ECE391
Computer System Engineering

Lecture 25

Dr. Zbigniew Kalbarczyk

University of Illinois at Urbana- Champaign

Fall 2018

Lecture Topics

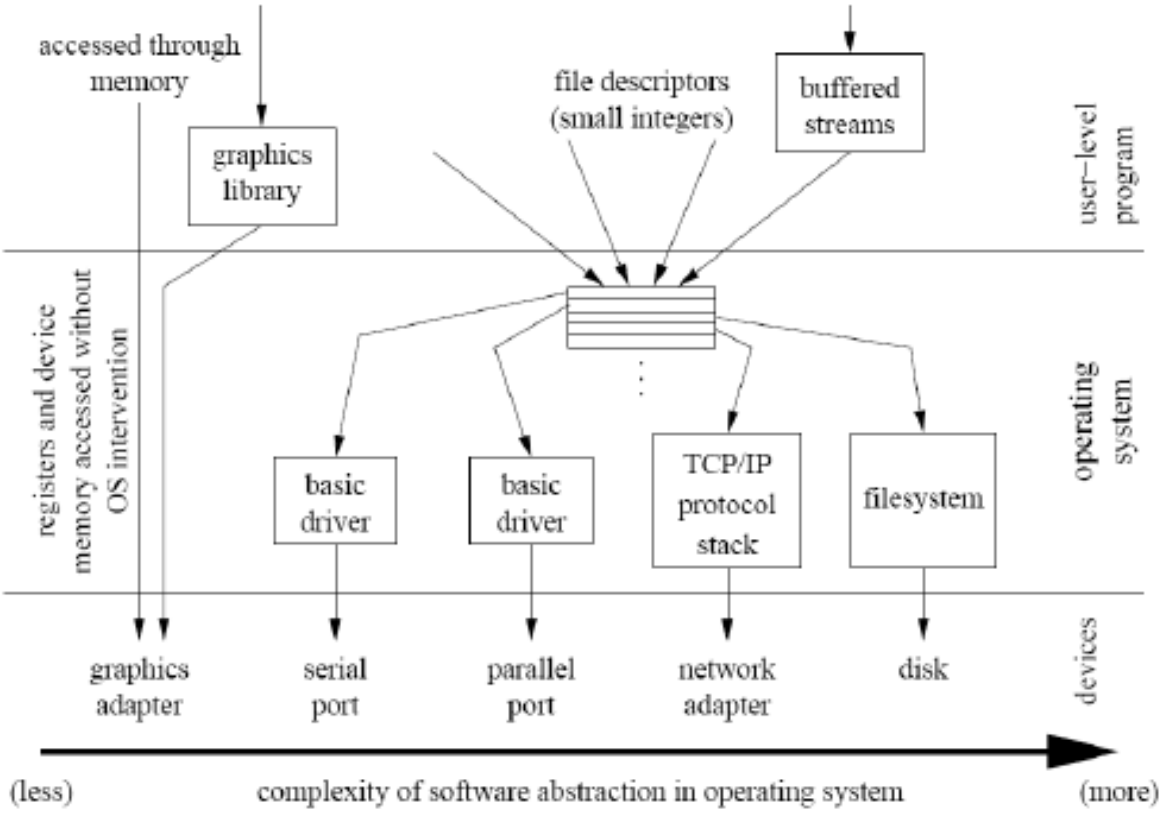
- Linux abstraction on device drivers
- Driver design process
- I-mail design

Device Drivers

- Kernel interacts with I/O devices by means of device drivers
  - operate at the kernel level
  - include data structures and functions that control one or more devises, e.g., keyboard, monitors, network interfaces
- Advantages:
  - device code can be encapsulated in a module
  - easy addition of new devices – no need to know the kernel source code
  - dynamic load/unload of device drivers

Example of Range of Device Driver Complexity

- Complexity grows from left to right



Range of Device Driver Complexity

- Leftmost example
  - no driver used (other than for obtaining/release access rights)
  - reads/writes go directly to device without OS intervention
  - lack of system calls improves performance
  - often used for graphics, and often with user-level libraries

- Remaining examples use file descriptor abstraction
  - sometimes with user-level libraries
  - all reduced to using file descriptors to do system calls
  - operating system
    - looks up file in array
    - uses file operations structure to hand off system call appropriately

Range of Device Driver Complexity

- Simple drivers such as serial port/parallel port
- Drivers for network adapters add state for protocol, kernel-side buffering
- Drivers for disks support block transfer, readahead, ….
  - usually used indirectly by file systems (inside kernel)
  - some user code (e.g., databases) uses raw disk commands

Kernel Abstraction for Devices – Device Files

- I/O devices are treated as special files called device files
  - same system calls used to interact with files on disk can be used to work with I/O devices
  - e.g., same *write()* used to
    - write to a regular file or
    - send data to a printer by writing to /dev/lp0 device file
- According to the characteristics of the underlying device drivers, device files can be of two types
  - block
  - character

Kernel Abstractions for Devices: Block Devices

- Block devices (underlies file systems)
  - data accessible only in blocks of fixed size, with size determined by device
  - can be addressed randomly
  - transfers to/from device are usually buffered (to) and cached (from) for performance
  - examples: disks, CD ROM, DVD

Kernel Abstractions for Devices: Character Devices

- Character device:
  - Almost everything else, except network cards
    - Network cards are not directly associated with device files

- A more constructive definition
  - contiguous space (or spaces) of bytes
  - some allow random access (e.g., magnetic tape driver)
  - others available only sequentially (e.g., sound card)
  - examples: keyboard, terminal, printers

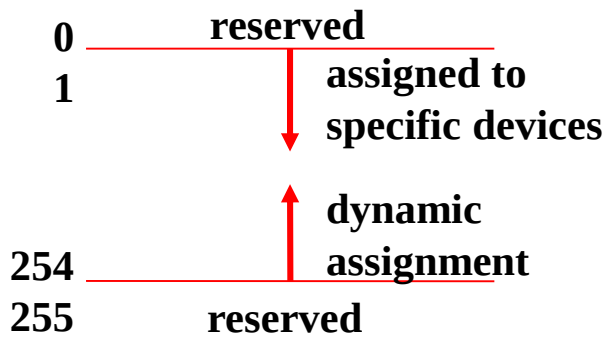
Kernel Abstractions for Devices

- A device file is usually stored as a real file
  - *Inode* includes an identifier of the hardware device corresponding to the character or block device file

- Devices identified by major and minor numbers (traditionally both 8-bit)
  - major number is device type (e.g., specific model of disk, not just “disk”)
  - minor number is instance number (if driver allows you to have more than one of a given model attached to your computer)

Kernel Abstractions for Devices

- Major numbers each have
  - associated fops structure
  - name (see **`/proc/devices`**)



Example of device files:

Name	Type	Major	Minor	Description
/dev/hda	block	3	0	First IDE disk
/dev/hda2	block	3	2	Secondary primary partion of first IDE disk
/dev/tty0	char	3	0	Terminal

Kernel Abstractions for Devices

- Registering and unregistering a device (see linux/fs.h)

```
int register_chrdev (unsigned int major,
      const char* name, struct file_operations* fops);
```

  - to request a specific major #, pass it as input argument
    - returns 0 on success
    - returns negative value on failure
  - for a dynamically assigned major #, pass 0 as input argument (major)
    - returns assigned major # on success (not 0!)
    - returns negative value on failure

Kernel Abstractions for Devices

```
int unregister_chrdev (unsigned int major,
                       const char* name);
```

- both parameters must match those of registration call
- returns 0 on success
- returns negative value on failure

Kernel Abstractions for Devices

To create a special device file for accessing a device, type (as root):

```
mknod <file> c <major> <minor>
```

- When open is called
  - major/minor #'s come in as **`inode->i_rdev`** **`(real device identifier)`**
  - can be split using **`MAJOR()`** and **`MINOR()`** macros
  - or **`imajor`**/**`iminor`** functions (argument is inode pointer)

File Operations

- File operations structure
  - jump table of file operations / character driver operations
  - generic instance for files on disk
  - distinct instances for sockets, etc.
  - one instance per device type

File Operations Structure

include/linux/fs.h

```
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 (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
        ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, 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);
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
        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 *);
        int (*fsync) (struct file *, struct dentry *, int datasync);
        int (*aio_fsync) (struct kiocb *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);

        /* ... plus a couple more rarely used functions */
};
```

Comments on File Operations

- Several direct mappings from system calls
  - llseek, read, write, etc.
  - arguments are identical
  - fsync: flashes the file by writing all cached data to disk

- offset pointer (**`loff_t*`**) argument
  - usually points to file's **`f_pos`**
  - but some system calls allow override, thus passed as pointer

Comments on File Operations

- flush is called each time a file is closed (may be open more than once)
- release is called after the last close (after the flush call)

- lock call used for file locking operations

- readv and writev
  - read and write vector operations (gather/scatter)
  - Emulated if function pointer is NULL

I-mail and Device Driver Design

- Example is Illinois mail (*I-mail*)
  - instant messaging within the Linux kernel
  - (perhaps not a best idea) but good example
    - somewhat complex abstraction
    - issues similar to other communication drivers (e.g., TCP/IP)
    - simpler than most real implementations
    - can be described from ground up

- Course Notes 6a on the class web site

Driver Design Process

0. “contemplate” security (hard/impossible to add in correctly as afterthought!)
1. write out all ops (in terms of visible interface)
2. design data structures
3. pick a locking strategy
4. determine locks needed, pick lock ordering
5. consider blocking issues & wakeup

Driver Design Process

6. consider dynamic allocation issues & hazards
7. write code
8. write subfunctions and synchronization rules
9. return to 3 or 4 if 7 or 8 fails
10. unit test