Linux Device Drivers

Otávio Basso Gomes

Programação de periféricos
Prof. Eduardo Bezerra

Roteiro

- Introdução
- Device drivers no Linux
- Tipos de drivers
- Programando no Kernel Space
- Hello World
- Major & minor numbers
- Pepe, char driver básico

Introdução

•Device drivers são módulos que oferecem comunicação entre o sistema operacional e um determinado periférico.

•O Sistema Operacional através de uma API oferece uma interface para o programador conectar o dispositivo ao usuário.

Device Drivers no Linux

Drivers no linux são como códigos objetos. Posteriomente são linkados ao kernel em tempo de execução.

"*Device driver está provendo mecanismo, não política*"[1].

- O programador do driver não deve pressupor como o dispositivo será usado, apenas disponibilizar o recurso.
- Exemplo: um driver floopy deve apenas mostrar blocos contínuos de dados. Permissões e sistema de arquivos é por conta do sistema.
- Segue a filosofia UNIX.

Tipos de drivers

Os três tipos básicos de drivers oferecidos pelo Kernel:

- Char Devices: funcionamento como de um stream de bytes. Acesso de blocos de dados com tamanho aleatório. O usuário usa o dispositivo através de um arquivo no diretório /dev (Ex. /dev/ttyS0).

- Block Devices: Além de ter uma interface como dos char devices. Também oferece acesso usando blocos de tamanho multiplo de um número determinado pelo driver. Normalmente representa dispositivos que são **MOUNT**ados.

- Network Interfaces: orientado a dispositivos de rede, não possui entrada no diretório /dev. Transmite/Recebe pacotes não bytes de um stream.

Programando no kernel space

- Liberdade de usar qualquer instrução do processador.
- O código deve ser reentrante.
- Sem biblioteca padrão do C (o kernel tem suas proprias rotinas de auxílio).

Hello World (Linux 2.6)

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

MODULE_LICENSE("Dual BSD/GPL");

static int __init hello_init(void)
{
   printk(KERN_INFO "Hello\n");
   return 0;
}

static void __exit hello_exit(void)
{
   printk(KERN_INFO "Bye");
}

module_init(hello_init);
module_exit(hello_exit);
```

Hello World: compilação

Através de Makefile. Para compilar é necessário ter configurações do kernel atual, por isso o “ /lib/modules/$(shell uname -r)/build”

```
obj-m += hello.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
```

Hello World: carregando/descarregando

Usando uma shell digite:

```
$insmod hello.ko
$dmesg
$rmmod hello.ko
$dmesg
```

Arquivo compilado da module

Comando que mostra as mensagens do kernel (para conferir os printk's)

Major & minor numbers

Major e Minor numbers:

- Major number serve como identificador do módulo para o sistema. Alguns são padronizados. O kernel pode alocar um não usado para você.
- Minor number identificador para uso interno do módulo.

Os major numbers usados pelo módulos podem ser vistos pelo arquivo /proc/devices

```
$cat /proc/devices
Character devices:
 ...
189 usb_device
195 nvidia
216 rfcomm
 ...
```

Pepe, char driver básico

A seguir será apresentado o código de um char driver básico. O falso dispositivo “pepe” terá a função leitura e escrita de strings. Exemplo*:

```
$cat /dev/pepe
...
$echo 'Acorda' > /dev/pepe
$cat /dev/pepe
...
$echo 'Pepeeee, jah tirei a vela!' > /dev/pepe
$cat /dev/pepe
Jah vo, jah vo.
```

*os “...” são escritos literalmente.

Pepe, char driver básico: struct file_operations

```
// Arquivo linux/fs.h
struct file_operations {
        ...
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        int (*open) (struct inode *, struct file *);
        int (*release) (struct inode *, struct file *);
        ...
};
```

"A estrutura file_operations oferece ponteiros para funções que o driver usa para implementar a interface do kernel para o driver."[man page]

Pepe, char driver básico: struct file_operations

```
static int pepe_open(struct inode *inode, struct file *file);
static int pepe_release(struct inode *inode, struct file *file);
static ssize_t pepe_read(struct file *filp, char __user *user_buffer,
                         size_t length, loff_t *offset);
static ssize_t pepe_write(struct file *filp, const char __user *buffer,
                          size_t len, loff_t *off);

static struct file_operations fops = {
   .read = pepe_read,
   .write = pepe_write,
   .open = pepe_open,
   .release = pepe_release
};
```

Pepe, char driver básico: registração

```
#include <linux/fs.h>

int register_chrdev(unsigned int major,  const  char*name,
      struct file_operations*ops);

int   unregister_chrdev(unsigned  int  major,  const  char
      *name);
```

Parâmetros:

major — o major number associado ao dispositivo. Na função register_chrdev, se este valor for 0 então o kernel aloca um valor não utilizado.

name — o nome curto para o dispositivo, aparece em /proc/devices.

*ops — a estrutura file_operations preenchida com ponteiros com as funções definidas.

Pepe, char driver básico: registração

```
static int __init pepe_init(void)
{
   major_number = register_chrdev(0, PEPE_DEVICE_NAME, &fops);

   if ( major_number < 0 ) {
      printk(KERN_ALERT "Nao pude registrar major number: %d",
major_number);
      return major_number;
   }

   return 0;
}

static void __exit pepe_exit(void)
{
   int r = unregister_chrdev(major_number, PEPE_DEVICE_NAME);
   if ( r < 0 )
      printk(KERN_ALERT "Não pude desregistrar o device");

}
```

Pepe, char driver básico: open e release

```
static int pepe_open(struct inode *inode, struct file *file)
{
   try_module_get(THIS_MODULE);

   if ( !pepe_already_call )
      msg_ptr = message_before_pepe;
   else
      msg_ptr = message_after_pepe;

   return 0;
}

static int pepe_release(struct inode *inode, struct file *file)
{
   module_put(THIS_MODULE);
   return 0;
}
```

Os drivers não podem ser removidos a mera vontade do usuário root.
O kernel fornece aos módulos um contador para indicar se estão em uso.

#include <linux/module.h>
try_module_get(THIS_MODULE)
incrementar o contador
module_put(THIS_MODULE)
decrementa o contador

Pepe, char driver básico: read

```
static ssize_t pepe_read(struct file *filp, char __user *user_buffer,
                         size_t length, loff_t *offset)
{
   ssize_t readed = 0;

   if ( *msg_ptr == 0 )
      return 0;

   while ( length > 0 && *msg_ptr != 0 ) {
      put_user(*msg_ptr++, user_buffer++);
      length--;
      readed++;
   }

   return readed;
}
```

Pepe, char driver básico: read

•O código/dados do kernel e o código/dados do usuário estão em dois espaços de endereçamento distintos.

•Para copiar do dispositivo para um buffer do usuário (parâmetro user_buffer) podemos usar estas rotinas:

- unsigned long copy_to_user (void __user * to, const void * from, unsigned long n);
- put_user (x, ptr);

•Para copiar do usuário para o driver:

- unsigned long copy_from_user (void * to, const void __user * from, unsigned long n);
- get_user(x, ptr);

Pepe, char driver básico: write

```
static ssize_t pepe_write(struct file *filp, const char __user *buffer,
                          size_t len, loff_t *off)
{
   ssize_t writed = 0;

   while ( writed < len ) {
      const int position = (write_position % WRITE_BUFFER_SIZE);
      write_position++;

      get_user(write_buffer[position], buffer + writed);
      writed++;
   }

   write_buffer[write_position-1] = 0;
   write_position = 0;

   printk(KERN_INFO "%s", write_buffer);

   if ( strcmp(message_to_call_pepe, write_buffer) == 0 ){
      pepe_already_call = 1;
   }

   return writed;
}
```

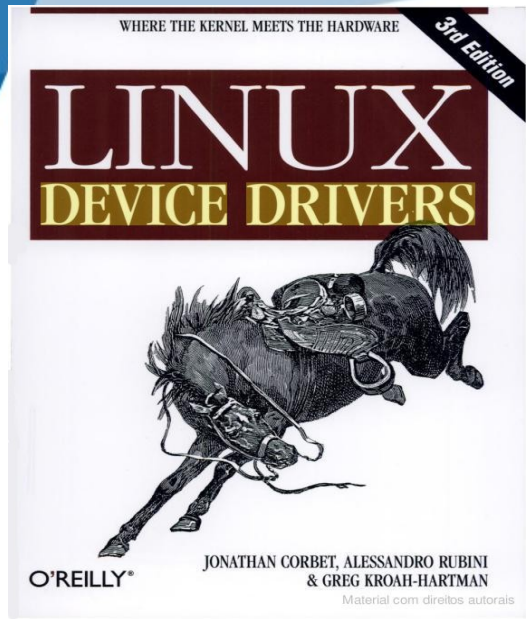
Pepe, char driver básico: insmod e mknod

Antes de começarmos a ler e escreve no driver, precisamos criar o arquivo. O comando mknod faz isto para nós.

mknod /dev/dispositivo c <major number> <minor number>

```
$insmod ./pepe.ko
$cat /proc/devices | grep pepe
252 pepe
$mknod /dev/pepe c 252 0
```

Referências



[1]Rubini, Alessandro & Corbet, Jonathan (2005), Linux Device Drivers.
Disponível em http://lwn.net/Kernel/LDD3/

The Linux Kernel Module Programming Guide, disponível em
http://tldp.org/LDP/lkmpg/2.6/html/index.html