I/O Management

I/O Management Overview

1. Have protocols
    - Interfaces for device I/O
2. Have dedicated handlers
    - Device drivers, interrupt handlers, ...
3. Decouple I/O details from core processing
    - Abstract I/O device detail from applications

I/O Device Features

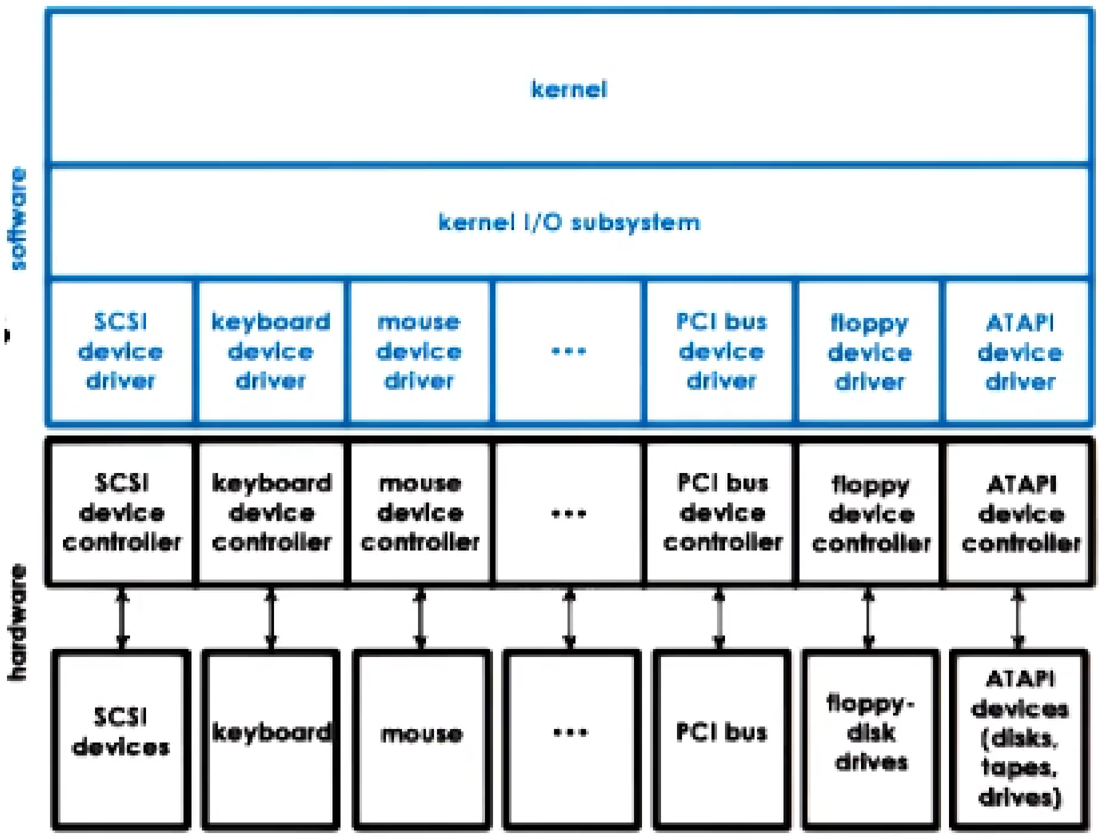
1. Control Registers
    - Command
    - Data transfers
    - Status
2. Microcontroller == Device's CPU
3. On device memory
4. Other logic (ADC, DAC, etc.)

CPU/Device Interconnect

1. Peripheral Component Interconnect (PCI)
    - Standard method for connecting devices to CPU
    - PCI-X: PCI Extended (better than PCI)
    - PCIE: PCI Express (more bandwidth, better than PCI-X)
2. Other types of interconnects
    - SCSI bus
    - Peripheral bus
    - Bridges handle differences

Device Drivers

1. Device-specific software components (per each device type)
    - Responsible for device access, management, and control
2. Provided by device manufacturers per OS version
3. Each OS standardizes interfaces
    - Device independence (OS doesn't need to be specialized for a type of functionality)
    - Device diversity (OS can support arbitrarily different devices)



Device Drivers

Types of Devices

1. Block: disk
   - Read/write block of data
   - Direct access to arbitrary block
2. Character: keyboard
   - Get/put character
3. Network devices
   - Stream of data (not a fixed block size)
4. OS representation of a device is a special device file
5. UNIX-like systems
   - /dev
   - tmpfs
   - devfs

CPU/Device Interactions

1. Memory-mapped I/O: Access device registers is equivalent to loading/ storing in memory
   - Part of 'host' physical memory dedicated for device interactions
   - Base address registers (BAR)
   - Configured during boot process
2. I/O port: Dedicated in/out instructions for device access
   - Target device (I/O port) and value in register
3. Interrupt

    - Pros: Can be generated as soon as possible
    - Cons: Interrupt handling steps
4. Polling
    - When convenient for OS
    - Delay or CPU overhead
5. Interrupt vs polling depends on kind of device and objectives

Device Access: Programmed I/O (PIO)

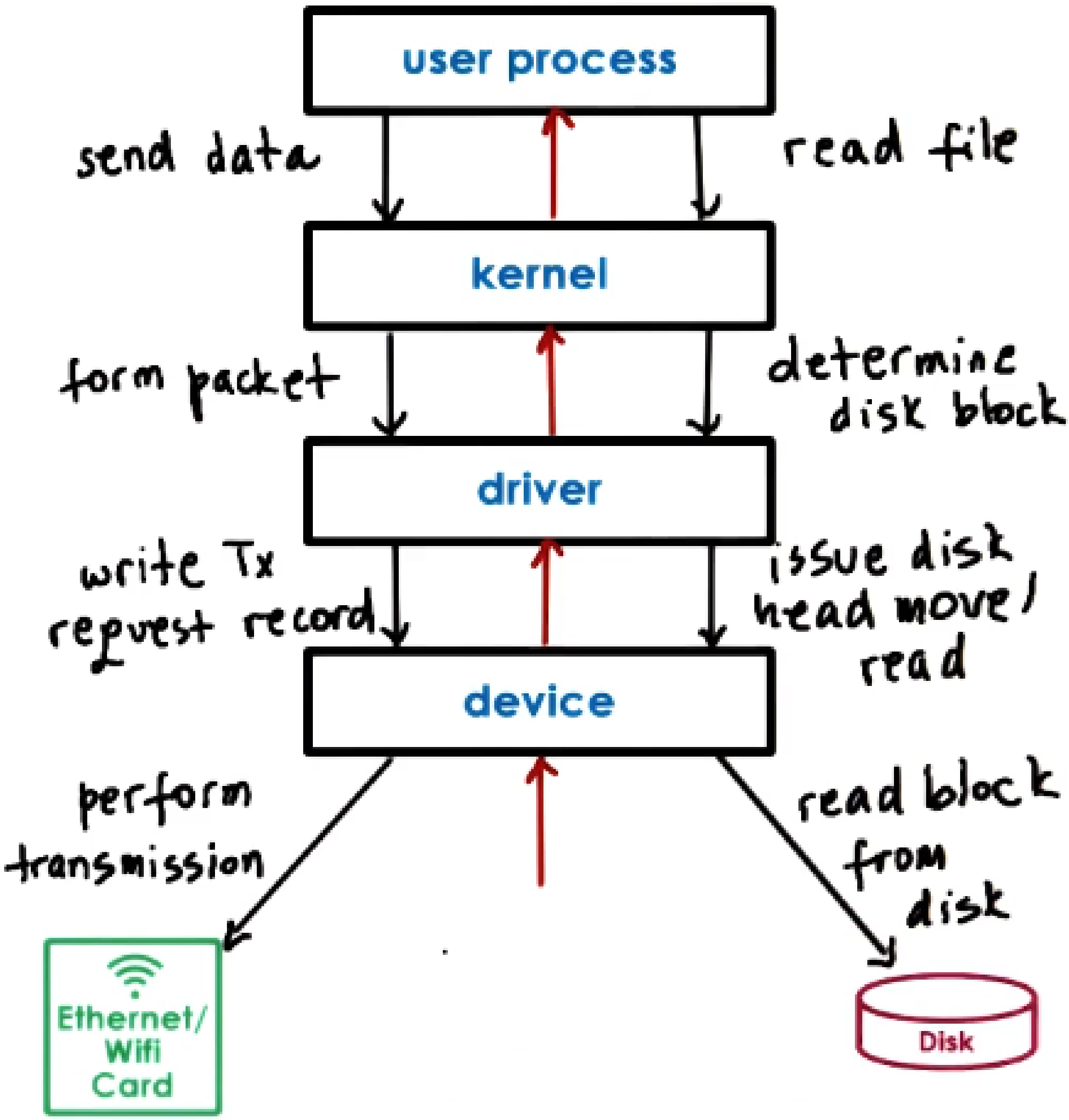
1. CPU "programs" the device by writing to command registers
    - Controls data movement by accessing data registers
2. No additional hardware support
3. Example: NIC (data is a network packet)
    - Write command to request packet transmission
    - Copy packet to data registers
    - Repeat until packet sent
    - 1500B packet; 8 byte registers/bus
        - 1 (for bus command) + 188 (for data)
        - 189 total CPU store instructions

Device Access: Direct Memory Access (DMA)

1. CPU "programs" the device via command registers
    - Controls data movement via DMA controls
    - DMA controller used for CPU/device communication
2. Example: NIC (data is a network packet)
    - Write command to request packet transmission
    - Configure DMA controller with in-memory address and size of packet buffer
    - 1500B packet; 8 byte registers/bus
        - 1 (for bus command) + 1 (DMA configure)
        - Fewer steps, but DMA configuration is more complex
3. Data buffer must be in physical memory until transfer completes for DMA
    - Must pin regions; not swappable

Typical Device Access

1. Perform system call
2. In-kernel stack
3. Driver invocation
4. Device request configuration
5. Device performs request



Typcial Device Access

Operating System Bypass

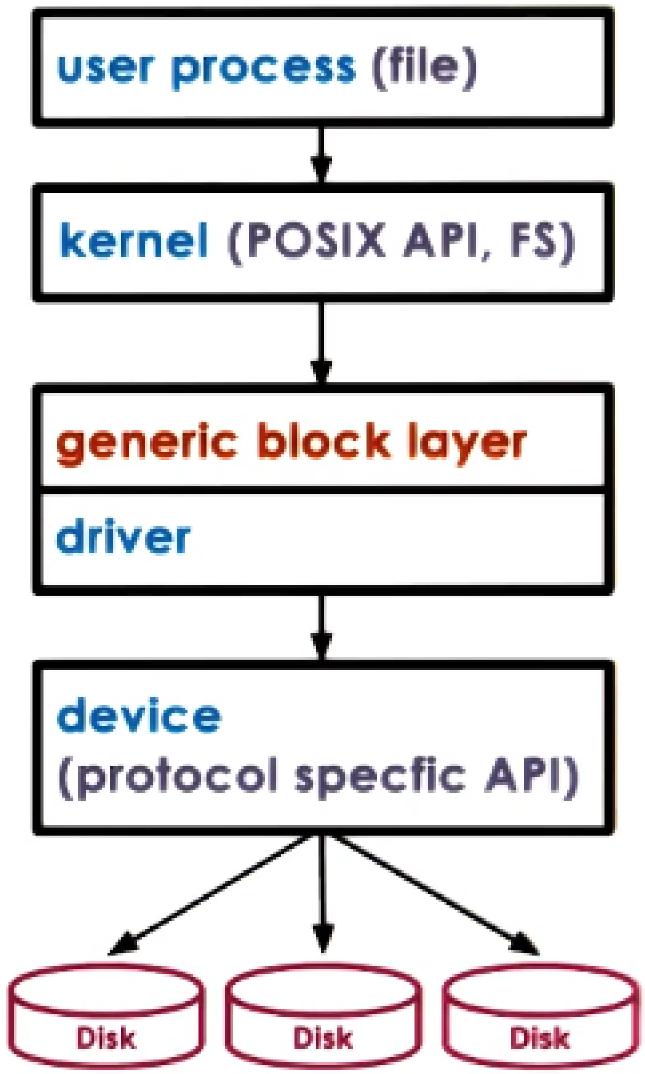
1. OS Bypass: Not required to go through the kernel
   - Device registers/data directly accessible
   - OS configures then gets out of the way
   - "User-level driver" (library)
2. OS still retains coarse-grained control
3. Relies on device features
   - Sufficient registers
   - De-multiplex capability for different processes
   - Kernel typically performs these operations, so device must perform

Synchronous vs Asynchronous Access

1. Synchronous I/O operations
    - Process blocks
2. Asynchronous I/O operations
    - Process continues
    - Later…
        - Process checks and retrieves result
        - Process is notified that the operation completed and results are ready

Block Device Stack

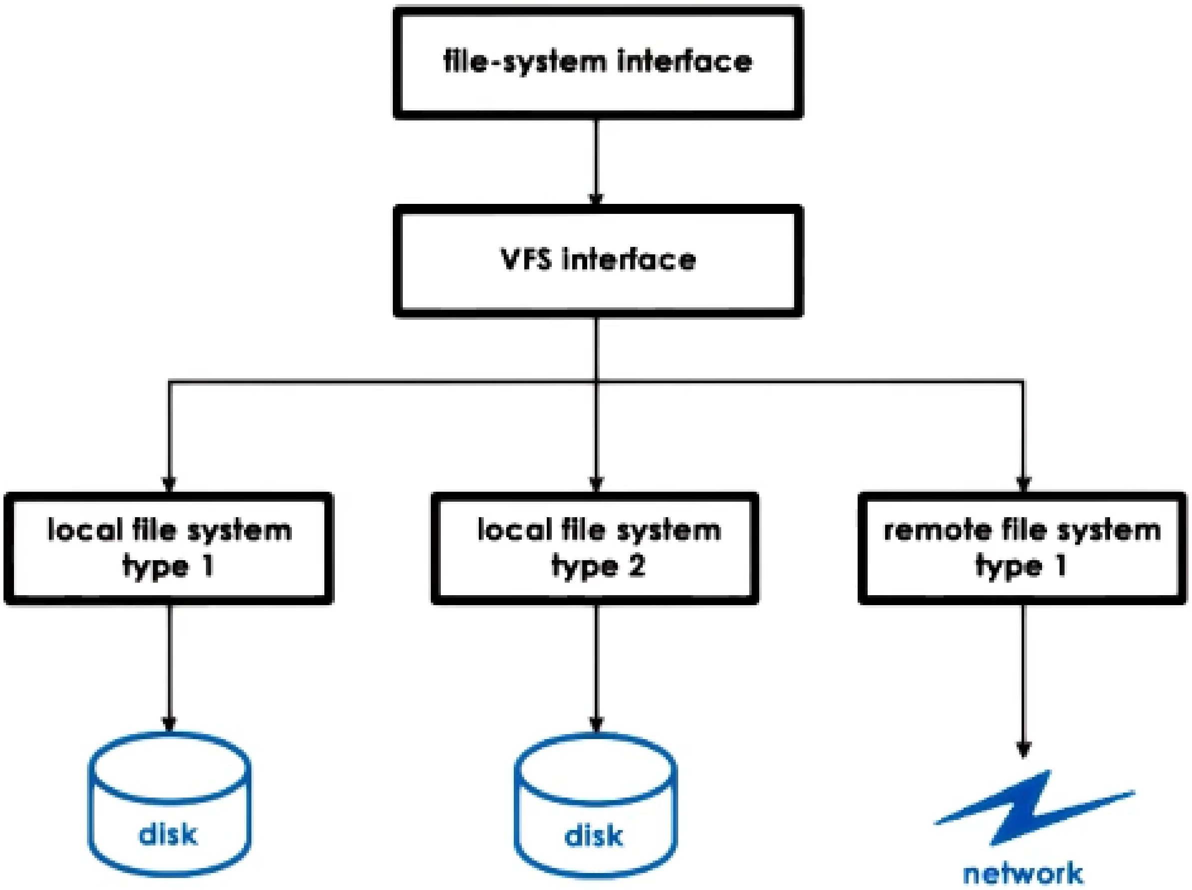
1. Processes use files -> Logical storage unit
2. Kernel file system (FS, POSIX API)
    - How to find and access file
    - OS specifies interface
3. Generic block layer
    - OS standardized block interface
    - Allows OS to interact with different interfaces in a uniform way
4. Device driver
5. Device (protocol-specific API)



Block Device Stack

Virtual File System

1. What if files are on more than one device?
2. What if devices work better with different filesystem implementations?
3. What if files are not on a local device (accessed via network)?
4. Linux implements a virtual filesystem that the user interacts with
   - Each underlying file system must implement a set of file system abstractions
   - Allows the user to interact with different devices in a uniform way



Virtual Filesystem

Virtual File System Abstractions

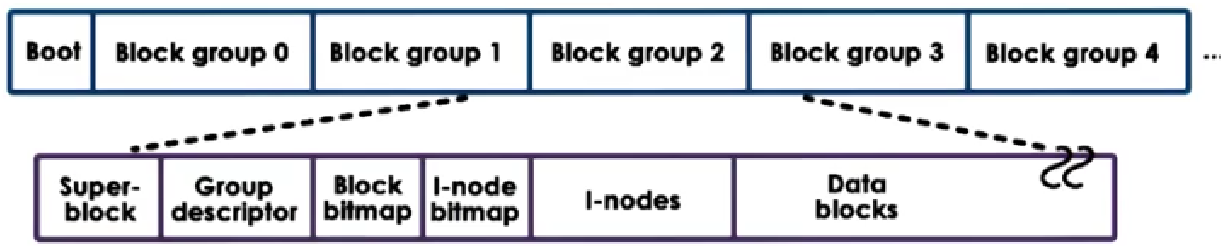
1. Files are the elements on which the VFS operates
2. File descriptor: OS representation of file
    - Open, read, write, sendfile, lock, close, ...
3. inode: Persistent representation of file “index”
    - List of all data blocks
    - Device, permissions, size, ...
4. dentry: directory entry, corresponds to single path component
    - /users/ada -> /, /users, /users/ada
    - Filesystem maintains cache of dentry entries
5. superblock: filesystem-specific information regarding FS layout

Virtual File System on Disk

1. file: Data blocks on disk
2. inode: Track files’ blocks
    - Also resides on disk in some block
3. superblock: overall map of disk blocks
    - inode blocks
    - data blocks
    - free blocks

Extended Filesystem v2.0 (ext2)

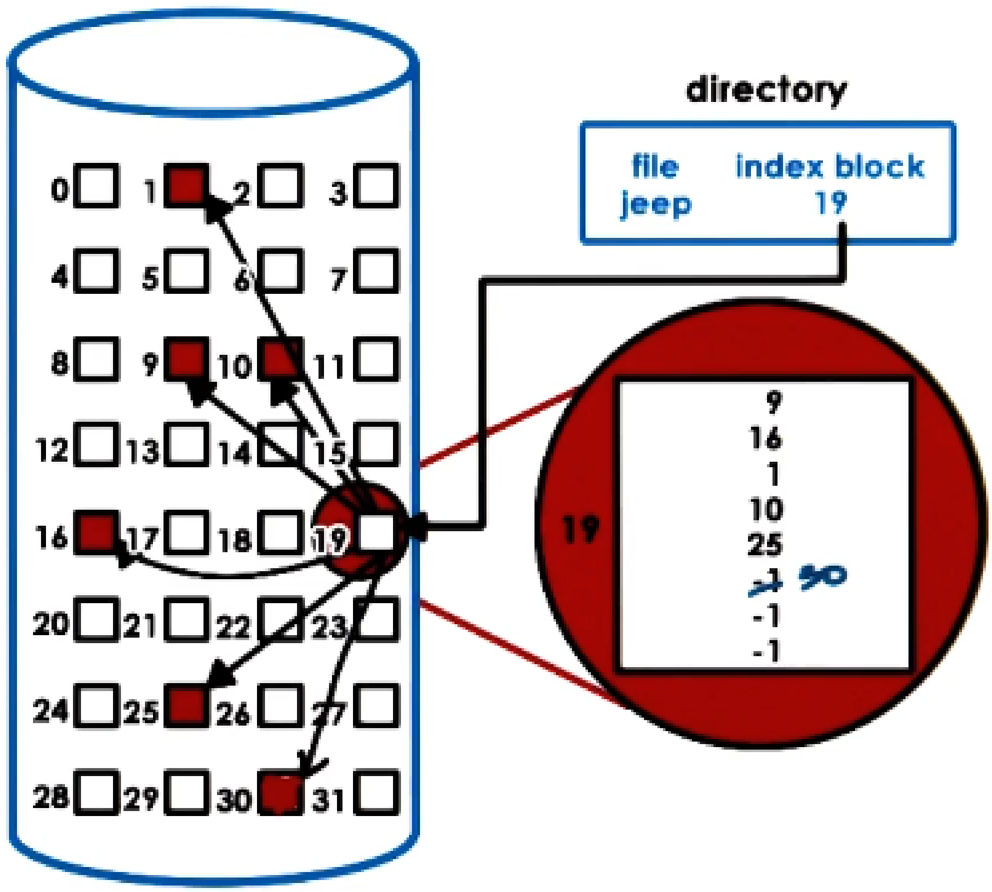
1. ext2: Second extended filesystem
2. For each block group…
   - Superblock: #inodes, #disk blocks, start of free blocks
   - Group descriptor: bitmaps, #free nodes, #directories
   - bitmaps: tracks free blocks and inodes
   - inodes: 1 to max number, 1 per file
   - data blocks: file data



ext2 Filesystem

inodes

1. Index of all disk blocks corresponding to a file
   - file: identified by inode
   - inode: list of all blocks + other metadata
2. Pros: Easy to perform sequential or random accesses to the file
3. Cons: Limit on file size (limited by total number of blocks)



inode

inodes with Indirect Pointers

1. Can use indirect pointers to solve file size issues
   - Index of all disk blocks corresponding to a file
2. inodes contain...
   - metadata
   - pointers to blocks
3. Direct pointer: Points to a data block
   - 1 kB per entry
4. Indirect pointer: Points to a block of pointers
   - 256 kB per entry
5. Double Indirect pointer: Points to a pointer to a block of pointers
   - 64 MB per entry
6. Pros: Small inode but large file sizes
7. Cons: File access slowdown (many disk accesses)

Disk Access Optimizations

1. Caching/buffering: Reduce #disk accesses
   - Buffer cache in main memory
   - read/write from cache

   - periodically flush to disk - fsync()
2. I/O scheduling: Reduce disk head movement
   - Maximize sequential vs random access
   - Write block 25, write block 17; If disk head is at 15, schedule 17->25
3. Prefetching: Increases cache hits
   - Leverages locality
   - Read block 17 -> also 18, 19
4. Journaling/logging: Reduce random access (ext3, ext4)
   - “Describe” write in log: block, offset, value
   - Periodically apply updates to proper disk locations