

Linux - *Devices and Modules*

Anderson Coelho Weller

Universidade Estadual de Campinas
Instituto de Computação

18 de novembro de 2013

Roteiro

1 Devices and Modules

2 Criar Módulo

3 Criar CDevs

Classes de Dispositivo

- O Linux subdivide os dispositivos em 3 tipos principais:
 - *Character devices* (cdevs)
 - *Block devices* (blkdevs)
 - *Network devices*



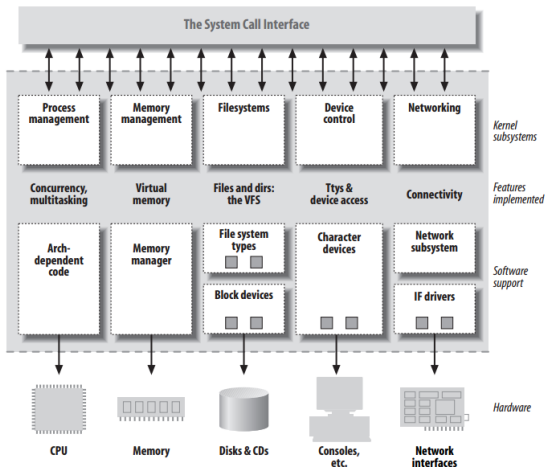
Classes de Dispositivo

- O Linux subdivide os dispositivos em 3 tipos principais:
 - *Character devices* (cdevs)
 - *Block devices* (blkdevs)
 - *Network devices*



Classes de Dispositivo

- Uma visão geral do *kernel* [1]



features implemented as modules



Classes de Dispositivo

- Nem todos os dispositivos são dispositivos físicos.
- Os *pseudo drivers* são dispositivos virtuais que proporcionam acesso à funcionalidades do kernel, tipo:
 - Gerador de números aleatórios do Kernel (/dev/random ou /dev/urandom)
 - Dispositivo Null (/dev/null)
 - Dispositivo Zero (/dev/zero)
 - Dispositivo Full (/dev/full)
 - Memória Principal (/dev/mem)



Classes de Dispositivo

- Nem todos os dispositivos são dispositivos físicos.
- Os *pseudo drivers* são dispositivos virtuais que proporcionam acesso à funcionalidades do kernel, tipo:
 - Gerador de números aleatórios do Kernel (/dev/random ou /dev/urandom)
 - Dispositivo Null (/dev/null)
 - Dispositivo Zero (/dev/zero)
 - Dispositivo Full (/dev/full)
 - Memória Principal (/dev/mem)



Classes de Dispositivo

- Nem todos os dispositivos são dispositivos físicos.
- Os *pseudo drivers* são dispositivos virtuais que proporcionam acesso à funcionalidades do kernel, tipo:
 - Gerador de números aleatórios do Kernel (/dev/random ou /dev/urandom)
 - Dispositivo Null (/dev/null)
 - Dispositivo Zero (/dev/zero)
 - Dispositivo Full (/dev/full)
 - Memória Principal (/dev/mem)



Classes de Dispositivo

- Nem todos os dispositivos são dispositivos físicos.
- Os *pseudo drivers* são dispositivos virtuais que proporcionam acesso à funcionalidades do kernel, tipo:
 - Gerador de números aleatórios do Kernel (/dev/random ou /dev/urandom)
 - Dispositivo Null (/dev/null)
 - Dispositivo Zero (/dev/zero)
 - Dispositivo Full (/dev/full)
 - Memória Principal (/dev/mem)



Classes de Dispositivo

- Nem todos os dispositivos são dispositivos físicos.
- Os *pseudo drivers* são dispositivos virtuais que proporcionam acesso à funcionalidades do kernel, tipo:
 - Gerador de números aleatórios do Kernel (/dev/random ou /dev/urandom)
 - Dispositivo Null (/dev/null)
 - Dispositivo Zero (/dev/zero)
 - Dispositivo Full (/dev/full)
 - Memória Principal (/dev/mem)



Classes de Dispositivo

- Nem todos os dispositivos são dispositivos físicos.
- Os *pseudo drivers* são dispositivos virtuais que proporcionam acesso à funcionalidades do kernel, tipo:
 - Gerador de números aleatórios do Kernel (/dev/random ou /dev/urandom)
 - Dispositivo Null (/dev/null)
 - Dispositivo Zero (/dev/zero)
 - Dispositivo Full (/dev/full)
 - Memória Principal (/dev/mem)



Classes de Dispositivo

- Nem todos os dispositivos são dispositivos físicos.
- Os *pseudo drivers* são dispositivos virtuais que proporcionam acesso à funcionalidades do kernel, tipo:
 - Gerador de números aleatórios do Kernel (/dev/random ou /dev/urandom)
 - Dispositivo Null (/dev/null)
 - Dispositivo Zero (/dev/zero)
 - Dispositivo Full (/dev/full)
 - Memória Principal (/dev/mem)



Módulos

- Os módulos são imagens binárias (carregáveis no *kernel*) contendo:
 - as sub-rotinas,
 - os dados e
 - os pontos de entrada e saída.
- Que implementam um dos tipos de dispositivos.



Módulos

- Os módulos são imagens binárias (carregáveis no *kernel*) contendo:
 - as sub-rotinas,
 - os dados e
 - os pontos de entrada e saída.
- Que implementam um dos tipos de dispositivos.



Módulos

- Os módulos são imagens binárias (carregáveis no *kernel*) contendo:
 - as sub-rotinas,
 - os dados e
 - os pontos de entrada e saída.
- Que implementam um dos tipos de dispositivos.



Módulos

- Os módulos são imagens binárias (carregáveis no *kernel*) contendo:
 - as sub-rotinas,
 - os dados e
 - os pontos de entrada e saída.
- Que implementam um dos tipos de dispositivos.



Módulos

- Os módulos são imagens binárias (carregáveis no *kernel*) contendo:
 - as sub-rotinas,
 - os dados e
 - os pontos de entrada e saída.
- Que implementam um dos tipos de dispositivos.



Módulos

- O suporte a módulos permite aos sistemas:
 - Manter uma imagem mínima do kernel, e
 - Carregar somente os drivers necessários.



Módulos

- O suporte a módulos permite aos sistemas:
 - Manter uma imagem mínima do kernel, e
 - Carregar somente os drivers necessários.



Módulos

- O suporte a módulos permite aos sistemas:
 - Manter uma imagem mínima do kernel, e
 - Carregar somente os drivers necessários.



Preparação do ambiente

- Primeiro temos que instalar e preparar os *headers* do *kernel*:

```
1 sudo -i
2 apt-get install module-assistant
3 m-a prepare
```

- Obs.: O seguinte comando também instala os pacotes que precisamos (equivalente ao “m-a prepare”):

```
1 sudo apt-get install build-essential linux-headers-$(uname
-r)
```



Preparação do ambiente

- Primeiro temos que instalar e preparar os *headers* do *kernel*:

```
1 sudo -i
2 apt-get install module-assistant
3 m-a prepare
```

- Obs.: O seguinte comando também instala os pacotes que precisamos (equivalente ao “m-a prepare”):

```
1 sudo apt-get install build-essential linux-headers-$(uname
-r)
```



Arquivo fonte 1

- Crie um diretório e insira o arquivo 'hello.c' [2] [3], contendo o seguinte código:

```
1 // Definindo __KERNEL__ e MODULE nos permite acessar o
   código no nível do kernel, geralmente não disponível
   para programas userspace.
2 #undef __KERNEL__
3 #define __KERNEL__
4 #undef MODULE
5 #define MODULE
6
7 // Linux Kernel/LKM headers: module.h é necessário para
   todos os módulos e kernel.h é necessário para KERN_INFO.
8 #include <linux/module.h> // incluído para todos os módulos
   do kernel
9 #include <linux/kernel.h> // incluído para KERN_INFO
10 #include <linux/init.h> // incluído para __init e __exit
   macros
11
```



Arquivo fonte II

```
12 static int __init hello_init(void)
13 {
14     printk(KERN_ALERT "Inicio do modulo.\n");
15     return 0; // Retorno diferente de zero significa que o
               módulo não pôde ser carregado.
16 }
17
18 static void __exit hello_cleanup(void)
19 {
20     printk(KERN_ALERT "Fim do modulo.\n");
21 }
22
23 module_init(hello_init);
24 module_exit(hello_cleanup);
25
26 MODULE_LICENSE("GPL");
27 MODULE_AUTHOR("Anderson");
28 MODULE_DESCRIPTION("Modulo Exemplo");
```



Makefile

- Crie o arquivo 'Makefile' [3] com as seguintes informações:

```
1 obj-m := hello.o
2 KDIR  := /lib/modules/$(shell uname -r)/build
3 PWD   := $(shell pwd)
4
5 all:
6     $(MAKE) -C $(KDIR) M=$(PWD) modules
7
8 clean:
9     $(MAKE) -C $(KDIR) M=$(PWD) clean
```



Compilar o código

- Para compilar o código basta executar o 'make':

```
1 make
```

- Exemplo de retorno do comando 'make':

```
make -C /lib/modules/3.2.0-4-686-pae/build  
M=/home/anderson/modulos modules  
make[1]: Entrando no diretorio  
  '/usr/src/linux-headers-3.2.0-4-686-pae'  
Building modules, stage 2.  
MODPOST 1 modules  
make[1]: Saindo do diretorio  
  '/usr/src/linux-headers-3.2.0-4-686-pae'
```



Compilar o código

- Para compilar o código basta executar o 'make':

```
1 make
```

- Exemplo de retorno do comando 'make':

```
make -C /lib/modules/3.2.0-4-686-pae/build  
M=/home/anderson/modulos modules  
make[1]: Entrando no diretorio  
  '/usr/src/linux-headers-3.2.0-4-686-pae'  
Building modules, stage 2.  
MODPOST 1 modules  
make[1]: Saindo do diretorio  
  '/usr/src/linux-headers-3.2.0-4-686-pae'
```



Inserir/Remover Módulo

- Para inserir o novo módulo no *kernel*:

```
1 sudo insmod hello.ko
```

- Para remover o módulo do *kernel*:

```
1 sudo rmmod hello
```



Inserir/Remover Módulo

- Para inserir o novo módulo no *kernel*:

```
1 sudo insmod hello.ko
```

- Para remover o módulo do *kernel*:

```
1 sudo rmmod hello
```



-
- ```

graph TD
 insmod --> init_function[init function]
 init_function -.-> blk_init_queue[blk_init_queue()]
 init_function -.-> add_disk[add_disk()]
 add_disk -.-> struct_gendisk[struct gendisk]
 struct_gendisk --> block_device_ops[block_device ops]
 struct_gendisk --> request_queue[request_queue_]
 request_queue --> request[request()]
 request --> request_handling[]
 request_handling -.-> request_handling
 request_handling -.-> del_gendisk[del_gendisk()]
 request_handling -.-> blk_cleanup_queue[blk_cleanup_queue()]
 blk_cleanup_queue --> cleanup_function[cleanup function]
 rmmod --> cleanup_function

```

- > Data operation
- > Data pointer
- - -> Function call
- > Function pointer

Data

[1]



# Verificar o Log do Linux

- Verificar as informações do *Log*:

```
1 tail /var/log/syslog
2 # OU
3 tail /var/log/messages
```

- Exemplo de conteúdo do *Log*:

```
Nov 11 15:38:35 debian-modulo kernel: Inicio do modulo.
Nov 11 15:38:40 debian-modulo kernel: Fim do modulo.
```



# Verificar o Log do Linux

- Verificar as informações do *Log*:

```
1 tail /var/log/syslog
2 # OU
3 tail /var/log/messages
```

- Exemplo de conteúdo do *Log*:

```
Nov 11 15:38:35 debian-modulo kernel: Inicio do modulo.
Nov 11 15:38:40 debian-modulo kernel: Fim do modulo.
```



# Character devices

- Vamos criar um driver de dispositivo de caractere.
- Esse dispositivo (`scull`) irá atuar sobre uma área da memória.
- A ideia é demonstrar a interface entre o *kernel* e o dispositivo.



# Character devices

- Vamos criar um driver de dispositivo de caractere.
- Esse dispositivo (scull) irá atuar sobre uma área da memória.
- A ideia é demonstrar a interface entre o *kernel* e o dispositivo.



# Character devices

- Vamos criar um driver de dispositivo de caractere.
- Esse dispositivo (`scull`) irá atuar sobre uma área da memória.
- A ideia é demonstrar a interface entre o *kernel* e o dispositivo.



# Design do dispositivo

- Primeiro, iremos definir as capacidades (o mecanismo) que dispositivo irá disponibilizar para os programas.
- Serão criados os seguintes tipos de dispositivos:
  - `scull0` a `scull3`: Área global e persistente da memória, acessível aos programas.



# Design do dispositivo

- Primeiro, iremos definir as capacidades (o mecanismo) que dispositivo irá disponibilizar para os programas.
- Serão criados os seguintes tipos de dispositivos:
  - **scull0** a **scull3**: Área global e persistente da memória, acessível aos programas.



# Design do dispositivo

- Primeiro, iremos definir as capacidades (o mecanismo) que dispositivo irá disponibilizar para os programas.
- Serão criados os seguintes tipos de dispositivos:
  - **scull0** a **scull3**: Área global e persistente da memória, acessível aos programas.



# Números Maiores e Menores

- Dispositivos de caractere são acessados pelo nome no sistema de arquivos.
- Eles são arquivos especiais localizados em */dev*.
- Dispositivos de caractere são identificados pela letra “c” (através do comando *'ls -l'*)
- Dispositivos de bloco são identificados pela letra “b”



# Números Maiores e Menores

- Dispositivos de caractere são acessados pelo nome no sistema de arquivos.
- Eles são arquivos especiais localizados em `/dev`.
- Dispositivos de caractere são identificados pela letra “c” (através do comando `'ls -l'`)
- Dispositivos de bloco são identificados pela letra “b”





# Números Maiores e Menores

- Dispositivos de caractere são acessados pelo nome no sistema de arquivos.
- Eles são arquivos especiais localizados em */dev*.
- Dispositivos de caractere são identificados pela letra “c” (através do comando *'ls -l'*)
- Dispositivos de bloco são identificados pela letra “b”



# Números Maiores e Menores

- Dispositivos de caractere são acessados pelo nome no sistema de arquivos.
- Eles são arquivos especiais localizados em `/dev`.
- Dispositivos de caractere são identificados pela letra “c” (através do comando `'ls -l'`)
- Dispositivos de bloco são identificados pela letra “b”



# Números Maiores e Menores

- Os dois números que aparecem antes da data de modificação são os números de identificação do dispositivo:
  - Major Number:** Identifica o driver associado ao dispositivo;
  - Minor Number:** Usado pelo *kernel* para identificar o dispositivo referenciado.

```
crw-rw-rw- 1 root root 1, 3 Apr 11 2002 null
crw----- 1 root root 10, 1 Apr 11 2002 psaux
crw----- 1 root root 4, 1 Oct 28 03:04 tty1
crw-rw-rw- 1 root tty 4, 64 Apr 11 2002 ttys0
crw-rw---- 1 root uucp 4, 65 Apr 11 2002 ttyS1
crw--w---- 1 vcsa tty 7, 1 Apr 11 2002 vcs1
crw--w---- 1 vcsa tty 7, 129 Apr 11 2002 vcsa1
crw-rw-rw- 1 root root 1, 5 Apr 11 2002 zero
```



# Character devices

- Para criar um driver de dispositivo de caractere precisamos codificar alguns métodos e ligá-los através das estruturas:
  - file\_operations
  - seq\_operations

```
1 static struct file_operations scull_proc_ops = {
2 .owner = THIS_MODULE,
3 .open = scull_proc_open,
4 .read = seq_read,
5 .llseek = seq_lseek,
6 .release = seq_release
7 };
```



# Character devices

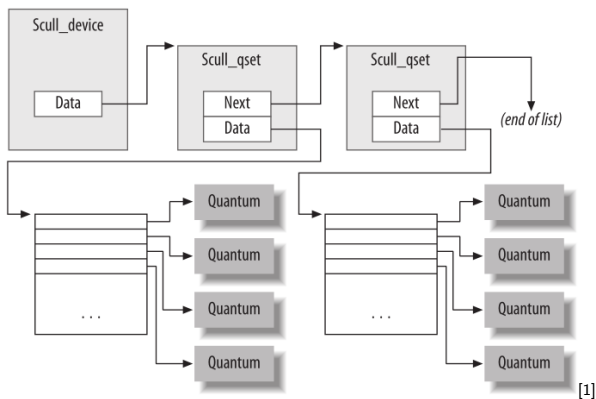
```
1 struct file_operations scull_fops = {
2 .owner = THIS_MODULE,
3 .llseek = scull_llseek,
4 .read = scull_read,
5 .write = scull_write,
6 .unlocked_ioctl = scull_ioctl,
7 .open = scull_open,
8 .release = scull_release,
9 };
```

```
1 static struct seq_operations scull_seq_ops = {
2 .start = scull_seq_start,
3 .next = scull_seq_next,
4 .stop = scull_seq_stop,
5 .show = scull_seq_show
6 };
```



# Scull Device

- Estrutura do dispositivo criado:







# Método de criptografia XOR

- Exemplo de criptografia dos caracteres (inserido no método Write) [4]:

```
1 ssize_t scull_read(struct file *filp, char __user *buf,
2 size_t count, loff_t *f_pos) {
3 // (...) Criptografia cada uma das letras da memoria
4 char key[21] = "12345678901234567890";
5 int key_count = 0; int i; int letra; int chave;
6 char *buf_ok = dptr->data[s_pos]+q_pos;
7 for (i=0 ; i<count ; i++)
8 {
9 letra = buf_ok[i];
10 chave = key[key_count];
11 buf_ok[i] = letra ^ chave;
12 key_count++;
13 if(key_count == strlen(key))
14 key_count = 0;
15 } // (...)
```



# Referências I

-  J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers*, 3rd ed. Sebastopol, CA: O'Reilly Media, Inc., February 2005.
-  R. Love, *Linux Kernel Development*, 3rd ed. Indiana, US: Pearson Education, Inc., 2010.
-  M. Loiseau, ““Hello World” Loadable Kernel Module,” April 2012. [Online]. Available: <http://blog.markloiseau.com/2012/04/hello-world-loadable-kernel-module-tutorial/> [Accessed: Nov. 11, 2013]
-  ShadenSmith, “C Tutorial - XOR Encryption,” Codecall, June 2009. [Online]. Available: <http://forum.codecall.net/topic/48889-c-tutorial-xor-encryption/> [Accessed: Nov. 12, 2013]





# Linux - *Devices and Modules*



Anderson Coelho Weller