la copia con `g_io_request`()
- El controlador de disco también obtiene la petición bio al llamar a la función `.start`() del hilo `g_down`. Habla con el hardware, obtiene los datos y llama a `g_io_deliver`() en el bio.
- Ahora, la notificación de bio completada "sube" en el hilo `g_up`. Primero se llama a `.done`() del particionador en el hilo `g_up`, usa la información almacenada en el bio para liberar la estructura `bio` clonada (con `g_destroy_bio`()) y llama a `g_io_deliver`() en la petición original.
- El sistema de archivos obtiene los datos y los transfiere al espacio de usuario.

Consulta la página de manual g_bio(9) para obtener información sobre cómo se pasan los datos de un lado para otro en la estructura `bio` (en particular date cuenta cómo se manejan los campos `bio_parent` y `bio_children`).

Una característica importante es que: *NO PUEDE HABER HILOS G_UP Y G_DOWN QUE SE VAYAN A DORMIR*. Esto significa que en esos hilos no se puede hacer ninguna de las siguientes cosas (la lista por supuesto no está completa, es sólo informativa):

- Llamadas a `msleep`() y `tsleep`(), evidentemente.
- Llamadas a `g_write_data`() y `g_read_data`(), porque estas duermen entre el paso de datos hacia los consumidores y la vuelta.
- Esperar por la E/S.
- Llamadas a malloc(9) y `uma_zalloc`() con el flag `M_WAITOK` establecido
- sx y otros sleepable locks

Esta restricción está aquí para impedir que el código GEOM obstruya la ruta de las solicitudes de E/S, ya que el dormir no tiene limite de tiempo y puede no haber garantías sobre cuánto tiempo tardará (también hay algunas otras razones más técnicas). También significa que no hay mucho que se pueda hacer en estos hilos; por ejemplo, prácticamente cualquier cosa compleja requiere asignación de memoria. Afortunadamente, hay una salida: crear hilos adicionales en el kernel.

4.9. Hilos del kernel para usar en el código GEOM

Los hilos del kernel se crean con la función kthread_create(9) y se comportan de una forma similar a los hilos en espacio de usuario, sólo que no pueden volver al llamante para indicarles que han terminado sino que tienen que invocar kthread_exit(9).

En el código GEOM, el uso habitual de los hilos es para descargar el procesamiento de peticiones del hilo `g_down`(la función `.start`()). Estos hilos parecen "manejadores de eventos": tienen vinculada una lista de eventos asociados a ellos (en los cuales los eventos pueden publicarse mediante varias funciones en varios hilos, por lo que deben estar protegidos por un mutex), toma los eventos de la lista, uno por uno, y los procesa en una gran instrucción `switch`().

La principal ventaja de utilizar un hilo para manejar las solicitudes de E/S es que pueden dormir (sleep) cuando sea necesario. Ahora, esto suena bien, pero se debe pensar cuidadosamente. Dormir (sleeping) es bueno y muy conveniente, pero puede ser muy efectivo destruyendo el rendimiento de la transformación de geom. Las clases extremadamente sensibles al rendimiento probablemente deberían hacer todo el trabajo en la llamada a la función `.start`(), teniendo mucho cuidado de manejar los errores de falta de memoria y similares.

El otro beneficio de tener un hilo manejador de eventos como este es serializar en un sólo hilo todas las peticiones y respuestas que vienen de los distintos hilos de geom. Esto también es muy cómodo pero puede ser lento. En la mayoría de los casos, el manejo de las peticiones de `.done`() se puede dejar en manos del hilo `g_up`.

Los mutex en el kernel de FreeBSD (consulta mutex(9)) tienen una característica distintiva respecto de sus primos en espacio de usuario - el código no puede dormir mientras se tiene un mutex cogido. Si el código necesita dormir a menudo, los locks sx(9) podrían ser más apropiados. Por otro lado, si haces casi todo en un solo hilo, podrías no necesitar mutex en absoluto.