Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Campo Mourão
Bacharelado em Ciência da Computação

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

DISCIPLINA DE SISTEMAS OPERACIONAIS

ALAN RODRIGO PATRIARCA

LUCAS SOUZA SANTOS

ESCREVENDO MÓDULOS DO NÚCLEO LINUX

Driver de dispositivo USB

CAMPO MOURÃO

2021

SUMÁRIO

1. Introdução

O kernel do Linux possui um design modular, o que permite o carregamento de um módulo kernel dinamicamente na memória mesmo depois que o kernel já tenha sido inicializado (carregado na memória). A linguagem de programação C pode ser usada para a criação de módulos kernel Linux em conjunto com algumas bibliotecas que o Linux fornece.

Este relatório tem o objetivo de apresentar o processo de implementação de um módulo kernel Linux, mais especificamente um driver de dispositivo USB. Primeiramente é apresentada uma descrição geral da atividade desenvolvida, seguindo para a seção 3 que expõe os métodos utilizados na solução, logo após a seção 4 apresenta os resultados com uma explicação passo a passo do código implementado, seguindo para a seção 5 onde o módulo é testado, e por fim, a seção 6 expõe as conclusões tiradas a partir de todo esse processo.

2. Descrição da atividade

Neste projeto foi desenvolvido um módulo para o núcleo do Linux utilizando a linguagem de programação C. O módulo implementado pode ser descrito como um driver de dispositivo USB que reconhece quando um ou mais dispositivos USB específicos são conectados e desconectados ao sistema no qual o módulo está instalado.

O módulo foi implementado utilizando as bibliotecas Linux para a linguagem C `module.h` e `usb.h`, que contém as funções e estruturas necessárias para a implementação de um driver de dispositivo USB.

3. Métodos

Para a criação do módulo foi necessário um certo conhecimento de conceitos relacionados a sistemas operacionais e a implementação de recursos. Nesse sentido, aspectos como o funcionamento do núcleo do sistema Linux e de implementação de módulos que possam ser utilizados pelo sistema operacional tiveram uma grande importância para o desenvolvimento da atividade.

3.1. Kernel Linux

O kernel do Linux como em outros sistemas operacionais é composto por subsistemas, que têm como responsabilidade implementar funcionalidades como o controle de acesso dos processos ao processador e a memória, gerenciamento de dispositivos na rede, comunicação entre processos, dentre outros.

No entanto, diferente de outros sistemas operacionais, o Linux trabalha com design modular, ou seja, somente o núcleo mínimo é carregado em memória no momento da inicialização e sempre que um usuário requisitar uma funcionalidade que o núcleo não possui, um módulo do kernel é dinamicamente adicionado na memória.

3.2. Módulo Kernel

O Linux utiliza a arquitetura monolítica, ou seja, que opera como um bloco maciço de código, e para adicionar novos recursos à essa arquitetura é utilizado o conceito de módulos. Um módulo do Linux é composto por um bloco de código que pode ser inserido ou removido do núcleo em execução, adicionando interfaces, drivers, dentre outros recursos.

3.3. Driver de dispositivo

Os drivers de dispositivos são programas que têm como responsabilidade fazer a comunicação entre o sistema operacional e o hardware, como monitores, placas de vídeo, impressoras, som e até mesmo dispositivos USB, que será o foco desta atividade. Na maioria das vezes os drivers de dispositivos são inseridos ao núcleo do Linux por meio de módulos que são acoplados ao sistema.

Corbet et al. [1] apresenta uma visão geral sobre drivers de dispositivos Linux, e serviu de grande influência para a implementação da solução proposta neste artigo, pois possui um capítulo dedicado a drivers de dispositivos, explicando não só os conceitos, como também as funções, tipos de dados e variáveis necessárias para a implementação.

4. Resultados

Nesta seção será apresentado com detalhes como a solução foi desenvolvida, bem como o código passo a passo da implementação. O Código-fonte do módulo descrito neste artigo foi escrito em um único arquivo, vamos chamar o arquivo de `modulo.c`.

4.1. Bibliotecas

A maioria dos códigos do kernel acabam incluindo um grande número de bibliotecas para obter a definição de funções, tipos de dados e variáveis necessárias. Neste projeto foram utilizadas apenas duas bibliotecas. Veja no Código-fonte 1.

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

Código-fonte 1 - Bibliotecas usadas no módulo.

A biblioteca `module.h` contém muitas definições de símbolos e funções necessárias para módulos carregáveis. Já a usb.h contém tudo relacionado à drivers de dispositivo USB.

4.2. Definições descritivas do módulo

A biblioteca `module.h` oferece suporte a algumas definições descritivas que podem ser aplicadas ao módulo desenvolvido, como por exemplo a linha `MODULE_LICENSE` que especifica a licença que se aplica ao código, `MODULE_AUTHOR` para informar quem escreveu o módulo, e `MODULE_DESCRIPTION` para descrever o que aquele módulo faz. Neste caso a licença utilizada foi a “GPL” (Licença Pública Geral), uma licença reconhecida pelo kernel para qualquer versão do GNU. O Código-fonte 2 apresenta como elas foram usadas.

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alan Patriarca e Lucas Souza");
MODULE_DESCRIPTION("Este driver reconhece a conexão e desconexão de um ou
mais dispositivos USBs específicos");
```

Código-fonte 2 - Definições descritivas do módulo.

4.3. Tipos de dados

A estrutura `usb_device_id` fornece uma lista dos diferentes tipos de dispositivos USB que o driver deve suportar, é ela que vai definir qual driver automaticamente carregar quando um dispositivo USB é conectado ao sistema. Como o driver que está sendo implementado corresponde a dispositivos USB específicos, para inicializar a estrutura utilizamos o macro `USB_DEVICE`, dessa forma para cada dispositivo USB a ser registrado,

deve-se fornecer como parâmetro os identificadores do fornecedor e do produto. Neste caso foram registrados três dispositivos USB, com seus identificadores obtidos através do comando `lsusb` no Linux. Veja no Código-fonte 3.

```
// Estrutura dos dispositivos USB
static struct usb_device_id tabela_de_dispositivos[] = {
  // HD externo
  { USB_DEVICE(0x13fd, 0x3920) },   // Informacao obtida atraves do
comando "lsusb" no linux
  // Smartphone Samsung
  { USB_DEVICE(0x04e8, 0x6860) },
  // Dispositivo Multilaser (mouse e teclado)
  { USB_DEVICE(0x062a, 0x4c01) },
  {} // A estrutura usb_device_id deve terminar com uma entrada NULL
};
```

Código-fonte 3 - Estrutura dos dispositivos USB usb_device_id.

Para permitir que as ferramentas de espaço de usuário descubram quais dispositivos este driver pode controlar, a macro `MODULE_DEVICE_TABLE` é utilizada para exportar a lista de dispositivos para o espaço de usuário, como pode ser visto no Código-fonte 4.

```
// Exportando tabela de IDs de dispositivos para o user space
MODULE_DEVICE_TABLE(usb, tabela_de_dispositivos);
```

Código-fonte 4 - Exportando a estrutura dos dispositivos para o espaço de usuário.

Todos os drivers USB devem criar uma estrutura `usb_driver`. É nela que ficam as funções, retornos de chamada e variáveis que descrevem o driver USB para o kernel. Na solução implementada, os campos preenchidos podem ser vistos na tabela 1.

.name	Ponteiro que aponta para o nome do driver.
.id_table	Ponteiro que aponta para a estrutura `usb_device_id`.

.probe	Ponteiro que aponta para a função de sonda no driver USB. Neste caso o atributo apontará para um função que imprime uma mensagem no log do Kernel toda vez que um dispositivo USB deste driver for conectado.
.disconnect	Ponteiro para a função de desconexão no driver USB. Neste caso o atributo apontará para um função que imprime uma mensagem no log do Kernel toda vez que um dispositivo deste driver for desconectado.

Tabela 1 - Descrição dos campos usados na estrutura usb_driver.

```
// Driver de dispositivo que será fornecido ao kernel
static struct usb_driver driver_de_dispositivo = {
  .name = "Identificador de Dispositivo USB",
  .id_table = tabela_de_dispositivos,
  .probe = dispositivo_conectado,
  .disconnect = dispositivo_desconectado
};
```

Código-fonte 5 - Estrutura usb_driver.

4.4. Conexão e desconexão dos dispositivos

Como foi citado no tópico 4.3, os atributos `.probe` e `.disconnect` na estrutura `usb_driver` foram utilizados para imprimir uma mensagem no log do kernel no momento da conexão e desconexão dos dispositivos USB registrados. Para que isso aconteça é necessário que estes atributos apontem para uma função que será chamada pelo núcleo neste momento. Vide Código-fonte 6.

```
// Funcao chamada quando algum dispositivo registrado for conectado
static int dispositivo_conectado(struct usb_interface *interface, const
struct usb_device_id *id) {
  printk(KERN_INFO "[*] Dispositivo (%04X:%04X) conectado\n",
id->idVendor, id->idProduct);
  return 0;
}

// Funcao chamada quando algum dispositivo registrado for desconectado
static void dispositivo_desconectado(struct usb_interface *interface) {
  printk(KERN_INFO "[*] Dispositivo desconectado\n");
```

```
}
```

Código-fonte 6 - Funções de sondagem e desconexão/descarregamento.

Onde a função `printk` é utilizada para imprimir as mensagens no log do Kernel nos momentos de conexão e desconexão dos dispositivos USB.

4.5. Inicialização e descarregamento do módulo

Outro ponto imprescindível em um módulo kernel são suas macros de inicialização e descarregamento, vide Código-fonte 7. O uso de `module_init` é obrigatório, pois ela informa onde a função de inicialização do módulo pode ser encontrada. Sem esta definição, a função de inicialização nunca é chamada. Já o `module_exit` informa onde pode ser encontrada a função de descarregamento do módulo. Sem esta definição, o kernel não permite que o módulo seja descarregado.

```
module_init(iniciar_modulo);
module_exit(descarregar_modulo);
```

Código-fonte 7 - Macros de inicialização e descarregamento.

A função `iniciar_modulo` é o ponto de entrada de inicialização do driver e é chamada durante a inicialização do sistema. Já a função `descarregar_modulo` é o ponto de saída do driver, é chamada ao descarregar o módulo do kernel Linux. O driver usb é registrado pela chamada da função `usb_register` e descarregado pela função `usb_deregister`, contidas nas funções `iniciar_modulo` e `descarregar_modulo`, respectivamente. As funções podem ser vistas no Código-fonte 8.

```
// Executa ao inicializar modulo
static int __init iniciar_modulo(void) {
  int ret;
  printk(KERN_INFO "Registrando driver de dispositivo USB");
  ret = usb_register(&driver_de_dispositivo);
  if (ret) {
      printk("\tFalha ao registrar módulo, erro número %d", ret);
  } else {
      printk(KERN_INFO "\tDriver registrado com sucesso");
```

```
  }
  return ret;
}

// Executa ao descarregar modulo
static void __exit descarregar_modulo(void) {
  printk(KERN_INFO "Descarregando driver de dispositivo USB");
  usb_deregister(&driver_de_dispositivo);
  printk(KERN_INFO "\tDriver descarregado com sucesso");
}
```

Código-fonte 8 - Funções para iniciar e descarregar módulo.

Este é o Código-fonte do driver de dispositivo USB, todo o Código-fonte apresentado neste relatório está em um repositório do GitHub e pode ser acessado através deste link: https://github.com/souzalucas/usb_device_driver.

5. Executando o módulo

Com o driver de dispositivo implementado, já podemos partir para a fase de execução, onde iremos realizar compilação do Código-fonte, instalação do módulo gerado a partir da compilação, e por fim realizar o teste com um dispositivo USB que tenha sido registrado na estrutura `usb_device_id`.

5.1. Compilação

Para compilar o Código-fonte utilizaremos o utilitário `make`, que realiza a compilação através das instruções contidas em um arquivo que iremos chamar de `Makefile`.

```
obj-m := modulo.o

KERNEL_DIR = /lib/modules/$(shell uname -r)/build

all:
    $(MAKE) -C $(KERNEL_DIR) M=$(shell pwd) modules

clean:
    rm -rvf *.o *.ko *.mod.c *.mod *.cmd *.symvers *.order .*.cmd
```

Código-fonte 9 - Makefile, para a compilação do módulo.

Este é o arquivo `Makefile` (Código-fonte 9), onde o atributo `obj-m` especifica o arquivo de objeto que será construído como um módulo de kernel carregável. Observe que é necessário especificar o diretório de construção do kernel para o qual procuramos compilar o módulo.

Com o `Makefile` pronto, já podemos compilar o módulo com o comando `make`. A saída deve ser semelhante à seguinte à do Código-fonte 10.

```
lucas@dell:~/Documentos/usb_device_identifier$ make
make -C /lib/modules/5.8.0-53-generic/build
M=/home/lucas/Documentos/usb_device_identifier modules
make[1]: Entrando no diretório '/usr/src/linux-headers-5.8.0-53-generic'
  CC [M]  /home/lucas/Documentos/usb_device_identifier/modulo.o
  MODPOST /home/lucas/Documentos/usb_device_identifier/Module.symvers
  CC [M]  /home/lucas/Documentos/usb_device_identifier/modulo.mod.o
  LD [M]  /home/lucas/Documentos/usb_device_identifier/modulo.ko
make[1]: Saindo do diretório '/usr/src/linux-headers-5.8.0-53-generic'
```

Código-fonte 10 - Saída do comando make.

5.2. Instalação

Se tudo der certo, após a compilação, vários arquivos serão criados no diretório do módulo, esses arquivos representam o módulo criado, símbolos externos definidos no módulo, lista da ordem de compilação e marcadores escritos no código. Para inserir o módulo no kernel basta executar o comando:

```
sudo insmod modulo.ko
```

Código-fonte 11 - Comando para instalar o módulo.

Para confirmar se o módulo foi instalado, basta executar o comando `lsmod | grep modulo`, que irá filtrar os resultados pelo nome do módulo criado. Uma resposta parecida com essa deverá ser obtida:

```
lucas@dell:~/Documentos/usb_device_identifier$ lsmod | grep modulo
modulo                 16384  0
```

Código-fonte 12 - Saída do comando lsmod | grep modulo.

Ou então podemos confirmar a instalação com o comando `sudo dmesg`, que irá devolver como saída, a lista dos logs do kernel. Você deve procurar as seguintes linhas para confirmar a instalação:

```
[ 6248.368868] Registrando driver de dispositivo USB
[ 6248.369380] usbcore: registered new interface driver Identificador de
Dispositivo USB
[ 6248.369381]  Driver registrado com sucesso
```

Código-fonte 13 - Mensagens que confirmam a instalação do módulo.

5.3. Teste

Recapitulando, o nosso módulo se trata de um driver de dispositivo USB que identifica a conexão e desconexão de dispositivos USB específicos, imprimindo alertas no log do kernel. Dessa forma, para testá-lo devemos ter em mãos um dispositivo USB que foi registrado no driver. Com o dispositivo em mãos, basta inseri-lo em uma entrada USB da máquina no qual o sistema foi instalado, e depois removê-lo. Agora devemos verificar o log do kernel com o comando `sudo dmesg`, o módulo está funcionando perfeitamente se mensagens como estas aparecem no log:

```
[ 6396.860443] [*] Dispositivo (062A:4C01) conectado
[ 6396.860580] [*] Dispositivo desconectado
```

Código-fonte 14 - Mensagens de conexão e desconexão de um dispositivo USB.

Se as mensagens não aparecem no log, possivelmente algum driver de dispositivo USB já instalado no sistema pode estar impedindo o funcionamento completo do driver que nós implementamos. Para resolver este problema, basta buscar pelos drivers USB instalados no sistema com o comando `lsmod`. Após identificá-los, você pode removê-los com o comando `sudo modprobe -r nome_do_modulo`, e repetir os passos da seção 5.3. Após o teste, os módulos removidos podem ser inseridos novamente com o comando `sudo modprobe nome_do_modulo`.

O módulo implementado também pode ser removido posteriormente com o comando `sudo rmmod modulo.ko`. A seguinte mensagem deve ser impressa no log do kernel:

```
[ 6224.031092] Descarregando driver de dispositivo USB
[ 6224.031093] usbcore: deregistering interface driver Identificador de
Dispositivo USB
[ 6224.031094] Driver descarregado com sucesso
```

Código-fonte 15 - Mensagem do descarregamento do módulo.

6. Conclusões

Com a atividade, foi necessário utilizar diversos conceitos relacionados a sistemas operacionais e posteriormente aplicá-los na prática para a construção de um driver que identificasse inserções e remoções de dispositivos USB. O módulo USB desenvolvido foi criado para identificar dispositivos com base em uma lista pré-configurada. A lista consiste em uma estrutura de dispositivos USB, no qual cada dispositivo possui um identificador do fornecedor e um identificador do produto.

Com base na lista criada, inserimos o módulo ao núcleo do Linux e assim foi possível visualizar mensagens no log do kernel após a conexão e desconexão dos dispositivos. Para a visualização das mensagens foi utilizado o comando `dmesg`, responsável por mostrar o log de mensagens do núcleo. Portanto, com a atividade foi possível acompanhar a criação de um módulo de um driver USB para o núcleo do Linux, mostrando que é possível construir o seu próprio módulo kernel aplicando os conceitos de sistemas operacionais apresentados na disciplina.

7. Referências

[1] Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman. (2005). Linux Device Drivers, 3rd Edition, páginas 327-361.

[2] Carlos A. Maziero. (2005). Sistemas Operacionais: Conceitos e Mecanismos, páginas 27-37.