

Lecture 10

I/O

Prof. Yinqian Zhang

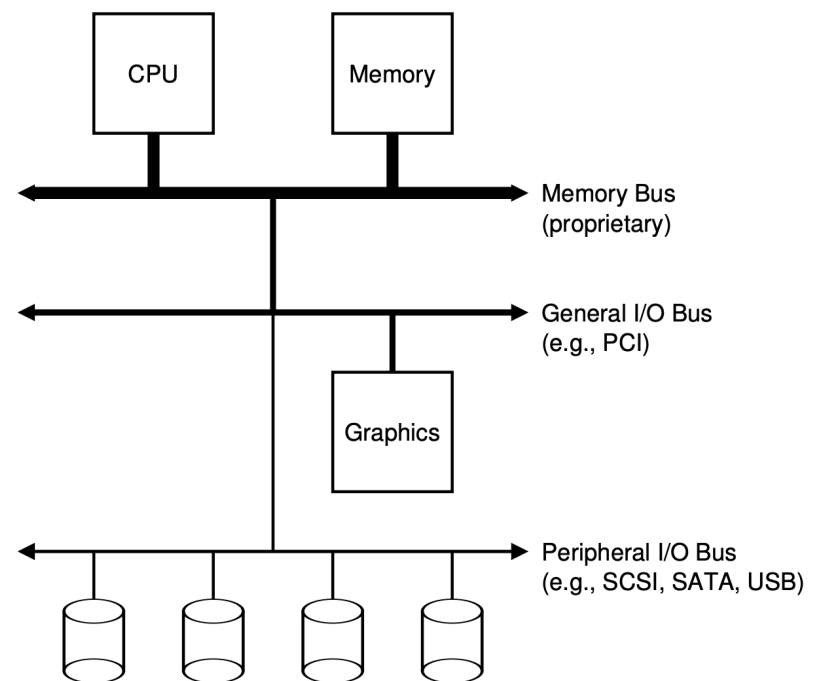
Fall 2024

I/O Management

- So far, we have learned how to manage CPU and memory
- Challenges of I/O management
 - Diverse devices: each device is slightly different
 - How can we standardize the interfaces to these devices?
 - Unreliable device: media failures and transmission errors
 - How can we make them reliable?
 - Unpredictable and slow devices
 - How can we manage them if we do not know what they will do or how they will perform?

A Classic View of Computer System

- A **single CPU** attached to the **main memory** of the system via memory bus or interconnect.
- Some devices (**graphics** and some other **higher-performance I/O devices**) are connected to the system via a general I/O bus (e.g., PCI)
- Finally, a peripheral bus, such as SCSI, SATA, or USB, connects slow devices to the system, including **disks, mice, and keyboards**



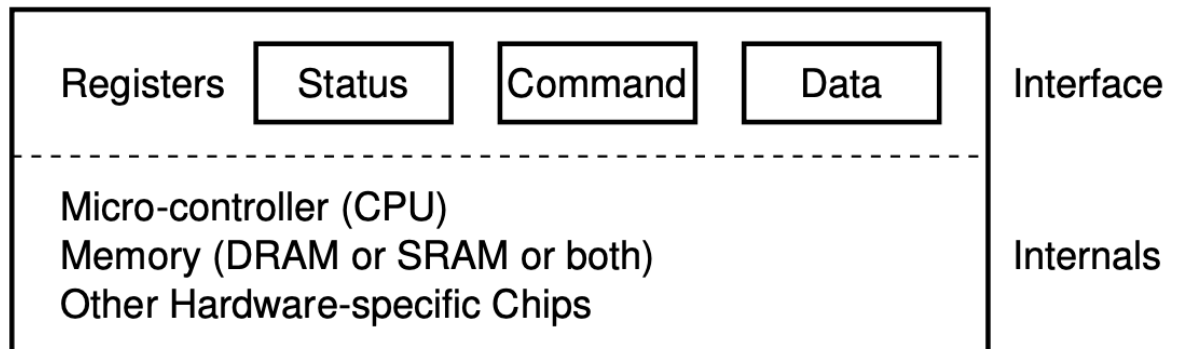
A Canonical View of Devices

- Interface

- The hardware interface a device present to the rest of the system
- **Status** registers: check the current status of the device
- **Command** register: tell the device to perform a certain task
- **Data** register: pass data to the device or get data from the device.

- Internal structures

- Implementation of the abstract of the device



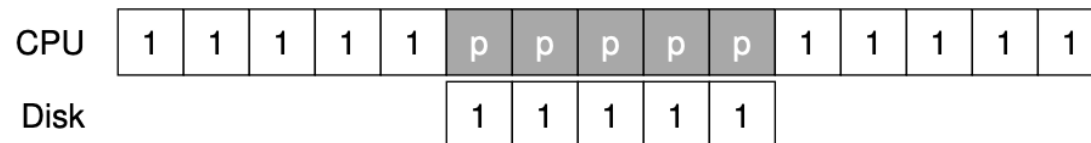
Basic I/O: Polling

- To write a byte of data to device
 - Step 1: OS waits until the device is ready to receive a command by repeatedly reading the status register;
 - Step 2: OS sends some data down to the data register;
 - Step 3: OS writes a command to the command register
 - Step 4: OS waits for the device to finish by again polling it in a loop, waiting to see if it is finished

```
While (STATUS == BUSY)
    ; // wait until device is not busy
Write data to DATA register
Write command to COMMAND register
    (starts the device and executes the command)
While (STATUS == BUSY)
    ; // wait until device is done with your request
```

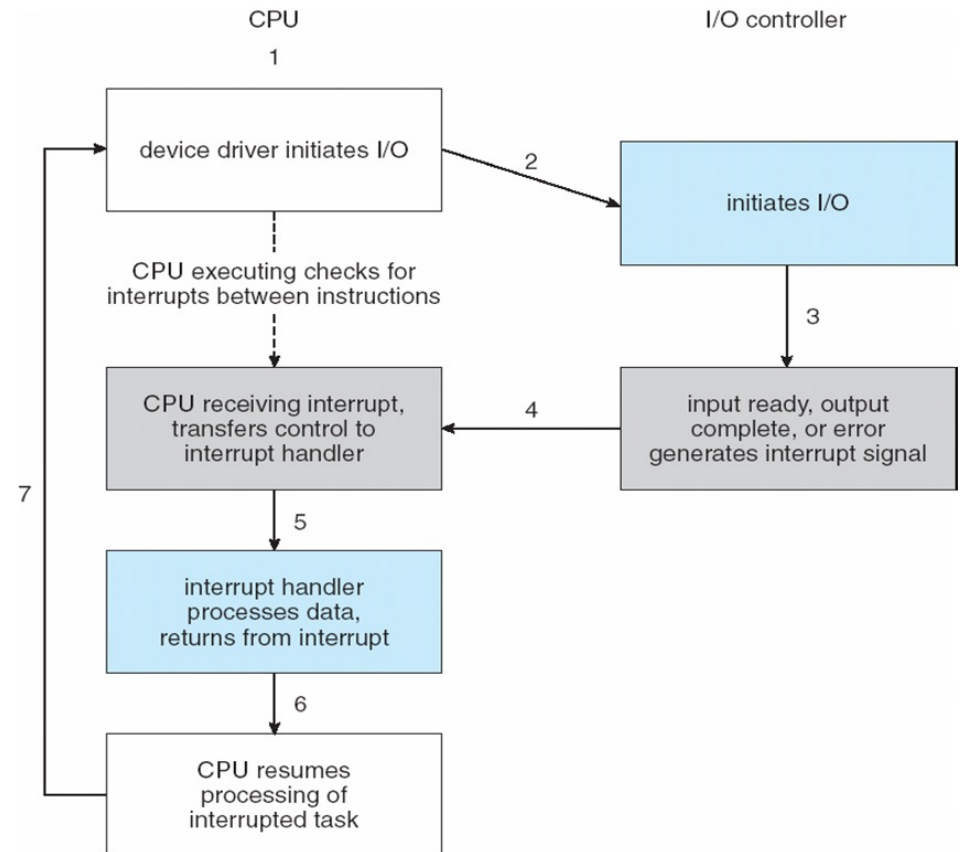
Issues of Polling

- Polling: frequent checking the status of I/O devices
- Polling is inefficient and inconvenient
 - Polling wastes CPU time waiting for slow devices to complete its activity
 - If CPU switches to other tasks, data may be overwritten
 - e.g., keyboard data overflow the buffer



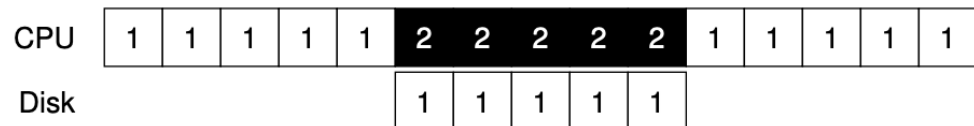
Efficient I/O: Interrupts

- Instead of polling the device repeatedly, the OS can issue a request, put the calling process to sleep, and context switch to another task.
- When the device is finally finished with the operation, it will raise a hardware interrupt, causing the CPU to jump into the OS at a predetermined **interrupt handler**



Polling or Interrupt?

- Polling works better for fast devices
 - Data fetched with first poll
- Interrupt works better for slow devices
 - Context switch is expensive
- Hybrid approach if speed of the device is not known or unstable
 - Polls for a while
 - Then use interrupts



Hardware Support for Interrupts

- **Interrupt-request line**, a CPU wire, triggered by I/O device
 - Checked by processor after each instruction
 - Save CPU state and jumps to the interrupt handler
- **Interrupt-controller hardware**
 - Defer interrupt handling during critical processing
 - Dispatch to proper interrupt handler
 - Support multi-level interrupts, high- and low-priority interrupts

Software Support for Interrupts

- **Interrupt handler** receives interrupts
 - **Maskable** to ignore or delay some interrupts
 - Some are **nonmaskable**, e.g., unrecoverable memory errors.
- A table of **interrupt vectors** to specify interrupt-handling routine
 - Dispatch interrupt to correct handler
 - **Interrupt chaining** if more than one device at the same interrupt number
 - Interrupt handlers on the corresponding chain are called one by one
 - The size of the interrupt table (i.e., number of interrupt vectors) and length of interrupt chains are results of system design trade-off.
 - Priority of interrupts: high-priority interrupts can preempt low-priority interrupts

Aside: Interrupts and Exceptions

- Interrupt mechanism also used for **exceptions**
 - Page fault is an exception that raises interrupts
 - Dividing by 0
 - Attempting to execute a privileged instruction from user mode
- Software interrupts, or **traps**
 - System call is made by executing a special instruction called software interrupts to trigger kernel to execute request
 - Lower priority interrupt
- Multi-CPU systems can process interrupts concurrently
 - If operating system designed to handle it

Programmed I/O

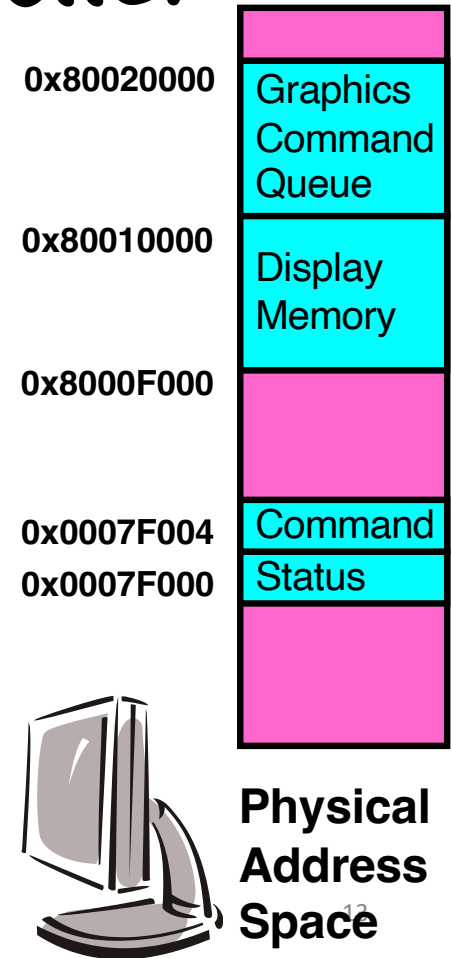
- Explicit I/O instructions
 - in/out instructions on x86:
out 0x21,AL
 - I/O instructions are privileged instructions
- Memory-mapped I/O
 - Registers/memory appear in physical address space
 - I/O accomplished with load and store instructions
 - I/O protection with address translation

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

Memory-Mapped Display Controller

- Memory-Mapped I/O

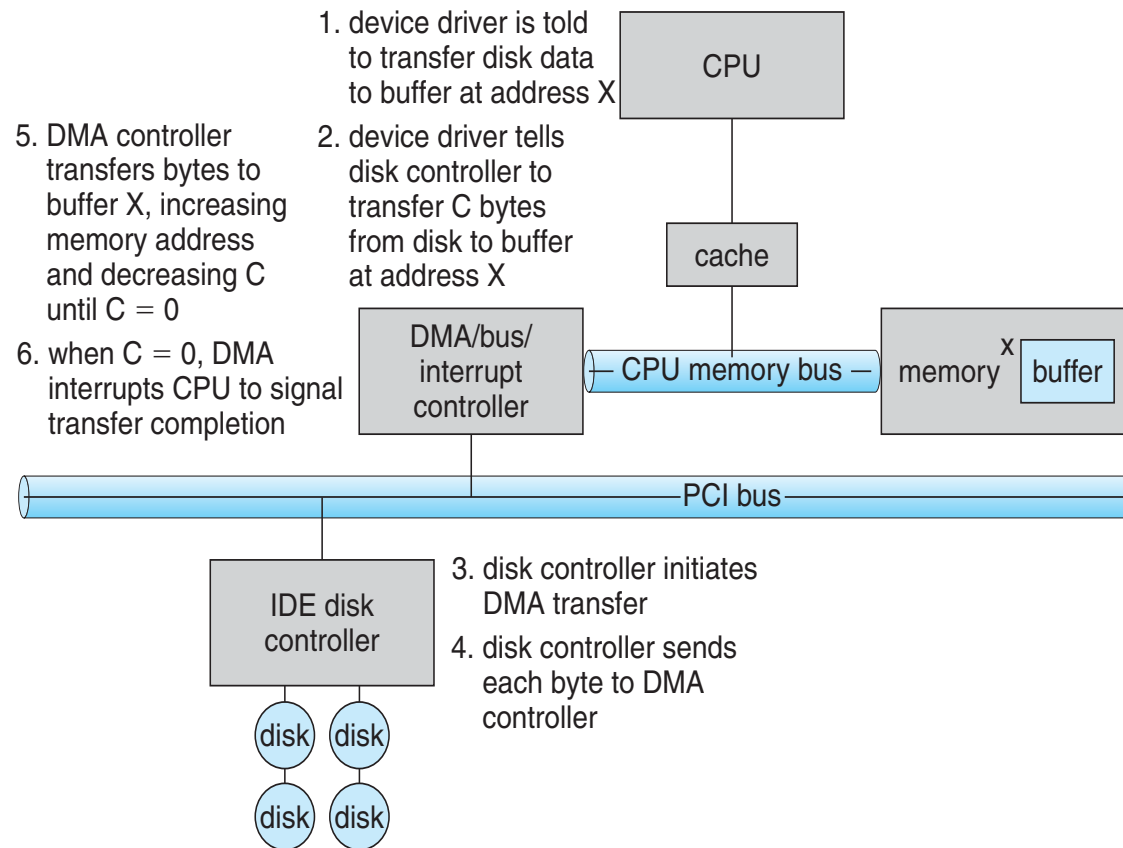
- Hardware maps control registers and display memory into physical address space
 - Addresses set by HW jumpers or at boot time
- Simply writing to display memory (also called the “frame buffer”) changes image on screen
 - Addr: 0x8000F000 — 0x8000FFFF
- Writing graphics description to cmd queue
 - Say enter a set of triangles describing some scene
 - Addr: 0x80010000 — 0x8001FFFF
- Writing to the command register may cause on-board graphics hardware to do something
 - Say render the above scene
 - Addr: 0x0007F004



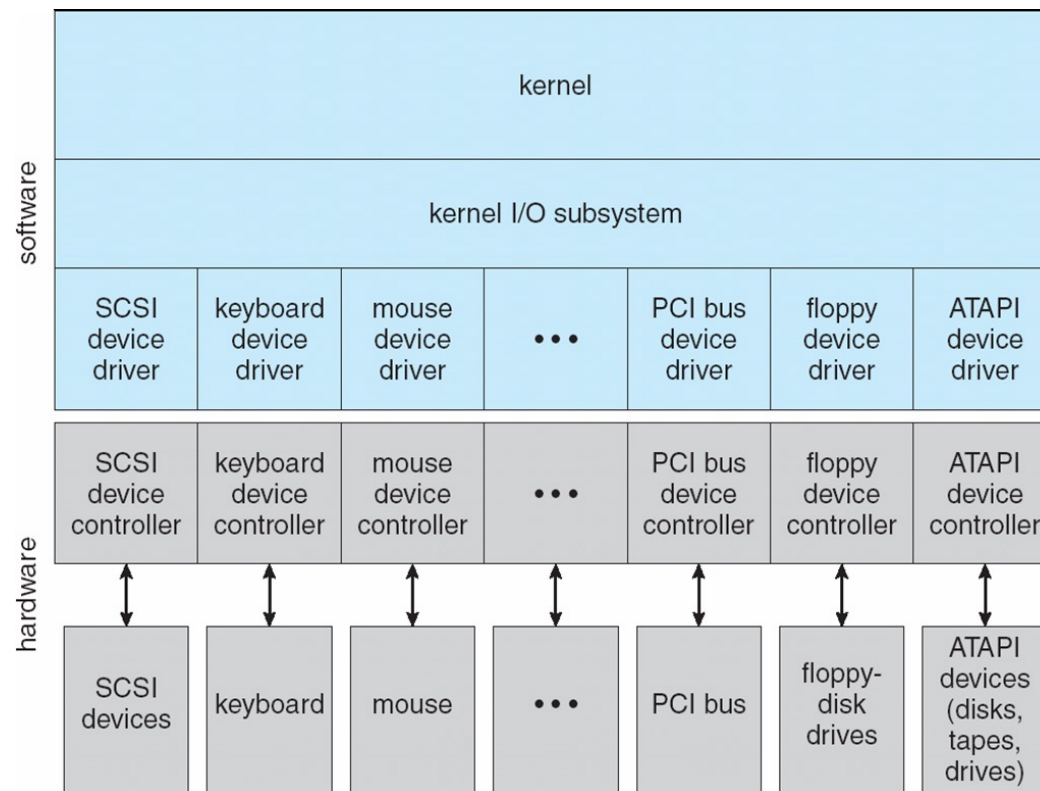
More Efficient Data Movement: DMA

- DMA is used to avoid **programmed I/O** for large data movement
 - Programmed I/O (PIO): when CPU is involved in data movement
 - PIO consumes CPU time
 - bypasses CPU to transfer data directly between I/O device and memory
- OS writes DMA command block into memory
 - Source and destination addresses
 - Read or write mode
 - Count of bytes
 - Writes location to DMA controller
 - Bus mastering of DMA controller – grabs bus from CPU
 - **Cycle stealing** from CPU but still much more efficient
 - When done, interrupts to signal completion

More Efficient Data Movement: DMA



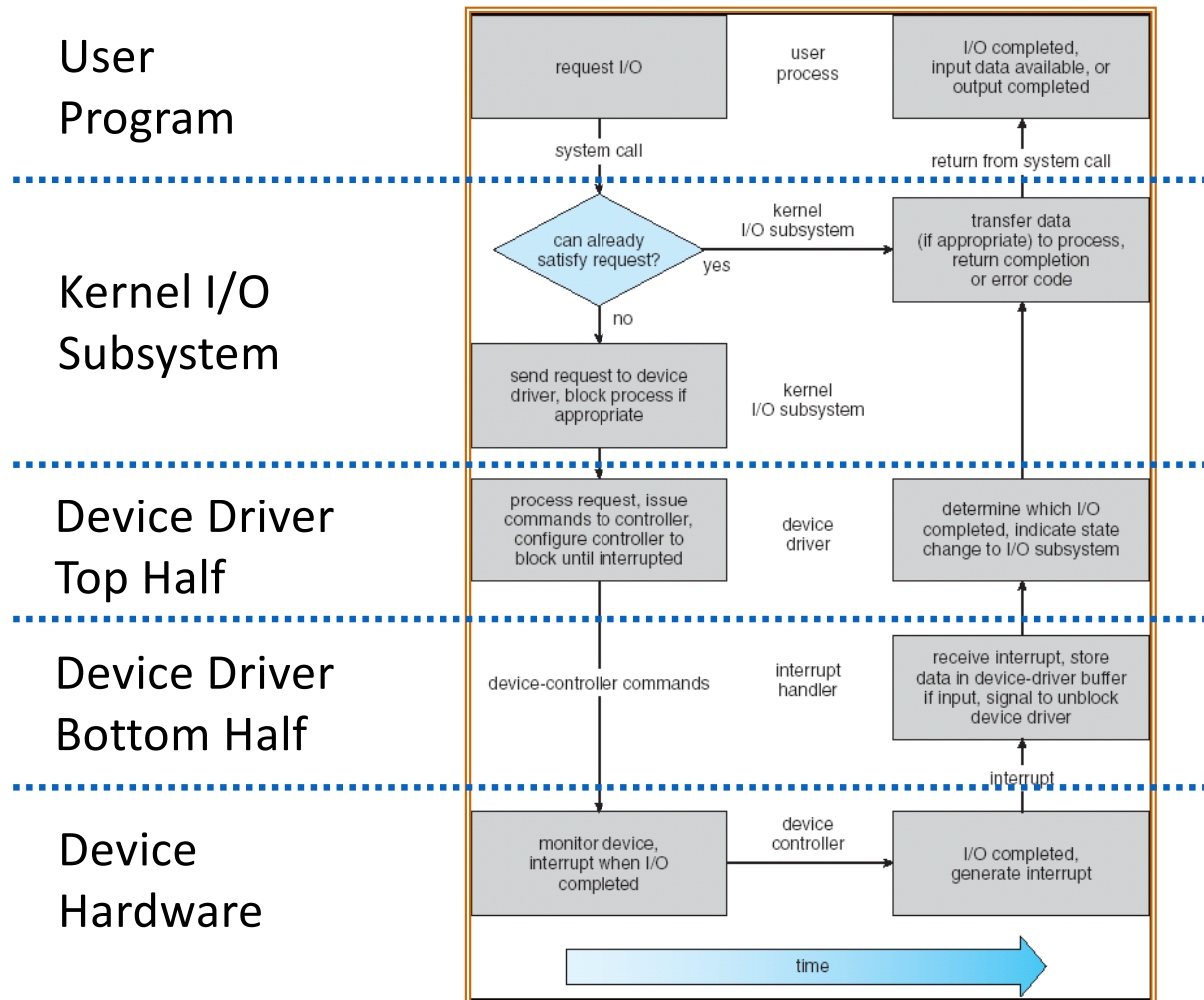
Kernel I/O Structure



Device Drivers

- Device Driver: Device-specific code in the kernel that interacts directly with the device hardware
 - Supports a standard, internal interface
 - Same kernel I/O system can interact easily with different device drivers
 - Special device-specific configuration supported with the `ioctl()` system call
- Device Drivers typically divided into two pieces:
 - Top half: accessed in call path from system calls
 - implements a set of standard, cross-device calls like `open()`, `close()`, `read()`, `write()`, `ioctl()`, `strategy()`
 - This is the kernel's interface to the device driver
 - Top half will *start* I/O to device, may put thread to sleep until finished
 - Bottom half: run as interrupt routine
 - Gets input or transfers next block of output
 - May wake sleeping threads if I/O now complete

Life Cycle of An I/O Request



Application I/O Interface

- I/O system calls encapsulate device behaviors in generic classes
- Device-driver layer hides differences among I/O controllers from kernel
- New devices talking already-implemented protocols need no extra work
- Each OS has its own I/O subsystem structures and device driver frameworks
- Devices vary in many dimensions
 - **Character-stream** or **block**
 - **Sequential** or **random-access**
 - **Synchronous** or **asynchronous**
 - **Sharable** or **dedicated**
 - **Speed of operation**
 - **read-write, read only, or write only**

Characteristics of I/O Devices (Cont'd)

- Subtleties of devices handled by device drivers
- Broadly I/O devices can be grouped by the OS into
 - Block I/O
 - Character I/O (Stream)
 - Memory-mapped file access
 - Network sockets
- For direct manipulation of I/O device specific characteristics from user-space applications
 - Unix `ioctl()` call to send arbitrary bits to a device control register and data to device data register

Block and Character Devices

- Block devices include disk drives
 - Commands include read, write, seek
 - **Raw I/O**, allows direct file-system access
 - Applications (e.g., database) do not need buffering or locking by filesystems
 - Memory-mapped file access possible
 - File mapped to virtual memory via demand paging
 - DMA
- Character devices include keyboards, mice, serial ports
 - Commands include `get()`, `put()` of one character
 - Libraries layered on top allow line editing

Nonblocking and Asynchronous I/O

- **Blocking** – process suspended until I/O completed
 - Processes moved from run queue to wait queue
- **Nonblocking** – I/O call returns as much as available
 - Returns quickly with count of bytes read or written
- **Asynchronous** – process runs while I/O executes
 - An alternative to nonblocking I/O
 - I/O request will be completed at some future time
 - I/O subsystem signals process when I/O completed
 - Software interrupt
 - Signal
 - Callback routine

Summary

- Two techniques to make I/O more efficient
 - Interrupts
 - DMA
- Two approaches to control devices
 - Explicit I/O instructions
 - Memory-mapped I/O
- The notion of device drivers
 - OS encapsulates low-level details and makes it easier to build the rest of the OS in a device-neutral fashion.

Thank you!

