ECE391
Computer System Engineering

Lecture 25

Dr. Zbigniew Kalbarczyk

University of Illinois at Urbana- Champaign

Fall 2022

Lecture Topics

- Linux abstraction on device drivers

- *Next:* Driver design process: I-mail design

Aministrivia

- **MP3.5 Code cutoff:**
  - **6:00 pm, Sunday, December 4**
  - Hand-in/demo (all group members should be present)
  - Sunday (12/4), Monday (12/5), and Tuesday (12/6)

- Time slots posted at:

https://docs.google.com/spreadsheets/d/1tA7q5zk303KAQcpiIaxGlfpNFfmJdn6AAWeKzVBs-zw/edit#gid=0

Aministrivia

- **Class Competition**
  - Thursday December 8
  - More details to be provided
- **Final Exam:**
  - Tuesday December 13
  - Time: 7:00 pm - 10:00 pm

Kernel Role in the System (1)

- ***Process management***
  - implements the abstraction of several processes executing on top of a single/ multiple CPU(s)
  - e.g., the kernel is responsible for creating/destroying and scheduling processes
- ***Memory management***
  - supports virtual memory for all processes on top of the limited available resources
- ***Filesystem(s)***
  - supports different file systems
  - almost everything in Linux can be treated as a file

Kernel Role in the System (2)

- ***Device control***
  - supports the system interactions with hardware devices
  - almost every system operation eventually maps to a physical device
  - device control operations are defined/implemented by a *device driver* specific to a given hardware device

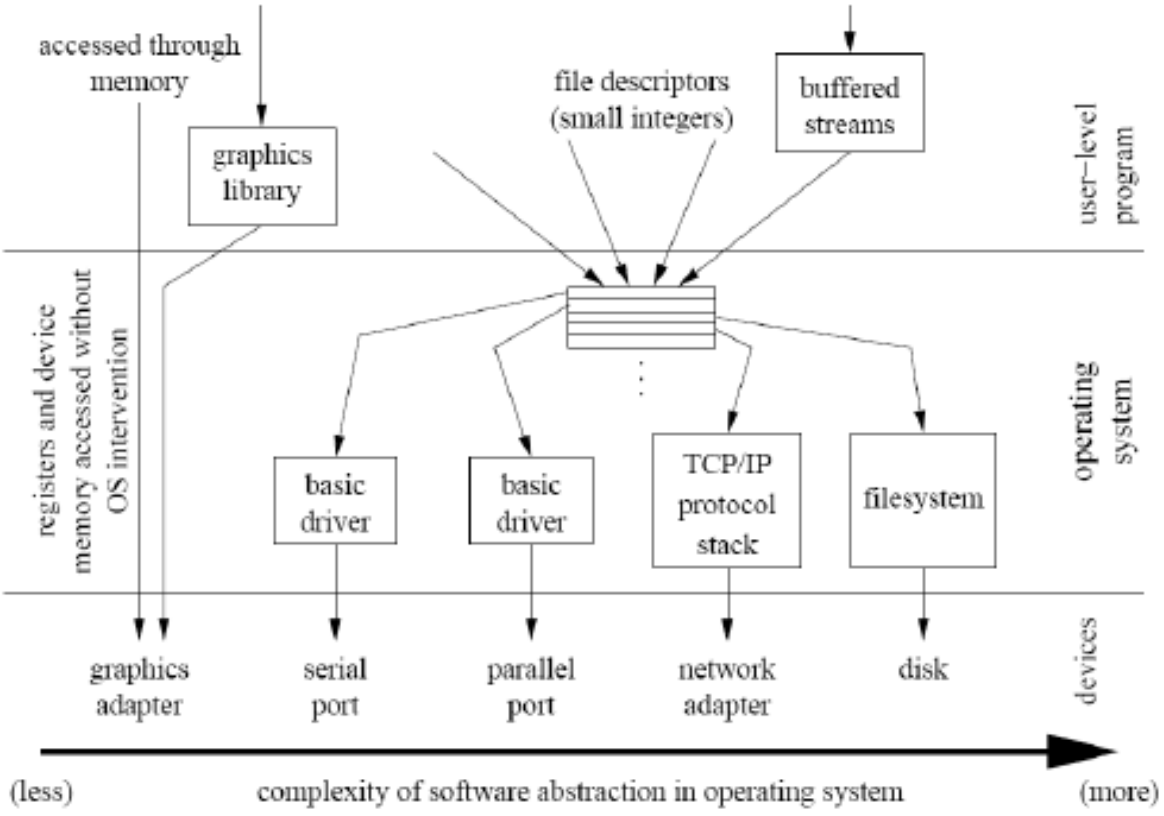
- ***Networking***
  - manages delivering data packets across program and network interfaces,
  - handles routing and address resolution

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
  - HW device responds to a well-defined programming interface
  - hide the details on how the device works
  - 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, read ahead, ….
  - 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

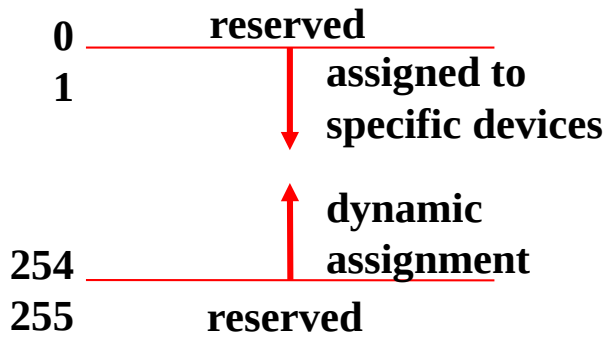
Kernel Abstractions for Devices

- 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

Device Driver Registration

- Each system call issued on a device file is translated by the kernel into an invocation of a function of a corresponding device driver
- To achieve this, a device driver must register itself
  - A device driver statically compiled in the kernel is registered during the kernel initialization phase
  - A device driver compiled as a kernel module is registered when the module is loaded
  - ***insmod*** e.g., insmode ./hello.ko #link the module to the running kernel
  - ***rmmod*** e.g., rmmod ./hello #unlink the module from the running kernel

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

Important Data Structures Used by the Driver

- Driver operations involve use of three kernel data structures:
  - ***file operations***
    - jump table of functions in the driver that implement specific file operations
  - ***file structure (or file object)***
    - represents an open file
    - no image in the permanent storage
    - created by the kernel on “open” system call
  - ***inode***
    - used internally by the kernel to represent file metadata
    - store in the permanent storage

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

Example: File Operation Structure in Imail

- static struct file_operations *Imail_fops* = {

```
        owner: THIS_MODULE,
        read:       Imail_read,
        write:      Imail_write,
        poll:       Imail_poll,
        ioctl:      Imail_ioctl,
        release:    Imail_release,
        fsync:      Imail_fsync,
};
```

Example of ***module_init() and module_exit()*** *for Imail*

- Kernel module must define two functions:

module_init (*Imail_init)* *# invoked when the module is loaded into the kernel*

module_exit (*Imail_exit*) *# invoked when the module is removed from the kernel*