

# CS 378 (Spring 2003)

## Linux Kernel Programming

**Yongguang Zhang**

**([ygz@cs.utexas.edu](mailto:ygz@cs.utexas.edu))**

# This Lecture

---

- Device Driver
  - Hardware Abstraction
  - Device File
  - Data Structure
  - Steps in writing a device driver
  
- Questions?

# Linux Device Driver

---

- Role of device driver in Linux
  - Hides hardware details
  - Provides well-defined interface for kernel/user
  - Manages hardware
- Usually as loadable modules
  - Some critical device drivers often compiled in

# Hardware Detail

---

- CPU side: I/O bus and I/O address
  - Address to communicate with hardware device
  - Can be mapped to physical memory address space
- Device side: hardware controller
  - Control and status registers (CSRs)
- Three ways to interact with hardware device
  - Polling (e.g., floppy driver)
  - Interrupt-driven (e.g, most other devices)
  - DMA (direct memory access)

# DMA

---

- For high throughput I/O
- RAM region accessible by both CPU and device
  - Mapped into physical address space
  - Devices to transfer data to/from RAM without CPU's intervention
- To use DMA in device driver
  - Set up DMA channel (address, registers, direction)
  - Tell the DMA controller to start
  - Prepare and register interrupt service routine
  - When transfer completes, controller will interrupt

# Types of Device Driver in Linux

---

- Character device
  - Stream of bytes
  - Read and written directly without buffering
  - Example: serial ports, monitor, keyboard
- Block device
  - Read and written in multiples of block, random access
  - Example: disk, tape, cdrom
- Network device
  - Specially handled in Linux networking stack
  - Example: Ethernet card

# Special Device File

---

- Abstraction and interface for character and block devices (but not for network device)
  - One device file corresponds to one device
  - To drive the device is to access the file: open, close, read, write, lseek, poll, etc.
- Data structure for all device file operations
  - Standard, same as any other type of file
  - Object type: `struct file_operations` (in `include/linux/fs.h`)

# Standard File Operations Structure

---

```
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 *);  
    ...  
}
```

# Device File Specifics

---

- Normal file systems
  - One `file_operations` object for each file system type
- Character device
  - One `file_operations` object for each device driver
  - Each device driver defines its own `read()`, `write()`, ...
- Block device
  - One `file_operations` object for all block device drivers
  - Additional structure for block device operations, one object per device
  - Each device driver needs only provide these operations

# Block Device File Operations

---

- In fs/block\_dev.c

```
struct file_operations def_blk_fops = {
    open:            blkdev_open,
    release:         blkdev_close,
    llseek:          block_llseek,
    read:            generic_file_read,
    write:           generic_file_write,
    mmap:            generic_file_mmap,
    fsync:           block_fsync,
    ioctl:           blkdev_ioctl,
};
```

# Block Device Specific Operations

---

- Additional operations for block device only
- Data type defined in `include/linux/fs.h` :

```
struct block_device_operations {
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
    int (*ioctl) (struct inode *, struct file *, unsigned,
unsigned long);
    int (*check_media_change) (kdev_t);
    int (*revalidate) (kdev_t);
    struct module *owner;
};
```

# /dev Directory

---

- Special directory in Linux for all device files
- The old way: through major/minor number
  - Any possible device has distinct major/minor number assigned (e.g., major number 3 for harddrive, 4 for tty)
  - Major number (1-254): to identify type of device
  - Minor number (0-255): to identify an individual device
  - Device file created by mknod command  
`mknod /dev/tty0 c 4 0`
- The new and better way: device file system

# The Old Device Files

---

- Implemented as fix-size array
  - Two arrays for character and block devices:  
static struct device\_struct chrdevs[MAX\_CHRDEV];  
static struct { ..... } blkdevs[MAX\_BLKDEV];
  - Major number corresponding to the array subscript
- Dynamic major number (0)
  - Dynamic allocated (from chrdevs[] or blkdevs[])
  - To use the device, need to check /dev special file to find out the major number allocated
- /dev directory static and crowded
  - All possible devices need a file in /dev

# Device File System

---

- devfs (device file system)
  - /dev is a special file system (c.f. /proc)
  - Each device correspond to a path name under /dev
- Dynamic file sytem
  - The device driver creates the file when it is loaded and removes the file when is unloaded
  - A file presented only if the device driver is loaded
- Benefits
  - Major/minor number is optional
  - Has no limit on number of devices

# Device File Data Structure

---

- Include `include/linux/devfs_fs_kernel.h`
  - Data type for devfs entry: `devfs_handle_t`  
`typedef struct devfs_entry * devfs_handle_t;`
  - Actual struct `devfs_entry` defined in `fs/devfs/base.c`
- Functions and operations
  - `devfs_register()`, `devfs_unregister()`
  - `devfs_mk_symlink()`, `devfs_mk_dir()`
  - `devfs_get_handle()`, `devfs_find_handle()`
  - Many many more
- Implementation: `fs/devfs/base.c`

# Registering a Device

---

- To create a device file under devfs:

```
devfs_handle_t devfs_register (  
    devfs_handle_t dir,  
    const char *name,  
    unsigned int flags,  
    unsigned int major, unsigned int minor,  
    umode_t mode,  
    void *ops,  
    void *info);
```

- To remove the device file:

```
extern void devfs_unregister (devfs_handle_t de);
```

# devfs\_register() Parameters

---

- **dir**: handle to parent directory
  - NULL means root directory (i.e., /dev)
- **name**: name of entry (device name)
- **flags**: options (usually `DEVFS_FL_DEFAULT`)
- **major and minor**: optional
- **mode**: file type (char or block) and access mode
- **ops**: the device operation structure
- **info**: pointer to private data for open file structure

# Device File Flags

---

- For device file option
  - See `DEVFS_FL*` in `include/linux/devfs_fs_kernel.h`
  - `DEVFS_FL_AUTO_OWNER`: when the device file is first open, the file's ownership is changed to the opening process, until the file is closed (ownership reverts back)
  - `DEVFS_FL_CURRENT_OWNER`: set the initial ownership to the current process that register the device
  - `DEVFS_FL_AUTO_DEVNUM`: automatically generate the device number (major/minor)
  - More

# Other Parameters

---

- **mode**: for device file type and permission
  - See `S_I*` macros in `include/linux/stat.h`
  - `S_IFCHR`: character device
  - `S_IFBLK`: block device
  - Other file types (`S_IFREG`, `S_IFDIR`, etc.)
  - And file permission bits (-rwx-)
- **ops**: pointer to device operation structure
  - Character device: must be pointer to object with type `struct file_operations`
  - Block device: must be pointer to object with type `struct block_device_operations`

# Registering a Device (Alternatives)

---

- These are for backward compatibility only
- To support old style device file under devfs
  - Find a major number (or 0 for dynamic allocation)
  - Use `devfs_register_chrdev()` or `devfs_register_blkdev()` in addition to `devfs_register()`
- To support old style device file only
  - Find a major number (or 0 for dynamic allocation)
  - Use `register_chrdev()` or `register_blkdev()` in stead of `devfs_register()`

# Steps To Write a Device Driver

---

- Write all the functions for the device operation structure
- Create and fill a struct object of the proper operation structure type
- Write an “init” function and an “exit” function to register and unregister the device driver
- Specify the “init” and “exit” functions as  
    `module_init(xxx_init)`  
    `module_exit(xxx_exit)`

# Fill the Device Operation Structure

---

- Write the required functions for the device operation structure
  - Character device: `struct file_operations`
    - `owner`, `llseek`, `read`, `write`, `readdir`, `poll`, `ioctl`, `mmap`, `open`, `flush`, `lock`, `release`, `fsync`, `fasync`, `lock`, `readv`, `writv`, `sendpage`, `get_unmapped_area`
  - Block device: `struct block_device_operations`
    - `open`, `release`, `ioctl`, `check_media_change`, `revalidate`
  - Not all functions are required if the operation isn't needed in the device driver

# Skeleton Character Device

---

- Device operation structure:

```
static struct file_operations xxx_fops =  
{  
    owner:    THIS_MODULE,  
    read:     xxx_read,  
    write:    xxx_write,  
    open:     xxx_open,  
    release:  xxx_release,  
};
```

# Skeleton “Init” Function

---

```
static int __init xxx_init(void)
{
    /* probe the hardware, request irq, ... */

    devfs_dir = devfs_mk_dir(NULL, "xxx_dir", NULL);

    devfs_handle = devfs_register(devfs_dir, "xxx",
                                DEVFS_FL_DEFAULT, XXX_MAJOR, 0,
                                S_IFCHR | S_IRUSR | S_IWUSR,
                                &xxx_fops, NULL);

    /* rest of the initial setup */
    printk( ... ); return 0;
}
```

# Skeleton “Exit” Function

---

```
static void __exit xxx_exit (void)
{
    /* clean up */

    devfs_unregister(devfs_handle);
    devfs_unregister(devfs_dir);

    /* clean up */
}

module_init(xxx_init);
module_exit(xxx_exit);
```

# Skeleton “Open” Operation

---

```
static int xxx_open(struct inode * inode, struct file * filp)
{
    /* Private data (info) is carried in filp->private_data */
    /* Check to see which device is being open (if more than one) */
    /* You can replace it with something else (device-specific data) */
    /* The old way: check minor number MINOR(inode->i_rdev) */

    MOD_INC_USE_COUNT;
}
```

- Who calls `xxx_open()`?
  - `sys_open()` creates the inode, the open file structure, then calls the special file’s open operation

# Example Character Device Driver

---

- Barebone example to show how things work
- Read and write to a 1K buffer in the device
- Allow multiple processes to open the same device
  - Need semaphore to synchronize among them
- Device internal data structure

```
struct xxx_dev {  
    char data[1024];  
    int len;  
    devfs_handle_t devfs_handle;  
    struct semaphore sem;  
};
```

# Example Open and Release

---

```
static struct xxx_dev xxx_dev;
```

```
static int xxx_open(struct inode * inode, struct file * filp)
{
    filp->private_data = &xxx_dev;

    MOD_INC_USE_COUNT;
    return 0;
}
```

```
static ssize_t xxx_release(struct inode *inode, struct file *filp)
{
    MOD_DEC_USE_COUNT;
    return 0;
}
```

# Example Read Functions

```
static ssize_t xxx_read(struct file *filp, char *buf, size_t count, loff_t*ppos)
{
    struct xxx_dev *d = filp->private_data;
    if (down_interruptible(&d->sem))
        return -ERESTARTSYS;
    if (*ppos >= d->len) {
        up(&d->sem); return 0;
    }
    if (*ppos + count > d->len)
        count =d->len - *ppos;
    if (copy_to_user(buf, d->data+*ppos, count)) {
        up(&d->sem); return -EFAULT;
    }
    up(&d->sem);
    *ppos += count;
    return(count);
}
```

# Example Write Functions

```
static ssize_t xxx_write(struct file *filp, const char *buf, size_t count, loff_t
    *ppos)
{
    struct xxx_dev *d = filp->private_data;
    if (down_interruptible(&d->sem))
        return -ERESTARTSYS;
    if (*ppos >= 1024) {
        up(&d->sem); return 0;
    }
    if (*ppos + count > 1024)
        count = 1024 - *ppos;
    if (copy_from_user(d->data+*ppos, buf, count)) {
        up(&d->sem); return -EFAULT;
    }
    up(&d->sem);
    *ppos += count;
    return(count);
}
```

# Discussions

---

- Q: if this is to deal with a real device, who is likely to fill the data[] so that xxx\_read() can fetch it out?
- Blocking read?
  - If there is no data, set up a wait queue, put itself into the wait queue
  - When data is available (likely in a tasklet of an interrupt handler), wake up all processes in this wait queue
- Blocking write?
  - Similar

# Summary

---

- Device Driver:
  - LKP §7
  - ULK §13
  - LDD2 §3
- Next Lecture: Block Device