Storage Systems

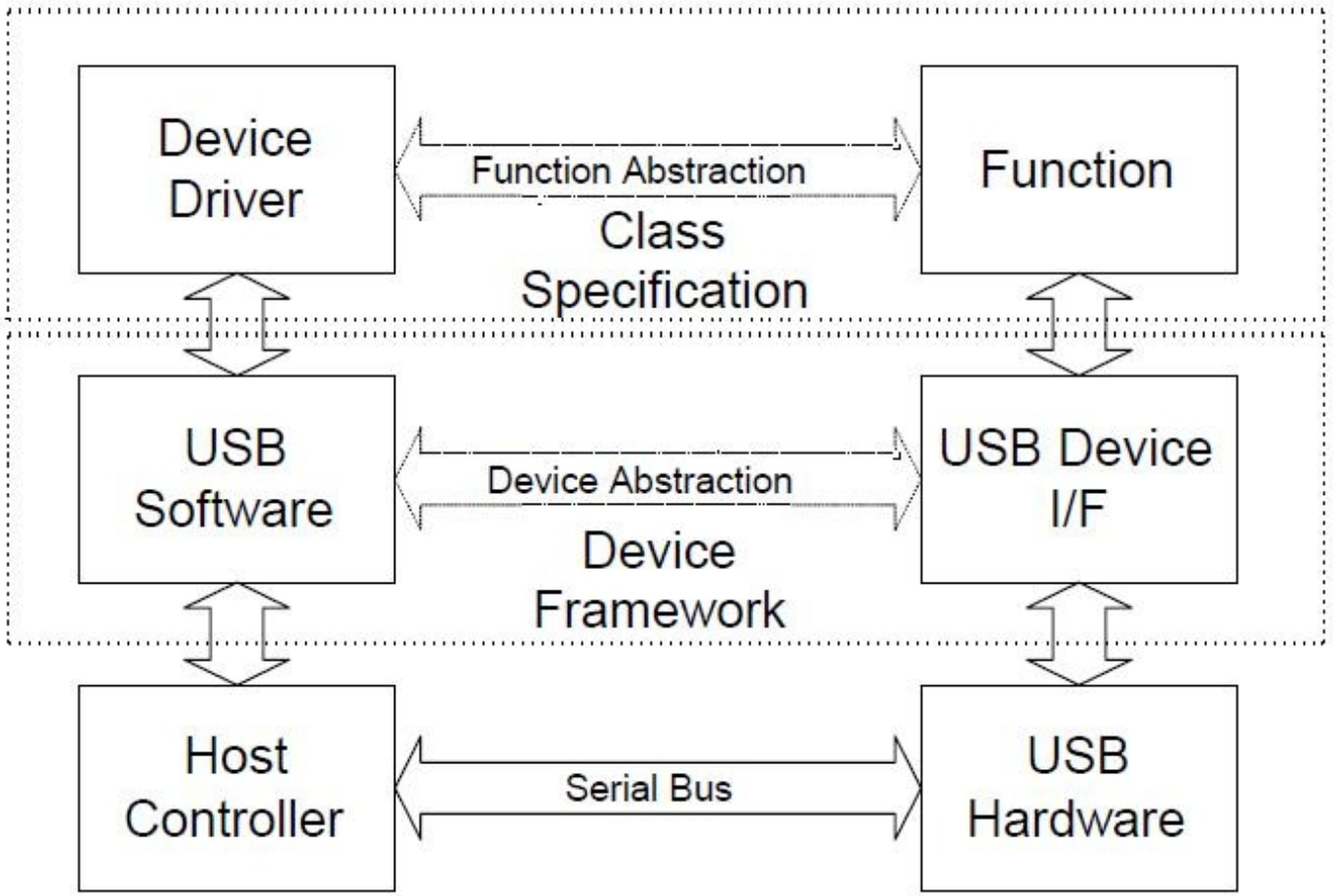
NPTEL Course
Jan 2013
(Lecture 11)

K. Gopinath
Indian Institute of Science

USB Mass Storage Device

- A USB has
  - a microcontroller that handles USB protocol
  - a media controller that handles device specific part (eg. storage)
- In a USB mass-storage device, hardware or firmware must
  - Detect and respond to generic USB requests and other events on bus Examples: requests to identify attached devices, manage traffic and power the bus.
  - Detect and respond to USB mass-storage requests for information such as status or actions from device.
    - These commands use the previous requests for transport.
  - Detect and respond to SCSI commands received in USB transfers such as read and write blocks of data in the storage media, request status information, and control device operation.
  - Optionally implement a filesystem if host commands are not required.

USB layering (from USB spec)



USB Layers

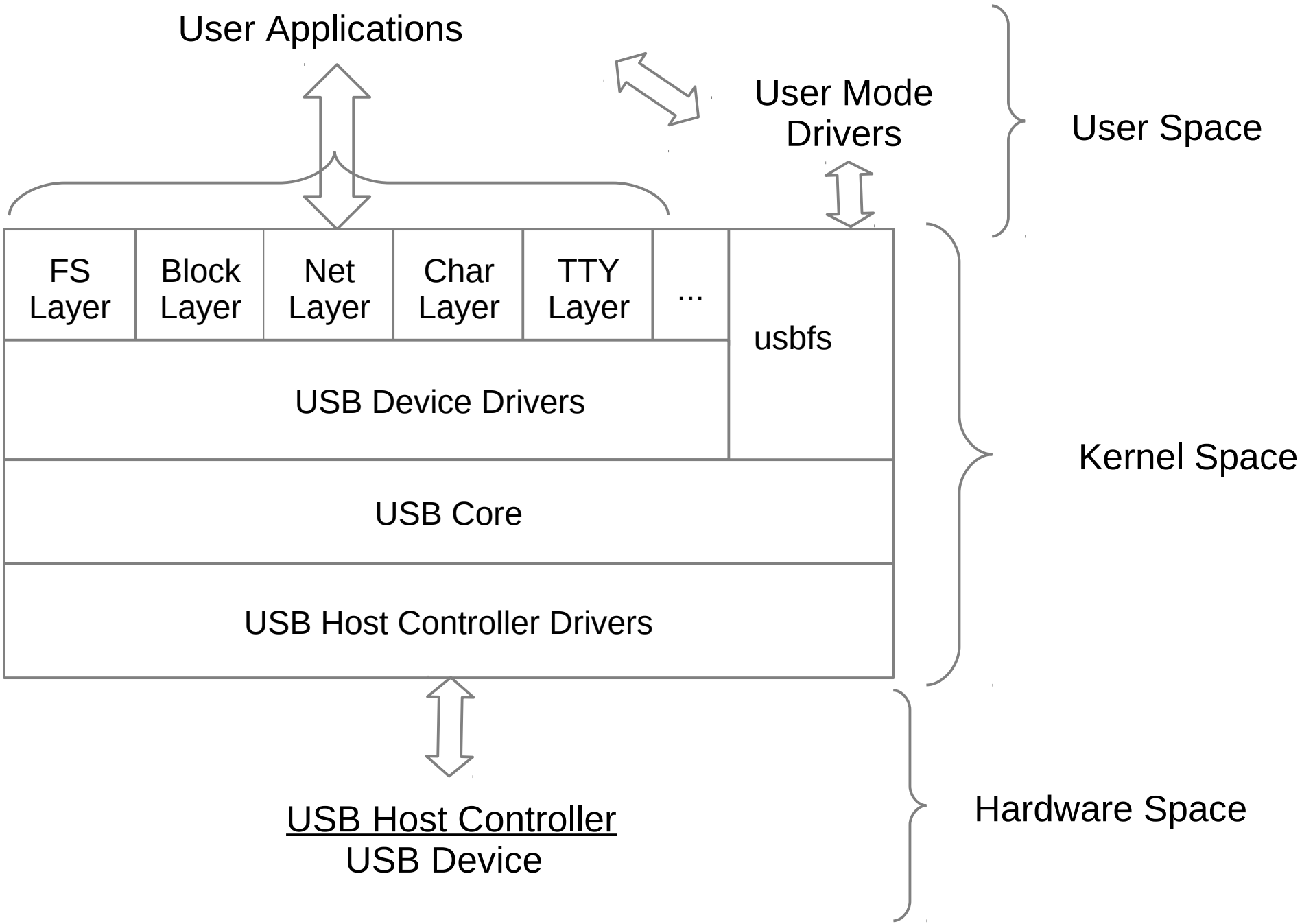
- The lowest layer concerns itself with aspects relating to serial transmission.
- The next upper layer handles USB protocol specific aspects; this is common to all USB devices.
- The next higher layer is function specific; for mass storage, this involves SCSI commands.
- Devices like camera have another layer; this is the filesystem layer as they should be able to store pictures taken without any outside help.
- Legacy devices are emulated

An explicit layering model helps in understanding the design.

USB Block Device

- USB mass storage specification provides a block interface
  - only the system that mounted it has access to the storage.
- Possible to provide multiple systems access to different files at the same time
  - requires a protocol such as MTP that operates at the level of a file rather than a block.
  - needed in a device, such as a camera with USB storage, when connected to a host.
- Note that only generic access such as read and write block
  - even if we have a USB SATA disk, usually no support for SATA features such as native command queuing (NCQ) which allows multiple disk operations to be sent.

Linux USB Framework



Insertion of a USB stick

- Hardware senses and interrupts the CPU.
- An agent (kernel thread, etc) identified in the interrupt routine to handle this insertion. Interrupt routine notes details of hardware that interrupted (such as USB port, controller, speed, PCI address).
- The kernel thread gets scheduled sometime later and notes the PCI information (that it is a USB mass storage device, USB wifi device, etc).
- The kernel thread upcalls an user program: in a GNU/Linux system, typically the program /sbin/hotplug, with “usb” as parameter.
  - a user mode policy agent
  - typically getting system resources needed, configure the device, helping to load driver or other modules, etc. by using configuration files.
- Once the device and its driver have been initialised, the filesystem(s) residing on it may be mounted and displayed to the user.

More Details

- Even if no driver for a USB device on, say, a Linux system, a valid USB device detected by hardware and later known to kernel
  - As design (and detection) as per USB protocol specs
- Hardware detection by the USB host controller: typically a native bus device, like a PCI device.
- Host controller driver gets low-level physical layer information and converts it into higher-level USB protocol-specific information.
- information about USB device then populated into the generic USB core layer (the usbcore driver) in the kernel, thus enabling the detection of a USB device at the kernel level, even without having its specific driver.
- Then, various drivers, interfaces, and applications (depends on the specific Linux distribution), give a userspace view of the detected devices.

- Many configurations for a USB device
  - Default also
  - For every configuration, the device may have 1+ interfaces.
    - An interface corresponds to a function provided by the device.
  - MFD (multi-function device) eg. USB printer: printing, scanning and faxing: 3 interfaces
- unlike other device drivers, a USB device driver typically associated/written per interface, rather than device as a whole
  - one USB device may have multiple device drivers
  - different device interfaces may have the same driver

USB Core APIs

- <linux/usb.h>
  - int usb_register(struct usb_driver *driver);
    - Gives info about name of driver, what probe/disconnect func to use etc.
  - void usb_deregister(struct usb_driver *);
  - FS register with VFS layer but USB devices to USB core
- USB core then uses the foll.
  - int (*probe)(struct usb_interface *interface, const struct usb_device_id *id);
  - void (*disconnect)(struct usb_interface *interface);
- endpoint-specific data transfer functions
  - usb_control_msg()
  - usb_interrupt_msg()
  - usb_rcvbulkpipe(), usb_sndbulkpipe()
  - Uses URBs (USB request block) for asynch I/O
  - Have to send SCSI cmds with USB mass storage devices

Outline of a Write to USB device

- When driver has data to send to the USB device (driver's write function), allocates
  - urb
  - DMA buffer
- Copies user data into DMA buffer
- urb initialized before sending to USB core
- After urb successfully transmitted to the USB device (or something happens in transmission), the urb callback called by the USB core

Summary

- Many subsystems work together
- Similar (high-level) structure in large as well as small storage systems