Escrevendo uma classe GEOM

Resumo

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

Índice

1. Introdução

1.1. Documentação

A documentação sobre programação de kernel é escassa - é uma das poucas áreas em que quase não há tutoriais amigáveis, e a frase "use o código fonte!" realmente é verdadeira. No entanto, existem alguns fragmentos (alguns deles seriamente desatualizados) circulando que devem ser estudados antes de começar a codificar:

- O Handbook do Desenvolvedor do FreeBSD - parte do projeto de documentação, não contém nada específico sobre programação do kernel, mas sim algumas informações úteis em geral.
- O Handbook 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 dispositivos FreeBSD.
- A seção Blueprints do site http://www.freebsddiary.org contém vários artigos interessantes sobre as facilidades do kernel.
- As páginas de manual da seção 9 - para documentação importante sobre as funções do kernel.
- A página de manual geom(4) e os slides sobre o GEOM do PHK em http://phk.freebsd.dk/pubs/ - para uma introdução geral ao subsistema GEOM.
- As páginas de manual g_bio(9), g_event(9), g_data(9), g_geom(9), g_provider(9), g_consumer(9), g_access(9) e outras vinculadas a elas, para documentação sobre funcionalidades específicas.
- A página de manual style(9) - para documentação sobre as convenções de estilo de codificação que devem ser seguidas para qualquer código que seja commitado para a árvore do FreeBSD.

2. Preliminares

A melhor maneira de desenvolver para o kernel é ter (pelo menos) dois computadores separados. Um deles conteria o ambiente de desenvolvimento e as fontes, e o outro seria usado para testar o código recém-escrito por meio do boot e montagem de sistemas de arquivos por rede do primeiro. Dessa forma, se o novo código contiver erros e travar a máquina, ele não afetará as fontes (e outros dados "ao vivo"). O segundo sistema nem mesmo precisa de uma tela adequada. Em vez disso, ele pode ser conectado com um cabo serial ou KVM para o primeiro.

No entanto, como nem todo mundo tem dois ou mais computadores disponíveis, existem algumas coisas que podem ser feitas para preparar um sistema "ao vivo" para o desenvolvimento de código do kernel. Essa configuração também é aplicável para desenvolvimento em uma máquina virtual VMWare ou QEmu (a próxima melhor opção depois de uma máquina de desenvolvimento dedicada).

2.1. Modificando um sistema para desenvolvimento

Para qualquer programação de kernel, é essencial ter um kernel com `INVARIANTS` ativado. Portanto, adicione as seguintes opções no arquivo de configuração do kernel:

```
options INVARIANT_SUPPORT
options INVARIANTS
```

Para obter mais depuração, você também deve incluir o suporte a WITNESS, que alertará sobre erros de bloqueio:

```
options WITNESS_SUPPORT
options WITNESS
```

Para depurar despejos de falhas (crash dumps), 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/NOME_DO_KERNEL/. Por conveniência, ele deve ser copiado para /boot/kernel/.

Outra conveniência é habilitar o depurador do kernel para que você possa examinar um kernel panic quando ele ocorrer. Para isso, adicione as seguintes linhas no 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
```

Panics do kernel podem acontecer, portanto, é preciso ter cuidado com o cache do sistema de arquivos. Em particular, ter softupdates pode significar que a versão mais recente de um arquivo pode ser perdida se ocorrer um panic antes que seja gravada no armazenamento. Desabilitar o softupdates implica em uma grande perda de desempenho e ainda não garante a consistência dos dados. É necessário montar o sistema de arquivos com a opção "sync" para garantir isso. Como um compromisso, os atrasos do cache do softupdates podem ser encurtados. Existem três sysctl que são úteis para isso (melhor configurados em /etc/sysctl.conf):

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

Os números representam segundos.

Para depurar panics do kernel, são necessários os despejos de núcleo do kernel. Como um kernel panic pode tornar os sistemas de arquivos inutilizáveis, este despejo de falhas é primeiro gravado em uma partição raw. Geralmente, isso é feito na partição de swap. Esta partição deve ter pelo menos o tamanho da RAM física da máquina. No próximo boot, o despejo é copiado para um arquivo regular. Isso acontece após a verificação e montagem dos sistemas de arquivos e antes que o swap seja ativado. Isso é controlado com duas variáveis do arquivo /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 realocar o core dump no reboot.

Gravar o core dump do kernel é lento e leva muito tempo, portanto, se você tiver muita memória (>256 MB) e muitos panics, pode ser frustrante esperar enquanto isso é feito (duas vezes - primeiro para gravá-lo na troca, depois para realocá-lo no sistema de arquivos). É conveniente, portanto, limitar a quantidade de RAM que o sistema usará por meio de um ajuste no /boot/loader.conf:

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

Se os panics forem frequentes e os sistemas de arquivos forem grandes (ou se você simplesmente não confiar no softupdates + fsck em segundo plano), é aconselhável desativar o fsck em segundo plano por meio da seguinte variável no arquivo /etc/rc.conf:

```
background_fsck="NO"
```

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

2.2. Começando o projeto

Para criar uma nova classe GEOM, um subdiretório vazio deve ser criado em 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 arquivos Makefile para todos os projetos de codificação não triviais, o que inclui, é claro, módulos do kernel.

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

```
SRCS=g_journal.c
KMOD=geom_journal

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

Este Makefile (com nomes de arquivo 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-os na variável `SRCS`, separados por espaço de outros nomes de arquivo.

3. Programação do kernel do FreeBSD

3.1. Alocação de memória

Consulte malloc(9). A alocação básica de memória é apenas ligeiramente diferente da sua equivalente no espaço do usuário. 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 de código fonte, por exemplo desta forma:

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

Para usar essa 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). Consulte uma(9) para obter detalhes, mas é um tipo especial de alocador usado principalmente para alocação rápida de listas compostas por itens do mesmo tamanho (por exemplo, matrizes dinâmicas de estruturas).

3.2. Listas e filas

Consulte queue(3). Existem MUITOS casos em que uma lista de coisas precisa ser mantida. Felizmente, essa estrutura de dados é implementada (de várias maneiras) por macros em C incluídas no sistema. O tipo de lista mais usado é TAILQ porque é o mais flexível. Também é o que tem os maiores requisitos de memória (seus elementos são duplamente vinculados) e também o mais lento (embora a variação de velocidade seja da ordem de algumas instruções de CPU a mais, então não deve ser levado a sério).

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

3.3. BIOS

A estrutura `bio` é usada para todas e quaisquer operações de entrada/saída relacionadas ao GEOM. Basicamente, ela contém informações sobre qual dispositivo ('provider') deve satisfazer a solicitação, tipo de solicitação, deslocamento, comprimento, ponteiro para um buffer e um conjunto de flags e campos "específicos do usuário" que podem ajudar a implementar vários ajustes.

O importante aqui é que os `bio`s são manipulados de forma assíncrona. Isso significa que, na maioria das partes do código, não há um análogo das chamadas read(2) e write(2) do espaço do usuário que não retornam até que uma solicitação seja concluída. 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 assíncrono (também chamado de "event-driven") é um pouco mais difícil do que o muito usado modelo imperativo usado no espaço do usuário (pelo menos leva um tempo para se acostumar). 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

As classes GEOM são transformações nos dados. Essas transformações podem ser combinadas de maneira semelhante a uma árvore. As instâncias de classes GEOM são chamadas de *geoms*.

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

- O `.init` é chamado quando o GEOM toma conhecimento de uma classe GEOM (quando o módulo do kernel é carregado.)
- O `.fini` é chamado quando o GEOM abandona a classe (quando o módulo é descarregado)
- O `.taste` é chamado em seguida, uma vez para cada provider que o sistema tem disponível. Se aplicável, esta função geralmente criará e iniciará uma instância geom.
- O `.destroy_geom` é chamado quando o geom deve ser desmontado
- O `.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, é uma estrutura (mais precisamente, um ponteiro para uma estrutura) que pode ser anexada a uma instância geom para manter quaisquer dados que sejam privados à instância geom. A maioria das classes GEOM tem os seguintes membros:

- `struct g_provider *provider`: Instância geom criada a partir do provider correspondente
- `uint16_t n_disks`: Número de consumidores que esta instância geom consome
- `struct g_consumer **disks`: Array de `struct g_consumer*`. (Não é possível usar apenas uma única indireção porque os `struct g_consumer*` são criados em nosso nome pelo GEOM).

A estrutura `softc` contém todo o estado da instância geom. Cada instância geom tem sua própria estrutura `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 comandos alternativos para o mesmo utilitário)
- O utilitário determina qual classe geom ele deve manipular e procura pela biblioteca geom_CLASSNAME.so (geralmente em /lib/geom).
- O utilitário utiliza a função dlopen(3) para carregar dinamicamente a biblioteca, extrair as definições dos parâmetros de linha de comando e funções auxiliares.

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

- O comando geom(8) procura na linha de comando pelo comando (geralmente `label`) e chama uma função auxiliar correspondente.
- A função auxiliar verifica parâmetros e reúne metadados, que são gravados em todos os provedores envolvidos.
- Isso "anula" os geoms existentes (se houver) e inicializa uma nova rodada de "degustação" dos providers. A classe geom pretendida reconhece os metadados e coloca o geom em funcionamento.

(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 auxiliar geom_CLASSNAME.so exporta a estrutura `class_commands`, que é um array de elementos `struct g_command`. Os comandos têm um formato uniforme e se parecem com:

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

Verbos comuns são:

- label - para escrever metadados nos dispositivos para que possam ser reconhecidos durante o processo de "tasting" e trazidos à tona em geoms
- destroy - para destruir metadados, fazendo com que os geoms sejam destruídos

Opções comuns são:

- `-v` : ser verboso (mostrar mais informações)

- `-f` : forçar

Muitas ações, como rotular e destruir metadados, podem ser executadas no espaço de usuário. Para isso, `struct g_command` fornece o campo `gc_func`, que pode ser definido como uma função (no mesmo arquivo .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

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

As funções de evento são:

- `.access` : calcula as permissões (leitura/escrita/exclusiva)
- `.dumpconf` : uma função que retorna informações formatadas em XML sobre o geom
- `.orphan` : chamado quando algum provedor subjacente é desconectado
- `.spoiled` : chamado quando algum provedor subjacente é escrito
- `.start` : lida com operações de entrada/saída (I/O)

Essas funções são chamadas a partir da thread do kernel `g_down` e não é permitido dormir nesse contexto (consulte a definição de dormir em outro lugar), o que limita bastante o que pode ser feito, mas força o tratamento a ser rápido.

A função mais importante para realizar trabalho útil é a função `.start()`, que é chamada quando uma solicitação BIO chega para um provider gerenciado por uma instância de classe geom.

4.8. Threads GEOM

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

- `g_down` : responsável por lidar com solicitações vindas de entidades de alto nível (como uma solicitação do espaço do usuário) a caminho de dispositivos físicos
- `g_up` : Lida com as respostas dos drivers de dispositivo às solicitações feitas por entidades de nível superior
- `g_event` : lida com todos os outros casos: criação de instâncias de geom, contagem de acesso, eventos de "spoil", etc.

Quando um processo do usuário emite uma solicitação para "ler dados X no deslocamento Y de um arquivo", o seguinte 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.

- Esta instância geom de nível superior (por exemplo, o "partition slicer") determina que a solicitação deve ser encaminhada para uma instância de nível inferior (por exemplo, o driver de disco). Ela faz uma cópia da solicitação bio (solicitações bio PRECISAM SEMPRE ser copiadas entre instâncias, com `g_clone_bio`()!), modifica o deslocamento dos dados e os campos do provider de destino e executa a cópia com `g_io_request`()
- O driver de disco também recebe a requisição bio como uma chamada para `.start`() na thread `g_down`. Ele conversa com o hardware, recebe os dados de volta e chama `g_io_deliver`() na bio.
- Agora, a notificação da conclusão do bio "sobe" na thread `g_up`. Primeiro, o particionador recebe `.done`() chamado na thread `g_up`, usa as informações armazenadas no bio para liberar a estrutura de `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.

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

Uma característica importante é que *NÃO PODEM HAVER CHAMADAS DE FUNÇÃO QUE BLOQUEIEM O PROCESSO (DURMAM) NAS THREADS G_UP E G_DOWN*. Isso significa que nenhuma das seguintes coisas pode ser feita nesses threads (a lista é apenas informativa e não completa):

- Chamadas para `msleep`() e `tsleep`(), obviamente.
- Chamadas para `g_write_data`() e `g_read_data()`, pois elas dormem entre a passagem dos dados para os consumidores e o retorno.
- Aguardando I/O.
- Chamadas a malloc(9) e `uma_zalloc`() com a flag `M_WAITOK` definida
- sx e outros tipos de bloqueios sleepable

Essa restrição foi imposta para evitar que o código GEOM obstrua o caminho de solicitação de E/S, já que a espera geralmente não está relacionada ao tempo e não há garantias sobre quanto tempo levará (há outras razões técnicas também). Isso também significa que não há muito o que se possa fazer nessas threads; por exemplo, quase qualquer coisa complexa requer alocação de memória. Felizmente, há uma saída: criar threads adicionais do kernel.

4.9. Threads de kernel para uso no código GEOM

Threads do Kernel são criados com a função kthread_create(9), e eles são parecidos com threads de espaço de usuário em termos de comportamento, apenas que não podem retornar ao chamador para indicar término, mas devem chamar kthread_exit(9).

No código do GEOM, o uso usual de threads é para descarregar o processamento de solicitações da thread `g_down` (a função `.start()`). Essas threads se parecem com "manipuladores de eventos": elas têm uma lista vinculada de eventos associados a elas (na qual eventos podem ser postados por várias funções em várias threads, então ela deve ser protegida por um mutex), pegam os eventos da lista um por um e os processam em uma grande declaração `switch()`.

O principal benefício de usar uma thread para lidar com as solicitações de E/S é que ela pode

dormir quando necessário. Agora, isso parece bom, mas deve ser cuidadosamente pensado. Dormir é bem conveniente, mas pode destruir efetivamente o desempenho da transformação geom. As classes extremamente sensíveis ao desempenho provavelmente devem fazer todo o trabalho na chamada de função `.start()`, tendo muito cuidado para lidar com erros de falta de memória e similares.

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

Mutexes no kernel do FreeBSD (veja mutex(9)) possuem uma distinção em relação às suas contrapartes mais comuns no userland - o código não pode dormir enquanto segura um mutex. Se o código precisa dormir muito, as travas sx(9) podem ser mais apropriadas. Por outro lado, se você fizer quase tudo em um único thread, pode se livrar completamente do uso de mutexes.