LABORATÓRIO TPSE I

Lab 08: Desenvolvendo Device Drivers

Prof. Francisco Helder

12 de novembro de 2022

1 Módulos no Kernel

Os Device Drivers assumem um papel especial no kernel do Linux. São “caixas pretas” distintas que fazem uma determinada peça de hardware responder a uma interface de programação interna bem definida; eles escondem completamente os detalhes de como o dispositivo funciona. As atividades do usuário são realizadas por meio de um conjunto de chamadas padronizadas e independentes do driver específico; mapear essas chamadas para operações específicas do dispositivo que atuam no hardware real é, então, a função do Device Driver.

Esta interface de programação é tal que os drivers podem ser construídos separadamente do resto do kernel e “plugados” em tempo de execução quando necessário. Essa modularidade torna os drivers em Linux fáceis de escrever, a ponto de agora existirem centenas deles disponíveis.

Conforme você aprende a escrever drivers, você descobre muito sobre o kernel Linux em geral; isso pode ajudá-lo a entender como sua máquina funciona e por que as coisas nem sempre são tão rápidas quanto você espera ou não fazem exatamente o que você deseja.

2 Carregando Módulos no Linux

Um dos bons recursos do Linux é a capacidade de estender em tempo de execução o conjunto de recursos oferecidos pelo kernel. Isso significa que você pode adicionar funcionalidades ao kernel (e remover funcionalidades também) enquanto o sistema está funcionando.

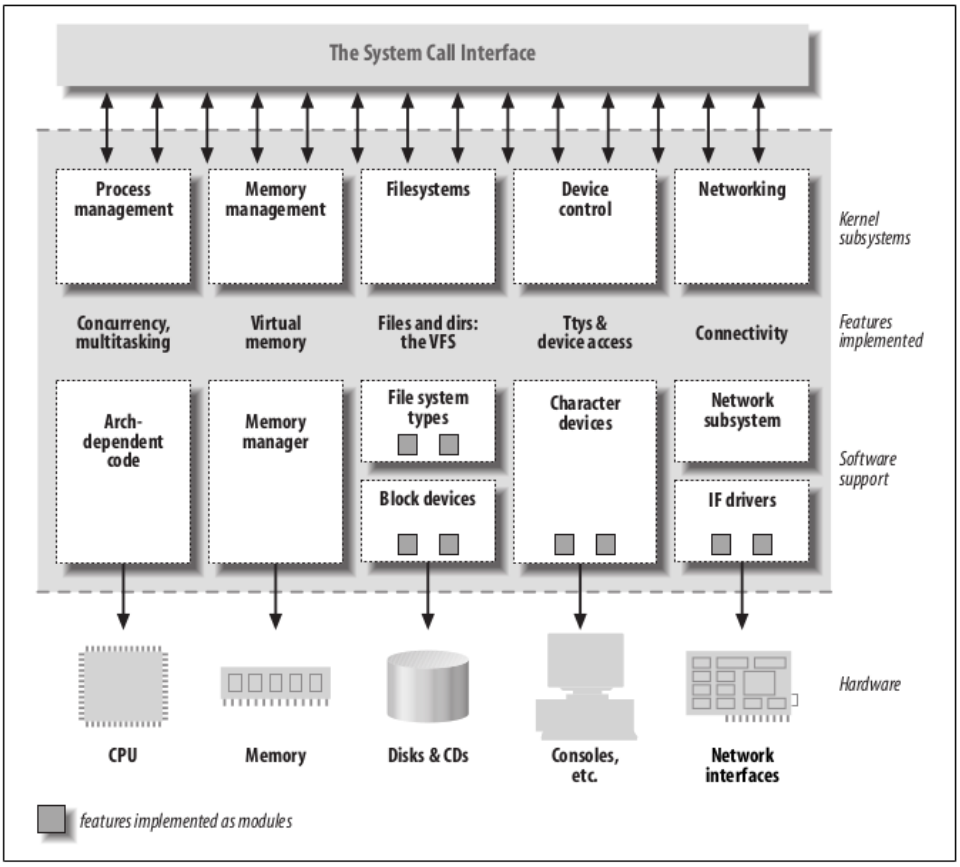

Figura 1: Um diagrama de blocos do Kernel.

Cada pedaço de código que pode ser adicionado ao kernel em tempo de execução é chamado de módulo. O kernel do Linux oferece suporte para vários tipos (ou classes) de módulos diferentes, incluindo, mas não limitado a, drivers de dispositivo. Cada módulo é composto de código objeto (não vinculado a um executável completo) que pode ser vinculado dinamicamente ao kernel em tempo de execução digitando insmod e pode ser desvinculado digitando rmmod.

A Figura 1 identifica diferentes classes de módulos responsáveis por tarefas específicas – diz-se que um módulo pertence a uma classe específica de acordo com a funcionalidade que oferece. A colocação dos módulos na Figura 1 cobre as classes mais importantes, mas está longe de ser completa porque cada vez mais funcionalidades no Linux estão sendo modularizadas.

3 Classes de Devices e Módules

A fomra como o Linux vê os dispositivos distingue entre três tipos fundamentais de dispositivos. Cada módulo geralmente implementa um desses tipos e, portanto, é classificável como **char module**, **block module** ou **network module**. Essa divisão de módulos em diferentes tipos, ou classes, não é rígida; o programador pode optar por construir módulos enormes implementando diferentes drivers em um único pedaço de código. Essas três classes são:

Character devices: Um dispositivo de caractere (char) é aquele que pode ser acessado como um fluxo de bytes (como um arquivo); um driver char é responsável por implementar esse comportamento. Esse driver geralmente implementa pelo menos as chamadas de sistema, tais como open, close, read e write. O console de texto (/dev/console) e as portas seriais (/dev/ttyS0) são exemplos de dispositivos char, pois são bem representados pela abstração de fluxo. Os dispositivos Char são acessados via sistema de arquivos, tais como /dev/ttyUSB0 e /dev/lp0.

Block devices: Assim como os dispositivos char, os dispositivos de bloco são acessados via sistema de arquivos no diretório /dev. Um dispositivo de bloco é um dispositivo (por exemplo, um disco) que pode hospedar um sistema de arquivos. Na maioria dos sistemas Unix, um dispositivo de bloco só pode lidar com operações de E/S que transferem um ou mais blocos inteiros, que geralmente têm 512 bytes (ou uma potência maior de dois) bytes de comprimento.

Network interfaces: Qualquer transação de rede é feita por meio de uma interface, ou seja, um dispositivo capaz de trocar dados com outros hosts. Normalmente, uma interface é um dispositivo de hardware, mas também pode ser um dispositivo de software puro, como a interface de loopback. Uma interface de rede é responsável por enviar e receber pacotes de dados, acionados pelo subsistema de rede do kernel, sem saber como as transações individuais são mapeadas para os pacotes reais que estão sendo transmitidos.

Existem outras maneiras de classificar os módulos de driver que são ortogonais aos tipos de dispositivos acima. Em geral, alguns tipos de drivers funcionam com camadas adicionais de funções de suporte ao kernel para um determinado tipo de dispositivo. Por exemplo, pode-se falar de módulos de barramento serial universal (USB), módulos seriais, módulos SCSI e assim por diante. Cada dispositivo USB é acionado por um módulo USB que funciona com o subsistema USB, mas o próprio dispositivo aparece no sistema como um char device (uma porta serial USB, digamos), um dispositivo de bloco (um leitor de cartão de memória USB) ou um dispositivo de rede (uma interface Ethernet USB).

4 Configurando, Compilando e Instalando os Módulos

> **Atenção:** Gerar a imagem do kernel e o sistema de arquivo conforme realizado nas práticas 03 e 04.

No terminal do Linux entre no diretório do código-fonte do kernel ou do ambinete do buildroot e então entre no menu de configurações do kernel, para realizar alterações para o módulo de usb_storage, ou seja para poder conectar um pendrive no usb da placa:

```
$ cd ~/XXX/lab_03/linux
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- menuconfig
```

Em seguida acesse o menu “Device Drivers” e o submenu “SCSI device support” e marque com um “M”, para o kernel inicialize esse dispositivo no modo módulo e não no modo in-built, como visto da Figura 2.

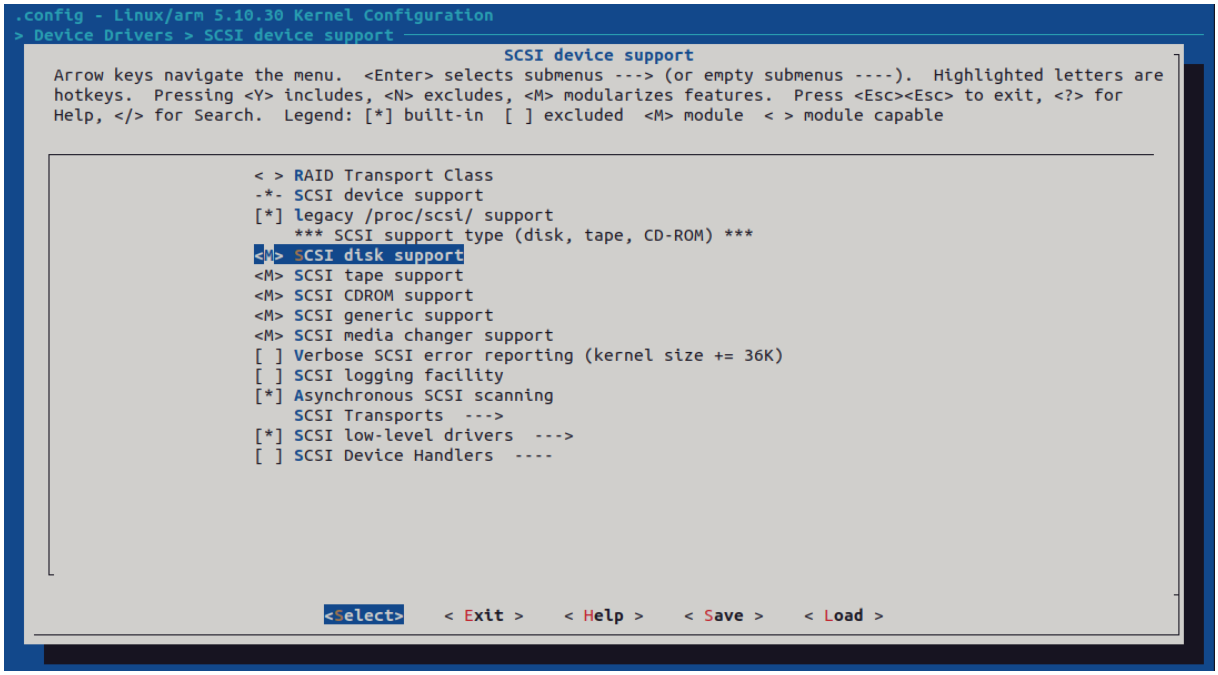

Figura 2: Processos rodando no linux gerado.

Ainda no menu “Device Drivers”, acesse o submenu “USB support” e marque com um "M", o suporte a dispositivos de armazenamento USB, opção “USB Mass Storage support” como mostrado na Figura 3.

Salve e saia do menu de configuração. Agora o kernel está pronto para ser compilado, então execute o seguinte comando::

```
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- modules -j4
```

E agora instale os módulos no rootfs.

```
$ make ARCH=arm INSTALL_MOD_PATH=/XXX/rootfs modules_install
```

Perceba que o kernel instalou os módulos em lib/modules/

```
$ ls nfs/lib/modules/5.10.131/kernel/drivers/usb/storage/
usb-storage.ko
```

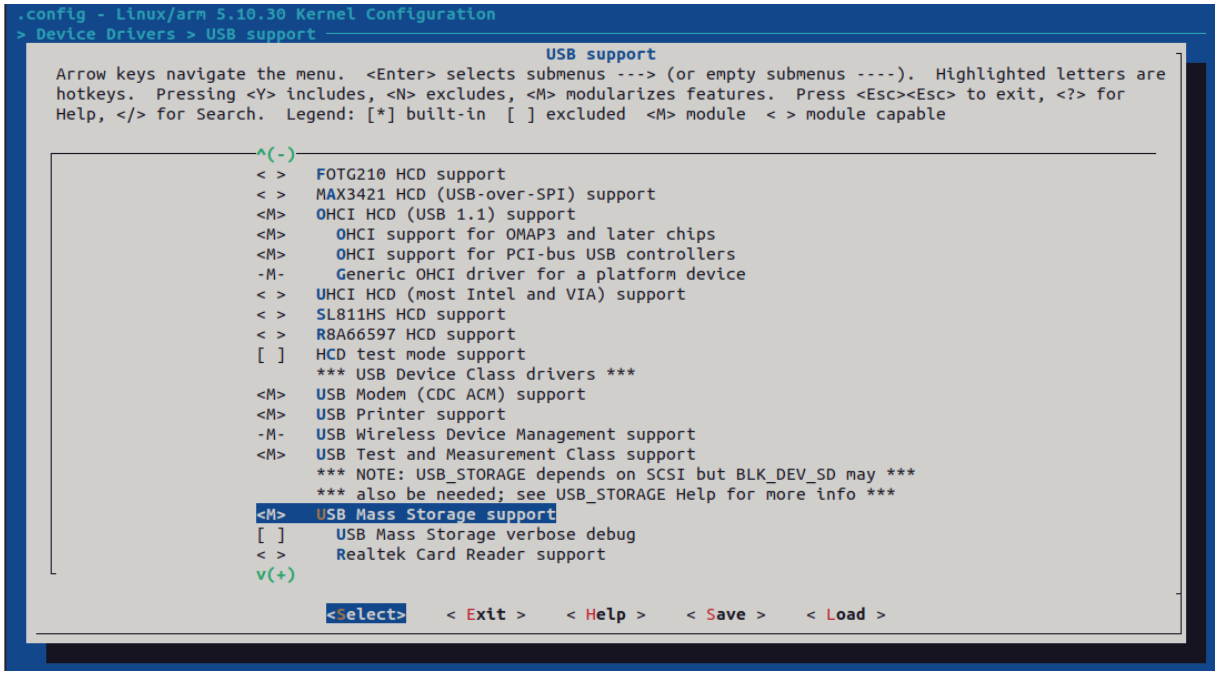

Figura 3: Processos rodando no linux gerado.

Inicialize a placa Insira um pendrive na porta usb da Beaglebone, perceba que o linux não conseguiu identificar o pendrive. Então insira o modulo com os seguintes comandos:

```
$ insmod /lib/modules/4.1.30/kernel/drivers/usb/storage/usb-storage.
   ko
$ modprobe usb-storage
```

Agora o Linux iniciou o modulo usb de armazenamento e está pronto para ser utilizando, como mostra a Figura 4

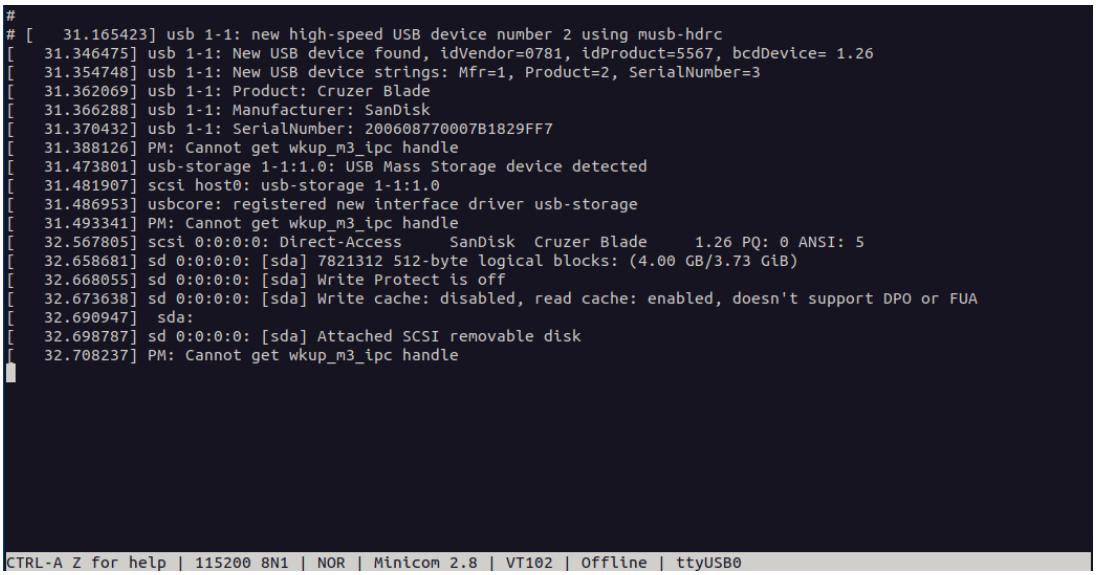

Figura 4: Processos rodando no linux gerado.

5 Construindo e Rodando Módulos

Nestas seção, vamos construir e executar um módulo completo (embora relativamente sem utilidade) e examinar alguns dos códigos básicos compartilhados por todos os módulos. Desenvolver esse conhecimento é uma base essencial para qualquer tipo de driver modular. Para evitar lançar muitos conceitos de uma só vez, vamos focar apenas sobre módulos, sem se referir a nenhuma classe de dispositivo específica.

5.1 Configurando seu Módulo de Teste

O módulo de exemplo devem funcionar com quase qualquer kernel 5.10.x, incluindo aqueles fornecidos por fornecedores de distribuição. No entanto, recomendamos que você utilize o kernel que você baixo para a prática 03.

5.2 O Módulo Hello

Todas as aulas de programação começam com um exemplo "hello world" como forma de mostrar o programa mais simples possível. Esta prática trata de módulos do kernel em vez de programas; então, o código a seguir é um módulo "hello world" completo. Nesta prática iremos criar o driver seguindo os seguintes passos:

1. Crie um diretório para o driver compilado:

```
$ mkdir /XXX/rootfs/lib/modules/5.10.131/kernel/drivers/hello
```

2. Crie um diretório para o código fonte:

```
$ mkdir /XXX/driver_hello/
$ cd driver_hello
```

3. Crie o arquivo hello.c em /driver_hello, com o seguinte conteúdo:

```
#include <linux/module.h>
#include <linux/kernel.h>
#error Are we building this file?
static int hello_init(void){
        printk(KERN_ALERT ``Hello, world\n'');
        return(0);
}
static void hello_exit(void){
        printk(KERN_ALERT ``Goodbye, cruel world\textbackslash{}n'');
}
module_init(hello_init);
module_exit(hello_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Author name <author@email.com>");
MODULE_DESCRIPTION("Test kernel module");
MODULE_VERSION("1.0");
```

4. Altere as informações do módulo para incluir seus dados.

5. Após o #include é adicionado a seguinte linha: #error Are we building this file?

   – Quando compilado, isso gerará um erro que provará que você está construindo este arquivo.

   – Mais tarde, quando tiver certeza de que seu arquivo está sendo compilado corretamente, você comenta essa linha, mas no momento ela servirá como nosso teste de que o processo está funcionando.

6. Crie Makefile em /XXX/driver_hello/ com o seguinte conteúdo:

```
# Makefile for driver
# if KERNELRELEASE is defined, we've been invoked from the
# kernel build system and can use its language.
ifneq (${KERNELRELEASE},)
        obj-m := hello.o
        # Otherwise we were called directly from the command line.
        # Invoke the kernel build system.
else
        KERNEL_SOURCE := ${HOME}/XXX/lab_03/linux
        PWD := $(shell pwd)
        # Linux kernel 5.10 (one line)
        CC=${HOME}/XXX/toolchain/arm-linux-gnueabihf/bin/arm-linux-
           gnueabihf-
        BUILD=boneblack
        CORES=4
        image=zImage
        PUBLIC_DRIVER_PWD=${HOME}/XXX/roots/lib/modules/5.10.131/
           kernel/drivers/hello
default:
        # Trigger kernel build for this module
        ${MAKE} -C ${KERNEL_SOURCE} M=${PWD} -j${CORES} ARCH=arm \
        LOCALVERSION=${BUILD} CROSS_COMPILE=${CC} ${address} \
        ${image} modules
        # copy result to public folder
        cp *.ko ${PUBLIC_DRIVER_PWD}
clean:
        ${MAKE} -C ${KERNEL_SOURCE} M=${PWD} clean
endif
```

- **Importante:**

  – O nome do arquivo deve ser Makefile (o kernel é case sensitive!); não o nomeie como “makefile”.

  – A instrução “CC” é uma linha, sem espaços ou avanços de linha.

- Leia os comentários para entender o que está acontecendo. Os básicos são:

- Caso a seguinte afirmação seja falsa: ifneq (${KERNELRELEASE},)
  então o make irá executar a parte "else" da declaração "ifneq". Isso configura parâmetros para o sistema de compilação do kernel e, em seguida, chama o sistema de compilação do kernel solicitando que ele compile essa pasta.
- A compilação do kernel começa a ser executada (via make) na pasta deste device driver.
- Caso a seguinte afirmação seja verdadeira: ifneq (${KERNELRELEASE},)
  então ele adiciona nosso(s) driver(s) à variável obj-m, que é usada pelo resto do sistema de construção do kernel para construir drivers.
- Observe que este Makefile sozinho não é suficiente para construir seu driver por conta própria; ele aproveita o sistema geral de compilação do kernel.

7. Teste se seu Makefile funciona para construir seu código fazendo com que a diretiva #error quebre a build:

```
driver_hello$ make
# Trigger kernel build for this module
make -C /home/heldercs/UFC/disciplinas/QXD0150/lab/lab_03/linux M=/
   home/heldercs/UFC/disciplinas/QXD0150/lab/lab_08/driver_hello -j4
   ARCH=arm \
LOCALVERSION=boneblack CROSS_COMPILE=/home/heldercs/UFC/disciplinas/
   QXD0150/lab/toolchain/arm-linux-gnueabihf/bin/arm-linux-gnueabihf-
     \
zImage modules
make[1]: Entering directory '/home/heldercs/UFC/disciplinas/QXD0150/
   lab/lab_03/linux'
Kernel: arch/arm/boot/Image is ready
CC [M]  /home/heldercs/UFC/disciplinas/QXD0150/lab/lab_08/
   driver_hello/hello.o
/home/heldercs/UFC/disciplinas/QXD0150/lab/lab_08/driver_hello/hello.
   c:7:2: error: #error Are we building this file?
#error Are we building this file?
^~~~~
make[2]: *** [scripts/Makefile.build:280: /home/heldercs/UFC/
   disciplinas/QXD0150/lab/lab_08/driver_hello/hello.o] Error 1
make[1]: *** [Makefile:1825: /home/heldercs/UFC/disciplinas/QXD0150/
   lab/lab_08/driver_hello] Error 2
make[1]: Leaving directory '/home/heldercs/UFC/disciplinas/QXD0150/
   lab/lab_03/linux'
make: *** [Makefile:19: default] Error 2
```

Você deve ver a mensagem **error: #error Are we building this file?**. Se sim, ótimo! Você está construindo o arquivo que espera!
Comente a instrução #error em seu hello.c e continue; caso contrário, verifique se o Makefile e a pasta de compilação do kernel estão corretos.

8. Construa o driver somente rodando make, quando tiver sucesso, ficará assim:

```
# Trigger kernel build for this module
make -C /home/heldercs/UFC/disciplinas/QXD0150/lab/lab_03/linux M=/
   home/heldercs/UFC/disciplinas/QXD0150/lab/lab_08/driver_hello -j4
   ARCH=arm \
LOCALVERSION=boneblack CROSS_COMPILE=/home/heldercs/UFC/disciplinas/
   QXD0150/lab/toolchain/arm-linux-gnueabihf/bin/arm-linux-gnueabihf-
     \
zImage modules
make[1]: Entering directory '/home/heldercs/UFC/disciplinas/QXD0150/
   lab/lab_03/linux'
Kernel: arch/arm/boot/Image is ready
CC [M]  /home/heldercs/UFC/disciplinas/QXD0150/lab/lab_08/
   driver_hello/hello.o
MODPOST /home/heldercs/UFC/disciplinas/QXD0150/lab/lab_08/
   driver_hello/Module.symvers
CC [M]  /home/heldercs/UFC/disciplinas/QXD0150/lab/lab_08/
   driver_hello/hello.mod.o
Kernel: arch/arm/boot/zImage is ready
LD [M]  /home/heldercs/UFC/disciplinas/QXD0150/lab/lab_08/
   driver_hello/hello.ko
make[1]: Leaving directory '/home/heldercs/UFC/disciplinas/QXD0150/
   lab/lab_03/linux'
# copy result to public folder
cp *.ko /home/heldercs/UFC/disciplinas/QXD0150/lab/nfs/lib/modules
   /5.10.131/kernel/drivers/hello
```

- Confira que o arquivo .ko foi corretamente copiado para o diretório /XXX/roofs/lib/modules/5.10.131/kernel/drivers/hello

```
$ ls nfs/lib/modules/5.10.131/kernel/drivers/hello/
hello.ko
```

5.3 Manipulando o Driver

Para usar o driver, ele deve estar disponível na placa Beaglebone em tempo de execução. No destino, mude para o diretório que contém o arquivo .ko. Se montado via NFS:

```
$ cd lib/modules/5.10.131/kernel/drivers/hello/
```

Liste os módulos carregados na placa, e você verá que o helloDriver não está listado:

```
$ lsmod
```

Carregue o driver:

```
$ insmod hello.ko
```

Você pode não ver nenhuma saída na tela, pois o driver imprime apenas no log do kernel. Para ver isso, talvez seja necessário executar dmesg:

```
$ dmesg | tail -1
[ 938.788651] Hello, wolrd
```

Se você vir insmod: cannot insert 'testdriver.ko': invalid module format, provavelmente significa que o kernel atual foi compilado com uma string de versão diferente da que seu host está compilando atualmente. Reconstrua e baixe o kernel ou altere o host para compilar a mesma versão que o destino (encontrado executando uname -r). Caso você vir uma mensagem loading out-of-tree module taints kernel, você pode ignorá-la.

Veja o módulo carregado na placa, então você deve ver o hello carregado.

```
$ lsmod
```

Para remover o driver na placa, você deve executar o seguinte comando:

```
$ rmmod hello
Goodbye, cruel world
```