Tutorial

Criação de um módulo de *kernel* no Linux

O que é um módulo de *kernel*?

Módulos são peças de código que podem ser carregadas para o *kernel*, e também descarregadas dele, sob demanda. Eles estendem as funcionalidades do *kernel* sem a necessidade de reiniciar o sistema. Assim, por exemplo, quando um novo dispositivo é ligado, seu *driver* pode ser carregado como um módulo para o *kernel*, de modo que este possa acessar tal dispositivo.

Sem os módulos teríamos que refazer o *kernel*, monolítico, e reiniciar o sistema, sempre que fosse necessário incorporar novas funcionalidades, além da desvantagem de tornar o *kernel* maior.

O *kernel* Linux é bastante diferente do espaço do usuário: muitas abstrações são abandonadas e você tem que tomar cuidado extra, já que um *bug* no seu código afeta o sistema inteiro. Não há maneira fácil de se realizar cálculos em ponto flutuante, a pilha é fixa e pequena, o código que você escreve é sempre assíncrono de forma que você precisa pensar na concorrência. A despeito de todos esses pontos, o *kernel* Linux é apenas um programa em C muito grande e complexo que é aberto para qualquer um ler, aprender e melhorar.

Provavelmente a forma mais fácil de iniciar na programação de *kernel* seja escrever um módulo. Há limites para o que os módulos podem fazer – por exemplo, eles não podem adicionar ou remover campos em estruturas de dados comuns como descritores de processos. Mas, de qualquer forma, um módulo é código em nível de *kernel* de fato e pode sempre ser compilado no *kernel*, (removendo, assim, todas as restrições) se necessário.

Quando não escrever um módulo de *kernel*

A programação do *kernel* é interessante, mas escrever código de *kernel* em projetos do mundo real requer certas habilidades. Em geral, você deve descer ao nível do *kernel* apenas se não houver nenhuma outra forma de resolver seu problema. Há chances de você conseguir resolvê-lo em espaço de usuário se:

- ✔ Você estiver desenvolvendo um *driver* USB – dê uma olhada em [3]
- ✔ Você estiver desenvolvendo um sistema de arquivos – pode tentar [4]
- ✔ Você está estendendo Netfilter – [5] pode ajudar

Em geral, código nativo do *kernel* terá desempenho melhor, mas para muitos projetos esta perda de desempenho não é crucial.

Recomendações antes de começar

Antes de começar a programar há algumas questões que devemos abordar.

1. Um módulo compilado para um *kernel* não vai carregar se você inicia o sistema com uma versão de *kernel* diferente, a menos que habilite `CONFIG_MODVERSIONS`, no *kernel*.
2. É altamente recomendável que você digite, compile e carregue os exemplos deste tutorial em um console. Você não deve trabalhar com eles no ambiente gráfico X. Módulos não podem imprimir na tela como `printf()`, mas eles podem armazenar (*log*) informações e alertas (*warnings*), que até podem ser impressos na tela, mas apensa no console.
3. Frequentemente, distribuições Linux distribuem fontes do *kernel* que foram alterados

(*patched*) de várias formas fora do padrão. Isto pode causar problemas. E ainda, algumas distribuições entregam *kernel headers* incompletos.

4. Para evitar os problemas expostos no ítem anterior, recomenda-se baixar, compilar e reiniciar uma versão nova e intacta do *kernel* (disponível, por exemplo, em *kernel.org*)
5. Um *bug* no seu módulo pode levar a um *system crash* (improvável, mas possível) e perda de dados. Por isso, assegure-se de ter feito cópia de todos os dados importantes antes de começar ou, ainda melhor, experimente em uma máquina virtual.

Começando: um exemplo simples mas completo

Neste tutorial nós vamos desenvolver um módulo de *kernel* simples que cria um dispositivo `/dev/reserve`. Uma *string* escrita neste dispositivo é devolvida com as palavras em ordem inversa (“Hello World” se torna “World Hello”). Por padrão, este dispositivo é disponível apenas para o `root`, de forma que você terá que executar seus programas de teste usando `sudo`.

A anatomia de um módulo

Como a maioria dos módulos do *kernel* Linux são escritos em C (a menos algumas partes de baixo específicas de arquitetura), é recomendável que você mantenha seu módulo em um arquivo único (por exemplo, `reverse.c`). Colocamos o código completo do exemplo em um anexo deste tutorial. Aqui vamos estudar alguns fragmentos dele.

Para começar vamos incluir alguns arquivos de cabeçalho e descrever o uso de macros predefinidas.

```
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Valentine Sinitsyn <valentine.sinitsyn@gmail.com>");
MODULE_DESCRIPTION("In-kernel phrase reverser");
```

Tudo simples até aqui, exceto por MODULE_LICENCE(). Este não é um mero marcador. O *kernel* favorece fortemente código GPL-compatível, de forma que, se você definir a licença como algo não GPL-compatível, digamos “Proprietary”, certas funções do *kernel* não estarão disponíveis para seu módulo.

Como programação de *kernel* é sempre assíncrona, não há nenhuma função main() que o Linux executa sequencialmente para seu código. Ao invés disso, seu código provê *callbacks* para vários eventos, como os abaixo.

```
static int __init reverse_init(void) {
      printk(KERN_INFO "reverse device has been registered\n");
      return 0;
}

static void __exit reverse_exit(void) {
      printk(KERN_INFO "reverse device has been unregistered\n");
}

module_init(reverse_init);
module_exit(reverse_exit);
```

Aqui, definimos funções a serem chamadas no instante de inserção do módulo no *kernel*

(comando insmod) e no evento da sua remoção (comando rmmod). Apenas a primeira é necessária. Para o momento, elas simplesmente imprimem, usando printk(), uma mensagem no arquivo `/var/log/messages` (acessível, a partir do espaço do usuário, via comando dmesg); `KERN_INFO` é um nível de *log* (note que não há nenhuma vírgula).

`__init` e `__exit` são atributos que instruem o compilador GCC sobre instruções que estão fora da especificação da linguagem. Muito código é necessário para preparar o *kernel* na inicialização. Muitas vezes, este código nunca mais é usado. Por isso, depois de executadas, o *kernel* limpa essas funções da memória. As mensagens de "Freeing unusing kernel memory: ..." que aparecem ao final da inicialização do sistema (podem ser vistas com dmesg), mostram isso.

Muitos *drivers* são implementados como módulos. Nesse caso, eles "saem" (em algum momento) do *kernel*. No entanto, se eles são compilados dentro do *kernel*, eles não necessariamente "saem". Nesse caso, essa área de código também é dispensável.

Atributos são raros de serem vistos em código C no espaço de usuário mas bem comuns no *kernel*. Finalmente, as macros `module_init()` e `module_exit()` definem as funções `reverse_init()` e `reverse_exit()` como *callbacks* para os eventos de inserção e remoção para nosso módulo. O nome das funções não importa; você pode dar o nome que quiser.

Módulos podem aceitar parâmetros, como por exemplo:

```
#> modprobe foo bar=1
```

Para tornar o exemplo mais interessante vamos fazer nosso módulo aceitar parâmetros.

O comando `modinfo` mostra todos os parâmetros aceitos por um módulo. Estes também são disponíveis em `/sys/module/<modulename>/parameters` como arquivos. Nosso módulo vai precisar de um *buffer* para armazenar frases – vamos, então, permitir que o usuário forneça este parâmetro. Para tal vamos adicionar as seguintes linhas logo após `MODULE_DESCRIPTION()`:

```
static unsigned long buffer_size = 8192;
module_param(buffer_size, ulong, (S_IRUSR | S_IRGRP | S_IROTH));
MODULE_PARM_DESC(buffer_size, "Internal buffer size");
```

Aqui, nós definimos uma variável para armazenar o valor, empacotamos esta em um parâmetro e o fizemos visível por qualquer um via sysfs. Este sistema de arquivos pode ser acessado a partir do diretório /sys, dentro do qual há o diretório module, contendo um subdiretório por módulo carregável. Dentro do subdiretório de um módulo particular há um diretório parameters, dentro do qual se encontram os arquivos representando os parâmetros do módulo. No caso deste exemplo, haverá um arquivo com o nome de buffer_size nesse diretório. É para ele que são definidas as permissões de acesso na macro module_param(). Aqui foram definidas três permissões de leitura com o conector '|' (ou) fazendo com que as três sejam ativadas em conjunto, mas poderia ter sido utilizada a forma numérica 0444, equivalente.

S_IRUSR (R USR = permissão de leitura para proprietário)
S_IRGRP (R GRP = permissão de leitura para o grupo)
S_IROTH (R OTH = permissão de leitura para outros)

A descrição do parâmetro, na última linha, aparece na saída do comando `modinfo`. Como o usuário pode definir `buffer_size` diretamente, precisamos validar o valor passado por ele na função `reverse_init()`. Sempre se deve checar dados provenientes de fora do *kernel* – se você não o fizer, estará admitindo a possibilidade de *kernel panic* ou, ainda, brechas de segurança. Veja código abaixo. Valores diferentes de zero retornados por um módulo indicam uma falha.

```
static int __init reverse_init()
{
      if (!buffer_size)
            return -1;
      printk(KERN_INFO
            "reverse device has been registered, buffer size is %lu bytesn",
            buffer_size);
      return 0;
}
```

Agora, é hora de compilar o módulo. Você precisará dos arquivos de cabeçalho para a versão de *kernel* que você está rodando (`linux-headers` ou pacotes equivalentes) e `build-essential` (ou análogo). Depois, deverá ser criado um `Makefile` típico, como o que segue:

```
obj-m += reverse.o
all:
      make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
      make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Execute o comando `make` para compilar seu primeiro módulo. Se digitou tudo corretamente, você encontrará `reverse.ko` no diretório corrente. Insira-o no *kernel* utilizando o comando:

```
sudo insmod reverse.ko
```

Depois, verifique a mensagem de log gerada:

```
$ dmesg | tail -1
[ 5905.042081] reverse device has been registered, buffer size is 8192 bytes
```

No entanto, esta mensagem está mentirosa por enquanto – ainda não há de fato um *device node* no sistema. Vamos consertar isso, então.

O dispositivo `misc`

No Linux, há um tipo de dispositivo de caracteres especial chamado “miscellaneous” (ou simplesmente “misc”). Ele é projetado para pequenos *drivers* de dispositivos com um único ponto de entrada, exatamente o que precisamos. Todo dispositivo `misc` compartilha o mesmo *major number* (10), de forma que um *driver* (`drivers/char/misc.c`) pode cuidar de todos eles, distinguindo-os por seus *minor numbers*. No mais, eles são apenas dispositivos normais de caracteres.

Para registrar um *minor number* (e um ponto de entrada) para o dispositivo, você pode declarar `struct misc_device`, preencher seus campos (note a sintaxe) e chamar `misc_register()` com um ponteiro para esta estrutura. Para isto funcionar, você também precisará incluir arquivo de cabeçalho `linux/miscdevice.h`.

```
static struct miscdevice reverse_misc_device = {
      .minor = MISC_DYNAMIC_MINOR,
      .name = "reverse",
      .fops = &reverse_fops
};
```

```
static int __init reverse_init()
{
	…
	misc_register(&reverse_misc_device);
	printk(KERN_INFO …
}
```

Aqui, nós requeremos o primeiro (dinâmico) *minor number* disponível para o dispositivo chamado “reverse”; as reticências indicam código omitido que já foi visto. É preciso fazer o *unregsiter* do dispositivo quando o módulo for removido.

```
static void __exit reverse_exit(void)
{
	misc_deregister(&reverse_misc_device);
	…
}
```

O campo 'fops' armazena um ponteiro para uma `struct file_operations` (declarada em `linux/fs.h`), e este é o ponto de entrada para nosso módulo. `reverse_fops` é definido como:

```
static struct file_operations reverse_fops = {
	.owner = THIS_MODULE,
	.open = reverse_open,
	...
	.llseek = noop_llseek
};
```

Novamente, `reverse_fops` contém um conjunto de *callbacks* (também conhecidos como métodos) para serem executados quando código do espaço do usuário abre um dispositivo, lê dele, escreve nele ou fecha o descritor de arquivo. Se você omitir qualquer dessas operações, uma *sensible fallback* será usada no lugar. Por isso, definimos explicitamente o método `llseek` para `noop_llseek()`, que (como o nome indica) não faz nada. A implementação padrão altera um ponteiro de arquivo e nós não queremos que nosso dispositivo seja passível dessa operação.

Implementando os métodos `open` e `close`

Vamos começar a implementar os métodos. Nós vamos alocar um novo *buffer* para cada descritor de arquivo aberto, e liberá-lo ao fechar. Isto não é muito seguro: se uma aplicação do espaço do usuário (talvez intencionalmente) disparar um grande volume de descritores, esta pode encher a RAM, e deixar o sistema inutilizável. Você deve sempre pensar sobre estas possibilidades no mundo real. No entanto, para este tutorial, isto é aceitável.

Precisaremos de uma estrutura para descrever o *buffer*. O *kernel* provê muitas estruturas de dados genéricas: listas ligadas, tabelas *hash*, árvores e outras. No entanto, *buffers* são usualmente implementados do zero. Vamos chamar o nosso de `struct buffer`:

```
struct buffer {
	char *data, *end, *read_ptr;
	unsigned long size;
};
```

`data` é um ponteiro para a *string* que este *buffer* armazena e `end` é o primeiro byte depois do fim da *string*. `read_ptr` é onde `read()` deve começar a leitura. O tamanho do *buffer* é armazenado por questão de completude da estrutura – por agora, não usamos este campo. Você não deve assumir que os usuários de sua estrutura vão inicializar todos os campos corretamente. Assim,

é melhor encapsular a alocação e liberação de *buffer* em funções. Estas são usualmente chamadas de `buffer_alloc()` e `buffer_free()`.

```
static struct buffer *buffer_alloc(unsigned long size)
{
       struct buffer *buf;
       buf = kzalloc(sizeof(*buf), GFP_KERNEL);
       if (unlikely(!buf))
            goto out;
       ...
       out:
            return buf;
}
```

A memória do *kernel* é alocada com `kmalloc()` e liberada com `kfree()`; a variação `kzallock()` preenche a memória alocada com zeros. Diferente da função padrão `malloc()`, sua correspondente para o *kernel* recebe *flags* especificando o tipo de memória requerido no segundo argumento. Aqui, `GFP_KERNEL` significa que precisamos de memória normal do *kernel* (não em zonas de DMA ou memória alta) e a função pode dormir (reescalonar o processo) se necessário. `sizeof(*buf)` é a forma comum de se obter o tamanho de uma estrutura acessível via ponteiro.

Você deve sempre verificar o valor de retorno de `kmalloc()`: seguir um ponteiro `NULL` resultará em *kernel panic*. Ainda, note o uso da macro `unlikely()`. Ela (e a macro oposta `likely()`) é bastante usada no *kernel* para significar que a condição é quase sempre verdadeira (ou falsa). Ela não afeta o controle de fluxo, mas ajuda processadores modernos aumentar o desempenho com predições de desvio.

Finalmente, note os `goto`s. Eles são frequentemente considerados ruins, no entanto, o *kernel* Linux (e alguns outros sistemas) o empregam para implementar saída centralizada de função. Isto resulta em menos código com aninhamento profundo e código mais legível.

Com buffer_alloc() e buffer_free() prontas, a implementação dos métodos open e close se torna bem simples.

```
static int reverse_open(struct inode *inode, struct file *file)
{
      int err = 0;
      file->private_data = buffer_alloc(buffer_size);
      ...
      return err;
}
```

`struct file` é a estrutura de dados padrão do *kernel* para armazenar informações sobre um arquivo aberto, como posição atual (`file->f_pos`), *flags* (`file->f_flags`) ou modo de abertura (`file->f_mode`). Um outro campo, `file->private_data` é usado para associar o arquivo com algum dado arbitrário, em nosso caso, o *buffer* que definimos anteriormente.

Se a alocação do *buffer* falha, nós indicamos isto para o código no espaço do usuário retornando um valor negativo (`-ENOMEM`). Uma biblioteca C realizando a chamada de sistema `open(2)` (provavelmente, **glibc**) vai detectar isto e definir a variável global `errno` adequadamente, isto é, para uma mensagem textual sobre o erro ocorrido (ENOMEM: Out of memory). As definições dessas constantes podem ser vistas em errno-base.h.

Implementando os métodos `read` e `write`

Nos métodos **read** e **write** é onde o trabalho real é realizado. Quando dados são escritos

para um *buffer*, nós jogamos fora o conteúdo prévio e invertemos a frase colocada, sem nenhum armazenamento temporário. O método **read** simplesmente copia os dados do *buffer* do *kernel* para o espaço do usuário. Mas o que o método `reverse_read()` deve fazer se não houver nenhum dado no *buffer* ainda? No espaço do usuário, a chamada **read()** bloquearia até que dados estivessem disponíveis. No *kernel*, deve-se esperar. Por sorte, existe um mecanismo para isto. Este é chamado de filas de espera ('*wait queues*').

A ideia é simples. Se o processo atual precisa esperar algum evento, seu descritor (uma `struct task_struct` armazenada como '**current**') é colocada em estado 'não-executável' (dormindo) e é adicionada a uma fila. Então, o escalonador, `schedule()`, é chamado para selecionar outro processo para executar. Um código que gera o evento, usa a fila para acordar os processos que estão dormindo colocando-os de volta no estado `TASK_RUNNING`. O escalonador selecionará um deles em algum momento no futuro. O Linux tem vários estados 'não-executáveis' para processos, mais notadamente `TASK_INTERRUPTIBLE` (um estado de bloqueio que pode ser interrompido com um sinal) e `TASK_KILLABLE` (um processo dormindo que pode ser 'morto'). Tudo isso deve ser manipulado corretamente e as filas de espera fazem isso por você.

Um local natural para o ponteiro de início de fila de espera é a estrutura `struct buffer`. Assim, adicionamos o campo `wait_queue_head_t read_queue` a ela. Você também deverá incluir o arquivo de cabeçalho `linux/sched.h`. Uma fila de espera pode ser declarada estaticamente com a macro `DECLARE_WAITQUEUE()`. No nosso caso, inicialização dinâmica é necessária. Por isso, adicionamos esta linha a `buffer_alloc()`:

```
init_waitqueue_head(&buf->read_queue);
```

Nós esperamos que os dados estejam disponíveis, ou seja, que a condição read_ptr != end se torne verdadeira. Também queremos que a espera seja passível de ser interrompida (digamos, por um Ctrl+C). Assim, o método “read” deve iniciar dessa forma:

```
static ssize_t reverse_read(struct file *file, char __user * out,
                            size_t size, loff_t * off)
{
      struct buffer *buf = file->private_data;
      ssize_t result;
      while (buf->read_ptr == buf->end) {
            if (file->f_flags & O_NONBLOCK) {
                  result = -EAGAIN;
                  goto out;
            }
            if (wait_event_interruptible
            (buf->read_queue, buf->read_ptr != buf->end)) {
                  result = -ERESTARTSYS;
                  goto out;
            }
      }
      ...
```

Nós realizamos o laço até que os dados estejam disponíveis e usamos `wait_event_interruptible()` (é uma macro, não uma função, é por isso que a fila é passada por valor) para esperar enquanto eles não chegam.

`wait_event_interruptible(wq, condition)`

Esta macro coloca o processo para dormir (TASK_INTERRUPTIBLE) até que `condition` se torne verdadeira ou um sinal seja recebido. A condição é testada é checada toda vez que a fila `wq` é 'despertada' (isto acontece com a chamada de `wake_up_interruptible()`,

que aqui se encontra na função `reverse_write()`.
Os valores de retorno são: `-ERESTARTSYS` se for interrompida por um sinal ou `0` se a condição se tornar verdadeira. O código indica que a chamada de sistema (read) deve ser reiniciada.

Se `wait_event_interruptible()` for interrompida, ela retorna um valor diferente de zero, que nós traduzimos para `-ERESTARTSYS`. Este código significa que a chamada de sistema deve ser reiniciada. `file->f_flags` verifica se o arquivo foi aberto com `O_NONBLOCK` ativo, caso em que deve-se terminar a chamada com o erro `EAGAIN`, indicando que uma tentativa de operação que pode bloquear, read(), foi realizada em um arquivo com `O_NONBLOCK` ativo. O código de erro EAGAIN indica que uma tentativa futura pode resultar em sucesso, já que um evento externo (dados que chegam no *buffer*) pode já ter acontecido.

Não podemos usar `if()` ao invés de `while()`, desde que pode haver muitos processos esperando pelos dados. Quando o método **write** acorda-os, o escalonador escolhe um para executar de forma não previsível. Assim, quando for dada a chance desse código executar, o *buffer* pode estar vazio novamente.

Agora precisamos copiar os dados de `buf->data` para o espaço do usuário. A função `copy_to_user()` do *kernel* faz exatamente isto.

`copy_to_user(dst,src,size)`

Obtendo sucesso, esta função copia `size` *bytes* apontados por `src`, que deve estar no espaço do *kernel*, para `dst`, que deve estar no espaço do usuário.
Retorna o número de bytes não copiados. Portanto, se retornar 0, significa sucesso.
Esta não é uma cópia de memória para memória comum. Por isso, não se pode simplesmente utilizar memcpy(), por exemplo. Isto por duas razões. Primeio, o kernel é capaz de escrever em qualquer memória, mas processo de usuário não. `copy_to_user()` precisa checar se `dst` é acessível e se poder ser escrito pelo processo atual. Segundo, dependendo da arquitetura, não se pode simplesmente copiar dados do *kernel* para o espaço do usuário. É necessário, antes, configuração especial ou utilizar operações especiais.
Uma discussão mais detalhada sobre o funcionamento desta função pode ser encontrada em [10].

```
size = min(size, (size_t) (buf->end - buf->read_ptr));
if (copy_to_user(out, buf->read_ptr, size)) {
      result = -EFAULT;
      goto out;
}
```

A chamada pode falhar se o ponteiro para o espaço do usuário estiver errado. Se isto acontece, retornamos `-EFAULT` (*Bad address*).

```
      buf->read_ptr += size;
      result = size;
out:
      return result;
```

Aritmética simples é necessária para que os dados sejam lidos em pedaços arbitrários. O método retorna o número de bytes lidos ou um código de erro.

O método **reverse_write()** é mais simples e mais curto. Primeiro, checamos se o *buffer* tem espaço suficiente, então usamos a função `copy_from_user()` para obter os dados. Então, os ponteiros `read_ptr` e `end` são reiniciados e o conteúdo do *buffer* é invertido.

Os motivos da necessidade de se utilizar a função copy_from_user() são os mesmos explicados acim para copy_to_user(). Além disso, seu funcionamento e valor de retorno são os mesmos, a menos que a cópia ocorre em sentido inverso, claro.

```
buf->end = buf->data + size;
buf->read_ptr = buf->data;
if (buf->end > buf->data)
      reverse_phrase(buf->data, buf->end - 1);
```

Aqui, `reverse_phrase()` faz todo o trabalho de inversão se apoiando na função `reverse_word()`, que é bem curta e marcada como `inline`. Esta é outra otimização comum. No entanto, você não deve usá-la em excesso, desde que isto tornaria a imagem do *kernel* desnecessariamente grande.

Finalmente, precisamos acordar processos esperando pelos dados na fila de espera, como descrito antes. `wake_up_interruptible()` faz isso.

```
wake_up_interruptible(&buf->read_queue);
```

Você agora tem um módulo de *kernel* que, pelo menos, compila com sucesso. Agora é hora de testá-lo. Mas faremos algumas melhorias após o teste.

Compilando o código do módulo

Compile o módulo e carregue-o no *kernel*:

```
$ make
$ sudo insmod reverse.ko buffer_size=2048
$ lsmod
  reverse 2419 0
$ ls -l /dev/reverse
  crw-rw-rw- 1 root root 10, 58 Feb 22 15:53 /dev/reverse
```

Tudo parece estar no lugar. Agora, para testar como o módulo funciona, escreveremos um pequeno programa que inverte seu primeiro argumento de linha de comando. O conteúdo da função `main()` pode ser assim:

```
int fd = open("/dev/reverse", O_RDWR);
write(fd, argv[1], strlen(argv[1]));
read(fd, argv[1], strlen(argv[1]));
printf("Read: %sn", argv[1]);
```

Execute-o assim:

```
$ ./test 'A quick brown fox jumped over the lazy dog'
Read: dog lazy the over jumped fox brown quick A
```

Implementando suporte a concorrência

Para provar a falta de suporte a concorrência, vamos criar dois processos que compartilham

o descritor de arquivo (e portanto, o *buffer* no *kernel*). Um processo irá escrever strings continuamente no dispositivo enquanto o outro fará leituras nele. A chamada de sistema fork() é usada no exemplo abaixo, mas pthreads também funcionaria. Novamente, o código que abre e fecha o dispositivo e faz a checagem de erro foi omitido.

```
char *phrase = "A quick brown fox jumped over the lazy dog";
if (fork())
      /* Parent is the writer */
      while (1)
            write(fd, phrase, len);
else
      /* child is the reader */
      while (1) {
            read(fd, buf, len);
            printf("Read: %sn", buf);
      }
```

O que você espera da saída desse programa? Abaixo, mostramos o que obtivemos no nosso teste.

```
Read: dog lazy the over jumped fox brown quick A
Read: A kcicq brown fox jumped over the lazy dog
Read: A kciuq nworb xor jumped fox brown quick A
Read: A kciuq nworb xor jumped fox brown quick A
...
```

O que aconteceu é o resultado de uma condição de corrida onde não houve controle adequado (nesse caso, nenhum ainda) de exclusão mútua. Pensamos que **read** e **write** são atômicos, ou executados uma instrução por vez do início até o fim. No entanto, o *kernel* pode reescalonar o processo executando a parte *kernel-mode* da operação **write**, isto é, reverse_write() em algum tempo qualquer antes que a função `reverse_phrase()` tenha terminado seu trabalho. Se o processo que faz **read()** é escalonado antes que o processo escritor tenha a chance de terminar, ele verá os dados em estado inconsistente. Tais *bugs* são realmente difíceis de rastrear. Mas como consertar isso?

Basicamente, precisamos garantir que nenhum método **read** seja executado até que o método **write** retorne. Isto é implementado utilizando-se de primitivas de sincronização (*locks*) como *mutexes* ou semáforos, que também existem no *kernel*. Em nosso caso, um simples *mutex* – um objeto que apenas um processo por vez pode possuir – é suficiente.

Travas (*locks*) sempre protegem algum dado (em nosso caso, uma instância "`struct buffer`") e é muito comum declará-lo dentro da estrutura que estão protegendo. Assim, adicionamos um *mutex* (`struct mutex lock`) dentro de "`struct buffer`". Também precisamos inicializar o *mutex* usando `mutex_init()`. A função `buffer_alloc()` é um bom local para isso. O código que usa *mutexes* deve também incluir `linux/mutex.h`.

Um *mutex* funciona como um sinaleiro – é inútil a menos que os *drivers* olhem para ele e sigam os seus sinais. Assim, precisamos atualizar `reverse_read()` e `reverse_write()` para adquirirem o *mutex* antes que façam qualquer coisa no *buffer* e liberarem o *mutex* quando terminarem.

Vejamos como fica o método **read**. O método **write** funciona da mesma forma.

```
static ssize_t reverse_read(struct file *file, char __user * out,
                            size_t size, loff_t * off)
{
      struct buffer *buf = file->private_data;
      ssize_t result;
```

```
      if (mutex_lock_interruptible(&buf->lock)) {
            result = -ERESTARTSYS;
            goto out;
...
}
```

Nós adquirimos o *lock* logo no início da função. `mutex_lock_interruptible()` ou obtém o *mutex* e retorna ou coloca o processo para dormir até que o *mutex* esteja disponível. Como antes, o sufixo `_interruptible` significa que o sono pode ser interrompido com um sinal.

```
      while (buf->read_ptr == buf->end) {
            mutex_unlock(&buf->lock);
            /* ... wait_event_interruptible() here ... */
            if (mutex_lock_interruptible(&buf->lock)) {
                  result = -ERESTARTSYS;
                  goto out;
            }
      }
```

Acima, está nosso laço de espera pelos dados. Nunca se deve deixar um processo dormir enquanto possui um *lock*. Isto pode levar a um *deadlock*. Assim, se não há dados, soltamos o *mutex* e chamamos `wait_event_interruptible()`. Quando esta retorna, readquirimos o *mutex* e continuamos normalmente.

```
if (copy_to_user(out, buf->read_ptr, size)) {
      result = -EFAULT;
      goto out_unlock;
}
...
out_unlock:
      mutex_unlock(&buf->lock);
out:
      return result;
```

Finalmente, o *mutex* é liberado quando a função termina ou se um erro ocorre enquanto o *mutex* está sendo mantido.

Recompile o módulo (não se esqueça de recarregá-lo) e execute o segundo teste novamente. Você não deverá ver nenhum dado corrompido desta vez.

Os conceitos mostrados permanecem os mesmos em cenários mais complexos. Concorrência, tabela de métodos, registro de *callbacks*, colocar processos para dormir e acordá-los são pontos com os quais um programador de *kernel* deve se sentir confortável.

Bibliografia

1. Salzman, Peter J., Burian, M., Pomerantz, O. *The Linux Kernel Module Programming Guide, 2007* Disponível em: http://www.linuxtopia.org/online_books/linux_kernel/linux_kernel_module_programming_2.6/index.html
2. Gregory, A. *Write your first Linux Kernel module*, Linux Voice, 2014. Disponível em: http://www.linuxvoice.com/be-a-kernel-hacker/
3. http://www.libusb.org/
4. http://fuse.sf.net/
5. http://www.linuxvoice.com/be-a-kernel-hacker/www.netfilter.org/projects/libnetfilter_queue

6. Love, R. *Linux kernel Development*, 3ed., Addison-Wesley, 2010
7. *An introduction to Linux kernel programming*, Disponível em: http://crashcourse.ca/ Acessado em: 21/11/2014.
8. *Linux Cross Reference*, Disponível em: http://lxr.free-electrons.com/source/Documentation/
9. Corbet, J., Rubini, A., Kroah-Hartman, G. *Linux Device Drivers*, Disponível em: http://lwn.net/Kernel/LDD3/
10. Love, Robert *Linux Kernel: How does copy_to_user work?* Disponível em: http://www.quora.com/Linux-Kernel/How-does-copy_to_user-work