Lecture 18: Device Drivers

Fall 2019
Jason Tang

Slides based upon Linux Device Drivers, 3rd Edition
http://lwn.net/Kernel/LDD3/

Topics

- Parts of a Linux Device Driver
- Character Devices
- File Operations
- User Space Context

Linux Devices

- In Linux (and other Unix-based systems), block and character devices have major and minor device numbers, traditionally as follow:
  - major number: identifies which driver to handle device
  - minor number: identifies which instance of device is being managed
- Module is any bit of runtime loaded kernel code; a device driver is a module that controls access to a device
- Usually, kernel instantiates multiple instances of same device driver when dealing with duplicate hardware

Linux Devices

```
$ ls -l /dev/tty[0-3]
crw--w---- 1 root tty 4, 0 Mar 14 15:45 /dev/tty0
crw-rw---- 1 root tty 4, 1 Mar 14 15:45 /dev/tty1
crw-rw---- 1 root tty 4, 2 Mar 14 15:45 /dev/tty2
crw-rw---- 1 root tty 4, 3 Mar 14 15:45 /dev/tty3
```

- In this example, the same serial driver is handling multiple devices, all with major number 4
  - See include/uapi/linux/major.h for mapping of major numbers to devices
- Core kernel will call serial driver's probe function multiple times, once for each serial hardware detected
  - Many drivers have a global static variable that counts how many times its probe function has been invoked

Driver Cleanup

- Opposite of probe is a remove function
  - Called by kernel when device is removed (e.g., unplugged)
- When module is loaded, kernel calls module’s init function
  - For each device detected, kernel calls driver’s probe function
  - When device removed, kernel calls driver’s remove function
- When module is unloaded, kernel calls module’s exit function
  - For each device still plugged in, remove function is called first

Device Registration

- Within a module's init function, a device driver registers itself on to a bus
  - Examples of buses: USB, PCI, SCSI, Infiniband, platform, …
- As part of registration, device driver registers something that identifies its hardware from others on the same bus
  - Example: Manufacturers hardwire device codes into PCI devices
- After kernel has initialized all core code, it then scans each bus
  - For each discovered device code, it calls the registered probe function

Device Registration Example

- Example: VirtualBox emulates an Intel Pro/1000 Gigabit Ethernet adapter
  - This PCI device has the device code 0x100E
  - In this device driver's module init, it registers itself with the core kernel as capable of handing a PCI device 0x100E
- During PCI initialization, kernel scans all devices and device codes on PCI bus
  - When it finds a device with code 0x100E, it invokes the probe function registered for that code

Device Driver Operations

- Drivers typically have three parts:

Part	Usage	Example
Registration (required)	Notifies core kernel of driver ID	`pci_device_register()`
Interrupt Handling (almost always)	Handles hardware interrupts generated by device	`register_threaded_irq()`
User Space API (usually)	Lets programs interact with driver	`miscdevice_register()`

- Interrupt handling deals with responding to interrupts, DMA, and other low-level details

- User space API involves creating entries in /dev, responding to system calls, etc., so that users can actually use the hardware

Interrupt Handling

- Device driver must service interrupts generated by hardware, by installing interrupt service routines (ISR)
- Example: a serial port raises an interrupt upon input (it received electrical signals on input pins)
    - That interrupt line is associated to an interrupt request (IRQ) number
    - Device driver registers itself as a handler to that IRQ
    - Kernel invokes callback(s) that are registered to the IRQ whenever interrupt is raised

User Space API

- Application programmers need some way to interact with hardware
- Example: Applications interact with /dev/tty0 character device by:
  - Writing data to /dev/tty0 eventually sends electrical signals out serial port
  - Reading data from /dev/tty0 returns inputs from serial port
    - Many hardware serial ports have a 16-byte circular input buffer
  - Invoking an `ioctl()` (I/O control) on /dev/tty0 to change baud rate, flow control, and other low-level hardware settings

Creating Device Nodes

- Several mechanisms exist to create different types of device nodes
- All mechanisms involve registering callbacks to respond to user operations
- Within a module's probe, that driver registers callbacks
  - During module remove, driver unregisters those callbacks
  - As part of device node creation, need to specify device name ("tty0"), major number ("4"), and minor number ("0")
  - May also specify default permissions ("0660")
- Kernel code **must check and handle all return values**

File Operations

```
struct file_operations {
    struct module *owner;
    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 *);
    ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
    ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
    …
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    …
}
```

- Driver creates a `struct file_operations` (a function pointer table) and sets its fields to desired callbacks
    - If a callback is unspecified (is set to `NULL`), than that operation is not supported; kernel returns an error if a user tries to perform that operation

https://elixir.bootlin.com/linux/latest/source/include/linux/fs.h

Miscellaneous Devices

- Many steps involved when creating new character devices
- Miscellaneous device (miscdevice): simpler interface for smaller drivers
  - Represented as a `struct miscdevice`, from include/linux/miscdevice.h
  - Need a `struct file_operations` that specifies callbacks
- In Linux, all miscellaneous devices share major number 10
  - Core kernel can dynamically assign minor numbers

Creating Miscellaneous Device

```
struct miscdevice  {
    int minor;
    const char *name;
    const struct file_operations *fops;
    …
    umode_t mode;
}
```

- Set `minor` field to desired minor value, or the constant `MISC_DYNAMIC_MINOR` to let kernel choose
- Set `name` field to name of device, as it should appear in /dev
- Set `fops` to point to an instance of a `file_operations` table
- (Optional) Set `mode`, ala `chmod` command

Miscellaneous Device Example

```c
static const struct file_operations rtc_fops = {
	.owner		= THIS_MODULE,
	.llseek		= no_llseek,
	.read		= rtc_read,
#ifdef RTC_IRQ
	.poll		= rtc_poll,
#endif
	.unlocked_ioctl	= rtc_ioctl,
	.open		= rtc_open,
	.release	= rtc_release,
	.fasync		= rtc_fasync,
};

static struct miscdevice rtc_dev = {
	.minor		= RTC_MINOR,
	.name		= "rtc",
	.fops		= &rtc_fops,
};
```

```c
static int __init rtc_init(void)
{
	…
	if (misc_register(&rtc_dev)) {
		…
		return -ENODEV;
	}
	…
}

static void __exit rtc_exit(void)
{
	…
	misc_deregister(&rtc_dev);
	…
}
```

- Real-time clock (RTC) is implemented as a character device

https://elixir.bootlin.com/linux/latest/source/drivers/char/rtc.c

File Operations: Open

```
static int foo_open(struct inode *inode, struct file *filp);
```

- Callback invoked when process opens device node
  - `inode` is a pointer to the inode as exists on disk; often unused
  - `filp` is a pointer to the kernel object representing the calling process
- Return value is 0 on success, or negative on error (file will not be opened)

Error Values

- For most kernel functions, a negative return value indicates error
  - libc will set errno to the absolute value of the return value
- Example: In standard C, the `open()` function returns the newly created file descriptor (a non-negative integer) on success, or -1 on failure
  - If kernel's open callback returns 0, libc sets return value to be the file descriptor
  - If instead callback returns `-EACCESS`, libc sets return value to -1 and sets `errno` to `EACCESS`

File Operations: Release

```
static int foo_release(struct inode *inode, struct file *filp);
```

- Callback invoked when process closes device node
    - `inode` and `filp` are as per *foo*_open() callback
- Return value is 0 on successful close, or negative on error (file will not be closed)

File Operations: read and write

```
static ssize_t foo_read(struct file *filp, char __user *ubuf,
                        size_t count, loff_t *ppos);
static ssize_t foo_write(struct file *filp, const char __user *ubuf,
                         size_t count, loff_t *ppos);
```

- *foo*_read() called when process is reading from device, *foo*_write() when process is writing to device
- `filp` is same file object from *foo*_open()
- `ubuf` is pointer to user buffer of where to read/write data
- `count` is number of bytes **requested** to read / number of bytes within `ubuf`
- `ppos` is offset into file; often ignored in streaming devices

User Context

- Kernel invokes callbacks on behalf of a user process
  - This is called user context, as opposed to interrupt context
- User memory exists within its process's virtual address space, but kernel memory is in kernel's virtual address space
  - Drivers **may not** simply copy data between kernel and user memory

```
static ssize_t foo_read(struct file *filp, char __user *ubuf,
                        size_t count, loff_t *ppos) {
    memcpy(ubuf, kernel_data, sizeof(*kernel_data));
    …
}
```

Wrong! Does not work! Do not do this!

Copying Data to User Space

- When copying data into user's memory, must check that destination is valid location and entirely within process's address space

- Kernel has macro `copy_to_user()` that makes necessary checks prior to copying; it returns 0 on successful copy, non-zero on error

- Copy the smaller of *requested* amount and the amount *actually available*

- Return value from *foo*_read() is the number of bytes written to `ubuf`, or negative on error

Copying Data from User Space

- Likewise, **always** use macro `copy_from_user()` when copying data from user space into kernel memory
- Copy the smaller of *provided* amount and the space *actually available* within the kernel
- *foo*_write() returns the number of bytes in `ubuf` that was consumed

Correct Read/Write Implementations

```
static char some_kernel_buffer[80];

static ssize_t foo_read(struct file *filp, char __user *ubuf,
                        size_t count, loff_t *ppos) {
    int retval;
    if (count < sizeof(some_kernel_buffer))
        count = sizeof(some_kernel_buffer);
    retval = copy_to_user(ubuf, some_kernel_buffer, count);
    if (retval < 0)
        return -EINVAL;
    return count;
}

static ssize_t foo_write(struct file *filp, const char __user *ubuf,
                         size_t count, loff_t *ppos) {
    int retval;
    if (count < sizeof(some_kernel_buffer))
        count = sizeof(some_kernel_buffer);
    retval = copy_from_user(some_kernel_buffer, ubuf, count);
    if (retval < 0)
        return -EINVAL;
    return count;
}
```