Escrevendo uma classe GEOM

Resumo

Este texto documenta alguns pontos de partida no desenvolvimento de classes GEOM e módulos do kernel em geral. Supõe-se que o leitor esteja familiarizado com a programação C do userland.

Índice

1. Introdução

1.1. Documentação

A documentação sobre programação do kernel é escassa - é uma das poucas áreas na qual não há quase nada de tutoriais amigáveis, e a frase "usa a fonte!" realmente é verdadeira. No entanto, existem alguns pedaços (alguns deles seriamente desatualizados) flutuando por ai e que devem ser estudados antes de começar a codificar:

- O Manual do Desenvolvedor do FreeBSD - parte do projeto de documentação, ele não contém nenhum informação específica para a programação do kernel, mas possui algumas informações gerais úteis.
- O Manual de Arquitetura do FreeBSD - também do projeto de documentação, contém descrições de várias instalações e procedimentos de baixo nível. O capítulo mais importante é o 13, Escrevendo drivers de dispositivo FreeBSD.
- A seção Blueprints do site do FreeBSD Diary contém vários artigos interessantes sobre os recursos do kernel.
- As páginas de manual na seção 9 - para documentação importante sobre as funções do kernel.
- A página man geom(4) e os Slides sobre o GEOM de PHK - para uma introdução geral do subsistema GEOM.
- Páginas de manual g_bio(9), g_event(9), g_data(9), g_geom(9), g_provider(9) g_consumer(9), g_access(9) & outros ligados a partir deles, para documentação sobre funcionalidades específicas.
- A página do manual style(9) - para documentação sobre as convenções de estilo de codificação que devem ser seguidas para qualquer código que se destine a ser incorporado à Árvore do Subversion do FreeBSD.

2. Preliminares

A melhor maneira de fazer o desenvolvimento do kernel é ter (pelo menos) dois computadores separados. Um deles conteria o ambiente de desenvolvimento e o código fonte, e o outro seria usado para testar o código recém escrito, inicializando por meio da rede e montando seu sistema de arquivo a partir do primeiro computador. Desta forma, se o novo código contiver erros e travar a máquina, isso não irá atrapalhar o código fonte (e nem nenhum outros dado "vivo"). O segundo sistema nem sequer requer um monitor adequado. Em vez disso, ele pode ser conectado por meio de um cabo serial ou KVM ao primeiro computador.

Mas, como nem todo mundo tem dois ou mais computadores à mão, há algumas coisas que podem ser feitas para preparar um sistema "vivo " para desenvolver código para o kernel. Esta configuração também é aplicável para desenvolvimento em uma máquina virtual criada com o VMWare ou com o QEmu (a próxima melhor coisa depois de uma máquina de desenvolvimento dedicada).

2.1. Modificando um sistema para desenvolvimento

Para qualquer programação do kernel, um kernel com a opção `INVARIANTS` ativada é obrigatório. Então, digite estas linhas no seu arquivo de configuração do kernel:

```
options INVARIANT_SUPPORT
options INVARIANTS
```

Para ter um maior nível de depuração, você também devrá incluir o suporte ao WITNESS, o qual irá alertá-lo sobre erros relacionados a bloqueios (locking):

```
options WITNESS_SUPPORT
options WITNESS
```

Para depurar despejos de memória, é necessário um kernel com símbolos de depuração:

```
  makeoptions    DEBUG=-g
```

Com a maneira usual de instalar o kernel (`make installkernel`) o kernel de depuração não será instalado automaticamente. Ele é chamado de kernel.debug e fica localizado em /usr/obj/usr/src/sys/KERNELNAME/. Por conveniência, deve ser copiado para /boot/kernel/.

Outra conveniência é habilitar o depurador do kernel para que você possa examinar o panic do kernel quando isso acontece. Para isso, insira as seguintes linhas no seu arquivo de configuração do kernel:

```
options KDB
options DDB
```

```
options KDB_TRACE
```

Para que isso funcione, você pode precisar definir um sysctl (se ele não estiver ativado por padrão):

```
  debug.debugger_on_panic=1
```

Kernel panics acontecerão, portanto, deve-se ter cuidado com o cache do sistema de arquivos. Em particular, ter o softupdates habilitado pode significar que a versão mais recente do arquivo pode ser perdida se um panic ocorrer antes de ser committed para armazenamento. Desativar o softupdates produz um grande impacto na performance e ainda não garante a consistência dos dados. A montagem do sistema de arquivos com a opção "sync" é necessária para isso. Para um compromisso, os atrasos do cache de softupdates podem ser encurtados. Existem três sysctl's que são úteis para isso (melhor ser configurado em /etc/sysctl.conf):

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

Os números representam segundos.

Para depurar os panics do kernel, os dumps do núcleo do kernel são necessários. Como um kernel panic pode tornar os sistemas de arquivos inutilizáveis, esse despejo de memória é primeiramente gravado em uma partição bruta. Normalmente, esta é a partição de swap. Essa partição deve ser pelo menos tão grande quanto a RAM física na máquina. Na próxima inicialização, o despejo é copiado para um arquivo normal. Isso acontece depois que os sistemas de arquivos são verificados e montados e antes que o swap seja ativado. Isto é controlado com duas variáveis /etc/rc.conf:

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

A variável `dumpdev` especifica a partição de swap e `dumpdir` informa ao sistema onde no sistema de arquivos ele deverá realocar o dump principal na reinicialização.

A gravação de core dumps é lenta e leva muito tempo, então se você tiver muita memória (>256M) e muitos panics, pode ser frustrante sentar e esperar enquanto isso é feito (duas vezes - primeiro para gravar para o swap, depois para realocá-lo para o sistema de arquivos). É conveniente limitar a quantidade de RAM que o sistema usará através de uma variável do /boot/loader.conf:

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

Se os panics são frequentes e os sistemas de arquivos são grandes (ou você simplesmente não confia em softupdates + background fsck), é aconselhável desligar o fsck em background através da variável /etc/rc.conf:

```
background_fsck="NO"
```

Dessa forma, os sistemas de arquivos sempre serão verificados quando necessário. Observe que, com o fsck em segundo plano, um novo panic pode acontecer enquanto ele está verificando os discos. Novamente, a maneira mais segura é não ter muitos sistemas de arquivos locais, usando o outro computador como um servidor NFS.

2.2. Começando o projeto

Para o propósito de criar uma nova classe GEOM, um subdiretório vazio deve ser criado sob um diretório arbitrário acessível pelo usuário. Você não precisa criar o diretório do módulo em /usr/src.

2.3. O Makefile

É uma boa prática criar Makefiles para cada projeto de codificação não trivial, o que obviamente inclui módulos do kernel.

Criar o Makefile é simples graças a um extenso conjunto de rotinas auxiliares fornecidas pelo sistema. Em suma, aqui está um exemplo de como um Makefile mínimo para um módulo do kernel se parece:

```
SRCS=g_journal.c
KMOD=geom_journal

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

Este Makefile (com nomes de arquivos alterados) serve para qualquer módulo do kernel, e uma classe GEOM pode residir em apenas um módulo do kernel. Se mais de um arquivo for necessário, liste-o na variável `SRCS`, separado com espaço em branco de outros nomes de arquivos.

3. Programação do kernel do FreeBSD

3.1. Alocação de memória

Veja o malloc(9). A alocação básica de memória é apenas ligeiramente diferente do seu userland equivalente. Mais notavelmente, `malloc`() e `free`() aceitam parâmetros adicionais conforme descrito na página do manual.

Um "malloc type" deve ser declarado na seção de declaração de um arquivo fonte, assim:

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

Para usar esta macro, os cabeçalhos sys/param.h, sys/kernel.h e sys/malloc.h devem ser incluídos.

Existe outro mecanismo para alocar memória, o UMA (Universal Memory Allocator). Veja uma(9) para detalhes, mas ele é um tipo especial de alocador usado principalmente para alocação rápida de listas compostas de itens do mesmo tamanho (por exemplo, matrizes dinâmicas de estruturas).

3.2. Listas e filas

Veja queue(3). Há MUITOS casos quando uma lista de coisas precisa ser mantida. Felizmente, essa estrutura de dados é implementada (de várias maneiras) por macros C incluídas no sistema. O tipo de lista mais usado é o TAILQ, porque é o mais flexível. É também aquele com os maiores requisitos de memória (seus elementos são duplamente vinculados) e também o mais lento (embora a variação de velocidade seja mais da ordem de várias instruções da CPU, portanto, ela não deve ser levada a sério).

Se a velocidade de recuperação de dados for muito importante, veja tree(3) e hashinit(9).

3.3. BIOS

A estrutura `bio` é usada para todas e quaisquer operações de Input/Output relativas ao GEOM. Ele basicamente contém informações sobre qual dispositivo ('provedor') deve satisfazer a solicitação, tipo de pedido, offset, comprimento, ponteiro para um buffer e um monte de sinalizadores "específicos do usuário" e campos que podem ajudar a implementar vários hacks.

O importante aqui é que os `bio`-s são tratados de forma assíncrona. Isso significa que, na maior parte do código, não há nenhum análogo as chamadas read(2) e write(2) que não retornam até que uma solicitação seja feita. Em vez disso, uma função fornecida pelo desenvolvedor é chamada como uma notificação quando a solicitação é concluída (ou resulta em erro).

O modelo de programação assíncrona (também chamado de "orientado a eventos") é um pouco mais difícil do que o imperativo muito mais usado no userland (pelo menos leva um tempo para se acostumar com isso). Em alguns casos, as rotinas auxiliares `g_write_data`() e `g_read_data`() podem ser usadas, mas *nem sempre*. Em particular, elas não podem ser usadas quando um mutex é mantido; por exemplo, o mutex de topologia GEOM ou o mutex interno mantido durante as funções `.start`() e `.stop`().

4. Programação GEOM

4.1. Ggate

Se o desempenho máximo não for necessário, uma maneira muito mais simples de fazer uma transformação de dados é implementá-lo na área do usuário por meio do recurso ggate (GEOM gate). Infelizmente, não existe uma maneira fácil de converter ou até mesmo compartilhar código entre as duas abordagens.

4.2. Classe GEOM

Classes GEOM são transformações nos dados. Essas transformações podem ser combinadas em uma

forma de árvore. Instâncias de classes GEOM são chamadas de *geoms*.

Cada classe GEOM possui vários "métodos de classe" que são chamados quando não há nenhuma instância geom disponível (ou simplesmente não estão vinculados a uma única instância):

- `.init` é chamada quando o GEOM toma conhecimento de uma classe GEOM (quando o módulo do kernel é carregado).
- `.fini` é chamada quando o GEOM abandona a classe (quando o módulo é descarregado)
- `.taste` é chamada next, uma vez para cada provedor que o sistema tiver disponível. Se aplicável, essa função geralmente criará e iniciará uma instância geom.
- `.destroy_geom` é chamada quando o geom deve ser desfeito
- `.ctlconf` é chamado quando o usuário solicita a reconfiguração do geom existente

Também são definidas as funções de evento GEOM, que serão copiadas para a instância geom.

O campo `.geom` na estrutura `g_class` é uma LISTA de geoms instanciados a partir da classe.

Estas funções são chamadas a partir da thread g_event do kernel.

4.3. Softc

O nome "softc" é um termo legado para "dados privados do driver". O nome provavelmente vem do termo arcaico "bloco de controle de software". No GEOM, ele é uma estrutura (mais precisamente: ponteiro para uma estrutura) que pode ser anexada a uma instância geom para armazenar quaisquer dados que sejam privados para a instância geom. A maioria das classes GEOM possui os seguintes membros:

- `struct g_provider *provider` : O "provedor" que este geom instância
- `uint16_t n_disks` : Número de consumidores que este geom consome
- `struct g_consumer **disks`: Array de `struct g_consumer*`. (Não é possível usar apenas uma única via indireta porque o struct g_consumer* é criado em nosso nome pela GEOM).

A estrutura `softc` contém todo o estado da instância geom. Cada instância geom possui seu próprio softc.

4.4. Metadados

O formato dos metadados é mais ou menos dependente da classe, mas DEVE começar com:

- Buffer de 16 bytes para uma assinatura de terminação nula (geralmente o nome da classe)
- ID da versão uint32

Assume-se que as classes geom sabem como lidar com metadados com ID de versão menores que os deles.

Os metadados estão localizados no último setor do provedor (e, portanto, devem caber nele).

(Tudo isso depende da implementação, mas todo o código existente funciona assim, e é suportado por bibliotecas.)

4.5. Rotulando/criando um GEOM

A sequência de eventos é:

- o usuário chama o utilitário geom(8) (ou um de seus equivalentes hardlinked)
- o utilitário descobre qual classe geom ele é suposto manipular e procura pela biblioteca geom_CLASSNAME.so (geralmente em /lib/geom).
- ele dlopen(3)-s a biblioteca, extrai as definições dos parâmetros da linha de comandos e funções auxiliares.

No caso da criação/rotulação de um novo geom, isso é o que acontece:

- O geom(8) procura no argumento da linha de comando pelo comando (geralmente `label`) e chama uma função auxiliar .
- A função auxiliar verifica parâmetros e reúne metadados, que são gravados em todos os provedores envolvidos.
- Este "estraga" geoms existentes (se existirem) e inicializa uma nova rodada de "degustação" dos provedores. A classe geom pretendida reconhece os metadados e carrega o geom.

(A sequência de eventos acima é dependente da implementação, mas todo o código existente funciona assim, e é suportado pelas bibliotecas.)

4.6. Estrutura do Comando GEOM

A biblioteca helper geom_CLASSNAME.so exporta a estrutura `class_commands`, que é uma matriz dos elementos `struct g_command`. Os comandos são uniformes no formato e se parecem com:

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

Verbos comuns são:

- label - para gravar metadados em dispositivos para que eles possam ser reconhecidos em degustações e criados em geoms
- destroy - para destruir metadados, para que as geoms sejam destruídas

Opções comuns são:

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

Muitas ações, como rotular e destruir metadados, podem ser executadas no userland. Para isso, `struct g_command` fornece o campo `gc_func` que pode ser definido para uma função (no mesmo .so) que será chamada para processar um verbo. Se `gc_func` for NULL, o comando será passado para o

módulo do kernel, para a função `.ctlreq` da classe geom.

4.7. Geoms

Geoms são instâncias de classes GEOM. Eles possuem dados internos (uma estrutura softc) e algumas funções com as quais eles respondem a eventos externos.

As funções de evento são:

- `.acess`: calcula permissões (leitura / escrita / exclusiva)
- `.dumpconf`: retorna informações formatadas em XML sobre o geom
- `.orphan`: chamado quando algum provedor subjacente é desconectado
- `.spoiled`: chamado quando algum provedor subjacente é gravado
- `.start`: lida com I/O

Estas funções são chamadas a partir da thread `g_down` do kernel e não pode haver sleeping neste contexto, (veja a definição de sleeping em outro lugar) o que limita um pouco o que pode ser feito, mas força o manuseio a ser rápido .

Destes, a função mais importante para fazer o trabalho útil real é a função `.start`(), que é chamada quando uma requisição BIO chega para um provedor gerenciado por uma instância da classe geom.

4.8. Threads GEOM

Existem três threads de kernel criados e executados pelo framework GEOM:

- `g_down` : trata de solicitações provenientes de entidades de alto nível (como uma solicitação do userland) no caminho para dispositivos físicos
- `g_up` : lida com respostas de drivers de dispositivos para solicitações feitas por entidades de nível superior
- `g_event` : lida com todos os outros casos: criação de instâncias geom, contagem de acessos, eventos "spoil", etc.

Quando um processo do usuário emite um pedido de "leitura de dados X no deslocamento Y de um arquivo", isto é o que acontece:

- O sistema de arquivos converte o pedido em uma instância struct bio e o transmite para o subsistema GEOM. Ele sabe o que a instância geom deve manipular porque os sistemas de arquivos são hospedados diretamente em uma instância geom.
- A requisição termina como uma chamada para a função `.start`() feita para a thread g_down e atinge a instância geom de nível superior.
- Essa instância geom de nível superior (por exemplo, o segmentador de partições) determina que a solicitação deve ser roteada para uma instância de nível inferior (por exemplo, o driver de disco). Ele faz uma cópia da solicitação bio (solicitações bio *SEMPRE* precisam ser copiadas entre instâncias, com `g_clone_bio`()!), modifica os campos de dados offset e de provedor de destino e executa a cópia com `g_io_request`()

- O driver de disco obtém a solicitação bio também como uma chamada para `.start`() na thread `g_down`. Ela fala com o hardware, recupera os dados e chama `g_io_deliver`() na bio.
- Agora, a notificação de bio conclusão "borbulha" na thread `g_up`. Primeiro, o slicer de partição obtém `.done`() chamado na thread `g_up`, ele usa as informações armazenadas na bio para liberar a estrutura `bio` clonada (com `g_destroy_bio`()) e chama `g_io_deliver`() no pedido original.
- O sistema de arquivos obtém os dados e os transfere para o usuário.

Veja a página de manual para o g_bio(9) para obter informações sobre como os dados são passados para frente e para trás na estrutura `bio` (observe em particular os campos `bio_parent` e `bio_children` e como eles são manipulados).

Uma característica importante: *NAS THREADS G_UP E G_DOWN NÃO SE PODE DORMIR (SELEEPING)*. Isso significa que nenhuma das seguintes coisas pode ser feita nessas threads (a lista não é completa, mas apenas informativa):

- Chamadas para `msleep`() e `tsleep`(), obviamente.
- Chamadas para `g_write_data`() e `g_read_data`(), porque estes dormem entre passar os dados para os consumidores e retornar.
- Esperando I/O.
- Chamadas para malloc(9) e `uma_zalloc`() com o conjunto de flags `M_WAITOK`
- sx e outros sleepable locks

Esta restrição está aqui para impedir que o código GEOM obstrua o caminho da solicitação de I/O, já que sleeping normalmente não é limitado pelo tempo e não pode haver garantias sobre quanto tempo levará (também existem algumas outras razões mais técnicas). Isso também significa que não existe muito o que possa ser feito nessas threads; por exemplo, quase qualquer coisa complexa requer alocação de memória. Felizmente, existe uma saída: criar threads adicionais no kernel.

4.9. Threads de kernel para uso no código GEOM

As threads do kernel são criadas com a função kthread_create(9), e elas são semelhantes aos threads do userland no comportamento, eles somente não podem retornar ao chamador para exprimir a conclusão, mas deve chamar kthread_exit(9).

No código GEOM, o uso usual de threads é para descarregar o processamento de requisições da thread `g_down` (a função `.start`). Estas threads se parecem com um "event handlers": elas têm uma lista encadeada de eventos associados a elas (nos quais eventos podem ser postados por várias funções em várias threads, portanto, devem ser protegidos por um mutex), pegam os eventos da lista, um por um, e processa-os em uma grande instrução `switch`().

A principal vantagem de usar uma thread para lidar com solicitações de I/O é que ela pode dormir quando necessário. Agora, isso parece bom, mas deve ser cuidadosamente pensado. Dormir é bom e muito conveniente, mas pode ser muito efetivo em destruir o desempenho da transformação geom. As classes extremamente sensíveis ao desempenho provavelmente devem fazer todo o trabalho na chamada de função `.start`(), tomando muito cuidado para lidar com erros de falta de memória e similares.

O outro benefício de ter uma thread de manipulação de eventos como essa é serializar todas as solicitações e respostas provenientes de diferentes threads geom em uma thread. Isso também é muito conveniente, mas pode ser lento. Na maioria dos casos, o tratamento de pedidos `.done`() pode ser deixado para a thread `g_up`.

Mutexes no kernel do FreeBSD (veja mutex(9)) têm uma distinção de seus primos mais comuns do userland - o código não pode dormir enquanto estiver segurando um mutex). Se o código precisar dormir muito, os bloqueios sx(9) podem ser mais apropriados. Por outro lado, se você faz quase tudo em um único thread, você pode se safar sem utilizar mutexes.