Desenvolvendo para o Kernel Linux

Módulos e Drivers

Allef P. Araujo

Espaços de kernel e de usuário

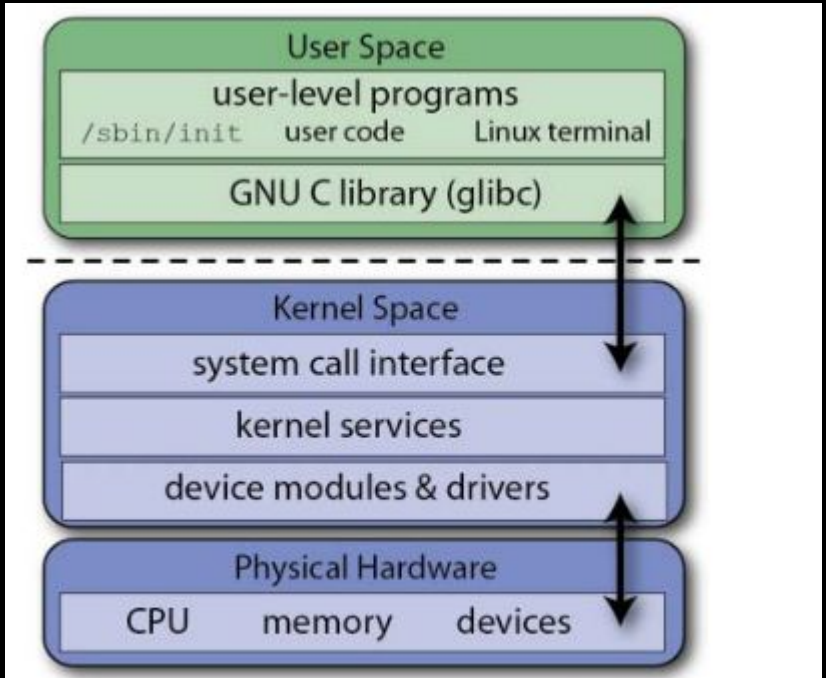


Figure 16-1: The Linux kernel and user space architecture

Módulos do kernel Linux

- Porta de entrada para desenvolvimento em espaço de kernel.

- Podem ser compilados junto ao kernel e incluídos na imagem ou compilados de forma separada e carregados dinamicamente.
    - Faremos de forma separada.

- Uma vez carregado, um módulo roda em espaço de kernel, com o mesmo nível de privilégio e acesso a todos os símbolos que o kernel exporta.

- Drivers são implementados como módulos do kernel.

Quando eu preciso criar um módulo/driver?

- Quase nunca!
    - Possivelmente já existe um driver no kernel para o dispositivo que você quer usar

- Adicionar suporte a um novo dispositivo de hardware no kernel.
    - Principalmente fabricantes de hardware

- Extender o comportamento do kernel para propósitos específico.
    - Ex: Alguns fabricantes de celular implementam funcionalidades de segurança em espaço de kernel.

Anatomia de um módulo

```c
#include <linux/init.h> // kernel macros e.g. __init and __exit
#include <linux/module.h> // We are writing a module
#include <linux/kernel.h> // It is kernel work, so we need kernel data types and functions

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Allef P. Araujo"); // TODO: Change to your name
MODULE_DESCRIPTION("My First Kernel Module");
MODULE_VERSION("1.0");

// Initialiaztion function
static int __init hello_1_init(void)
{
        return 0;
}

static void __exit hello_1_exit(void)
{
}

module_init(hello_1_init);
module_exit(hello_1_exit);
```

Makefile

```
obj-m += hello-1.o

PWD := $(CURDIR)
KDIR ?= "/lib/modules/$(shell uname -r)/build"

all:
	make -C $(KDIR) M=$(PWD) modules

clean:
	make -C $(KDIR) M=$(PWD) clean
```

Particularidades

- Podemos desenvolver módulos em C ou Rust (Versões mais recentes do kernel).
- Não tem libc! Nem outras bibliotecas de espaço de usuário.
- Temos que usar as versões de bibliotecas e funções providas pelo próprio kernel.
    - printk em vez de printf.
    - kmalloc em vez de malloc.

Particularidades

- Outras implementações particulares do kernel:
  - Kernel Threads (kthreads em vez de pthreads)
  - Listas Ligadas
  - Árvores binárias
  - Mapas

Compilação Cruzada

Hardware

- Led: Pino 11 - GPIO17
- Botão: Pino 13 - GPIO27

Instruções gerais

- Neste treinamento vamos compilar os nossos módulos fora da árvore do kernel.

- É necessário compilar os módulos utilizando os headers compatíveis com a versão da build que está rodando na placa.
    - Precisamos dos headers do kernel compilado disponíveis na máquina de compilação.
    - Para isto, vamos compilar o kernel.

Informações do kernel

- Precisamos de algumas informações a respeito do kernel rodando em nossa placa.

- Conecte-se a placa via ssh:
  - $ ssh pi@raspberrypi.local

- Anote a versão do kernel retornada por:
  - $ uname -r

- Anote a arquitetura do kernel retornada por:
  - $ uname -m

Toolchain

- Precisamos instalar uma toolchain para ARM.

- Se a arquitetura do kernel for aarch64 (64 bits):
  - $ sudo apt install gcc-aarch64-linux-gnu

- Se for um kernel 32 bits:
  - $ sudo apt install gcc-arm-linux-gnueabihf

Fontes do kernel

- Baixe os fontes do kernel da Raspberry Pi:
  - $ git clone https://github.com/raspberrypi/linux.git
- Faça checkout para a branch compatível com a versão que você encontrou com o comando ‘uname -r’. Supondo que você encontrou 6.1.21:
  - $ git checkout rpi-6.1.y
- Pode ser que a branch já tenha evoluído e o topo não seja mais compatível com o kernel da sua placa. Neste caso, podemos pegar o commit exato da versão 6.1.21. Anote o sha do commit retornado por:
  - git log - -oneline | grep 6.1.21

Fontes do kernel

- Faça checkout para o comando específico:
  - $ git checkout <commit_sha>

Configurando o kernel

- Vamos copiar as configurações do kernel que já está instalado na placa de desenvolvimento.

- Via ssh na placa rode:
  - $ sudo modprobe configs

- Na pasta raíz do kernel que você baixou rode:
  - $ scp pi@raspberrypi.local:/proc/config.gz .
  - $ zcat config.gz > .config

Variáveis de ambiente e compilação

- Se o kernel for 64 bits:
  - $ export ARCH=arm64
  - $ export CROSS_COMPILE=aarch64-linux-gnu-
- Se o kernel for 32 bits:
  - $ export ARCH=arm
  - $ export CROSS_COMPILE=arm-linux-gnueabihf-
- Compile
  - $ make -j8

Mão na massa

Compilando um módulo

- Execute os seguintes comandos:
    - $ source setenv.sh *# Disponível no repositório de exercícios*
    - $ make

Comandos Básicos

- Instalar um módulo:
  - $ sudo insmod <mod_name>.ko
- Remover um módulo:
  - $ sudo rmmod <mod_name>
- Listar módulos instalados:
  - $ sudo lsmod
- Informações do módulo:
  - $ modinfo <mod_name>.ko
- Logs do kernel (últimas 5 linhas):
  - $ dmesg -t | tail -5
- Logs do kernel continuamente:
  - $ dmesg -w

Exercício 1

- Pasta ex/01

- Completar o código nos locais indicados com logs utilizando a função printk.
    - `printk(KERN_INFO "Message: %s!\n", arg);`

- Compilar o módulo.

- Instalar e exercitar os comandos básicos.

Parâmetros do módulo

- É possível parametrizar o nosso módulo durante a instalação. Ex:
    - $ sudo insmod module.ko param=value

- Adicionando um parâmetro ao módulo:

```
module_param(param, charp, S_IRUGO); // S_IRUGO can be read/not changed, charp means char pointer
MODULE_PARM_DESC(param, "The param description.");
```

Exercício 2

- Pasta ex/02

- O módulo possui uma variável ‘name’ que é impressa na inicialização e remoção do módulo.

- Transforme a variável em parâmetro para ser modificado na instalação do módulo. Para isso, procure as instruções nos comentários TODO.

sysfs

- Podemos interagir com o nosso módulo por meio do sysfs.

```
pi@raspberrypi:~ $ ls /sys/module/hello_2/
coresize  initsize   notes       refcnt    srcversion  uevent
holders   initstate  parameters  sections  taint       version
pi@raspberrypi:~ $ cat /sys/module/hello_2/parameters/name
allef
pi@raspberrypi:~ $ cat /sys/module/hello_2/version
1.0
pi@raspberrypi:~ $
```

- Documentação?
  - https://www.kernel.org/doc/Documentation/ABI/stable/sysfs-module

sysfs

- Em versões mais recentes do kernel devemos exportar as interfaces dos nossos drivers no /dev, respeitando as classes de dispositivo.
    - Caractere (/dev)
    - Bloco (/dev)
    - Rede (interface via socket)
- Hoje vamos brincar com interfaces no sysfs e no /dev!

Drivers

- Módulos que abstraem o hardware.
- Leds, porta serial, botões, teclado, etc.
- Vamos fazer 2 drivers simples:
    - Driver de Led (Pino 11 - GPIO 17)
    - Ler botão por interrupção e trocar o estado do LED (GPIO 17 e GPIO 27)

Exercício 3

- Driver de Led.
- Led conectado à GPIO 17 do processador.
- O kernel Linux abstrai o acesso a GPIO por meio da lib disponível em linux/gpio.h.
  - Também é possível acessar via endereço de memória (Memory-Mapped I/O), forma mais trabalhosa e menos flexível.
- Nosso driver será implementado como um dispositivo de caractere com interface no /dev.

Exercício 3

- linux/gpio.h

```
static inline bool gpio_is_valid(int number)
static inline int  gpio_request(unsigned gpio, const char *label)
static inline int  gpio_export(unsigned gpio, bool direction_may_change)
static inline int  gpio_direction_input(unsigned gpio)
static inline int  gpio_get_value(unsigned gpio)
static inline int  gpio_direction_output(unsigned gpio, int value)
static inline int  gpio_set_debounce(unsigned gpio, unsigned debounce)
static inline int  gpio_sysfs_set_active_low(unsigned gpio, int value)
static inline void gpio_unexport(unsigned gpio)
static inline void gpio_free(unsigned gpio)
static inline int  gpio_to_irq(unsigned gpio)
```

Exercício 3 - Inicialização do Módulo

- Registrar o driver de caractere

```
major = register_chrdev(0, DEVICE_NAME, &led_fops);
if (major < 0) {
            printk("LED_DRIVER: Failed to registet Led device major:  %d\n", major);
            return -ENODEV;
}

printk("LED_DRIVER: I've got a major  %d\n", major);
cls = class_create(THIS_MODULE, DEVICE_NAME);
cls->dev_uevent = set_permissions;
device_create(cls, NULL, MKDEV(major, 0), NULL, DEVICE_NAME);
printk("LED_DRIVER: Led created on /dev/ %s\n", DEVICE_NAME);
```

Exercício 3 - Inicialização do Módulo

- Configurar a GPIO onde o LED está conectado

```
    gpio_request(gpioLed, "led");                   // request LED GPIO
    gpio_direction_output(gpioLed, 0);              // set in output mode and
off
```

Exercício 3 - Encerramento do Módulo

- Se o Led estiver ligado, desligar e liberar a GPIO

```
    gpio_set_value(gpioLed, 0);                 // turn the LED off
    gpio_free(gpioLed);                         // free the LED GPIO
    gpio_free(gpioButton);                      // free the button GPIO
    printk(KERN_INFO "GPIO_TEST: Goodbay from the LKM! \n");
```

Exercício 3 - File operations

- Nosso LED é exportado como um arquivo /dev/led
- Nós interagimos com o nosso LED utilizando as operações padrão de escrita/leitura em arquivos: open, read, write, close.
- Precisamos implementar estas operações:

```
static struct file_operations led_fops = {
    .read = led_read,
    .write = led_write,
    .open = led_open,
    .release = led_release,
};
```

Exercício 3 - File operations

```
struct file_operations {
   struct module *owner;
   loff_t (*llseek) (struct file *, loff_t, int);
   ssize_t (*read) (struct file *, char *, size_t, loff_t *);
   ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
   int (*readdir) (struct file *, void *, filldir_t);
   unsigned int (*poll) (struct file *, struct poll_table_struct *);
   int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
   int (*mmap) (struct file *, struct vm_area_struct *);
   int (*open) (struct inode *, struct file *);
   int (*flush) (struct file *);
   int (*release) (struct inode *, struct file *);
   int (*fsync) (struct file *, struct dentry *, int datasync);
   int (*fasync) (int, struct file *, int);
   int (*lock) (struct file *, int, struct file_lock *);
     ssize_t (*readv) (struct file *, const struct iovec *, unsigned long,
      loff_t *);
     ssize_t (*writev) (struct file *, const struct iovec *, unsigned long,
      loff_t *);
};
```

Exercício 3 - File operations

- Fazer o exercício 3 de acordo com as instruções dos comentários marcados com TODO.
- Se necessário utilizar o exemplo modules/chardev-led disponibilizado na pasta de exercícios.
- Testar na placa.
- Analise o código do aplicativo disponibilizado na pasta ~/app do target. Execute o programa.

Exercício 4

- Vamos criar um módulo do kernel que configura um pino de GPIO como entrada e captura o pressionar de um botão via interrupção.

```
irqNumber = gpio_to_irq(gpioButton);          // map GPIO to IRQ number

// This next call requests an interrupt line
result = request_irq(irqNumber,             // interrupt number requested
                     (irq_handler_t) erpi_gpio_irq_handler,  // handler function
                     IRQF_TRIGGER_RISING,    // on rising edge (press, not release)
                     "erpi_gpio_handler",    // used in /proc/interrupts
                     NULL);
```

Exercício 4

- O que eu vou fazer quando a interrupção ocorrer?

```
static irq_handler_t erpi_gpio_irq_handler(unsigned int irq, void *dev_id, struct
pt_regs *regs) {

    // TODO: Invert Led state

    return (irq_handler_t) IRQ_HANDLED;          // anounce IRQ handled
}
```

Exercício 4

- Vamos implementar o que falta no exercício seguindo os comentários marcados como TODO no exercício 4.

- Testar no target.

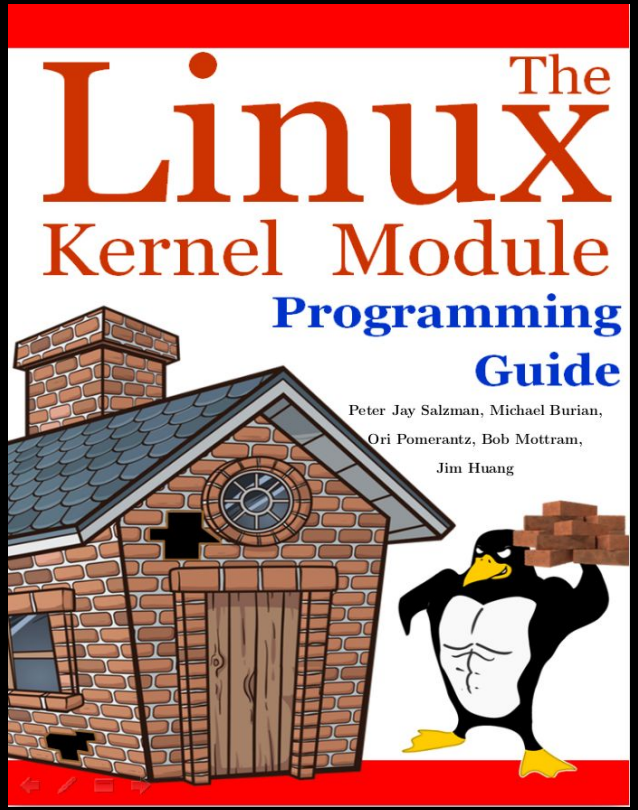
- Alguém não sabe o que é uma interrupção?

O que estudar depois

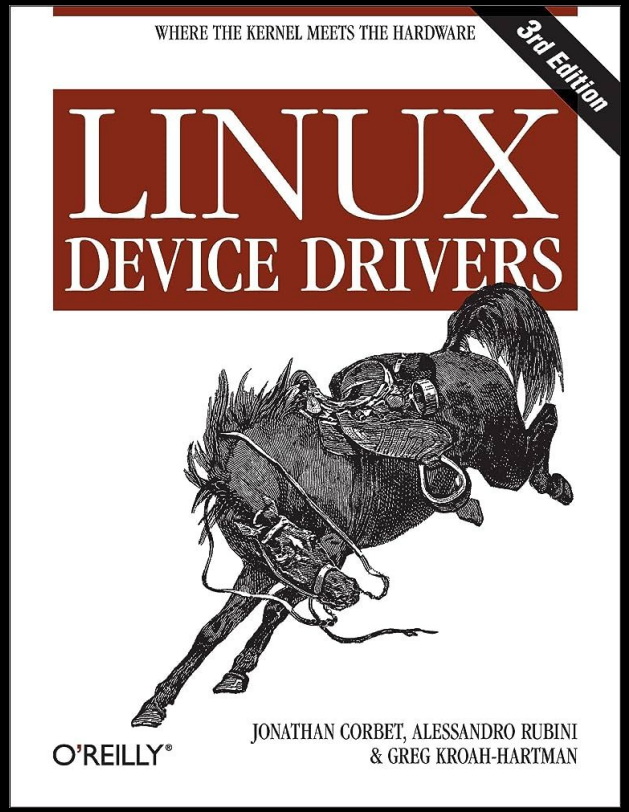
- Drivers de dispositivos de caracteres. Ver exemplos.
- Device Tree.
- Kernel Threads.
- Kprobes.
- kernelnewbies.org
- Principalmente: Ler o código de drivers existentes!

Bibliografia disponível

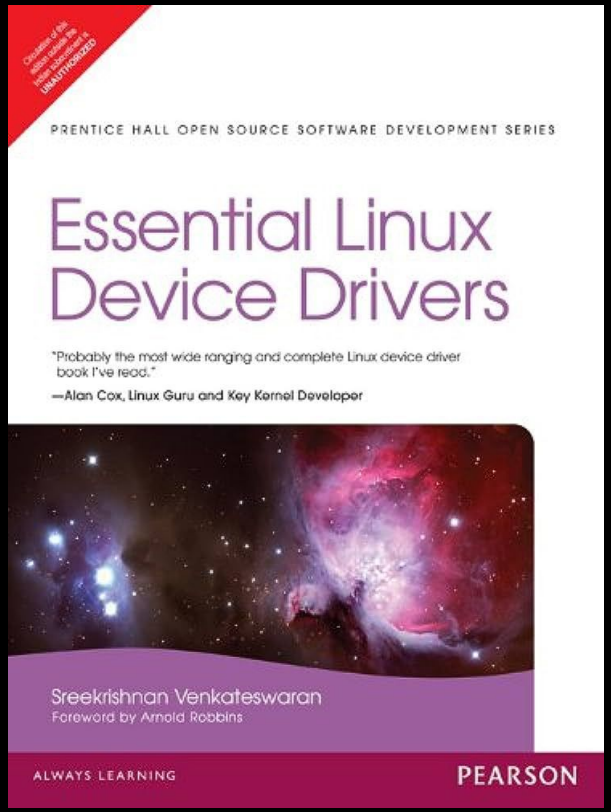
Livro: The LKM Programming Guide



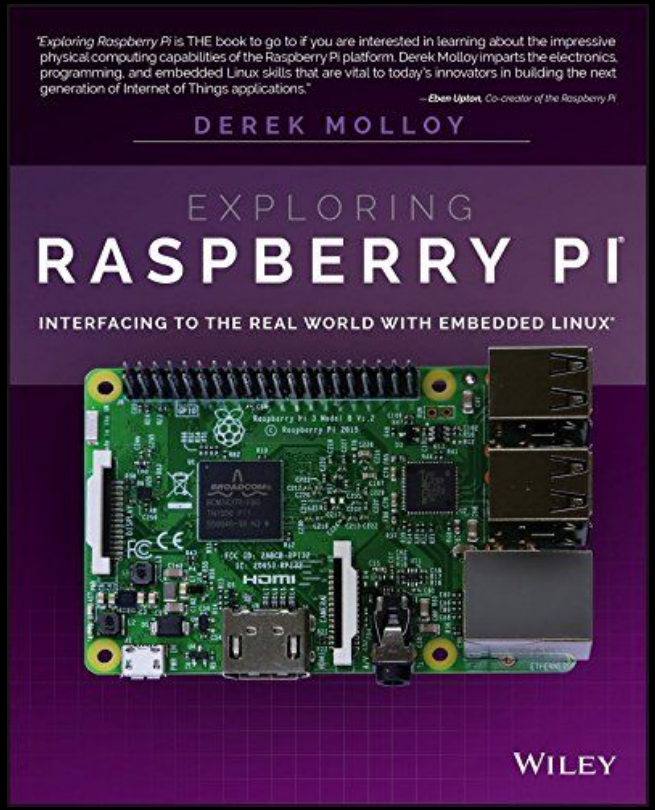
Livro: Linux Device Drivers



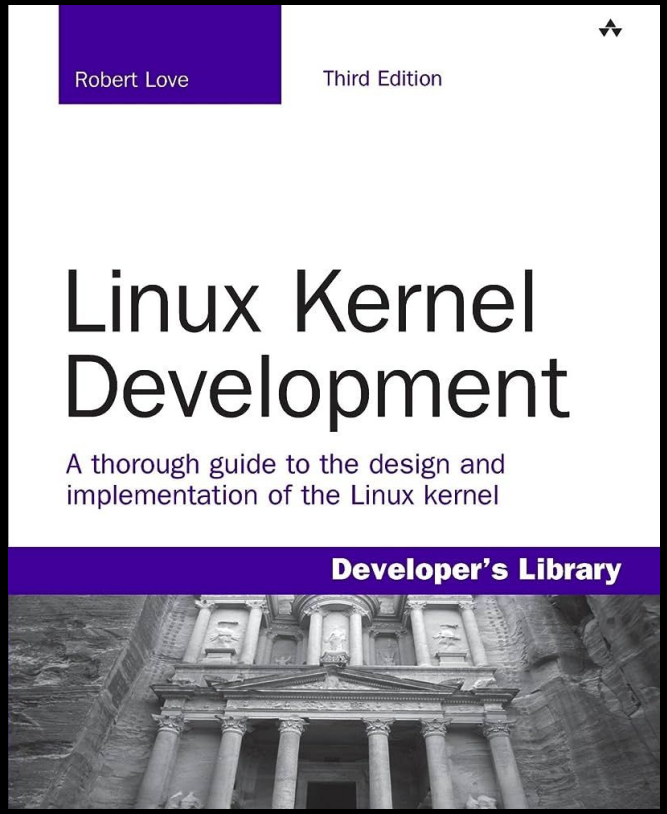
Livro: Essential Linux Device Drivers



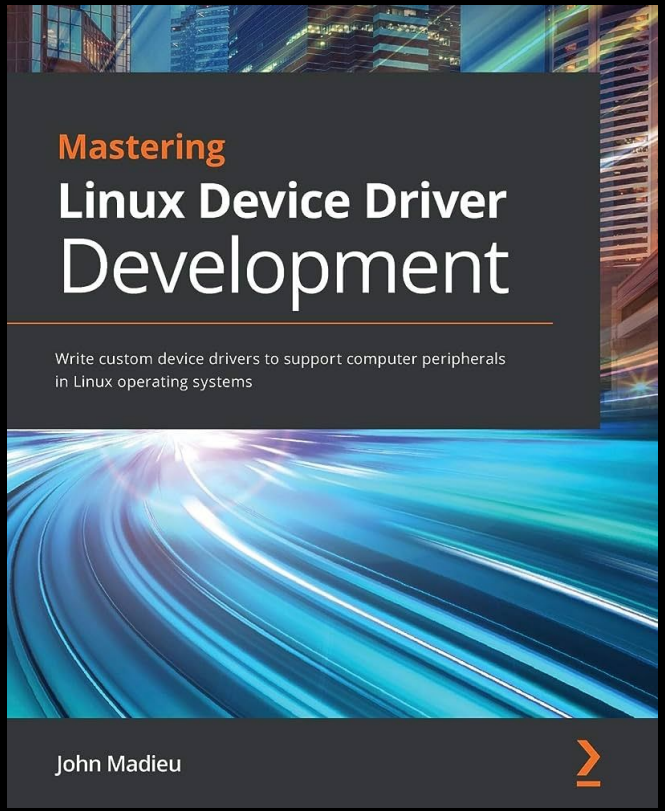
Livro: Exploring Raspberry Pi



Livro: Linux Kernel Development



Livro: Mastering Linux Device Driver Development



Treinamento da Embedded Labworks (Sérgio Prado)

- Slides: https://e-labworks.com/training/ldd/slides.pdf
- Página do curso: https://e-labworks.com/en/treinamentos/linux-device-drivers/